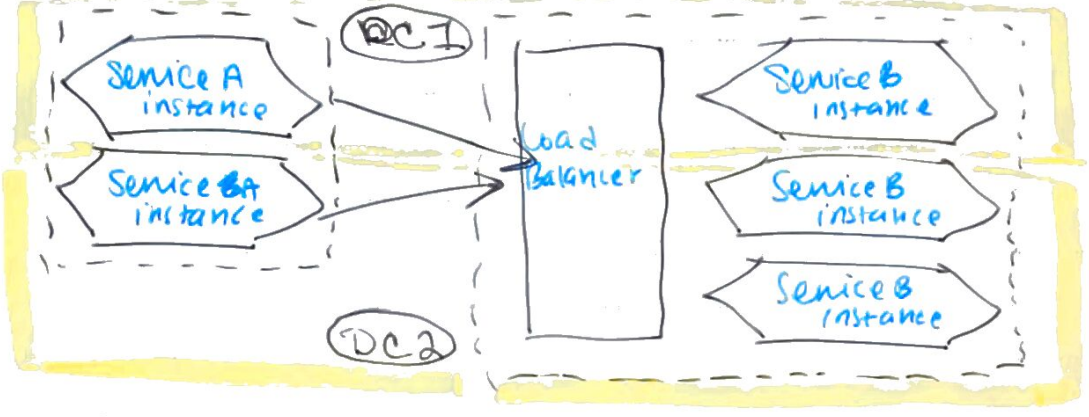
Building Microservices Ch8

Deployment

- From Logical to Physical

"A logical view of an architecture typically abstracts away underlying physical deployment concerns — that notion needs to change for the scope of this chapter."

∞ instances : ↑ load, ↑ robustness, ↑ failure tolerance of single instance



- the # of instances depends on: required redundancy, expected load.
  + where will the instances run? ex. different hardware, data centers

Database "don't share databases"

* any db used by a service to manage its state is hidden inside the service
  + instances of the same service should be able to access the same db

"The logic for accessing & manipulating state is still held within a single logical service"

→ can have read replicas & primary DB node since w/ relational dbs you can't typically scale writes w/ more db nodes so free up space w/ read replicas

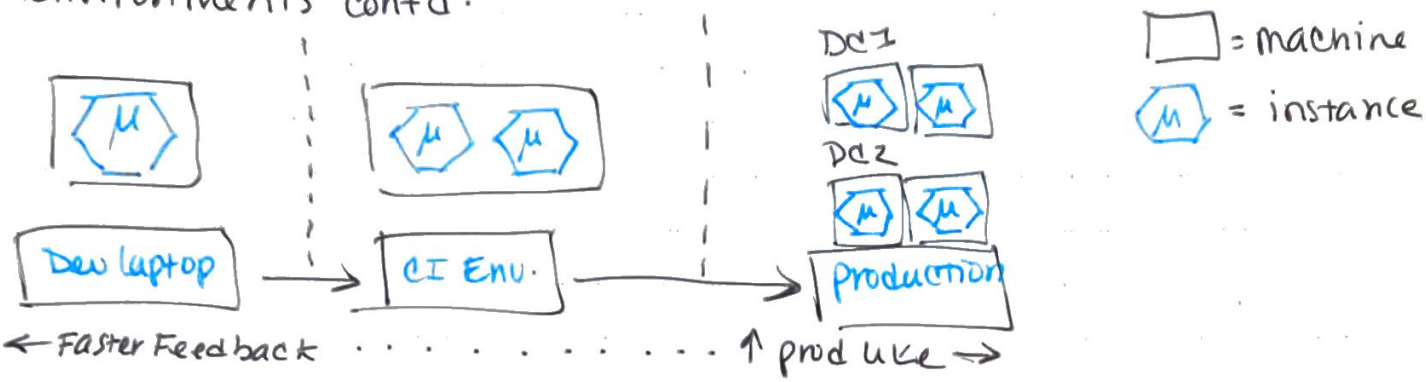
* the same DB hardware can support multiple logically isolated databases. BEWARE if shared hardware fails
* cost is ↓ for hardware w/ cloud providers like AWS

Environments

"...environments closer to the developer will be tuned to provide fast feedback ... as envs get closer to prod ... more & more like the end prod env. to ensure that we catch problems."

Infrastructure As Code
2nd edition
Kief Morris

Environments cont'd.



