Microservices + Oracle | A Bright Future

Kelly Goetsch
Director, Product Management ORACLE
January 28th 2016

Java ORACLE

Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Program Agenda

1. Introduction
2. History of Microservices
3. Non-technical Prerequisites
4. Architectural Prerequisites
5. Technical Prerequisites
6. Implementing Microservices
7. How Oracle Products Support Microservices
8. Summary

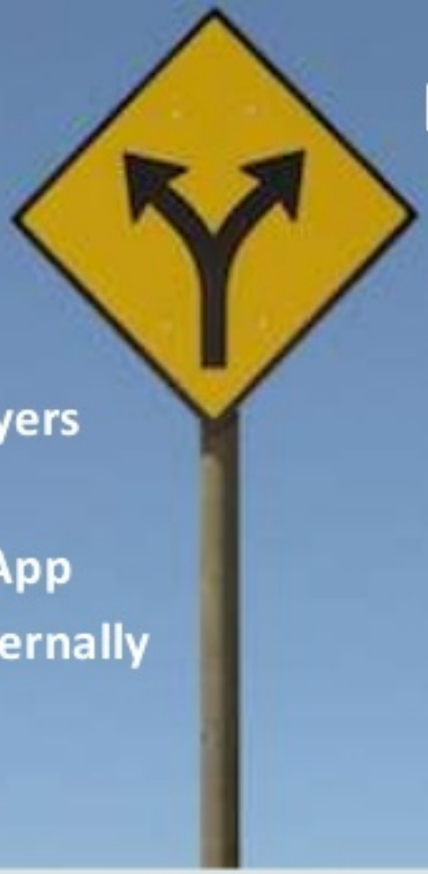
Introduction

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 Often Has Its Own Datastore
- Organized Around Business Capabilities
- State is Externalized
- Choice of Technology for Each Microservice
- REST Calls Over HTTP, Messaging, or Binary

Microservices Are Analogous to Unix Utilities

Same concept, different decade

"Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface."

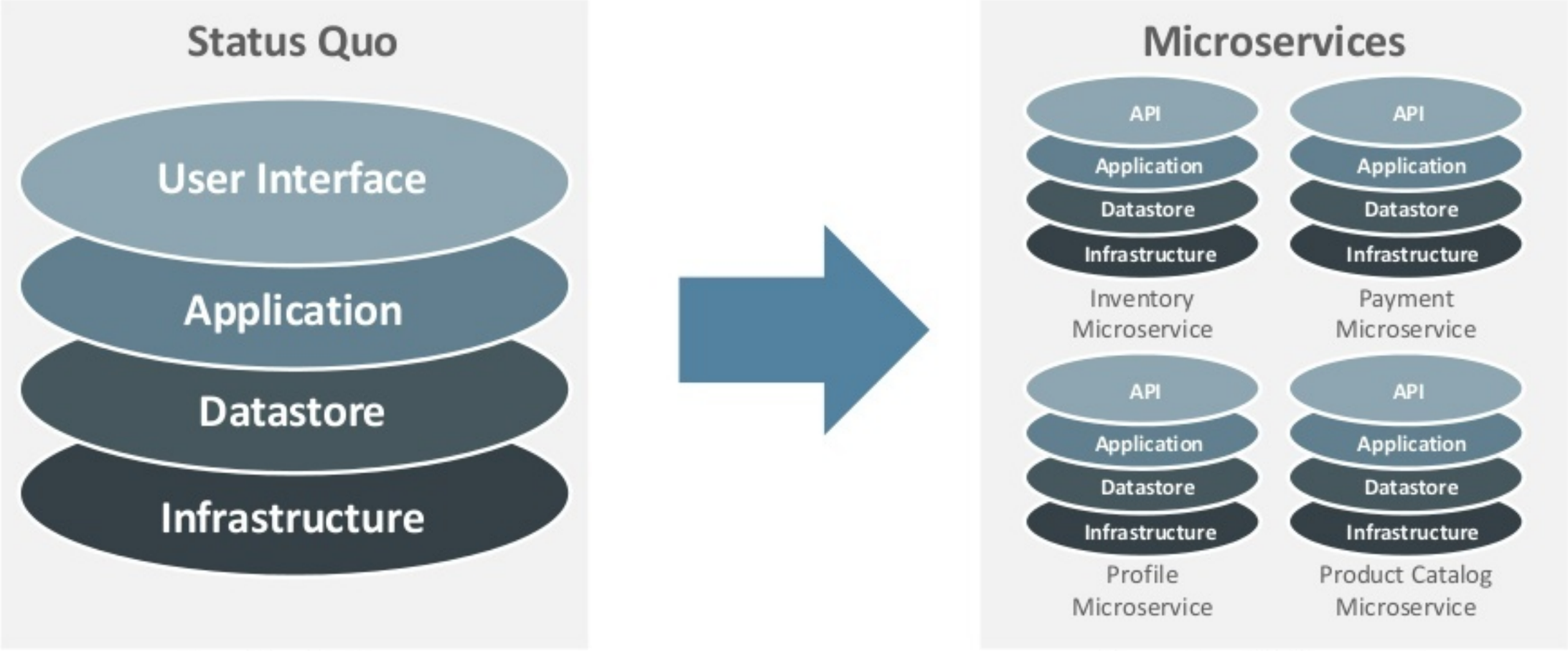
Doug McIlroy
Inventor of the Unix Pipe

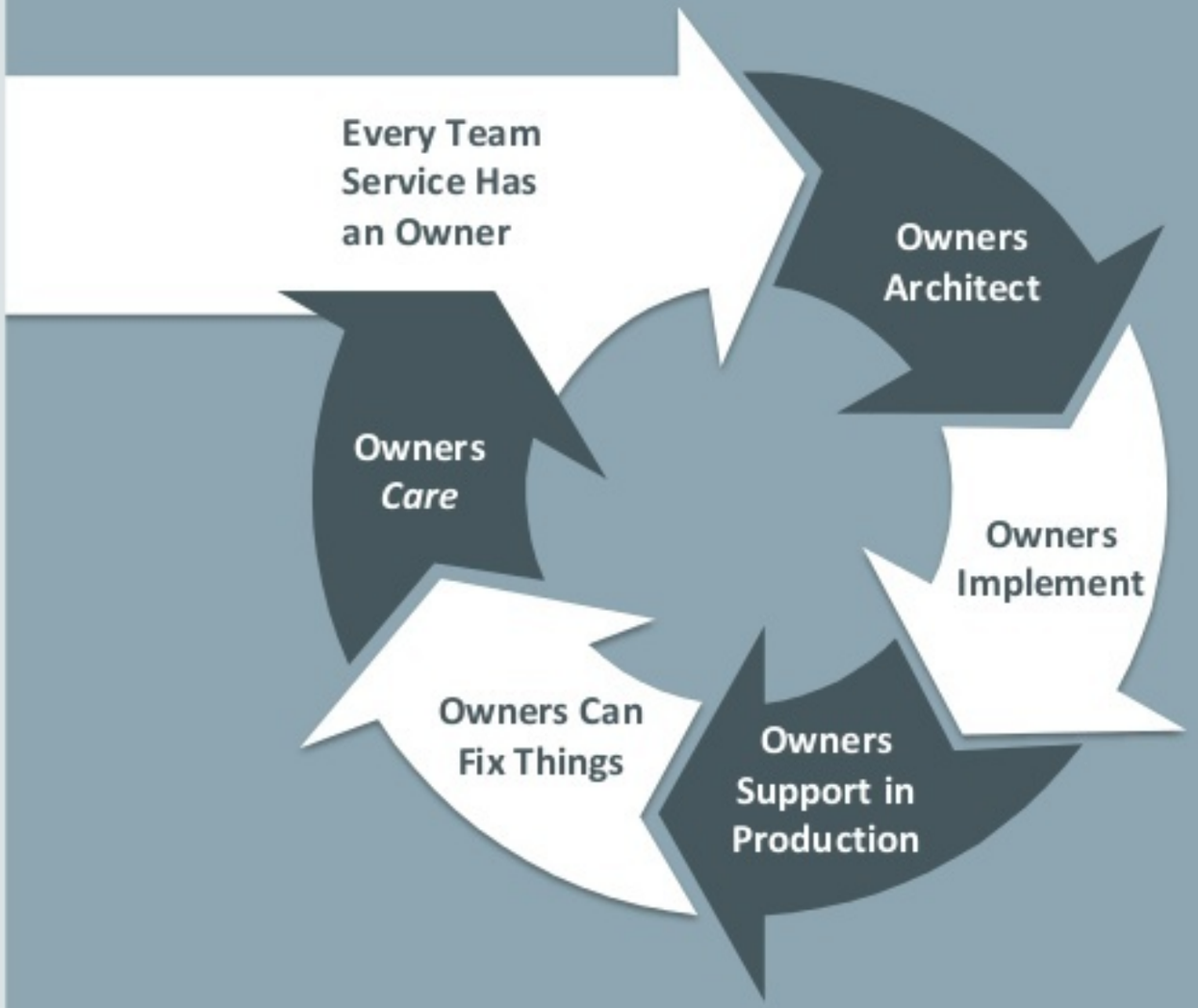
- Unix Executable: Does one thing and does it well
- Runs independent of other commands
- Produces text-based response

```
curl -v -H "Accept: application/json" -H "Content-type: application/json" -X POST
-d '{"productId":645887","quantity":"1"}'
"http://localhost:8840/rest/ShoppingCart/"
```

- Microservice: Does one thing and does it well
- Runs independent of other microservices
- Produces text-based response to clients

Microservices Apps Are Developed/Deployed Independently





Ownership is Key to the Success of Microservices

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

Fundamentally, Microservices is a Tradeoff

Do you want...

Traditional App Development

Easier Deployment/Ops

- One big block of code, sometimes broken into semi-porous modules
- Complexity handled inside the big block of code
- Each big block is hard to develop but easy to deploy

Microservices

Easier Development

- Many small blocks of code, each developed and deployed independently
- Complexity encapsulated in each microservice
- Each microservice is easy to develop but hard to deploy

Top 5 Signs It's Time To Look at Microservices

1. 100+ developers for an app
2. 5m lines of code for an app
3. Monthly or quarterly releases to production
4. > 1 quarter backlog of development work
5. > 20% developer turnover

Common Microservice Adoption Use Cases

I want to **extend** my existing monolithic application by adding microservices on the periphery.

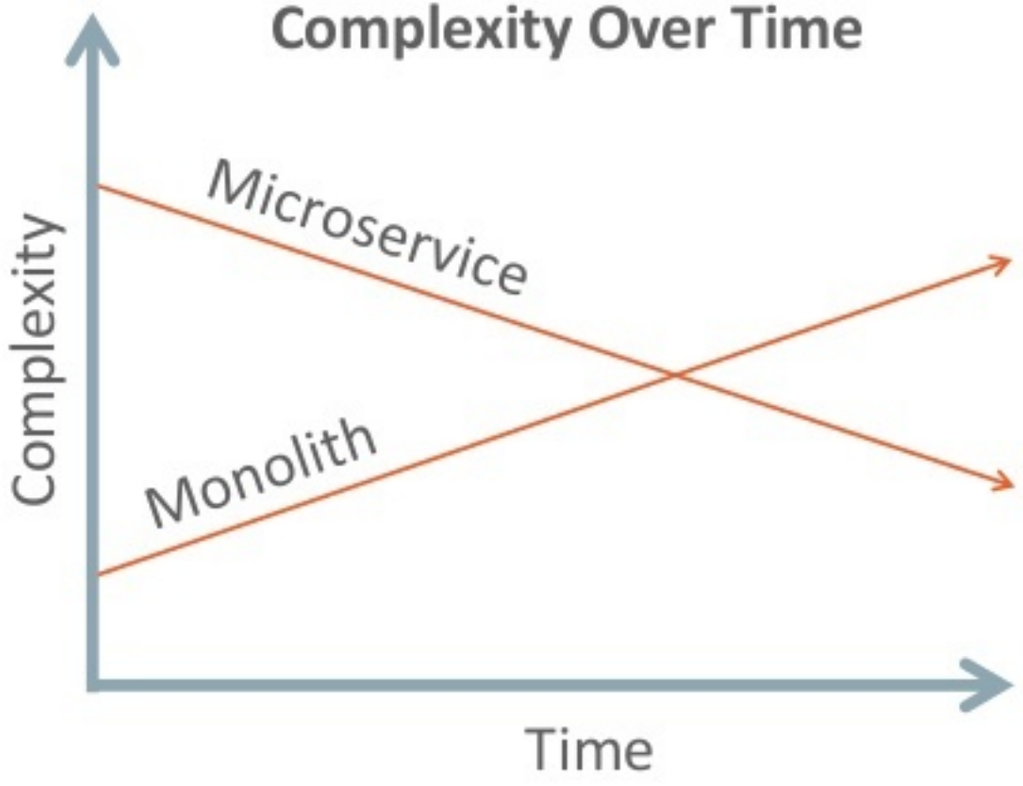
I want to **decompose** an existing modular application into a microservices-style application

I want to build a **net new** microservices-style application from the ground up.

Sometimes Monolithic Applications Are Still a Good Fit

Microservices add complexity

- For less complex applications, monoliths are always better in both the long and short-run
- For moderately complex applications, monoliths are still probably better in both the long and short-run
- For complex apps, microservices may pay off over time but it takes a long time to offset the high up-front investment required to do it



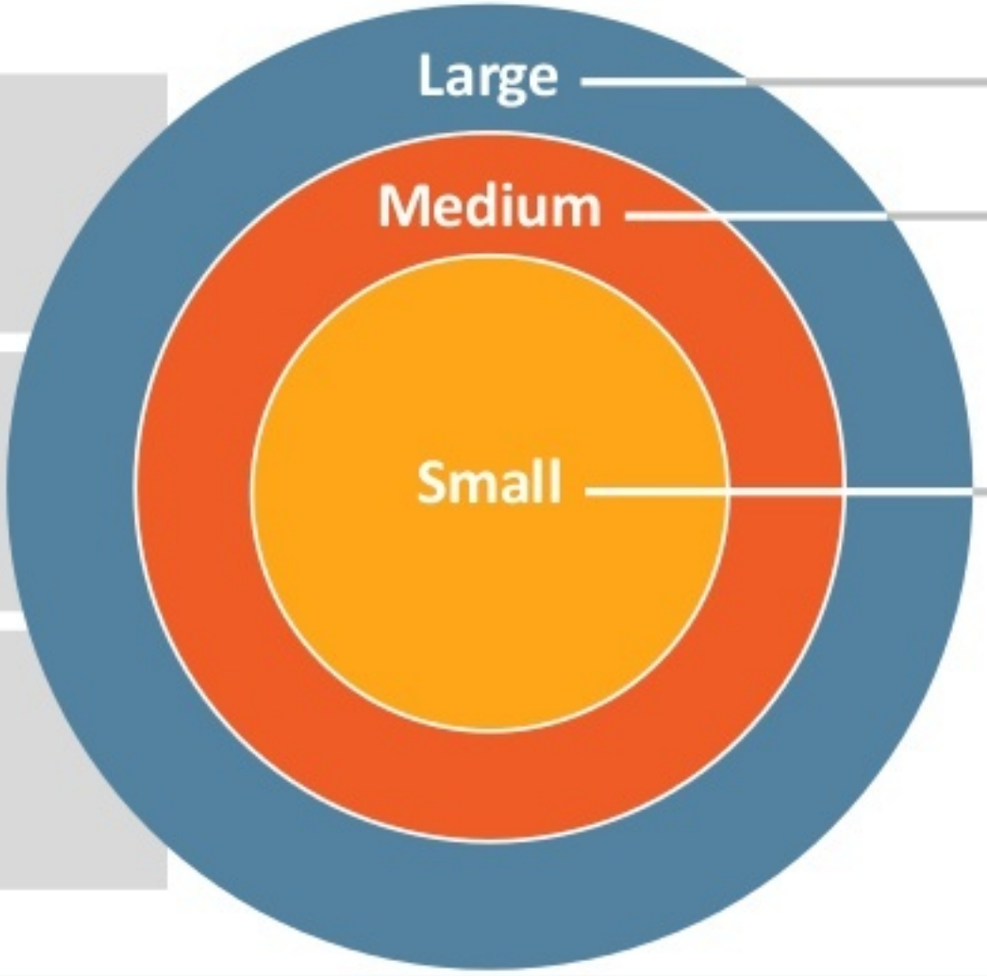
How Big Should a Microservice Be?

Can have hundreds of microservices for a larger application

Do One Thing and Do It Well

Avoid Inter-dependencies

Focus on Business Capabilities



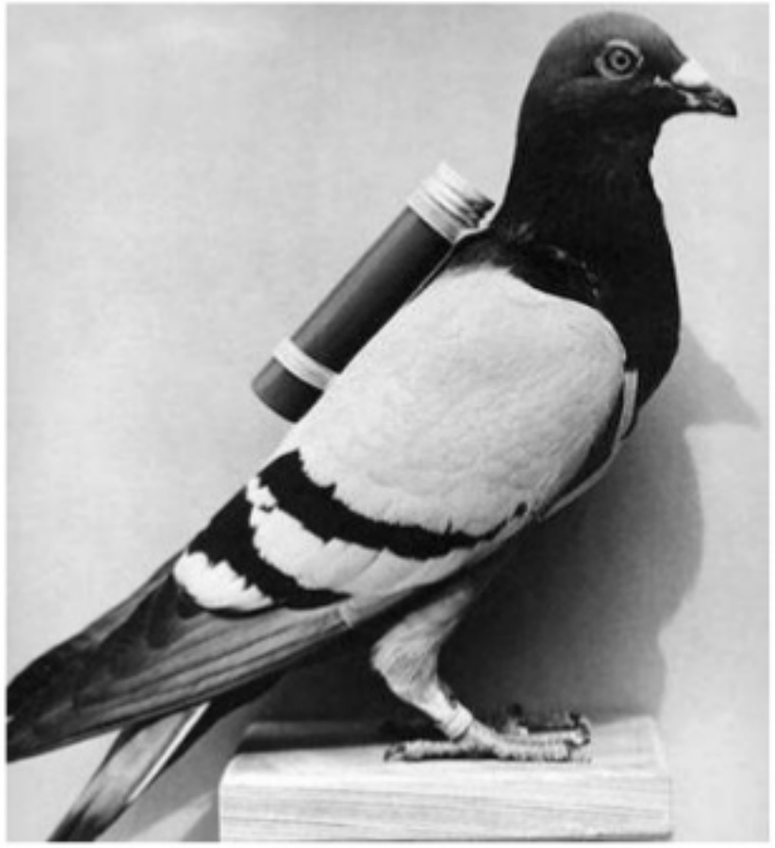
11-15 People
Example: Order Microservice

4-10 People
Example: Inventory Microservice

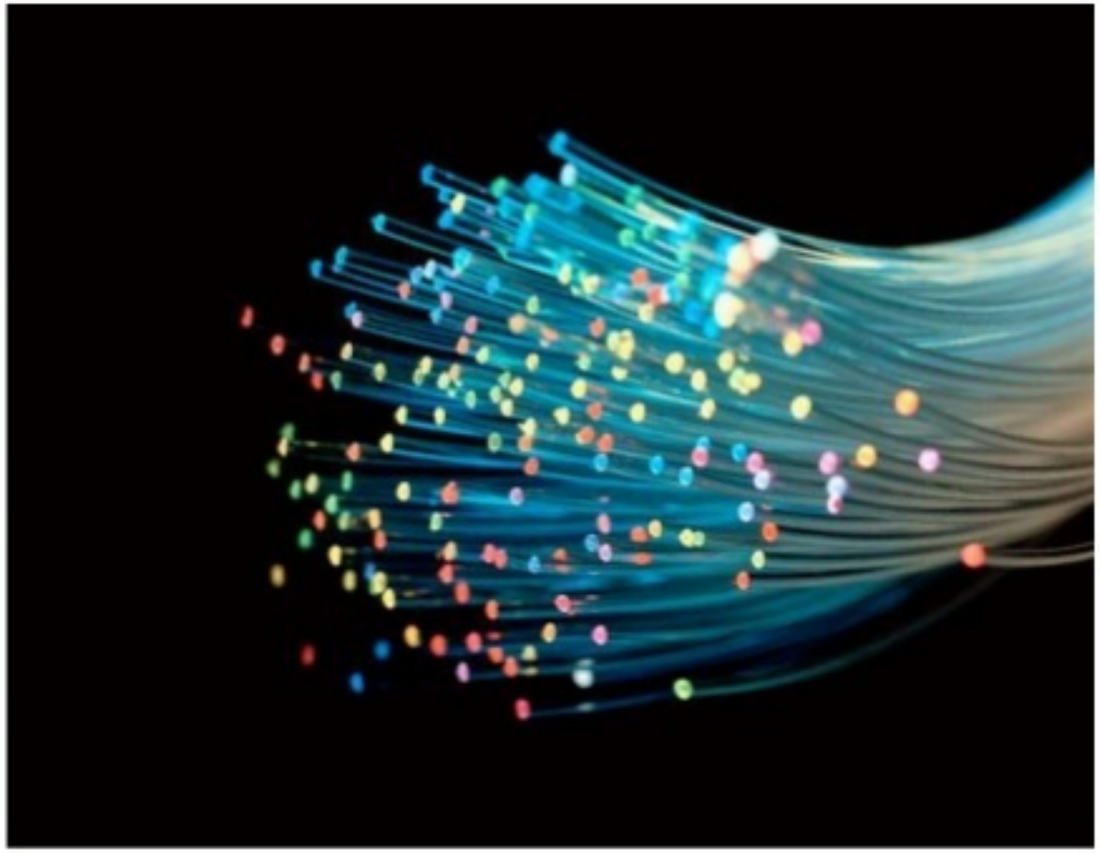
1-3 People
Example: Order Status Microservice

