Cloud Messaging Services

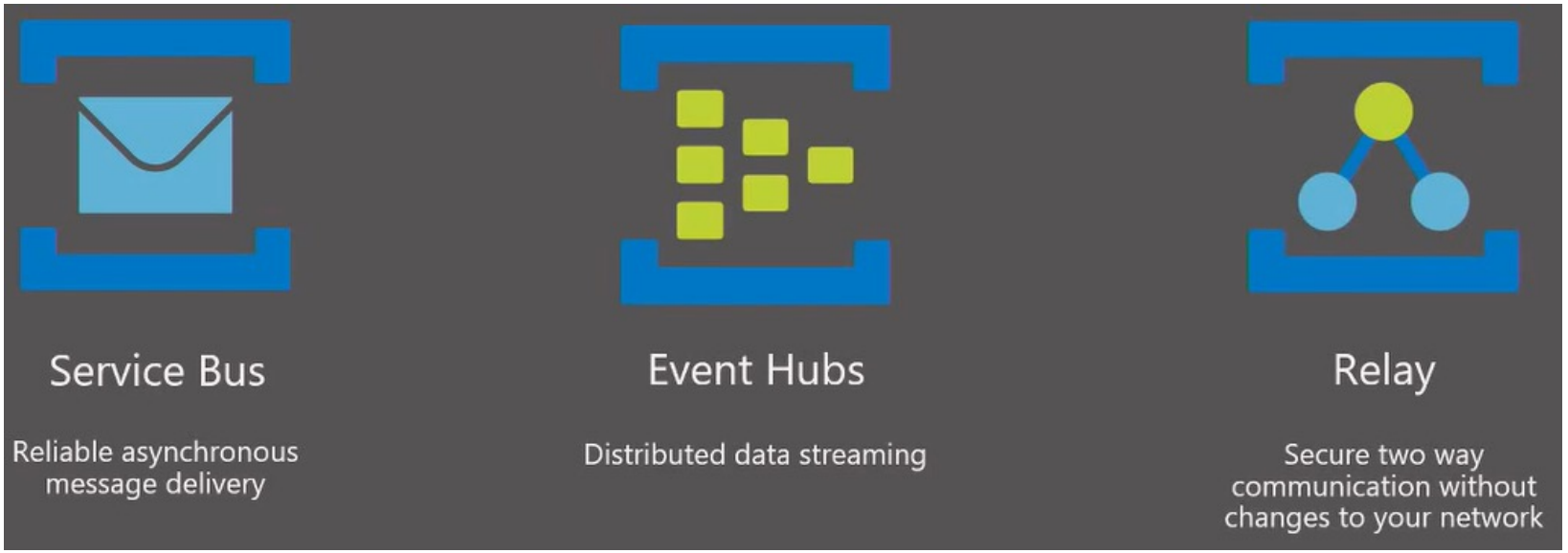
Messaging Services

- Used for machine – machine and/or service – service communication
- Provides decoupling between producers of messages and consumers of messages
- Data streaming, ingestion and retention scenarios
- Connectivity between on-premise and cloud scenarios
- Key glue that enables loosely coupled and distributed cloud based solutions

Where messaging services are used?

Pricing and Quote Updates

Azure Messaging Services



Azure Service Bus

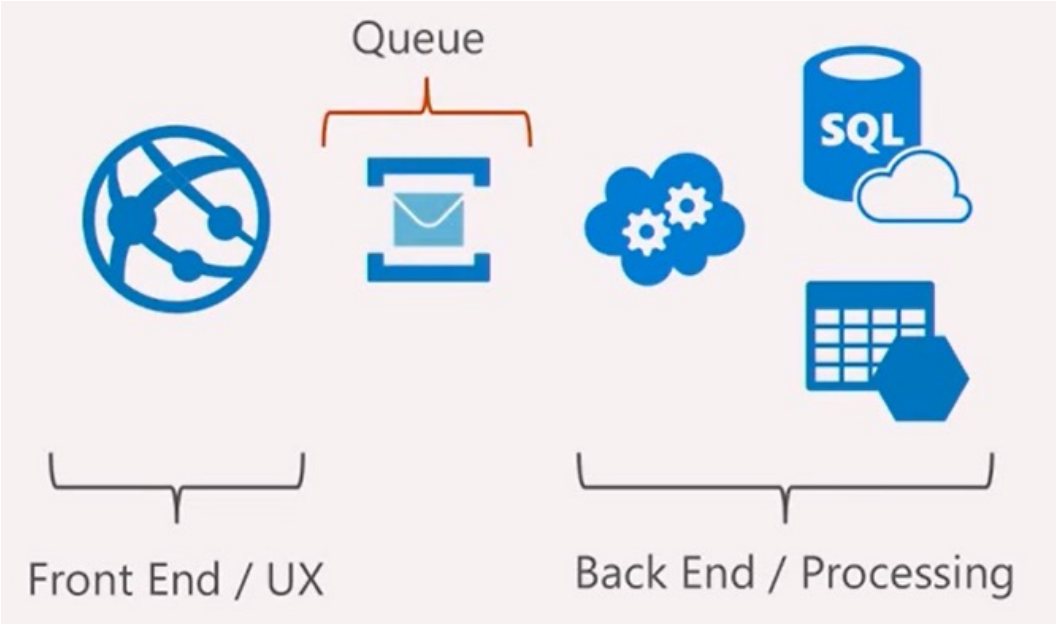
- Messaging as a service

Queues and Topics

- Reliable asynchronous communication
- Rich features for temporal control
- Routing and filtering
- Convoys & Sessions (related message with state)

Where does Service bus fit in?

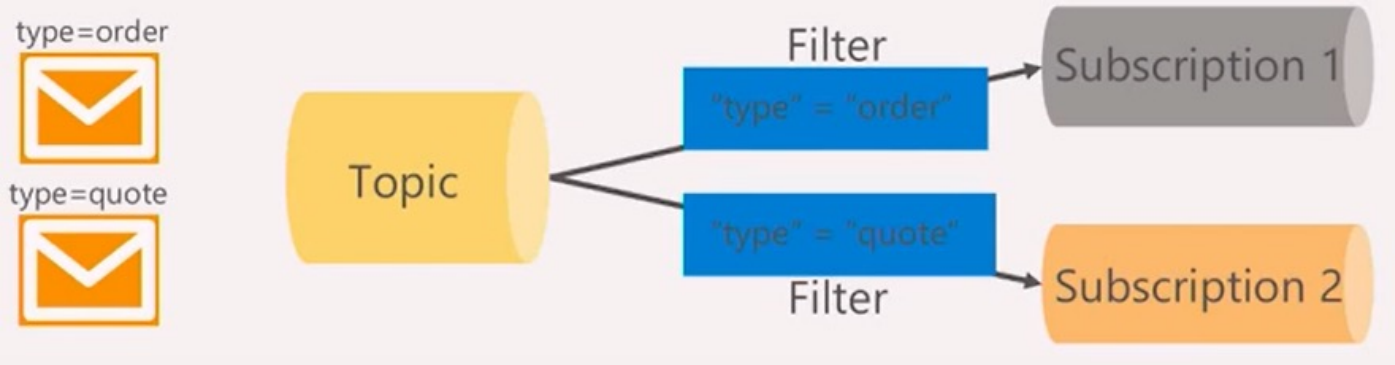
- Decoupling and providing durability are primary drivers for using service bus



