Scalable Web Application Architecture and Tools

1. Scalable Web Application Architecture
2. Distributed Computation
3. Apache Ignite (In Memory Distributed Database)
4. Mongo DB (Distributed Document Database)
5. Object Storage
   Azure Blob, Distributed Cloud Storage
6. Docker Container, Compose, Swarm and Kubernetes
7. Microservices

DevOps and Tools

1. DevOps
2. Deployment Tool (msdeploy)
3. Continuous Deployment (SaltStack, MS DevOps, Jenkins)
4. Code as Infrastructure(Chef, SaltStack)

Azure Cloud

1. Cloud Basics XaaS
   IaaS, PaaS, SaaS, DBaaS,… etc
2. Azure Compute
   Azure VM, Availability Sets, Connections, Load Balancer, Application Gateway
3. Azure DevOps
4. Azure Network
   Virtual, Network, Public IP, Load balancers, App Gateway
5. Azure Databases
   SQL Server, MySQL, Maria DB, SQL Elastic Pools, Cosmos DB

Programming Language

1. Python
2. Django
3. Go
4. Web Assembly and Blazor
5. Nodejs

Scalable Web Application Architecture

Scalable (lity) vs Elastic (ity)

- Scalability: "Increasing" the capacity to meet the "increasing" workload.
- Elasticity: "Increasing or reducing" the capacity to meet the "increasing or reducing" workload.
- Scalability: In a scaling environment, the available resources may exceed to meet the "future demands".
- Elasticity: In the elastic environment, the available resources match the "current demands" as closely as possible.
- Scalability: Scalability enables a corporate to meet expected demands for services with "long-term, strategic needs".
- Elasticity: Elasticity enables a corporate to meet unexpected changes in the demand for services with "short-term, tactical needs".
- Scalability: Scalability adapts only to the "workload increase" by "provisioning" the resources in an "incremental" manner.
- Elasticity: Elasticity adapts to both the "workload increase" as well as "workload decrease" by "provisioning and deprovisioning" resources in an "autonomic" manner.

- Scalability: Increasing workload is served with increasing the power of a single computer resource or with increasing the power by a group of computer resources.
- Elasticity: Varying workload is served with dynamic variations in the use of computer resources.

- Scalability: It is "increasing" the capacity to serve an environment where workload is increasing.

This scalability could be "Scaling Up" or "Scaling Out".

(Example:

Scaling Up - increasing the ability of an individual server

Scaling out - increasing the ability by adding multiple servers to the individual server.)

Elasticity: It is the ability to "scale up or scale down" the capacity to serve at will.

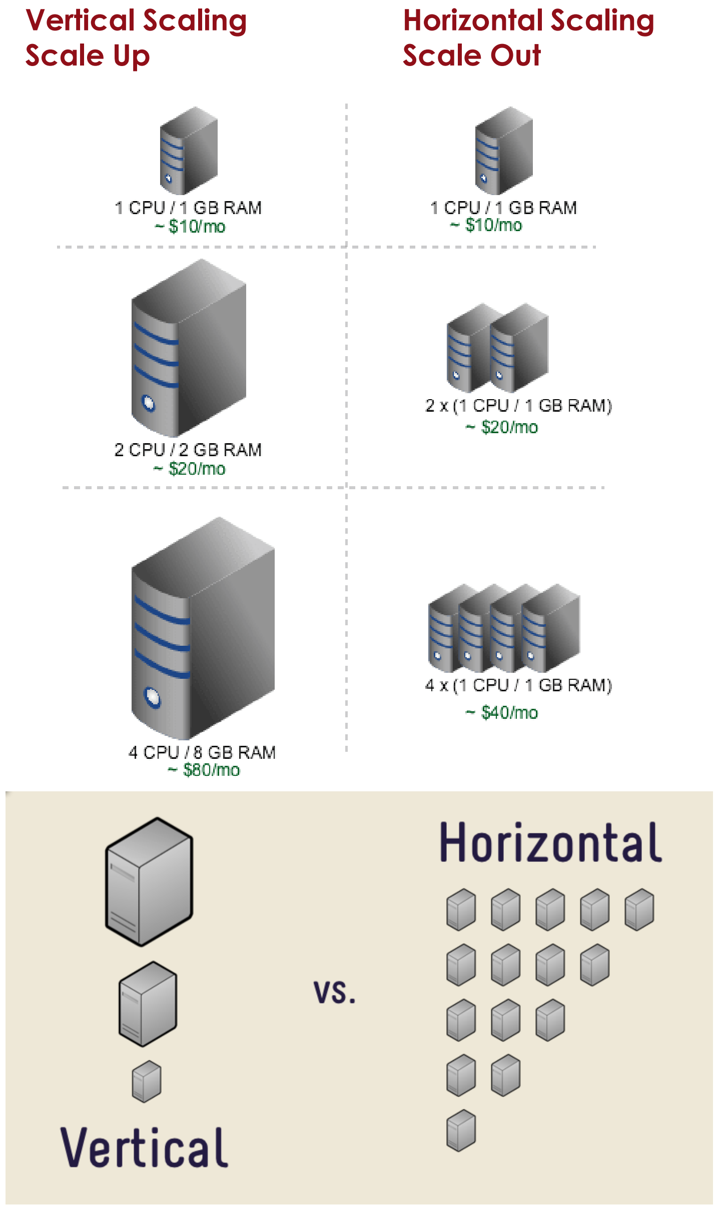
- Scalability: To use a simile, "scaling up" is an individual increasing her power to meet the increasing demands, and "scaling out" is building a team to meet the increasing demands.
- Elasticity: To use a simile, a film actor increasing or reducing her body weight to meet differing needs of the film industry.

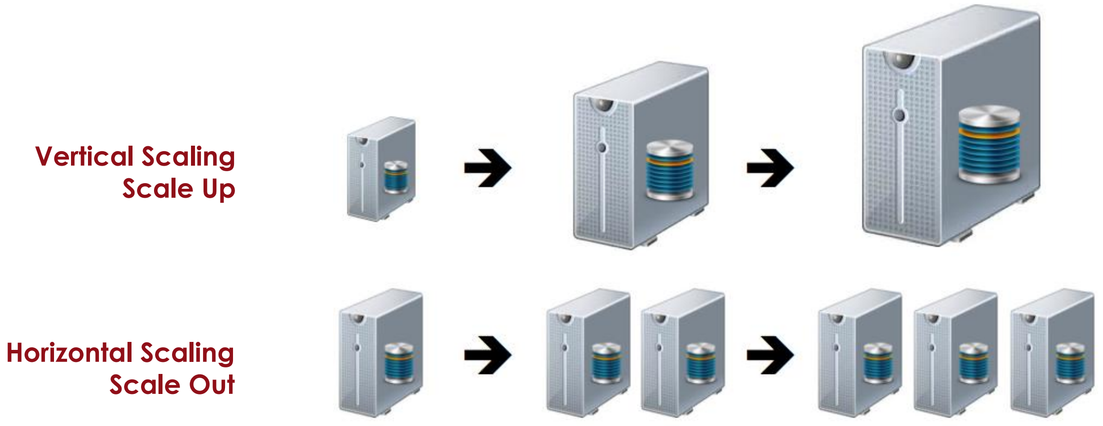
Vertical Scaling
Scale Up

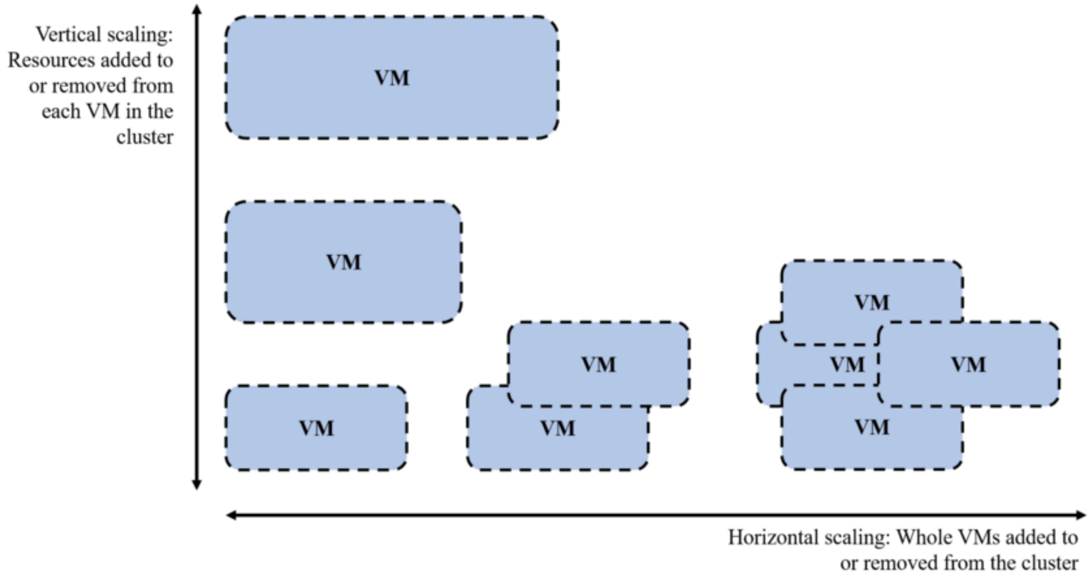
Horizontal Scaling
Scale Out

Compare two method by management and cost !!!

- Risks Management
- Vendor Limitation
- Open and Closed Source (Free) Apps (Windows vs Linux)



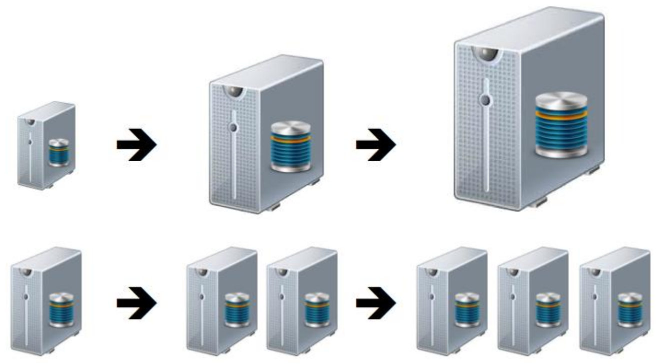



Vertical scaling: Resources added to or removed from each VM in the cluster
VM
VM
VM
VM
VM
VM
VM
VM
VM
Horizontal scaling: Whole VMs added to or removed from the cluster

Granularity

Granularity is the extent to which a system is broken down into small parts, either the system itself or its description or observation. It is the extent to which a larger entity is subdivided. For example, a yard broken into inches has finer granularity than a yard broken into feet.

Coarse-grained - larger components than fine-grained, large subcomponents. Simply wraps one or more fine-grained services together into a more coarse-grained operation.
Fine-grained - smaller components of which the larger ones are composed, lower level service



Fine grained

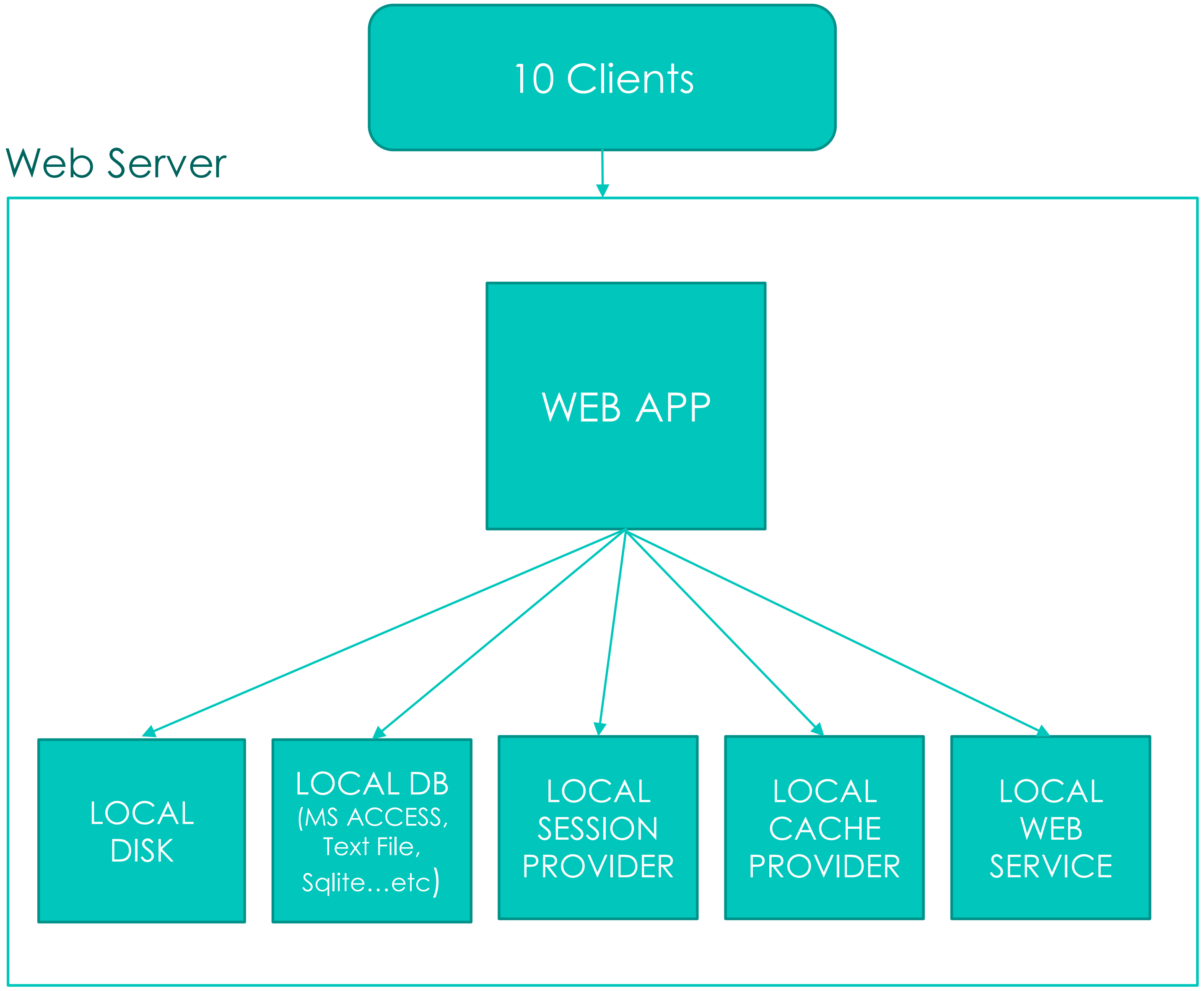
- less computation time
- Program broken large no. of small task
- Task assigned individual processors
- Load balancing
- help to achieve Parallel computation
- High communication and synchronisation overhead
- Higher degree of parallelism
- Detect using compiler
- Grain size 4–5microsec
- few data element assign to single processor

