**Microservices Part 1 of 8**

Contents

[Microservices 2](#_Toc212784601)

[Architecture Style, Benefits, Challenges and Anti Patterns 2](#_Toc212784602)

[[ Pending ] Introduction to Microservices Architecture 6](#_Toc212784603)

[Domain Driven Design [ DDD ] 6](#_Toc212784604)

[Strategic DDD 8](#_Toc212784605)

[Tactical DDD 11](#_Toc212784606)

[Design a microservices architecture 18](#_Toc212784607)

[Design Patterns for Microservices 41](#_Toc212784608)

[Microservices 42](#_Toc212784609)

[Cloud Pattern Catalog 67](#_Toc212784610)

[Monolith to Microservices 67](#_Toc212784611)

[Enterprise Application Architecture – Martin Fowler 67](#_Toc212784612)

[Architecting Container and Microservice Based Applications 68](#_Toc212784613)

[Microservice Pre-requisites 68](#_Toc212784614)

[Martin Fowler 68](#_Toc212784615)

[IntelliJ ShortCuts 68](#_Toc212784616)

[Free a Occupied port on Windows 69](#_Toc212784617)

# Microservices

## Architecture Style, Benefits, Challenges and Anti Patterns

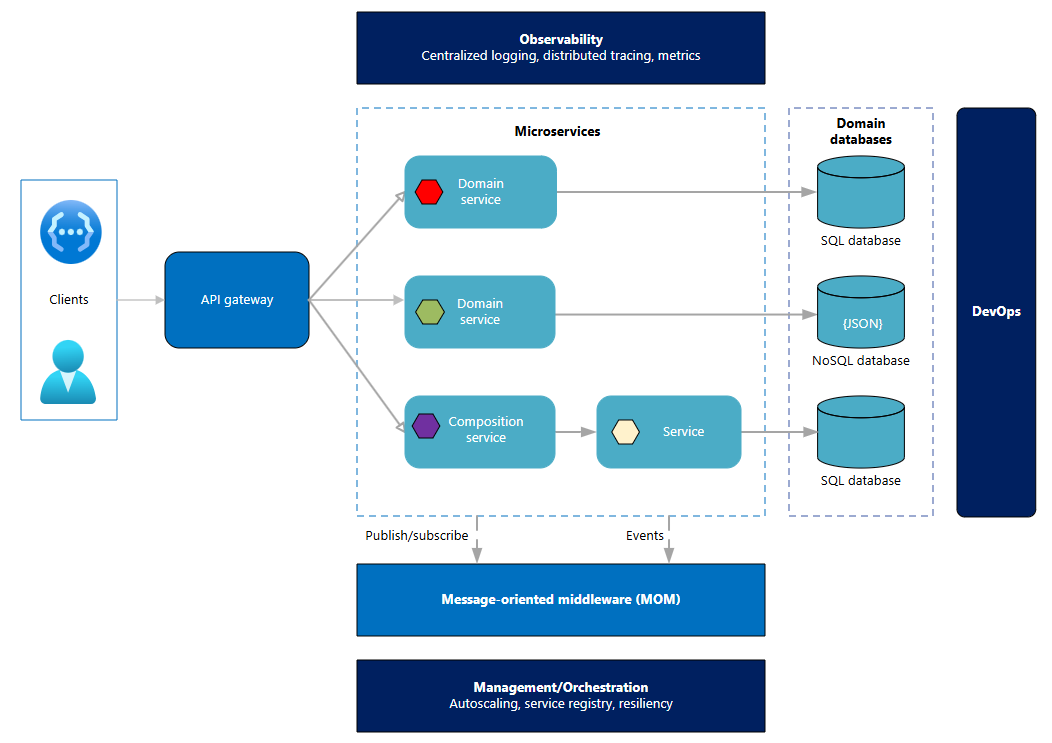
Architecture Style

Microservices are a popular architectural style for building applications that are resilient, highly scalable, independently deployable, and able to evolve quickly. Building a successful microservices architecture requires a fundamental shift in mindset. It goes beyond decomposing an application into smaller services. You must also rethink how systems are designed, deployed, and operated.

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

1. a natural division within a business and
2. provides an explicit boundary within which a domain model exists.



Microservices are small, independent, and loosely coupled components that a single small team of developers can write and maintain. Each service is managed as a separate codebase, which allows a small team to handle it efficiently. Because services can be deployed independently, teams can update existing services without rebuilding or redeploying the entire application. Unlike traditional models that have a centralized data layer, microservices are responsible for persisting their own data or external state. They communicate through well-defined APIs, which keeps internal implementations hidden from other services. This architecture also supports polyglot programming, which means that services don't need to share the same technology stack, libraries, or frameworks.

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 you find a bug in one part of the application, it can block the entire release process. For example, a bug can stall new features if you need to integrate, test, and publish a bug fix.

**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 to be less productive because communication is slower, management overhead increases, and agility diminishes.

**Small code base**: In a monolithic application, code dependencies often become tangled over time. Adding a new feature might require changes in many parts of the codebase. A microservices architecture avoids this problem by not sharing code or data stores. This approach minimizes dependencies and makes it easier to introduce new features.

**Mix of technologies**: Teams can pick the technology that best suits their service by using a mix of technology stacks as appropriate.

**Fault isolation**: If an individual microservice becomes unavailable, it doesn'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 by using asynchronous messaging patterns.

**Scalability**: Services can be scaled independently. This approach lets you scale out subsystems that require more resources without scaling out the entire application. Use an orchestrator such as Kubernetes to add a higher density of services onto a single host, which allows for more efficient resource usage.

**Data isolation**: Updating a schema is simpler in a microservices architecture because only one microservice is affected. In contrast, monolithic applications can complicate schema changes, since multiple components often interact with the same data. This shared access makes any modification potentially risky.

Challenges

The benefits of microservices come with trade-offs. Consider the following challenges before you create 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. Make sure that you consider challenges like service discovery, data consistency, transaction management, and interservice communication when you design your application.

**Development and testing**: Writing a small service that relies on other dependent services requires a different approach than writing a traditional monolithic or layered application. Existing tools aren't always designed to work with service dependencies. Refactoring across service boundaries can be difficult. It's 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 result in 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 method 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 extra latency can become a problem. You need to design APIs carefully. Avoid overly chatty APIs, think about serialization formats, and look for places to use asynchronous communication patterns like the Queue-Based Load Leveling pattern.

**Data integrity**: Each microservice is responsible for its own data persistence. As a result, data consistency across multiple services can be a challenge. Different services persist data at different times, using different technology, and with potentially different levels of success. When more than one microservice is involved in persisting new or changed data, it's unlikely that the complete data change could be considered an atomic, consistent, isolated, and durable (ACID) transaction. Instead, the technique is more aligned to Basically Available, Soft State, Eventual Consistency (BASE). Embrace eventual consistency where possible.

**Management**: A successful microservice architecture 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. Use DDD to identify bounded contexts and define clear service boundaries. Avoid creating overly granular services, which can increase complexity and reduce performance.

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

**Standardize** your **technology** choices by limiting the number of languages and frameworks that you use. Establish platform-wide standards for logging, monitoring, and deployment.

**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.

Improve security by using mutual Transport Layer Security (**mTLS**) for service-to-service encryption. Implement role-based access control and use API gateways to enforce policies.

Offload cross-cutting concerns, such as authentication and Secure Sockets Layer termination, to the gateway. Service meshes and frameworks like Dapr can also help with common cross-cutting concerns like mTLS authentication and resiliency.

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 requires updating the other service. Overly chatty communication between two services might be a symptom of tight coupling and low cohesion.

Use continuous integration and continuous deployment (**CI**/**CD**) pipelines to automate testing and deployment. Deploy services independently and monitor rollout health.

Isolate failures. Use resiliency strategies to prevent failures within a service from cascading. For more information, see Resiliency patterns and Design reliable applications.

Use chaos engineering to **test** **the** resiliency **of** **your** **microservice** **architecture** and its dependencies. Evaluate and improve how the system handles partial failures.

Implement centralized logging, distributed tracing (**OpenTelemetry**), and metrics collection to ensure observability.

Antipatterns for microservices

When you design and implement microservices, specific pitfalls frequently occur that can undermine the benefits of this architectural style. Recognizing these antipatterns helps teams avoid costly mistakes and build more resilient, maintainable systems. Avoid the following antipatterns:

Implementing microservices without a deep understanding of the business domain results in poorly aligned service boundaries and undermines the intended benefits.

Designing events that depend on past or future events violates the principle of atomic and self-contained messaging. This dependency forces consumers to wait and reduces system reliability.

Using database entities as events exposes internal service details and often fails to convey the correct business intent, which leads to tightly coupled and unclear integrations.

Avoiding data duplication at all costs is an antipattern. Using patterns like materialized views to maintain local copies improves service autonomy and reduces cross-service dependencies.

Using generic events forces consumers to interpret and filter messages. This approach adds unnecessary complexity and reduces clarity in event-driven communication.

Sharing common libraries or dependencies between microservices creates tight coupling, which makes changes risky and widespread and goes against the principle of self-contained services.

Exposing microservices directly to consumers results in tight coupling, scalability problems, and security risks. Using an API gateway provides a clean, manageable, and secure entry point.

Keeping configuration values inside microservices tightly couples them to specific environments, which makes deployments harder. However, externalizing configuration promotes flexibility and environment portability.

Embedding security logic like token validation directly inside microservices complicates their code and maintenance. Alternatively, offloading security to dedicated components keeps services focused and cleaner.

Failing to abstract common microservices tasks results in repetitive, error-prone code and limits flexibility. Alternatively, using abstraction frameworks like Dapr simplifies development by decoupling business logic from infrastructure concerns.

## [ Pending ] Introduction to Microservices Architecture

<https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices>

# Domain Driven Design [ DDD ]

DDD - Domain Analysis to model Microservices

One of the biggest challenges of microservices is to define the boundaries of individual services. The general rule is that a service should do only one thing, but putting that rule into practice requires careful thought. There's no mechanical process that produces the correct design. You have to think deeply about your business domain, requirements, architecture characteristics (also known as nonfunctional requirements), and goals. Otherwise, you can end up with a haphazard design that exhibits some undesirable characteristics, such as hidden dependencies between services, tight coupling, or poorly designed interfaces. This article shows a domain-driven approach to designing microservices. Evaluating service boundaries is an ongoing effort on evolving workloads. Sometimes the evaluation results in redefined definitions of existing boundaries that require more application development to accommodate the changes.

This article uses a drone delivery service as a running example. For more information about the scenario and the corresponding reference implementation, see [Design a microservices architecture](https://learn.microsoft.com/en-us/azure/architecture/microservices/design/).

Microservices should be designed around business capabilities, not horizontal layers such as data access or messaging. In addition, they should have loose coupling and high functional cohesion. Microservices are loosely coupled if you can change one service without requiring other services to be updated at the same time.

A microservice is cohesive if it has a single, well-defined purpose, such as managing user accounts or tracking delivery history.

A service should encapsulate domain knowledge and abstract that knowledge from clients.

For example, a client should be able to schedule a drone without knowing the details of the scheduling algorithm or how the drone fleet is managed.

Architecture characteristics have to be defined for each microservice to match its domain concerns, rather than being defined for the entire system. For example, a customer-facing microservice may need to have performance, availability, fault tolerance, security, testability, and agility.

A backend microservice may need to have only fault tolerance and security.

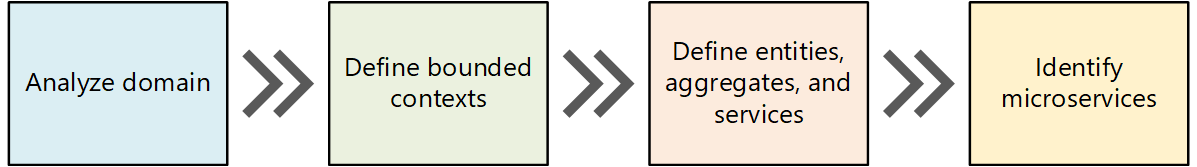
If microservices have synchronous communications with each other, the dependency between them often produces the need to share the same architecture characteristics.

Domain-driven design (DDD) provides a framework that can get you most of the way to a set of well-designed microservices.

**DDD has two distinct phases, strategic and tactical.**

* In Strategic DDD, you define the large-scale structure of the system. Strategic DDD helps to ensure that your architecture remains focused on business capabilities.
* Tactical DDD provides a set of design patterns that you can use to create the domain model. These patterns include entities, aggregates, and domain services.

These tactical patterns help you to design microservices that are both loosely coupled and cohesive.



In this article and the next, we'll walk through the following steps, applying them to the Drone Delivery application:

Start by analyzing the business domain to understand the application's functional requirements. The output of this step is an informal description of the domain, which can be refined into a more formal set of domain models.

Next, define the bounded contexts of the domain. Each bounded context contains a domain model that represents a particular subdomain of the larger application.

Within a bounded context, apply tactical DDD patterns to define entities, aggregates, and domain services.

Use the results from the previous step to identify the microservices in your application.

In this article, we cover the first three steps, which are primarily concerned with DDD. In the next article, we'll identify the microservices. However, it's important to remember that:

DDD is an iterative, ongoing process. Service boundaries aren't fixed in stone. As an application evolves, you may decide to break apart a service into several smaller services.

Note: This article doesn't show a complete and comprehensive domain analysis. We deliberately kept the example brief, to illustrate the main points. For more background on DDD, we recommend Eric Evans' Domain-Driven Design, the book that first introduced the term. Another good reference is Implementing Domain-Driven Design by Vaughn Vernon.

Example Scenario: Drone Delivery

Fabrikam, Inc. is starting a drone delivery service. The company manages a fleet of drone aircraft. Businesses register with the service, and users can request a drone to pick up goods for delivery. When a customer schedules a pickup, a backend system assigns a drone and notifies the user with an estimated delivery time. While the delivery is in progress, the customer can track the location of the drone, with a continuously updated ETA.

This scenario includes a fairly complex domain. Some of the key business concerns include scheduling drones, tracking packages, managing user accounts, and storing and analyzing historical data. Fabrikam also aims to get to market quickly and iterate rapidly, adding new functionality and capabilities. The application must operate at cloud scale and meet a high service-level objective. Also, Fabrikam expects different parts of the system to have varying requirements for data storage and querying. These considerations lead Fabrikam to adopt a microservices architecture for the Drone Delivery application.

## Strategic DDD

### Domain Analysis - Analyze the domain

A DDD approach helps you design microservices so that every service forms a natural fit to a functional business requirement. It can help you to avoid the trap of letting organizational boundaries or technology choices dictate your design.

Before you write any code, you should have a high-level understanding of the system that you build. DDD starts by modeling the business domain and creating a domain model. The domain model is an abstract model of the business domain. It distills and organizes domain knowledge and provides a common language for developers and domain experts.

### Map all business functions and their Connections

Start by mapping all the business functions and their connections. This effort can be a collaboration that includes domain experts, software architects, and other stakeholders. You don't need to use any particular formalism. Sketch a diagram or draw on whiteboard.

As you fill in the diagram, you might start to identify discrete subdomains.

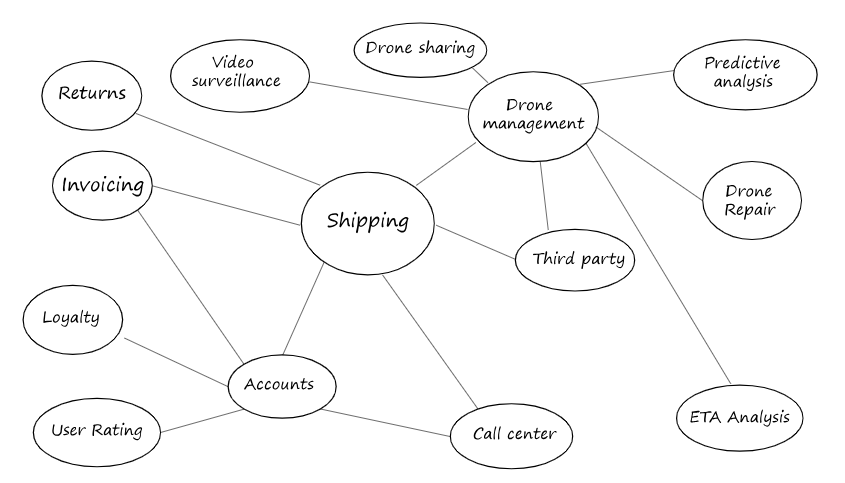
* Which functions are closely related?
* Which functions are core to the business?
* Which functions provide ancillary services?
* What's the dependency graph?

During this initial phase, you aren't concerned with technologies or implementation details.

That said, you should note the place where the application needs to integrate with external systems, such as customer relationship management, payment processing, or billing systems.

**Example: Drone delivery application**

After some initial domain analysis, the Fabrikam team came up with a rough sketch that depicts the Drone Delivery domain.



* Shipping is placed in the center of the diagram, because it's core to the business. Everything else in the diagram exists to enable this functionality.
* Drone management is also core to the business. Functionality that is closely related to drone management includes drone repair and using predictive analysis to predict when drones need servicing and maintenance.
* ETA analysis provides time estimates for pickup and delivery.
* Third-party transportation will enable the application to schedule alternative transportation methods if a package cannot be shipped entirely by drone.
* Drone sharing is a possible extension of the core business. The company may have excess drone capacity during certain hours, and could rent out drones that would otherwise be idle. This feature will not be in the initial release.
* Video surveillance is another area that the company might expand into later.
* User accounts, Invoicing, and Call center are subdomains that support the core business.

Notice that at this point in the process, we haven't made any decisions about implementation or technologies. Some of the subsystems may involve external software systems or third-party services. Even so, the application needs to interact with these systems and services, so it's important to include them in the domain model.

Note: When an application depends on an external system, there's a risk that the external system's data schema or API can leak into the application. This kind of leakage can compromise the architectural design. It's especially common with legacy systems that don't follow modern best practices and might use convoluted data schemas or outdated APIs. In these cases, it's important to establish a well-defined boundary between the external system and the application. Consider using the Strangler Fig pattern or the Anti-Corruption Layer pattern to enforce this boundary.

### Define bounded contexts

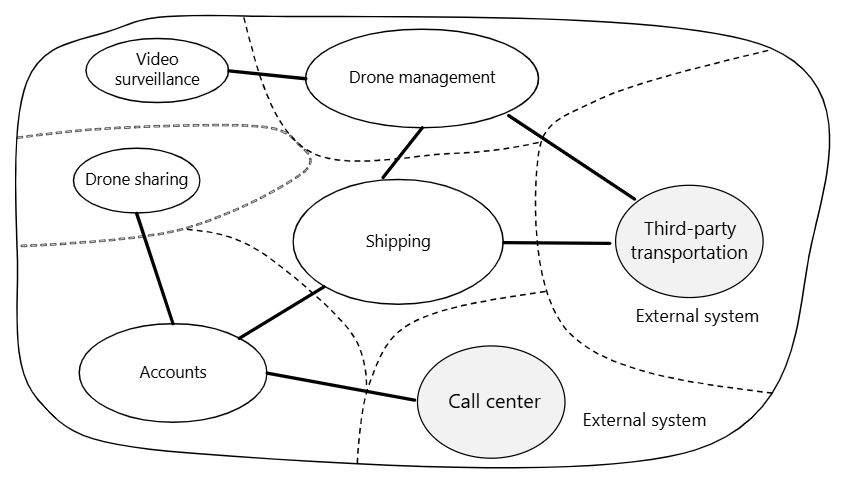
The domain model will include representations of real things in the world — users, drones, packages, and so forth. But that doesn't mean that every part of the system needs to use the same representations for the same things.

For example, subsystems that handle drone repair and predictive analysis will need to represent many physical characteristics of drones, such as their maintenance history, mileage, age, model number, performance characteristics, and so on. But when it's time to schedule a delivery, we don't care about those things. The scheduling subsystem only needs to know whether a drone is available, and the ETA for pickup and delivery.

If we tried to create a single model for both of these subsystems, it would be unnecessarily complex. It would also become harder for the model to evolve over time, because any changes will need to satisfy multiple teams working on separate subsystems. Therefore, it's often better to design separate models that represent the same real-world entity (in this case, a drone) in two different contexts. Each model contains only the features and attributes that are relevant within its particular context.

The DDD concept of bounded contexts comes into play here. A bounded context defines the boundary within a domain where a specific domain model applies.

Referring to the previous diagram, you can group functionality based on whether different functions share the same domain model.



Bounded contexts are not necessarily isolated from one another. In this diagram, the solid lines connecting the bounded contexts represent places where two bounded contexts interact. For example, Shipping depends on User Accounts to get information about customers, and on Drone Management to schedule drones from the fleet.

In the book Domain Driven Design, Eric Evans describes several patterns for maintaining the integrity of a domain model when it interacts with another bounded context. One of the main principles of microservices is that services communicate through well-defined APIs. This approach corresponds to two patterns that Evans calls Open Host Service and Published Language. The idea of Open Host Service is that a subsystem defines a formal protocol (API) for other subsystems to communicate with it. Published Language extends this idea by publishing the API in a form that other teams can use to write clients. In the article Designing APIs for microservices, we discuss using OpenAPI Specification (formerly known as Swagger) to define language-agnostic interface descriptions for REST APIs, expressed in JSON or YAML format.

For the rest of this journey, we will focus on the Shipping bounded context.

**Next steps**

After completing a domain analysis, the next step is to apply tactical DDD, to define your domain models with more precision.

## Tactical DDD

Using tactical DDD to design microservices

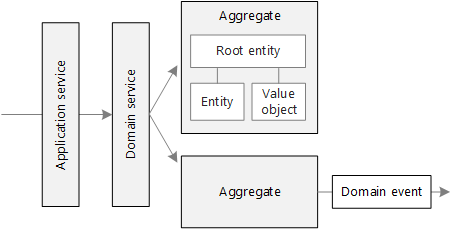
Domain-driven design (DDD) opposes the idea of having a single unified model for the entire system. Instead, it encourages dividing the system into bounded contexts, each with its own model. During the strategic phase of DDD, you map out the business domain and define bounded contexts for your domain models.

Tactical DDD is when you define your domain models with more precision. The tactical patterns are applied within a single bounded context. In a microservices architecture, where each bounded context is a microservice candidate, the entity and aggregate patterns are of note. Applying these patterns helps identify natural boundaries for the services in your application. For more information, see [Identify microservice boundaries](https://learn.microsoft.com/en-us/azure/architecture/microservices/model/microservice-boundaries).

General principle: A microservice should be no smaller than an aggregate & no larger than a bounded context.

This article reviews the tactical patterns and then applies them to the Shipping bounded context in the Drone Delivery application.

This section provides a brief summary of the tactical DDD patterns. If you're familiar with DDD, you might choose to skip it. These patterns are described in more detail in chapters 5 and 6 of Eric Evans' book, and in Implementing Domain-Driven Design by Vaughn Vernon.



### Patterns

#### Entities

Entities. An entity is an object with a unique identity that persists over time. For example, in a banking application, customers and accounts would be entities.

An entity has a unique identifier in the system, which can be used to look up or retrieve the entity. That doesn't mean the identifier is always exposed directly to users. It could be a GUID or a primary key in a database.

An identity can span multiple bounded contexts and might persist beyond the lifetime of the application. For example, bank account numbers or government-issued IDs aren't tied to a specific application.

The attributes of an entity can change over time. For instance, a person's name or address might change, but they remain the same individual.

#### Value Objects

Value objects. A value object has no identity. It's defined only by the values of its attributes. Value objects are immutable. To update a value object, a new instance is created to replace the old one. Value objects can include methods that encapsulate domain logic, but those methods must not produce side effects or modify the object's state. Common examples of value objects include colors, dates and times, and currency values.

#### Aggregates.

An aggregate defines a consistency boundary around one or more entities. Exactly one entity in an aggregate is the root. Lookup is done using the root entity's identifier. Any other entities in the aggregate are children of the root, and are referenced by following pointers from the root.

The purpose of an aggregate is to model transactional invariants. Things in the real world have complex webs of relationships. Customers create orders, orders contain products, products have suppliers, and so on. If the application modifies several related objects, how does it guarantee consistency? How do we keep track of invariants and enforce them?

Traditional applications have often used database transactions to enforce consistency. In a distributed application, however, that's often not feasible. A single business transaction may span multiple data stores, or may be long running, or may involve third-party services. Ultimately it's up to the application, not the data layer, to enforce the invariants required for the domain. That's what aggregates are meant to model.

An aggregate might consist of a single entity, without child entities. What makes it an aggregate is the transactional boundary.

#### Domain and Application Services.

In DDD terminology, a service is an object that implements some logic without holding any state. Evans distinguishes between domain services, which encapsulate domain logic, and application services, which provide technical functionality, such as user authentication or sending an SMS message. Domain services are often used to model behavior that spans multiple entities.

The term service is overloaded in software development. The definition used here isn't directly related to microservices.

#### Domain events.

Domain events can notify other parts of the system when something occurs. As the name suggests, domain events should represent something meaningful within the domain. For example, "a record was inserted into a table" isn't a domain event. "A delivery was canceled" is a domain event. Domain events are especially important in a microservices architecture. Because microservices are distributed and don't share data stores, domain events enable coordination between services. For more information about asynchronous messaging, see Interservice communication.

There are a few other DDD patterns not covered here, including factories, repositories, and modules. These patterns can be helpful when you implement a microservice, but they're less relevant when you design the boundaries between microservices.

### Applying the patterns - Drone delivery

We start with the scenarios that the Shipping bounded context must handle.

* A customer can request a drone to pick up goods from a business that is registered with the drone delivery service.
* The sender generates a tag (barcode or RFID) to put on the package.
* A drone will pick up and deliver a package from the source location to the destination location.
* When a customer schedules a delivery, the system provides an ETA based on route information, weather conditions, and historical data.
* When the drone is in flight, a user can track the current location and the latest ETA.
* Until a drone has picked up the package, the customer can cancel a delivery.
* The customer is notified when the delivery is completed.
* The sender can request delivery confirmation from the customer, in the form of a signature or finger print.
* Users can look up the history of a completed delivery.