Features of Service Bus

- Scheduled delivery
- Poison message handling
- ForwardTo
- Defer
- Sessions
- Batching
- Ordering
- Auto-delete on idle
- OnMessage
- Duplicate detection
- Filters
- Actions
- Transactions

Topics & Subscriptions



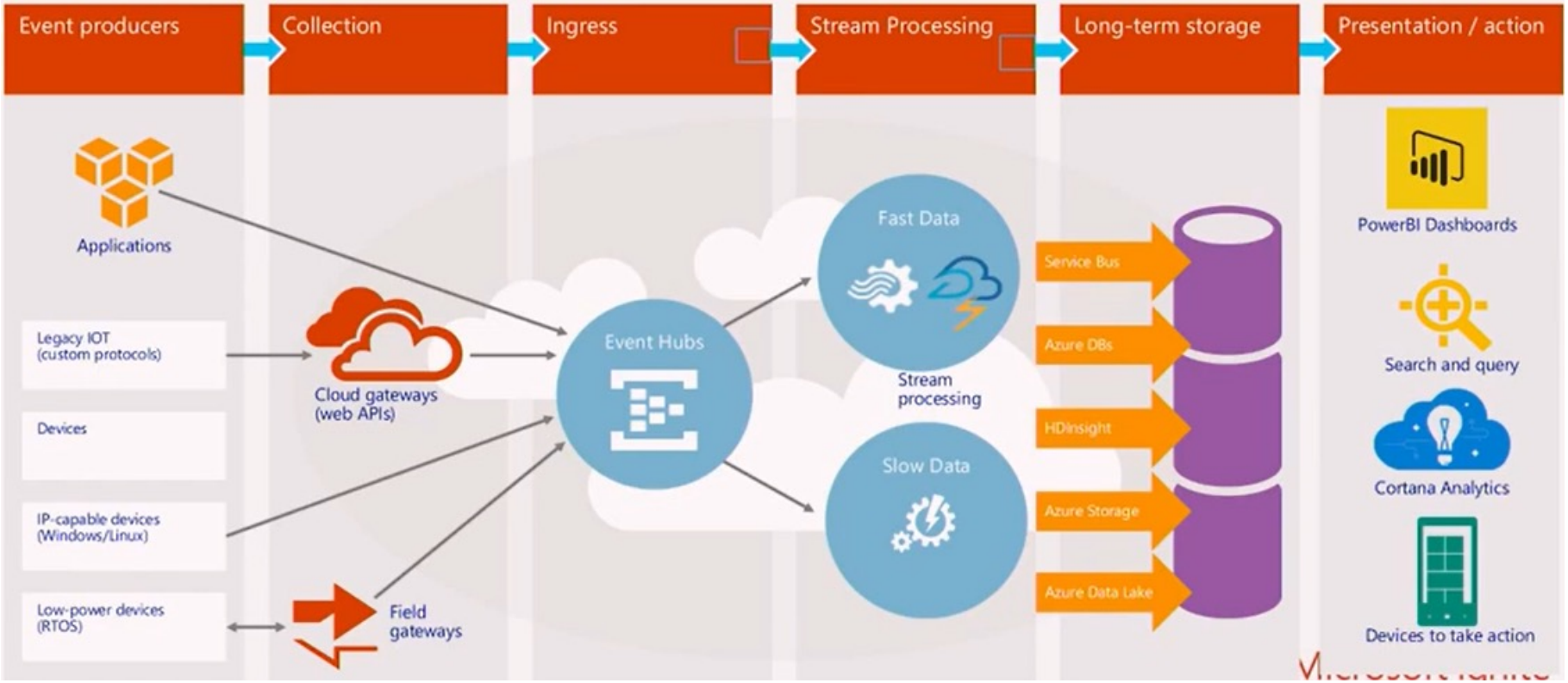
Sender knows only about Topic
Receivers know only about Subscription
Filters and Actions exist on Subscription

Azure Event Hubs

Distributed Data Streaming solution

Event Hubs

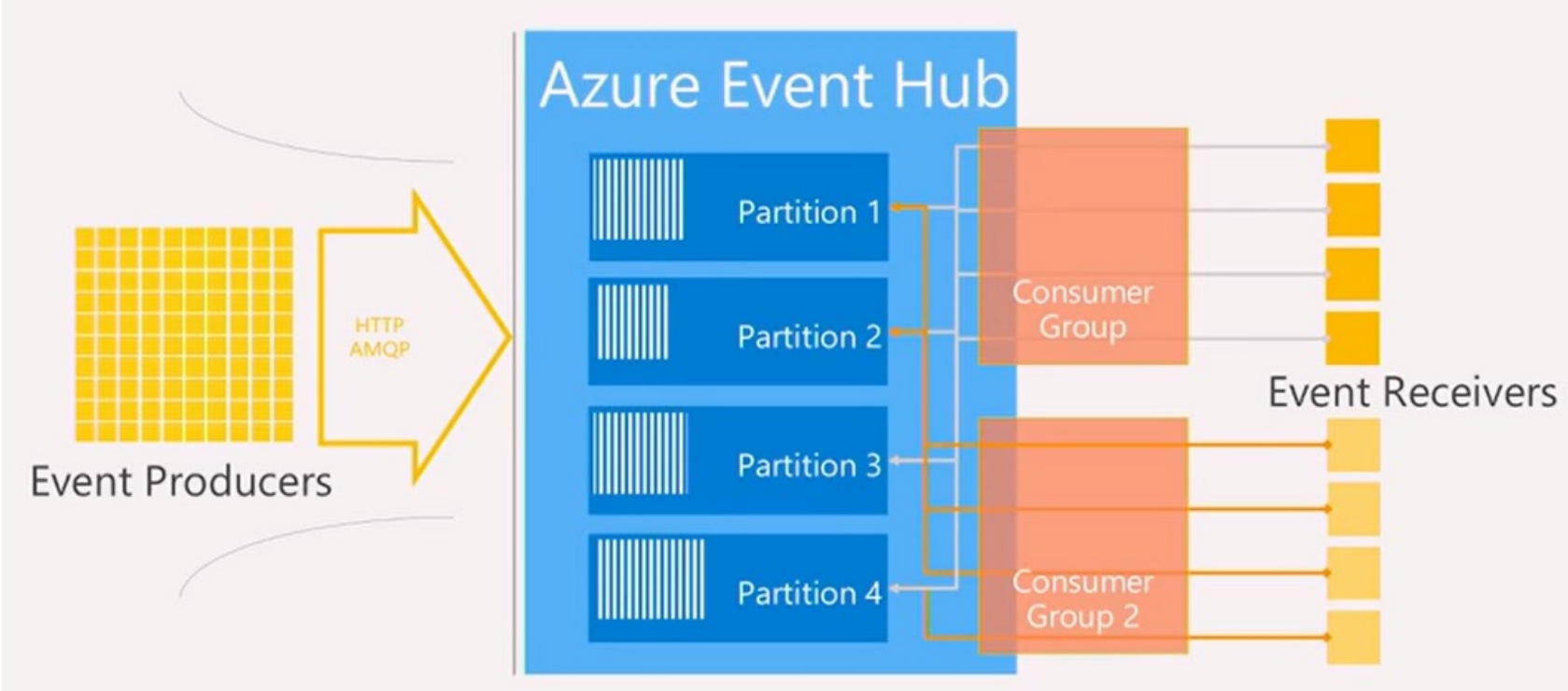
- A streaming service designed to do a low latency distributed stream ingress (i.e. going-in)
- A partitioned consumer scale model
- A time retention buffer
- An elastic component in the middle of your telemetry / IoT / big data chain


