

Introducing Cloud Native Architecture

Used to build Cloud Native applications



What is Cloud Native?

A new style of architecture

Competency

Distributed Computing

Clip slide

4

- Multi-master
- Many Data Centers
- Many Fault Domains
- Many Regions
- Global Server Load Balancing
- Replication

Microservices

3

- Minimal Function
- Service Discovery
- API-first
- Polyglot
- Choreography
- Loose Coupling

* as a Service

2

- Consume Infrastructure and Software as a Service
- Auto-scaling
- Infinite Elasticity
- Fault Tolerant by Definition

DevOps

1

- Automated Provisioning
- Automated Setup
- Continuous Integration
- Continuous Delivery
- Automated Testing
- Agile
- Culture Change

Prerequisite #1 - DevOps

A prerequisite to consuming infrastructure and software as a service

Culture Constrained by Technology

Culture

Respect

Discuss

Avoid Blaming

"Done" Means Released

Get development and ops to work together

Technology

Infrastructure as Code

Shared Version Control

One Step Build/Deploy

Don't Fix Anything

Assume Your Infrastructure is Unreliable
Even though it is actually pretty reliable these days

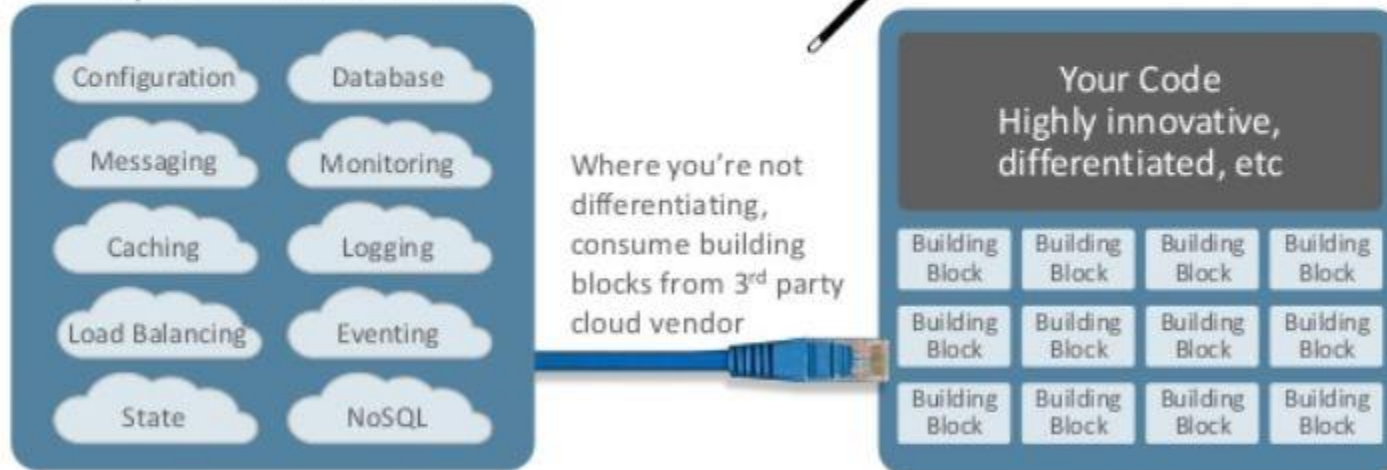


Java

Prerequisite #2 - * as a Service

Requires fundamental shift in how applications are built

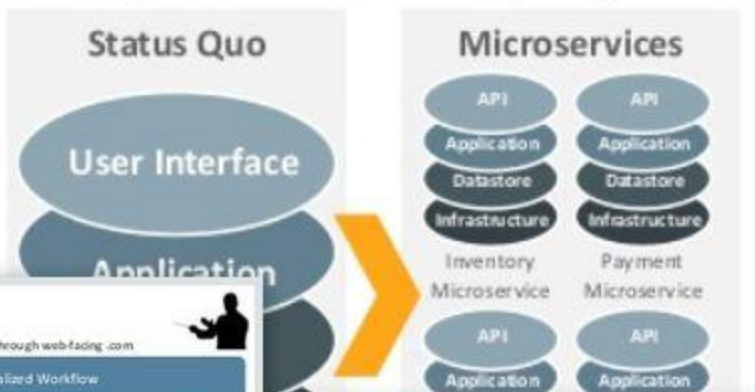
3rd Party Cloud – On or Off Premise



Prerequisite #3 – Microservices

Break up your application into many small pieces to get features to market quickly

