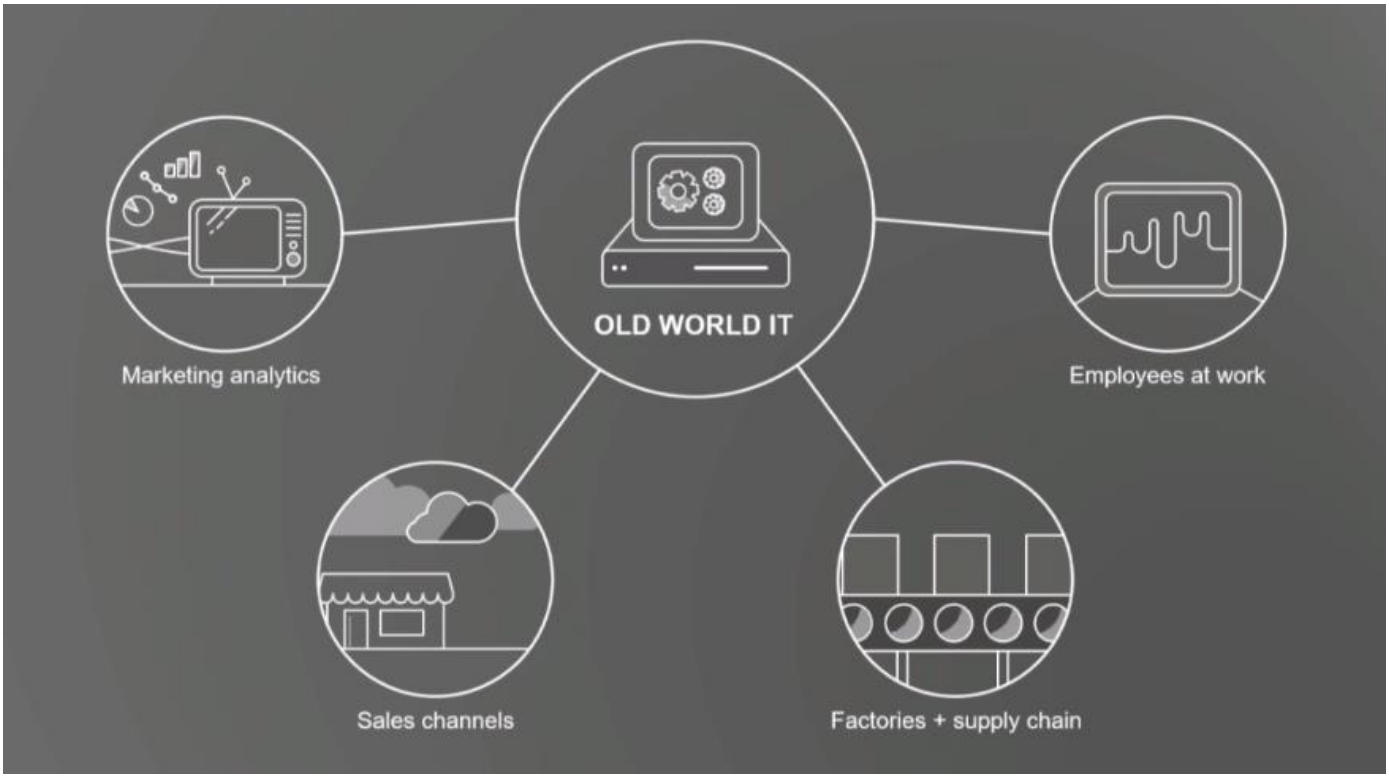
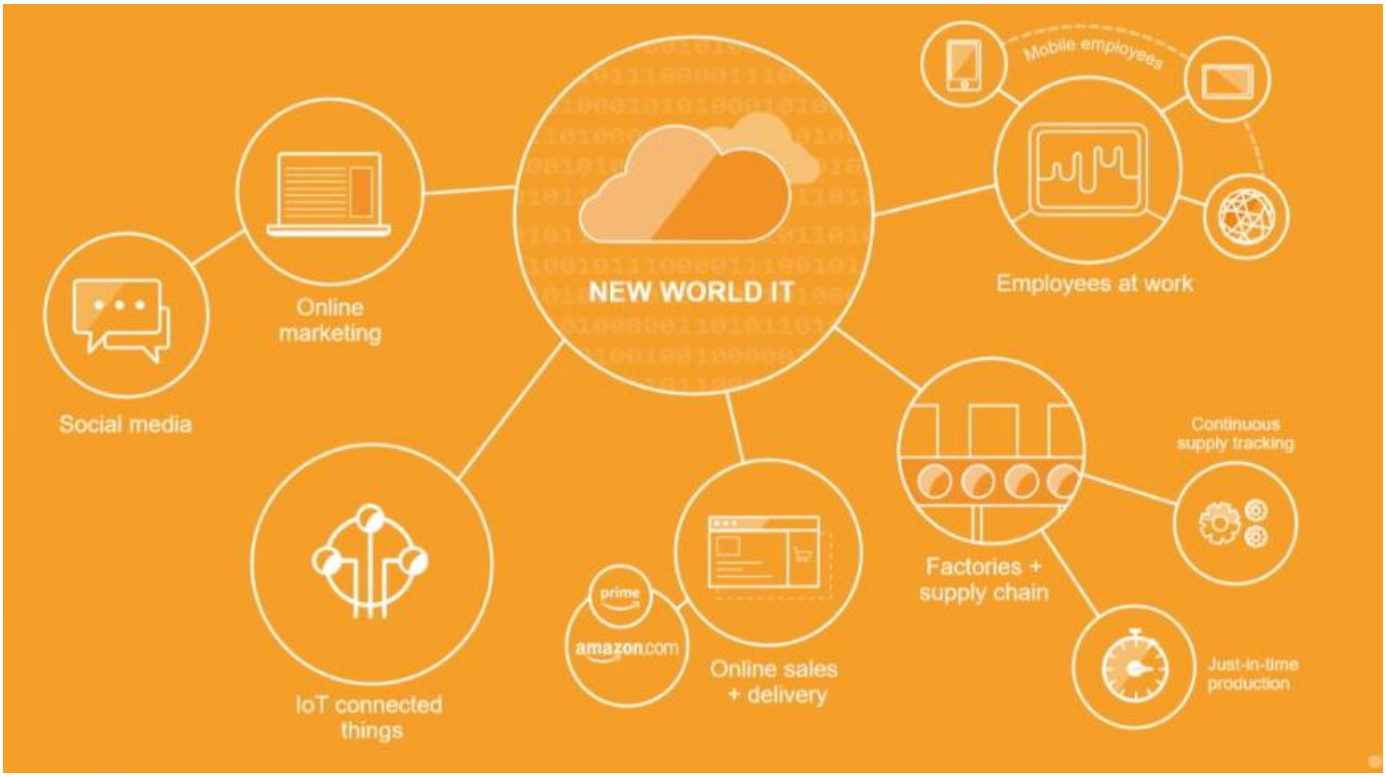
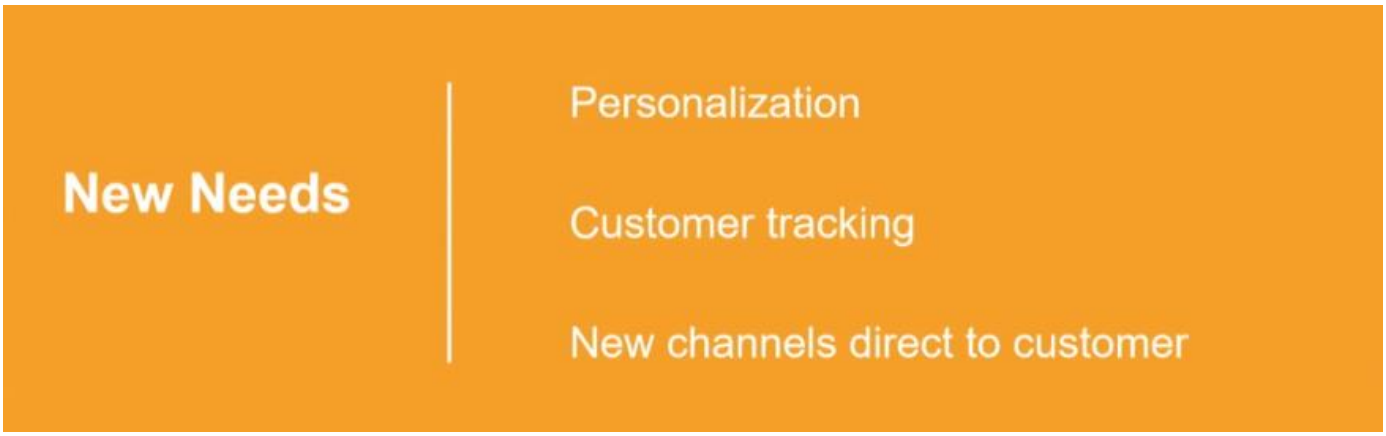
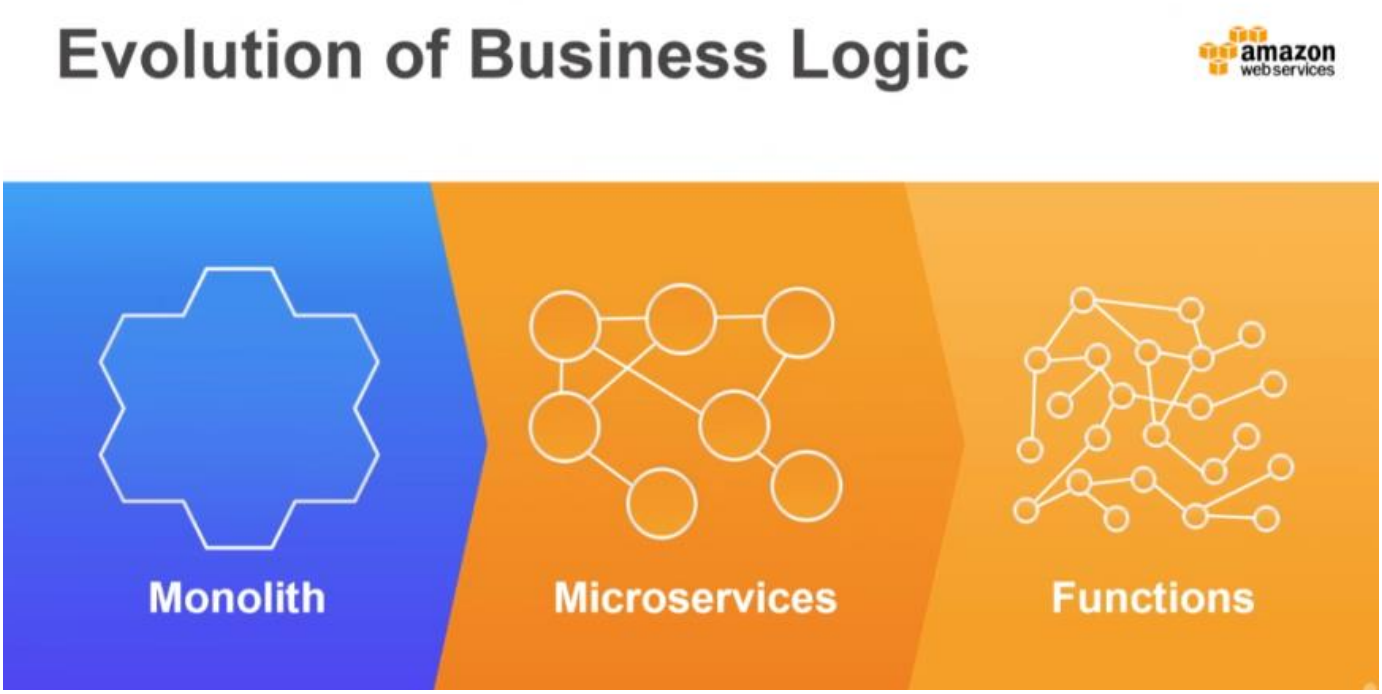
Many industries are going through a digital transformation as their existing business models are being disrupted and new competitors emerge. The key driver is a need for faster time-to-value as a direct relationship with customers provides analytics that drive personalization and rapid product development. There's a cultural aspect to the change, as well as new organizational patterns that go along with a migration to cloud native services. Application architectures are evolving from monoliths to microservices and serverless deployments, and they becoming more distributed, highly available, and resilient. The highly automated practices that have built up around DevOps are moving to the mainstream, and some new techniques are emerging around security red teams and chaos engineering.


