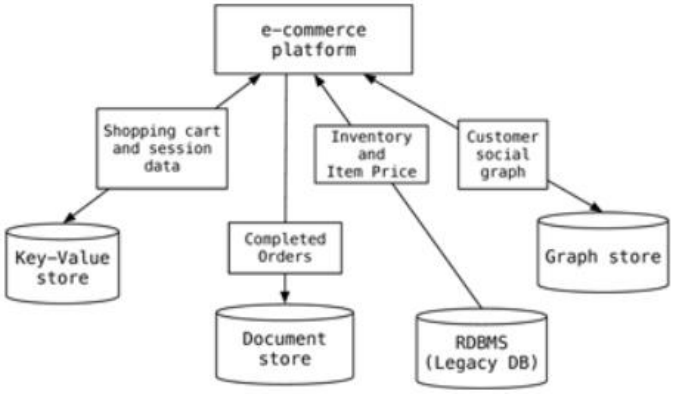
Traditional on-premises	Modern cloud
Monolithic	Decomposed / Distributed
Designed for predictable scalability	Designed for elastic scale
Relational database	Polyglot persistence* (multiple data storage technologies)
Synchronized processing	Asynchronous processing
Occasional large updates	Frequent small updates
Manual management	Automated self-management using DevOps
Snowflake servers**	Immutable infrastructure

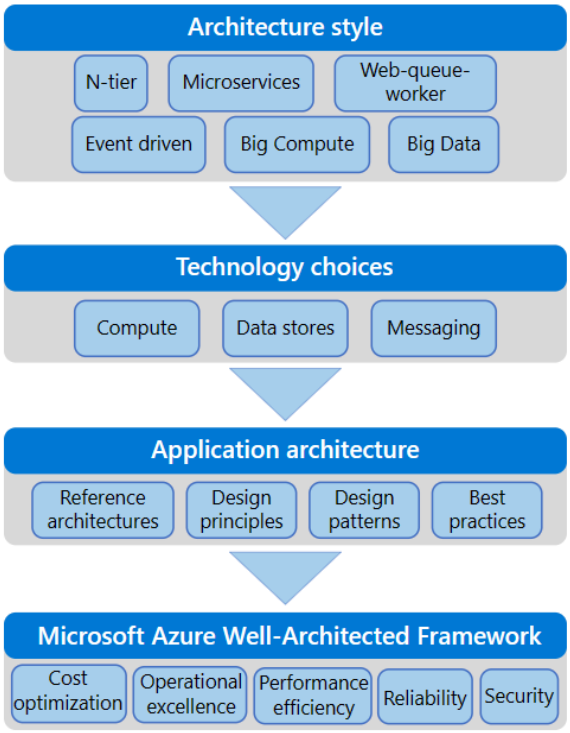
*Polyglot Persistence:



**Snowflake servers are the server whose actual configuration has drifted far more than what was actually required.

Azure Application Architecture

The Azure application architecture guidance is organized as a series of steps, from the architecture and design to implementation. For each step, there is supporting guidance that will help you design your application architecture.



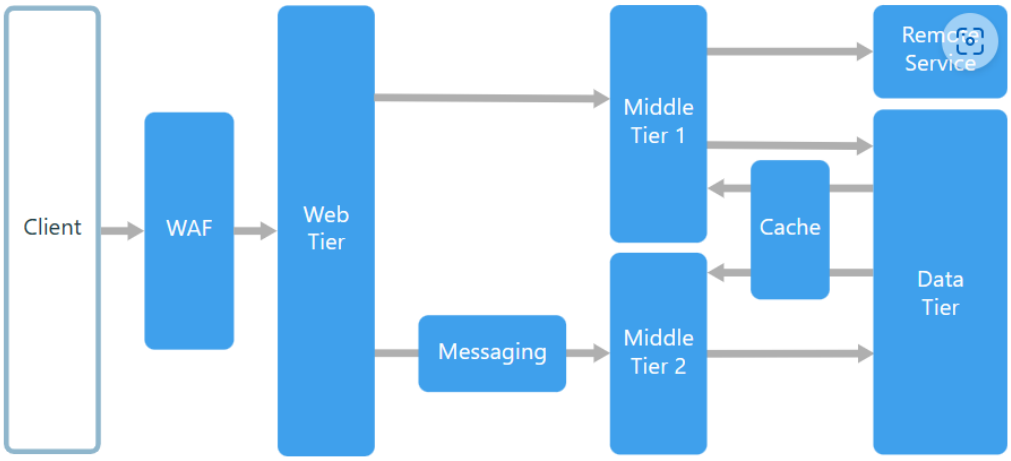
Architectural Styles

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.