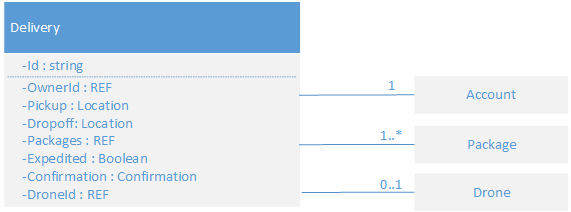
From these scenarios, the development team identified the following entities.

* Delivery
* Package
* Drone
* Account
* Confirmation
* Notification
* Tag

The first four, Delivery, Package, Drone, and Account, are all aggregates that represent transactional consistency boundaries. Confirmations and Notifications are child entities of Deliveries, and Tags are child entities of Packages.

The value objects in this design include Location, ETA, PackageWeight, and PackageSize.

To illustrate, here is a UML diagram of the Delivery aggregate. Notice that it holds references to other aggregates, including Account, Package, and Drone.

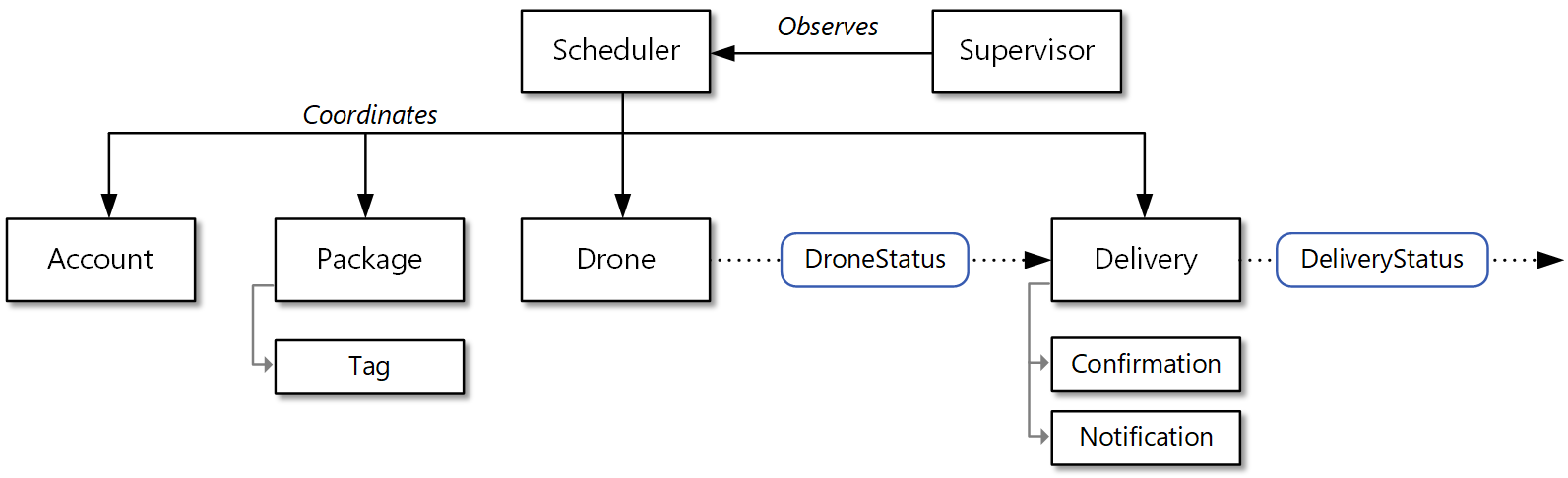


There are two domain events:

* While a drone is in flight, the Drone entity sends DroneStatus events that describe the drone's location and status (in-flight, landed).
* The Delivery entity sends DeliveryTracking events whenever the stage of a delivery changes. The DeliveryTracking events include DeliveryCreated, DeliveryRescheduled, DeliveryHeadedToDropoff, and DeliveryCompleted.

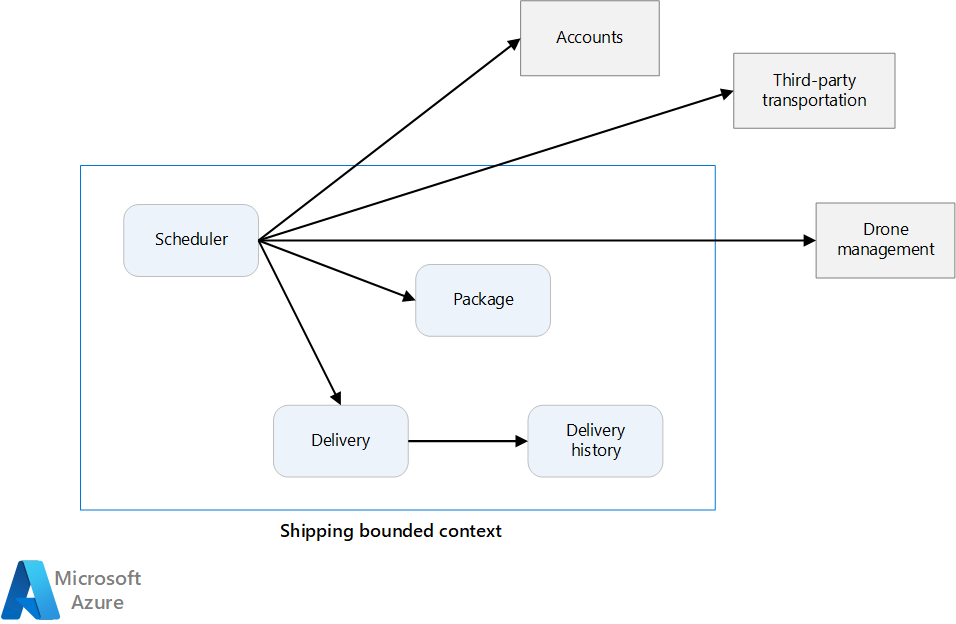
Notice that these events describe things that are meaningful within the domain model. They describe something about the domain, and aren't tied to a particular programming language construct.

The development team identified one more area of functionality, which doesn't fit neatly into any of the entities described so far. Some part of the system must coordinate all of the steps involved in scheduling or updating a delivery. Therefore, the development team added two domain services to the design: a Scheduler that coordinates the steps, and a Supervisor that monitors the status of each step, in order to detect whether any steps failed or timed out. This approach is a variation of the Scheduler Agent Supervisor pattern.



### Identify Microservice Boundaries

Architects and developers struggle to define the correct size for a microservice. Guidance often emphasizes avoiding extremes of too large or too small, but that advice can be vague in practice. But if you start from a carefully designed domain model, you can more easily define the correct size and scope of each microservice.



This article uses a drone delivery service as a running example. You can read more about the scenario and the corresponding reference implementation [here](https://learn.microsoft.com/en-us/azure/architecture/microservices/design/).

From domain model to microservices

In the previous article, we defined a set of bounded contexts for a Drone Delivery application. Then we looked more closely at one of these bounded contexts, the Shipping bounded context, and identified a set of entities, aggregates, and domain services for that bounded context.

Now we're ready to go from domain model to application design. Here's an approach that you can use to derive microservices from the domain model.

1. Start with a bounded context. In general, the functionality in a microservice shouldn't span more than one bounded context. By definition, a bounded context marks the boundary of a specific domain model. If you find that a microservice mixes different domain models together, you might need to go back and refine your domain analysis.
2. Next, look at the aggregates in your domain model. Aggregates are often good candidates for microservices. A well-designed aggregate exhibits many of the characteristics of a well-designed microservice:

An aggregate is derived from business requirements, rather than technical concerns such as data access or messaging.

An aggregate should have high functional cohesion.

An aggregate is a boundary of persistence.

Aggregates should be loosely coupled.

1. Domain services are also good candidates for microservices. Domain services are stateless operations across multiple aggregates. A typical example is a workflow that includes several microservices. The Drone Delivery application shows an example.
2. Consider nonfunctional requirements. Look at factors such as team size, data types, technologies, scalability requirements, availability requirements, and security requirements. These factors might cause you to break a microservice into multiple smaller services. In other cases, they might cause you to merge several microservices into a single microservice.

After you identify the microservices in your application, validate your design against the following criteria:

* Each service has a single responsibility.
* There are no chatty calls between services. If splitting functionality into two services causes them to be overly chatty, it might be a symptom that these functions belong in the same service.
* Each service is small enough that it can be built by a small team working independently.
* There are no interdependencies that require two or more services to be deployed together. Each service should be deployable independently, without needing to redeploy others.
* Services aren't tightly coupled and can evolve independently.
* Service boundaries are designed to avoid problems with data consistency or integrity. In some cases, maintaining data consistency means grouping related functionality into a single microservice. However, strong consistency isn't always required. Distributed systems provide strategies for handling eventual consistency, and the benefits of decomposing services often outweigh the complexity of managing it.

Above all, it's important to be pragmatic, and remember that domain-driven design is an iterative process. When in doubt, start with more coarse-grained microservices. Splitting a microservice into two smaller services is easier than refactoring functionality across several existing microservices.

Example: Defining microservices for the Drone Delivery application

The development team previously identified the four aggregates, Delivery, Package, Drone, and Account, and two domain services, Scheduler and Supervisor.

Delivery and Package are obvious candidates for microservices. The Scheduler and Supervisor coordinate the activities performed by other microservices, so it makes sense to implement these domain services as microservices.

Drone and Account are interesting because they belong to other bounded contexts. One option is for the Scheduler to call the Drone and Account bounded contexts directly. Another option is to create Drone and Account microservices inside the Shipping bounded context. These microservices would mediate between the bounded contexts, by exposing APIs or data schemas that are more suited to the Shipping context.

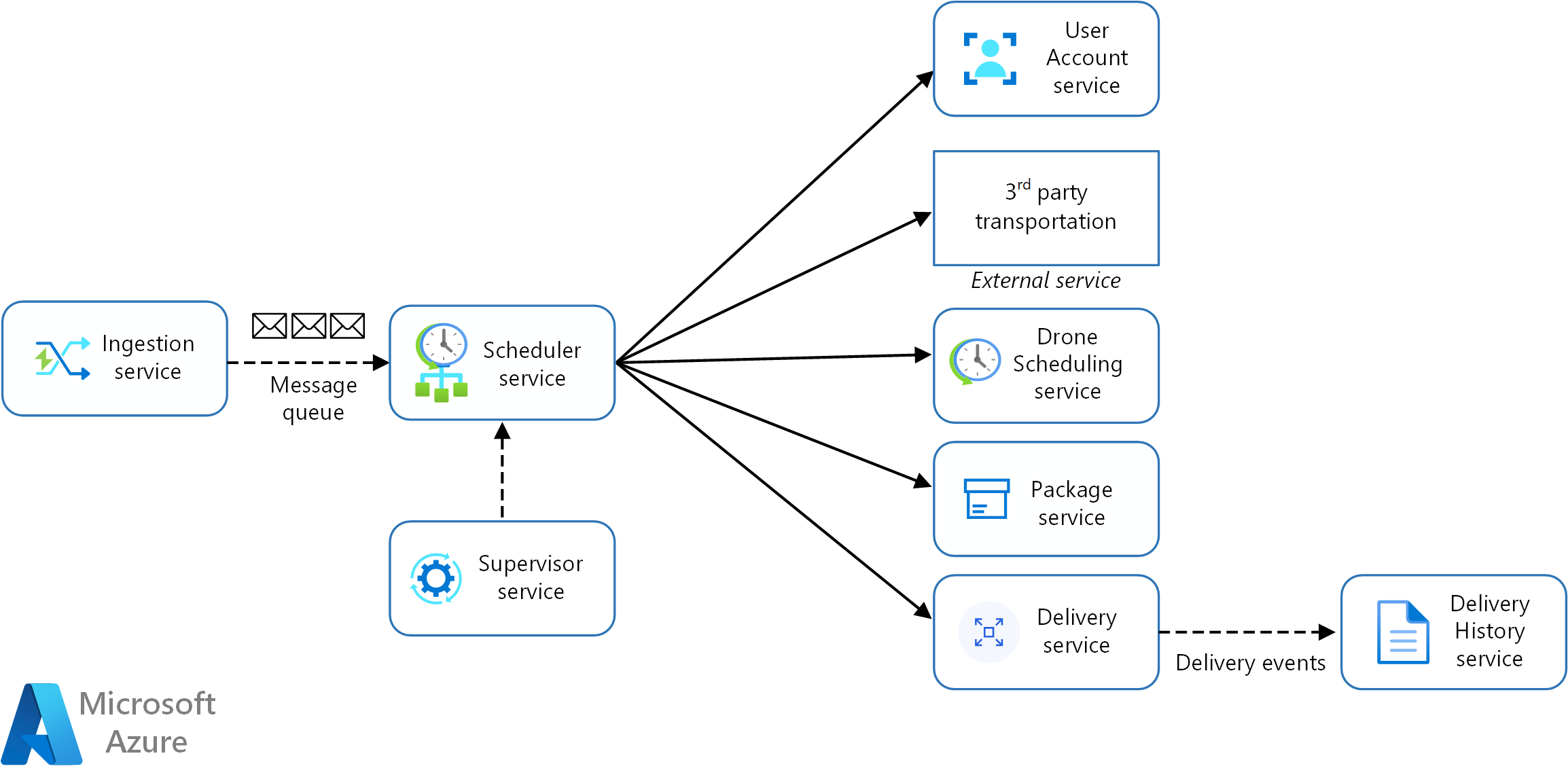
The details of the Drone and Account bounded contexts are beyond the scope of this guidance, so we created mock services for them in our reference implementation. But here are some factors to consider in this situation:

* What is the network overhead of calling directly into the other bounded context?
* Is the data schema for the other bounded context suitable for this context, or is it better to have a schema tailored to this bounded context?
* Is the other bounded context a legacy system? If so, you might create a service that acts as an anti-corruption layer to translate between the legacy system and the modern application.
* What is the team structure? Is it easy to communicate with the team responsible for the other bounded context? If not, creating a service that mediates between the two contexts can help to mitigate the cost of cross-team communication.

So far, the team hasn't considered any nonfunctional requirements. After evaluating the application's throughput needs, the development team creates a separate Ingestion microservice to handle client requests. This microservice implements load leveling by placing incoming requests into a buffer for processing. The Scheduler then reads requests from the buffer and implements the workflow.

Nonfunctional requirements also lead the team to create one more service. The existing services focus on scheduling and delivering packages in real time. However, the system must also store delivery history in long-term storage for data analysis. Initially, the team considered making this task part of the Delivery service. But the data storage requirements for historical analysis differ from the requirements for in-flight operations. For more information, see Data considerations. As a result, the team created a separate Delivery History service. This service listens for DeliveryTracking events from the Delivery service and writes them to long-term storage.

The following diagram shows the design at this point:



At this point, you should have a clear understanding of the purpose and functionality of each microservice in your design. Now you can architect the system.

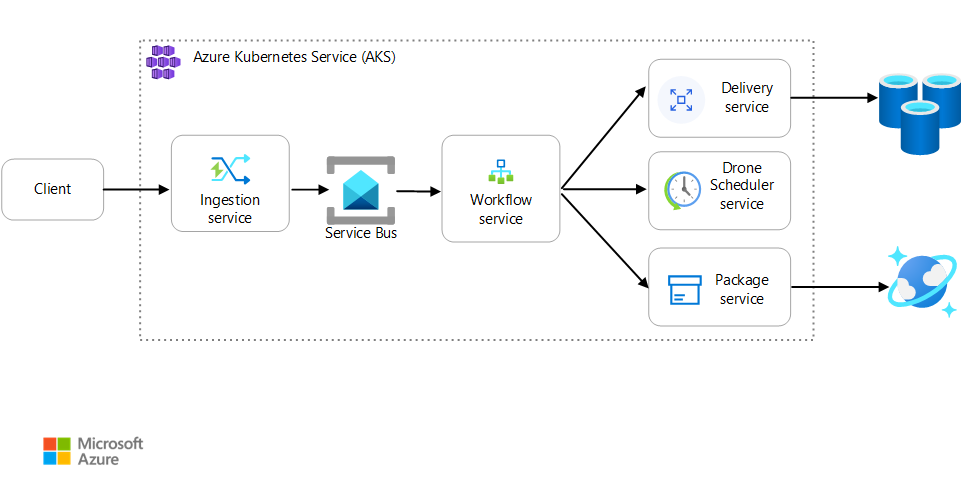
## Design a microservices architecture

Microservices have become a popular architectural style for building cloud applications that are resilient, highly scalable, independently deployable, and able to evolve quickly. To be more than just a buzzword, however, microservices require a different approach to designing and building applications.

In this set of articles, we explore how to build a microservices architecture on Azure. Topics include:

1. Compute options for microservices
2. Interservice communication
3. API design
4. API gateways
5. Data considerations
6. Design patterns

Example Architecture



Fabrikam, Inc. is starting a drone delivery service. The company manages a fleet of drone aircraft. Businesses register with the service, and users can request a drone to pick up goods for delivery. When a customer schedules a pickup, a backend system assigns a drone and notifies the user with an estimated delivery time. While the delivery is in progress, the customer can track the location of the drone, with a continuously updated ETA.

This solution is ideal for the aerospace and aircraft industries.

This scenario involves a fairly complicated domain. Some of the business concerns include scheduling drones, tracking packages, managing user accounts, and storing and analyzing historical data. Moreover, Fabrikam wants to get to market quickly and then iterate quickly, adding new functionality and capabilities. The application needs to operate at cloud scale, with a high service level objective (SLO). Fabrikam also expects that different parts of the system will have very different requirements for data storage and querying. All of these considerations lead Fabrikam to choose a microservices architecture for the Drone Delivery application.

For help in choosing between a microservices architecture and other architectural styles, see the [Azure Application Architecture Guide](https://learn.microsoft.com/en-us/azure/architecture/guide/).

This architecture uses Kubernetes with [Azure Kubernetes Service (AKS)](https://learn.microsoft.com/en-us/azure/aks/). However, many of the high-level architectural decisions and challenges will apply to any container orchestrator.

#### Azure compute option

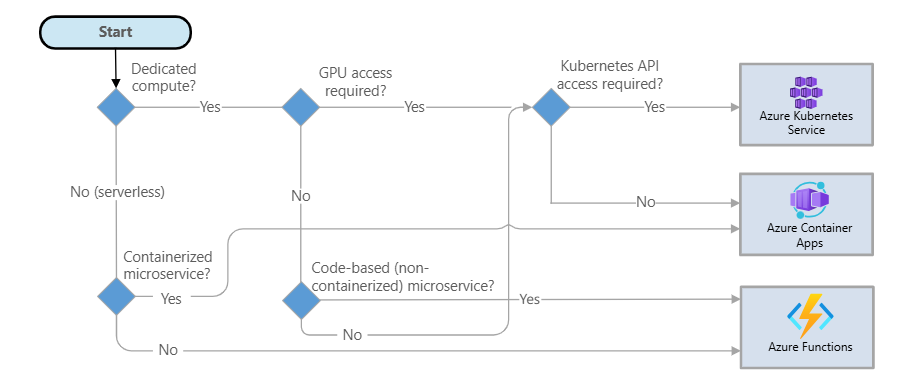
Choose an Azure compute option for microservices

The term compute refers to the hosting model for the computing resources that your application runs on. This article provides prescriptive guidance to help you choose a compute platform for microservices. Your microservice compute platform selection might depend on more nuanced requirements.

For a microservices architecture, the following approaches are popular:

* Deploy microservices on dedicated compute platforms, typically by using a microservice orchestrator.
* Deploy microservices on a serverless platform.

Although these options aren't the only ones, they're both proven approaches to building microservices. An application might include both approaches.



##### Serverless platform

You can use serverless platforms to deploy microservices on Azure Container Apps or Azure Functions. Both Container Apps and Functions provide serverless compute options that bill based on the volume of requests rather than compute consumption. Both platforms also give you the option to host the workloads on dedicated capacity.

##### Deploy code-based microservices

If you want to deploy your microservices as code instead of containerizing them, you might want to use Azure Functions. For more information, see the list of programming and scripting languages supported by Functions. For microservices that you develop in other languages, you might want to implement a custom handler in Functions or consider containerizing the application.

##### GPU model

If your microservice requires GPU capacity, for example, to run machine learning tasks, consider choosing Container Apps or Azure Kubernetes Service (AKS) for your platform. AKS can use any GPU models in Azure, and Container Apps offers a subset of GPU models to choose from.

##### Service Orchestrators

An orchestrator handles tasks that relate to deploying and managing a set of services. These tasks include placing services on nodes, monitoring the health of services, restarting unhealthy services, load balancing network traffic across service instances, service discovery, scaling the number of instances of a service, and applying configuration updates. Popular orchestrators include Kubernetes, Azure Service Fabric, DC/OS, and Docker Swarm.

On the Azure platform, consider the following options:

* Azure Kubernetes Service (AKS) is a managed Kubernetes service. AKS provisions Kubernetes and exposes the Kubernetes API endpoints, hosts and manages the Kubernetes control plane, and performs automated upgrades, automated patching, autoscaling, and other management tasks. AKS provides direct access to Kubernetes APIs.
* Container Apps is a managed service built on Kubernetes that abstracts the complexities of container orchestration and other management tasks. Container Apps simplifies the deployment and management of containerized applications and microservices in a serverless environment while providing the features of Kubernetes. Container Apps is ideal for scenarios where direct access to Kubernetes APIs isn't required.
* Service Fabric is a distributed systems platform for packaging, deploying, and managing microservices. You can deploy microservices to Service Fabric as containers, as binary executables, or as Reliable Services. By using the Reliable Services programming model, services can directly use Service Fabric programming APIs to query the system, report health, receive notifications about configuration and code changes, and discover other services.
* Use Azure Red Hat OpenShift to deploy fully managed OpenShift clusters. Azure Red Hat OpenShift extends Kubernetes. Azure Red Hat OpenShift is jointly engineered, operated, and supported by Red Hat and Microsoft.
* Other options, such as Docker Enterprise Edition, can run in a cloud-computing environment on Azure. You can find deployment templates on Azure Marketplace.

##### Use Kubernetes APIs

Access to Kubernetes APIs is often a deciding factor when you choose a compute option. AKS provides direct access to Kubernetes APIs, but Container Apps doesn't. Container Apps hides the complexities of Kubernetes and simplifies the container deployment experience. If you design your microservice deployment to directly interact with Kubernetes APIs, AKS might be the right choice.

##### Other decision factors

There might be other factors that affect your microservice compute platform selection. These factors include service mesh options, platform scalability, and skill sets that you might use within the organization.

###### Considerations

These considerations implement the pillars of the Azure Well-Architected Framework, which is a set of guiding tenets that can be used to improve the quality of a workload. For more information, see Microsoft Azure Well-Architected Framework.

###### Reliability

Reliability ensures your application can meet the commitments you make to your customers. For more information, see Design review checklist for Reliability.

One of the key pillars of reliability is resiliency. The goal of resiliency is to return the workload to a fully functioning state after a failure occurs.

If you choose Azure Functions as your microservice computing platform, consider deploying the Functions Premium plan or Azure App Service plan in a zone-redundant configuration. For more information, see Reliability in Functions.

If you choose AKS as your microservice computing platform, you can enhance microservice reliability by deploying an AKS cluster that uses availability zones, by using the Standard or Premium tier for Azure Kubernetes clusters, and by increasing the minimum number of pods and nodes. For more information, see Deployment and cluster reliability best practices for AKS.

If you choose Container Apps as your microservice computing platform, you can enhance reliability by using availability zones. For more information, see Reliability in Container Apps.

###### Security

Security provides assurances against deliberate attacks and the abuse of your valuable data and systems. For more information, see Design review checklist for Security.

If you choose Azure Functions as your compute platform to deploy microservices, the principles of securing Azure Functions apply to microservices as well.

If you choose AKS as your compute platform to deploy microservices, the AKS security baseline architecture provides guidance for securing the compute platform. For best practices on microservice security on AKS, see Advanced AKS microservice architecture.

If you choose Container Apps as your compute platform to deploy microservices, see the security baseline for Container Apps for security best practices.

###### Cost Optimization

Cost Optimization is about looking at ways to reduce unnecessary expenses and improve operational efficiencies. For more information, see Design review checklist for Cost Optimization.

When you use an orchestrator, you pay for the virtual machines that run in the cluster. When you use a serverless application, you pay only for the actual compute resources that you consume. In both cases, you need to factor in the cost of any extra services, such as storage, databases, and messaging services.

Azure Functions, Container Apps, and AKS provide autoscaling options. Container Apps and Functions provide serverless platforms where the cost is based on consumption and can be zero. AKS provides only dedicated compute options.

If you choose AKS as the compute platform to deploy microservices, you need to understand cost optimization best practices. For more information, see Optimize costs in Azure Kubernetes Service.

If you choose Container Apps as your microservices compute platform, you need to understand the various billing models and decide on the deployment model for your microservices based on your workload requirements. For more information, see Billing in Container Apps.

If you choose Azure Functions as your microservices compute platform, you need to understand the various billing models and decide on the Functions plan based on your workload requirements. For more information, see Estimate consumption-based costs and Azure Functions plan details.

##### Operational Excellence

Operational Excellence covers the operations processes that deploy an application and keep it running in production. For more information, see Design review checklist for Operational Excellence.

You can deploy all of the microservice compute choices that this article describes in an automated manner by using Terraform, Bicep, and other scripting languages. You can use Application Insights, Azure Monitor, and other monitoring solutions to monitor these compute platforms and microservices.

Consider the following factors when you choose between an orchestrator approach and a serverless approach:

* Flexibility and control: An orchestrator gives you control over configuring and managing your services and the cluster. The trade-off is more complexity. With a serverless architecture, you give up some degree of control because these details are abstracted.
* Portability: All of the orchestrators listed in this article, including Kubernetes, DC/OS, Docker Swarm, and Service Fabric, can run on-premises or in multiple public clouds.
* Application integration: It can be challenging to build a complex application that uses a serverless architecture because you need to coordinate, deploy, and manage many small, independent functions. One option in Azure is to use Azure Logic Apps to coordinate a set of Azure functions. For an example of this approach, see Create a function that integrates with Logic Apps.