OLD WORLD IT
Marketing analytics
Employees at work
Sales channels
Factories + supply chain


NEW WORLD IT
Mobile employees
Employees at work
Online marketing
Social media
Continuous supply tracking
Factories + supply chain
Just-in-time production
Online sales + delivery
IoT connected things



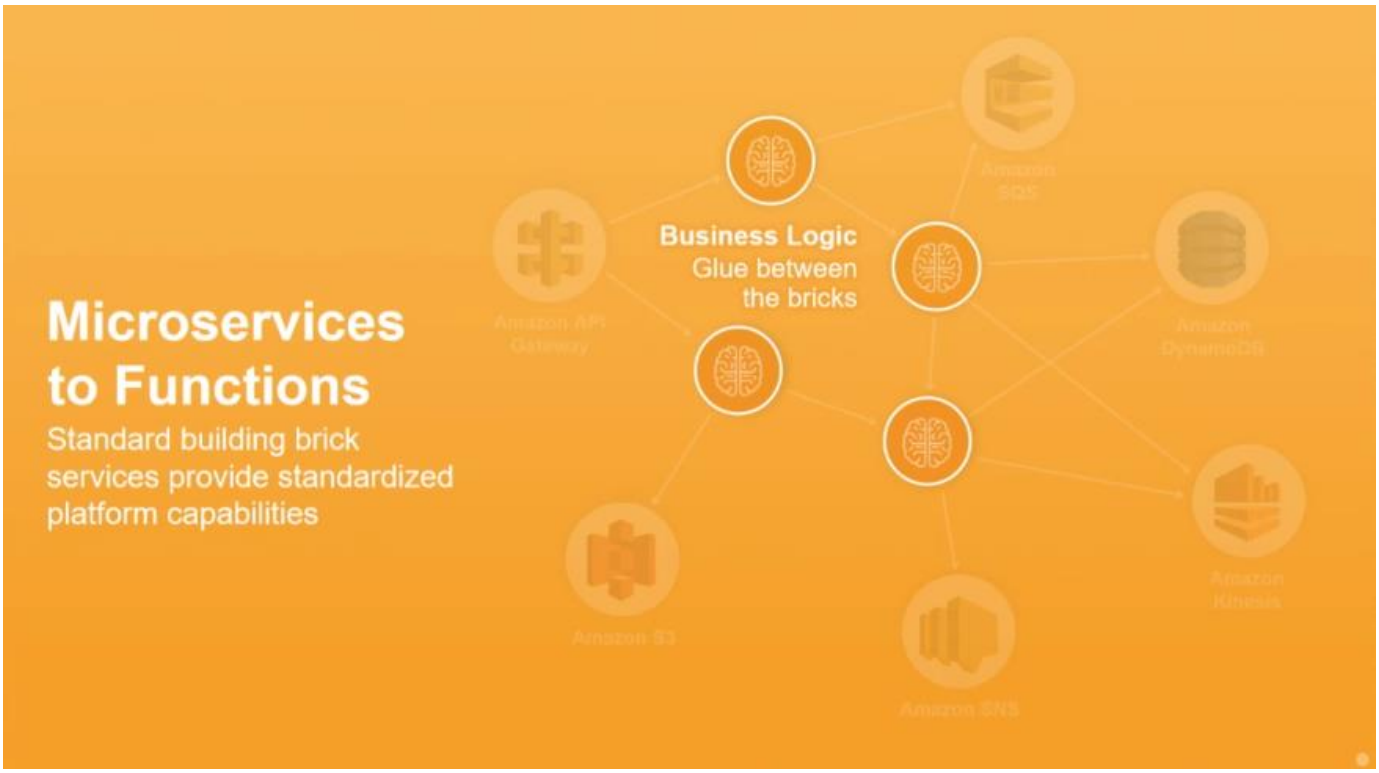
You need to build things that include the above, personalization because you can now know who is watching TV or seeing your ads or products in near real-time and you can now optimize their experiences.



Monoliths usually pass around big fat XML and SOAP chunks between their sub-components and this made them slow

This allows us to iterate and increase development time while working on the smaller services independently

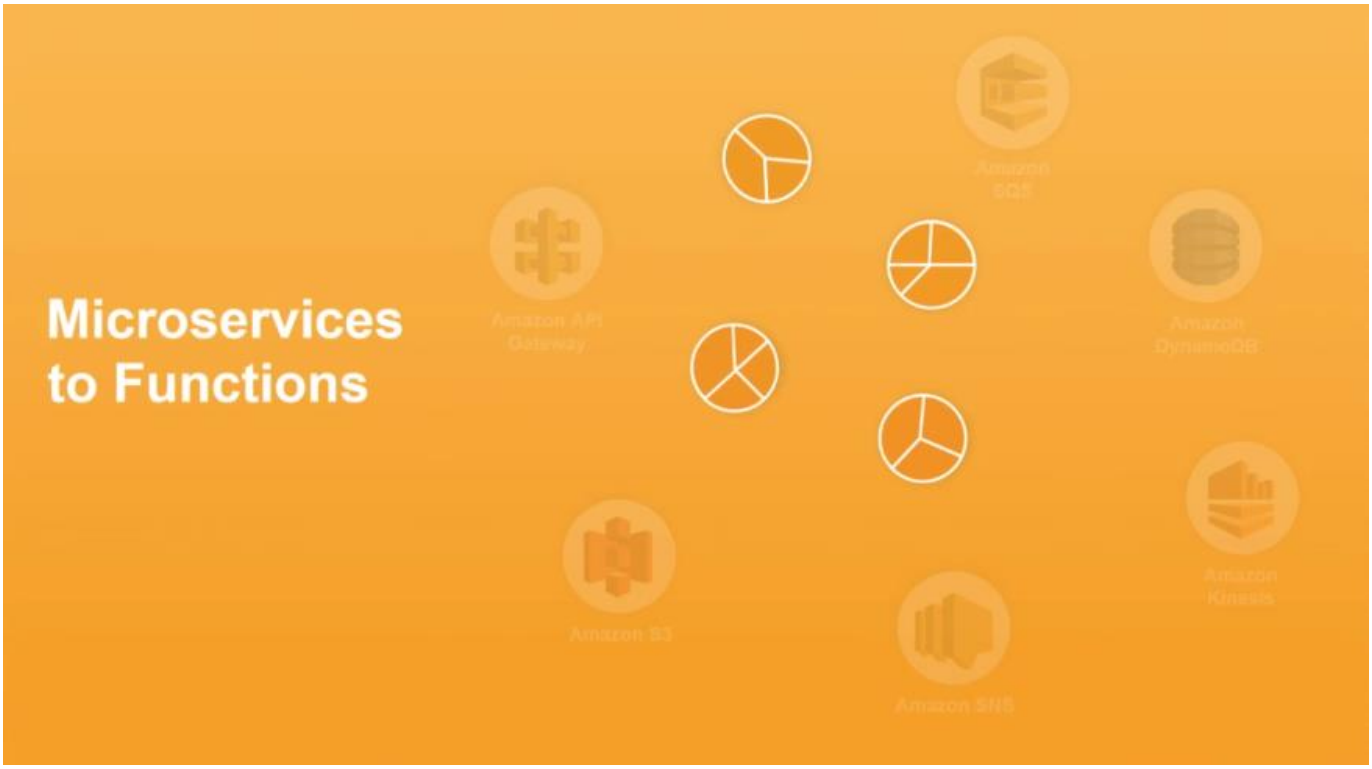
We can now start replacing a lot of the common services with better managed services like SQS, DynamoDB, Kinesis, etc.



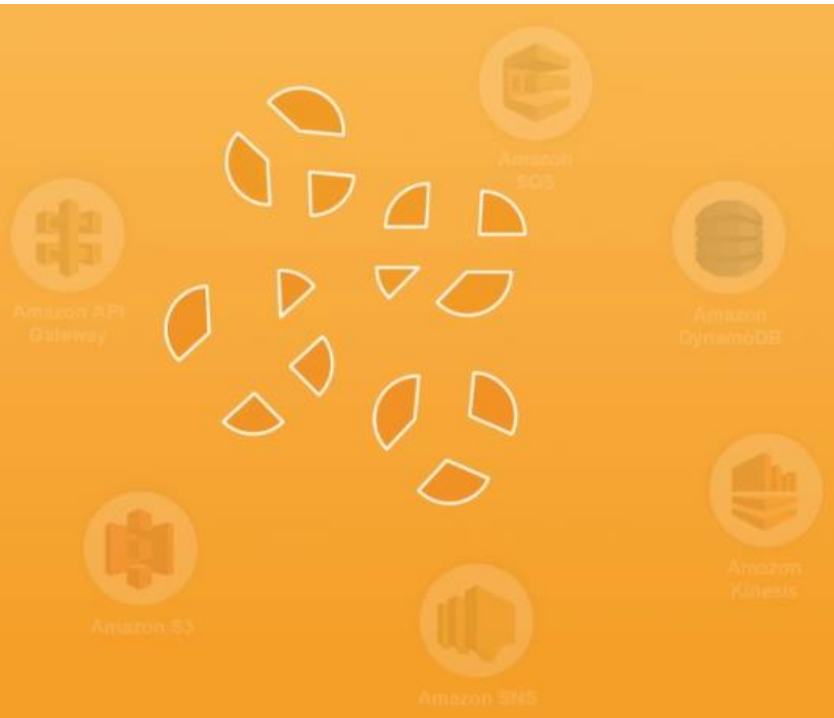
This are now our unique services that we now have to write.

