# SE 577
# Software Architecture

## Cloud Native Architectures

# What are cloud native architectures?

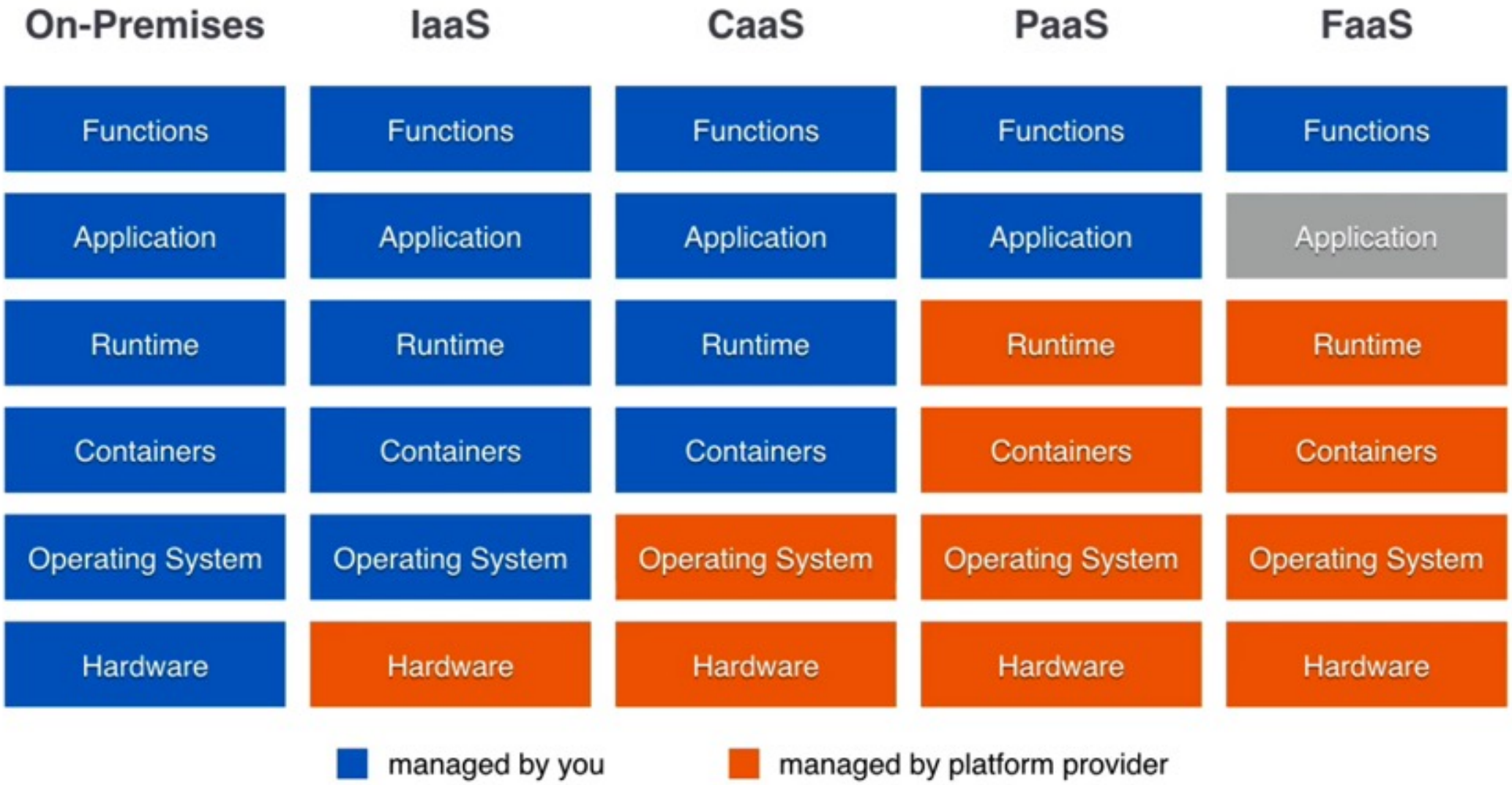
- Cloud native architectures interestingly have nothing directly to do with the cloud, although to realize its goals, they tend to utilize cloud-based platforms
- Attributes of cloud native architectures
  - Support rapid and continuous deployment
  - Elastic Scale (both up and down)
  - Embrace automation for everything – think "*x as code*" – infrastructure provisioning, build processes, deployment
  - Are "supervised" to promote fault tolerance and self-healing
  - Pay for what you use vs pay for software licenses
  - Shift engineering attention towards building solutions vs managing infrastructure, security patching, updating/upgrading and configuration
  - Are "antifragile" – as we increase stress on the system via speed and scale the system improves its ability to respond
  - Are API first, every aspect of the system can be managed via APIs

# What are cloud native architectures?

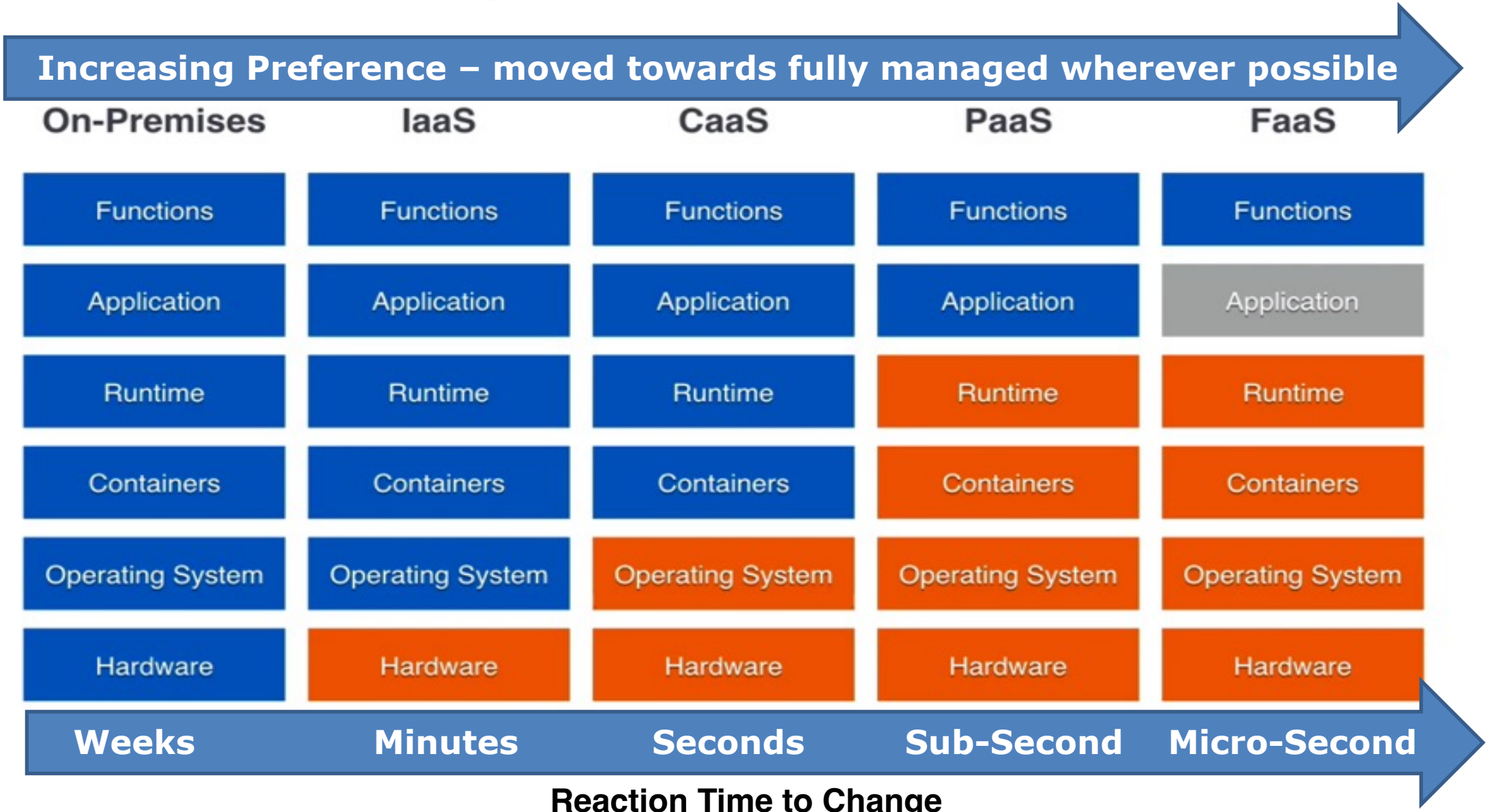
- What is needed for a cloud native architecture given the fact that they don't actually have to run in the cloud?
- Regardless of on prem, or public cloud, all cloud native architectures require
  - Software defined control of technical resources that support, compute, network, security policy and storage
  - APIs that allow engineering to control every aspect of infrastructure configuration and deployment
  - An underlying platform that provides isolation and redundancy of workloads
  - An underlying platform that is hardened against any single point of failure
  - An underlying platform that can apply course- and fine-grained security everywhere
  - An underlying platform that is polyglot – allows workloads to be developed in any language on any platform
  - An underlying platform that supports granular collection of data to support billing (direct, internal chargeback) based on resources consumed

**The above list is basically best served by a mainstream cloud provider – AWS, Azure, GCP, etc; however you can also do these things to a large degree on premise – example, VMWare has a number of these offerings, and many cloud providers provide virtual machines that you can run on premise to take the end of the cloud on prem.**

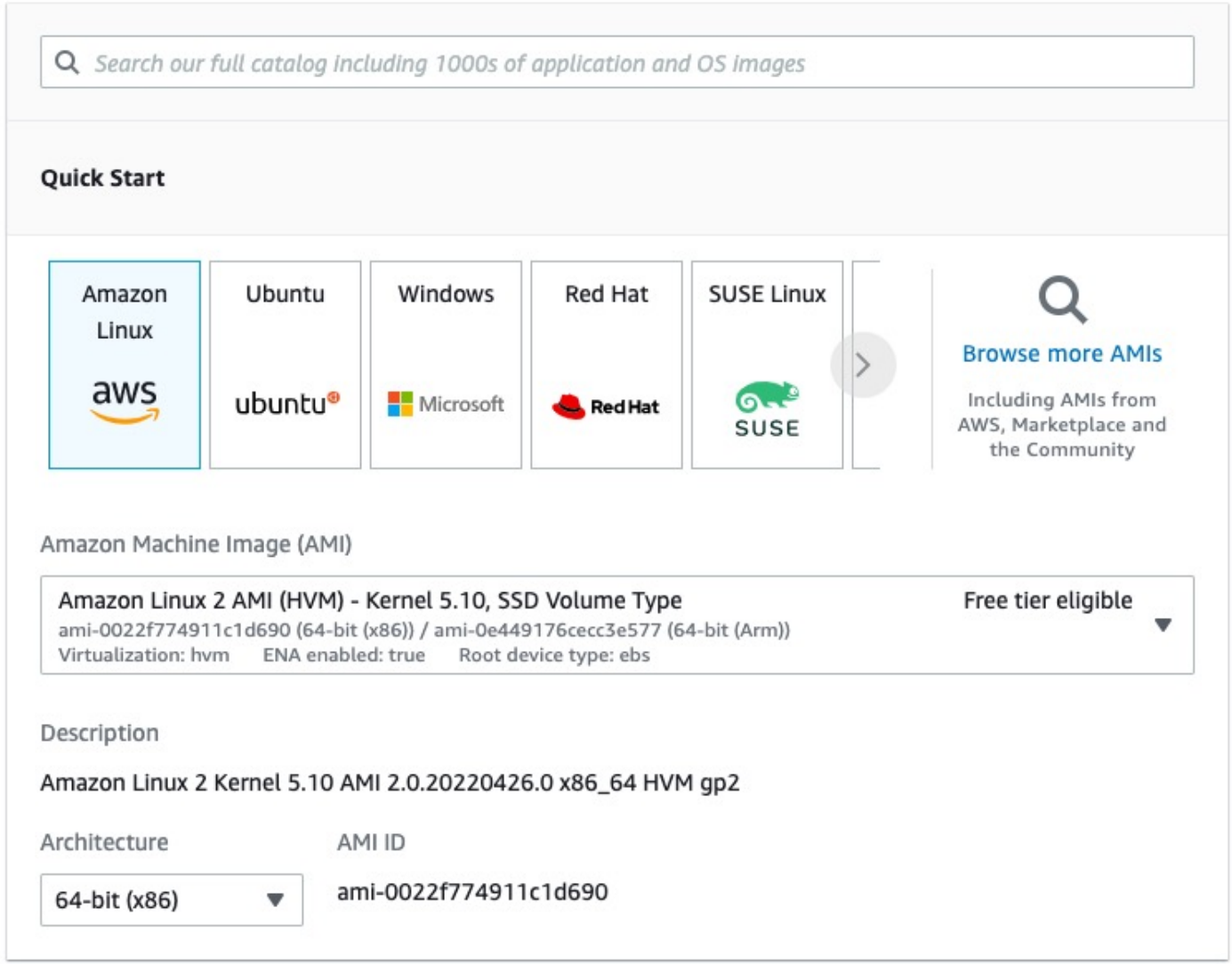
# Cloud Computing Models



# Cloud Computing Models



# IaaS Example – Aws EC2



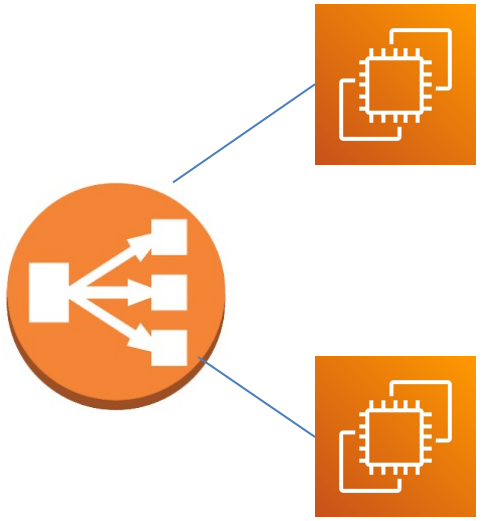

AWS EC2 allows you to launch a virtual machine inside of a “virtual data center” – VPC