#### Inter Service Communication

Communication between microservices must be efficient and robust. With lots of small services interacting to complete a single business activity, this can be a challenge. In this article, we look at the tradeoffs between asynchronous messaging versus synchronous APIs. Then we look at some of the challenges in designing resilient interservice communication.

##### Challenges

Here are some of the main challenges arising from service-to-service communication. Service meshes, described later in this article, are designed to handle many of these challenges.

###### Resiliency

Resiliency. There may be dozens or even hundreds of instances of any given microservice. An instance can fail for any number of reasons. There can be a node-level failure, such as a hardware failure or a VM reboot. An instance might crash, or be overwhelmed with requests and unable to process any new requests. Any of these events can cause a network call to fail. There are two design patterns that can help make service-to-service network calls more resilient:

* Retry. A network call may fail because of a transient fault that goes away by itself. Rather than fail outright, the caller should typically retry the operation a certain number of times, or until a configured time-out period elapses. However, if an operation is not idempotent, retries can cause unintended side effects. The original call might succeed, but the caller never gets a response. If the caller retries, the operation may be invoked twice. Generally, it's not safe to retry POST or PATCH methods, because these are not guaranteed to be idempotent.
* Circuit Breaker. Too many failed requests can cause a bottleneck, as pending requests accumulate in the queue. These blocked requests might hold critical system resources such as memory, threads, database connections, and so on, which can cause cascading failures. The Circuit Breaker pattern can prevent a service from repeatedly trying an operation that is likely to fail.

###### Load Balancing

Load balancing. When service "A" calls service "B", the request must reach a running instance of service "B". In Kubernetes, the Service resource type provides a stable IP address for a group of pods. Network traffic to the service's IP address gets forwarded to a pod by means of iptable rules. By default, a random pod is chosen. A service mesh (see below) can provide more intelligent load balancing algorithms based on observed latency or other metrics.

###### Distributed Tracing

Distributed tracing. A single transaction may span multiple services. That can make it hard to monitor the overall performance and health of the system. Even if every service generates logs and metrics, without some way to tie them together, they are of limited use.

###### Service Versioning

Service versioning. When a team deploys a new version of a service, they must avoid breaking any other services or external clients that depend on it. In addition, you might want to run multiple versions of a service side-by-side, and route requests to a particular version. See API Versioning for more discussion of this issue.

###### TLS Encryption and Mutual TLS Authentication

TLS encryption and mutual TLS authentication. For security reasons, you may want to encrypt traffic between services with TLS, and use mutual TLS authentication to authenticate callers.

##### Synchronous versus asynchronous messaging

There are two basic messaging patterns that microservices can use to communicate with other microservices.

1. Synchronous communication. In this pattern, a service calls an API that another service exposes, using a protocol such as HTTP or gRPC. This option is a synchronous messaging pattern because the caller waits for a response from the receiver.
2. Asynchronous message passing. In this pattern, a service sends message without waiting for a response, and one or more services process the message asynchronously.

It's important to distinguish between asynchronous I/O and an asynchronous protocol. Asynchronous I/O means the calling thread is not blocked while the I/O completes. That's important for performance, but is an implementation detail in terms of the architecture. An asynchronous protocol means the sender doesn't wait for a response. HTTP is a synchronous protocol, even though an HTTP client may use asynchronous I/O when it sends a request.

There are tradeoffs to each pattern. Request/response is a well-understood paradigm, so designing an API may feel more natural than designing a messaging system. However, asynchronous messaging has some advantages that can be useful in a microservices architecture:

* **Reduced coupling**. The message sender does not need to know about the consumer.
* **Multiple** **subscribers**. Using a pub/sub model, multiple consumers can subscribe to receive events. See Event-driven architecture style.
* **Failure** **isolation**. If the consumer fails, the sender can still send messages. The messages will be picked up when the consumer recovers. This ability is especially useful in a microservices architecture, because each service has its own lifecycle. A service could become unavailable or be replaced with a newer version at any given time. Asynchronous messaging can handle intermittent downtime. Synchronous APIs, on the other hand, require the downstream service to be available or the operation fails.
* **Responsiveness**. An upstream service can reply faster if it does not wait on downstream services. This is especially useful in a microservices architecture. If there is a chain of service dependencies (service A calls B, which calls C, and so on), waiting on synchronous calls can add unacceptable amounts of latency.
* **Load** **leveling**. A queue can act as a buffer to level the workload, so that receivers can process messages at their own rate.
* **Workflows**. Queues can be used to manage a workflow, by check-pointing the message after each step in the workflow.

However, there are also some challenges to using asynchronous messaging effectively.

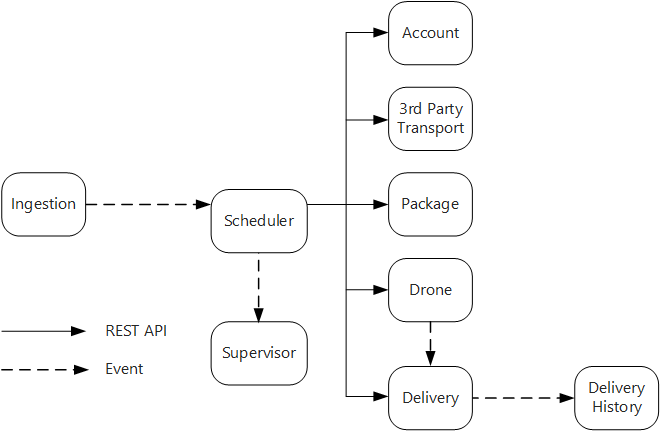
* **Coupling with the messaging infrastructure**. Using a particular messaging infrastructure may cause tight coupling with that infrastructure. It will be difficult to switch to another messaging infrastructure later.
* **Latency**. End-to-end latency for an operation may become high if the message queues fill up.
* **Cost**. At high throughputs, the monetary cost of the messaging infrastructure could be significant.
* **Complexity**. Handling asynchronous messaging is not a trivial task. For example, you must handle duplicated messages, either by de-duplicating or by making operations idempotent. It's also hard to implement request-response semantics using asynchronous messaging. To send a response, you need another queue, plus a way to correlate request and response messages.
* **Throughput**. If messages require queue semantics, the queue can become a bottleneck in the system. Each message requires at least one queue operation and one dequeue operation. Moreover, queue semantics generally require some kind of locking inside the messaging infrastructure. If the queue is a managed service, there may be additional latency, because the queue is external to the cluster's virtual network. You can mitigate these issues by batching messages, but that complicates the code. If the messages don't require queue semantics, you might be able to use an event stream instead of a queue. For more information, see Event-driven architectural style.

##### Drone Delivery: Choosing the messaging patterns

This solution uses the Drone Delivery example. It's ideal for the aerospace and aircraft industries.

With these considerations in mind, the development team made the following design choices for the Drone Delivery application:

* The Ingestion service exposes a public REST API that client applications use to schedule, update, or cancel deliveries.
* The Ingestion service uses Event Hubs to send asynchronous messages to the Scheduler service. Asynchronous messages are necessary to implement the load-leveling that is required for ingestion.
* The Account, Delivery, Package, Drone, and Third-party Transport services all expose internal REST APIs. The Scheduler service calls these APIs to carry out a user request. One reason to use synchronous APIs is that the Scheduler needs to get a response from each of the downstream services. A failure in any of the downstream services means the entire operation failed. However, a potential issue is the amount of latency that is introduced by calling the backend services.
* If any downstream service has a nontransient failure, the entire transaction should be marked as failed. To handle this case, the Scheduler service sends an asynchronous message to the Supervisor, so that the Supervisor can schedule compensating transactions.
* The Delivery service exposes a public API that clients can use to get the status of a delivery. In the article API gateway, we discuss how an API gateway can hide the underlying services from the client, so the client doesn't need to know which services expose which APIs.
* While a drone is in flight, the Drone service sends events that contain the drone's current location and status. The Delivery service listens to these events in order to track the status of a delivery.
* When the status of a delivery changes, the Delivery service sends a delivery status event, such as DeliveryCreated or DeliveryCompleted. Any service can subscribe to these events. In the current design, the Delivery History service is the only subscriber, but there might be other subscribers later. For example, the events might go to a real-time analytics service. And because the Scheduler doesn't have to wait for a response, adding more subscribers doesn't affect the main workflow path.



Notice that delivery status events are derived from drone location events. For example, when a drone reaches a delivery location and drops off a package, the Delivery service translates this into a DeliveryCompleted event. This is an example of thinking in terms of domain models. As described earlier, Drone Management belongs in a separate bounded context. The drone events convey the physical location of a drone. The delivery events, on the other hand, represent changes in the status of a delivery, which is a different business entity.

##### Using a service mesh

A service mesh is a software layer that handles service-to-service communication. Service meshes are designed to address many of the concerns listed in the previous section, and to move responsibility for these concerns away from the microservices themselves and into a shared layer. The service mesh acts as a proxy that intercepts network communication between microservices in the cluster. Currently, the service mesh concept applies mainly to container orchestrators, rather than serverless architectures.

Service mesh is an example of the [Ambassador pattern](https://learn.microsoft.com/en-us/azure/architecture/patterns/ambassador) — a helper service that sends network requests on behalf of the application.

* Right now, the main options for a service mesh in Kubernetes are [Linkerd](https://linkerd.io/) and [Istio](https://istio.io/). Both of these technologies are evolving rapidly. However, some features that both Linkerd and Istio have in common include:
* Load balancing at the session level, based on observed latencies or number of outstanding requests. This can improve performance over the layer-4 load balancing that is provided by Kubernetes.
* Layer-7 routing based on URL path, Host header, API version, or other application-level rules.
* Retry of failed requests. A service mesh understands HTTP error codes, and can automatically retry failed requests. You can configure that maximum number of retries, along with a timeout period in order to bound the maximum latency.
* Circuit breaking. If an instance consistently fails requests, the service mesh will temporarily mark it as unavailable. After a backoff period, it will try the instance again. You can configure the circuit breaker based on various criteria, such as the number of consecutive failures,
* Service mesh captures metrics about interservice calls, such as the request volume, latency, error and success rates, and response sizes. The service mesh also enables distributed tracing by adding correlation information for each hop in a request.
* Mutual TLS Authentication for service-to-service calls.

Do you need a service mesh? It depends. Without a service mesh, you'll need to consider each of the challenges mentioned at the beginning of this article. You can solve problems like retry, circuit breaker, and distributed tracing without a service mesh, but a service mesh moves these concerns out of the individual services and into a dedicated layer. On the other hand, a service mesh adds complexity to the setup and configuration of the cluster. There may be performance implications, because requests now get routed through the service mesh proxy, and because extra services are now running on every node in the cluster. You should do thorough performance and load testing before deploying a service mesh in production.

##### Distributed Transactions

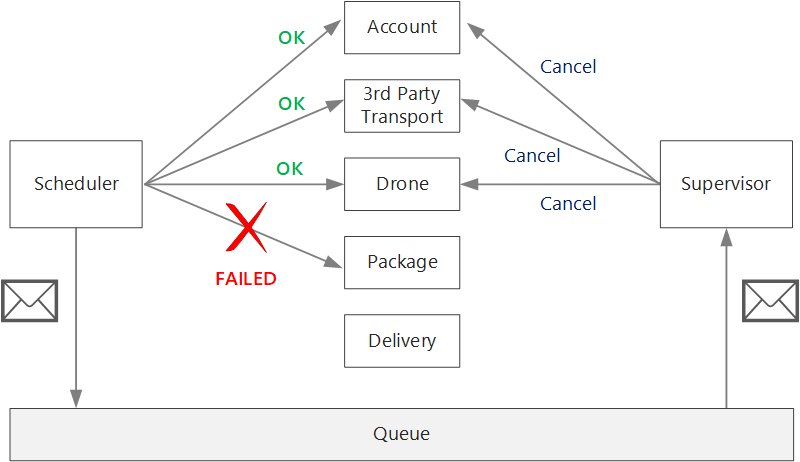
A common challenge in microservices is correctly handling transactions that span multiple services. Often in this scenario, the success of a transaction is all or nothing — if one of the participating services fails, the entire transaction must fail.

There are two cases to consider:

* A service may experience a transient failure such as a network timeout. These errors can often be resolved simply by retrying the call. If the operation still fails after a certain number of attempts, it's considered a nontransient failure.
* A nontransient failure is any failure that's unlikely to go away by itself. Nontransient failures include normal error conditions, such as invalid input. They also include unhandled exceptions in application code or a process crashing. If this type of error occurs, the entire business transaction must be marked as a failure. It may be necessary to undo other steps in the same transaction that already succeeded.

After a nontransient failure, the current transaction might be in a partially failed state, where one or more steps already completed successfully. For example, if the Drone service already scheduled a drone, the drone must be canceled. In that case, the application needs to undo the steps that succeeded, by using a [Compensating Transaction](https://learn.microsoft.com/en-us/azure/architecture/patterns/compensating-transaction). In some cases, this action must be done by an external system or even by a manual process. In your design, remember that compensating measures are also subject to failure.

If the logic for compensating transactions is complex, consider creating a separate service that is responsible for this process. In the Drone Delivery application, the Scheduler service puts failed operations onto a dedicated queue. A separate microservice, called the Supervisor, reads from this queue and calls a cancellation API on the services that need to compensate. This is a variation of the [Scheduler Agent Supervisor pattern](https://learn.microsoft.com/en-us/azure/architecture/patterns/scheduler-agent-supervisor). The Supervisor service might take other actions as well, such as notify the user by text or email, or send an alert to an operations dashboard.



The Scheduler service itself might fail (for example, because a node crashes). In that case, a new instance can spin up and take over. However, any transactions that were already in progress must be resumed.

One approach is to save a checkpoint to a durable store after each step in the workflow is completed. If an instance of the Scheduler service crashes in the middle of a transaction, a new instance can use the checkpoint to resume where the previous instance left off. However, writing checkpoints can create a performance overhead.

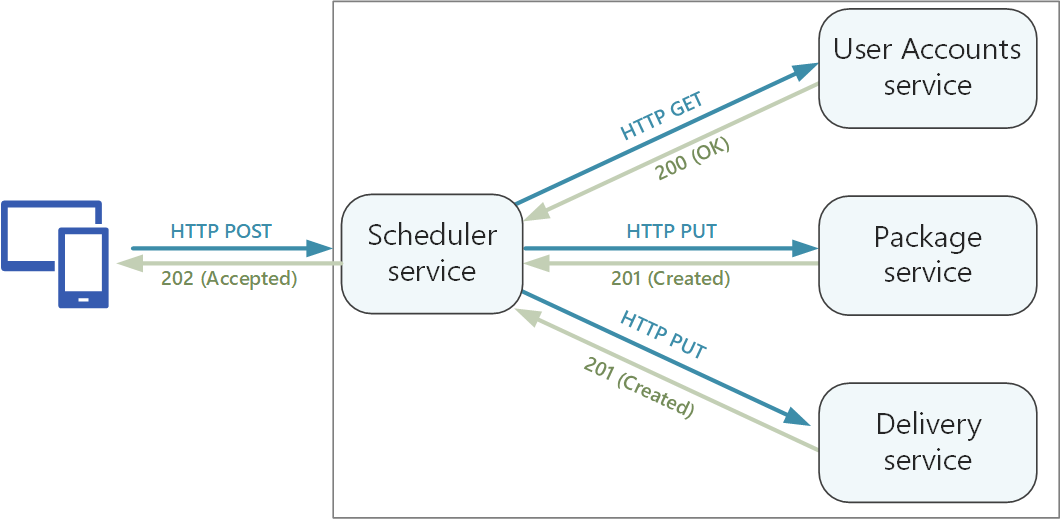
Another option is to design all operations to be idempotent. An operation is idempotent if it can be called multiple times without producing additional side-effects after the first call. Essentially, the downstream service should ignore duplicate calls, which means the service must be able to detect duplicate calls. It's not always straightforward to implement idempotent methods. For more information, see [Idempotent operations](https://learn.microsoft.com/en-us/azure/architecture/microservices/design/api-design#idempotent-operations).

For microservices that talk directly to each other, it's important to create well-designed APIs.

#### API Design

Design APIs for Microservices

Good API design is important in a microservices architecture, because all data exchange between services happens either through messages or API calls. APIs must be efficient to avoid creating chatty I/O. Because services are designed by teams working independently, APIs must have well-defined semantics and versioning schemes, so that updates don't break other services.



It's important to distinguish between two types of API:

* Public APIs that client applications call.
* Backend APIs that are used for interservice communication.

These two use cases have somewhat different requirements. A public API must be compatible with client applications, typically browser applications or native mobile applications. Most of the time, that means the public API will use REST over HTTP. For the backend APIs, however, you need to take network performance into account. Depending on the granularity of your services, interservice communication can result in a lot of network traffic. Services can quickly become I/O bound. For that reason, considerations such as serialization speed and payload size become more important. Some popular alternatives to using REST over HTTP include gRPC, Apache Avro, and Apache Thrift. These protocols support binary serialization and are generally more efficient than HTTP.

##### Considerations

Here are some things to think about when choosing how to implement an API.

REST versus RPC. Consider the tradeoffs between using a REST-style interface versus an RPC-style interface.

* REST models resources, which can be a natural way to express your domain model. It defines a uniform interface based on HTTP verbs, which encourages evolvability. It has well-defined semantics in terms of idempotency, side effects, and response codes. And it enforces stateless communication, which improves scalability.
* RPC is more oriented around operations or commands. Because RPC interfaces look like local method calls, it may lead you to design overly chatty APIs. However, that doesn't mean RPC must be chatty. It just means you need to use care when designing the interface.

For a RESTful interface, the most common choice is REST over HTTP using JSON. For an RPC-style interface, there are several popular frameworks, including gRPC, Apache Avro, and Apache Thrift.

Efficiency. Consider efficiency in terms of speed, memory, and payload size. Typically a gRPC-based interface is faster than REST over HTTP.

Interface definition language (IDL). An IDL is used to define the methods, parameters, and return values of an API. An IDL can be used to generate client code, serialization code, and API documentation. IDLs can also be consumed by API testing tools. Frameworks such as gRPC, Avro, and Thrift define their own IDL specifications. REST over HTTP does not have a standard IDL format, but a common choice is OpenAPI (formerly Swagger). You can also create an HTTP REST API without using a formal definition language, but then you lose the benefits of code generation and testing.

Serialization. How are objects serialized over the wire? Options include text-based formats (primarily JSON) and binary formats such as protocol buffer. Binary formats are generally faster than text-based formats. However, JSON has advantages in terms of interoperability, because most languages and frameworks support JSON serialization. Some serialization formats require a fixed schema, and some require compiling a schema definition file. In that case, you'll need to incorporate this step into your build process.

Framework and language support. HTTP is supported in nearly every framework and language. gRPC, Avro, and Thrift all have libraries for C++, C#, Java, and Python. Thrift and gRPC also support Go.

Compatibility and interoperability. If you choose a protocol like gRPC, you may need a protocol translation layer between the public API and the back end. A gateway can perform that function. If you are using a service mesh, consider which protocols are compatible with the service mesh. For example, Linkerd has built-in support for HTTP, Thrift, and gRPC.

Our baseline recommendation is to choose REST over HTTP unless you need the performance benefits of a binary protocol. REST over HTTP requires no special libraries. It creates minimal coupling, because callers don't need a client stub to communicate with the service. There are rich ecosystems of tools to support schema definitions, testing, and monitoring of RESTful HTTP endpoints. Finally, HTTP is compatible with browser clients, so you don't need a protocol translation layer between the client and the backend.

However, if you choose REST over HTTP, you should do performance and load testing early in the development process, to validate whether it performs well enough for your scenario.

##### RESTful API design

There are many resources for designing RESTful APIs. Here are some that you might find helpful:

[API design](https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design)

[API implementation](https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-implementation)

[Microsoft REST API Guidelines](https://github.com/Microsoft/api-guidelines)

Here are some specific considerations to keep in mind.

* Watch out for APIs that leak internal implementation details or simply mirror an internal database schema. The API should model the domain. It's a contract between services, and ideally should only change when new functionality is added, not just because you refactored some code or normalized a database table.
* Different types of client, such as mobile application and desktop web browser, may require different payload sizes or interaction patterns. Consider using the [Backends for Frontends pattern](https://learn.microsoft.com/en-us/azure/architecture/patterns/backends-for-frontends) to create separate backends for each client, which expose an optimal interface for that client.
* For operations with side effects, consider making them idempotent and implementing them as PUT methods. That will enable safe retries and can improve resiliency. The article Interservice communication discuss this issue in more detail.
* HTTP methods can have asynchronous semantics, where the method returns a response immediately, but the service carries out the operation asynchronously. In that case, the method should return an [HTTP 202](https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html) response code, which indicates the request was accepted for processing, but the processing is not yet completed. For more information, see [Asynchronous Request-Reply pattern](https://learn.microsoft.com/en-us/azure/architecture/patterns/async-request-reply).

##### Mapping REST to DDD patterns

Patterns such as entity, aggregate, and value object are designed to place certain constraints on the objects in your domain model. In many discussions of DDD, the patterns are modeled using object-oriented (OO) language concepts like constructors or property getters and setters. For example, value objects are supposed to be immutable. In an OO programming language, you would enforce this by assigning the values in the constructor and making the properties read-only:

typescript

export class Location {

readonly latitude: number;

readonly longitude: number;

constructor(latitude: number, longitude: number) {

if (latitude < -90 || latitude > 90) {

throw new RangeError('latitude must be between -90 and 90');

}

if (longitude < -180 || longitude > 180) {

throw new RangeError('longitude must be between -180 and 180');

}

this.latitude = latitude;

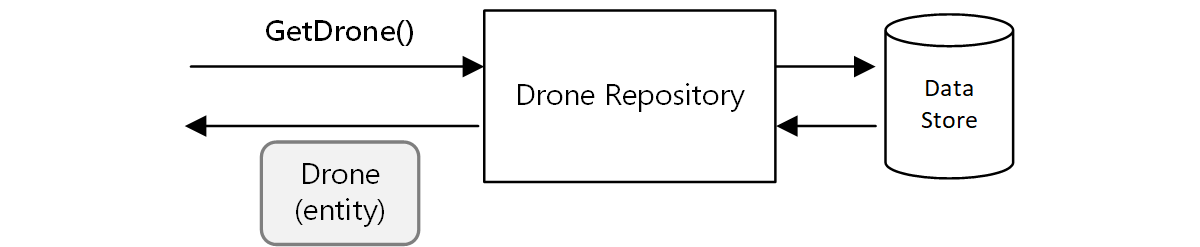
this.longitude = longitude;

}

}

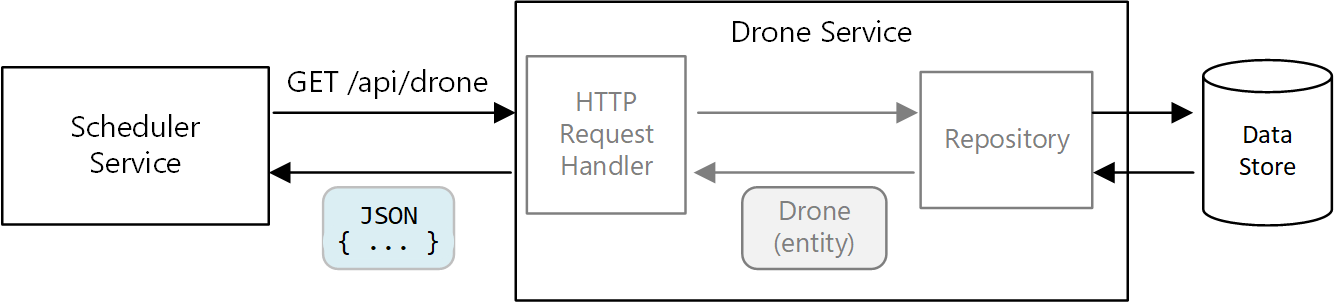
These sorts of coding practices are particularly important when building a traditional monolithic application. With a large code base, many subsystems might use the Location object, so it's important for the object to enforce correct behavior.

Another example is the Repository pattern, which ensures that other parts of the application do not make direct reads or writes to the data store:



In a microservices architecture, however, services don't share the same code base and don't share data stores. Instead, they communicate through APIs. Consider the case where the Scheduler service requests information about a drone from the Drone service. The Drone service has its internal model of a drone, expressed through code. But the Scheduler doesn't see that. Instead, it gets back a representation of the drone entity — perhaps a JSON object in an HTTP response.

This example is ideal for the aircraft and aerospace industries.



The Scheduler service can't modify the Drone service's internal models, or write to the Drone service's data store. That means the code that implements the Drone service has a smaller exposed surface area, compared with code in a traditional monolith. If the Drone service defines a Location class, the scope of that class is limited — no other service will directly consume the class.

For these reasons, this guidance doesn't focus much on coding practices as they relate to the tactical DDD patterns. But it turns out that you can also model many of the DDD patterns through REST APIs.

For example:

* Aggregates map naturally to resources in REST. For example, the Delivery aggregate would be exposed as a resource by the Delivery API.
* Aggregates are consistency boundaries. Operations on aggregates should never leave an aggregate in an inconsistent state. Therefore, you should avoid creating APIs that allow a client to manipulate the internal state of an aggregate. Instead, favor coarse-grained APIs that expose aggregates as resources.
* Entities have unique identities. In REST, resources have unique identifiers in the form of URLs. Create resource URLs that correspond to an entity's domain identity. The mapping from URL to domain identity may be opaque to client.
* Child entities of an aggregate can be reached by navigating from the root entity. If you follow [HATEOAS](https://en.wikipedia.org/wiki/HATEOAS) principles, child entities can be reached via links in the representation of the parent entity.
* Because value objects are immutable, updates are performed by replacing the entire value object. In REST, implement updates through PUT or PATCH requests.
* A repository lets clients query, add, or remove objects in a collection, abstracting the details of the underlying data store. In REST, a collection can be a distinct resource, with methods for querying the collection or adding new entities to the collection.

