Architecture styles

05/14/2019 • 5 minutes to read • Contributors 🧑 🧑 🧑 🧑 👤 all

In this article

An *architecture style* is a family of architectures that share certain characteristics. For example, N-tier is a common architecture style. More recently, microservice architectures have started to gain favor. Architecture styles don't require the use of particular technologies, but some technologies are well-suited for certain architectures. For example, containers are a natural fit 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. 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 a natural fit 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 application that use a mix 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. The use of managed services simplifies deployment and operations. But with a 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, this can reduce the frequency of updates and limit innovation.

Microservices

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 built by a small, focused development team. Individual services can be deployed without a lot of coordination between teams, which encourages frequent updates. A microservice architecture is more complex to build and manage than either N-tier or web-queue-worker. It requires a mature development and DevOps culture. But done right, this style can lead to 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 very low latency, such as 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 fit certain specific profiles. Big data divides a very large dataset into chunks, performing paralleling processing across the entire set, for analysis and reporting. Big compute, also called high-performance computing (HPC), makes parallel computations across a large number (thousands) of cores. Domains include simulations, modeling, and 3-D 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 do not 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.

Before choosing an architecture style, make sure that you understand the underlying principles and constraints of that style. Otherwise, you can end up with a design that conforms to the style at a superficial level, but does not achieve the full potential of that style. It's also important to be pragmatic. Sometimes it's better to relax a constraint, rather than insist on architectural purity.

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

Architecture style	Dependency management	Domain type
N-tier	Horizontal tiers divided by subnet	Traditional business domain. Frequency of updates is low.

Architecture style	Dependency management	Domain type
Web-Queue-Worker	Front and backend jobs, decoupled by async 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/consumer. Independent view per sub-system.	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**. Is the complexity of the architecture justified for your domain? Conversely, is the style too simplistic for your domain? In that case, you risk ending up with a "[big ball of mud](#)", because the architecture does not help you to manage dependencies cleanly.

- **Asynchronous messaging and eventual consistency**. Asynchronous messaging can be used to decouple services, and increase reliability (because messages can be retried) and scalability. However, this also creates challenges such as always-once semantics and eventual consistency.

- **Inter-service communication**. As you decompose an application into separate services, there is a risk that communication between services will cause unacceptable latency or create network congestion (for example, in a microservices architecture).

- **Manageability**. How hard is it to manage the application, monitor, deploy updates, and so on?