Cloud Application Architecture

Ing. Guillermo Zepeda Selman

Source: https://docs.microsoft.com/en-us/azure/architecture/guide/

Introduction

Traditional on-premises

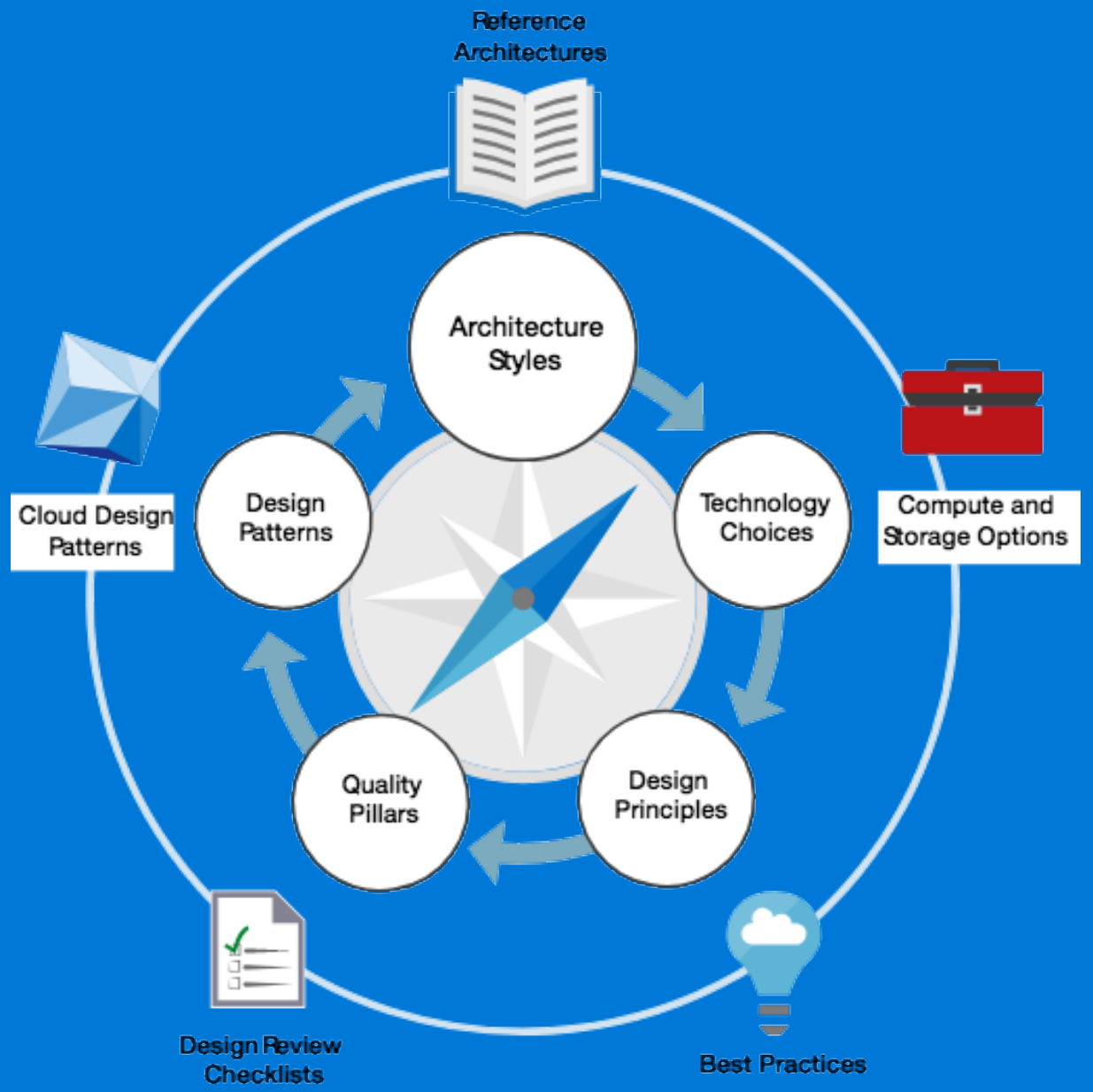
- Monolithic, centralized
- Designed for predictable scalability
- Relational database
- Strong consistency
- Serial and synchronized
- processing
- Design to avoid failures (MTBF)
- Occasional big updates
- Manual management
- Snowflake servers

Modern cloud

- Decomposed, de-centralized
- Design for elastic scale
- Polyglot persistence (mix of storage technologies)
- Eventual consistency
- Parallel and asynchronous processing
- Design for failure (MTTR)
- Frequent small updates
- Automated self-management
- Immutable infrastructure

This guide…

- Architecture styles
- Technology choices
- Design principles
- Quality Pillars
- Cloud design patterns



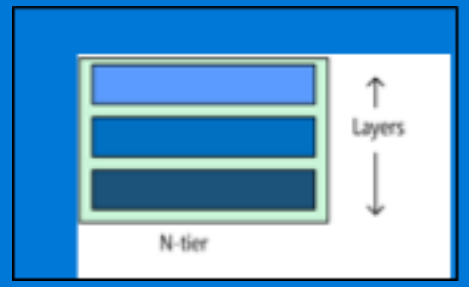
Architecture style

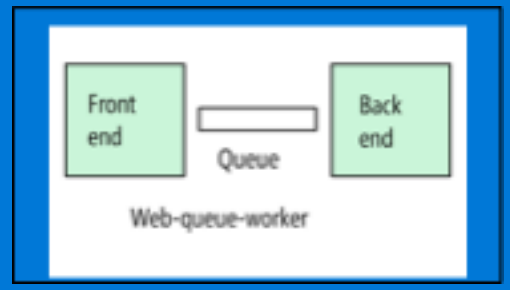
Choose an architecture style

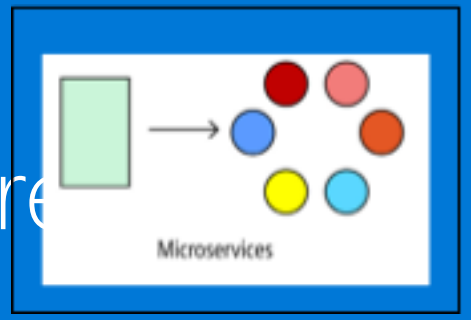
- First decision
- Base on:
  - Complexity
  - Type of domain
  - IaaS or Paas
  - What it will do
- Skills of the team
- Existing architecture?

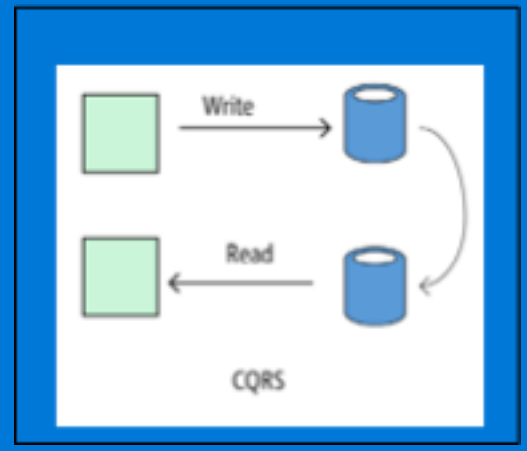
Quick tour of the styles

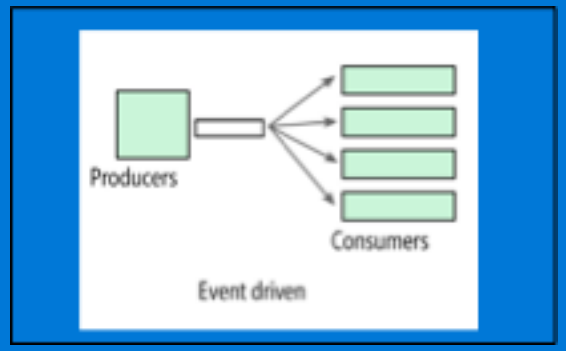
- N-tier
- Web-Queue-Worker
- Microservices
- CQRS
- Event-driven Architecture
- Big Data, Big Compute







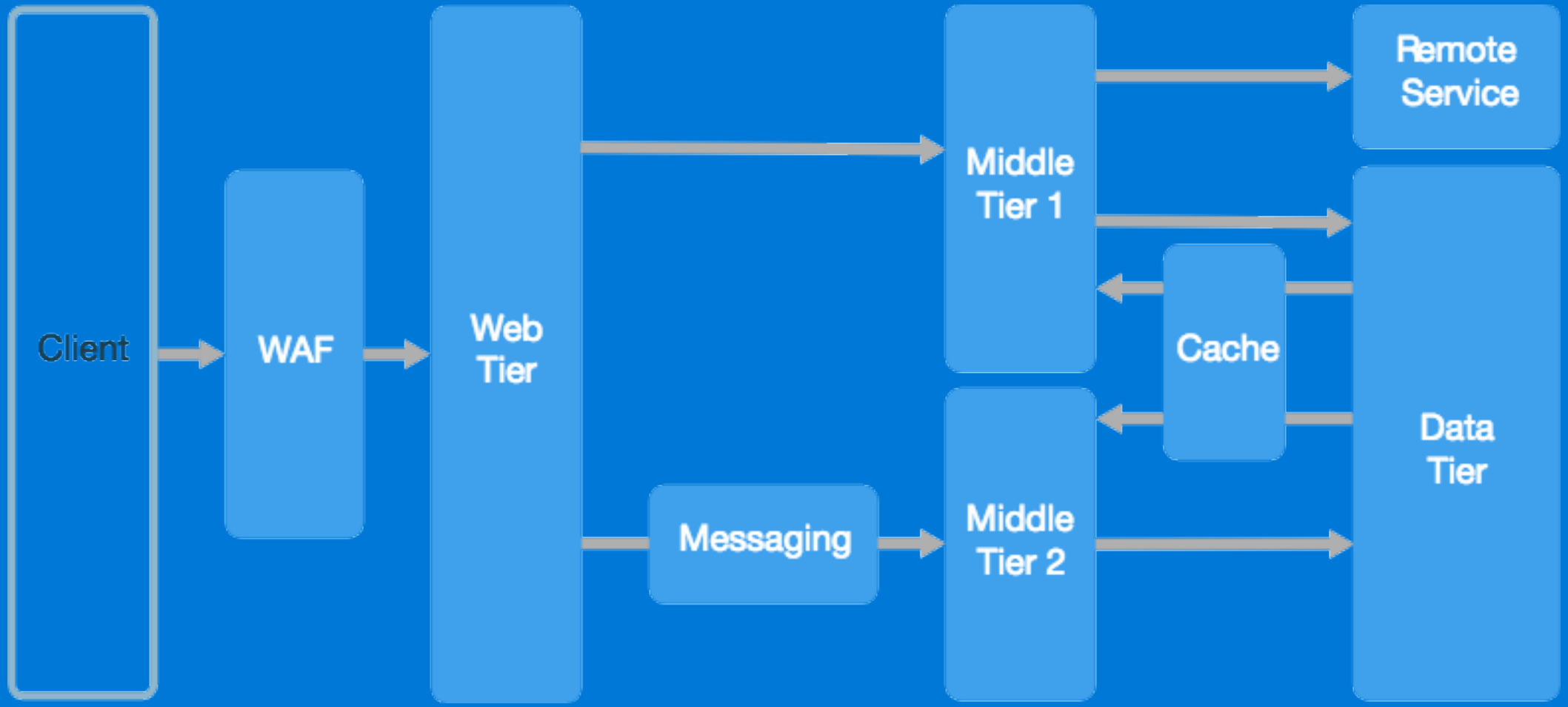




Styles as constraints

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 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.
CQRS	Read/write segregation. Schema and scale are optimized separately.	Collaborative domain where lots of users access the same data.
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.

N-tier architecture style



N-tier architecture style

- When to use?
  - Simple web applications.
  - Migrating an on-premises application to Azure with minimal refactoring.
  - Unified development of on-premises and cloud applications.

- Benefits
  - Portability between cloud and on-premises, and between cloud platforms.
  - Less learning curve for most developers.
  - Natural evolution from the traditional application model.
  - Open to heterogeneous environment (Windows/Linux)

N-tier architecture style

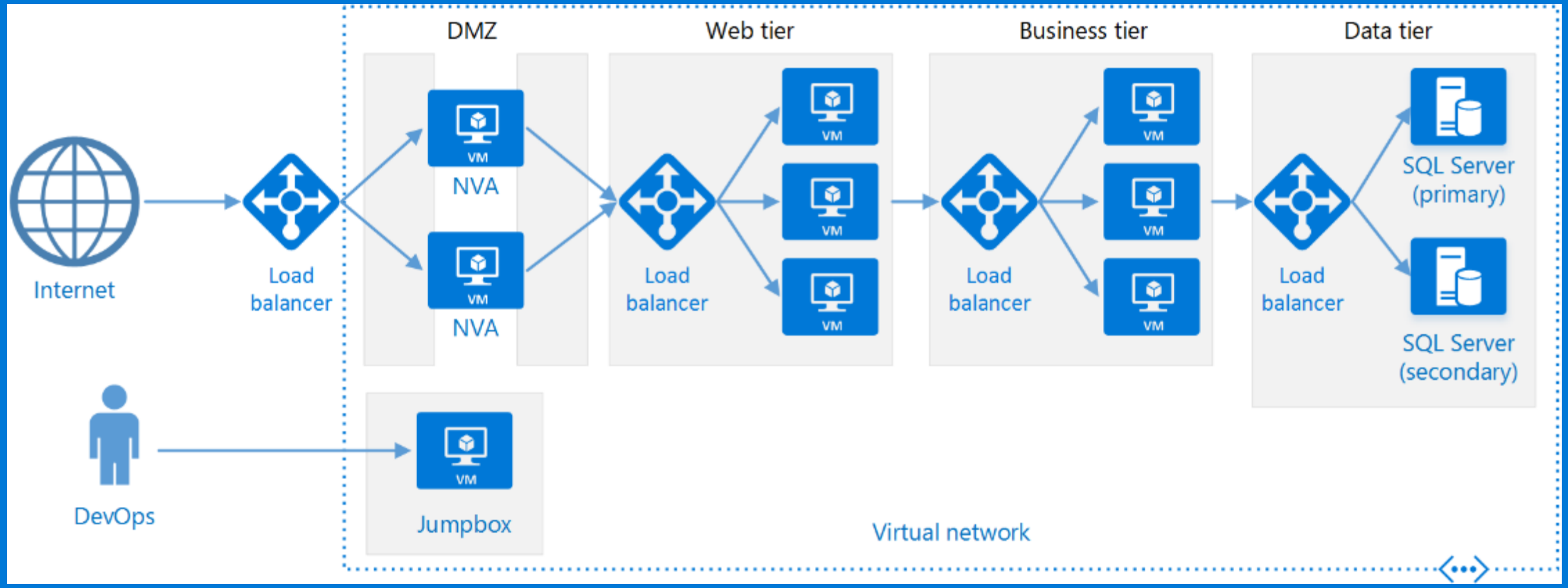
- Benefits
  - Portability between cloud and on-premises, and between cloud platforms.
  - Less learning curve for most developers.
  - Natural evolution from the traditional application model.
  - Open to heterogeneous environment (Windows/Linux)

- 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.

N-tier architecture style

- Best practices
  - Use autoscaling
  - Use asynchronous messaging to decouple tiers
  - Cache semi-static data
  - Configure database tier for high availability
  - Place a web application firewall between the frontend and the internet
  - Place each tier on its own subnet
  - Restrict access to the data tier to the middle tiers only

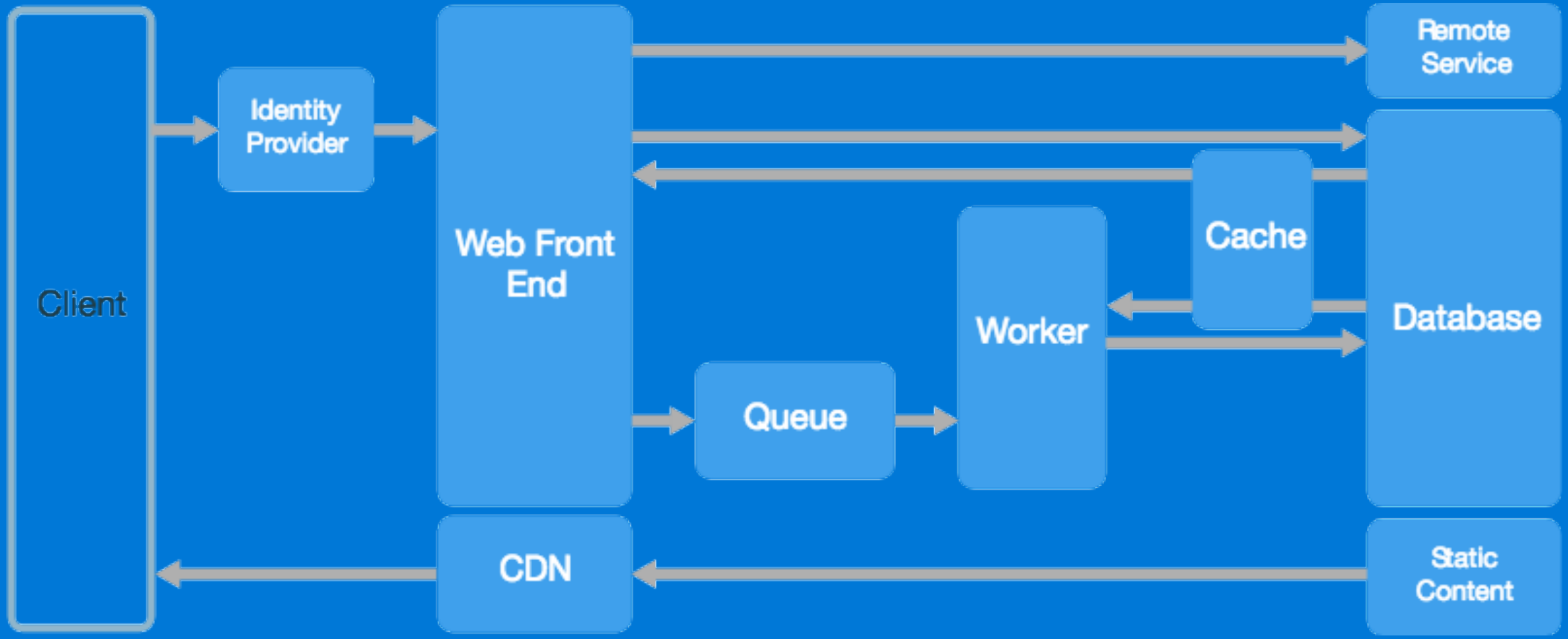
N-tier architecture on VMs



N-tier: additional considerations

- Not restricted to 3 tiers
- Different requirements for scalability, reliability and security for each tier.
- Use VM Scale Sets for autoscaling.
- Look for places in the architecture where you can use a managed service without significant refactoring. In particular, look at caching, messaging, storage, and databases.
- Others…

Web-Queue-Worker architecture style



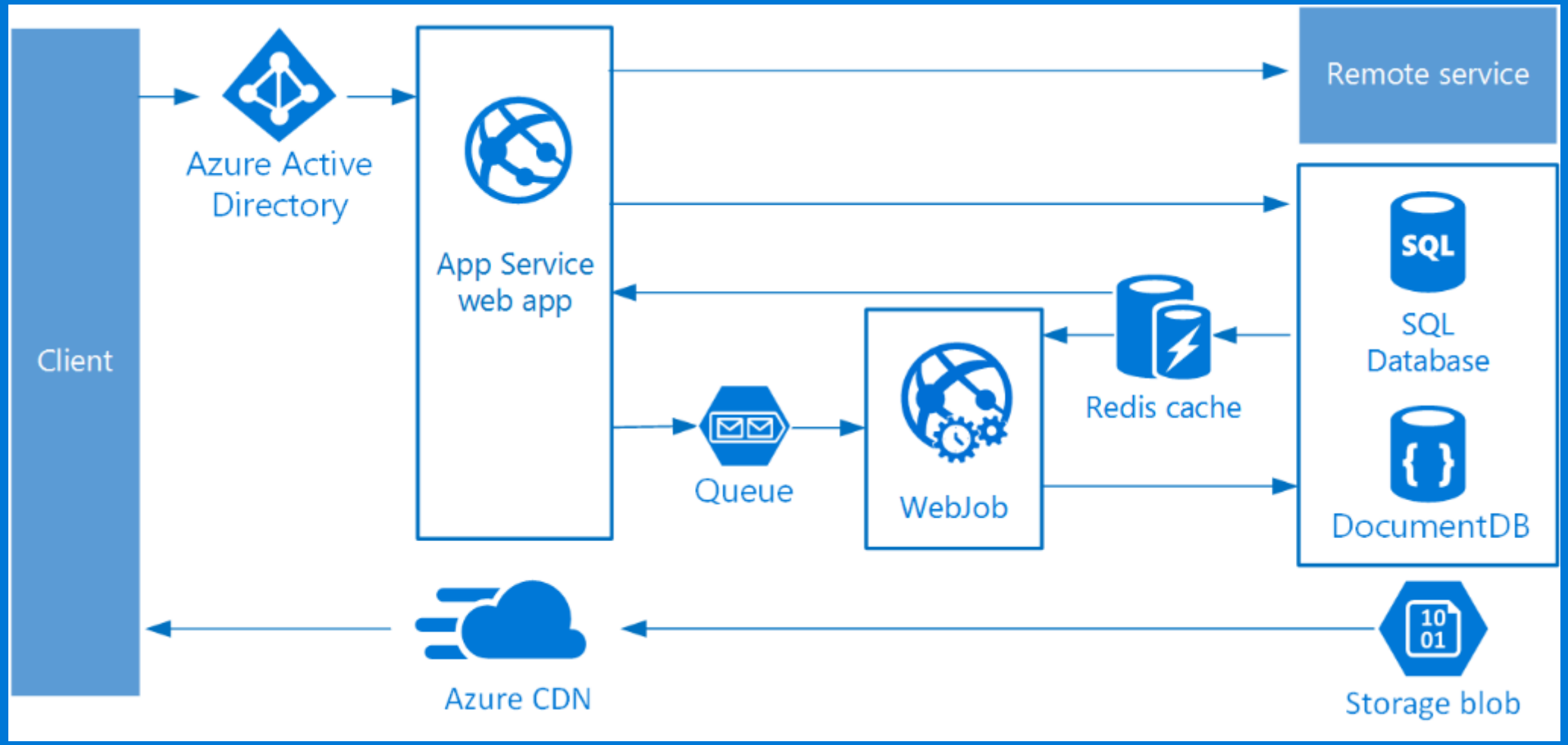
Web-Queue-Worker architecture style

- When to use
  - 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).
- 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.

Web-Queue-Worker architecture style

- Challenges
  - Without careful design, the front end and the worker can become large, monolithic components that are difficult to maintain and update.
  - There may be hidden dependencies, if the front end and worker share data schemas or code modules.
- Best Practices
  - Use polyglot persistence when appropriate.

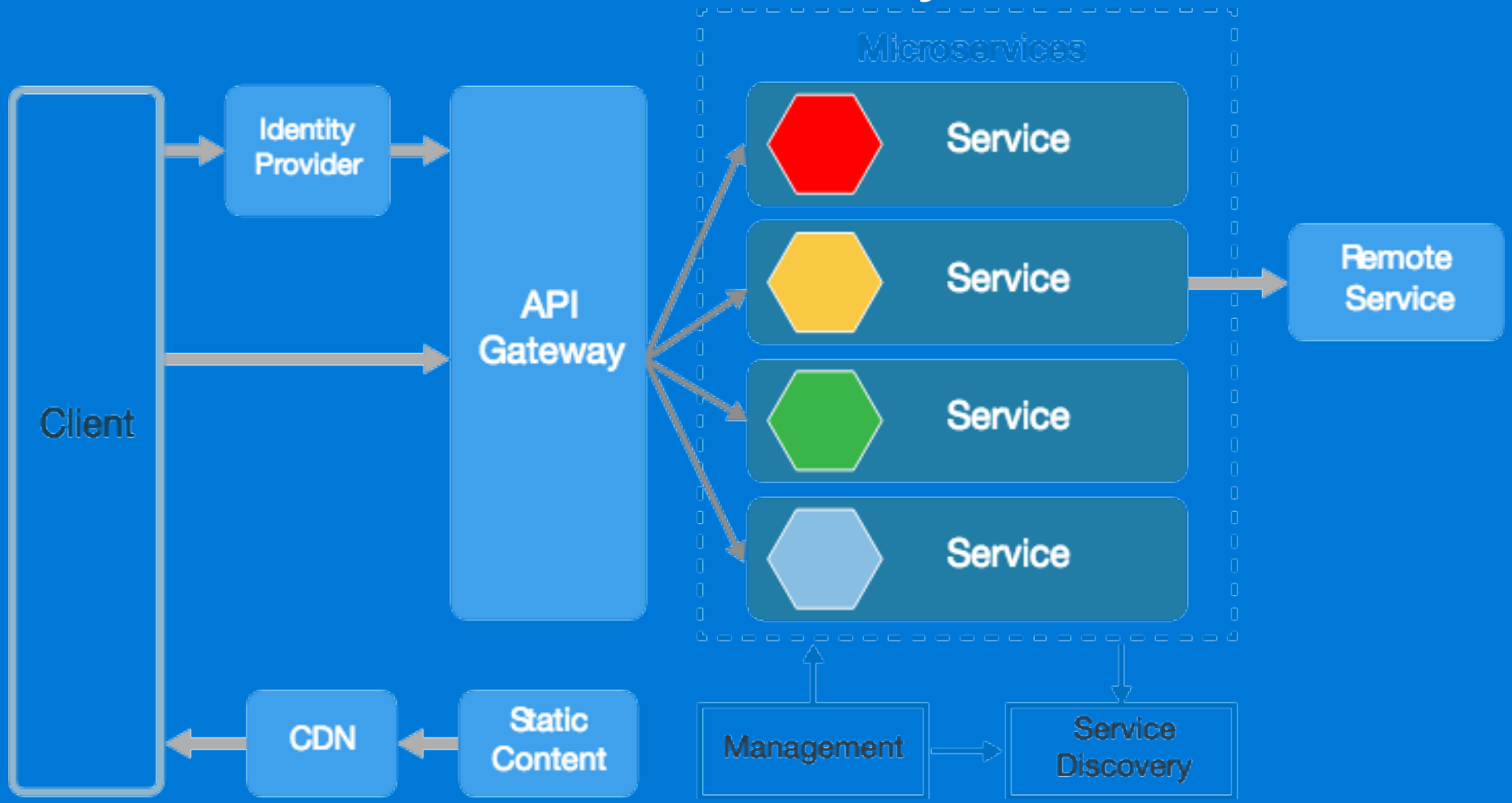
WQW on Azure App Service



WQW additional considerations

- Not every transaction has to go through the queue and worker to storage.
- Use the built-in autoscale feature of App Service to scale out the number of VM instances.
- Consider putting the web app and the WebJob into separate App Service plans.
- Use separate App Service plans for production and testing.
- Use deployment slots to manage deployments.

