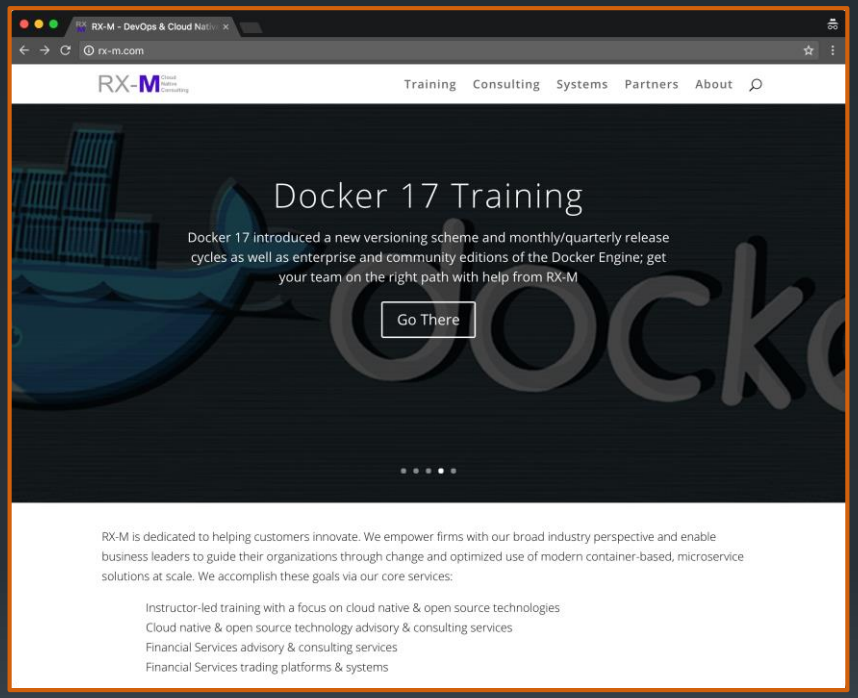
RX-M Cloud Native Consulting

Microservices

An in depth look at the microservices architecture pattern

RX-M Cloud Native Training

- Microservice Oriented
  - Microservices Foundation [3 Day]
  - Microservices on AWS [3 Day]
  - Microservices on Azure [3 Day]
  - Microservices on GCP [3 Day]
  - Microservices on Bluemix [3 Day]
  - Microservices on Oracle Cloud [3 Day]
  - Building Microservices with Go [3 Day]
  - Building Microservices with Apache Thrift [3 Day]
  - Building Microservices with gRPC [3 Day]
- Container Packaged
  - Docker Foundation [3 Day]
  - Docker Advanced [2 Day]
  - OCI [2 Day]
  - Container Technology [2 Day]
  - CRI-O [2 Day]
- Dynamically Managed
  - Docker Orch. (Compose/Swarm) [2 Day]
  - Kubernetes Foundation [2 Day]
  - Kubernetes Advanced [3 Day]
  - Kubernetes for Developers [3 Day]
  - Mesos Foundation [2 Day]
  - Nomad [2 Day]



RX-M Cloud Native Consulting

Overview

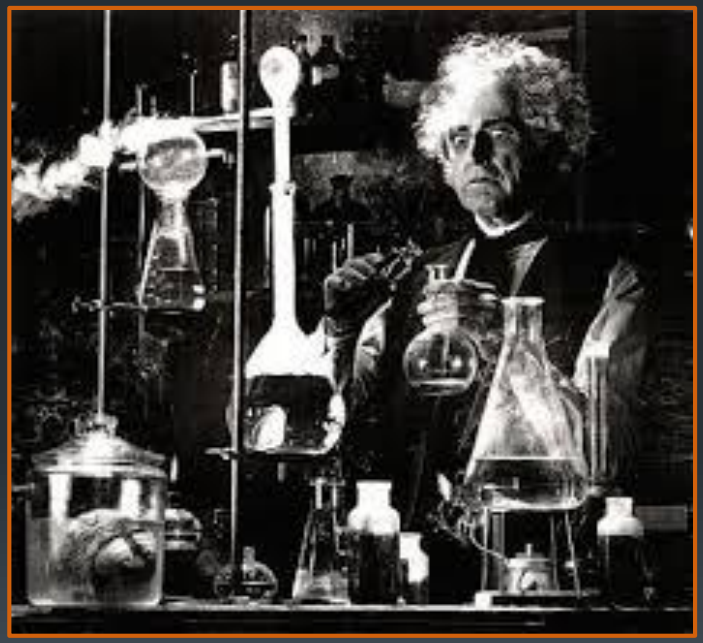
1. Microservice Overview
2. Microservice Communications I - Client/Server
3. Container Packaging
4. Microservice Communications II - Messaging
5. Cloud Native Transactions and Event Sourcing
6. Stateless Services and Polyglot Persistence
7. Microservice Orchestration
8. Gateways and Meshes
9. Serverless

Administrative Info

- Format: Lecture/Labs/Discussion
- Schedule: 9:00AM – 5:00PM
  15 minute break, AM & PM
  1 hour lunch at noon
  Lab work after each module
- Location: Fire exits, Restrooms, Security, other matters
- Attendees: Name/Role/Experience/Goals for the Course

Lecture and Lab

- Our Goals in this class are two fold:
  1. Introduce concepts and ecosystems
     - Covering concepts and where things fit in the world is the primary purpose of the lecture/discussion sessions
     - The instructor will take you on a tour of the museum
       - Like a museum tour, you should interact with the instructor (tour guide), ask questions, discuss
       - Like a museum tour, you will not have time to read the slides during the tour, instead, the instructor will discuss and point out the highlights of the slides (exhibits) which will be waiting for you to read in depth later should you like to dig deeper
  2. Impart practical experience
     - This is the primary purpose of the labs
     - Classes rarely have time for complete real world projects so think of the labs as thought experiments
       - Like hands on exhibits at the museum



Day 1 – Sateless Microservices

- Microservice Architecture
- Communications Part I - APIs
- Container Packaging

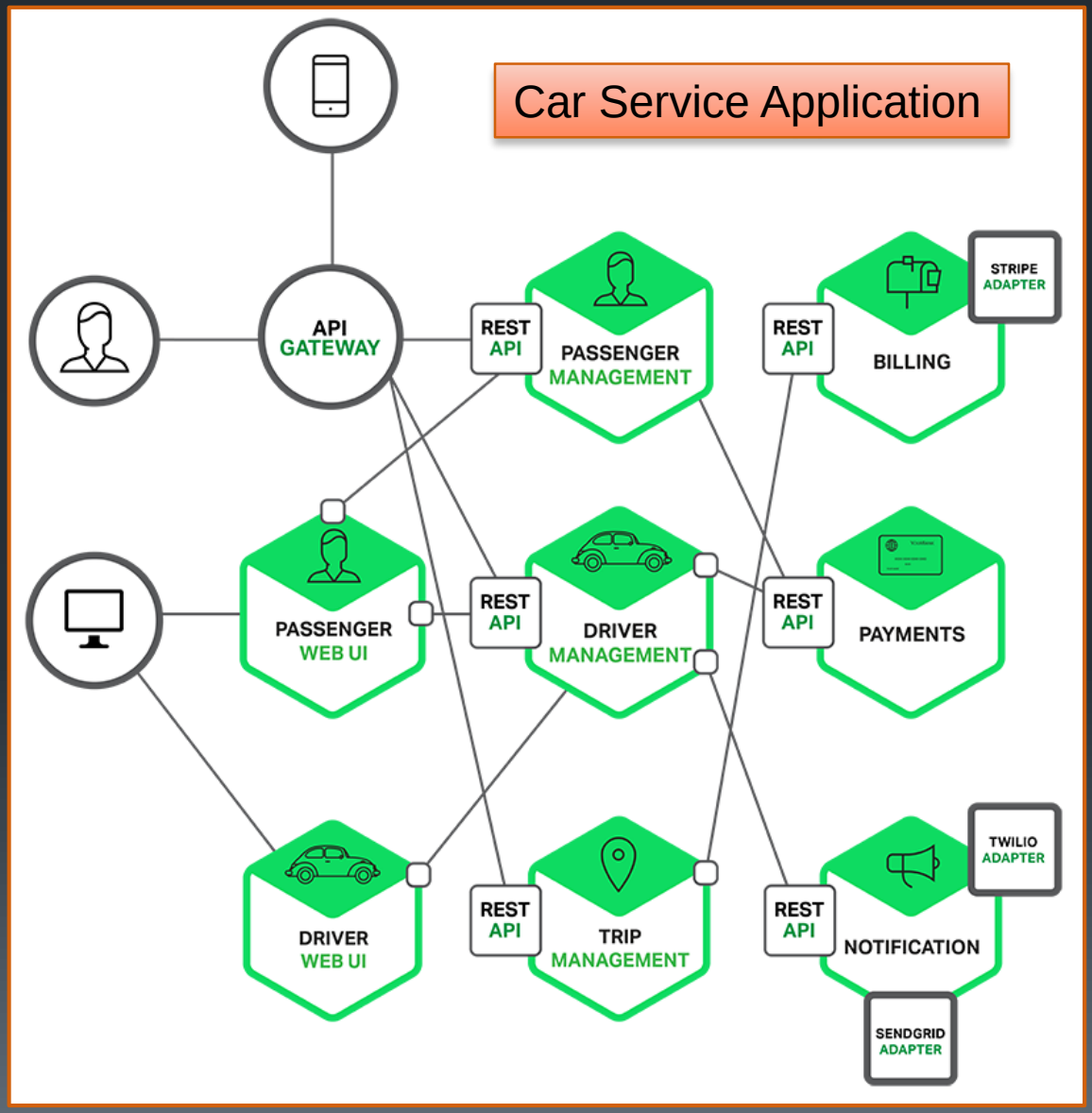
1: Microservice Overview

Objectives

- Define microservice
- Explain the microservice architecture (MSA)
- List the perceived benefits of microservices
- Explore the down sides of microservices
- Describe processes and approaches used successfully with microservices

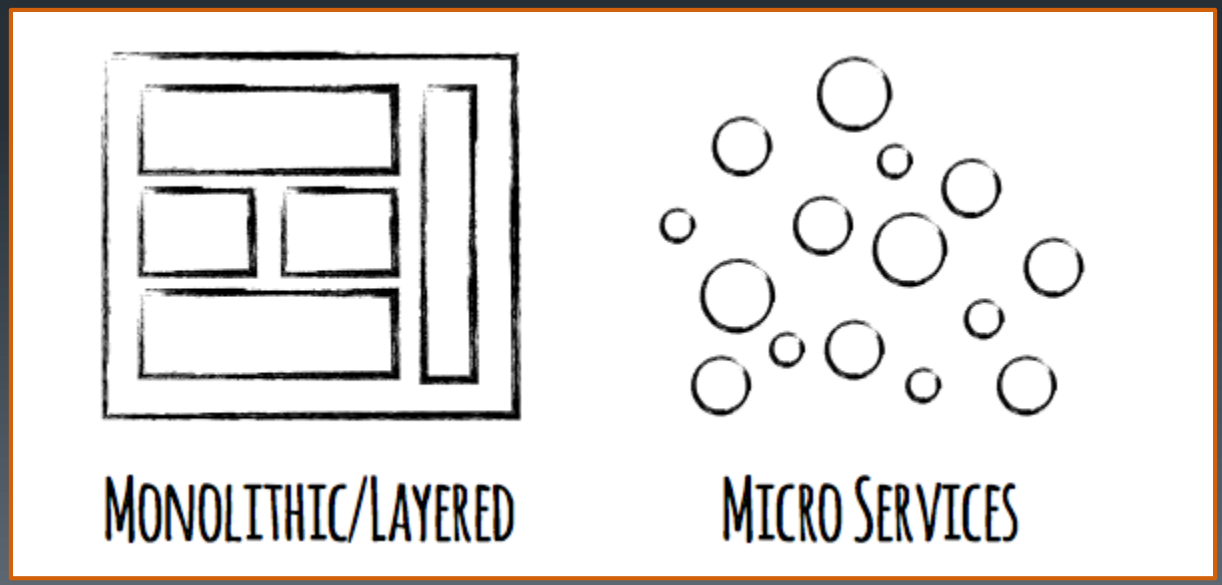
What is a microservice?

- Microservice
  - A fine grained atomically deployable service accessed via a platform agnostic network API
- Microservice Architecture (MSA)
  - An architectural pattern used to build distributed software systems
  - Processes communicate over networks
  - Composed of business aligned services
  - Supports design evolution and self organization
  - An approach favoring symmetry over hierarchy (peer to peer not layers)
  - An architectural approach that defines an application, not an enterprise
- A specialization of Service-Oriented Architectures (SOA)
  - MSA is the first realization of SOA designed to complement Agile and DevOps methodologies (both of which matured after SOA)



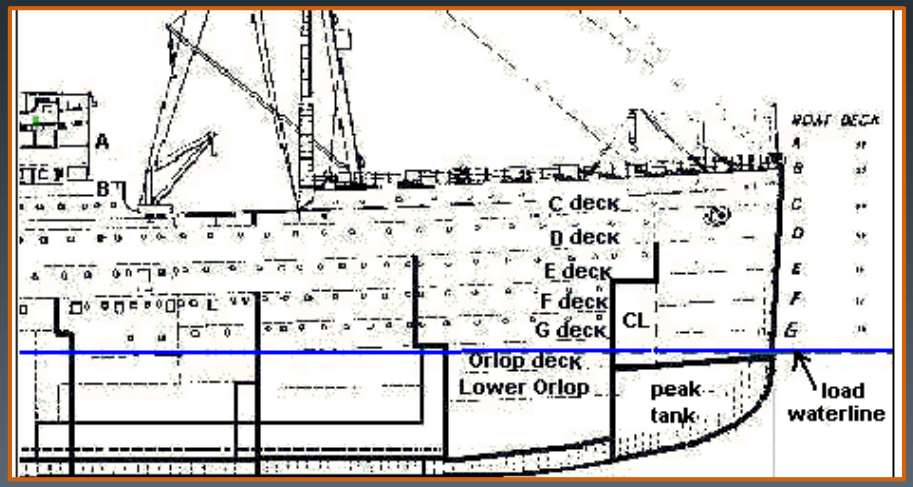
Microservice Attributes

- Small crisply defined services
- Decomposed by business/problem domain, not technology
- Well defined interfaces, completely abstract implementations
- Light-weight, technology-agnostic inter-service communications
- Polyglot friendly
- Changes are single service and incremental
- Services are atomically deployable and designed to be replaced
- Services are not changed they are replaced (and can be rolled back)
- Services are loosely coupled (discovering each other dynamically if need be)
- Single responsibility (high cohesion)
- Persistent state is encapsulated behind service interfaces and managed using cluster friendly network distributed schemes
- State is decomposed by service not centralized or shared



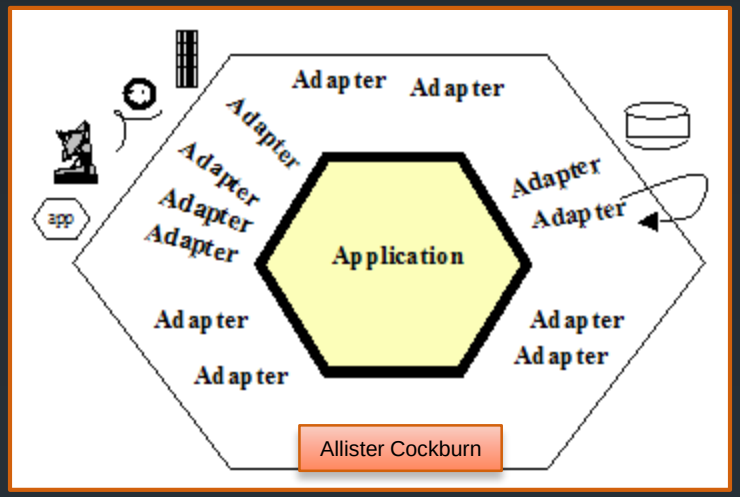
Microservices benefits

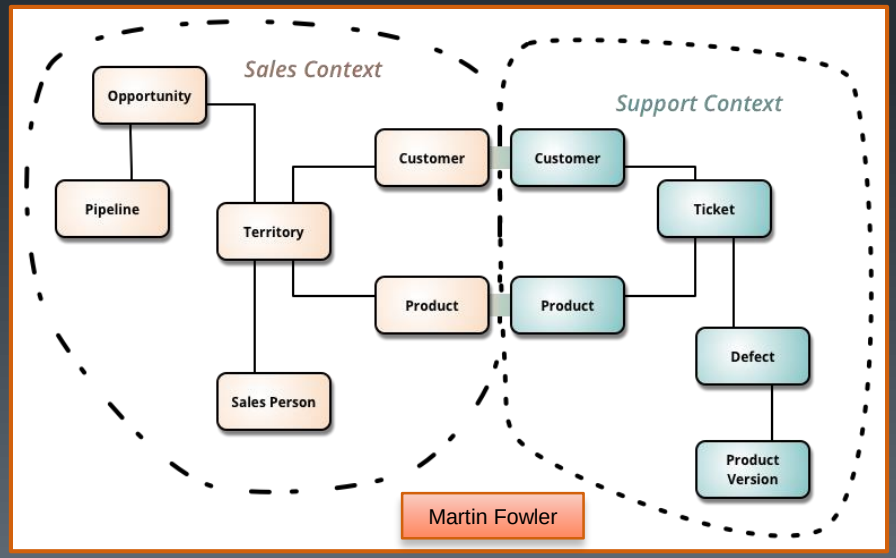
- Deliver software faster
  - A change to a microservice typically requires only the deployment of the microservice
  - Monolithic applications require large deployments for small changes
- Fine grained scaling
  - Microservices allow you to scale only the parts that need to be scaled
- Flexibility to embrace newer technologies
  - Risk is a key barrier to adopting new technologies
  - With a monolithic application a new programming language, database, or framework will impact a large amount of the system
  - With microservices this can be a small experiment and rolled back easily and without large cost if it fails
- Organizational Alignment
  - Each team can own one or more services, reducing coordination overhead
- Respond faster to change
  - Microservices are easier to rewrite wholesale, making it possible to try different ideas and track changes in business structure and patterns
- Composability
  - Microservices are atomic enough to be easily consumed by a range of clients for different purposes
- Resilience
  - When a single microservice fails only a small part of the application is inoperable
  - Enables microservice applications to operate more effectively in the face of partial failure
    - Resilience engineering bulkhead concept



Architecture Evolution

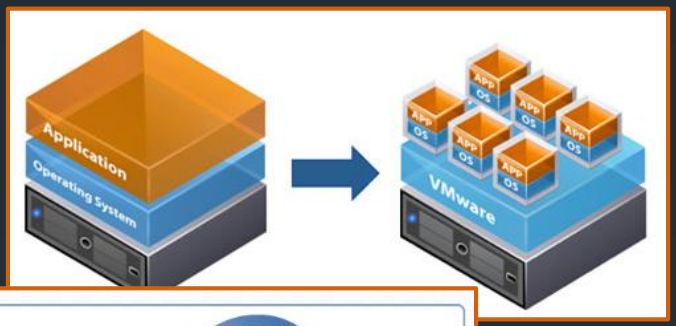
- There is no perfect system architecture
  - Only those who have built a complete system can clearly reflect upon what the perfect solution for a given problem might be
- Microservice architectures allow the right architecture to emerge naturally over time
  - While controlling risk and cost
- Microservice based systems focus on business goals
  - Services align with the business and its needs
    - Bounded Context
  - Business changes are likely to have a one to one correspondence services which need to be changed in the solution
- Minimizing service scope limits the effect of bad design decisions
  - Old services with bad designs can be replaced wholesale, even using completely different languages, frameworks and state management solutions
- Microservice interfaces can evolve in backward compatible ways
  - REST, Thrift, gRPC/ProtoBuf all allow for backward compatible interface evolution
- Interfaces that are found to be unsound can be superseded progressively
- Microservices are typically designed without either a UI or a database
  - Allows them to be easily tested
  - Allows them to work when the database becomes unavailable or changes
  - Allows applications to collaborate in varying and flexible ways, independent of UI design and evolution

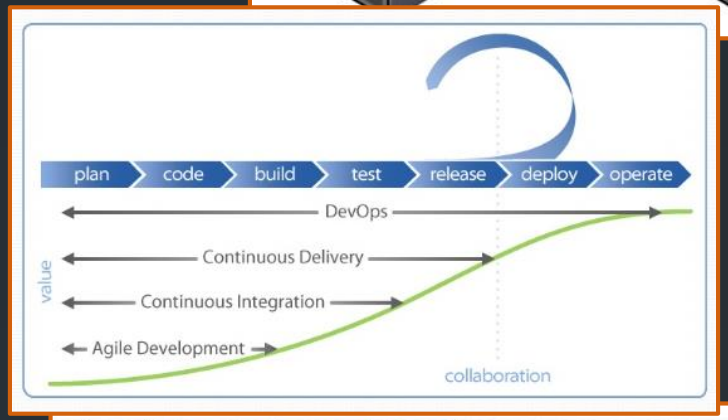


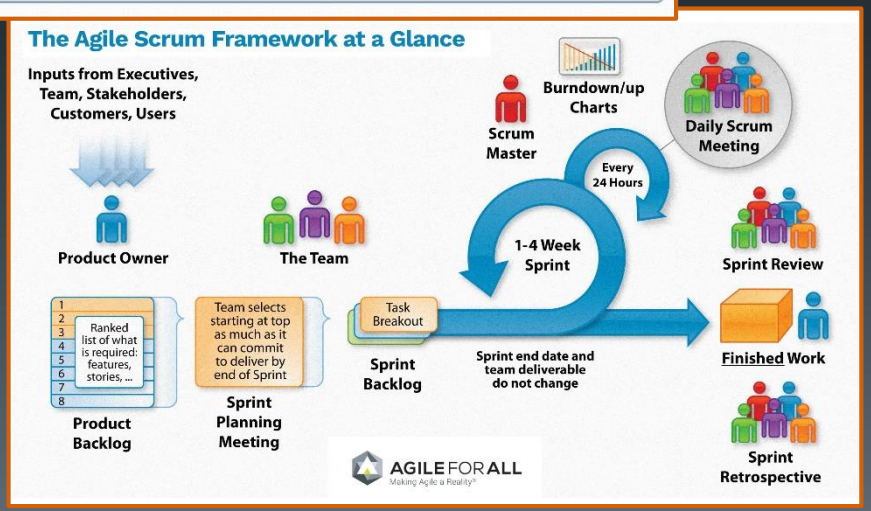


Microservice Precursors

- Service Oriented Architecture (SOA)
  - Promoted the construction of applications from networked services using vender agnostic interfaces
- Domain-Driven Design and its precursors
  - Instilled the importance of representing the real world in code
- Hexagonal Architecture
  - Allister Cockburn argues for avoiding layered architectures where business logic can hide
- Agile
  - Describes the benefits of small teams owning the full lifecycle of their services
- Continuous Integration and Delivery
  - Demonstrated the benefits of moving software to production early and often, treating every check-in as a release candidate
- Virtualization
  - Enabled dynamic provisioning of systems
- DevOps
  - Infrastructure automation (Puppet, Chef, Ansible, Salt, etc.) and the unified views of development and operations give us ways to handle dynamic infrastructure and CI/CD at scale

