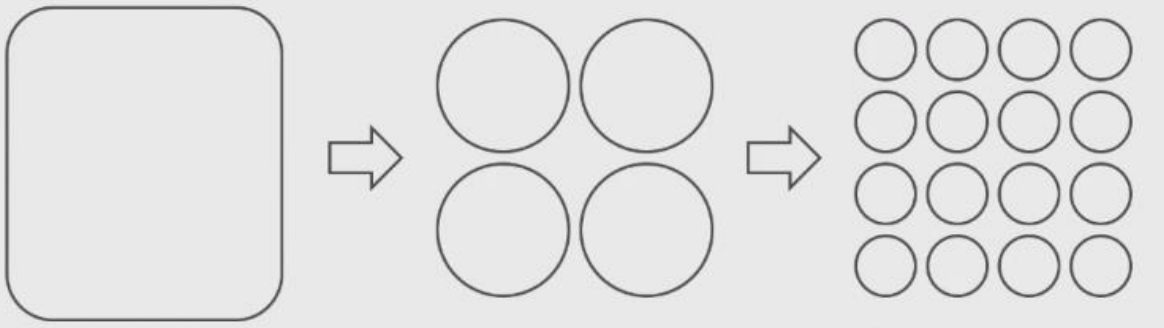
Running and managing large scale applications with microservices architectures is hard and often requires operating complex container management infrastructure. Amazon EC2 Container Service (ECS) is a highly scalable, high performance container management service that supports Docker containers and allows you to easily run applications on a managed cluster of Amazon EC2 instances. In this session, we will walk through a number of patterns used by our customers to run their microservices platforms. We will dive deep into some of the challenges of running microservices, such as load balancing, service discovery, and secrets management, and we'll see how Amazon ECS can help address them. We'll also hear from Instacart how they use a blue/green deployment process to deploy services to ECS and how they manage configuration with a RDS-based metadata service.

Overview of Microservices Architecture

What are microservices?

"**A software architecture style** in which complex applications are **composed of small, independent processes** communicating with each other using language-agnostic APIs. These services are **small, highly decoupled** and focus on **doing a small task**, facilitating a **modular approach** to system-building." - Wikipedia

