# Architecture and Kubernetes

Understanding Kubernetes's place in software engineering

# An idea of what we will do

- We won't do Kubernetes in a vacuum
  - Don’t want to leave with a "and then"
- We will need to go from the bottom to understand its place
  - We start with system design, then move into architecture
- The goal is to understand:
  - What Kubernetes is
  - What its purpose is
  - What it can't do
  - When to use it
- Day 1 will not include much Kubernetes as its needed to set the stage
  - We will focus largely on a case study to help explain all the considerations needed for Kubernetes

# Overview

- A peak at Kubernetes
- When applications grow
- System Design
- Software Architecture
- Kubernetes theory

# A peak at Kubernetes

Covering just enough to get a starting point

# A peak at Kubernetes

## What is Kubernetes?

- Portable, open source, extensile platform
  - For managing **containerized** workloads and services
  - Made by Google, intended for **production scale**
  - Is an **orchestration** tool
- Provides declarative configuration and automation
  - You supply what you want (a desired state/configuration)
  - Kubernetes makes sure to meet and maintain that
- Will create new instances as needed, replace broken ones, and handle rollout updates
  - Basically does "docker run" for you (gross simplification)

# A peak at Kubernetes

## Why don’t we just dive in?

- Teaching Kubernetes without its context is like showing you how to use a hammer without telling you why and when its used.
  - And importantly, when its not needed!
- *If all you have is a hammer, everything looks like a nail*
- Learning about the context will help when learning more advanced aspects, or dealing with larger projects.
- It will also help preventing over-engineering solutions
  - Many start-ups fail because of this

# When applications grow

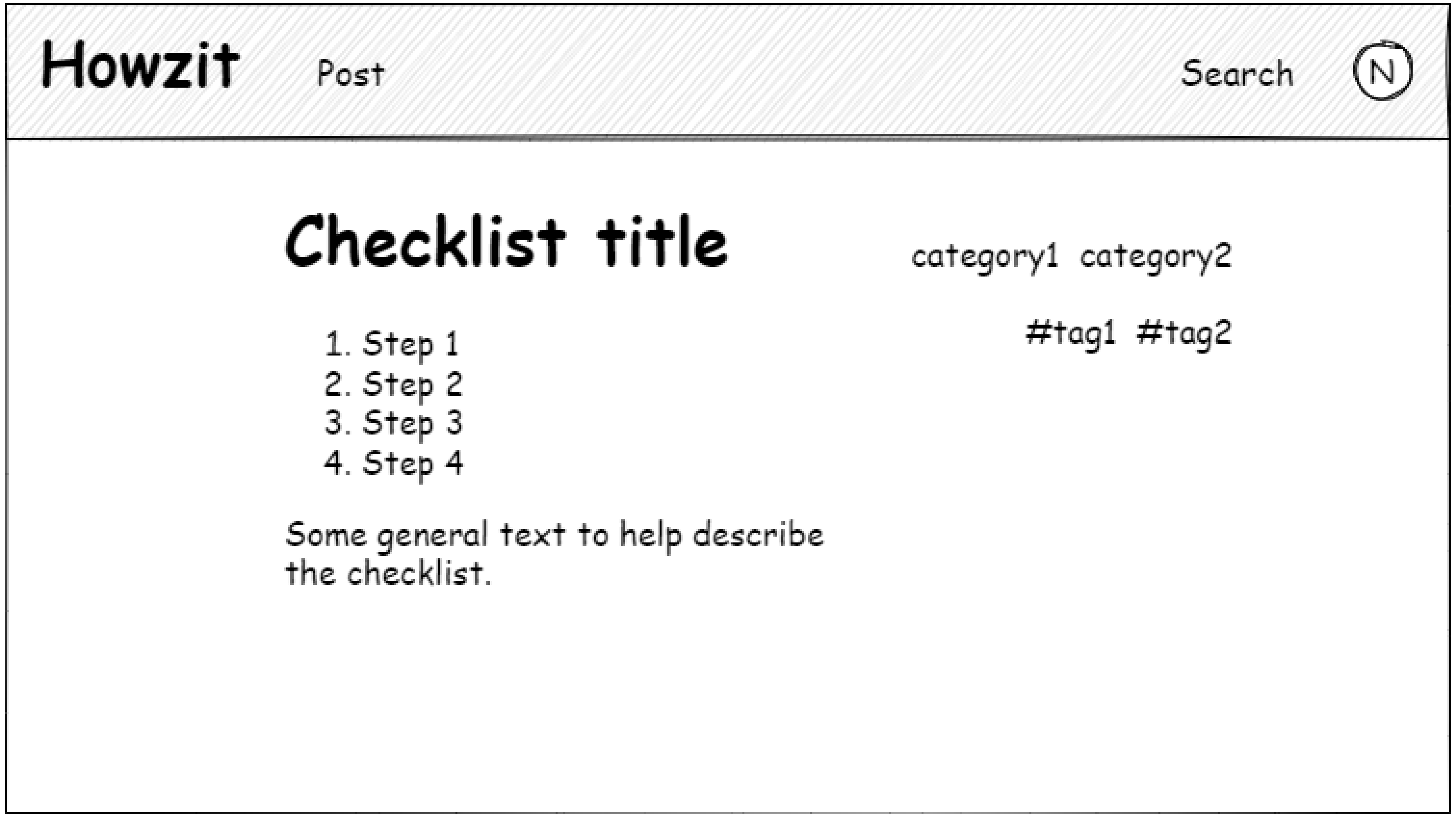
A case study of growing pains

# When applications grow

## Setting the scene

- Our Todo idea sparked something, a new app
- Like a Todo list, but backwards 👀
  - People post checklists for various things (basically how to guides in an ordered list)
  - These checklists can be for many categories
  - Users search for and favourite lists
    - Search based on category, date, title, and any tags
  - Users can also make their own lists to share with the community