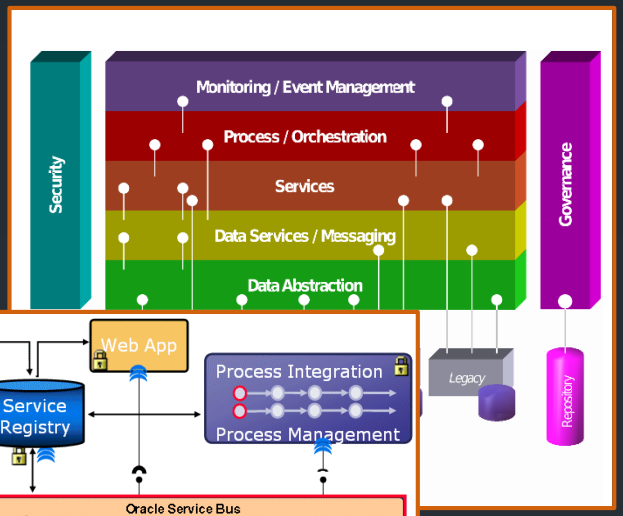


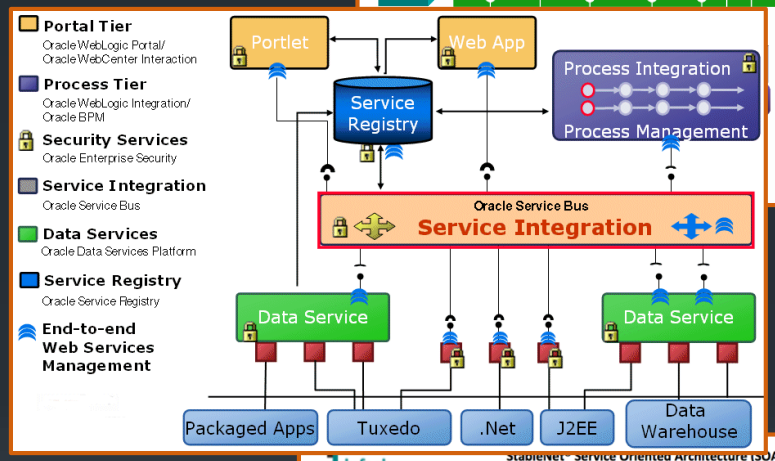


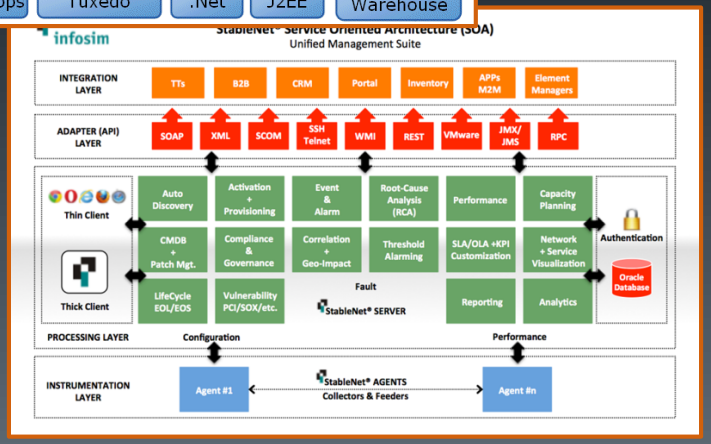


SOA

- Service-Oriented Architecture (SOA)
  - An architectural style where services are provided through a communication protocol over a network
  - A fundamental principle is independence
    - of vendors
    - of products
    - of technologies
- A service is a discrete unit of functionality that can be accessed remotely and acted upon and updated independently
  - e.g retrieving a credit card statement online
- The four properties of a service:
  1. Represents a business activity
  2. It is self-contained
  3. It is a black box for its consumers
  4. It may consist of other underlying services
- Different services can be used in conjunction to provide the functionality of a large software application
- The goal _was_ flexibility and independence
  - Many SOA solutions were heavy weight and scaled poorly
    - SOAP
      - Oasis offers >30 standards for SOAP
      - The W3C SOAP 1.2 standard is >100 pages long
    - Enterprise Service Buses
    - Layering
    - Etc.

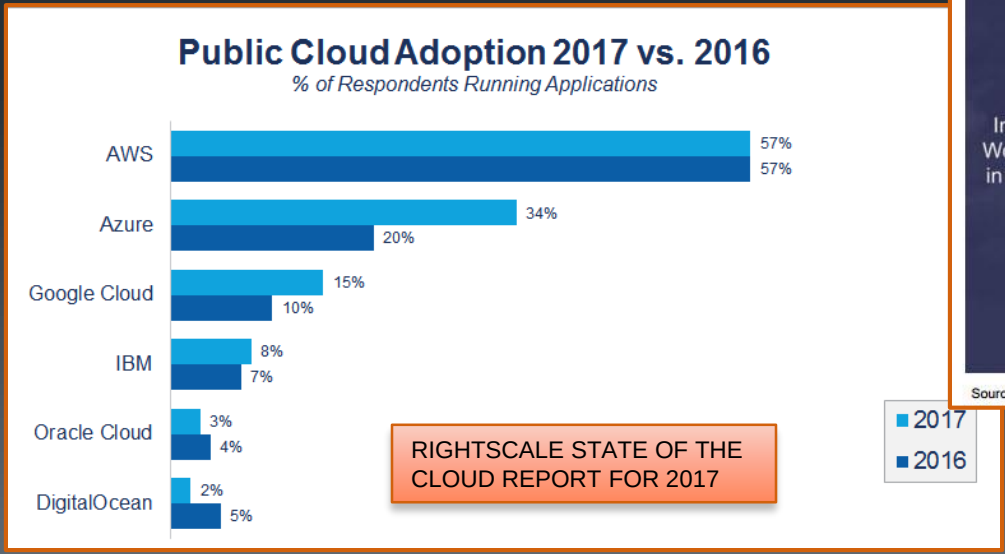


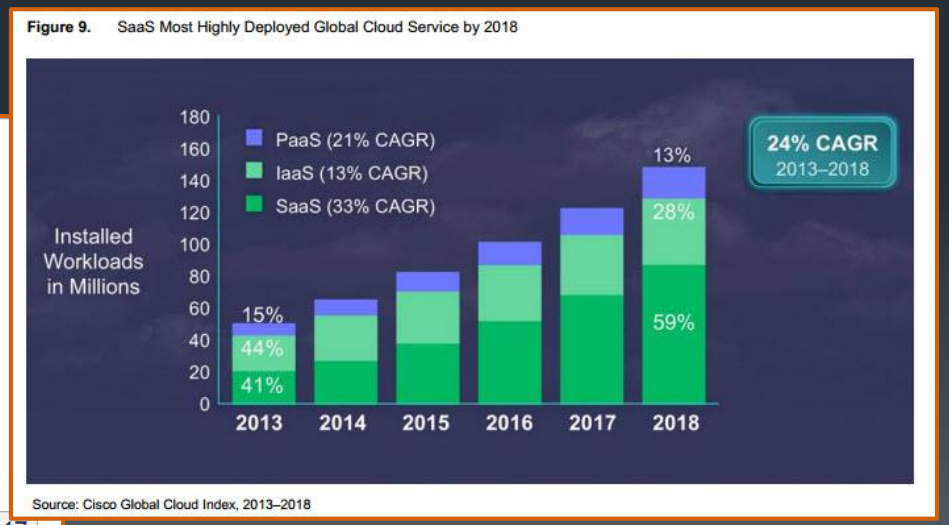




Growing dominance of the Cloud

- The 10/2015 Cisco GCI research report predicts that by 2019:
  - Global data center traffic will grow nearly 3-fold
  - Global data center traffic will reach 10.4 zettabytes per year
  - 83 percent of all data center traffic will come from the cloud
  - 4 out of 5 data center workloads will be processed in the cloud
- This mass migration to the cloud places critical challenges in from of development teams:
  1. How to build the right software for the cloud
  2. … and therefore, how clouds work and how to effectively leverage them





12 Factor Apps

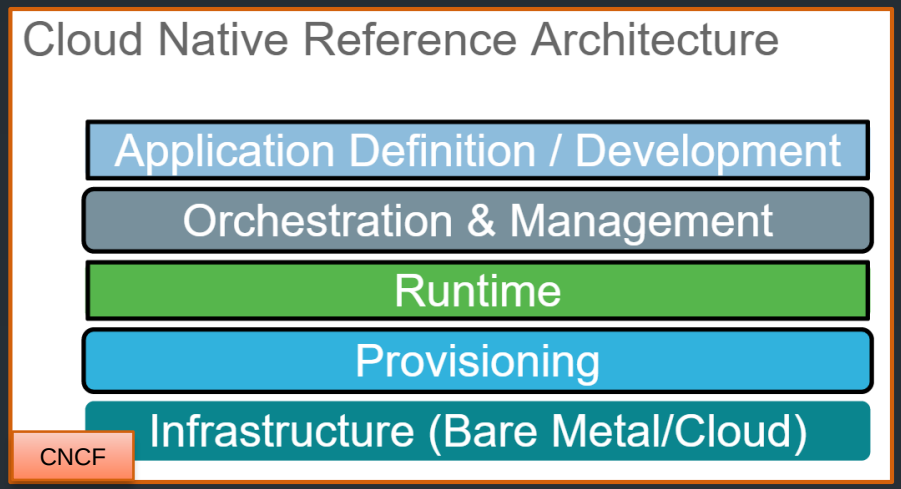
- Codebase
  - Each deployable app is tracked as one codebase in revision control (may have many deployed instances across multiple environments)
- Dependencies
  - An app explicitly declares and isolates dependencies via appropriate tooling (e.g., Maven, Bundler, NPM) rather than depending on implicit dependencies in its deployment environment
- Config
  - Configuration, or anything that is likely to differ between deployment environments (e.g., development, staging, production) is injected via operating system-level environment variables
- Backing services
  - Backing services, such as databases or message brokers, are treated as attached resources and consumed identically across all environments
- Build, release, run
  - The stages of building an artifact, combining that artifact with configuration, and starting one or more processes from that artifact/configuration combination, are strictly separated
- Processes
  - The app executes as one or more stateless processes (e.g., master/workers) that share nothing, state is externalized to backing services (cache, object store, etc.)
- Port binding
  - The app is self-contained and exports any/all services via port binding (including HTTP)
- Concurrency
  - Concurrency is usually accomplished by scaling out app processes horizontally (though processes may also multiplex work via internally managed threads if desired)
- Disposability
  - Robustness is maximized via processes that start up quickly and shut down gracefully, these aspects allow for rapid elastic scaling, deployment of changes, and recovery from crashes
- Dev/prod parity
  - Continuous delivery and deployment are enabled by keeping development, staging, and production environments as similar as possible
- Logs
  - Rather than managing logfiles, treat logs as event streams, allowing the execution environment to collect, aggregate, index, and analyze the events via centralized services
- Admin processes
  - Administrative or managements tasks, such as database migrations, are executed as one-off processes in environments identical to the app’s long-running processes

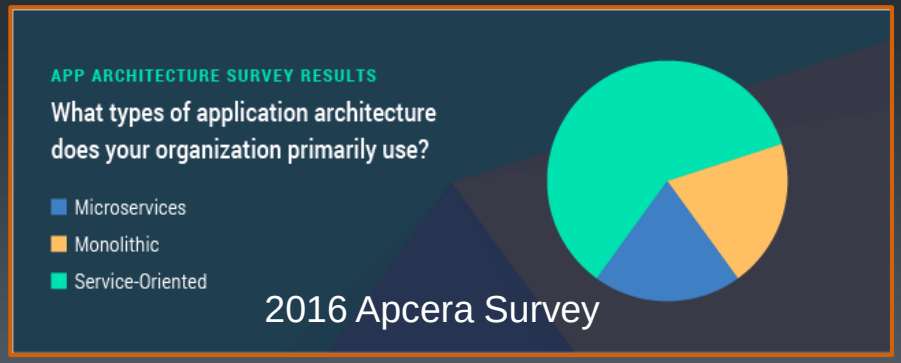
12 Factor Apps

- Codebase
  - Each deployable app is tracked as one codebase in revision control (may have many deployed instances across multiple environments)
  - Microservices: SAME
- Dependencies
  - An app explicitly declares and isolates dependencies via appropriate tooling (e.g., Maven, Bundler, NPM) rather than depending on implicit dependencies in its deployment environment
  - Microservices: DIFF This conflates Build Operations with Distribution. In a PaaS you push code in a CaaS you push container images. Local activities use one build and PaaS activities use another. The image history should be immutable and no build should be required to recall any prior image for dev, testing or production deployment.
- Config
  - Configuration, or anything that is likely to differ between deployment environments (e.g., development, staging, production) is injected via operating system-level environment variables
  - Microservices: DIFF Microservices should rely principally on dynamic discovery mechanisms (DNS, etc.) for environmental variations.
- Backing services
  - Backing services, such as databases or message brokers, are treated as attached resources and consumed identically across all environments
  - Microservices: SAME
- Build, release, run
  - The stages of building an artifact, combining that artifact with configuration, and starting one or more processes from that artifact/configuration combination, are strictly separated
  - Microservices: DIFF Building and packaging operations should be atomic with microservices, an unpackaged service is not deployable/testable/etc.
- Processes
  - The app executes as one or more stateless processes (e.g., master/workers) that share nothing, state is externalized to backing services (cache, object store, etc.)
  - Microservices: SAME
- Port binding
  - The app is self-contained and exports any/all services via port binding (including HTTP)
  - Microservices: DIFF Containerized microservices can listen on any port they desire and benefit from consistant port usage across replicas
- Concurrency
  - Concurrency is usually accomplished by scaling out app processes horizontally (though processes may also multiplex work via internally managed threads if desired)
  - Microservices: SAME
- Disposability
  - Robustness is maximized via processes that start up quickly and shut down gracefully, these aspects allow for rapid elastic scaling, deployment of changes, and recovery from crashes
  - Microservices: SAME
- Dev/prod parity
  - Continuous delivery and deployment are enabled by keeping development, staging, and production environments as similar as possible
  - Microservices: SAME
- Logs
  - Rather than managing logfiles, treat logs as event streams, allowing the execution environment to collect, aggregate, index, and analyze the events via centralized services
  - Microservices: SAME
- Admin processes
  - Administrative or managements tasks, such as database migrations, are executed as one-off processes in environments identical to the app's long-running processes
  - Microservices: SAME (Blue/Green approach)

Next Generation Models

- Cloud Native Computing
  - A new computing paradigm optimized for modern distributed systems environments
  - Capable of scaling to tens of thousands of self healing multi-tenant nodes
- Cloud native systems have the following properties:
  - Container packaged
    - Running applications and processes in software containers as an isolated unit of application deployment, and as a mechanism to achieve high levels of resource isolation
    - Improves overall developer experience, fosters code and component reuse and simplify operations for cloud native applications
  - Dynamically managed
    - Actively scheduled and actively managed by a central orchestrating process
    - Radically improve machine efficiency and resource utilization while reducing the cost associated with maintenance and operations
  - Micro-services oriented
    - Loosely coupled with dependencies explicitly described (e.g. through service endpoints)
    - Significantly increase the overall agility and maintainability of applications



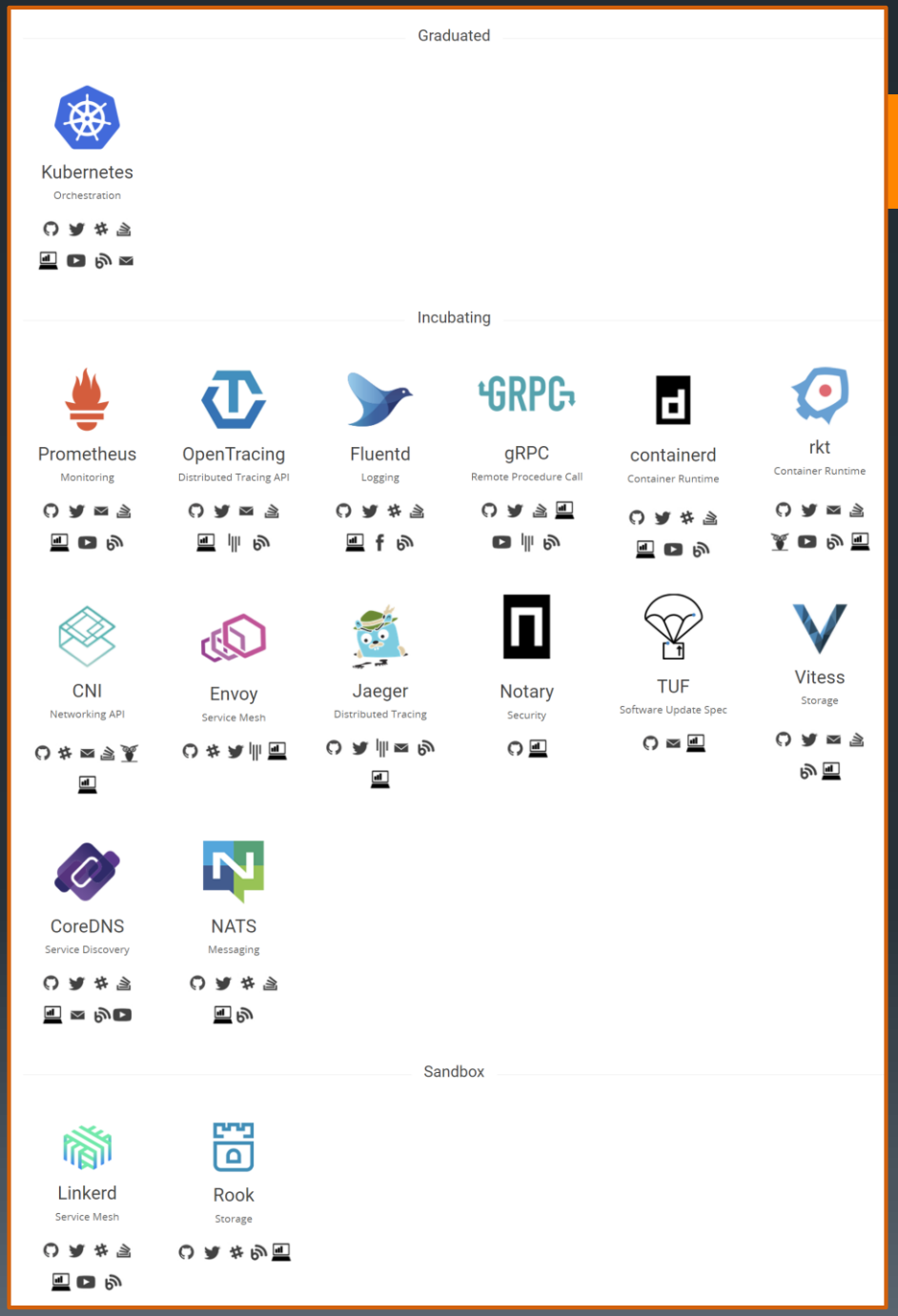


Cloud Native System Traits

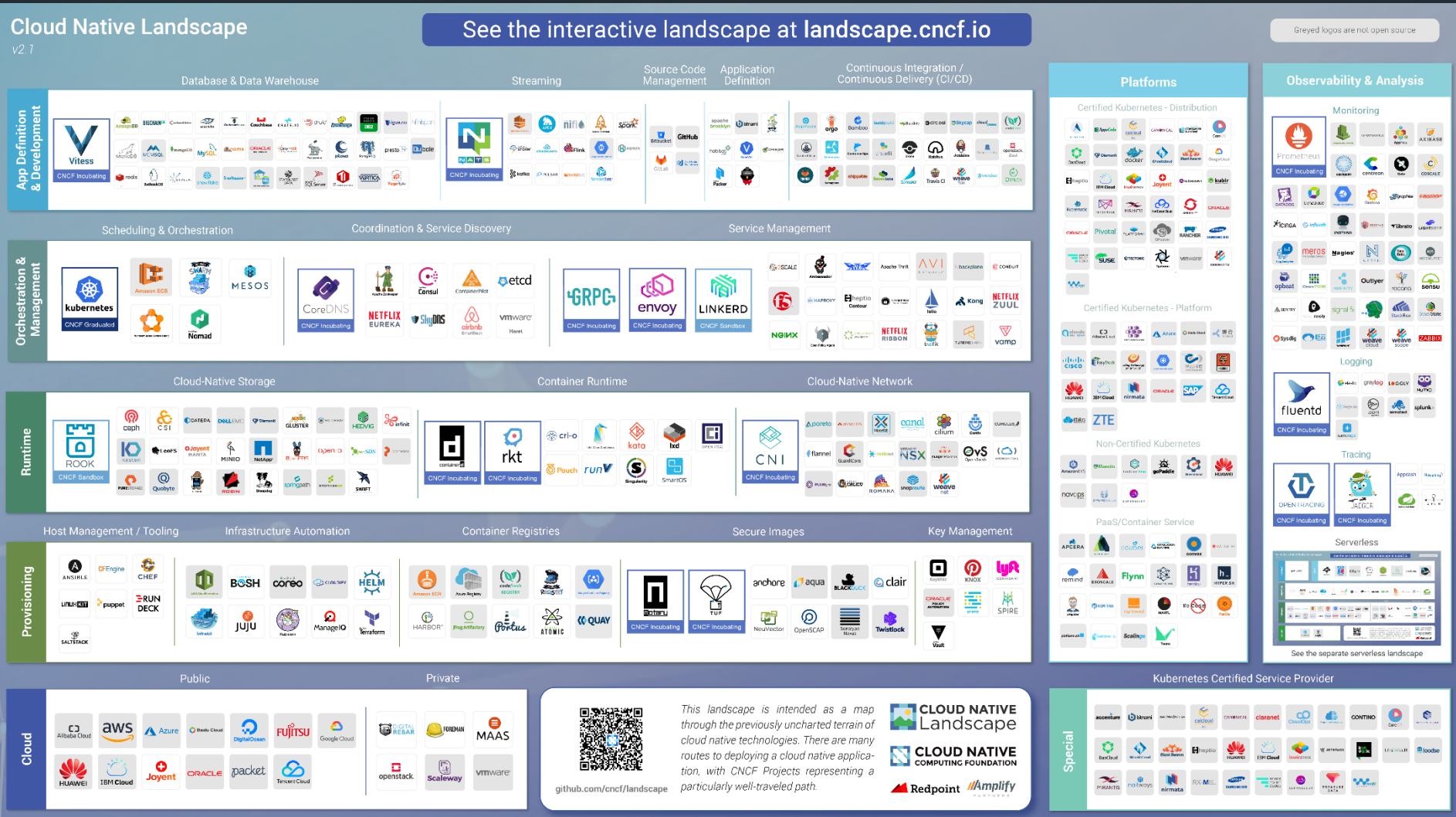
- In 2018 the CNCF refocused the official CNCF definition of cloud native systems of a more abstract set of attributes
  - Operability: Expose control of application/system lifecycle
  - Observability: Provide meaningful signals for observing state, health, and performance
  - Elasticity: Grow and shrink to fit in available resources and to meet fluctuating demand
  - Resilience: Fast automatic recovery from failures
  - Agility: Fast deployment, iteration, and reconfiguration.