Architecture styles that are commonly found in cloud applications.

1. N-tier
2. Microservices
3. Web-queue Worker
4. Event Driven
5. Big Compute
6. Big Data

N-tier Architectural Style



An N-tier architecture divides an application into **logical layers** and **physical tiers**.

Dependencies are managed by dividing the application into *layers* that perform logical functions, such as presentation, business logic, and data access.

An N-tier application can have a **closed layer architecture** or an **open layer architecture**:

- In a closed layer architecture, a layer can only call the next layer immediately down.
- In an open layer architecture, a layer can call any of the layers below it.

When to use this architecture:

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. Often, it's advantageous to use managed services for some parts of the architecture, particularly caching, messaging, and data storage.

Consider an N-tier architecture for:

- Simple web applications.
- Migrating an on-premises application to Azure with minimal refactoring. (Lift and Shift)
- Unified development of on-premises and cloud applications.

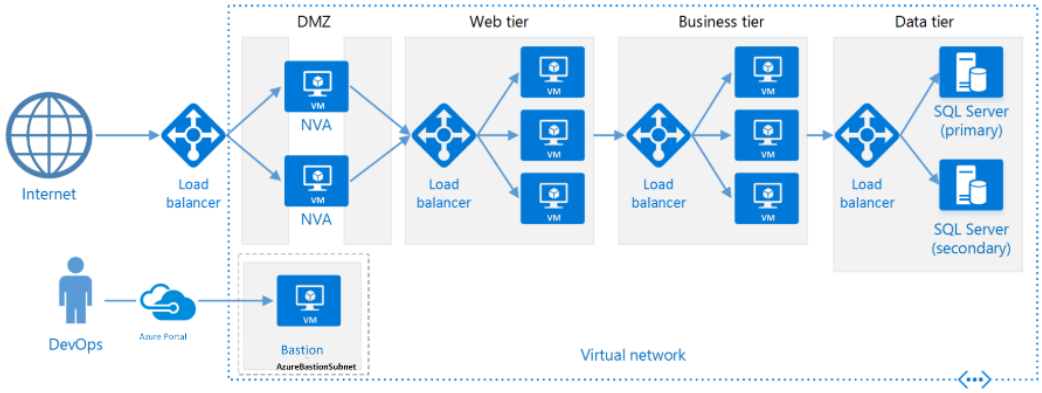
Challenges:

- It's easy to end up with a middle tier that just does CRUD operations on the database, adding extra latency without doing any useful work.
- Monolithic design prevents independent deployment of features.
- Managing an IaaS application is more work than an application that uses only managed services.
- It can be difficult to manage network security in a large system.

Downside and Challenges:

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 architecture on virtual machines



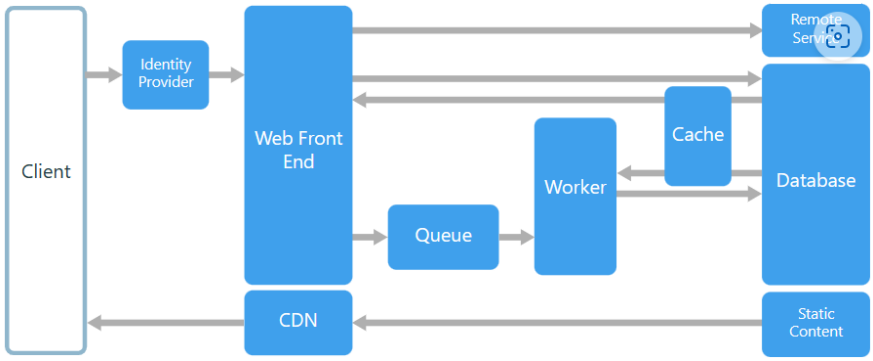
Each tier consists of **two or more VMs**, placed in an availability set or **virtual machine scale set**. Load balancers are used to distribute requests across the VMs in a tier. A tier can be scaled horizontally by adding more VMs to the pool.

Each tier is also placed inside its **own subnet**, meaning their internal IP addresses fall within the same address range. That makes it easy to apply **network security group** rules and route tables to individual tiers.

The web and business tiers are **stateless**. Any VM can handle any request for that tier. The data tier should consist of a replicated database.

Network security groups **restrict access** to each tier. For example, the database tier only allows access from the business tier.