Event producers
Collection
Ingress
Stream Processing
Long-term storage
Presentation / action
Applications
Legacy IOT
(custom protocols)
Devices
IP-capable devices
(Windows/Linux)
Low-power devices
(RTOS)
Cloud gateways
(web APIs)
Field
gateways
Event Hubs
Fast Data
Stream
processing
Slow Data
Service Bus
Azure DBs
HDInsight
Azure Storage
Azure Data Lake
PowerBI Dashboards
Search and query
Cortana Analytics
Devices to take action

Event Hub features

- Archive
- Proximity (related data is grouped together)
- Order
- Consistent Playback
- Tremendous scale

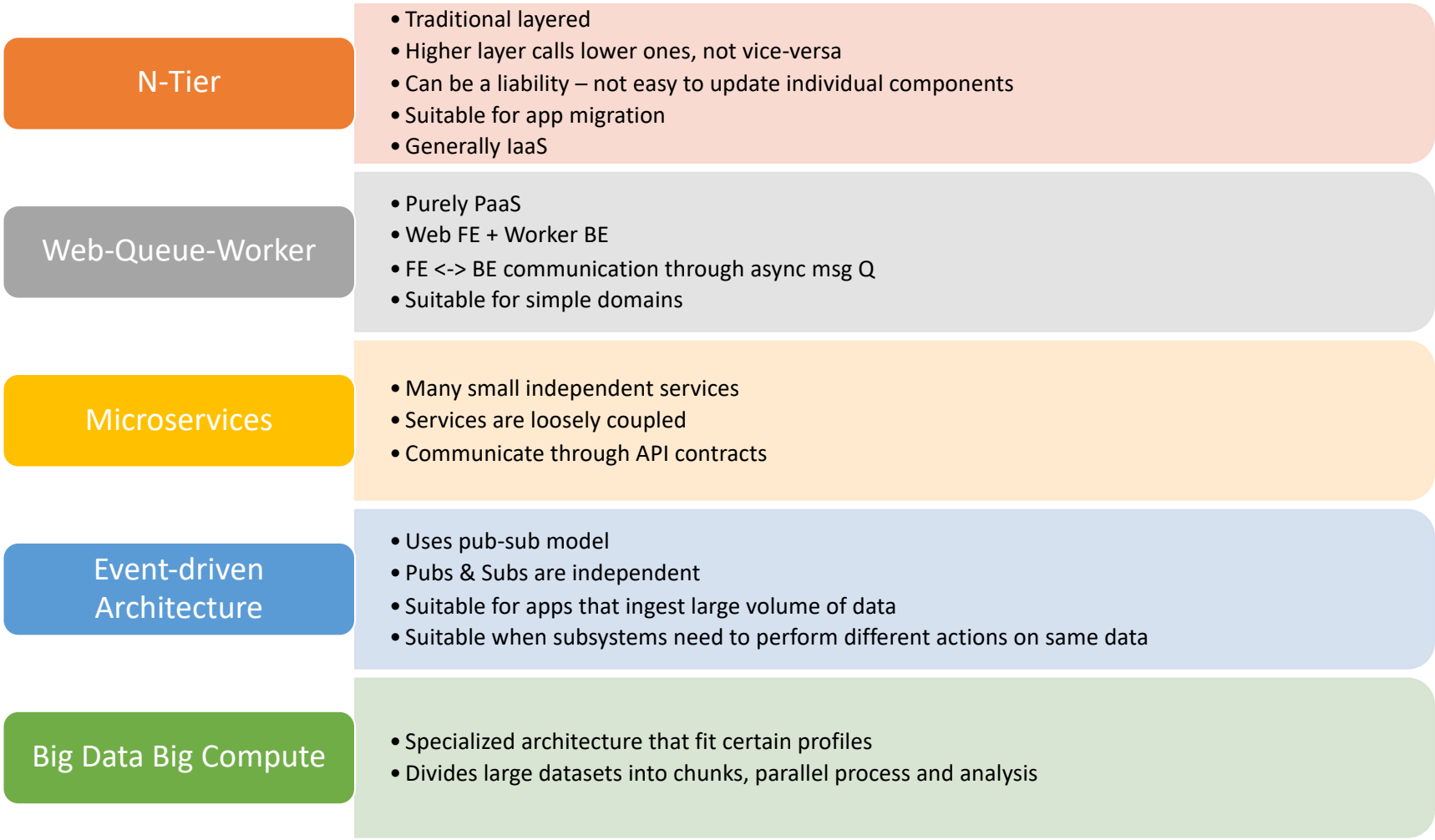
Event Hubs conceptual architecture



Relays & Hybrid Connections

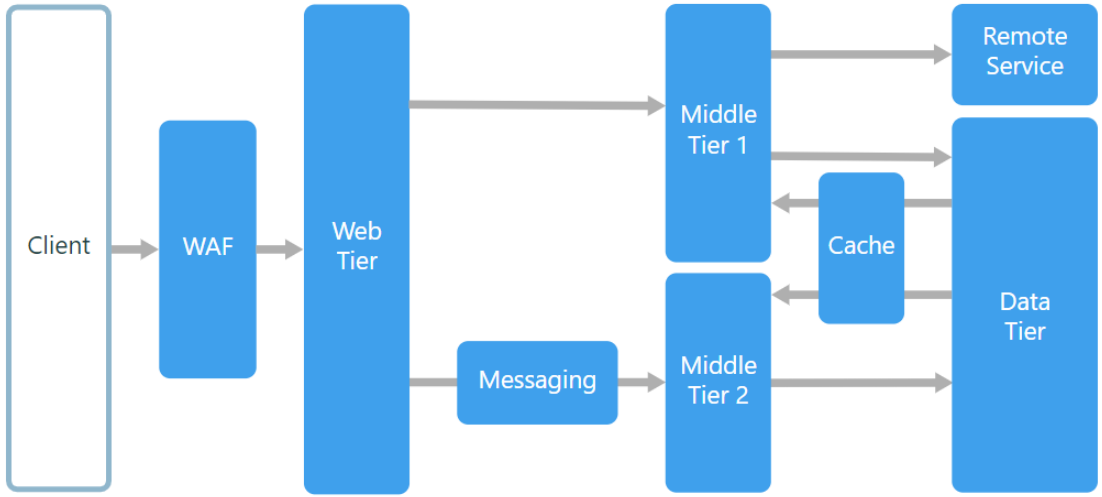
- Securely connect to on-premises data and services from anywhere
- Load balance multiple sites behind a single endpoint
- Protect internal services by projecting a cloud endpoint
- Does not require any network changes
- Live request-response and streaming services
- Hybrid cloud + on premises