When you design your APIs, think about how they express the domain model, not just the data inside the model, but also the business operations and the constraints on the data.

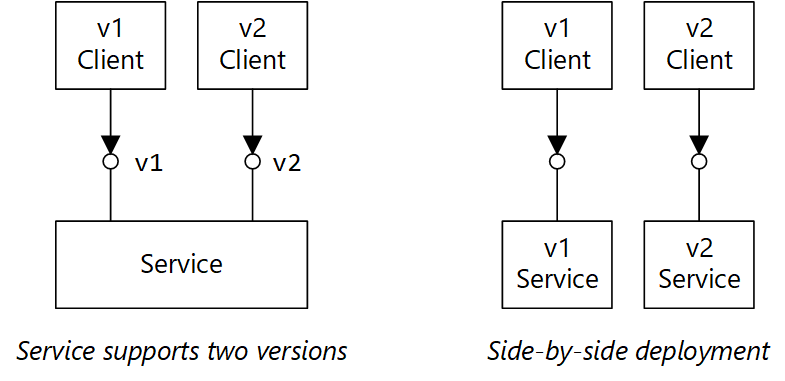
| **DDD concept** | **REST equivalent** | **Example** |
| --- | --- | --- |
| Aggregate | Resource | { "1":1234, "status":"pending"... } |
| Identity | URL | https://delivery-service/deliveries/1 |
| Child entities | Links | { "href": "/deliveries/1/confirmation" } |
| Update value objects | PUT or PATCH | PUT https://delivery-service/deliveries/1/dropoff |
| Repository | Collection | https://delivery-service/deliveries?status=pending |

##### API versioning

An API is a contract between a service and clients or consumers of that service. If an API changes, there is a risk of breaking clients that depend on the API, whether those are external clients or other microservices. Therefore, it's a good idea to minimize the number of API changes that you make. Often, changes in the underlying implementation don't require any changes to the API. Realistically, however, at some point you will want to add new features or new capabilities that require changing an existing API.

Whenever possible, make API changes backward compatible. For example, avoid removing a field from a model, because that can break clients that expect the field to be there. Adding a field does not break compatibility, because clients should ignore any fields they don't understand in a response. However, the service must handle the case where an older client omits the new field in a request.

Support versioning in your API contract. If you introduce a breaking API change, introduce a new API version. Continue to support the previous version, and let clients select which version to call. There are a couple of ways to do this. One is simply to expose both versions in the same service. Another option is to run two versions of the service side-by-side, and route requests to one or the other version, based on HTTP routing rules.



There's a cost to supporting multiple versions, in terms of developer time, testing, and operational overhead. Therefore, it's good to deprecate old versions as quickly as possible. For internal APIs, the team that owns the API can work with other teams to help them migrate to the new version. This is when having a cross-team governance process is useful. For external (public) APIs, it can be harder to deprecate an API version, especially if the API is consumed by third parties or by native client applications.

When a service implementation changes, it's useful to tag the change with a version. The version provides important information when troubleshooting errors. It can be very helpful for root cause analysis to know exactly which version of the service was called. Consider using semantic versioning for service versions. [Semantic versioning](https://semver.org/) uses a MAJOR.MINOR.PATCH format. However, clients should only select an API by the major version number, or possibly the minor version if there are significant (but non-breaking) changes between minor versions. In other words, it's reasonable for clients to select between version 1 and version 2 of an API, but not to select version 2.1.3. If you allow that level of granularity, you risk having to support a proliferation of versions.

For further discussion of API versioning, see [Versioning a RESTful web API](https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design#implement-versioning).

##### Idempotent operations

An operation is idempotent if it can be called multiple times without producing additional side-effects after the first call. Idempotency can be a useful resiliency strategy, because it allows an upstream service to safely invoke an operation multiple times. For a discussion of this point, see [Distributed transactions](https://learn.microsoft.com/en-us/azure/architecture/microservices/design/interservice-communication#distributed-transactions).

The HTTP specification states that GET, PUT, and DELETE methods must be idempotent. POST methods are not guaranteed to be idempotent. If a POST method creates a new resource, there is generally no guarantee that this operation is idempotent. The specification defines idempotent this way:

A request method is considered "idempotent" if the intended effect on the server of multiple identical requests with that method is the same as the effect for a single such request. ([RFC 7231](https://tools.ietf.org/html/rfc7231#section-4))

It's important to understand the difference between PUT and POST semantics when creating a new entity. In both cases, the client sends a representation of an entity in the request body. But the meaning of the URI is different.

* For a POST method, the URI represents a parent resource of the new entity, such as a collection. For example, to create a new delivery, the URI might be /api/deliveries. The server creates the entity and assigns it a new URI, such as /api/deliveries/39660. This URI is returned in the Location header of the response. Each time the client sends a request, the server will create a new entity with a new URI.
* For a PUT method, the URI identifies the entity. If there already exists an entity with that URI, the server replaces the existing entity with the version in the request. If no entity exists with that URI, the server creates one. For example, suppose the client sends a PUT request to api/deliveries/39660. Assuming there is no delivery with that URI, the server creates a new one. Now if the client sends the same request again, the server will replace the existing entity.

Here is the Delivery service's implementation of the PUT method.

[HttpPut("{id}")]

[ProducesResponseType(typeof(Delivery), 201)]

[ProducesResponseType(typeof(void), 204)]

public async Task<IActionResult> Put([FromBody]Delivery delivery, string id)

{

logger.LogInformation("In Put action with delivery {Id}: {@DeliveryInfo}", id, delivery.ToLogInfo());

try

{

var internalDelivery = delivery.ToInternal();

// Create the new delivery entity.

await deliveryRepository.CreateAsync(internalDelivery);

// Create a delivery status event.

var deliveryStatusEvent = new DeliveryStatusEvent { DeliveryId = delivery.Id, Stage = DeliveryEventType.Created };

await deliveryStatusEventRepository.AddAsync(deliveryStatusEvent);

// Return HTTP 201 (Created)

return CreatedAtRoute("GetDelivery", new { id= delivery.Id }, delivery);

}

catch (DuplicateResourceException)

{

// This method is mainly used to create deliveries. If the delivery already exists then update it.

logger.LogInformation("Updating resource with delivery id: {DeliveryId}", id);

var internalDelivery = delivery.ToInternal();

await deliveryRepository.UpdateAsync(id, internalDelivery);

// Return HTTP 204 (No Content)

return NoContent();

}

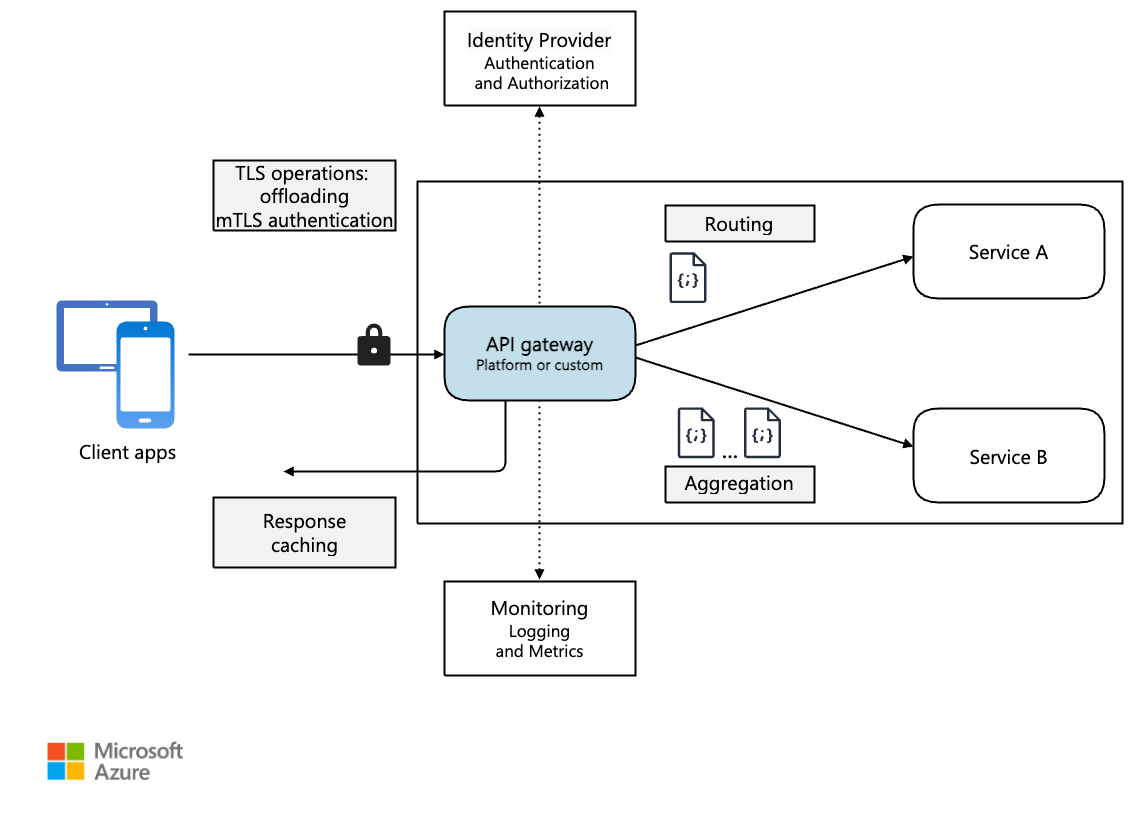
}

It's expected that most requests will create a new entity, so the method optimistically calls CreateAsync on the repository object, and then handles any duplicate-resource exceptions by updating the resource instead.

#### API Gateways

Use API gateways in microservices

In a microservices architecture, a client might interact with more than one front-end service. Given this fact, how does a client know what endpoints to call? What happens when new services are introduced, or existing services are refactored? How do services handle SSL termination, mutual TLS, authentication, and other concerns? An API gateway can help to address these challenges.



##### What is an API gateway?

An API gateway provides a centralized entry point for managing interactions between clients and application services. It acts as a reverse proxy and routes clients requests to the appropriate services. It can also perform various cross-cutting tasks such as authentication, SSL termination, mutual TLS, and rate limiting.

##### Why use an API gateway?

An API gateway simplifies communication, enhances client interactions, and centralizes the management of common service-level responsibilities. It acts as an intermediary, and it prevents direct exposure of application services to clients. Without an API gateway, clients must communicate directly with individual application services, which can introduce the following challenges:

* Complex client code: It can result in complex client code. Clients must track multiple endpoints and handle failures resiliently.
* Tight coupling: It creates coupling between the client and the backend. Clients need to understand decomposition of individual services, complicating service maintenance and refactoring.
* Increased latency: A single operation might require calls to multiple services. The result can be multiple network round trips between the client and the server, adding significant latency.
* Redundant handling of concerns: Each public-facing service must handle concerns such as authentication, SSL, and client rate limiting.
* Protocol limitations: Services must expose a client-friendly protocol such as HTTP or WebSocket. This exposure limits [communication protocols](https://learn.microsoft.com/en-us/azure/architecture/microservices/design/interservice-communication) options.
* Expanded attack surface: Public endpoints increase the potential attack surface and require hardening.

##### How to use an API gateway

An API gateway can be tailored to your application’s requirements by using specific design patterns. These design patterns address key functionality such as routing, request aggregation, and cross-cutting concerns:

* [Gateway routing](https://learn.microsoft.com/en-us/azure/architecture/patterns/gateway-routing). You can use an API gateway as a reverse proxy to route client requests to different application services. The API gateway uses layer-7 routing and provides a single endpoint for clients to use. Use API gateway routing when you want to decouple clients from application services.
* [Gateway aggregation](https://learn.microsoft.com/en-us/azure/architecture/patterns/gateway-aggregation). You can use the API gateway to aggregate multiple client requests into a single request. Use this pattern when a single operation requires calls to multiple application services. In API aggregation, the client sends one request to the API gateway. Then, the API gateway routes requests to the various services required for the operations. Finally, the API gateway aggregates the results and sends them back to the client. The aggregation helps reduce chattiness between the client and the application services.
* [Gateway offloading](https://learn.microsoft.com/en-us/azure/architecture/patterns/gateway-offloading). You can use an API gateway to provide cross-cutting functionality, so individual services don't have to provide it. It can be useful to consolidate cross-cutting functionality into one place, rather than making every service responsible. Here are examples of functionality that you could offload to an API gateway:
* SSL termination
* Mutual TLS
* Authentication
* IP allowlist or blocklist
* Client rate limiting (throttling)
* Logging and monitoring
* Response caching
* Web application firewall
* GZIP compression
* Servicing static content
* API gateway options

Here are some options for implementing an API gateway in your application.

* Reverse proxy server. Nginx and HAProxy are open-source reverse proxy offerings. They support features such as load balancing, SSL termination, and layer-7 routing. They have free versions and paid editions that provide extra features and support options. These products are mature with rich feature sets, high performance, and extensible.
* [Service mesh ingress controller](https://learn.microsoft.com/en-us/azure/aks/servicemesh-about/). If you use a service mesh, evaluate the ingress controller’s features specific to that service mesh. Check for AKS-supported add-ons like Istio and Open Service Mesh. Look for third-party open-source projects like Linkerd or Consul Connect. For example, the Istio ingress controller supports layer 7 routing, HTTP redirects, retries, and other features.
* [Azure Application Gateway](https://learn.microsoft.com/en-us/azure/application-gateway/). Application Gateway is a managed load balancing service. It provides perform layer-7 routing, SSL termination, and a web application firewall (WAF).
* [Azure Front Door](https://learn.microsoft.com/en-us/azure/frontdoor/front-door-overview). Azure Front Door is a content delivery network (CDN). It uses global and local points of presence (PoPs) to provide fast, reliable, and secure access to your applications' static and dynamic web content globally.
* [Azure API Management](https://learn.microsoft.com/en-us/azure/api-management/). API Management is a managed solution for publishing APIs to external and internal customers. It provides features to manage public-facing APIs, including rate limiting, IP restrictions, and authentication using Microsoft Entra ID or other identity providers. API Management doesn't perform any load balancing, so you should use it with a load balancer, such as Azure Application Gateway, or a reverse proxy. For information, see [API Management with Azure Application Gateway](https://learn.microsoft.com/en-us/azure/api-management/api-management-howto-integrate-internal-vnet-appgateway).

##### Choose an API gateway technology

When selecting an API gateway, consider the following factors:

* Support all requirements. Choose an API gateway that supports your required features. All the previous [API gateway options](https://learn.microsoft.com/en-us/azure/architecture/microservices/design/gateway#api-gateway-options) support layer-7 routing. But their support for other features, such as authentication, rate limiting, and SSL termination, can vary. Assess whether a single gateway meets your needs or if multiple gateways are necessary.
* Prefer built-in offerings. Use built-in API gateway and ingress solutions provided by your platform, such as Azure Container Apps and AKS, whenever they meet your security and control requirements. Only use a custom gateway if the built-in options lack necessary flexibility. Custom solutions require a governance model, such as GitOps, to manage its lifecycle effectively.
* Choose the right deployment model. Use managed services like Azure Application Gateway and Azure API Management for reduced operational overhead. If you use general-purpose reverse proxies or load balancers, deploy them in a way that aligns with your architecture. You can deploy general-purpose API gateways to dedicated virtual machines or inside an AKS cluster in their Ingress Controller offerings. To isolate the API gateway from the workload, you can deploy them outside the cluster, but this deployment increases the management complexity.
* Manage changes. When you update services or add new ones, you might need to update the gateway routing rules. Implement processes or workflows to manage routing rules when adding or modifying services, SSL certificates, IP allowlists, and security configurations. Use infrastructure-as-code and automation tools to streamline API gateway management.

Next steps

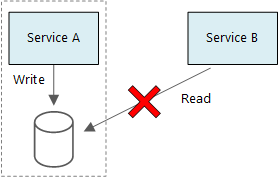
Previous articles explored the interfaces between microservices and between microservices and client applications. These interfaces treat each service as a self-contained, opaque unit. A critical principle of microservices architecture is that services should never expose internal details about how they manage data. This approach has significant implications for maintaining data integrity and consistency, which is the subject of the next article.

#### Data considerations for microservices

This article describes considerations for managing data in a microservices architecture. Because every microservice manages its own data, data integrity and data consistency are critical challenges.

A basic principle of microservices is that each service manages its own data. Two services should not share a data store. Instead, each service is responsible for its own private data store, which other services cannot access directly.

The reason for this rule is to avoid unintentional coupling between services, which can result if services share the same underlying data schemas. If there is a change to the data schema, the change must be coordinated across every service that relies on that database. By isolating each service's data store, we can limit the scope of change, and preserve the agility of truly independent deployments. Another reason is that each microservice may have its own data models, queries, or read/write patterns. Using a shared data store limits each team's ability to optimize data storage for their particular service.



This approach naturally leads to [polyglot persistence](https://martinfowler.com/bliki/PolyglotPersistence.html) — the use of multiple data storage technologies within a single application. One service might require the schema-on-read capabilities of a document database. Another might need the referential integrity provided by an RDBMS. Each team is free to make the best choice for their service.

It's fine for services to share the same physical database server. The problem occurs when services share the same schema, or read and write to the same set of database tables.

##### Challenges

Some challenges arise from this distributed approach to managing data. First, there may be redundancy across the data stores, with the same item of data appearing in multiple places. For example, data might be stored as part of a transaction, then stored elsewhere for analytics, reporting, or archiving. Duplicated or partitioned data can lead to issues of data integrity and consistency. When data relationships span multiple services, you can't use traditional data management techniques to enforce the relationships.

Traditional data modeling uses the rule of "one fact in one place." Every entity appears exactly once in the schema. Other entities may hold references to it but not duplicate it. The obvious advantage to the traditional approach is that updates are made in a single place, which avoids problems with data consistency. In a microservices architecture, you have to consider how updates are propagated across services, and how to manage eventual consistency when data appears in multiple places without strong consistency.

##### Approaches to managing data

There is no single approach that's correct in all cases, but here are some general guidelines for managing data in a microservices architecture.

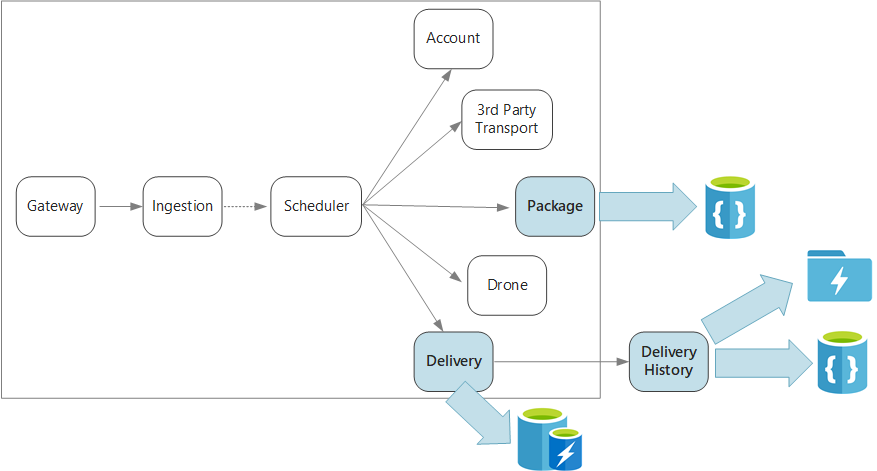
* Define the required consistency level per component, preferring eventual consistency where possible. Understand the places in the system where you need strong consistency or ACID transactions, and the places where eventual consistency is acceptable. Review Using tactical DDD to design microservices for further component guidance.
* When you need strong consistency guarantees, one service may represent the source of truth for a given entity, which is exposed through an API. Other services might hold their own copy of the data, or a subset of the data, that is eventually consistent with the master data but not considered the source of truth. For example, imagine an e-commerce system with a customer order service and a recommendation service. The recommendation service might listen to events from the order service, but if a customer requests a refund, it is the order service, not the recommendation service, that has the complete transaction history.
* For transactions, use patterns such as [Scheduler Agent Supervisor](https://learn.microsoft.com/en-us/azure/architecture/patterns/scheduler-agent-supervisor) and [Compensating Transaction](https://learn.microsoft.com/en-us/azure/architecture/patterns/compensating-transaction) to keep data consistent across several services. You may need to store an additional piece of data that captures the state of a unit of work that spans multiple services, to avoid partial failure among multiple services. For example, keep a work item on a durable queue while a multi-step transaction is in progress.
* Store only the data that a service needs. A service might only need a subset of information about a domain entity. For example, in the Shipping bounded context, we need to know which customer is associated to a particular delivery. But we don't need the customer's billing address — that's managed by the Accounts bounded context. Thinking carefully about the domain, and using a DDD approach, can help here.
* Consider whether your services are coherent and loosely coupled. If two services are continually exchanging information with each other, resulting in chatty APIs, you may need to redraw your service boundaries, by merging two services or refactoring their functionality.
* Use an [event driven architecture style](https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/event-driven). In this architecture style, a service publishes an event when there are changes to its public models or entities. Interested services can subscribe to these events. For example, another service could use the events to construct a materialized view of the data that is more suitable for querying.
* A service that owns events should publish a schema that can be used to automate serializing and deserializing the events, to avoid tight coupling between publishers and subscribers. Consider JSON schema or a framework like [Microsoft Bond](file:///C:\Users\Lalit\AppData\Roaming\Microsoft\Word\v), Protobuf, or Avro.
* At high scale, events can become a bottleneck on the system, so consider using aggregation or batching to reduce the total load.

##### Example: Choosing data stores for the Drone Delivery application

The previous articles in this series discuss a drone delivery service as a running example. You can read more about the scenario and the corresponding architecture in [Design a microservices architecture](https://learn.microsoft.com/en-us/azure/architecture/microservices/design/interservice-communication).

To recap, this application defines several microservices for scheduling deliveries by drone. When a user schedules a new delivery, the client request includes information about the delivery, such as pickup and dropoff locations, and about the package, such as size and weight. This information defines a unit of work.

The various backend services care about different portions of the information in the request, and also have different read and write profiles.



Delivery service

The Delivery service stores information about every delivery that is currently scheduled or in progress. It listens for events from the drones, and tracks the status of deliveries that are in progress. It also sends domain events with delivery status updates.

It's expected that users will frequently check the status of a delivery while they are waiting for their package. Therefore, the Delivery service requires a data store that emphasizes throughput (read and write) over long-term storage. Also, the Delivery service does not perform any complex queries or analysis, it simply fetches the latest status for a given delivery. The Delivery service team chose Azure Cache for Redis for its high read-write performance. The information stored in Redis is relatively short-lived. Once a delivery is complete, the Delivery History service is the system of record.

Delivery History service

The Delivery History service listens for delivery status events from the Delivery service. It stores this data in long-term storage. There are two different use-cases for this historical data, which have different data storage requirements.

The first scenario is aggregating the data for the purpose of data analytics, in order to optimize the business or improve the quality of the service. Note that the Delivery History service doesn't perform the actual analysis of the data. It's only responsible for the ingestion and storage. For this scenario, the storage must be optimized for data analysis over a large set of data, using a schema-on-read approach to accommodate a variety of data sources. [Azure Data Lake Store](https://learn.microsoft.com/en-us/azure/data-lake-store/data-lake-store-performance-tuning-guidance) is a good fit for this scenario. Data Lake Store is an Apache Hadoop file system compatible with Hadoop Distributed File System (HDFS), and is tuned for performance for data analytics scenarios.

The other scenario is enabling users to look up the history of a delivery after the delivery is completed. Azure Data Lake is not optimized for this scenario. For optimal performance, Microsoft recommends storing time-series data in Data Lake in folders partitioned by date. (See [Tuning Azure Data Lake Store for performance](https://learn.microsoft.com/en-us/azure/data-lake-store/data-lake-store-performance-tuning-guidance)). However, that structure is not optimal for looking up individual records by ID. Unless you also know the timestamp, a lookup by ID requires scanning the entire collection. Therefore, the Delivery History service also stores a subset of the historical data in Azure Cosmos DB for quicker lookup. The records don't need to stay in Azure Cosmos DB indefinitely. Older deliveries can be archived — say, after a month. This could be done by running an occasional batch process. Archiving older data can reduce costs for Cosmos DB while still keeping the data available for historical reporting from the Data Lake.

Package service

The Package service stores information about all of the packages. The storage requirements for the Package are:

* Long-term storage.
* Able to handle a high volume of packages, requiring high write throughput.
* Support simple queries by package ID. No complex joins or requirements for referential integrity.

