Architecture styles

07/11/2025

An *architecture style* is a family of architectures that share specific characteristics. For example, N-tier is a common architecture style. More recently, microservice architectures are starting to gain favor. Architecture styles don't require the use of specific technologies, but some technologies are better suited for certain architectures. For example, containers are well-suited for microservices.

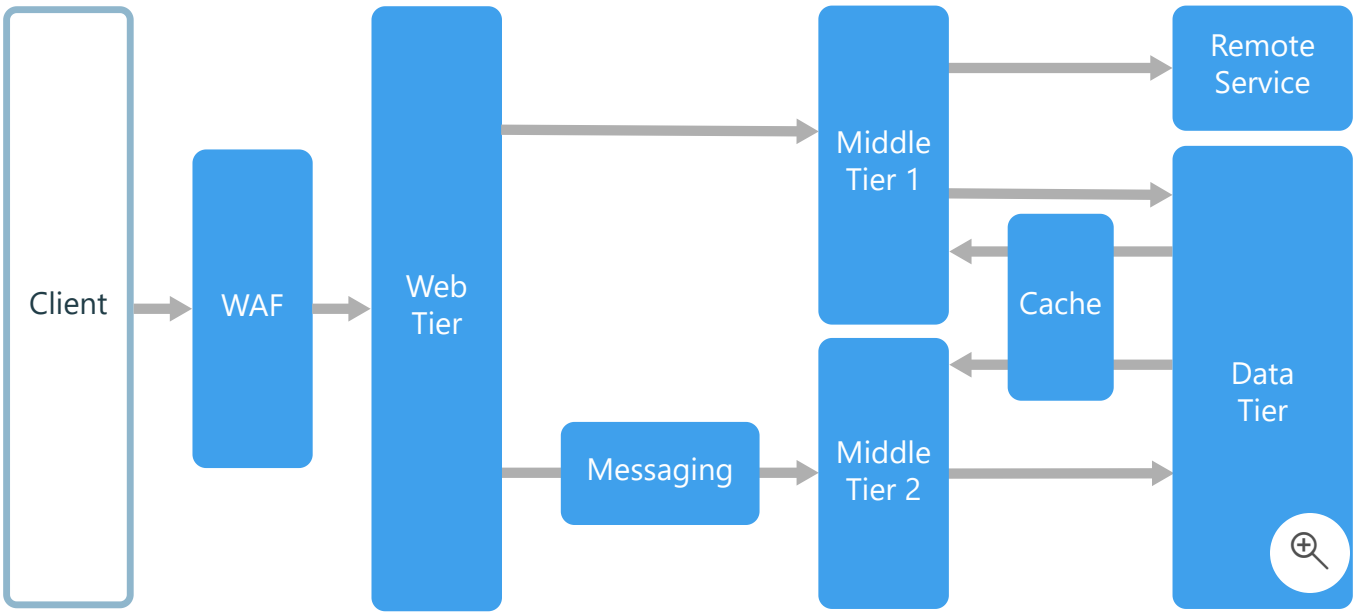
We have identified a set of architecture styles that are commonly found in cloud applications. The article for each style includes:

- A description and logical diagram of the style.
- Recommendations for when to choose this style.
- Benefits, challenges, and best practices.
- A recommended deployment using relevant Azure services.

A quick tour of the styles

This section gives a quick tour of the architecture styles that we've identified, along with some high-level considerations for their use. This list isn't exhaustive. Read more details in the linked topics.