Microservices to Functions

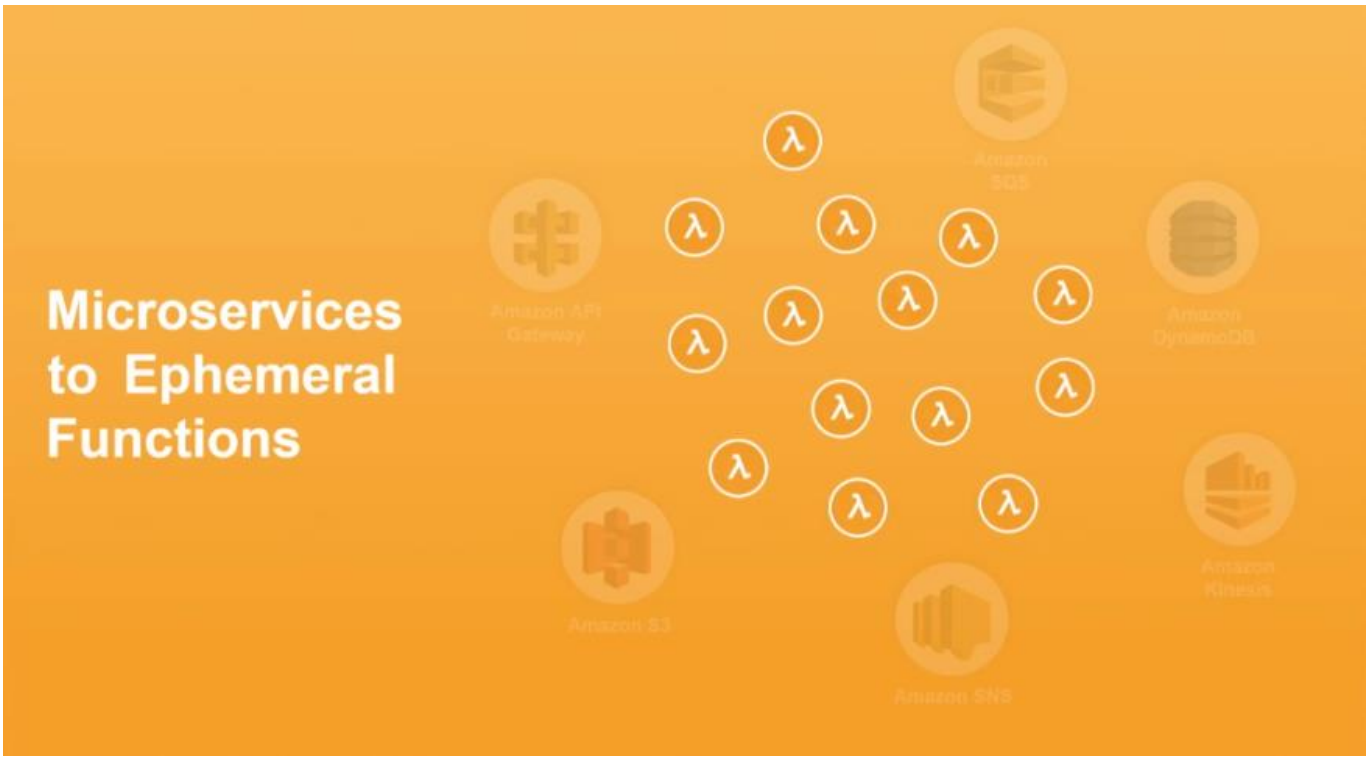
Microservices to Functions



Microservices to Functions

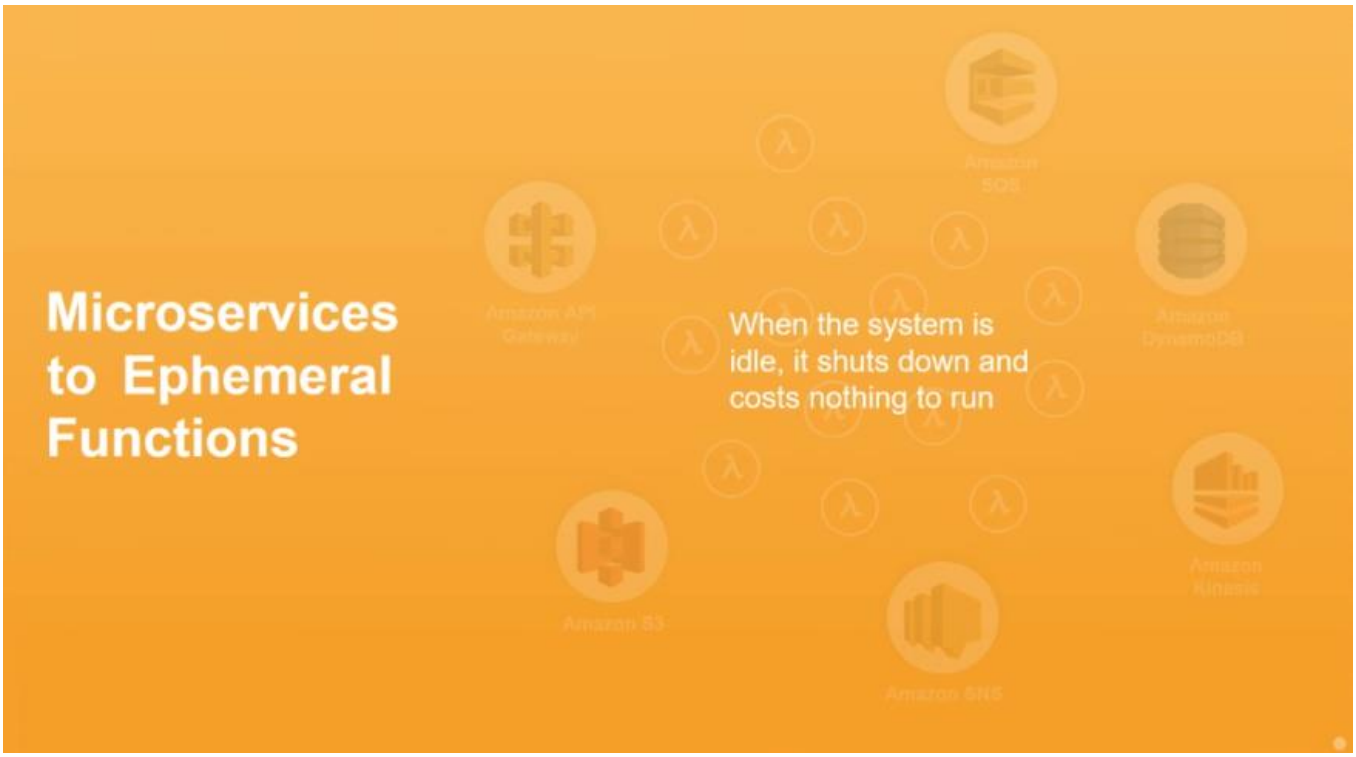


Microservices to Functions

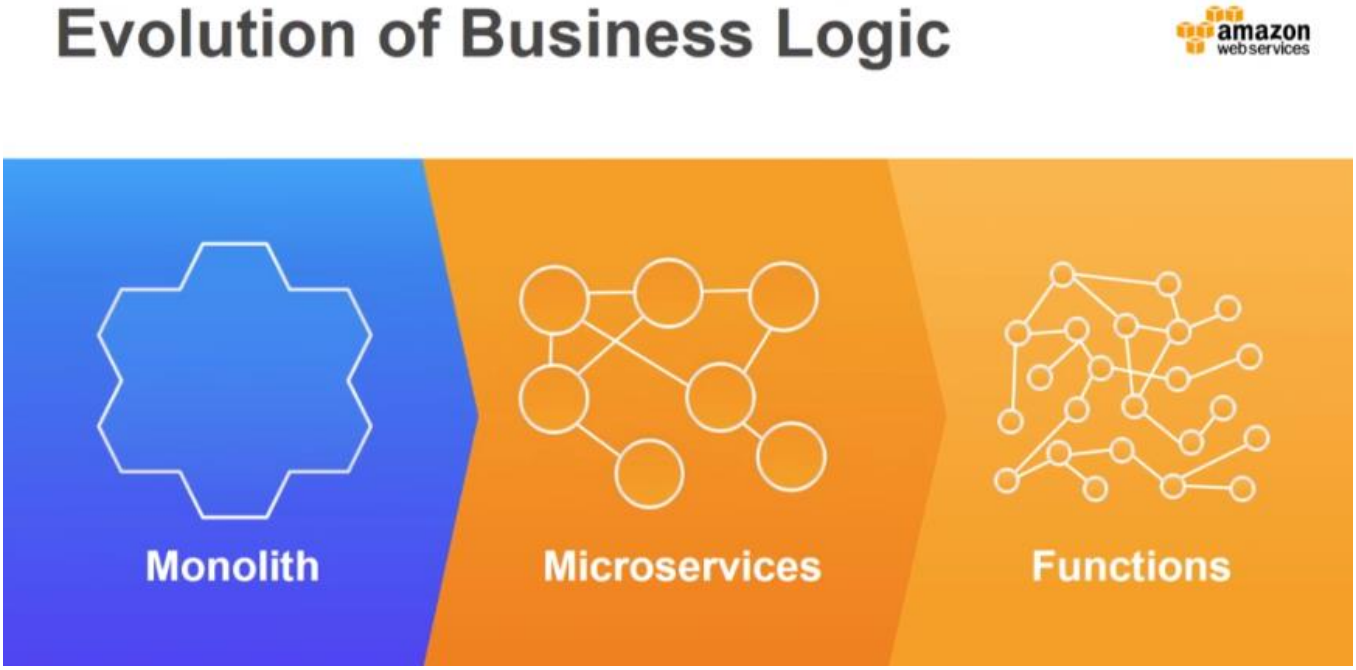


These lambda functions are now ephemeral and can sleep while not being used

When a request now comes in, it wakes up some lambda functions and services to do the particular work needed and then goes back to sleep



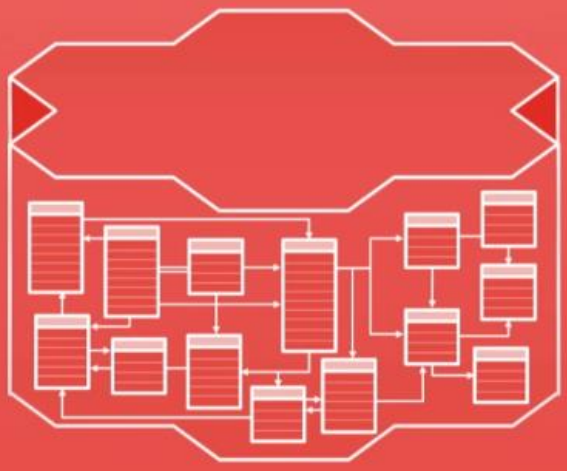
This is the concept of the Serverless evolution



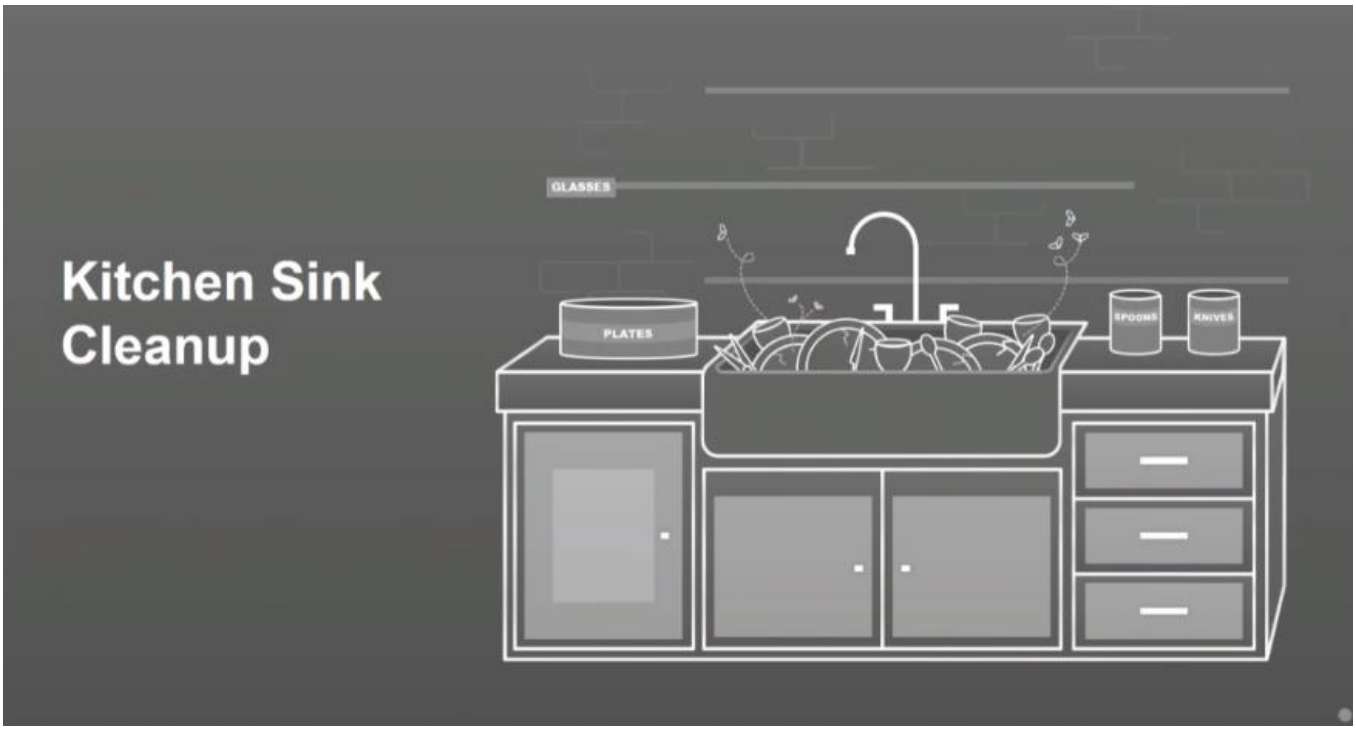
We are trying to get to a de-normalized model

Expensive,
Hard to Create
and Run

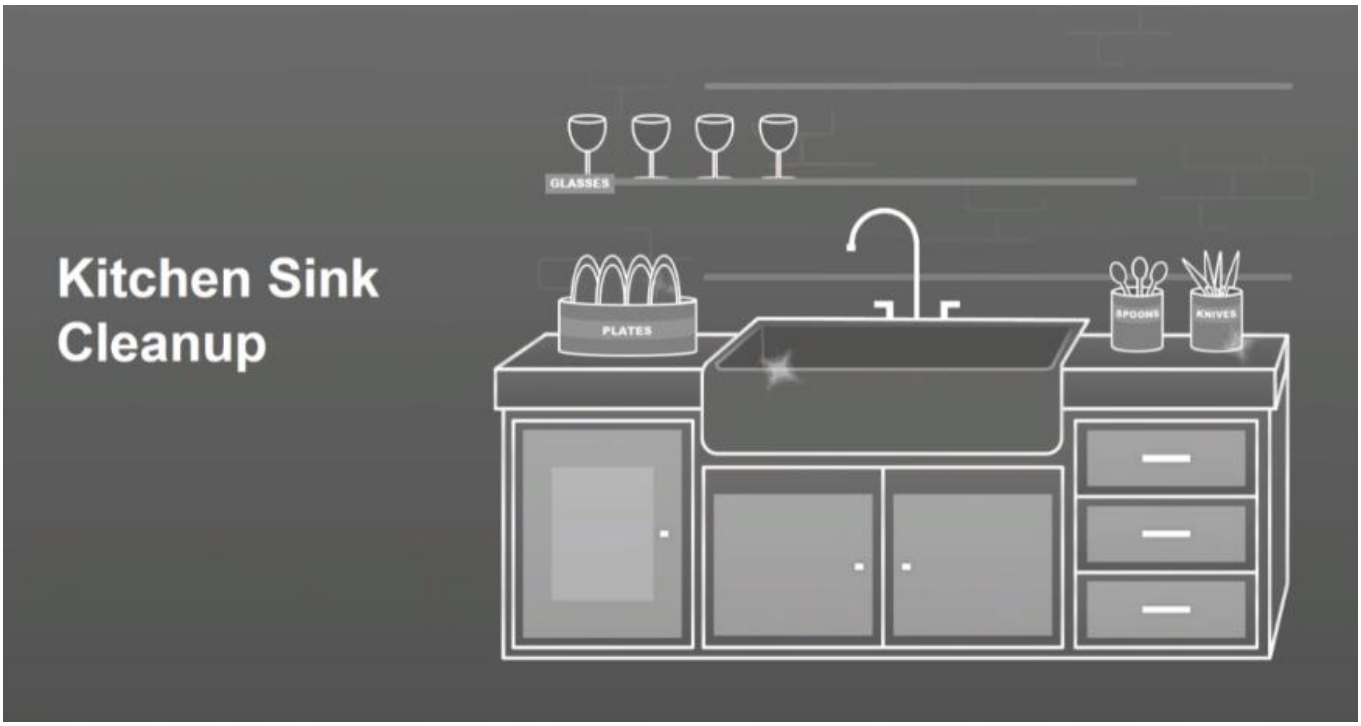
Database Schema
Entity Relationship



They usually have a very complicated schema structure and becomes very hard to untangle the database



You want to untangle the mess as below



But we are missing a spoon. It means we have to have a mechanism for keeping all our components in sync because microservices usually have their own individual databases and queues

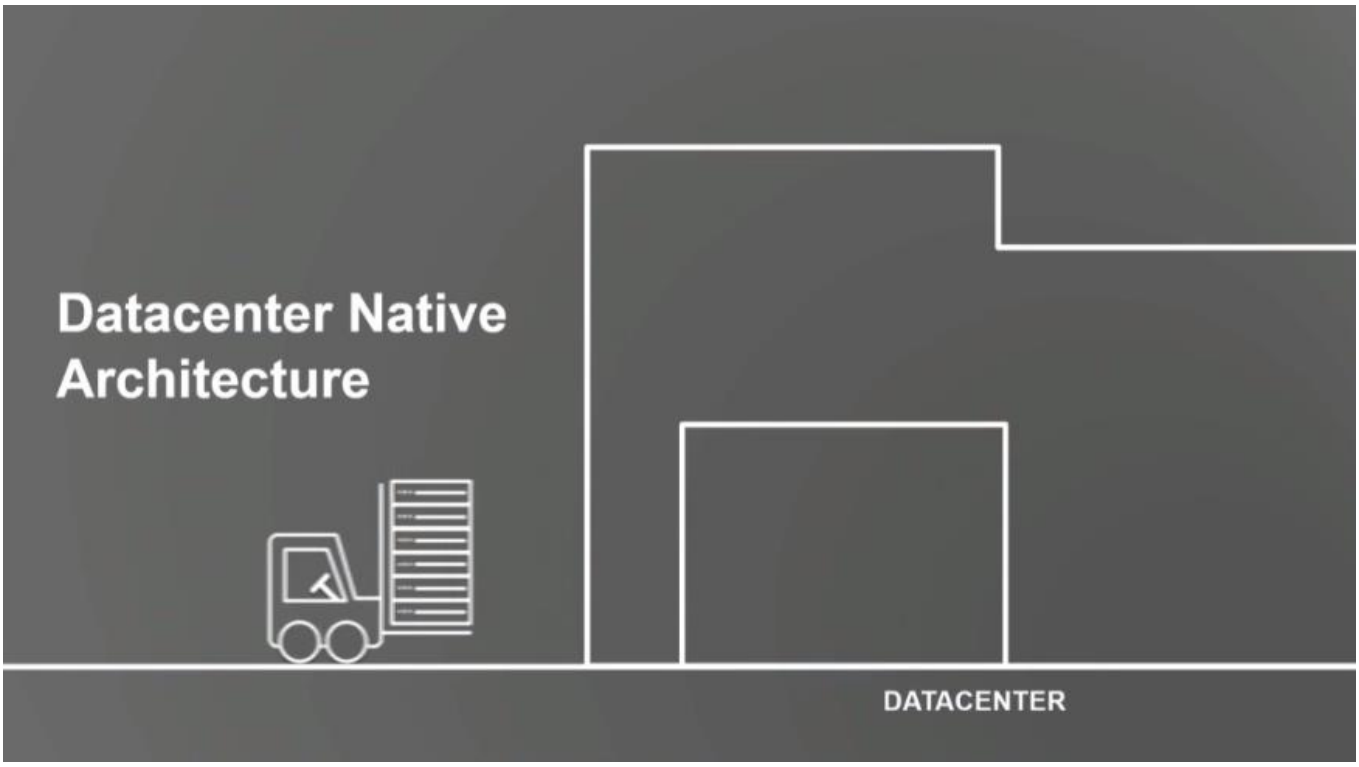
We are going to have to create a new place to put a new dataset instead of lumping it together with other data