Coarse grained

- High computation time
- Program broken large task
- Task assigned not individual processors
- Maybe Load imbalancing
- less Parallel computation
- Low communication and synchronisation overhead
- Lower degree of parallelism
- Detect using interprocedural parallezing compiler
- Many data element assigned to single processor

Dependencies

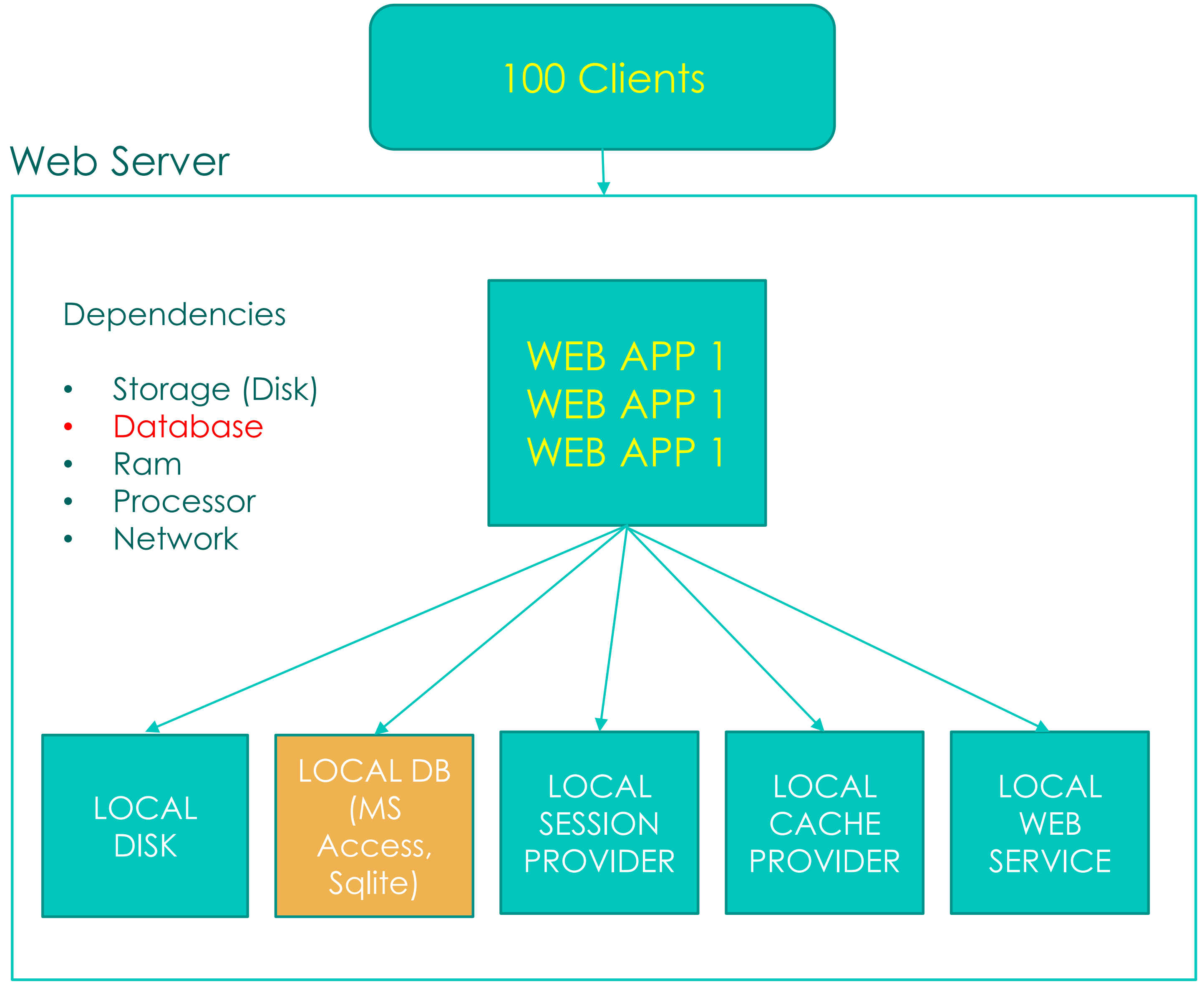
- Storage (Disk)
- Database
- Ram
- Processor
- Hardware
- Network
- Others



Step 1:
Increase Web App Process

Result:
Increased Local Disk Space and Speed
Increased Ram (session, cache)
Increased Processor

Cannot Increase Local Db

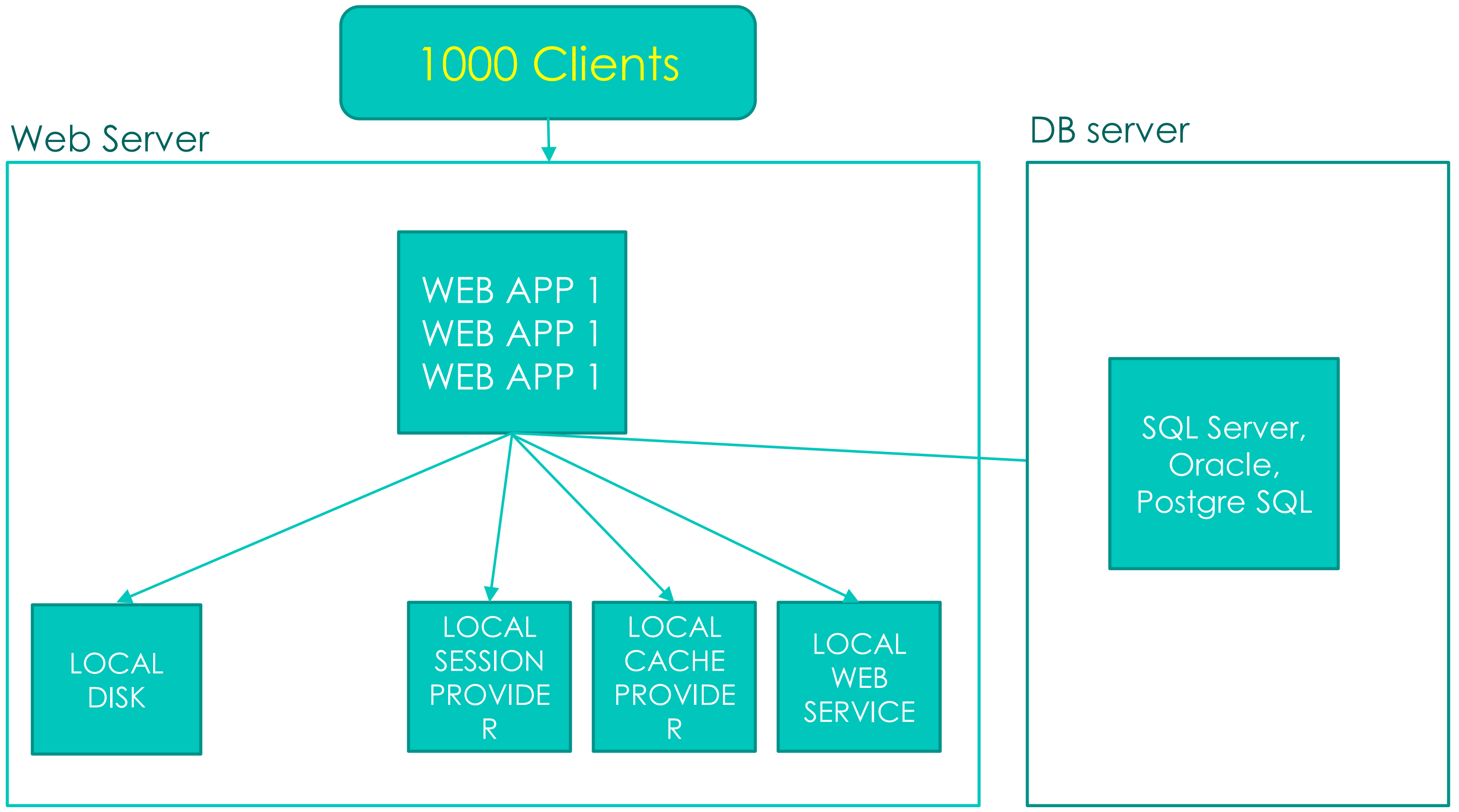


Step 2:
Change Database

Result:
Still needs more local storage , Ram and Processor

Options:

- Get more local resources (create a supercomputer like system) – Too Expensive
- Get new server



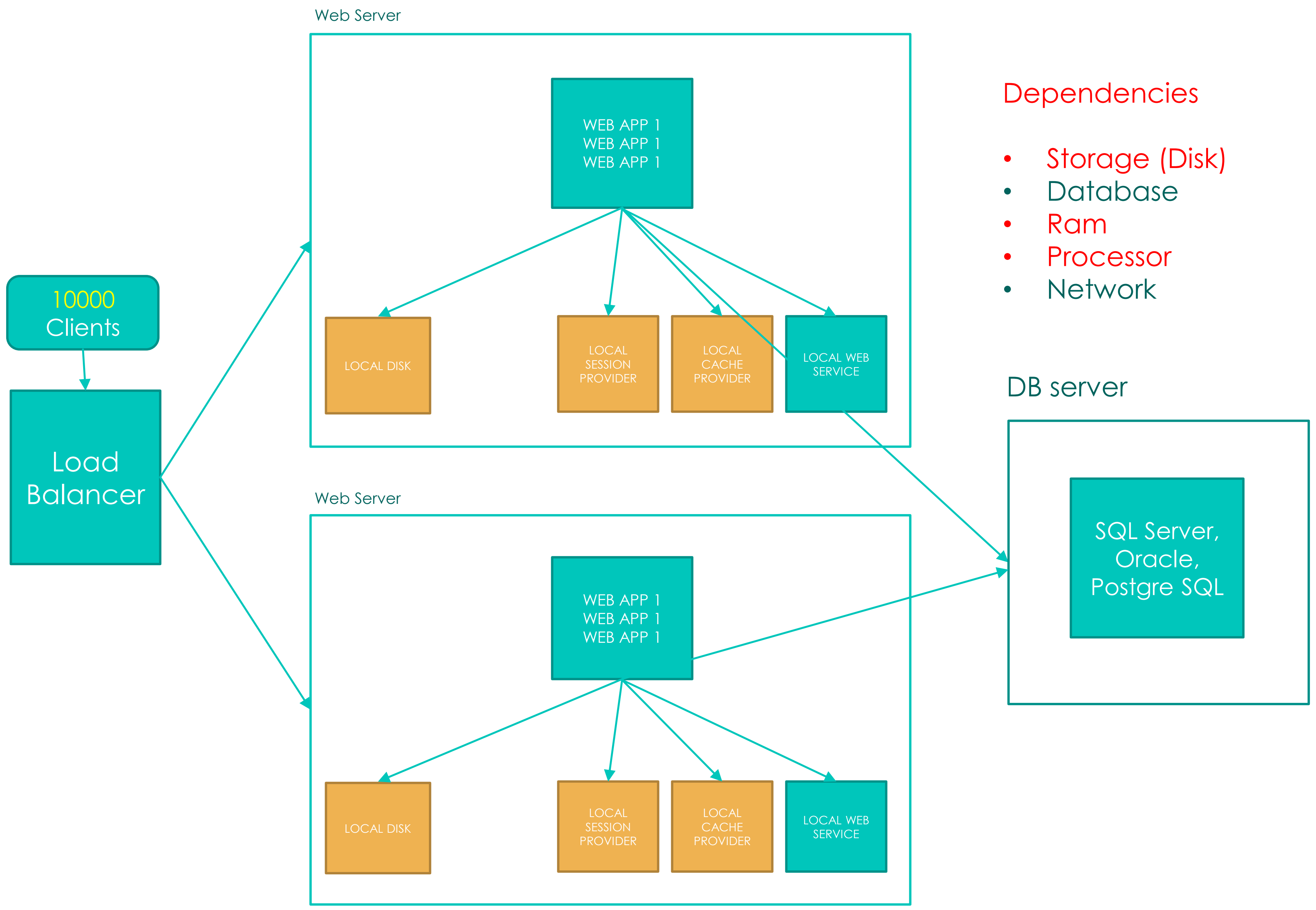
Dependencies

- Storage (Disk)
- Database
- Ram
- Processor
- Network

Step 3:
Increase Web Server

Result:
Storage, session and cache should be synced

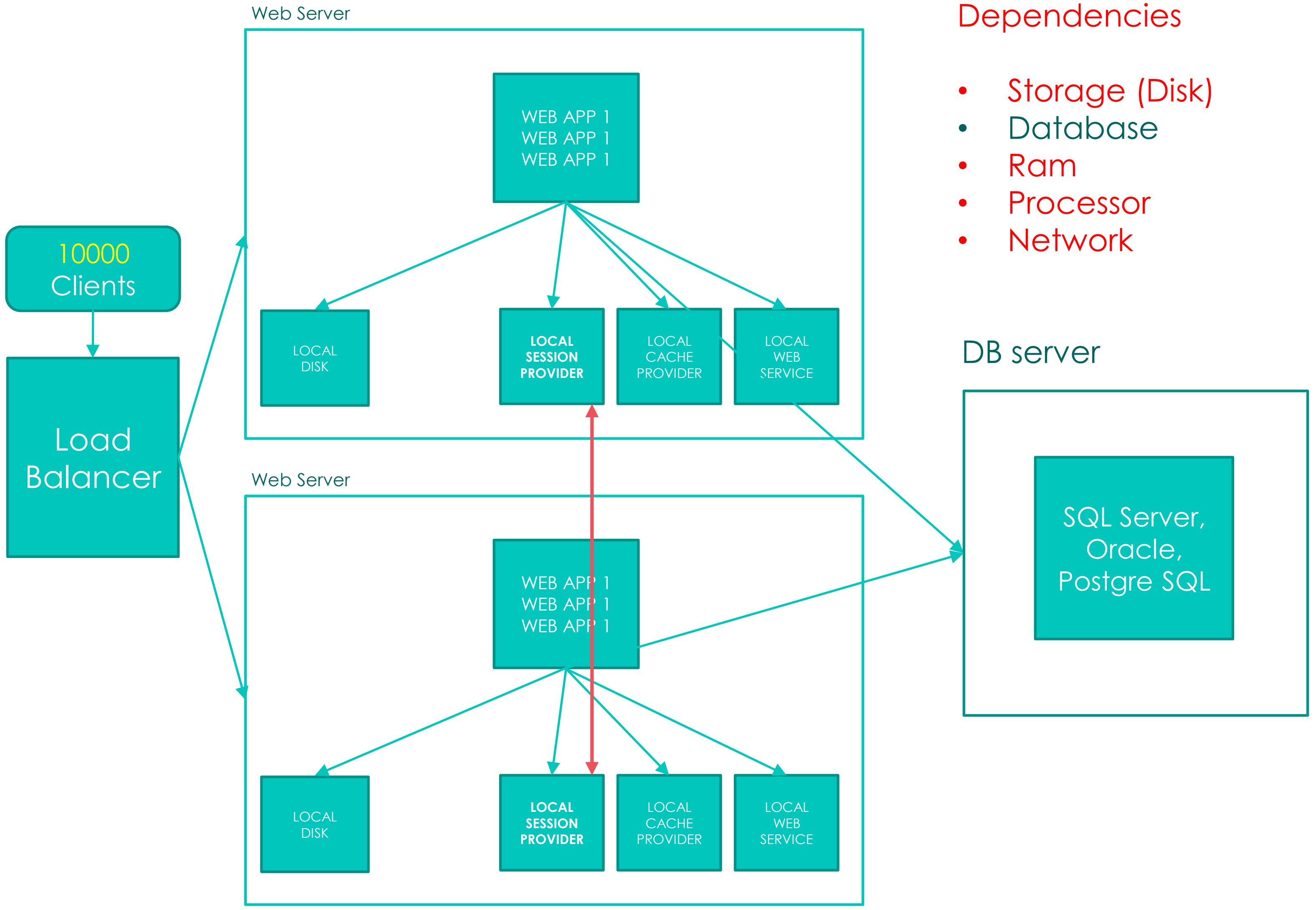
Options:
- Sync ram and disk
- Use shared-distributed system