"A μservice can vary in how it is deployed from one environment to the next

+ need to have proper config / environment

* Principles of μservice Deployment

① Isolated Execution : each μservice has its own computing resources + don't impact others

② Focus on Automation

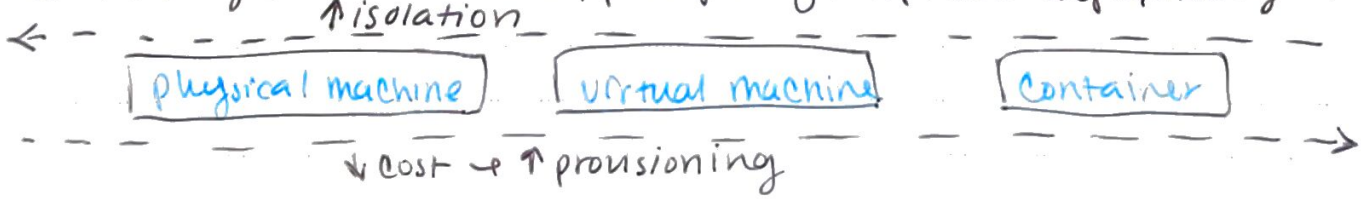
③ Infrastructure as code

④ Zero-downtime deployment

⑤ Desired State management : use a platform that maintains μservice state, launching new instances if required

① isolated execution

- running ∞ μservices on one host can make monitoring more difficult, load management, dependency manageme for ∞ μservices ↑difficult + ↓autonomy of teams
- drastically undermines the key principle of independent deployability



* container isolation is now good enough that it's the default isolation solution

② Focus on automation

→ good place to start is how you manage hosts, is it easy to build + tear down?

③ Infrastructure as Code (IAC) Puppet, Chef, Ansible, Bash, Terraform, Pulumi, AWS CloudFormation + Cloud Dev Kit (CDK)

↑ transparency, ↑ reproduceability

④ Zero downtime deployment
- more likely to be able to release during a work day
- or do rolling deployment where old versions are ramped down while new versions are ramped up
- easier to implement right away vs. having an existing repo retrofitted for 0 downtime deploys.

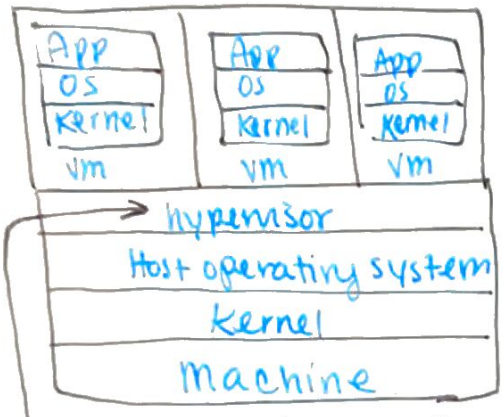
⑤ Desired state management
Is "the ability to specify the infrastructure requirements you have for your application, and for those requirements to be maintained w/o manual intervention.
- ex # μservice instances, how much memory + CPU

Kubernetes, autoscaling groups on Azure/AWS, Nomad
* need to remember you have this management in place
* Fully automated deployments are required
  + taking into consideration how long it takes for an instance to be launched
→ can wait to implement these tools if you don't have many processes at first that need them

Git Ops
↳ where desired codestate is defined in code + stored in source control + make use of tooling inside of Kubernetes (for example) vs just infrastructure
Flux

Deployment Options
(↓) Physical machine, no virtualization, direct on machine
Virtual machine, VMware, AWS EC2, vSphere, Xen, KVM has overhead
Container, Docker, Solaris zones, OpenVZ, LXC ↓startup time, ↑control, ↑containers on a host
(↓) Application Container → ↑cost, tech lockin, ↓isolation, ↓option, ↑startup time, ↑complex, ↓monitoring
Platform as a Service (PaaS) ex. Heroku, * Google App Engine, Netlify, AWS Beanstalk
Function as a Service (FaaS) ex. AWS Lambda, Azure Functions → durable functions
↳ core component of "serverless", ↓control + possible limits on call time / exec. time → stateless

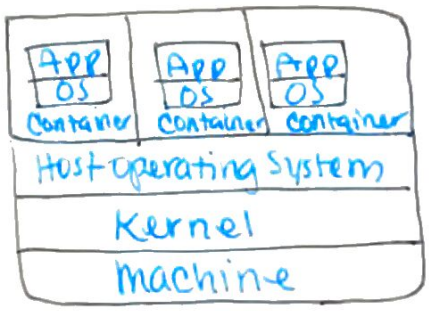
* Serverless: the developer doesn't have to think about the server, it's still there, just managed by someone else.