Container Orchestration Initiatives

- CNCF [circa 7/2015] http://cncf.io
  - Cloud Native Computing Foundation
    - Hosted by the Linux Foundation
  - Cloud Native Applications are: container packaged, dynamically managed and micro-services oriented
  - Mission: To create and drive the adoption of a new set of common container technologies informed by technical merit and end user value, and inspired by Internet-scale computing
  - Aims to host the reference stack of technologies around container orchestration
    - Google contributed Kubernetes to the kick off the foundation, more recent additions:
      - Prometheus, Fluentd, CoreDNS, Linkerd, Jaeger, OpenTracing, gRPC, Containerd, rkt, Envoy, CNI, Notary, TUF, Vitess, NATS, Rook
  - 80+ Members including:
    - Cisco, CoreOS, Docker, Fujitsu, Google, Huawei, IBM, Intel, Joyent, Mesosphere, Red Hat, Samsung SDS, Supernap, AT&T, NetApp Amihan, Apcera, Apigee Aporeto, Apprenda, AVI Networks, Caicloud, Canonical, Capital One, Centrify, ChaoSuan, Chef, Cloudsoft, Container Solutions, Crunchy, Dao Cloud, Deis, Digital Ocean, Easy Stack, eBay, Eldarion, Exoscale, Galactic Fog, Goldman Sachs, Gronau, Heptio, Iguaz.io, Infoblox, Juniper, Livewyer, Loodse, Minio, Mirantis, NCSoft, NEC, Nginx, Packet, Plexistor, Portworx, Rancher, RX-M, Stack Point Cloud, Storage OS, Sysdig, Tigera, Treasure Data, Twistlock, Twitter, Univa, Virtuozzo, Vmware, Weaveworks, Box, Ticketmaster, Zhejiang University

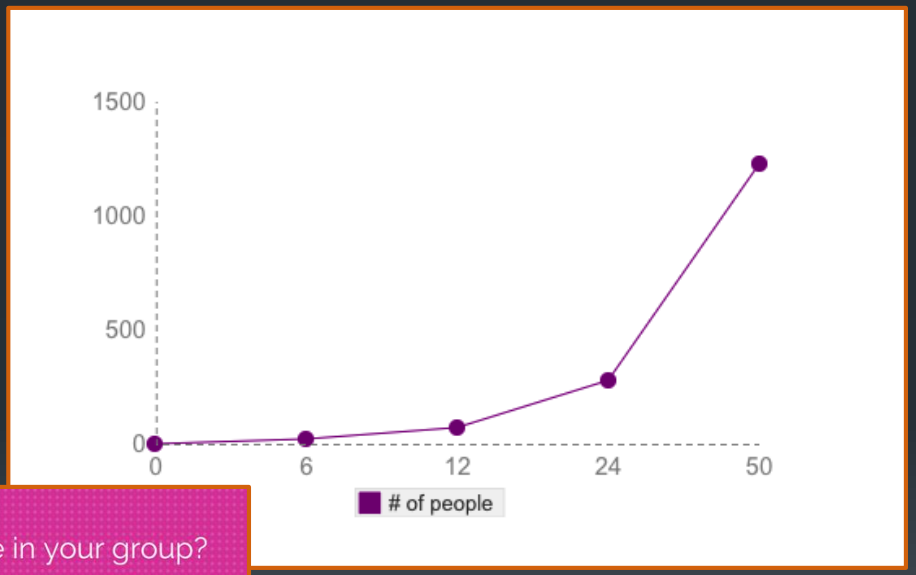


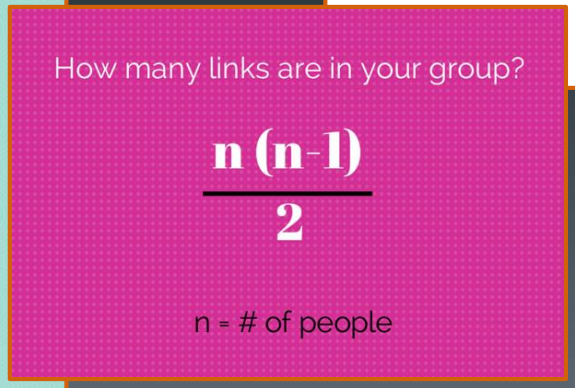
Players in the space



How micro is micro

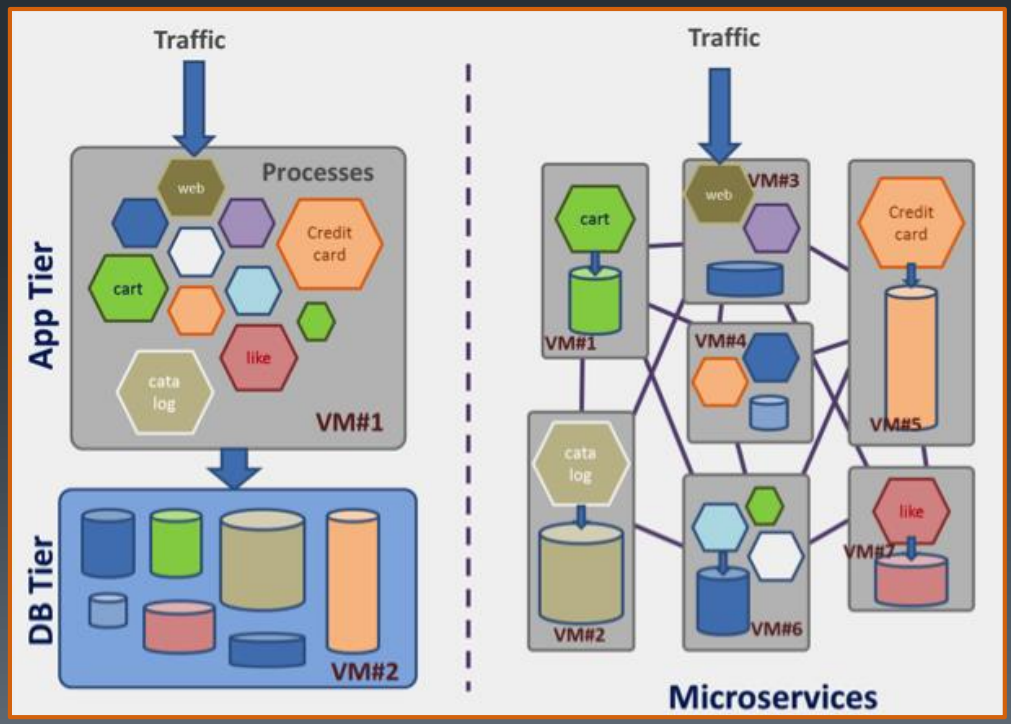
- You should be able to rewrite a microservice from scratch in 2 weeks
- A microservice should have a single crisp responsibility
- A microservice should be owned by a single team





Microservice challenges

- Explosion in the number of processes to manage
  - Requires reliable deployment solutions and automated dynamic orchestration to manage
- New and not well understood by many teams
  - Easy to slip into old habits making things worse not better
- Heavy network utilization and increased latency
  - Microservices communicate through out-of-process network interfaces
  - Slower, perhaps much slower, that in process function calls, though mitigated by async communications in some cases
- Small to medium applications may be harder to deploy and manage
  - No transactions, no single integration database
- Integration is no longer anyone in development's problem
  - Isolating teams within a single process may lead to poor integration practices in development making life harder for test and production



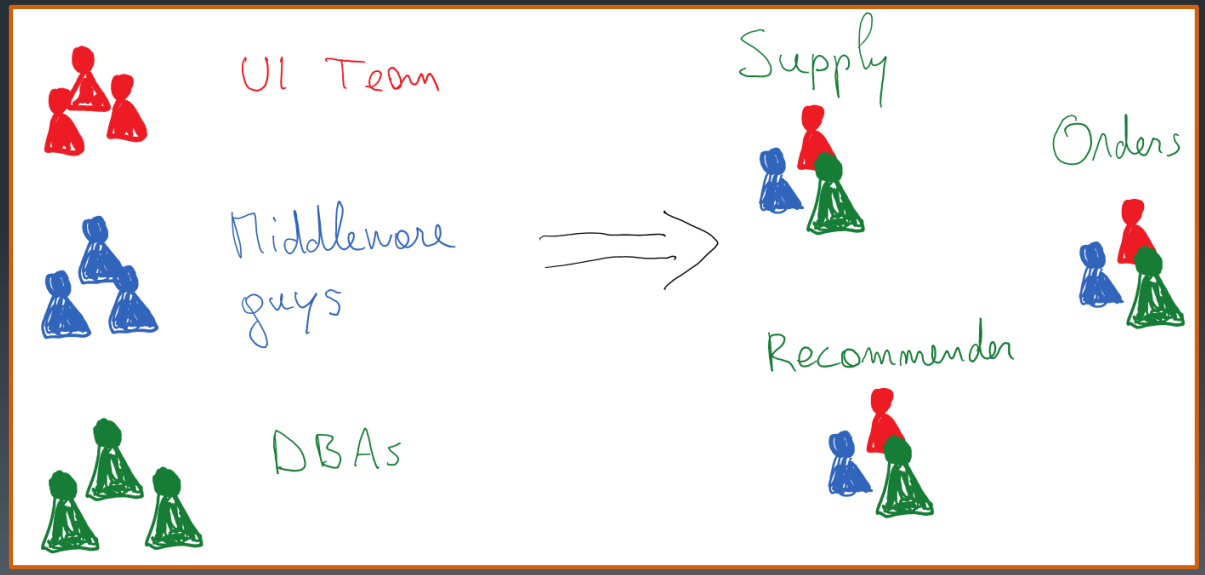
Anti-patterns

- Packing multiple services onto the same package
  - Single service packages can be cost prohibitive with VMs but works well with containers
- Too much cross service code sharing
  - Services can become coupled to internal representations and/or code
  - Decreases autonomy and requires additional coordination when making changes
- Too much polyglotism
  - Too many languages, stores or platform tool choices can make operations costly and problematic
- Shared database between different services
  - Creates tight coupling between services
  - Schema updates force updates across services
- Hard Coded Endpoints
  - Services should discover one another
- Transactions (particularly distributed)
  - Transactions require tight coupling of components
  - Scalable, loosely coupled distributed applications use eventual consistency, atomic updates and other approaches to consistency
- Persistent messaging
  - Message transactions and/or delivery guarantees
- Service instance identity
  - Microservices should implement a service, not embody it
  - End points should support elastic, scalable, multi-instance services
- Selective message delivery
  - Implies service instance identity and dependency

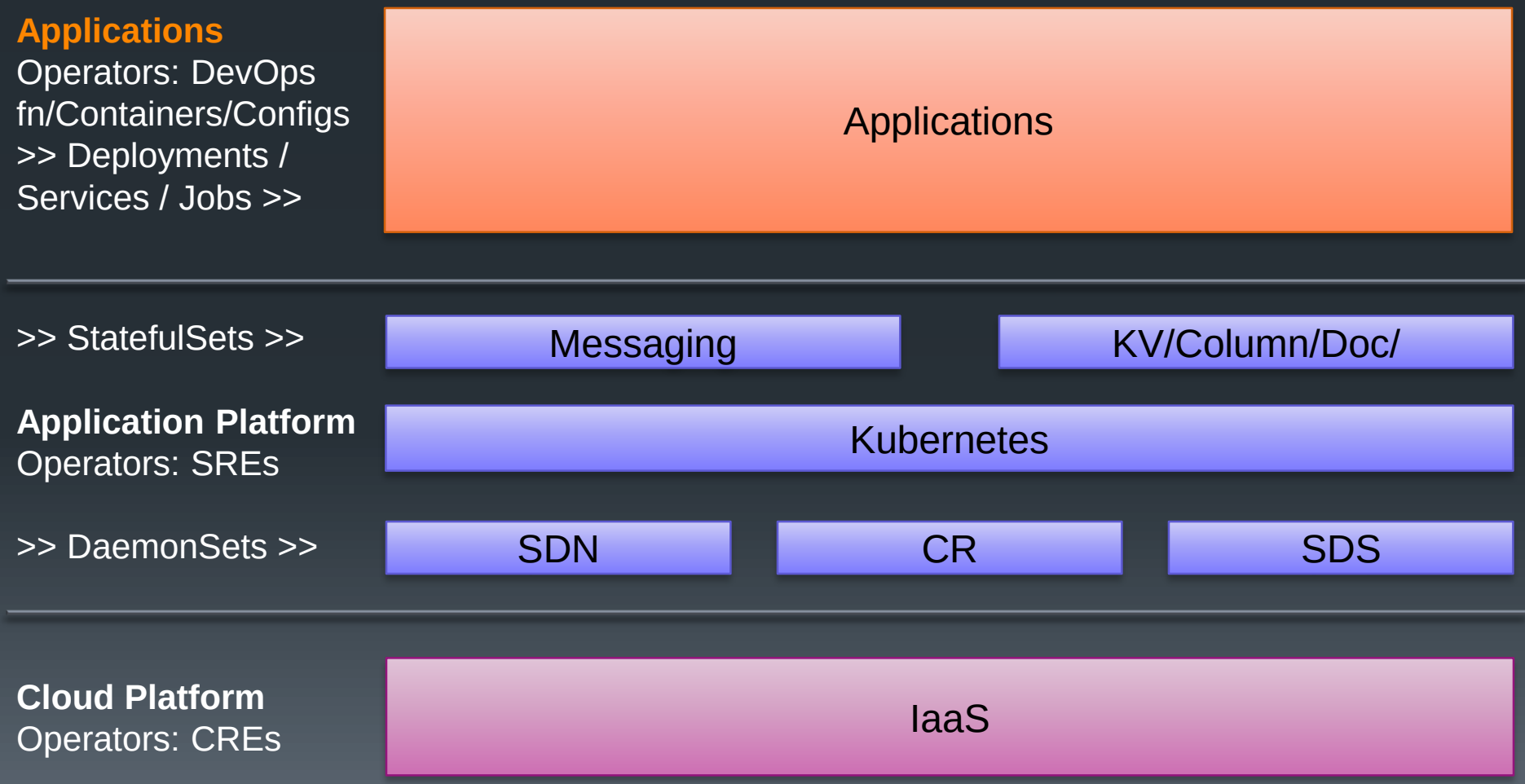
Microservices in Practice

- Evolution at 3 major tech firms:
- eBay
  - 5th generation today
    - Monolithic Perl
    - Monolithic C++
    - Java
    - microservices
- Twitter
  - 3rd generation today
    - Monolithic Rails
    - JS / Rails / Scala
    - microservices
- Amazon
  - Nth generation today
    - Monolithic C++
    - Perl / C++
    - Java / Scala
    - microservices

Very small autonomous teams achieve great things



Division of Labor in a Cloud Native World



Summary

- Microservice Architecture refers to an architectural pattern derived from SOA but focused on small single responsibility services with ubiquitous network based interfaces
- While microservices offer a range of benefits they have proven particularly powerful for organizations seeking:
  - Extreme scale
  - Low technology lock in and ability to adopt new productive tools dynamically
  - Rapid CI/CD support
- Microservice also carry risks, particularly when incomplete or inappropriate processes, tools or designs are applied

Lab 1

- Configuring a microservice development environment and creating a hello world microservice

2: Communication I

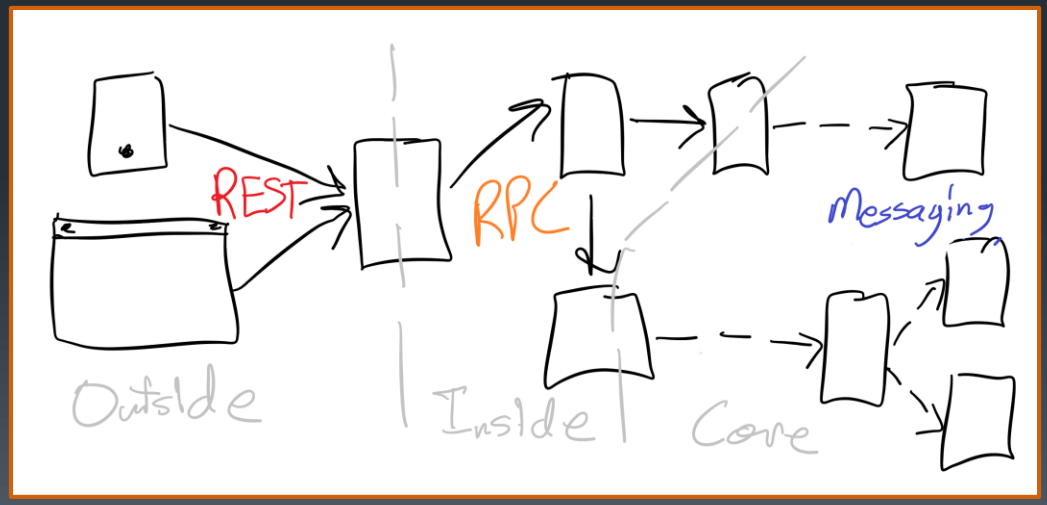
Objectives

- Describe the range on microservice communication schemes
- List the Request/Response style service communications types
- Explore interface evolution
- Define RESTful interface
- Examine REST in detail
- Examine RPC in Detail

Communications Options

- Communications are the lifeblood of microservices
- Schemes
  - Request/Response
    - REST
    - RPC
      - gRPC/ProtoBuf
      - Apache Thrift
  - Messaging
    - Broadcast
    - Pub/Sub (aka multicast)
      - Kafka
      - Nats
    - Anycast
    - Unicast
  - Streaming
    - Characterized by a continuous flow of data from a server to a connected client

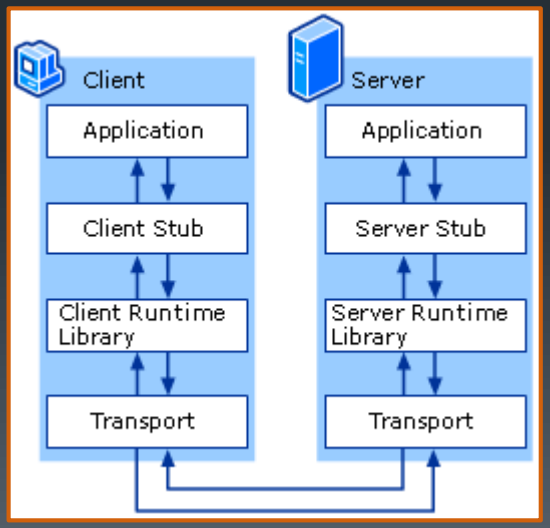
Each scheme has unique characteristics and requirements



Request/Response

- REST
  - Language/platform agnostic
  - Many supporting frameworks
  - Two main data formats
    - XML
    - JSON
- RPC
  - Long standing approach to distributed computing
  - Extends the concept of an in process function call across a network
  - Several modern schemes allow evolution and are suitable for microservice development
    - gRPC and Protocol Buffers
    - Apache Thrift
    - JSON RPC

Resource	POST create	GET read	PUT update	DELETE delete
/dogs	create a new dog	list dogs	bulk update dogs	delete all dogs
/dogs/1234	error	show Bo	if exists update Bo if not error	delete Bo



HTTP

- The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems and is the foundation of data communication for the World Wide Web
- The HTTP standard was coordinated by the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C)
  - RFC 2616 (June 1999) defines HTTP/1.1, the version in common use
- HTTP functions as a request-response protocol in the client-server computing model
  - The client submits an HTTP request message to the server
    - A web browser is typically the client
  - The server returns a response message to the client
    - The server provides resources such as HTML files or performs other functions on behalf of the client
    - The response contains completion status information about the request and may also contain requested content in its message body
- Part of HTTPs appeal is that it is simple
  - You can code a decent Java or C++ HTTP server in a day …
  - But you don't need to because there are tens already written to choose from