Cloud Architectures


N-Tier
• Traditional layered
• Higher layer calls lower ones, not vice-versa
• Can be a liability – not easy to update individual components
• Suitable for app migration
• Generally IaaS
Web-Queue-Worker
• Purely PaaS
• Web FE + Worker BE
• FE <-> BE communication through async msg Q
• Suitable for simple domains
Microservices
• Many small independent services
• Services are loosely coupled
• Communicate through API contracts
Event-driven Architecture
• Uses pub-sub model
• Pubs & Subs are independent
• Suitable for apps that ingest large volume of data
• Suitable when subsystems need to perform different actions on same data
Big Data Big Compute
• Specialized architecture that fit certain profiles
• Divides large datasets into chunks, parallel process and analysis

N-tier Architecture Style

- An N-tier architecture divides an application into **logical layers** and **physical tiers**.
- Layers are a way to separate responsibilities and manage dependencies.
- Tiers are physically separated, running on separate machines.

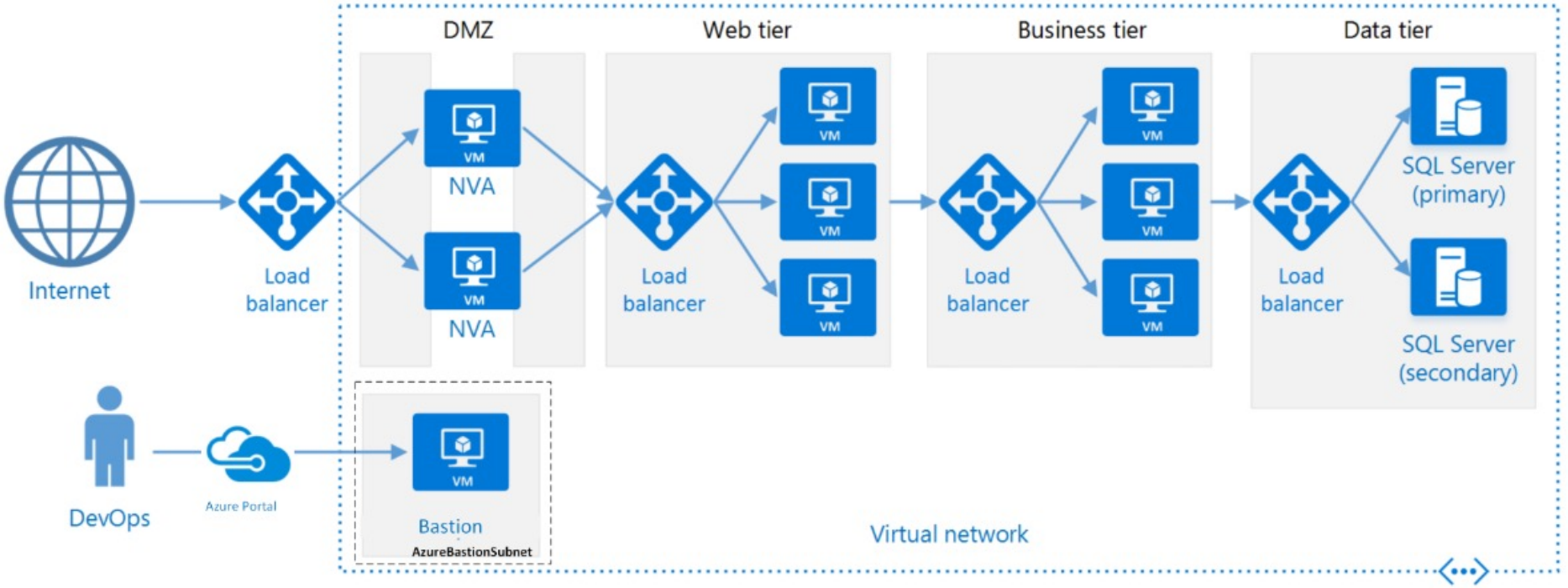


When to use this architecture

- Typically implemented as infrastructure-as-service (IaaS) applications, with each tier running on a separate set of VMs.
- Consider an N-tier architecture for:
  - Simple web applications.
  - Migrating an on-premises application to Azure with minimal refactoring.
  - Unified development of on-premises and cloud applications.

- **Best practices**
- Use autoscaling to handle changes in load. See Autoscaling best practices.
- Use asynchronous messaging to decouple tiers.
- Cache semi static data. See Caching best practices.
- Configure the database tier for high availability, using a solution such as SQL Server Always On availability groups.
- Place a web application firewall (WAF) between the front end and the Internet.
- Place each tier in its own subnet, and use subnets as a security boundary.
- Restrict access to the data tier, by allowing requests only from the middle tier(s).