- Single monolithic application -> small, independently deployable microservices
- Each microservice:
 - Has its own team that designs, builds, deploys and maintains it
 - Exposes an API, which can be elsewhere



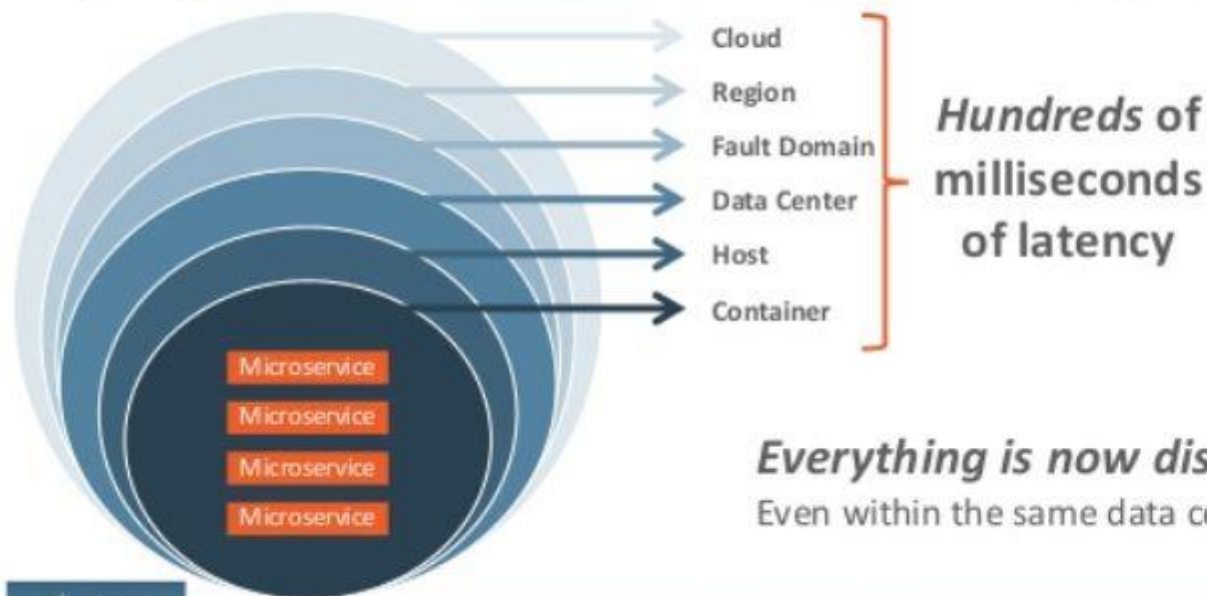
and async messaging

Many Small Microservices

Java

Prerequisite #4 – Distributed Computing

Run your applications across multiple data centers, fault domains, regions, etc



Everything is now distributed
Even within the same data center

What's the Value of Cloud Native?

#1 Value – Quickly Capitalize on Business Opportunities

SPEED

Quickly move changes

RESILIENCE

Survive

Single of Orchestration

How to orchestrate your cloud native application and testing, etc.



Example of Diversity

Example of Diversity: Multiple instances of an application running on different hardware and software configurations.



Assume Your Infrastructure is Unreliable

Even though it is actually pretty reliable these days.



Strip Your Applications Headings

Strip your applications down to a single language, but don't strip the logic.



Consider Deploying Front/Back Ends To Different Locations

Offering users that a cloud native app.



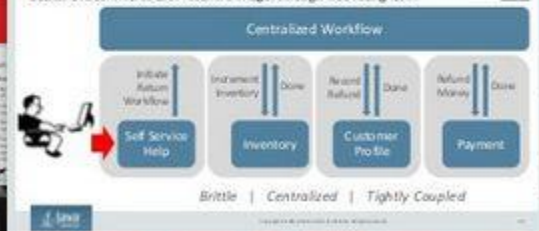
java

Clip slide

DevOps + * as a Service + Microservices + Distributed *Make Cloud Native Available to Everyone Else*

Example of Orchestration

Scenario: a Commerce user returns a widget through web-facing .com



Geographically Distribute Your Workflows

- Multiple data centers
- Multiple edge locations
- Multiple regions
- Multiple providers

Choreography Over Orchestration

Advantages:
- No shared state or global context
- No shared state or global context
- No shared state or global context
- No shared state or global context

Adopt your API Gateway

API Gateway is a single entry point for all the APIs in your system. It acts as a reverse proxy, routing requests to the appropriate service.