Monolithic vs. SOA vs. Microservices

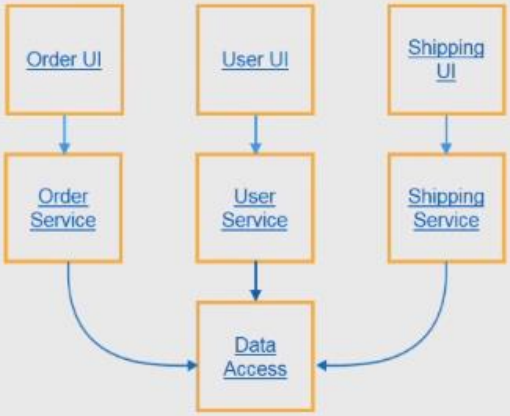


Monolithic
Single Unit

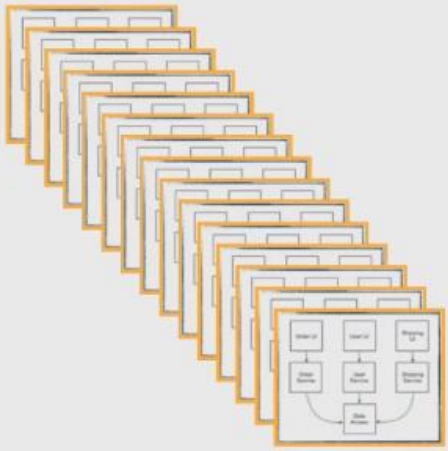
SOA
Coarse-grained

Microservices
Fine-grained

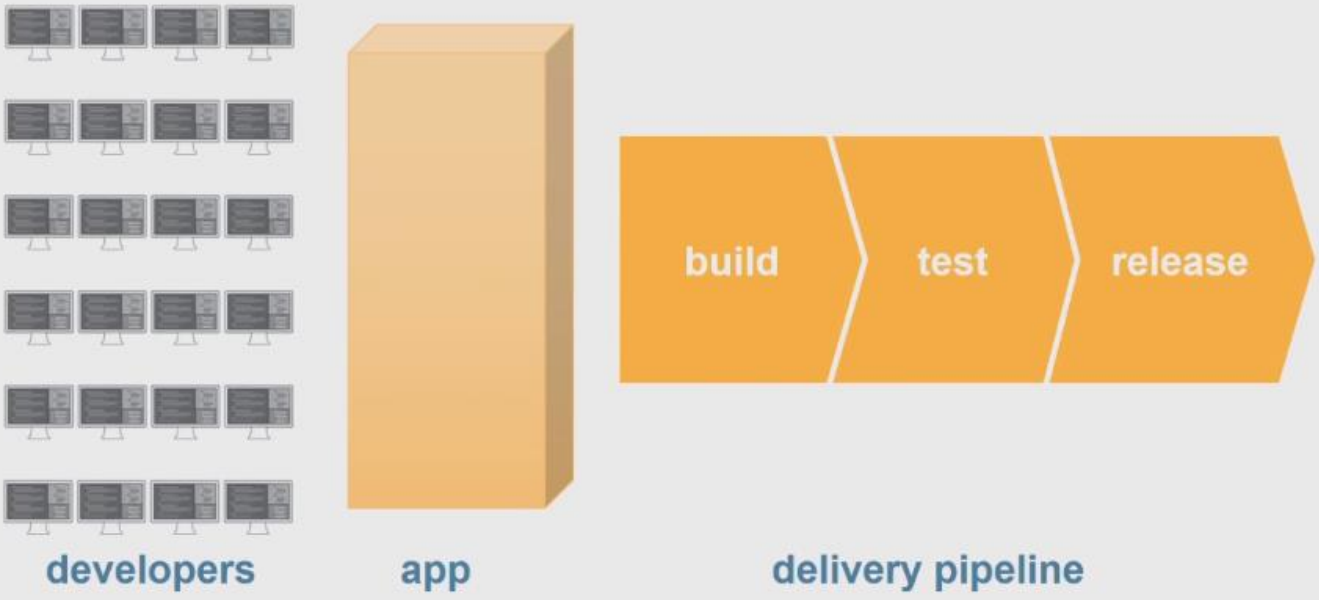
Monolithic Architecture



Monolithic Architecture – Scaling

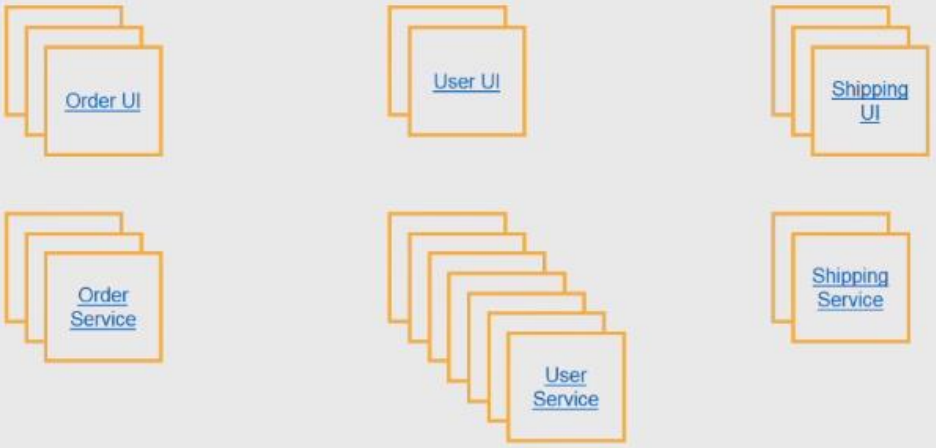


Monolith Development Lifecycle

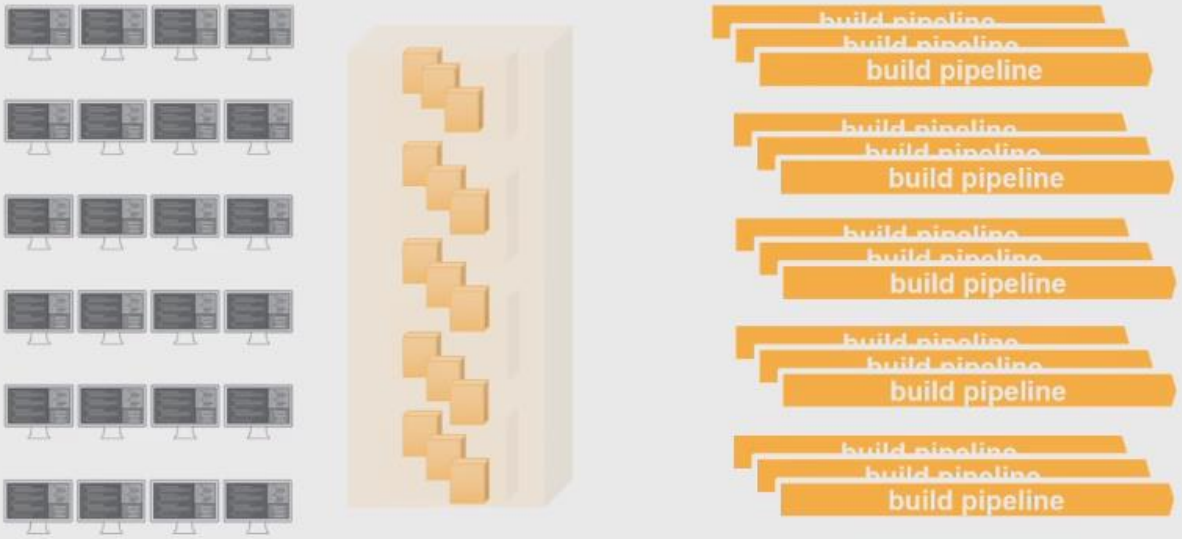


Microservices Architecture

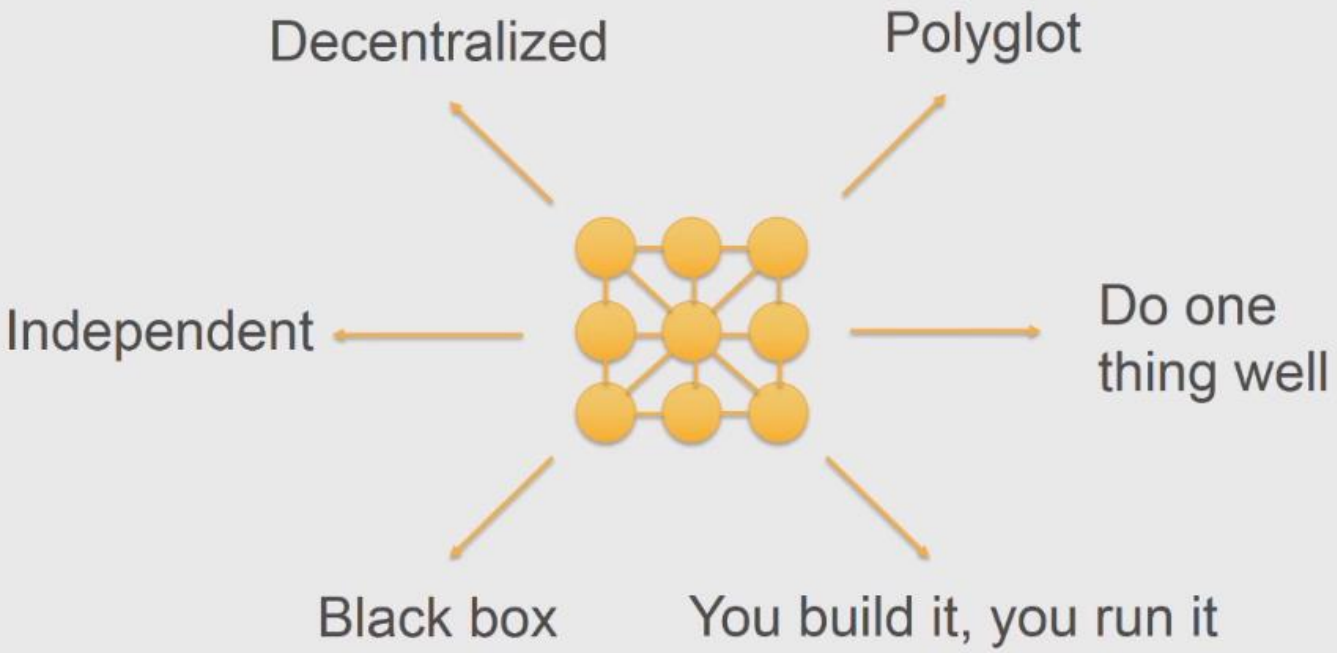
Microservices Architecture – Scaling



Microservices Development Lifecycle



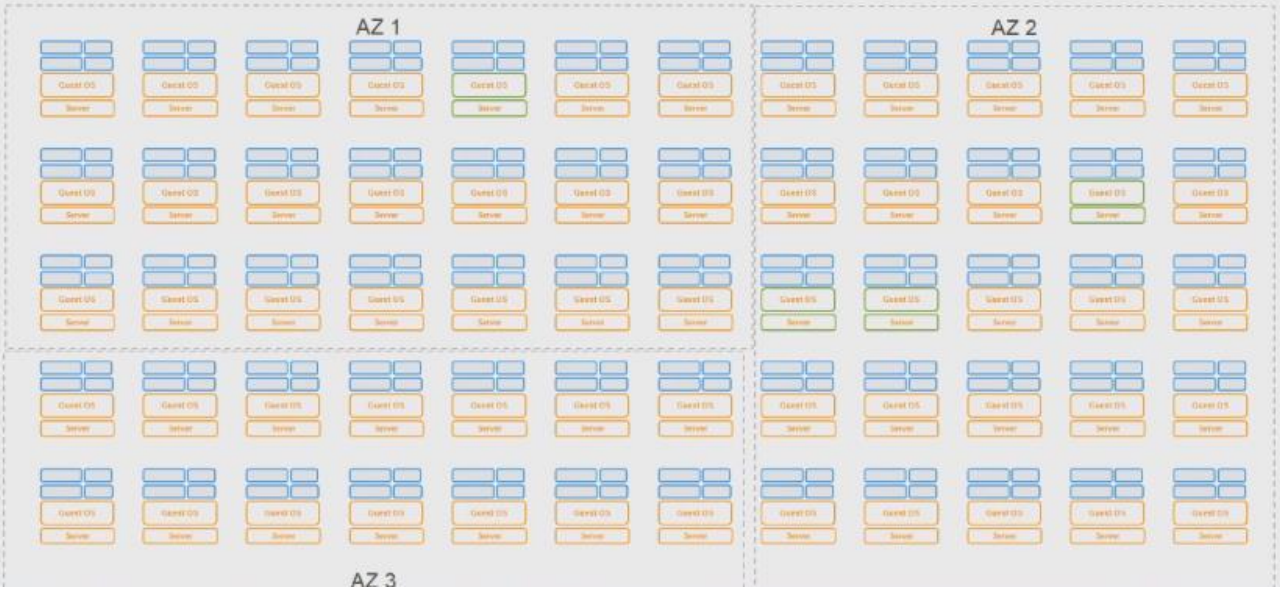
Characteristics of Microservice Architectures



Challenges in running microservices

Microservice Challenge #1 – Resource Management

Managing a large fleet by hand is impossible:



Microservices Challenge #2 – Monitoring

A microservices architecture will have 10s, 100s, 1000s, maybe even 10,000s of individual services:

- How do you know if an individual service is healthy?