Dependencies

- Storage (Disk)
- Database
- Ram
- Processor
- Network

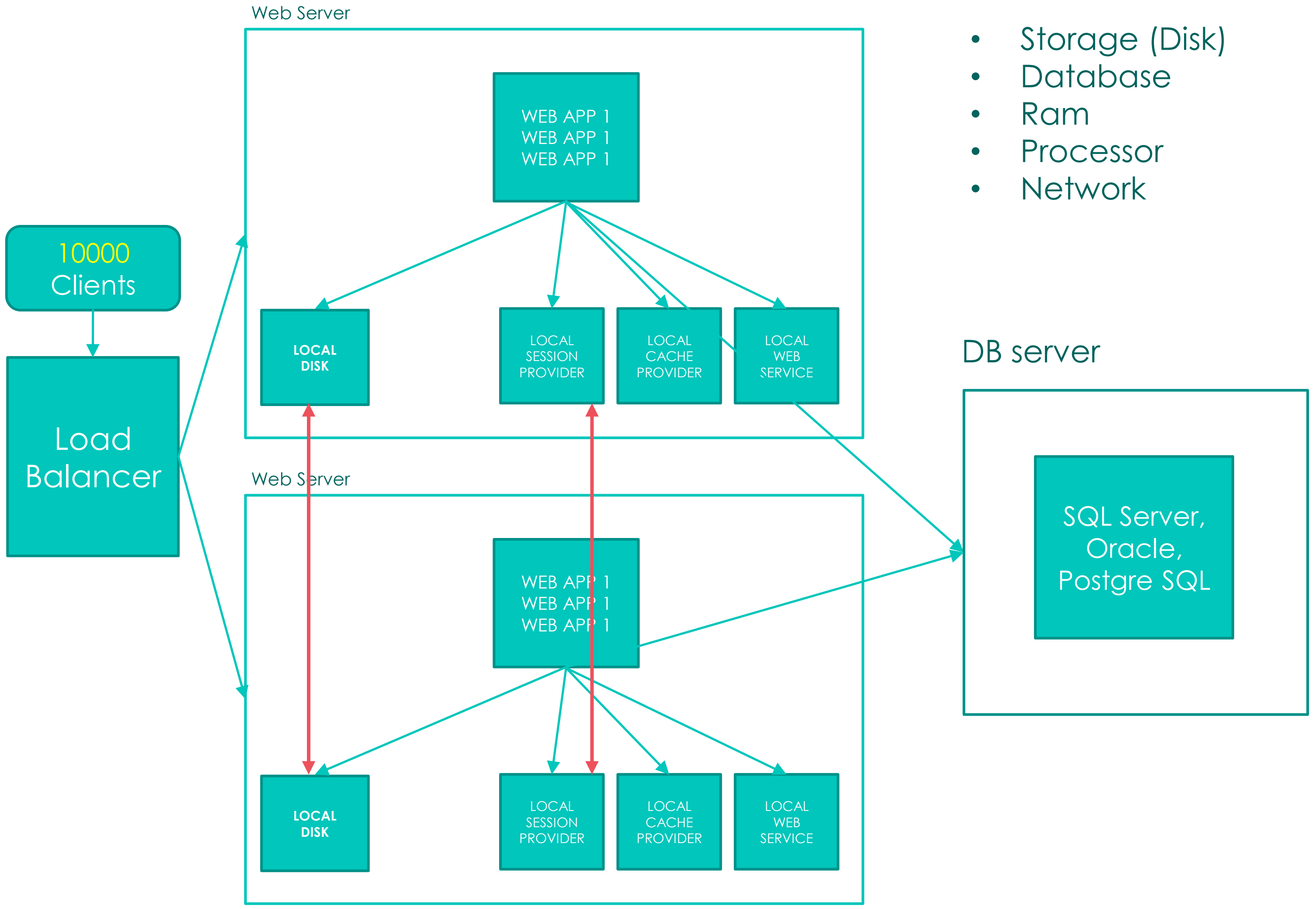
Step 3 – Session Problem: Sync sessions with IIS session state



Dependencies

- Storage (Disk)
- Database
- Ram
- Processor
- Network

Step 3 – Disk Problem:
Sync Disks with MS Storage Replication or MS Distributed File System



Dependencies

- Storage (Disk)
- Database
- Ram
- Processor
- Network

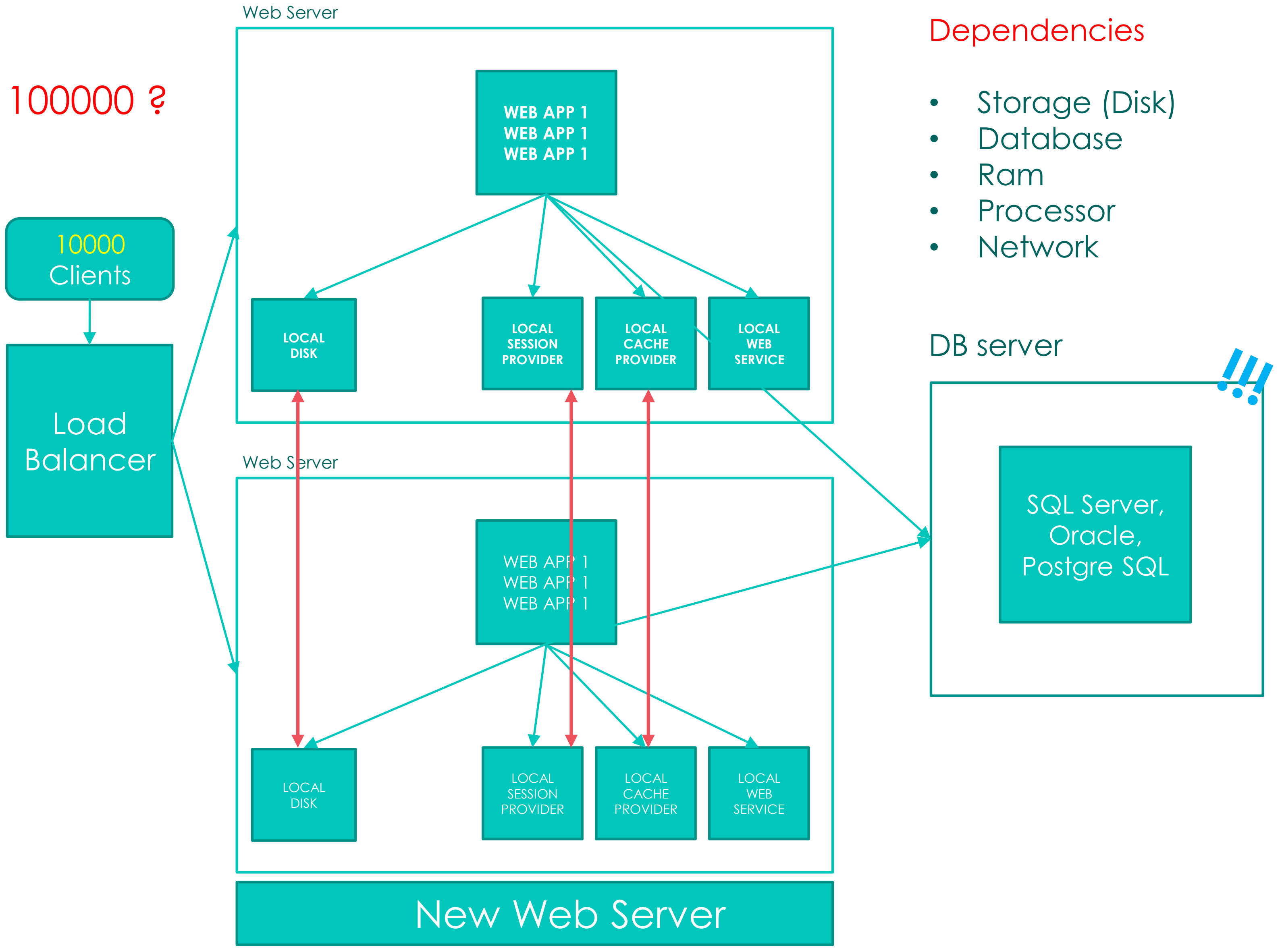
Step 3 – Cache Problem:
MS AppFabric for distributed cache

Results:
If we need new web app, it must be part of this system.

If we need new environment for new app, we must setup same system again.

If one of the components (eg session, cache) need more ram or processor, we cannot increase the resource for it separately.

AppFabric is not supported … etc.



Dependencies

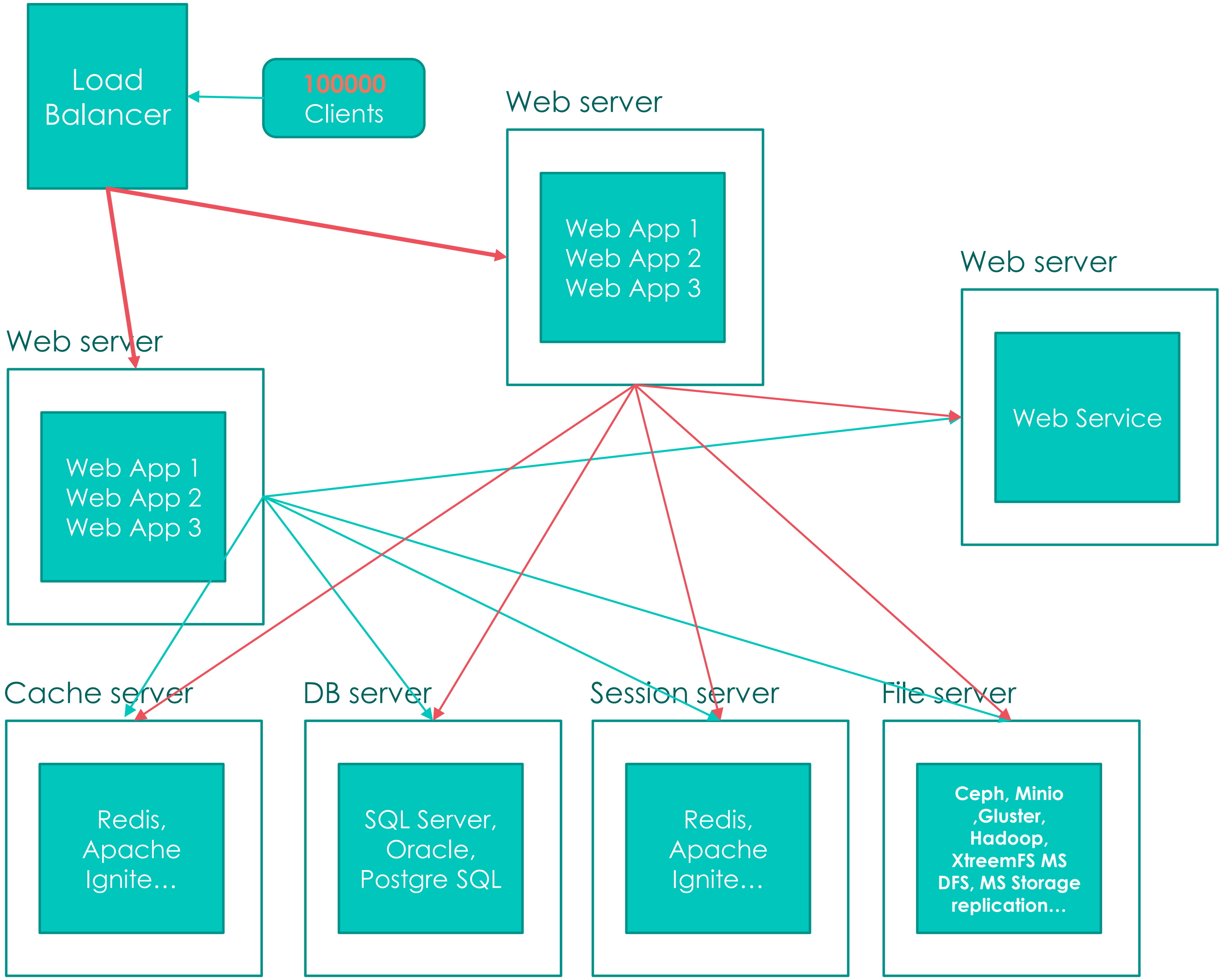
- Storage (Disk)
- Database
- Ram
- Processor
- Network

Step 4 – First Optimization:
isolate database, hard disk and session from the system.

Results:
All Components can be increased separately.

SQL Server, Postgres, Oracle are not distributed by default. Also they are really slow on some cases.

