McDonald's is the world's largest restaurant company with 37,000 locations serving 64 million people per day. Using AWS, McDonalds built ***Home Delivery—a platform that integrates local restaurants with delivery partners such as UberEats***.

McDonald's built and launched the Home Delivery platform in less than four months using a microservices architecture running on Amazon Elastic Container Service, ***Amazon Elastic Container Registry***, ***Application Load Balancer***, ***Amazon Elasticache***, ***Amazon SQS***, ***Amazon RDS***, and ***Amazon S3***.

The ***cloud-native microservices architecture allows the platform to scale to 20,000 orders per second*** with less than 100-millisecond latency, and open APIs allow McDonald's to easily integrate with multiple global delivery partners. Using AWS also means the system provides McDonald's with a return on its investment, even for its average $2–5 order value.

Contents

- About **McDonald's** and **Digital Acceleration**
- McDonald's **Home Delivery Platform**: Business Problem and Architecture
- Under the Covers: How we used ECS for **Scale, Speed, Security, DevOps and Monitoring**
- Final Thoughts and **Takeaways**

AWS re:Invent

aws

About McDonald's and Digital Acceleration

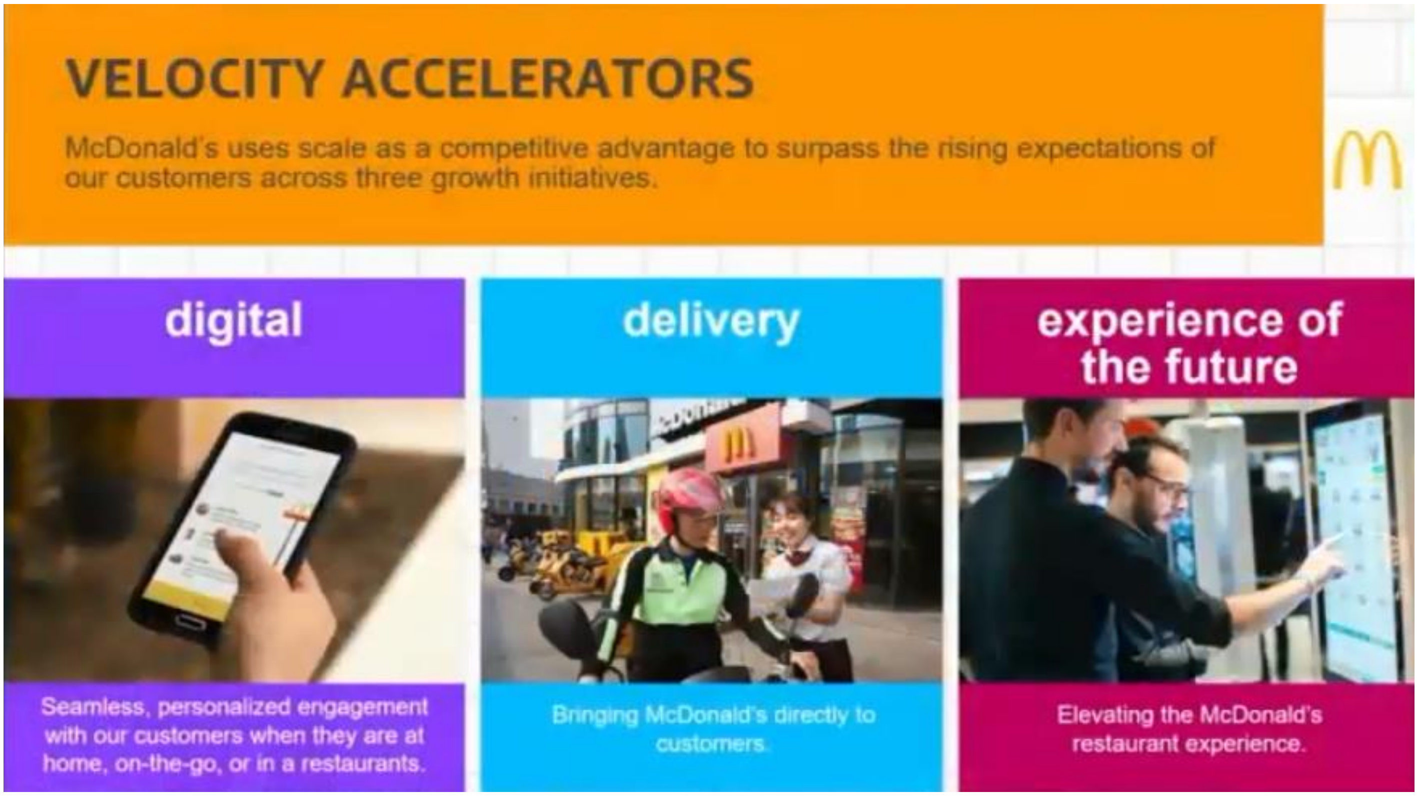
WORLD'S LARGEST RESTAURANT COMPANY

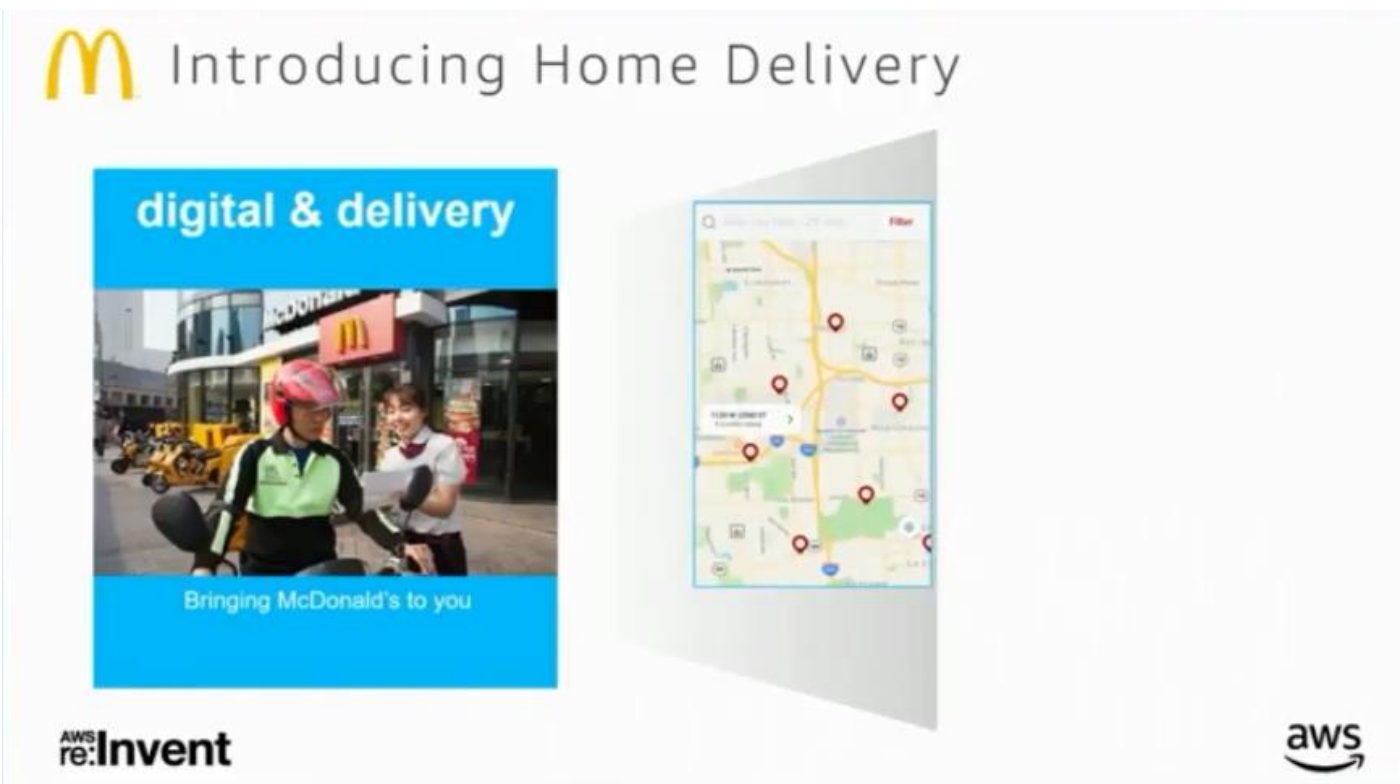
37K RESTAURANTS