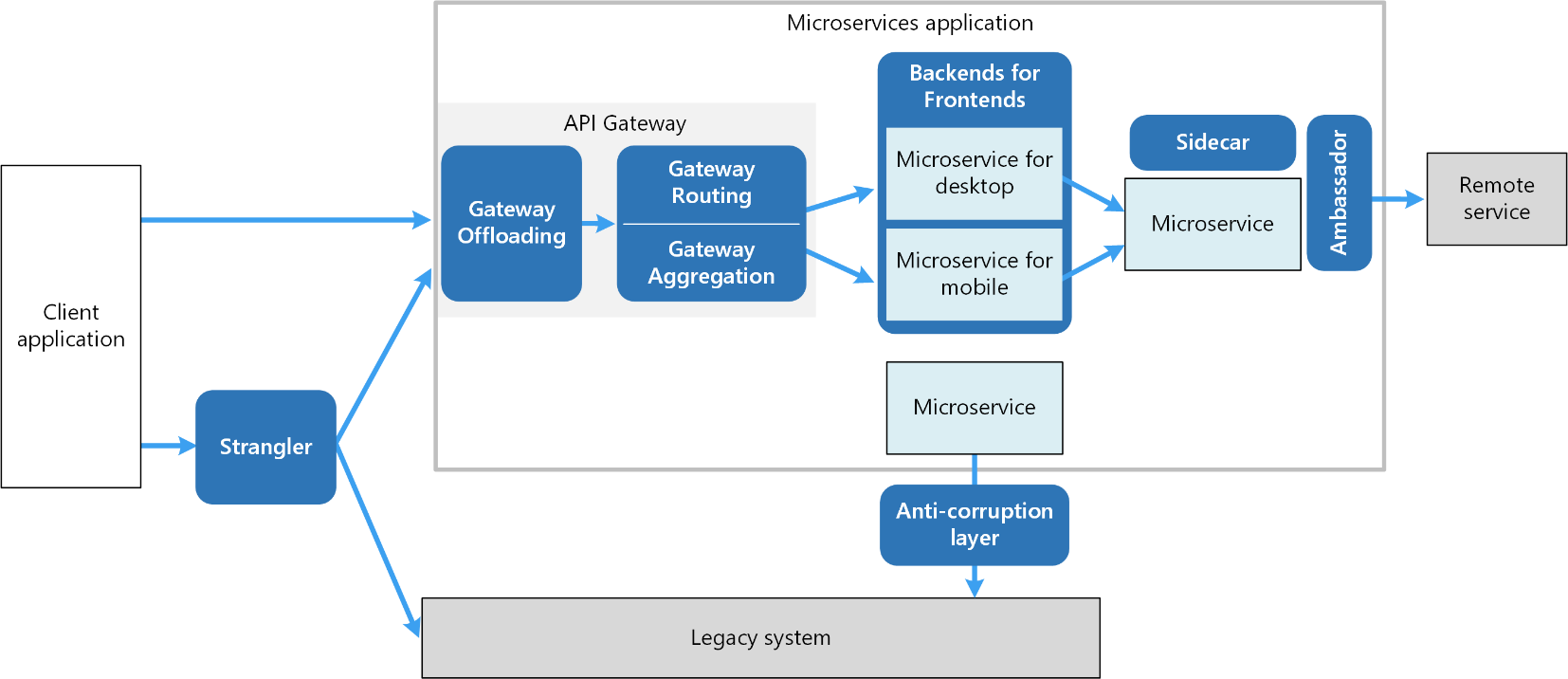
Because the package data is not relational, a document-oriented database is appropriate, and Azure Cosmos DB can achieve high throughput by using sharded collections. The team that works on the Package service is familiar with the MEAN stack (MongoDB, Express.js, AngularJS, and Node.js), so they select the MongoDB API for Azure Cosmos DB. That lets them leverage their existing experience with MongoDB, while getting the benefits of Azure Cosmos DB, which is a managed Azure service.

Next steps

Learn about design patterns that can help mitigate some common challenges in a microservices architecture.

# Design Patterns for Microservices

The goal of microservices is to increase the velocity of application releases, by decomposing the application into small autonomous services that can be deployed independently. A microservices architecture also brings some challenges. The design patterns shown here can help mitigate these challenges.



## Microservices

### Ambassador

[Ambassador](https://learn.microsoft.com/en-us/azure/architecture/patterns/ambassador) can be used to offload common client connectivity tasks such as monitoring, logging, routing, and security (such as TLS) in a language agnostic way. Ambassador services are often deployed as a sidecar (see below).

Create helper services that send network requests on behalf of a consumer service or application. An ambassador service can be thought of as an out-of-process proxy that is co-located with the client.

This pattern can be useful for offloading common client connectivity tasks such as monitoring, logging, routing, security (such as TLS), and resiliency patterns in a language agnostic way. It's often used with legacy applications, or other applications that are difficult to modify, in order to extend their networking capabilities. It can also enable a specialized team to implement those features.

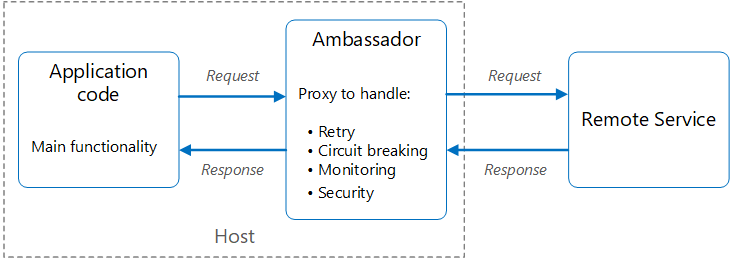
Context and problem

Resilient cloud-based applications require features such as circuit breaking, routing, metering and monitoring, and the ability to make network-related configuration updates. It may be difficult or impossible to update legacy applications or existing code libraries to add these features, because the code is no longer maintained or can't be easily modified by the development team.

Network calls may also require substantial configuration for connection, authentication, and authorization. If these calls are used across multiple applications, built using multiple languages and frameworks, the calls must be configured for each of these instances. In addition, network and security functionality may need to be managed by a central team within your organization. With a large code base, it can be risky for that team to update application code they aren't familiar with.

Solution

Put client frameworks and libraries into an external process that acts as a proxy between your application and external services. Deploy the proxy on the same host environment as your application to allow control over routing, resiliency, security features, and to avoid any host-related access restrictions. You can also use the ambassador pattern to standardize and extend instrumentation. The proxy can monitor performance metrics such as latency or resource usage, and this monitoring happens in the same host environment as the application.



Features that are offloaded to the ambassador can be managed independently of the application. You can update and modify the ambassador without disturbing the application's legacy functionality. It also allows for separate, specialized teams to implement and maintain security, networking, or authentication features that have been moved to the ambassador.

Ambassador services can be deployed as a sidecar to accompany the lifecycle of a consuming application or service. Alternatively, if an ambassador is shared by multiple separate processes on a common host, it can be deployed as a daemon or Windows service. If the consuming service is containerized, the ambassador should be created as a separate container on the same host, with the appropriate links configured for communication.

Issues and considerations

* The proxy adds some latency overhead. Consider whether a client library, invoked directly by the application, is a better approach.
* Consider the possible impact of including generalized features in the proxy. For example, the ambassador could handle retries, but that might not be safe unless all operations are idempotent.
* Consider a mechanism to allow the client to pass some context to the proxy, and back to the client. For example, include HTTP request headers to opt out of retry or specify the maximum number of times to retry.
* Consider how you'll package and deploy the proxy.
* Consider whether to use a single shared instance for all clients or an instance for each client.

When to use this pattern

Use this pattern when you:

* Need to build a common set of client connectivity features for multiple languages or frameworks.
* Need to offload cross-cutting client connectivity concerns to infrastructure developers or other more specialized teams.
* Need to support cloud or cluster connectivity requirements in a legacy application or an application that is difficult to modify.

This pattern may not be suitable:

* When network request latency is critical. A proxy introduces some overhead, although minimal, and in some cases this may affect the application.
* When client connectivity features are consumed by a single language. In that case, a better option might be a client library that is distributed to the development teams as a package.
* When connectivity features can't be generalized and require deeper integration with the client application.

Pillar - How this pattern supports pillar goals

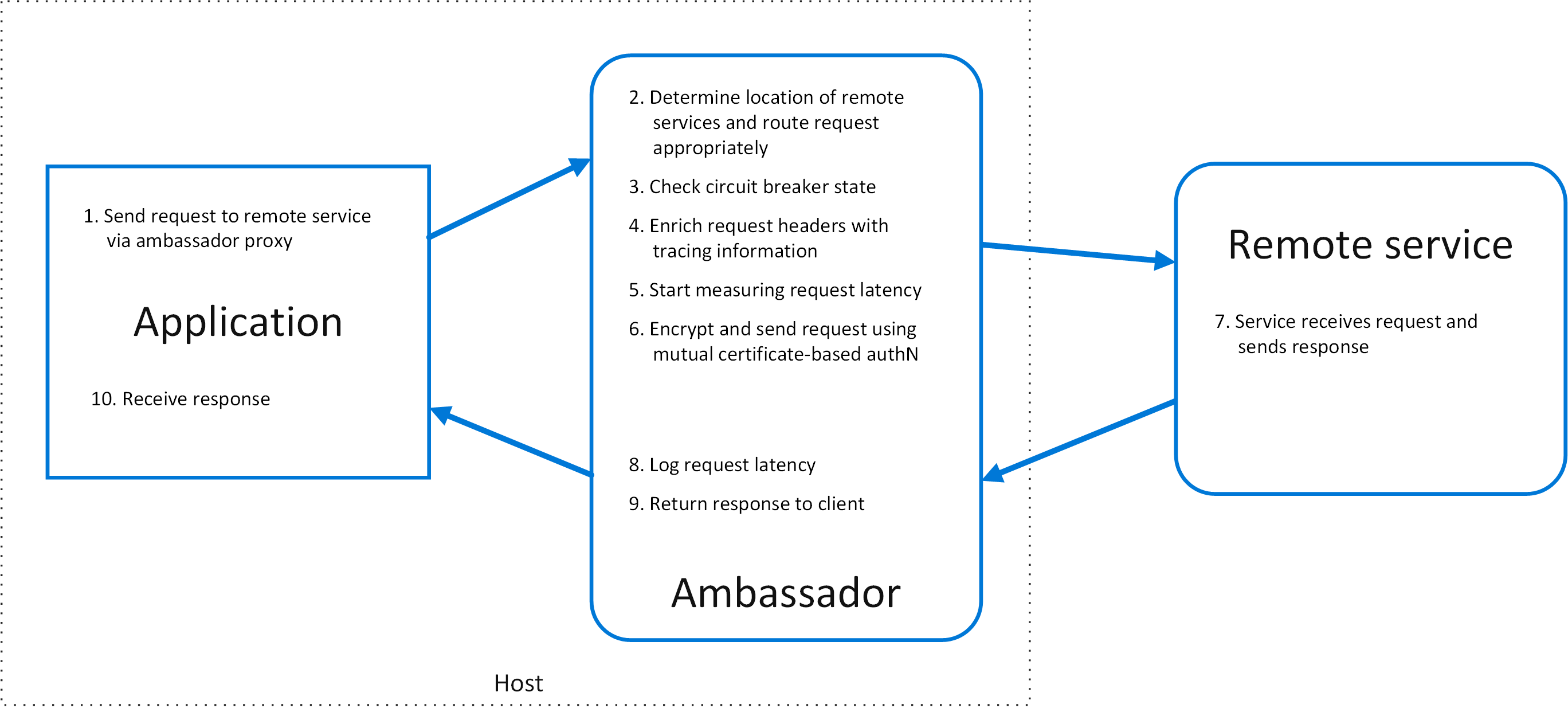
|  |  |
| --- | --- |
| [Reliability](https://learn.microsoft.com/en-us/azure/well-architected/reliability/checklist) design decisions help your workload become resilient to malfunction and to ensure that it recovers to a fully functioning state after a failure occurs. | The network communications mediation point facilitated by this pattern provides an opportunity to add reliability patterns to network communication, such as retry or buffering.  [RE:07 Self-preservation](https://learn.microsoft.com/en-us/azure/well-architected/reliability/self-preservation) |
| [Security](https://learn.microsoft.com/en-us/azure/well-architected/security/checklist) design decisions help ensure the confidentiality, integrity, and availability of your workload's data and systems. | This pattern provides an opportunity to augment security on network communications that couldn't have been handled by the client directly.  [SE:06 Network controls](file:///C:\Users\Lalit\AppData\Roaming\Microsoft\Word\v)  [SE:07 Encryption](https://learn.microsoft.com/en-us/azure/well-architected/security/encryption) |

As with any design decision, consider any tradeoffs against the goals of the other pillars that might be introduced with this pattern.

Workload design

An architect should evaluate how the Ambassador pattern can be used in their workload's design to address the goals and principles covered in the Azure Well-Architected Framework pillars. For example:

The following diagram shows an application making a request to a remote service via an ambassador proxy. The ambassador provides routing, circuit breaking, and logging. It calls the remote service and then returns the response to the client application:



Related Pattern – Sidecar

### Sidecar

[Sidecar](https://learn.microsoft.com/en-us/azure/architecture/patterns/sidecar) deploys helper components of an application as a separate container or process to provide isolation and encapsulation.

Deploy components of an application into a separate process or container to provide isolation and encapsulation. This pattern can also enable applications to be composed of heterogeneous components and technologies.

This pattern is named Sidecar because it resembles a sidecar attached to a motorcycle. In the pattern, the sidecar is attached to a parent application and provides supporting features for the application. The sidecar also shares the same lifecycle as the parent application, being created and retired alongside the parent. The sidecar pattern is sometimes referred to as the sidekick pattern and is a decomposition pattern.

Context and problem

Applications and services often require related functionality, such as monitoring, logging, configuration, and networking services. These peripheral tasks can be implemented as separate components or services.

If they're tightly integrated into the application, they can run in the same process as the application, making efficient use of shared resources. However, this also means they're not well isolated, and an outage in one of these components can affect other components or the entire application. Also, they usually need to be implemented using the same language as the parent application. As a result, the component and the application have close interdependence on each other.

If the application is decomposed into services, then each service can be built using different languages and technologies. While this gives more flexibility, it means that each component has its own dependencies and requires language-specific libraries to access the underlying platform and any resources shared with the parent application. In addition, deploying these features as separate services can add latency to the application. Managing the code and dependencies for these language-specific interfaces can also add considerable complexity, especially for hosting, deployment, and management.

Solution

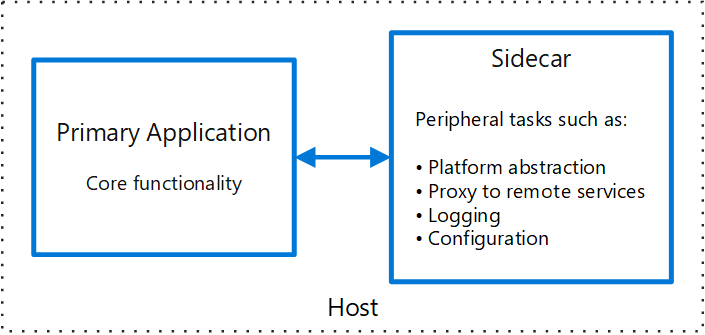
Co-locate a cohesive set of tasks with the primary application, but place them inside their own process or container, providing a homogeneous interface for platform services across languages.

A sidecar service isn't necessarily part of the application, but is connected to it. It goes wherever the parent application goes. Sidecars are supporting processes or services that are deployed with the primary application. On a motorcycle, the sidecar is attached to one motorcycle, and each motorcycle can have its own sidecar. In the same way, a sidecar service shares the fate of its parent application. For each instance of the application, an instance of the sidecar is deployed and hosted alongside it.

**Advantages of using a sidecar pattern include:**

* A sidecar is independent from its primary application in terms of runtime environment and programming language, so you don't need to develop one sidecar per language.
* The sidecar can access the same resources as the primary application. For example, a sidecar can monitor system resources used by both the sidecar and the primary application.
* Because of its proximity to the primary application, there's no significant latency when communicating between them.
* Even for applications that don't provide an extensibility mechanism, you can use a sidecar to extend functionality by attaching it as its own process in the same host or sub-container as the primary application.

The sidecar pattern is often used with containers and referred to as a sidecar container or sidekick container.



Issues and considerations

* Consider the deployment and packaging format you will use to deploy services, processes, or containers. Containers are particularly well suited to the sidecar pattern.
* When designing a sidecar service, carefully decide on the interprocess communication mechanism. Try to use language- or framework-agnostic technologies unless performance requirements make that impractical.
* Before putting functionality into a sidecar, consider whether it would work better as a separate service or a more traditional daemon.
* Also consider whether the functionality could be implemented as a library or using a traditional extension mechanism. Language-specific libraries may have a deeper level of integration and less network overhead.

When to use this pattern

Use this pattern when:

* Your primary application uses a heterogeneous set of languages and frameworks. A component located in a sidecar service can be consumed by applications written in different languages using different frameworks.
* A component is owned by a remote team or a different organization.
* A component or feature must be co-located on the same host as the application.
* You need a service that shares the overall lifecycle of your main application, but can be independently updated.
* You need fine-grained control over resource limits for a particular resource or component. For example, you may want to restrict the amount of memory a specific component uses. You can deploy the component as a sidecar and manage memory usage independently of the main application.

This pattern may not be suitable:

* When interprocess communication needs to be optimized. Communication between a parent application and sidecar services includes some overhead, notably latency in the calls. This may not be an acceptable trade-off for chatty interfaces.
* For small applications where the resource cost of deploying a sidecar service for each instance is not worth the advantage of isolation.
* When the service needs to scale differently than or independently from the main applications. If so, it may be better to deploy the feature as a separate service.

Workload design

An architect should evaluate how the Sidecar pattern can be used in their workload's design to address the goals and principles covered in the [Azure Well-Architected Framework pillars](https://learn.microsoft.com/en-us/azure/well-architected/pillars). For example:

|  |  |
| --- | --- |
| Pillar | How this pattern supports pillar goals |
| Security design decisions help ensure the confidentiality, integrity, and availability of your workload's data and systems. | By encapsulating these task and deploying them out-of-process, you can reduce the surface area of sensitive processes to only the code that's needed to accomplish the task. You can also use sidecars to add cross-cutting security controls to an application component that's not natively designed with that functionality.  [SE:04 Segmentation](https://learn.microsoft.com/en-us/azure/well-architected/security/segmentation)  [SE:07 Encryption](https://learn.microsoft.com/en-us/azure/well-architected/security/encryption) |
| Operational Excellence helps deliver workload quality through standardized processes and team cohesion. | This pattern provides an approach to implementing flexibility in tool integration that might enhance the application's observability without requiring the application to take direct implementation dependencies. It enables the sidecar functionality to evolve independently and be maintained independently of the application's lifecycle.  [OE:04 Tools and processes](https://learn.microsoft.com/en-us/azure/well-architected/operational-excellence/tools-processes)  [OE:07 Monitoring system](https://learn.microsoft.com/en-us/azure/well-architected/operational-excellence/observability) |
| Performance Efficiency helps your workload efficiently meet demands through optimizations in scaling, data, code. | You can move cross-cutting tasks to a single process that can scale across multiple instances of the main process, which reduces the need to deploy duplicate functionality for each instance of the application.  [PE:07 Code and infrastructure](https://learn.microsoft.com/en-us/azure/well-architected/performance-efficiency/optimize-code-infrastructure) |

Example

The sidecar pattern is applicable to many scenarios. Some common examples:

* Infrastructure API. The infrastructure development team creates a service that's deployed alongside each application, instead of a language-specific client library to access the infrastructure. The service is loaded as a sidecar and provides a common layer for infrastructure services, including logging, environment data, configuration store, discovery, health checks, and watchdog services. The sidecar also monitors the parent application's host environment and process (or container) and logs the information to a centralized service.
* Manage NGINX/HAProxy. Deploy NGINX with a sidecar service that monitors environment state, then updates the NGINX configuration file and recycles the process when a change in state is needed.
* Ambassador sidecar. Deploy an ambassador service as a sidecar. The application calls through the ambassador, which handles request logging, routing, circuit breaking, and other connectivity related features.
* Offload proxy. Place an NGINX proxy in front of a node.js service instance, to handle serving static file content for the service.

### Anti Corruption Layer

[Anti-corruption layer](https://learn.microsoft.com/en-us/azure/architecture/patterns/anti-corruption-layer) implements a façade between new and legacy applications, to ensure that the design of a new application is not limited by dependencies on legacy systems.

Implement a façade or adapter layer between different subsystems that don't share the same semantics. This layer translates requests that one subsystem makes to the other subsystem. Use this pattern to ensure that an application's design is not limited by dependencies on outside subsystems. This pattern was first described by Eric Evans in Domain-Driven Design.

Context and problem

Most applications rely on other systems for some data or functionality. For example, when a legacy application is migrated to a modern system, it may still need existing legacy resources. New features must be able to call the legacy system. This is especially true of gradual migrations, where different features of a larger application are moved to a modern system over time.

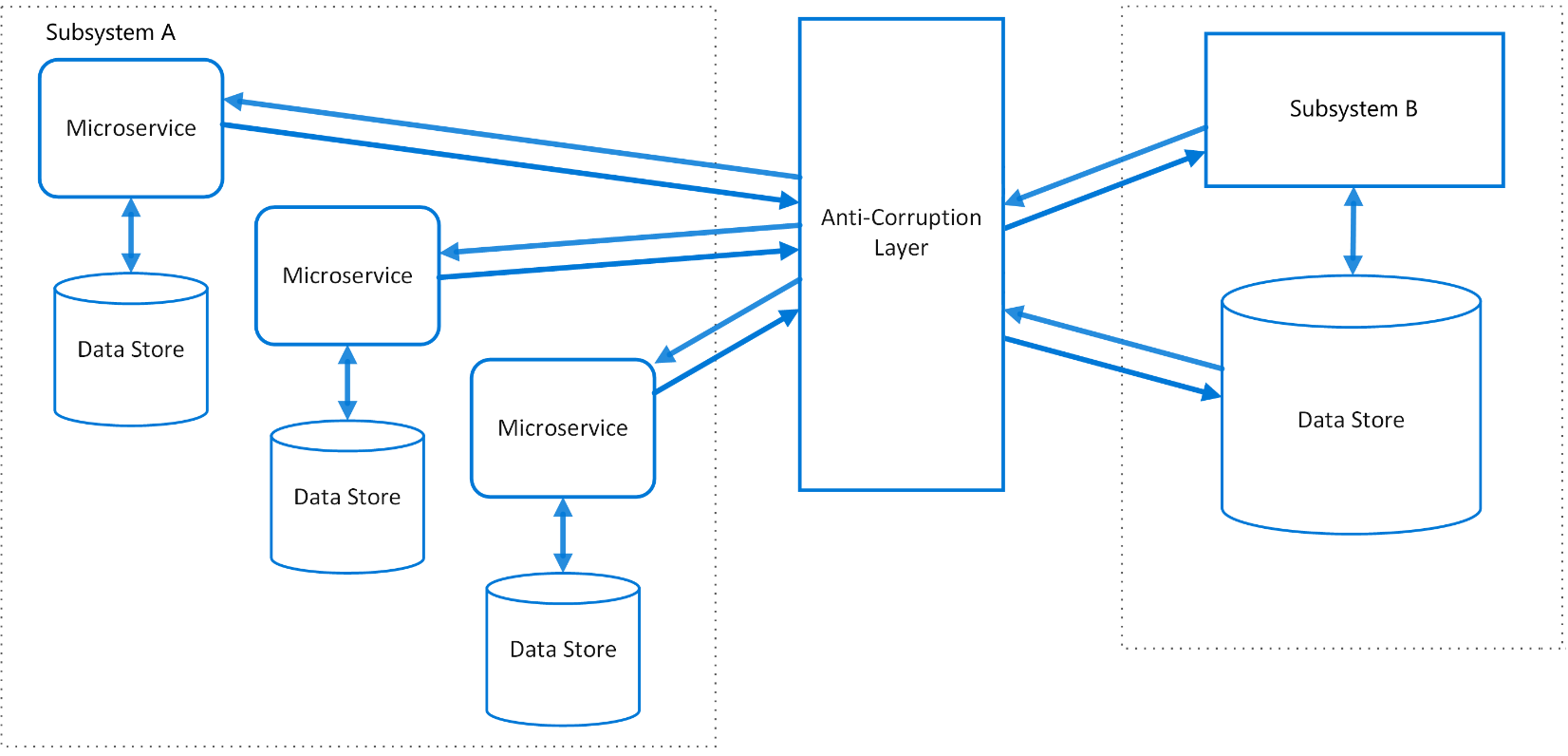
Often these legacy systems suffer from quality issues such as convoluted data schemas or obsolete APIs. The features and technologies used in legacy systems can vary widely from more modern systems. To interoperate with the legacy system, the new application may need to support outdated infrastructure, protocols, data models, APIs, or other features that you wouldn't otherwise put into a modern application.

Maintaining access between new and legacy systems can force the new system to adhere to at least some of the legacy system's APIs or other semantics. When these legacy features have quality issues, supporting them "corrupts" what might otherwise be a cleanly designed modern application.

Similar issues can arise with any external system that your development team doesn't control, not just legacy systems.

Solution

Isolate the different subsystems by placing an anti-corruption layer between them. This layer translates communications between the two systems, allowing one system to remain unchanged while the other can avoid compromising its design and technological approach.



The diagram above shows an application with two subsystems. Subsystem A calls to subsystem B through an anti-corruption layer. Communication between subsystem A and the anti-corruption layer always uses the data model and architecture of subsystem A. Calls from the anti-corruption layer to subsystem B conform to that subsystem's data model or methods. The anti-corruption layer contains all of the logic necessary to translate between the two systems. The layer can be implemented as a component within the application or as an independent service.

Issues and considerations

* The anti-corruption layer may add latency to calls made between the two systems.
* The anti-corruption layer adds an additional service that must be managed and maintained.
* Consider how your anti-corruption layer will scale.
* Consider whether you need more than one anti-corruption layer. You may want to decompose functionality into multiple services using different technologies or languages, or there may be other reasons to partition the anti-corruption layer.
* Consider how the anti-corruption layer will be managed in relation with your other applications or services. How will it be integrated into your monitoring, release, and configuration processes?
* Make sure transaction and data consistency are maintained and can be monitored.
* Consider whether the anti-corruption layer needs to handle all communication between different subsystems, or just a subset of features.
* If the anti-corruption layer is part of an application migration strategy, consider whether it will be permanent, or will be retired after all legacy functionality has been migrated.
* This pattern is illustrated with distinct subsystems above, but can apply to other service architectures as well, such as when integrating legacy code together in a monolithic architecture.

When to use this pattern

Use this pattern when:

A migration is planned to happen over multiple stages, but integration between new and legacy systems needs to be maintained.

Two or more subsystems have different semantics, but still need to communicate.

This pattern may not be suitable if there are no significant semantic differences between new and legacy systems.

Workload design

An architect should evaluate how the Anti-corruption Layer pattern can be used in their workload's design to address the goals and principles covered in the [Azure Well-Architected Framework pillars](https://learn.microsoft.com/en-us/azure/well-architected/pillars). For example:

|  |  |
| --- | --- |
| Pillar | How this pattern supports pillar goals |
| Operational Excellence helps deliver workload quality through standardized processes and team cohesion. | This pattern helps ensure that new component design remains uninfluenced by legacy implementations that might have different data models or business rules when you integrate with these legacy systems and it can reduce technical debt in new components while still supporting existing components.  - OE:04 Tools and processes |

As with any design decision, consider any tradeoffs against the goals of the other pillars that might be introduced with this pattern.