Startup time = a few minutes

# IaaS Example – Aws EC2


When you provision a virtual machine, you can specify a “startup script” to install and configure your instance

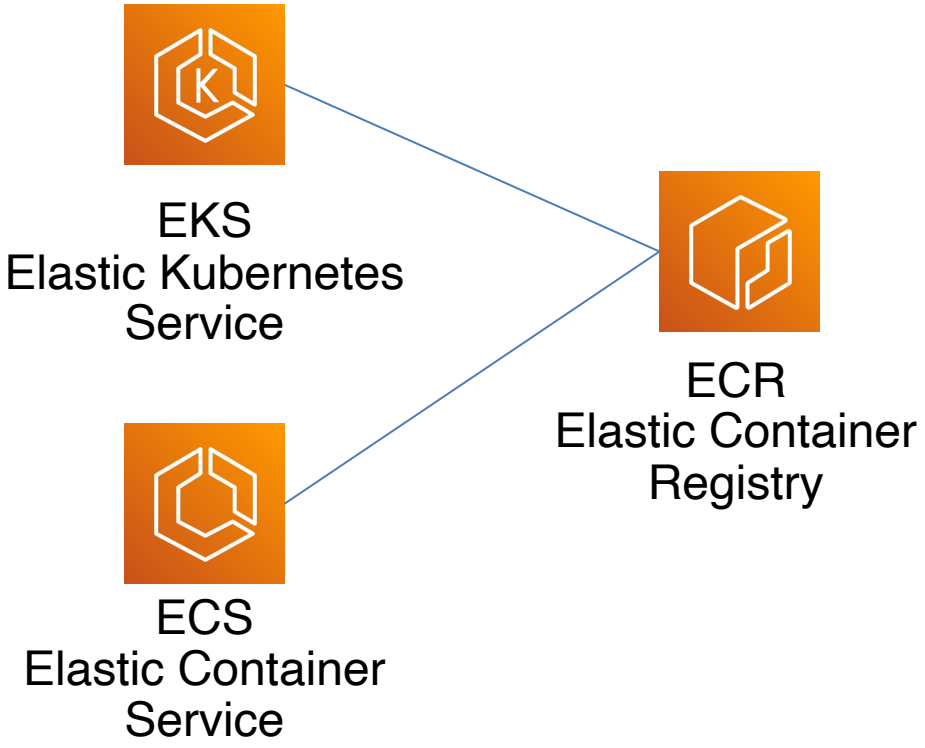
You can also take a bare instance, ssh into it, configure it, and then create your own custom AMI (Amazon Machine Image) – like a docker image but for a full virtual machine



You can setup an elastic load balancer (a fully managed service) to send traffic to EC2 instances, they can also run health checks to scale them

**The biggest drawback with IaaS based solutions is that you are responsible for everything including patching the OS, software installation and configuration, and defining how the virtual machines will be managed, scaled, etc. Also, scaling EC2 is slow, takes several minutes since its an entire VM reboot**

# CaaS Example – Aws ECS



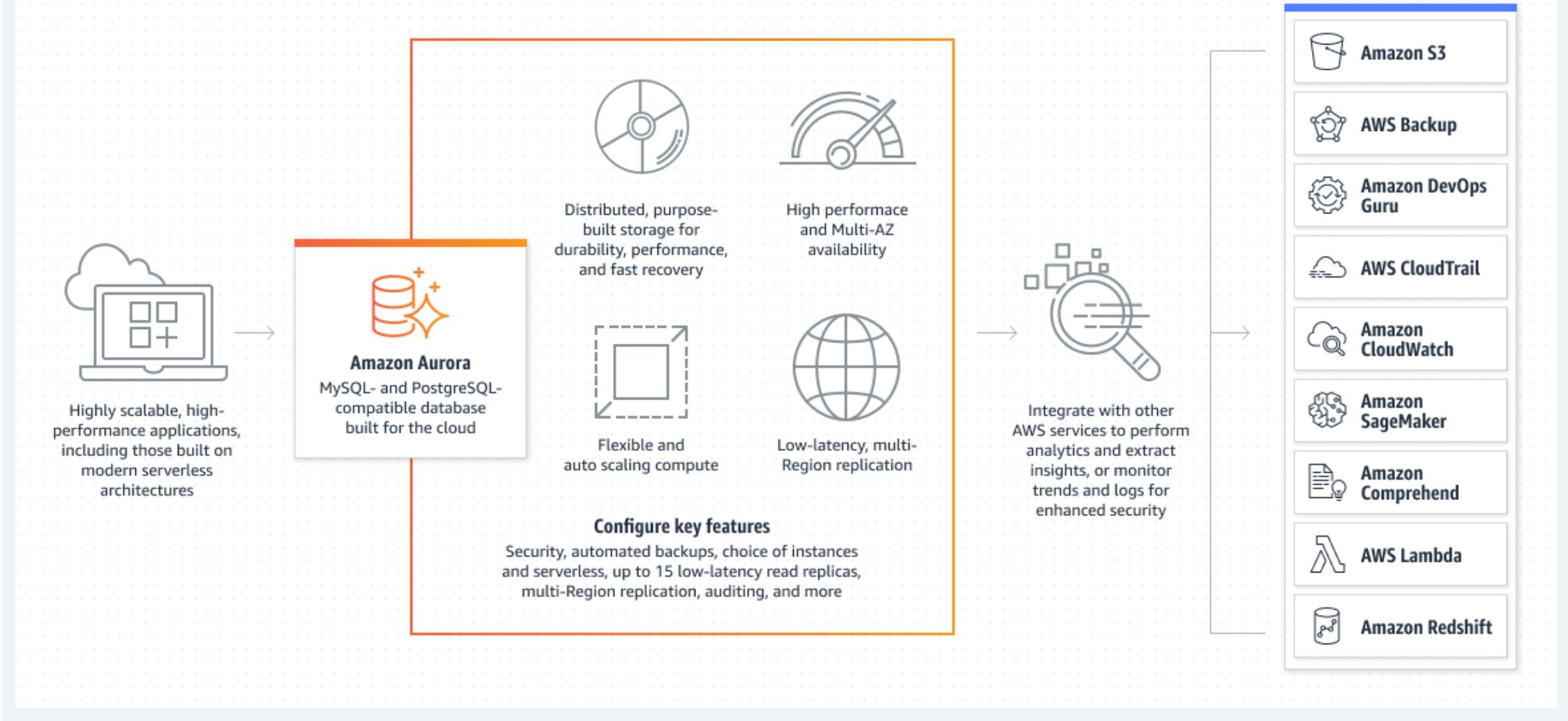

CaaS solutions require you to manage fewer things, you just manage your container images, the cloud provider manages the runtime (data plane) that executes the containers

You are still responsible for things like patching and upgrading containers (base images, dependencies)

In this example EKS and ECS are both fully managed services, the architecture tradeoff is a standards based solution EKS, which is more difficult to manage vs ECS which is cloud specific to AWS

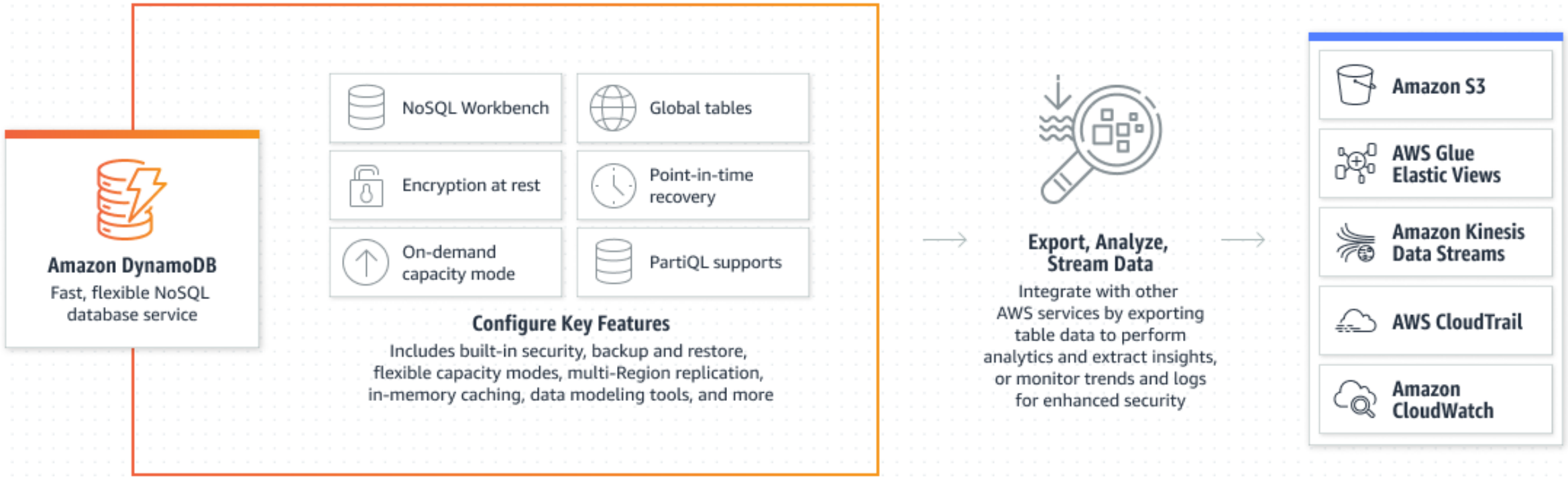
# Fully Managed Service Example

Amazon Aurora provides built-in security, continuous backups, serverless compute, up to 15 read replicas, automated multi-Region replication, and integrations with other AWS services.



# Fully Managed Service Example II

Amazon DynamoDB is a fully managed, serverless, key-value NoSQL database designed to run high-performance applications at any scale. DynamoDB offers built-in security, continuous backups, automated multi-Region replication, in-memory caching, and data export tools.

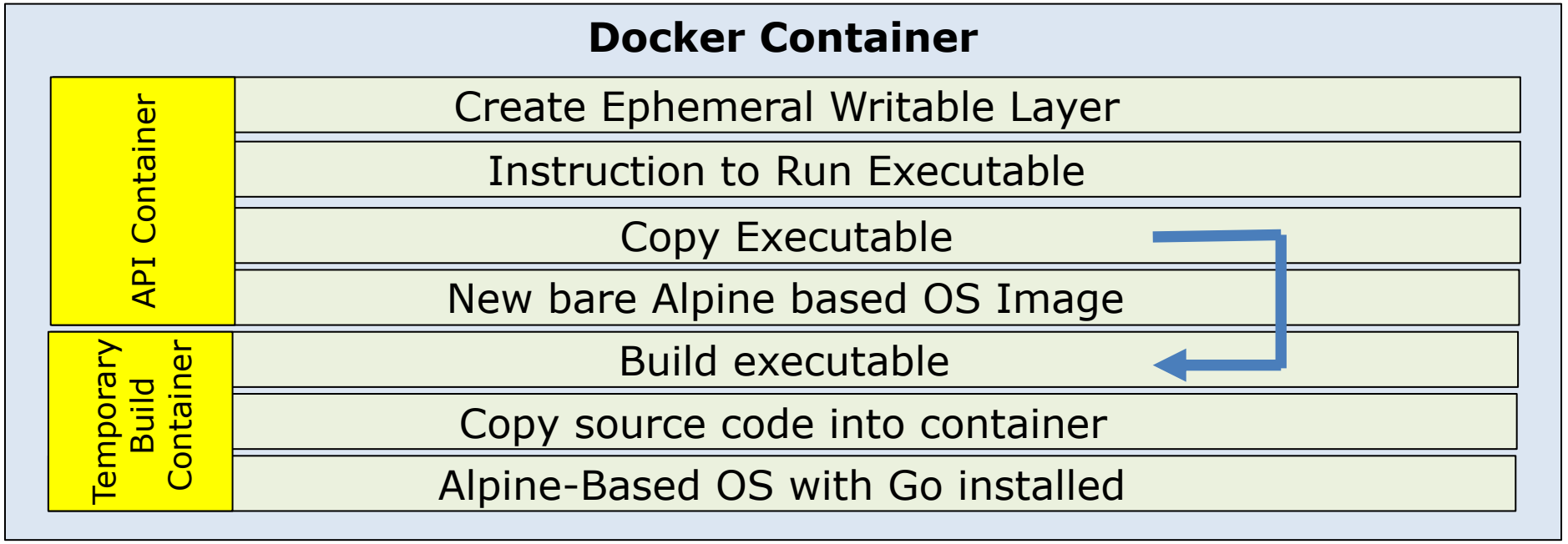



**Of architectural interest, is that fully managed distributed database services look more like microservices than they do typical databases**