Microservices architecture style



Microservices

- When to use
  - Large applications that require a high release velocity.
  - Complex applications that need to be highly scalable.
  - Applications with rich domains or many subdomains.
  - An organization that consists of small development teams.
- Benefits
  - Independent deployments.
  - Independent development
  - Small, focused teams
  - Fault isolation
  - Mixed technology stacks
  - Granular scaling

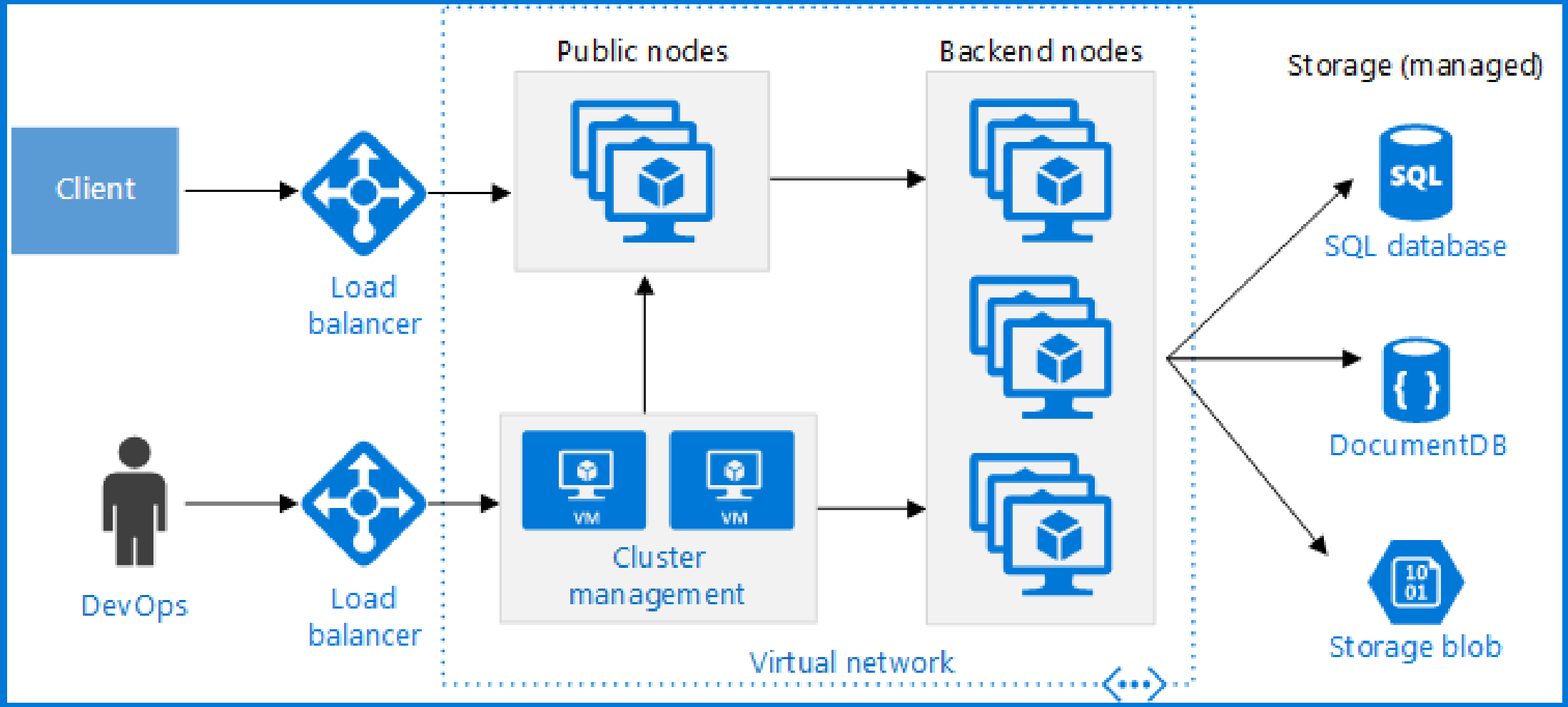
Microservices – Challenges

- Complexity
- Development and test
- Lack of governance
- Network congestion and latency
- Data integrity
- Management
- Versioning
- Skillset

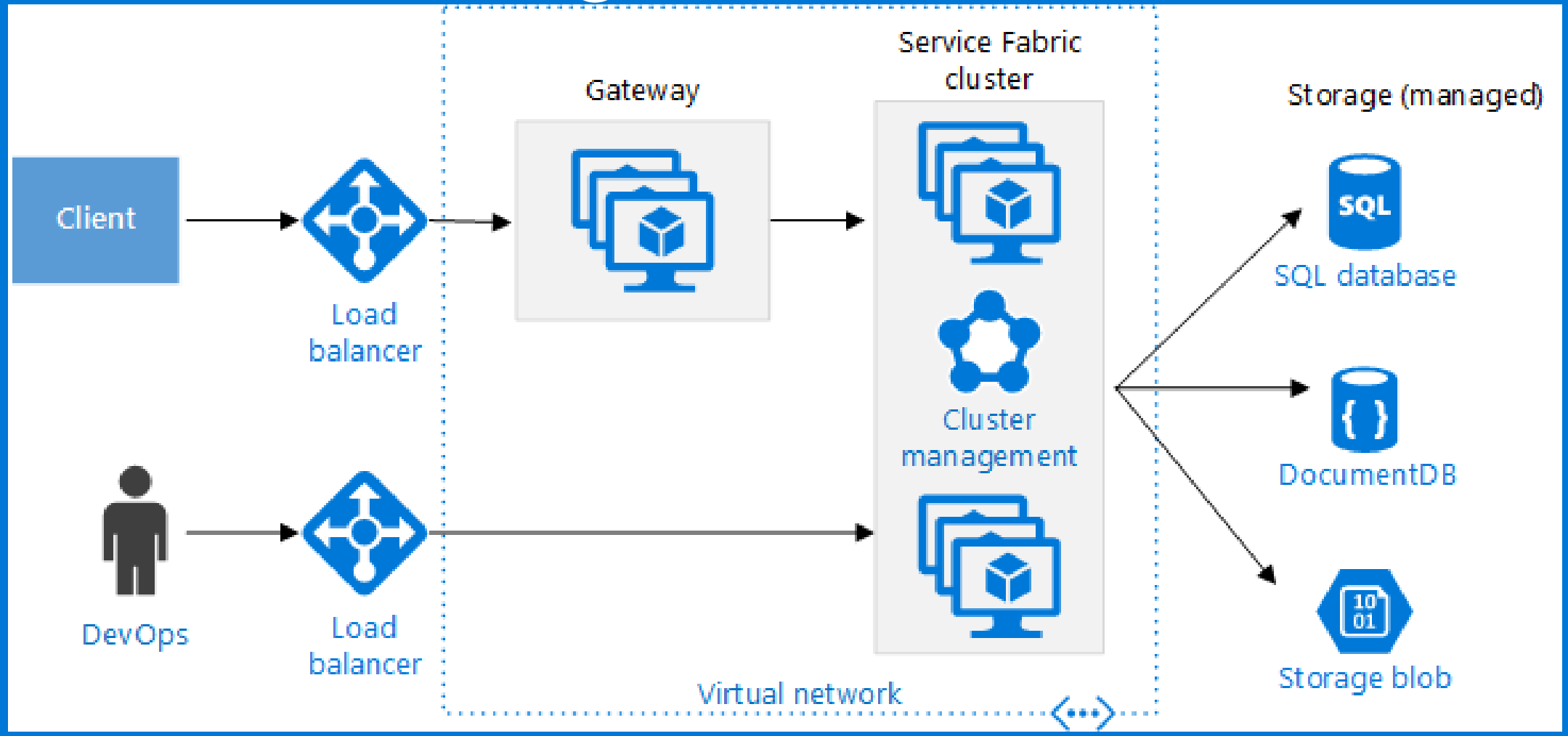
Microservices – Best practices

- Model services around business domain
- Decentralize everything. Individual teams are responsible for designing and building services.
- Avoid sharing code or data schemas.
- 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.
- Offload cross-cutting concerns, such as authentication and SSL termination, to the gateway.
- 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 will require updating the other service. Overly chatty communication between two services may be a symptom of tight coupling and low cohesion.
- Isolate failures. Use resiliency strategies to prevent failures within a service from cascading.

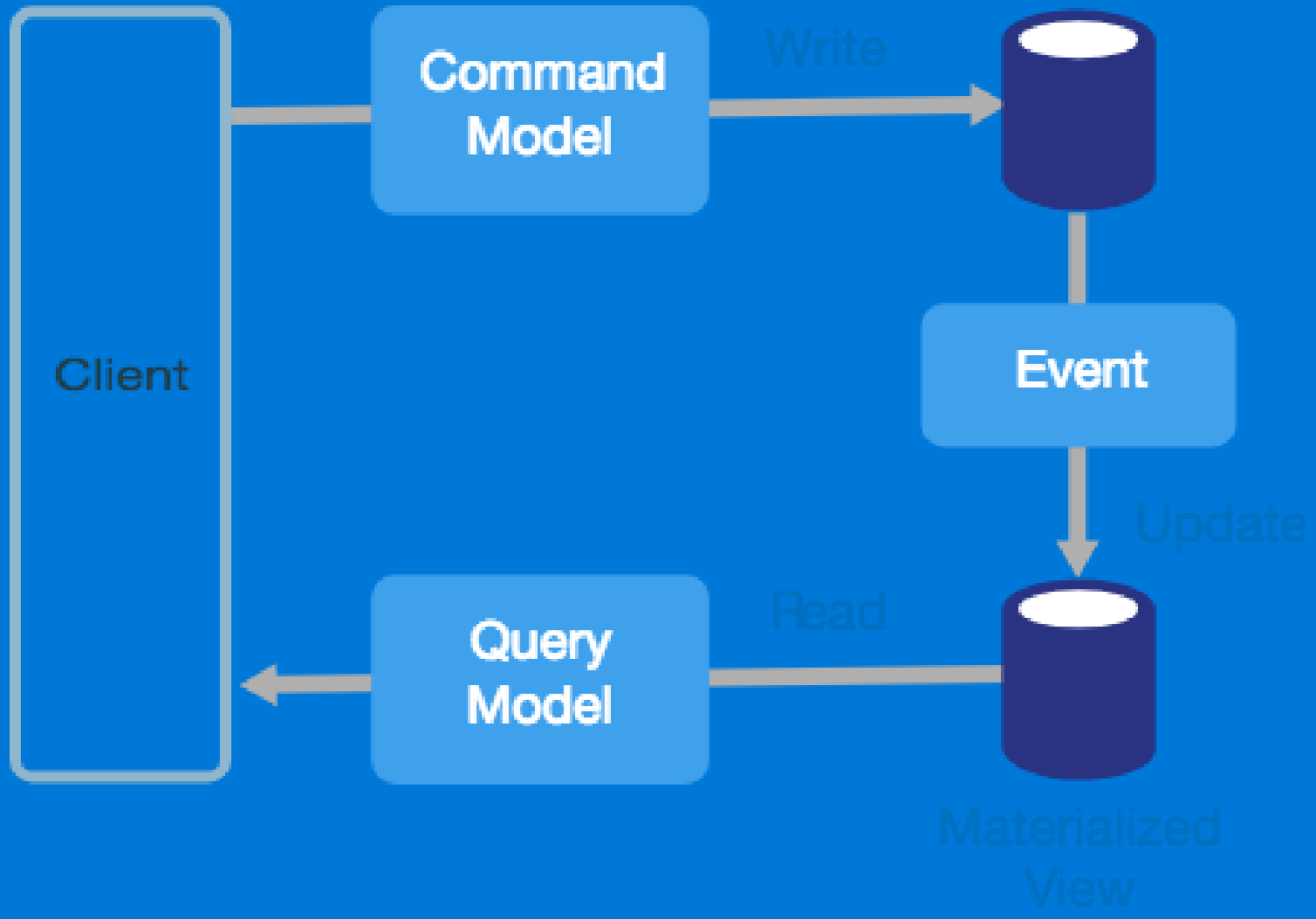
Microservices - Azure Container Service



Microservices using Azure Service Fabric



CQRS Architecture style



CQRS – When to use

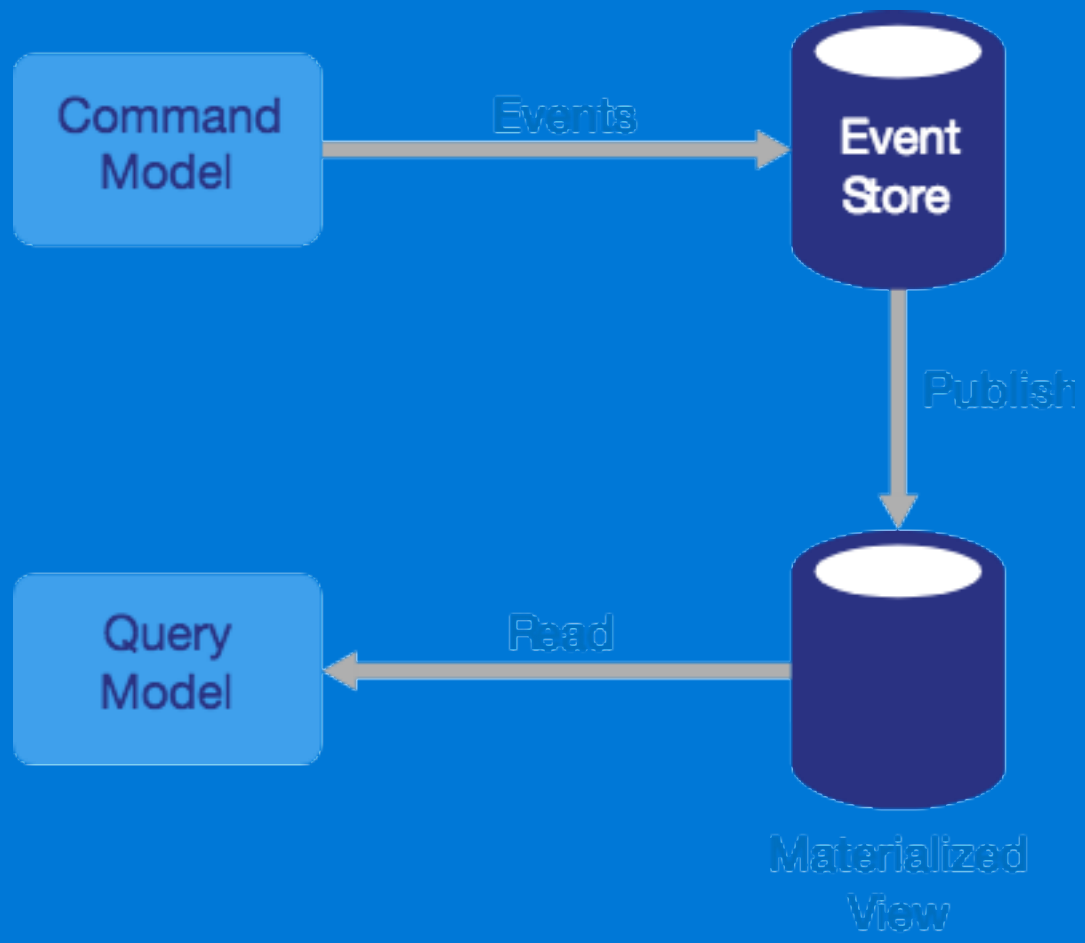
- Consider CQRS for collaborative domains where many users access the same data, especially when the read and write workloads are asymmetrical.
- CQRS is not a top-level architecture that applies to an entire system. Apply CQRS only to those subsystems where there is clear value in separating reads and writes. Otherwise, you are creating additional complexity for no benefit.

CQRS – Benefits

- Independently Scaling
- Optimized Data Schemas
- Security
- Separation of concerns
- Simpler Queries

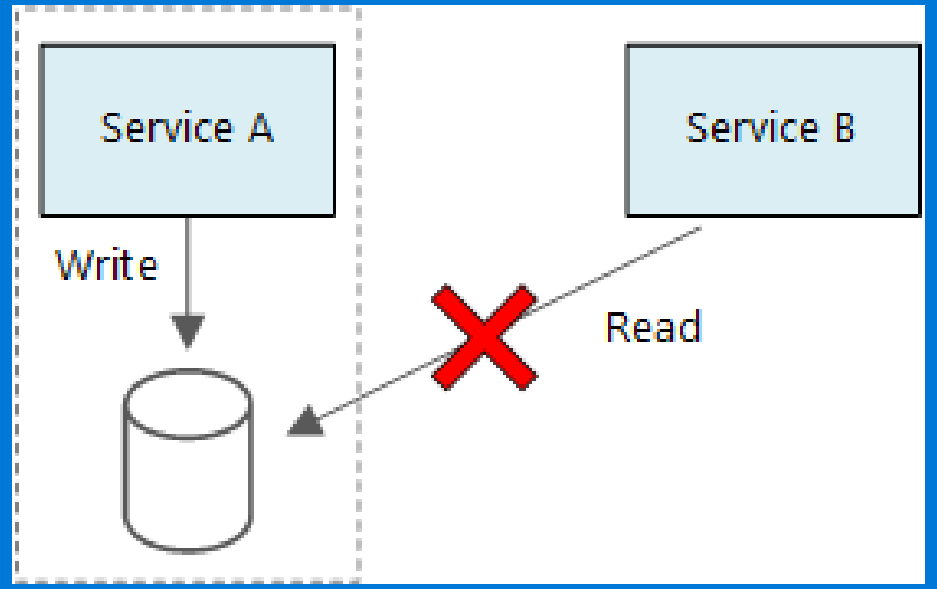
CQRS – Challenges

- Complexity
- Messaging
- Eventual consistency



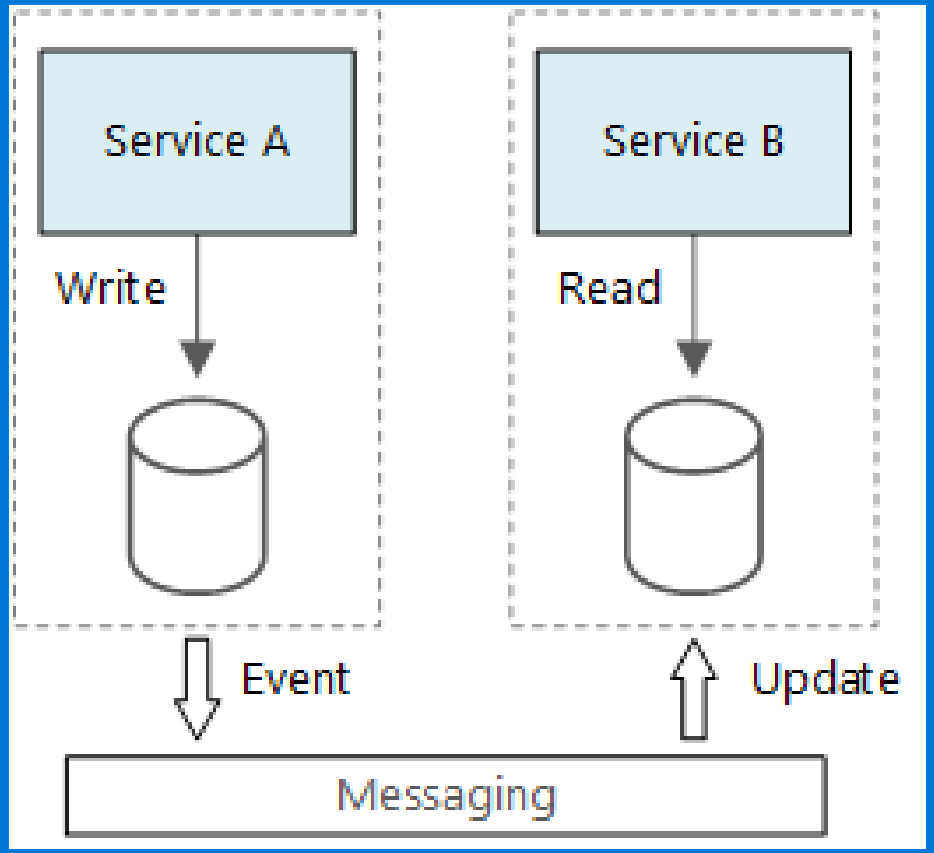
CQRS in microservices

- CQRS can be especially useful in a microservices architecture. One of the principles of microservices is that a service cannot directly access another service's data store.

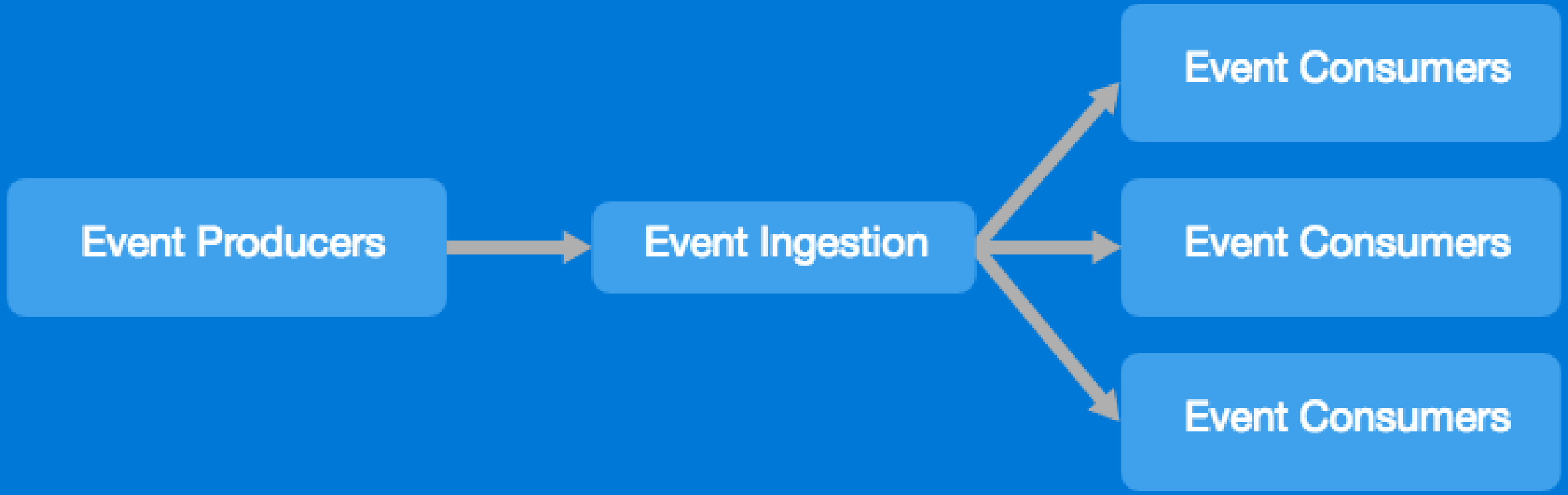


CQRS in microservices

- In the following diagram, Service A writes to a data store, and Service B keeps a materialized view of the data. Service A publishes an event whenever it writes to the data store. Service B subscribes to the event.



Event-driven architecture style



Event-driven architecture

- Two models
  - Pub/sub
  - Event streaming
- On the consumer side:
  - Simple event processing
  - Complex event processing
  - Event stream processing

Event-driven architecture – When to use

- 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.

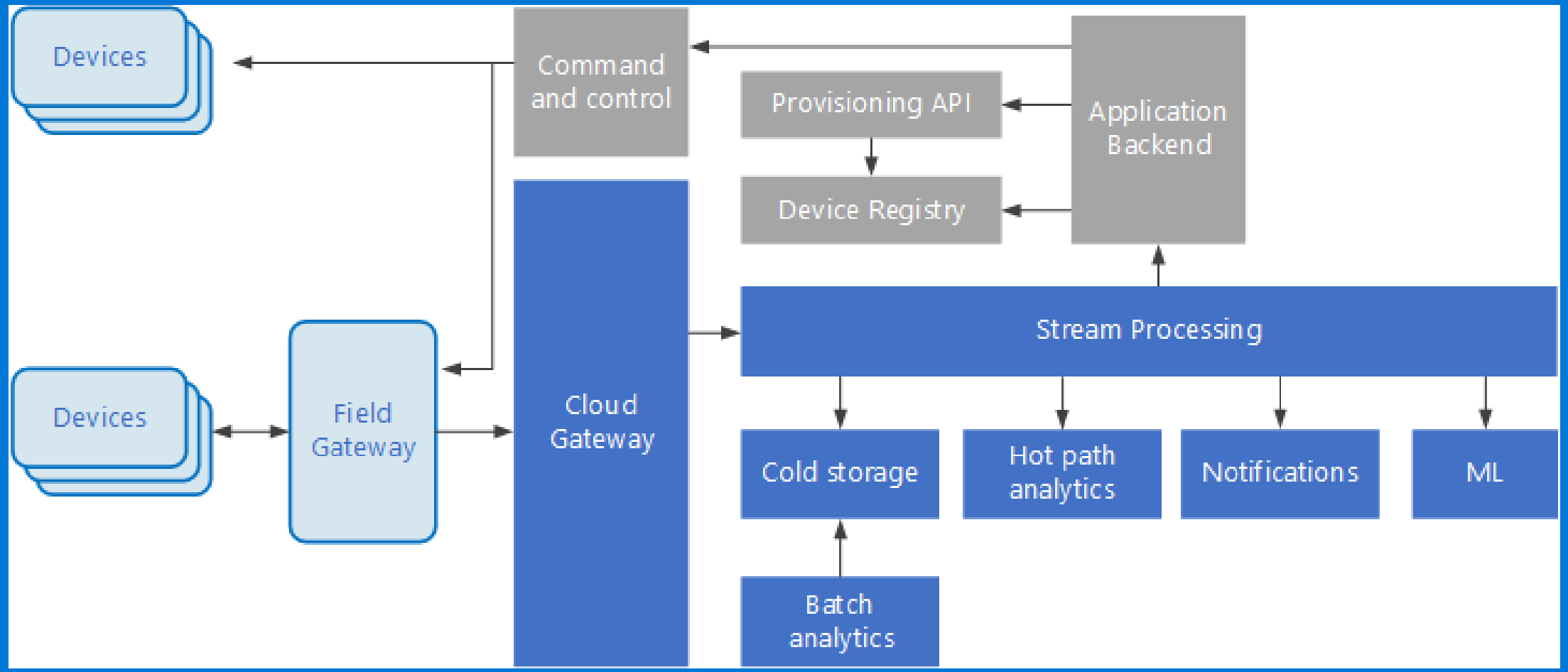
Event-driven architecture – Benefits

- 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.

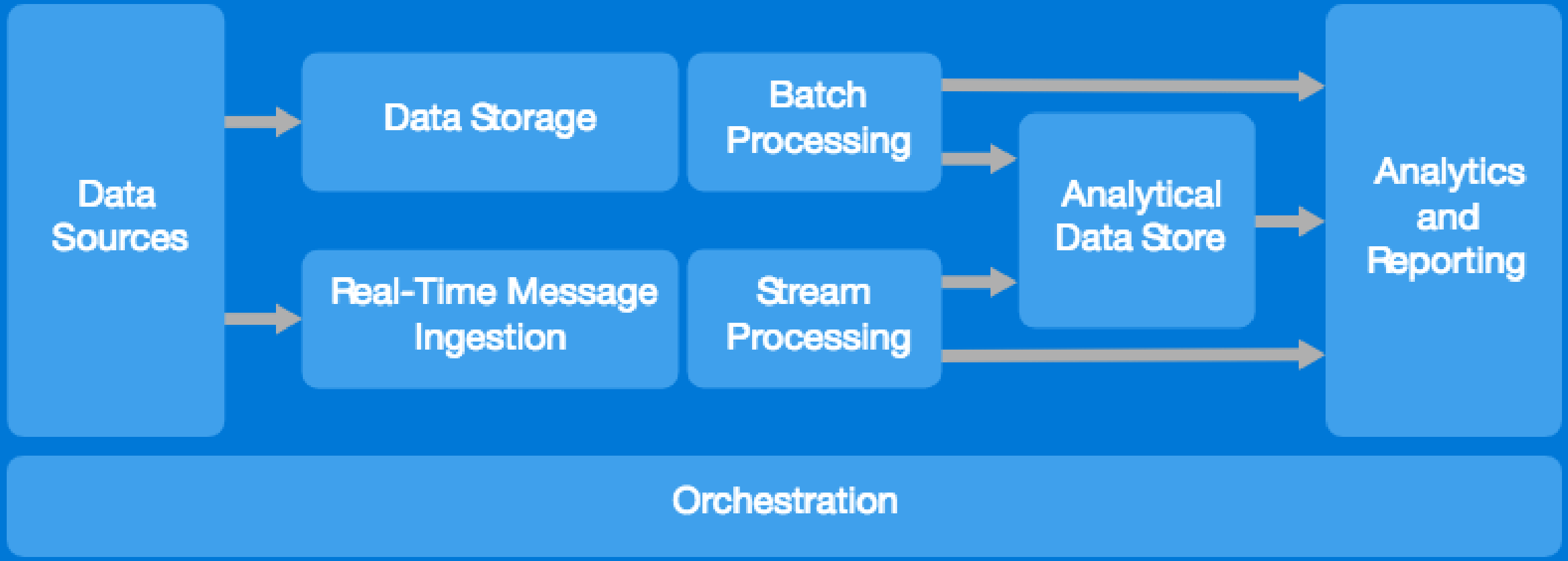
Event-driven architecture – Challenges

- 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.

Event-driven – IoT architecture



Big data architecture style



Big data architecture style – components

- Data sources
- Data storage
- Batch processing
- Real-time message ingestion
- Real-time processing
- Analytical data store
- Analysis and reporting
- Orchestration

- In Azure:
  - Managed services, including Azure Data Lake Store, Azure Data Lake Analytics, Azure Data Warehouse, Azure Stream Analytics, Azure Event Hub, Azure IoT Hub, and Azure Data Factory.
  - Open source technologies based on the Apache Hadoop platform, including HDFS, HBase, Hive, Pig, Spark, Storm, Oozie, Sqoop, and Kafka. These technologies are available on Azure in the Azure HDInsight service.

Big data architecture – When to use it

- 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.