Assemble Your Infrastructure by Composing

Infrastructure as Code (IaC) allows you to define and manage infrastructure as code, making it easier to deploy and scale your applications.

Adopt an API Gateway

API Gateway is a single entry point for all the APIs in your system. It acts as a reverse proxy, routing requests to the appropriate service.

Java

Cloud Native is Difficult

But not doing it is worse

*52% of the Fortune 500 have been merged,
acquired, and gone bankrupt since 2000*

Which path are you going to take?

Building Cloud Native Applications

Choreography Over Orchestration Between Microservices

Orchestration

- Top-down coordination of discrete actions
- Used in centralized, monolithic applications
- Brittle = centralized by nature
- Each "action" registers with centralized system – single point of failure that is not very flexible

Choreography

- Bottom-up coordination of discrete actions
- Used in distributed, microservice applications
- Resilient = distributed by nature
- Each microservice asynchronously throws up a message that other microservices can consume



Example of Choreography



Assume Your Infrastructure is Unreliable



Changes Required to Adopt Cloud Native



Cloud Native Takes Real Organizational Commitment

The legacy style has inertia

- **LARGE HORIZONTAL LAYERS FOCUSED ON PROJECTS-> SMALL VERTICAL TEAMS FOCUSED ON PRODUCTS**
- **CHANGE HOW PEOPLE ARE REWARDED**
- **CHANGE PEOPLE'S JOBS**

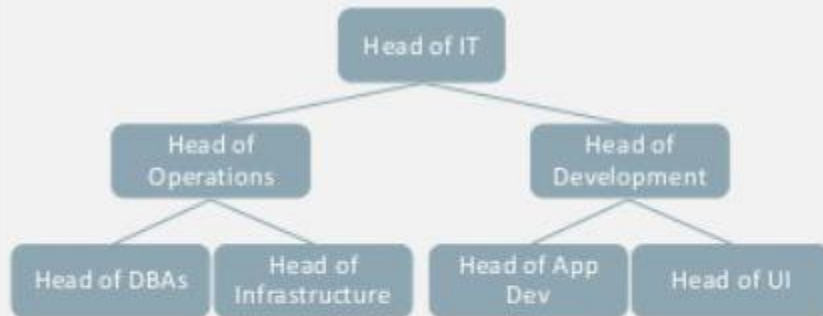
RETHINK **YOUR ORGANIZATION**



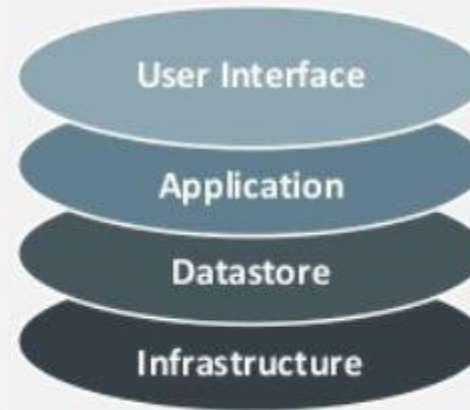
Horizontally Tiered Enterprises == Horizontally Tiered Apps

Conway's Law: Software reflects the structure of the organization that produced it

Typical Enterprise Organization Structure



Resulting Software



An Enormous Monolith

Even Simple Changes Are Hard to Implement With Monoliths

Organizational boundaries introduce the need to extensively coordinate

New requirement: Add a birthdate property to the customer's profile. How does this get implemented?

1. Application developer tickets DBAs to
2. Application developer adds the new property to the application-level code
3. Application developer tickets UI team to have them add that property to the profile screens



*Different Teams
Different Timelines
Different Priorities*

Adopt an API Gateway

API gateways provide a "backend for each frontend"



- Builds a XML or JSON response for each type of client - web, mobile, etc.
- Asynchronously calls each of the N microservices required to build a response
- Handles security and hides backend
- Load balances
- Applies limited business logic
- Makes APIs
- Logs centrally



Characteristics of Different Organization Types

Centralized Organizations Focused on Technology Layers

- Produce monoliths
- Simple change requires extensive coordination across all of the different layers
- Business logic is spread everywhere because it's easier to bury it in the layer you own
- No real ownership