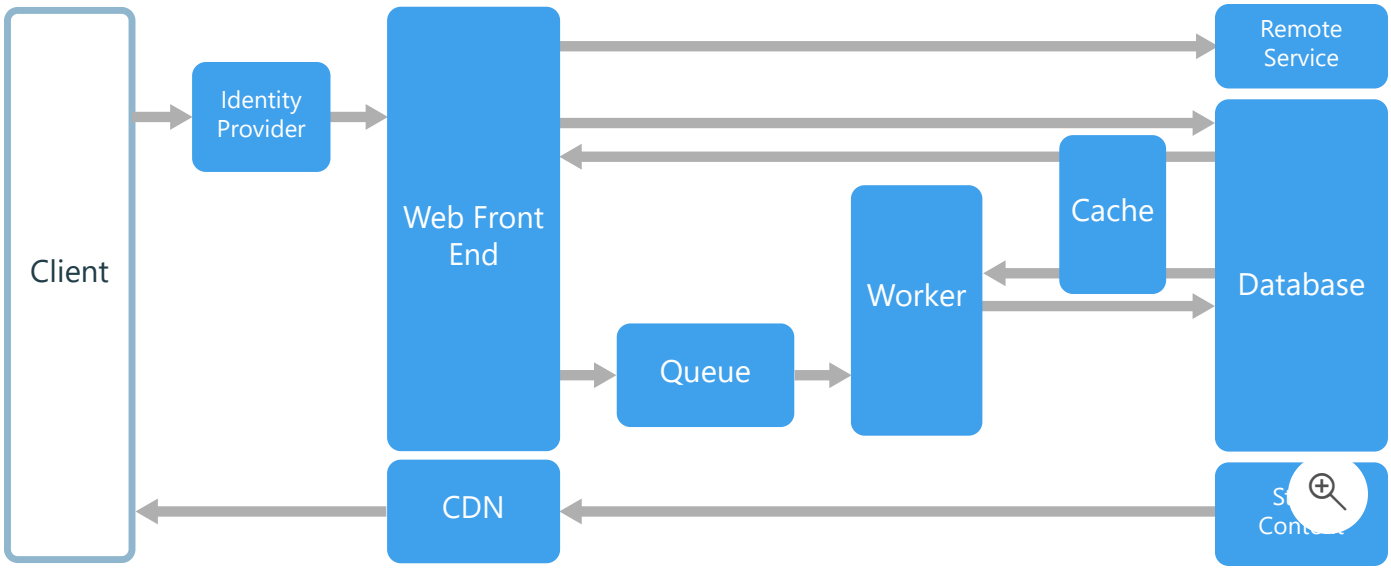
N-tier



N-tier is a traditional architecture for enterprise applications. Dependencies are managed by dividing the application into *layers* that perform logical functions, such as presentation, business logic, and data access. A layer can only call into layers that sit below it. However, this horizontal layering can be a liability. It can be hard to introduce changes in one part of the application without touching the rest of the application. That makes frequent updates a challenge, limiting how quickly new features can be added.

N-tier is well-suited for migrating existing applications that already use a layered architecture. For that reason, N-tier is most often seen in infrastructure as a service (IaaS) solutions or applications that use a combination of IaaS and managed services.