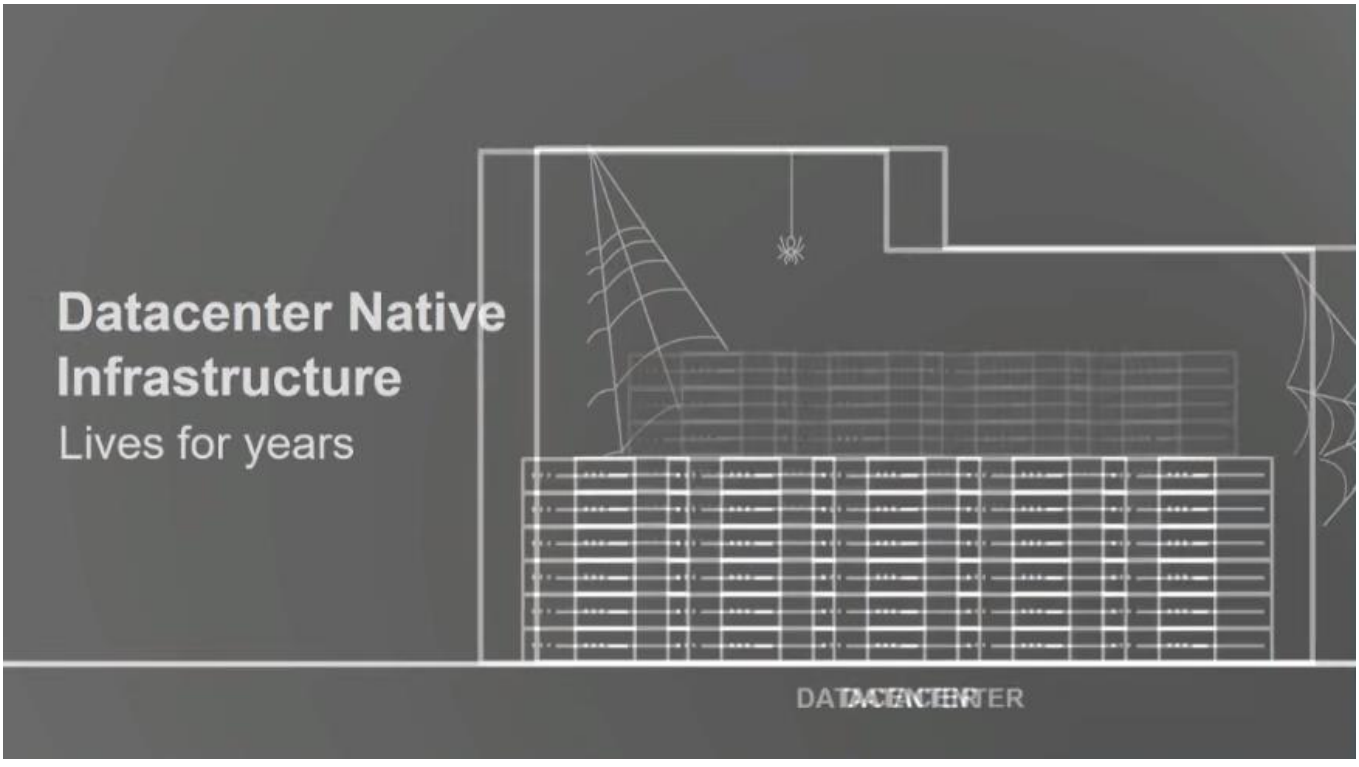
You can now simply create your new database for a new service easily using a managed database service like DynamoDB

You now also have much simpler schema and systems, you don't need to worry about transactions and joins anymore because your data is now in a single database.

The New De-Normal

Cloud Native Architecture


Datacenter Native
Architecture
DATACENTER


Datacenter Native
Infrastructure
Lives for years
DATACENTER

Cloud Migration
Pay as you go

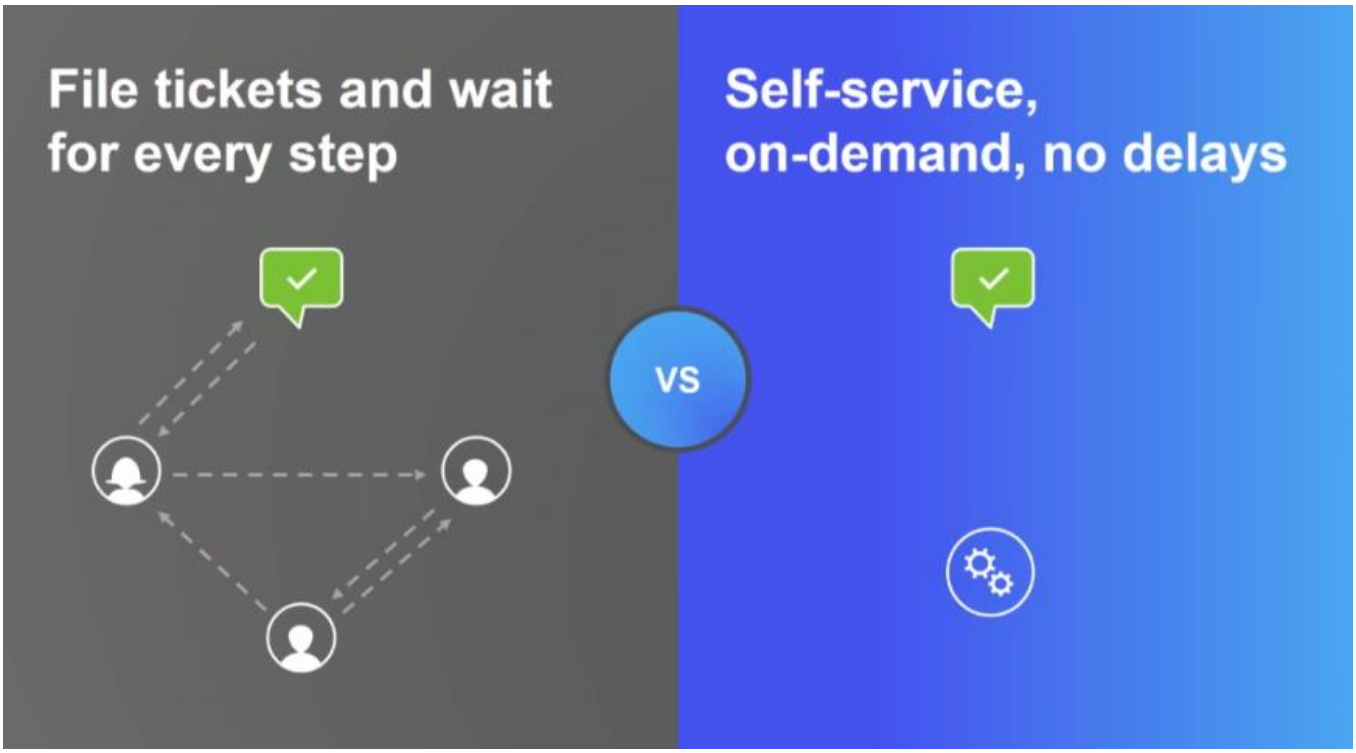
Pay up front and depreciate over 3 years

Pay a month later for the number of seconds used

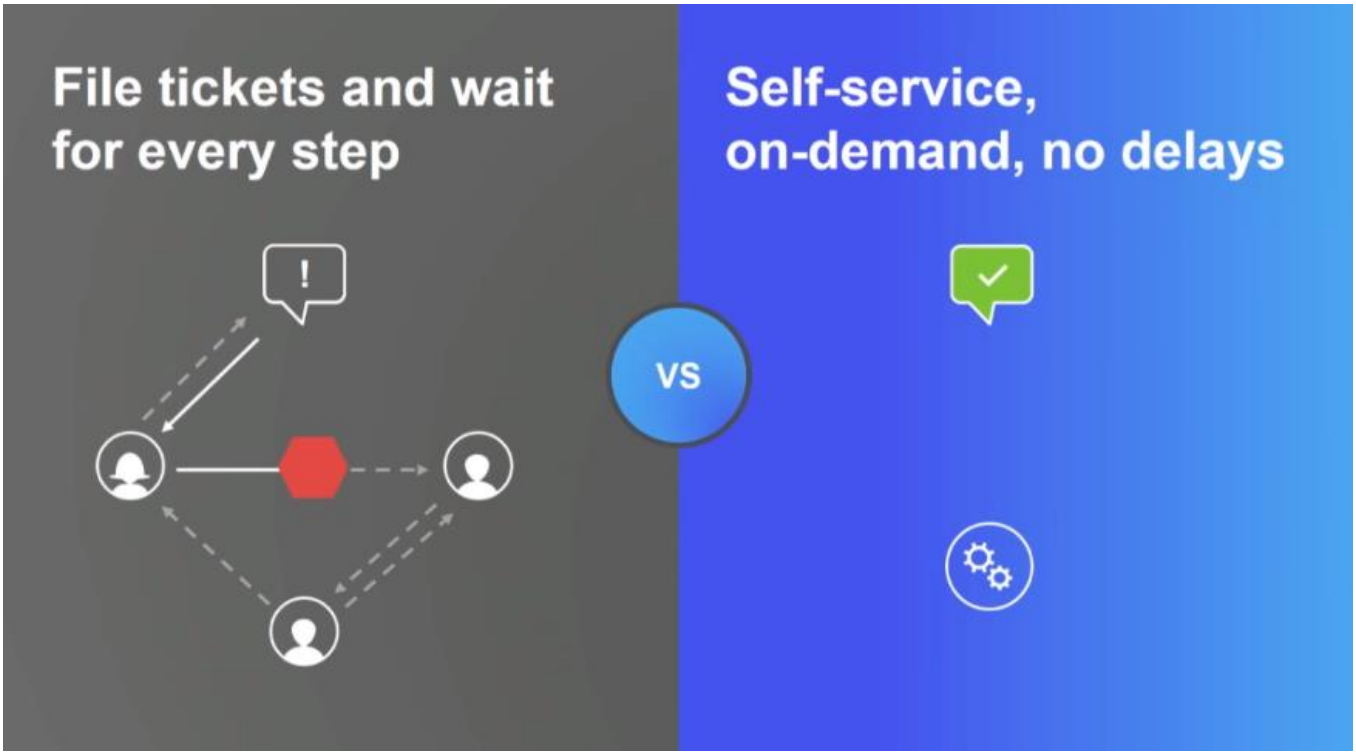
Cloud Native Principle

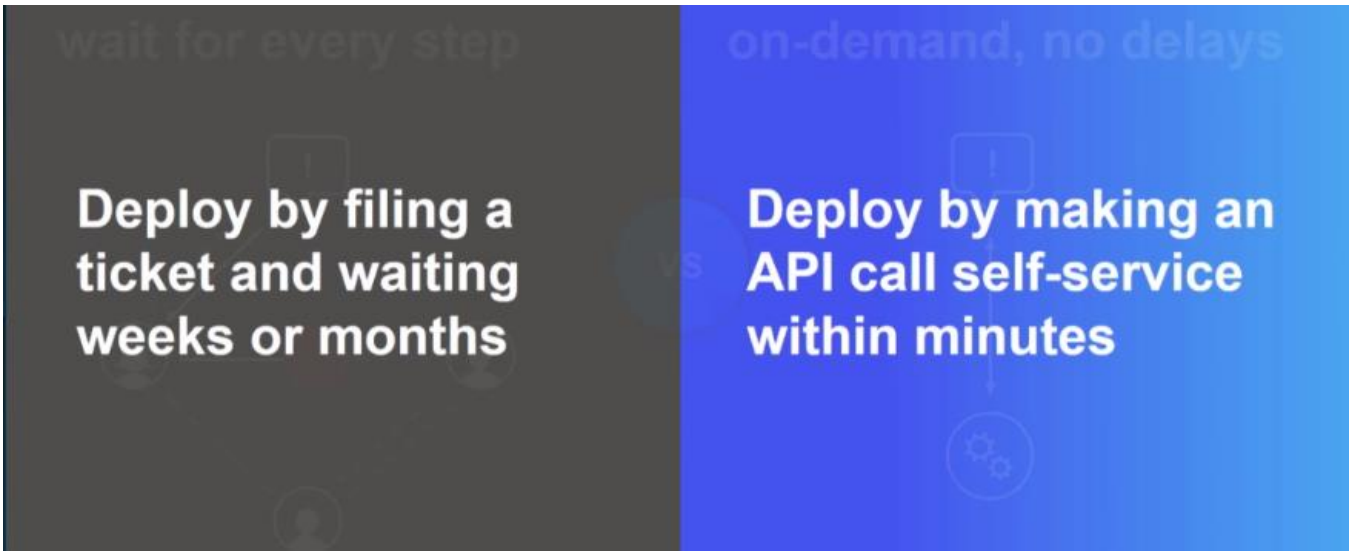
Pay for what you used last month…

…not what you guess you will need next year.