# FaaS – AWS Lambda

To best understand lambda, lets take a step back to look at the docker architecture to build containers

```
docker build –t architecting–software/se577–bc–go –f Dockerfile .
```

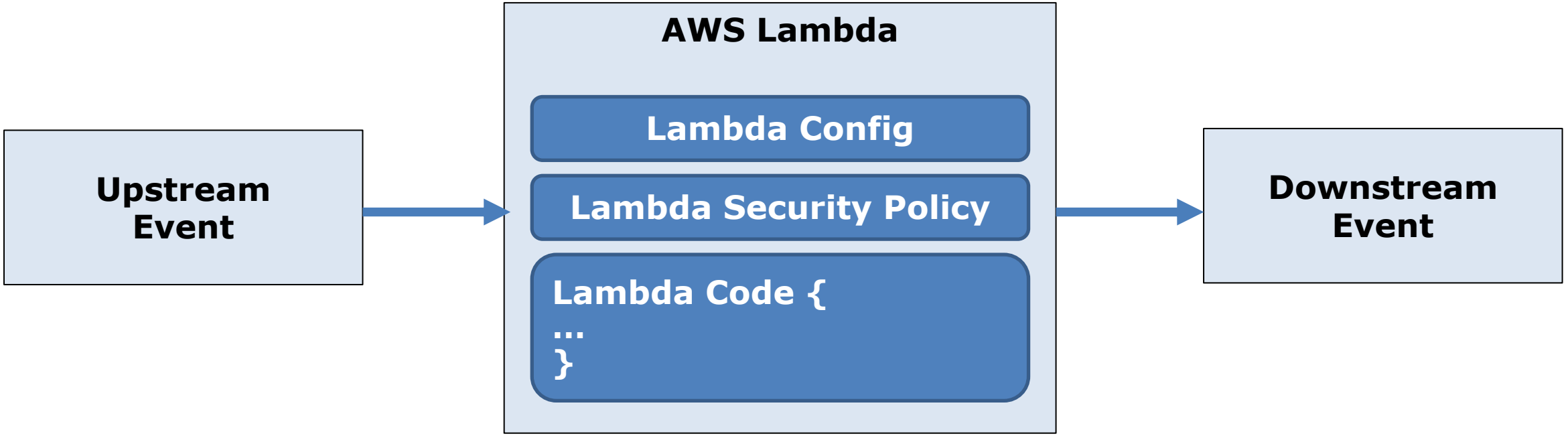



You can then run the code in the container

```
docker run –p 9095 architecting–software/se577–bc–go
```

This is a bit of work, what if you could just say “here is my code”, go run it?  That’s what AWS Lambda is about

# FaaS – AWS Lambda

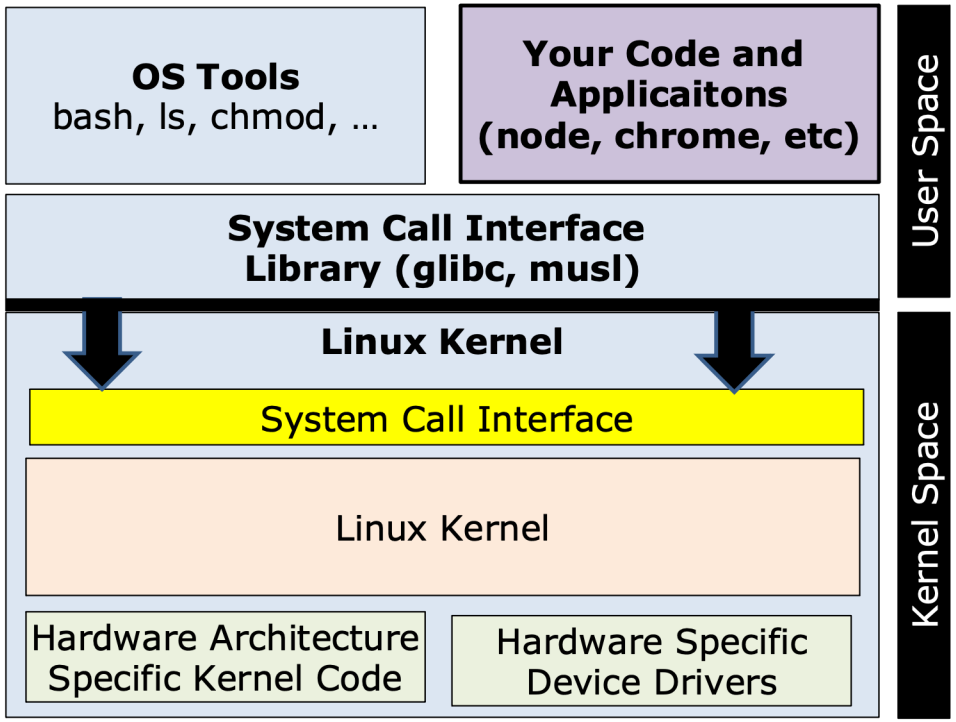



Lambdas run your code based on an upstream event (e.g., API Request, File Upload, IoT Event, etc), and can optionally trigger downstream events (e.g., send an SMS message)

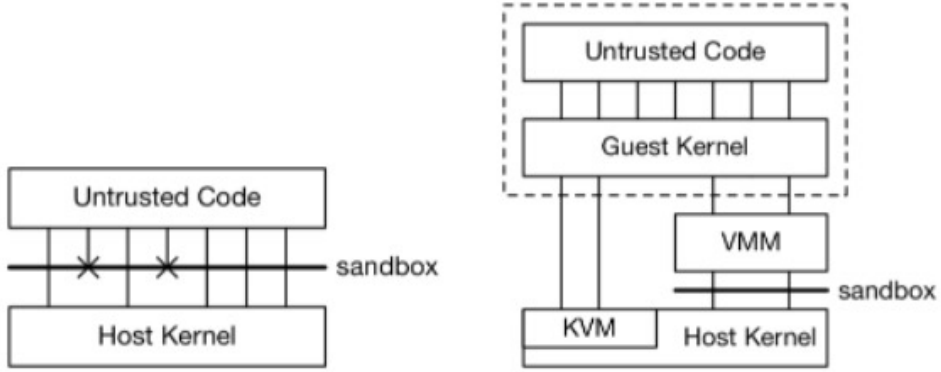
Lambdas must have really fast startup times to run your code at essentially infinite scale

# FaaS – AWS Lambda Architecture

Why do we need lambda if we have containers, they start fast after all?




Remember the container architecture, its very fast, but each container is visible to the host machine as a stand alone process and the containers share the host machine's kernel

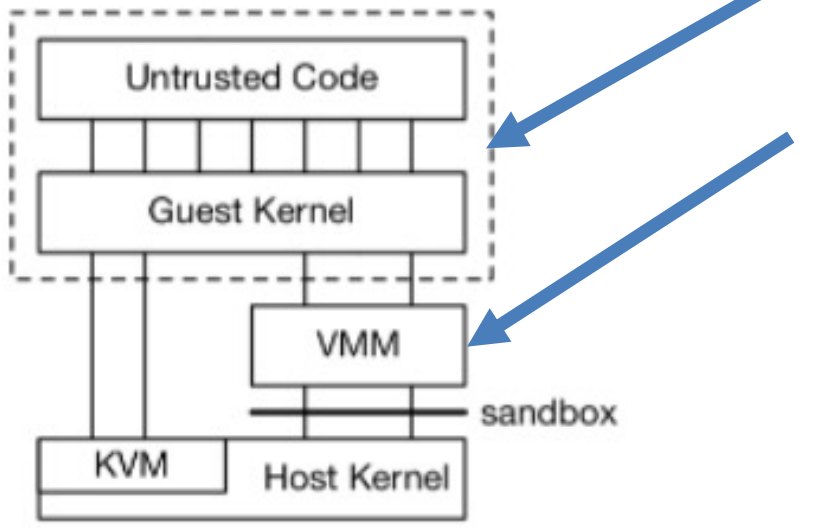



(a) Linux container model (b) KVM virtualization model

From the firecracker paper we see some of the potential security challenges of just using linux containers, a way to better isolate workloads is the KVM virtualization model on the left

# FaaS – AWS Lambda Architecture

Why do we need lambda if we have containers, they start fast after all?




(b) KVM virtualization model

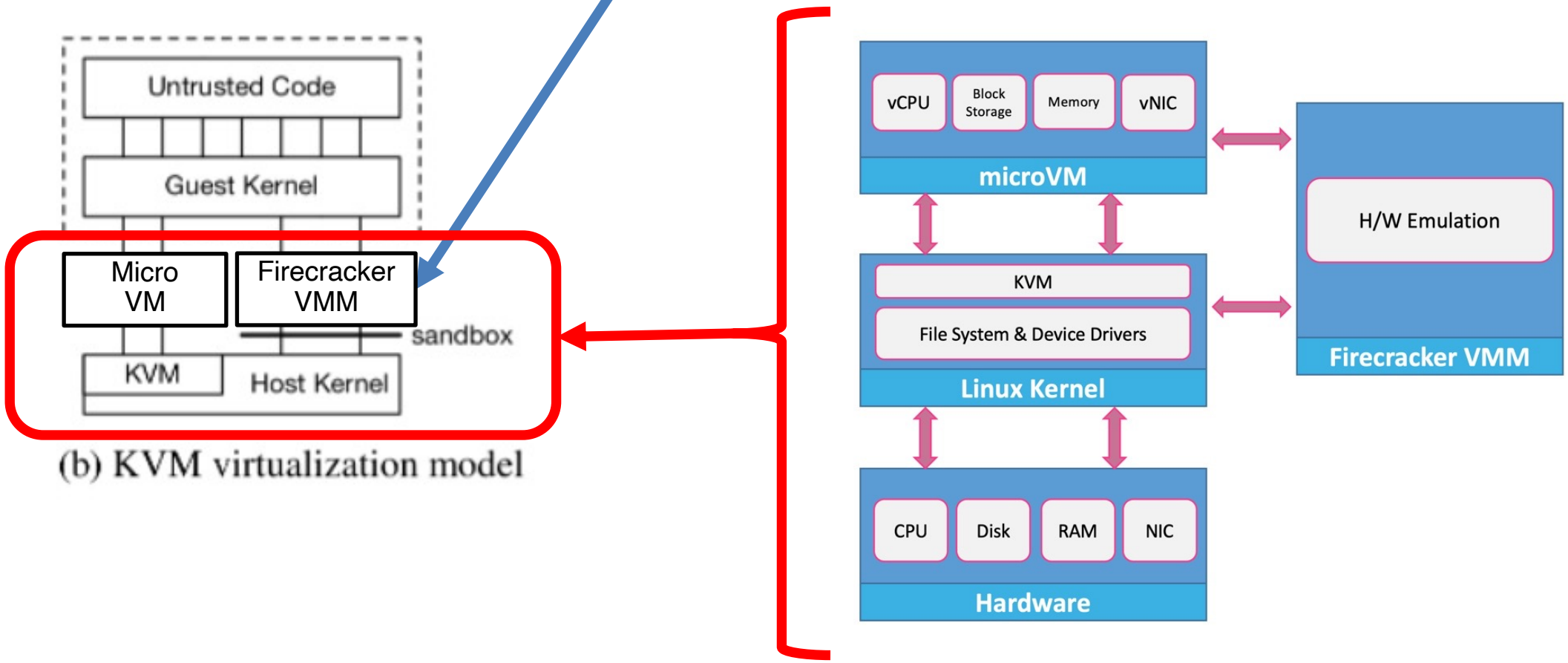
Equivalent to a container, but much more secure since they don't ride directly on the guest OS

Virtual Machine Managers tend to be large and complex pieces of software, and thus have slow startup times

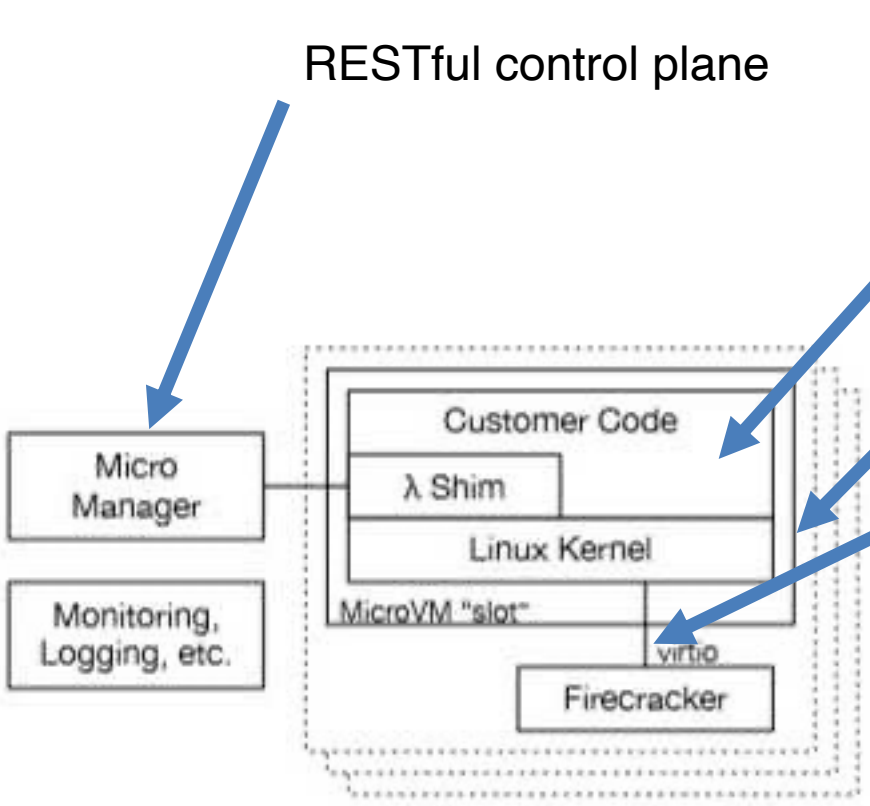
# FaaS – AWS Lambda Architecture

Why did I assign the firecracker paper?