# When applications grow



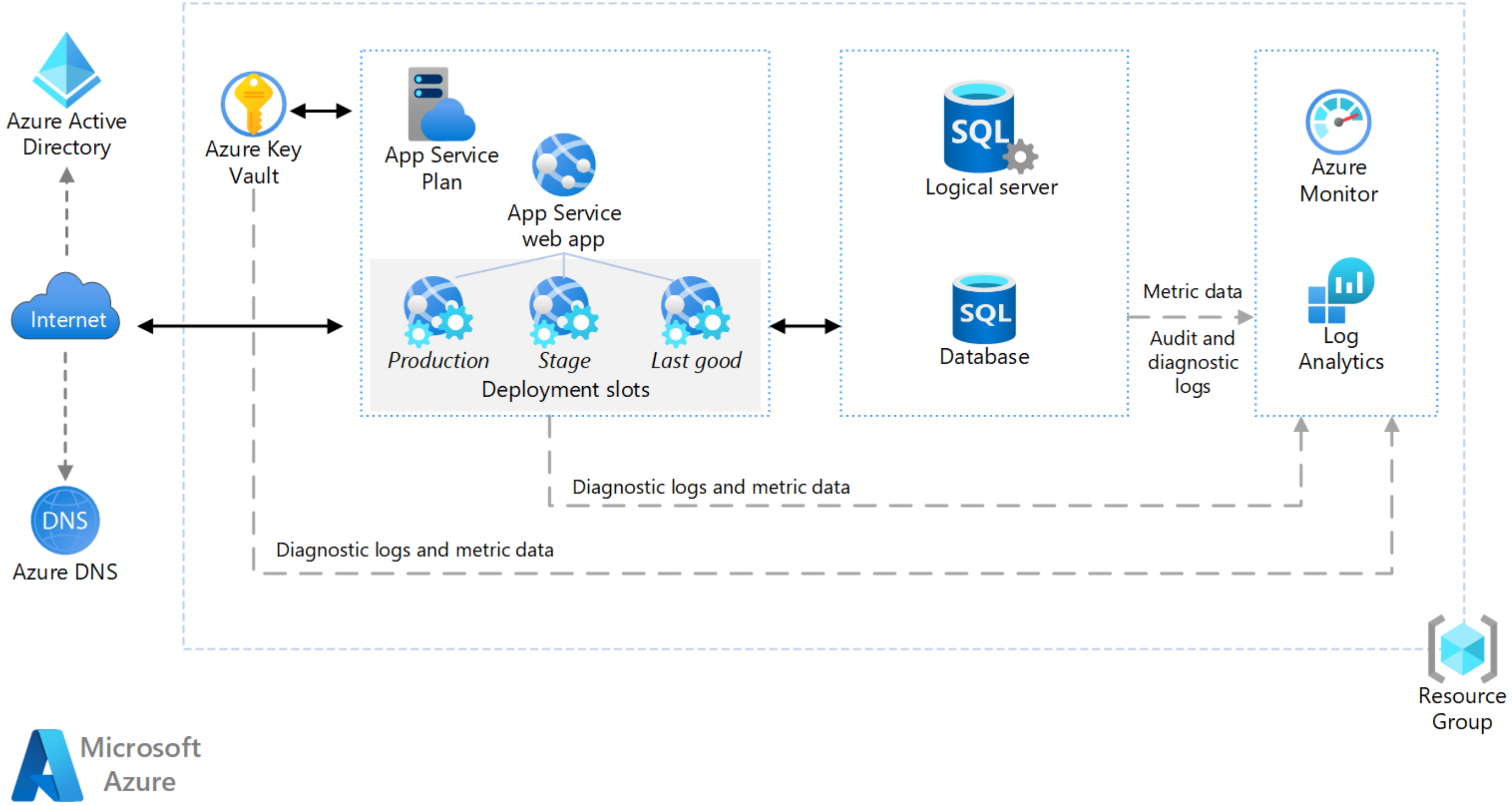

🔍 Mock UI (I know its comic sans)

# When applications grow

i A promising start

- We test it with your friends and colleagues, and it goes well
- We decide to host it on a cloud service to expose it to the public
  - We place the application in an Azure Web App
- We use an identity provider for authentication and authorization
  - The actual one doesn’t matter (Azure AD, Google, Auth0,…)

# When applications grow




Backend (link)

# When applications grow

## Feature requests

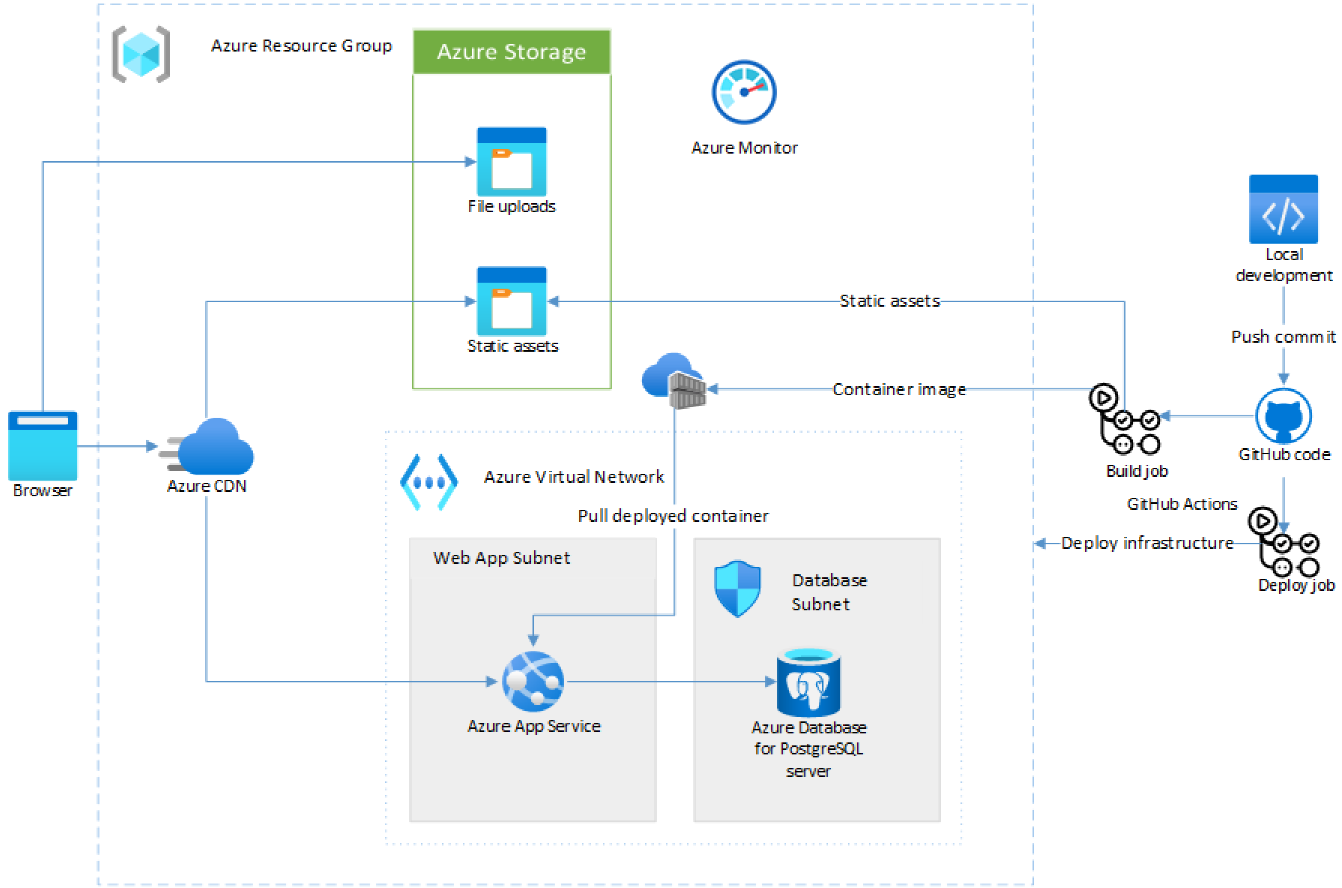
- As time goes on, users request some features
  - You also have some ideas of what to add
- The following is placed on backlog:
  - Add a rating system with a text portion
  - Add a comment system that will appear with the ratings (they get mixed up, but can filter)
  - Add media to checklists (associated videos and images)
  - Add a discovery page that shows you checklists you could be interested in (users can define interests and not interested to help the algorithm).

# When applications grow

## Rolling out the features

- We need to figure out a way to update the application as fast as possible and in the least likely to fail way
  - Also need to consider multiple devs working on it
- Containers and pipelines help here
  - Basically, adapting some aspects of DevOps to your project
- One issue remains – media
- *How would you look to store videos and images for checklists?*

# When applications grow




🔍 `Adopting the recommended method (`[link]()`)`

# When applications grow


New features

# When applications grow

**Its boom time!**

- Our new features resonate with the userbase
- They start sharing the app and using it daily
- The application was not ready for this
  - Users start to complain
- "It takes ages to load", "I must submit several times", "the service is down"
- We investigate the Web App and look at the usage
  - By using Azure Monitor, and monitoring some metrics

# When applications grow

i Bottlenecks rise up!