1.9M PEOPLE Working for McDonald's & Franchisees

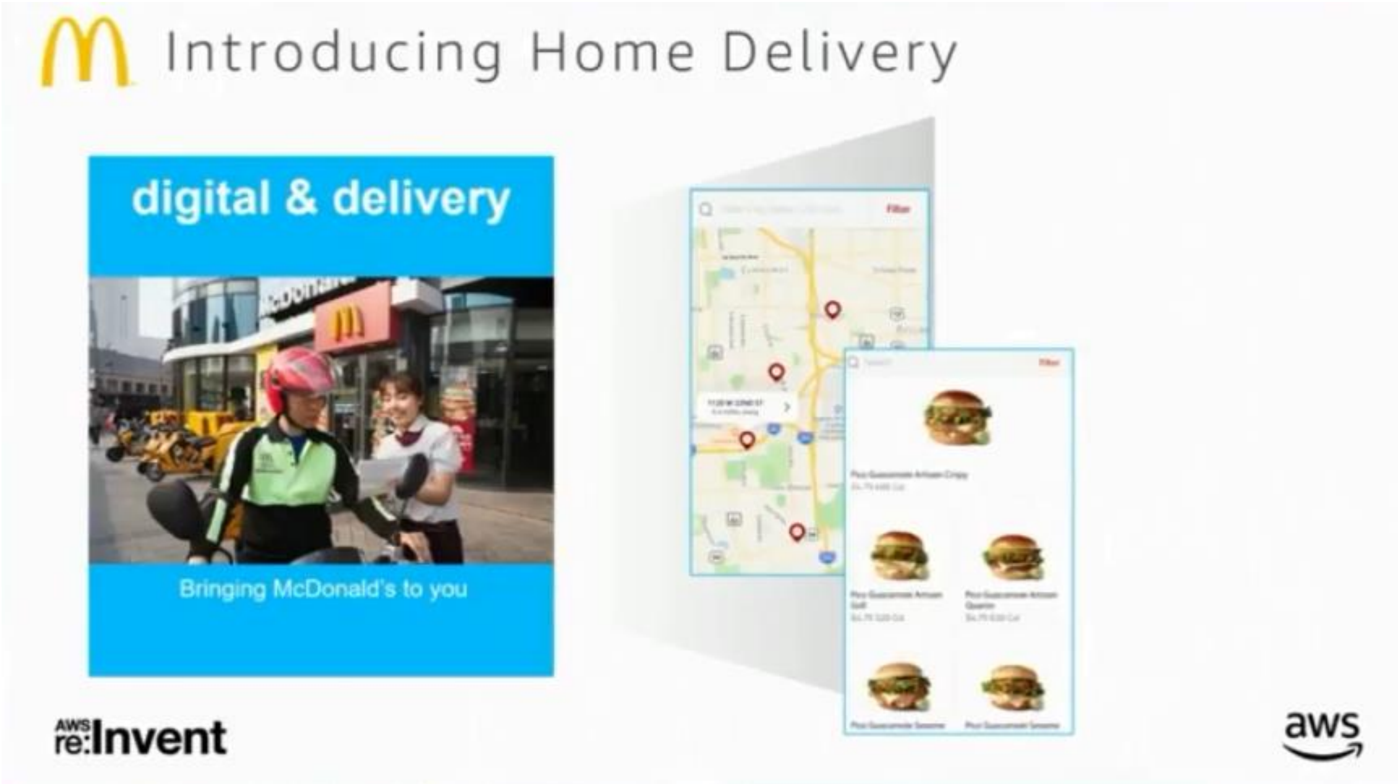
120 COUNTRIES

64M+ CUSTOMERS Served Every Day

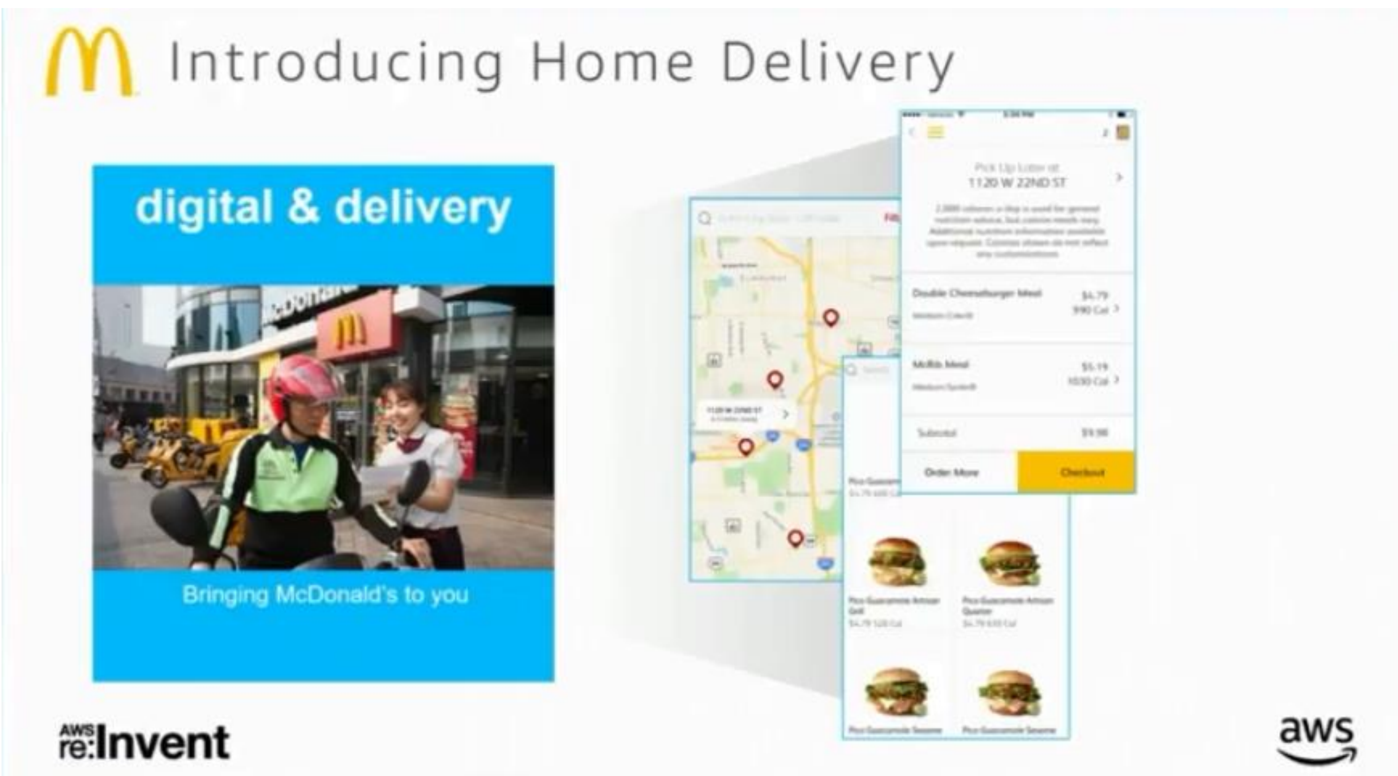




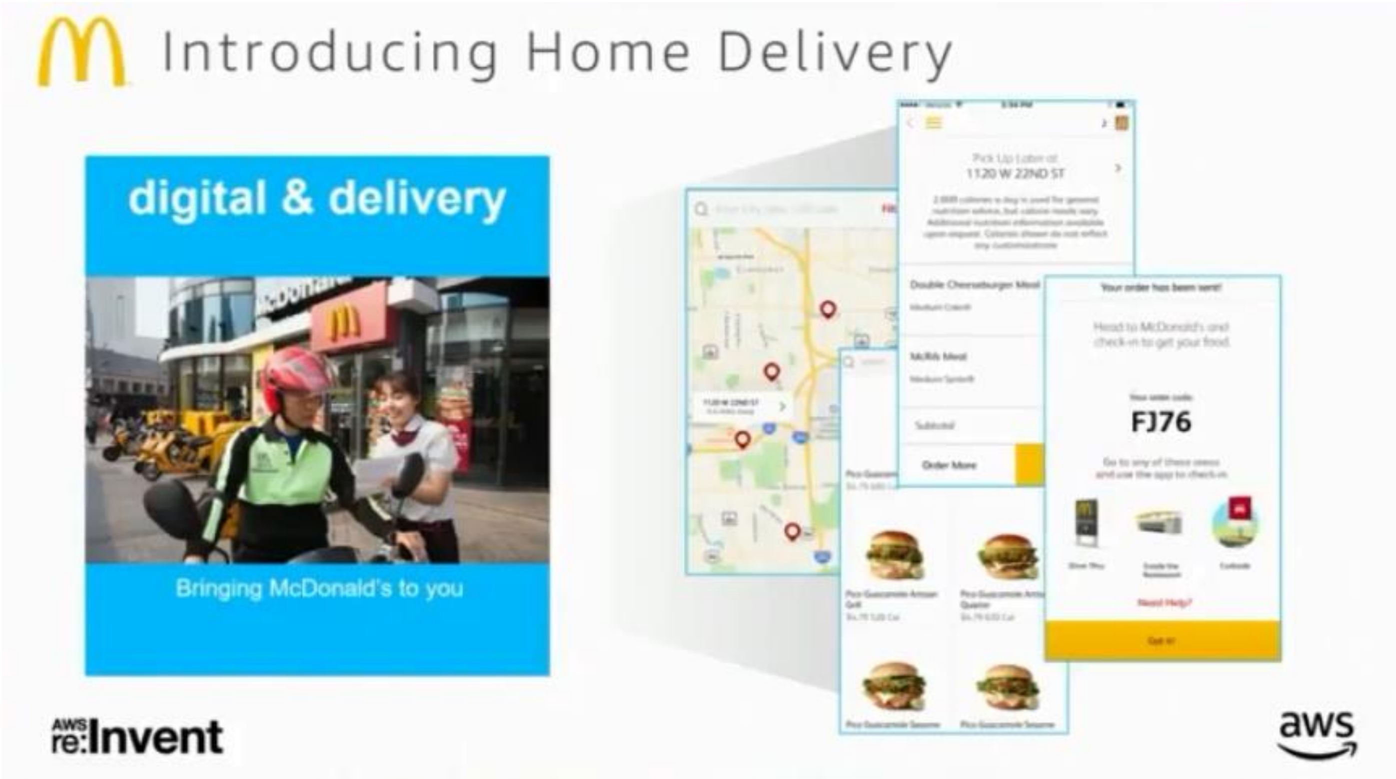
This journey starts when the customer picks a restaurant from a map on their phone to purchase their food from