Big data architecture – Benefits

- Technology choices
- Performance through parallelism
- Elastic scale
- Interoperability with existing solutions

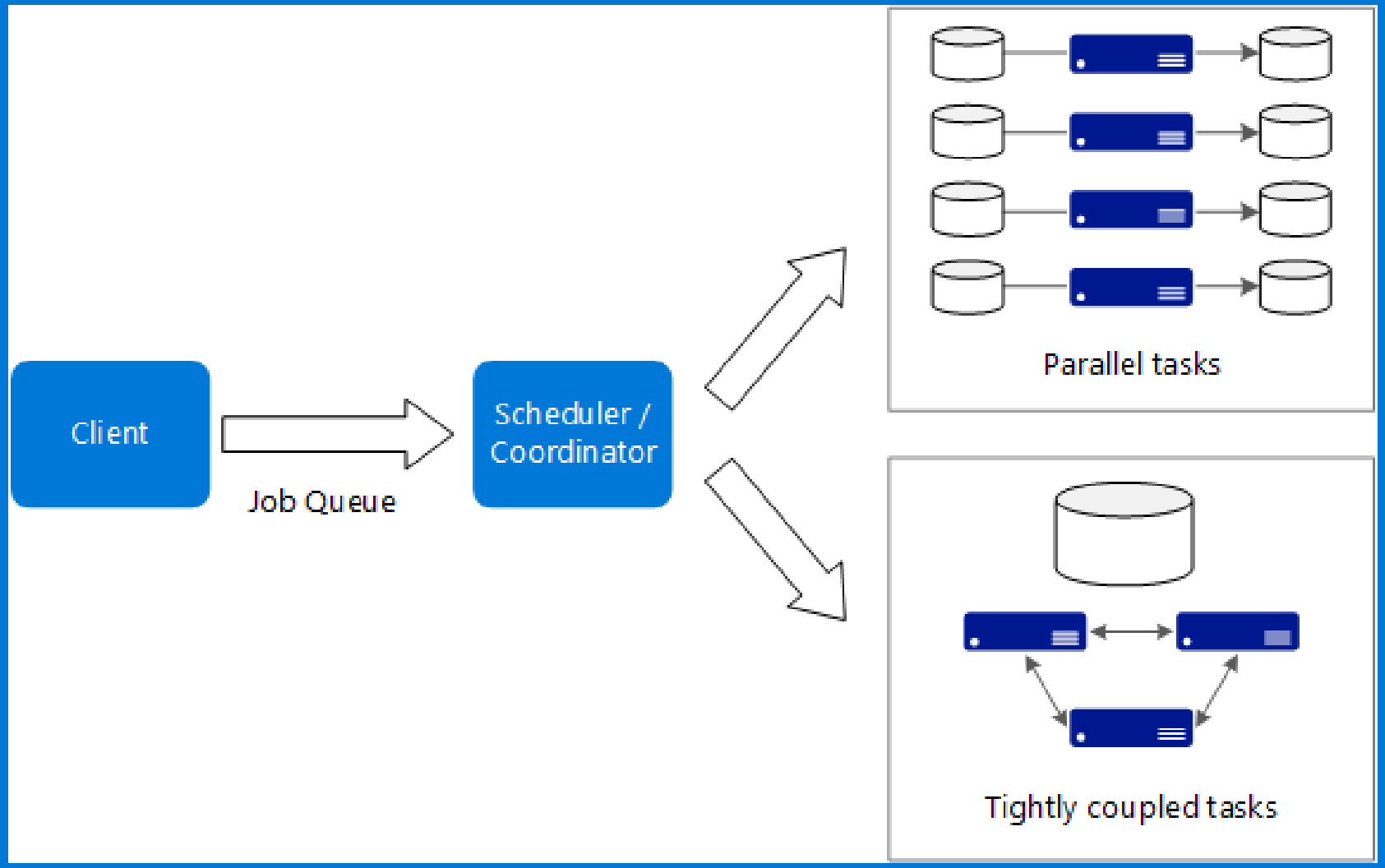
Big data architecture – challenges

- Complexity
- Skillset
- Technology maturity
- Security

Big data architecture – best practices

- Leverage parallelism
- Partition data
- Apply schema-on-read semantics
- Process data in-place
- Balance utilization and time costs
- Separate cluster resources
- Orchestrate data ingestion
- Scrub sensitive data early

Big compute architecture style



Big compute architecture – When to use

- 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.

Big compute architecture – 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.

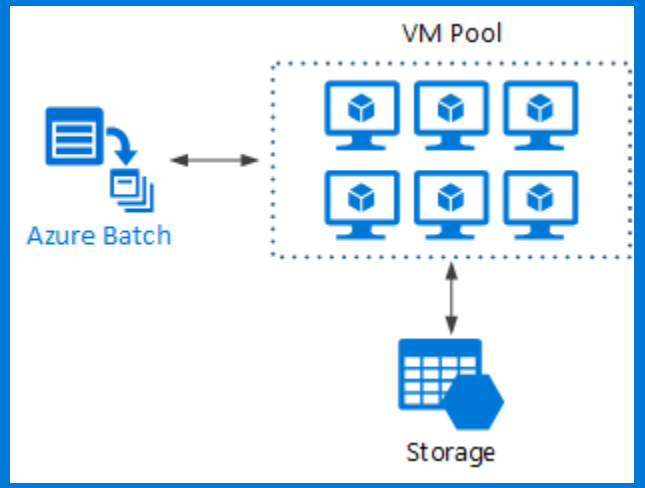
Big compute architecture – challenges

- 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

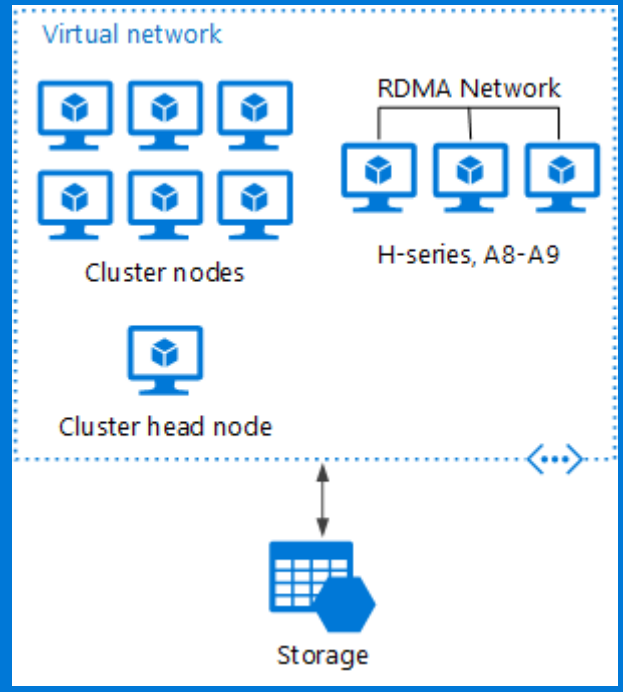
Azure Batch is a managed service for running large-scale high-performance computing (HPC) applications.

Using Azure Batch, you configure a VM pool, and upload the applications and data files. Then the Batch service provisions the VMs, assign tasks to the VMs, runs the tasks, and monitors the progress. Batch can automatically scale out the VMs in response to the workload. Batch also provides job scheduling.

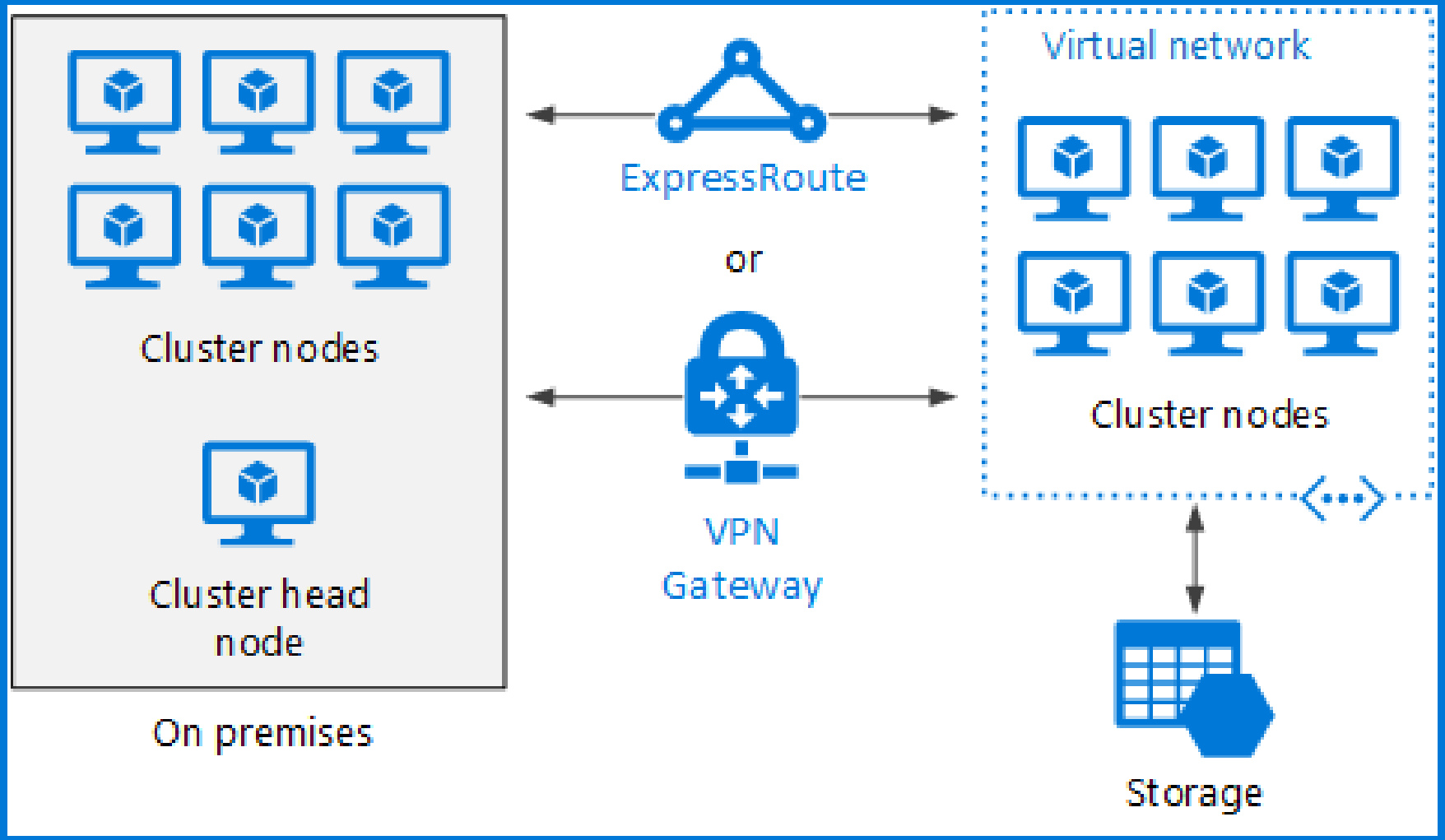


Big compute running on VMs

- HPC = High performance computing
- HPC Pack allows you to create and manage HPC clusters consisting of dedicated on-premises Windows or Linux compute nodes, part-time servers, workstation computers, and dedicated or on-demand compute resources that are deployed in Microsoft Azure.



Big compute running on VMs – Hybrid



Choose a compute service

Compute options

- The term *compute* refers to the hosting model for the computing resources that your application runs on.

- IaaS
- PaaS
- FaaS (Serverless)

Azure Compute options

- Virtual Machines
- App Service
- Service Fabric
- Azure Container Service
- Azure Container Instance
- Azure Function
- Azure Batch
- Cloud Services

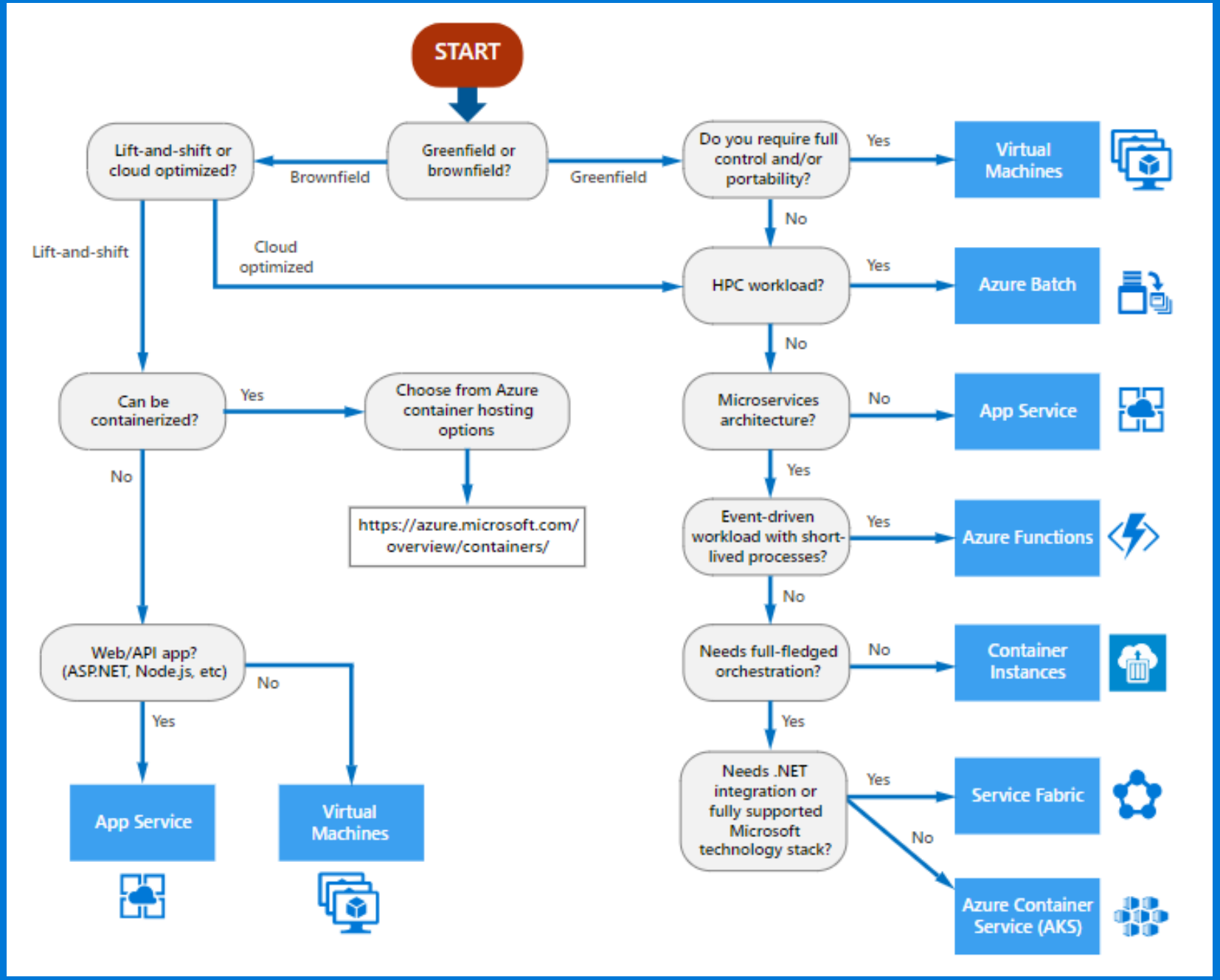
Factors to consider

- Hosting model
- DevOps
- Scalability
- Availability
- Cost
- Overall limitations
- Appropiate architectures

Decision tree – Definitions

- Greenfield
- Brownfield
- Lift and shift
- Cloud optimized

Decision tree



Hosting model

Criteria	Virtual Machines	App Service	Service Fabric	Azure Functions	Azure Container Service	Container Instances	Azure Batch
Application composition	Agnostic	Applications, containers	Services, guest executables, containers	Functions	Containers	Containers	Scheduled jobs
Density	Agnostic	Multiple apps per instance via app service plans	Multiple services per VM	Serverless [1]	Multiple containers per VM	No dedicated instances	Multiple apps per VM
Minimum number of nodes	1 [2]	1	5 [3]	Serverless [1]	3	No dedicated nodes	1 [4]
State management	Stateless or Stateful	Stateless	Stateless or stateful	Stateless	Stateless or Stateful	Stateless	Stateless
Web hosting	Agnostic	Built in	Agnostic	Not applicable	Agnostic	Agnostic	No
Can be deployed to dedicated VNet?	Supported	Supported [5]	Supported	Supported [5]	Supported	Not supported	Supported
Hybrid connectivity	Supported	Supported [6]	Supported	Supported [7]	Supported	Not supported	Supported

DevOps

Criteria	Virtual Machines	App Service	Service Fabric	Azure Functions	Azure Container Service	Container Instances	Azure Batch
Local debugging	Agnostic	IIS Express, others [1]	Local node cluster	Visual Studio or Azure Functions CLI	Local container runtime	Local container runtime	Not supported
Programming model	Agnostic	Web and API applications, WebJobs for background tasks	Guest executable, Service model, Actor model, Containers	Functions with triggers	Agnostic	Agnostic	Command line application
Application update	No built-in support	Deployment slots	Rolling upgrade (per service)	Deployment slots	Depends on orchestrator. Most support rolling updates	Update container image	Not applicable

Scalability

Criteria	Virtual Machines	App Service	Service Fabric	Azure Functions	Azure Container Service	Container Instances	Azure Batch
Auto-scaling	VM scale sets	Built-in service	VM Scale Sets	Built-in service	Not supported	Not supported	N/A
Load balancer	Azure Load Balancer	Integrated	Azure Load Balancer	Integrated	Azure Load Balancer	No built-in support	Azure Load Balancer
Scale limit	Platform image: 1000 nodes per VMSS, Custom image: 100 nodes per VMSS	20 instances, 100 with App Service Environment	100 nodes per VMSS	200 instances per Function app	100 [1]	20 container groups per subscription by default. Contact customer service for increase. [2]	20 core limit by default. Contact customer service for increase.

Availability

Criteria	Virtual Machines	App Service	Service Fabric	Azure Functions	Azure Container Service	Container Instances	Azure Batch
SLA	SLA for Virtual Machines	SLA for App Service	SLA for Service Fabric	SLA for Functions	SLA for Azure Container Service	SLA for Container Instances	SLA for Azure Batch
Multi region failover	Traffic manager	Traffic manager	Traffic manager, Multi-Region Cluster	Not supported	Traffic manager	Not supported	Not Supported

Other

Criteria	Virtual Machines	App Service	Service Fabric	Azure Functions	Azure Container Service	Container Instances	Azure Batch
SSL	Configured in VM	Supported	Supported	Supported	Configured in VM	Supported with sidecar container	Supported
Cost	Windows, Linux	App Service pricing	Service Fabric pricing	Azure Functions pricing	Azure Container Service pricing	Container Instances pricing	Azure Batch pricing
Suitable architecture styles	N-Tier, Big compute(HPC)	Web-Queue-Worker, N-Tier	Microservices, Event-driven architecture	Microservices, Event-driven architecture	Microservices, Event-driven architecture	Microservices, task automation, batch jobs	Big compute(HPC)

Choose a data store

Choose the right data store

- Business use data for a lot of different operations
- Single data store is not usually the best approach.
  - Polyglot persistence
- Data stores are categorized by
  - How data is structured
  - Types of operations they support.
- Not all data stores in a given category provide the same feature-set

Types of Data Stores

- RDBMS
- Key/Value Stores
- Document Databases
- Graph Databases
- Column-family Databases
- Data Analytics
- Search engine Databases
- Time Series databases
- Object Storage
- Shared files

Criteria for choosing a data store

- Functional requirements
  - Data Format
  - Data Size
  - Scale and structure
  - Data relationships
  - Consistency model
  - Schema flexibility
  - Concurrency
  - Data movement
  - Data lifecyle
  - Other supported features

Criteria for choosing a data store

- Non-functional requirements
  - Performance and scalability
  - Reliability
  - Replication
  - Limits
- Management and cost
  - Managed service
  - Region availability
  - Portability
  - Licensing
  - Overall cost
  - Cost effectiveness

Criteria for choosing a data store

- Security
  - Security
  - Auditing
  - Networking requirements
- DevOps
  - Skill set
  - Clients

Relational database management systems

- Azure SQL Database
- Azure Database for MySQL
- Azure Database for PostgreSQL

RDBMS –Workload

- Both the creation of new records and updates to existing data happen regularly.
- Multiple operations have to be completed in a single transaction.
- Requires aggregation functions to perform cross-tabulation.
- Strong integration with reporting tools is required.
- Relationships are enforced using database constraints.
- Indexes are used to optimize query performance.
- Allows access to specific subsets of data.

RDBMS – Data Type

- Data is highly normalized.
- Database schemas are required and enforced.
- Many-to-many relationships between data entities in the database.
- Constraints are defined in the schema and imposed on any data in the database.
- Data requires high integrity. Indexes and relationships need to be maintained accurately.
- Data requires strong consistency. Transactions operate in a way that ensures all data are 100% consistent for all users and processes.
- Size of individual data entries is intended to be small to medium-sized.

RDBMS – Examples

- Line of business (human capital management, customer relationship management, enterprise resource planning)
- Inventory management
- Reporting database
- Accounting
- Asset management
- Fund management
- Order management

Document databases

- Cosmos DB

Key	Document

1001	{
	"CustomerID": 99,
	"OrderItems": [
	{ "ProductID": 2010,
	"Quantity": 2,
	"Cost": 520
	},
	{ "ProductID": 4365,
	"Quantity": 1,
	"Cost": 18
	}],
	"OrderDate": "04/01/2017"
	}

1002	{
	"CustomerID": 220,
	"OrderItems": [
	{ "ProductID": 1285,
	"Quantity": 1,
	"Cost": 120
	}],
	"OrderDate": "05/08/2017"
	}

Document databases – Workload

- General purpose.
- Insert and update operations are common. Both the creation of new records and updates to existing data happen regularly.
- No object-relational impedance mismatch. Documents can better match the object structures used in application code.
- Optimistic concurrency is more commonly used.
- Data must be modified and processed by consuming application.
- Data requires index on multiple fields.
- Individual documents are retrieved and written as a single block.

Document databases – Data type

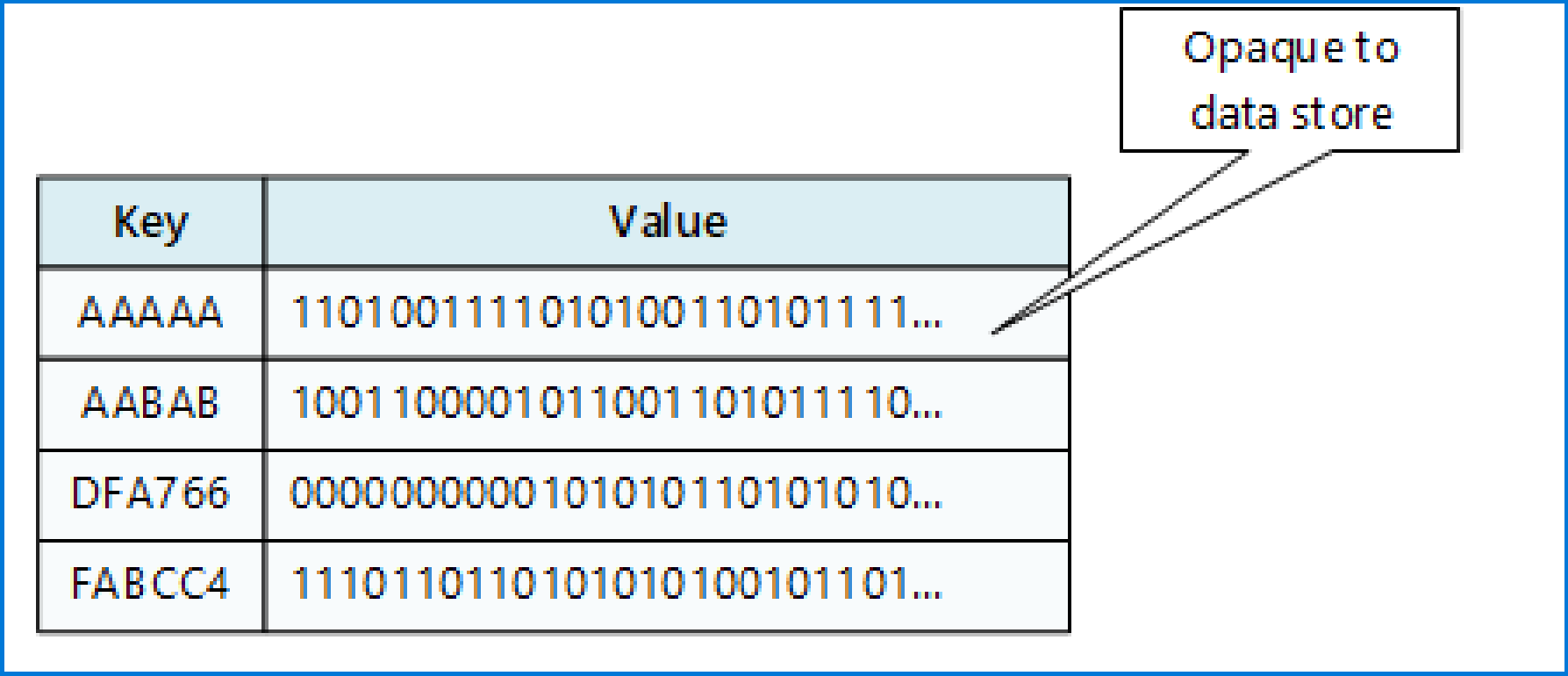
- Data can be managed in de-normalized way.
- Size of individual document data is relatively small.
- Each document type can use its own schema.
- Documents can include optional fields.
- Document data is semi-structured, meaning that data types of each field are not strictly defined.
- Data aggregation is supported..

Document databases – Examples

- Product catalog
- User accounts
- Bill of materials
- Personalization
- Content management
- Operations data
- Inventory management
- Transaction history data
- Materialized view of other NoSQL stores. Replaces file/BLOB indexing.

Key/value stores

- Cosmos DB
- Azure Redis Cache



Key/value stores – Workload

- Data is identified and accessed using a single ID key, like a dictionary.
- Massively scalable.
- No joins, lock, or unions are required.
- No aggregation mechanisms are used.
- Secondary indexes are generally not used.

Key/value stores – Data Type

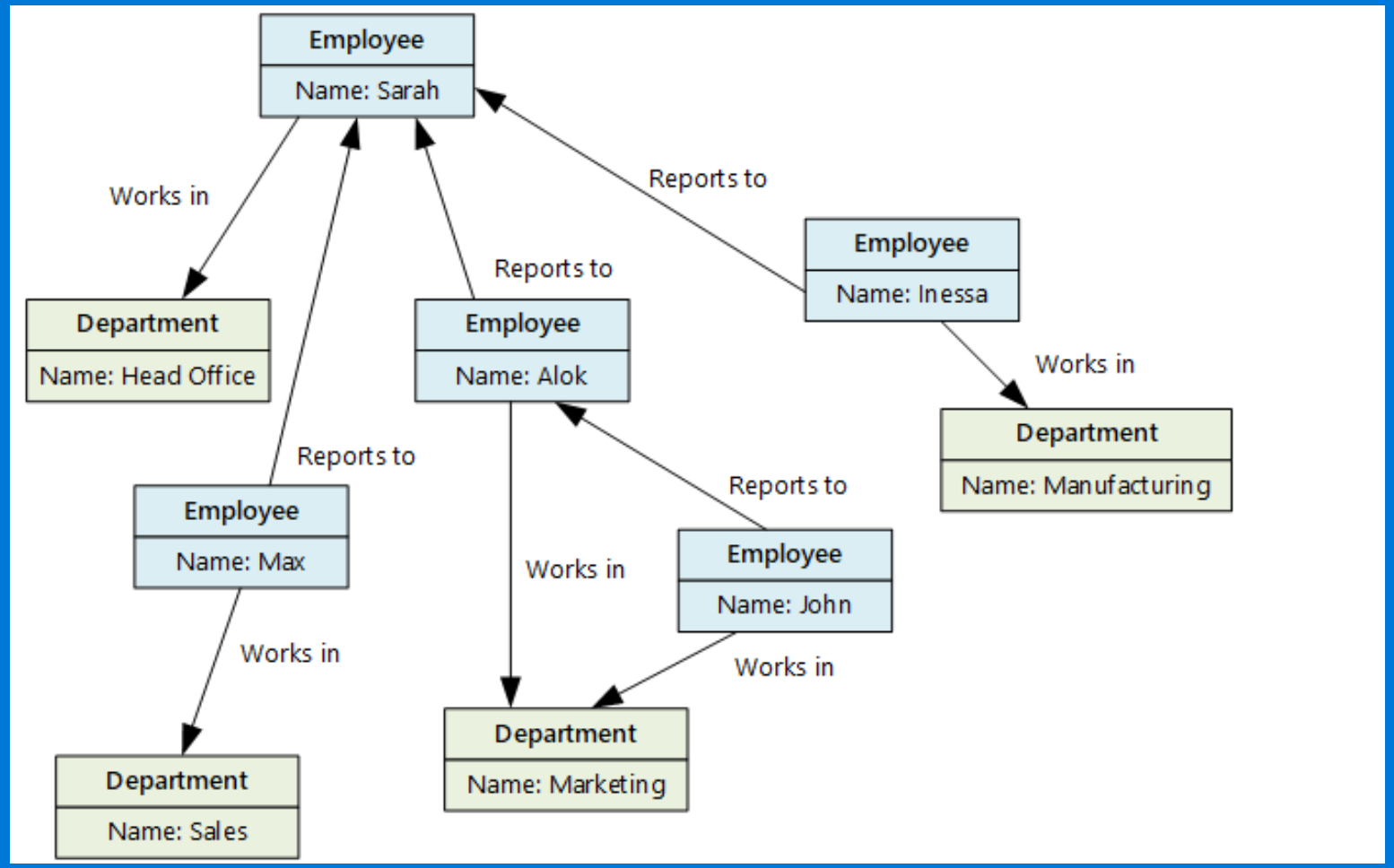
- Data size tends to be large.
- Each key is associated with a single value, which is an unmanaged data BLOB.
- There is no schema enforcement.
- No relationships between entities.