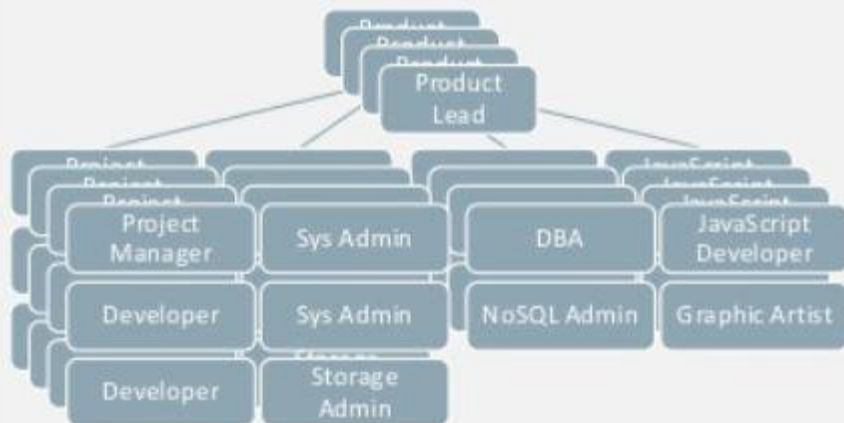
Distributed Organizations Focused on Products

- Produce microservices
- A team can make any change to their microservice
- Architecture more clean because it's not done by central committee
- **A lot of freedom and responsibilities**
- **True ownership**

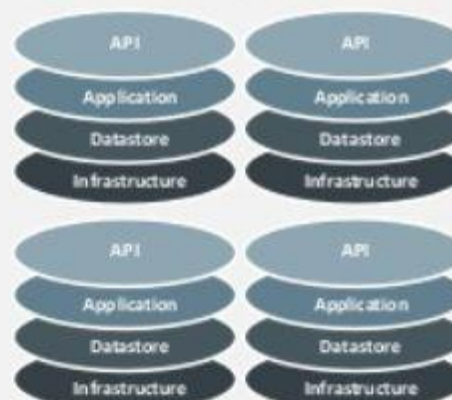
Re-structure Your Organization – Put Conway's Law to Work

Build small product-focused teams – strict one team to one microservice mapping

Microservices Organization Structure



Resulting Software



Many Small Microservices

Small Teams = Much Better Communication

Low Latency/High Bandwidth Communication



Remove All Hard-coded IPs, Host Names, etc

Use service discovery, DNS, etc instead. Everything should be dynamic

```

thisPort=192.168.1.122.1
httpsPort=7443
httpPort=7444
mailPort=80
lockServer=192.168.1.122.23
lockServerPort=7442
'Configuration: LockServer: 192.168.1.122.23

```



Legacy

services

Much Faster Turnaround With Microservices

Hey Jim, can you add that column to the database before lunch?

Low latency, high bandwidth communication

The profile microservice team – three people total, all sitting together

Turn around changes in hours vs. months

Adopt DevOps

Requires changes throughout your entire organization

Culture

Respect

- Dev respect for ops
- Ops respect for dev

Discuss

- Ops should be in dev discussions
- Dev should be in ops discussions
- Shared runbooks

Avoid Blaming

- No fingerprinting!
- Everyone should have some culpability

"Done" Means Released

- Dev's responsibility shouldn't ever end – production support required
- "Throwing it over the wall" is dead