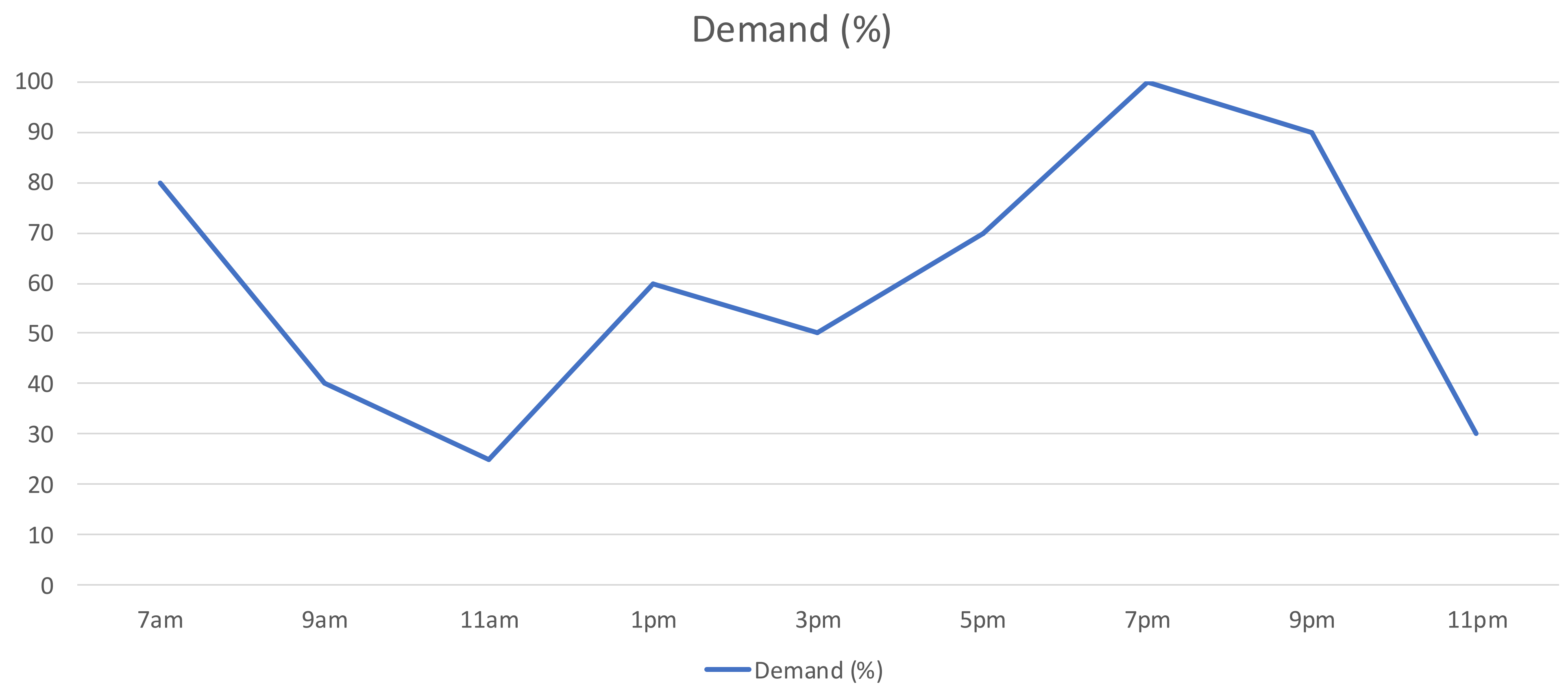
- We inspect your two services
  - Web App and Database
- We see your web app is spiking up to 100% utilization and staying there for hours
  - This also coincides with your users’ complaints
- The database is still fine
- This means our Web App is the bottleneck
- *How would you go about solving this?*

# When applications grow

## i Scalability

- Scalability is the property of a system to handle a growing amount of work by adding resources to the system.
- This can be done in many ways, but simply put:
  - You can get a bigger computer (scale up)
  - You can get more computers (scale out)
- This is known as vertical and horizontal scaling
  - Each have their own benefits and drawbacks
  - They often as used together to get more benefits

# When applications grow




🔍 Demand throughout the day

# When applications grow

## Vertical scaling

- Adding/removing resources, typically involving the addition of CPUs, memory or storage to a single computer.
  - In Azure you scale up by changing the price tier
- This raises the “ceiling” of your capacity
- Vertical scaling wont react to your needs
  - It caters for the peak and stays there
- Essentially “throw money at it until its not a problem”
- *What are some issues you can see with this?*

# When applications grow

## i Challenges with vertical scaling

- Catering only for peaks can increase costs
- Single point of failure
  - High risk of downtime
  - High risk of hardware failures and outages
  - Azure helps with this (they need to meet SLAs)
- Doesn’t actually scale well
  - Hardware limit

# When applications grow

## Horizontal scaling

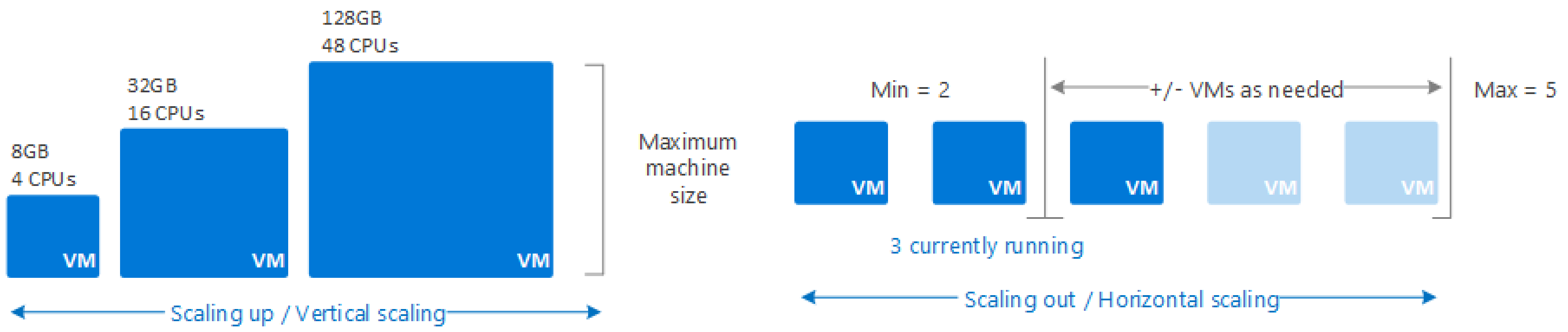
- When a hardware limit is reached, need to consider other methods of scaling
- Scale out (horizontal) allows you to have multiple instances of an application
- This splits the load, but you need a dedicated load balancer
  - Which adds complication
- Azure has many options for scaling out, and even implementing autoscaling
  - Autoscaling allows you to meet demand as it changes

# When applications grow

## Challenges with horizontal scaling

- More complicated to manage
  - At minimum you need a load balancer
  - It can get wildly more complicated than this
- Have increased latency due to internal network calls
  - Inter-process communication is faster (single instance)
- Can introduce data inconsistency
  - When scaling databases or stateful applications (will cover later)

# When applications grow




Scale up and out ([source]())