Key/value stores – Data caching

- Data caching
- Session management
- User preference and profile management
- Product recommendation and ad serving
- Dictionaries

Graph databases

- Cosmos DB



Graph databases – Workload

- The relationships between data items are very complex, involving many hops between related data items.
- The relationship between data items are dynamic and change over time.
- Relationships between objects are first-class citizens, without requiring foreign-keys and joins to traverse.

Graph databases – Data type

- Data is comprised of nodes and relationships.
- Nodes are similar to table rows or JSON documents.
- Relationships are just as important as nodes, and are exposed directly in the query language.
- Composite objects, such as a person with multiple phone numbers, tend to be broken into separate, smaller nodes, combined with traversable relationships

Graph databases – Examples

- Organization charts
- Social graphs
- Fraud detection
- Analytics
- Recommendation engines

Column-Family Databases

- HBase in HDInsight

CustomerID	Column Family: Identity
001	**First name:** Mu Bae **Last name:** Min
002	**First name:** Francisco **Last name:** Vila Nova **Suffix:** Jr.
003	**First name:** Lena **Last name:** Adamcyz **Title:** Dr.

CustomerID	Column Family: Contact Info
001	**Phone number:** 555-0100 **Email:** someone@example.com
002	**Email:** vilanova@contoso.com
003	**Phone number:** 555-0120

Column-family databases – Workload

- Most column-family databases perform write operations extremely quickly.
- Update and delete operations are rare.
- Designed to provide high throughput and low-latency access.
- Supports easy query access to a particular set of fields within a much larger record.
- Massively scalable.

Column-family databases – Data type

- Data is stored in tables consisting of a key column and one or more column families.
- Specific columns can vary by individual rows.
- Individual cells are accessed via get and put commands
- Multiple rows are returned using a scan command.

Column-family databases – Examples

- Recommendations
- Personalization
- Sensor data
- Telemetry
- Messaging
- Social media analytics
- Web analytics
- Activity monitoring
- Weather and other time-series data

Other technologies

- Data Analytics
  - SQL Data Warehouse
  - Azure Data Lake
- Search Engine Databases
  - Azure Search
- Time Series Databases
  - Time Series Insights
- Object storage
  - Blob Storage
- Shared files
  - File Storage

Design principles

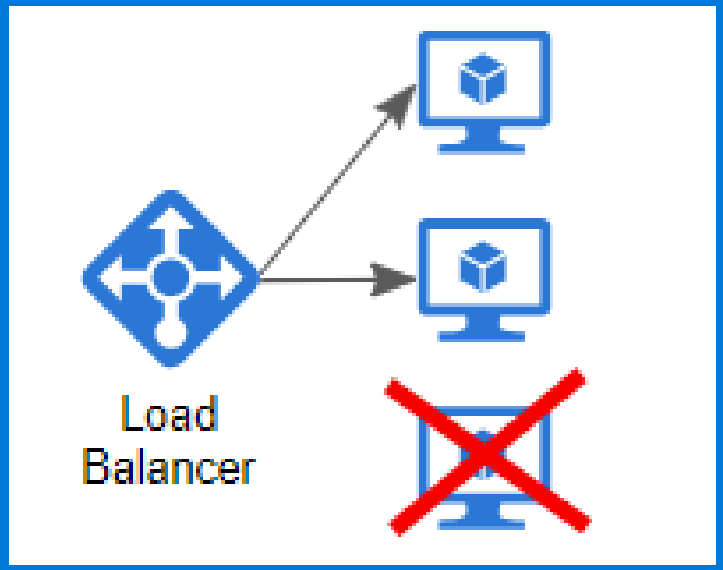
Design for self healing

- Design your application to be self healing when failures occur

- Recommendations
  - Retry failed operations
  - Protect failing remote services
  - Isolate critical resources
  - Perform load leveling
  - Fail over
  - Compensate failed transactions
  - Checkpoint long-running transactions
  - Degrade gracefully
  - Throttle clients
  - Block bad actors
  - Use leader election
  - Test with Fault injection
  - Embrace chaos engineering

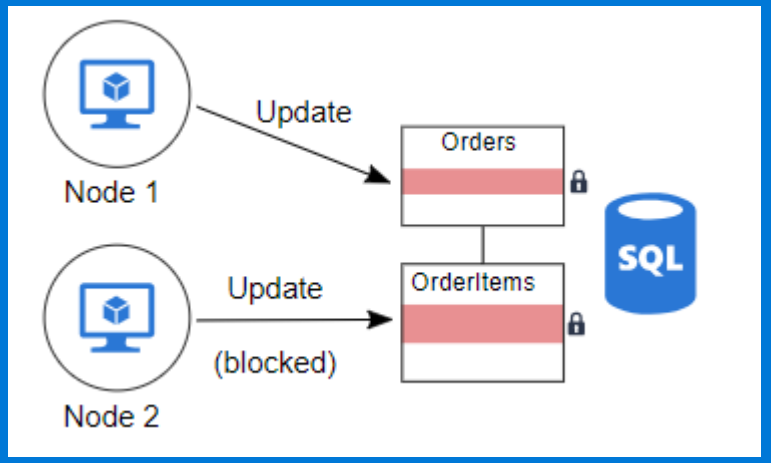
Make all things redundant

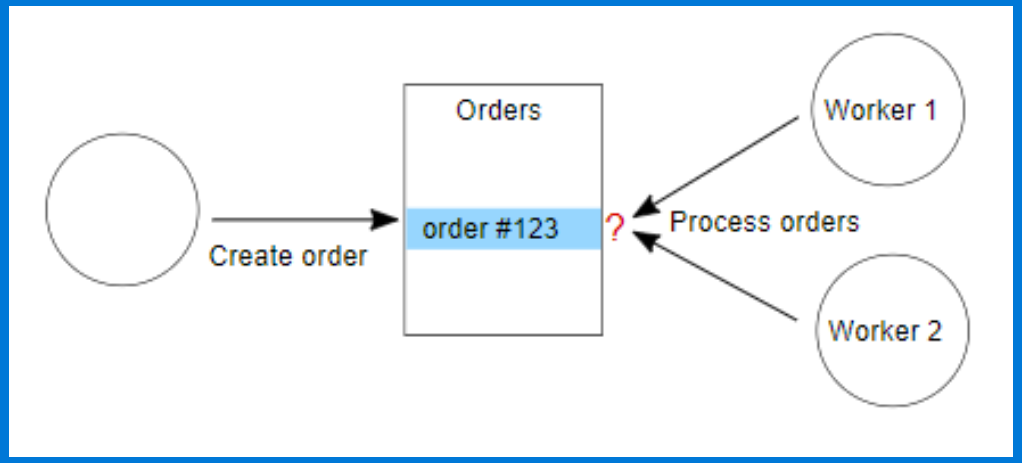
- Build redundancy into your application to avoid having single points of failure
- Recommendations
  - Consider business requirements
  - Place VMs behind load balancer
  - Replicate databases
  - Enable geo-replication
  - Partition for availability
  - Deploy to more than one región
  - Synchronize front and backend failover
  - Use automatic failover but manual failback



Minimize coordination

- Minimize coordination between application services to achieve scalability





Minimize coordination – Recommendations

- Embrace eventual consistency
- Use domain events to synchronize state
- Consider patterns such as CQRS and event sourcing
- Partition data
- Design idempotent operations
- Use asynchronous parallel processing
- Use optimistic concurrency when possible
- Consider MapReduce or other parallel, distributed algorithms
- Use leader election for coordination

Design to scale out

- Design your application so that it can scale horizontally
- Recommendations
  - Avoid instance stickiness
  - Identify bottlenecks
  - Decompose workloads by scalability requirements
  - Offload resource-intensive tasks
  - Use built-in autoscaling features
  - Design for scale in

Partition around limits

- Use partitioning to work around database, network, and compute limits
- Horizontal partitioning, subset of the total dataset.
- Vertical partitioing, subset of the fields for the ítems in the data store
- Functional partitioning, according how data is used by each bounded context in the system.

Partitioning around limits – Recommendations

- Partition different parts of the application
- Design the partition key to avoid hot spots.
- Partition around Azure subscription and service limits.
- Partition at different levels.

Design for operations

- Design an application so that the operations team has the tools they need
  - Deployment
  - Monitoring
  - Escalation
  - Incident response
  - Security auditing

Design for operations - Recommendations

- Make all things observable
- Instrument for monitoring
- Instrument for root cause analysis
- Use distributed tracing
- Standarize logs and metrics
- Automate management tasks
- Treat configuration as code

Use managed services

- When possible, use PaaS rather than IaaS

Instead of running...	Consider using...
Active Directory	Azure Active Directory Domain Services
Elasticsearch	Azure Search
Hadoop	HDInsight
IIS	App Service
MongoDB	Cosmos DB
Redis	Azure Redis Cache
SQL Server	Azure SQL Database

Use the best data store for the job

- Pick the storage technology that is the best fit for your data and how it will be used
- Recommendations
  - Don’t use a relational database for everything
  - Embrace polyglot persistence
  - Consider the type of data
  - Prefer availability over strong consistency
  - Consider the skill set of the dev team
  - Use compensating transactions
  - Look at bounded contexts

Design for evolution

- An evolutionary design is key for continuous innovation
- Recommendations
  - Enforce high cohesion and loose coupling
  - Encapsulate domain knowledge
  - Use asynchronous messaging
  - Don’t build domain knowledge into a gateway
  - Expose open interfaces
  - Design and test against service contracts
  - Abstract infrastructure away from domain logic
  - Offload cross-cutting concerns to a separate service
  - Deploy service independently

Build for the needs of the business

- Every design decision must be justified by a business requirement
- Recommendations
  - Define business objectives
  - Document service level agreements (SLA) and service level objectives (SLO)
  - Model the application around the business domain
  - Capture both functional and nonfunctional requirements
  - Decompose by workload
  - Plan for growth
  - Manage costs

Pillars of software quality

Pillars of software quality

- Scalability
- Availability
- Resiliency
- Management
- Security

Scalability

- The ability of a system to handle increased load
- Vertical (Scale up) or horizontal (scale out)
- Just adding more instances could just move to bottleneck to somewhere else.
- Always conduct performance and load testing to find this bottlenecks.
- Use the Scalability checklist to review your design from a scalability standpoint.

Availability

- Proportion of time the system is functional and working.
- Cloud app should have SLO
- Azure offers SLAs to describe commitments for uptime and connectivity.
- Use the Availability checklist to review your design from an availability standpoint.

Availability

% Uptime	Downtime per week	Downtime per month	Downtime per year
99%	1.68 hours	7.2 hours	3.65 days
99.9%	10 minutes	43.2 minutes	8.76 hours
99.95%	5 minutes	21.6 minutes	4.38 hours
99.99%	1 minute	4.32 minutes	52.56 minutes
99.999%	6 seconds	26 seconds	5.26 minutes

Resiliency

- Is the ability of the system to recover from failures and continue to function.
- Cloud apps must be designed to expect occasional failures and recover from them.
- How much downtime is acceptable? How much will potential downtime cost your business? How much should you invest in making the application highly available?
- Use the Resiliency checklist to review your design from a resiliency standpoint.

Management and DevOps

- Covers the operations processes that keep and application running in production.
- Deployments must be reliable and predictable.
- Monitoring and diagnostics are crucial.
- Use the DevOps checklist to review your design from a management and DevOps standpoint.

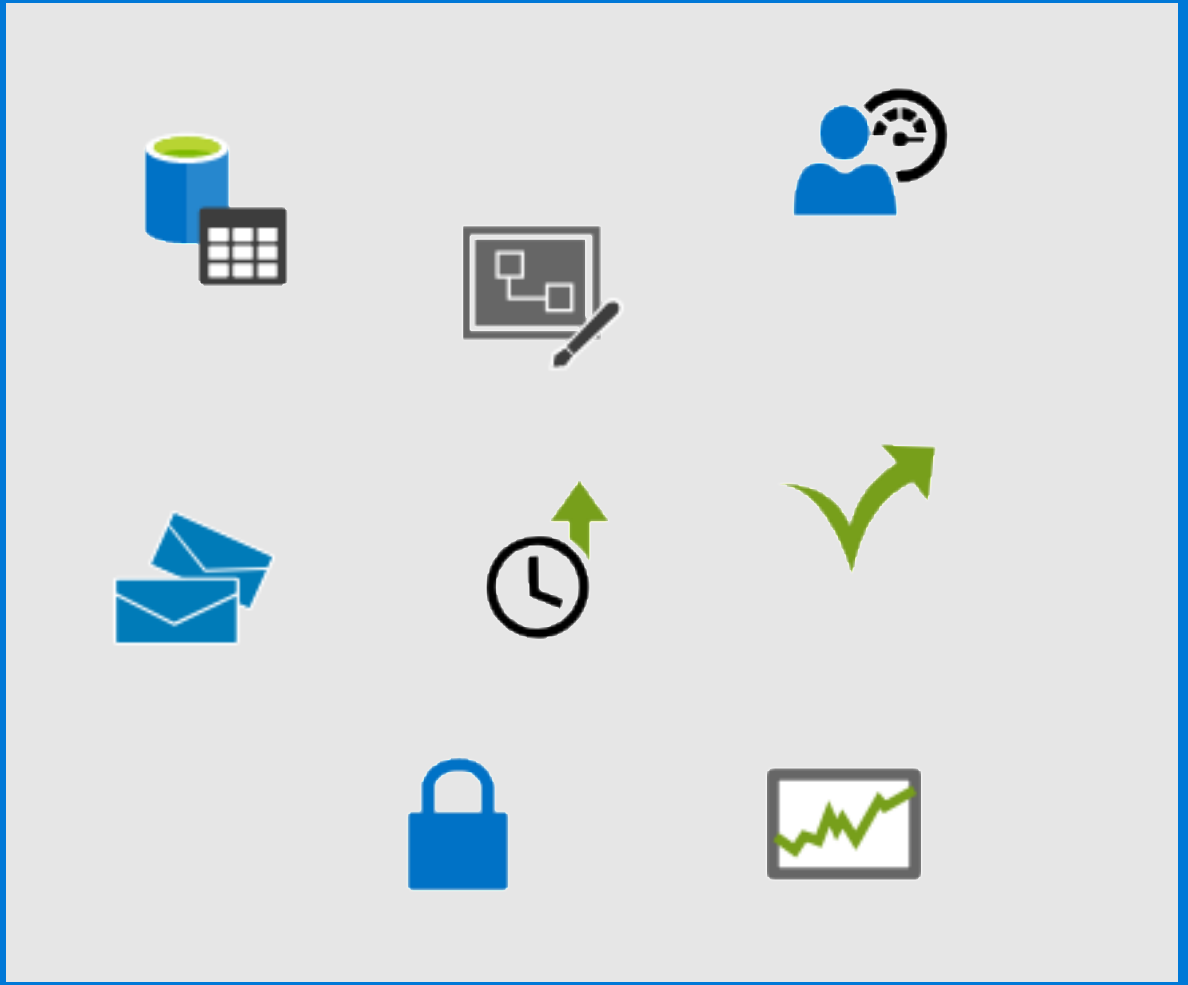
Security

- Identity management
- Protecting your infrastructure
- Application security
- Data sovereignty and encryption

Cloud Design Patterns

Challenges in Cloud development

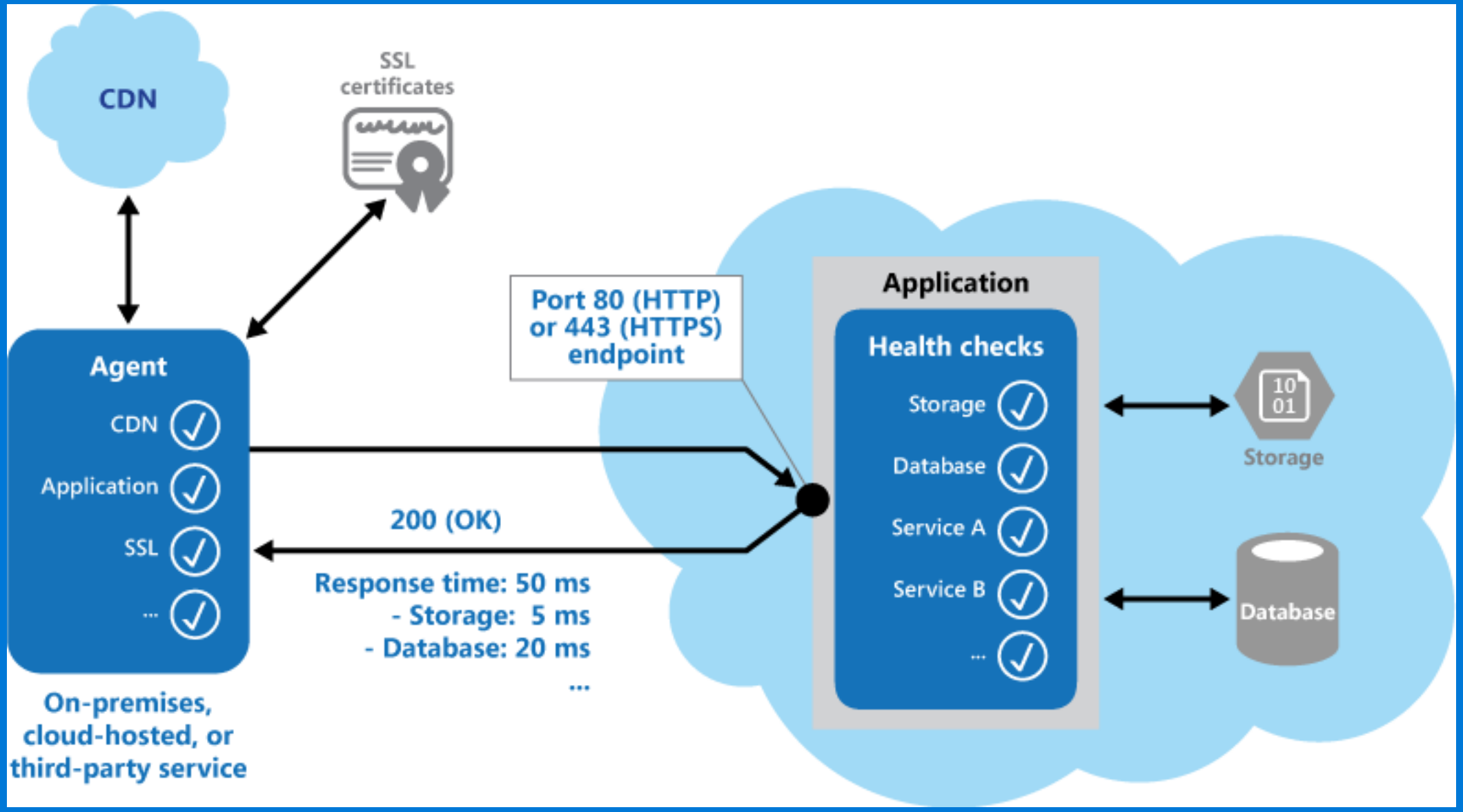
- Availability
- Data Management
- Design and Implementation
- Messaging
- Management and monitoring
- Performance and scalability
- Resiliency
- Security



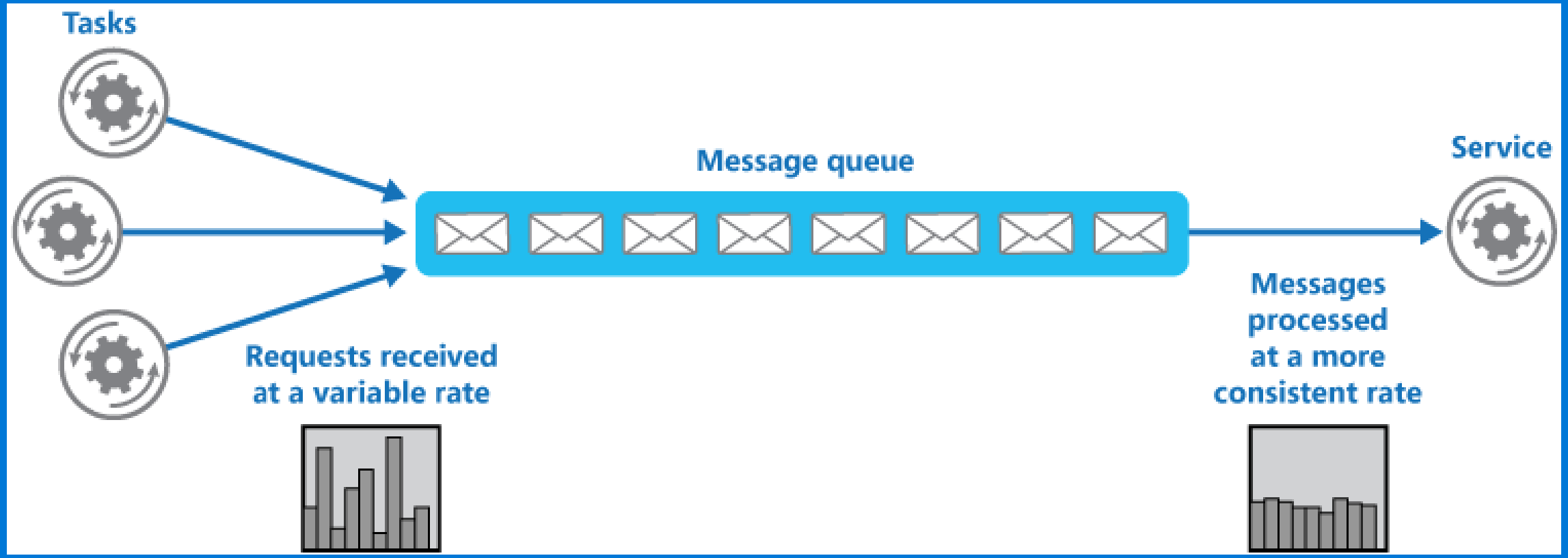
Availability patterns

- Health endpoint monitoring: Implement functional checks in an application that external tools can access through exposed endpoints at regular intervals.
- Queue-Based load monitoring: Use a queue that acts as a buffer between a task and a service that it invokes in order to smooth intermittent heavy loads
- Throttling: Control the consumption of resources used by an instance of an application, an individual tenant, or an entire service.

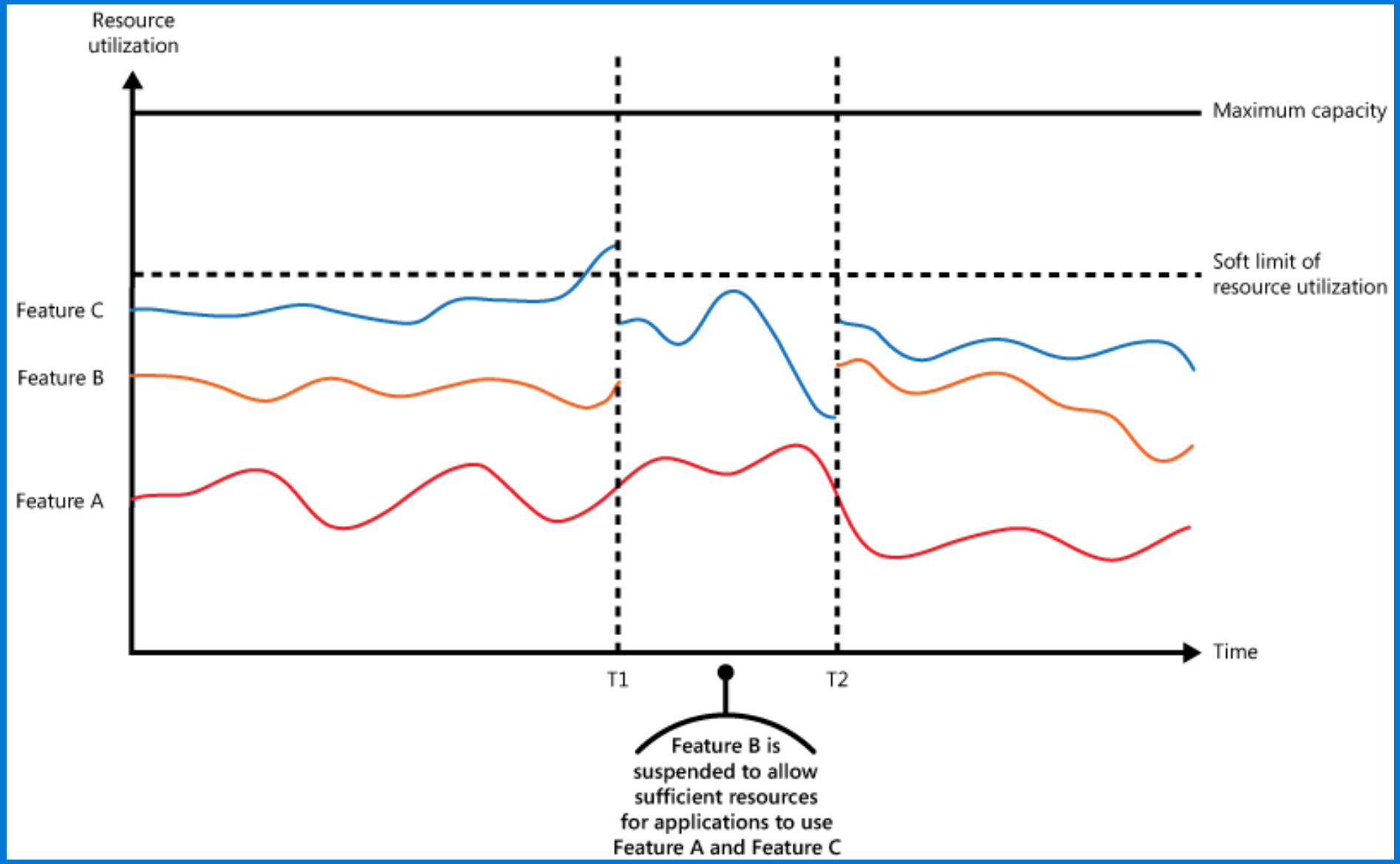
Health endpoint monitoring pattern



Queue-based load levelling



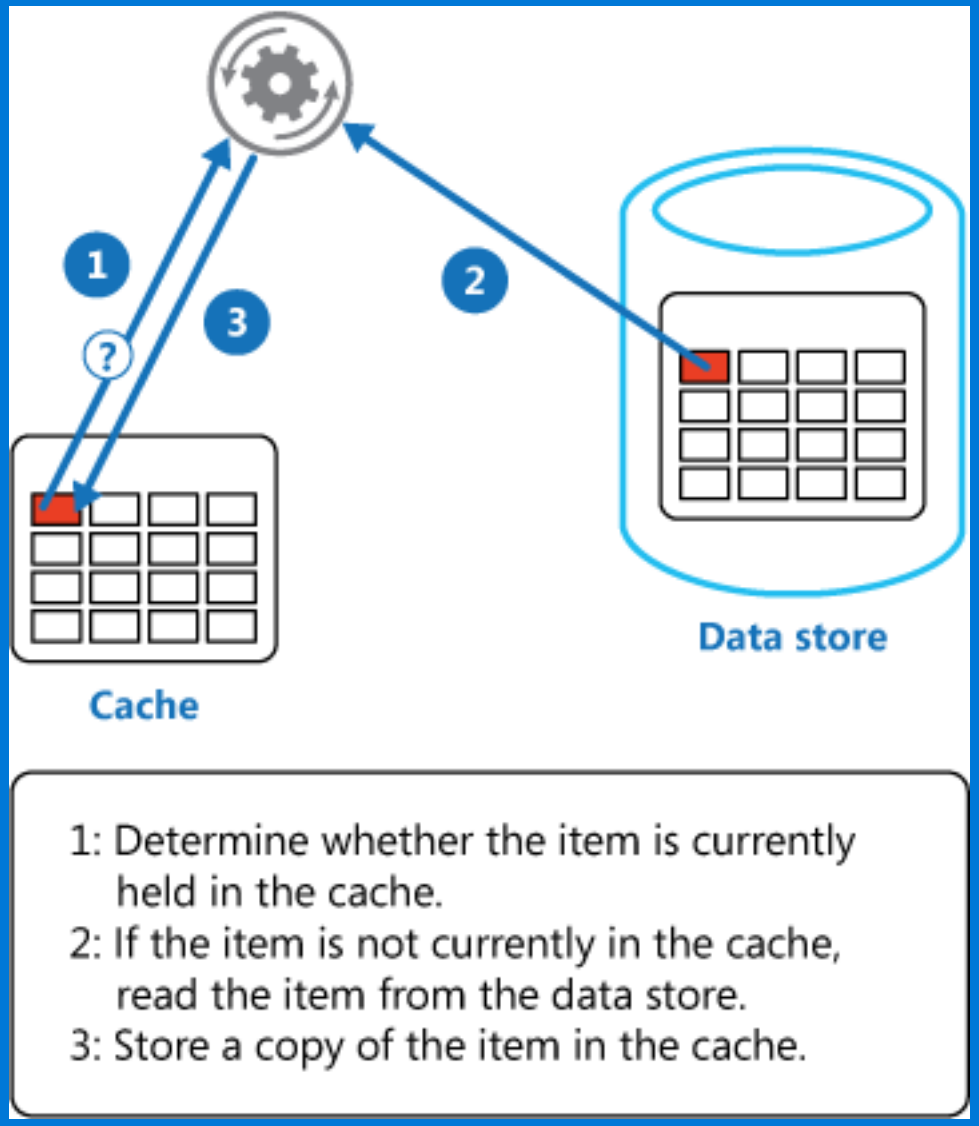
Throttling


Resource utilization
Maximum capacity
Soft limit of resource utilization
Feature C
Feature B
Feature A
Time
T1
T2
Feature B is suspended to allow sufficient resources for applications to use Feature A and Feature C