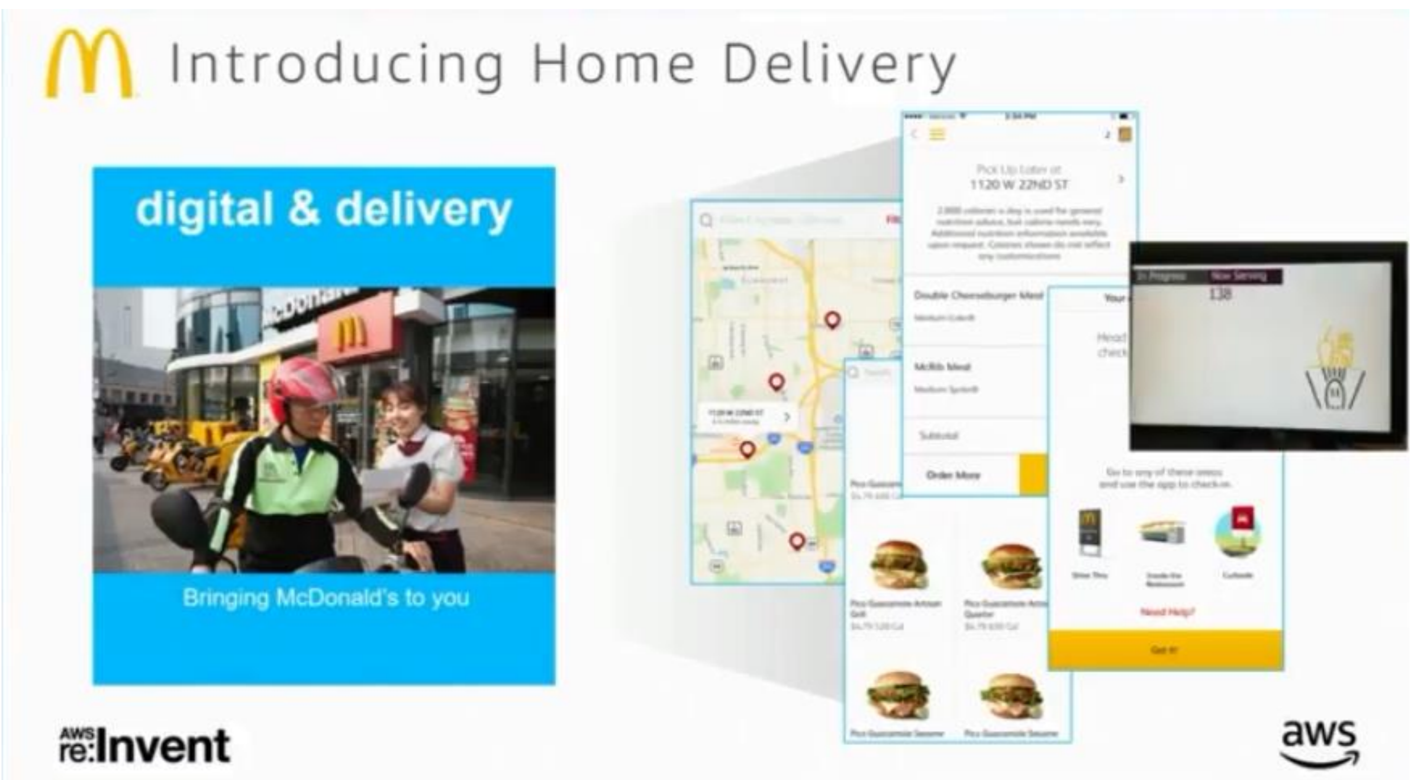
They then get shown the available menu at that particular restaurant