Small Teams = Much Better Communication

Low Latency/High Bandwidth Communication



Legacy

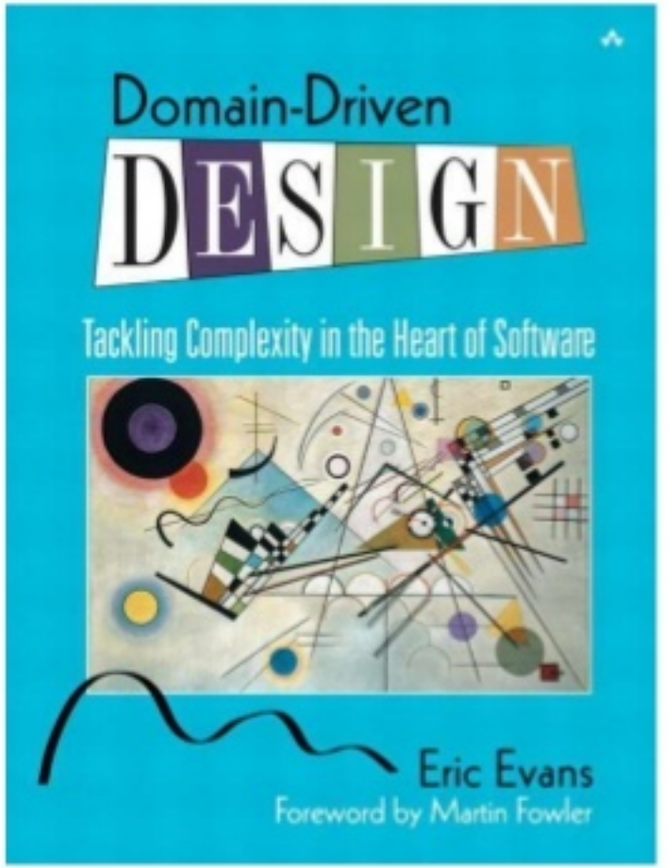


Microservices

Definitive Guide to Deciding Microservice Size

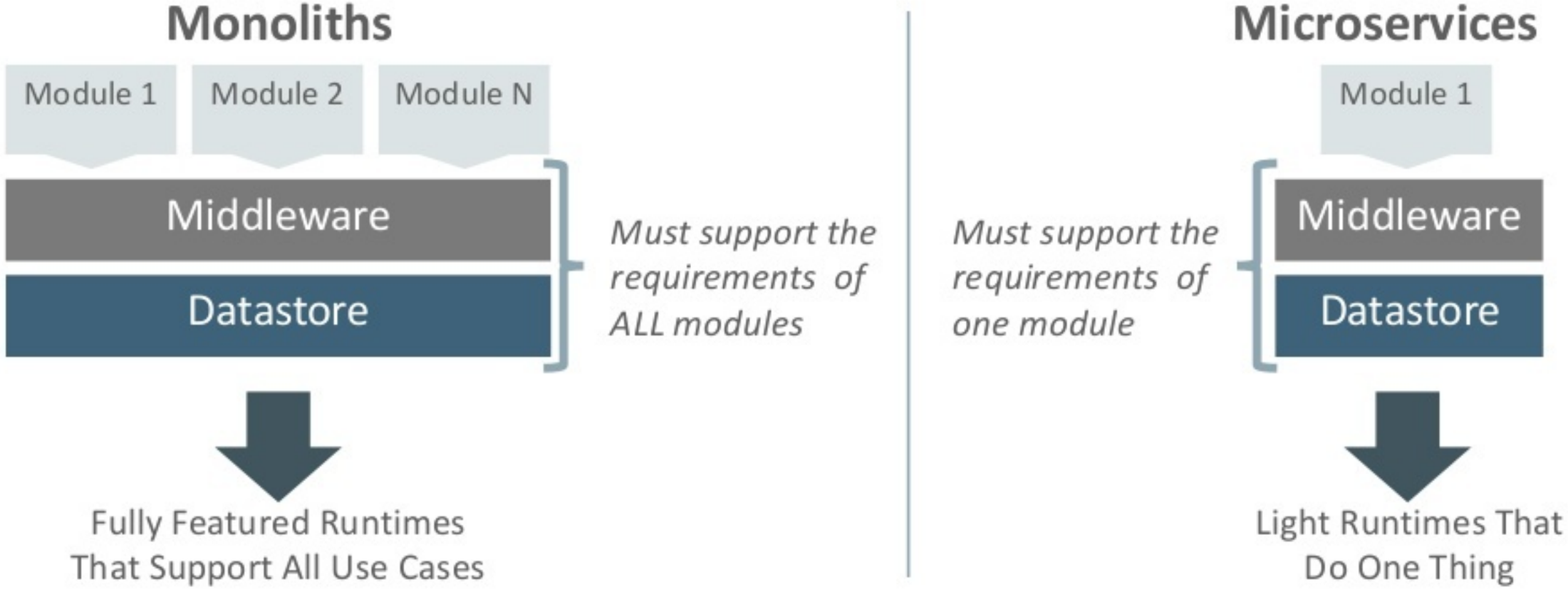
Domain-Driven Design: Tackling Complexity in the Heart of Software by Eric Evans

- Bounded Context is a central pattern in Domain-Driven Design
- Deals with designing software based on the underlying domain
- You can't build a big unified domain model for an entire system
- Divides a large system into Bounded Contexts, each of which can have a unified model



“Micro” in Microservices != Runtime Weight

Microservices *tend* to use smaller runtimes but you can use what you have today



You're Doing Microservices If You...

Microservices Are Not a Panacea

Technical Complexity

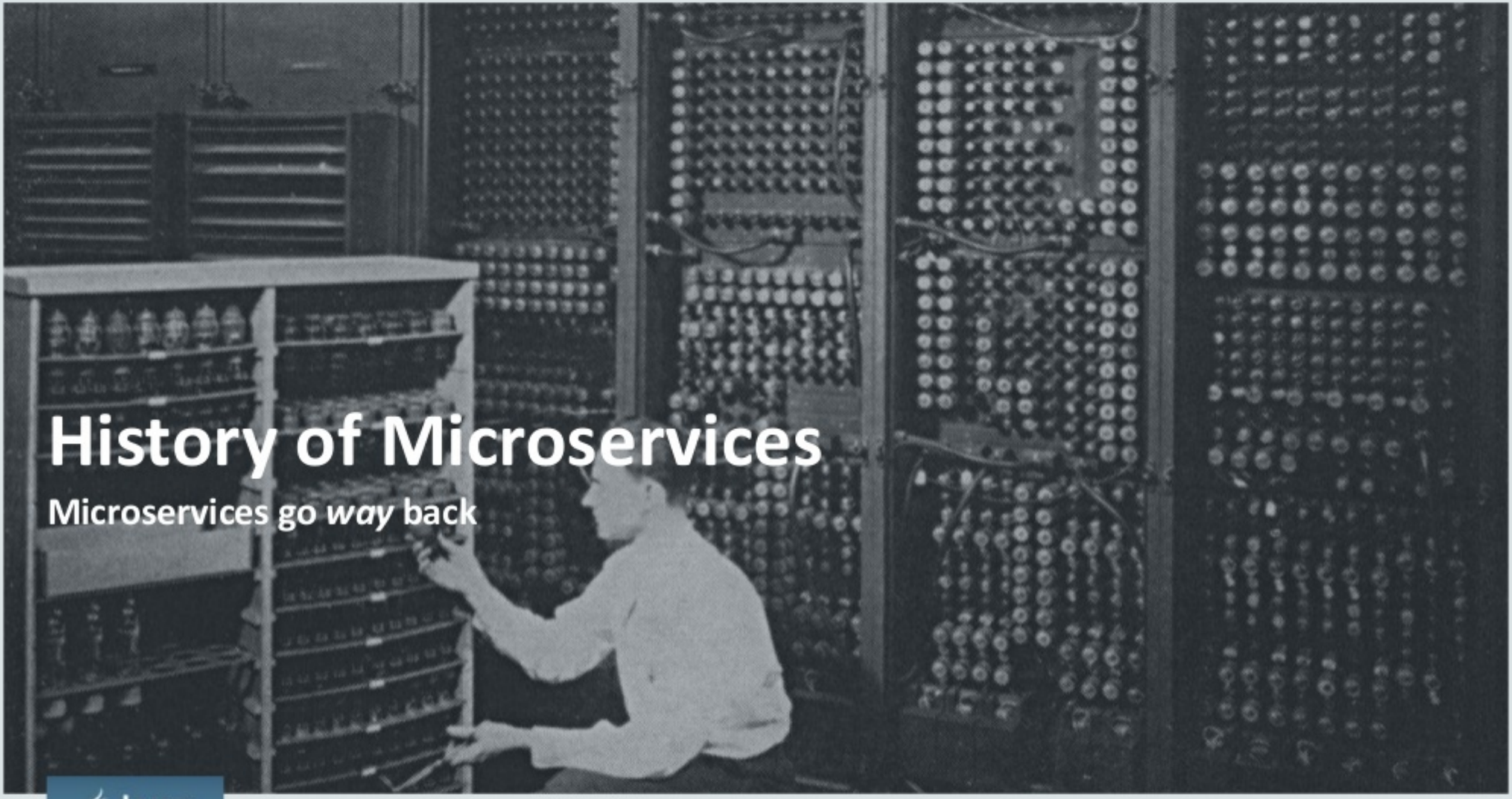
- Microservices doesn't eliminate complexity - it just moves it and often adds to it
- Monolithic applications allow you to deal with complexity in one body of code
- Forces move to distributed computing

Effort

- Testing, logging, monitoring, security, versioning, etc all become much harder
- Polyglot exponentially increases the number of lifecycles required
- A lot of duplicated effort since each team is independent and goal is to minimize dependencies

Organization

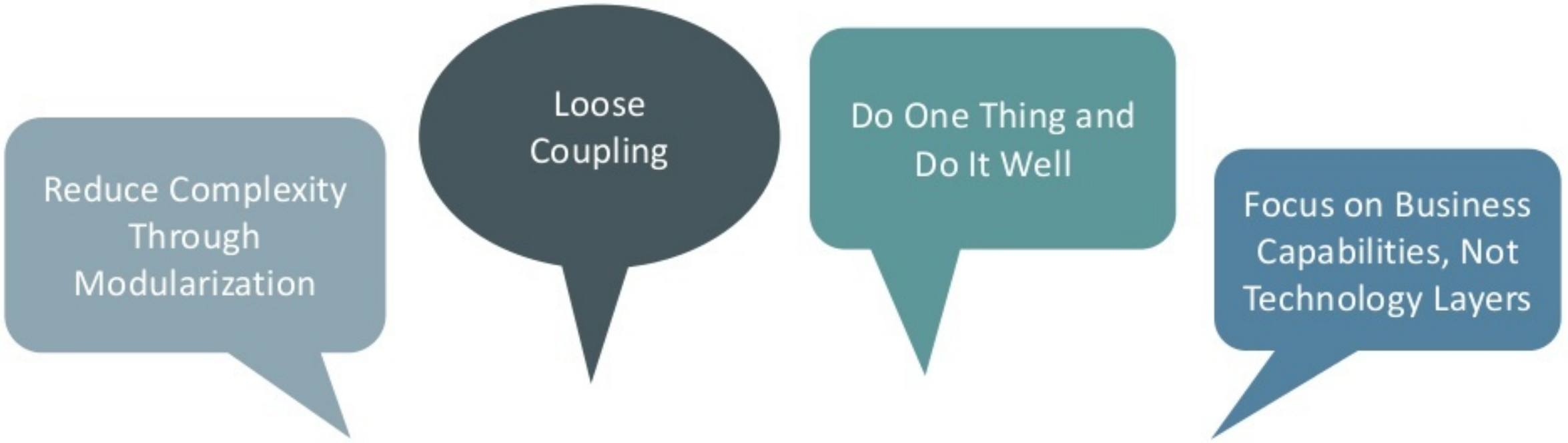
- Most organizations are organized around horizontal technology layers – need to build small product-focused teams
- Much higher skills required
- Many developers will not want to do production support



History of Microservices

Microservices go *way* back

Microservice Principles Have Been With Us For Decades

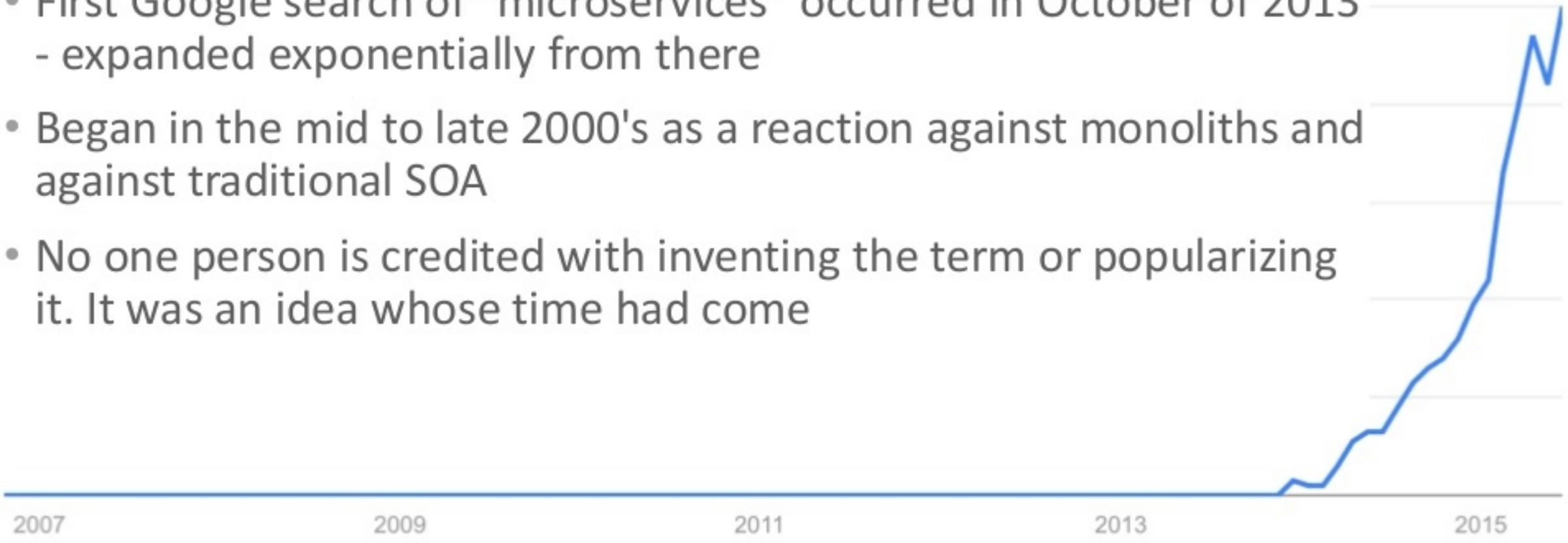


The principles behind microservices are often just good architecture principles

History of Microservices as a Term

A trend without a name, until “microservices” was used in 2013

- First Google search of "microservices" occurred in October of 2013 - expanded exponentially from there
- Began in the mid to late 2000's as a reaction against monoliths and against traditional SOA
- No one person is credited with inventing the term or popularizing it. It was an idea whose time had come



SOA vs. Microservices

SOA is the general idea, where microservices are a very specific way of achieving it

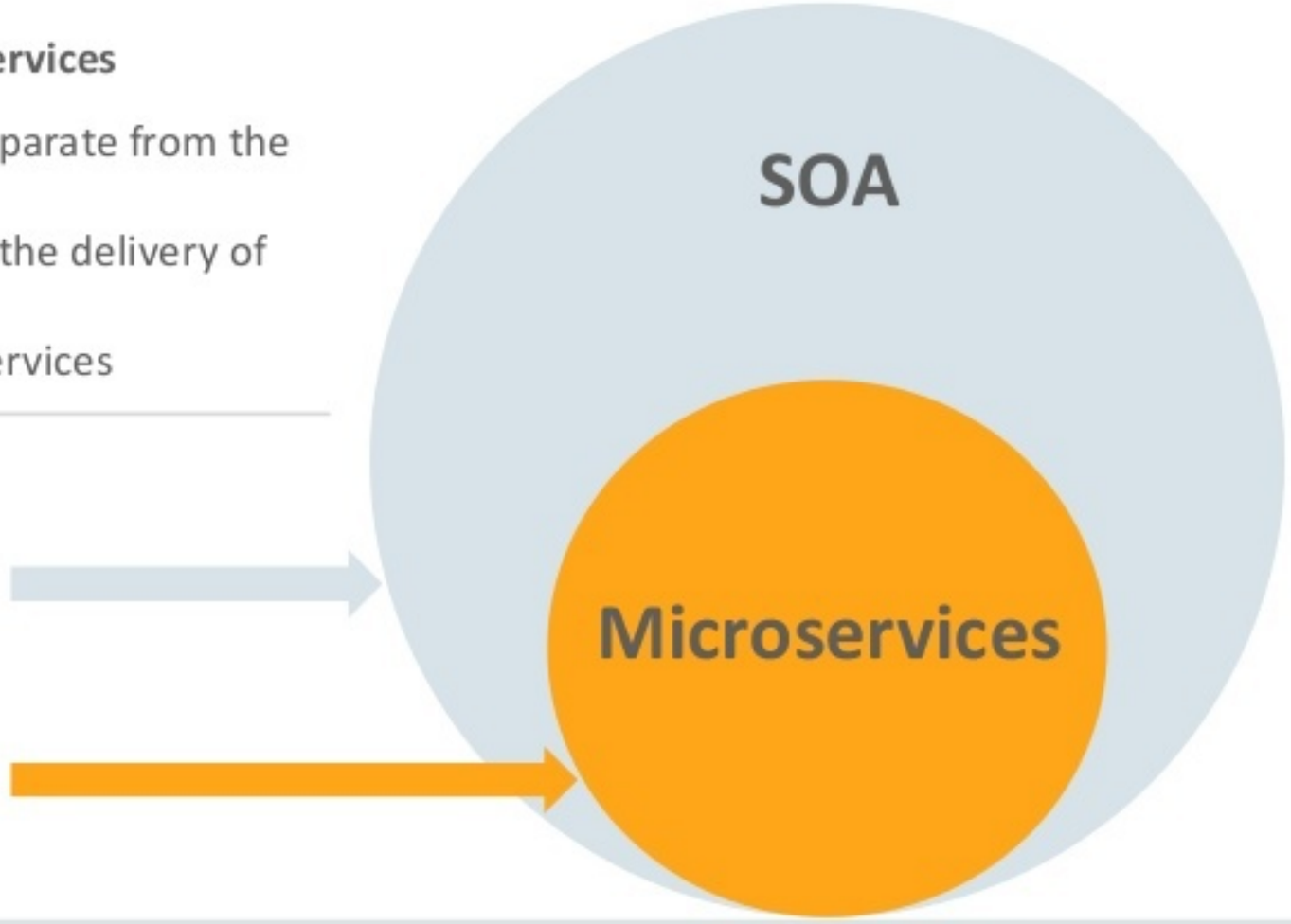
All of the tenets of SOA also apply to microservices

1. Keeping consumption of services separate from the provisioning of services
2. Separating infra management from the delivery of application capability
3. Separating teams and decoupling services

Implementation Differences

- Favors centralized orchestration
- Needlessly complicated by SOAP
- "Dumb endpoints, smart pipes"

- Favors distributed choreography
- REST + HTTP/S = simple
- "Smart endpoints, dumb pipes"



SOA vs. Microservices Misconceptions

"Microservices removes the need for an Enterprise Service Bus"

Don't confuse the product with the pattern

"Microservices solves the problems of SOA"

Don't confuse improper SOA deployments as problems with SOA

"Companies like Netflix and LinkedIn use microservices, so we should too"

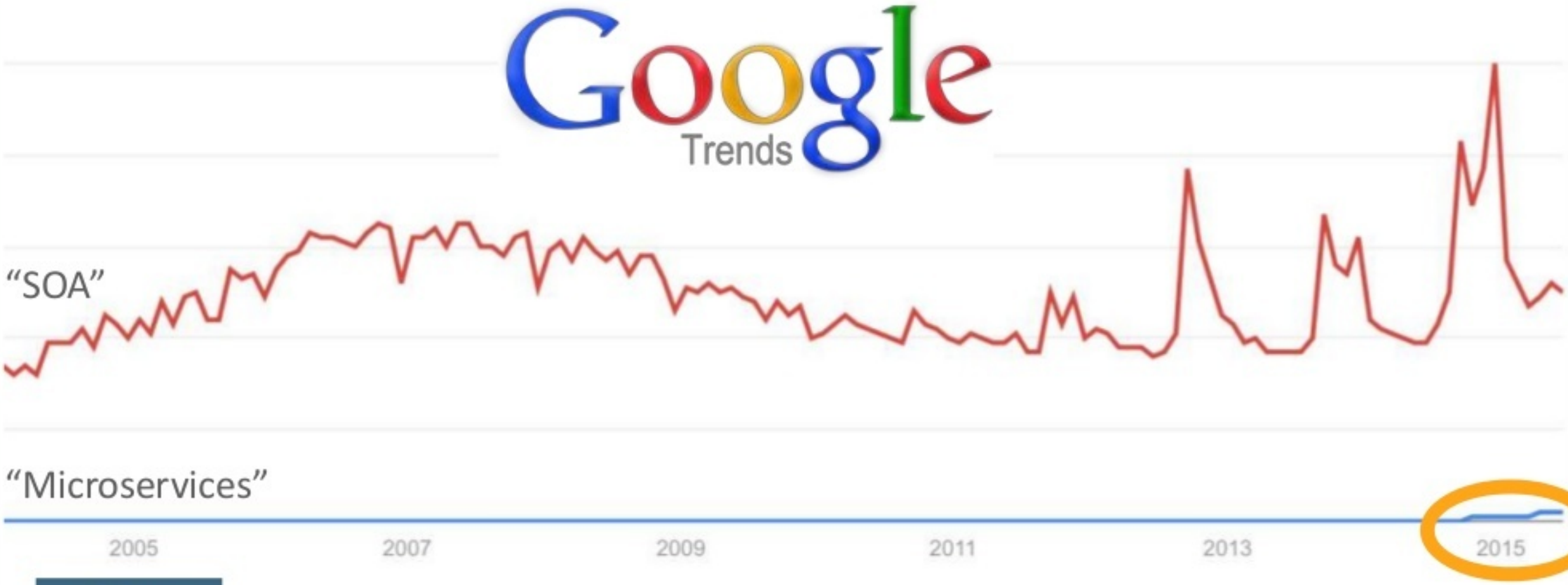
Netflix and LinkedIn are in the platform business. Is that your business too?