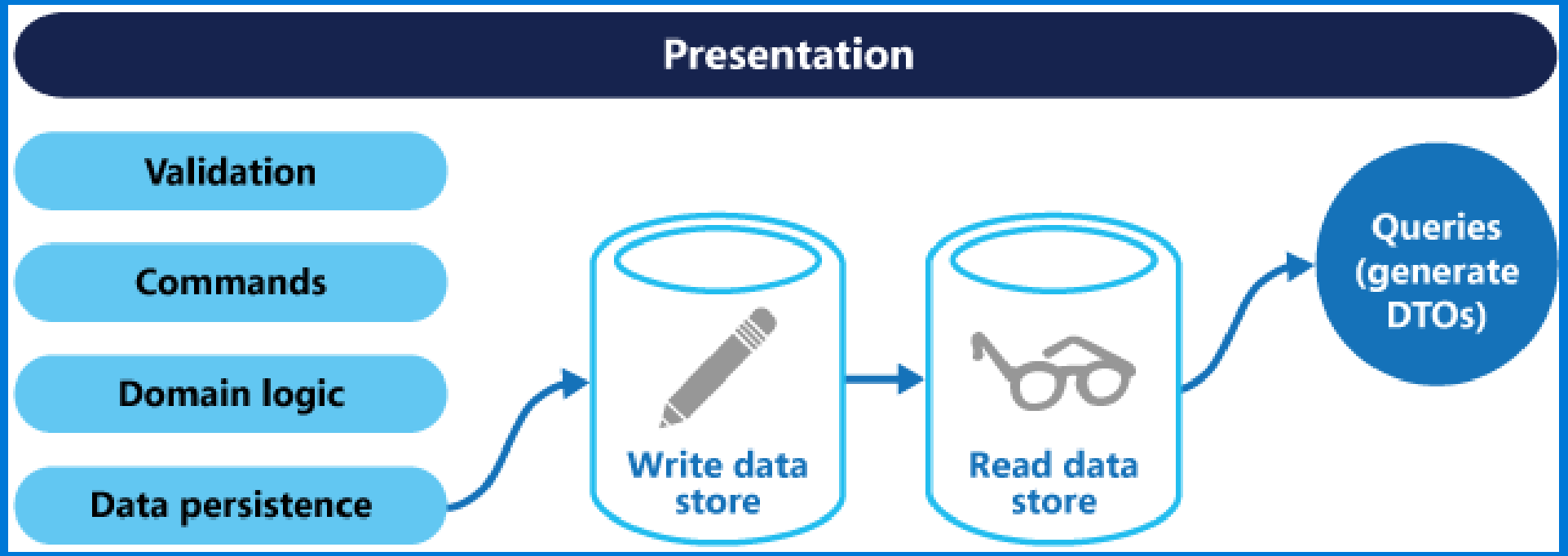
Data Management patterns

Pattern	Summary
Cache-Aside	Load data on demand into a cache from a data store
CQRS	Segregate operations that read data from operations that update data by using separate interfaces.
Event Sourcing	Use an append-only store to record the full series of events that describe actions taken on data in a domain.
Index Table	Create indexes over the fields in data stores that are frequently referenced by queries.
Materialized View	Generate prepopulated views over the data in one or more data stores when the data isn't ideally formatted for required query operations.
Sharding	Divide a data store into a set of horizontal partitions or shards.
Static Content Hosting	Deploy static content to a cloud-based storage service that can deliver them directly to the client.
Valet Key	Use a token or key that provides clients with restricted direct access to a specific resource or service.

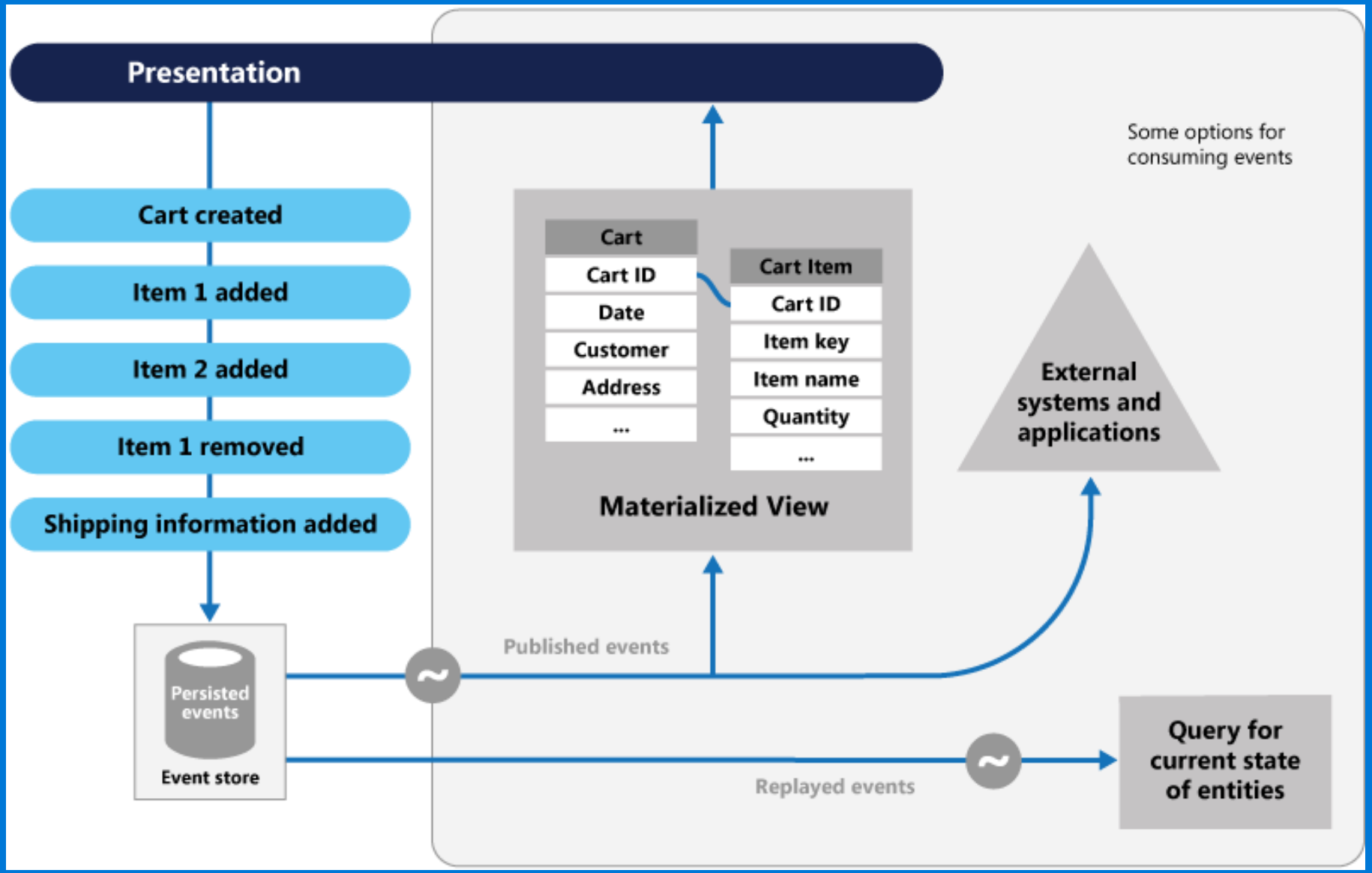
Cache Aside


1
2
3
?
Data store
Cache
1: Determine whether the item is currently held in the cache.
2: If the item is not currently in the cache, read the item from the data store.
3: Store a copy of the item in the cache.

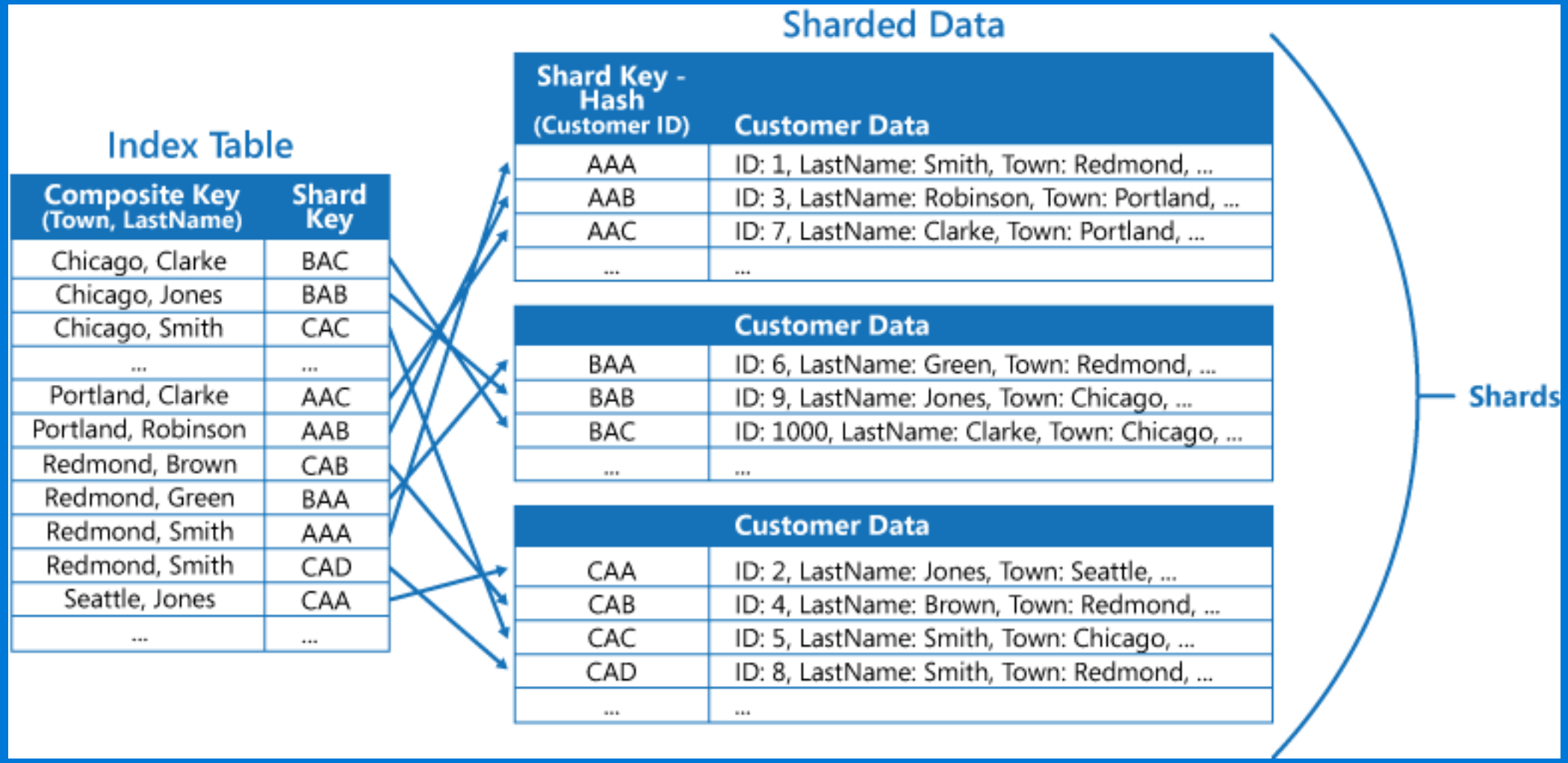
CQRS


Presentation
Validation
Commands
Domain logic
Data persistence
Write data store
Read data store
Queries (generate DTOs)

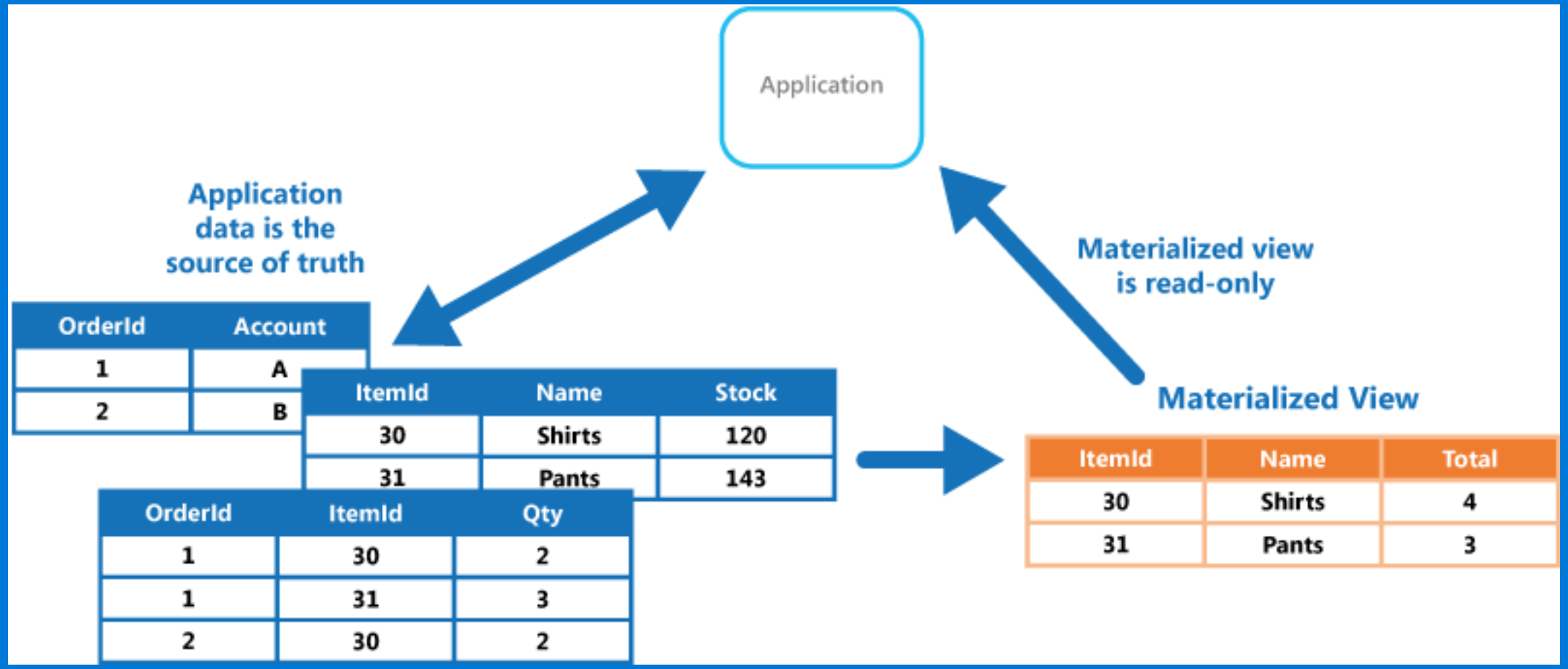
Event Sourcing



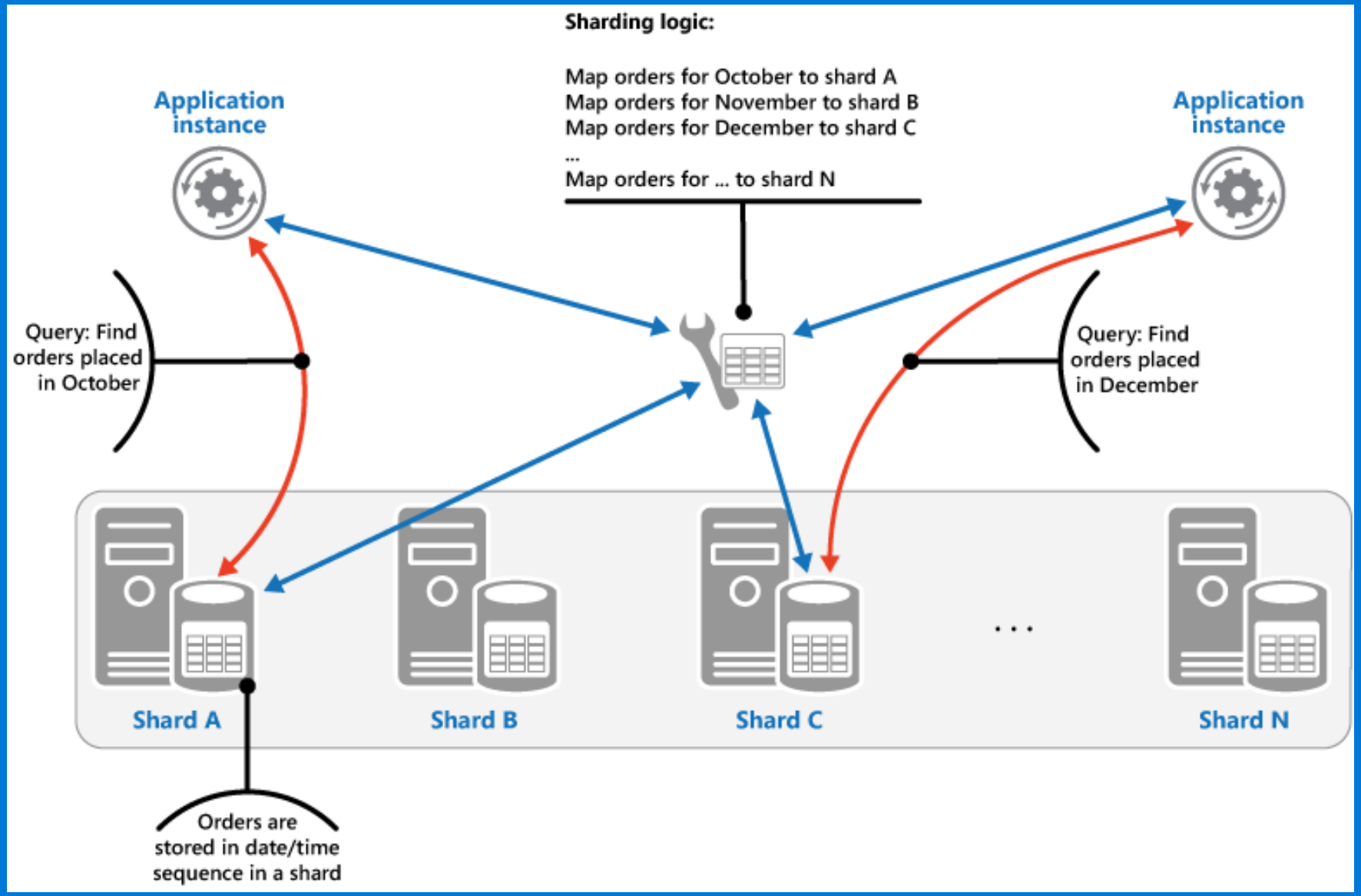
Index Table


Sharded Data
Index Table
Composite Key (Town, LastName)
Shard Key
Chicago, Clarke
BAC
Chicago, Jones
BAB
Chicago, Smith
CAC
...
...
Portland, Clarke
AAC
Portland, Robinson
AAB
Redmond, Brown
CAB
Redmond, Green
BAA
Redmond, Smith
AAA
Redmond, Smith
CAD
Seattle, Jones
CAA
...
...
Shard Key - Hash (Customer ID)
Customer Data
AAA
ID: 1, LastName: Smith, Town: Redmond, ...
AAB
ID: 3, LastName: Robinson, Town: Portland, ...
AAC
ID: 7, LastName: Clarke, Town: Portland, ...
...
...
Customer Data
BAA
ID: 6, LastName: Green, Town: Redmond, ...
BAB
ID: 9, LastName: Jones, Town: Chicago, ...
BAC
ID: 1000, LastName: Clarke, Town: Chicago, ...
...
...
Customer Data
CAA
ID: 2, LastName: Jones, Town: Seattle, ...
CAB
ID: 4, LastName: Brown, Town: Redmond, ...
CAC
ID: 5, LastName: Smith, Town: Chicago, ...
CAD
ID: 8, LastName: Smith, Town: Redmond, ...
...
...
Shards

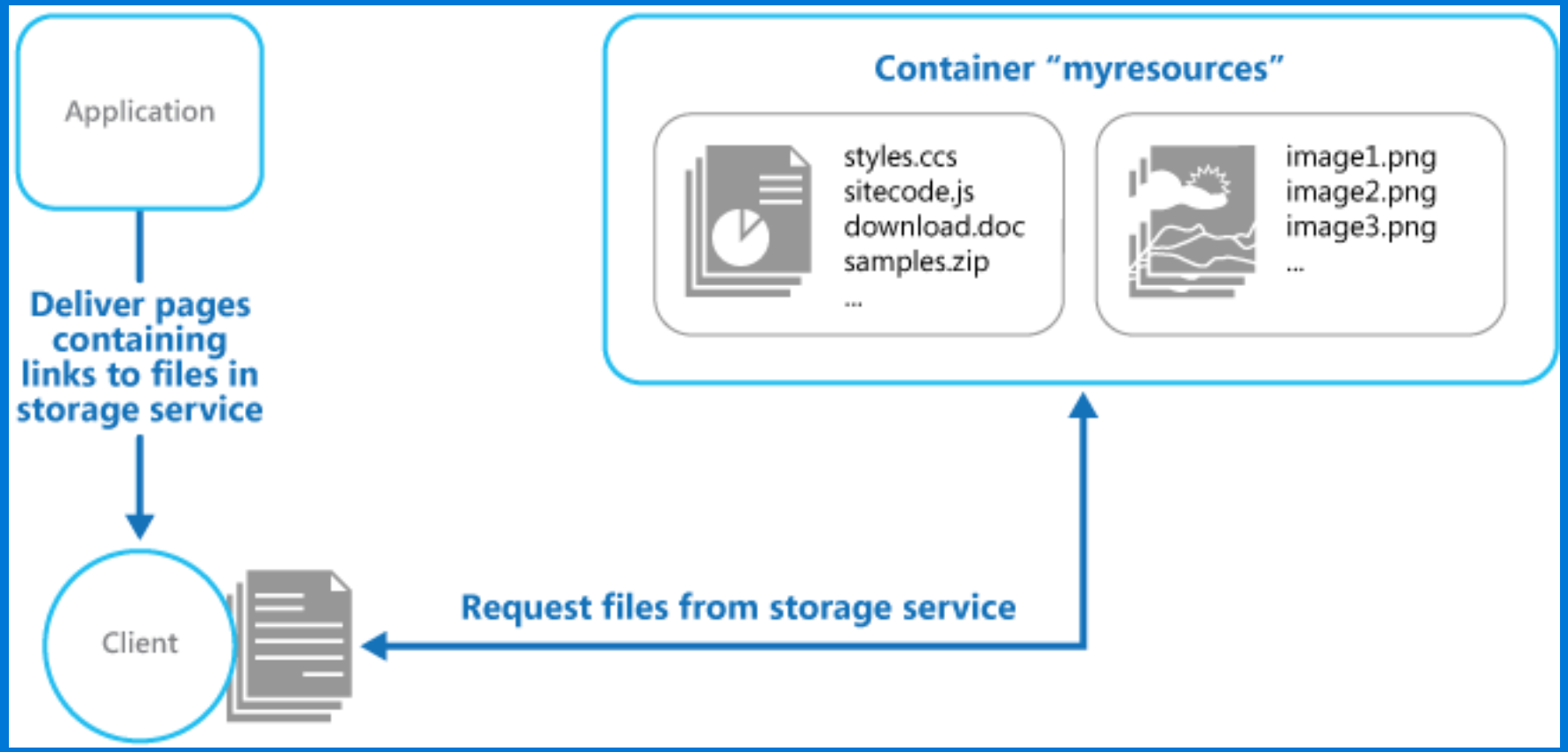
Materialized view



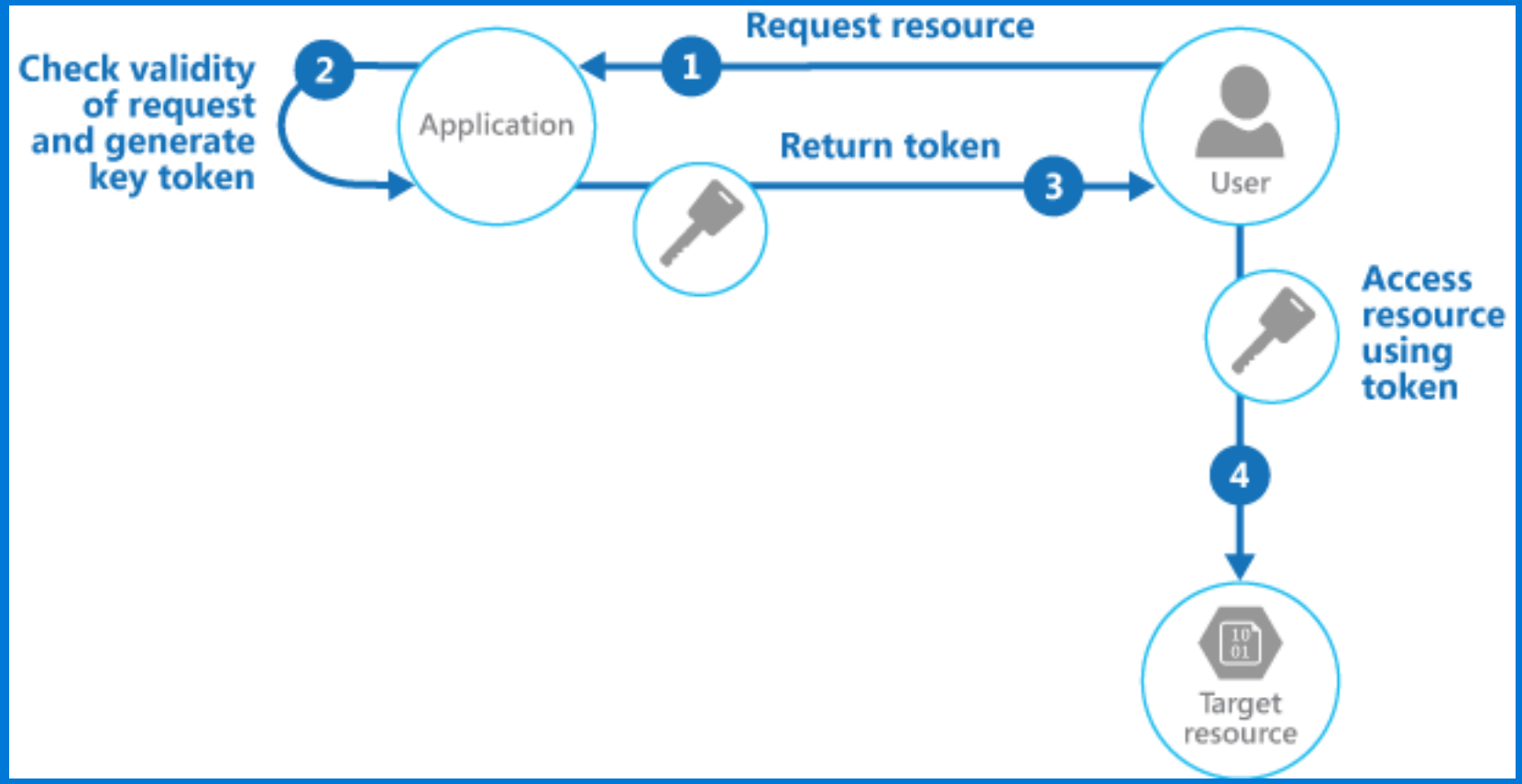
Sharding


Sharding logic:
Map orders for October to shard A
Map orders for November to shard B
Map orders for December to shard C
...
Map orders for ... to shard N
Application instance
Application instance
Query: Find orders placed in October
Query: Find orders placed in December
Shard A
Shard B
Shard C
Shard N
Orders are stored in date/time sequence in a shard