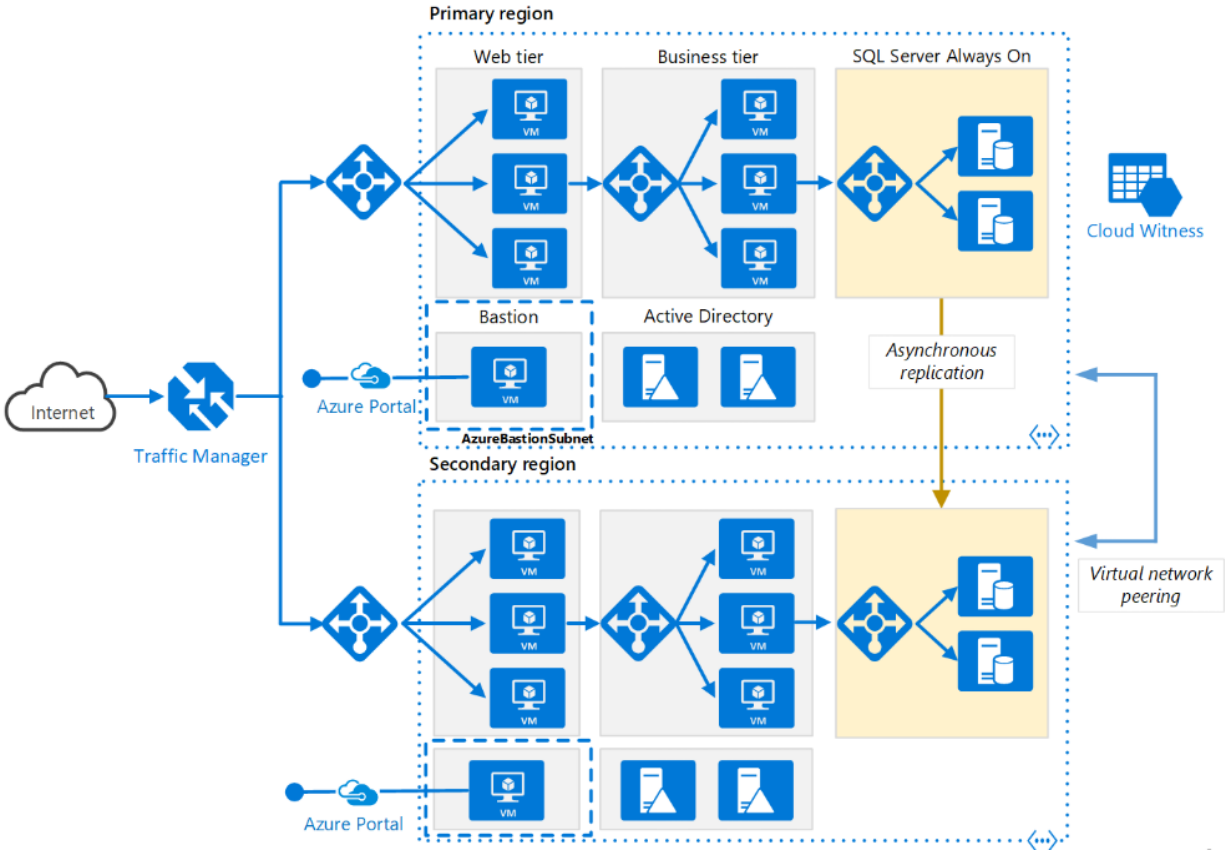
N-tier architecture on Virtual Machines



- Each tier consists of two or more VMs, placed in an availability set or virtual machine scale set.
- Multiple VMs provide resiliency in case one VM fails.
- 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.

Role of Application Gateway

- Web traffic load balancer enables us to manage traffic to our web applications
- Operates at application level (OSI level 7 - Application).
  - Traditional load balancers operate at transport layer (OSI level 4 – TCP & UDP) and route traffic based on IP address and port.
  - Application Gateway can route traffic based on URL
- SSL termination
  - supports terminate encryption at gateway, so data travels un-encrypted to the backend servers. Saves overhead.
- Web application firewall – protection against common exploits
- Multiple site hosting, Redirection, Session affinity etc.

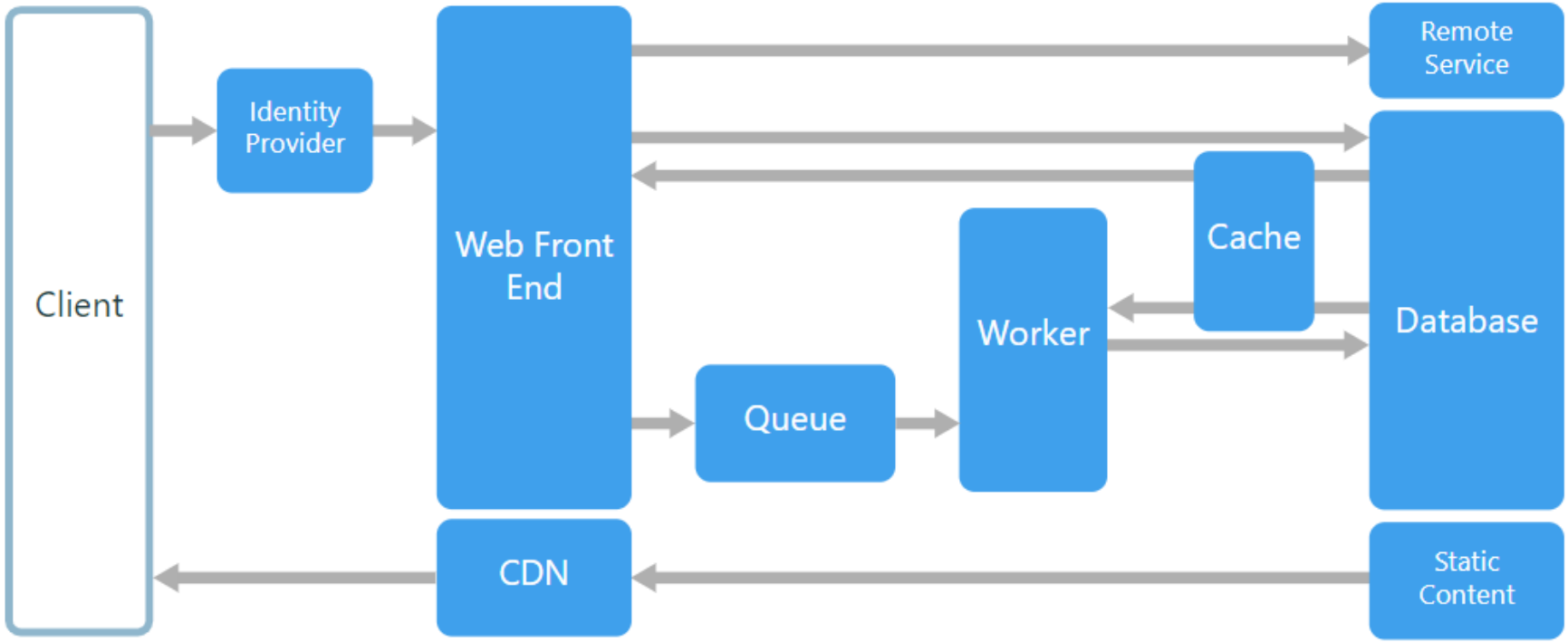


N-tier architecture – Multi-region deployment

Role of Traffic Manager in Multi-region deployment

- Traffic Manager allows users to control traffic distribution across various application endpoints.
- The Traffic Manager distributes the traffic using the configured routing method
  - Priority, Weighted, Performance, Geographic
- Constant monitoring of the endpoint health.
- Automatic failover in case the endpoints fail.
- Functions at the DNS level. It uses the DNS to route clients to precise service endpoints, according to the traffic-routing rules and methods.
- Traffic Manager operation modes:
  - Active | Passive – hot failover
  - Active | Passive – cold failover
  - Active | Active – load balanced