HTTP Request Methods

- GET

Requests a representation of the specified resource

- HEAD

Asks for the response but without the body

- POST

Requests that the server accept the entity enclosed in the request as a new subordinate of the web resource identified by the URI

- PUT

Requests that the enclosed entity be stored under the supplied URI

- DELETE

Deletes the specified resource

- TRACE

Echoes back the received request so that a client can see what (if any) changes or additions have been made by intermediate servers

- OPTIONS

Returns the HTTP methods that the server supports for the specified URL

- CONNECT

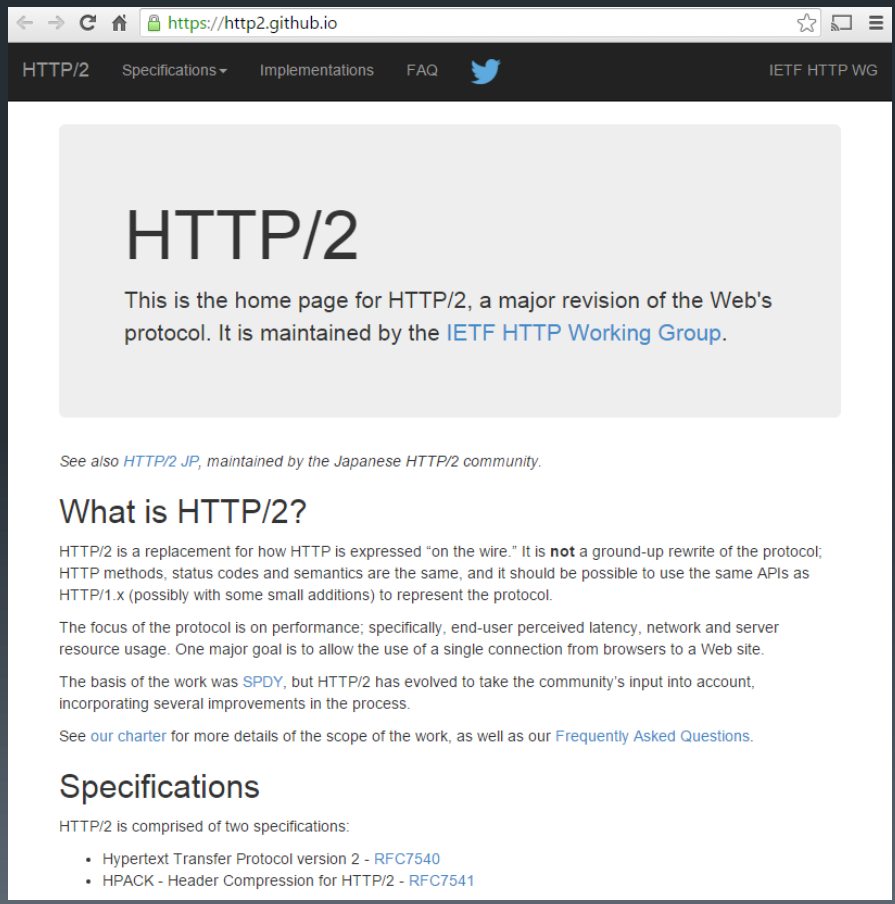
Converts the request connection to a transparent TCP/IP tunnel

- PATCH

Used to apply partial modifications to a resource

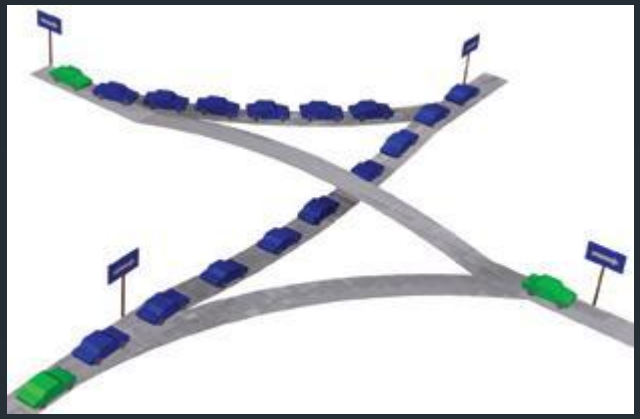
HTTP Evolution

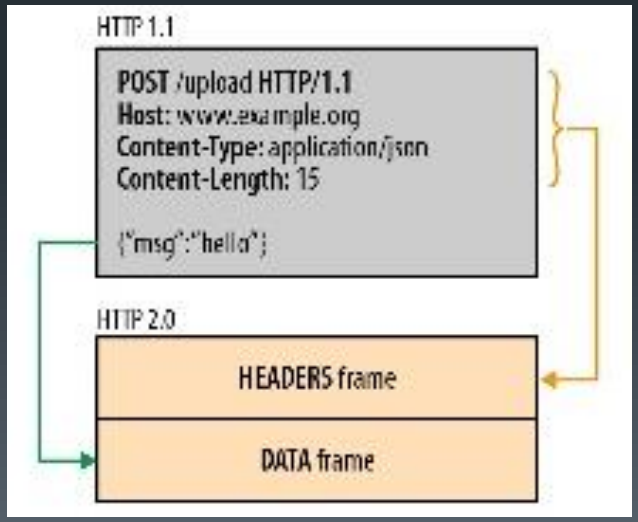
- The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems
- Standards development of HTTP is coordinated by the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C) through the RFC (Requests for Comments) process
- Timeline
  - 1989: Tim Berners-Lee proposes the "WorldWideWeb" project
  - 1990: HTTP becomes the foundation of data communication for the World Wide Web
  - 1991: HTTP v0.9 is established ad hoc (GET only)
  - 1995: Dave Raggett leads HTTP Working Group (HTTP WG) to expand the protocol with extended operations, extended negotiation, richer meta-information, tied with a security protocol
  - 1996: RFC 1945 establishes the HTTP V1.0 standard
  - 1997: RFC 2068 defined HTTP/1.1
  - 1999: RFC 2616 updates 2068 and is the version in common use
  - 2007: HTTPbis Working Group formed to revise and clarify the HTTP/1.1 specification
  - 2014- 06: HTTPbis WG releases six-part specification obsoleting RFC 2616:
    - RFC 7230, HTTP/1.1: Message Syntax and Routing
    - RFC 7231, HTTP/1.1: Semantics and Content
    - RFC 7232, HTTP/1.1: Conditional Requests
    - RFC 7233, HTTP/1.1: Range Requests
    - RFC 7234, HTTP/1.1: Caching
    - RFC 7235, HTTP/1.1: Authentication
  - 2015-05: HTTP/2 published as RFC 7540



HTTP/2

- Problems with old HTTP
  - No support for concurrent requests
    - HTTP/1.0 allowed only one request to be outstanding at a time on a given TCP connection
    - HTTP/1.1 added request pipelining to allow multiple outstanding requests, but still suffered from head-of-line blocking
    - HTTP/1.0 and HTTP/1.1 clients that need to make many requests in parallel use multiple connections to a server in order to achieve concurrency and reduce latency
  - HTTP header fields are often repetitive and verbose
    - This causes unnecessary network traffic and fills the initial TCP congestion window quickly, often resulting in excessive latency when multiple requests are made on a new TCP connection
- HTTP/2 solutions
  - HTTP/2 supports all of the core features of HTTP/1.1 but aims to be more efficient in several ways
  - HTTP/2 allows interleaving of request and response messages on the same connection and uses an efficient coding for HTTP header fields
    - Better network utilization because fewer network connections are required and connections live longer
  - HTTP/2 allows prioritization of requests, letting more important requests complete more quickly, improving performance
  - HTTP/2 enables more efficient processing of messages through use of binary message framing
    - Supporting server push and flow control
- What about WebSocket?!?
  - … oh yeah that … WebSocket integration with HTTP/2 was overlooked and does not exist today
  - You can upgrade from HTTP 1.1 to WebSocket or to HTTP/2
  - It appears the HTTP/2 standard will be amended to support multiple WebSocket channels concurrently with HTTP requests over a single TCP connection using HTTP/2 framing





REST

- The term representational state transfer was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation at UC Irvine (http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm)
- Representational state transfer (REST) is an architectural style
  - REST is a style not a protocol, principal protocol developed along REST guidelines: HTTP
  - While HTTP supports REST it does not enforce it
    - e.g. many GET requests are implemented in such a way as to change state on the server
- REST ignores the details of component implementation and protocol syntax in order to focus on the roles of these, and their interpretation of significant data elements
- REST has been used to describe desired web architecture, to identify existing problems, to compare alternative solutions, and to ensure that protocol extensions would not violate the core constraints that make the Web successful
  - Much like a design guide applied to a commercial software project
- Fielding used REST to design HTTP 1.1 and Uniform Resource Identifiers (URI)
- The Web is a REST system

The properties of REST are:

- Performance
- Scalability of component interactions
- Simplicity of interfaces
- Modifiability of components to meet changing needs (even while the application is running)
- Visibility of communication between components by service agents
- Portability of component deployment
- Reliability

REST Constraints

- Client–server
  - A uniform interface separates clients from servers
  - Clients are not concerned with data storage
  - Servers are not concerned with the user interface or user state
- Stateless
  - No client context is stored on the server between requests
  - Each request from any client contains all of the information necessary to service the request
  - Session state is held in the client, though session state can be transferred by the server to another service such as a database to maintain a persistent state for a period of time and allow authentication
  - The client sends requests when it is ready to make the transition to a new state
  - While one or more requests are outstanding, the client is considered to be in transition
- Cacheable
  - Clients can cache responses
  - Responses must implicitly or explicitly define themselves as cacheable, or not, to prevent clients reusing stale or inappropriate data in response to further requests
  - Well-managed caching partially or completely eliminates some client–server interactions
- Layered system
  - A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary
  - Intermediary servers may improve system scalability by enabling load-balancing and by providing shared caches
- Code on demand (optional)
  - Servers can temporarily extend or customize the functionality of a client by the transfer of executable code
- Uniform Interface
  - The uniform interface simplifies and decouples the architecture

RESTful Web Services

- A RESTful web service is a web API implemented using HTTP and REST principles
- It provides a collection of resources, with four defined aspects:
  1. The base URI for the web API, such as http://example.com/resources/
  2. The Internet media type of the data supported by the web API
     - This is often JSON but can be any other valid Internet media type provided that it is a valid hypertext standard
  3. The set of operations supported by the web API using HTTP methods (e.g., GET, PUT, POST, or DELETE).
  4. The API must be hypertext driven
- PUT and DELETE methods are idempotent
- GET is a safe method (or nullipotent), calling it produces no side-effects
- Unlike SOAP-based web services, there is no "official" standard for RESTful web APIs
- REST is an architectural style, unlike SOAP, which is a protocol
- Even though REST is not a standard, a RESTful implementation such as the Web typically uses standards like HTTP, URI, JSON, etc.

RESTful web API HTTP methods

Resource	GET	PUT	POST	DELETE
Collection URI, such as `http://example.com/resources`	**List** the URIs and perhaps other details of the collection's members.	**Replace** the entire collection with another collection.	**Create** a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation.	**Delete** the entire collection.
Element URI, such as `http://example.com/resources/item17`	**Retrieve** a representation of the addressed member of the collection, expressed in an appropriate Internet media type.	**Replace** the addressed member of the collection, or if it doesn't exist, **create** it.	Not generally used. Treat the addressed member as a collection in its own right and **create** a new entry in it.	**Delete** the addressed member of the collection.

GET

- The HTTP GET method is used to retrieve a representation of a resource
  - GET http://www.example.com/customers/12345
  - GET http://www.example.com/customers/12345/orders
  - GET http://www.example.com/bucket/sample
- In the OK path, GET returns a representation (typically in XML or JSON) and an HTTP response code of 200 (OK)
  - In an error case, usually a 404 (NOT FOUND) or 400 (BAD REQUEST)
- According to the HTTP specification GET, OPTIONS and HEAD are safe and should be used only to read data, not to change it
  - Do not expose unsafe operations via these methods, they should never change a resource
- GET, OPTIONS, HEAD, PUT and DELETE are idempotent
  - Making multiple identical requests produces the same representation every time without changing the resource beyond the first method call

HTTP Method	Idempotent	Safe
OPTIONS	yes	yes
GET	yes	yes
HEAD	yes	yes
PUT	yes	no
POST	no	no
DELETE	yes	no
PATCH	no	no

Popular REST Platforms

- Java
  - JAX-RS
    - Jersey
    - MOXy
- Python
  - Django REST Framework
  - Flask
- Scala
  - Play
- Node.js
  - Connect
  - Express
- Ruby
  - Sinatra
  - Rails-API
- PHP
  - F3

```
user@RONAM ~
$ curl http://localhost:8080/HelloWorld/rest/myresource
Got it!
user@RONAM ~
$ curl -v  http://localhost:8080/HelloWorld/rest/myresource
* Adding handle: conn: 0x1d27e80
* Adding handle: send: 0
* Adding handle: recv: 0
* Curl_addHandleToPipeline: length: 1
* - Conn 0 (0x1d27e80) send_pipe: 1, recv_pipe: 0
* About to connect() to localhost port 8080 (#0)
*   Trying ::1...
* Connected to localhost (::1) port 8080 (#0)
> GET /HelloWorld/rest/myresource HTTP/1.1
> User-Agent: curl/7.30.0
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 200 OK
* Server Apache-Coyote/1.1 is not blacklisted
< Server: Apache-Coyote/1.1
< Content-Type: text/plain
< Content-Length: 7
< Date: Thu, 17 Jul 2014 11:23:19 GMT
<
Got it!
* Connection #0 to host localhost left intact
```

```java
package com.example;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

/**
 * Root resource (exposed at "myresource" path)
 */
@Path("myresource")
public class MyResource {

    /**
     * Method handling HTTP GET requests. The returned object will be sent
     * to the client as "text/plain" media type.
     *
     * @return String that will be returned as a text/plain response.
     */
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String getIt() {
        return "Got it!";
    }
}
```

Clients and Connections

- Client application components use networks to access the application back end
- In order to scale to support huge numbers of clients many considerations must be taken into account at the back end perimeter
  - DNS routing
  - Load Balancing
  - Caching
  - Security
  - Etc.
- We'll need to take a deeper look at the cloud and its network

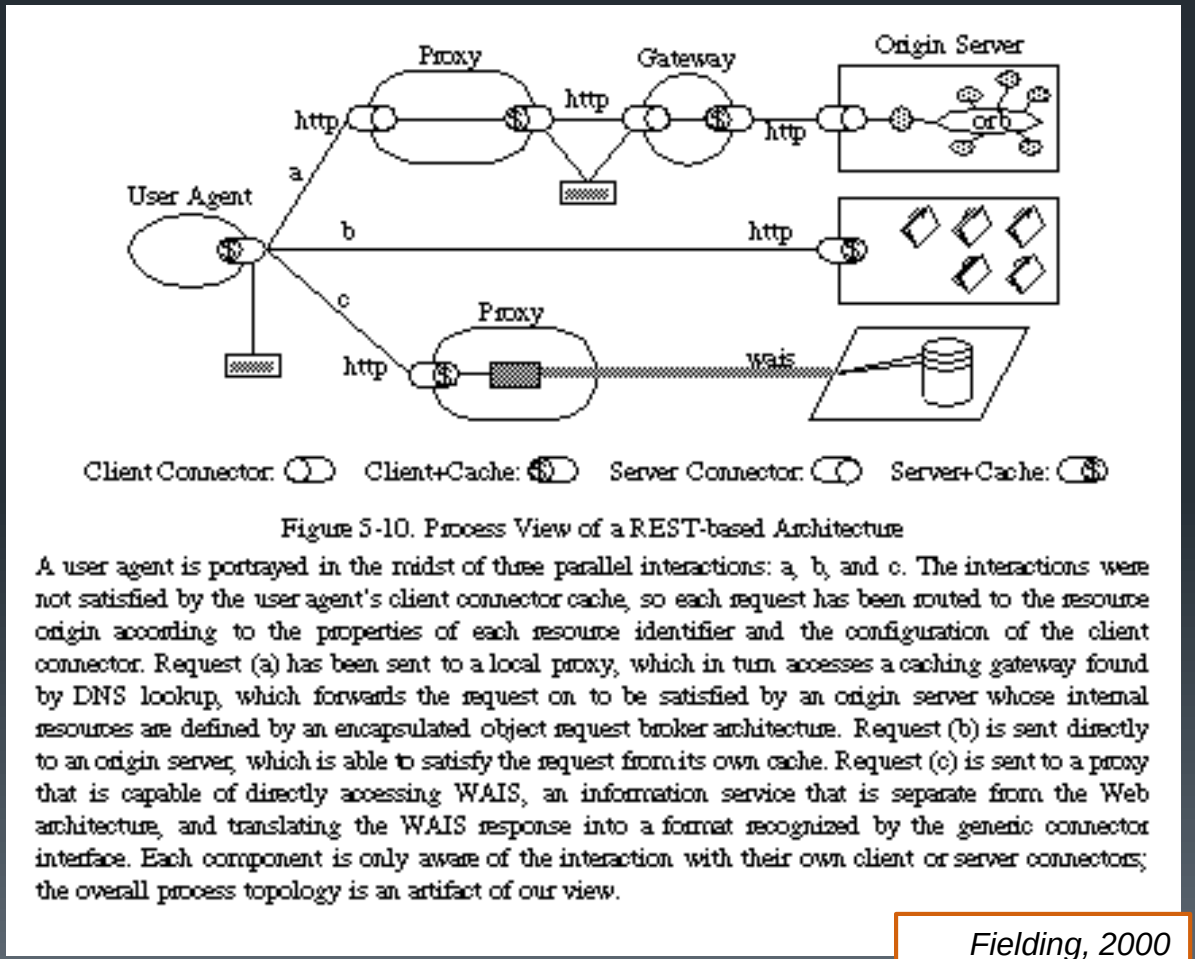


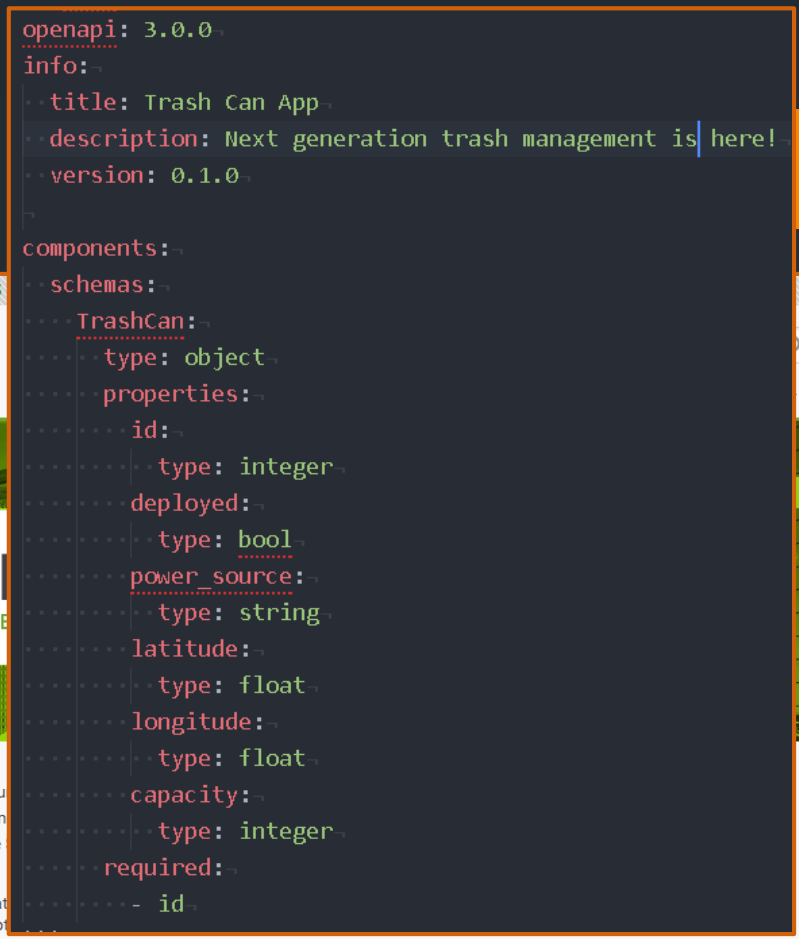
Figure 5-10. Process View of a REST-based Architecture

A user agent is portrayed in the midst of three parallel interactions: a, b, and c. The interactions were not satisfied by the user agent's client connector cache, so each request has been routed to the resource origin according to the properties of each resource identifier and the configuration of the client connector. Request (a) has been sent to a local proxy, which in turn accesses a caching gateway found by DNS lookup, which forwards the request on to be satisfied by an origin server whose internal resources are defined by an encapsulated object request broker architecture. Request (b) is sent directly to an origin server, which is able to satisfy the request from its own cache. Request (c) is sent to a proxy that is capable of directly accessing WAIS, an information service that is separate from the Web architecture, and translating the WAIS response into a format recognized by the generic connector interface. Each component is only aware of the interaction with their own client or server connectors; the overall process topology is an artifact of our view.