Static Content Hosting



Valet key


Request resource
Check validity of request and generate key token
2
1
Application
Return token
3
User
Access resource using token
4
Target resource

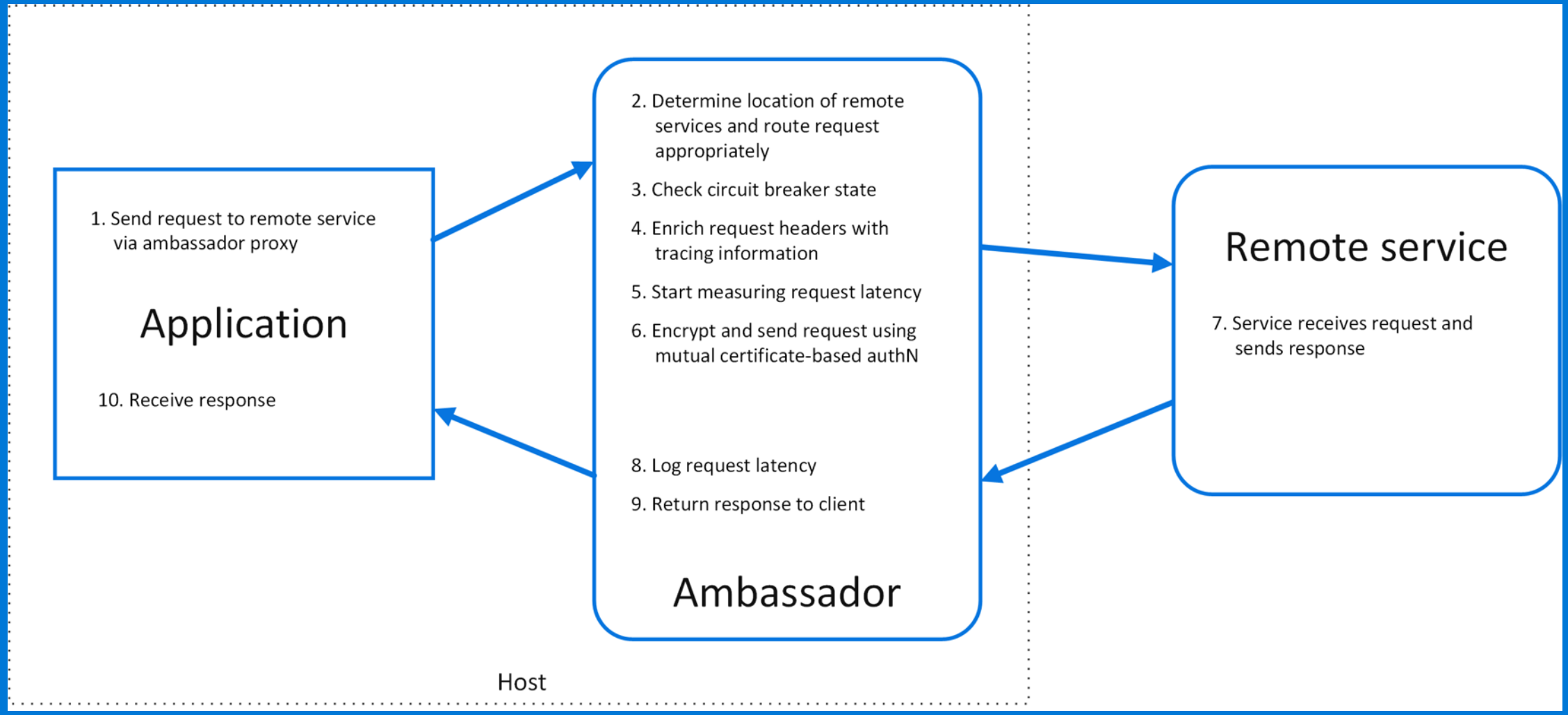
Design and implementation

Pattern	Summary
Ambassador	Create helper services that send network requests on behalf of a consumer service or application.
Anti-Corruption Layer	Implement a façade or adapter layer between a modern application and a legacy system.
Backends for Frontends	Create separate backend services to be consumed by specific frontend applications or interfaces.
Compute Resource Consolidation	Consolidate multiple tasks or operations into a single computational unit
External Configuration Store	Move configuration information out of the application deployment package to a centralized location.
Gateway Aggregation	Use a gateway to aggregate multiple individual requests into a single request.
Gateway Offloading	Offload shared or specialized service functionality to a gateway proxy.
Gateway Routing	Route requests to multiple services using a single endpoint.

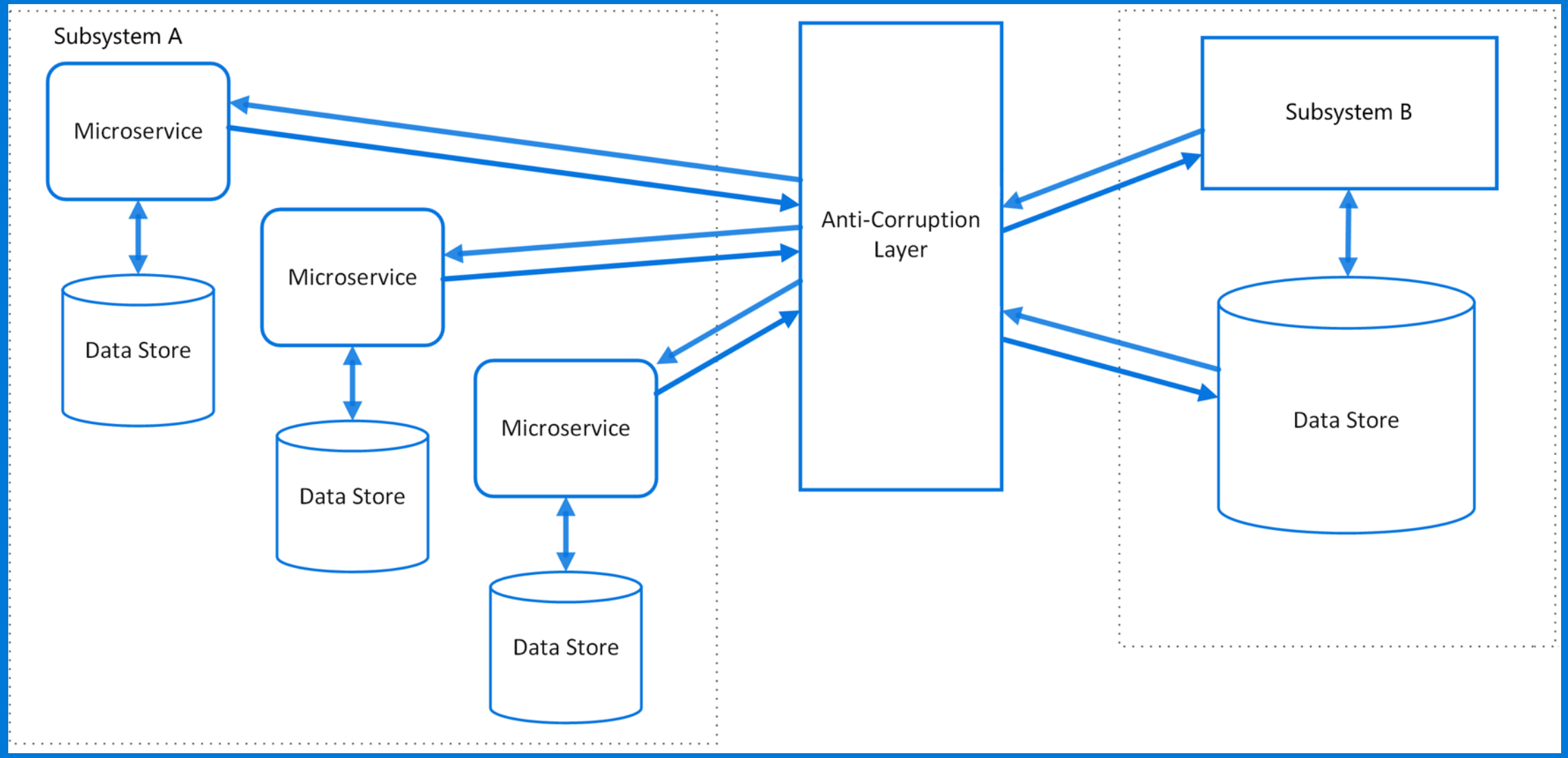
Design and Implementation

Leader Election	Coordinate the actions performed by a collection of collaborating task instances in a distributed application by electing one instance as the leader that assumes responsibility for managing the other instances.
Pipes and Filters	Break down a task that performs complex processing into a series of separate elements that can be reused.
Sidecar	Deploy components of an application into a separate process or container to provide isolation and encapsulation.
Strangler	Incrementally migrate a legacy system by gradually replacing specific pieces of functionality with new applications and services.

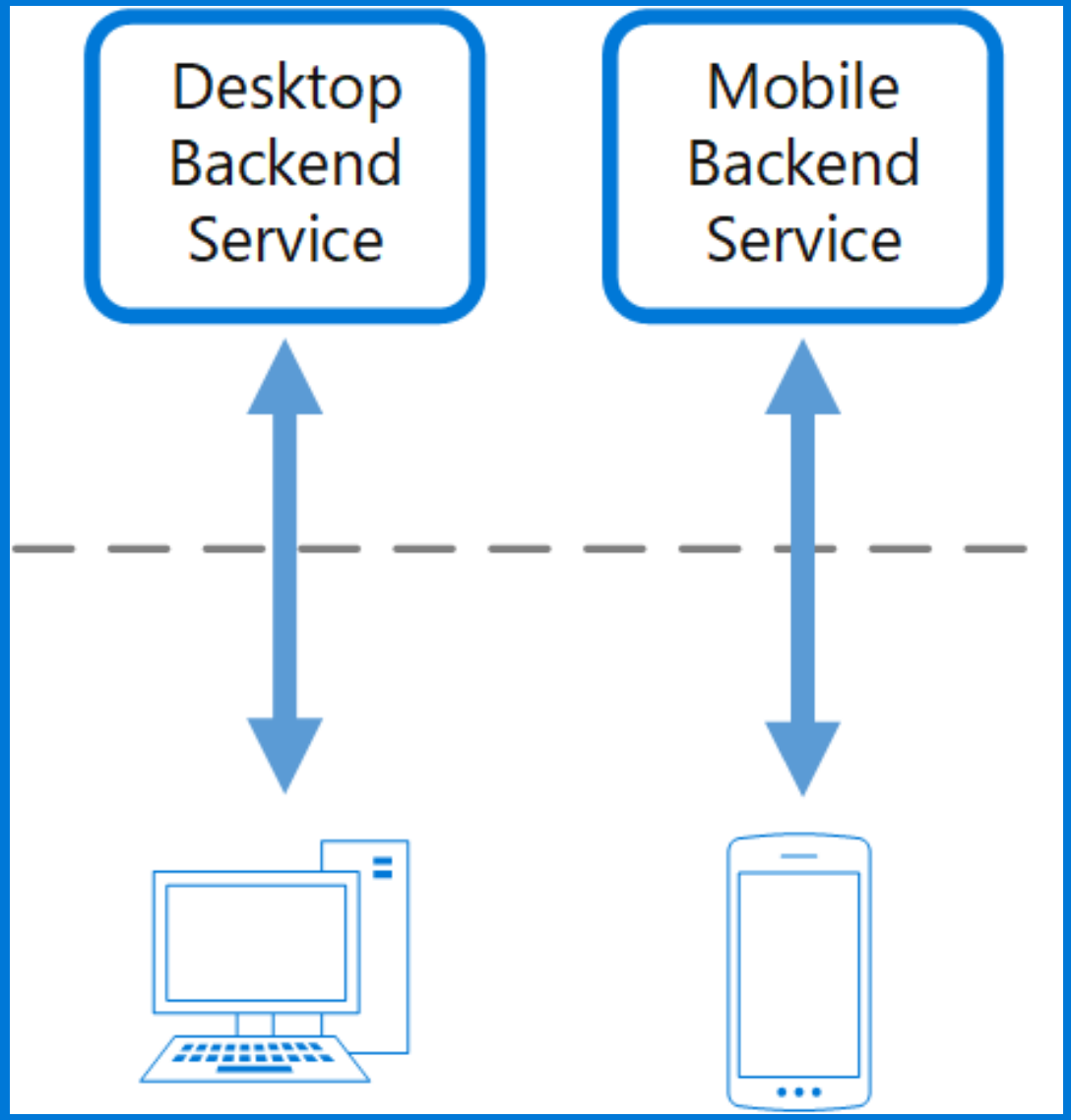
Ambassador


1. Send request to remote service via ambassador proxy
Application
10. Receive response
2. Determine location of remote services and route request appropriately
3. Check circuit breaker state
4. Enrich request headers with tracing information
5. Start measuring request latency
6. Encrypt and send request using mutual certificate-based authN
8. Log request latency
9. Return response to client
Ambassador
Remote service
7. Service receives request and sends response
Host