# When applications grow

## Where does Kubernetes fit in?

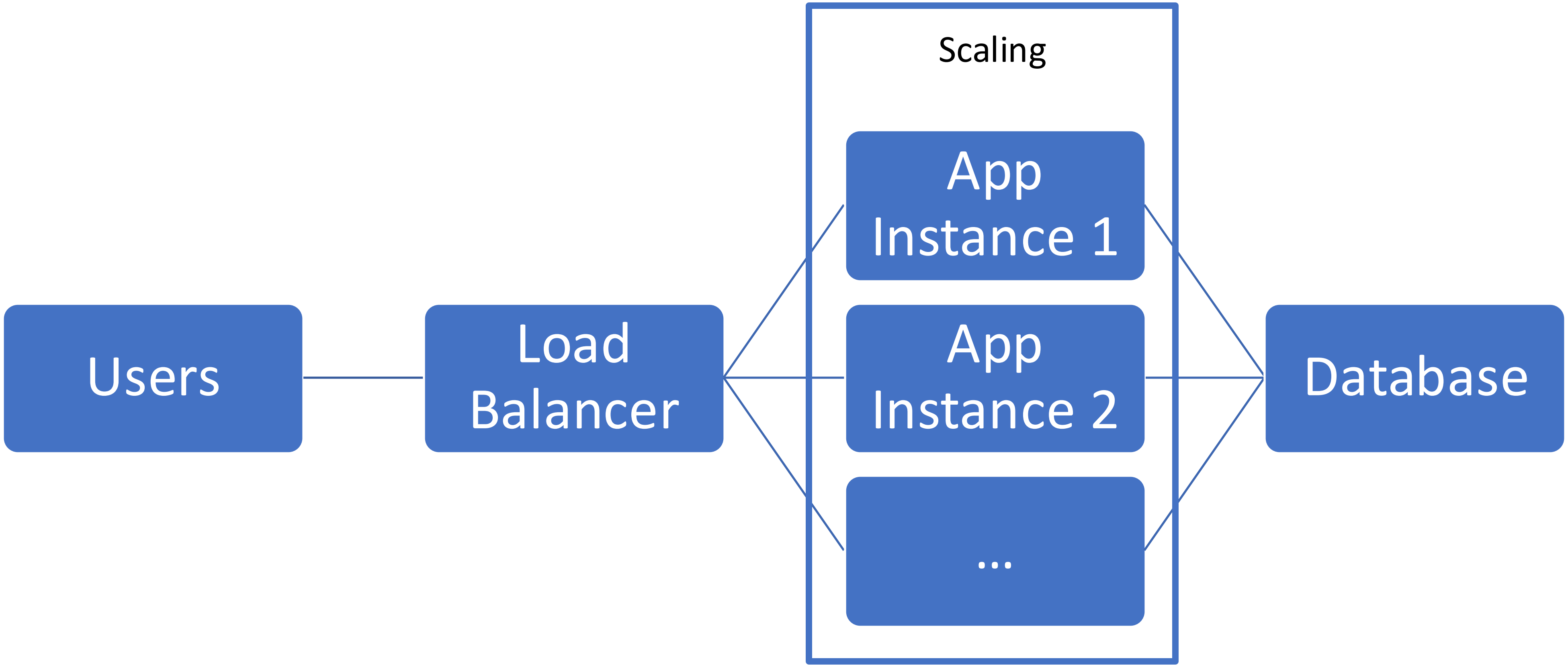
- When discussing scale out (more instances), we are discussing the “what”
- Kubernetes is one of the “how” solutions
  - It has autoscaling built in
  - It rolls out updates with no downtime
  - It can do a lot more than this
- Azure does help with scaling management with autoscaling
  - Comes at a cost and has vendor lock in
  - It also abstracts a lot away, limiting control
- *Basically, it can be used, but it's not needed. Why?*

# When applications grow

i Back to our scenario

- We implement some combination of vertical and horizontal scaling
- The average demand is higher (vertical helps)
  - You upgrade your pricing tier
- The demand fluctuates (horizontal helps)
  - We let Azure scale based on rules (when close to max util, make an instance)
- This helps reduce the problems, but not all

# When applications grow




`Our current application organisation`

# Any questions?

A checkpoint before we keep going

# When applications grow

## Our new bottleneck

- Our new scaling technique helps with performance
- However, we still are getting complaints around our peaks
- Investigating our Web App shows nothing that would alarm us, so we move on
- Investigating our database shows its being fully used
  - Makes sense, as we didn’t scale it up
- *How would you scale a database?*
  - *What are some challenges you see with scaling?*

# When applications grow

## How to scale databases?

- Vertically scaling databases is easy and the first option
- Horizontally scaling databases is a *bit* more tricky
- There are several factors that need to be considered when deciding on how to scale out databases
- *What do you think some considerations that need to be made?*
- *Are there ways to improve database performance without using more hardware?*

# When applications grow

i Non-hardware DB optimisations

- Improved indexing
    - Database indexing helps with searching values in a column
    - In a very basic sense, it's the same as an index in a book
    - Term "john" can be found on pages 25, 152, and 180.
    - Its more complicated than this, but outside of our scope
- Query optimisation
    - JOINs slow down queries heavily, try limit this
    - You can be fetching more data than needed and doing processing in the server
- Implementing connection pools
    - Some have it built in (Spring has Hikari)
- This all can be identified by looking at latency

# When applications grow

## Scaling by replication

- A form of horizontal scaling
- You replicate databases and route traffic (essentially load balance)
  - Normally in database clusters
  - Also, can just have a dedicated DB for each app instance
- This is a lot of effort and requires extra software to be written
  - Ensuring consistency of data

# When applications grow

## Problem with replication

- When we have distributed databases, we face something called the [CAP theorem].
- It's too long to explain in depth here, but it states that you can only have 2 of the 3 following features:
  - **Consistency** – reads receive the most recent data (or an error)
  - **Availability** – all requests return data (data could be old)
  - **Partition tolerance** – continued operation despite network failures
  - We always want tolerance, so it's really between C and A

# When applications grow

i `CAP considerations are complicated`

- This requires deep knowledge of distributed systems
  - Which will take far more time than a 2-day workshop
- So, for our use case, we will not consider this
  - However, the theory behind it is very relevant to what we need to cover
- I'm sure you're tired of hearing me speak, lets watch a 7 min video explaining CAP theorem and move on
  - https://www.youtube.com/watch?v=9SSvdLnmDiI&t=16s&ab_channel=MarkRichards

# When applications grow

i What replication options do we have?

- We will not look at how to achieve a distributed database solution
- We will instead just touch on some main ways replication can be done
  - Creating full copies
  - Creating read only replicas
  - Sharding our database
  - Caching results

# When applications grow

## Replication

- If we do a full copy, we have good consistency, but increased latency
- If our application is read-heavy, having read only copies helps distribute load
  - But will struggle with syncing
- Our application is read-heavy (people browsing and viewing checklists) so maybe this can help us

# When applications grow

## Caching

- A cache is essentially a fast datastore that is pre-queried
  - organised to give you what you need as fast as possible
- It helps decrease the network latency as you scale
- The main issue is that cache needs to be invalidated
  - This is not a trivial task at all
  - Redis is the most popular cache solution
- A nice video about cache