Technology

Infra as Code

- Don't build envs by hand
- Scripts checked in and managed as src

Shared Version Control

- Single system
- Ship trunk
- Enable features through flags

One Step Build/Deploy

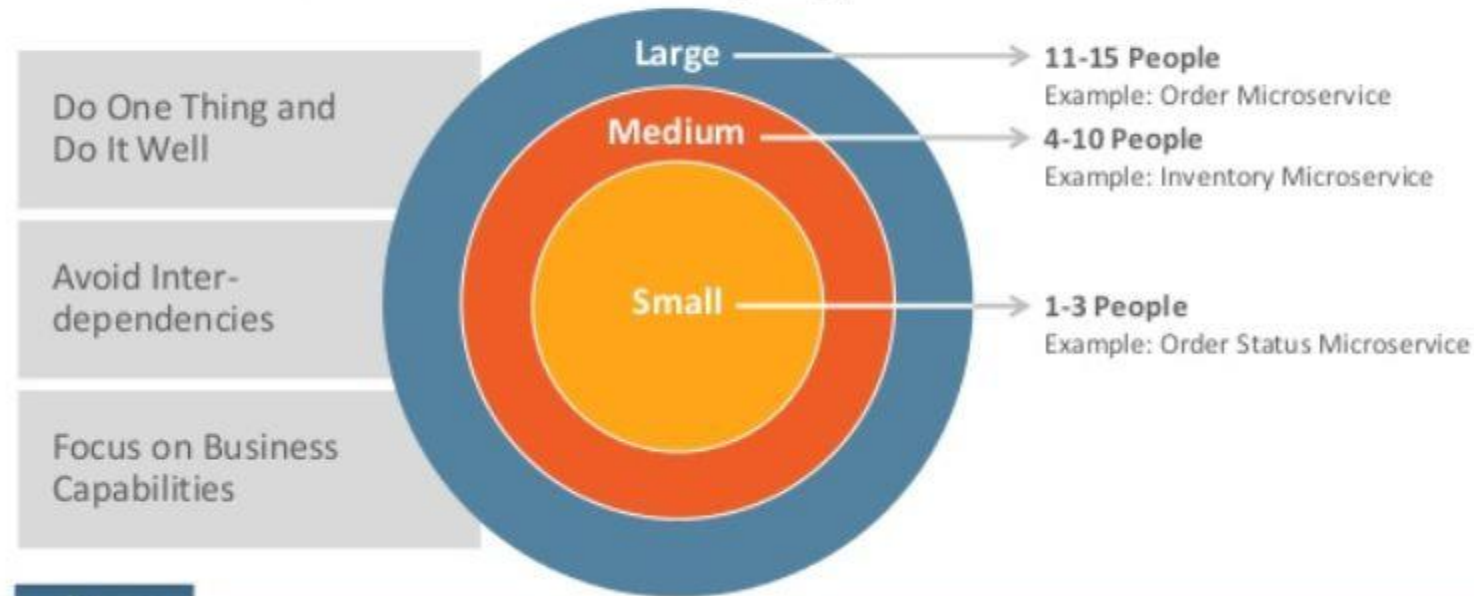
- One button build/deploy
- If verification fails, stop and alert or take action

Don't Fix Anything

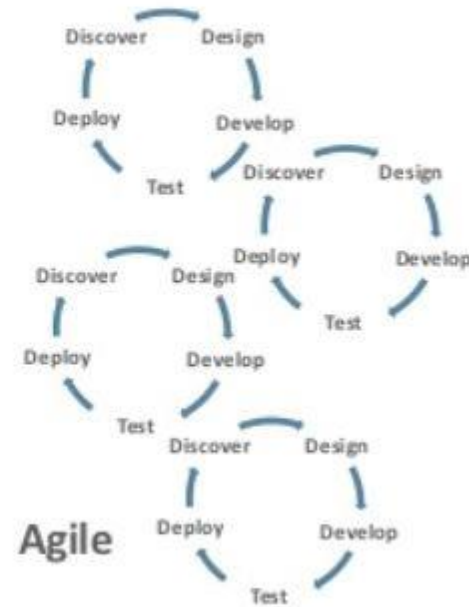
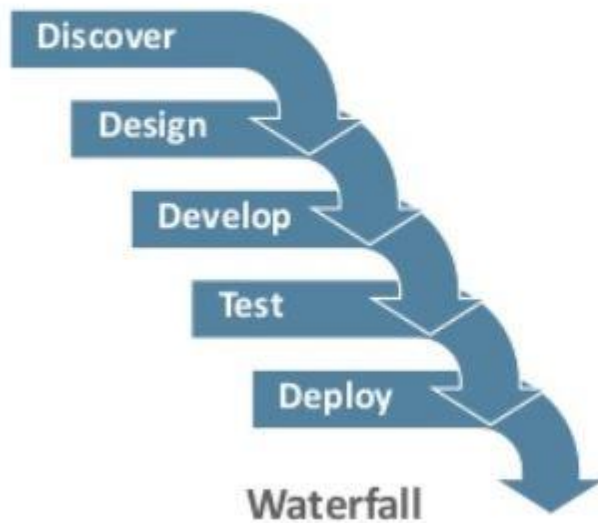
- If something breaks, re-deploy. Don't fix
- Fix environment setup scripts