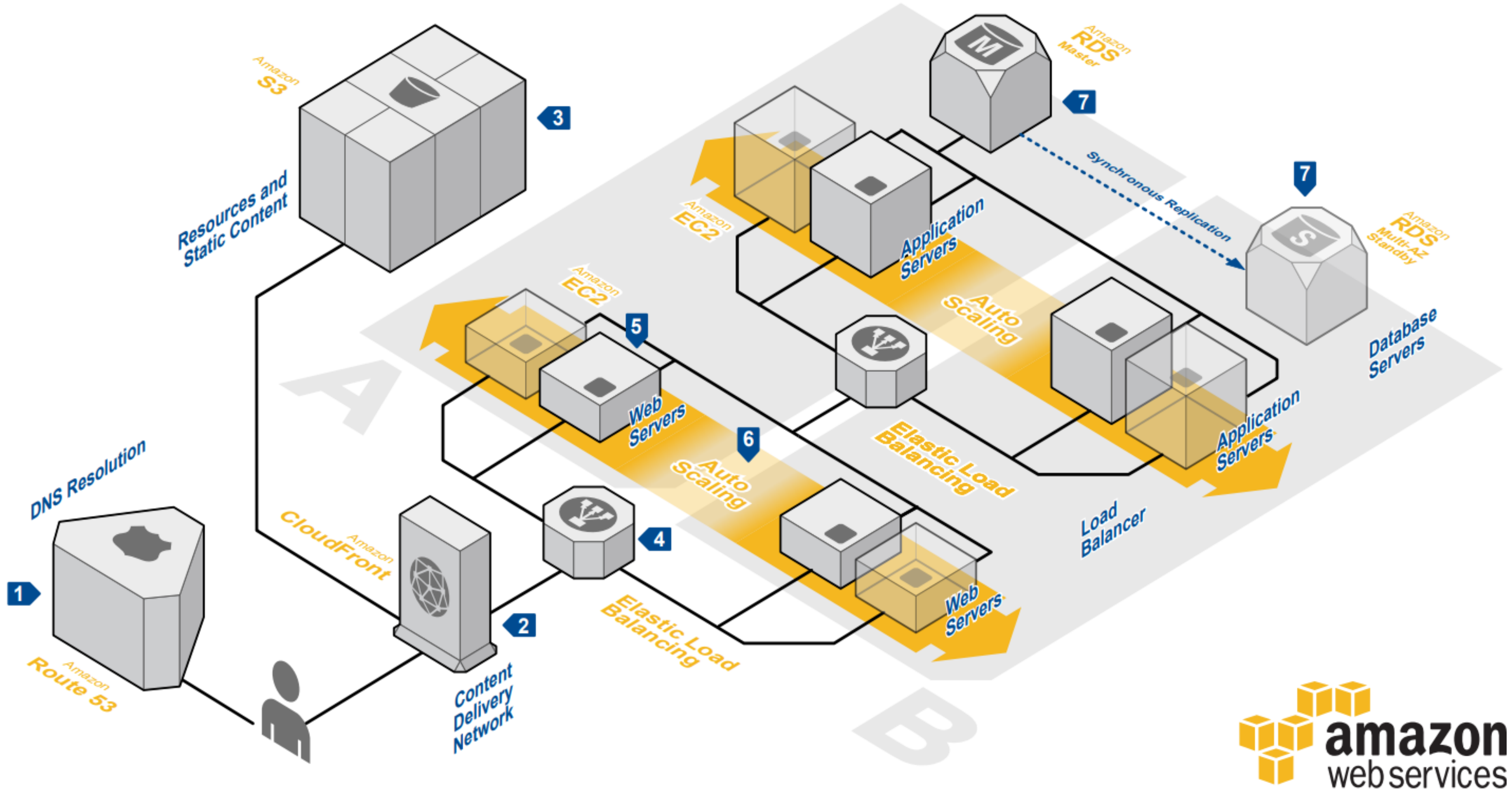
Options:
Distributed DBs can be used (Document and Column Based DBs)



AWS Architecture Center

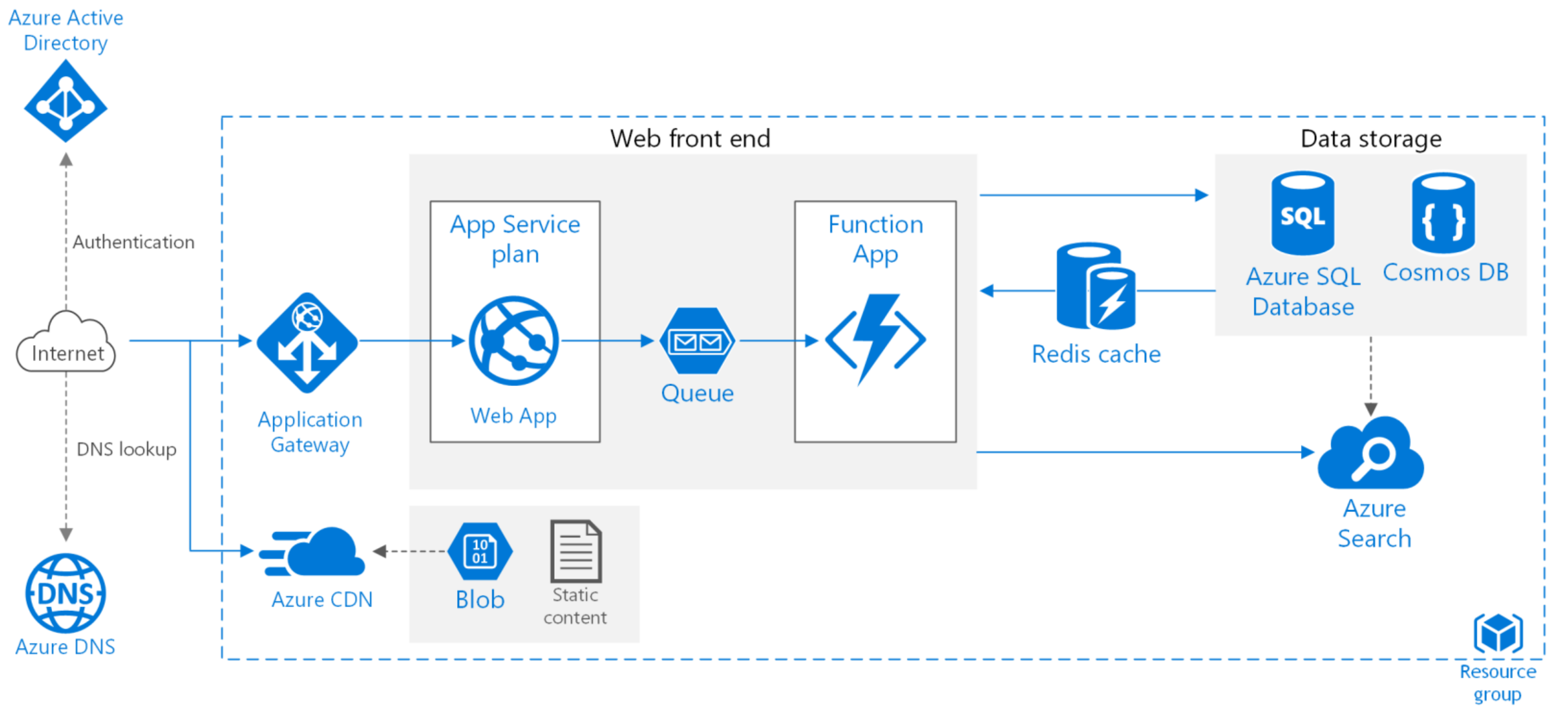
https://aws.amazon.com/tr/architecture/?awsf.quickstart-architecture-page-filter=highlight%23new

https://aws.amazon.com/tr/architecture/



Azure Architecture Center

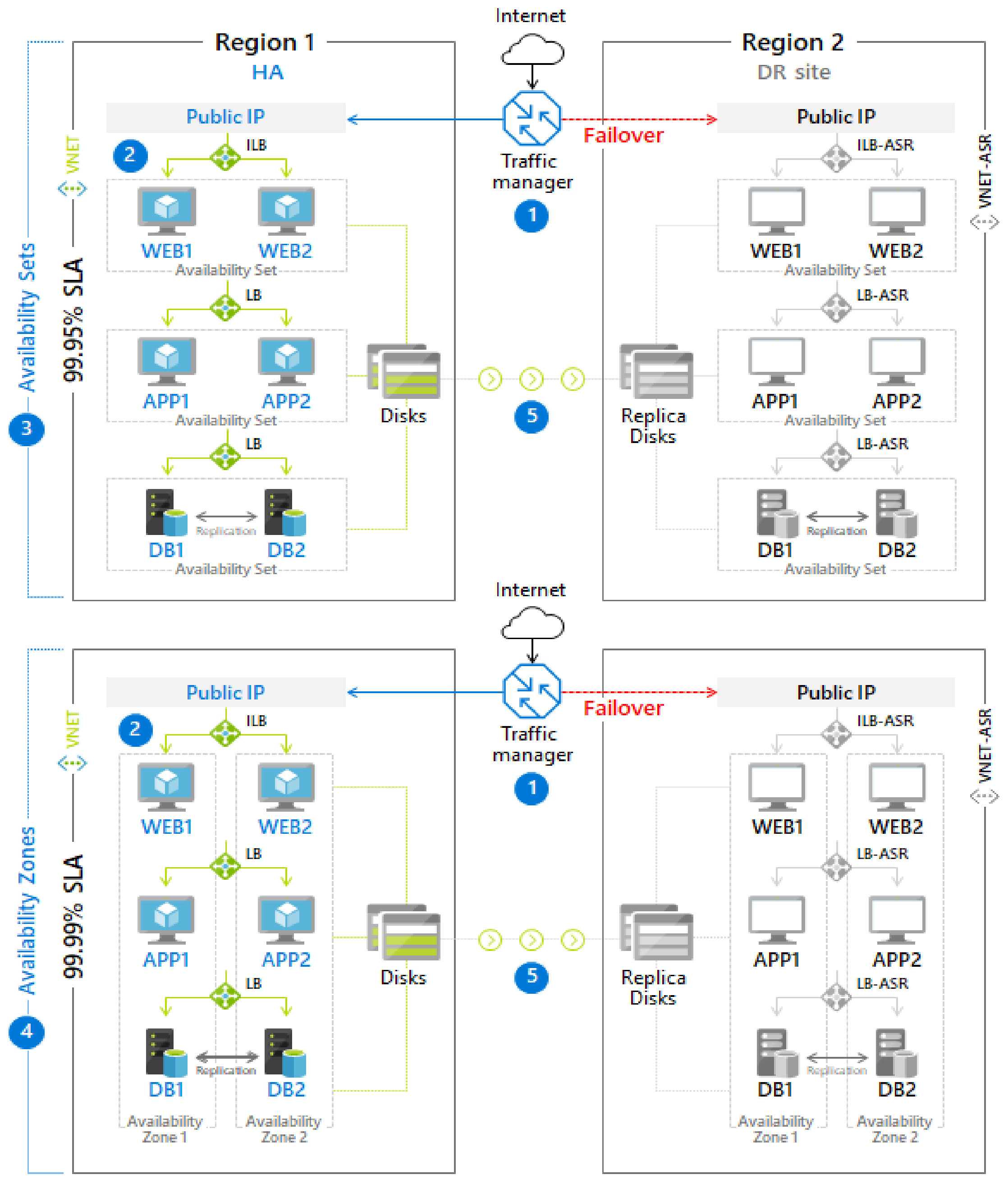
https://docs.microsoft.com/en-us/azure/architecture/



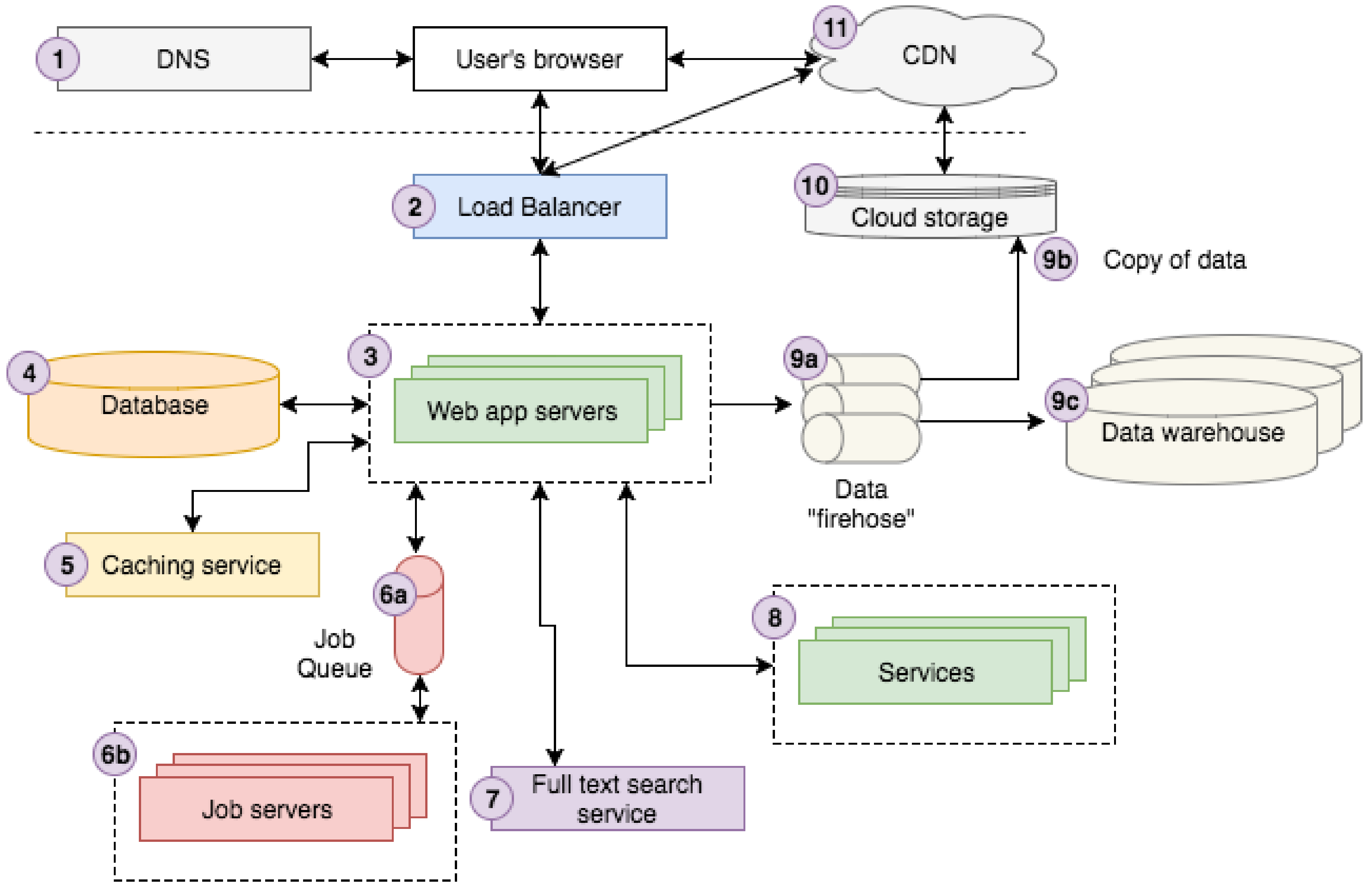
Azure Architecture Center

High Scalable vs High Available

https://docs.microsoft.com/en-us/azure/architecture/example-scenario/infrastructure/multi-tier-app-disaster-recovery