Firecracker is a highly optimized VMM along with a Micro VM that is designed from the ground up to support serverless workloads.



# FaaS – AWS Lambda Architecture

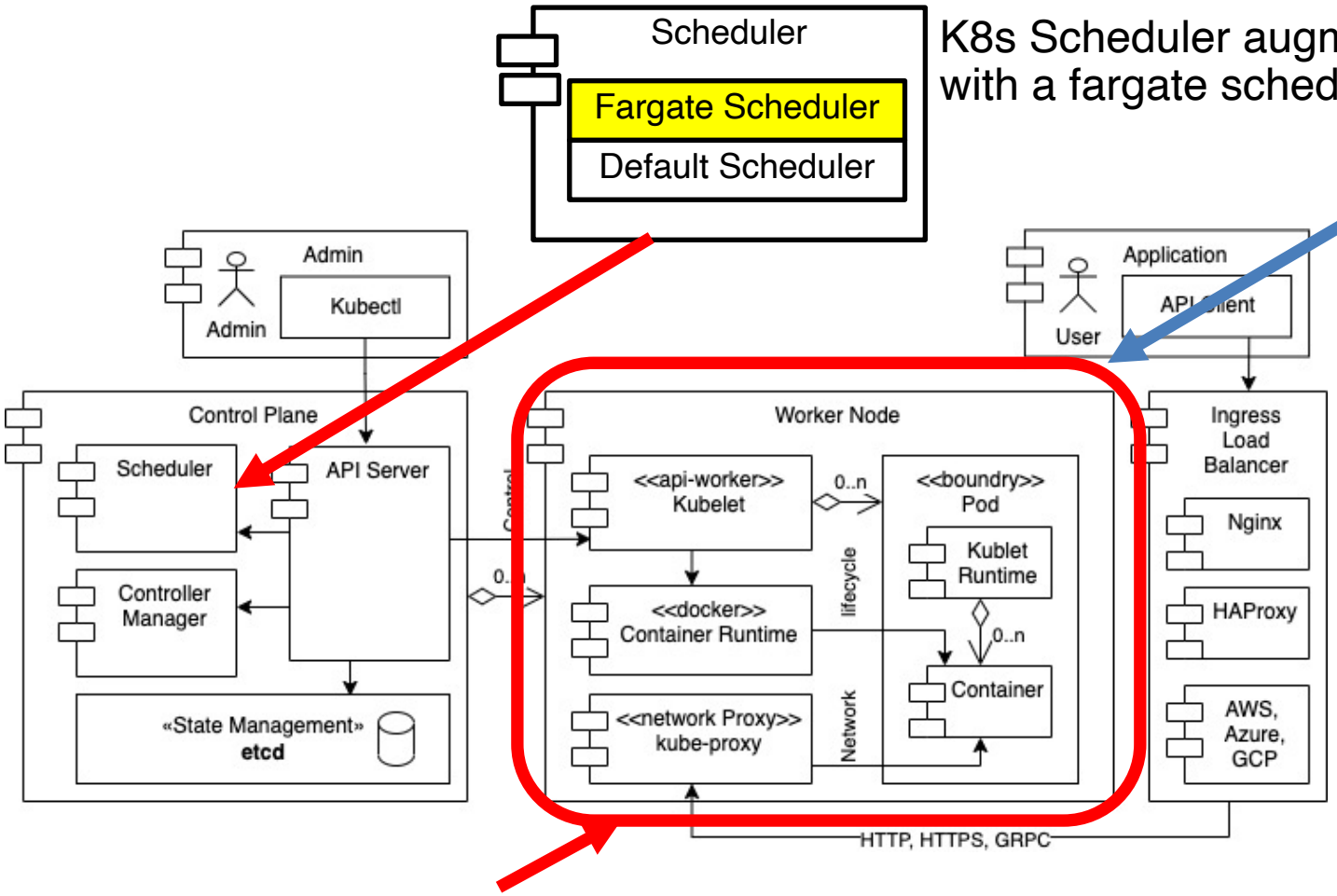



Firecracker has been tested and proven via billions of lambda serverless executions and fargate container invocations

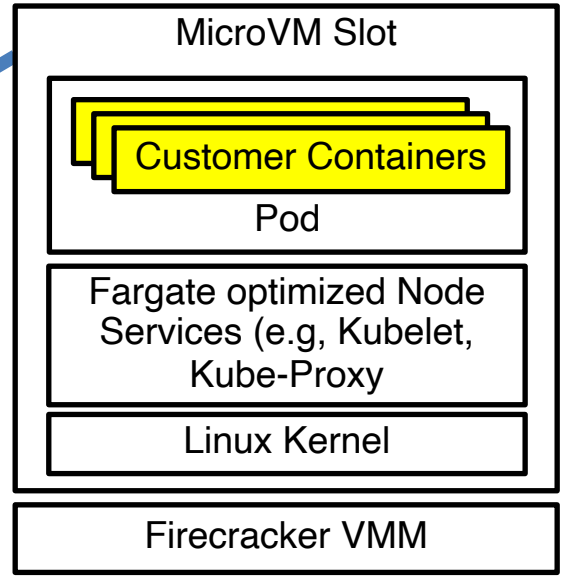
Startup time is about 100ms

Density is about 4000 instances per host

# Caas/FaaS Hybird – AWS Fargate




K8s Scheduler augmented with a fargate scheduler




Fargate nodes preallocated and spun up to act like a dynamic worker node, the pods are basically started serverless

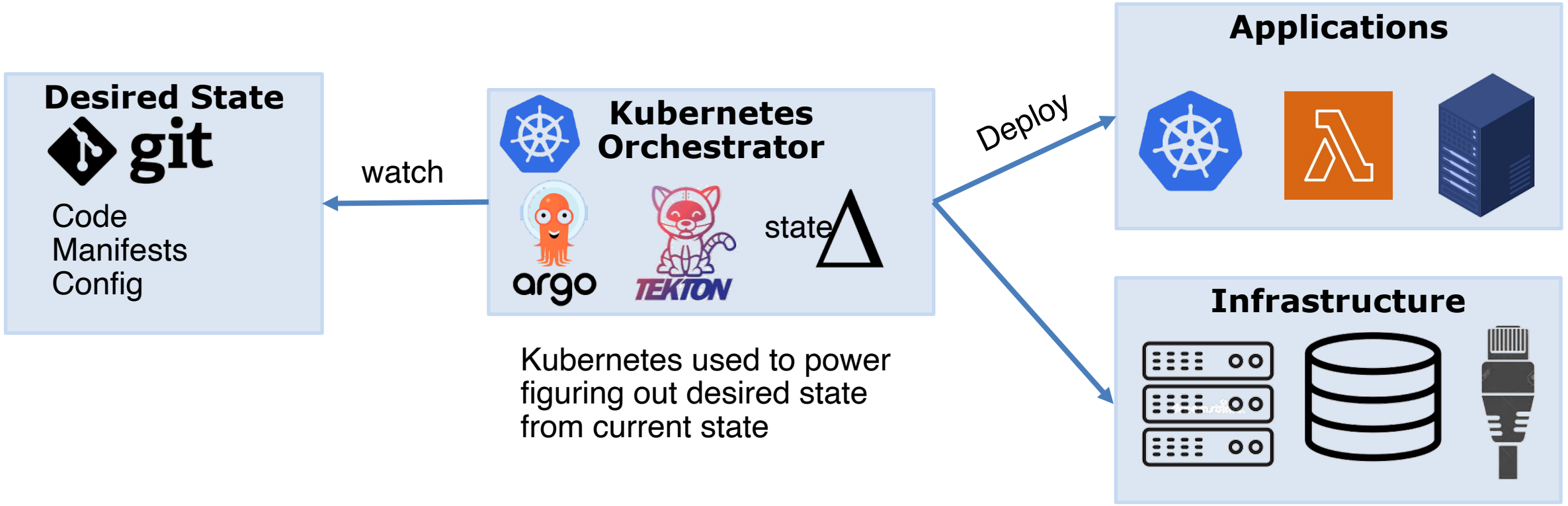
Recall, the Kubernetes architecture has predefined nodes spun up, where each node runs one or more workloads via pods. Generally, nodes have a preconfigured number of pods that they start and can scale up and down from there. Scaling pods generally takes in the order of seconds with the default scheduler, microseconds with fargate and firecracker

# Back to Cloud Native

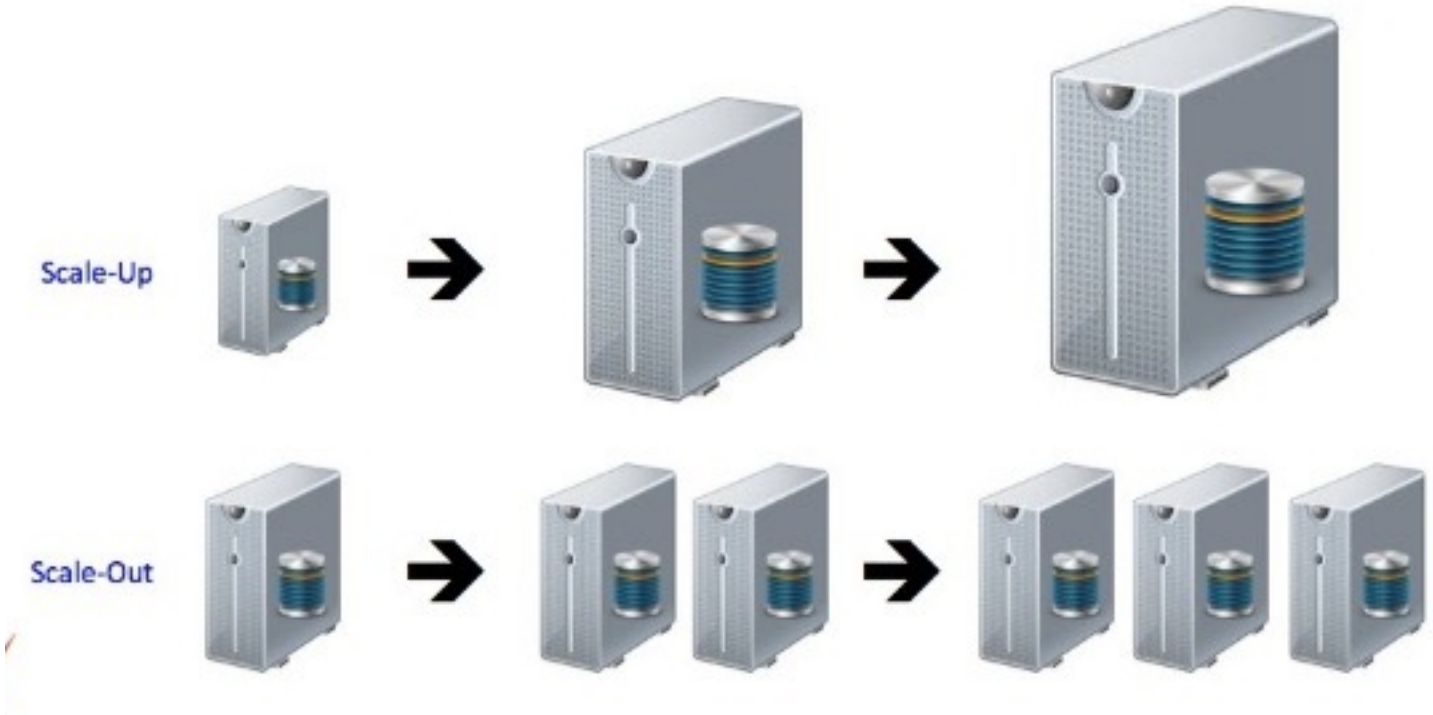
- Now that we have a good handle on different types of workloads in cloud native architectures
- Some other things to consider
  - Managing cloud resources and application assets via automation – gitops
  - 12 factor apps and architectural considerations for software design of cloud native apps
  - Best practices based on learnings over the years – example, AWS Well architected framework