# When applications grow

## i Sharding

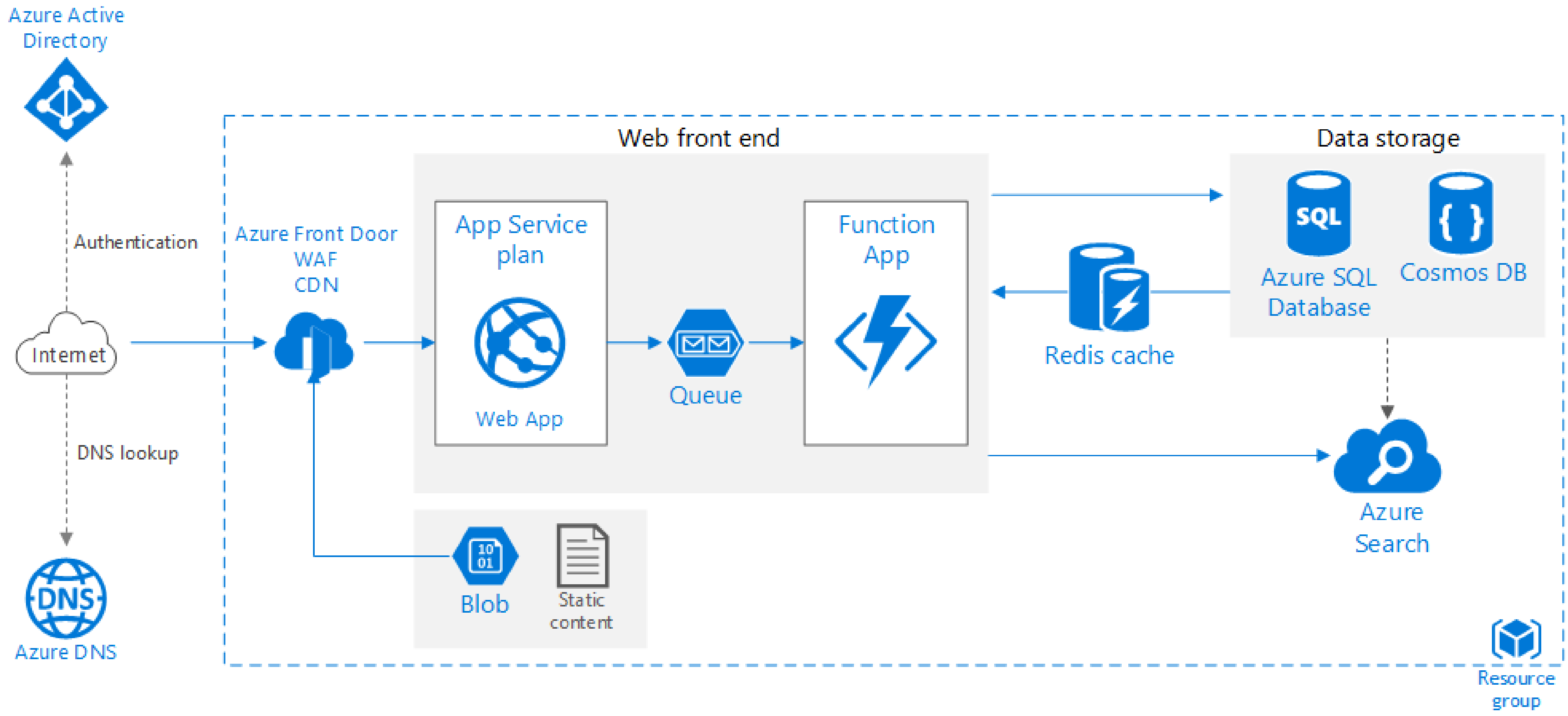
- This is simply splitting up your database over multiple databases or shards
- Many ways this can be done:
  - Vertical sharding (rows 1-1000, 1001-2000, or by hashcode)
  - Horizontal sharding (id + cols 2-5, id + cols 6-8)
  - Geo sharding
- The choice is yours, there are tools that help automatic sharding
- Microsoft doc on sharding

# When applications grow

`What do you think?`

- Our application is read-heavy
- We want to keep it as simple as possible for now
  - Can grow in complexity later with some help
- *Based on what we briefly looked at, what approach do you think we need?*
  - Replication, cache, sharding, or some combination?

# When applications grow




What does our overlord recommend?

# When applications grow

## i And Kubernetes?

- Amazingly, still not needed.
- Why did we spend the last several dozen slides not covering Kubernetes?
  - Simple, Kubernetes helps with **scaling**
  - However, it's not the only way to scale
  - We saw many, non-Kubernetes solutions (giving you more tools than the hammer)
- We also learnt some important lessons about scaling
  - Planning is important, and everything needs to scale together
- Good article on scaling
- Design to scale out by Microsoft

# Any questions?

The final checkpoint before moving on

# When applications grow

i `Where to from here?`

- We covered scaling and scalability, a non-functional requirement
- When considering scalability, we need to understand **system architecture**
    - What we have been seeing in articles and diagrams
- System architecture is part of **system design**
    - This is the context mentioned in the beginning

# Quiz 1: Scalability

Please complete Quiz 1 on Moodle (~10 mins)

# System Design

How to think about creating efficient software solutions

# System Design

## What is it?

- Process of designing the *architecture*, *components*, *data*, and *interfaces* for a system
  - Done to meet customer needs
- Customer requirements -> Physical system
  - Required careful planning and consideration
  - Need to have descriptions of what customers can do
  - How their data is handled
  - How the system is architected to meet needs
- It’s a very systematic approach that requires clear communication

# System Design

## Functional requirements

- What users can do (represented through use cases)
- What data needs to be stored to meet business needs
- Any business rules that need to be applied
- How users will authenticate themselves, and any authorization that is needed
- How the system will be administrated

# System Design

## Applied to our checklist app

- Users can sign in with most social apps
  - Google, Facebook, LinkedIn, etc.
- Users can browse and search for checklists
- Users can add checklists to their favourites
  - They can also leave comments and reviews (or report lists)
- Users can create their own checklists
  - Add categories and tags for searching
- Moderators can edit all checklists to remove inappropriate content
  - They require a moderator access level and have their own dashboard

# System Design

## Non-functional requirements

- Not-so-direct requirements about the systems performance as a whole
- Referred to as the –ilities
  - security, reliability, scalability, maintainability, availability, etc
- Each of them are quite deep in their own right
  - We just looked at scalability previously
- They exist to serve as various constraints to the system
- Link to a quick summary of them

# System Design

## i Applied to our system

- Can be a very broad range of aspects, but will list some
- Security – we allow external providers to authenticate and control user data
- Our system should be highly available, with eventual consistency
  - It's not urgent to have the most up to date data every request, a few seconds or minute delay is fine (we don’t want errors)
- Our discover page shouldn’t take more than 1 second to load

# Software Architecture

Structuring our software

# Software Architecture

## What is it?

- High level structure of the actual software
- Creates the solutions to the business needs
  - The services needed to meet requirements
- Defines the various modules and components needed
  - This helps ensure maintainability and scalability
- Has various patterns and principles which guide the design
- Focusses on the externally visible parts of the system and their interaction

# Software Architecture

## i vs System Architecture

- It is a bigger picture, that focusses on more than just software
  - Hardware, networking, and so on
  - It can also define multiple different subsystems of software that interact
- All the diagrams seen so far have been of system architecture
- Software and system architecture are often mixed – to provide different "zoom" levels

# Software Architecture

## What about Software Design?

- This is the granular look at the individual modules and components
- The actual code inside the applications
  - How the various components interact
- It is the implementation of the architecture
- SOLID principles and design patterns help guide it

# Software Architecture

**System Design**

- High level planning of overall system to meet customer needs. Functional requirements, Non-Functional requirements, High level Architecture, Data considerations, etc.

**System Architecture**

- Planning of how infrastructure looks. Includes aspects such as networking, applications, storage, etc.

**Software Architecture**

- How the applications themselves are structured. Layered vs Microservice vs Event-Driven (Architectural Patterns).

**Software Design**

- How the various modules of the application interact. A "how" of the architecture. SOLID principles and Software Design Patterns.