"We must choose microservices, or SOA"

Use both

SOA Still Reigns Supreme

SOA isn't going anywhere



Microservice Principles Are Old; Implementation is New

Microservices is not just rebranded SOA

- Teams independently architect, develop, deploy and maintain each microservice
- Each microservice often has its own datastore, which may not always be 100% up-to-date
- Microservices is fully decentralized - no ESBs, no single database, no top-down anything
- Responses from microservices are not manipulated by an intermediary, like an ESB
- Microservices favors simple transports - XML or JSON over HTTP/S. No SOAP
- Any given instance of a microservice is stateless - state, config, and data pushed externally
- Microservices support polyglot - each microservice team is free to pick the best technology
- DevOps principles - automated setup and developers owning production support
- Use of containers, which allow for simple app packaging and fast startup time
- Use of cloud, for the elastic infrastructure, platform and software services

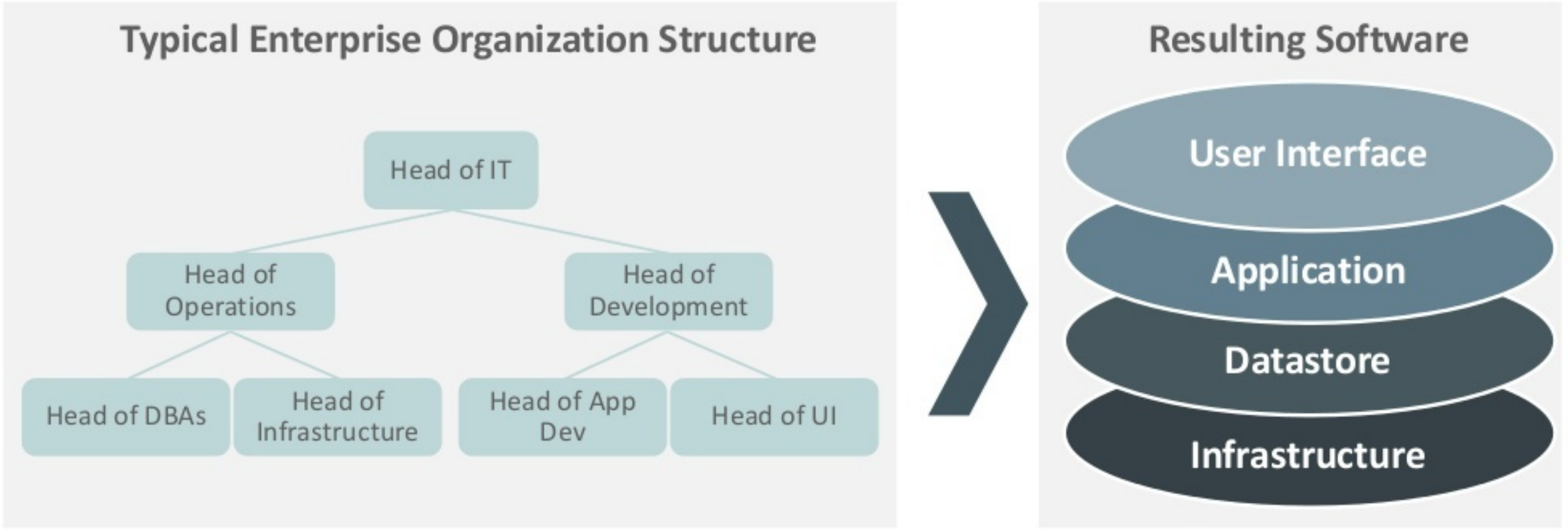
Conway's Law

"Any piece of software ***reflects the organizational structure*** *that produced it"*

Melvin Conway
1968

Conway's Law In Action

Any piece of software reflects the organizational structure that produced it



An Enormous Monolith

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

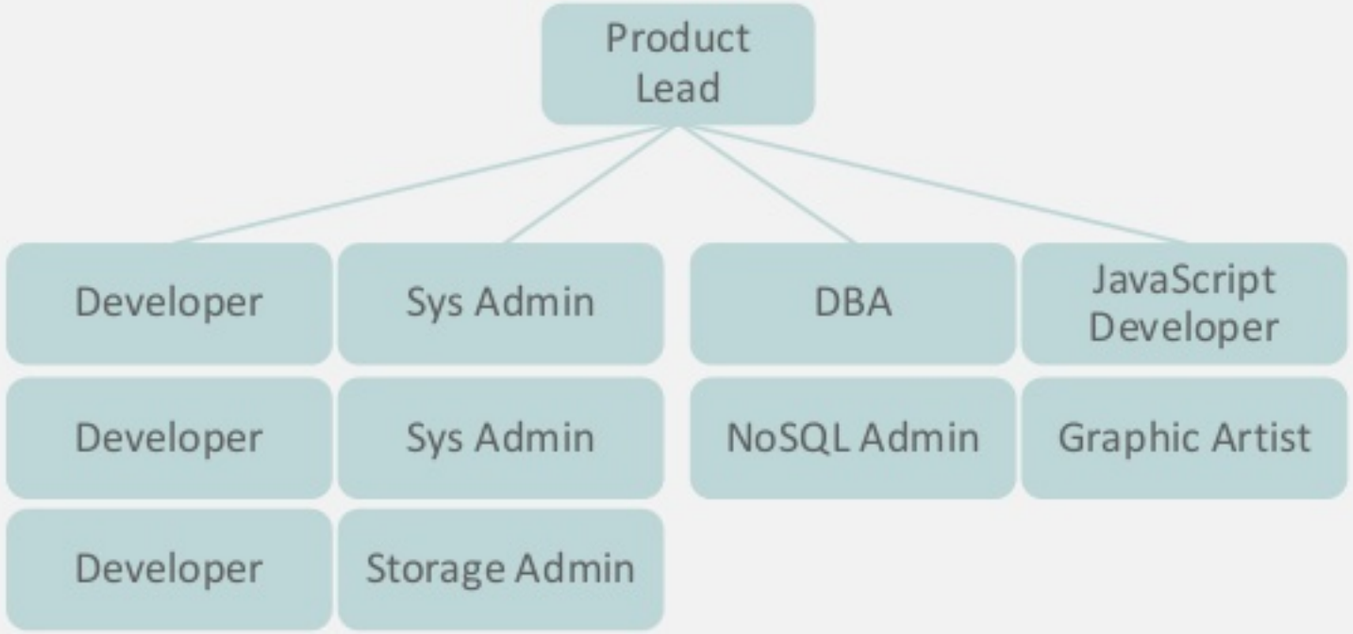
Distributed Organizations
Focused on Products

- Produce microservices
- Small teams can make any change they want to an individual microservice
- Architecture clean because developers have 100% control
- True ownership

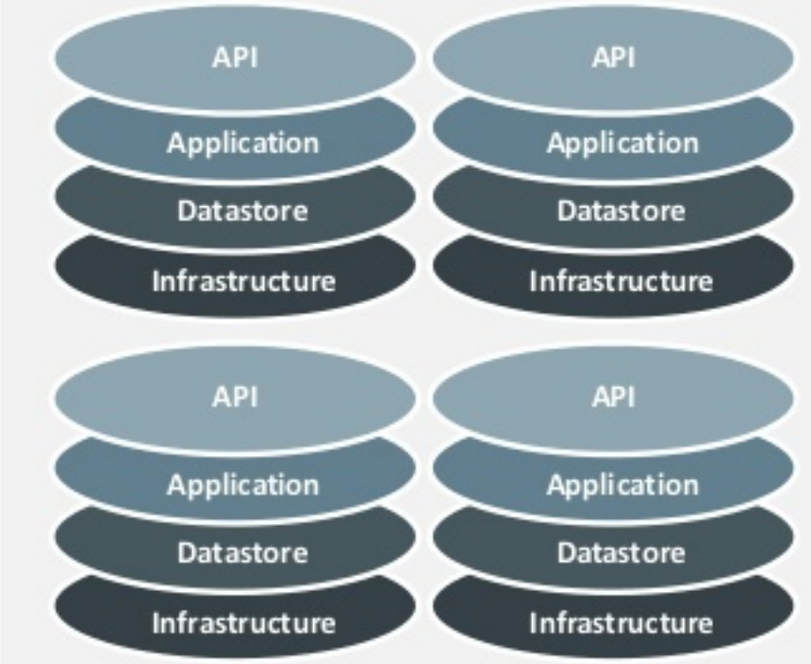
Successful Teams Structure Their Teams Around Products

Build small vertical teams

Microservices Organization Structure



Resulting Software

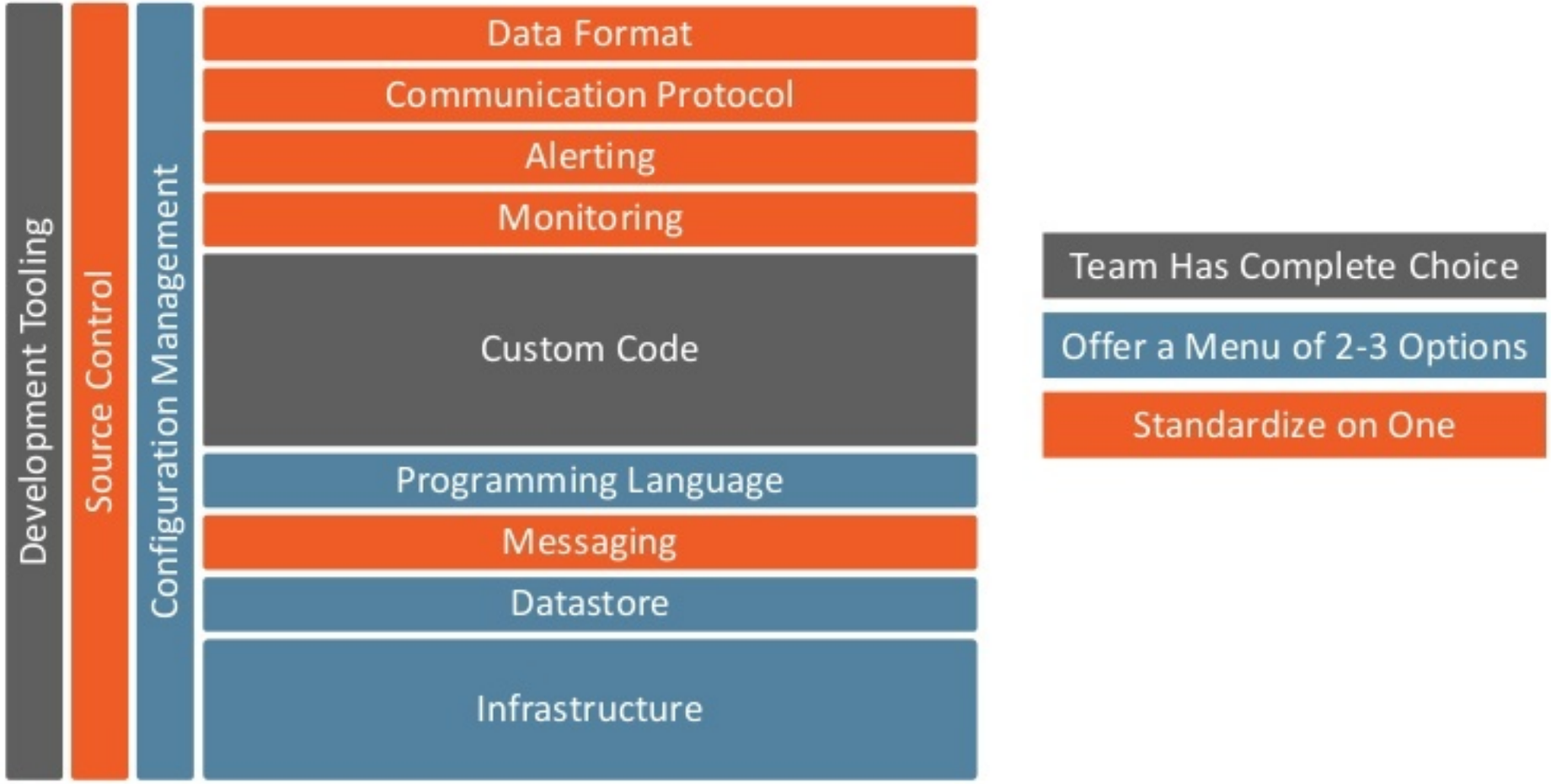


Many Small Microservices

Horizontal -> Vertical Teams

Standardize Where it Makes Sense. Be Pragmatic

Teams do not have 100% freedom



Each Team Should Treat Other Teams as External Vendors

Good fences make good neighbors

- Clear interfaces
- Clear SLAs
- Chargeback

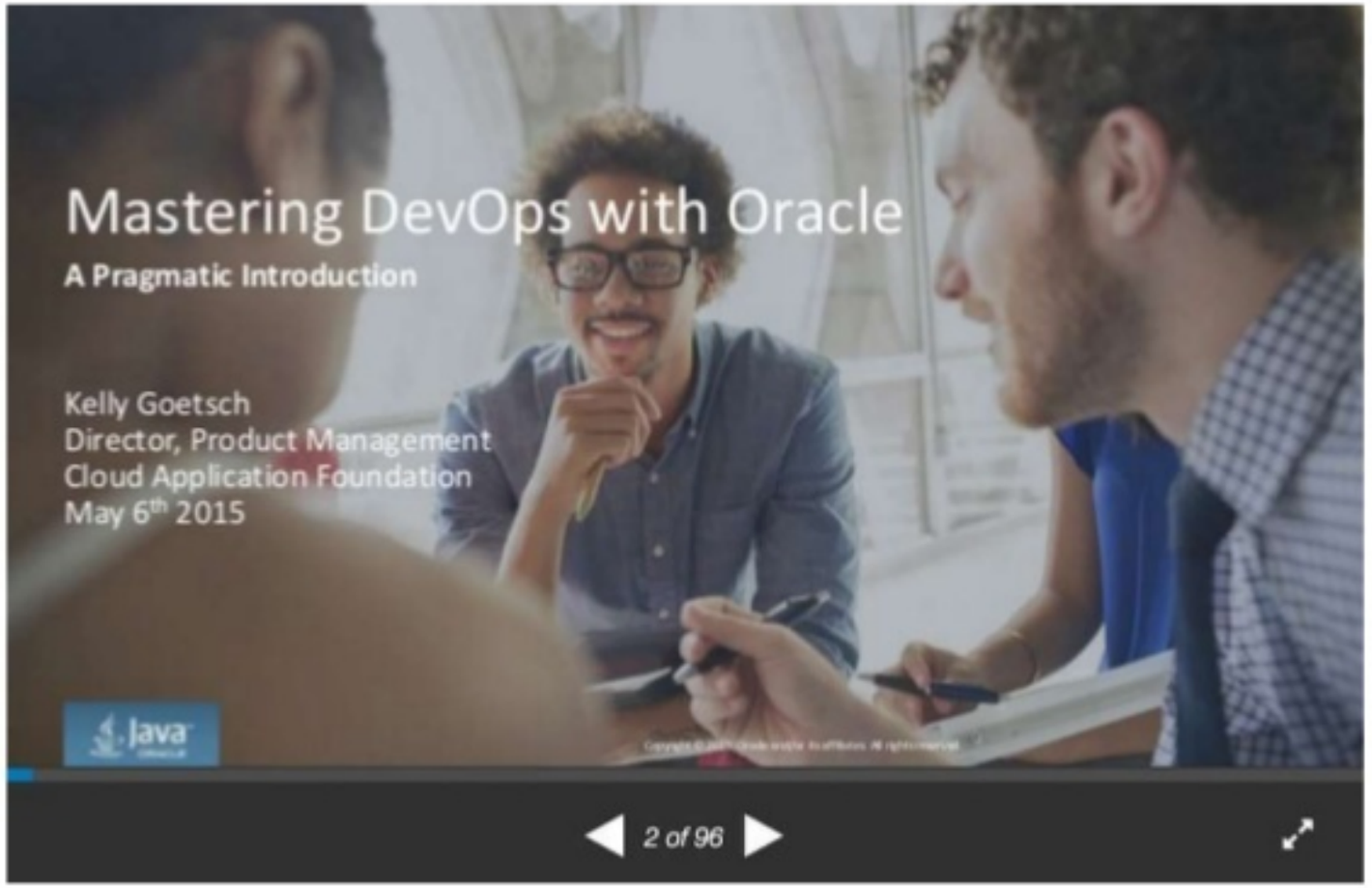
DevOps Must Also Be Adopted Prior to Microservices

Mostly a function of culture

Culture		Technology	
Respect	• Dev respect for ops • Ops respect for dev	Infra as Code	• Don't build envs by hand • Scripts checked in and managed as src
Discuss	• Ops should be in dev discussions • Dev should be in ops discussions • Shared runbooks	Shared Version Control	• Single system • Ship trunk • Enable features through flags
Avoid Blaming	• No fingerpointing! • Everyone should have some culpability	One Step Build/Deploy	• One button build/deploy • If verification fails, stop and alert or take action
"Done" Means Released	• Dev's responsibility shouldn't ever end – production support required • "Throwing it over the wall" is dead	Don't Fix Anything	• If something breaks, re-deploy. Don't fix • Fix environment setup scripts

More About DevOps

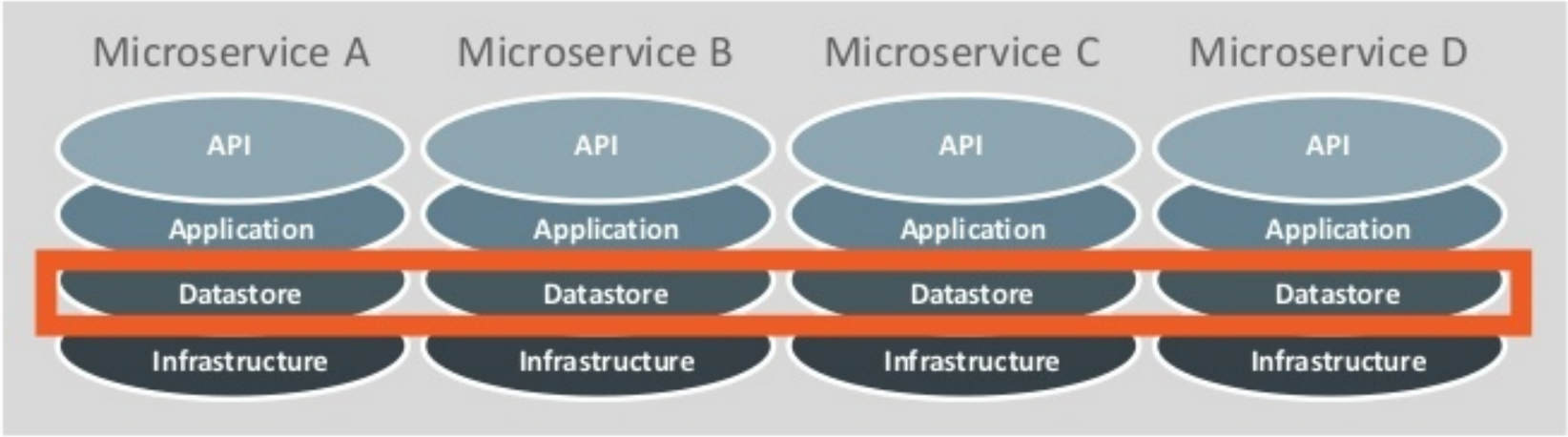
http://www.slideshare.net/KellyGoetsch/mastering-devops-with-oracle



Architectural Prerequisites

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 between microservices. REST + HTTP probably isn't fast enough
- May end up duplicating data across datastores – e.g. a customer's profile

Rules of Distributed Computing

Computer science is about tradeoffs

Theory

CAP Theorem – Pick Any Two

Consistency
Each node shows the same data at all times

Availability
Each node is available for writes at all times

Partition Tolerance
Able to handle network outages

More Information

Practice

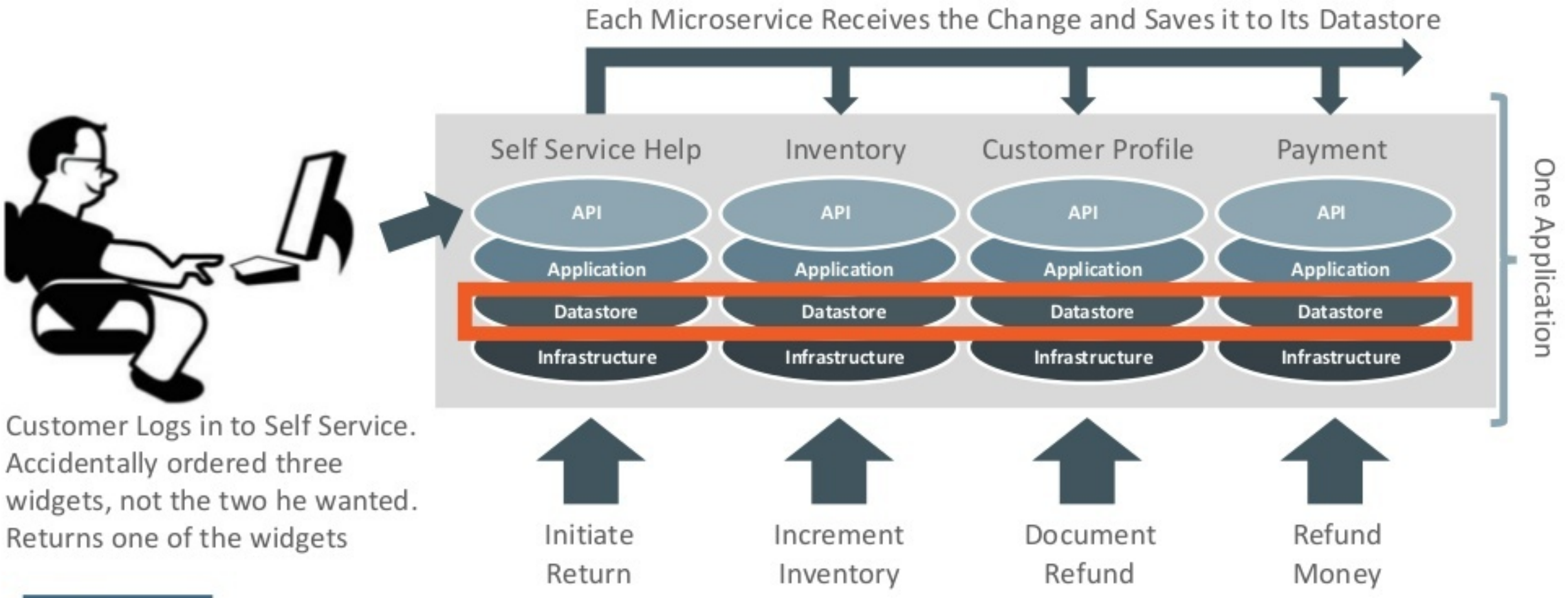
Pick One
Partition Tolerance is non-negotiable because we have networks that can always fail

Enterprise IT Systems: Often CP
Microservice Systems: Often AP

Each microservice can be CP, AP or CA but the system as a whole is always CP or AP

Microservices Usually Forces Eventual Consistency

Synchronous communication leads to availability and performance problems



Synchronous Calls With Microservices Are Very Bad

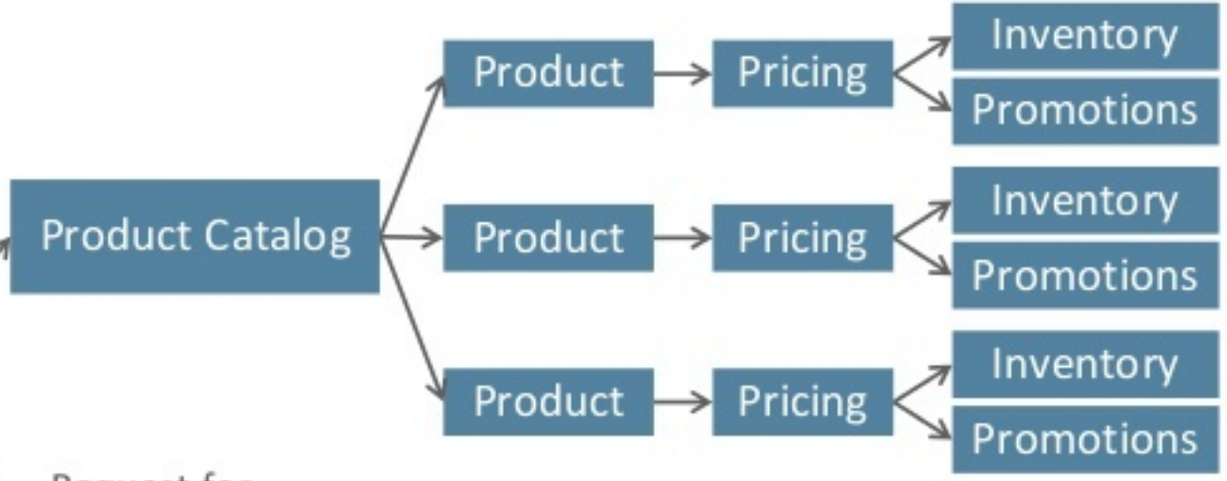
Chaining == coupling == downtime

The availability of microservice A depends on B, B depends on C, etc

Avoid Synchronous Calls for Read-only Data By Copying

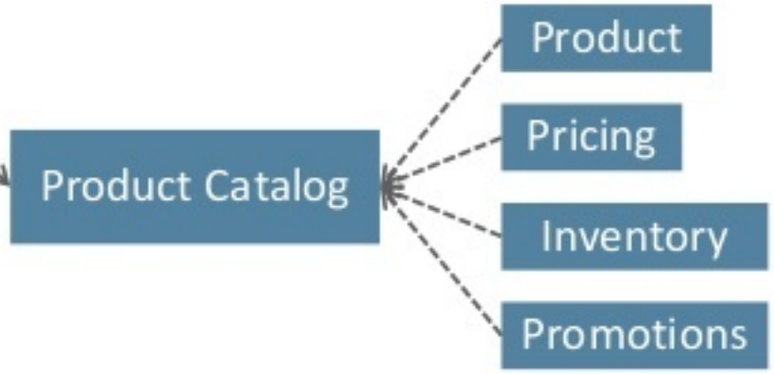
Traditional Monoliths

Request for Category Page



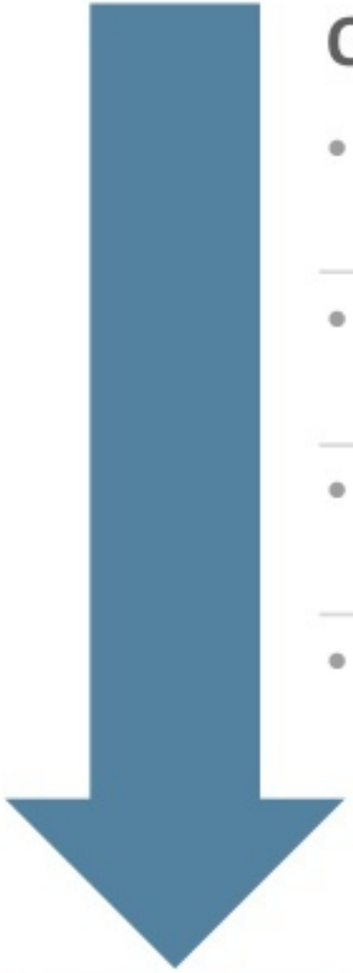
- Synchronous in-memory calls
- Data is 100% consistent
- No data is duplicated