Best Practice for Monolithic Apps



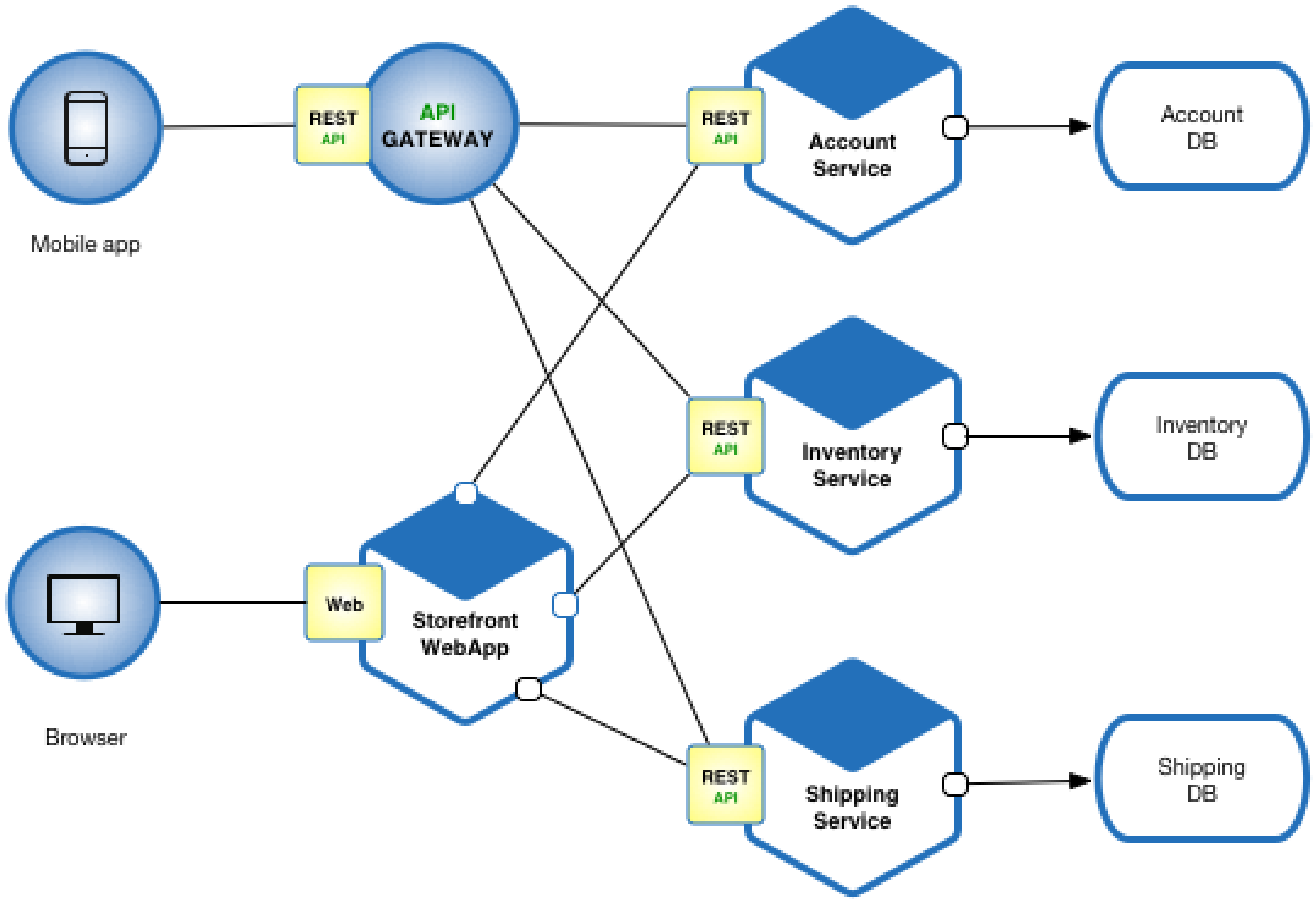
Microservices

E-Commerce Components

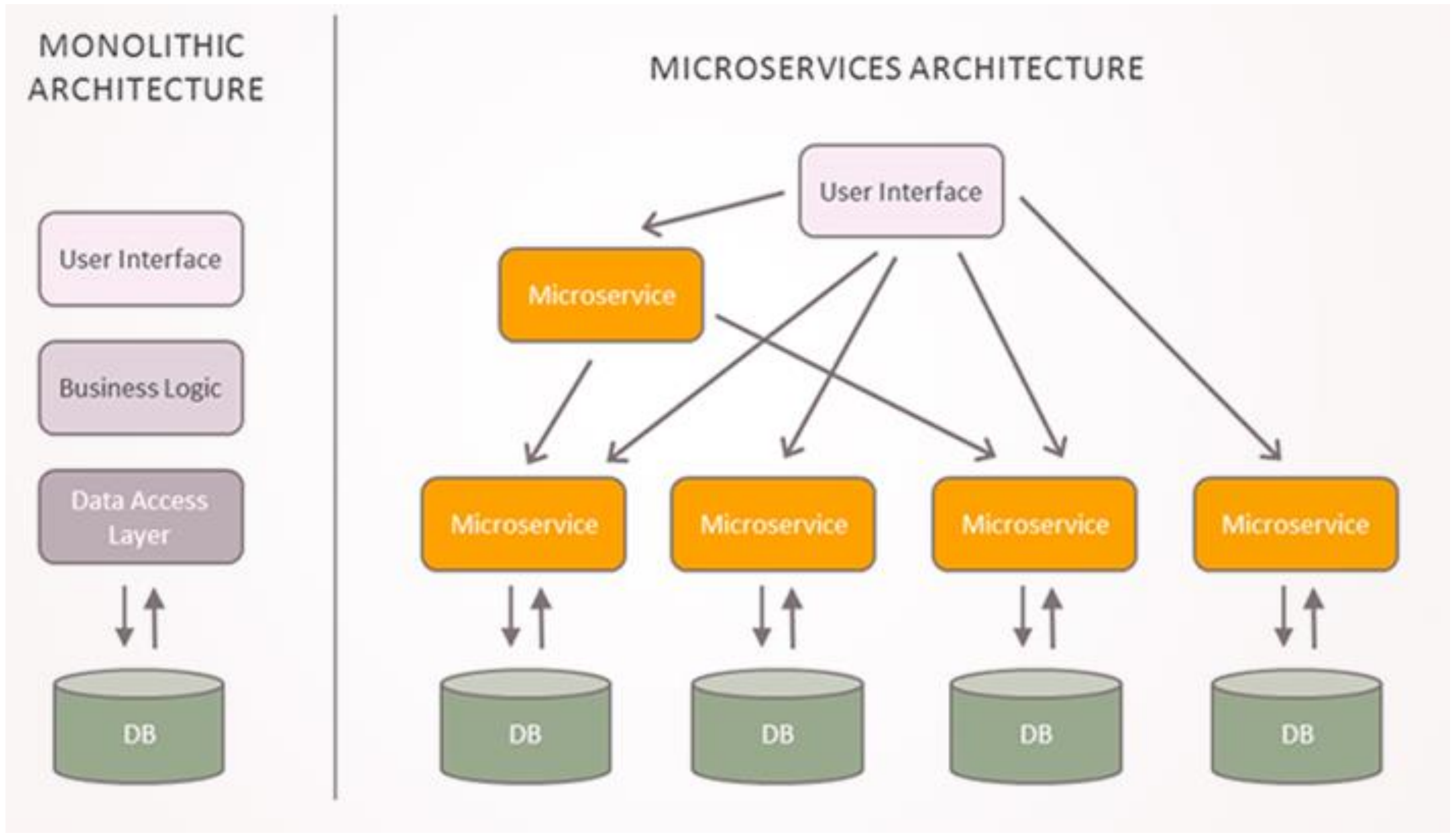
- Account (Identity) Management
- Shipping Management
- Inventory Management
- Order Management
- Marketing Management
- Supplier Management
- Etc.

Cross Cutting Concern

Microservices



Monolith vs Microservice



Why Microservices

https://dzone.com/articles/why-microservices

https://docs.microsoft.com/en-us/azure/architecture/microservices/introduction

https://blog.newrelic.com/technology/microservices-what-they-are-why-to-use-them/

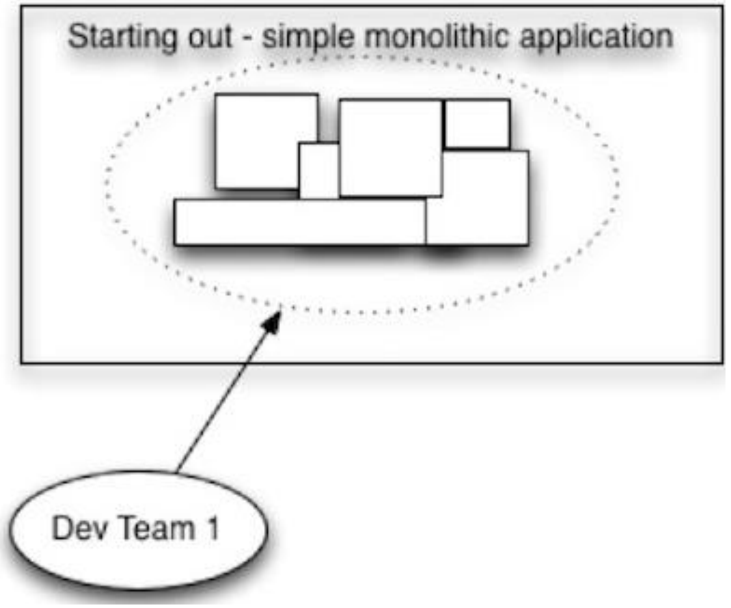
Concerns With the Monolith

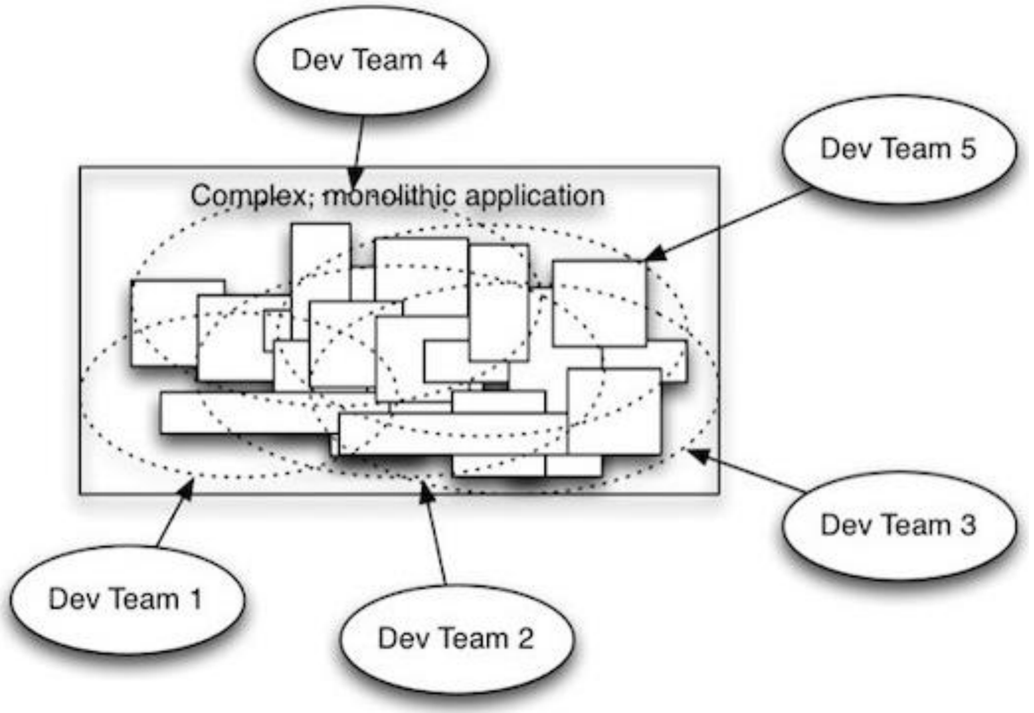
- Large applications that require a high release velocity.
- Complex applications that need to be highly scalable.
- Applications with rich domains or many subdomains.
- Difficult to Scale
- No Clear Ownership
- Failure Cascade
- Wall Between Dev and Ops
- Stuck in a Technology/Language

When switch to Microservices?

App Based Needs

https://blog.newrelic.com/technology/microservices-what-they-are-why-to-use-them/

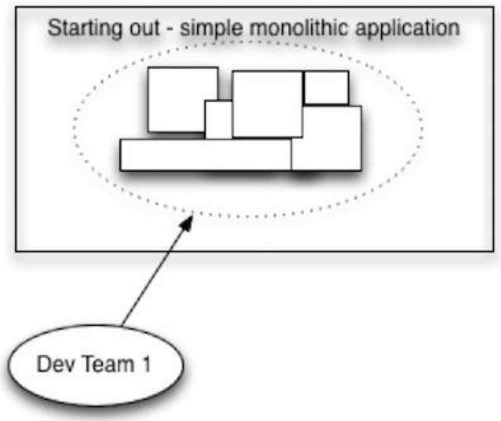




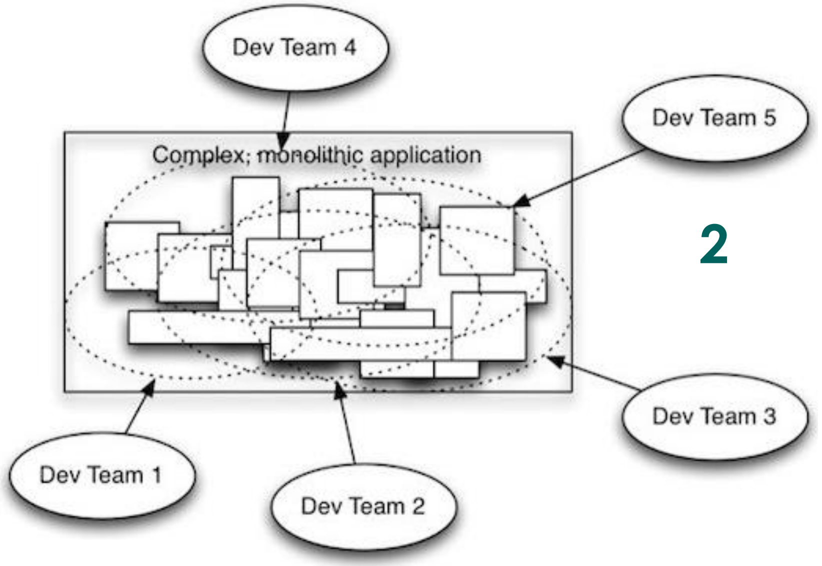
When switch to Microservices?