**Related Patterns:** Strangler Fig Pattern, Message Bridge Pattern

### Strangler Fig

[Strangler Fig](https://learn.microsoft.com/en-us/azure/architecture/patterns/strangler-fig) supports incremental refactoring of an application, by gradually replacing specific pieces of functionality with new services.

This pattern incrementally migrates a legacy system by gradually replacing specific pieces of functionality with new applications and services. As you replace features from the legacy system, the new system eventually comprises all of the old system's features. This approach suppresses the old system so that you can decommission it.

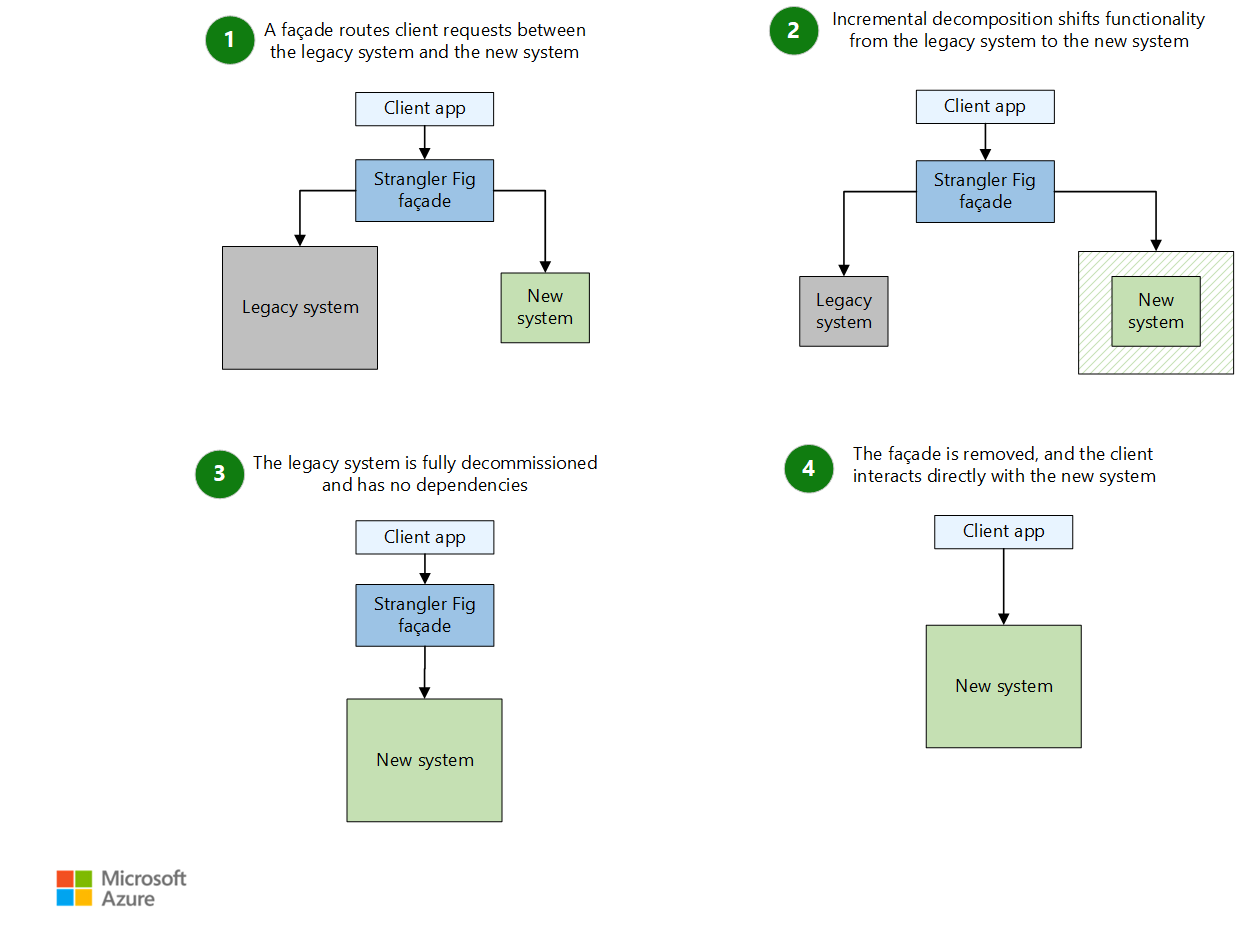
Context and problem

As systems age, the development tools, hosting technology, and system architectures that they're built on can become obsolete. As new features and functionality are added, these applications become more complex, which can make them harder to maintain or extend.

Replacing an entire complex system is a huge undertaking. Instead, many teams prefer to migrate to a new system gradually and keep the old system to handle unmigrated features. However, running two separate versions of an application forces clients to track which version has individual features. Every time teams migrate a feature or service, they must direct clients to the new location. To overcome these challenges, you can adopt an approach that supports incremental migration and minimizes disruptions to clients.

Solution

Use an incremental process to replace specific pieces of functionality with new applications and services. Customers can continue using the same interface, unaware that this migration is taking place.



The Strangler Fig pattern provides a controlled and phased approach to modernization. It allows the existing application to continue functioning during the modernization effort. A façade (proxy) intercepts requests that go to the back-end legacy system. The façade routes these requests either to the legacy application or to the new services.

This pattern reduces risks in migration by enabling your teams to move forward at a pace that suits the complexity of the project. As you migrate functionality to the new system, the legacy system becomes obsolete, and you decommission the legacy system.

1. The Strangler Fig pattern begins by introducing a façade (proxy) between the client app, the legacy system, and the new system. The façade acts as an intermediary. It allows the client app to interact with the legacy system and the new system. Initially, the façade routes most requests to the legacy system.
2. As the migration progresses, the façade incrementally shifts requests from the legacy system to the new system. With each iteration, you implement more pieces of functionality in the new system.

This incremental approach gradually reduces the legacy system's responsibilities and expands the scope of the new system. The process is iterative. It allows the team to address complexities and dependencies in manageable stages. These stages help the system remain stable and functional.

1. After you migrate all of the functionality and there are no dependencies on the legacy system, you can decommission the legacy system. The façade routes all requests exclusively to the new system.
2. You remove the façade and reconfigure the client app to communicate directly with the new system. This step marks the completion of the migration.

Problems and considerations

Consider the following points as you decide how to implement this pattern:

Consider how to handle services and data stores that both the new system and the legacy system might use. Make sure that both systems can access these resources at the same time.

Structure new applications and services so that you can easily intercept and replace them in future strangler fig migrations. For example, strive to have clear demarcations between parts of your solution so that you can migrate each part individually.

After the migration is complete, you typically remove the strangler fig façade. Alternatively, you can maintain the façade as an adaptor for legacy clients to use while you update the core system for newer clients.

Make sure that the façade keeps up with the migration.

Make sure that the façade doesn't become a single point of failure or a performance bottleneck.

When to use this pattern

Use this pattern when:

You gradually migrate a back-end application to a new architecture, especially when replacing large systems, key components, or complex features introduces risk.

The original system can continue to exist for an extended period of time during the migration effort.

This pattern might not be suitable when:

Requests to the back-end system can't be intercepted.

You migrate a small system and replacing the whole system is simple.

You need to fully decommission the original solution quickly.

Workload design

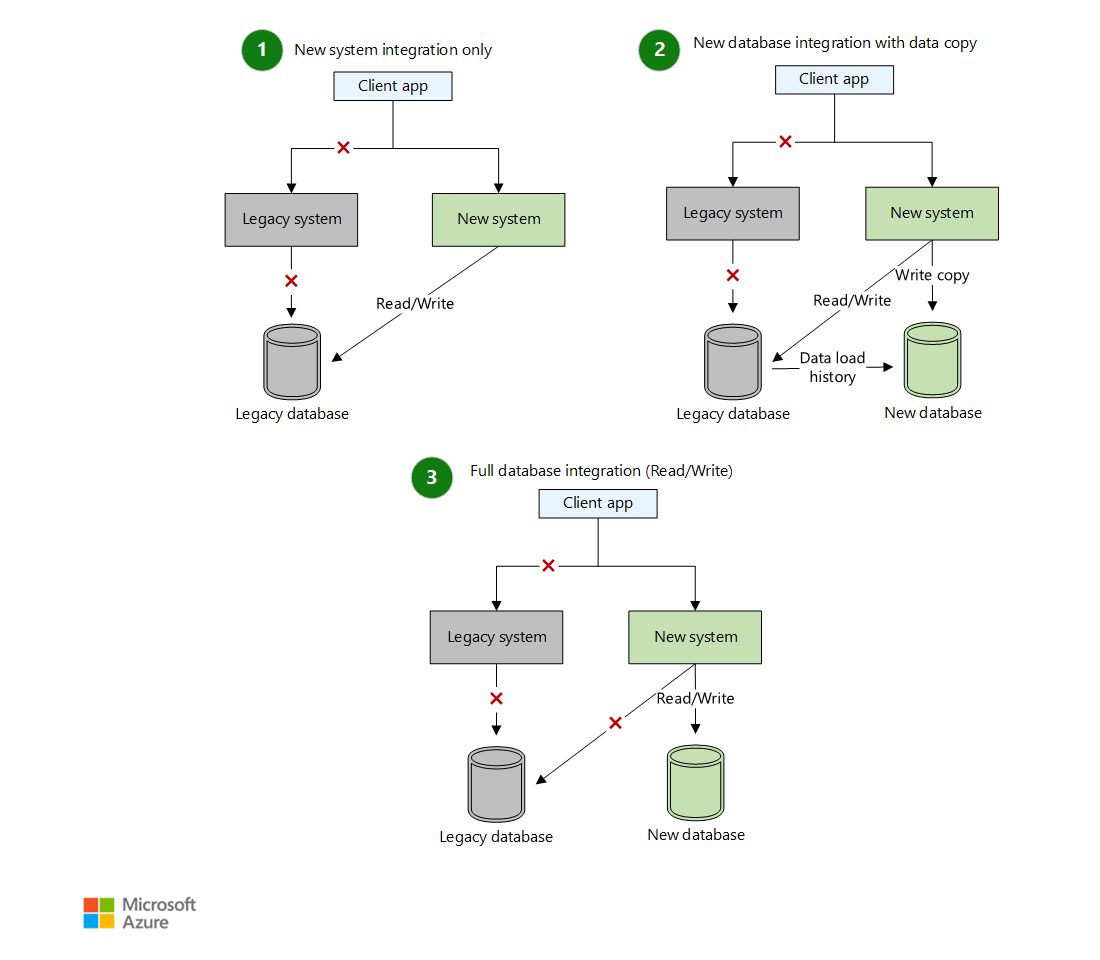
Evaluate how to use the Strangler Fig pattern in a workload's design to address the goals and principles of the Azure Well-Architected Framework pillars. The following table provides guidance about how this pattern supports the goals of each pillar.

|  |  |
| --- | --- |
| Pillar | How this pattern supports pillar goals |
| Reliability design decisions help your workload become resilient to malfunction and to ensure that it recovers to a fully functioning state after a failure occurs. | This pattern's incremental approach can help mitigate risks during a component transition compared to making large systemic changes all at once.  - RE:08 Testing |
| Cost Optimization focuses on sustaining and improving your workload's return on investment (ROI). | The goal of this approach is to maximize the use of existing investments in the currently running system while modernizing incrementally. It enables you to perform high-ROI replacements before low-ROI replacements.  - CO:07 Component costs  - CO:08 Environment costs |
| Operational Excellence helps deliver workload quality through standardized processes and team cohesion. | This pattern provides a continuous improvement approach. Incremental replacements that make small changes over time are preferable to large systemic changes that are riskier to implement.  - OE:06 Supply chain for workload development  - OE:11 Safe deployment practices |

Consider any trade-offs against the goals of the other pillars that this pattern might introduce.

Example

Legacy systems typically depend on a centralized database. Over time, a centralized database can become difficult to manage and evolve because of its many dependencies. To address these challenges, various database patterns can facilitate the transition away from such legacy systems. The Strangler Fig pattern is one of these patterns. Apply the Strangler Fig pattern as a phased approach to gradually transition from a legacy system to a new system and minimize disruption.



1. You introduce a new system, and the new system starts handling some requests from the client app. However, the new system still depends on the legacy database for all read and write operations. The legacy system remains operational, which facilitates a smooth transition without immediate structural changes.
2. In the next phase, you introduce a new database. You migrate data load history to the new database by using an extract, transform, and load (ETL) process. The ETL process synchronizes the new database with the legacy database. During this phase, the new system performs shadow writes. The new system updates both databases in parallel. The new system continues to read from the legacy database to validate consistency.
3. Finally, the new database becomes the system of record. The new database takes over all read and write operations. You can start deprecating the legacy database and legacy system. After you validate the new database, you can retire the legacy database. This retirement completes the migration process with minimal disruption.

Related Pattern is Message Bridge

[Martin Fowler – Strangler Fig Application](https://martinfowler.com/bliki/StranglerFigApplication.html)

For the complete catalog of cloud design patterns on the Azure Architecture Center, see Cloud Design Patterns.

### Message Bridge

[Messaging Bridge](https://learn.microsoft.com/en-us/azure/architecture/patterns/messaging-bridge) integrates disparate systems built with different messaging infrastructures.

This article describes the Messaging Bridge pattern, which is a technique that you can use to integrate disparate systems that are built on top of different messaging infrastructures.

Context and problem

Many organizations and workloads can inadvertently have IT systems that use multiple messaging infrastructures like Microsoft Message Queueing (MSMQ), RabbitMQ, Azure Service Bus, and Amazon SQS. This problem can occur due to mergers, acquisitions, or due to extending current on-premises systems to cloud-hosted components for cost-effectiveness and the ease of maintenance.

Developers might address this challenge by modifying the systems being integrated to communicate by using HTTP-based web services. However, this approach has drawbacks, including:

The systems must be modified by adding an HTTP client on one side and an HTTP request handler on the other. The systems must then be retested and redeployed.

HTTP endpoints must be hosted, which adds complexity when you make web services secure and highly available.

Frequent network connectivity problems that require custom-built retry mechanisms.

Solution

If the systems being integrated consist of components that communicate by exchanging messages, the Messaging Bridge pattern improves integration and mitigates drawbacks.

In this scenario, each system connects to one messaging infrastructure. To integrate across different messaging infrastructures, introduce a bridge component that connects to two or more messaging infrastructures at the same time. The bridge pulls messages from one and pushes them to the other without changing the payload.

The systems being integrated don't need to recognize the others or the bridge. The sender system is configured to send specific messages to a designated queue on its native messaging infrastructure. The bridge picks up those messages and forwards them to another queue in a different messaging infrastructure where the receiver system picks them up.

Benefits

The systems being integrated via the Messaging Bridge don't have to be modified. Ideally, the endpoints aren't aware that the messages are bridged.

The integration is more reliable compared to the HTTP alternative due to the at-least-once message delivery mechanism guarantee.

Migration scenarios can be more flexible. For example, endpoints can be migrated from one messaging infrastructure to another as the schedule permits instead of all at once.

Drawbacks

Advanced features of one or both messaging technologies might not be available on the bridged route.

The bridged route needs to consider both technologies' limitations. For example, the maximum message size might be 4 MB in MSMQ but only 64 KB in Azure Storage queues.

Issues and considerations

Consider the following points when implementing the Messaging Bridge pattern:

If one of the integrated systems relies on distributed transactions, for example Microsoft Distributed Transaction Coordinator (DTC), for correctness, you must implement a deduplication mechanism in the bridge.

If one of the systems being integrated doesn't use any messaging infrastructure and can't be modified, you can build the Messaging Bridge between the infrastructure that's used by the other system and a SQL Server-emulated queue. The legacy system can send messages by using the change data capture feature for SQL Server to push its changes to a dedicated queue table. The bridge can forward these messages to the actual messaging infrastructure.

You can use a single queue in each messaging infrastructure, designated as the bridging queue. In this topology, configure the sending system to use that specific queue as the destination for message types that are sent to the other system. You can also use multiple pairs of queues in each messaging infrastructure, so the sender is unaware of the bridge. A shadow queue is created for each destination queue in the destination system's messaging infrastructure. The bridge forwards messages between the shadow queues and their counterparts.

In order to meet the desired availability service-level agreements (SLAs), you might need to scale out the Messaging Bridge by using the Competing consumers approach.

Regular message-processing components use the Retry pattern to handle transient failures. The retry counter limit enables components to detect poison messages and remove them from the queue to unblock processing. The bridge might require a different retry policy to prevent falsely identifying messages as poison if an infrastructure failure occurs. You might use the Circuit Breaker pattern to pause forwarding.

When to use this pattern

Use the Messaging Bridge pattern when you need to:

* Integrate existing systems with minimal need for modification.
* Integrate legacy applications that can't use other messaging technologies.
* Extend existing on-premises applications with cloud-hosted components.
* Connect geo-distributed systems when the internet connection isn't stable.
* Migrate a single distributed system from one messaging infrastructure to another incrementally without the need to migrate the whole system in one effort.

This pattern might not be suitable if:

* At least one of the systems involved relies on a feature of one messaging infrastructure that isn't present in the other.
* Integration is synchronous in nature, and the initiating system requires immediate response.
* Integration has specific functional or nonfunctional requirements, such as security or privacy concerns.
* The volume of data for the integration exceeds the capacity of the messaging system or makes messaging an expensive solution to the problem.

Workload design

An architect should evaluate how the Messaging Bridge pattern can be used in their workload's design to address the goals and principles covered in the Azure Well-Architected Framework pillars. For example:

|  |  |
| --- | --- |
| Pillar | How this pattern supports pillar goals |
| Cost Optimization is focused on sustaining and improving your workload's return on investment. | This intermediary step can increase the longevity of your existing system without the need for rewrites by allowing interoperability with systems that use a different messaging or eventing technology.  [CO:07 Component costs](https://learn.microsoft.com/en-us/azure/well-architected/cost-optimization/optimize-component-costs#determine-the-future-of-the-feature) |
| Operational Excellence helps deliver workload quality through standardized processes and team cohesion. | This decoupling provides flexibility when you transition messaging and eventing technology within your workload or when you have heterogeneous requirements from external dependencies.  [OE:06 Deploying workload changes](https://learn.microsoft.com/en-us/azure/well-architected/operational-excellence/workload-supply-chain) |

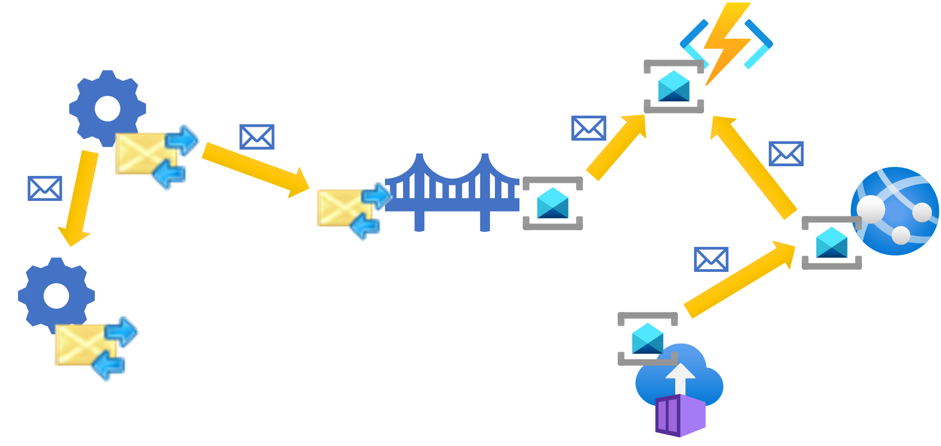
As with any design decision, consider any tradeoffs against the goals of the other pillars that might be introduced with this pattern.

Example

There's an application written in a .NET framework for managing employee scheduling hosted on-premises. The application is well-structured with separate components communicating via MSMQ. The application works, and the workload team has no intention of rewriting it. A new consumer of the scheduling data needs to be built to meet a business need, and the IT strategy calls for building new software as cloud-native applications to optimize the costs and delivery time.

The asynchronous queue-based architecture worked for the workload team in the past, so the team is going to use the same architectural approach but with the modern technology, Service Bus. The workload team doesn't want to introduce synchronous communication between the cloud and the on-premises deployment to mitigate the latency or unavailability of one affecting the other.

The team decides to use the Messaging Bridge pattern to connect the two systems. The pattern consists of two parts. One part receives messages from the existing MSMQ queue and forwards them to Service Bus. The other part takes messages from the Service Bus and forwards them to the existing MSMQ queue.



When the implementation team uses this approach, they utilize existing infrastructure in the existing application to integrate with the new components. The existing application isn't aware that the new components are hosted in Azure. Similarly, the new components communicate with the legacy application in the same way that they communicate between themselves, by sending Service Bus messages. The bridge forwards messages between the two systems.

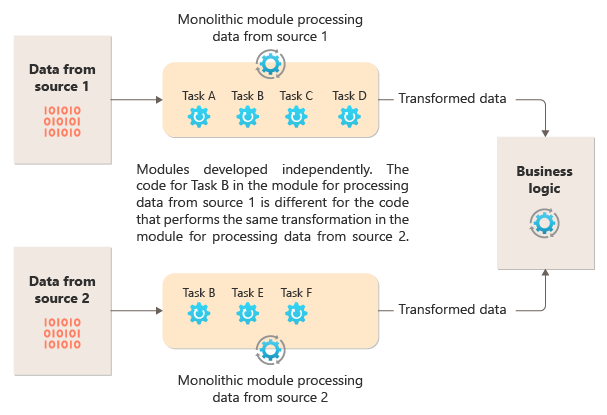
### [Pipes and Filters](https://learn.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters)

Decompose a task that performs complex processing into a series of separate elements that can be reused. Doing so can improve performance, scalability, and reusability of initial steps by allowing task elements that perform the processing to be deployed and scaled independently. Pipes and Filters pattern supports a high level of modularity.

Context and problem

You have a pipeline of sequential tasks that you need to process. A straightforward but inflexible approach to implement this application is to perform this processing in a monolithic module. However, this approach is likely to reduce the opportunities for refactoring the code, optimizing it, or reusing it if parts of the same processing are required elsewhere in the application.

The following diagram illustrates one of the problems with processing data using a monolithic approach, the inability to reuse code across multiple pipelines. In this example, an application receives and processes data from two sources. A separate module processes the data from each source by performing a series of tasks to transform the data before passing the result to the business logic of the application.



Some of the tasks that the monolithic modules perform are functionally similar, but the code has to be repeated in both modules and is likely tightly coupled within its module. In addition to the inability to reuse logic, this approach introduces a risk when requirements change. You must remember to update the code in both places.

There are other challenges with a monolithic implementation unrelated to multiple pipelines or reuse. With a monolith, you don't have the ability to run specific tasks in different environments or scale them independently. Some tasks might be compute-intensive and would benefit from running on powerful hardware or running multiple instances in parallel. Other tasks might not have the same requirements. Further, with monoliths, it's challenging to reorder tasks or to inject new tasks in the pipeline. These changes require retesting the entire pipeline.

Solution

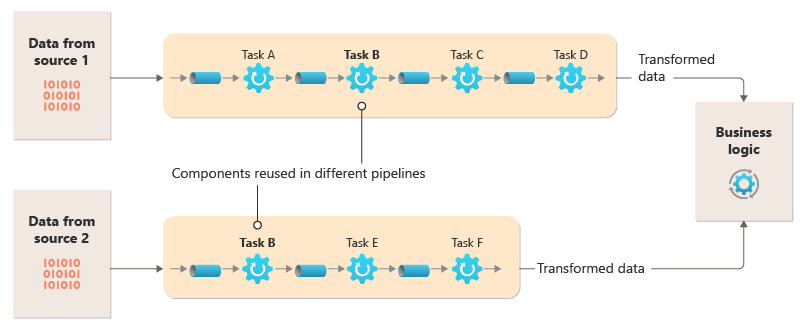
Break down the processing required for each stream into a set of separate components (or filters), each performing a single task. Composite tasks should use multiple filters rather than one. The filters are composed into pipelines by connecting the filters with pipes. Filters are independent, self-contained, and typically stateless. Filters receive messages from an inbound pipe and publish messages to a different outbound pipe. Filters can transform the message or test it against one or more criteria to include conditional logic. Pipes don't perform routing or any other logic. They only connect filters, passing the output message from one filter as the input to the next.

Filters act independently and are unaware of other filters. They're only aware of their input and output schemas. As such, the filters can be arranged in any order so long as the input schema for any filter matches the output schema for the previous filter. Using a standardized schema for all filters enhances the ability to reorder filters. Pipes and filters architecture encourages compositional reuse.

The loose coupling of filters makes it easy to:

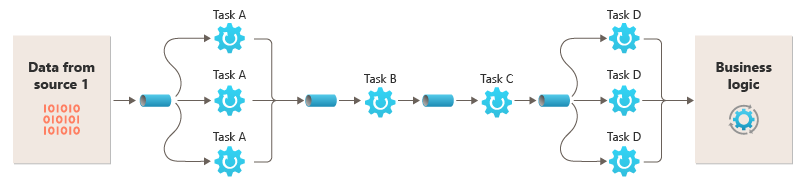
* Create new pipelines composed of existing filters
* Update or replace logic in individual filters
* Reorder filters, when necessary
* Run filters on differing hardware, where required
* Run filters in parallel

This diagram shows a solution implemented with pipes and filters:



The time it takes to process a single request depends on the speed of the slowest filters in the pipeline. One or more filters could be bottlenecks, especially if a high number of requests appear in a stream from a particular data source. The ability to run parallel instances of slow filters enables the system to spread the load and improve throughput.