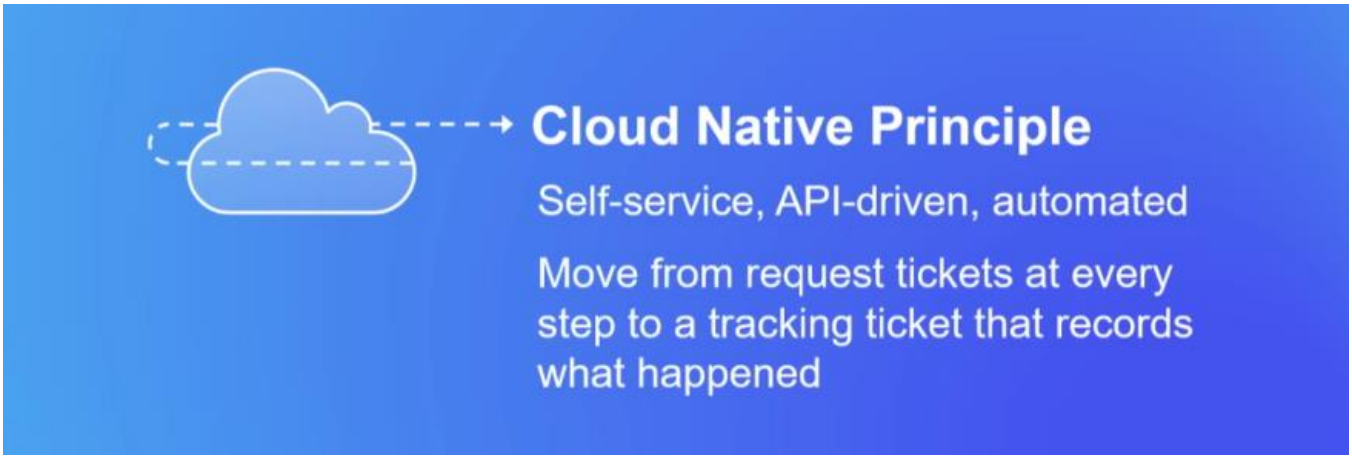


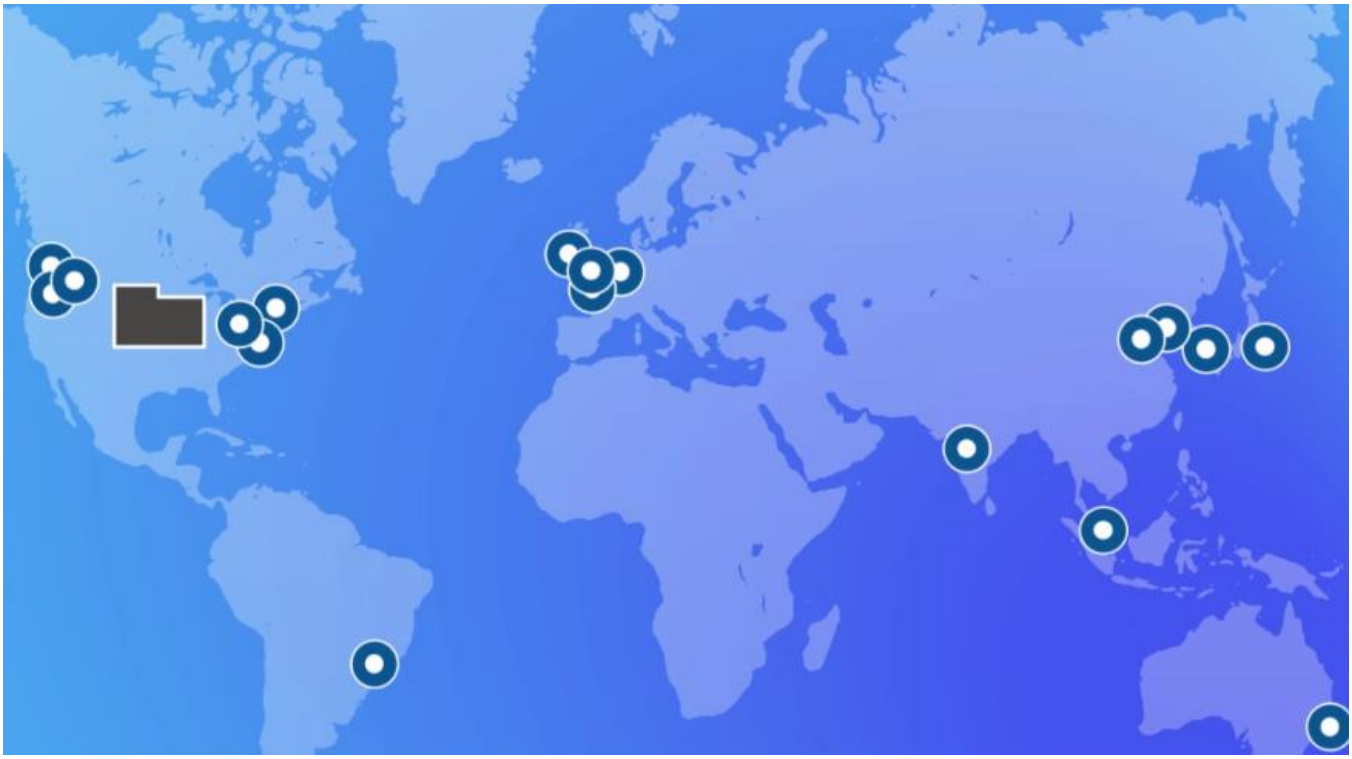
What you want is self-service, on-demand, API driven services

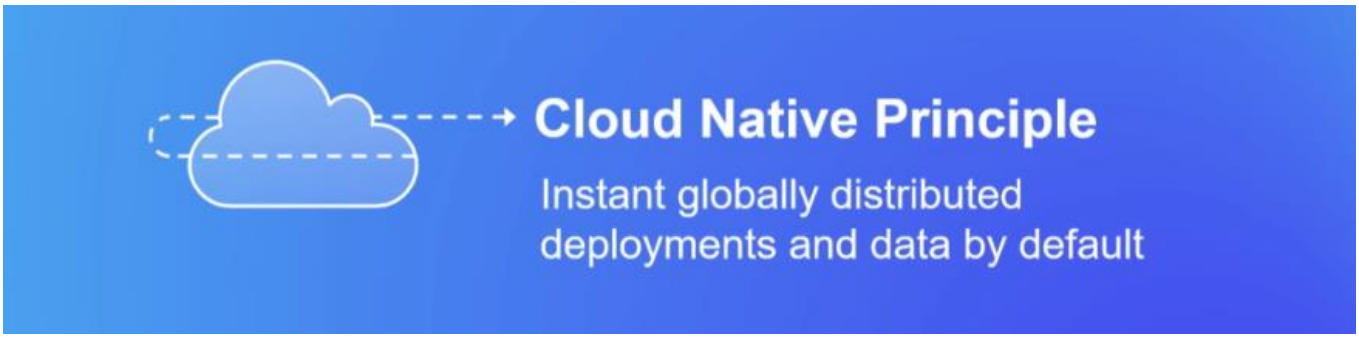


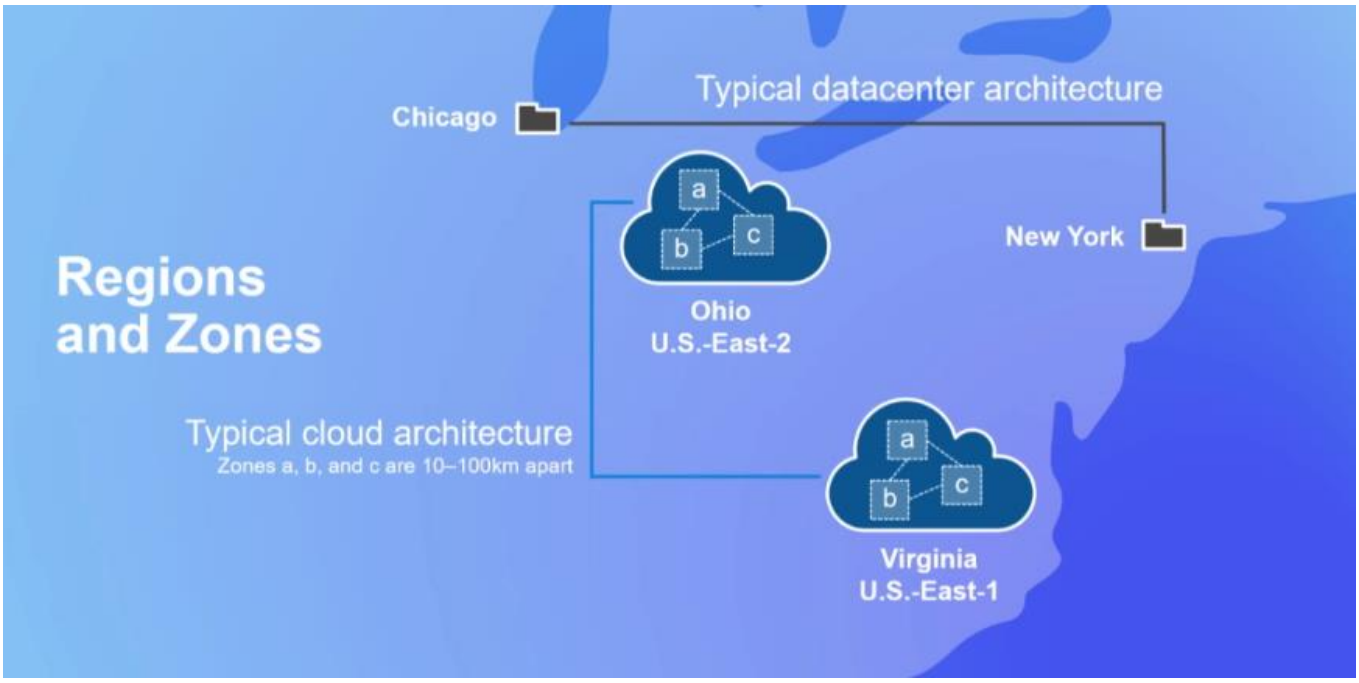


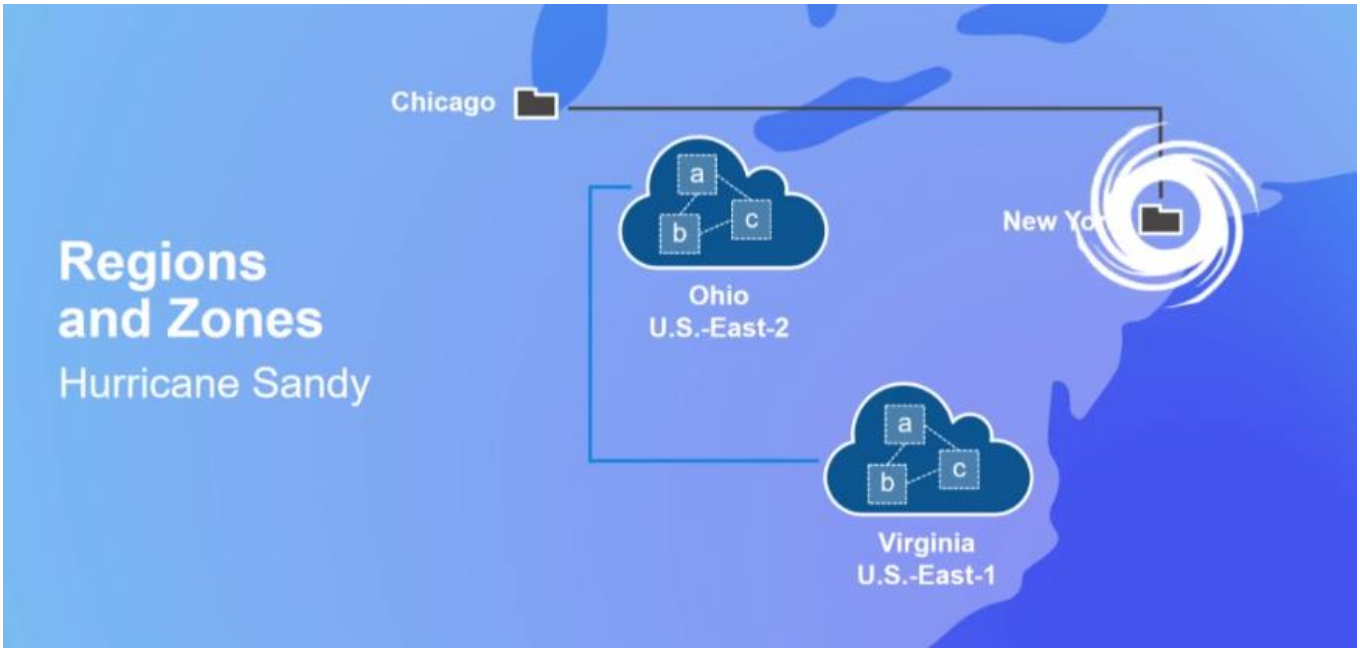
You need to make all your operational infrastructure be API call driven to be truly cloud-native