- How do you measure the performance of an individual service?
- How do you troubleshoot and debug an individual service?

Microservices Challenge #3: Service Discovery

Each microservice scales up and down independently of one another:

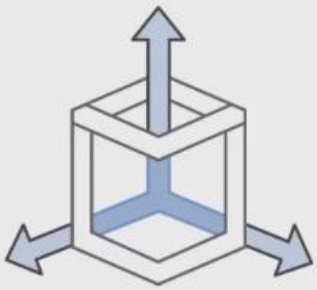
- How does Service A know the URLs for all instances of Service B?
- How do you allow services to scale independently while still using load balancers?
- How does a new instance of a service announce itself to other services?

Microservices Challenge #4: Deployment

A microservices architecture will have 10s, 100s, 1000s, maybe even 10,000s of individual services:

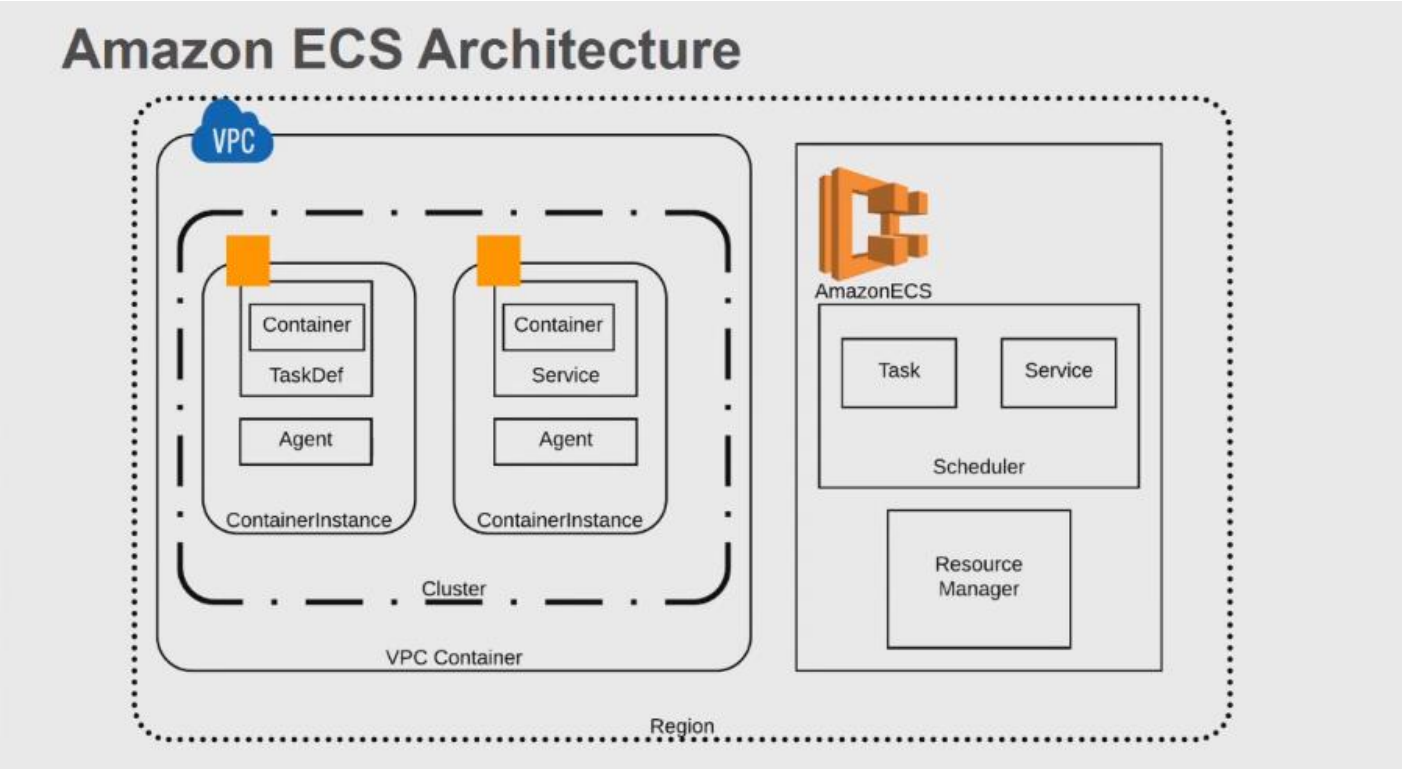
- Each service will be developed, tested, and deployed on its own timeline – How do you manage this across large numbers of services?
- Services are polyglot – different languages, frameworks – how do you efficiently deploy them?
- How do you decide which hosts to deploy a service on?