# GitOps – Best practices for automation of cloud native workloads

GitOps is about putting git in the center of managing the CI/CD process. Git maintains the desired state, and CI/CD tools such as argo and tekton leverage Kubernetes to figure out how to transition the current state into the desired state



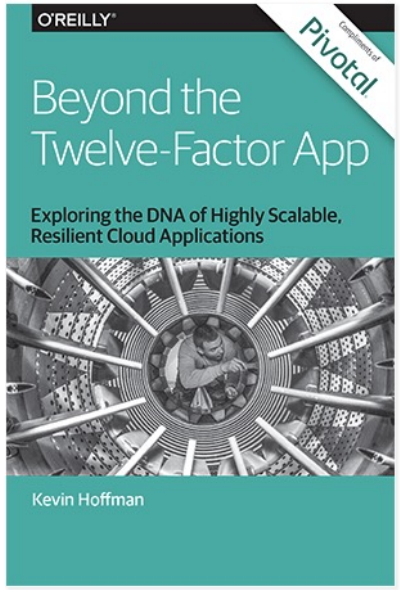
# Moving from Scale-Up to Scale-Out architectures




**One of the biggest considerations for architectures that support cloud native computing concepts is to favor scale-out over scale-up. If your architecture cannot support scale-out, it is not really aligned with cloud native computing concepts. WARNING WARNING WARNING Scale-out takes you into the realm of needing to be very good at distributed computing**

# Design and Architecture best practices that have created the foundation for cloud native


**12 software design practices proposed by Herouku (https://www.heroku.com/) for building modern scalable apps – Published 2011**



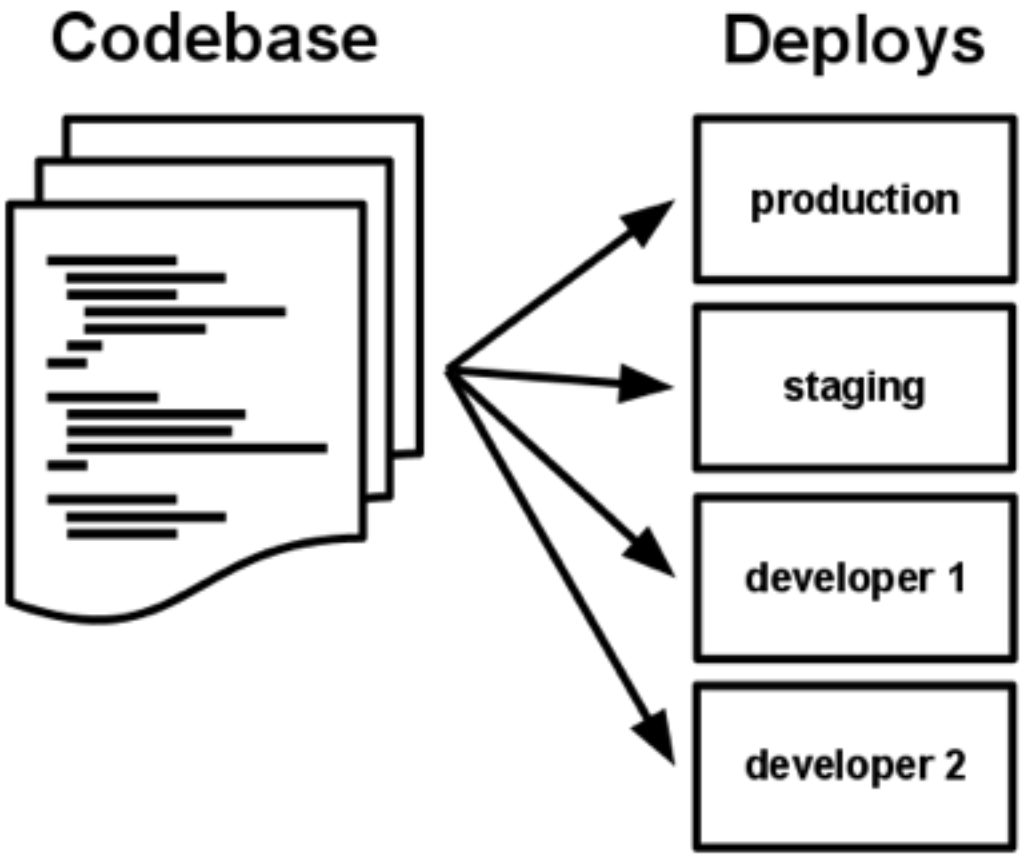

**In 2016, these concepts were updated with 15 practices by Pivotal**

# Architectural best practices for modern applications

1. **One codebase, one application**
2. **API first**
3. **Dependency management**
4. **Design, build, release, and run**
5. **Configuration, credentials, and code**
6. **Logs**
7. **Disposability**
8. **Backing services**
9. **Environment parity**
10. **Administrative processes**
11. **Port binding**
12. **Stateless processes**
13. **Concurrency**
14. **Telemetry**
15. **Authentication and authorization**

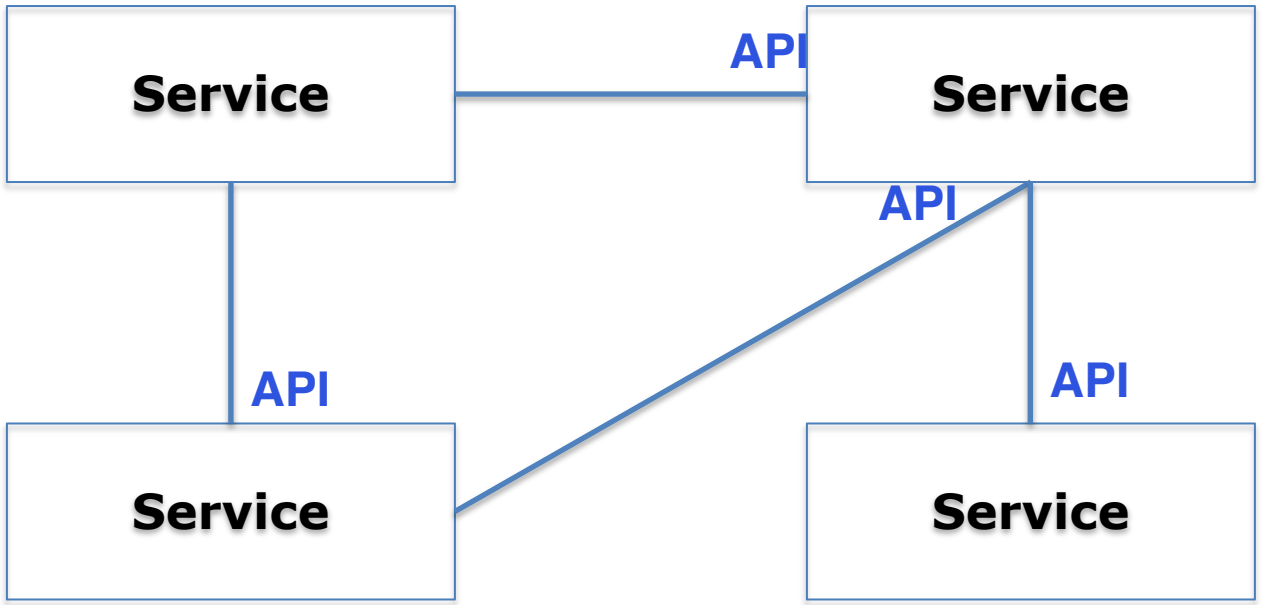
These practices apply towards modern distributed apps – web, mobile, cloud, IoT

# 1. One Codebase



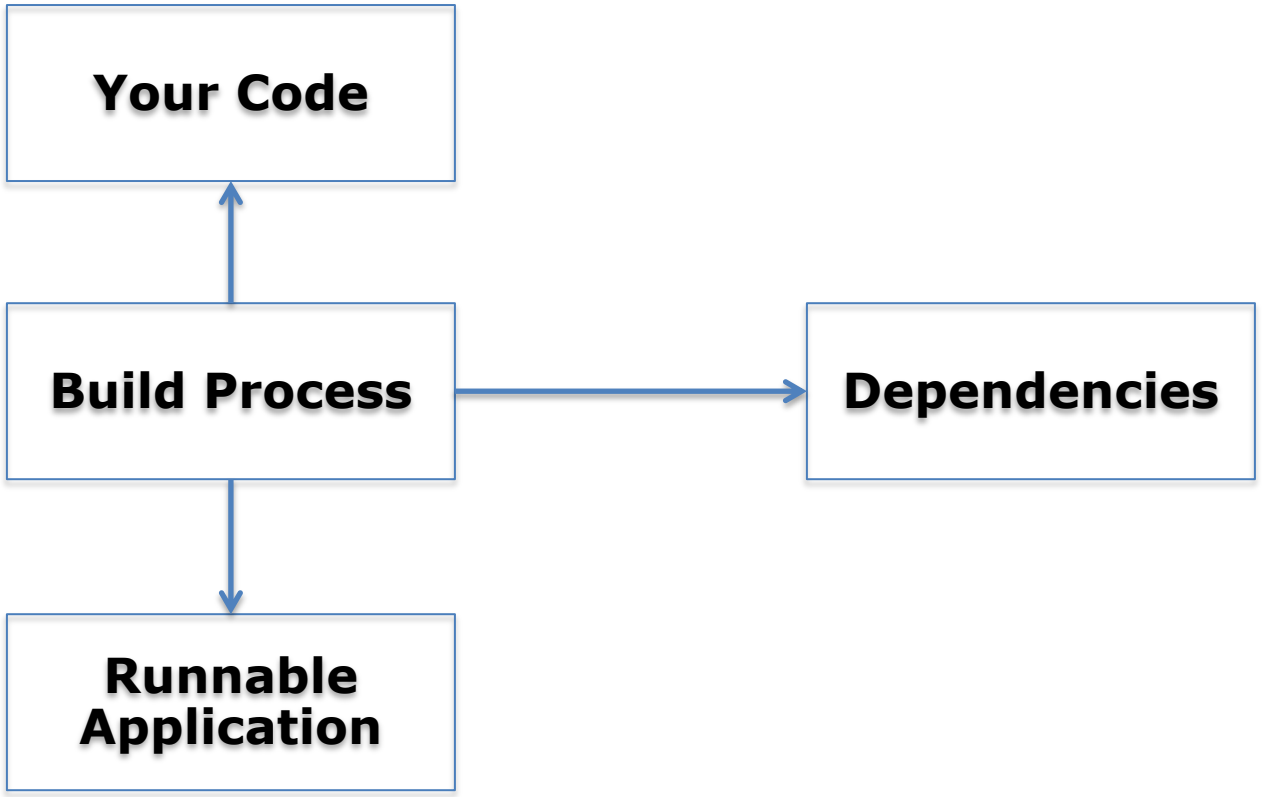

**A codebase for an application is used to create any number of immutable releases that**
**are likely destined for different environments**

# 2. API First



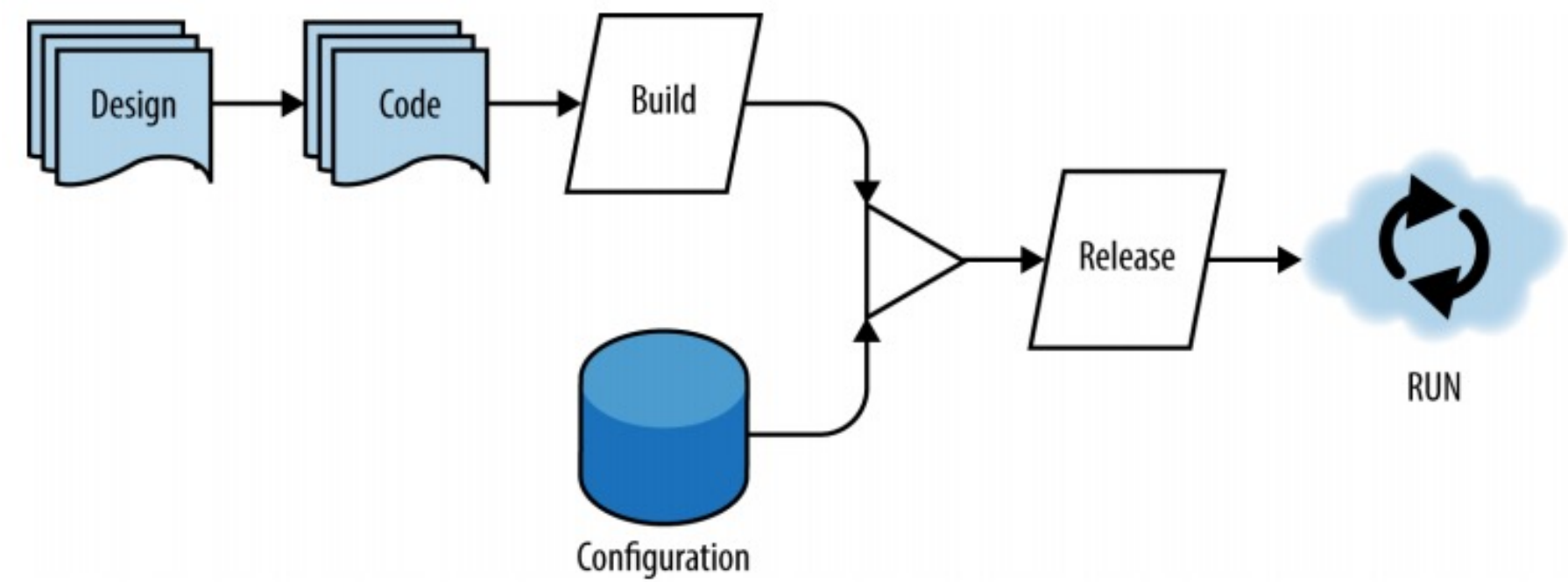

**All externally visible application interfaces are exposed via well-defined APIs. Define the API-interfaces first for clients and other services**