Web-Queue-Worker Architectural Style



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 web front end communicates with the worker through a **message queue**. The web and worker are both **stateless**. Session state can be stored in a **distributed cache**.

Based on purely PaaS solution.

Other components that are commonly incorporated into this architecture include:

- One or more databases.
- A cache to store values from the database for quick reads.
- CDN to serve static content
- Remote services, such as email or SMS service. Often these features are provided by third parties.
- Identity provider for authentication.

When to use this architecture:

It is suitable for relatively simple domains with some **resource-intensive** tasks.

The web and worker are both stateless. Session state can be stored in a distributed cache. The use of managed services simplifies deployment and operations.

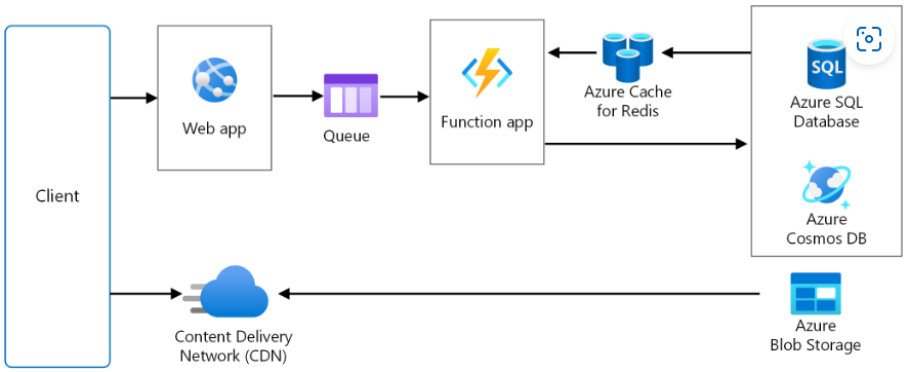
Benefits

- Relatively simple architecture that is easy to understand.
- Easy to deploy and manage.
- Clear separation of concerns.
- The front end is decoupled from the worker using asynchronous messaging.
- The front end and the worker can be scaled independently.

Downside and Challenges

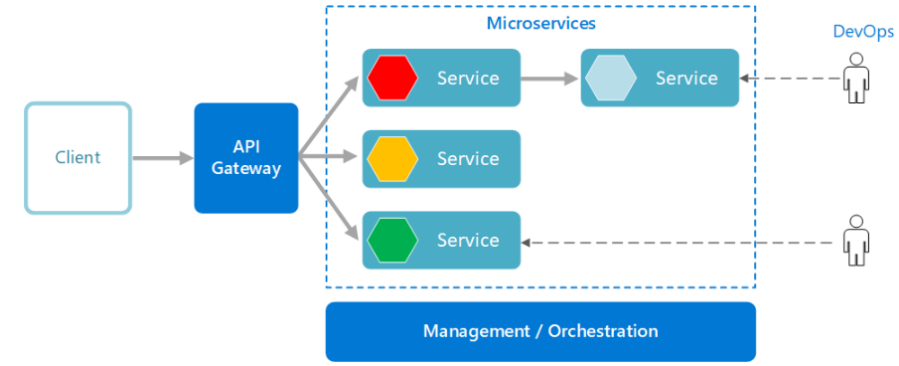
- With complex domains, it can be hard to manage dependencies. The front end and the worker can easily become large, monolithic components that are hard to maintain and update. As with N-tier, this can reduce the frequency of updates and limit innovation.
- There may be hidden dependencies, if the front end and worker share data schemas or code modules.

Web-Queue-Worker on Azure App Service



- The front end is implemented as an Azure App Service web app, and the worker is implemented as an Azure Functions app. The web app and the function app are both associated with an App Service plan that provides the VM instances.
- You can use either Azure Service Bus or Azure Storage queues for the message queue. (The diagram shows an Azure Storage queue.)
- Azure Cache for Redis stores session state and other data that needs low latency access.
- Azure CDN is used to cache static content such as images, CSS, or HTML.
- For storage, choose the storage technologies that best fit the needs of the application. You might use multiple storage technologies (polyglot persistence). To illustrate this idea, the diagram shows Azure SQL Database and Azure Cosmos DB.

Microservices



Microservices application is composed of many small, independent services. Each service implements a single business capability. Services are loosely coupled, communicating through API contracts.

When to use this Architecture:

Best suited for Complex Domain. Individual services can be deployed without a lot of coordination between teams, which encourages frequent updates.