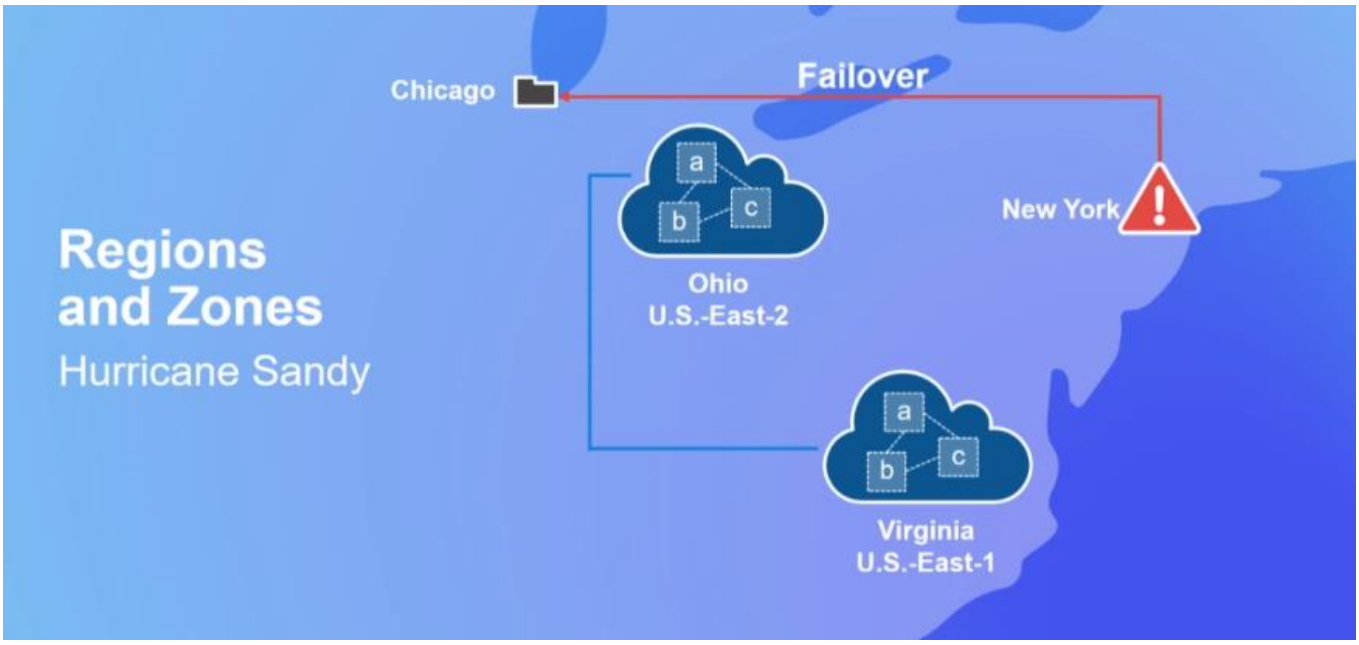


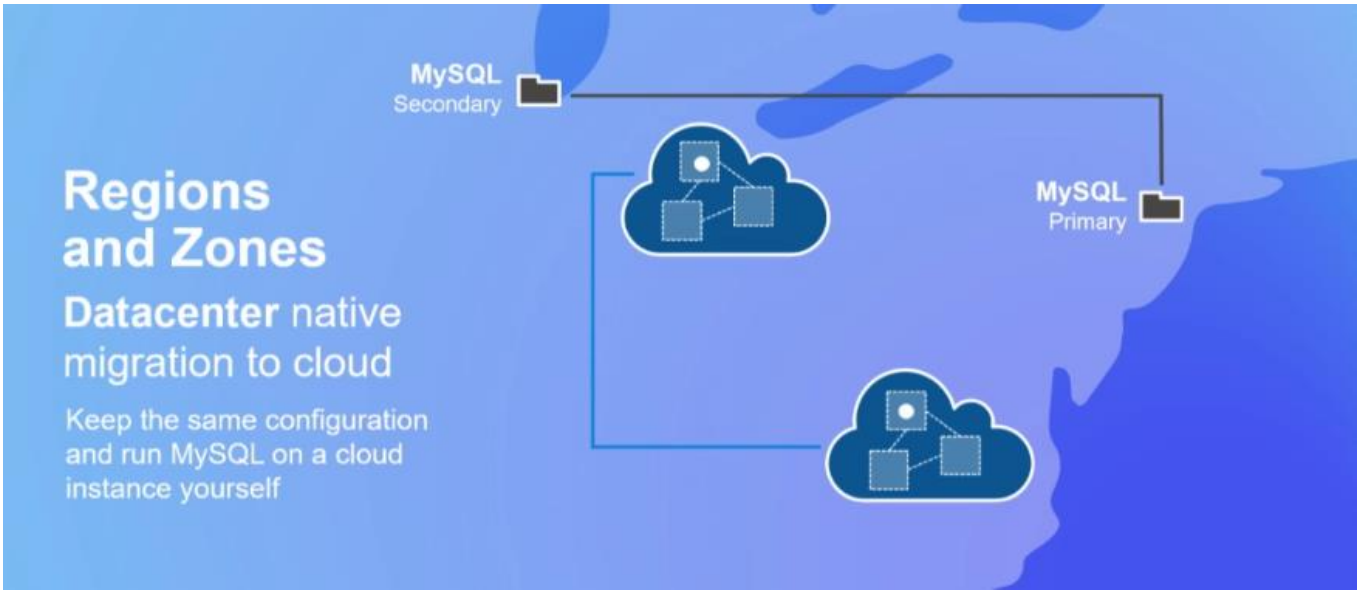



Cloud Native Principle
Instant globally distributed
deployments and data by default

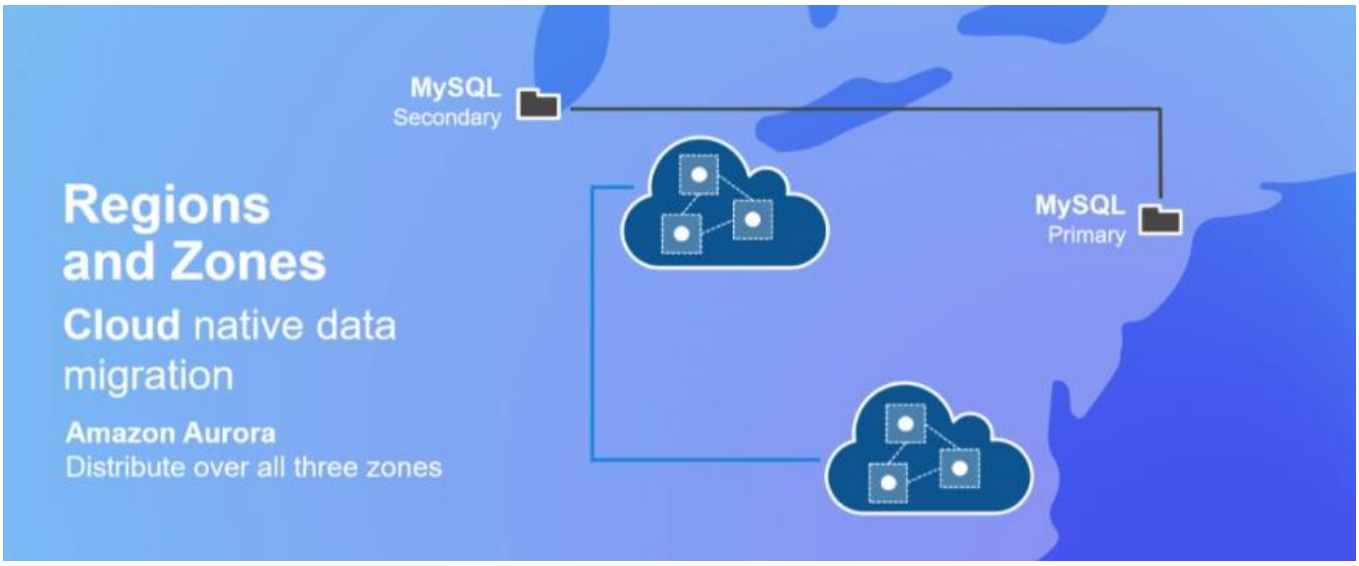

Regions
and Zones
Typical datacenter architecture
Chicago
New York
a
b
c
Ohio
U.S.-East-2
Typical cloud architecture
Zones a, b, and c are 10–100km apart
Virginia
U.S.-East-1

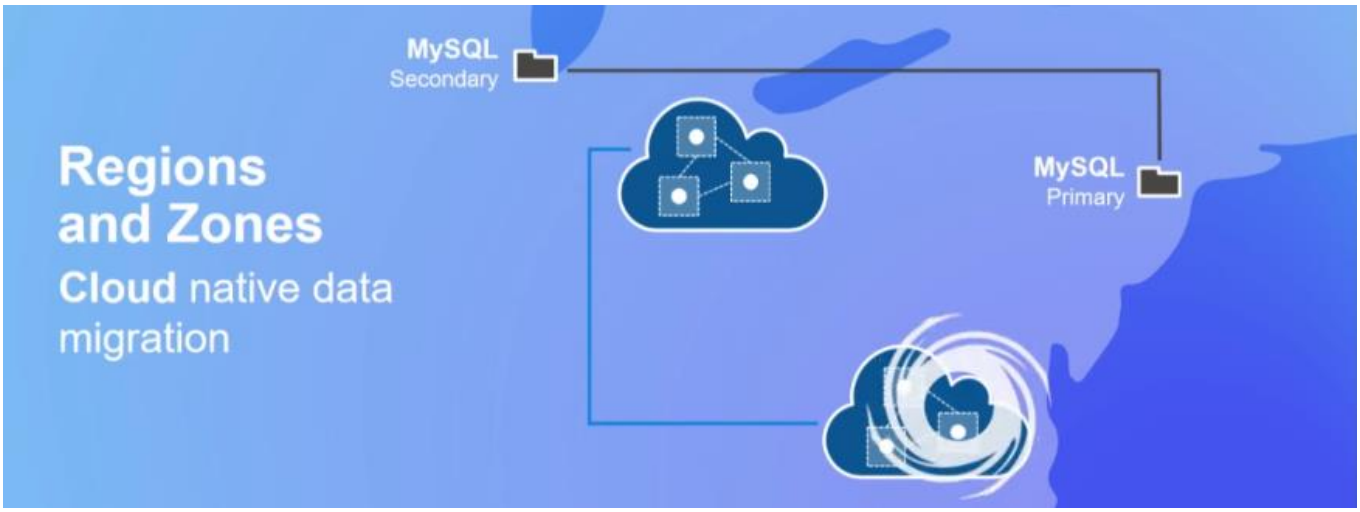

Regions
and Zones
Hurricane Sandy
Chicago
a
b
c
Ohio
U.S.-East-2
Virginia
U.S.-East-1





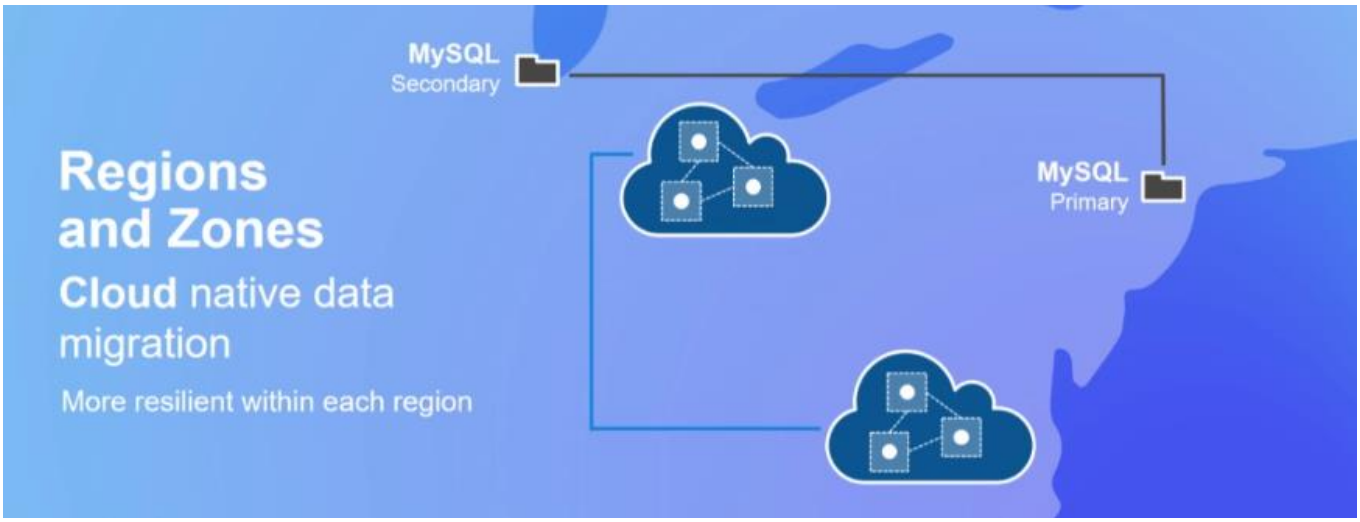
What you really want to do to have a distributed database is to use the Aurora service as below