Start Managing Small, Vertical Teams

Can have hundreds of microservices for a larger application



Change to a Form of Agile Project Management



Hold Each Person on Each Team Accountable

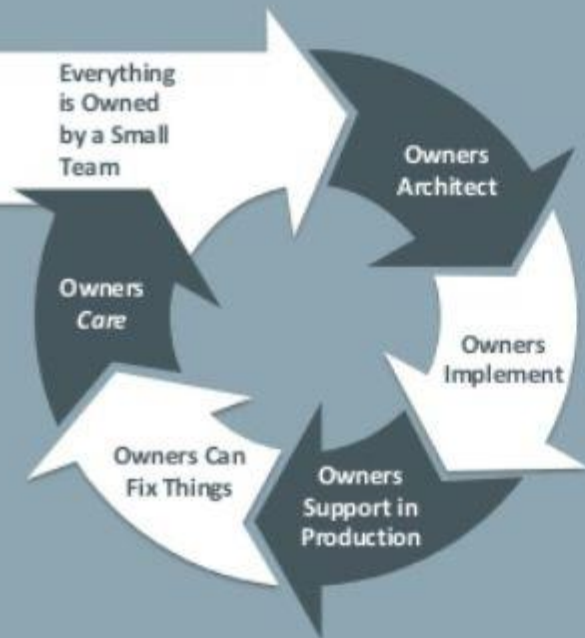
Much easier with microservices-style architecture



Example of Orchestration

Scenario: e-Commerce user returns a widget through web-facing .com





Ownership is Key to the Success of Cloud Native

In traditional enterprises, any one individual has very low ownership of anything. It's classic tragedy of the commons

Consume Application Building Blocks as Software

Cloud (*-as-a-Service) is key to innovation

Start consuming resources as a *service*

Building Blocks

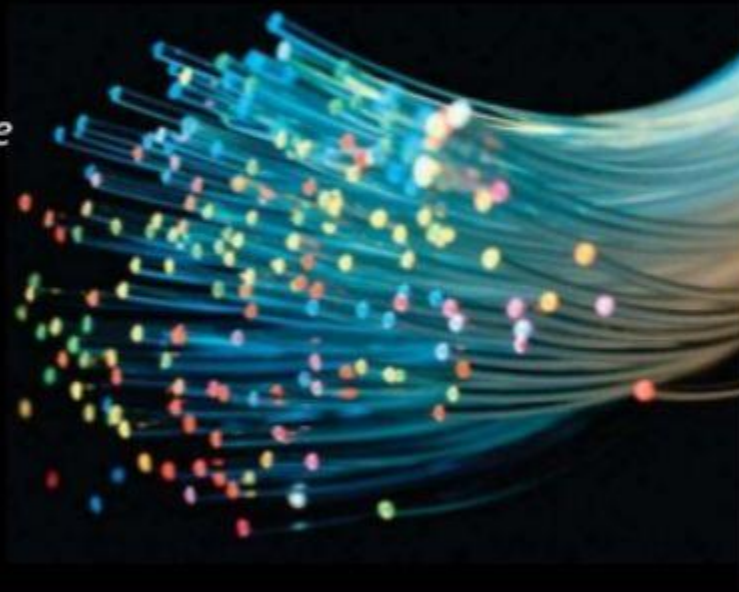
Database, NoSQL, Messaging, etc

Platforms

Java EE, Java SE, Node, etc

Infrastructure

Compute, Networking, Storage






Empower Developers to Make Procurement Decisions

- #1 focus of cloud native: time to market. Long-term maintenance should not be a big consideration
- Let developers who are innovating pick the absolute best technology for their own use
- Each small team supports their own microservice in perpetuity. No need to have large maintenance teams
- Standardization is best for system of record-style applications



Each Microservice Can Now Use Its Own Database/Datastore

RDBMS is great but not necessary for all use cases

Relational Database	NoSQL	Object Grids
ACID-compliant, suitable for a wide range of workloads. Trusted, reliable, wide client support, easy to use	Non-relational organization of data, including key/value, graph, document, tabular, etc. Always BASE and sometimes ACID compliant	Store objects in and move business logic into the server-side grid.
		



What Are Microservices?