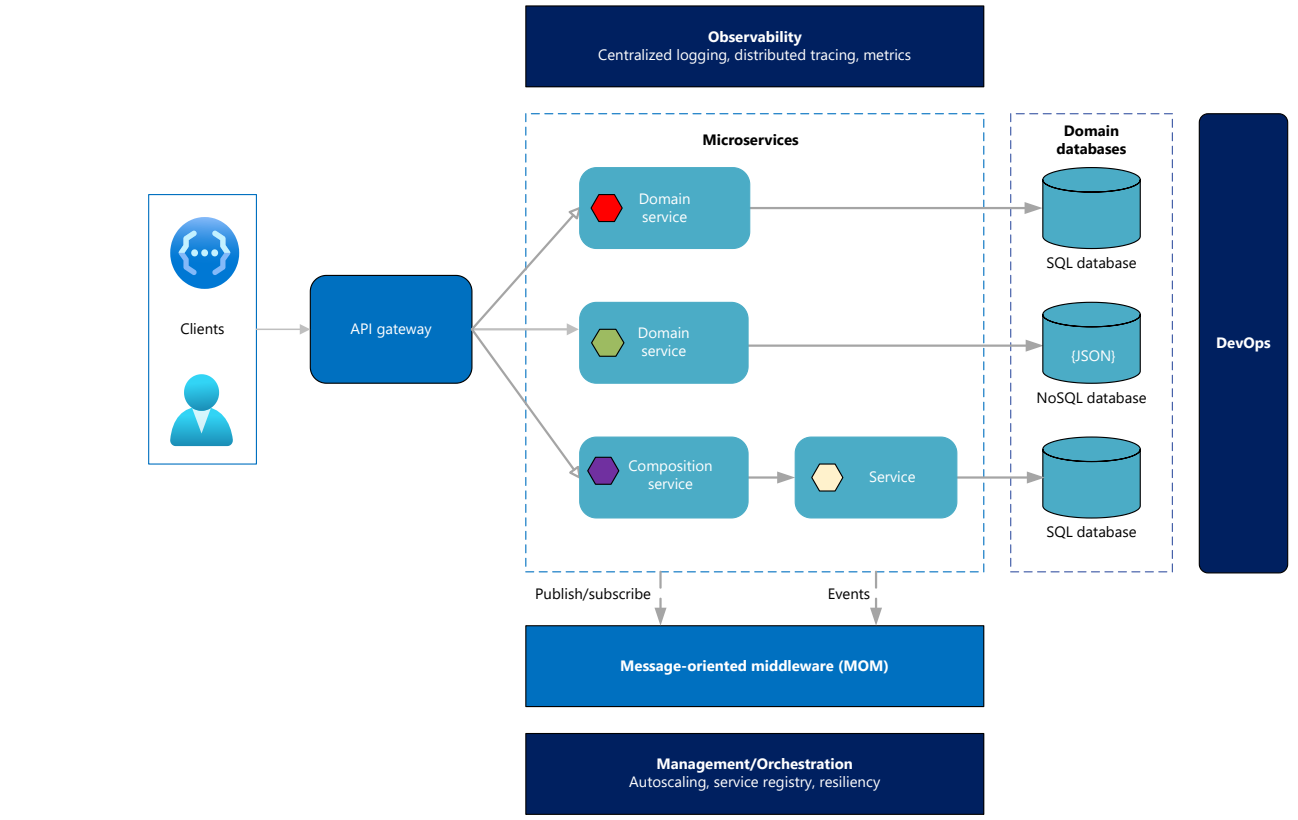
Web-Queue-Worker



For a purely PaaS solution, consider a Web-Queue-Worker architecture. In this style, the application has a web front end that handles HTTP requests and a back-end worker that performs CPU-intensive tasks or long-running operations. The front end communicates to the worker through an asynchronous message queue.

Web-Queue-Worker is suitable for relatively simple domains with some resource-intensive tasks. Like N-tier, the architecture is easy to understand. Managed services simplify deployment and operations. But with complex domains, it can be hard to manage dependencies. The front end and the worker can easily become large, monolithic components that are hard to maintain and update. As with N-tier, Web-Queue-Worker can reduce the frequency of updates and limit innovation.

Microservices

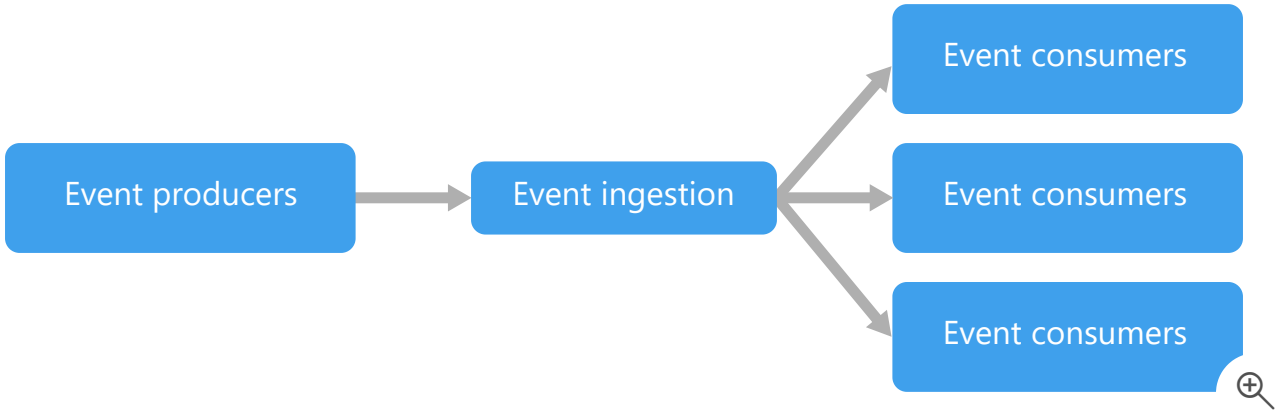


Microsoft Azure

If your application has a more complex domain, consider moving to a **Microservices** architecture. A microservices application is composed of many small, independent services. Each service implements a single business capability. Services are loosely coupled, communicating through API contracts.

Each service can be developed by a small, focused team. Individual services can be deployed with minimal coordination across teams, which supports frequent updates. Compared to N-tier or Web-Queue-Worker architectures, a microservice architecture is more complex to build and operate. It requires a mature development and DevOps culture. However, with the right practices in place, this approach can result in higher release velocity, faster innovation, and a more resilient architecture.