Regions and Zones

Cloud native data migration



Regions and Zones

Cloud native data migration

More resilient within each region

Cloud Native Principle

Distribute over zones within a region by default

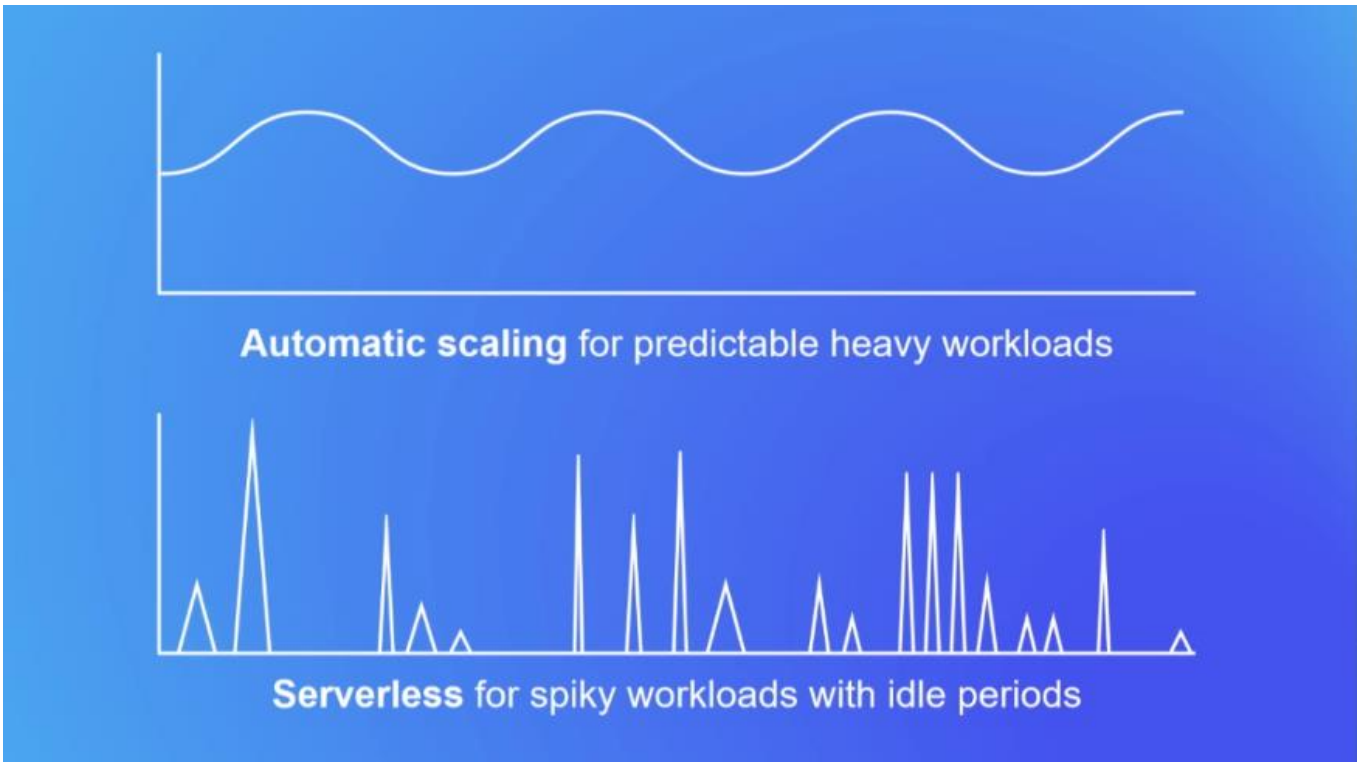
Elasticity

DATACENTER

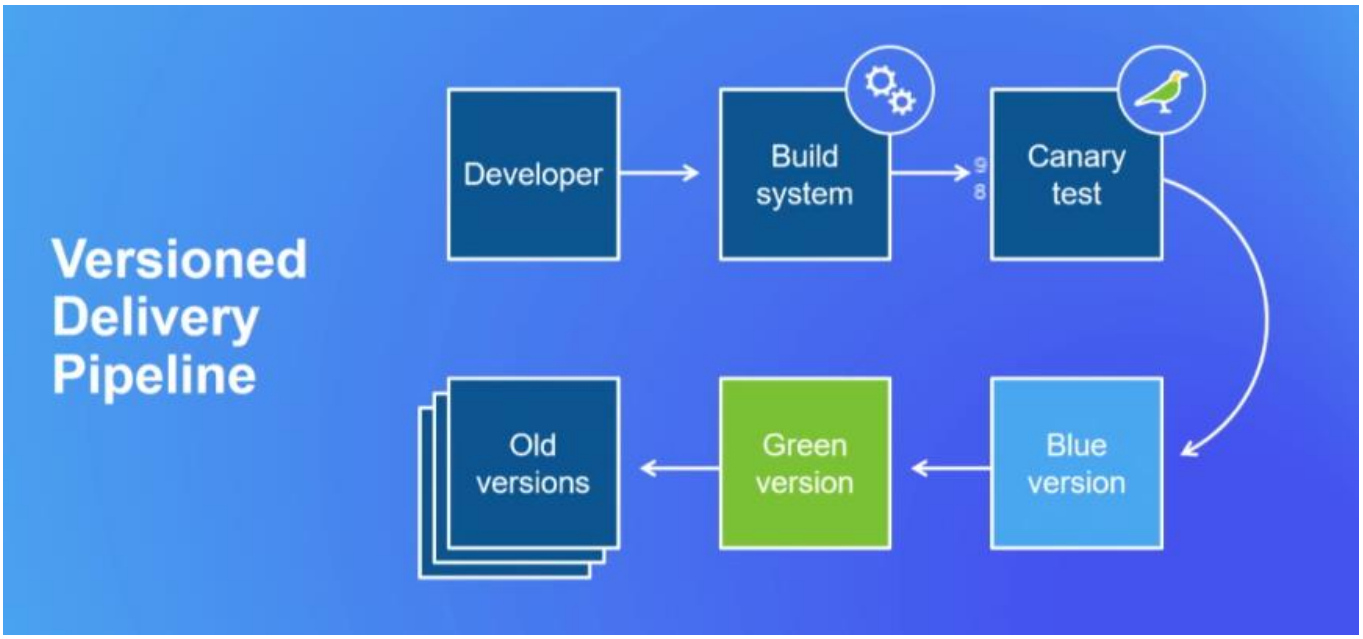
Hard to get over 10 percent utilization—need extra capacity in case of peak

CLOUD

Target over 40 percent utilization—no capacity overload issues


Automatic scaling for predictable heavy workloads
Serverless for spiky workloads with idle periods


Cloud Native Principle
Turn it off when it's idle
Many times higher utilization
Huge cost savings
Avoids capacity overloads


Versioned Delivery Pipeline
Developer
Build system
Canary test
Blue version
Green version
Old versions

Cloud Native Principle

Immutable code

Automated builds
Ephemeral instances, containers, and functions
Blue–Green deployments
Versioned services

Cloud Native Principles

Pay as you go, afterward
Self-service—no waiting
Globally distributed by default
Cross-zone/region availability models
High utilization—turn idle resources off
Immutable code deployments

Cloud Native Principles

Remain constant as practices evolve

Cloud Native Architecture

Building on Cloud

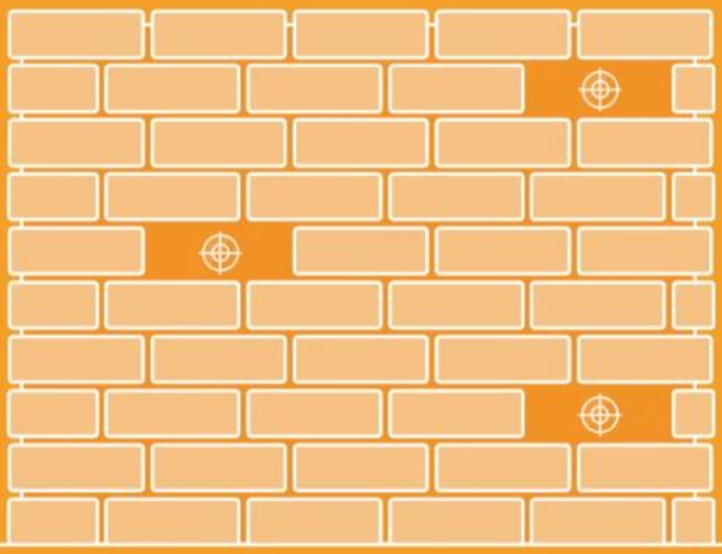
Startup Spaces

Incremental

Gap fillers

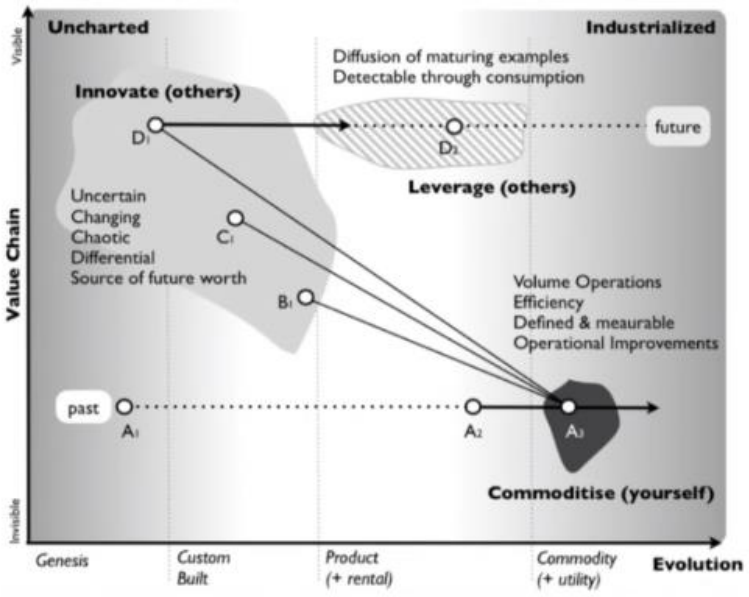
Leverage

Category creator



Leverage Wardley Maps

Move up the value chain



With Leverage Comes Dependencies…

Lock-in and the Lifecycle of Dependencies

Choosing, using, and losing

What is the return on investment (ROI) for each phase?

Choosing

Using

Losing

What is the ROI for each phase?

How has ROI changed with advances in technology and practices?

Choosing

Using

Losing

Choosing

Choosing

Investments

Negotiating, learning, experimenting
Hiring experts, building
Installing, customizing
Developing, training

Choosing

How much time elapses?

"The best decision is the right decision. The next-best decision is the wrong decision. The worst decision is no decision."

— *Scott McNealy*

Choosing

Analysis paralysis vs. Snap judgement

Choosing

Making a commitment

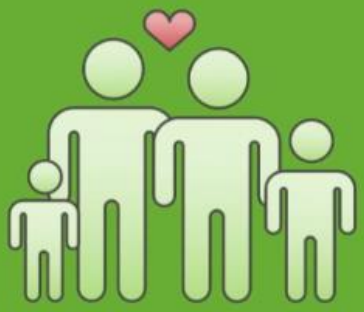
Whenever development is frozen and the operations team takes over, the key is turned in the lock

Choosing—What Changed?

Old World	New World
Monolith—all in one	Microservice—fine grain
Proof of concept install	Web service/open source
Enterprise purchase cycle	Free tier/free trial
Months	Minutes
$100K–Millions	$0–$1,000s

Using

Using



Investments

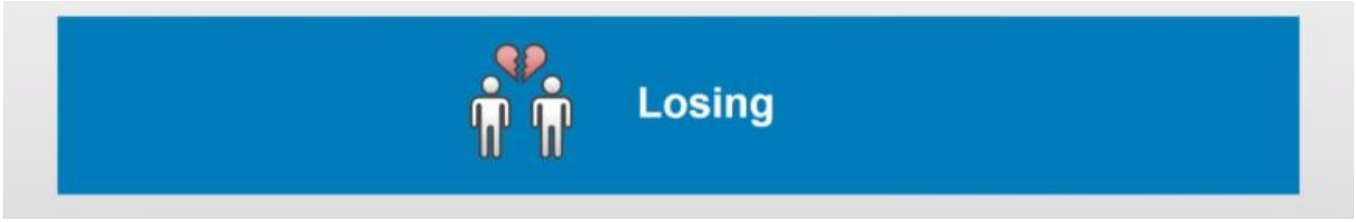
Cost of setup
Cost of operation
Capacity planning
Scenario planning
Incident management
Tuning performance and utilization

Using

Returns

Service capabilities
Availability, functionality
Scalability, agility
Efficiency

The Using phase is about moving from a Project to a Product



Losing

Returns

Reduced spending
More advanced technology
Better service, agility, scalability
Choose again, the cycle continues…

Losing—What Changed?

Old World	New World
Monolithic—all or nothing	Microservices—fine grain
Frozen waterfall projects	Agile continuous delivery
Long-term contracts	Pay as you go
Local dependencies	Remote web services

Old World Monolithic on-prem waterfall lock-in	New World Agile cloud-native micro-dependencies
Years	Weeks
Millions of dollars	Hundreds of dollars
Hundreds of dev years	A few dev weeks
Lock-in	Refactoring
Lawyers and contracts	Self-service

Bottom Line

ROI for choosing, using, losing has changed radically. Stop talking about lock-in, it's just refactoring dependencies

The cost of each dependency is far lower

Frequency of refactoring is far higher

Investment and return are much more incremental

Chaos Architecture

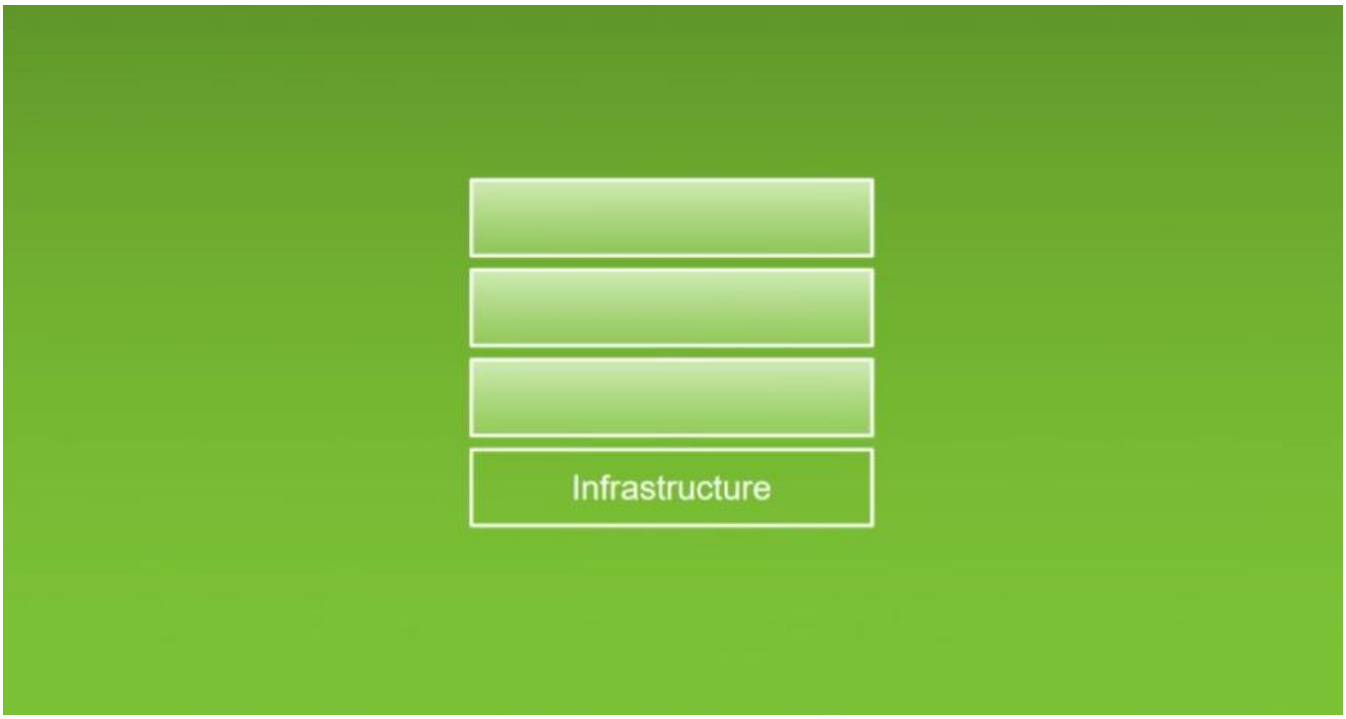
Infrastructure and Services

No single point of failure

No single point of failure

No single point of failure

Distributed Replicated
Automated Cloud



You now need to build an architecture that defines how does the data get to 2 places and how does the data stay in sync? You need a strategy for failovers and have a way of re-routing customers to the backup source when the main source fails and goes down reliably, how do you route back when the main source comes back on stream?

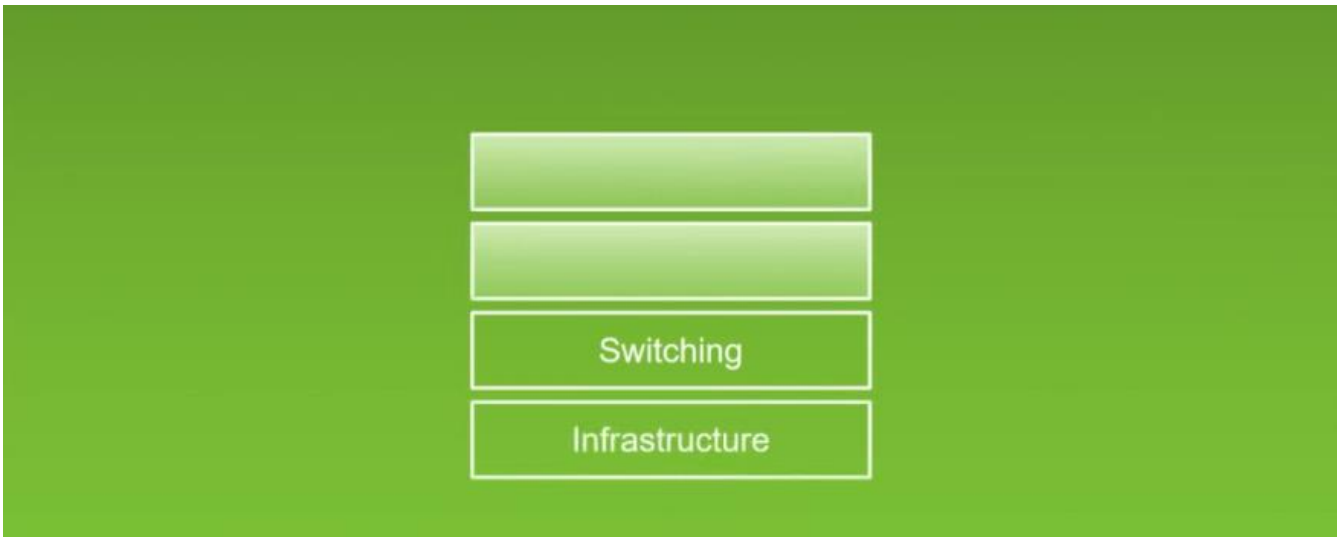
Who has a backup datacenter? What's the best description of it?