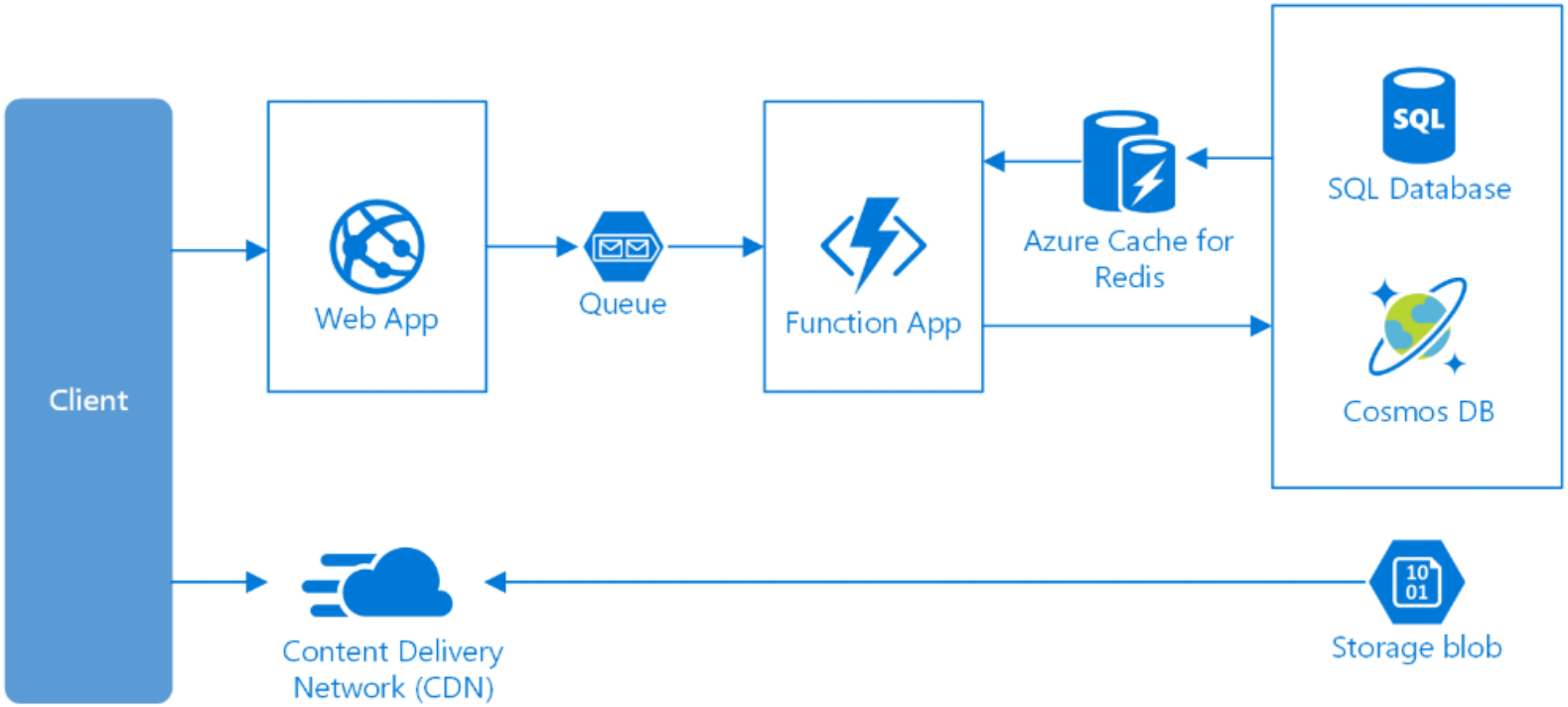
Web-Queue-Worker architecture



- The core components of this architecture are
  - a **web front end** that serves client requests,
  - a **worker** that performs resource-intensive tasks, long-running workflows, or batch jobs.
  - a **message queue -** the web front end communicates with the worker through it.
- 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 are provided by third parties.
  - Identity provider for authentication.

- The web and worker are both stateless.
- Session state can be stored in a distributed cache.
- Any long-running work is done asynchronously by the worker.
- The worker can be triggered by messages on the queue or run on a schedule for batch processing.
  - The worker is an optional component. If there are no long-running operations, the worker can be omitted.
- The front end might consist of a web API.
- On the client side, the web API can be consumed by a single-page application that makes AJAX calls, or by a native client application.

Web-Queue-Worker on Azure



- Consider this architecture style for:
  - Applications with a relatively simple domain.
  - Applications with some long-running workflows or batch operations.
  - When you want to use managed services, rather than infrastructure as a service (IaaS).

Microservices Architecture

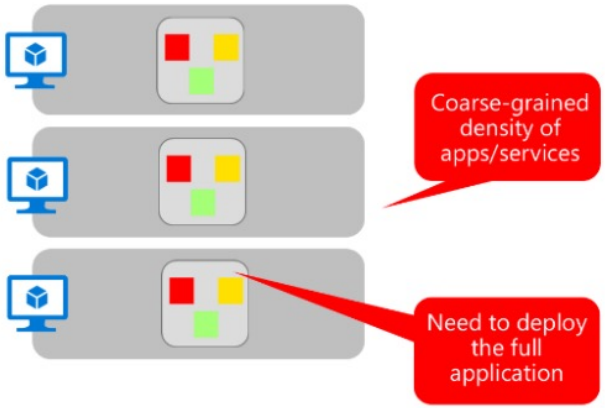
- a microservices architecture is an approach to building a server application as a set of small services
- Each service runs in its own process and communicates with other processes using protocols such as HTTP/HTTPS, WebSockets, or AMQP
- Each microservice implements a specific end-to-end domain or business capability within a certain context boundary, and each must be developed autonomously and be deployable independently
- each microservice should own its related domain data model and domain logic (sovereignty and decentralized data management) and could be based on different data storage technologies (SQL, NoSQL) and different programming languages

Why a microservices architecture?

- it provides long-term agility. Microservices enable better maintainability in complex, large, and highly-scalable systems by letting you create applications based on many independently deployable services that each have granular and autonomous lifecycles.
- As an additional benefit, microservices can scale out independently
- Instead of having a single monolithic application that you must scale out as a unit, you can instead scale out specific microservices
- That way, you can scale just the functional area that needs more processing power or network bandwidth to support demand, rather than scaling out other areas of the application that don't need to be scaled. That means cost savings because you need less hardware