The ability to run filters on different compute instances enables them to be scaled independently and take advantage of the elasticity that many cloud environments provide. A filter that's computationally intensive can run on high-performance hardware, while other less-demanding filters can be hosted on less-expensive commodity hardware. The filters don't even need to be in the same datacenter or geographic location, enabling each element in a pipeline to run in an environment that's close to the resources it requires. These efforts require specific design techniques such as messaging, multi-threading, and so on to maximize the elasticity of each pipe or filter. This diagram shows an example applied to the pipeline for the data from Source 1:



If the input and output of a filter are structured as a stream, you can perform the processing for each filter in parallel. The first filter in the pipeline can start its work and output its results, which are passed directly to the next filter in the sequence before the first filter completes its work.

Using the Pipes and Filters pattern together with the Compensating Transaction pattern is an alternative approach to implementing distributed transactions. You can break a distributed transaction into separate, compensable tasks, each of which can be implemented via a filter that also implements the Compensating Transaction pattern. You can implement the filters in a pipeline as separate hosted tasks that run close to the data that they maintain.

Issues and considerations

Consider the following points when you decide how to implement this pattern:

* Monolithic nature. This pattern is usually implemented as a monolithic pipeline, so for any change, the entire filter chain should be tested end to end. Also, fault-tolerance for the whole process needs to be considered; if a filter or pipe fails, the whole pipeline is likely to fail.
* Complexity. The increased flexibility that this pattern provides can also introduce complexity, especially if the filters in a pipeline are distributed across different servers.
* Reliability. Use an infrastructure that ensures that data flowing between filters in a pipe aren't lost.
* Idempotency. If a filter in a pipeline fails after receiving a message and the work is rescheduled to another instance of the filter, part of the work might already be complete. If the work updates some aspect of the global state (like information stored in a database), a single update could be repeated. A similar issue might occur if a filter fails after it posts its results to the next filter, but before it indicates that it completed its work successfully. In these cases, another instance of the filter could repeat this work, causing the same results to be posted twice. This scenario could result in subsequent filters in the pipeline processing the same data twice. Therefore, filters in a pipeline should be designed to be idempotent. For more information, see Idempotency Patterns on Jonathan Oliver's blog.
* Repeated messages. If a filter in a pipeline fails after it posts a message to the next stage of the pipeline, another instance of the filter might be run, and it would post a copy of the same message to the pipeline. This scenario could cause two instances of the same message to be passed to the next filter. To avoid this problem, the pipeline should detect and eliminate duplicate messages.

Note: If you implement the pipeline by using message queues (like Azure Service Bus queues), the message queuing infrastructure might provide automatic duplicate message detection and removal.

* Context and state. In a pipeline, each filter essentially runs in isolation and shouldn't make any assumptions about how it was invoked. Therefore, each filter should be provided with sufficient context to perform its work. This context could include a significant amount of state information. If filters use external state, such as data in a database or external storage, then you must consider the impact on performance. Every filter has to load, operate, and persist that state, which adds overhead over solutions that load the external state a single time.
* Message tolerance. Filters must be tolerant of data in the incoming message that they don't operate against. They operate on the data pertinent to them and ignore other data and pass it along unchanged in the output message.
* Error handling - Every filter must determine what to do in the case of a breaking error. The filter must determine if it fails the pipeline or propagates the exception.

When to use this pattern

Use this pattern when:

* The processing required by an application can easily be broken down into a set of independent steps.
* The processing steps performed by an application have different scalability requirements.

Note

* You can group filters that should scale together in the same process. For more information, see the Compute Resource Consolidation pattern.
* You require the flexibility to allow reordering of the processing steps the application performs, or to allow the capability to add and remove steps.
* The system can benefit from distributing the processing for steps across different servers.
* You need a reliable solution that minimizes the effects of failure in a step while data is being processed.

This pattern might not be useful when:

* The application follows a request-response pattern.
* The task processing must be completed as part an initial request, such as a request/response scenario.
* The processing steps performed by an application aren't independent, or they have to be performed together as part of a single transaction.
* The amount of context or state information a step requires makes this approach inefficient. You might be able to persist state information to a database, but don't use this strategy if the extra load on the database causes excessive contention.

Workload design

An architect should evaluate how the Pipes and Filters pattern can be used in their workload's design to address the goals and principles covered in the Azure Well-Architected Framework pillars. For example:

|  |  |
| --- | --- |
| Pillar | How this pattern supports pillar goals |
| Reliability design decisions help your workload become resilient to malfunction and to ensure that it recovers to a fully functioning state after a failure occurs. | The single responsibility of each stage enables focused attention and avoids the distraction of commingled data processing.  - RE:01 Simplicity  - RE:07 Background jobs |

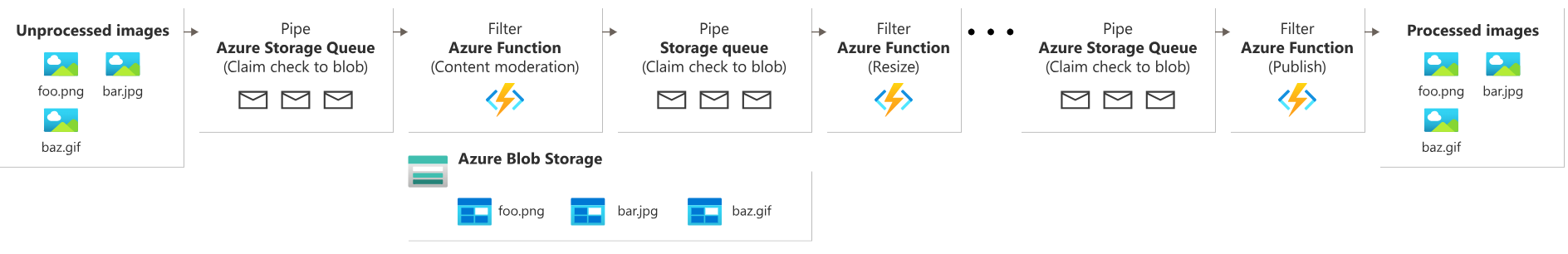
Example

You can use a sequence of message queues to provide the infrastructure required to implement a pipeline. An initial message queue receives unprocessed messages that become the start of the pipes and filters pattern implementation. A component implemented as a filter task listens for a message on this queue, performs its work, and then posts a new or transformed message to the next queue in the sequence. Another filter task can listen for messages on this queue, process them, post the results to another queue, and so on, until the final step that ends the pipes and filters process. This diagram illustrates a pipeline that uses message queues:



An image processing pipeline could be implemented using this pattern. If your workload takes an image, the image could pass through a series of largely independent and reorderable filters to perform actions such as:

* content moderation
* resizing
* watermarking
* reorientation
* Exif metadata removal
* Content delivery network (CDN) publication



In this example, the filters could be implemented as individually deployed Azure Functions or even a single Azure Function app that contains each filter as an isolated deployment. The use of Azure Function triggers, input bindings, and output bindings can simplify the filter code and work automatically with a queue-based pipe using a claim check to the image to process.

Here's an example of what one filter, implemented as an Azure Function, triggered from a Queue Storage pipe with a claim Check to the image, and writing a new claim check to another Queue Storage pipe might look like. We've replaced the implementation with pseudocode in comments for brevity. More code like this can be found in the demonstration of the Pipes and Filters pattern available on GitHub.

**C# Code**

// This is the "Resize" filter. It handles claim checks from input pipe, performs the

// resize work, and places a claim check in the next pipe for anther filter to handle.

[Function(nameof(ResizeFilter))]

[QueueOutput("pipe-fjur", Connection = "pipe")] // Destination pipe claim check

public async Task<string> RunAsync(

[QueueTrigger("pipe-xfty", Connection = "pipe")] string imageFilePath, // Source pipe claim check

[BlobInput("{QueueTrigger}", Connection = "pipe")] BlockBlobClient imageBlob) // Image to process

{

\_logger.LogInformation("Processing image {uri} for resizing.", imageBlob.Uri);

// Idempotency checks

// ...

// Download image based on claim check in queue message body

// ...

// Resize the image

// ...

// Write resized image back to storage

// ...

// Create claim check for image and place in the next pipe

// ...

\_logger.LogInformation("Image resizing done or not needed. Adding image {filePath} into the next pipe.", imageFilePath);

return imageFilePath;

}

The Spring Integration Framework has an implementation of the pipes and filters pattern.

[Idempotency Patterns](https://blog.jonathanoliver.com/idempotency-patterns/)

### [Claim Check](https://learn.microsoft.com/en-us/azure/architecture/patterns/claim-check)

### [Competing Consumers](https://learn.microsoft.com/en-us/azure/architecture/patterns/competing-consumers)

### [Compute Resource Consolidation](https://learn.microsoft.com/en-us/azure/architecture/patterns/compute-resource-consolidation)

### [Compensating Transaction](https://learn.microsoft.com/en-us/azure/architecture/patterns/compensating-transaction)

### Backends for Frontends

[Backends for Frontends](https://learn.microsoft.com/en-us/azure/architecture/patterns/backends-for-frontends) creates separate backend services for different types of clients, such as desktop and mobile. That way, a single backend service doesn't need to handle the conflicting requirements of various client types. This pattern can help keep each microservice simple, by separating client-specific concerns.

### Bulkhead

[Bulkhead](https://learn.microsoft.com/en-us/azure/architecture/patterns/bulkhead) isolates critical resources, such as connection pool, memory, and CPU, for each workload or service. By using bulkheads, a single workload (or service) can't consume all of the resources, starving others. This pattern increases the resiliency of the system by preventing cascading failures caused by one service.

### Gateway Aggregation

[Gateway Aggregation](https://learn.microsoft.com/en-us/azure/architecture/patterns/gateway-aggregation) aggregates requests to multiple individual microservices into a single request, reducing chattiness between consumers and services.

### Gateway Offloading

[Gateway Offloading](https://learn.microsoft.com/en-us/azure/architecture/patterns/gateway-offloading) enables each microservice to offload shared service functionality, such as the use of SSL certificates, to an API gateway.

### Gateway Routing

[Gateway Routing](https://learn.microsoft.com/en-us/azure/architecture/patterns/gateway-routing) routes requests to multiple microservices using a single endpoint, so that consumers don't need to manage many separate endpoints.

### [Asynchronous Request Reply](https://learn.microsoft.com/en-us/azure/architecture/patterns/async-request-reply)

### [Cache Aside](https://learn.microsoft.com/en-us/azure/architecture/patterns/cache-aside)

### [Choreography](https://learn.microsoft.com/en-us/azure/architecture/patterns/choreography)

### [Circuit Breaker](https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker)

### [CQRS](https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs)

### [Deployment Stamps](https://learn.microsoft.com/en-us/azure/architecture/patterns/deployment-stamp)

### [Edge Workload Configuration](https://learn.microsoft.com/en-us/azure/architecture/patterns/edge-workload-configuration)

### [Event Sourcing](https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing)

### [External Configuration Store](https://learn.microsoft.com/en-us/azure/architecture/patterns/external-configuration-store)

### [Federated Identity](https://learn.microsoft.com/en-us/azure/architecture/patterns/federated-identity)

### [Gatekeeper](https://learn.microsoft.com/en-us/azure/architecture/patterns/gatekeeper)

### [Geode](https://learn.microsoft.com/en-us/azure/architecture/patterns/geodes)

### [Health Endpoint Monitoring](https://learn.microsoft.com/en-us/azure/architecture/patterns/health-endpoint-monitoring)

### [Index Table](https://learn.microsoft.com/en-us/azure/architecture/patterns/index-table)

### [Leader Election](https://learn.microsoft.com/en-us/azure/architecture/patterns/leader-election)

### [Materialized View](https://learn.microsoft.com/en-us/azure/architecture/patterns/materialized-view)

### [Priority Queue](https://learn.microsoft.com/en-us/azure/architecture/patterns/priority-queue)

### [Publisher/ Subscriber](https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber)

### [Quarantine](https://learn.microsoft.com/en-us/azure/architecture/patterns/quarantine)

### [Queue Based Load Levelling](https://learn.microsoft.com/en-us/azure/architecture/patterns/queue-based-load-leveling)

### [Rate Limiting](https://learn.microsoft.com/en-us/azure/architecture/patterns/rate-limiting-pattern)

### [Retry](https://learn.microsoft.com/en-us/azure/architecture/patterns/retry)

### [Saga](https://learn.microsoft.com/en-us/azure/architecture/patterns/saga)

### [Scheduler Agent Supervisor](https://learn.microsoft.com/en-us/azure/architecture/patterns/scheduler-agent-supervisor)

### [Sequential Convoy](https://learn.microsoft.com/en-us/azure/architecture/patterns/sequential-convoy)

### [Scharding](https://learn.microsoft.com/en-us/azure/architecture/patterns/sharding)

### Sidecar

### Static Content Hosting

### Throttling

### Valet Key

## Cloud Pattern Catalog

<https://learn.microsoft.com/en-us/azure/architecture/patterns/>

## Monolith to Microservices

<https://learn.microsoft.com/en-us/training/modules/microservices-architecture/>

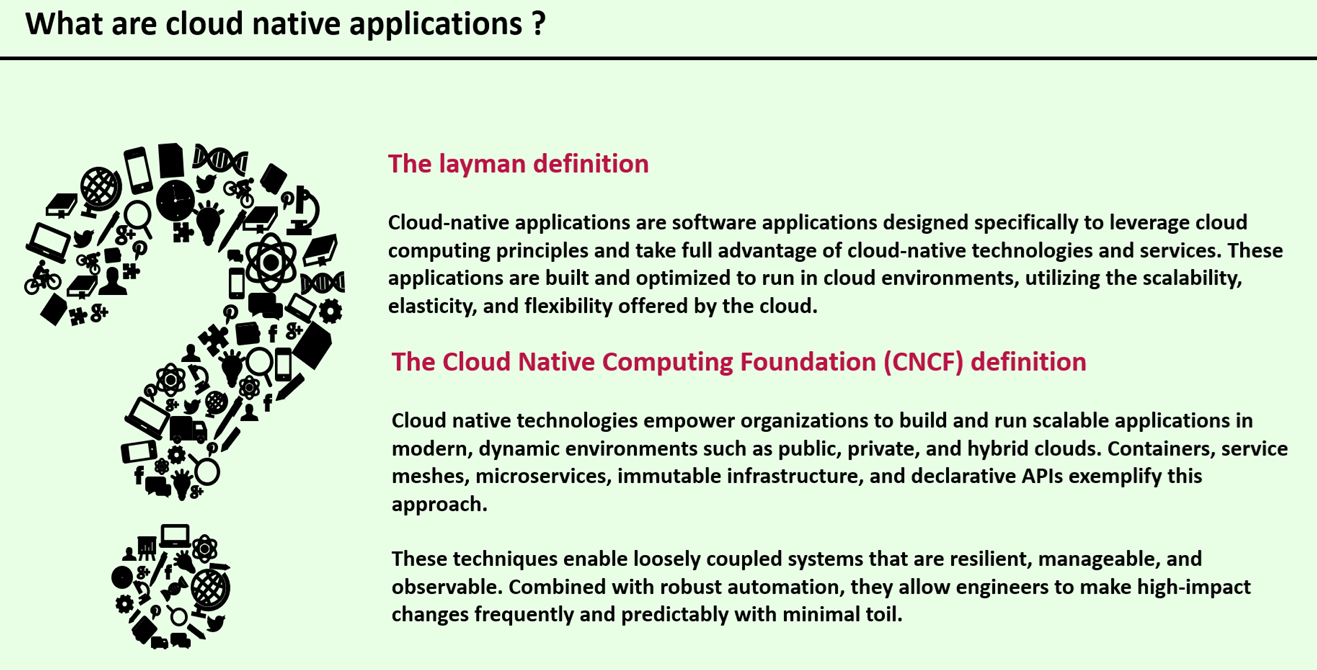
## Enterprise Application Architecture – Martin Fowler

<https://martinfowler.com/eaaCatalog/>

# Cloud Native Apps, 15 Factor Methodology

## Cloud Native Applications

This section is a small theory section where we are going to discuss about the definition of cloud native applications. What are the characters of Cloud Native applications and 12 factor methodology and 15 factor methodology? These are the standards that any microservice developer has to know because by following these standards only we are going to build many concepts inside the microservice in the upcoming sections. Now coming to the question, what is a cloud native application?



You might be hearing this jargon or this buzzword a lot inside your project or inside some blogs or podcasts or YouTube videos. So let's try to understand what are these cloud native applications and how they are related to the microservices. First, I'll try to provide a layman language. Like any non-technical person can understand this definitions.

**Layman Definition**

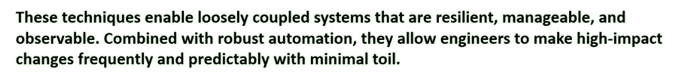
Cloud Native applications are the software applications that are designed and developed specifically to leverage the cloud computing principles and take the full advantage of cloud native technologies and services, which means these applications are built and optimized to run inside any cloud environments by utilizing the cloud advantages like scalability, elasticity and flexibility.

In simple words, cloud native applications are built for cloud environments so that the organizations can take complete advantage of cloud provider services and technologies. Now let's try to understand the official definition of cloud native applications from the Cloud Native Computing Foundation. So if you see here,

**Cloud Native Computing Foundation ( CNCF ) Definition**

Cloud native technologies empower organizations to build and run scalable applications in modern dynamic environments such as public, private and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

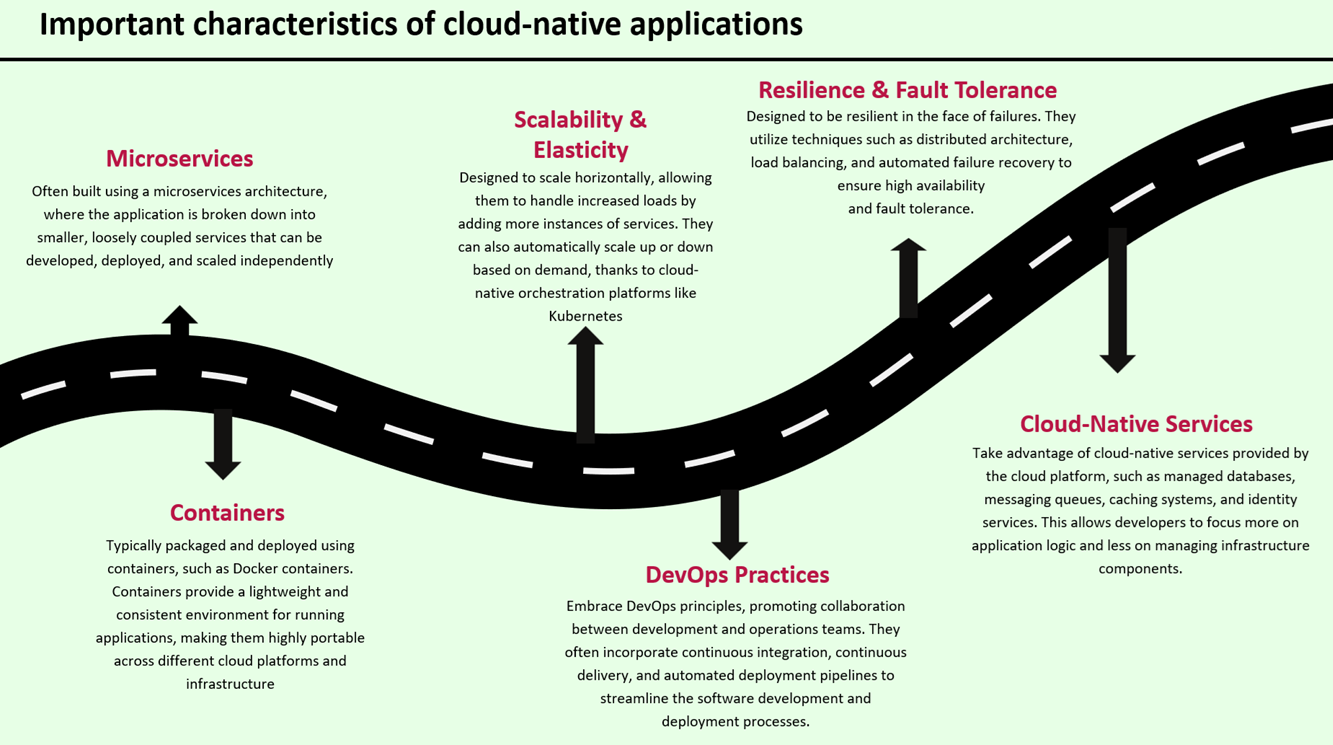
So the cloud environment can be anything, it can be it can be your own data center, it can be GCP, azure, or it can be a hybrid combination of private cloud and public cloud. Regardless of what type of cloud computing you're trying to use, your cloud native applications will work. So how this is possible is when we are trying to build cloud native applications, we will try to leverage technologies like containers, service meshes, microservices, immutable infrastructure and declarative APIs. So all these technologies make your applications to run on any cloud so you won't get vendor lock in with any particular cloud environment. On top of these cloud advantages.



These techniques also enable developing loosely coupled systems that are resilient, manageable and observable. Resilient means they can withstand any failures manageable you already know they are easy to manage and observable means we'll get to know everything about our applications, how it is working, are there any issues? There is also a separate section inside this course about the observability. We'll be discussing more details in that section. So with all these new techniques, along with the robust automation, these cloud native applications, they allow engineers to develop high impact changes frequently and predictably with minimal toil. So this will give freedom to the organizations and the developers if they want to make a small change or if they want to make a small enhancement, they can do that very quickly without any defects or regression issues being introduced. Because they are all loosely coupled and it is very easy to test them to develop new enhancements. I hope this is clear to you. So if anyone asks you what is the definition of cloud native application inside an interview or inside any project, first try to understand whether the opposite person who is asking is a technical person or a non-technical person. If he is a non-technical person, please share this layman definition. Whereas if the person is a technical person, you can share the official definition from the Cloud Native Computing Foundation.

## Important Characteristics of Cloud Native Applications

Right now we are clear about the definition of cloud native applications. inside this lecture, let me explain you about the important characteristics of cloud native applications. So whenever you see these characters for any application, then you can easily assume they are cloud native applications. The very first main character of Cloud Native application is microservice. When you are building microservice based applications, which are like loosely coupled and smaller in nature, then that gives a flexibility to you to develop them parallelly and deploy and scale independently. So this is the main advantage of the cloud native applications.



So cloud native application is a broader topic and under that topic microservices is one of the important feature. After building microservices like separating your business logic, you will obviously containerize your applications with the help of Docker or any other containerization software.

**Containers**

So what are these containers we already discussed? These are typically packaged and deployed using Docker containers. These containers provide a lightweight and consistent environment for running applications and making them highly portable across different cloud platforms and infrastructure. With the help of these containers, only the code will work very similarly regardless of which cloud environment you are trying to deploy. You take this container, you try to deploy inside your local system and you deploy inside the AWS, GCP, Azure Cloud in all the places it is going to work in very similar manner. Whereas this is not the scenario with a monolithic application. Your monolithic application won't give such kind of flexibility. You need to put a lot of efforts to bring that consistency across all the cloud platforms.

**Scalability and Elasticity**

The next character of cloud native applications is they provide scalability and elasticity. Since you are building your applications based upon microservices and with the help of containers, they can be easily scaled horizontally allowing them to handle any kind of traffic that comes towards your applications. So adding more instances of services is going to be super easy and this can be achieved automatically with the help of container orchestration platforms like Kubernetes. We also have separate section inside this course where I'll be showing you how automatically scale your microservice applications with the help of Kubernetes.

**Devops Practices**

Moving on to the next principle, these cloud native applications, they follow DevOps practices, by embracing all these DevOps principles. They promote a collaboration culture between the development and operations teams. Whenever these cloud native applications are being built, they will not be any blame game between the developers and operations team when they are building cloud native application because they are following these DevOps practices and with the help of these DevOps practices, they will incorporate continuous integration, continuous delivery and automated deployment pipelines to streamline the software development and deployment process. So these cloud native applications, they will give complete flexibility to the organizations, whether they want to do continuous integration only or whether they want to go with the continuous delivery or if they want to go with the continuous deployment as well.

**Resilience and Fault Tolerance**

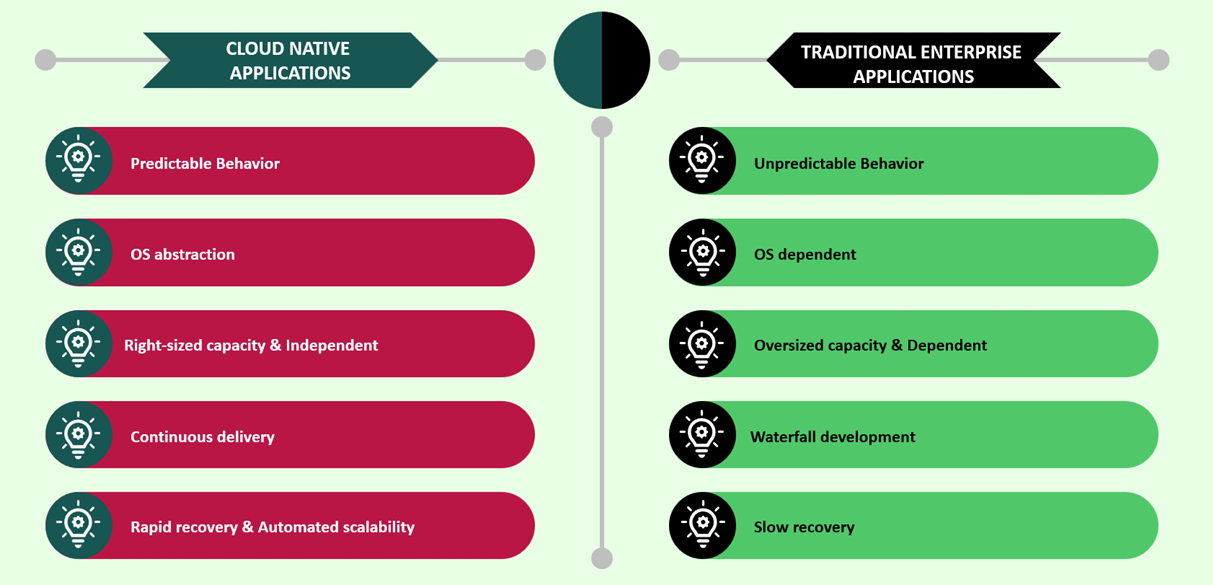
The next character of native applications is, they are resilient and fault tolerance in nature. Whatever applications that we are going to develop with the help of cloud native principles, they can withstand any kind of failures which will make them resilient and they will utilize techniques such as distributed architecture, load balancing and automated fail recovery to ensure high availability and fault tolerance. One example that I can give here is, think like you have a microservice by following the cloud native application principles, you will deploy this microservice in multiple locations. Even if one of the location has a downtime due to some power outage or due to some internet issue, the microservice will continue to work from the other location that you have deployed. And at the same time, for some reason, if one of your microservice instance is not working platforms like Kubernetes, they can automatically shut down that microservice instance and bring up a new instance. This way you are ensuring the failure recovery automatically and bringing fault tolerance inside your applications.