`Similar terms, different levels`

# Software Architecture

## Architectural patterns (styles)

- There are many different design patterns which help developers
- We will focus on contrasting just two
  - Layered (N-tier)
  - Microservice
- Reason: Modern development is mostly focussed on shifting a monolith into a microservice
  - Kubernetes main role is with microservices
- Article from Red Hat looking at 14 common patterns
  - Another from Red Hat looking at a few more closely

# Software Architecture

## N-tier

- A very common way of architecting software is with the layered approach
  - These layers are called tiers
- How most monoliths are defined, what you already know
- Presentation -> Business -> Persistence -> Database
  - Implemented in many ways (depending on stack)
- For Java Spring Data + Web:
  - Controller -> Service -> Repository -> Database

# Software Architecture

## Microservices

- Architectural approach to building systems
- System is composed of small, loosely coupled, distributed services
  - Changes won't break the entire app
- Increases speed of delivery of new features
  - Small applications are easier to create and deploy
  - Very well aligned with CI/CD
- The biggest factor – scales incredibly well

# Software Architecture

## i The problem with Microservices

- Red Hat have a good article, where they outline some challenges:
  - **Building** is challenging as it required a lot of planning and consideration for how service interacts and how data is handled
  - **Testing** becomes more complex as it may take a few different failures to know the source
  - **Deployment** and **versioning** becomes complex due to management
  - **Logging** and **monitoring** is mandatory and often required separate applications to achieve
  - **Debugging** is not trivial as you can't see all the services in one IDE
  - **Connectivity** challenges arise when you must consider visibility

# Software Architecture

## When to move to micro

- When you cannot scale feasibly with hardware and replications (previous methods we discussed)
  - I cannot stress enough *don't start with microservices*
- Also, if certain parts of your application are being used for most of the traffic
  - For example – the *discover* page is using 80% of the traffic. It's a waste to scale the entire app up, why not split it.
  - There are also non-microservice solutions (CQRS)
- When we move away from monoliths, everything gets more complex

# Software Architecture

## How to start shifting?

- Simply put – identify and break things up
  - Need to do an analysis, often with Domain-driven design
- Article from Microsoft outlines this nicely
  - Similar article from Google Cloud
- So, our goal is to see what services our monolith contains and break those out
- *Looking at our use case (the checklist app), what services do you see?*

# Software Architecture

| | | | |
|---|---|---|---|
| Auth Service | Discovery Service | Checklist Service | Comment Service |
| Rating Service | Review/Moderate Service | Search Service | User Service |
| Notification Service | Monitoring Service | Logging Service | Messaging Service (for monitoring) |

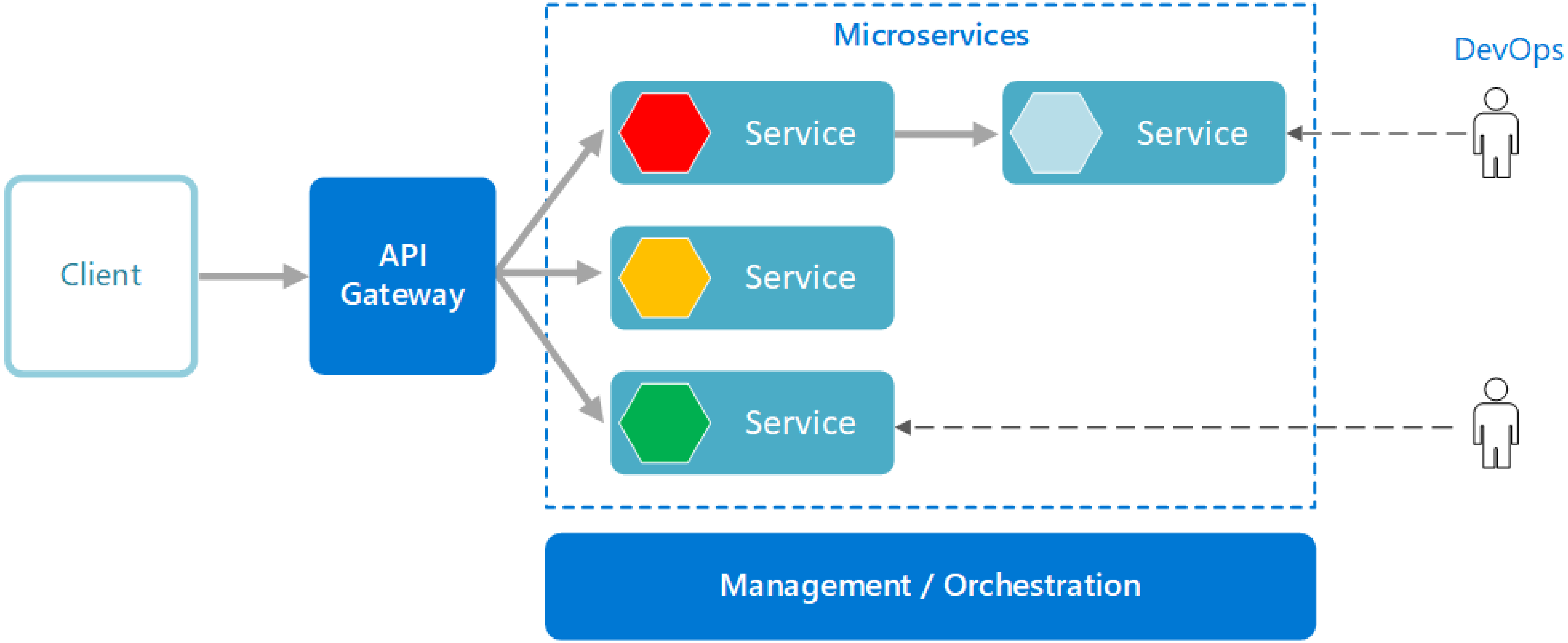
`Our services (Azure Article)`

# Software Architecture

## How do we manage this?

- 12+ services, multiple databases, networking, scaling?
  - Sounds like an actual nightmare, because it is
- For example, we want our *Discovery* service to have more resources than our *Moderation* service
- What happens when one of the services go down?
- Or we need to update a service, we can't just *turn off* notifications or comments while it updates
- It sounds like we need something to orchestrate this all
  - *What do you think we should do?*

# Software Architecture




[Microservices architecture design]

# Any questions?

Check up before we go into Kubernetes

# Software Architecture

## Refocussing to Kubernetes

- Orchestration is one job of a microservice-oriented solution
  - A job Kubernetes is made for, so we should use it here
- There are many options for how to use Kubernetes
  - Each with their own levels of interaction
  - We will use it primarily as-is (raw) and with Docker Desktop
- Now that we have established some context, we can start looking at Kubernetes with an understanding of its place

Noroff
School of technology and digital media

# Quiz 2: Design and Architecture

Please complete Quiz 2 on Moodle (~10 mins)

# Kubernetes Theory

Understanding the orchestra

# Kubernetes Theory

## What is orchestration?

- Organising individual containers to appear as a cohesive *whole*
    - To meet users' needs
- In more technical terms:
    - It automates the deployment, management, scaling, and networking of containers.
- Kubernetes isn't the only orchestration tool, it's just the most popular one
    - Mainly due to it being vendor-agnostic

# Kubernetes Theory

## i What can Kubernetes do?

- Handles networking (internal and externally exposed)
- Load balances
- Orchestrates storage as well as services
- Automatic rollout and rollback – you can change configuration and it will adapt
- Bin packing – allocate resources and Kubernetes maximises for that
- Configuration and secret management