Introducing Amazon ECS

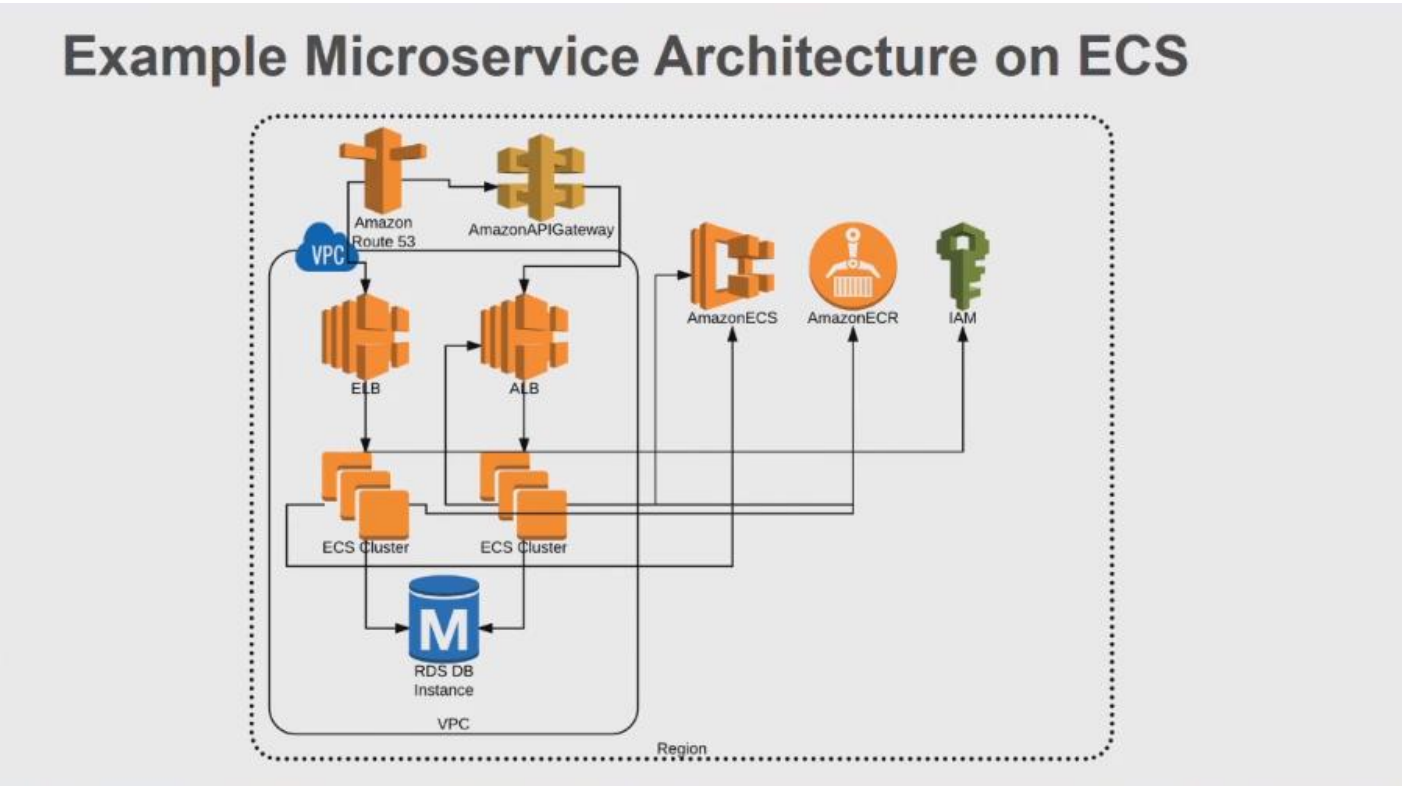


- Fully managed elastic service – You don't need to run anything, and the service scales as your microservices architecture grows
- Shared state optimistic scheduling
- Fully ACID compliant resource and state management
- Integration with CloudWatch service for monitoring and logging
- Integration with Code* services for continuous integration and delivery (CI/CD)

Amazon ECS Architecture



The customer owns and runs the VPC with a cluster that contains a set of EC2 instances running inside of it that share the name of the ECS cluster that they belong to. The container instances are really EC2 instances running an optimized AMI. On the instance you have the ECS agent running. On the ECS side we have a Scheduler and the Resource Manager running



Route53 can be used to provide the mappings between the ELB cnames and the friendlier domain names, you can use API Gateway in front of your services living in your ECS cluster that can also be talking to your databases. The managed services include ECS, ECR, and IAM to provide IAM roles to the services and tasks level in addition to the host level.