# 3. Dependency Management



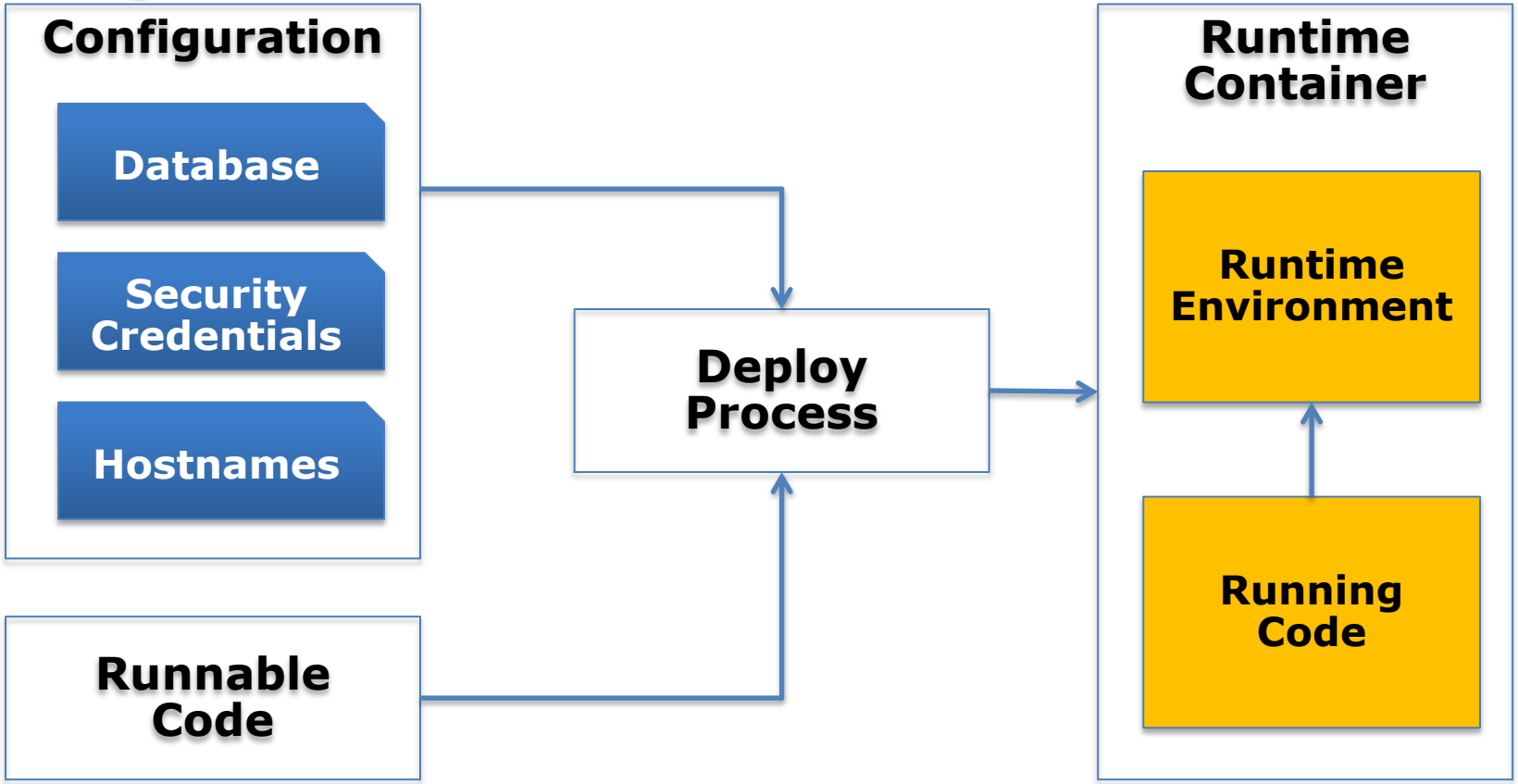

**Never rely on the existence of system-wide dependencies. Explicitly declare all dependencies (and versions) pulling them in dynamically at build time (maven, sbt, npm, yarn, go, cargo, etc)**

# 4. Design, Build, Release and Run



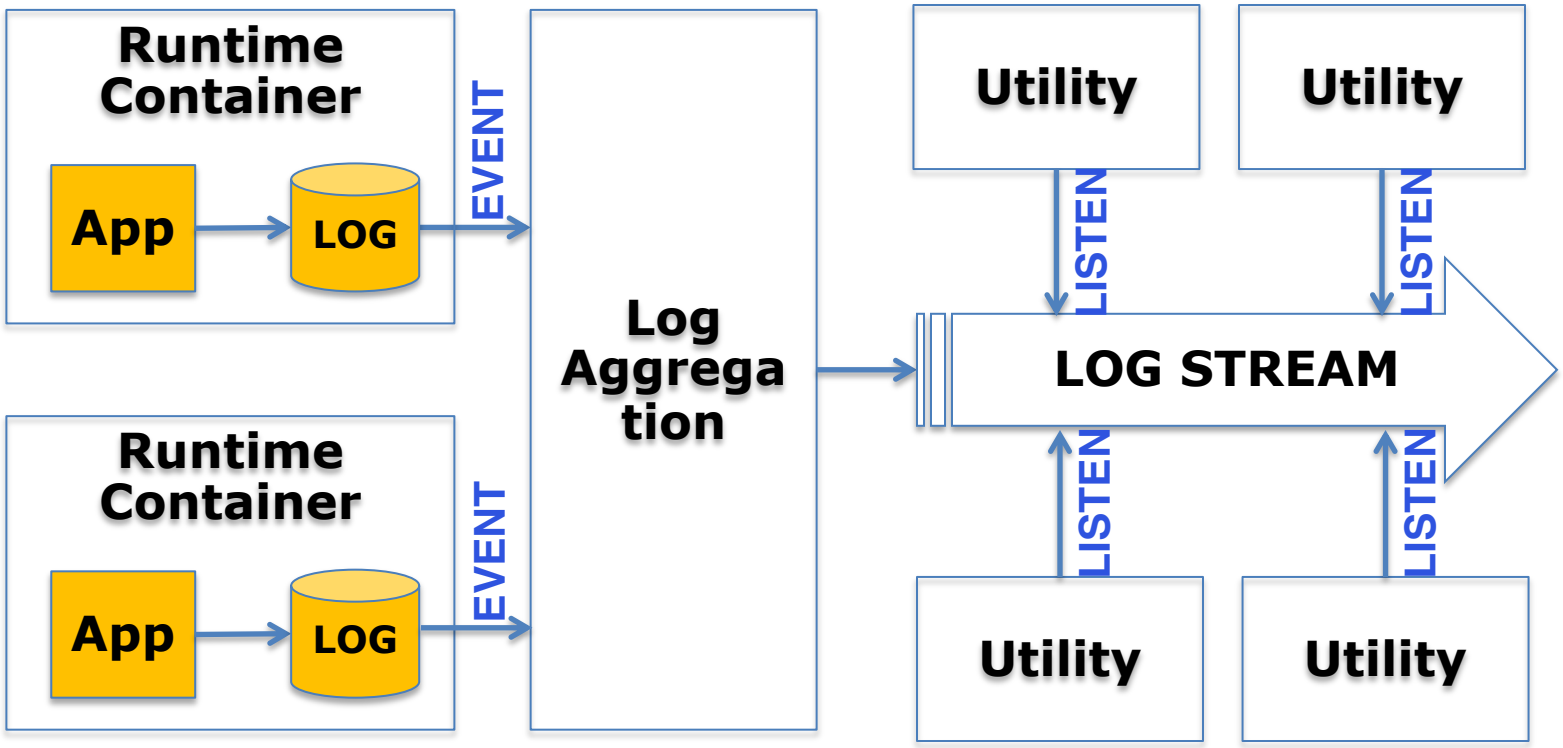

**Every release is coordinated via an automated build process that injects the proper runtime configuration. Supports a CI/CD pipeline. This is at the center of the gitops based patterns**

# 5. Configuration, Credentials and Code




**Treat configuration as code – configuration is everything likely to change between deploys. Ideally deployment configuration changes are injected into the runtime as environment variables. Again, we see this in gitops**

# 6. Logs

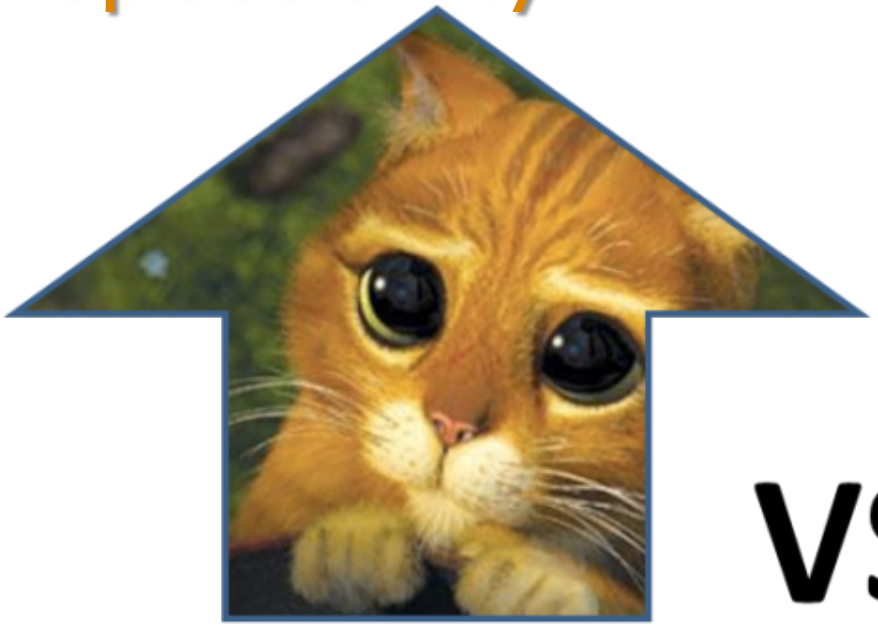



Logs provide insight into the behavior of a running app. Logs should be considered as an event stream, capturing all events from a system in a time ordered sequence. Each app instance is only concerned writing locally to its log, aggregation and streaming are handled outside of the application runtime. All modern public cloud offerings support these capabilities intrinsically, and third party solutions that aggregate logs are very powerful (e.g., splunk)