New Microservices



- Synchronous calls to product catalog microservice
- Product, pricing, inventory and promotions microservices are systems of record; they publish asynchronously to Product Catalog microservice when updated
- Product Catalog microservice is not always consistent
- Data is duplicated – two or more microservices may contain the same data

Choreography Tends to Be Better Than Orchestration

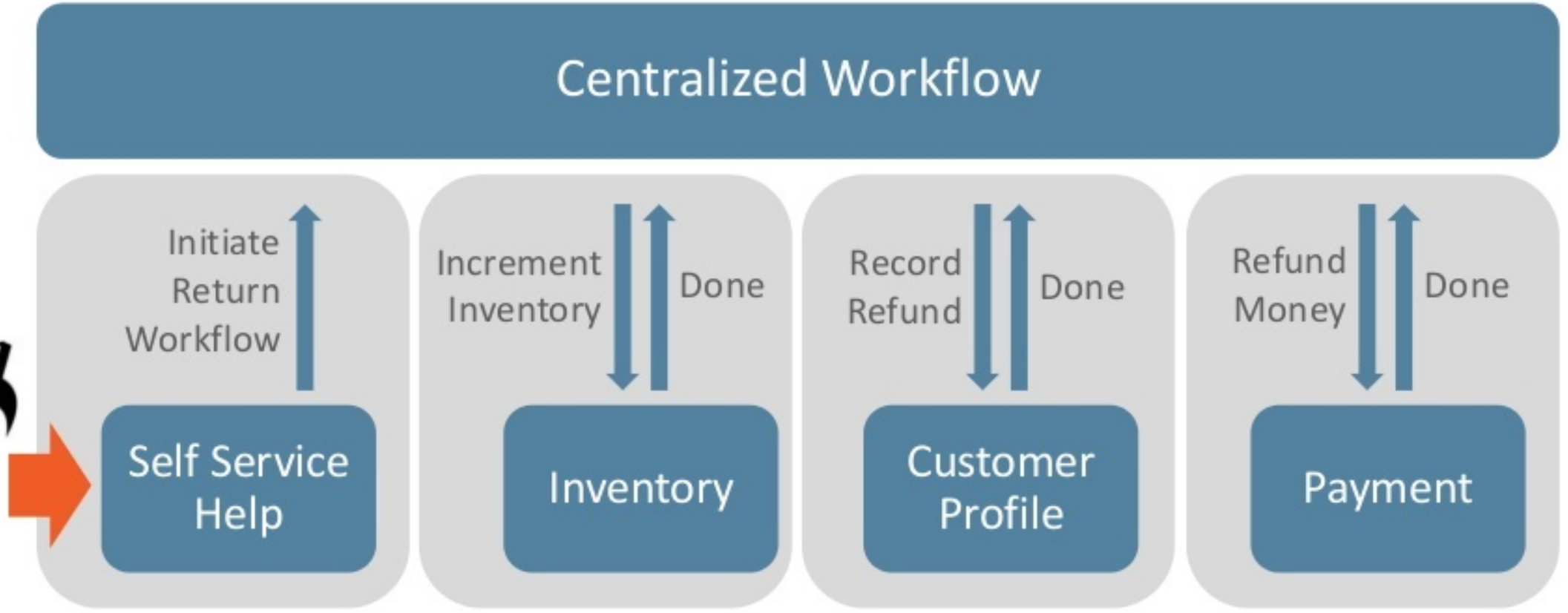


Orchestration	Choreography
• Top-down coordination of discrete actions	• Bottom-up coordination of discrete actions
• Used in centralized, monolithic applications	• Used in distributed, microservice applications
• Brittle – centralized by nature	• Resilient – distributed by nature
• Each “action” registers with centralized system – single point of failure that is not very flexible	• Each microservice asynchronously throws up a message that other microservices can consume

Orchestration vs. Choreography

Example of Orchestration

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



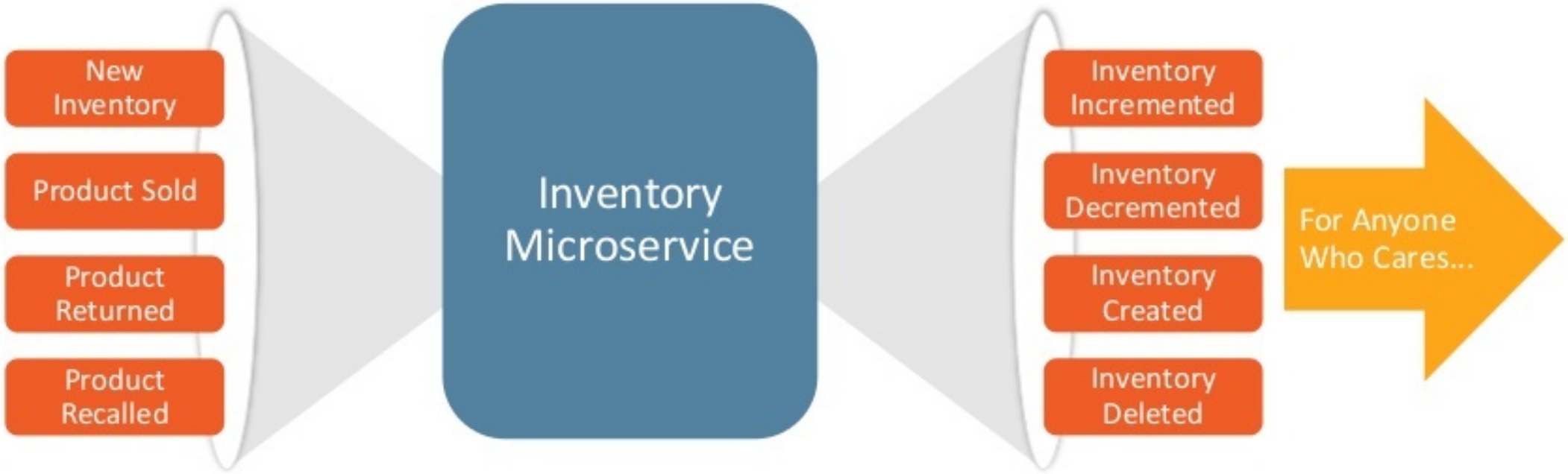
Brittle | Centralized | Tightly Coupled

Example of Choreography

Scenario: Inventory microservice

Events This Microservice Cares About

Events This Microservice Emits



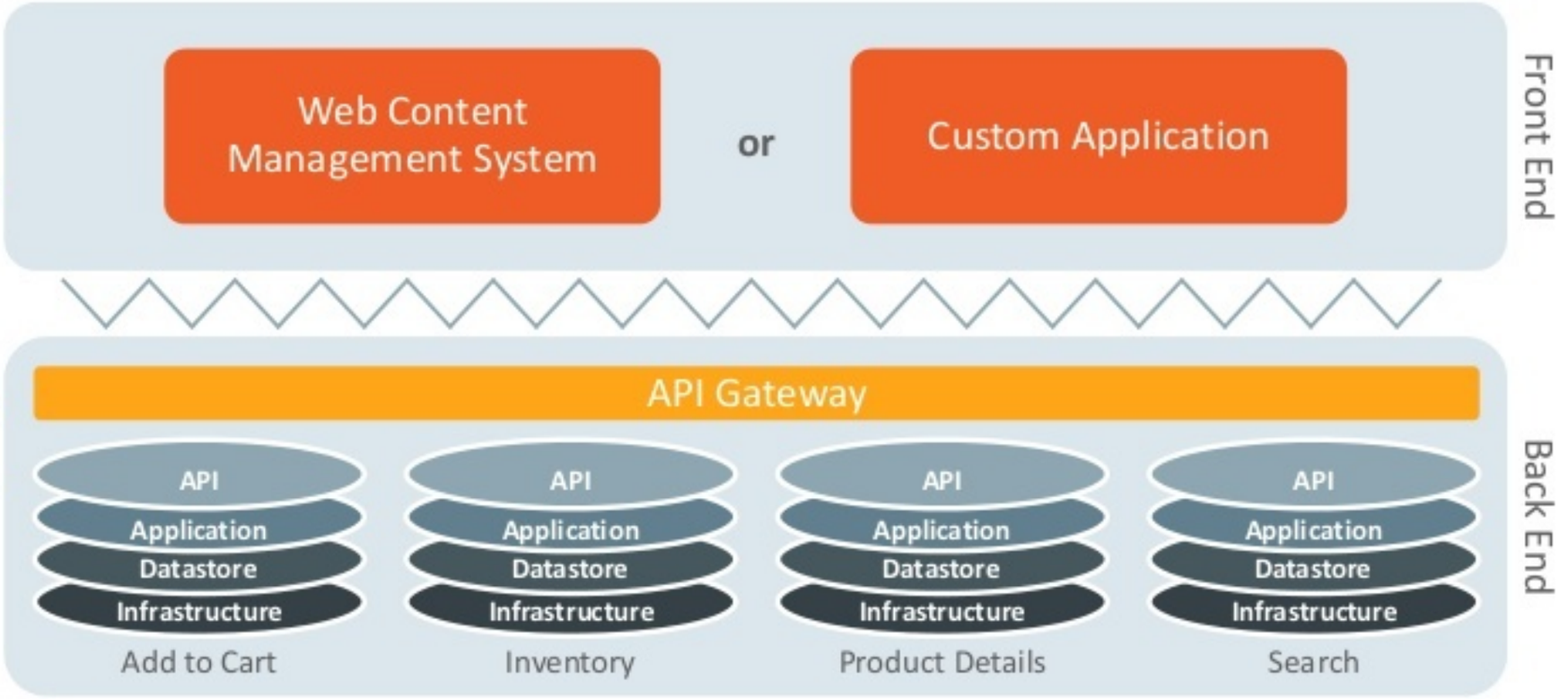
All asynchronous

Best to Ship Your Applications Headless

Put your front and back ends in different clouds, different geographies

Very Different Requirements

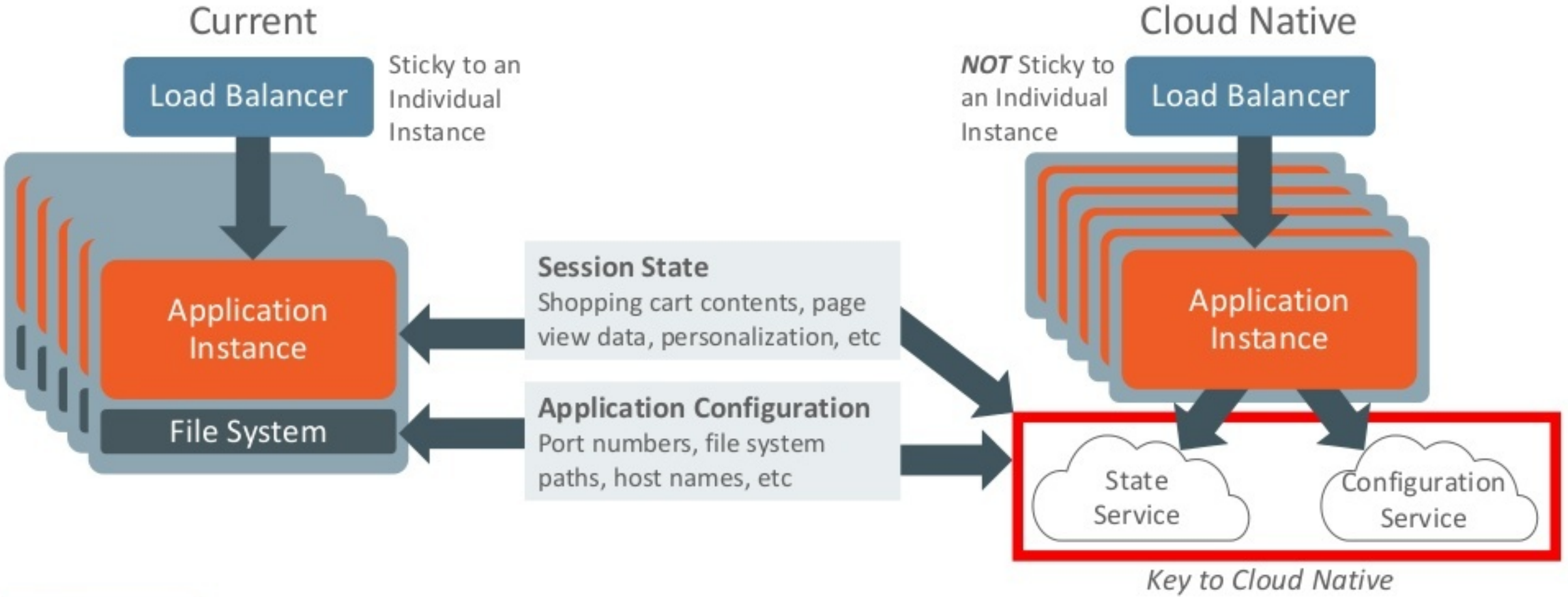
- Security
- Elasticity
- Performance
- Traffic patterns



Design, develop, deploy and manage your front and back ends differently

Make Your Middle Tier Stateless If You Can

Push all state and configuration down to highly available cloud services



Remove All Hard-coded IPs, Host Names, etc

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

Modify Messaging To Be Less Fragile

Messages should be constructed to be able to be applied over and over

Not Idempotent

```
<credit>
   <amount>100</amount>
   <forAccount>1234</account>
</credit>
```

- Can execute exactly once
- Works fine in a monolith
- Will break with microservices

Best For Microservices

Idempotent

```
<credit>
   <amount>100</amount>
   <forAccount>1234</account>
   <creditMemoID>4567</creditMemoId>
</credit>
```

- Can execute many times
- Works fine in monolith and microservices
- Allows the pipes to be dumb – message just needs to be applied once

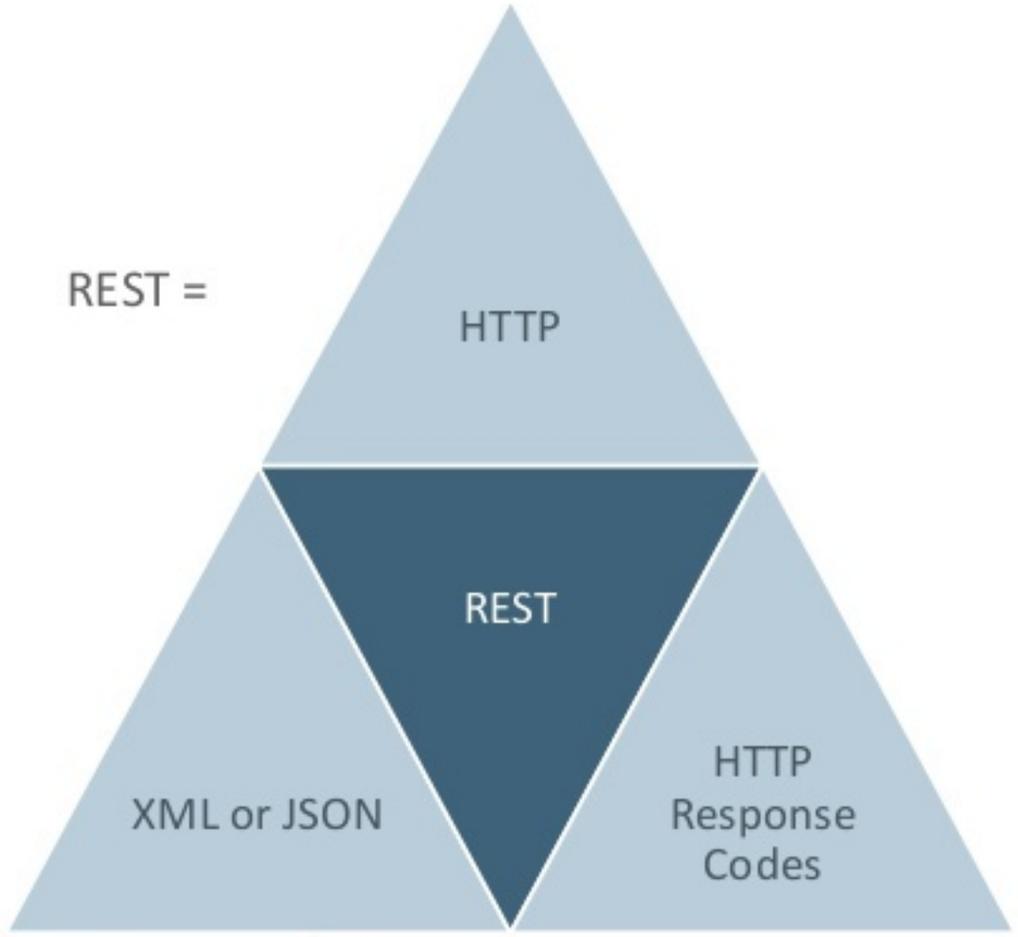
* Idempotent = the outcome doesn't change after the first application

Approaches to Synchronous Network Calls

	XML/JSON Over HTTP	Binary Over Wire
Primary Use	Communicating with clients over the public internet	Communicating with other microservices over a private network
Pros	▪ Universally understood format ▪ Easy to implement and understand	▪ Very fast
Cons	▪ Slow since it's text-based	▪ Can be hard to implement
Implementations	▪ No special software required – natively supported by all major programming languages ▪ HTTP is the language of the web!	▪ Oracle Portable Object Format ▪ Google Protocol Buffers ▪ Apache Avro ▪ Apache Thrift

REST: **Re**presentational **S**tate **T**ransfer

Strongly associated with microservices but not a technical requirement



- Much simpler alternative to SOAP
- Uses GET, POST, PUT, DELETE, etc – just like web browsers do
- Synchronous inter-microservice communication often occurs over binary
- Can version APIs - /v1.2/customer
- Can use XML or JSON
  - XML is often better - supports XPath, CSS selectors
- Can't generate strongly typed stubs

Microservices Requires a Higher Level of REST Use

Oracle fully supports level 3

- **Level 3**
  - Application itself tells client how to interact with it - similar to hyperlinks in a web page
  - <link rel = "delete" uri = "/OrderService/12345/delete"/> under <order id="12345">

- **Level 2**
  - Start using HTTP verbs - HTTP GET/POST/PUT/DELETE/etc
  - Responses come back using correct HTTP codes – e.g. HTTP 409 for a conflict

- **Level 1**
  - Start requesting individual resources
  - Interact with /OrderService/12345

- **Level 0**
  - Use HTTP as a tunneling mechanism only
  - Interact with /OrderService
  - Use HTTP GET/POST only

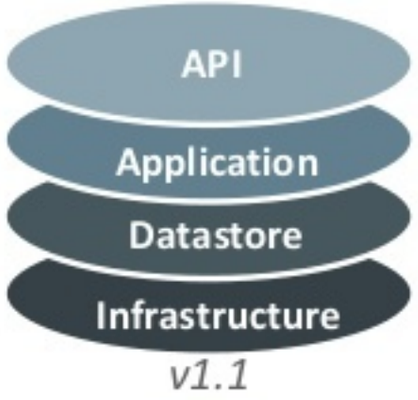
{JSON}

<XML>

Java ORACLE

Message Compatibility Across Versions

Design for backwards compatibility



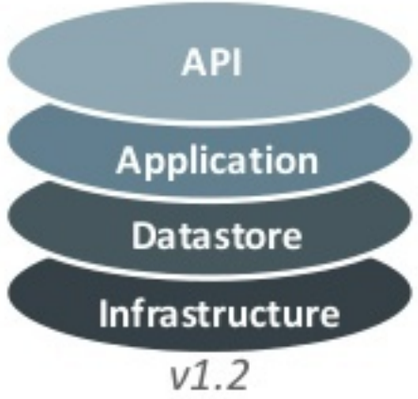
v1.1

Send update to Product Catalog microservice

```
<product>
   <id>329340224</id>
</product>
```

Don't make properties required!