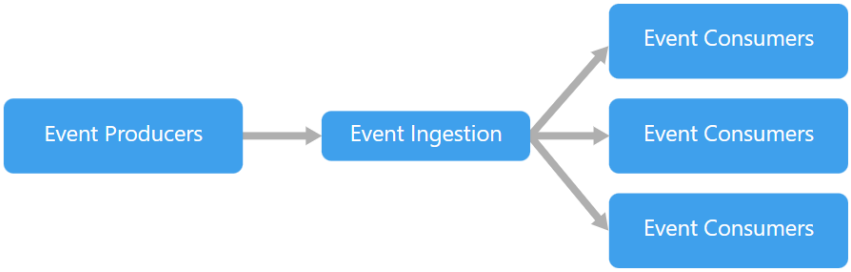
Benefits:

- Agility
- Small, focused teams
- Small code base
- Mix of technologies
- Fault isolation
- Scalability
- Data Isolation

Downside and Challenges:

- 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.
- Testing with dependencies specially asynchronous communicating microservices.
- Network congestion and latency
- Data integrity
- Versioning

Event-driven architecture



It uses 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.

Events are delivered in **near real time**, so consumers can respond immediately to events as they occur. Producers are decoupled from consumers — a producer doesn't know which consumers are listening. Consumers are also decoupled from each other, and every consumer sees all of the events.

An event driven architecture can use a publish/subscribe (also called *pub/sub*) model or an event stream model.

- **Pub/sub**: The messaging infrastructure keeps track of subscriptions. When an event is published, it sends the event to each subscriber. After an event is received, it cannot be replayed, and new subscribers do not see the event.
- **Event streaming**: Events are written to a log. Events are strictly ordered (within a partition) and durable. Clients don't subscribe to the stream, instead a client can read from any part of the stream. The client is responsible for advancing its position in the stream. That means a client can join at any time, and can replay events.

When to choose: Consider an event-driven architecture for applications that ingest and process a large volume of data with very low latency, such as IoT solutions.

- Multiple subsystems must process the same events.
- Real-time processing with minimum time lag.
- Complex event processing, such as pattern matching or aggregation over time windows.

- High volume and high velocity of data, such as IoT.

Benefits

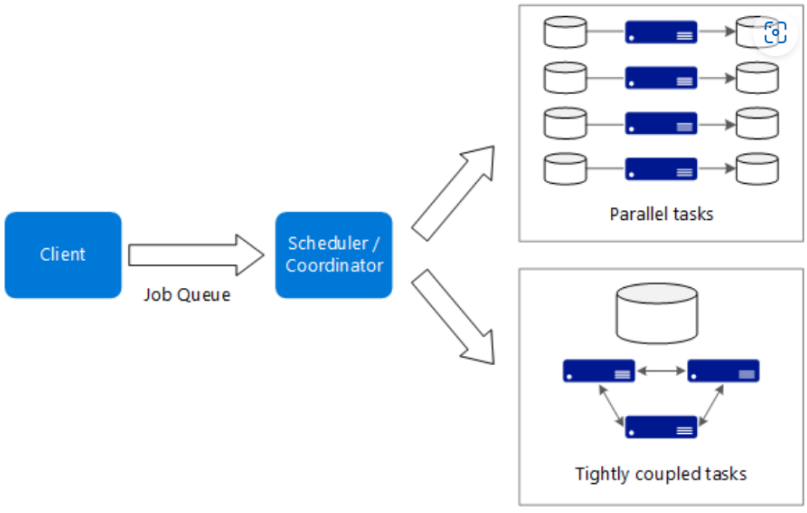
- Producers and consumers are decoupled.
- No point-to-point integrations. It's easy to add new consumers to the system.
- Consumers can respond to events immediately as they arrive.
- Highly scalable and distributed.
- Subsystems have independent views of the event stream.

Challenges:

- Guaranteed delivery. In some systems, especially in IoT scenarios, it's crucial to guarantee that events are delivered.
- Processing events in order or exactly once. Each consumer type typically runs in multiple instances, for resiliency and scalability. This can create a challenge if the events must be processed in order (within a consumer type), or if the processing logic is not idempotent.

Big Compute Architecture

The term big compute describes large-scale workloads that require a large number of cores, often numbering in the hundreds or thousands. Scenarios include image rendering, fluid dynamics, financial risk modeling, oil exploration, drug design, and engineering stress analysis, among others.



Typical characteristics of big compute applications:

- The work can be split into discrete tasks, which can be run across many cores simultaneously.
- Each task is finite. It takes some input, does some processing, and produces output. The entire application runs for a finite amount of time (minutes to days).