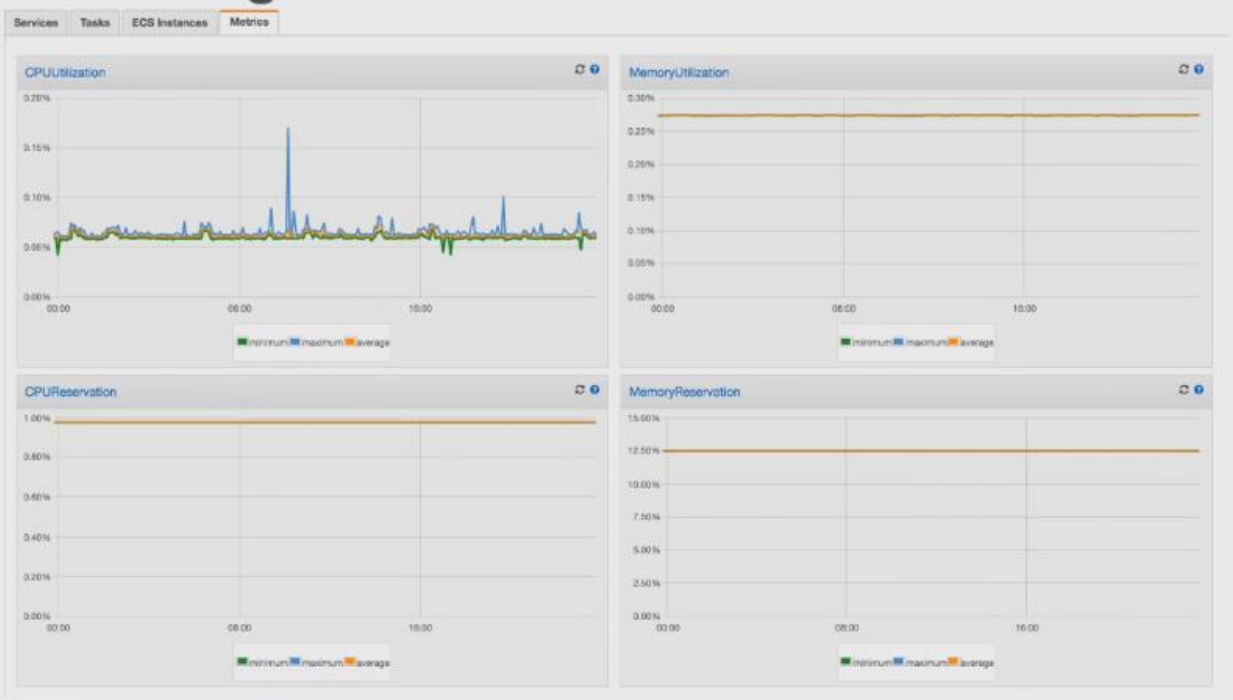
Monitoring with Amazon CloudWatch

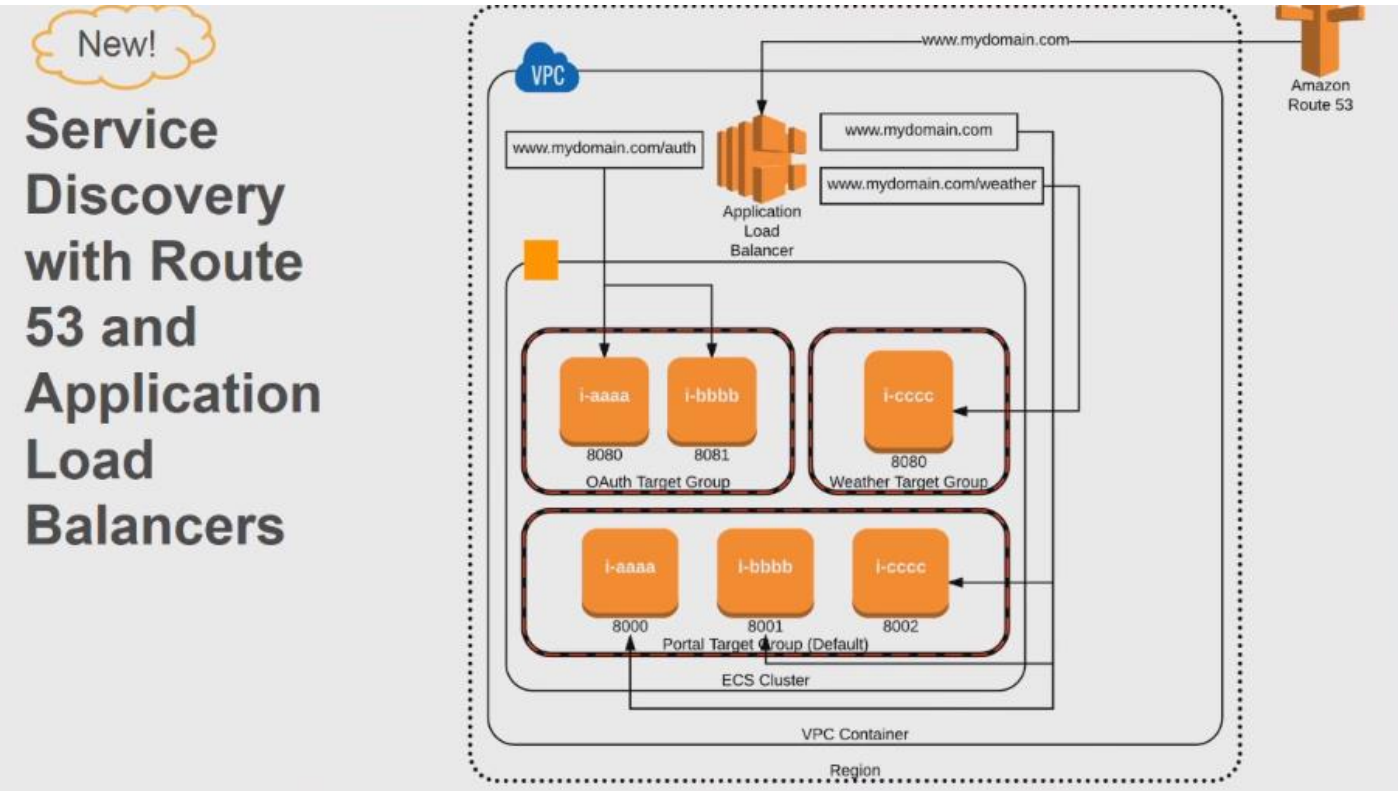
Metric data sent to CloudWatch in 1-minute periods and recorded for a period of two weeks