Assume any given microservice has two or more different versions running concurrently. Build for backwards compatibility



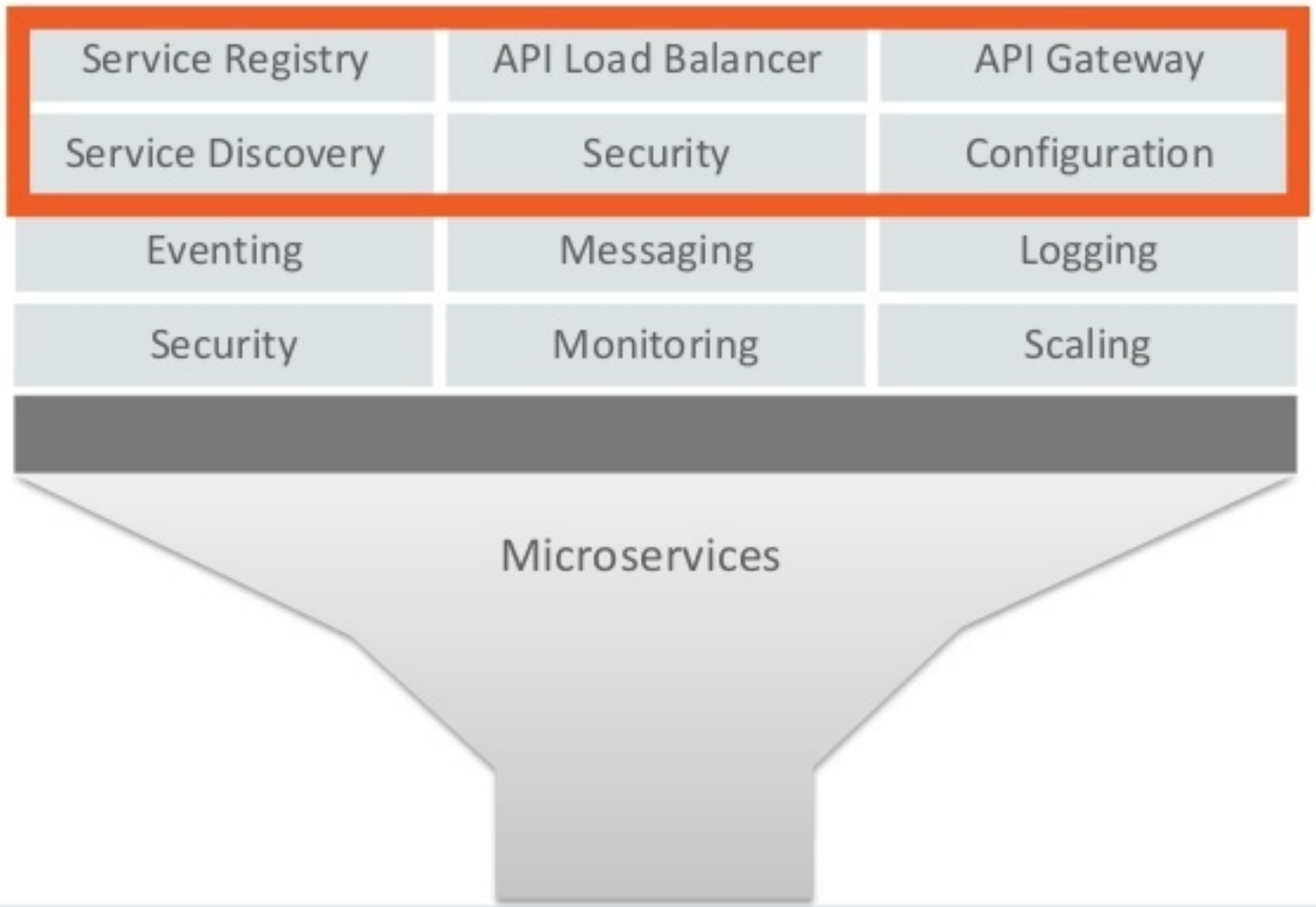
v1.2

Send update to Product Catalog microservice but v1.2 adds a new required property

```
<product>
   <id>329340224</id>
   <shipToStore>true</shipToStore>
</product>
```

Technical Prerequisites

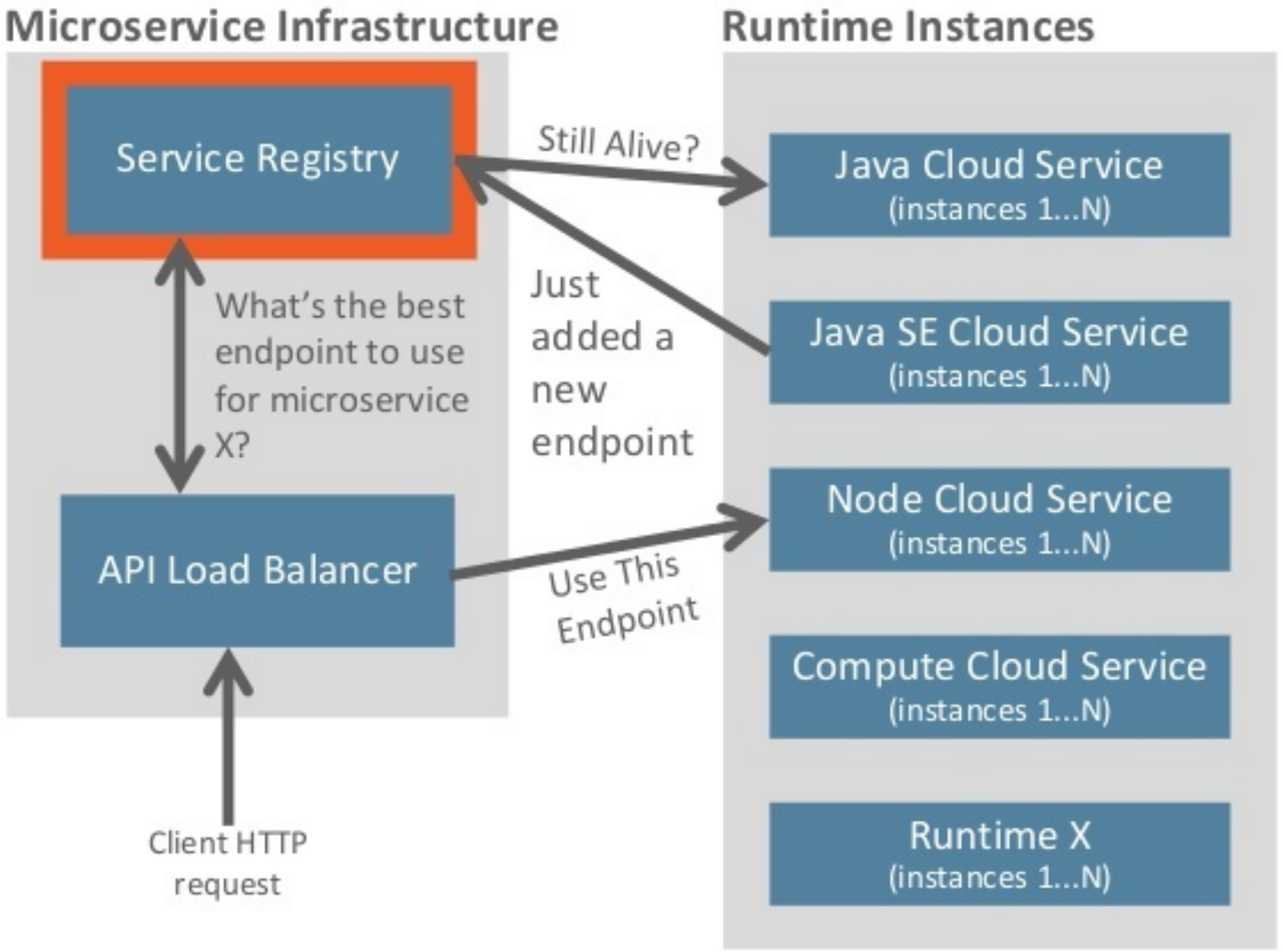
Requirements for Microservices Implementation



Generally Recommended for Microservices

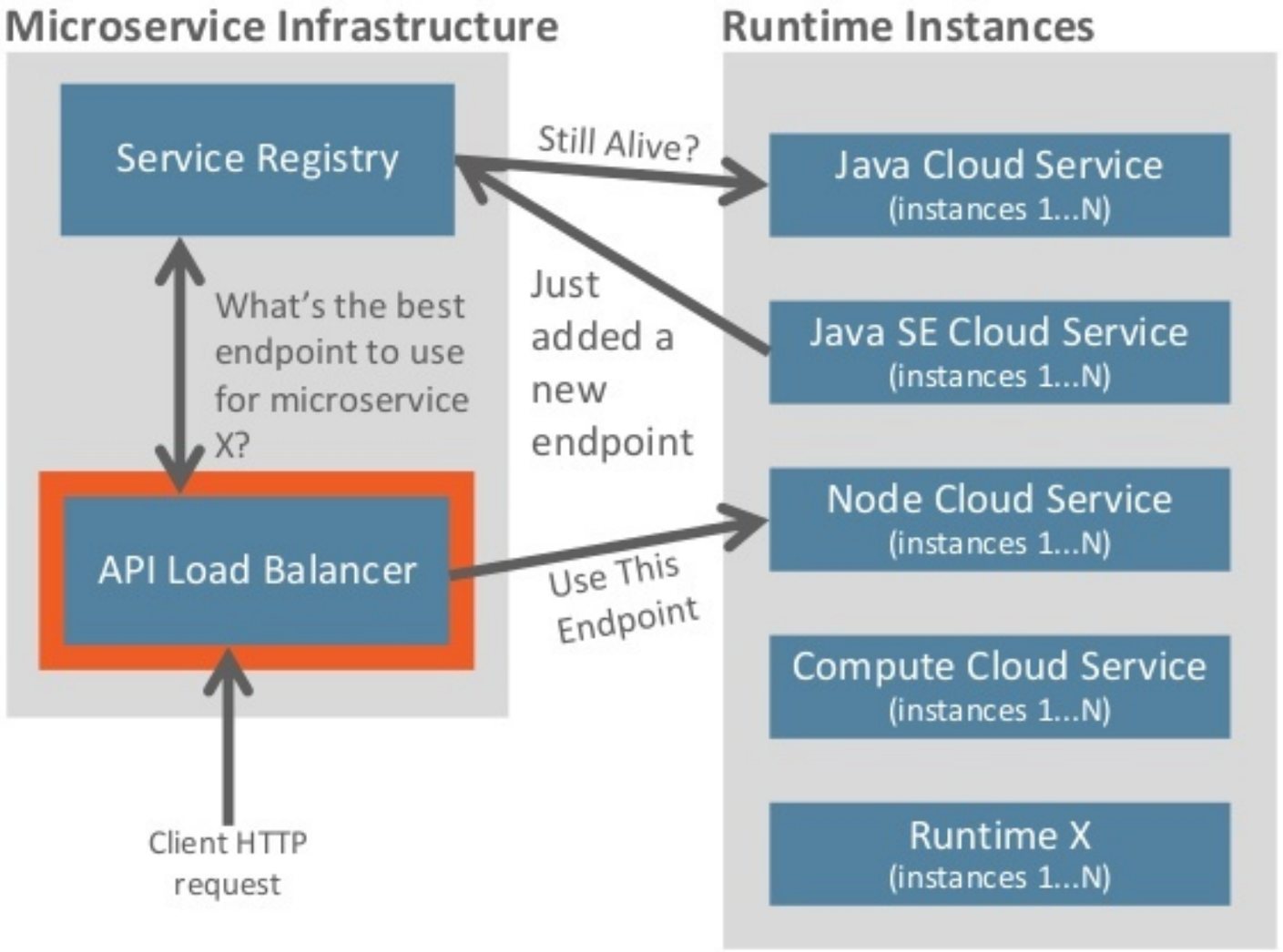
Key Microservices Technology – Service Registry

- **Manages the lifecycle of each microservice endpoint**
  - Newly-instantiated endpoints register with Service Registry
  - Service Registry continually polls each endpoint's health
- **Aware of tenants, microservice versions, and environments**
  - Health checking and selecting the most appropriate endpoint are very much dependent upon the tenant, version, and environment
  - Can query for an endpoint based on those attributes



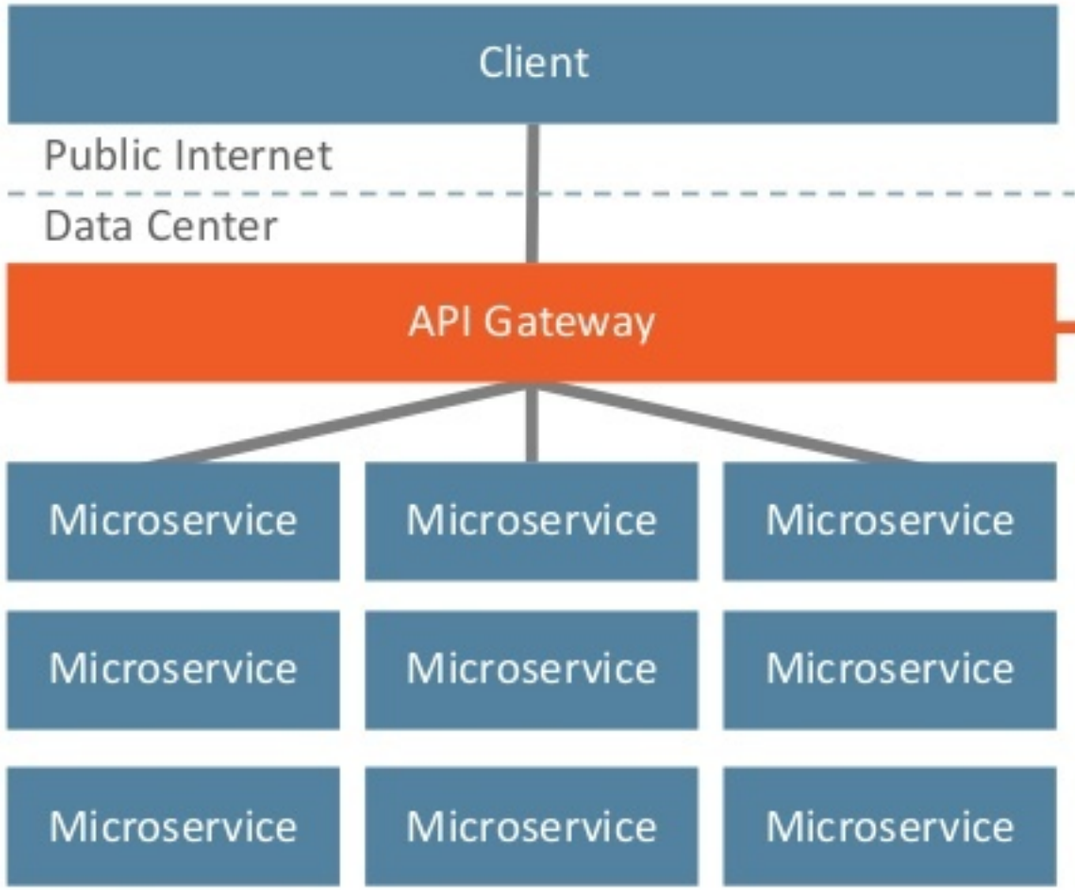
Key Microservices Technology – API Load Balancer

- Matches each request to the best endpoint
- Queries Service Registry to determine the best endpoint
- Ideally stateless – looks up the best endpoint for each HTTP request
- Uses load balancer or web server as the core. Typical implementations are customized Oracle Traffic Director, Nginx, or Apache mod_rewrite
- Handles logging, request rate limiting, authentication



API Gateways Load Balance and Aggregate Responses

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 back-end
- Load balances
- Applies *limited* business logic
- Meters APIs
- Logs centrally
- Common solutions: Netty, Vertex, Nginx, Kong, Apigee

API Load Balancer vs. API Gateway

API Load Balancer

- ✓ Matches each request to the best endpoint
- ✓ Queries Service Registry to determine the best endpoint
- ✓ Handles logging, request rate limiting, authentication

API Load Balancers are missing aggregation

but...

It's best to build a microservice that does aggregation. Keep the API Load Balancer or Gateway as free of business logic as possible

API Gateway

- ✓ Matches each request to the best endpoint
- ✓ Queries Service Registry to determine the best endpoint
- ✓ Handles logging, request rate limiting, authentication
- ✓ Aggregates the responses of many microservices

Service Discovery – Generally Not a Problem With Monoliths

Requirements

- Must resolve:
  - Host/port
  - Version for each microservice
- Must be discoverable

Because

- Many microservices in an environment
- Many environments
- Many versions of each microservice
- Hosts/ports can change quickly
- It's not practical to manage by hand

Solutions Include

- Any DNS provider
- Consul
- Zookeeper
- Custom coding to search instance tags

Approaches Include

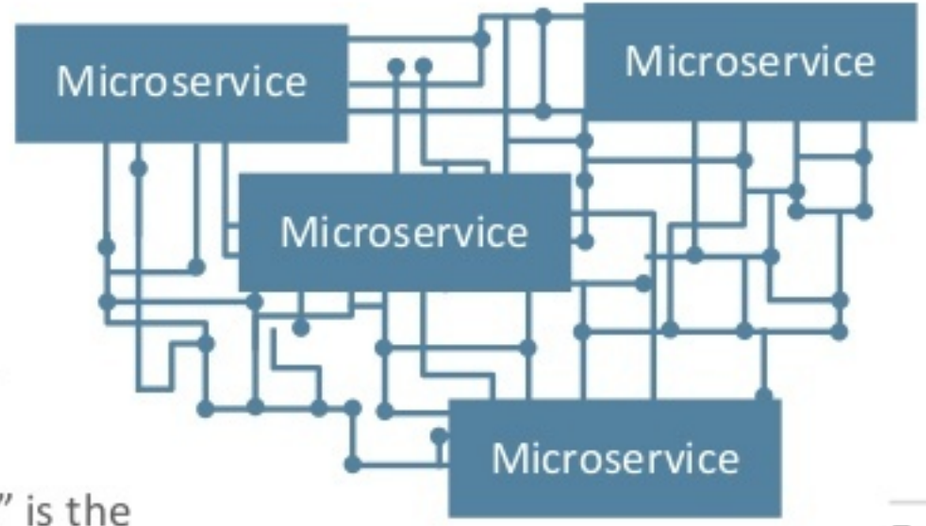
- Plain DNS - limited to IP only
- SRV DNS records - gives IP + port
- Hierarchical namespaces
- Instance tagging in the cloud

Microservices Requires Robust Messaging

Both traditional durable messaging and non-durable eventing

Why Use Messaging?

- Message broker can buffer messages until the consumer is able to process them – prevents synchronous coupling which leads to outages



Requirements for Messaging

- "Smart endpoints, dumb pipes" is the philosophy of microservices – messaging should just pass messages. Not manipulate them
- Should support a variety of communication patterns including one-way requests and publish-subscribe

Types of Messaging

Normal Messaging

- Durable, ordered
- Relatively low throughput
- Usually brokered
- Often used to keep the data across different microservices in sync, as each microservice has its own data store

Eventing

- Non-durable, un-ordered
- Very high throughput
- Usually non-brokered
- Often used to distribute notification events – scale up, scale down, etc

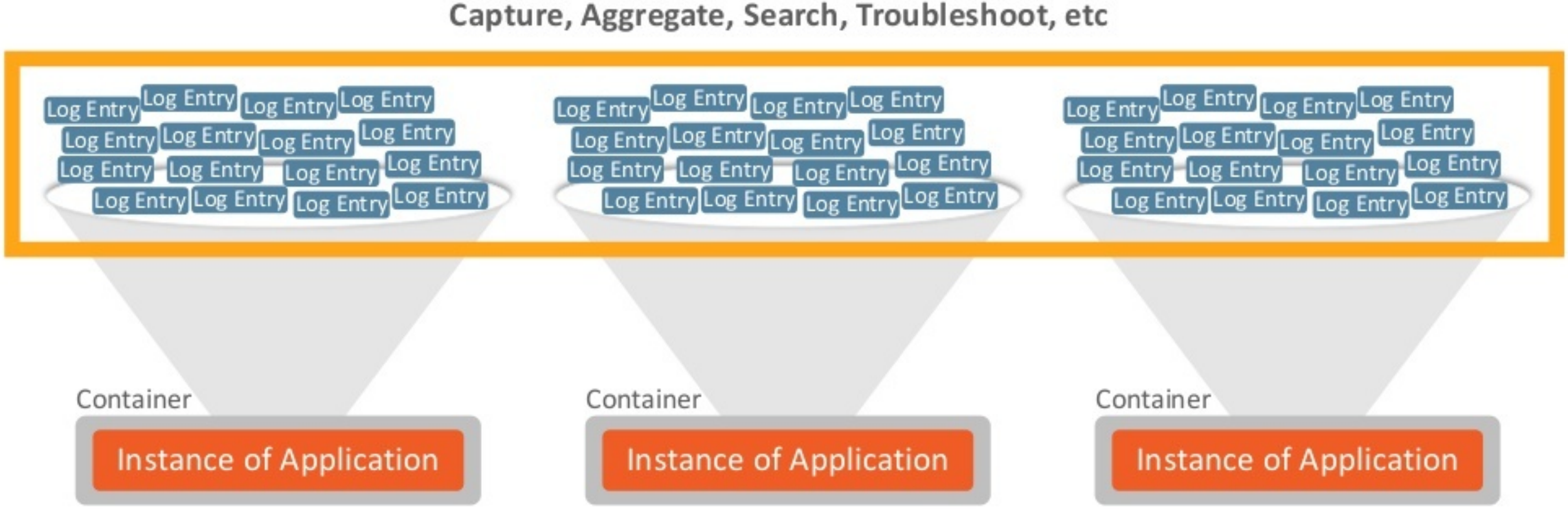
Circuit Breakers Prevent Cascading Failures

Cascading failures are more common with microservices

- Rule #1 of microservices – avoid coupling!
  - Synchronous = two systems are coupled
  - Asynchronous = no coupling
- Cascading failures happen when request-handling threads are waiting on a response from a remote system
- Circuit breakers make synchronous calls from another thread pool to avoid binding up request-handling threads
- Hystrix (Java-based) is well-known and solves this problem

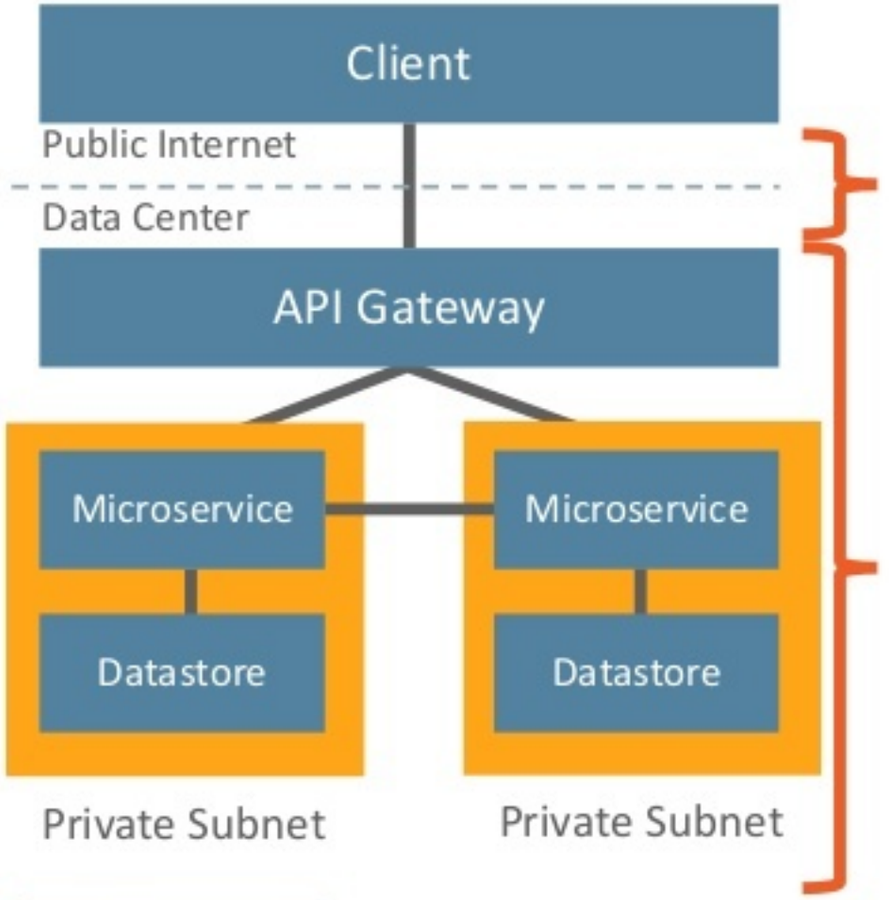
Emit Logs as Event Streams

Can't do anything with log files sitting on a container's local storage volume



Security: Requires New Paradigms to Properly Secure

More microservices, more principals, more technologies, more everything



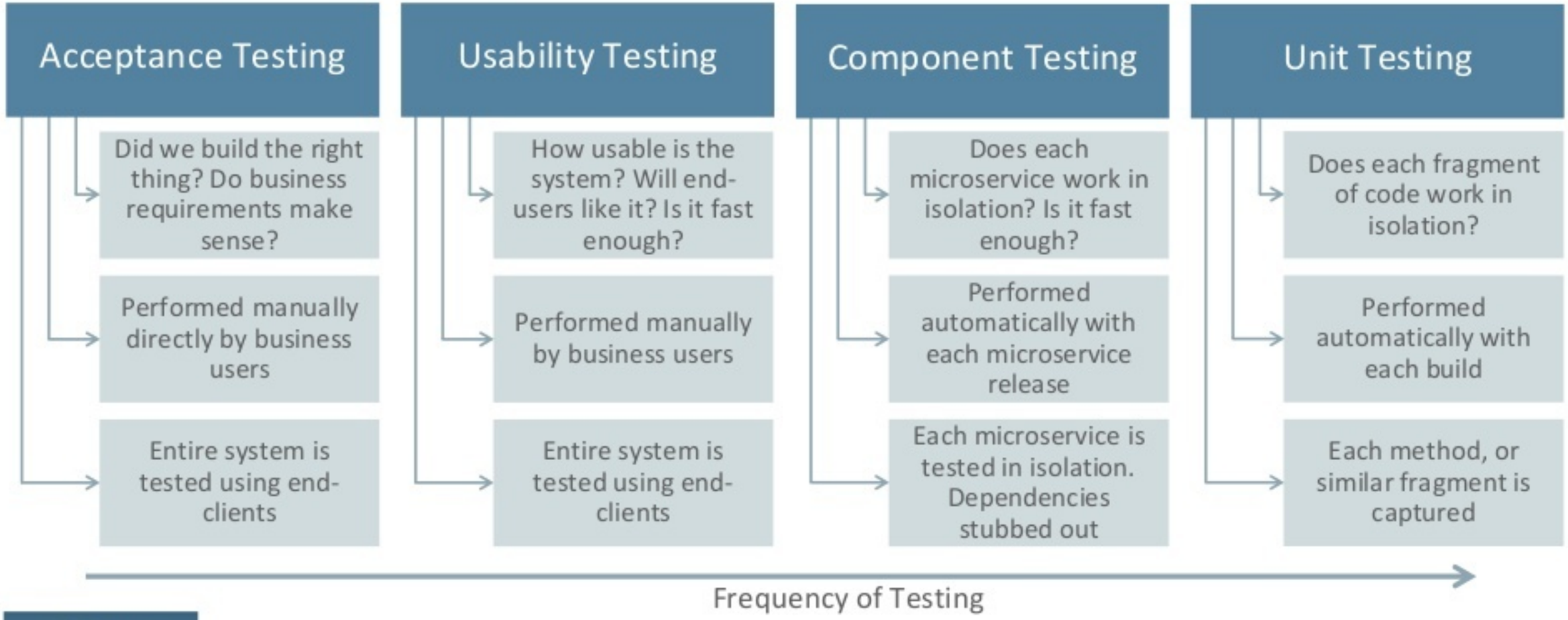
- HTTPS

- HTTPS
- HTTPS Basic
- Client certificates
- Public/private keys
- SAML
- OpenID Connect