# Kubernetes Theory

## What Kubernetes is not

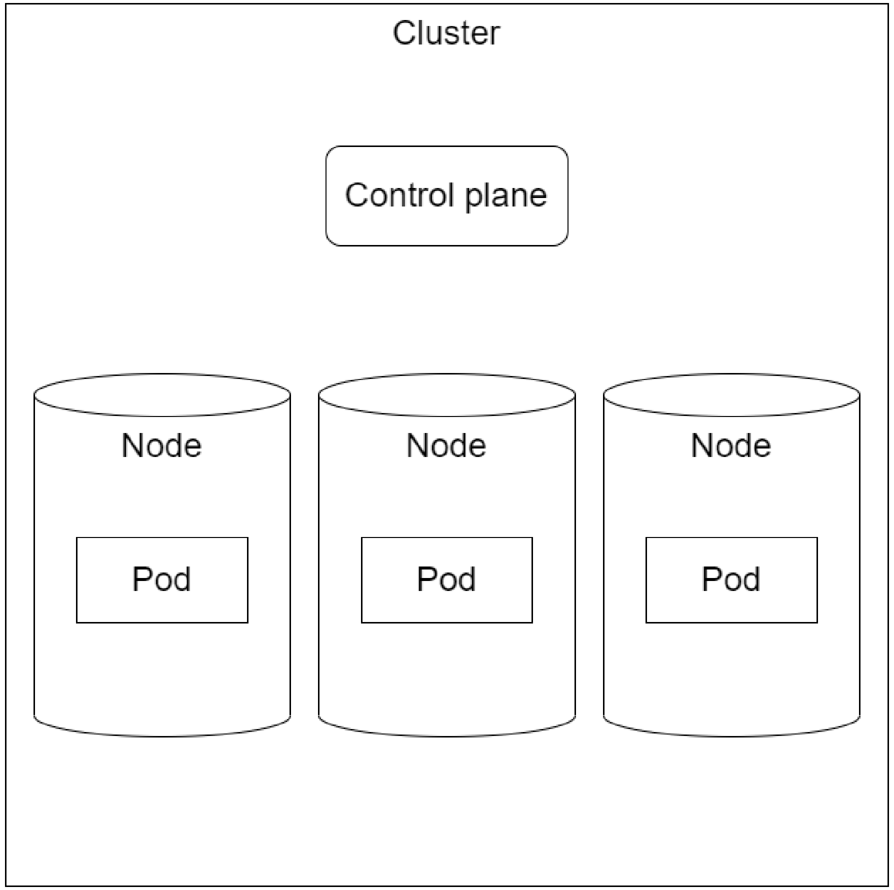
- Doesn’t build and deploy source code – it simply helps run it
- Does not provide application-level services (middleware, data-processing, databases, caches) built-in
  - It can run these, but it needs to be configured by you
- Does not dictate logging, monitoring, or alerting solutions.
  - Once again, it can run them, but it's up to you

# Kubernetes Theory

## i Architecture: Cluster, nodes, and pods

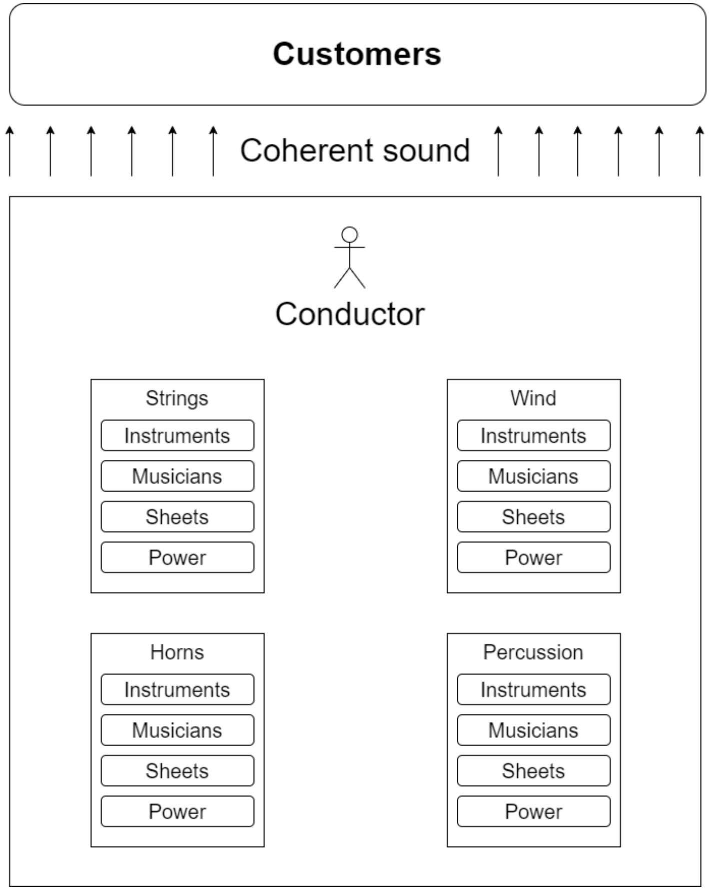
- When you run/deploy Kubernetes you get a *cluster*
- The cluster has *nodes* which do work (run containerized applications)
  - Every cluster has at least one
  - Nodes are either physical or virtual machines
- Nodes host *pods* which are a set of running containers
- Kubernetes controls this with a *control plane* that acts as a master node
  - It manages the worker nodes and pods
  - Pods can be spread over many nodes

# Kubernetes Theory




Architecture (simplified)

# Kubernetes Theory




Orchestra analogy

# Kubernetes Theory

## i A closer look at the "conductor"

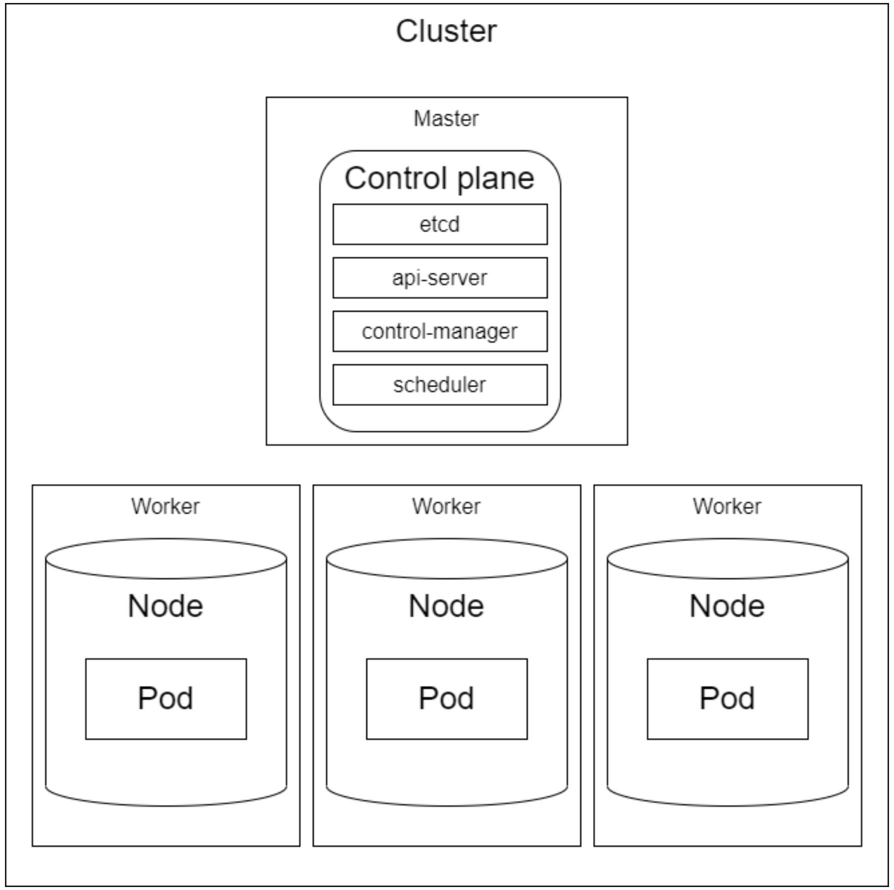
- The **control plane** makes global decisions about the cluster and lives in the master node
- It tells which worker nodes to run which containers
- It also controls the networking between them all
- It also has an API exposed for management by a dev
- *Drives actual state toward desired state*
- It also helps with scheduling (assigning new pods to nodes)

# Kubernetes Theory

## Control plane components summary

- **ETCD** – how cluster data is persisted (key-value store)
- **Scheduler** – watches for new pods with no node and tries to assign nodes to run them within the defined constraints.
- **Controller manager** – watches the current state and makes changes towards desired state.
  - It has different controllers for different jobs (node, replica, jobs, namespace, endpoints)
- **API server** - uses REST operations to manage the cluster
  - Also exposes itself to be externally managed