App Based Needs

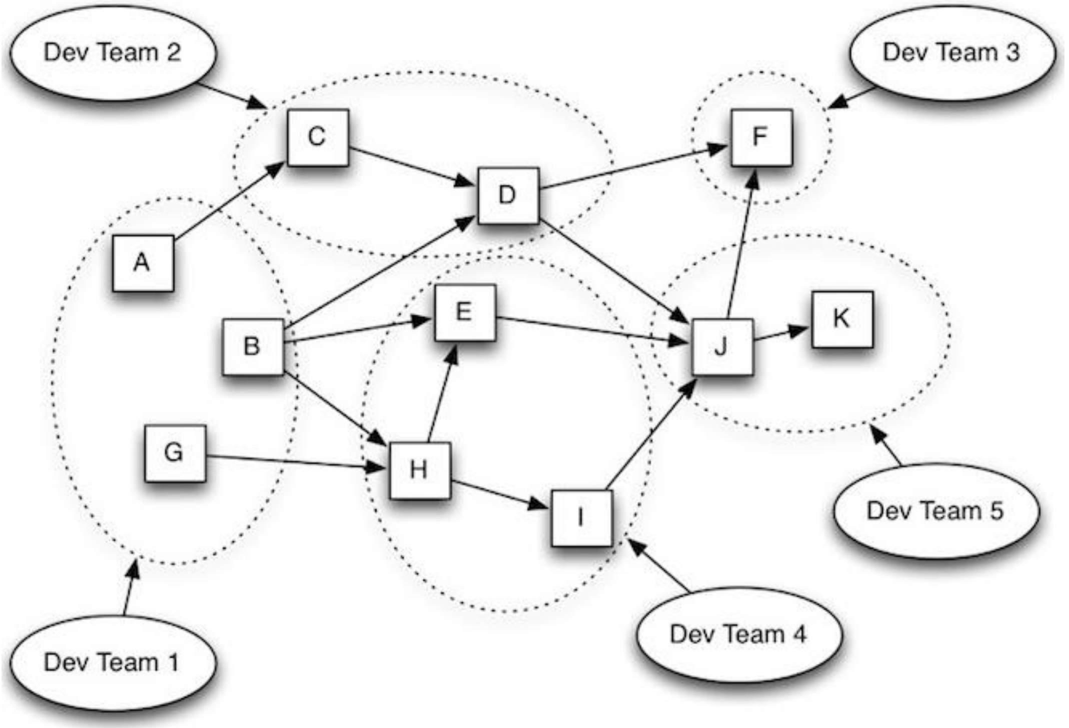
https://blog.newrelic.com/technology/microservices-what-they-are-why-to-use-them/



1



2

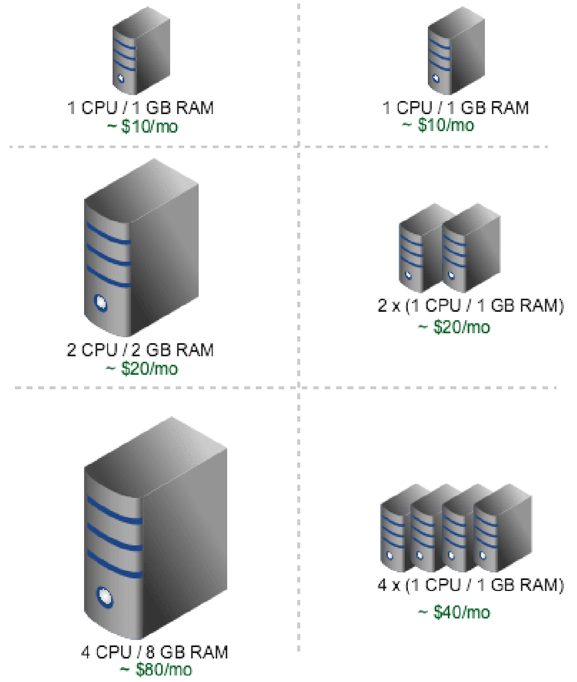


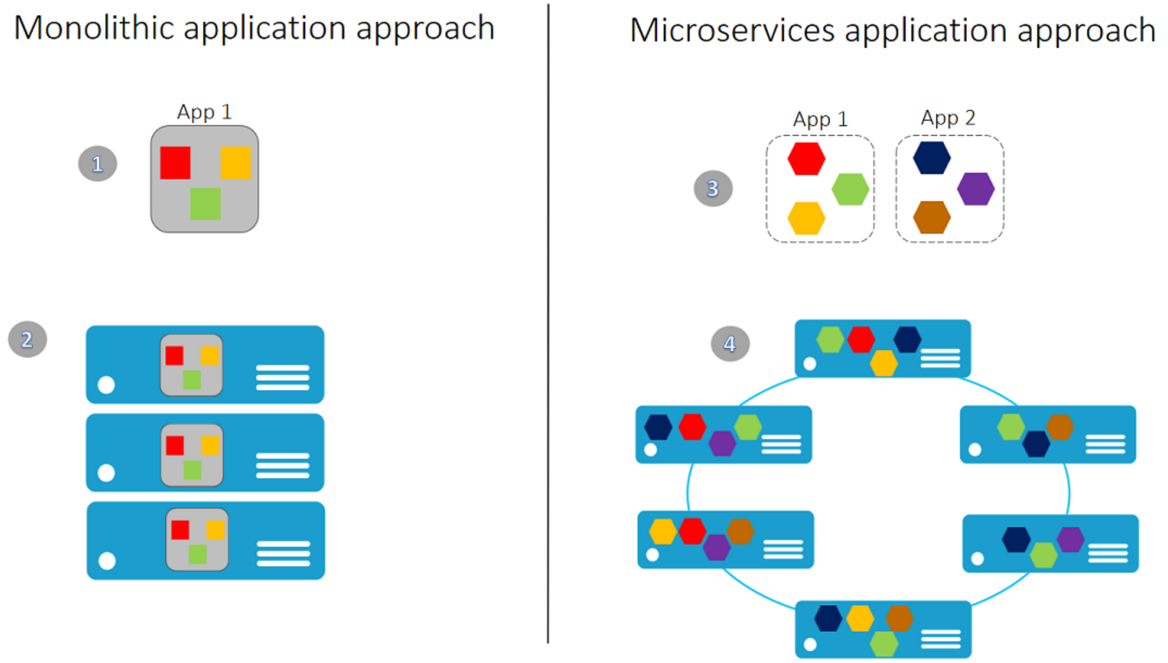
3

When switch to Microservices?

Hardware Based Needs

https://blog.newrelic.com/technology/microservices-what-they-are-why-to-use-them/





Monolithic application approach
App 1
1
2
Microservices application approach
App 1
App 2
3
4

Why Microservices

https://dzone.com/articles/why-microservices

https://docs.microsoft.com/en-us/azure/architecture/microservices/introduction

https://blog.newrelic.com/technology/microservices-what-they-are-why-to-use-them/

https://www.edureka.co/blog/top-10-reasons-to-learn-microservices/

The Benefits Microservices

- **Improved granular scalability.** Because microservices let you independently scale services up or down, the ease—and cost—of scaling is dramatically less than in a monolithic system. Adding new capabilities usually means adding discrete new microservices, not redoing the entire application, which increases both development speed and application stability.
- **Better fault isolation (Reduces Risks).** If one microservice fails, all the others will likely continue to work. This is a key part of the microservices architectural design.
- **Optimized scaling decisions.**With microservice architectures, scaling decisions can be made at a more granular level, allowing more efficient system optimization and organization.
- **Localized complexity.** Microservice architectures let developers think about services as black boxes. Owners of the service need to understand the complexity of only what is within their service. Other service owners need to know only what capabilities a service provides, without having to worry about how it works internally. This compartmentalization of knowledge and complexity makes it easier to create and manage large applications.
- **Increased business agility.** Microservices are relatively small and simple, and failure of a microservice affects only that service—not the whole application—so enterprises can afford to experiment with new processes, algorithms, and business logic. Microservices give you the freedom to experiment and “fail fast.”
- **Increased developer productivity.** New developers can get up to speed rapidly, since it’s easier to understand a small, isolated piece of functionality than an entire monolithic application.
- **Simplified debugging and maintenance.** For the same reasons that building individual microservices is easier than coding for a monolithic architecture, developers can be much more productive when debugging code and performing maintenance.
- **Better alignment of developers with business users.** Since microservice architectures are organized around business capabilities, developers can more easily understand the user perspective and create microservices that are better aligned with the business.
- **Future-proofed applications.** When innovations happen and new or updated technology disrupt your software development process, microservice architectures makes it easier to respond by replacing or upgrading the individual services affected without impacting the whole application.
- **Smaller and more agile development teams.** In modern software organizations, teams are often organized by the microservices they work on. These teams involve fewer people, and they’re more focused on the task at hand.
- **Promotes the best big data practices**
- **Flexibility to use various tools for the required task**
- **Provides Continous Delivery**

Challenges

https://docs.microsoft.com/en-us/azure/architecture/microservices/introduction

https://blog.newrelic.com/technology/microservices-what-they-are-why-to-use-them/

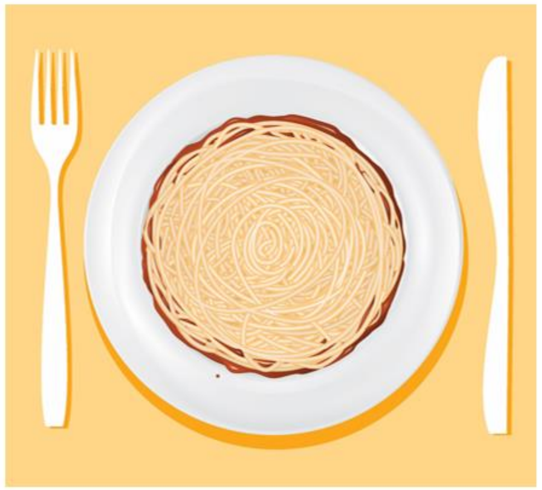
Challenges

The benefits of microservices don't come for free. Here are some of the challenges to consider before embarking on a microservices architecture.

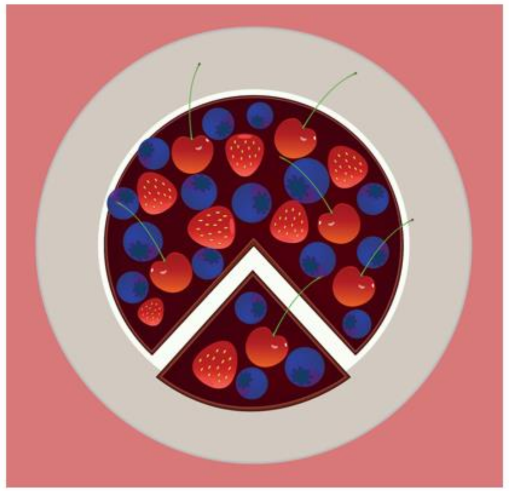
•**Complexity**. A microservices application has more moving parts than the equivalent monolithic application. Each service is simpler, but the entire system as a whole is more complex.
•**Development and testing**. Writing a small service that relies on other dependent services requires a different approach than a writing a traditional monolithic or layered application. Existing tools are not always designed to work with service dependencies. Refactoring across service boundaries can be difficult. It is also challenging to test service dependencies, especially when the application is evolving quickly.
•**Lack of governance**. The decentralized approach to building microservices has advantages, but it can also lead to problems. You may end up with so many different languages and frameworks that the application becomes hard to maintain. It may be useful to put some project-wide standards in place, without overly restricting teams' flexibility. This especially applies to cross-cutting functionality such as logging.
•**Network congestion and latency**. The use of many small, granular services can result in more interservice communication. Also, if the chain of service dependencies gets too long (service A calls B, which calls C...), the additional latency can become a problem. You will need to design APIs carefully. Avoid overly chatty APIs, think about serialization formats, and look for places to use asynchronous communication patterns.
•**Data integrity**. With each microservice responsible for its own data persistence. As a result, data consistency can be a challenge. Embrace eventual consistency where possible.
•**Management**. To be successful with microservices requires a mature DevOps culture. Correlated logging across services can be challenging. Typically, logging must correlate multiple service calls for a single user operation.
•**Versioning**. Updates to a service must not break services that depend on it. Multiple services could be updated at any given time, so without careful design, you might have problems with backward or forward compatibility.
•**Skillset**. Microservices are highly distributed systems. Carefully evaluate whether the team has the skills and experience to be successful.
•Cross-cutting Concerns Across Each Service

Monolithic vs. Microservices And SOA

https://nordicapis.com/should-you-start-with-a-monolith-or-microservices/
https://dev.to/alex_barashkov/microservices-vs-monolith-architecture-4l1m



With monolithic, tightly coupled applications, all changes must be pushed at once, making continuous deployment impossible.



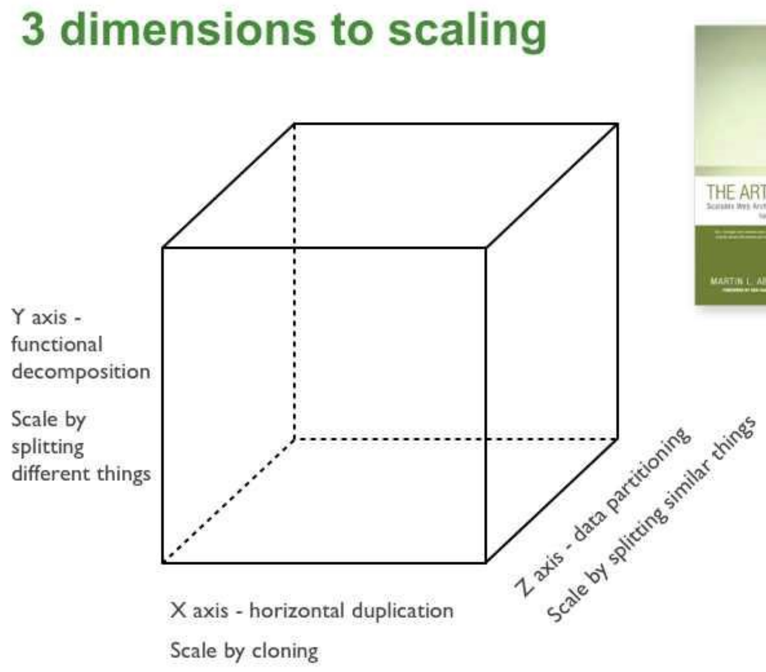
Traditional SOA allows you to make changes to individual pieces. But each piece must be carefully altered to fit into the overall design.

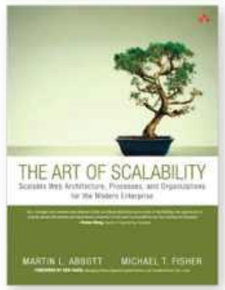
With a microservices architecture, developers create, maintain and improve new services independently, linking info through a shared data API.