Type 2 virtualization

① map resources (CPU/memory) from ~~###~~ virtual host to physical host.
② control layer to manipulate the VMs themselves

"If you need the stricter isolation levels that they can bring, or you don't have the ability to containerize your application, VMs can be a great choice."



Container-based virtualization

"think of a container as an abstraction over a subtree of the overall system process tree, with the kernel doing all the hard work."
* need to get routing from outside world to container
* it is possible for a container to "bust out" + interact w/ other containers or the underlying host
"View containers as a great way of isolating execution of trusted software."
* Microsoft's Hyper-V containers
* Firecracker

FaaS Challenges
- startup time can vary based on language
- possible max # of concurrent invocations
- make sure all aspects w/ scale w/ the functions

"Really, though, when it comes to managing ∞ containers on ∞ machines, K8s (Kubernetes) is king here, even if you might use the Docker toolchain for building + managing individual containers"

→ Webassembly (WASM): standard defined to give devs a way of running sandboxed programs on browsers "the goal... is to allow arbitrary code to run in a safe + efficient manner on client devices."
→ Webassembly System Interface (WASI): a way to let WASM move from browser to anywhere that has implemented the interface

* WASM has the potential to challenge the use of containers as the go-to deploy-format for server-side applications. *

Faas, mapping to μservices

① Function per μservice, sensible place to start
- invoked → trigger single entry point in function
- → some way to dispatch to different pieces of functionality in μservice

② Function per aggregate, based on DDD
- May have already explicitly modeled aggregates (typically refer to real-world concepts).
- ensures all logic for an aggregate is self-contained in the function
- μservice is now a logical concept w/ diff. ∞ functions that theoretically can be deployed independently
- * (++) coarse grained external interface so you can restructure w/o impacting upstream consumers
- * shared DB if same team manages all services
  - ↳ if needs diverge look into separating out data usage

"now one μservice is made up of multiple different indep. deployable units." "the μservice moves towards being more of a logical than a physical concept."

* Careful not to violate the core principles of an aggregate, should be treated as a single unit to ensure we can better manage the integrity of the aggregate itself.

* Sam's Really Basic Rules for Where To Deploy Stuff *

① if it ain't broke, don't fix it

② give up as much control as you feel happy with, and then give away just a little bit more

③ Containerizing μservices is not pain-free but is a good compromise for cost of isolation + local development w/ ↑ control over what happens
- * Expect K8s

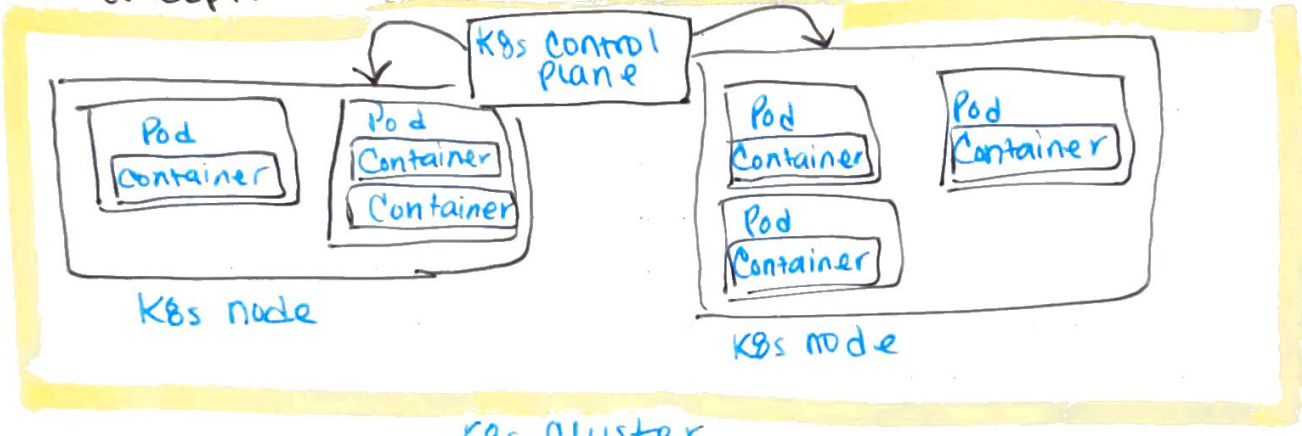
→ Unless your needs can be met w/ something simpler like PaaS or Faas