Available metrics: `CPUReservation`, `MemoryReservation`, `CPUUtilization`, `MemoryUtilization`

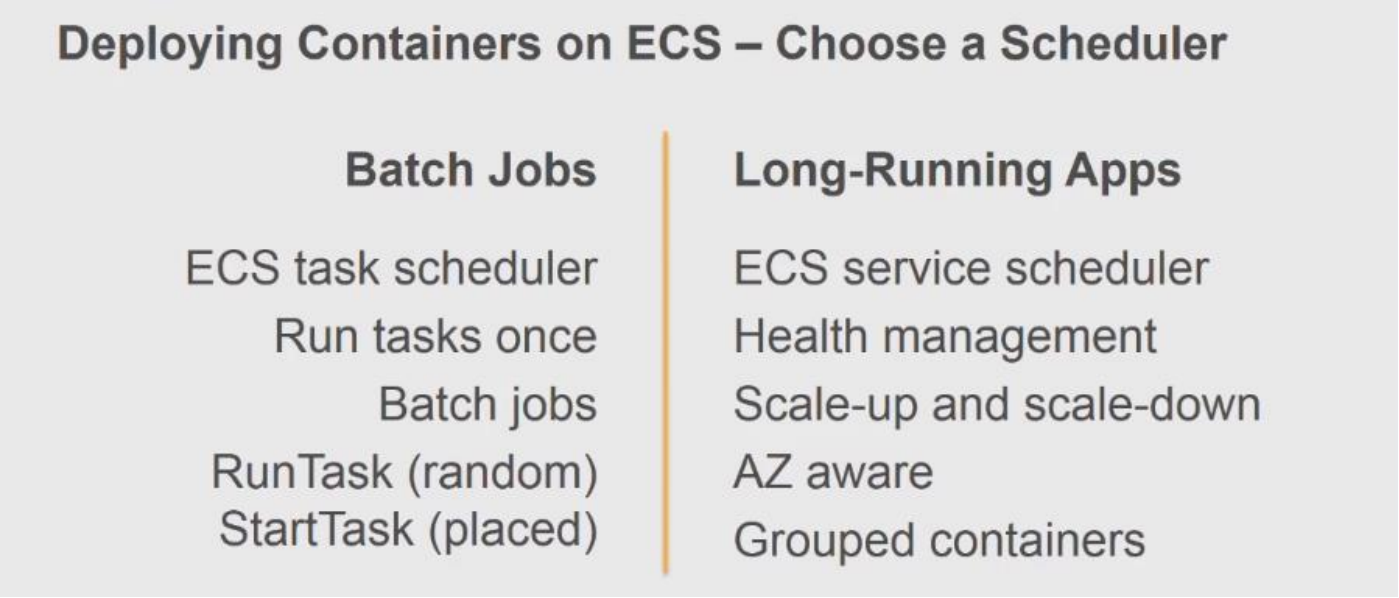
Available dimensions: `ClusterName`, `ServiceName`

Monitoring with Amazon CloudWatch





This architecture shows how the ALB service offers some interesting solutions for service discovery. Basically, ALB can route requests to different target groups. The target groups are hosts that are able to respond to requests of the same type and live behind a LB (layer 7). the ALB knows how to route based on the path information to the relevant hosts running behind it. This means that each service just needs to know the service name of any other service it needs to communicate with, build what the URL path information should look like to call, the ALB takes responsibility that the request gets to the hosts concerned.



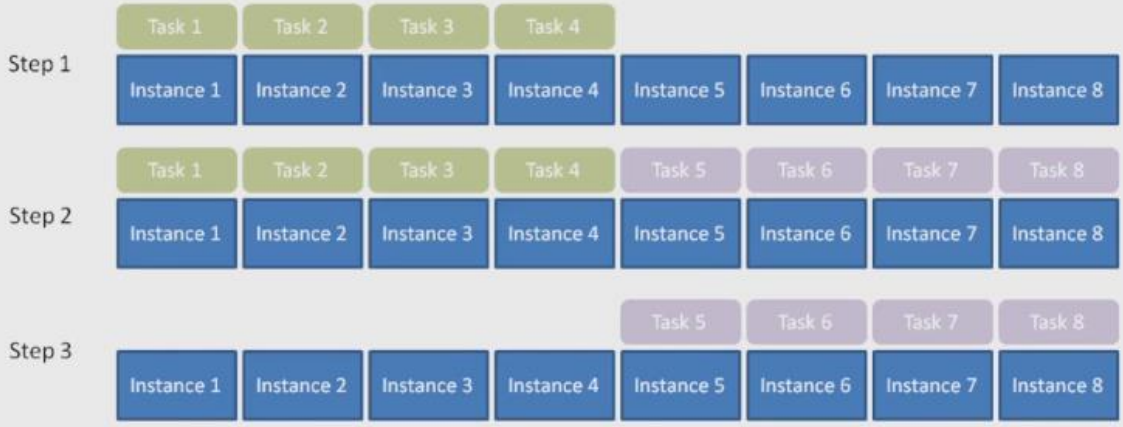
Long running things are like REST services, these will be deployed using the ECS service scheduler so that it can provide things like health management, scaling, etc.