- Must *authorize* and *authenticate* every single principal – best to use common approach
- Monolithic apps typically do authentication and authorization on their own
- Secure every single remote network call
- Use network segmentation to isolate microservices from each other

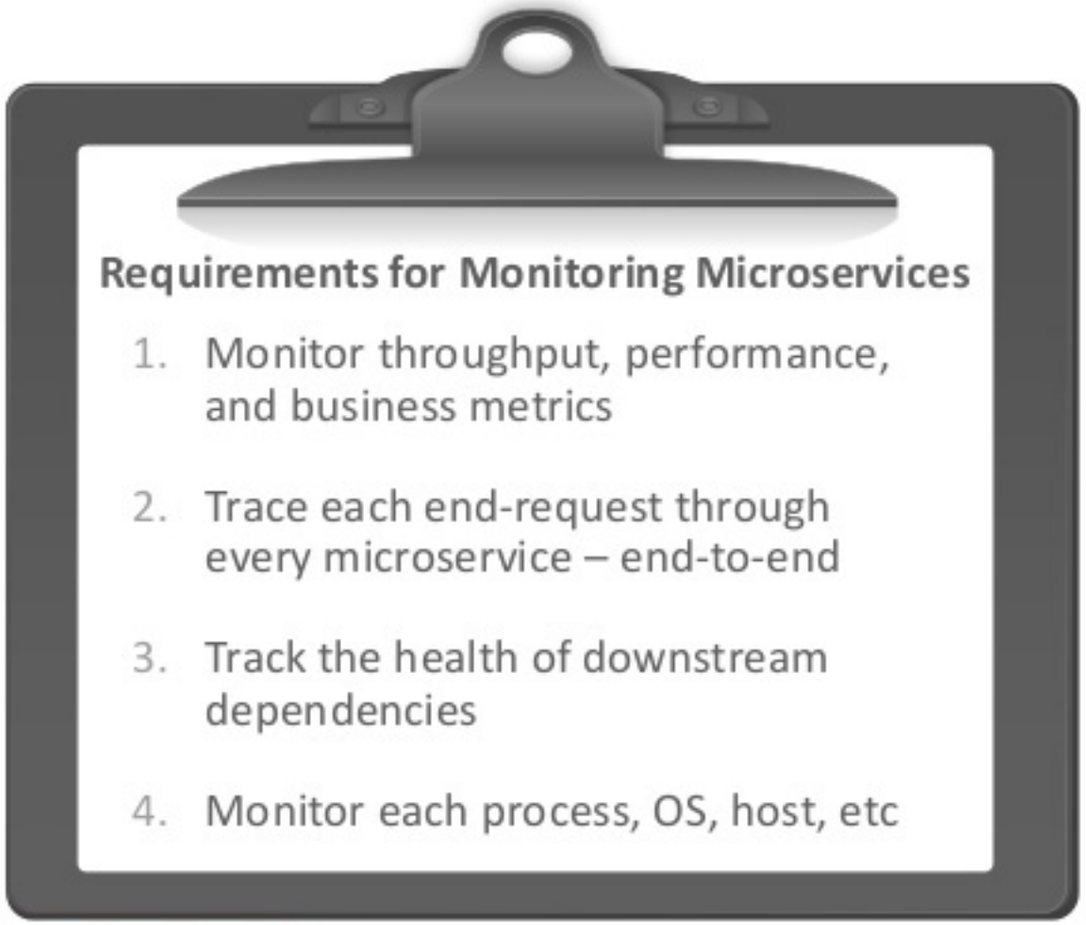
Microservices Complicates Testing

More code, more microservices to test



What/How To Monitor

Monitoring a monolith is relatively easy – one app. Microservices = many apps

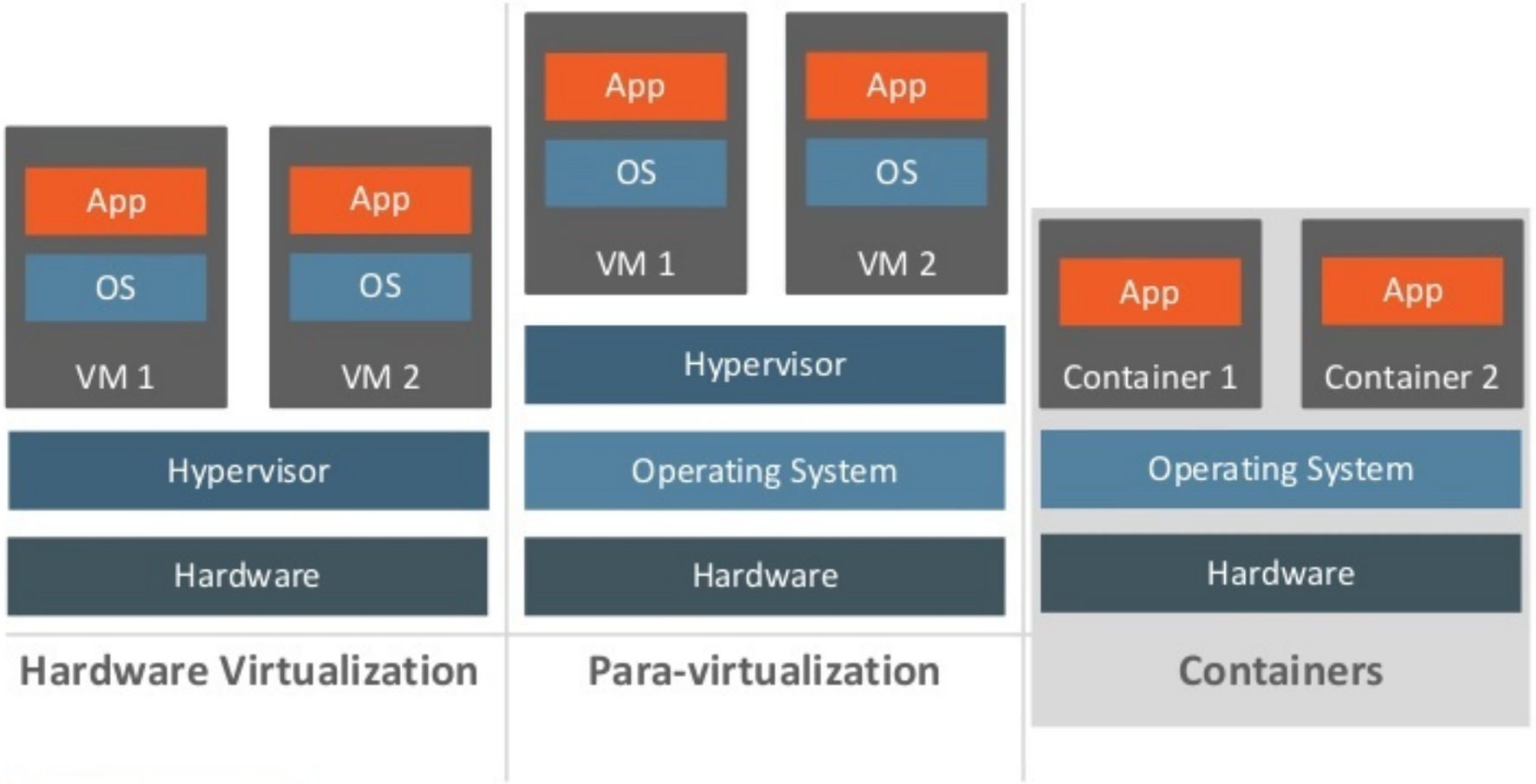


Popular Tooling

Dropwizard Metrics

Containers Make Microservices Easier

Helpful to microservices but not a requirement



- #1 value – app packaging
- Microservices doesn't rely on containers but they do help:
  - Higher density
  - Easy to start/stop
  - Portability
- Containers are lightweight, just like microservices themselves

Many See Containers As the Standard

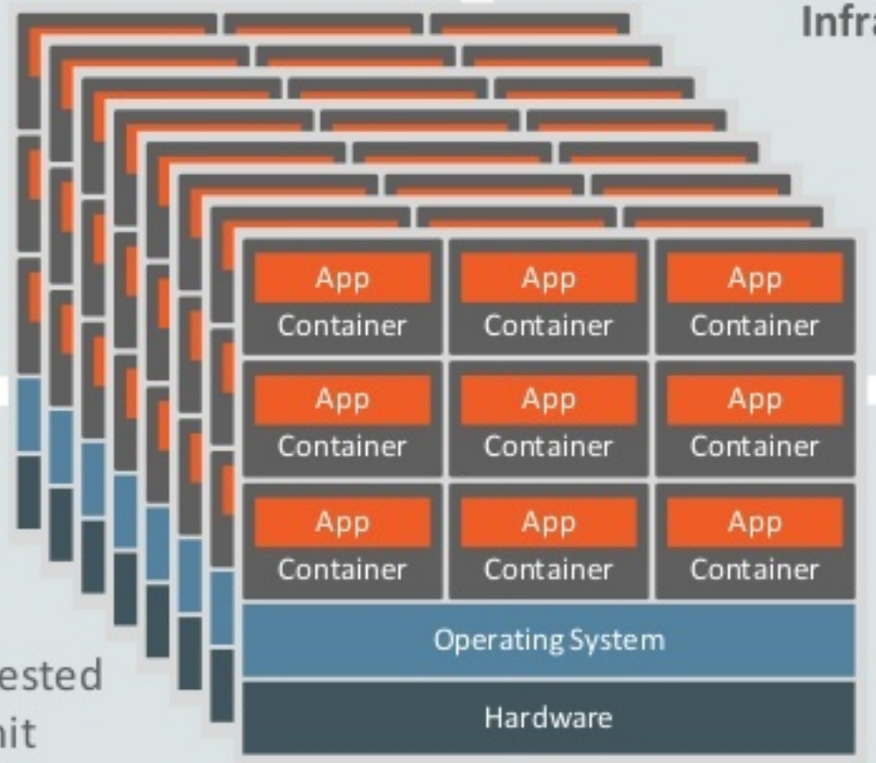
Four main use cases

Application Packaging

Neatly package applications and supporting environment in immutable, portable containers

Infrastructure Consolidation

Get infrastructure utilization up to 100% (vs 5-10% with VMs) due to over-subscription of resources and near bare metal performance.



Continuous Integration

All changes to an app are contained in one immutable container image. Container is tested and deployed as one atomic unit

DIY PaaS

Build a simple PaaS by wiring up containers to a load balancer. New code, patches, etc pushed as new immutable containers.

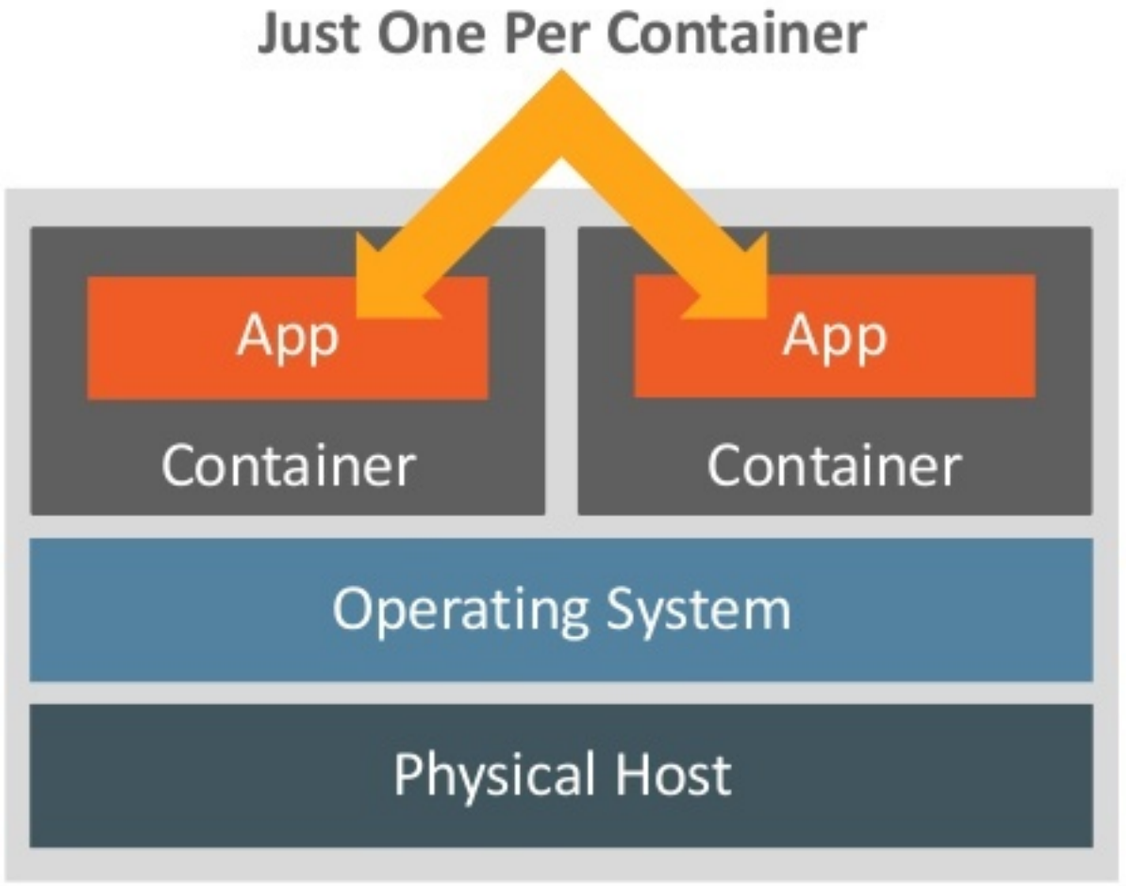
Containers Should Be Immutable

Build and deploy a new container

Never touch a container that's already been built

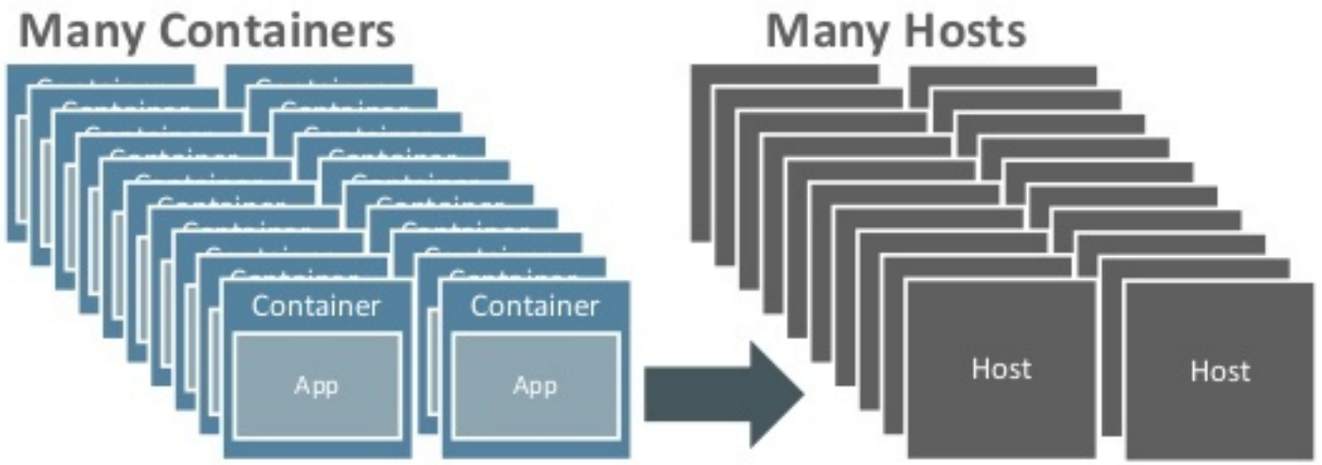
One Instance Per Container is Typical

- Best to run one instance (unique host/port combination) per container
- Running multiple instances of the same application or different applications will make scheduling very difficult
- Expose one port per container



How Do You Deploy Containers to Physical Hosts?

The emerging space of container orchestration



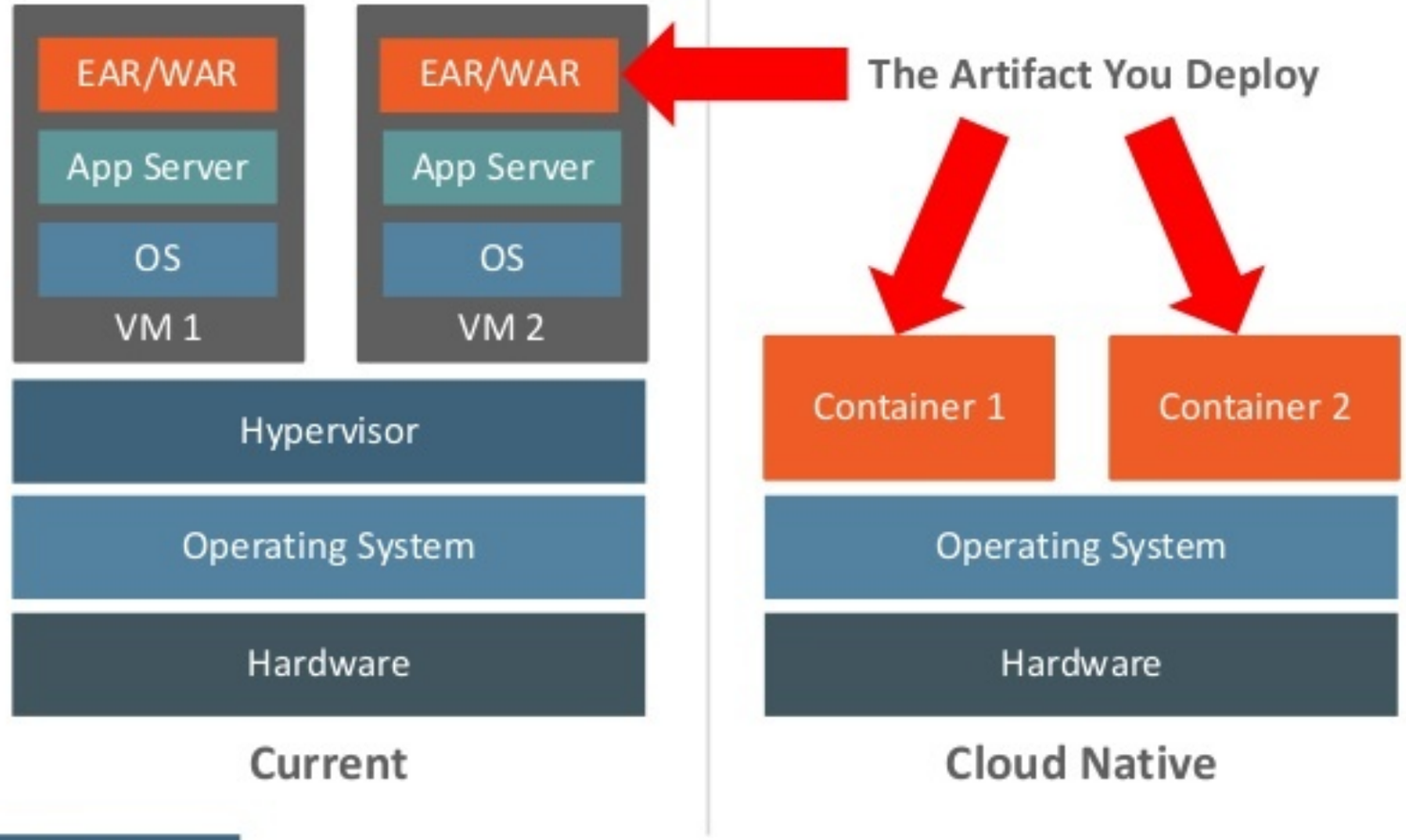
Emerging space. Solutions are very early and lack any real notion of an application. Still very much infrastructure-focused

What Do Container Orchestration Solutions Do?