The customer then completes their order in their basket



At this point, order is complete and the delivery process begins

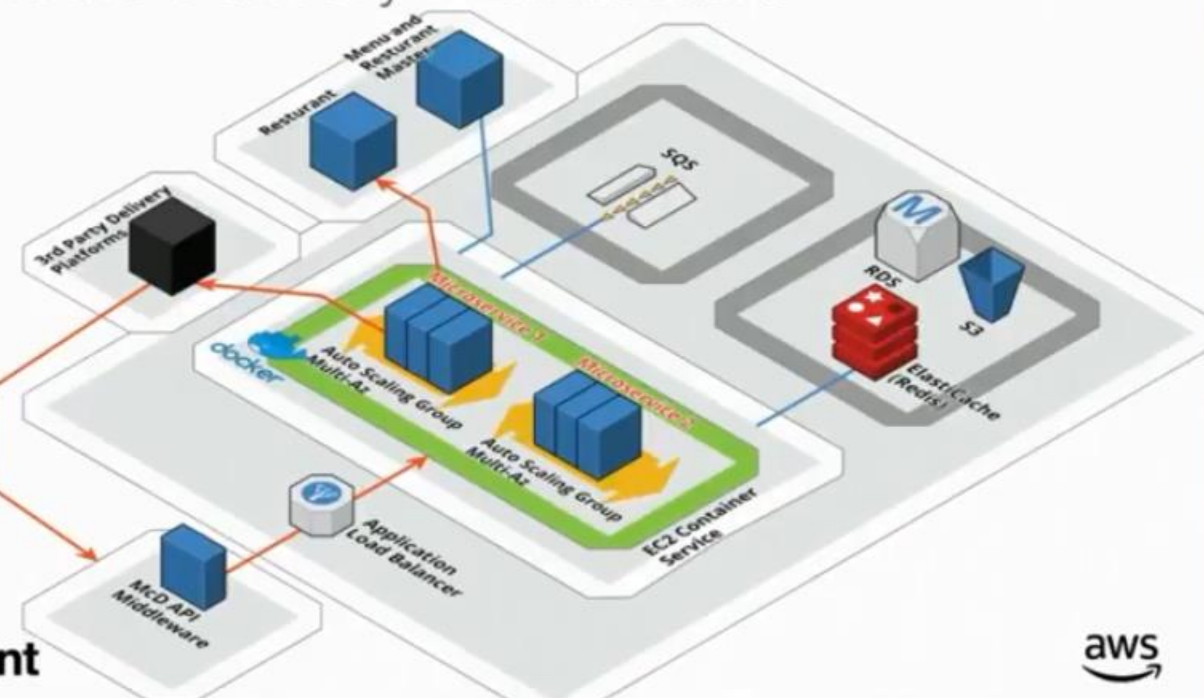


The delivery driver or rider close to the restaurant gets a ping with the details for delivery

Critical Business Requirements

- **Speed to market**, quick turn around for features and functionality from concept to production
- **Scalability and reliability** targets of, 250K-500K orders per hour
- **Multi country support** and integration with multiple 3rd party food delivery partners
- **Cost sensitivity**, cost model based on low average check amounts

Home Delivery Architecture



Key Architecture Principles

- **Microservices**, with clean APIs, service models, isolation, independent data models and deployability.
- **Containers and orchestration,** for handling massive scale, reliability and speed to market requirements
- **PaaS,** based architecture model by leveraging AWS platform components such as ECS, SQS, RDS and Elasticache
- **Synchronous and event based**, programming models based on requirements

Under the Covers: Using Amazon ECS for Scale, Speed, Security, DevOps and Monitoring

Why Amazon ECS?

- Speed to market
- Scalability and reliability
- Security
- DevOps – CI / CD
- Monitoring

Speed to Market

- **4 months** from concept to production with 2 week dev iterations
- **Polyglot tech stack** ported to containers
  - Existing .net code was refactored and complied with .net core
  - Java was used where .net core is not supported
- **Simplified Amazon ECS deployment model** with easy integration to AWS services
- **Less time** was spend on **application tuning/testing** to achieve the scale and reliability requirements
- **DevOps** – Faster feedback loops on release iterations

Scalability and Reliability

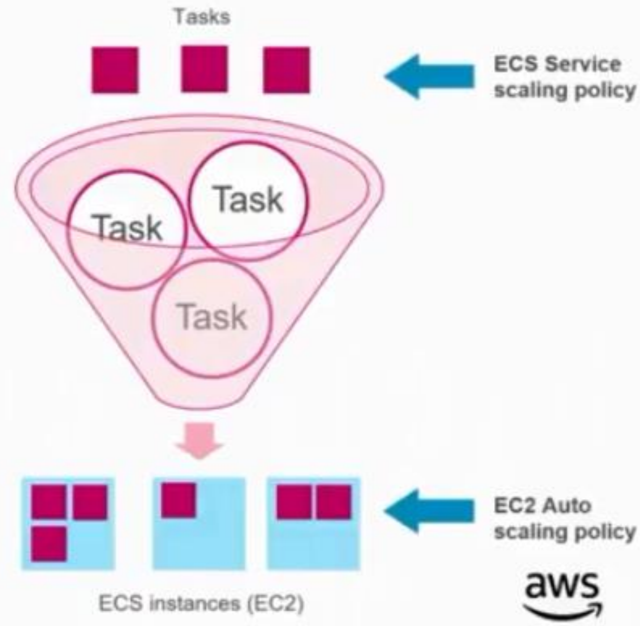
- **Scale Targets:** Achieved the scale targets of 250k-500k orders per hour with below ~100ms response times
- **Autoscaling:** Used Amazon ECS "out-of-box" resource based autoscaling
- **Task Placement:** Task placement strategies and constrains were used to fine-tune and achieve container isolation with country / 3rd party scaling requirements
- **Scalability and Reliability Requirements (By Service):**
  1. {Service 1} Contains synchronous APIs with intense scalability and reliability requirements based on traffic burst patterns
  2. {Service 2} Requires more batch mode processing. Work load optimization is a critical requirement
  3. Some instances of {Service 3} will need isolation from others