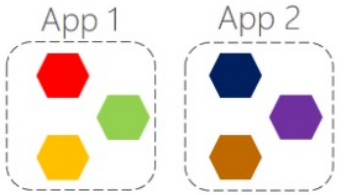
Monolithic deployment approach

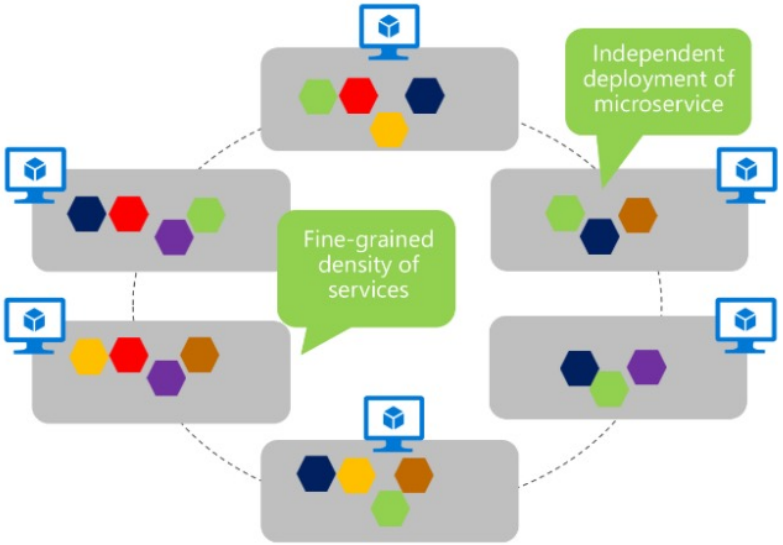
- A traditional application has most of its functionality within a few processes that are componentized with layers and libraries.
- Scales by cloning the app on multiple servers/VMs



Microservices application approach

- A microservice application segregates functionality into separate smaller services.
- Scales out by **deploying each service independently** with multiple instances across servers/VMs

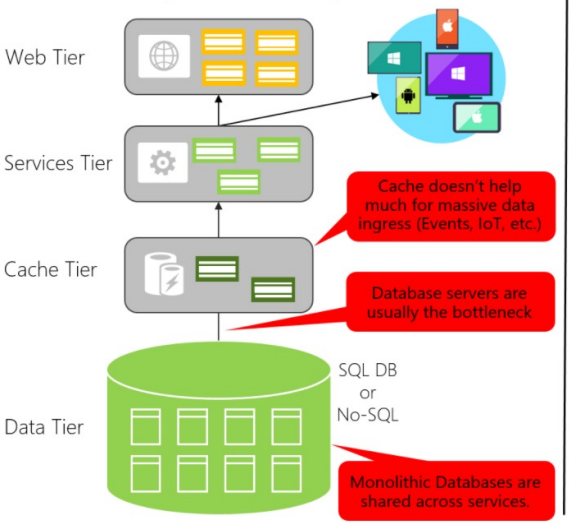




- Architecting fine-grained microservices-based applications enables continuous integration and continuous delivery practices.
- It also accelerates delivery of new functions into the application. Fine-grained composition of applications also allows you to run and test microservices in isolation, and to evolve them autonomously while maintaining clear contracts between them.
- As long as you don't change the interfaces or contracts, you can change the internal implementation of any microservice or add new functionality without breaking other microservices.

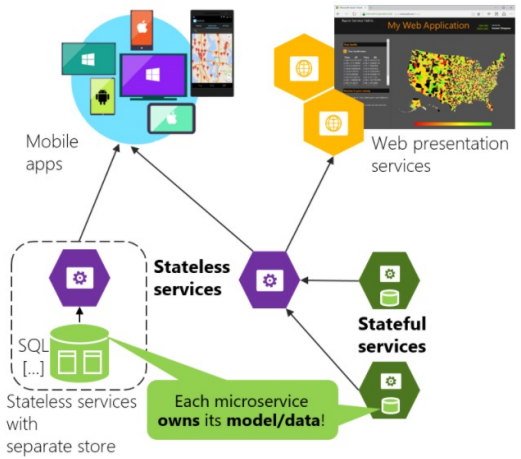
Data in Traditional approach

- Single monolithic database
- Tiers of specific technologies



Data in Microservices approach

- Graph of interconnected microservices
- State typically scoped to the microservice
- Remote Storage for cold data



Challenges and solutions for distributed data management

Challenge #1	Challenge #2	Challenge #3	Challenge #4
• How to define the boundaries of each microservice	• How to create queries that retrieve data from several microservices	• How to achieve consistency across multiple microservices	• How to design communication across microservice boundaries

Challenge #1: How to define the boundaries of each microservice

- focus on the application's logical domain models and related data
- Try to identify decoupled islands of data and different contexts within the same application
- The terms and entities that are used in those different contexts might sound similar, but you might discover that in a particular context, a business concept with one is used for a different purpose in another context, and might even have a different name
  - user: guest vs customer vs buyer
- always attempt to minimize the coupling between those microservices

Challenge #2: How to create queries that retrieve data from several microservices

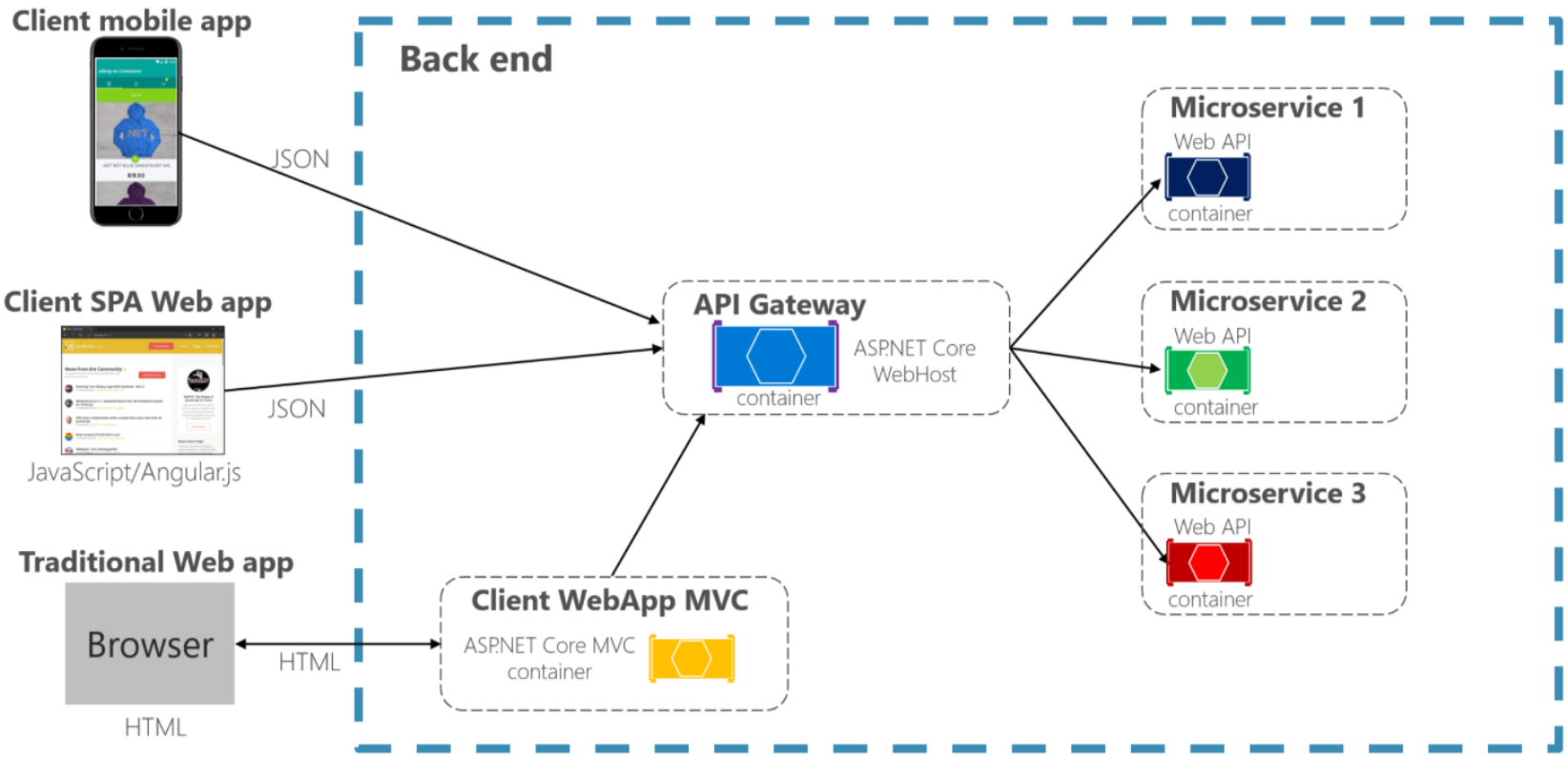
- how to implement queries that retrieve data from several microservices, while avoiding chatty communication to the microservices from remote client apps