- Map containers back to physical hosts, taking into account user-defined placement rules, the utilization of each host, and the needs of each container. Can be very complex
- Set up overlay networking, firewalls, ensure network QoS, etc
- Auto-scaling
- Local and external load balancers
- Service registry / discovery

Artifacts Are Now Immutable Containers – Not EARs, WARs

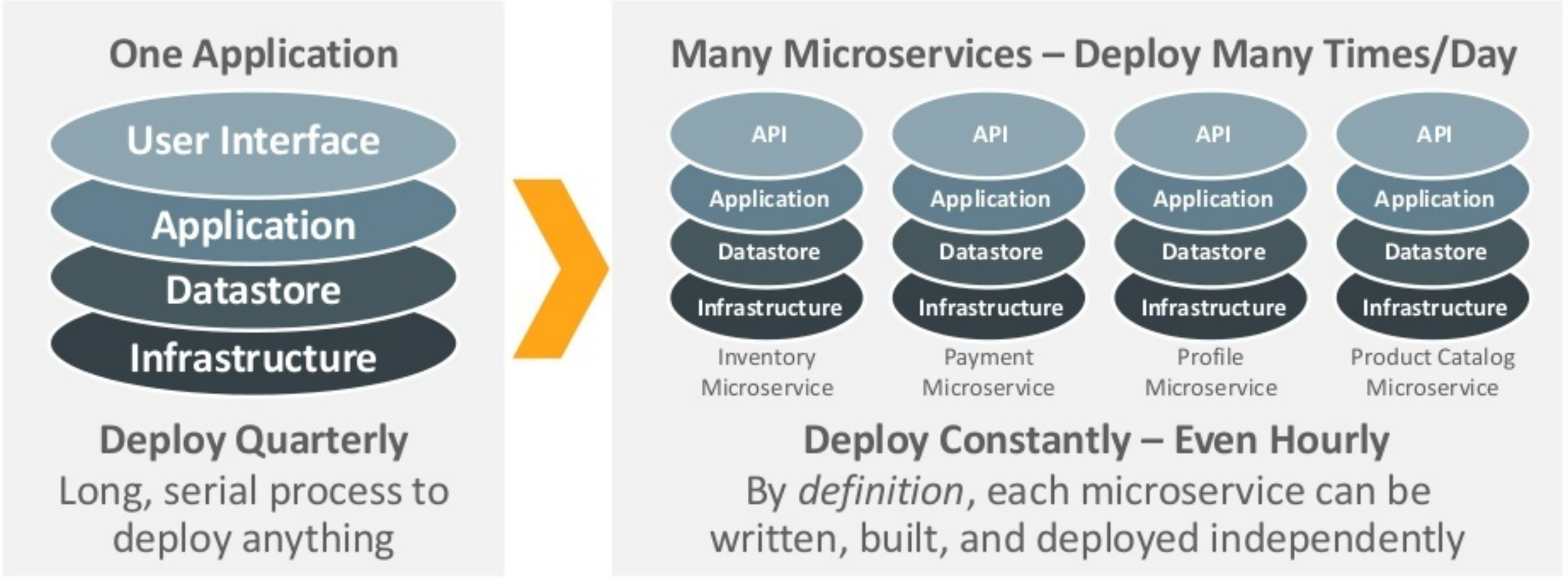
Containers can have anything in them and are highly portable



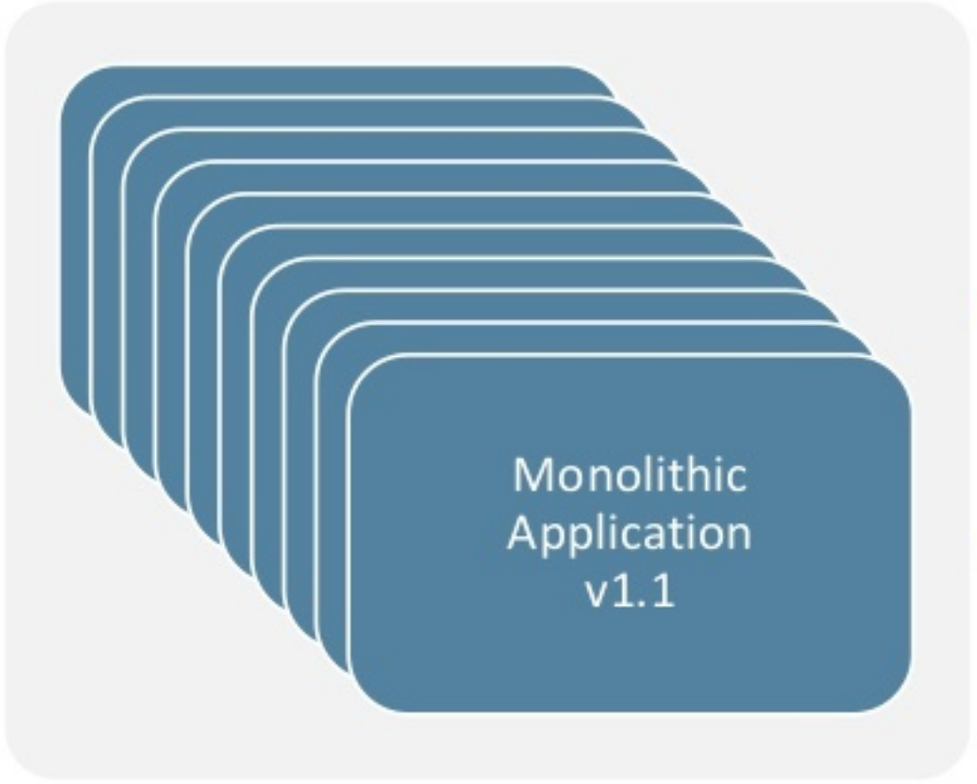
- No more installing a JVM, app server, and then deploying the artifacts to them
- Build the container once, deploy it anywhere. Can include complex environment variables, scripts, etc
- Containers should be free of state and configuration
- Containers should not assume they are able to write to a persistent local file system

Constantly Deploy Each Microservice

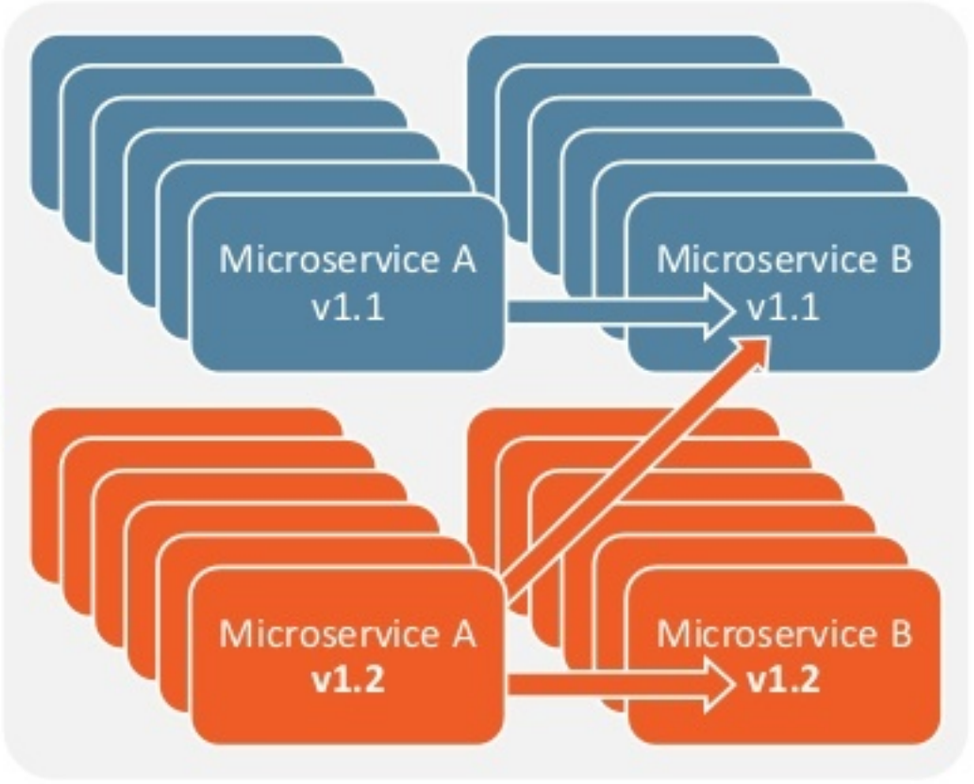
No need to batch together many changes to single app



Run Many Versions of the Same Microservice Concurrently



Run only *one* version of the same application in the same environment



Run *many* versions of each microservice in the same environment

Application Containers – Use What You Have Today

Characteristics of Existing Application Server

- Run a very large, complicated monolithic application
- Many advanced features
- Integrated dependency management
- Few instances per host
- Stateful

Characteristics of Microservices-friendly Application Server

- Run a smaller, microservice that does one thing really well
- Few features
- External dependency management
- Many instances per host
- Stateless

Run whatever works best – few firm requirements

How Oracle Products Support Microservices

Oracle's Microservices Strategy

	Oracle's Microservices Roadmap	Today With Oracle Products
State	**Planned** - Oracle Cloud State Service	Oracle Coherence or Oracle WebLogic
Configuration	**Planned** - Oracle Cloud Config Service	Oracle Coherence
Runtime	**Planned** - Runtime - Jersey + Grizzly	Oracle WebLogic, Node, Java SE, etc
Eventing	**Planned** - Oracle Cloud Eventing Service	Oracle Coherence
Messaging	Oracle Messaging Cloud Service	Oracle Messaging Cloud Service
Management/Logging/Alerting	Oracle Management Cloud Service	Oracle Management Cloud Service
Datastore	Oracle Database or NoSQL Cloud Service	Oracle Database or NoSQL Cloud Service
Central Source of Truth	**Planned** - Oracle Microservices Platform	Oracle Coherence
Service Discovery	**Planned** - Oracle Microservices Platform	Oracle App Container Cloud Service
API Gateway/Load Balancer	**Planned** - Oracle Microservices Platform	Oracle App Container Cloud Service
Container Orchestration	**Planned** - Oracle Microservices Platform	Oracle App Container Cloud Service
Infrastructure	Oracle Cloud	Oracle Cloud
Build/Deploy	Oracle Developer Cloud Service	Oracle Developer Cloud Service

An All-New Microservices Platform From Oracle

Coming Soon!

All New

Java Runtime — Light/fast, Jersey-based, Easy Packaging

RAILS · node · Microsoft .NET · python · Scala · Docker

ORACLE Microservices Platform

- Rolling Upgrades
- Stateless Support
- Stateful Support
- Rollbacks
- Service Discovery
- Blue/Green Releases
- Canary Testing
- Health Monitoring
- Language-specific Tooling
- Infrastructure Agnostic
- Placement Constraints
- High Availability
- High Density
- Hybrid Cloud
- Internal Load Balancing

Consume Platform as PaaS on Oracle Cloud

Install Platform On Any IaaS, On Or Off Premise (future)

ORACLE Cloud — Oracle Public Cloud

amazon web services · Microsoft Azure · Google Cloud Platform

ORACLE Cloud — Oracle Public Cloud *Machine*

PureSystems · vmware · VBLOCK · openstack · servers

ORACLE Cloud

- Config
- State
- Eventing

New!

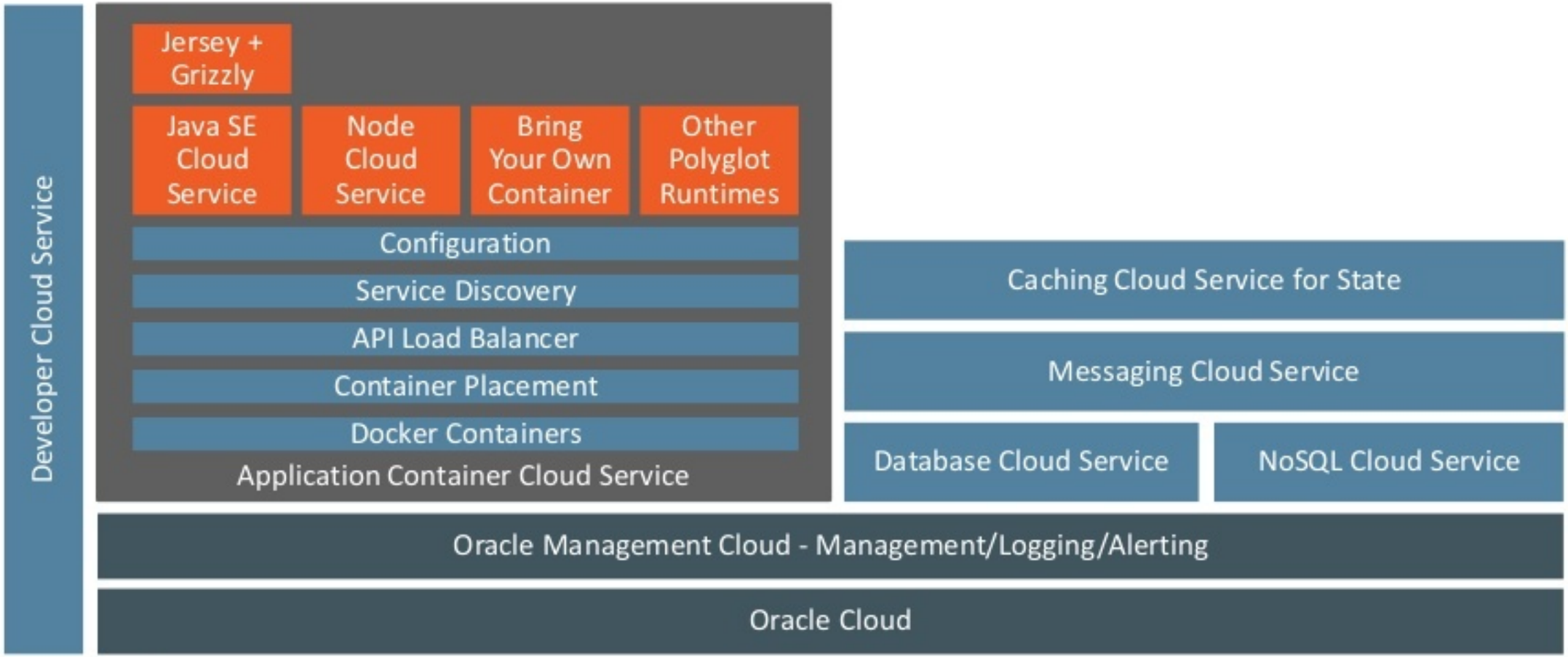
- Messaging
- Logging
- Monitoring
- Database
- NoSQL
- Caching
- Integration
- Big Data
- Mobile
- Process
- Developer

Services Available for Consumption Inside the Platform

Java ORACLE

Using Application Container Cloud Service For Microservices

A modern platform for lightweight application development



Introducing Oracle App Container Cloud Service

Key Features

- Useful for any Java, Node.js, or polyglot runtime
- IDE Choice - JDeveloper, Eclipse, NetBeans - and API access
- Continuous integration with Oracle Developer Cloud Service
- Cloud tooling for lifecycle management
- Integrated load balancer, support for service discovery
- Bring-your-own-container model supported shortly

Benefits

- Self-service application platform with advanced cloud tools
- Secure, Highly Available with Clustering
- Fully automated provisioning, patching, backup, and recovery

JAX-RS: The Java API for RESTful Web Services

Oracle is a spec lead

Server-side Code

```
@Path("/atm/{cardId}")
public class AtmService {
  @GET @Path("/balance")
  @Produces("text/plain")
  public String balance(
        @PathParam("cardId") String card,
        @QueryParam("pin") String pin) {
```

Simply annotate Java code to expose as REST

Client-side Code

```
Client client = ClientFactory.newClient();
String balance =
  client.target("http://xxxx/atm/{cardId}/
                                    balance")
  .pathParam("cardId", "1234567890123456")
  .queryParam("pin", "1111")
  .request("text/plain")
  .get(String.class);
```

Use REST without having to parse text

Originally defined in JSR 311 - 1.0
Part of Java EE 6 Spec

|

Updated as JSR 339 in 2013 - 2.0
Part of Java EE 7 Spec

Jersey: The Reference Implementation of JAX-RS

Oracle continues to lead the spec

JERSEY

- Oracle sponsored open source
- Implements the JSR 311 specification
- Contains
  - Standalone server
  - Client
  - JAXB/JSON support

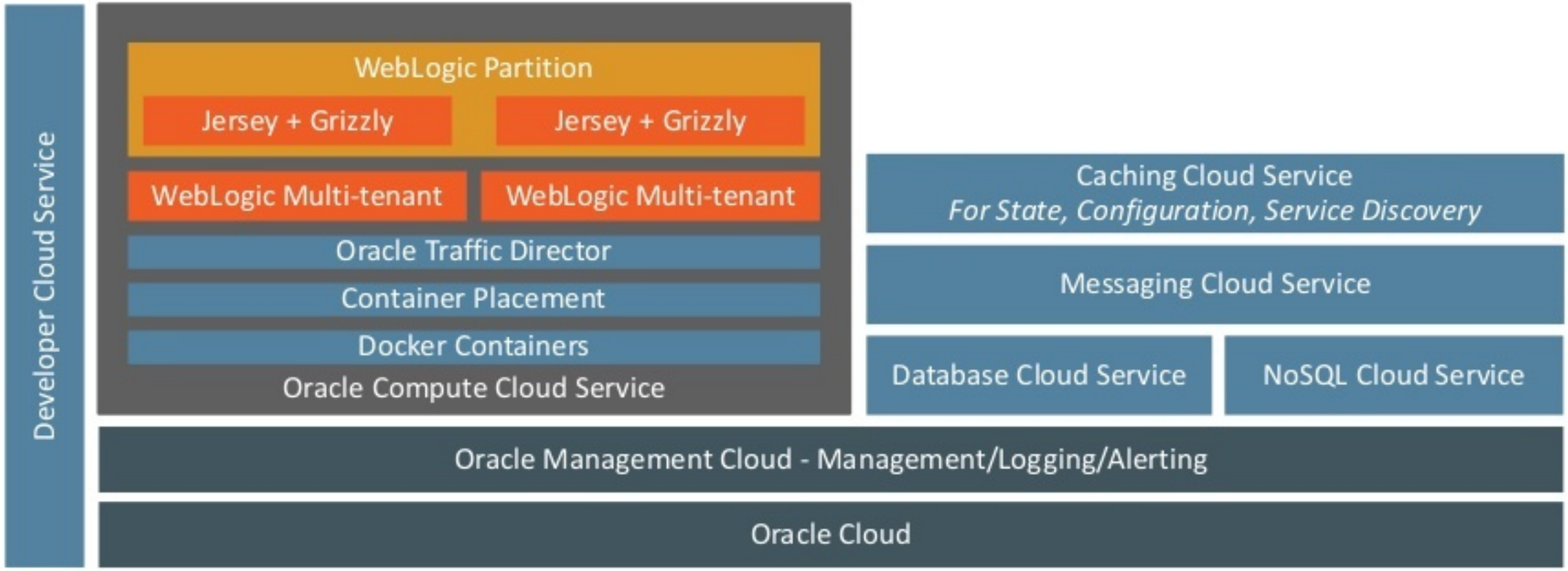
Grizzly: High Performance I/O

Great for inter-process communication

- Oracle sponsored open source
- Allows developers to take advantage of the Java NIO to provide very fast inter-process communication
- Brings non-blocking sockets to the protocol processing layer
  - Support for non-blocking HTTP processing
- WebSocket Support
- APIs make non-blocking interactions simple

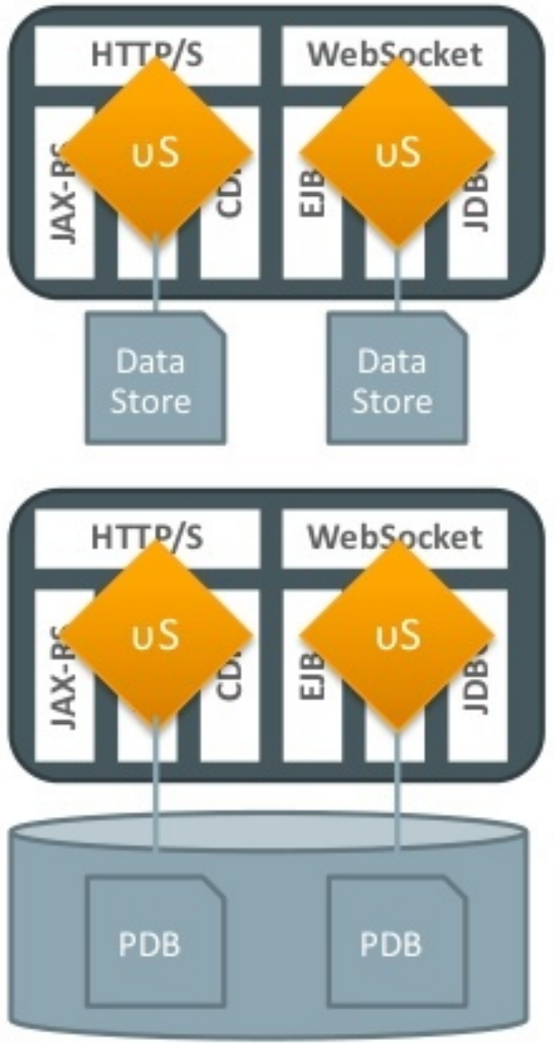
Using WebLogic + Java EE For Microservices

A proven, enterprise-grade platform for application development



Java EE Platform Supports Microservices

- Standards based infrastructure
  - Prescribed, validated set of APIs and services
  - WAR file deployment for service boundary
- Enables development of thin Microservices
  - Assemble application code and resources only
  - APIs and implementation libraries supplied by platform
- Freedom to choose best service topology
  - One service per server, multiple services per server
  - One server per container/host
  - Multiple containers/servers per host



ORACLE WEBLOGIC

WebLogic Supports Microservices (and then some!)

- Java EE Platform Support
  - With latest Java language support
- Fully automatable infrastructure
  - Scriptable provisioning, configuration, deployment supporting code-as-infrastructure
  - REST API, WLST, JMX
  - Callable from Maven, Gradle, Arquillian
  - Easily incorporated into Continuous Integration and Delivery workflows
- Flexible placement models
  - Single server, multiple servers, clusters of servers
- Diverse datastore options
  - Traditional multi-vendor relational database support
  - Integrated Oracle Database 12c pluggable database support as independent data stores for services
  - Simple configuration for Document/Graph databases
  - Container, application, embedded application scoped database resources

Java ORACLE

ORACLE WEBLOGIC

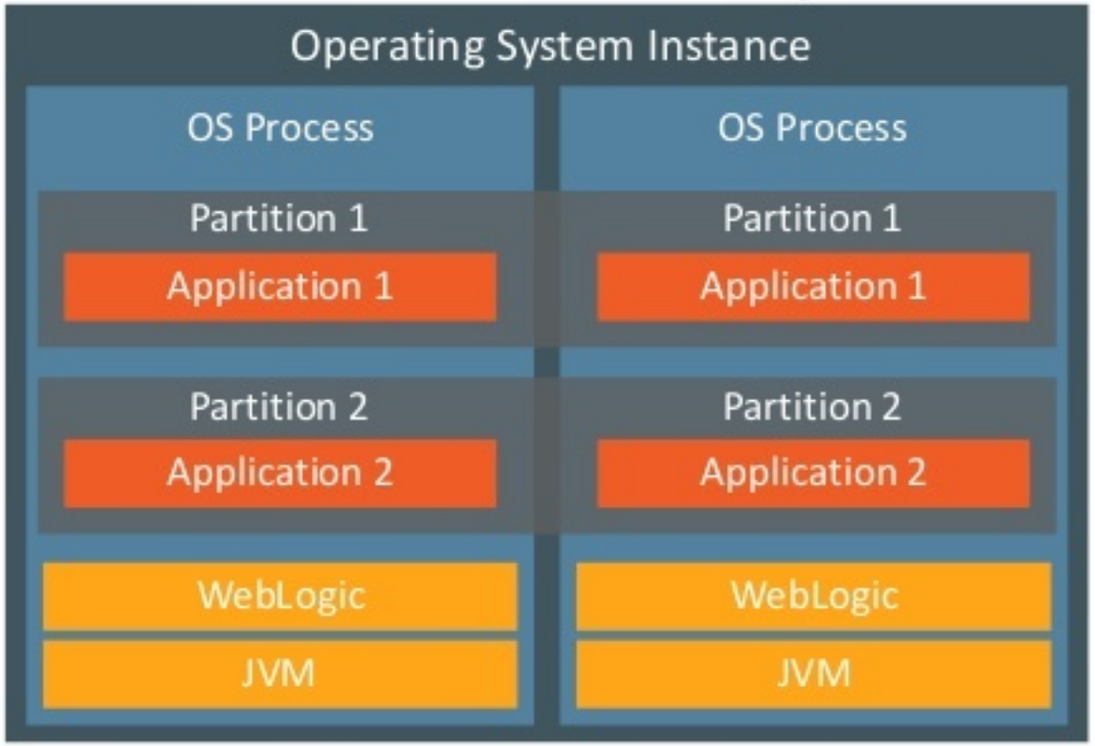
WebLogic Multi Tenant

Partition is now a first-class object

Standard WebLogic

Single-tenant

Multi Tenant WebLogic



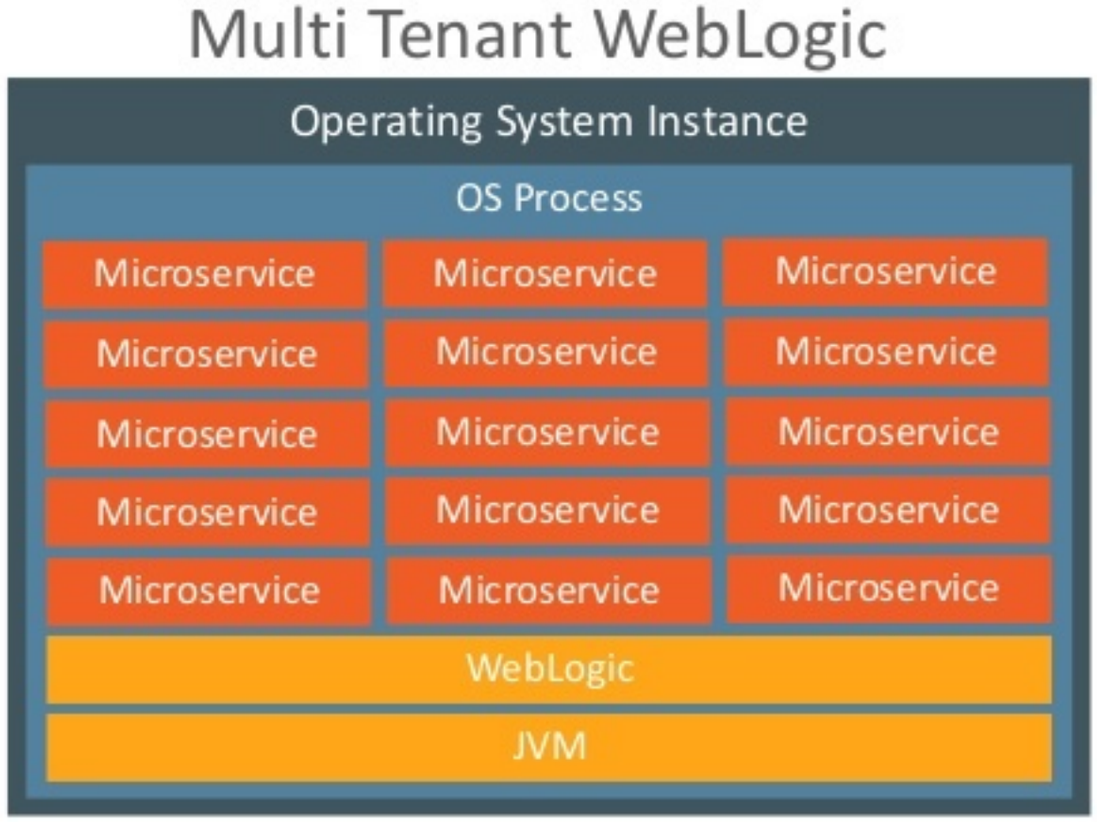
Multi-tenant – strong isolation between tenants