splunk>

# 7. Disposability

**“Pets vs Cattle”**



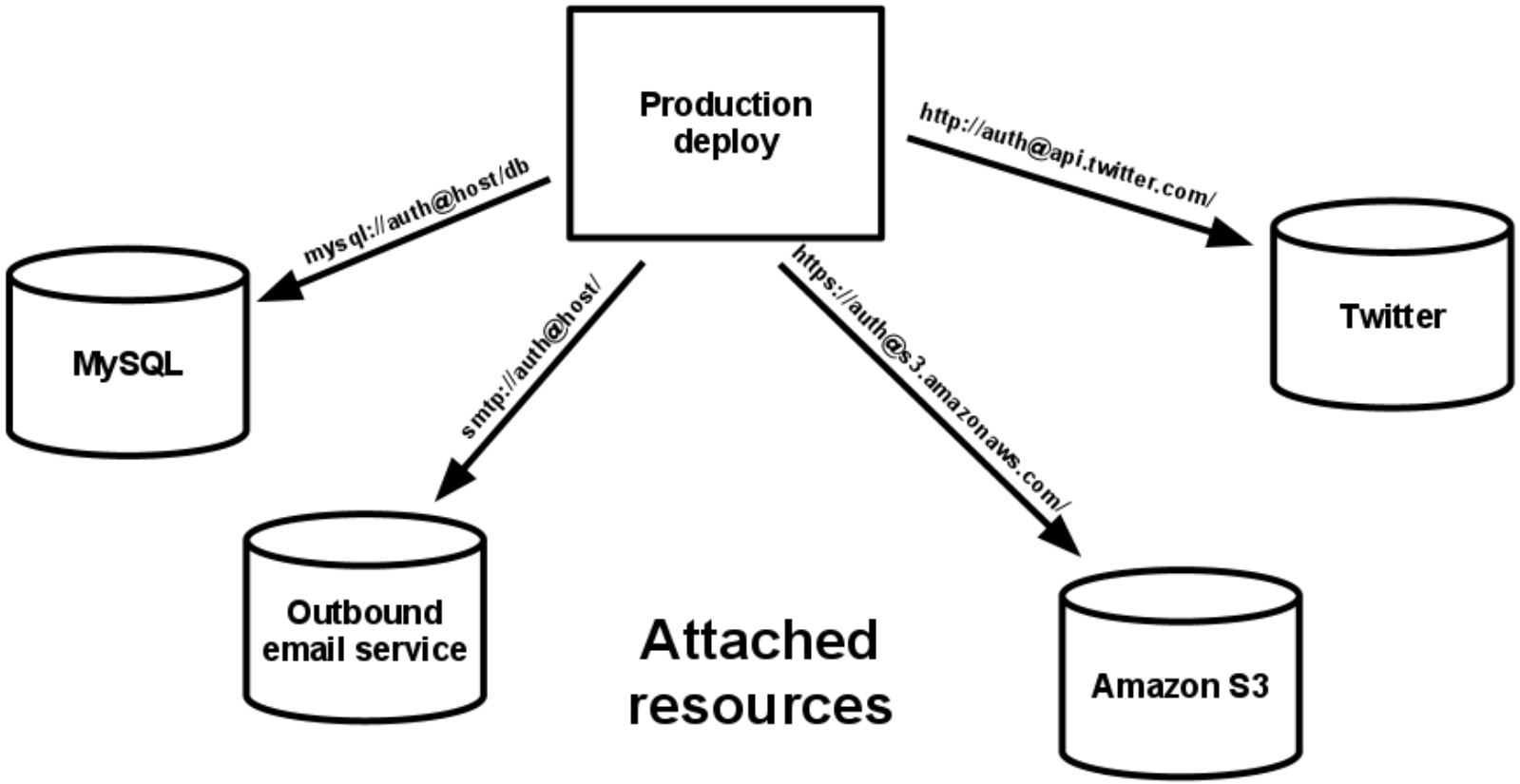
Scale Up

VS

Scale Out

**Application instances should be able to be started quickly, and terminate cleanly. Repair is never done, instances are disposed and replaced. Favor light/thin frameworks over heavy/fat middleware. This is the key enabler towards scaling out versus scaling up. EVERYTHING SHOULD BE EPHEMERAL in cloud native architectures.**

# 7. Backing Services




**Treat backing services as attached resources. They are attached via a URL or other location and and security credentials managed by the config. Resources should be able to be attached and detached to the application on demand without any code changes.**

# 9. Environment Parity

**Time**

***Becomes more important as the check-in to deployment cycle time decreases. Days? Months? Minutes***

**People**

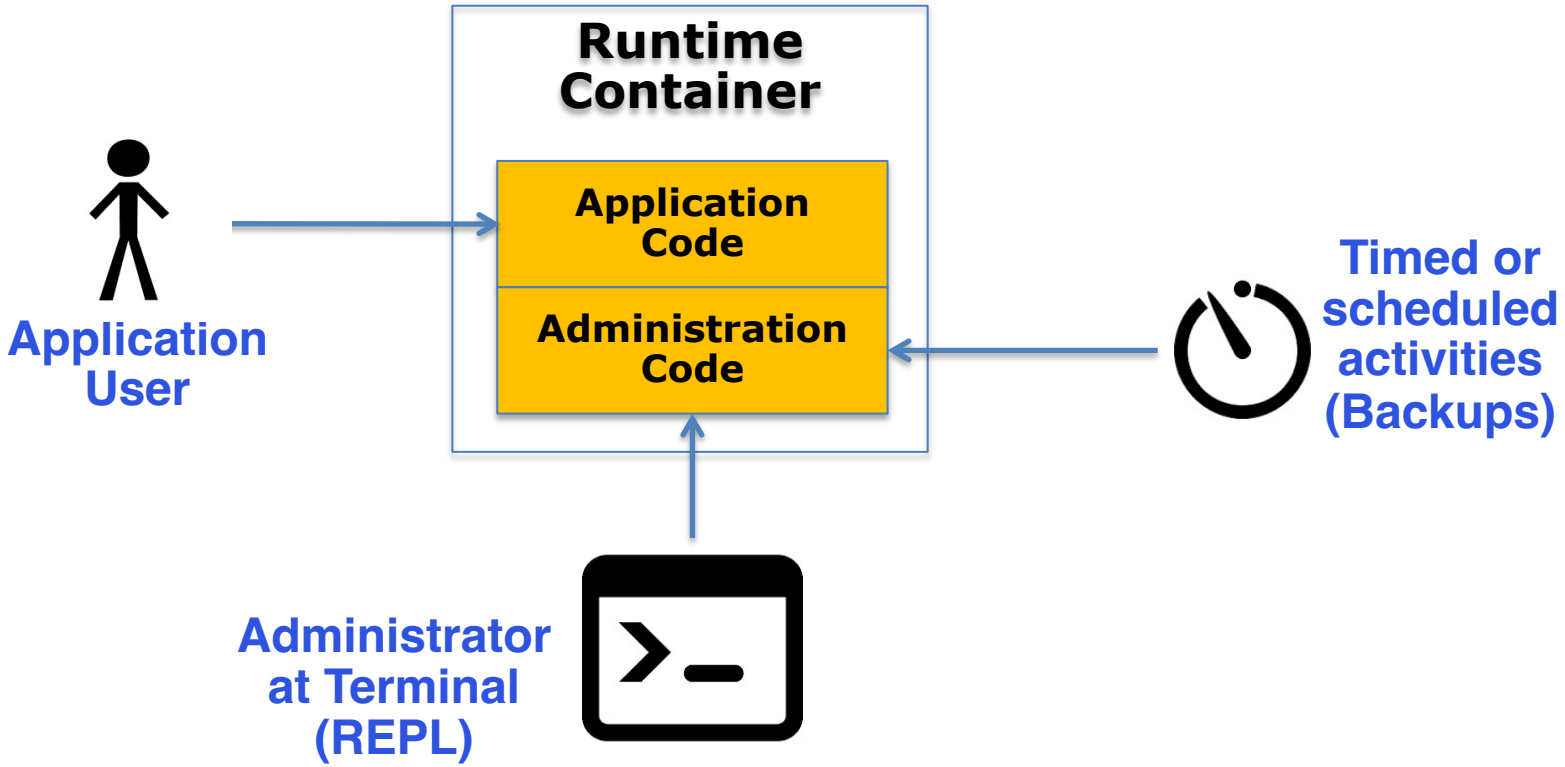
***Specialized Skillsets, or "if you build it, you deploy and run it"***

**Resources**

***Expensive, specialized resources (Oracle DB), or general purpose open source (Postgress)***

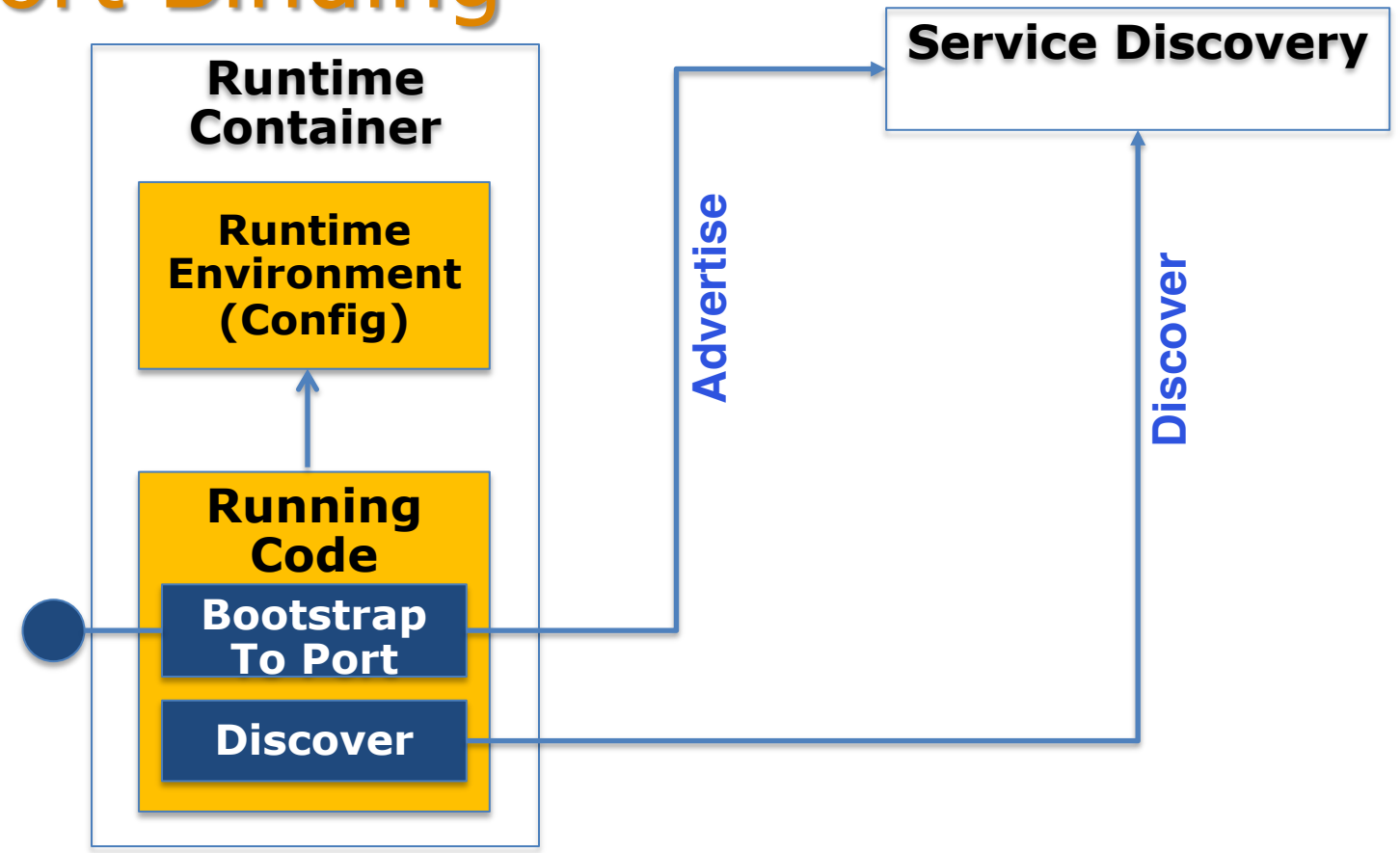
**Keep all environments as similar as possible. Need to avoid "it works on my machine" scenario. Might be impossible to totally maintain consistency, but it should be maximized as much as possible. Can be mitigated with other factors such as Backing Services, Configuration as Code, etc**

# 10. Administrative Processes



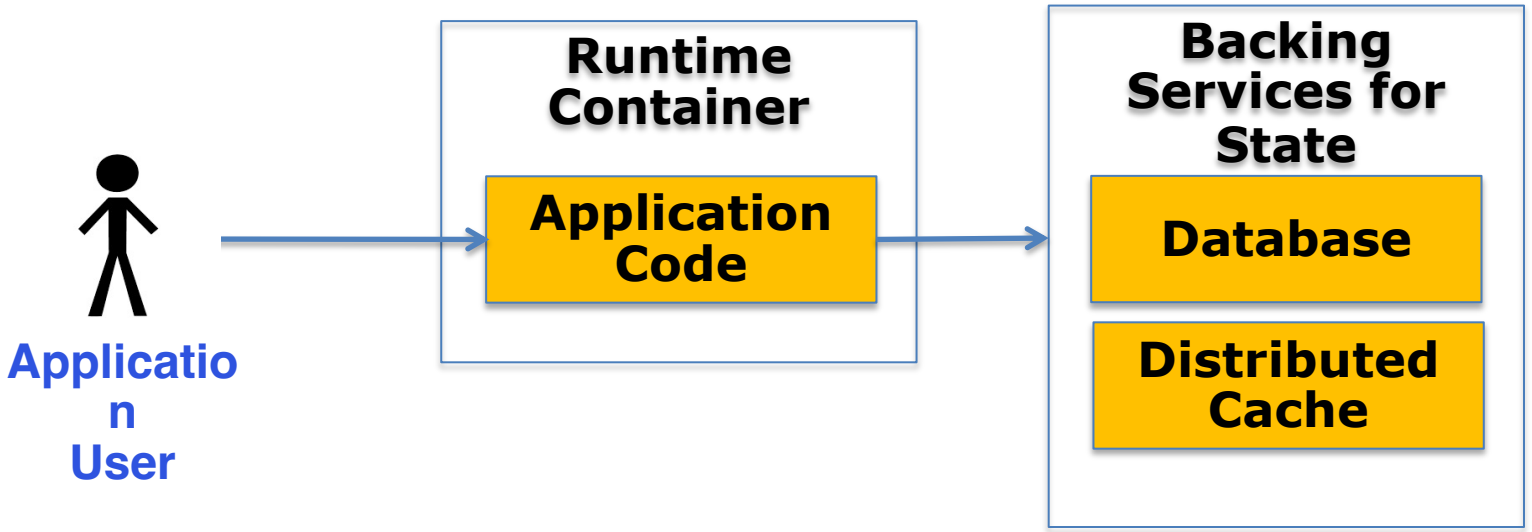

**Run administrative and management tasks as one-off processes. Admin code should be managed and deployed with the underlying application code. Admin actions should execute in the same runtime configuration as the underlying application**