# Kubernetes Theory



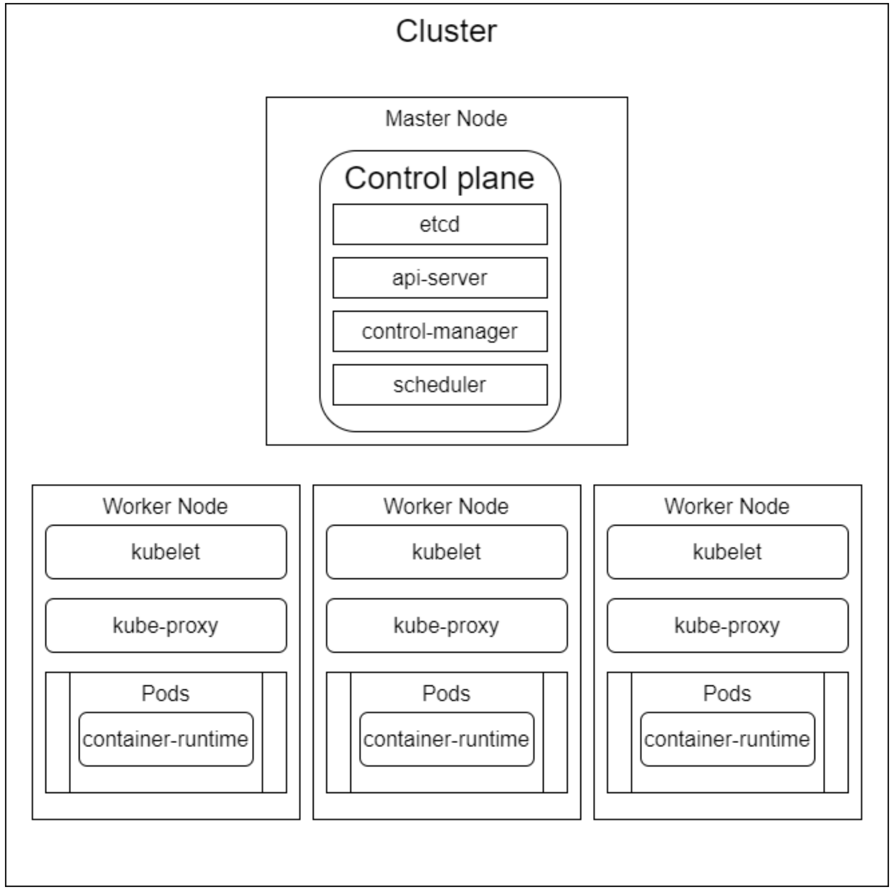

`Including control plane components`

# Kubernetes Theory

## Node components

- Node components run on every node
  - Maintaining running pods
  - Providing the Kubernetes runtime environment
- A **kubelet** runs on each node in the cluster
  - Makes sure that containers are running in a Pod.
- A **kube-proxy** maintains network rules on nodes
  - Implements part of the service concept (expose apps running on pods as services)
  - Allows network communication from inside or outside your cluster (helps with traffic forwarding)
- The **container runtime** is the software that is responsible for running containers (e.g., Docker Engine)
- A **DaemonSet** ensures that a specified collection of pods runs on the specified nodes
  - It makes sure one pod exists per node.

# Kubernetes Theory




Including Node components

# Kubernetes Theory

## Pods

- Smallest deployable units of computing that you can create and manage in Kubernetes.
- A group of one or more containers, with shared storage and network resources, and a specification for how to run the containers.
- You can have a pod be for a single container, or multiple tightly coupled containers.
  - They are placed on the same server and share dependencies and resources
  - Think of it like docker-compose
- Pods are not managed by you, but by the Kubernetes Scheduler and controllers

# Kubernetes Theory

## Pod management

- When a pod is created, it needs to be assigned a node by the scheduler to run.
- Kubernetes wont manage the containers, it manages the pods via controllers
- Pods run single instances of provided applications
  - If you need multiple, you need to create replica pods
  - Its simply an included flag or line in configuration
  - The controller will manage the replicas

# Kubernetes Theory

## Pods and controllers

- Pods are created by workload resources called controllers
  - Manage rollout, replication, and health of pods in the cluster
- If a node in the cluster fails, a controller detects that the pods on that node are unresponsive
  - Creates replacement pod(s) on other nodes
- The three most common types of controllers are:
  - **Jobs** for batch-type jobs that are ephemeral, and will run a task to completion
  - **Deployments** for applications that are stateless and persistent, such as web servers (HTTP servers)
  - **StatefulSets** for applications that are both stateful and persistent such as databases

# Kubernetes Theory

## Pod communication

- When pods are created they are assigned IP addresses
  - This changes each time they are created, so its pointless hardcoding them
- With multiple containers in a pod, localhost will work
  - Multi-container pods are not common and seen as advanced, so this is not really applicable to us
- You can expose a pod externally as a Service
- Kubernetes handles inter cluster IP assignment, so we don’t have to worry about creating links or mapping to host ports

# Any questions?

Now that the information dump is over, phew

# Kubernetes Theory

## Extensions and addons

- Kubernetes has some base functionality, but a lot is left out
  - This is on purpose to not be vendor-locked
- Addons are not mandatory, but greatly help the developer experience
- Some common addons relate to:
  - Web UI, DNS services, monitoring, logging, health checks (chaos monkey), and testing
  - These addons are added to the cluster as containers to provide their functionality

# Quiz 3: Kubernetes Architecture

Please complete Quiz 3 on Moodle (~10 mins)

# In Conclusion

## Key takeaways

- We looked at how to improve scalability of an application without using Kubernetes.
- When normal scaling doesn’t help, its time to reorganize our software architecture to migrate to a microservice architecture.
- This is a complicated process, but Kubernetes helps with management and deployment
- Kubernetes has a complicated architecture based on a master-slave pattern. This is implemented by nodes which act as workers.
- Each node contains one or more pods of running containers.
- Kubernetes manages this all through the control plane and tries to match the desired state provided by configuration.

# In Conclusion

## Looking Ahead

- Now we will practically use Kubernetes to configure and run our own microservice.
- We will not deal with state (databases) as that mean we need to engineer solutions to CAP – we don’t have time for that.
- We will start with Kubernetes locally and then move to Azure Kubernetes Service in the cloud and secure it.

# Additional Resources

## Hungry for more?

- The next few slides provide links to useful resources we also use from time to time:
  - Reference guides
    - A must have for any developer
  - Video demonstrations
    - Quality videos explaining selected topics or demonstrating skills
  - Articles, sample code and hands-on tutorials
    - Interesting articles
    - Additional code examples
    - Step-by-step guides for extra practice

# Additional Resources

💡 Articles, Sample Code & Tutorials

- System design cheatsheet
- Article - The complete guide to System Design
- Azure Architecture Center
- Books:
  - Clean Architecture – Robert Martin (I have this one)
  - Designing Data Intensive Applications
  - Systems Analysis and Design
- Microsoft Learn module on scale up and out

# Additional Resources

## Articles, Sample Code & Tutorials

- Microservices:
  - Articles from major cloud providers on "what is" - Azure, AWS, Google, and IBM
  - Book on monolith to microservice
  - microservices.io
  - Martin Fowler on microservices

# Additional Resources

💡 Articles, Sample Code & Tutorials

- YouTube channels and talks
  - Scott Hanselman channel
  - Mark Richards channel
  - Talk on how to think like an architect
  - Netflix talk on how they used microservices
- 12 factor app

# Additional Resources

## Articles, Sample Code & Tutorials

- Kubernetes:
  - Kubernetes docs
  - Kubernetes glossary (Red hat)
  - Kubernetes cheat sheet
  - Microsoft Learn Introduction to Kubernetes