- need a way to aggregate information if you want to improve the efficiency in the communications of your system

API Gateway

- In a microservices architecture, the client apps usually need to consume functionality from more than one microservice.
  - the client needs to handle multiple calls to microservice endpoints
- **Coupling**: Without the API Gateway pattern, the client apps are coupled to the internal microservices. The client apps need to know how the multiple areas of the application are decomposed in microservices. When evolving and refactoring the internal microservices, those actions impact maintenance because they cause breaking changes to the client apps due to the direct reference to the internal microservices from the client apps. Client apps need to be updated frequently, making the solution harder to evolve.
- **Too many round trips**: A single page/screen in the client app might require several calls to multiple services. That can result in multiple network round trips between the client and the server, adding significant latency. Aggregation handled in an intermediate level could improve the performance and user experience for the client app.
- **Security issues**: Without a gateway, all the microservices must be exposed to the "external world", making the attack surface larger than if you hide internal microservices that aren't directly used by the client apps. The smaller the attack surface is, the more secure your application can be.
- **Cross-cutting concerns**: Each publicly published microservice must handle concerns such as authorization and SSL. In many situations, those concerns could be handled in a single tier so the internal microservices are simplified.

API Gateway Pattern



- the API gateway sits between the client apps and the microservices.
- It acts as a reverse proxy, routing requests from clients to services.
- It can also provide additional cross-cutting features such as authentication, SSL termination, and cache.

CQRS with query/reads tables

Another solution for aggregating data from multiple microservices is the Materialized View pattern.

In this approach, you generate, in advance (prepare denormalized data before the actual queries happen), a read-only table with the data that's owned by multiple microservices.

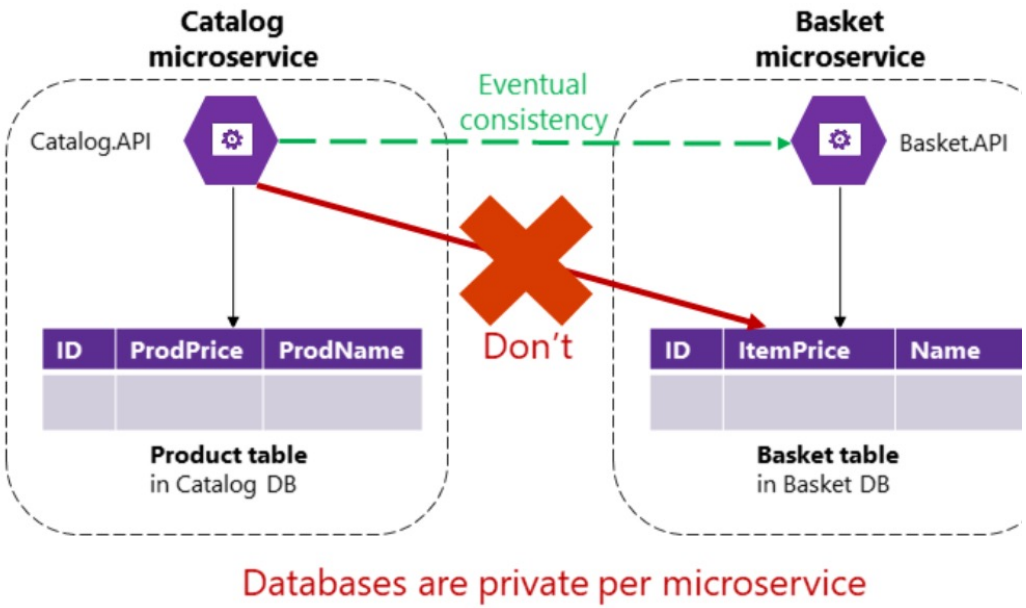
The table has a format suited to the client app's needs. —you create a denormalized table in a different database that's used just for queries.

it also improves performance considerably when compared with a complex join, because you already have the data that the application needs in the query table. Of course, using Command and Query Responsibility Segregation (CQRS)

"Cold data" in central databases

For complex reports and queries that might not require real-time data, a common approach is to export your "hot data" (transactional data from the microservices) as "cold data" into large databases that are used only for reporting.

That central database system can be a Big Data-based system, like Hadoop, a data warehouse like one based on Azure SQL Data Warehouse, or even a single SQL database that's used just for reports (if size won't be an issue).



- how to implement end-to-end business processes while keeping consistency across multiple microservices.
- Eg Price update for a product in Catalog Service

To make an update to the Basket microservice, the Catalog microservice should use eventual consistency probably based on asynchronous communication such as integration events (message and event-based communication).

Most microservice-based scenarios demand availability and high scalability as opposed to strong consistency.

Mission-critical applications must remain up and running, and developers can work around strong consistency by using techniques for working with weak or eventual consistency.

This is the approach taken by most microservice-based architectures.

Challenge #4: How to design communication across microservice boundaries

- Communicating across microservice boundaries is a real challenge.

- In this context, communication doesn't refer to what protocol you should use (HTTP and REST, AMQP, messaging, and so on). Instead, it addresses what communication style you should use, and especially how coupled your microservices should be.

- Depending on the level of coupling, when failure occurs, the impact of that failure on your system will vary significantly.

An HTTP-based approach is perfectly acceptable; the issue here is related to how you use it.

If you create long chains of synchronous HTTP calls across microservices, communicating across their boundaries as if the microservices were objects in a monolithic application, your application will eventually run into problems.

It's recommended that you use only asynchronous interaction for inter-microservice communication, either by using asynchronous message- and event-based communication, or by using (asynchronous) HTTP polling independently of the original HTTP request/response cycle.

(See details here)