Java ORACLE

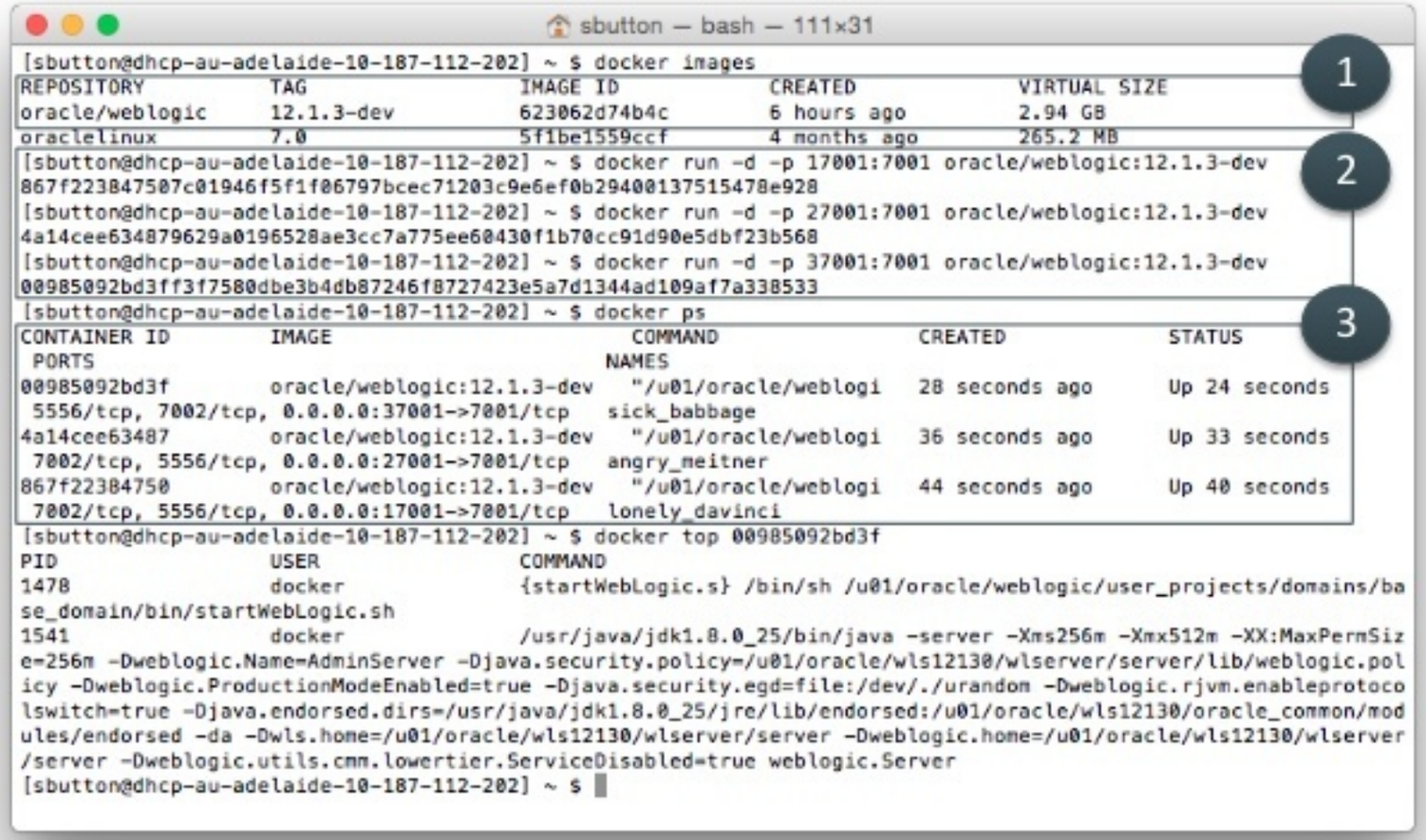
WebLogic Multi Tenant is Perfect for Microservices

Similar to Oracle Database pluggable/container databases

- Each microservice instance can have its own light-weight WebLogic container-like partition
- Easily move partitions between WebLogic hosts
- Each partition is exceptionally light
- Each WebLogic host can support hundreds of partitions



WebLogic Runs Great on Docker



- Build oracle/weblogic:12.1.3-dev image
- Start 3 WebLogic Server Containers in seconds
- View status

$ docker run –p 17001:7001 oracle/weblogic:12.1.3-dev

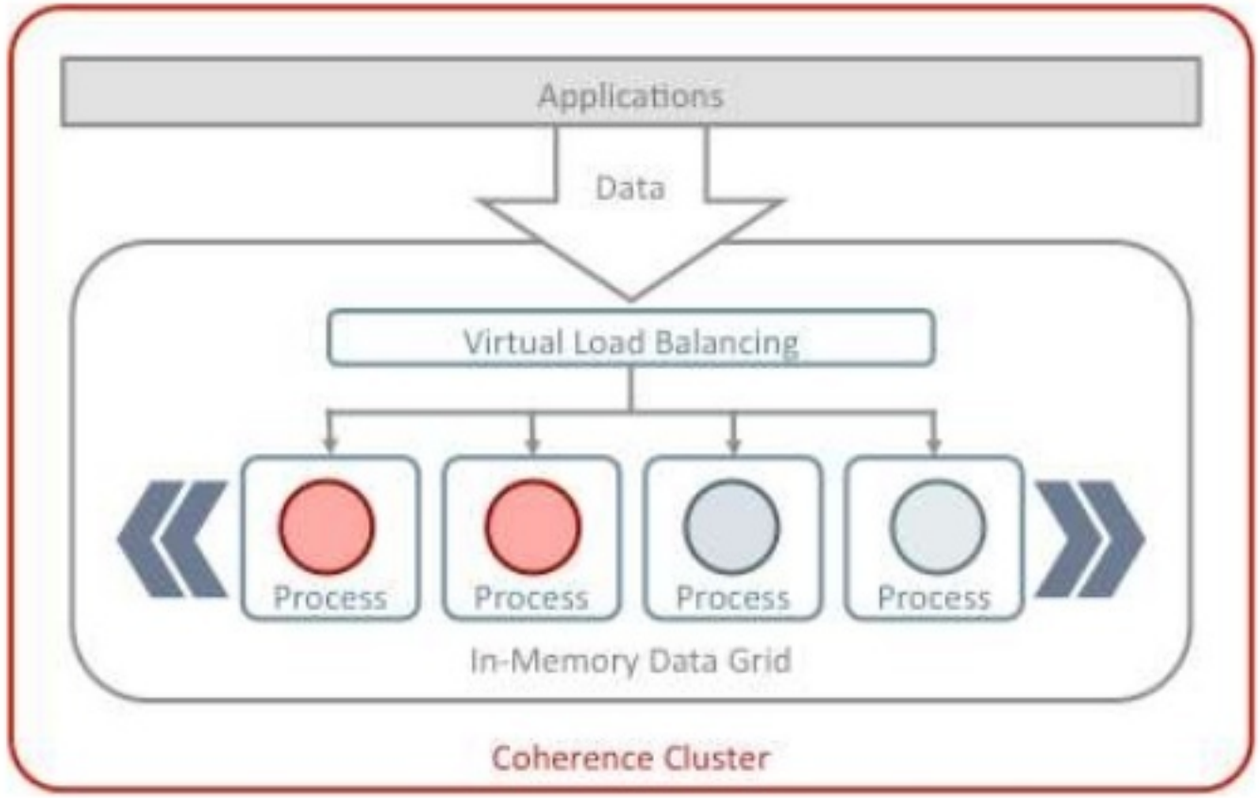
Other Supporting Products from Oracle

ORACLE COHERENCE

Oracle Coherence – Oracle's Distributed Data Grid

Architected for microservice state management

- Reliable in-memory key-value store
  - Dynamically scalable
  - Scale processing with data
  - Flexible topology support
- Java, .NET, C++, REST, Memcached, Jcache client support
- In-place distributed processing
- Queries & continuous queries
- Map-reduce aggregation
- Event notification / event programming model
- Distributed Lambdas and Streams



Java ORACLE

Oracle Messaging Cloud Is Perfect for Microservices

Implements AMQP standard

Standardized Interfaces	Versatile	Delivery Choices	Reliability Mechanisms
REST	*Oracle Cloud*	Pull	*Transactions*
JMS	*On Premise*	Push	*Acknowledgements*
Message push over HTTP	*Hybrid*	Filter	*Durable subscriptions*

Oracle Management Cloud Services – Initial Offerings

Application Performance Monitoring

Improve End-User Experience and System Performance; Diagnose Performance Issues Faster

Log Analytics

Extract Value from Logs by Collecting, Correlating, and Searching Any Kind of Log Data; Quickly Discover Anomalies

IT Analytics

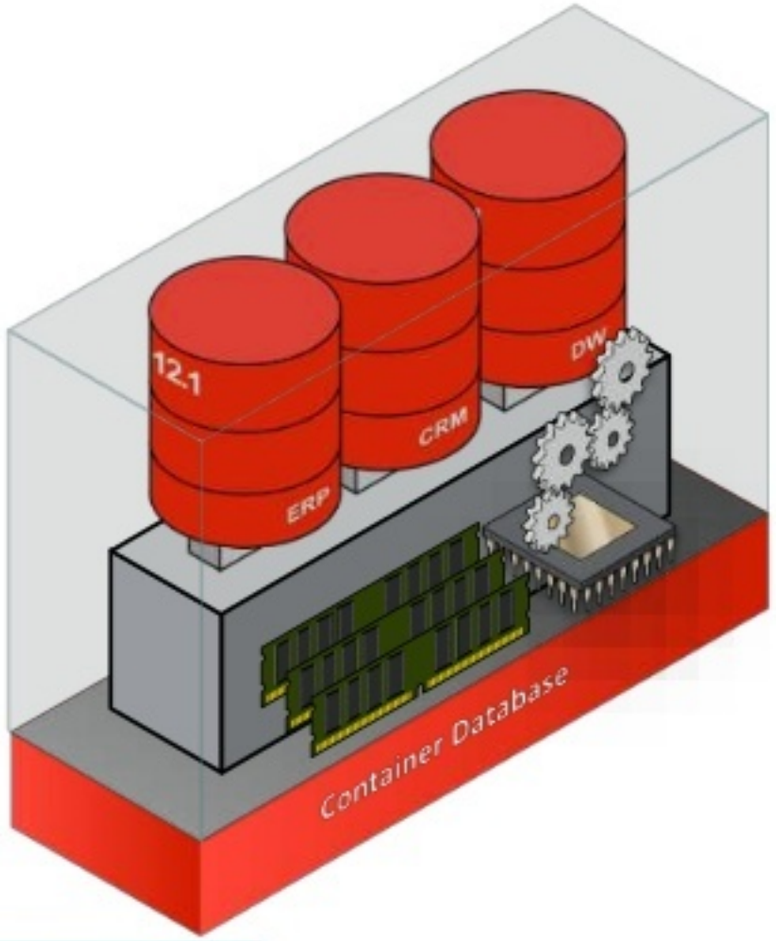
Make Critical Decisions About Your IT Estate; Plan For Growth, Run What-If Analyses, Compare Resource Usage

Oracle Data*store* Options – Offered On Premise and In Cloud

CATEGORY	PRODUCTS	DESCRIPTION
RDBMS	ORACLE® DATABASE	Supports JSON, XML, CLOBs, BLOBs, and multi-media. Accessible over client-specific APIs, REST
Key/Value Stores	ORACLE® NoSQL	Distributed key/value pairs, schema-less, nearly ACID compliant, scale out. Berkeley DB behind the scenes
Object Data Grid	ORACLE® COHERENCE	Distributed data grid that supports grid-side processing

Pluggable Databases

One Container Database per application, one Pluggable Database per microservice



- Container Database
  - Multi-tenant database that includes zero, one or many pluggable databases
  - Upgrades, etc are performed against container
- Pluggable Database
  - A full database to the client except that behind the scenes it doesn't have its own controlfiles, redo logs, undo, etc
  - Just a collection of datafiles and tempfiles to handle its own objects, including its own data dictionary
  - Can easily move Pluggable Databases from one container to another

Oracle Database: **S**imple **O**racle **D**ocument **A**ccess (SODA)

NoSQL backed by Oracle Database

- Enable schemaless development on top of an Oracle Database
  - Provide a simple NoSQL-style API for working with documents
- Make it easy to use Oracle as a NoSQL-style document store
- Supports all common application development environments
- Supports SQL!
- Works with Oracle Database tooling – backup, restore, security, etc

REST

```
POST /DBJSON/SCOTT/CUSTOMERS?action=query HTTP/1.0
Content-Type: application/json
Body:  {
  "company" : "Oracle",
  "$or" : {"$startsWith" : {"name": "Melissa"}}}
```

SQL

```
select "JKEY","VERSION", "LAST_MODIFIED", "CREATED"
  from "SCOTT"."CUSTOMERS"
 where JSON_EXISTS(
    "JVALUE",
       '$?( $.company == $B0 && ( $.name starts with $B2))'
       PASSING 'Oracle' AS "B0" , 'Melissa' AS "B1")
```

JSON

```
{
   "items": [ {
          "id": "09615A98B2B941AF94D84FD44D04AB9C",
          "etag": "D78FBD… B879E",
          "lastModified": "2015-02-10T01:15:13.631231",
         "created": "2015-02-10T01:15:13.631231"

       }, … {more records...}
   ],
   "hasMore": true,
   "count": 4,
   "offset": 0,
   "limit": 4
}
```

SODA For Java

Native library for Java

- Provides:
  - Connection to the database (replaces JDBC!)
  - Collection Management
  - CRUD operations
  - Query-by-Example
  - Utility and control functions
- SODA for Java applications can use a JDBC connection to talk to the database
- SODA for Java is transactional
  - Supports Hybrid model with JDBC and SODA based operations

```
// Create a Connection
OracleRDBMSClient client = new OracleRDBMSClient();
OracleDatabase database = client.getDatabase(conn);

// Now create a collection
OracleCollection collection =
database.getDatabaseAdmin().createCollection("MyCollection");

// Create a document
OracleDocument document = database.createDocumentFromString("{
\"name\" : \"Alexander\" }");

// Next, insert it into the collection
OracleDocument insertedDocument = collection.insertAndGet(document);

// Get the key of the inserted document
String key = insertedDocument.getKey();

// Get the version of the inserted document
String version = insertedDocument.getVersion();
```

SODA For REST

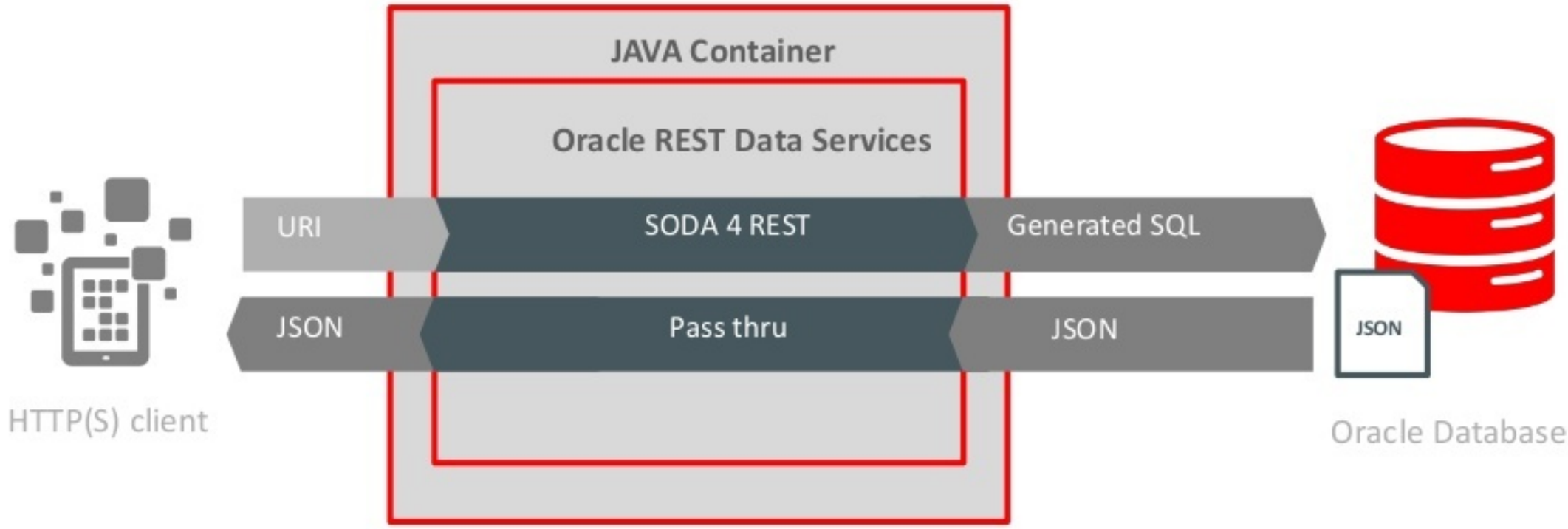
Can be invoked from any programming language

- Standard REST based model
- CRUD operations map to HTTP Verbs
  - Create / Update : PUT / POST
  - Retrieve : GET
  - Delete : DELETE
- Other operations, such as Query by Example, Bulk Insert and Indexing are mapped to variants of POST
- JSON document forms the payload of the HTTP Request or Response
- Stateless model, no transaction support
- Implemented as a Java Servlet

`GET /DBSODA/schema`	List all collections in a schema
`GET /DBSODA/schema/collection`	Get all objects in collection
`GET /DBSODA/schema/collection/id`	Get specific object in collection
`PUT /DBSODA/schema/collection`	Create a collection if necessary
`PUT /DBSODA/schema/collection/id`	Update object with id
`POST /DBSODA/schema/collection`	Insert object into collection
`POST`	Find objects matching filter in

SODA For REST Architecture

Clean architecture



- Data stored in Oracle Database as JSON documents
- App Developer make standard HTTP(S) calls to SODA for REST API

Summary

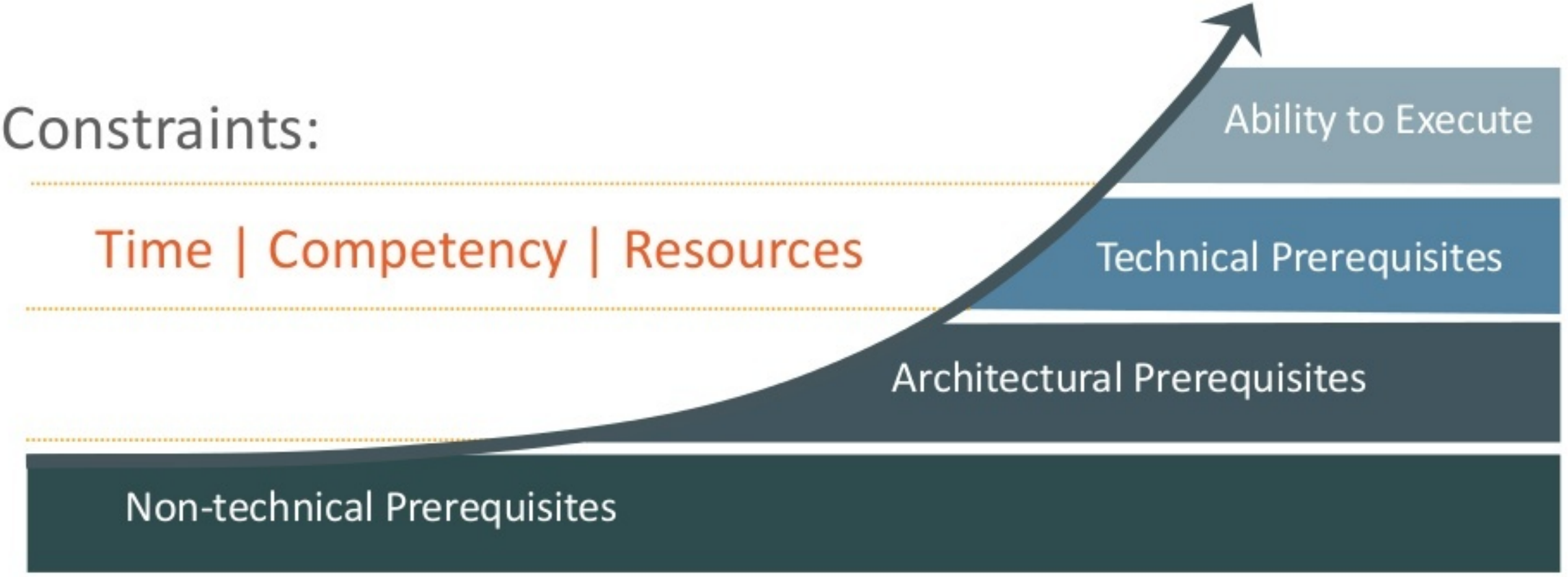
Problems to Solve With Microservices

Many, very sticky problems to solve

Microservices Adoption Patterns

Microservice Takeaways for Enterprises

The term "microservices" is probably a passing fad, but there is real business value



Remember: Microservices Are Not a Silver Bullet

It's probably best to stick with monoliths

"There is no single development, in either technology or management technique, which by itself promises even one order of magnitude [tenfold] improvement within a decade in productivity, in reliability, in simplicity."

"We cannot expect ever to see two-fold gains every two years" in software development, like there is in hardware development (Moore's law)

Integrated Cloud

Applications & Platform Services


Java™
ORACLE


ORACLE®