* Puppet, Chef, Ansible + Salt have a ↓ role now, but can help w/ legacy apps/infra. + for building the clusters that container workloads run on

* infra as code is still important, the tools have just changed
  Terraform (cloud) Pulumi (lang. flexibility)

K8s → Container Orchestration
↳ handle how & where container workloads are run
↳ " desired state management & workload distribution
- Mesos, Nomad, AWS ECS, Docker Swarm Mode, etc *but k8s won*

K8s Concepts



K8s Control Plane: controlling software that manages nodes
K8s Node: set of machines that the workloads will run on
K8s Pod: ≥ 1 containers deployed together, ephemeral
K8s Service: stable routing endpoint, a way to map from the pods running to a stable network interface w/in the cluster. Long-running.

"in Kubernetes you don't deploy a service — you deploy pods that map to a service"

K8s replica set: define desired state of a set of pods, you don't work w/ replica set directly, they're handled for you via deployment
K8s deployment: how you apply changes to your pods & replica sets
↳ issue rolling upgrades, rollbacks, scaling ↑ nodes, etc.

"So, to deploy your μservice, you define a ~~set~~ pod, which will contain your μservice instance inside it; you define a service, which will let k8s know how your μservice will be accessed; and you apply changes to the running pods using a deployment."

Borg, Omega, and Kubernetes
Brendan Burns et al.

Multitenancy & Federation
- different departments may want different degrees of control over various resources.
- These controls weren't built into K8s, so here are some workarounds
  ① adopt a platform on top of K8s to handle this, OpenShift, ↑cost & ↑to learn
  ② Federated model of ∞ separate clusters of software on top to make changes to all clusters if needed. Resource pooling is harder, can help w/ cluster upgrades

*these are problems of scale

- Cloud Native Computing Foundation (CNCF), curates ecosystem of projects to help ↑ cloud native development, in practice this means supporting K8s & projects that work w/ or build on K8s

"Most folks using K8s end up assembling their own platform by installing supporting software such as service meshes, message brokers, log aggregation tools, and more."

* applications built on K8s are portable across K8s clusters in theory, but not always in practice *

- How to manage deployment & life cycle of 3rd party apps & subsystems is a problem. Helm is "missing package manager" for K8s Operator is focused on the ongoing management of the application, you could use both together.
  Custom Resource ~~Definition~~ Definitions (CRDs), you can plug in new behavior into your cluster & basically allow you to implement your own K8s abstractions. Can use these for anything & there's no best practice consensus yet
  Knative: open source & aim to provide FaaS-style workflows to developers using K8s under the hood. A service mesh (specifically Istio) is required. May undergo big changes
  OpenFaaS is an alternative

* Author expects K8s will stick around & in the future will be abstracted so you'll be using K8s w/o knowing it.

Accelerate
Forsgren, Humble, Kim

Should You use K8s
- implementing & managing your own K8s cluster is not easy
  → K8s specific info is required right now, so some people use other companies or use other FaaS / PaaS solutions
- Micro K8s to test out & Katacoda for online tutorials
- "if you've got a handful of developers and only a few services, K8s is likely to be huge overkill, even if using a fully managed platform"

Progressive Delivery
- *shipping frequently & having ↓ failure rate goes hand in hand
  ↳ can use feature toggles, canary releases, parallel runs, etc. all falls under the banner of Progressive Delivery
- "we can separate the concept of deployment from that of release"
- Deployment: when you install a version of your software into an environment
- Release: when you make a system or some part of it available to users
- "progressive delivery as "continuous delivery w/ fine grained control over the blast radius""
  - Blue Green Deploy: one version (blue) live & another version (green) live make sure new version works then redirect customers.
  - Feature Toggles: hide functionality behind a toggle (Launch Darkly, Split or config file
  - Canary Release: limited subset of customers see new functionality if there's a problem, rollback, if not then rollout Spinnaker
  - Parallel Run: execute both versions, use result from your designated implementation. Scientist
- *these tools can be used in coordination

Summary, principles of deployment:
① Isolated Execution ② Focus on Automation ③ Infrastructure as Code
④ Aim for zero-downtime deploy
Guidelines: ⓐ ain't broke, don't fix ⓑ give up control ⓒ Containerize

* this space is going through a lot of churn