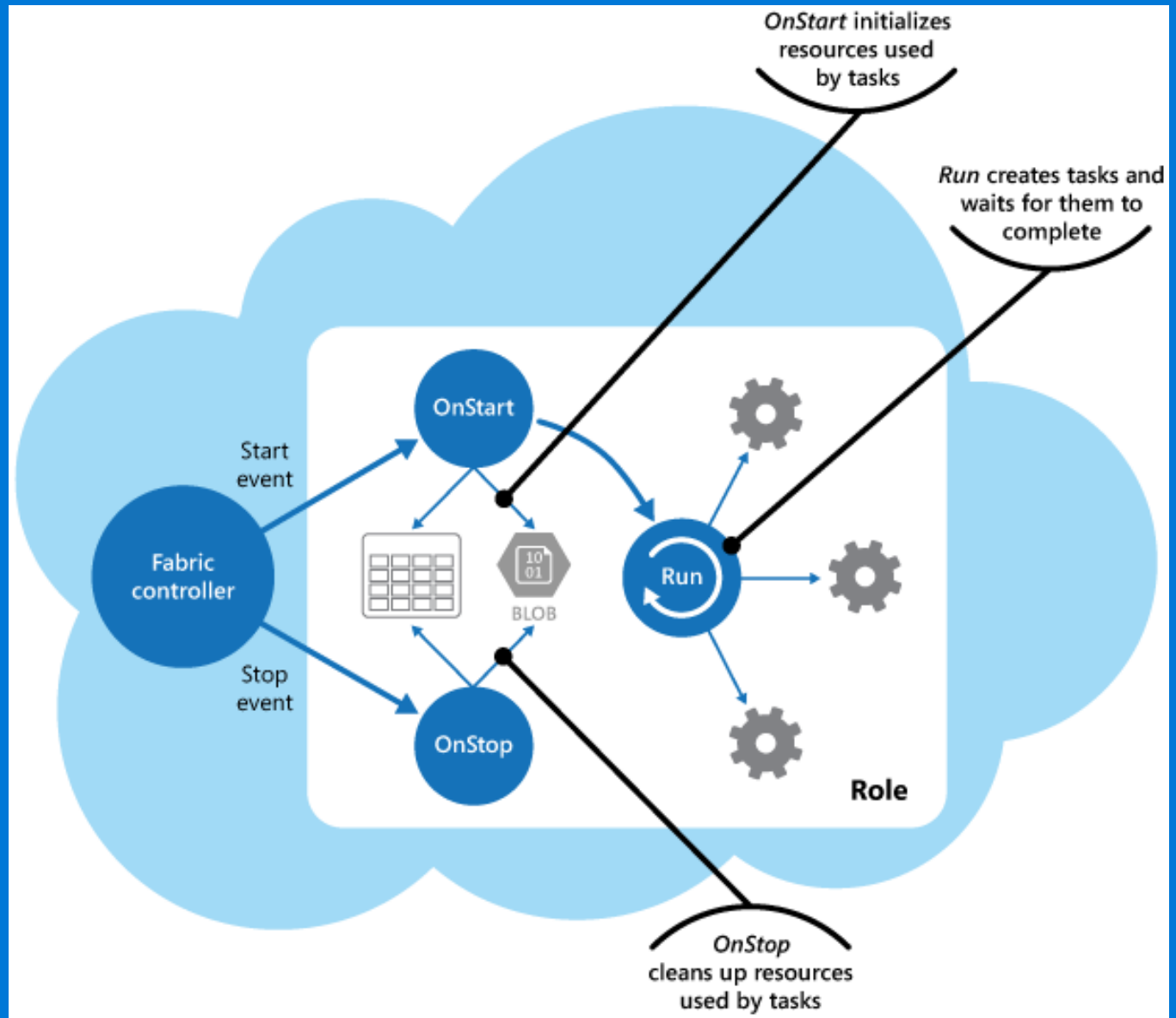
Anti-corruption layer pattern



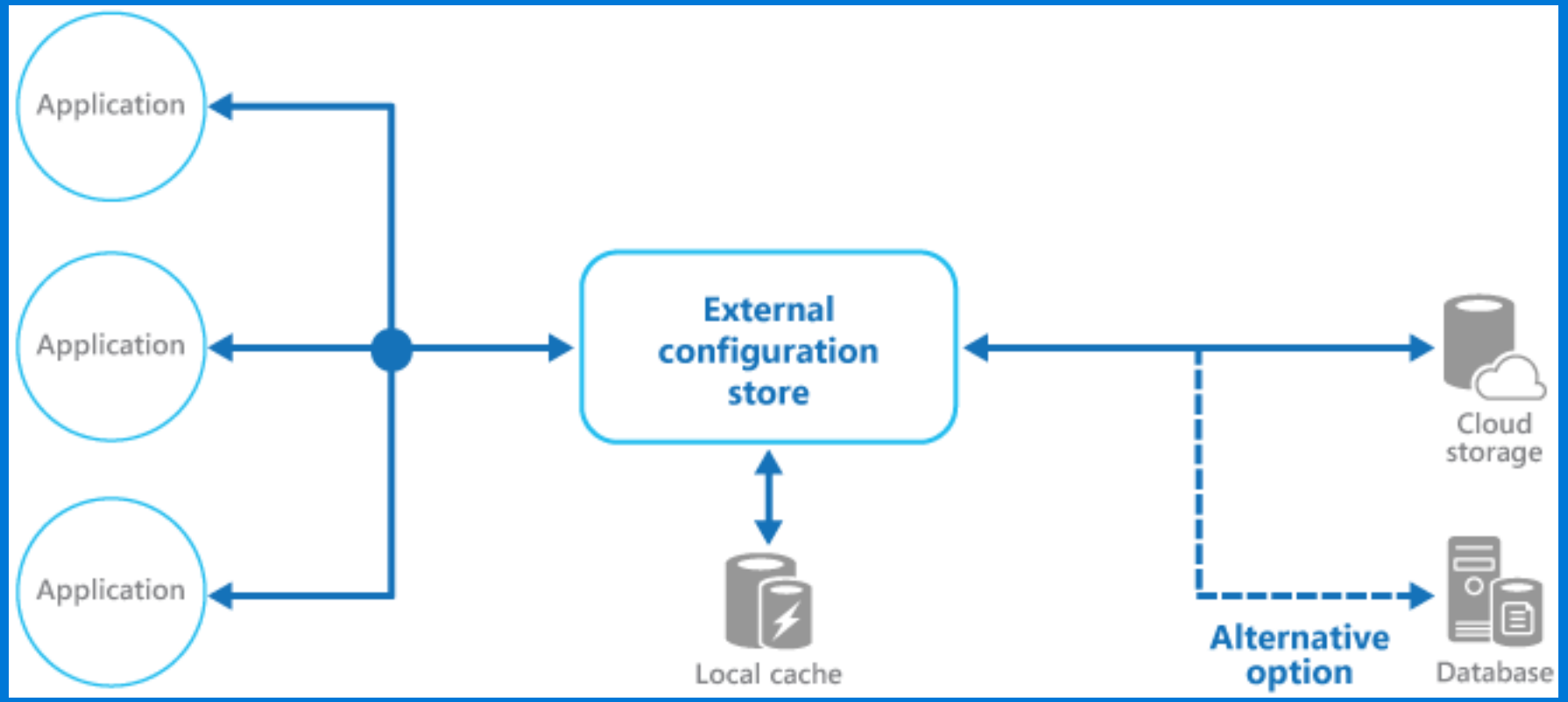
Backends for Frontends pattern



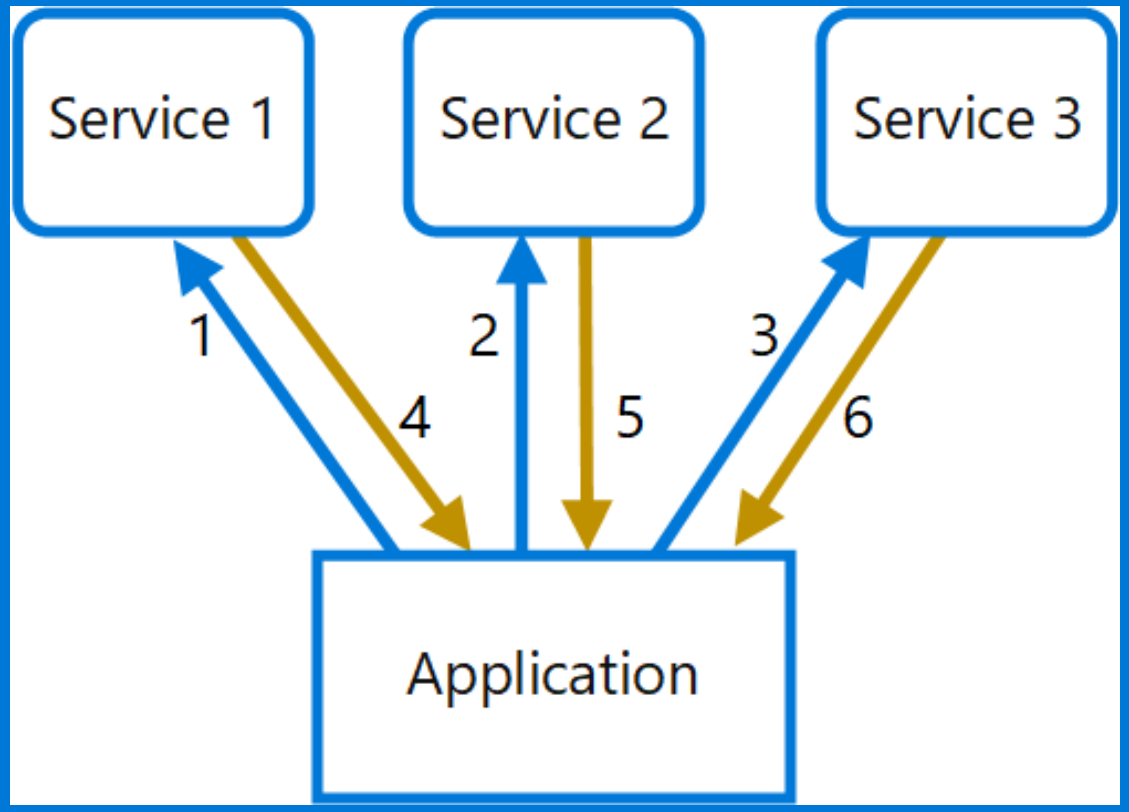
Compute resource consolidation

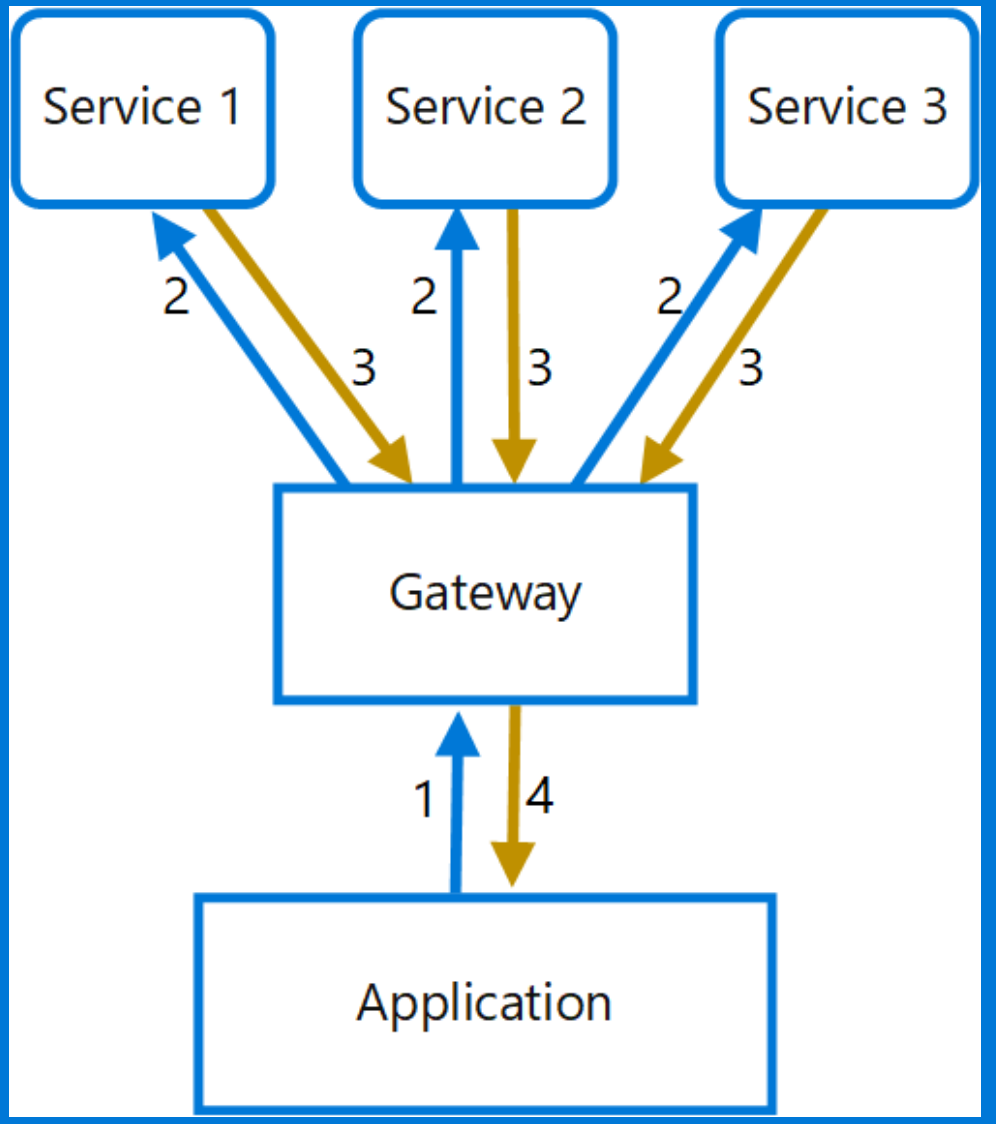


External configuration store

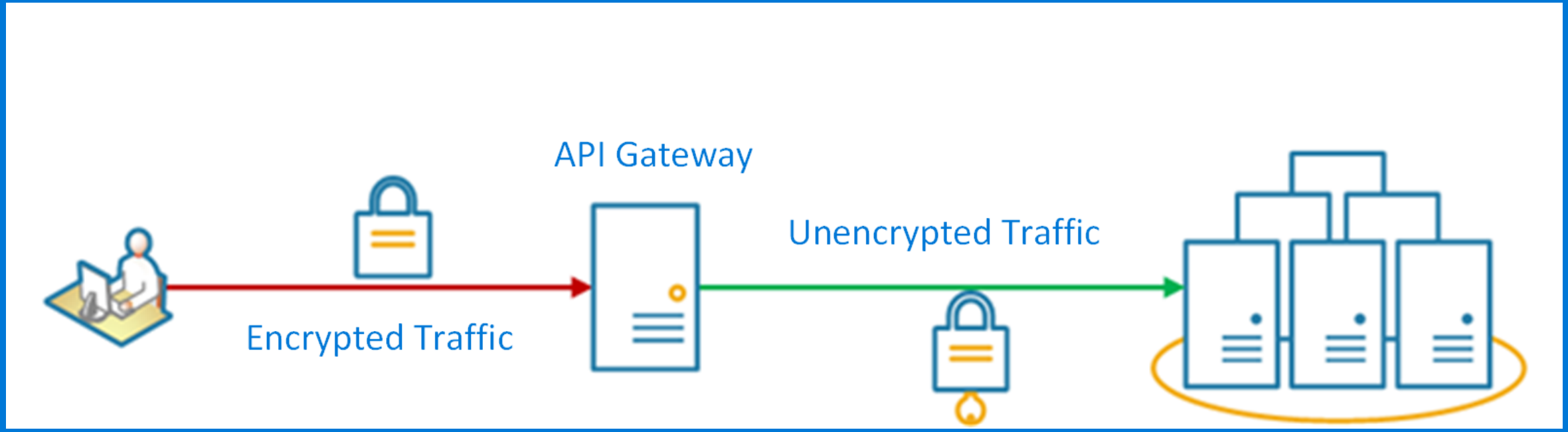


Gateway aggregation

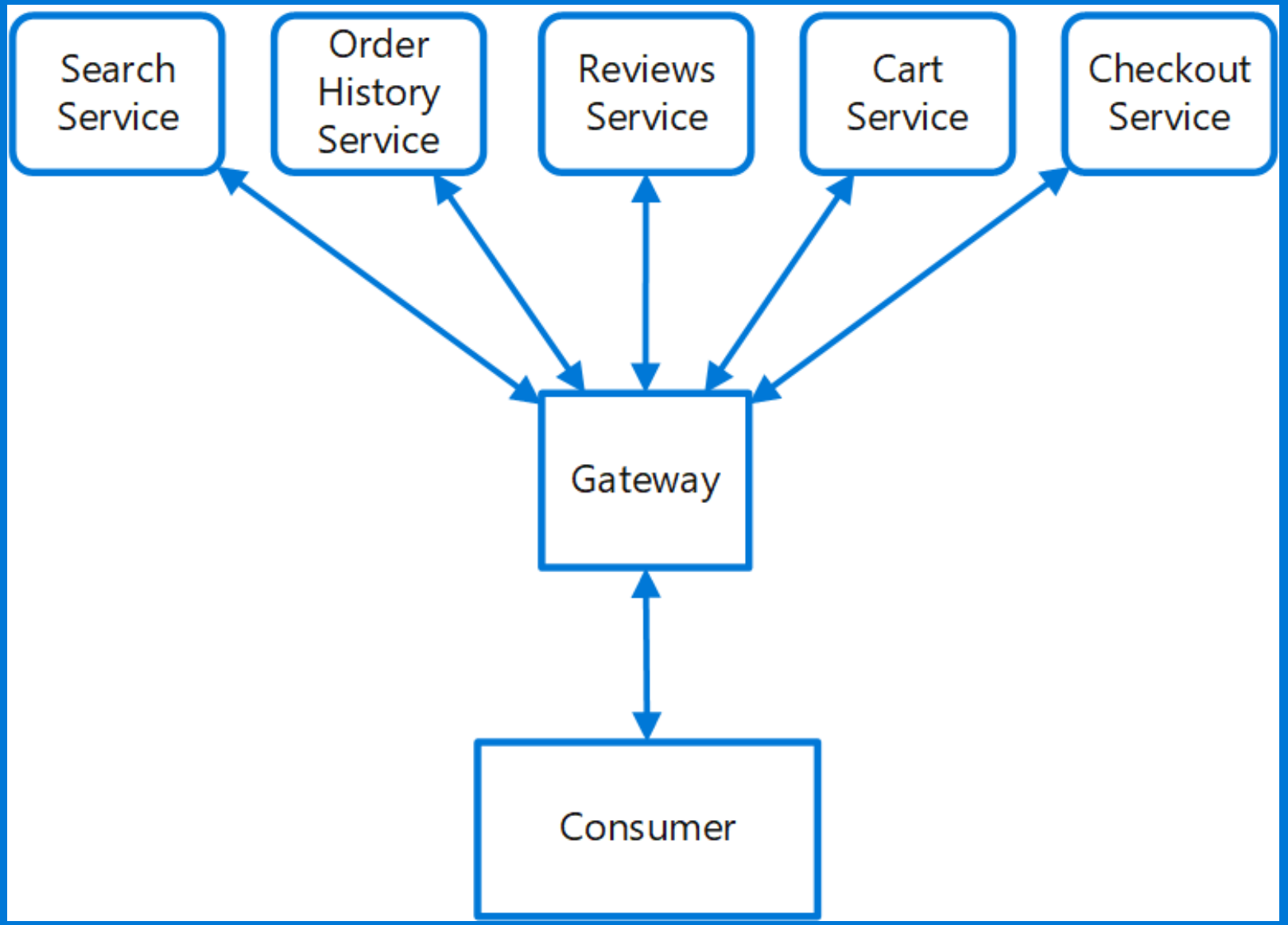




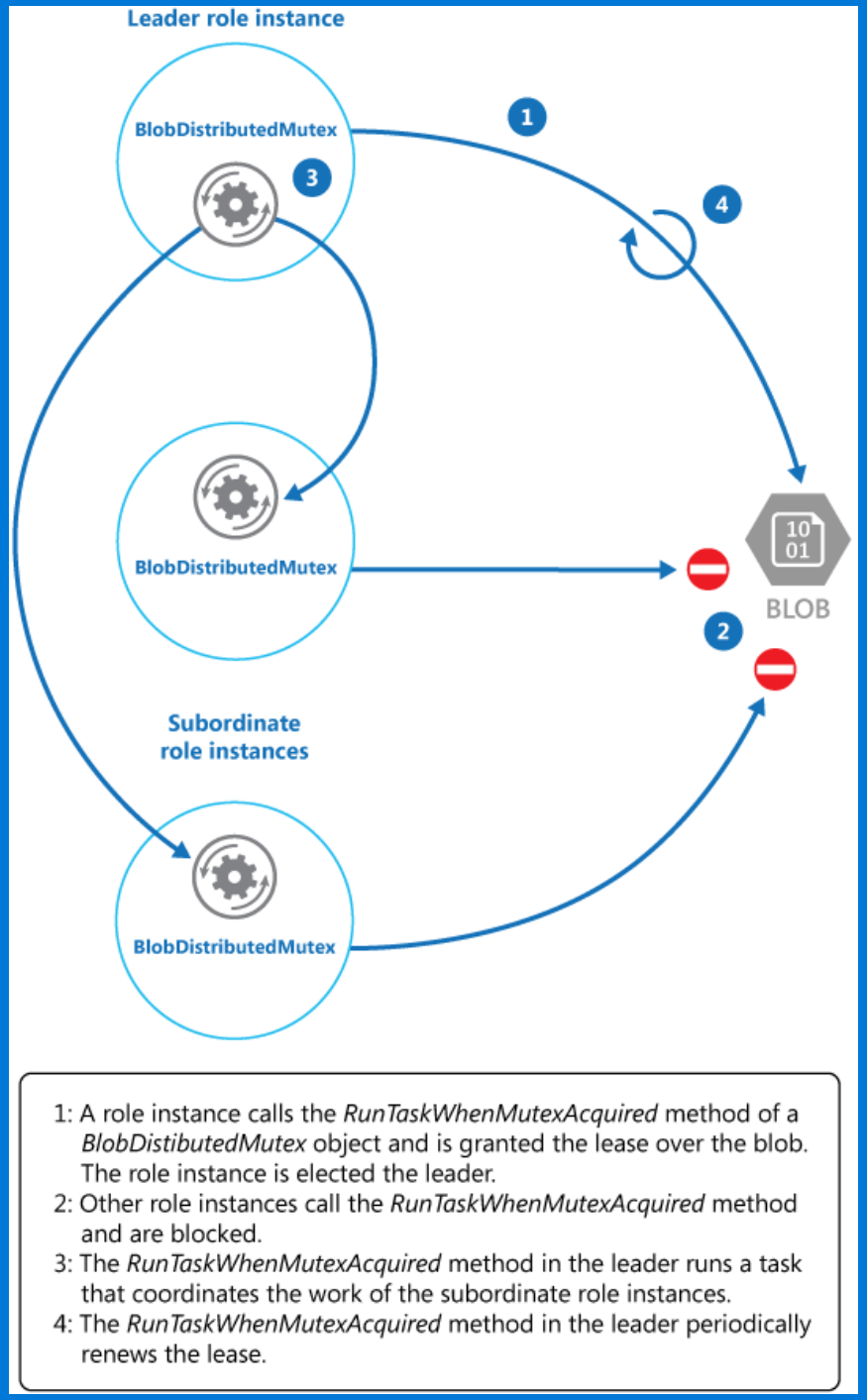
Gateway offloading



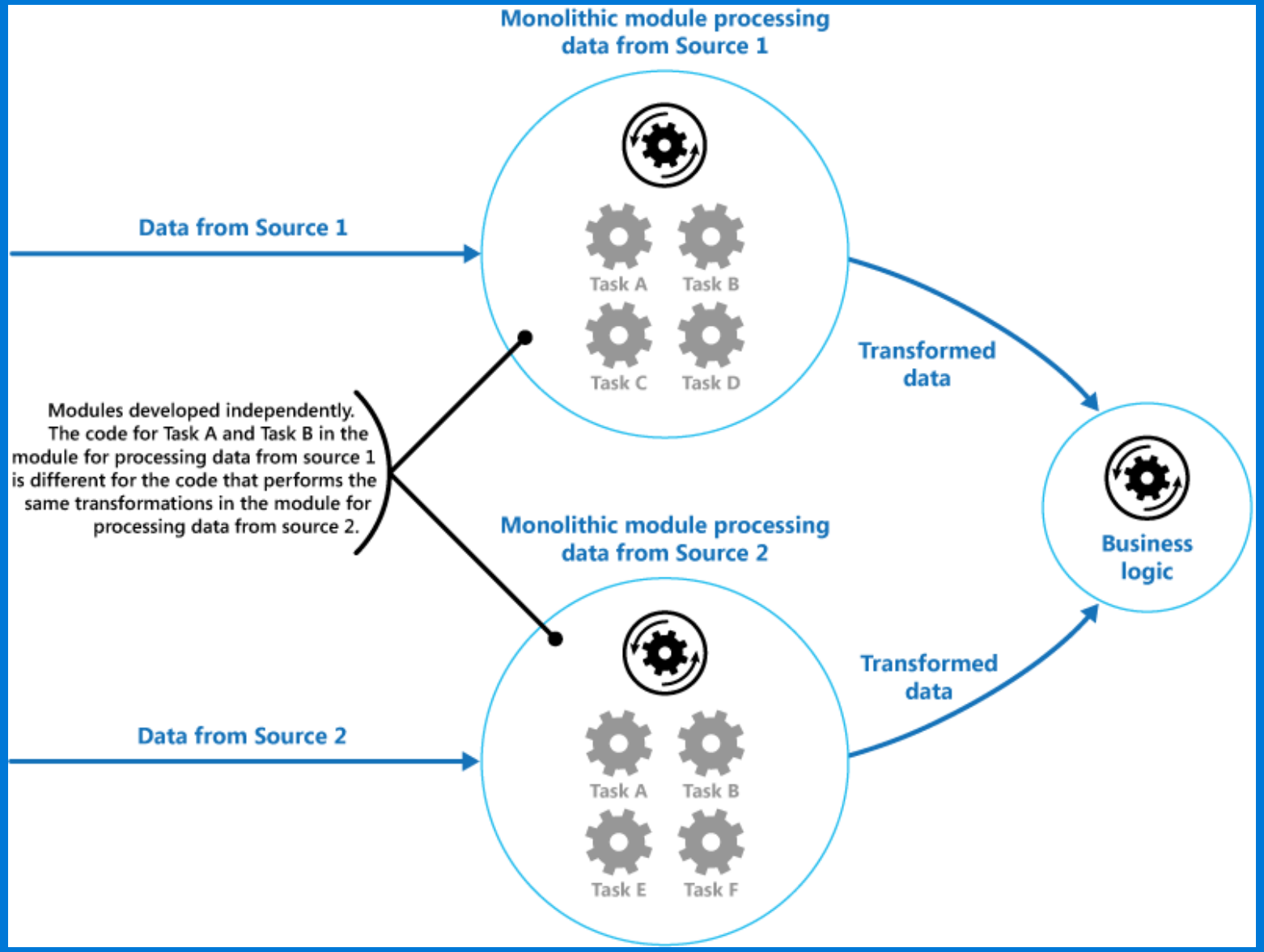
Gateway routing



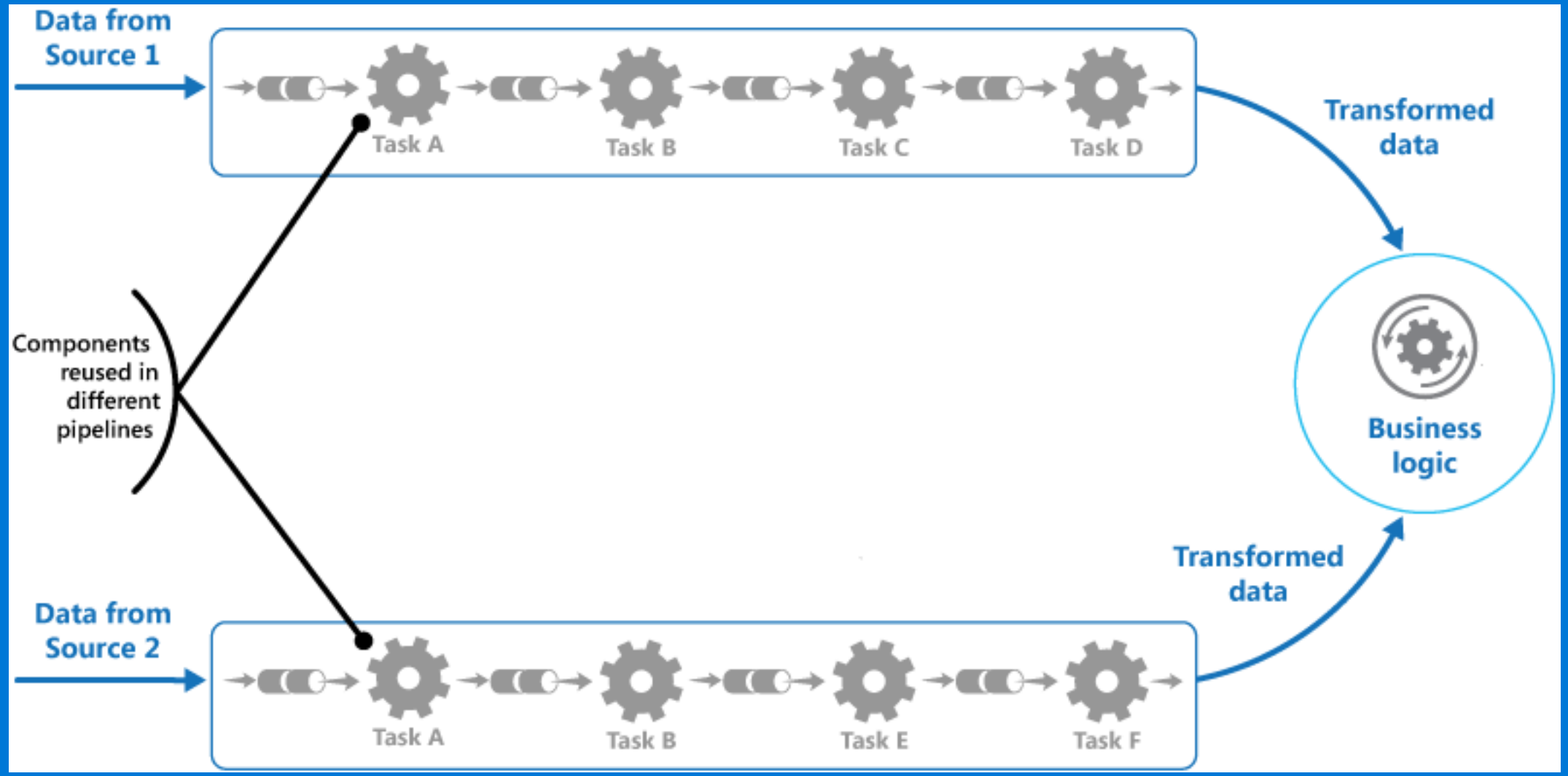
Leader election



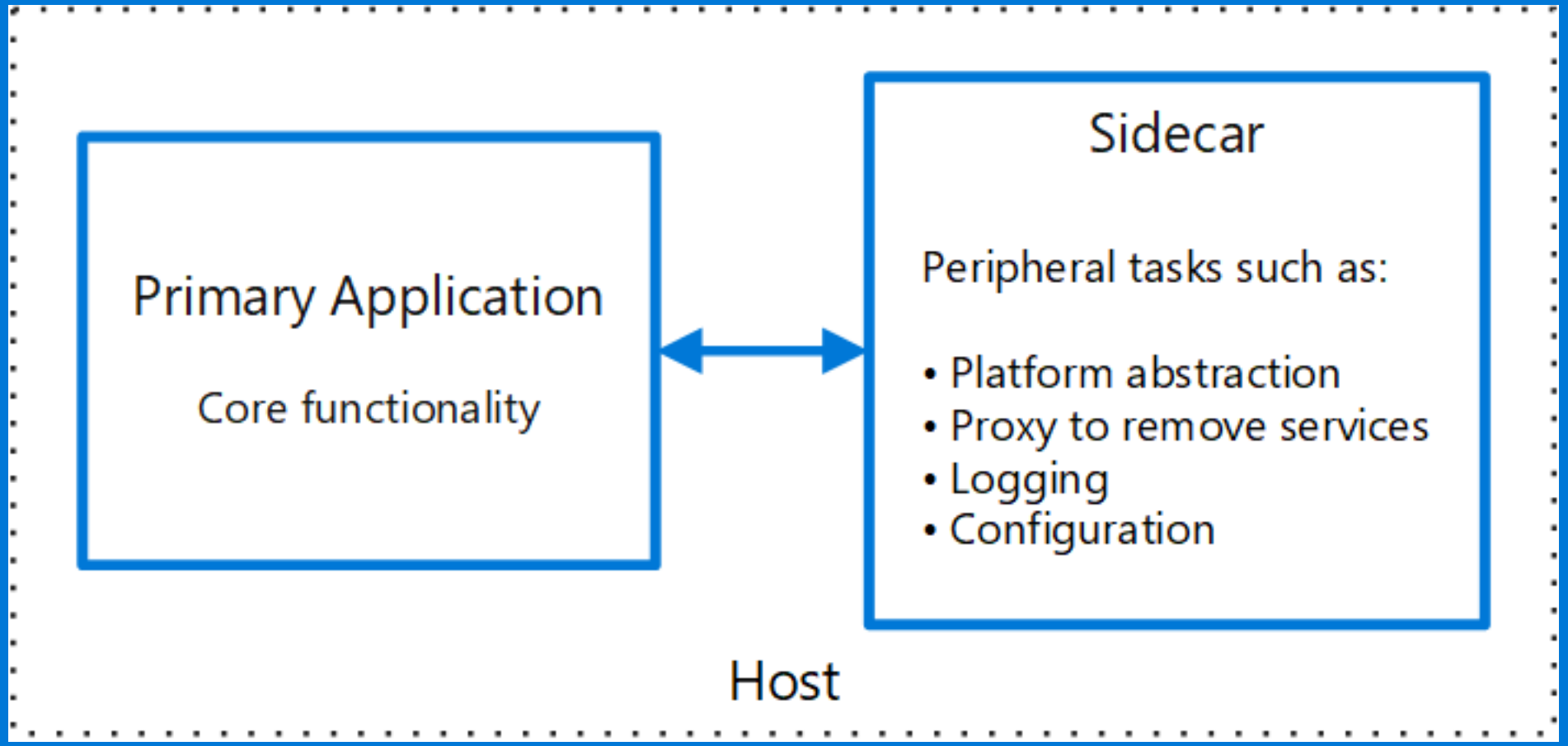
Pipes and filters – issue



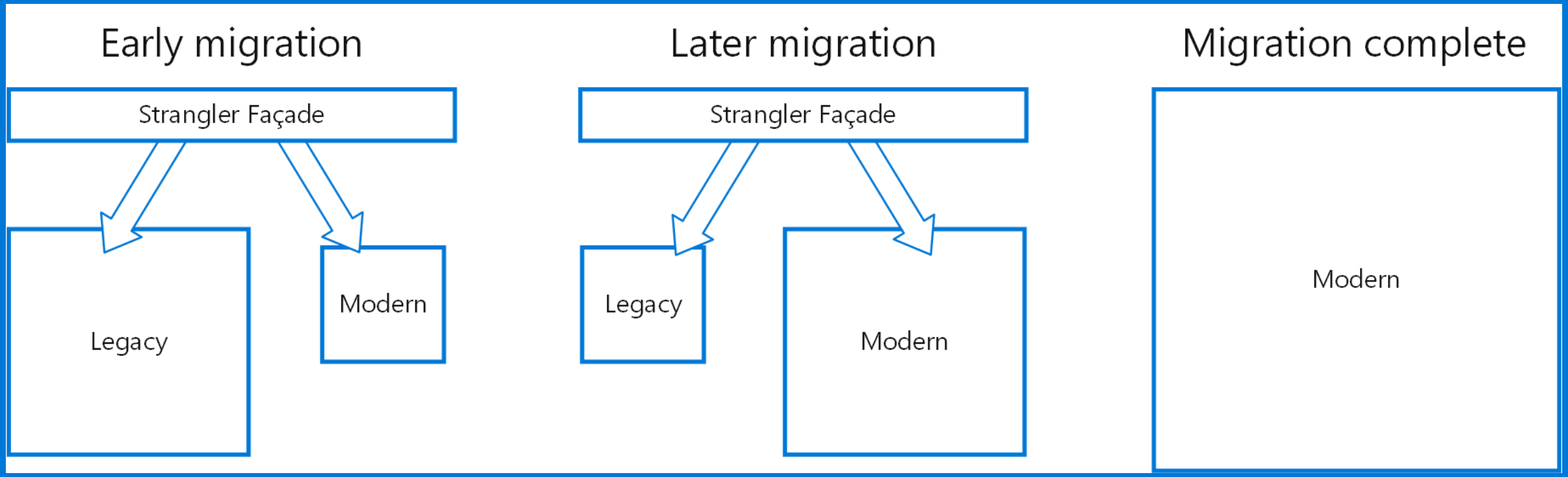
Pipes and filters



Sidecar


Primary Application
Core functionality
Sidecar
Peripheral tasks such as:
• Platform abstraction
• Proxy to remove services
• Logging
• Configuration
Host

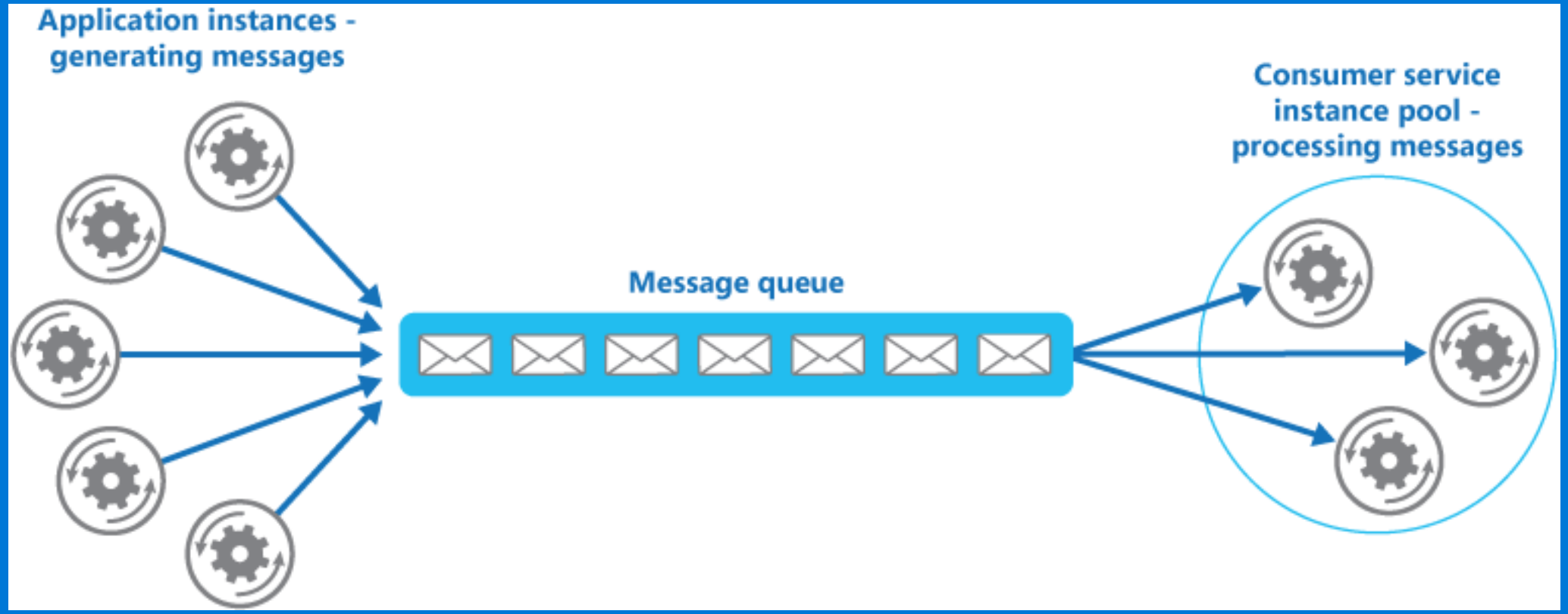
Strangler


Early migration
Strangler Façade
Legacy
Modern
Later migration
Strangler Façade
Legacy
Modern
Migration complete
Modern

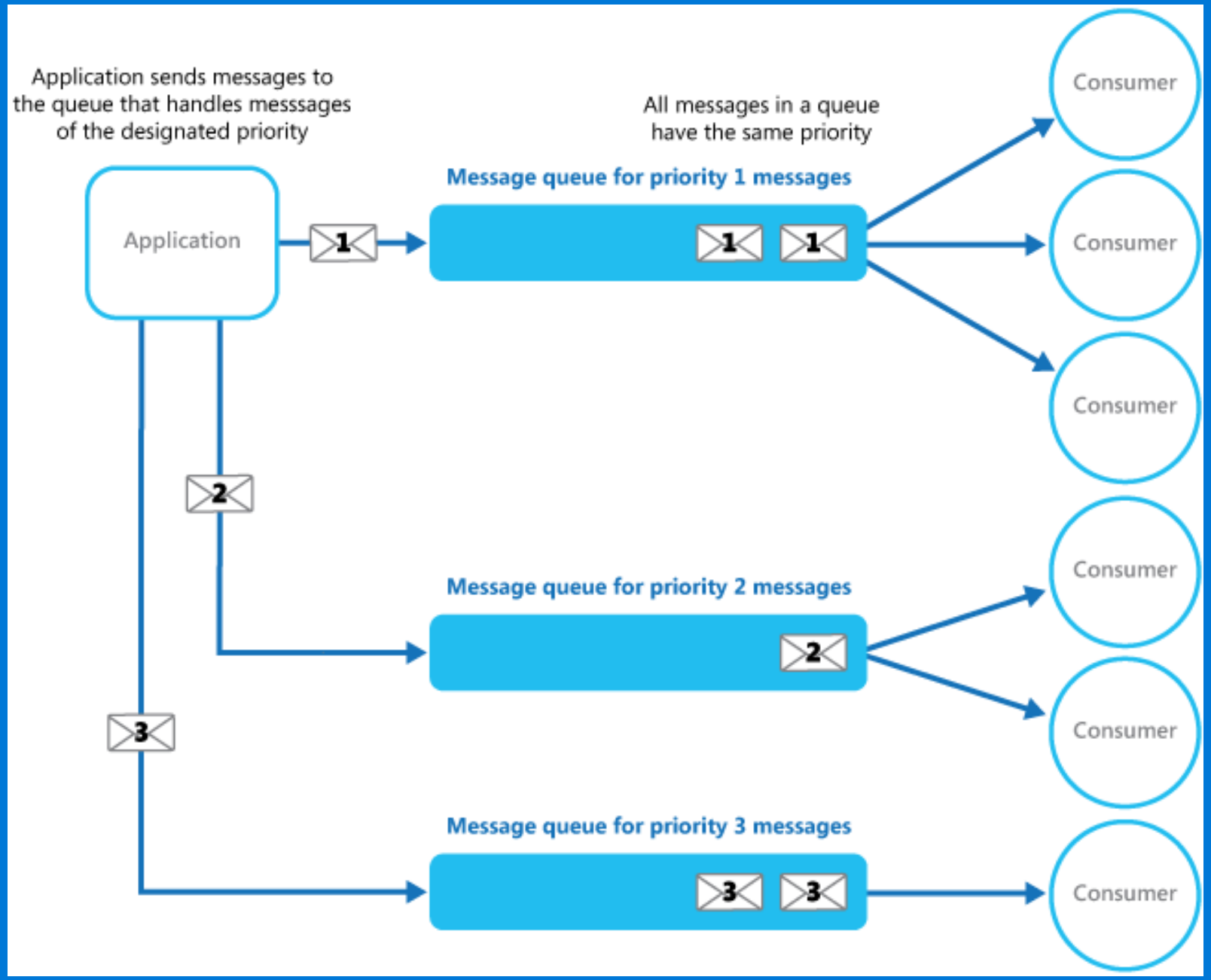
Messaging patterns

Pattern	Summary
Competing Consumers	Enable multiple concurrent consumers to process messages received on the same messaging channel.
Pipes and Filters	Break down a task that performs complex processing into a series of separate elements that can be reused.
Priority Queue	Prioritize requests sent to services so that requests with a higher priority are received and processed more quickly than those with a lower priority.
Queue-Based Load Leveling	Use a queue that acts as a buffer between a task and a service that it invokes in order to smooth intermittent heavy loads.
Scheduler Agent Supervisor	Coordinate a set of actions across a distributed set of services and other remote resources.