Fielding, 2000

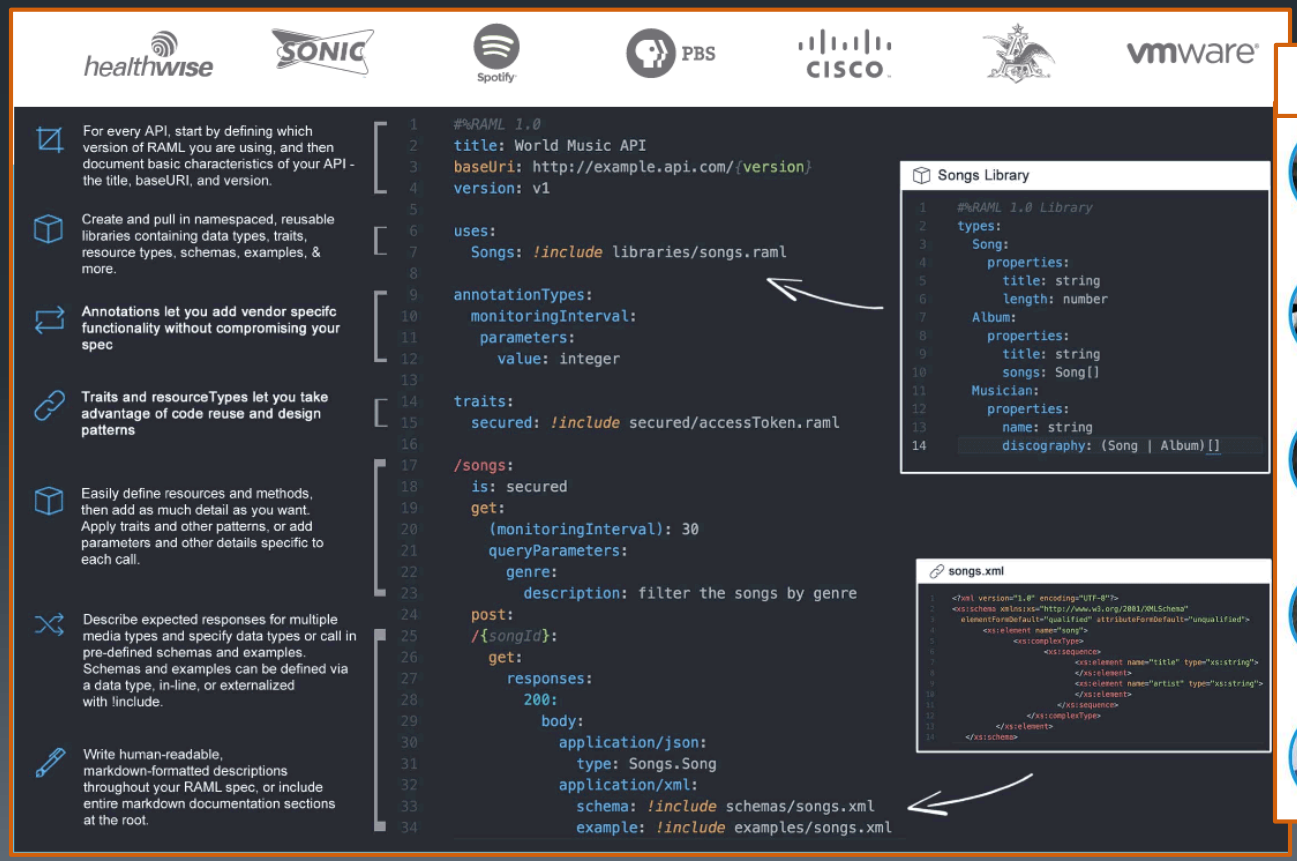
OAI

- Open API Initiative (OAI)
  - Created by an industry consortium interested in standardizing the way REST APIs are described
  - A vendor neutral open governance structure under the Linux Foundation
  - SmartBear Software donated the Swagger Specification
  - Swagger is the basis for the OAI specification
- APIs form the connecting glue between modern applications
- Nearly every application uses APIs to connect with corporate data sources, third party data services or other applications
- Creating an open description format for REST API services that is vendor neutral, portable and open is critical to accelerating the vision of a truly connected world



RAML

- RESTful API Modeling Language (RAML)
- A REST API codification competitor to OAI
- RAML was developed and is supported by a group of technology leaders dedicated to building an open, simple and succinct spec for describing APIs
- The RAML Workgroup provides ongoing contributions to both the RAML spec and a growing ecosystem of tools designed to make API-first design simple and API consumption frictionless
- RAML 1.0 was released on May 16, 2016



RAML Working Group

Uri Sarid
CTO
MuleSoft

Misko Hevery
Project Founder
AngularJS

Ivan Lazarov
Chief Enterprise Architect
Intuit

Peter Rexer
Director of Product - Developer Platform
Airware

John Musser
Founder
Programmable Web & API Science

Tony Gullotta
Director of Technology
Akana Software

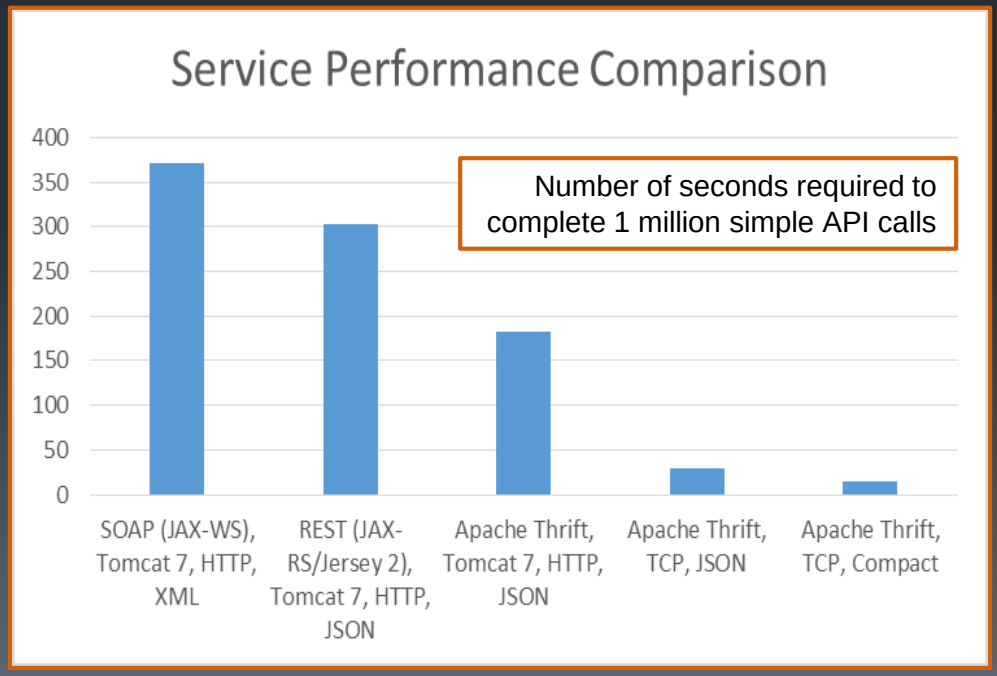
Jaideep Subedar
Sr. Manager, Product Management - Application Integration Solutions Group
Cisco

Kevin Duffey
Senior MTS Engineer
VMware

Rob Daigneau
Director of Architecture for Akamai's OPEN API Platform
Akamai Technologies

Backend API Performance

- REST is predominantly used over the infrastructure of the Web
  - Natural synergies and benefits occur here
- In backend systems web technologies can be cumbersome
  - The is becoming less the case with HTTP/2
- Many Internet giants have invented their own IDL based backend RPC technologies to improve processing performance in the datacenter/cloud
  - Facebook -> Apache Thrift
  - Twitter -> Apache Thrift, Scrooge/Finagle [Thrift for Scala]
  - Google -> Protocol Buffers
- There are benefits to a homogeneous solution but non functional requirements must also be considered when designing end to end corporate solutions



Evolution of RPC

1980 - Bruce Jay Nelson is credited with inventing the term RPC in early ARPANET documents
- The idea of treating network operations as procedure calls

1981 - Xerox Courier possibly the first commercial RPC system

1984 - Sun RPC (now Open Network Computing [ONC+] RPC, RFC 5531)

1991 - CORBA – Common Object Request Broker Architecture
- The CORBA specification defines an ORB through which an application interacts with objects
- Applications typically initialize the ORB and accesses an internal Object Adapter, which maintains things like reference counting, object (and reference) instantiation policies, and object lifetime policies
- General Inter-ORB Protocol (GIOP) is the abstract protocol by which object request brokers (ORBs) communicate
- Internet InterORB Protocol (IIOP) is an implementation of the GIOP for use over the Internet, and provides a mapping between GIOP messages and the TCP/IP layer

1993 - DCE RPC – An open (designed by committee) RPC solution integrated with the Distributed Computing Environment
- Packaged with a distributed file system, network information system and other platform elements

1994 - MS RPC (a flavor of DCE RPC and the basis for DCOM)

1994 - Java RMI – a Java API that performs the object-oriented equivalent of remote procedure calls (RPC), with support for direct transfer of serialized Java objects and distributed garbage collection
- RMI-IIOP (now deprecated) implements the RMI interface over CORBA
- Third party RMI implementations and wrappers are prevalent (e.g. Spring RMI)

1998 - SOAP (Simple Object Access Protocol) specifies a way to perform RPC using XML over HTTP or Simple Mail Transfer Protocol (SMTP) for message negotiation and transmission

2001 - Google Protocol Buffers – developed at Google to glue their servers together and interoperate between their three official languages (C++/Java/Python, JavaScript and others have since been added), used as a serialization scheme for custom RPC systems

2006 - Apache Thrift – developed at Facebook to solve REST performance problems and to glue their servers together across many languages
- The basis for Twitter Finagle, a cornerstone of the Twitter platform

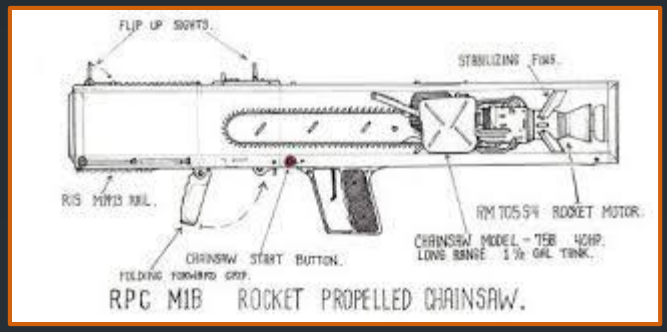
2008 - Apache Avro is a serialization framework designed to package the serialization schema with the data serialized, packaged with Hadoop

2015 - Google gRPC announced as an RPC framework operating over http/2 using protocol buffers for serialization

2017 - Google contributes gRPC to CNCF

Elements of RPC protocols

- Session Management
  - Connection or Connectionless operation
  - If connections are used how are they:
    - Established
    - Maintained
    - Terminated
- Security
  - Is authentication provided
  - Is integrity provided
  - Is confidentiality provided
- Calling Conventions
  - Message structure
  - Interactions of requests and responses
    - Normal Call/Return
    - Returnless calls (one way messages from client to server)
    - Notifications (one way messages from server to client)
  - Synchronous/Asynchronous call support
- Data Serialization
  - How are parameters and return values encoded for transmission



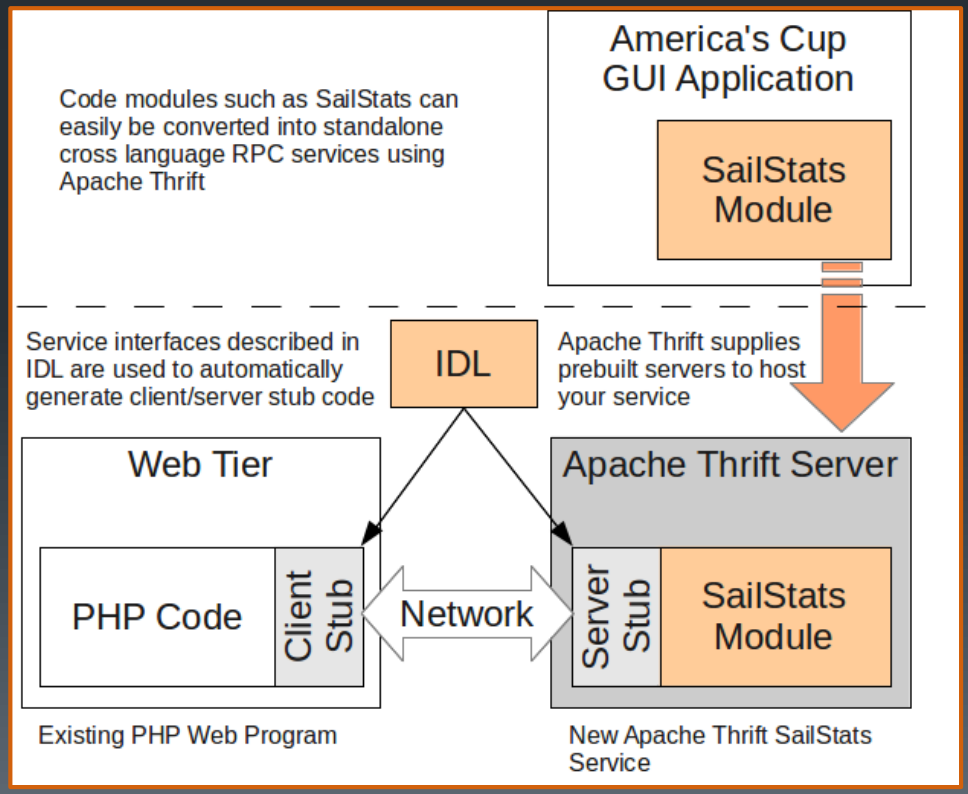
Not that RPC…

Apache Thrift (a case study)

- Apache Thrift is a light weight, fast, cross language RPC framework
  - A plug in transport layer
  - A plug in serialization layer
  - A complete RPC server library
- Commercial systems such as EverNote and open source projects such as Cassandra have adopted Apache Thrift as their principle API provider

C	C++	C#	D
Delphi	Erlang	Go	Haskell
Java	JavaScript	Objective-C	OCaml
Perl	PHP	Python	Ruby
Smalltalk			

Table 1.1 - Languages supported by Apache Thrift



Apache Thrift IDL

- Interface definition languages (IDLs) provide an implementation free view of the critical exchanges taking place between applications and application subsystems
- IDLs tend to be language and platform agnostic
- Apache Thrift IDL supports
  - Constants
  - Data Structures
  - Collections
  - Services, which are collections of functions
  - exceptions
  - And most important, interface evolution

```
service SailStats {
   double GetSailorRating(1: string SailorName),
   double GetTeamRating(1: string TeamName),
   double GetBoatRating(1: i64 BoatSerialNumber),
   list<string> GetSailorsOnTeam(1: string TeamName),
   list<string> GetSailorsRatedBetween(1: double MinRating,
                                       2: double MaxRating),
   string GetTeamCaptain(1: string TeamName),
}
```

```
namespace * FishTrade

enum Market {
    Unknown       = 0
    Portland      = 1
    Seattle       = 2
    SanFrancisco  = 3
    Vancouver     = 4
    Anchorage     = 5
}

typedef double USD

struct TimeStamp {
    1: i16  year
    2: i16  month
    3: i16  day
    4: i16  hour
    5: i16  minute
    6: i16  second
    7: optional i32 micros
}

union FishSizeUnit {
    1: i32  pounds
    2: i32  kilograms
    3: i16  standard_crates
    4: double  metric_tons
}

struct Trade {
    1: string       fish
    2: USD          price
    3: FishSizeUnit amount
    4: TimeStamp    date_time
    5: Market       market=Market.Unknown//Market where trade occured
}

exception BadFish {
    1: string       fish       //The problem fish
    2: i16          error_code //The service specific error code
}
```

```
exception BadFishes {
    1: map<string, i16>  fish_errors //The problem fish:error pairs
}

service TradeHistory {
    /**
     * Return most recent trade report for fish type
     *
     * @param fish the symbol for the fish traded
     * @return the most recent trade report for the fish
     * @throws BadFish if fish has no trade history or is invalid
     */
    Trade GetLastSale(1: string fish)
        throws (1: BadFish bf)

    /**
     * Return most recent trade report for multiple fish types
     *
     * @param fish the symbols for the fish to return trades for
     * @param fail_fast if set true the first invalid fish symbol is thrown
     *                  as a BadFish exception, if set false all of the bad
     *                  fish symbols are thrown using the BadFishes
     *                  exception. If no bas fish are passed this parameter
     *                  is ignored.
     * @return list of trades cooresponding to the fish supplied, the list
     *         returned need not be in the same order as the input list
     * @throws BadFish first fish discovered to be invalid or without a
     *                 trade history (only occurs if skip_bad_fish=false)
     */
    list<Trade> GetLastSaleList(1: set<string> fish
                                2: bool fail_fast=false)
        throws (1: BadFish bf  2: BadFishes bfs)
}
```

Apache Thrift IDL Example

Simple Thrift Service Handler in Java

IDL

```
service Message {
   string motd()
}
```

Listing 9.4 ~/thriftbook/servers/MessageHandler.java

```
import java.util.Arrays;
import java.util.List;
import org.apache.thrift.TException;

public class MessageHandler implements Message.Iface {    #A
  public MessageHandler() {
    msg_index = 0;
  }

  @Override
  public String motd() throws TException {
    System.out.println("Call count: " + ++msg_index);
    return msgs.get(Math.abs(msg_index%3));                #B
  }

  private int msg_index;
  private static List<String> msgs = Arrays.asList("Apache Thrift!!",
                                                   "Childhood is a short season",
                                                   "'Twas brillig");
}
```

This should be atomic with a multithreaded server (the book uses this example to demonstrate a race)

Server

```
$ ls -l
drwxr-xr-x 2 randy randy 4096 Jul 15 07:33 gen-cpp
drwxr-xr-x 3 randy randy 4096 Jul 15 07:42 gen-py
-rw-r--r-- 1 randy randy  534 Jul 16 00:53 MessageHandler.java
-rw-r--r-- 1 randy randy  470 Jul 15 22:37 simple_client.py
-rw-r--r-- 1 randy randy 1148 Jul 15 06:46 simple_server.cpp
-rw-r--r-- 1 randy randy   35 Jul 15 04:47 simple.thrift
-rw-r--r-- 1 randy randy  590 Jul 16 00:53 ThreadedServer.java
$ thrift -gen java simple.thrift
$ javac -cp /usr/local/lib/libthrift-1.0.0.jar:\
           /usr/local/lib/slf4j-api-1.7.2.jar:\
           /usr/local/lib/slf4j-nop-1.7.2.jar
           ThreadedServer.java MessageHandler.java gen-java/*.java
Note: gen-java/Message.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
$ java -cp /usr/local/lib/libthrift-1.0.0.jar:\
          /usr/local/lib/slf4j-api-1.7.2.jar:\
          /usr/local/lib/slf4j-nop-1.7.2.jar:\
          gen-java:\
          .
          ThreadedServer
```

Listing 9.5 ~/thriftbook/servers/ThreadedS

```
import org.apache.thrift.TProcessor;
import org.apache.thrift.server.TServer;
import org.apache.thrift.server.TThreadPoolServer;
import org.apache.thrift.transport.TServerSocket;
import org.apache.thrift.transport.TTransportException;

public class ThreadedServer {
  public static void main(String[] args) throws TTransportException {
    TServerSocket svrTrans = new TServerSocket(8585);                    #C
    TProcessor processor = new Message.Processor<>(new MessageHandler());#D
    TServer server = new TThreadPoolServer(
             new TThreadPoolServer.Args(svrTrans).processor(processor)); #E
    server.serve();                                                      #F
  }
}
```

Client

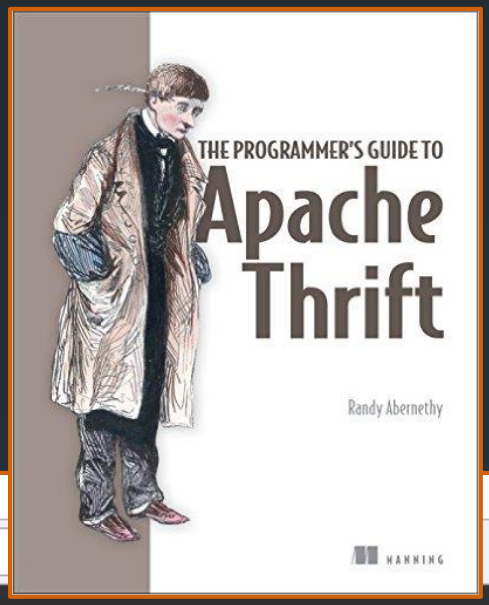
```
$ thrift -gen py simple.thrift
$ python simple_client.py
[Client] received: Childhood is a short season
Enter 'q' to exit, anything else to continue: q
$ python simple_client.py
[Client] received: 'Twas brillig
Enter 'q' to exit, anything else to continue:
[Client] received: Apache Thrift!!
Enter 'q' to exit, anything else to continue: q
$
```

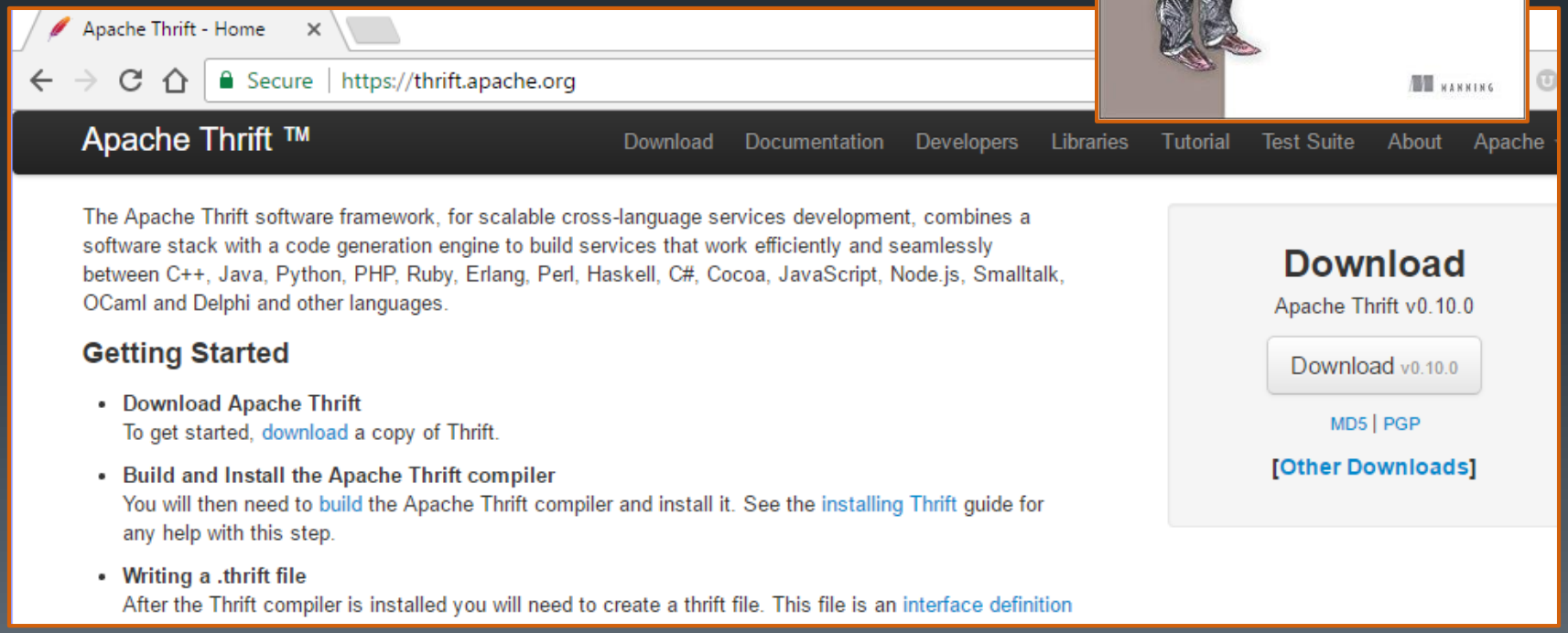
Listing 9.3 ~thriftbook/servers/simp

```
import sys
sys.path.append("gen-py")
from thrift.transport import TSocket
from thrift.protocol import TBinaryProtocol
from simple import Message

trans = TSocket.TSocket("localhost", 8585)
trans.open()
proto = TBinaryProtocol.TBinaryProtocol(trans)
client = Message.Client(proto)

while True:
    print("[Client] received: %s" % client.motd())
    line = raw_input("Enter 'q' to exit, anything else to continue: ")
    if line == 'q':
        break

trans.close()
```

Apache Thrift Info

- Docs on the Web (not super but will get you started)
- Books:
  - The Programmer's Guide to Apache Thrift
    - Randy Abernethy
    - Manning Publications





Summary

- Microservice communication schemes include
  - Request/Response
  - Messaging
  - Streaming
- Request/Response style service communications types include
  - REST
  - RPC
- RESTful interfaces can be described using OAI
- RPC interfaces can be described using IDL

Lab 2

- Creating a RESTful microservice

3: Container Packaging

Objectives

- Define application container
- List several container technologies
- Explain the purpose of Docker
- Contrast Virtual Machines and containers
- Explore the relationship between
  - IaaS and Virtual Machines
  - PaaS and Containers
- Understand the nature of Docker Images

What is a container?