Scheduling Containers: Long-Running App

Deploy using the least space: *minimumHealthyPercent* = 50%, *maximumPercent* = 100%

Scheduling Containers: Long-Running App

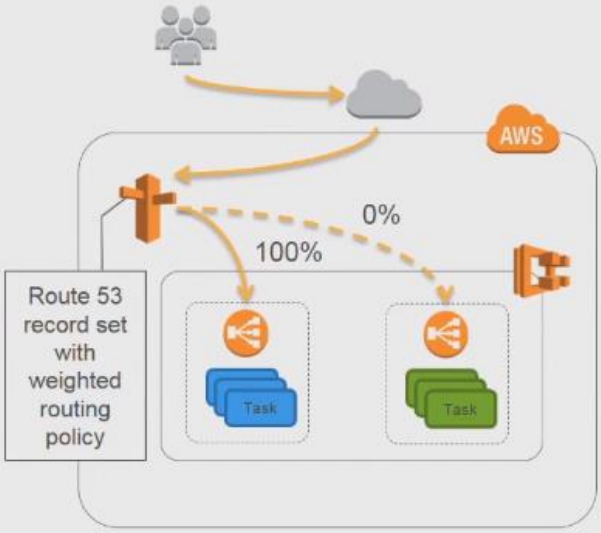
Deploy quickly without reducing service capacity: *minimumHealthyPercent* = 100%, *maximumPercent* = 200%



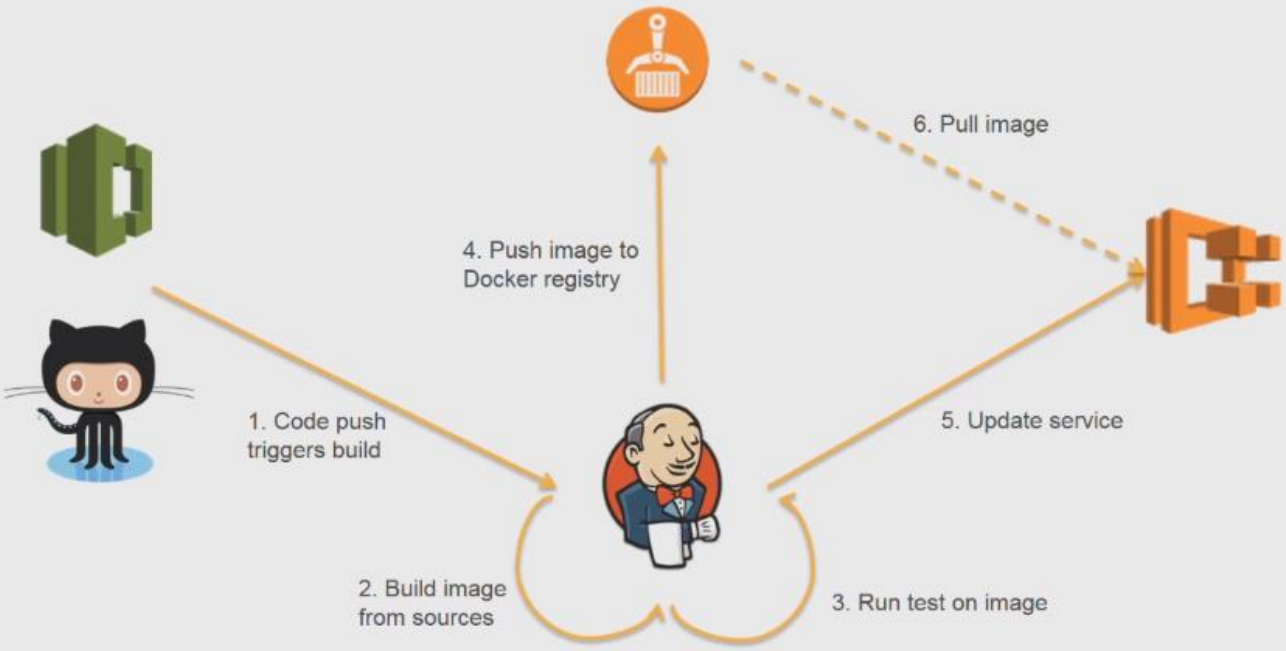
Scheduling Containers: Long-Running App

Blue-Green Deployments

- Define two ECS services
- Each service is associated w/ load balancer
- Both load balancers in Route 53 record set with weighted routing policy, 100% primary, 0% secondary
- Deploy to blue or green service and switch weights



Continuous Delivery to ECS with Jenkins



If you are already using Jenkins for non-containerized build process, for continuous integration you can simply include a Dockerfile at the root of your application folder to put your build artifact into a Docker container image that can be push to ECR. For the continuous delivery part, you can have a trigger that sits on top of the ECR image push event that will trigger a job to pull the container out and push it to the EC2 instances running in the cluster.

Continuous Delivery to ECS with Jenkins

Easy Deployment

Developers – Merge into master, done!

Jenkins Build Steps

Trigger via webhooks, monitoring, Lambda
Build Docker image via build and publish plugin
Push Docker image into registry
Register updated job with ECS API

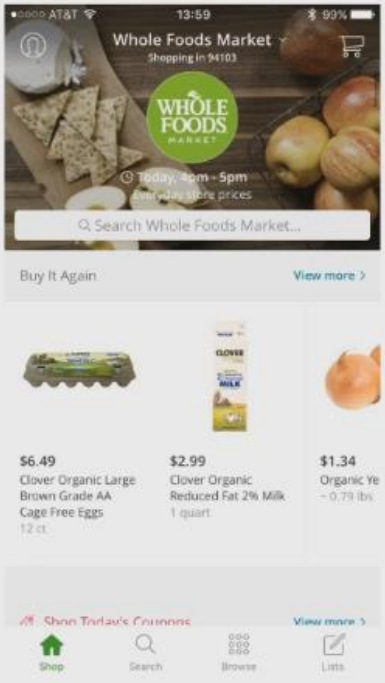
How Instacart Uses ECS

Nick Elser, Director of Engineering, Instacart

December 1, 2016

Instacart

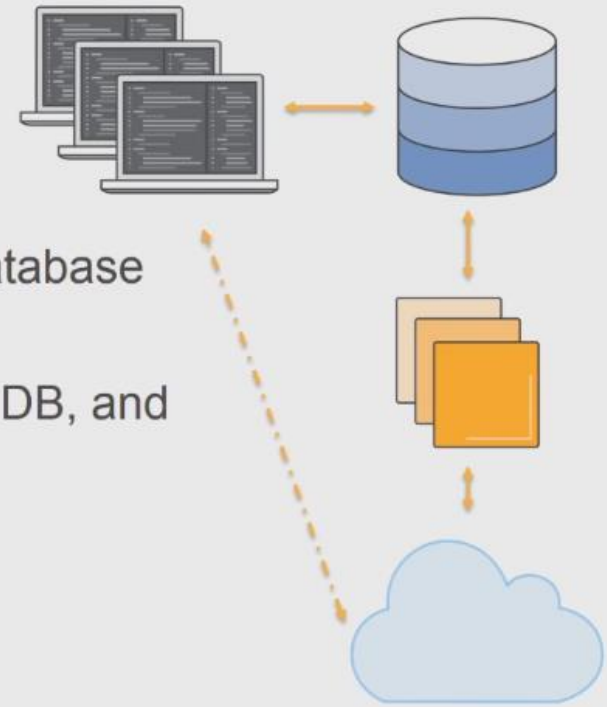
Started in 2012
Same-day delivery from local stores
Small engineering/infrastructure team
“Every minute counts”