- The application does not need to stay up 24/7. However, the system must handle node failures or application crashes.
- For some applications, tasks are independent and can run in parallel. In other cases, tasks are tightly coupled, meaning they must interact or exchange intermediate results.
- Depending on your workload, you might use compute-intensive VM sizes (H16r, H16mr, and A9) or Azure Batch Service

When to use this architecture

- Computationally intensive operations such as simulation and number crunching.
- Simulations that are computationally intensive and must be split across CPUs in multiple computers (10-1000s).
- Simulations that require too much memory for one computer, and must be split across multiple computers.
- Long-running computations that would take too long to complete on a single computer.
- Smaller computations that must be run 100s or 1000s of times, such as Monte Carlo simulations.

Benefits

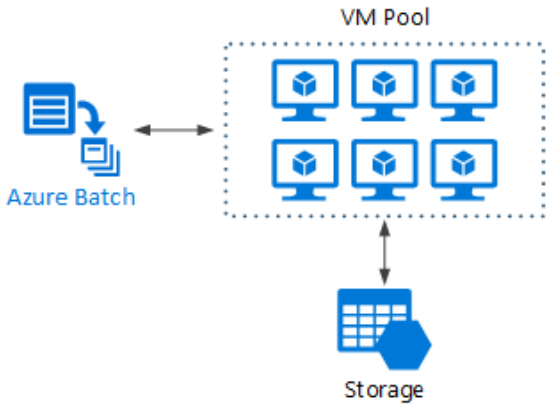
- High performance with "embarrassingly parallel" processing.
- Can harness hundreds or thousands of computer cores to solve large problems faster.
- Access to specialized high-performance hardware, with dedicated high-speed InfiniBand networks.
- You can provision VMs as needed to do work, and then tear them down.

Challenges

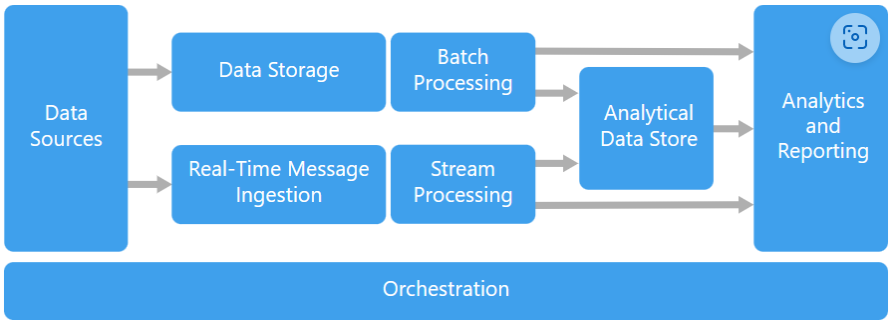
- Managing the VM infrastructure.
- Managing the volume of number crunching
- Provisioning thousands of cores in a timely manner.
- For tightly coupled tasks, adding more cores can have diminishing returns. You may need to experiment to find the optimum number of cores.

Big compute using Azure Batch:

Azure Batch is a managed service for running large-scale high-performance computing (HPC) applications.



Big Data Architectural Style



Big data solutions typically involve one or more of the following types of workload:

- Batch processing of big data sources at rest.
- Real-time processing of big data in motion.
- Interactive exploration of big data.
- Predictive analytics and machine learning.

When to use this architecture

- Store and process data in volumes too large for a traditional database.
- Transform unstructured data for analysis and reporting.
- Capture, process, and analyze unbounded streams of data in real time, or with low latency.
- Use Azure Machine Learning or Microsoft Cognitive Services.

Benefits

- **Technology choices**. You can mix and match Azure managed services and Apache technologies in HDInsight clusters, to capitalize on existing skills or technology investments.
- **Performance through parallelism**. Big data solutions take advantage of parallelism, enabling high-performance solutions that scale to large volumes of data.

- **Elastic scale**. All of the components in the big data architecture support scale-out provisioning, so that you can adjust your solution to small or large workloads, and pay only for the resources that you use.
- **Interoperability with existing solutions**. The components of the big data architecture are also used for IoT processing and enterprise BI solutions, enabling you to create an integrated solution across data workloads

Challenges:

- Complexity
- Skillset
- Technology maturity
- Security

Technology Choices

1. Compute
2. Data stores
3. Messaging

Application Architecture

1. References Architectures
2. Design Principles
3. Design Patterns
4. Best Practices

Microsoft Azure Well Architected Framework

1. Cost Optimization
2. Operational excellence
3. Performance efficiency
4. Reliability
5. Security