Minimal function services that are deployed separately but can interact together to achieve a broader use-case

Monolithic Applications

- Single, Monolithic App
- Must Deploy Entire App
- One Database for Entire App
- Organized Around Technology Layers
- State In Each Runtime Instance
- One Technology Stack for Entire App
- In-process Calls Locally, SOAP Externally



Microservices

- Many, Smaller Minimal Function Microservices
- Can Deploy Each Microservice Independently
- Each Microservice Has Its Own Datastore
- Organized Around Business Capabilities
- State is Externalized
- Choice of Technology for Each Microservice
- REST Calls Over HTTP, Messaging, or Binary

Three Key Rules to Microservices

Can **build** a microservice independently



Can **release** each microservice independently

Choreography Over Orchestration Between Microservices

Orchestration

- Top-down coordination of discrete actions
- Used in centralized, monolithic applications
- Brittle → centralized by nature
- Each "action" registers with centralized system → single point of failure that is not very flexible

Choreography

- Bottom-up coordination of discrete actions
- Used in distributed, microservice applications
- Resilient → distributed by nature
- Each microservice asynchronously throws up a message that other microservices can consume



Everything Is Now Distributed. Get Used To It



Applications

Applications are now broken up into small microservices, with separate frontends and backends



Infrastructure

Different data centers, fault domains, regions, etc. Even within the same data center, latency may be unacceptably high

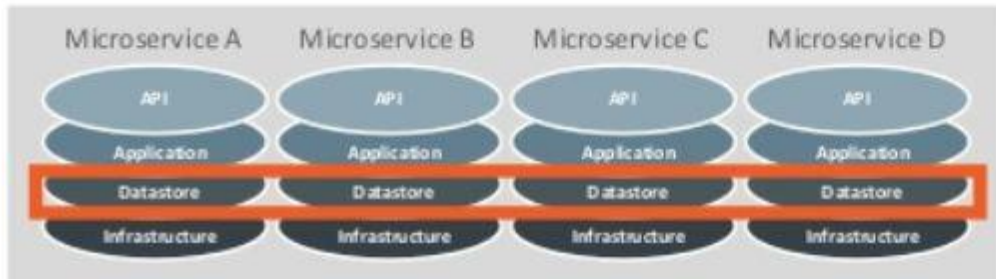


Teams

Many small teams, each responsible for their own microservice. Each team is often geographically distributed

Microservices Forces Move To Distributed Computing

Introduces enormous complexity – monoliths don't suffer from this



- Distributed computing is a natural consequence of microservices because each microservice has its own datastore
- Sharing datastores across microservices introduces coupling – very bad!
- There will always be latency between microservices
- Latency = eventual consistency

- All data exchange between microservices must be through API layer or messaging – no accessing datastores cross-microservices
- Must implement high-speed messaging for *synchronous* calls between microservices. REST + HTTP probably isn't fast enough
- May end up duplicating data across datastores – e.g. a customer's profile

Choreography Over Orchestration Between Microservices

Orchestration

- Top-down coordination of discrete actions
- Used in centralized, monolithic applications
- Brittle – centralized by nature
- Each “action” registers with centralized system – single point of failure that is not very flexible

Choreography

- Bottom-up coordination of discrete actions
- Used in distributed, microservice applications
- Resilient – distributed by nature
- Each microservice asynchronously throws up a message that other microservices can consume



Example of Orchestration

Scenario: eCommerce user returns a widget through web-facing .com



Centralized Workflow

Initiate
Return
Workflow

Increment
Inventory

Done

Record
Refund

Done

Refund
Money

Done

Choreography Over Orchestration Between Microservices

Orchestration

• Top-down coordination of discrete actions

• Used in centralized, monolithic applications

• Brittle – centralized by nature

• Each "action" registers with centralized system – single point of failure that is not very flexible

Choreography

• Bottom-up coordination of discrete actions

• Used in distributed, microservice applications

• Resilient – distributed by nature

• Each microservice asynchronously throws up a message that other microservices can consume

Example of Choreography

Scenario: User returns a widget

Step 1: User clicks 'Return'

Step 2: Inventory is incremented

Step 3: Refund is processed

Step 4: Confirmation is sent

Assume Your Infrastructure is Unavailable

How do you handle this?

How do you handle this?

How do you handle this?

How do you handle this?

How do you handle this?

Tightly Coupled



Example of Choreography

Scenario: Inventory microservice

Events This Microservice Cares About

Events This Microservice Emits