- Lightweight Linux environment
  - Now lightweight Windows environments as well
- Encapsulated and deployable
- Runnable
- A way to package applications for reliable deployment
  - Particularly popular for packaging microservices
- Made widely popular by Docker
  - Docker simplifies configuring, constructing and running containers
  - Docker Inc. provides a container platform including systems for publishing and sharing containers
- Container technology predates and enables Docker
  - Docker was the dominant mover in this space during 2013-2015
  - Competition is maturing
  - Standards exist and are evolving

```
$ time docker run ubuntu echo "hello world"
hello world

real    0m0.319s
user    0m0.005s
sys     0m0.013s
```

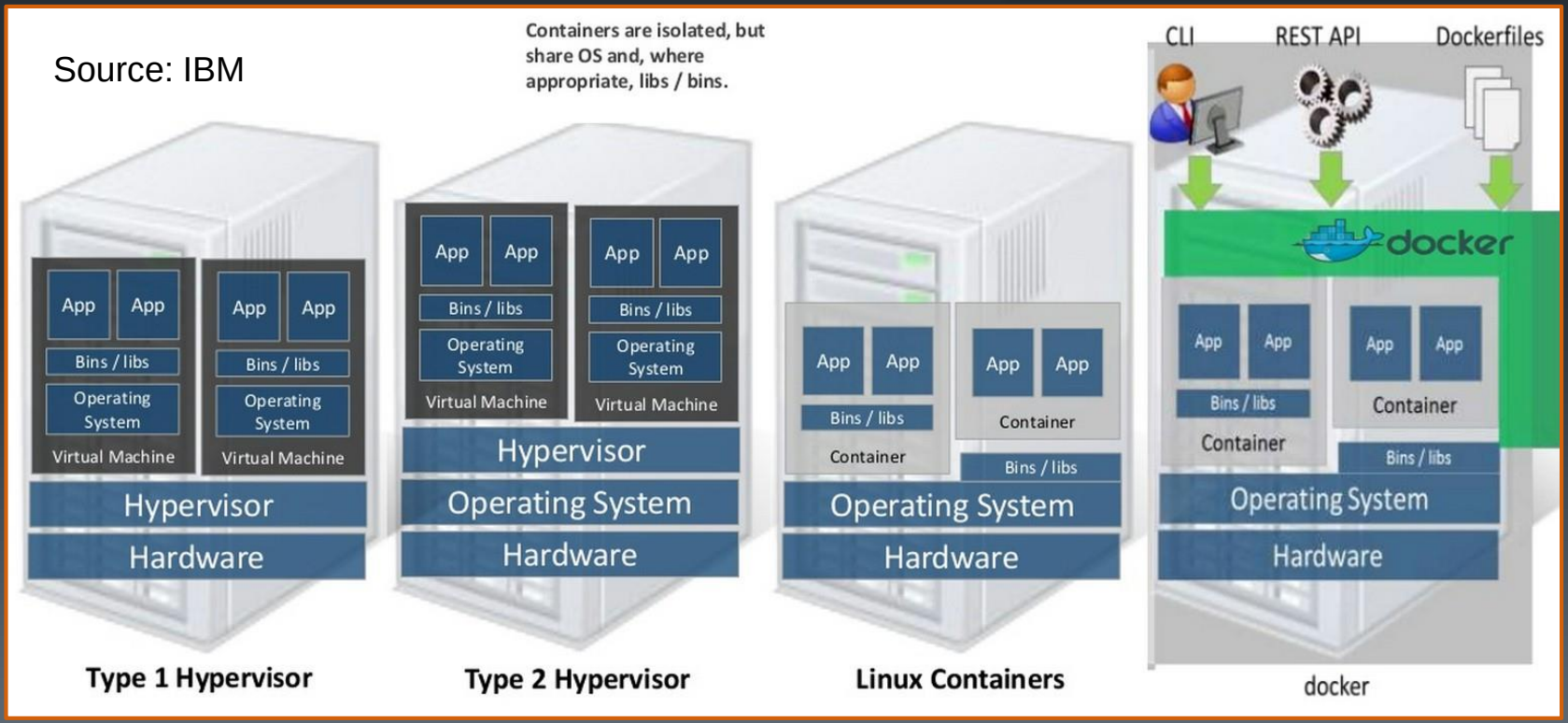
Disk usage: less than 100 kB
Memory usage: less than 1.5 MB

VMs and Containers

- VM
  - A virtual machine for operating systems
- Container
  - A virtual operating system for applications
- These are not mutually exclusive and many environments become optimal when properly combining both

Containers supply only the executables and library interfaces necessary to mimic the application dependent aspects of a Linux distribution (Ubuntu, RHEL, SUSE, etc.), leveraging the fact that all Linux systems use the same underlying kernel

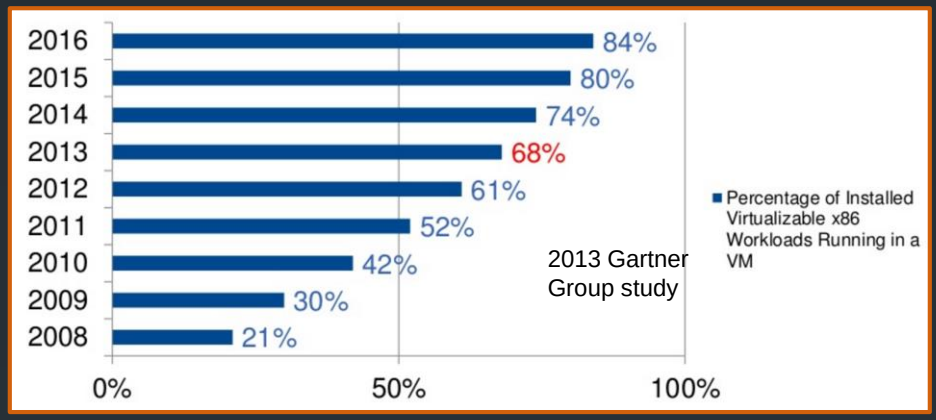
Source: IBM

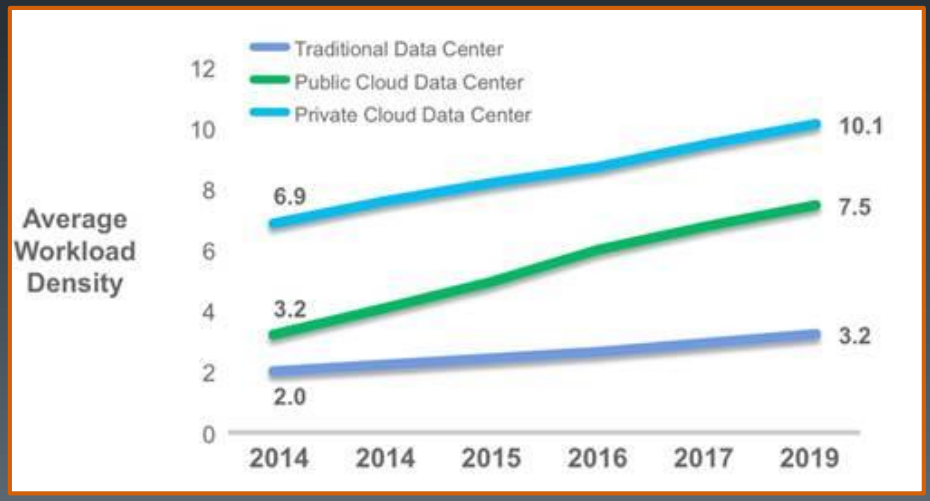


VMs, why do I care?

- VM's allow new machine instances to be deployed in real time (seconds-minutes)
  - Not months, as is typical with physical server purchase/deploy cycles
- VM's enable migration and elasticity
  - Machines can be created, moved and deleted rapidly
- VM's allow physical resources to be fully utilized
  - Physical CPU/RAM use can be maximized without mixing logical machine roles
  - Increased server density
- VM's enable repeatable static system environments to be used for development, testing and deployment
  - The same machine can be launched by Vagrant, OpenStack, Amazon EC2, Microsoft Azure, Google Cloud, etc.
- VM's enable cloud computing
  - Pay as you go
  - Self service

IaaS

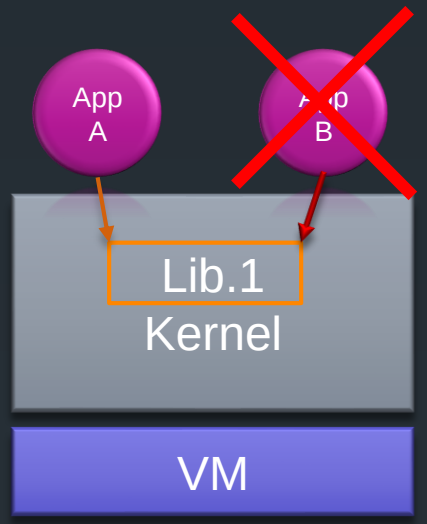


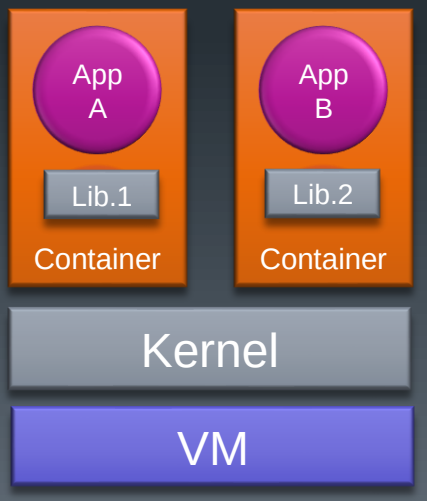


Containers, why do I care?

- Containers provide a static application runtime environment creating reliable deployments
  - Fundamentally changes the approach to operations
  - Removes an entire class of extremely complex operational problems

- VMs are typically used to host infrastructure roles (e.g. Web Server, Application Server, Database Server, …)
  - Applications running on such systems can have complex interactions with system services and other applications
  - This complexity makes it difficult to guarantee identical dev/test/prod environments, making application deployment complex and error prone
- Containers create a new layer of abstraction at the operating system level for application roles (web server, logging system, security tools, monitoring software, etc.)
  - Isolation
    - Allows multiple containerized applications to run on the same VM
  - Encapsulation
    - Each container is encapsulated with its own unique dependencies
  - Portability
    - Repeatable deployment across dev/test/prod environments and clouds





PaaS

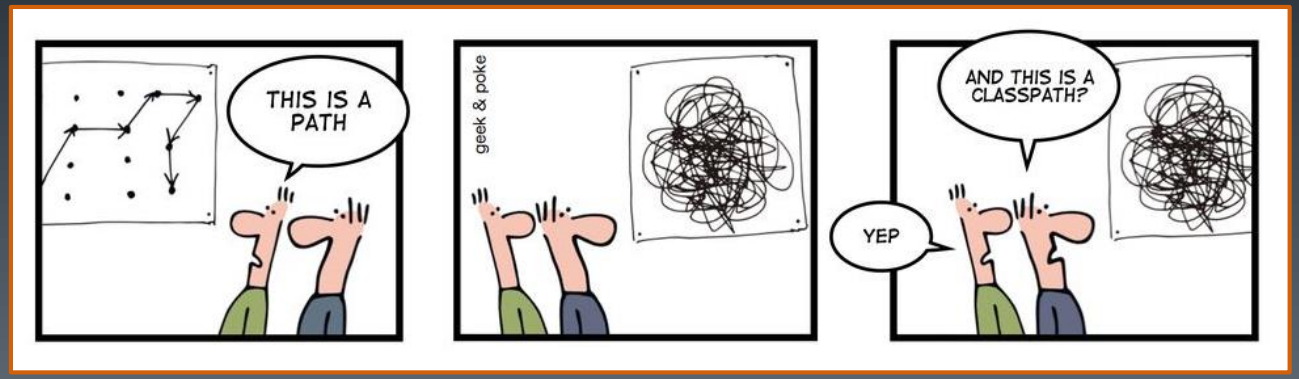
Encapsulation

- Containers can encapsulate:
  - Data
  - Code
  - Configuration files
  - Frameworks
  - Libraries
  - System Dependencies
  - Packaging
  - Linux Distributions
  - Environment Variables
  - Command line arguments
  - and more

- Virtually everything needed by an application can be packaged within the application's container
- We can ignore where and how the container runs
  - The container's internals do not interact with external aspects of the environment, removing an entire class of deployment problems

Empowering Agile Processes

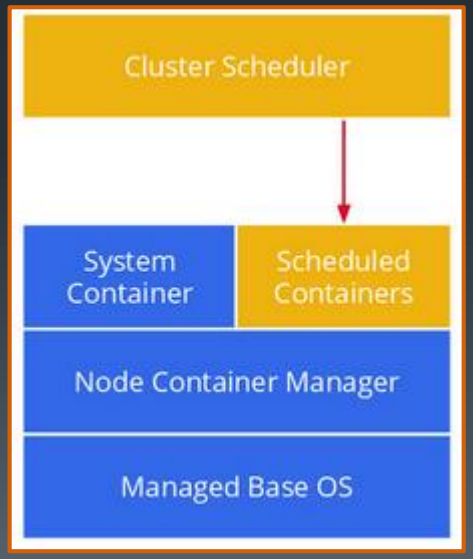
- Microservices
  - The more atomic the service:
    1. The more likely it is to be reusable
    2. The more easily it can be encapsulated
    3. The more of them you need to do something useful (!)
- Micro services and VMs have different cardinalities
  - 10:1 Ten services running on a single VM creates an ops integration challenge across all services
- Micro services and Containers have the same cardinality
  - 1:1 One service in one container, requires only ops support for the service encapsulated
- Containers allow each service to be packaged with its own dependencies
  - 10 containerized services map directly to 10 individual, isolated, reliable, repeatable ops events
  - Gives devops teams autonomy
    - Team owns software and configuration when using containers
  - Empowers CI/CD and incremental application evolution
    - Integration challenges are reduced to inter service communications, networking and volumes



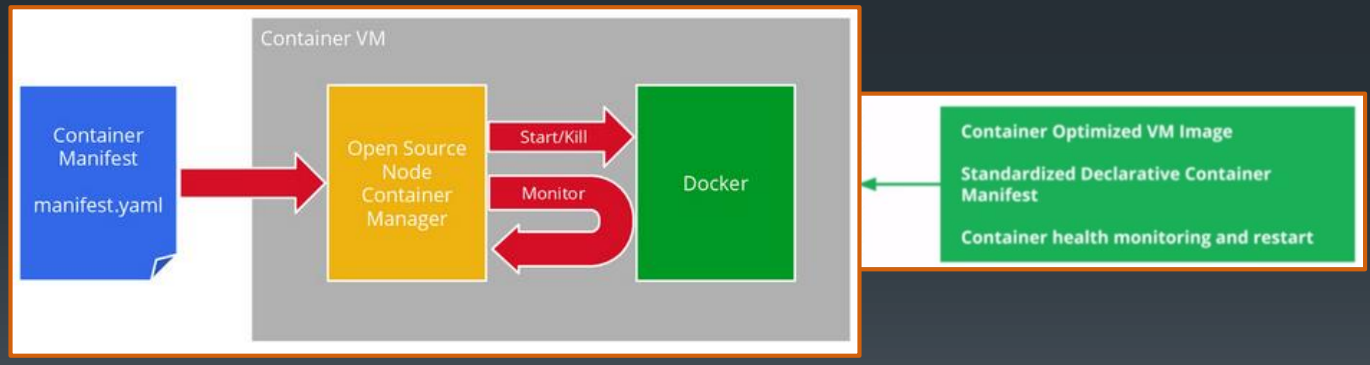
Containers at scale

- Containers - a key enabler of PaaS cloud environments
  - Google App Engine is one of the most visible container based cloud systems
- Everything at Google, from Search to Gmail, is packaged and run in a Linux container
  - Google's Borg system (the basis for Kubernetes) is a cluster manager that ran hundreds of thousands of jobs, from many thousands of different containerized applications, across a number of clusters each with up to tens of thousands of machines (Borg is now replaced within Google by the next gen Omega system)
  - Over 2 billion containers are started per week at Google (over 3,000 per second on average)
- lmctfy ("Let Me Contain That For You") open source version of Google's container stack
  - Google has ported the core lmctfy concepts and abstractions to Docker's libcontainer
  - libcontainer is now the basis for much of the OCI container standard

Google Container Infrastructure



Google Cloud Platform
Docker containers in Google Compute Engine

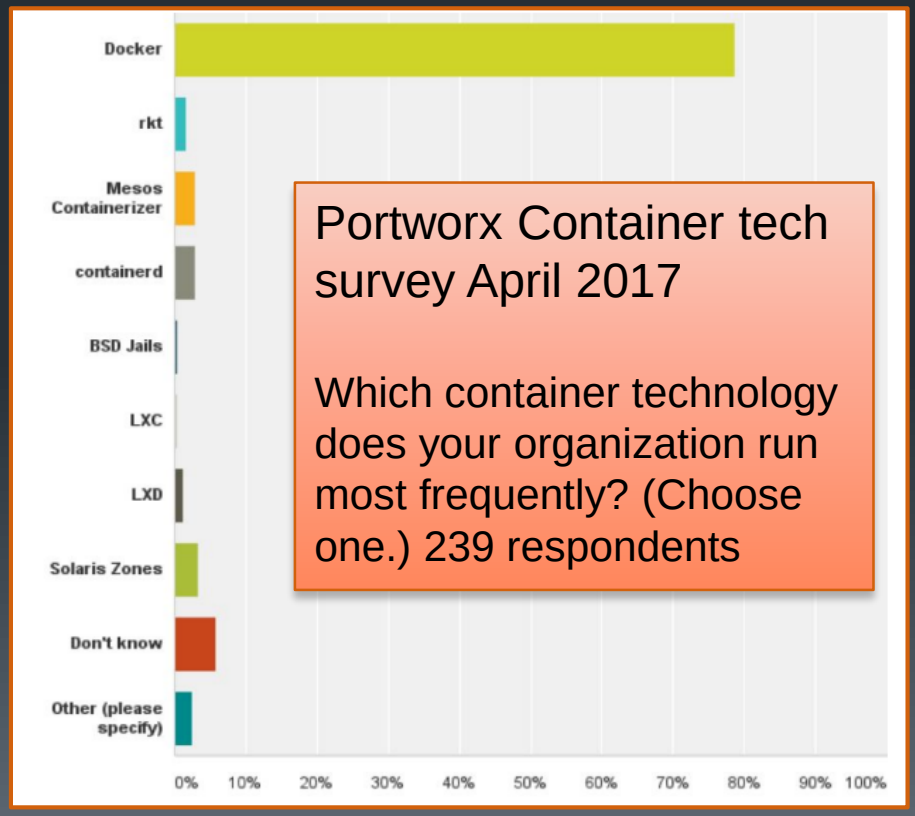


Containers At Scale by Joe Beda
Published May 22, 2014 in Technology

Borg Paper: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43438.pdf

Docker Alternatives

- Virtual Machines
  - Better isolation, worse performance, more complex deployment
  - VMware, AWS, Google, and Azure all run Docker-based workloads on behalf of cloud customers in a multi-tenant environment, but do so by putting each customer's Docker containers inside the logical boundaries of virtual machines
  - VMware VIC (VM Integrated Containers)
  - Kata Containers (combined projects: Hyper RunV and Intel Clear Containers)
- RedHat Solutions
  - CoreOS Inc. (acquired by RedHat) Tectonic is a minimal Linux OS with direct support for containers (similar to RHEL Atomic, Ubuntu Core and Windows 2016 Nano)
  - Rocket (rkt) lightweight docker like platform
  - CRI-O is an open source RedHat backed container manager
- Joyent SmartOS & Triton
  - Triton uses the docker client but supplies SmartOS (an OpenSolaris/KVM hypervisor) with Solaris Zone isolation (using Illumos Zones) for containers offering strong security
- Mesos
  - Distributed OS running ”frameworks”
- LXC
  - Original Linux container library (v2.0.3 – 6/2016)
- LXD/OpenVZ
  - Systems in containers (lightvisors)
- BSD Jails
  - Isolation features of BSD Unix
- OpenSolaris Container
  - Zones based isolation model