Kanban Solutions @kanbansolutions kanbansolutions.com

The Scale Cube

https://microservices.io/articles/scalecube.html

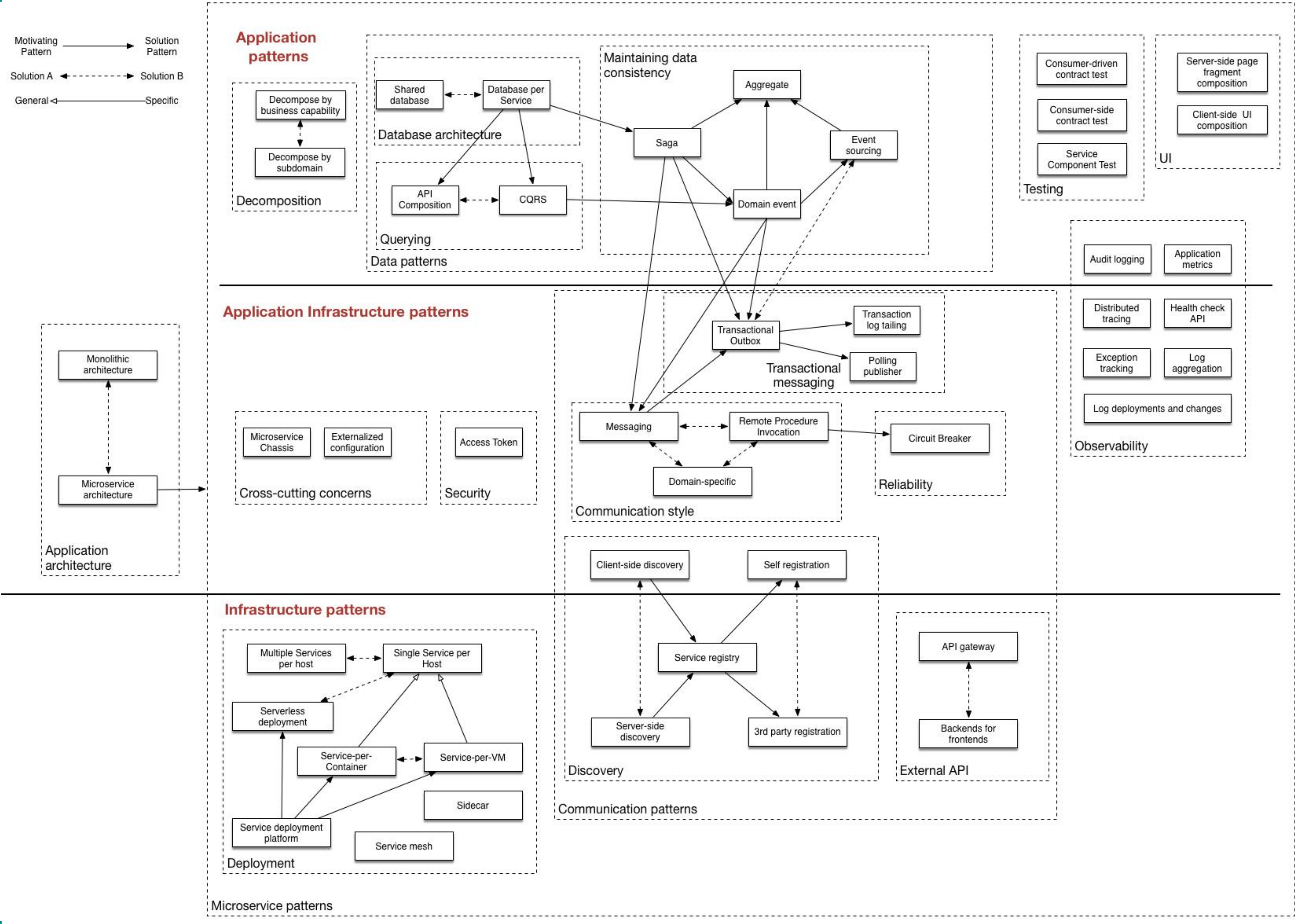

3 dimensions to scaling
Y axis - functional decomposition
Scale by splitting different things
X axis - horizontal duplication
Scale by cloning
Z axis - data partitioning
Scale by splitting similar things



X-axis scaling
X-axis scaling consists of running multiple copies of an application behind a load balancer. If there are N copies then each copy handles 1/N of the load. This is a simple, commonly used approach of scaling an application. One drawback of this approach is that because each copy potentially accesses all of the data, caches require more memory to be effective. Another problem with this approach is that it does not tackle the problems of increasing development and application complexity.
Y-axis scaling
Unlike X-axis and Z-axis, which consist of running multiple, identical copies of the application, Y-axis axis scaling splits the application into multiple, different services. Each service is responsible for one or more closely related functions. There are a couple of different ways of decomposing the application into services. One approach is to use verb-based decomposition and define services that implement a single use case such as checkout. The other option is to decompose the application by noun and create services responsible for all operations related to a particular entity such as customer management. An application might use a combination of verb-based and noun-based decomposition.
Z-axis scaling
When using Z-axis scaling each server runs an identical copy of the code. In this respect, it’s similar to X-axis scaling. The big difference is that each server is responsible for only a subset of the data. Some component of the system is responsible for routing each request to the appropriate server. One commonly used routing criteria is an attribute of the request such as the primary key of the entity being accessed. Another common routing criteria is the customer type. For example, an application might provide paying customers with a higher SLA than free customers by routing their requests to a different set of servers with more capacity.

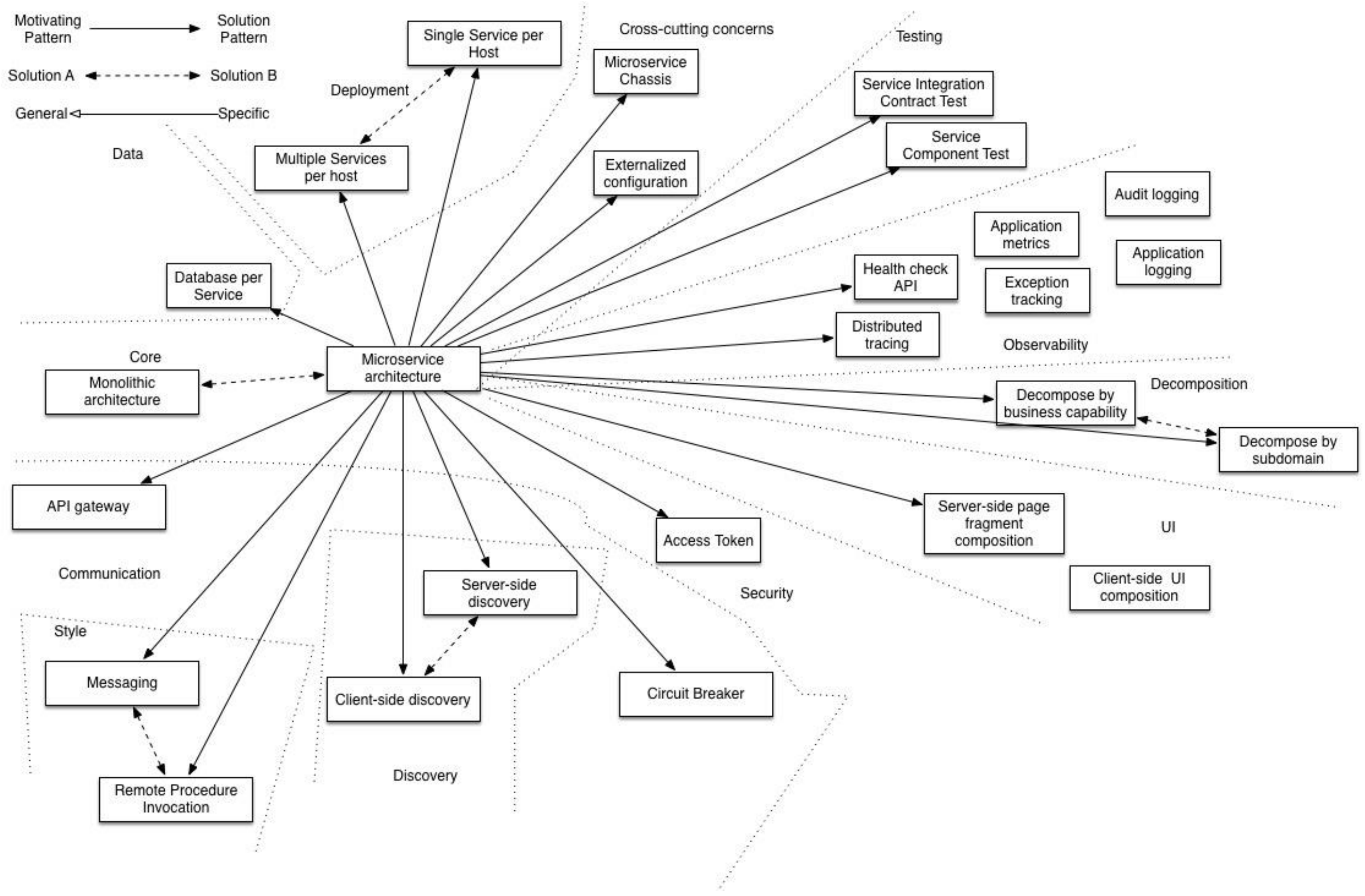
A pattern language for Microservices

https://microservices.io/patterns/index.html



Microservice Patterns

https://microservices.io/patterns/microservices.html



A pattern language for Microservices

https://microservices.io/patterns/index.html

Application architecture patterns
Which architecture should you choose for an application?
•Monolithic architecture - architect an application as a single deployable unit
•Microservice architecture - architect an application as a collection of loosely coupled, services

Decomposition
How to decompose an application into services?
•Decompose by business capability - define services corresponding to business capabilities
•Decompose by subdomain - define services corresponding to DDD subdomains

Deployment patterns
How to deploy an application's services?
•Multiple service instances per host - deploy multiple service instances on a single host
•Service instance per host - deploy each service instance in its own host
•Service instance per VM - deploy each service instance in its VM
•Service instance per Container - deploy each service instance in its container
•Serverless deployment - deploy a service using serverless deployment platform
•Service deployment platform - deploy services using a highly automated deployment platform that provides a service abstraction

Cr**oss cutting concerns**
How to handle cross cutting concerns?
•Microservice chassis - a framework that handles cross-cutting concerns and simplifies the development of services
•Externalized configuration - externalize all configuration such as database location and credentials

A pattern language for Microservices

https://microservices.io/patterns/index.html

Communication patterns

Style
Which communication mechanisms do services use to communicate with each other and their external clients?
- Remote Procedure Invocation - use an RPI-based protocol for inter-service communication
- Messaging - use asynchronous messaging for inter-service communication
- Domain-specific protocol - use a domain-specific protocol

External API
How do external clients communicate with the services?
- API gateway - a service that provides each client with unified interface to services
- Backend for front-end - a separate API gateway for each kind of client

Service discovery
How does the client of an RPI-based service discover the network location of a service instance?
- Client-side discovery - client queries a service registry to discover the locations of service instances
- Server-side discovery - router queries a service registry to discover the locations of service instances
- Service registry - a database of service instance locations
- Self registration - service instance registers itself with the service registry
- 3rd party registration - a 3rd party registers a service instance with the service registry

Reliability
How to prevent a network or service failure from cascading to other services?
- Circuit Breaker - invoke a remote service via a proxy that fails immediately when the failure rate of the remote call exceeds a threshold

Transactional messaging
How to publish messages as part of a database transaction?
- Transactional outbox
- Transaction log tailing
- Polling publisher

A pattern language for Microservices

https://microservices.io/patterns/index.html

Data management
How to maintain data consistency and implement queries?
•Database per Service - each service has its own private database
•Shared database - services share a database
•Saganew - use sagas, which a sequences of local transactions, to maintain data consistency across services
•API Compositionnew - implement queries by invoking the services that own the data and performing an in-memory join
•CQRS - implement queries by maintaining one or more materialized views that can be efficiently queried
•Event sourcing - persist aggregates as a sequence of events

Security
How to communicate the identity of the requestor to the services that handle the request?
•Access Token - a token that securely stores information about user that is exchanged between services

Testing
How to make testing easier?
•Consumer-driven contract test - a test suite for a service that is written by the developers of another service that consumes it
•Consumer-side contract test - a test suite for a service client (e.g. another service) that verifies that it can communicate with the service
•Service component sest - a test suite that tests a service in isolation using test doubles for any services that it invokes

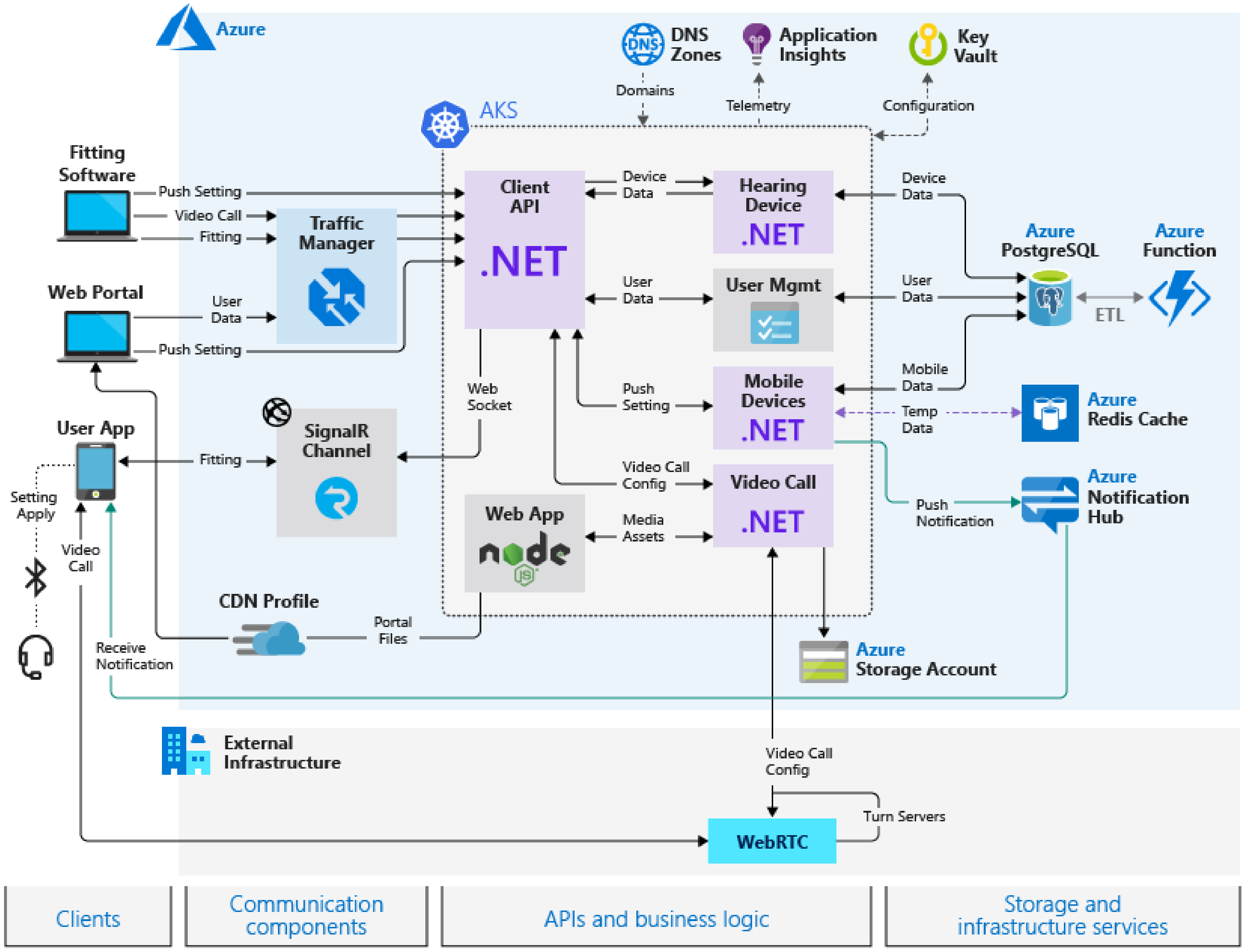
Observability
How to understand the behavior of an application and troubleshoot problems?
•Log aggregation - aggregate application logs
•Application metrics - instrument a service's code to gather statistics about operations
•Audit logging - record user activity in a database
•Distributed tracing - instrument services with code that assigns each external request an unique identifier that is passed between services. Record information (e.g. start time, end time) about the work (e.g. service requests) performed when handling the external request in a centralized service
•Exception tracking - report all exceptions to a centralized exception tracking service that aggregates and tracks exceptions and notifies developers.
•Health check API - service API (e.g. HTTP endpoint) that returns the health of the service and can be pinged, for example, by a monitoring service
•Log deployments and changesnew