Scalability and Reliability

```
{Service 1} Task Definition
"placementStrategy": [
    {
        "field": "attribute:ecs.availability-zone",
        "type": "spread"
    }

{Service 2} Task Definition
"placementStrategy": [
    {
        "field": "memory",
        "type": "binpack"
    }
]

{Service 3} Task Definition
"placementConstraints": [
    {
        "expression": "task:group == US",
        "type": "memberOf"
    }
]
```



AWS re:Invent

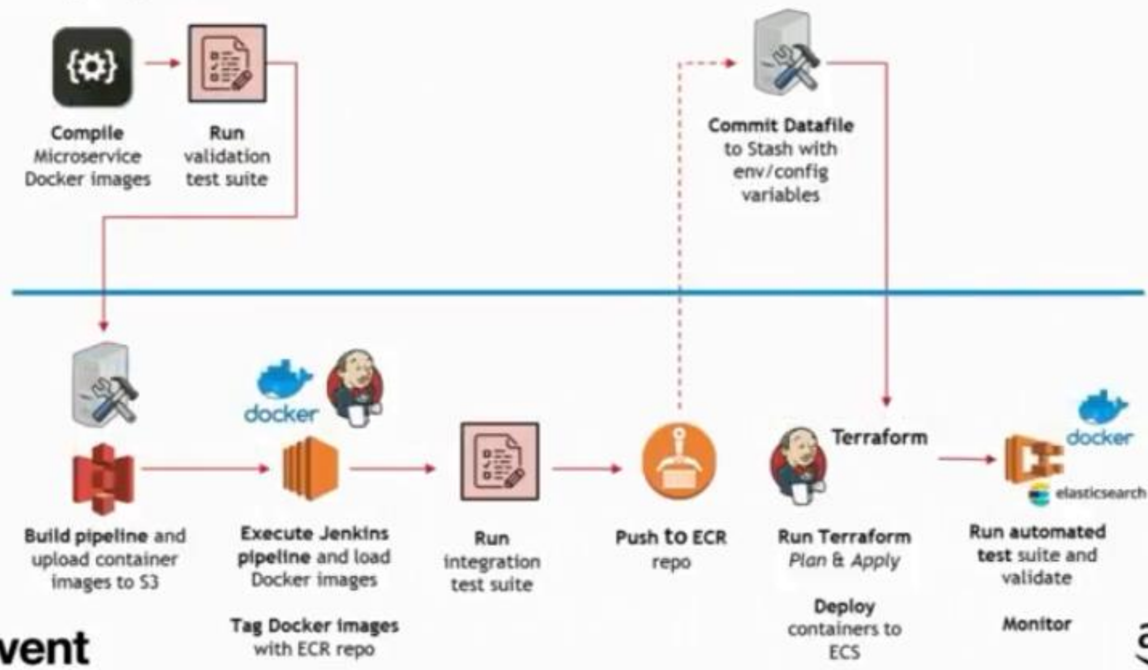
aws

Security

- **Container security**
  - Container isolation through IAM policies and security groups
  - Reduced container-to-container to communication (Reduce attack footprint).

- **ECS instance security**
  - Automated AMI factory → start with ECS optimized AMIs→ McD hardened gold AMIs→ Project specific AMIs
  - AMI factory pipeline listens to a SNS topic for obtaining updated AMIs

DevOps/CI&CD



Monitoring

- NewRelic agents configured to monitor the ECS instances, Containers and AWS PaaS components
- ELK stack configured for service logging and analytics

Major Technical Challenges

- "Out of memory error" due to containers not having access to *cgroup* file systems to get memory limits
    - **Solution**: Incorporated a new filesystem(lxcfs) to virtualized *cgroup* and virtualized views of **/proc** files

- Docker containers are not honoring the ECS instance routing rules
    - **Solution:** Custom implementation of a Docker bridge

Final Thoughts and Key Takeaways

- A thought out **microservice architecture** is key for **scalability, reliability, and containerization.**

- **Massive scale** achievable (**north of 20k TPS under 100ms**) in a controlled manner **using auto-scale policies and task placement strategies.**

- **Moving to containers** simplified our development and deployment models and in turn provided **quicker dev/test iterations.**

- ECS out of the box integration and deployment models further **simplified our DevOps pipeline.**