**Cloud Native Services**

Moving on to the next character of cloud Native applications, which is they utilize cloud native services to a great extent. Like I said, cloud native applications means they are developed to leverage the cloud environments to a great extent. When an organization or an application uses these cloud native services provided they don't have to focus on infrastructure because all these services will be monitored and maintained by the cloud platform provider itself. With that, the developers and the organizations they can simply focus more on the application logic and enhancing the business logic with a very less focus on the infrastructure components. Like you can see these are all the important characters of cloud native applications. Inside this course, we are going to touch all of these characters in the various sections. Whenever you see a application is following all these characters, then you can simply assume that is an cloud native application.

## Difference B/W Cloud Native and Traditional Apps

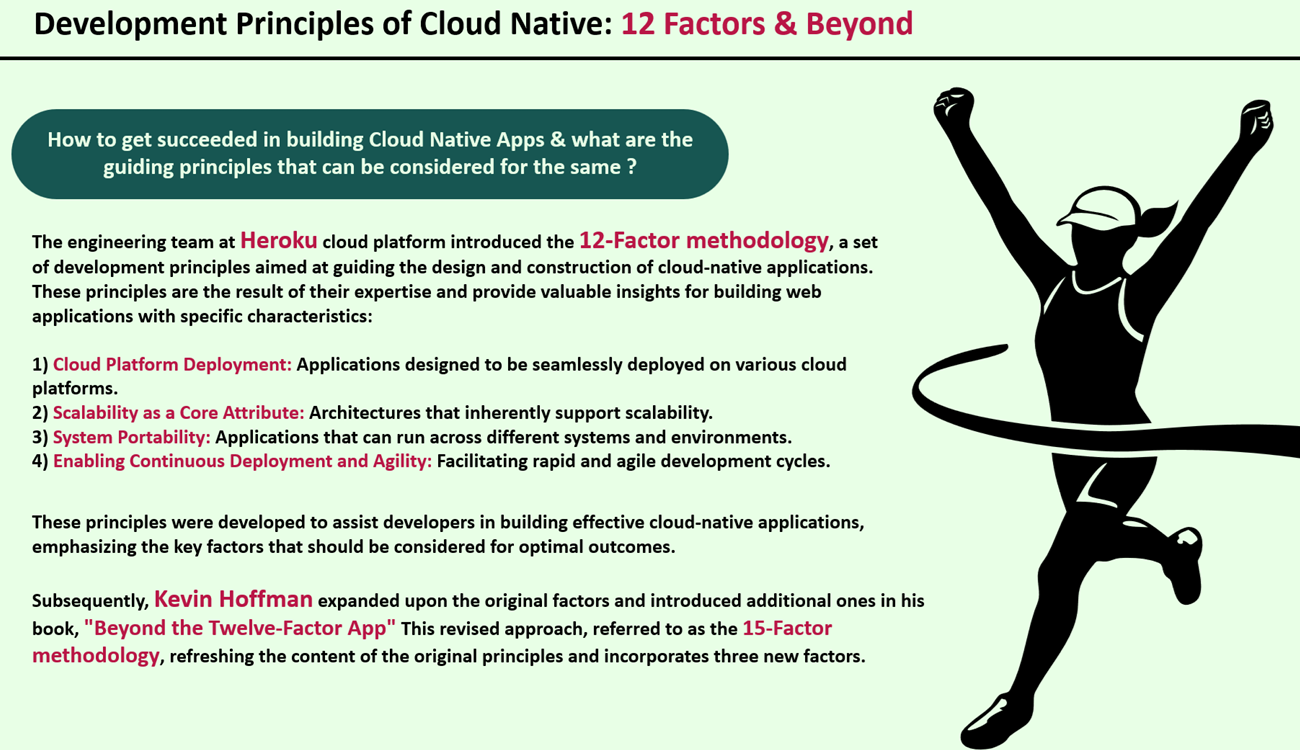
The very first difference of cloud native application and traditional applications is, cloud native applications will have predictable behavior, whereas with traditional applications, the behavior is going to be unpredictable. For example, inside microservice environment, if there is an issue, we can easily track that where the issue is because all your business logic is loosely coupled and separated into multiple microservices, so you can easily predict where the exception happens. Whereas inside monolithic application, since all your business logic clubbed together, you never know where the exception happens until your developer put a lot of efforts by debugging all the lines of code that you have inside your application.



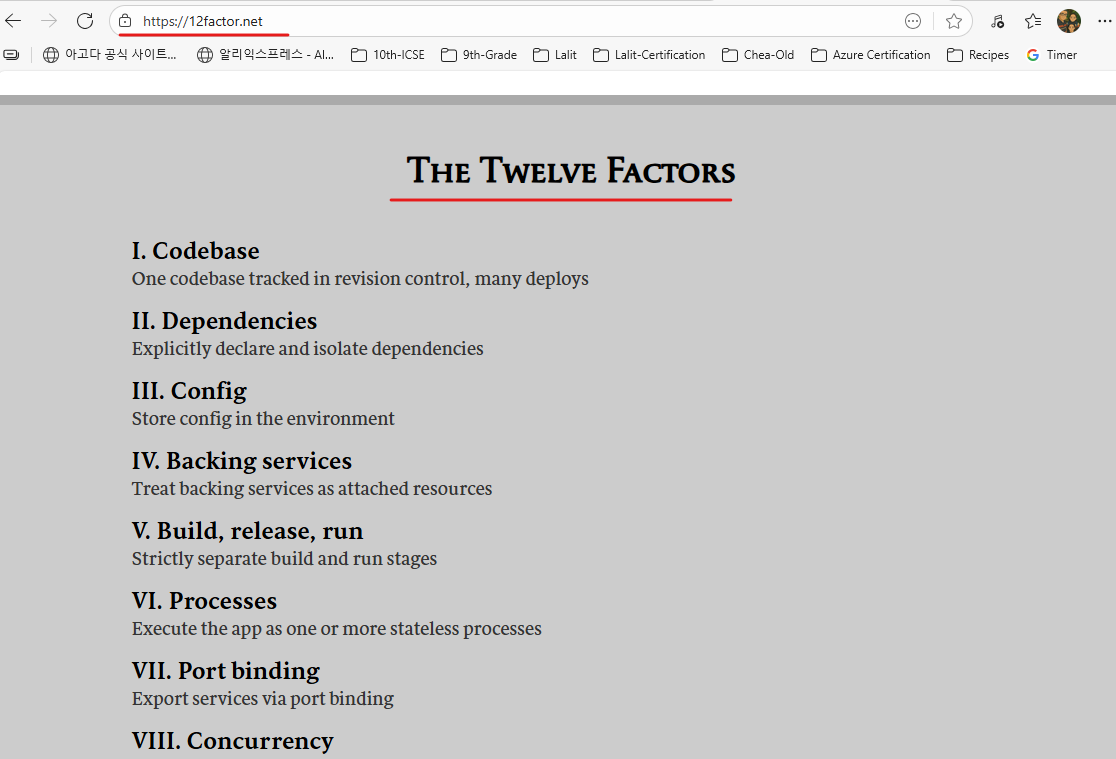
The next difference that we have here is cloud native applications they are not OS dependent. They will abstract the operating system, whereas traditional enterprise applications, they are dependent on operating system. This is because inside cloud native applications we are going to adopt docker containers and Docker images where they will abstract the operating system and they will work in any operating system. The next difference that we have here is, cloud native applications they are rightly sized and they are following the right capacity and they work in an independent manner. Whereas with traditional applications they are oversized capacity because you have all your business logic inside a single application or inside a single codebase. That's why most of the times traditional enterprise applications, they are dependent on each other. Moving on to the next difference cloud native applications they support continuous delivery with the help of DevOps principles and automation, whereas traditional enterprise applications, they follow waterfall development. They won't support Agile style of working inside any organization. Then the last difference that we have here is Cloud Native Application supports rapid recovery and automated scalability. Inside cloud native applications if one of the microservice instance is not working properly Kubernetes, they can automatically create a new instance and try to recover automatically and also scale your applications automatically based upon the incoming traffic. Whereas with traditional applications, since they are not going to use Docker containers, obviously they cannot rely on the platforms like Kubernetes with that reasons with the traditional enterprise applications, the recovery is going to be super slow and at the same time there won't be any automated style of recovery and scalability. From these differences It is very clear that Cloud Native application is a clear winner compared to the traditional enterprise applications. So now you agree that cloud native applications are good and that's why we are trying to learn and build microservices inside this course. But here you may have a question like do we have any principles or guidelines that we need to follow while building these cloud native applications in order to get all these advantages? Off course, there are good guidelines and principles that we can follow while building these cloud native applications or microservices.

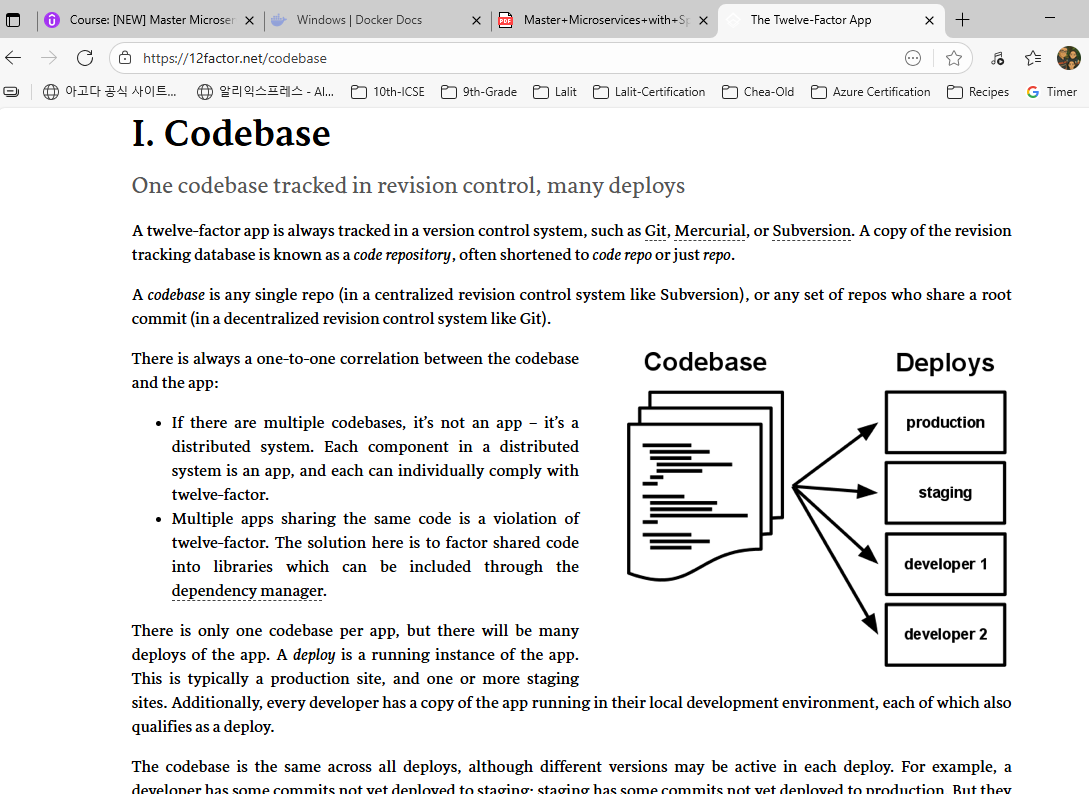
## 12 Factors and Beyond

Many of us who is trying to build cloud native applications or microservices will have a basic question, which is how to get succeeded in building a better cloud native application or a better microservices applications. Are there any guiding principles that can be considered for the same? To answer your question, yes, we do have standards, so let's try to understand them. Back in 2012, the engineering team at Heroku Cloud Platform, they introduced 12 factor methodology. So these are the 12 different development principles that are aimed at guiding the developers in designing and developing cloud native applications. These 12 different principles under the 12 factor methodology are the result of the Heroku team expertise from many, many years building these cloud native applications.

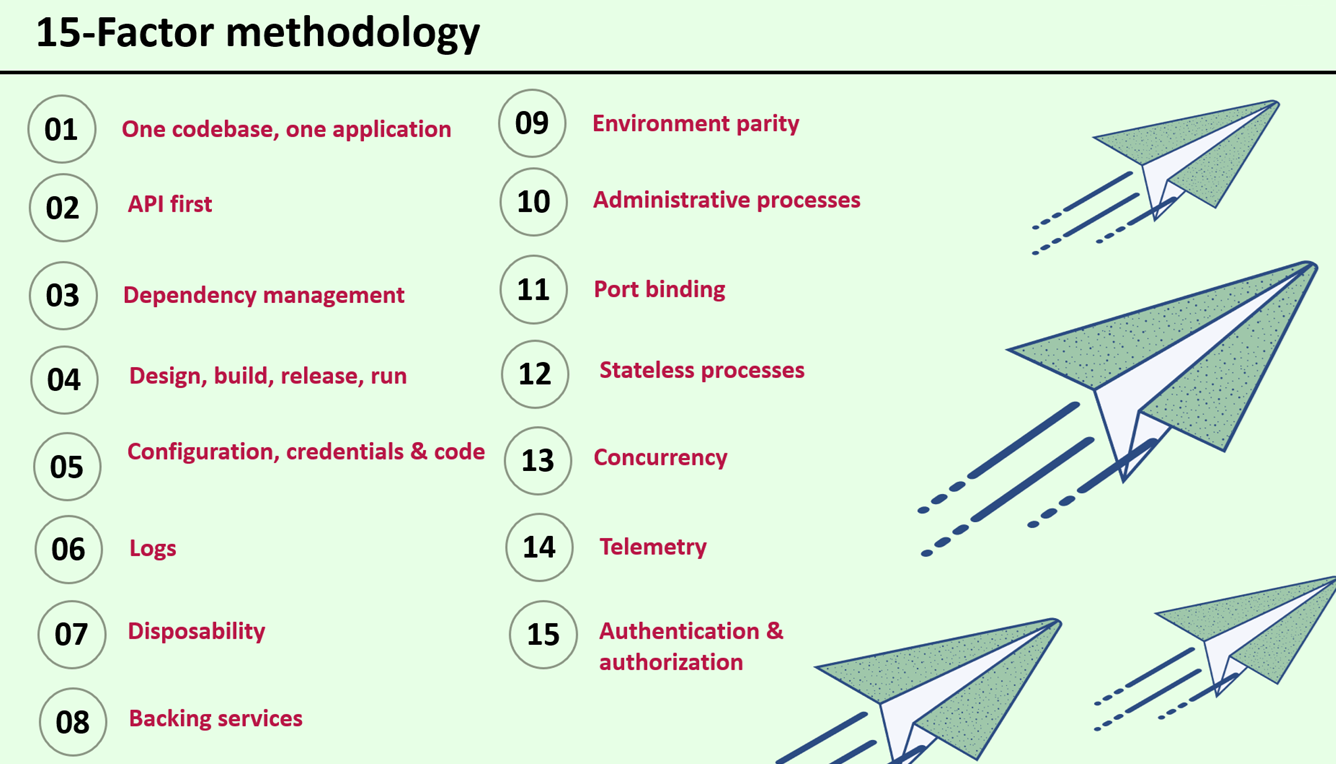


When you follow these 12 factor methodology, your applications will have advantages like ready for cloud platform deployment regardless of whatever cloud platform that you choose. You can seamlessly deploy your cloud native applications. And at the core of this cloud native applications, they are going to support scalability and elasticity. And very similarly, these cloud native systems that you are going to develop with the help of 12 factor methodology, they allow system portability, which means your applications can run across different systems and environments without any issues. And at last these applications will support continuous deployment and agility. So in the initial days, everyone are recommended to follow these 12 factor methodology. But later on, after few years since the technology is getting evolved day by day, a person with the name Kevin Hoffman, he expanded these 12 factor methodology by adding three more principles. And in his book with the name Beyond the 12 Factory app, he revised all these 12 factor methodology. And on top of that, he also introduced three more development principles. And with that, a new 15 factor methodology came into picture. And this is the most latest one right now everyone is referring to. So do you want to see what are these principles? Let's try to see the summary of these methodologies. So these are the 15 methodologies are the principles or guidelines that any developer or organization should follow while building cloud native applications. Without following these standards, they can't tell their applications as cloud native applications because they will end up with some problems since they are not following these standards. These standards are built based upon the development experience of cloud native applications. If you see first, this is the website like everyone used to refer 12factor.net So inside this website you can read all the 12 factor methodologies and if you can scroll down here, these are the 12 factors. If you can click on each of them, there is a detailed description about each of these methodologies.

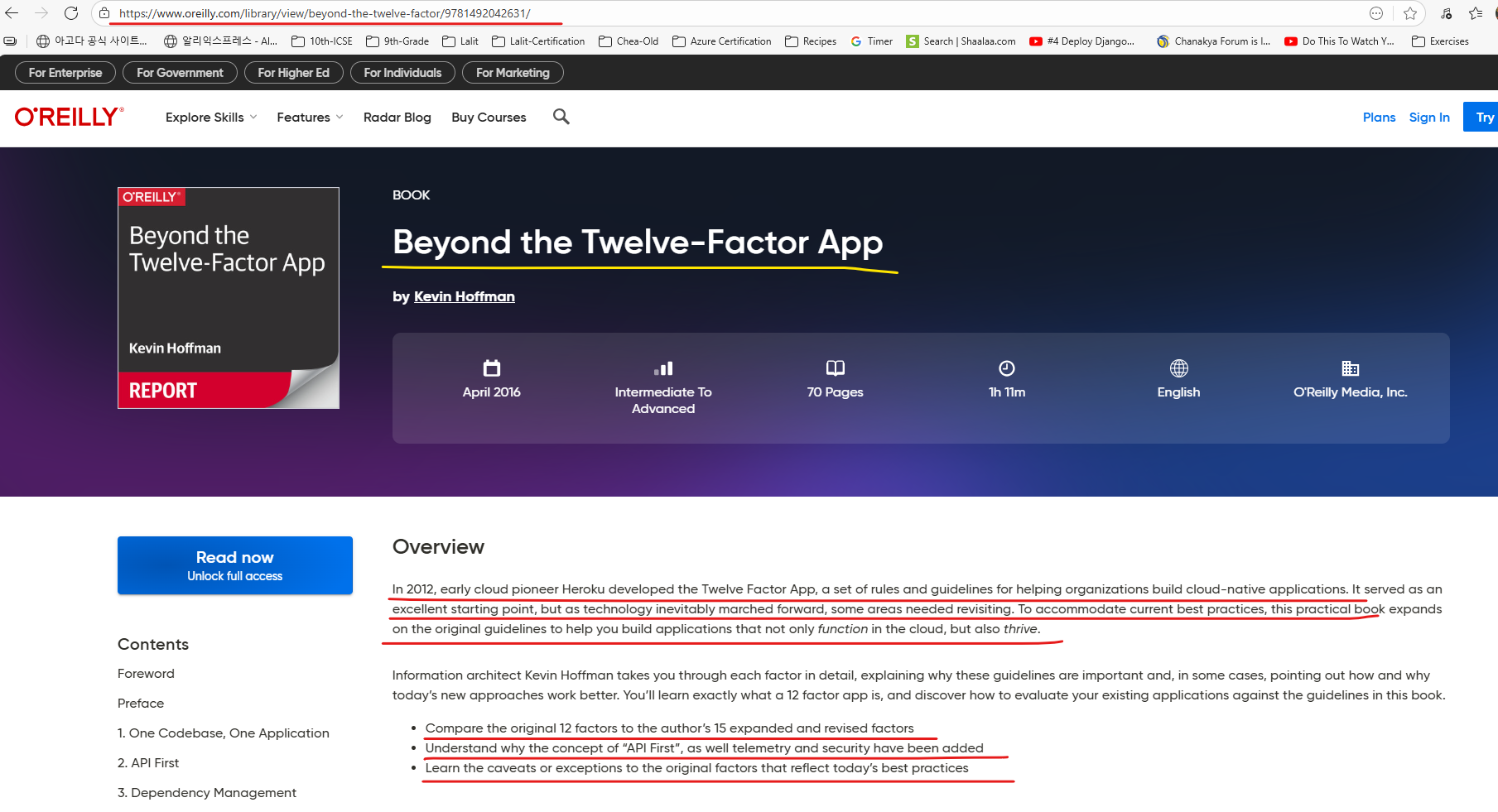




So don't worry, I'm going to explain them inside this course. You don't need to read all of them and digest them. I'm going to explain them in an easily understandable format.



But like I said, these Twelve-Factor app is right now outdated. Like I said, Kevin Hoffman, he wrote a book with the name Beyond the Twelve-Factor App. And inside this book he added three more principles or guidelines. And with that this became 15 factor methodology. You can see the description of this book. In 2012, Early cloud pioneer Heroku developed the 12 Factor app, a set of rules and guidelines for helping organizations for building cloud native applications. It served as an excellent starting point, but as a technology changes day by day, some areas need revisiting to accommodate the current best practices this practical book expands on the original guidelines to help you to build applications that not only function in the cloud but also thrive. I hope you are clear. Like what is the difference between 12 Factor app and the 15 factor app methodology? So 15 factor methodology is the most updated one.



<https://www.oreilly.com/library/view/beyond-the-twelve-factor/9781492042631/>

If you attend an interview focusing on microservices, definitely there will be some questions around these 12 factor methodology or 15 factor methodology. That's why it is very important to understand these methodologies one by one. So let's try to deep dive into these methodologies and learn quickly about them.

## \*\*\* 15-Factors Deep Dive [ Pending ] \*\*\*

Start from: <https://tsys.udemy.com/course/master-microservices-with-spring-docker-kubernetes/learn/lecture/39944388#overview>

# Architecting Container and Microservice Based Applications

<https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/>

# Microservice Pre-requisites

<https://martinfowler.com/bliki/MicroservicePrerequisites.html>

# Martin Fowler

<https://www.martinfowler.com/tags/microservices.html>

# IntelliJ ShortCuts

<https://www.jetbrains.com/help/idea/mastering-keyboard-shortcuts.html>

| **Shortcut** | **Action** |
| --- | --- |
| Double Shift | [Search Everywhere](https://www.jetbrains.com/help/idea/searching-everywhere.html)  Quickly find any file, action, symbol, tool window, or setting in IntelliJ IDEA, in your project, and in the current Git repository. |
| Ctrl + Shift + A | [Find Action](https://www.jetbrains.com/help/idea/searching-everywhere.html#find_action)  Find a command and execute it, open a tool window, or search for a setting. |
| Alt + Enter | [Show Context Actions](https://www.jetbrains.com/help/idea/intention-actions.html)  Quick-fixes for highlighted errors and warnings, intention actions for improving and optimizing your code. |
| F2  Shift + F2 | [Navigate between code issues](https://www.jetbrains.com/help/idea/navigating-through-the-source-code.html#navigate-errors-warnings)  Jump to the next or previous highlighted error. |
| Ctrl + E | [View recent files](https://www.jetbrains.com/help/idea/recent-files-and-changes.html#recent_files)  Select a recently opened file from the list. |
| Ctrl + Shift + Enter | [Complete Current Statement](https://www.jetbrains.com/help/idea/working-with-source-code.html#editor_statement_select)  Insert any necessary trailing symbols and place the caret where you can start typing the next statement. |
| Ctrl + Alt + L | [Reformat Code](https://www.jetbrains.com/help/idea/reformat-and-rearrange-code.html#reformat_code)  Reformat the whole file or the selected fragment according to the current code style settings. |
| Ctrl + Alt + Shift + T | [Invoke refactoring](https://www.jetbrains.com/help/idea/refactoring-source-code.html)  Refactor the element under the caret, for example, safe delete, copy, move, rename, and so on. |
| Ctrl + W  Ctrl + Shift + W | [Extend or shrink selection](https://www.jetbrains.com/help/idea/working-with-source-code.html#editor_code_selection)  Increase or decrease the scope of selection according to specific code constructs. |
| Ctrl + 0  Ctrl + Shift + 0 | [Add/remove line or block comment](https://www.jetbrains.com/help/idea/working-with-source-code.html#editor_lines_code_blocks)  Comment out a line or block of code. |
| Ctrl + B | [Go To Declaration](https://www.jetbrains.com/help/idea/navigating-through-the-source-code.html#go_to_declaration)  Navigate to the initial declaration of the instantiated class, called method, or field. |
| Alt + F7 | [Find Usages](https://www.jetbrains.com/help/idea/find-highlight-usages.html#find-usages)  Show all places where a code element is used across your project. |
| Alt + 1 | Focus the Project tool window |
| Esc | Focus the editor |

# Free a Occupied port on Windows

C:\Users\Lalit> netstat -ano | findstr :8080

TCP 0.0.0.0:8080 0.0.0.0:0 LISTENING 5820

TCP [::]:8080 [::]:0 LISTENING 5820

C:\Users\Lalit>tasklist | findstr 8080

svchost.exe 18080 Services 0 14,500 K

C:\Users\Lalit>taskkill /F /PID 5820

ERROR: The process with PID 5820 could not be terminated.

Reason: Access is denied.

Need to be in administration mode first

C:\Users\Lalit>taskkill /F /PID 5820

SUCCESS: The process with PID 5820 has been terminated.