UI patterns
How to implement a UI screen or page that displays data from multiple services?
•Server-side page fragment composition - build a webpage on the server by composing HTML fragments generated by multiple, business capability/subdomain-specific web applications
•Client-side UI composition - Build a UI on the client by composing UI fragments rendered by multiple, business capability/subdomain-specific UI components

Azure Architecture Center

Telehealth System

https://docs.microsoft.com/en-us/azure/architecture/example-scenario/apps/telehealth-system



Serverless

https://martinfowler.com/articles/serverless.html

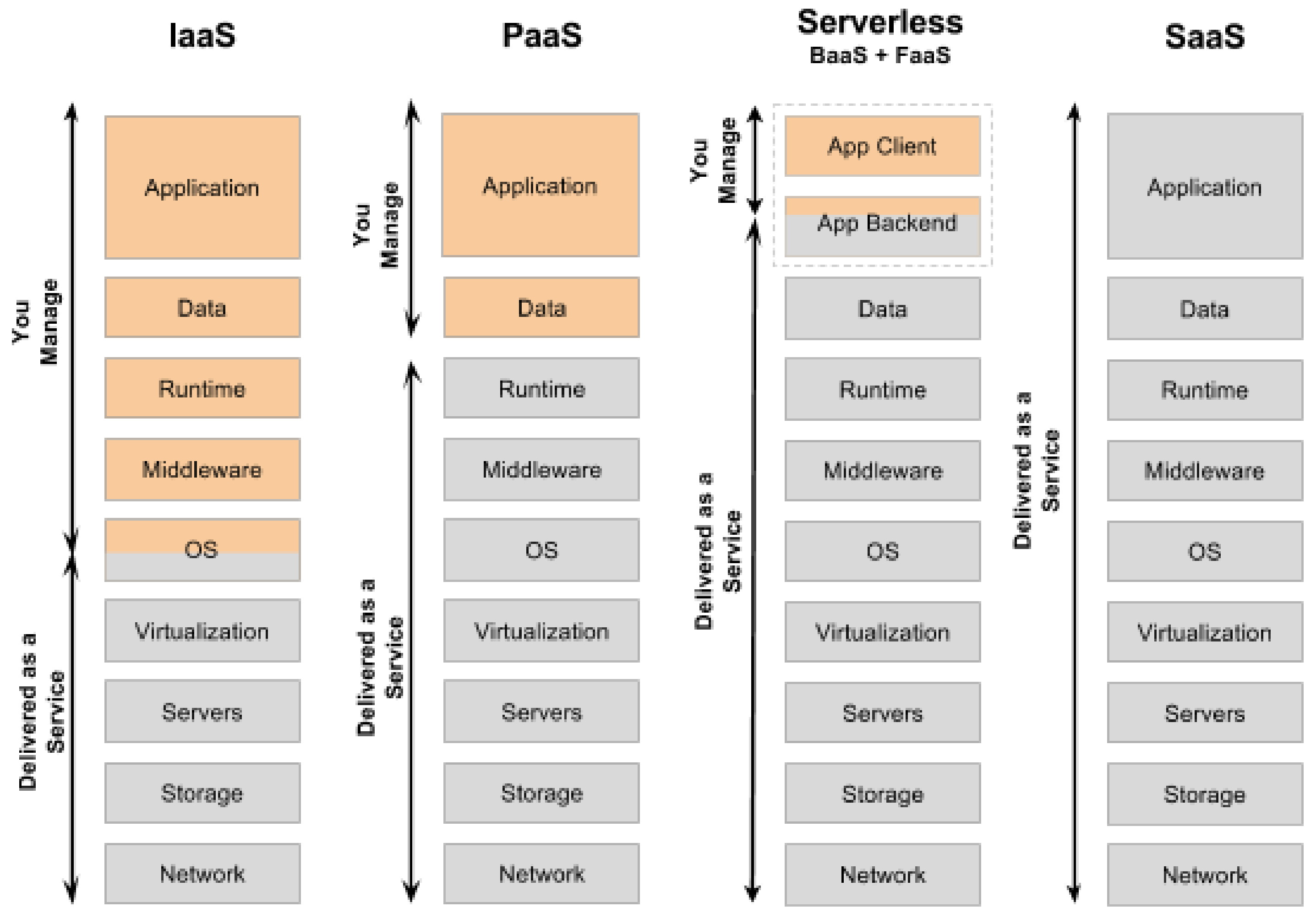
Serverless architectures are application designs that incorporate third-party “Backend as a Service” (BaaS) services, and/or that include custom code run in managed, ephemeral containers on a “Functions as a Service” (FaaS) platform. By using these ideas, and related ones like single-page applications, such architectures remove much of the need for a traditional always-on server component. Serverless architectures may benefit from significantly reduced operational cost, complexity, and engineering lead time, at a cost of increased reliance on vendor dependencies and comparatively immature supporting services.

Serverless computing (or serverless for short), is an execution model where the cloud provider (AWS, Azure, or Google Cloud) is responsible for executing a piece of code by dynamically allocating the resources. And only charging for the amount of resources used to run the code. The code is typically run inside stateless containers that can be triggered by a variety of events including http requests, database events, queuing services, monitoring alerts, file uploads, scheduled events (cron jobs), etc. The code that is sent to the cloud provider for execution is usually in the form of a function. Hence serverless is sometimes referred to as *“Functions as a Service”* or *“FaaS”*. Following are the FaaS offerings of the major cloud providers:

- AWS: AWS Lambda
- Microsoft Azure: Azure Functions
- Google Cloud: Cloud Functions

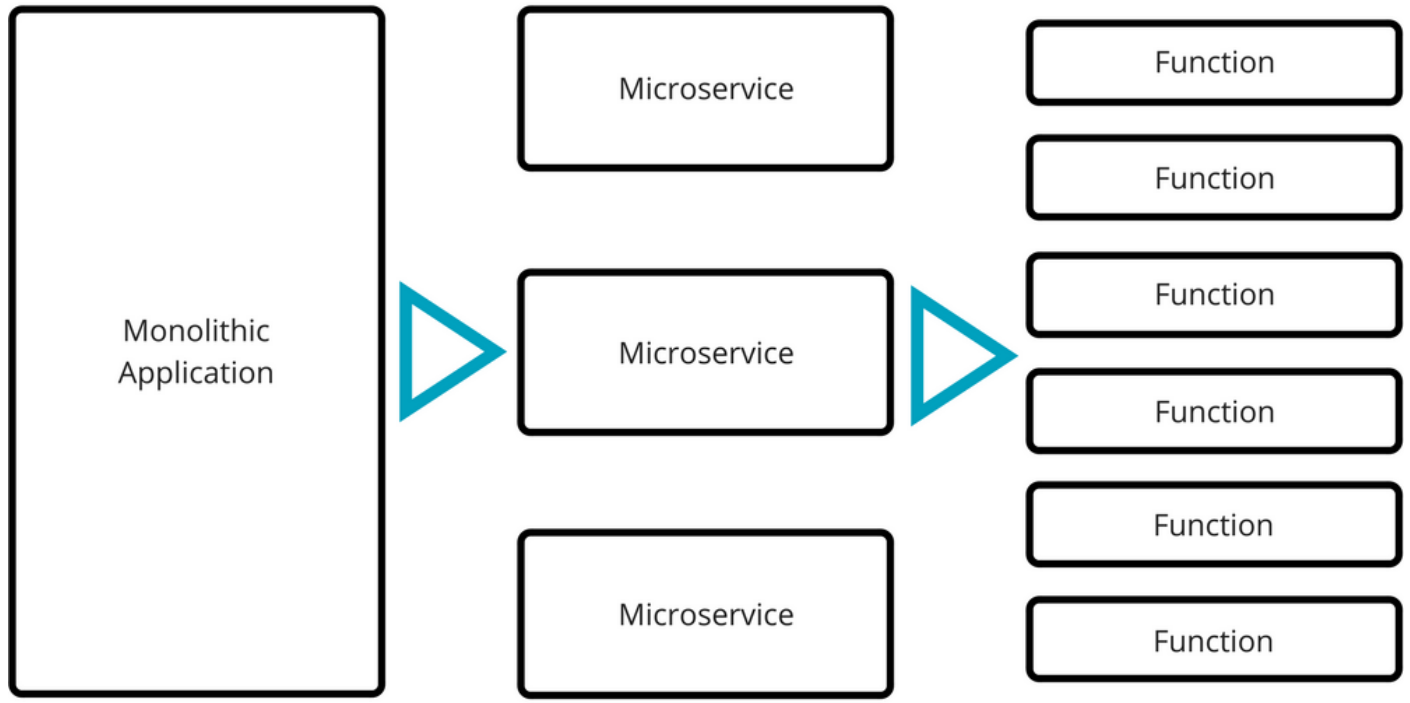
Serverless

https://specify.io/concepts/serverless-baas-faas



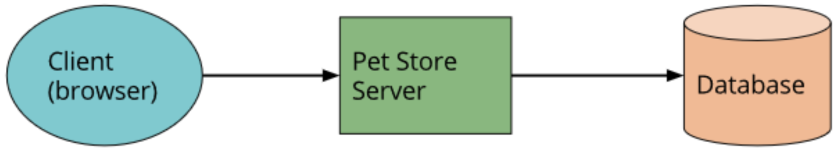
Serverless

https://specify.io/concepts/serverless-baas-faas

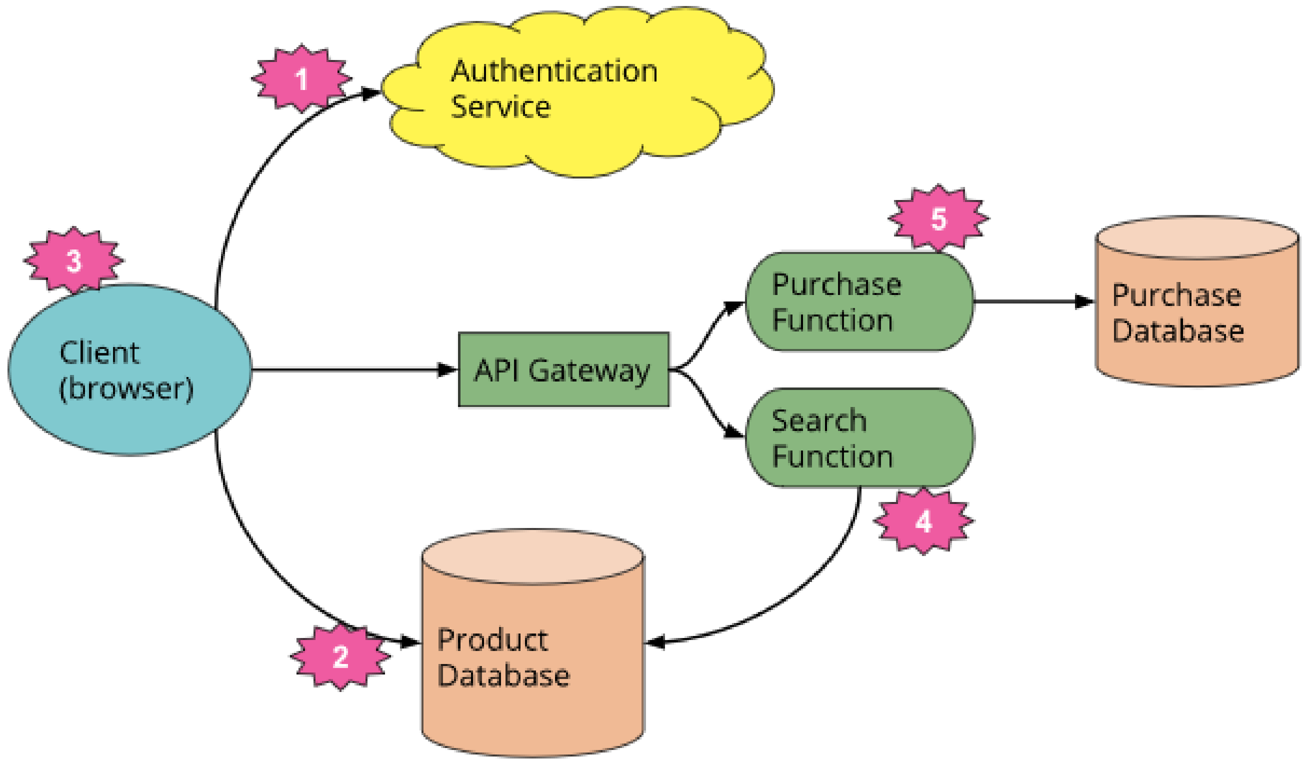


Serverless

https://martinfowler.com/articles/serverless.html



1. Third-party BaaS service (e.g., Auth0.)
2. Google Firebase, AzureCosmosDB
3. Server side rendering (Single Page App)
4. FaaS function that responds to HTTP requests via an API gateway
5. “purchase” functionality with another separate FaaS function



Serverless

https://martinfowler.com/articles/serverless.html

This is a massively simplified view, but even here we see a number of significant changes:

- We've deleted the authentication logic in the original application and have replaced it with a third-party BaaS service (e.g., Auth0.)
- Using another example of BaaS, we've allowed the client direct access to a subset of our database (for product listings), which itself is fully hosted by a third party (e.g., Google Firebase.) We likely have a different security profile for the client accessing the database in this way than for server resources that access the database.
- These previous two points imply a very important third: some logic that was in the Pet Store server is now within the client—e.g., keeping track of a user session, understanding the UX structure of the application, reading from a database and translating that into a usable view, etc. The client is well on its way to becoming a Single Page Application.
- We may want to keep some UX-related functionality in the server, if, for example, it's compute intensive or requires access to significant amounts of data. In our pet store, an example is "search." Instead of having an always-running server, as existed in the original architecture, we can instead implement a FaaS function that responds to HTTP requests via an API gateway (described later). Both the client and the server "search" function read from the same database for product data.

If we choose to use AWS Lambda as our FaaS platform we can port the search code from the original Pet Store server to the new Pet Store Search function without a complete rewrite, since Lambda supports Java and Javascript—our original implementation languages.

- Finally, we may replace our "purchase" functionality with another separate FaaS function, choosing to keep it on the server side for security reasons, rather than reimplement it in the client. It too is fronted by an API gateway. Breaking up different logical requirements into separately deployed components is a very common approach when using FaaS.