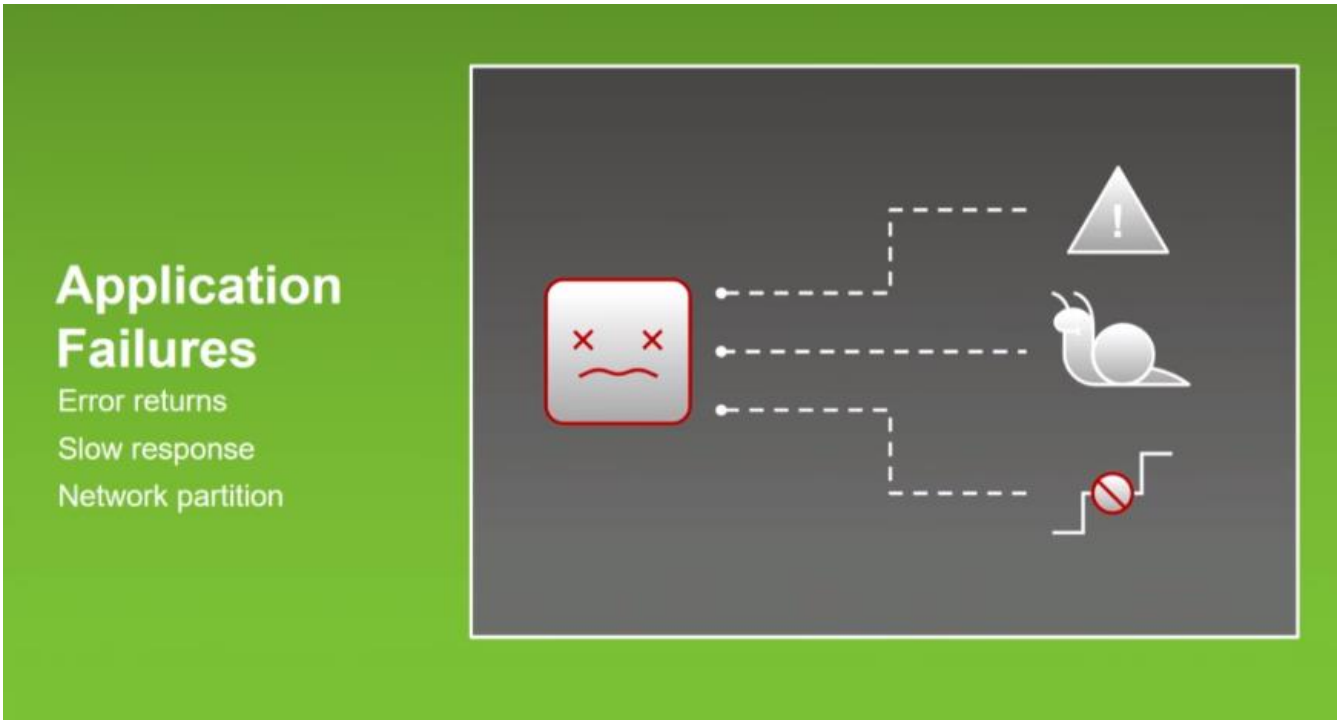
1. Availability theater—never tried to use it
2. Infrequent partial testing
3. Regular tests during maintenance
4. Frequent failovers during production to prove that no-one can tell it's happening

Route updates and customer requests to specific regions and services

Replicate data and re-route requests during incidents

Switching mechanism must be far more reliable than redundant elements you are switching between





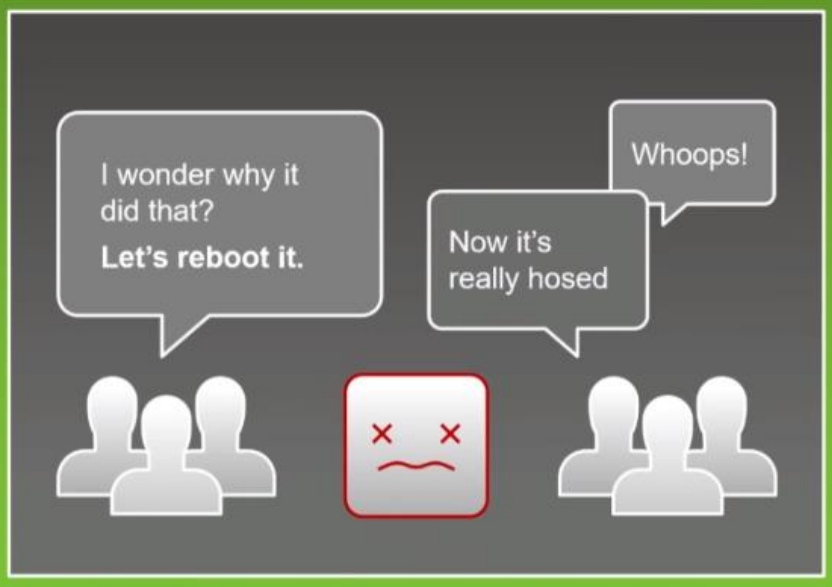
What happens when your app faces error returns from the APIs? Or slow responses? What do you want the reduced state or degraded state look like to your end user? Are you going to let them continue to di what they want to do if your service is down?

Microservices limit ‘blast radius’ for software incidents

Circuit breakers limit damage
Bulkheads prevent it from spreading
DITTO—Do Idempotent Things To Others
Avoid update and delete semantics

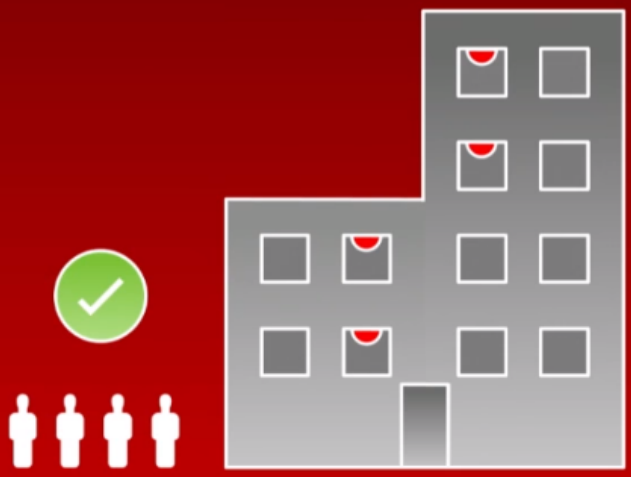
People

Unexpected application behavior often causes people to intervene and make the situation worse



People Training

A fire drill is a boring routine where we make everyone take the stairs and assemble in the parking lot

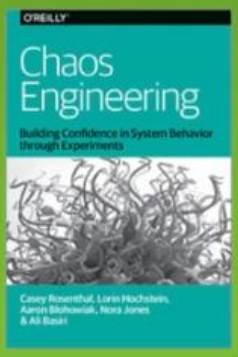


People Training

Fire drills save lives in the event of a real fire, because people are trained for how to react

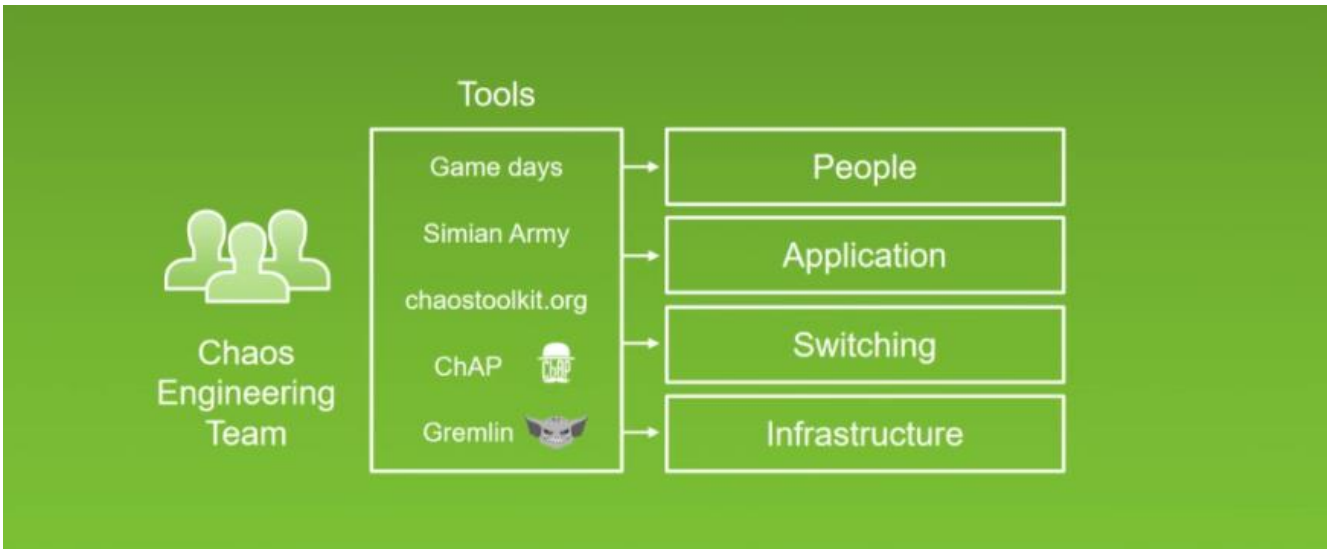
Who runs the 'fire drill' for IT?

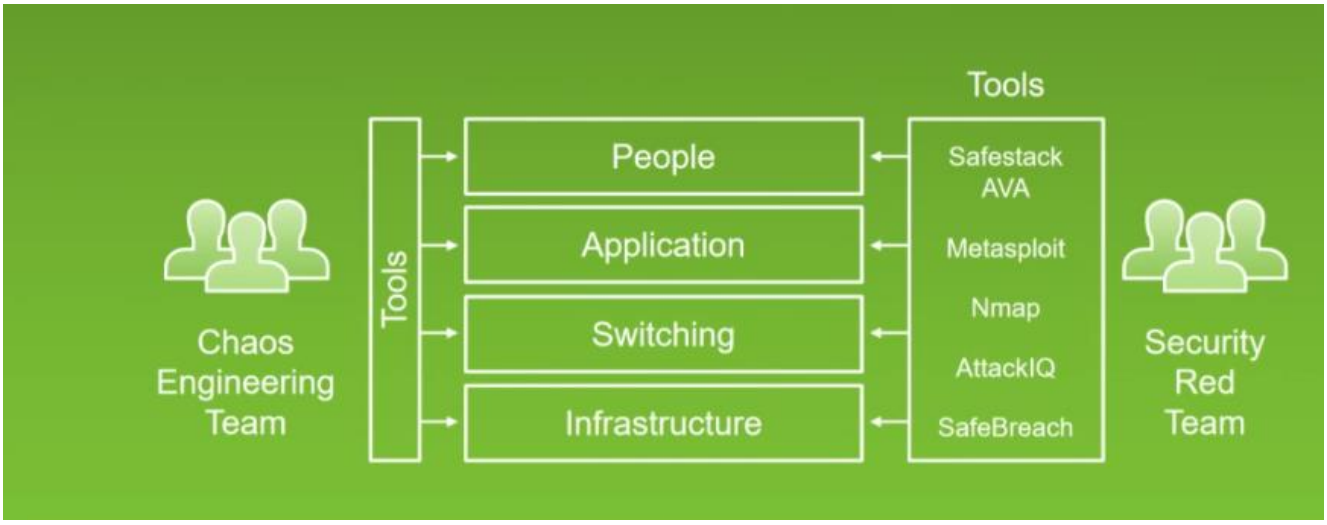
- People
- Application
- Switching
- Infrastructure



Chaos Engineering Team

- People
- Application
- Switching
- Infrastructure





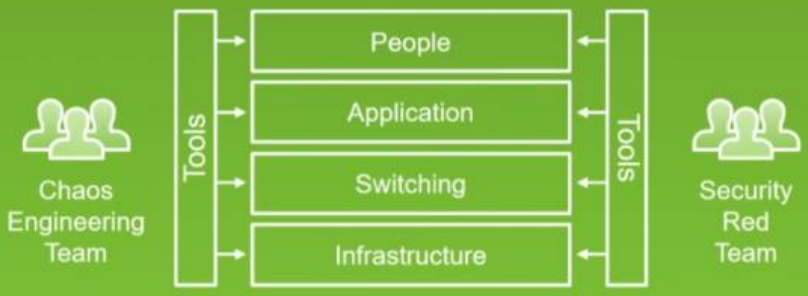
Chaos Architecture

Four layers

Two teams

An attitude—

Break it to make it better



Risk Tolerance

Who is at risk for what?
Is downtime a bigger risk?

Incident Lifecycle

Break It to Make It Safer

The “new view” of safety based on:
Todd Conklin’s pre-accident podcast
John Allspaw’s stella.report
Sydney Dekker—Drift into Failure

Failures are a System Problem— Lack of Safety Margin

Not something with a root cause of component or human error

Blindfolded on a cliff edge, what would you do?

Hypothesis Testing

- We think we have safety margin in this dimension, let's carefully test to be sure
- In production
- Without causing an issue

How to Select a Test?

LDFI—Lineage-driven Fault Injection
Dependency graphs for important flows

Experienced staff

Robust applications

Dependable switching fabric

Redundant service foundation

Chaos Architecture

aws

A cloud native availability model

Digital Transformation