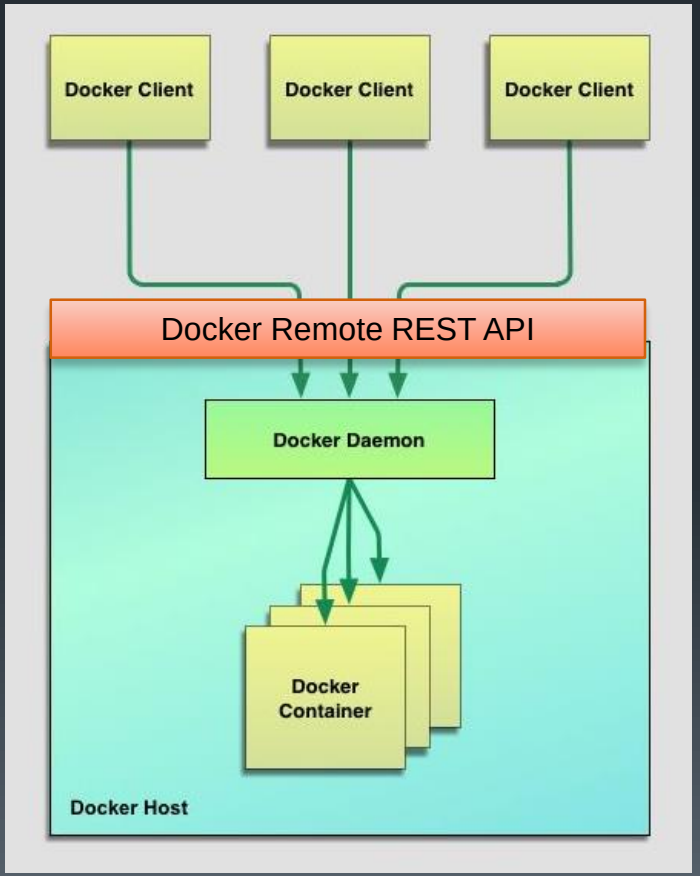
http://portworx.com/wp-content/uploads/2017/04/Portworx_Annual_Container_Adoption_Survey_2017_Report.pdf

Container Standardization

- OCI [circa 6/2015]
  - Open Container Initiative
    - Formerly Open Container Project
  - The OCI contains three specifications:
    - Runtime Specification (runtime-spec) 1.0
      - How to run a “filesystem bundle” that is unpacked on disk
    - Image Specification (image-spec) 1.0
      - The format of an OCI Image and how to unpack that image into an OCI Runtime filesystem bundle
    - Distribution Specification (distribution-spec) in progress
      - Standardizes container image distribution based on Docker Registry API v2 protocol
  - Run by the Linux Foundation
    - Non-profit which oversees the Linux open source operating system
  - Docker donated container format, runtime code (libcontainer), specifications and registry API
  - Members (All of the tech companies in the Fortune 100 except Apple, all of the key container startups):
    - Amazon, Apcera, Apprenda, Aqua, AT&T, ClusterHQ, Cisco, CoreOS, ContainerShip, Datera, Dell, Docker, EMC, Facebook, Fujitsu Limited, Goldman Sachs, Google, Hewlett Packard Enterprise, Huawei, IBM, Infoblox, Intel, Joyent, Kismatic, Kyup, Mesosphere, Microsoft, Midokura, Nutanix, Odin (Virtuozzo), Oracle, Pivotal, Polyverse, Portworx, Rancher Labs, Red Hat, Replicated, Resin.io, Robin, Scalock, Sysdig, SUSE, Twistlock, Twitter, Univa, Verizon Labs, Vmware, Weaveworks and WD

Core Docker Components

- Docker Engine
  - Daemon on the Linux host supporting docker container execution: /usr/bin/dockerd
    - Docker manages containers, the linux kernel runs containers
- Docker Client
  - The Docker client sends requests to local and remote Docker daemons: /usr/bin/docker (or docker.exe)
- Docker Remote API
  - The Docker client talks to the Docker daemon through the Docker Remote RESTful API
- Docker Images
  - Images are used to generate containers
    - Just as a VM image can create multiple VM instances
    - Just as an executable (/usr/bin/vi) can create multiple processes
- Registries
  - Registries are network services from which Docker Images can be saved and retrieved
  - The Docker Hub is an internet based registry with support for public and private images
  - Private registry servers can be created for an organization's internal use
- Docker Containers
  - A container is a software package generated from an image
  - Said to be running when processes are executing within it
  - Can be stopped and started

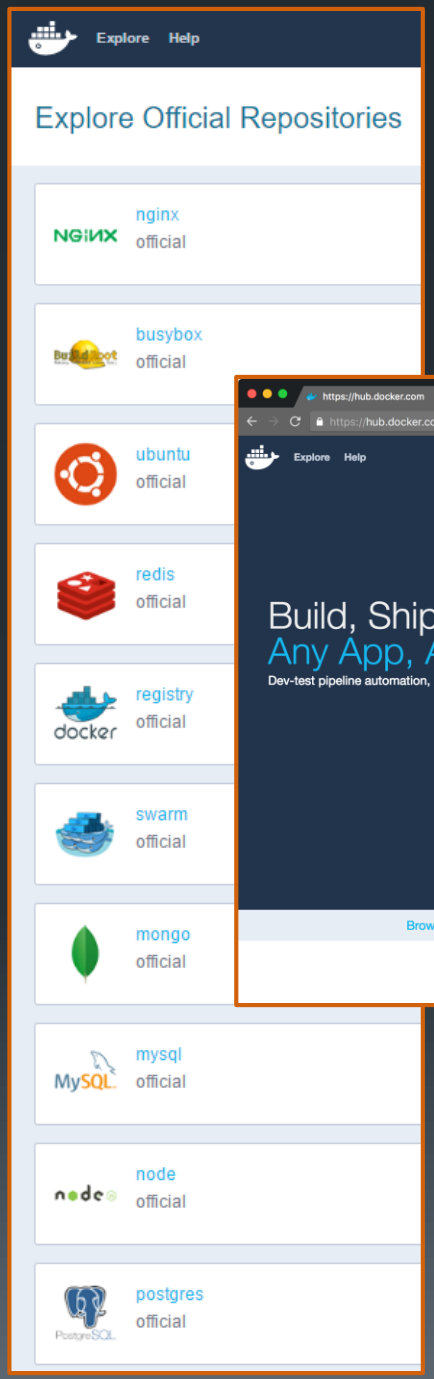


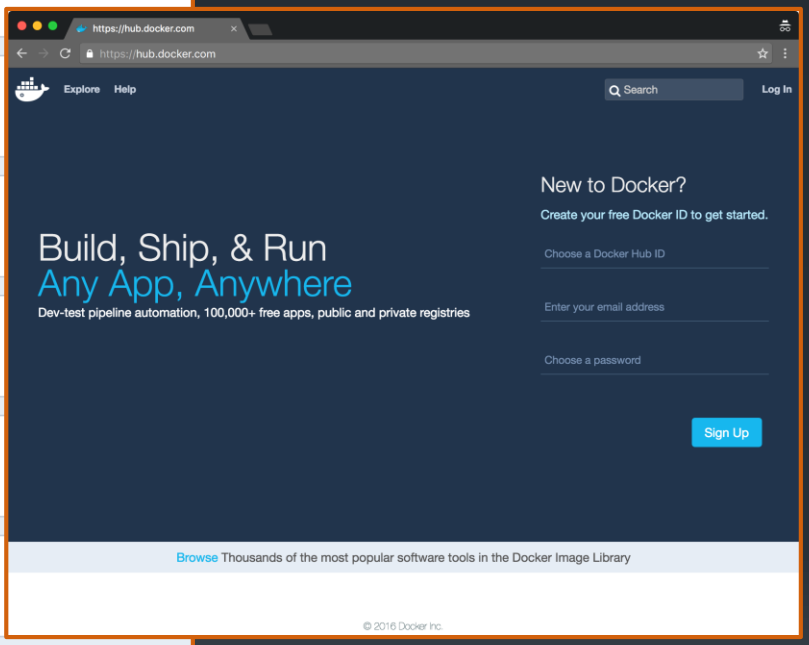
Container Control

```
user@ubuntu:~$ docker container --help

Usage:  docker container COMMAND

Manage containers

Commands:
  attach      Attach local standard input, output, and error streams to a running container
  commit      Create a new image from a container's changes
  cp          Copy files/folders between a container and the local filesystem
  create      Create a new container
  diff        Inspect changes to files or directories on a container's filesystem
  exec        Run a command in a running container
  export      Export a container's filesystem as a tar archive
  inspect     Display detailed information on one or more containers
  kill        Kill one or more running containers
  logs        Fetch the logs of a container
  ls          List containers
  pause       Pause all processes within one or more containers
  port        List port mappings or a specific mapping for the container
  prune       Remove all stopped containers
  rename      Rename a container
  restart     Restart one or more containers
  rm          Remove one or more containers
  run         Run a command in a new container
  start       Start one or more stopped containers
  stats       Display a live stream of container(s) resource usage statistics
  stop        Stop one or more running containers
  top         Display the running processes of a container
  unpause     Unpause all processes within one or more containers
  update      Update configuration of one or more containers
  wait        Block until one or more containers stop, then print their exit codes

Run 'docker container COMMAND --help' for more information on a command.
user@ubuntu:~$
```

The Docker Hub

- The Docker Hub is a registry supporting dynamic container image download
- Docker Hub hosts many base images
  - Cirros
  - Ubuntu
  - Fedora
  - Debian
  - Centos
  - etc .
- These images can be used as the basis for building custom application service images
  - Our prior example automatically downloaded the minimal Cirros Linux cloud image and then ran a Bourne shell within the container
- When docker can not find an image locally it looks for the image in a registry, like DockerHub
- Docker Hub also hosts ready to run service images
  - MongoDB, Cassandra, Apache, Nginx, Redis, WordPress, …
- And dev platform images
  - Java, NodeJS, Ruby, Python, …

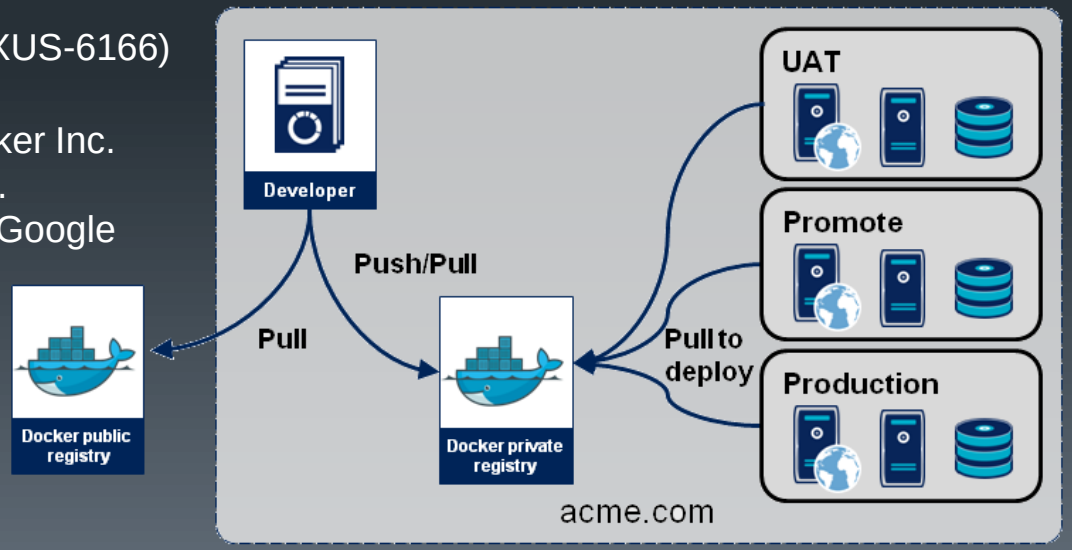




Registries, Repos and Tags

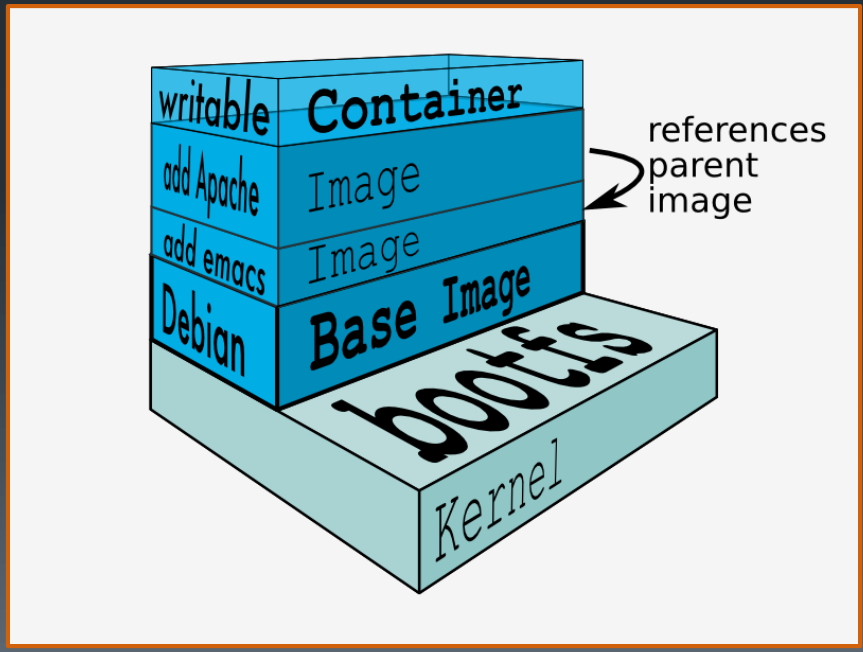
- Images house file system layers and metadata
- Repositories are named collections of images
- Tags are strings used to identify individual images within a repository
- Registries are services that allow you to store and retrieve images by repository:tag name using a REST (HTTP) API
  - Docker Hub is the central public registry
  - Private open source and commercial registries
    - Docker Trusted Registry (formerly known as: Docker Hub Enterprise) from Docker Inc. (commercial product)
    - Quay Enterprise part of the Tectonic platform from CoreOS (commercial product)
    - Docker Registry from Docker Inc. (free open source) (https://github.com/docker/docker-registry)
    - Harbor open source enterprise-class container registry server from Vmware (https://github.com/vmware/harbor)
    - Artifactory 3.4+ (http://www.jfrog.com/open-source/#os-arti)
    - Nexus 3 Milestone 5 (in preview) (https://issues.sonatype.org/browse/NEXUS-6166)
  - Hosted cloud solutions
    - Docker Hub (hub.docker.com) from Docker Inc.
    - Quay (quay.io), acquired by CoreOS Inc.
    - Google Container Registry (gcr.io) from Google
    - Amazon EC2 Container Registry [ECR]

Public registry images should be used with caution by the security minded



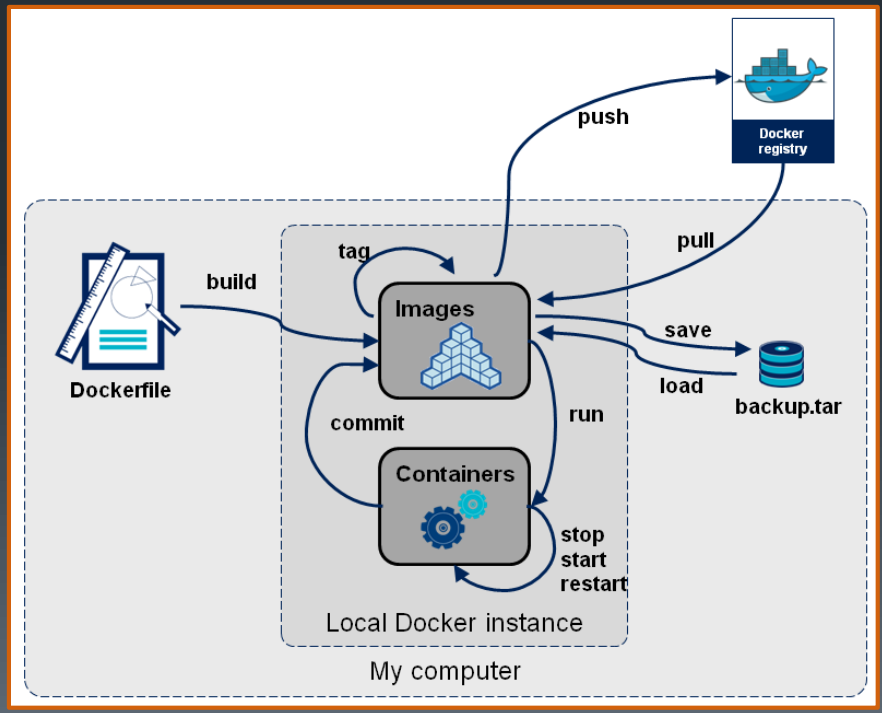
Docker Images

- Docker containers are constructed by sequentially applying meta data and file system layers from one or more images
- A Docker image includes metadata and an optional file system layer
  - Image meta data includes the ID of the image's parent, the default command to run, etc.
    - In Docker Engine <= v1.9 image IDs were random (UUIDs)
    - In Docker Engine >= v1.10 image IDs are SHA256 hashes of the image (content addressable)
  - Layered images tend to supply one specific feature on top of the parent image
  - Upper layer metadata and files mask metadata and files at lower layers with the same name/pathname
- An image with no parent image is called a Base Image
  - Base images tend to be largish and house operating system libraries and executables (e.g. Debian, Ubuntu, Centos, etc.)
- Image metadata and files are immutable
  - Allows one Image to support multiple Container instances with repeatable results
  - Reduces the disk and memory footprint of a given set of containers which share the same read only images
- Containers have a writable file system layer
  - The container file system layer is initially empty
  - All writes go to this file system layer and overlay any matching underlying image files
  - In this way, container file systems contain only the delta between their file system state and that of their underlying images
  - The view from the top down, including all file system layers in the stack, is called the union file system



Creating Images

- Docker images are the basis of containers
- Docker stores downloaded images on the Docker host
  - If a required image isn't already present on the host it is downloaded from a registry
- There are two ways to create new images
  - Update a container created from an image and commit the results to a new image
  - Use a Dockerfile to specify instructions to create an image



Dockerfiles

- docker commit
  - A practical way to create images interactively
  - An impractical way to create identical or incrementally improved images
- docker build
  - The docker build command creates images automatically from a Dockerfile
- Dockerfile
  - A text document containing all the commands you would normally execute manually in order to build a Docker image with docker commit
  - To build an image you can place a file called “Dockerfile” at the root of your repository and call “docker build” with the path to your repository
    - $ docker build .
  - Each command in a Dockerfile creates a new image layer

```
user@ubuntu:~$ mkdir thriftdev
user@ubuntu:~$ cd thriftdev/
user@ubuntu:~/thriftdev$ vim dockerfile
user@ubuntu:~/thriftdev$ cat dockerfile
# Version: 1.0.0
FROM ubuntu:14.04
RUN apt-get update
RUN apt-get install -y git
RUN echo 'thrift dev image v1.0.0' > /README.md
LABEL license "Apache 2.0"
user@ubuntu:~/thriftdev$ docker build -t my-image .
Sending build context to Docker daemon  2.048kB
Step 1/5 : FROM ubuntu:14.04
...
 ---> 3b853789146f
Step 2/5 : RUN apt-get update
 ---> Running in a9f3916d7afa
...
Removing intermediate container a9f3916d7afa
 ---> 13834b4d4d37
Step 3/5 : RUN apt-get install -y git
 ---> Running in d1957635c842
...
Removing intermediate container d1957635c842
 ---> 499d7350e19a
Step 4/5 : RUN echo 'thrift dev image v1.0.0' > /README.md
 ---> Running in 503ca71c7085
Removing intermediate container 503ca71c7085
 ---> 80787dfa4e12
Step 5/5 : LABEL license "Apache 2.0"
 ---> Running in 84267ff31dfe
Removing intermediate container 84267ff31dfe
 ---> 450665b7c688
Successfully built 450665b7c688
Successfully tagged my-image:latest
user@ubuntu:~/thriftdev$
```

```
user@ubuntu:~/thriftdev$ docker image ls -a
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
my-image            latest              450665b7c688        42 minutes ago      282MB
<none>              <none>              80787dfa4e12        42 minutes ago      282MB
<none>              <none>              499d7350e19a        42 minutes ago      282MB
<none>              <none>              13834b4d4d37        43 minutes ago      244MB
ubuntu              14.04               3b853789146f        4 days ago          223MB
user@ubuntu:~/thriftdev$
```

Summary

- Application containers are packaging systems and runtime environments that isolate and constrain the resources of a single service
- Docker is the prevalent container technology though several others are available and new options are emerging
- Docker automates the process of running containerized services
- Virtual Machines and containers have some features in common but are significantly different technologies
  - VMs are more heavy weight but provide better isolation
- Docker Images offer a layered system for creating containers
  - This may enable software sharing without strong coupling

Lab 3

- Container packaging

Day 2 – Microservices and State

- Microservice Communications II - Messaging
- Cloud Native Transactions and Event Sourcing
- Stateless Services and Polyglot Persistence

4: Communication II

Objectives

- Explain the benefits of loosely coupled systems
- Describe the difference between asynchronous and synchronous communications
- Map out the behavior of event based systems
- List some of the more important messaging platforms