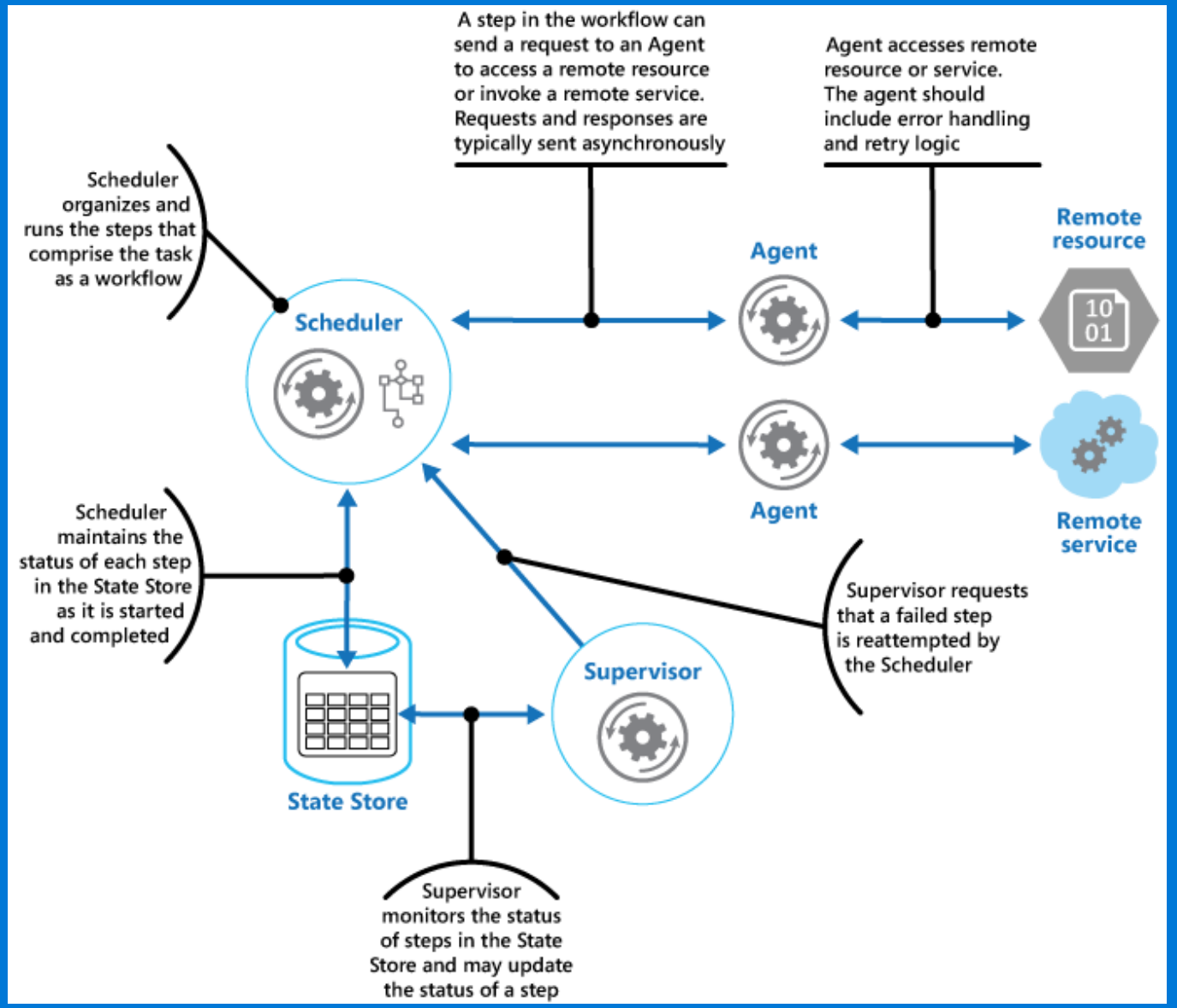
Competing consumers



Priority queue



Scheduler agent supervisor



Management and monitoring

Pattern	Summary
Ambassador	Create helper services that send network requests on behalf of a consumer service or application.
Anti-Corruption Layer	Implement a façade or adapter layer between a modern application and a legacy system.
External Configuration Store	Move configuration information out of the application deployment package to a centralized location.
Gateway Aggregation	Use a gateway to aggregate multiple individual requests into a single request.
Gateway Offloading	Offload shared or specialized service functionality to a gateway proxy.
Gateway Routing	Route requests to multiple services using a single endpoint.
Health Endpoint Monitoring	Implement functional checks in an application that external tools can access through exposed endpoints at regular intervals.
Sidecar	Deploy components of an application into a separate process or container to provide isolation and encapsulation.
Strangler	Incrementally migrate a legacy system by gradually replacing specific pieces of functionality with new applications and services.

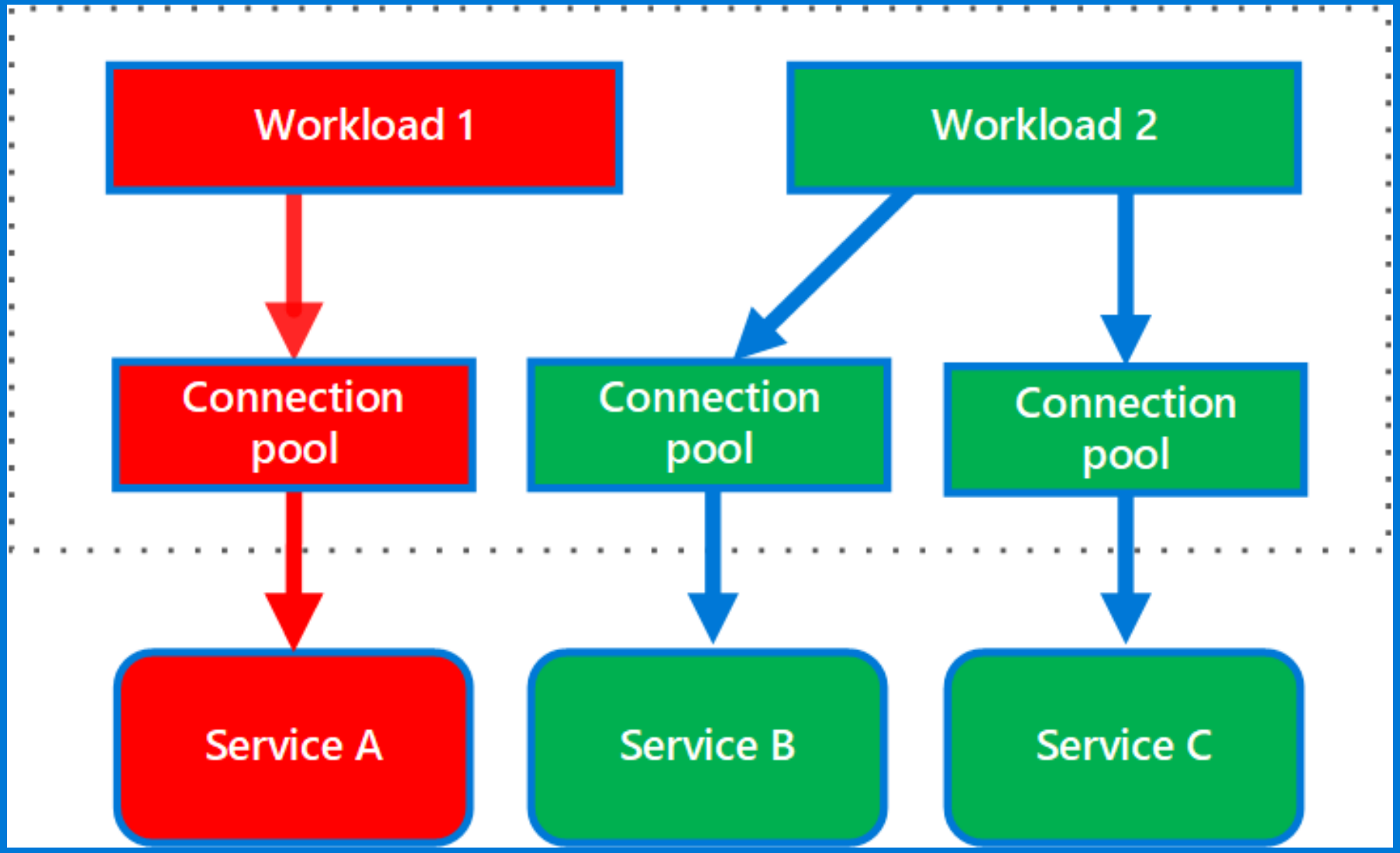
Performance and Scalability

Pattern	Summary
Cache-Aside	Load data on demand into a cache from a data store
CQRS	Segregate operations that read data from operations that update data by using separate interfaces.
Event Sourcing	Use an append-only store to record the full series of events that describe actions taken on data in a domain.
Index Table	Create indexes over the fields in data stores that are frequently referenced by queries.
Materialized View	Generate prepopulated views over the data in one or more data stores when the data isn't ideally formatted for required query operations.
Priority Queue	Prioritize requests sent to services so that requests with a higher priority are received and processed more quickly than those with a lower priority.
Queue-Based Load Leveling	Use a queue that acts as a buffer between a task and a service that it invokes in order to smooth intermittent heavy loads.
Sharding	Divide a data store into a set of horizontal partitions or shards.
Static Content Hosting	Deploy static content to a cloud-based storage service that can deliver them directly to the client.
Throttling	Control the consumption of resources used by an instance of an application, an individual tenant, or an entire service.

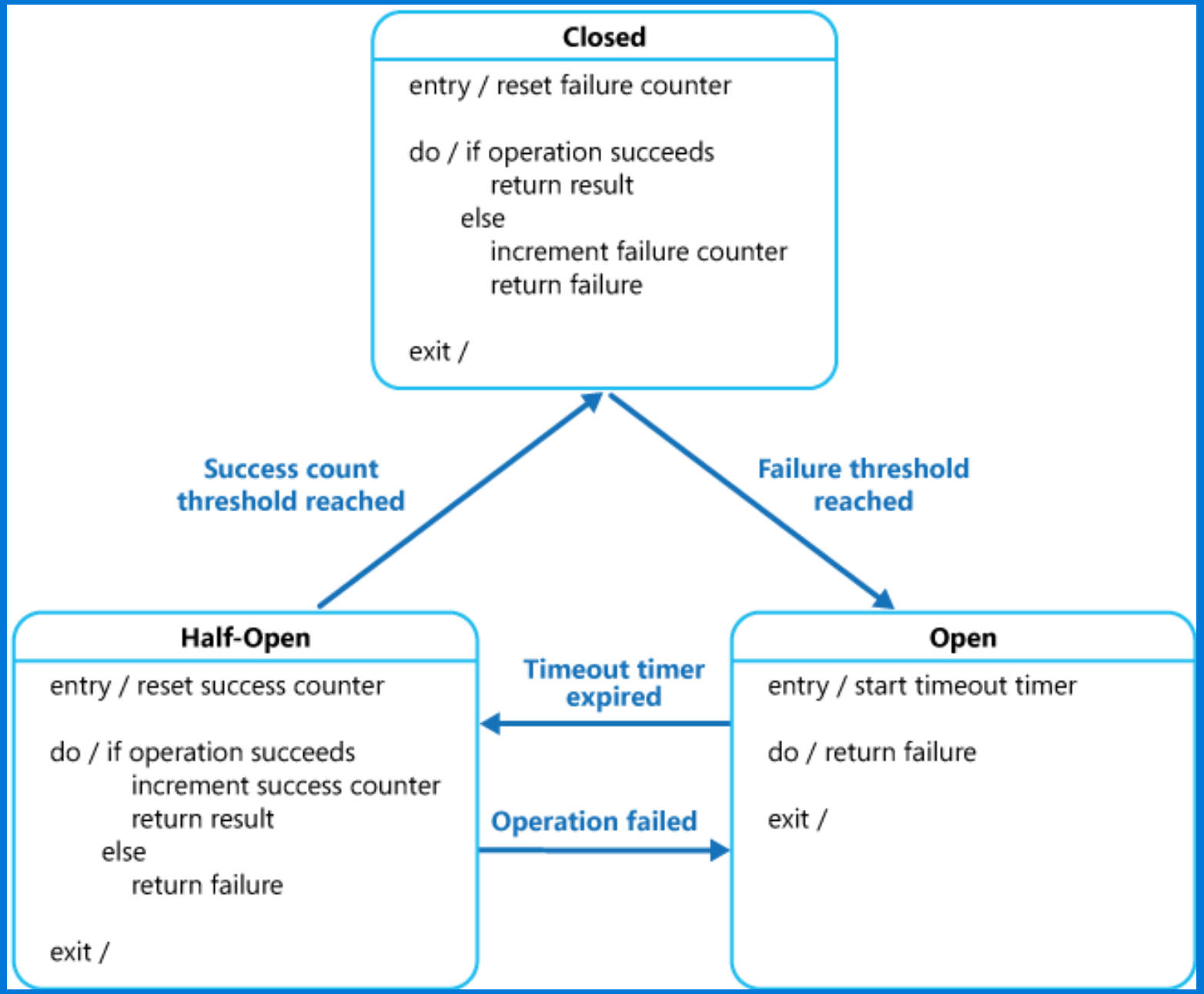
Resiliency

Pattern	Summary
Bulkhead	Isolate elements of an application into pools so that if one fails, the others will continue to function.
Circuit Breaker	Handle faults that might take a variable amount of time to fix when connecting to a remote service or resource.
Compensating Transaction	Undo the work performed by a series of steps, which together define an eventually consistent operation.
Health Endpoint Monitoring	Implement functional checks in an application that external tools can access through exposed endpoints at regular intervals.
Leader Election	Coordinate the actions performed by a collection of collaborating task instances in a distributed application by electing one instance as the leader that assumes responsibility for managing the other instances.
Queue-Based Load Leveling	Use a queue that acts as a buffer between a task and a service that it invokes in order to smooth intermittent heavy loads.
Retry	Enable an application to handle anticipated, temporary failures when it tries to connect to a service or network resource by transparently retrying an operation that's previously failed.
Scheduler Agent Supervisor	Coordinate a set of actions across a distributed set of services and other remote resources.

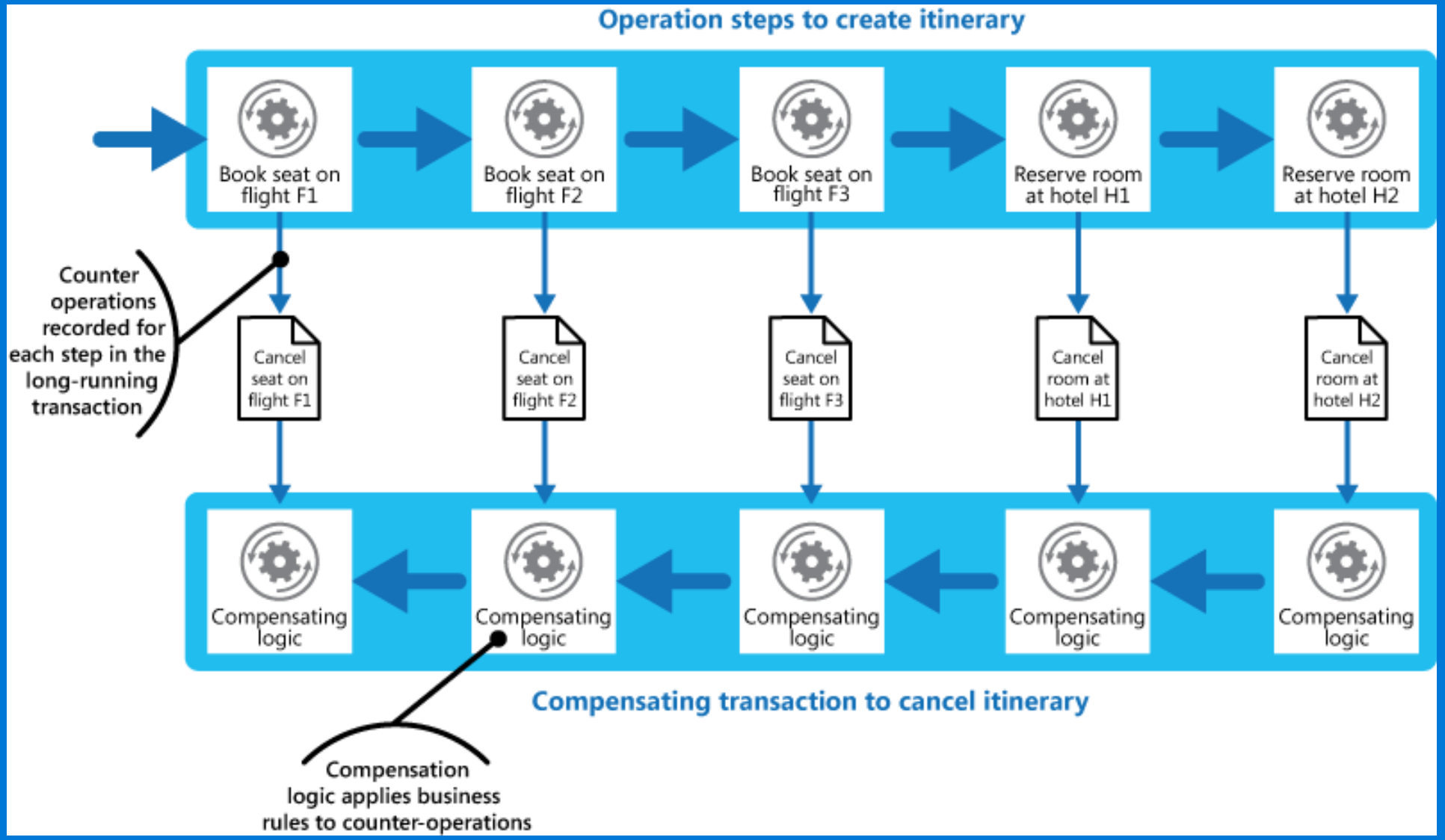
Bulkhead


Workload 1
Workload 2
Connection pool
Connection pool
Connection pool
Service A
Service B
Service C

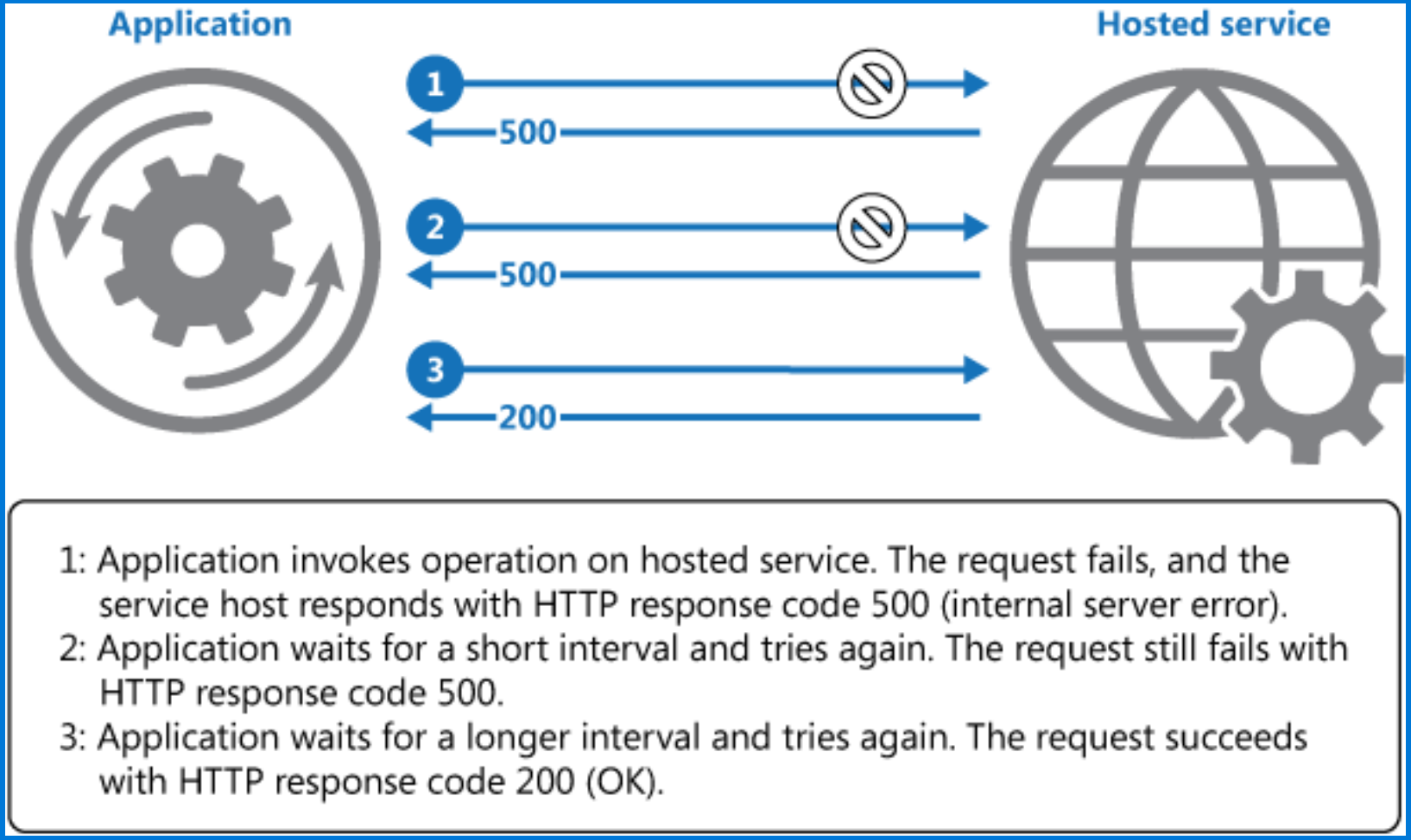
Circuit breaker



Compensating transaction



Retry pattern

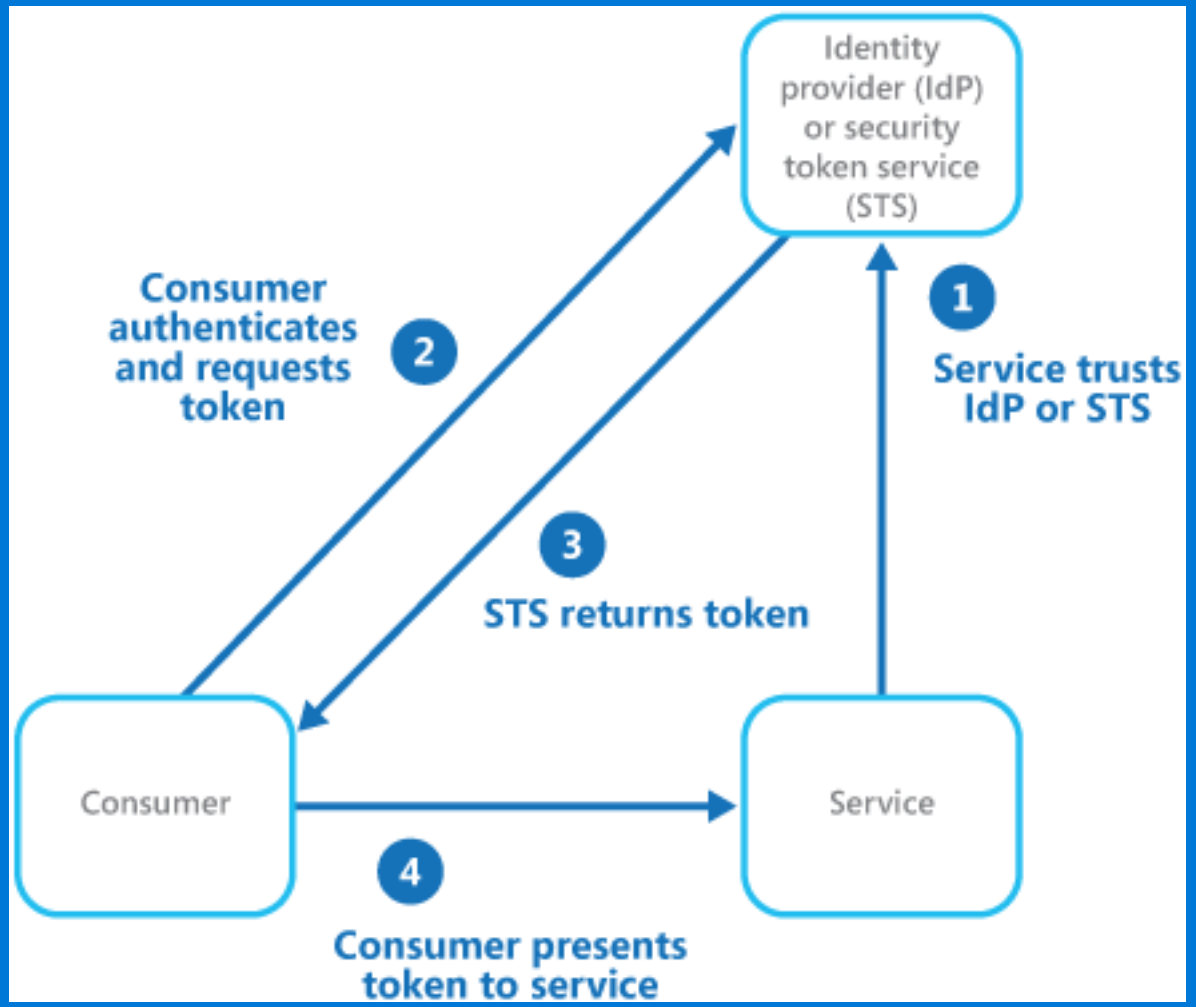


1: Application invokes operation on hosted service. The request fails, and the service host responds with HTTP response code 500 (internal server error).
2: Application waits for a short interval and tries again. The request still fails with HTTP response code 500.
3: Application waits for a longer interval and tries again. The request succeeds with HTTP response code 200 (OK).

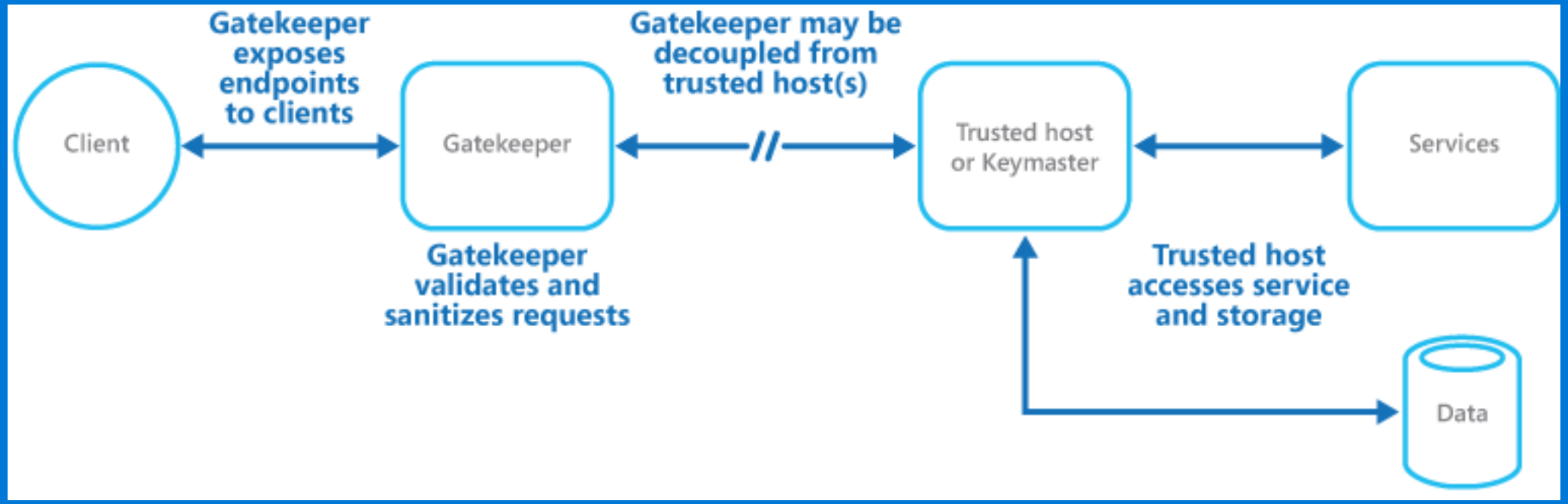
Security

Pattern	Summary
Federated Identity	Delegate authentication to an external identity provider.
Gatekeeper	Protect applications and services by using a dedicated host instance that acts as a broker between clients and the application or service, validates and sanitizes requests, and passes requests and data between them.
Valet Key	Use a token or key that provides clients with restricted direct access to a specific resource or service.

Federated identity



Gatekeeper pattern



¿Questions?