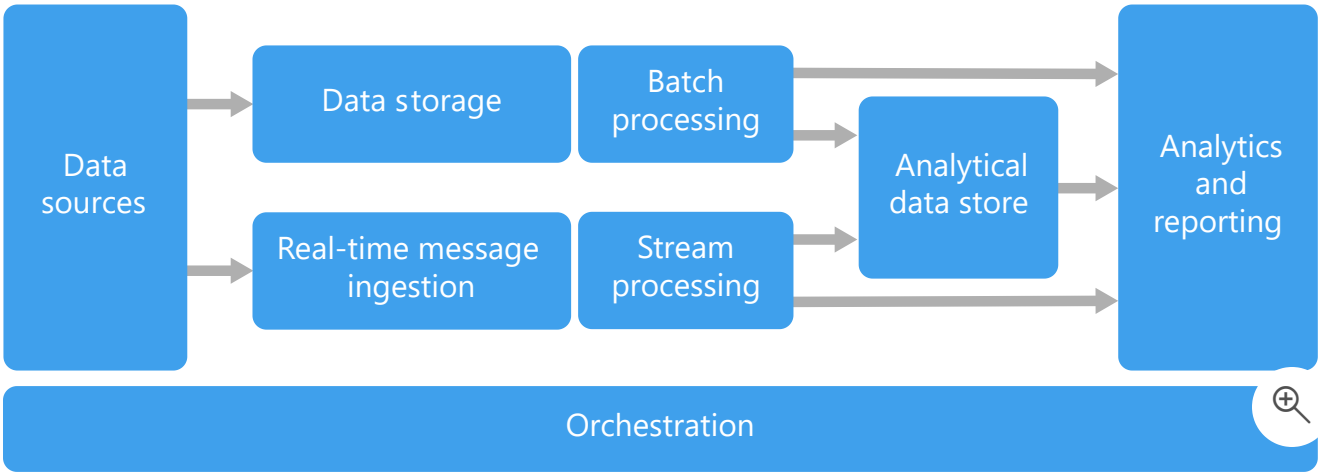
Event-driven architecture



Event-Driven Architectures use a publish-subscribe (pub-sub) model, where producers publish events, and consumers subscribe to them. The producers are independent from the consumers, and consumers are independent from each other.

Consider an event-driven architecture for applications that ingest and process a large volume of data with low latency, such as Internet of Things (IoT) solutions. The style is also useful when different subsystems must perform different types of processing on the same event data.

Big data, big compute



Big data and **big compute** are specialized architecture styles for workloads that match specific profiles. Big data splits a large dataset into chunks and performs parallel processing across the entire set for analysis and reporting. Big compute, also known as *high-performance computing*, performs parallel computations across thousands of cores. Common domains include simulations, modeling, and 3D rendering.

Architecture styles as constraints

An architecture style places constraints on the design, including the set of elements that can appear and the allowed relationships between those elements. Constraints guide the "shape" of an architecture by restricting the universe of choices. When an architecture conforms to the constraints of a particular style, certain desirable properties emerge.

For example, the constraints in microservices include:

- A service represents a single responsibility.
- Every service is independent of the others.
- Data is private to the service that owns it. Services don't share data.

By adhering to these constraints, what emerges is a system where services can be deployed independently, faults are isolated, frequent updates are possible, and it's easy to introduce new technologies into the application.

Each architecture style has its own trade-offs. Before you choose an architectural style, it's essential to understand the underlying principles and constraints. Without that understanding, you risk creating a design that superficially conforms to the style without realizing its full benefits. Focus more on why you're selecting a specific style than on how to implement it. Be practical. Sometimes it's better to relax a constraint than to chase architectural purity.

Ideally, the choice of architectural style should be made with input from informed workload stakeholders. The workload team should start by identifying the nature of the problem that they're solving. They should then define the key business drivers and the corresponding architecture characteristics, also known as *nonfunctional requirements*, and prioritize them. For example, if time to market is critical, the team might prioritize maintainability, testability, and reliability to enable rapid deployment. If the team has tight budget constraints, feasibility and simplicity might take precedence. Selecting and sustaining an architectural style isn't a one-time task. It requires ongoing measurement, validation, and refinement. Because changing architectural direction later can be costly, it's often worthwhile to invest more effort upfront to support long-term efficiency and reduce risks.

The following table summarizes how each style manages dependencies, and the types of domain that are best suited for each.

Expand table

Architecture style	Dependency management	Domain type
N-tier	Horizontal tiers divided by subnet	Traditional business domain. Frequency of updates is low.
Web-Queue-Worker	Front-end and back-end jobs, decoupled by asynchronous messaging.	Relatively simple domain with some resource intensive tasks.
Microservices	Vertically (functionally) decomposed services that call each other through APIs.	Complicated domain. Frequent updates.
Event-driven architecture	Producer or consumer. Independent view for each subsystem.	IoT and real-time systems.
Big data	Divide a huge dataset into small chunks. Parallel processing on local datasets.	Batch and real-time data analysis. Predictive analysis using ML.
Big compute	Data allocation to thousands of cores.	Compute intensive domains such as simulation.

Consider challenges and benefits

Constraints also create challenges, so it's important to understand the trade-offs when adopting any of these styles. Do the benefits of the architecture style outweigh the challenges, *for this subdomain and bounded context*.

Here are some of the types of challenges to consider when selecting an architecture style:

- **Complexity.** The architecture's complexity must match the domain. If it's too simplistic, it can result in a big ball of mud , where dependencies aren't well managed and the structure breaks down.
- **Asynchronous messaging and eventual consistency.** Asynchronous messaging is used to decouple services and improve reliability because messages can be retried. It also enhances scalability. However, asynchronous messaging also creates challenges in handling eventual consistency and the possibility of duplicate messages.
- **Interservice communication.** Decomposing an application into separate services might increase communication overhead. In microservices architectures, this overhead often results in latency problems or network congestion.
- **Manageability.** Managing the application includes tasks such as monitoring, deploying updates, and maintaining operational health.