# 11. Port Binding



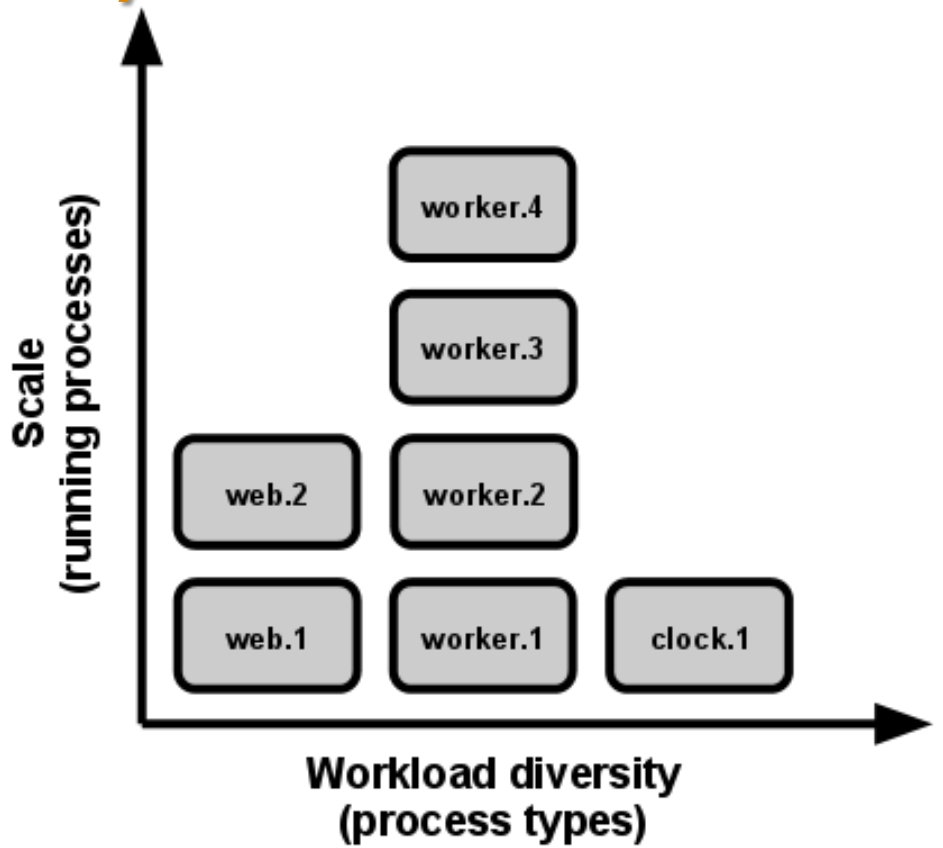

**Export services via port binding. Use a service discovery mechanism to locate and bind to other services. Examples, ETCD and Consul – basically a highly dynamic DNS service. In AWS, all resources can be discovered by ARN, also many managed services provide these capabilities.**

# 12. Stateless Processes



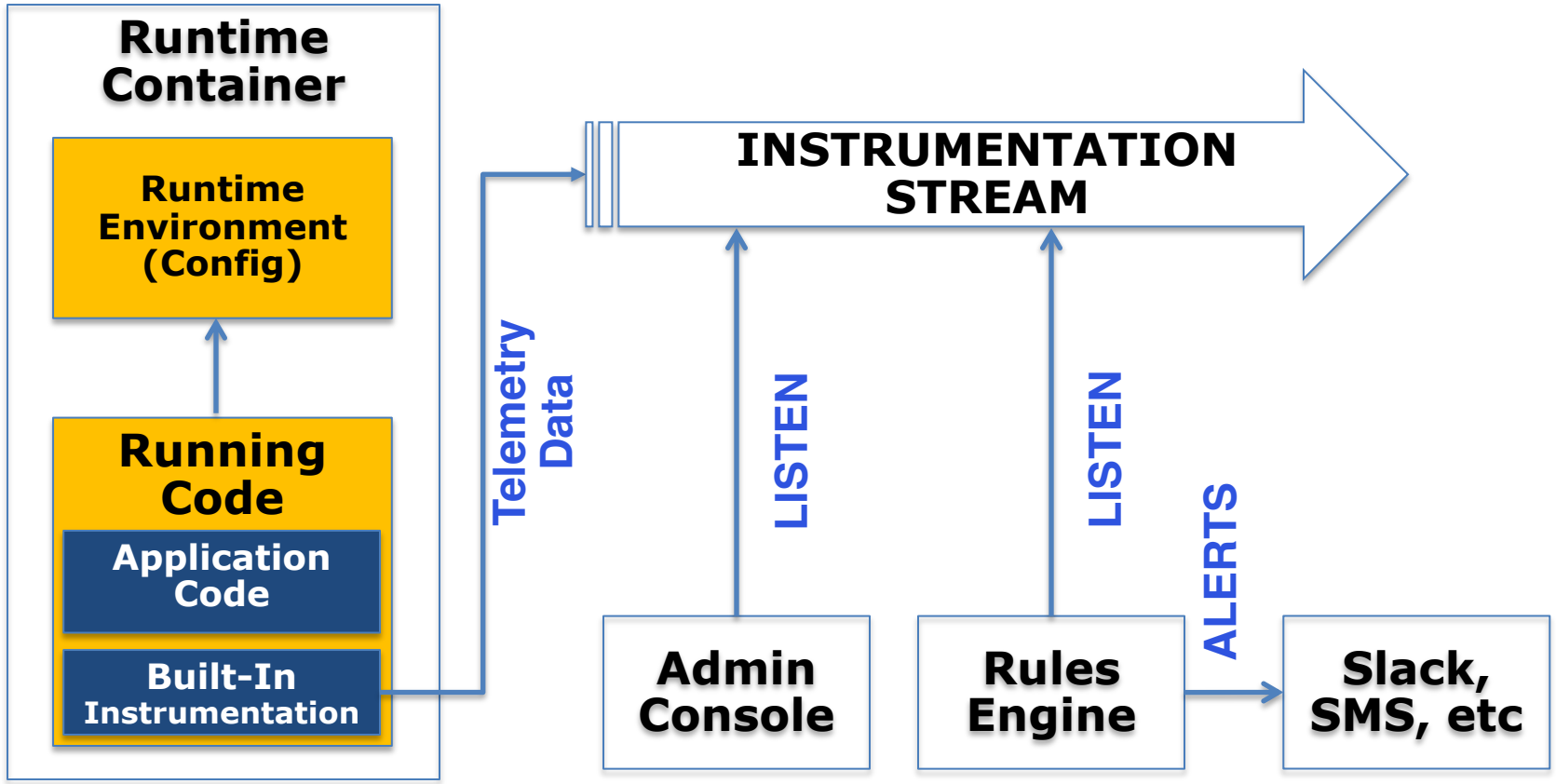

**Services should be "share nothing" and stateless. Any state or persistence operations should be handled in a backing service such as a cache or database. This is a key attribute of resiliency and scale**

# 13. Concurrency



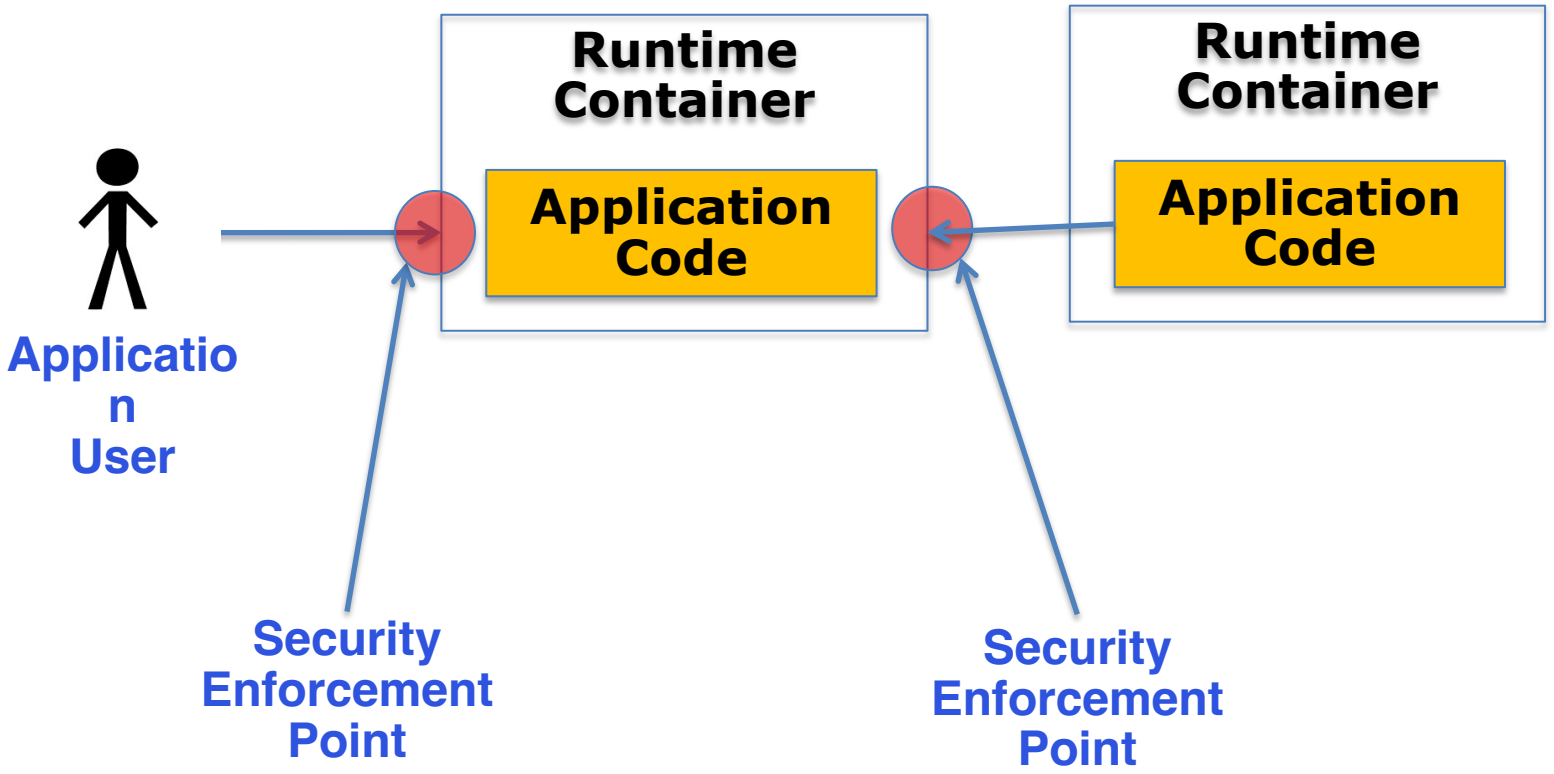

**Scale out via the process model. Dynamically be able to start and stop various aspects of your application based on load or needs. This is different from a scale-up model where scaling involves increasing all services equally.**

# 14. Telemetry



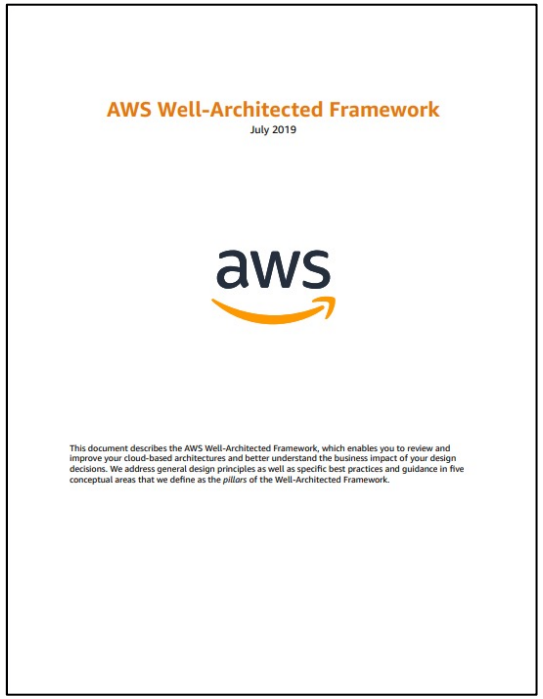

**All well designed application have built in instrumentation to communicate on application health (e.g., performance, errors, resources, etc). Examples, AWS Cloud Watch, Dynatrace, New Relic**

# 15. Authentication and Authorization




**Every request to your application should know about who (person or service) who is making the request and the roles that they have that dictate access. See OIDC and oAuth as examples. AWS handles this via security groups, network NACL, and a variety of other policy based services in IAM**

# Well Architected – Cloud Native – Example AWS Well Architected Framework





**https://aws.amazon.com/architecture/well-architected**