The Stack

Primarily Ruby/Rails/JS, Python, & R
Few primary domains, each with own SLA
Domains comprised of many small-ish services
Monorepo
Custom PaaS

Instacart's PaaS



Coordinated through RDS database
Same tooling local & cloud
Jobs take desired state from DB, and turn it into cloud reality

The Instacart PaaS is a set of libraries and code written in Python that talk to the Amazon APIs and to the RDS and PostGres databases. We are storing all the changes that the developers have requested into the database whether that is a new deployment, a configuration change, etc. We then have jobs waiting in the cloud that take those requests and make them real by implementing the calls to the services.

Why not EC2?

Relatively slow
Deployments/rollbacks not atomic
Constrained to EC2 instance sizes

Previously, we were deploying to a fleet of EC2 instances over CodeDeploy to run our services. But this was limited for our services and we were under-utilizing our EC2 instances.

From EC2 to ECS

Transparent to developers: container vs. instance
Each container gets an IP address & DNS entry
SSH access, FS access

Deployment

Atomic through AWS CloudFormation
Rollbacks are instant (& atomic)
Custom “GC” policies per application type
Applications sized ”correctly”
Continuous delivery possible

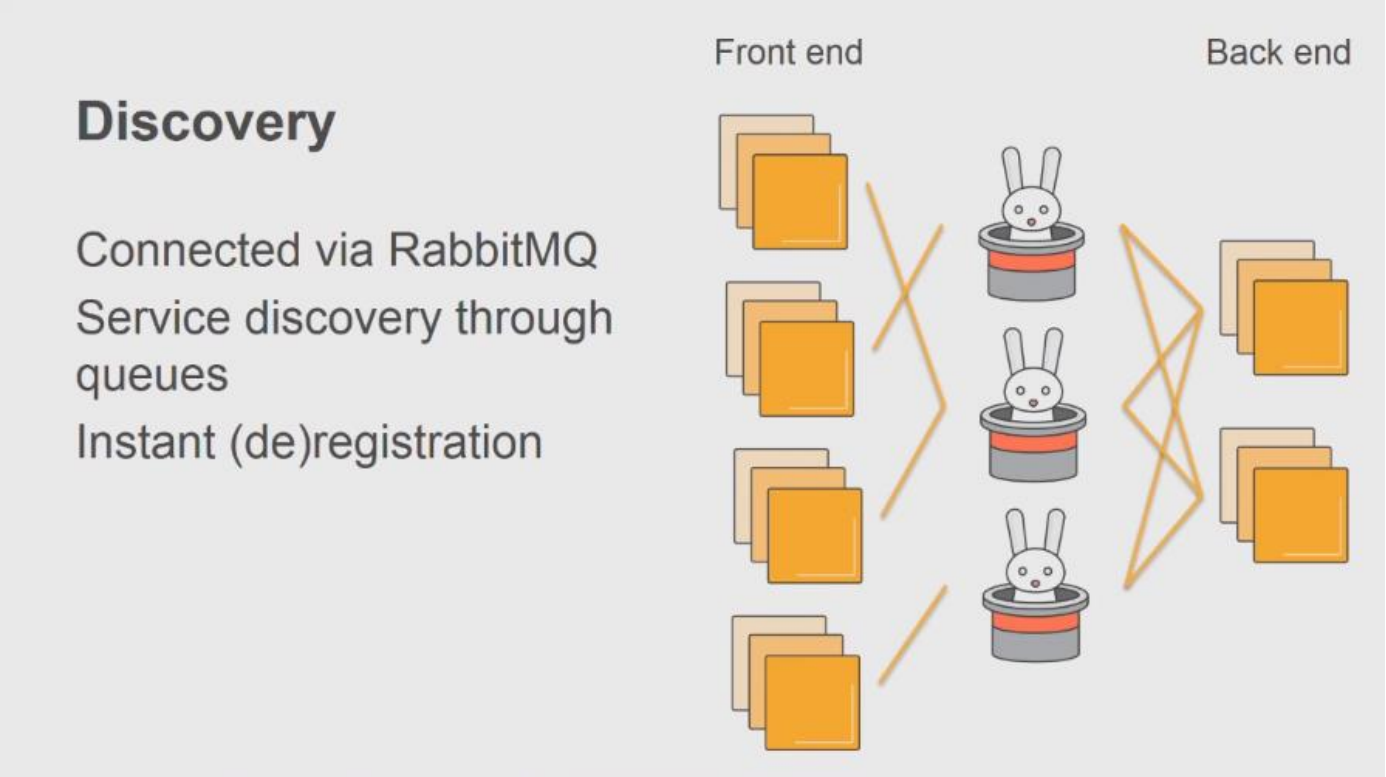
Deployment Pipelines

Webhooks on Jenkins servers
Deploys pipelined to 10% of users, then to 100%
Trust your metrics: 5xx & latency are king

Our deployment pipeline consists of Git pushes that trigger a webhook on our Jenkins servers that then build the artifact and push the Docker image to ECR. The Jobs system then detects the new image push event to ECR and run integration tests on the image to ensure it passes, record all the results to our infrastructure database. We have workers that are looking at the database for the result of the integration tests and will deploy the image in a rollout phase to 10% of the instances and then to 100% of the instances if all the 5xx and latency thresholds are met, otherwise they will roll back the 10% deployments.

Blue/Green Deployments

ALB registration/deregistration for blue/green
Keep one deploy running for instant rollback
Separate ports for separate deploys



For service discovery, we have a set of independent RabbitMQ nodes that are separate, stateless nodes running on EC2 directly and not clustered. Any frontend service connects to any one of the rabbits at random, while any backend service responsible to requests from the frontends connects to every single one of the rabbits. We do service discovery using the queues, any of the backends can pull a request out of a rabbit queue and respond to it.