Loosely Coupled Systems

- A loosely coupled system is one in which each of its components has, or makes use of, little or no knowledge of the definitions of other separate components
- One of the most critical enablers of loosely coupled systems is messaging

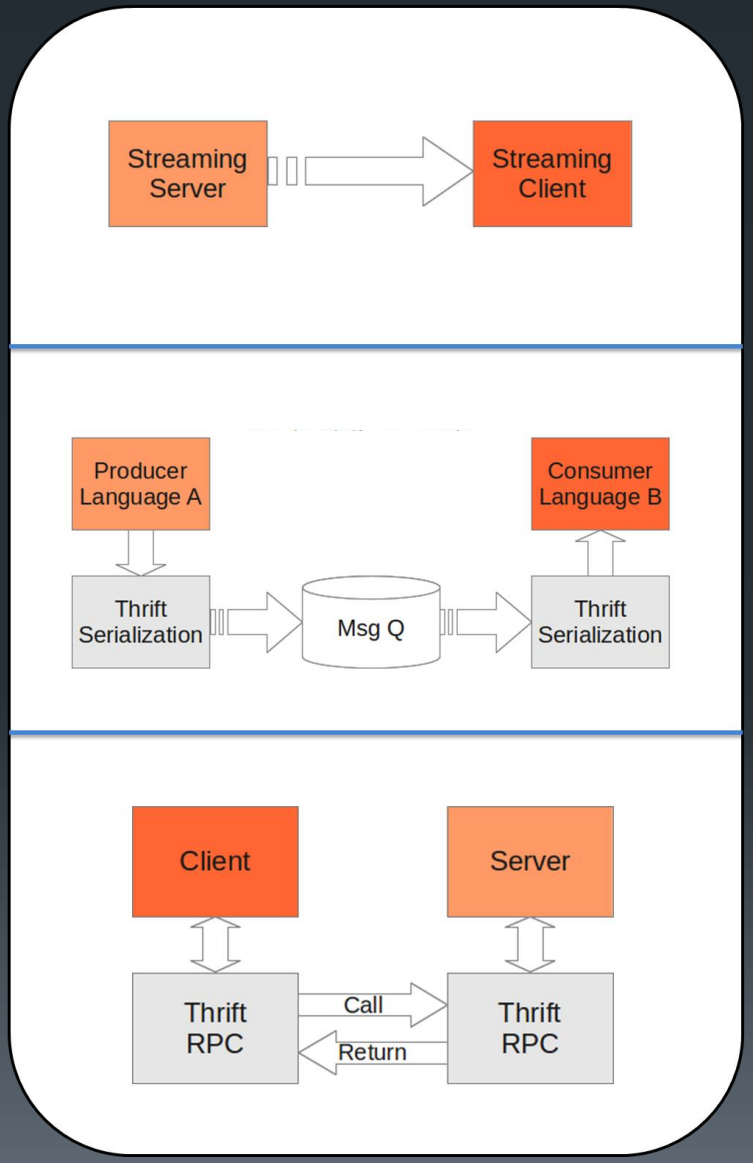
Loose Coupling

- Systems that are loosely coupled have independent lifespans
  - If service A crashes or is taken down, service B should not crash
  - Rather service B should either
    - Not know (as would be the case in many messaging scenarios)
    - Rediscover and reconnect (trying repeatedly over time as necessary)
  - Services directly depending upon other service (e.g. client/server relationships) should degrade favorably until the dependency is resolved
- Loose coupling is critical to overall application safety
  - Isolating failure modes rather than propagating them

Level	Tight Coupling	Loose Coupling
Physical coupling	Direct physical link required	Physical intermediary
Communication style	Synchronous	Asynchronous
Type system	Strong type system (e.g., interface semantics)	Weak type system (e.g., payload semantics)
Interaction pattern	OO-style navigation of complex object trees	Data-centric, self-contained messages
Control of process logic	Central control of process logic	Distributed logic components
Service discovery and binding	Statically bound services	Dynamically bound services
Platform dependencies	Strong OS and programming language dependencies	OS- and programming language independent

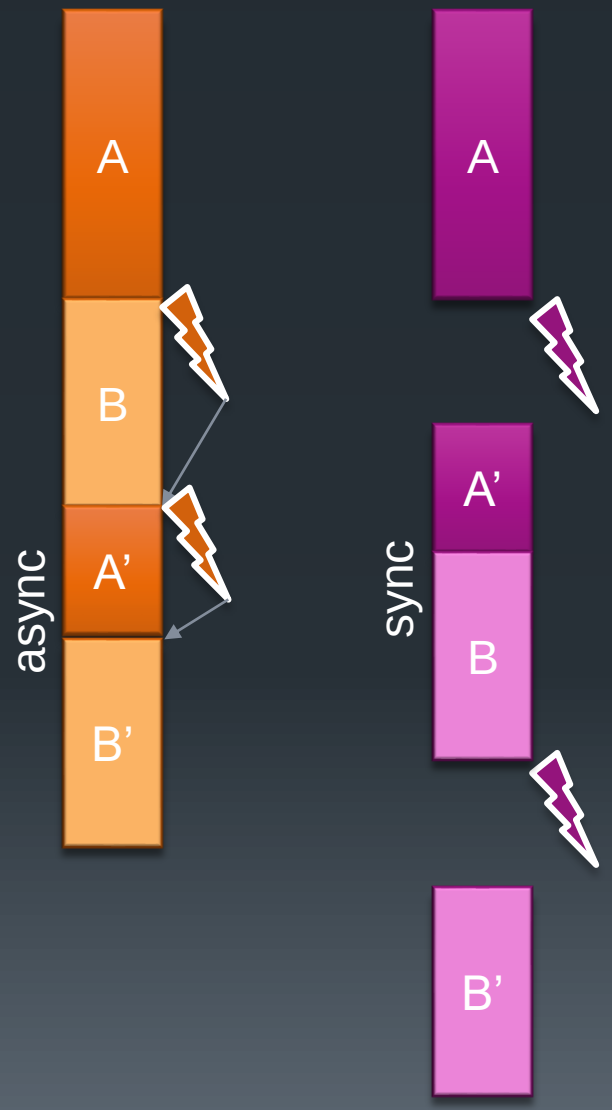
Communications Schemes

- **Streaming** – Communications characterized by an ongoing flow of bytes from a server to one or more clients.
  - Example: An internet radio broadcast where the client receives bytes over time transmitted by the server in an ongoing sequence of small packets.
- **Messaging** – Message passing involves one way asynchronous, often queued, communications, producing loosely coupled systems.
  - Example: Sending an email message where you may get a response or you may not, and if you do get a response you don't know exactly when you will get it.
- **RPC** – Remote Procedure Call systems allow function calls to be made between processes on different computers.
  - Example: An iPhone app calling a service on the Internet which returns the weather forecast.



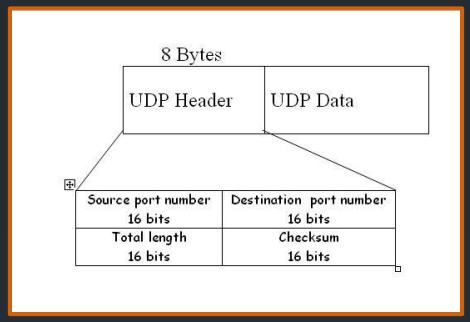
Asynchrony

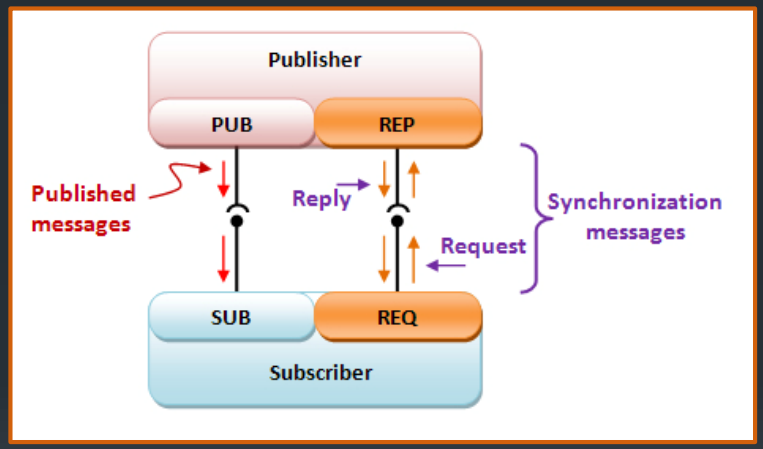
- Synchronous communication
  - Call block until the operation completes
  - Easy to reason about
  - One knows when things complete and what the status is
- Asynchronous communication
  - The caller doesn't wait for the operation to complete
  - May not even care whether or not the operation completes
  - Useful for long-running jobs
  - Good for low latency operations
- Streaming can be sync but is usually async
- Messaging is almost always async
- RPC can be either sync or async
- REST is sync by definition when over HTTP
  - Schemes like chunked responses and long polling attempt to alleviate

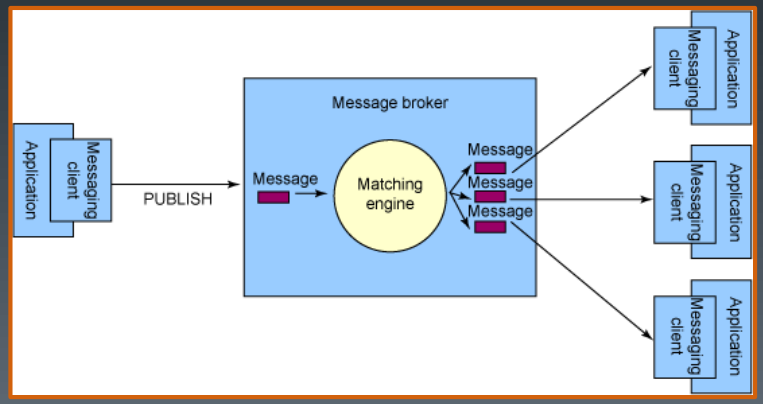


Types of Messaging

- Raw Network Messaging
  - UDP
  - No store, no forward
  - True Multicast, True Broadcast
  - Fast as it gets
- Library Based
  - Can offer some delivery assurances (retry)
  - Deploys in process
  - No server or middleware required
  - Examples
    - Nanomsg
    - ZeroMQ
- Message Oriented Middleware (MOM)
  - Store and forward
  - Pub/sub
  - Perhaps various Transaction Levels
  - Routing
  - More complicated deployment
  - Not the fastest messaging solution
  - Examples
    - NATS
    - Kafka
    - RabbitMQ
    - ActiveMQ
    - MSMQ
    - Redis







AMQP

- Advanced Message Queuing Protocol (AMQP)
- An OASIS open standard
- Application layer protocol for message-oriented middleware
  - message orientation
  - Queuing
  - routing (including point-to-point and publish-and-subscribe)
  - reliability
  - security
- AMQP mandates the behavior of the messaging provider and client to the extent that implementations from different vendors are truly interoperable, in the same way as SMTP, HTTP, FTP, etc. have created interoperable systems
- Previous attempts to standardize middleware have happened at the API level (e.g. JMS) and thus did not ensure interoperability
- Unlike JMS, which merely defines an API, AMQP is a wire-level protocol
- Any tool that can create and interpret messages that conform to this data format can interoperate with any other compliant tool irrespective of implementation language
- Broker Implementations
  - SwiftMQ, a commercial JMS, AMQP 1.0 and AMQP 0.9.1 broker and a free AMQP 1.0 client
  - Windows Azure Service Bus, Microsoft's cloud based messaging service
  - Apache Qpid, an open-source project at the Apache Foundation
  - Apache ActiveMQ, an open-source project at the Apache Foundation
  - Apache Apollo, open-source modified version of the ActiveMQ project at the Apache Foundation (threading functionality replaced and non-blocking techniques implemented more widely)
  - RabbitMQ an open-source project sponsored by Pivotal, supports AMQP 1.0 and other protocols via plugins

Java JMS 2.0

- Java Message Service (JMS) originally released in 2001
- Java based Message Oriented Middleware (MOM) API
- Used to send messages between two or more clients
- JMS is a part of the Java Platform, Enterprise Edition
  - JSR 914
- Allows the communication between different components of a distributed application to be loosely coupled, reliable, and asynchronous
- JMS 2.0 is the first update in over 10 years
  - Simplified API
  - Async send
  - Delayed delivery
  - Scaling through shared topics
- Not the best choice for microservices as it is Java specific (like RMI)

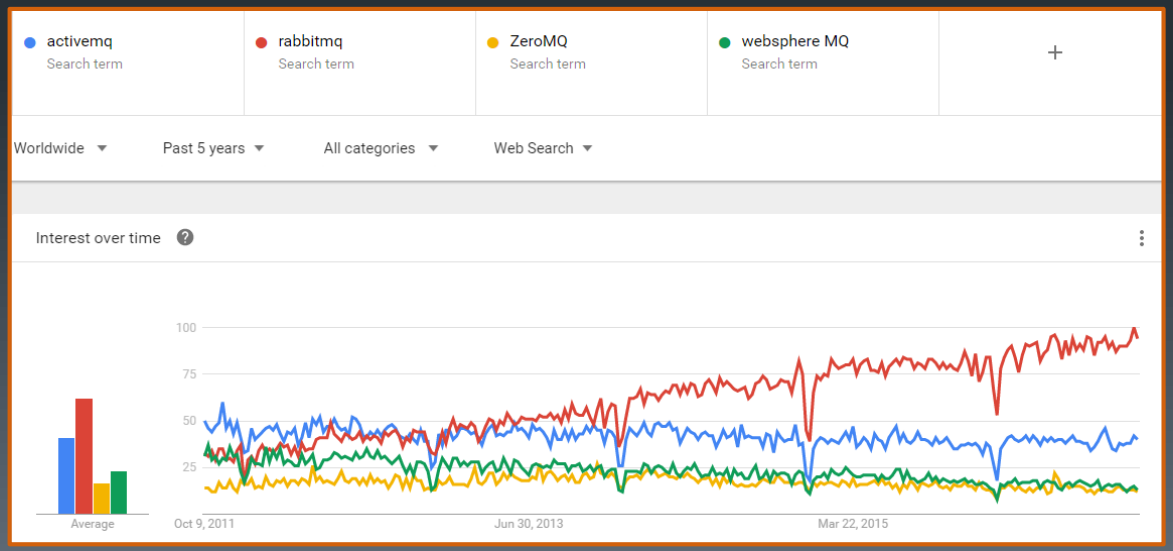
- Many Provider implementations
  - Apache ActiveMQ
  - Apache Qpid, using AMQP[5]
  - Oracle Weblogic and AQ from Oracle
  - EMS from TIBCO
  - FFMQ, GNU LGPL licensed
  - JBoss Messaging and HornetQ
  - JORAM, from the OW2 Consortium
  - Open Message Queue, from Oracle
  - OpenJMS, from The OpenJMS Group
  - Solace JMS from Solace Systems
  - RabbitMQ by Pivotal
  - SAP Process Integration ESB
  - SonicMQ from Progress Software
  - SwiftMQ
  - Tervela
  - Ultra Messaging from Informatica
  - webMethods from Software AG
  - WebSphere MQ FioranoMQ

MSMQ

- Microsoft Message Queuing [MSMQ]
- A message queue implementation developed by Microsoft and deployed in its Windows Server operating systems since Windows NT 4 and Windows 95
- The latest Windows 8 also includes this component
- In addition to its mainstream server platform support, MSMQ has been incorporated into Microsoft Embedded platforms since 1999 and the release of Windows CE 3.0
- Support for three transmission modes
  - Express – not written to disk, no ack
  - Reliable – written to disk, acked
  - Transactional – ACID, DTC eligible
- In .Net Systems, implemented through WCF Reliable Messaging
- Not the best choice for microservices as it is Windows specific

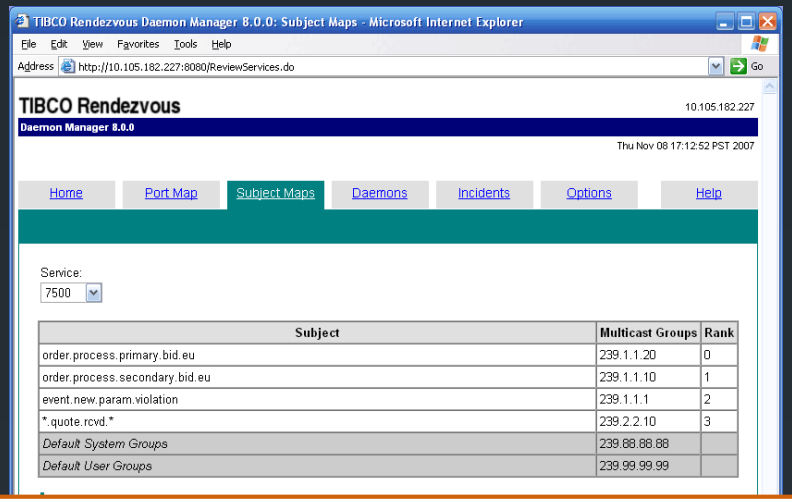
High Profile Messaging Platforms

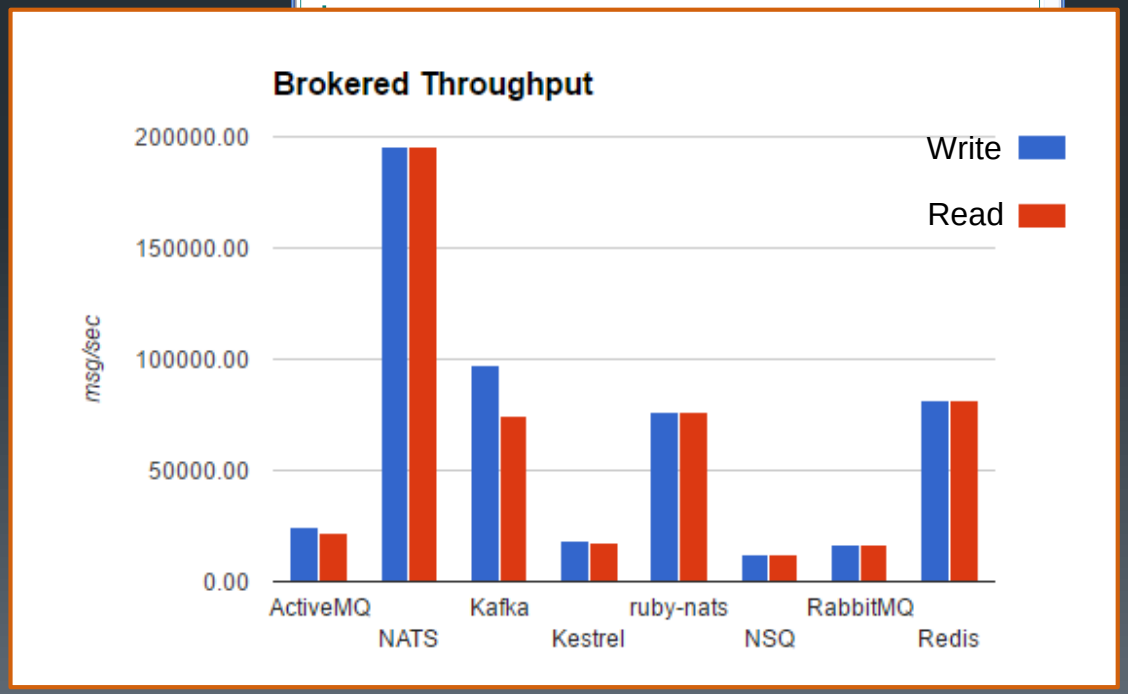
- Websphere MQ – The first MOM system released in 1992
  - Originally known as MQ Series
  - A revelation
  - Still important but loosing the spotlight to open source and others
- ActiveMQ – Most popular community open source MQ system
  - Supports everything (JMS, JDBC, STOMP, XMPP, MQTT, REST, OpenWrite, …)
  - **Apache Apollo** – rewrite of ActiveMQ in Scala, much simpler and faster
- Apache QPID – AMQP driven messaging (100% AMQP support)
  - RedHat MRG enterprise messaging product is based on this
- RabbitMQ – Most popular commercially backed open source MQ system
  - Erlang based, elegant, fast, reliable
- ZeroMQ – Library based messaging, thus very fast, but brokerless, requiring rendezvous



High Performance Messaging Systems

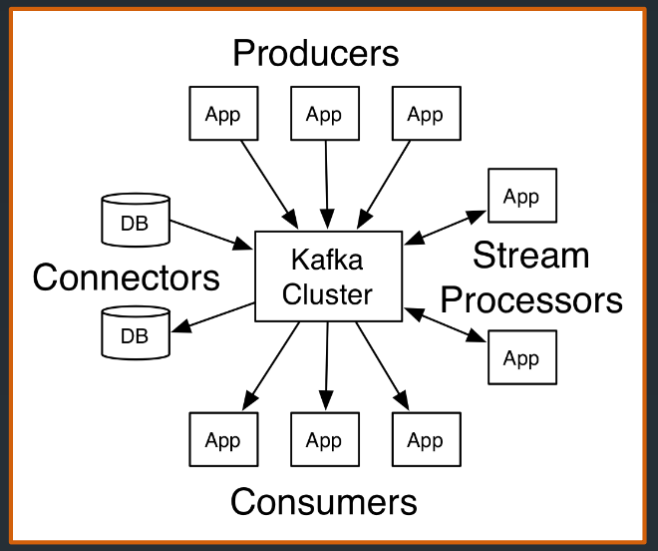
- Tibco Rendezvous [RV] – frequently used in financial applications, fast, expensive
- Informatica Ultra Messaging – Formerly 29West, fast, expensive (a little cheaper than Tibco)
- ZeroMQ – High performance library only messaging, open source
- NATS - open source messaging broker designed for microservices and cloud native applications
- Apache Kafka - a distributed platform supporting pub/sub, processing and storage for streams of data in a distributed, replicated cluster, originally created at LinkedIn
- Apache Pulsar – a distributed platform supporting queuing and pub-sub messaging, originally created at Yahoo

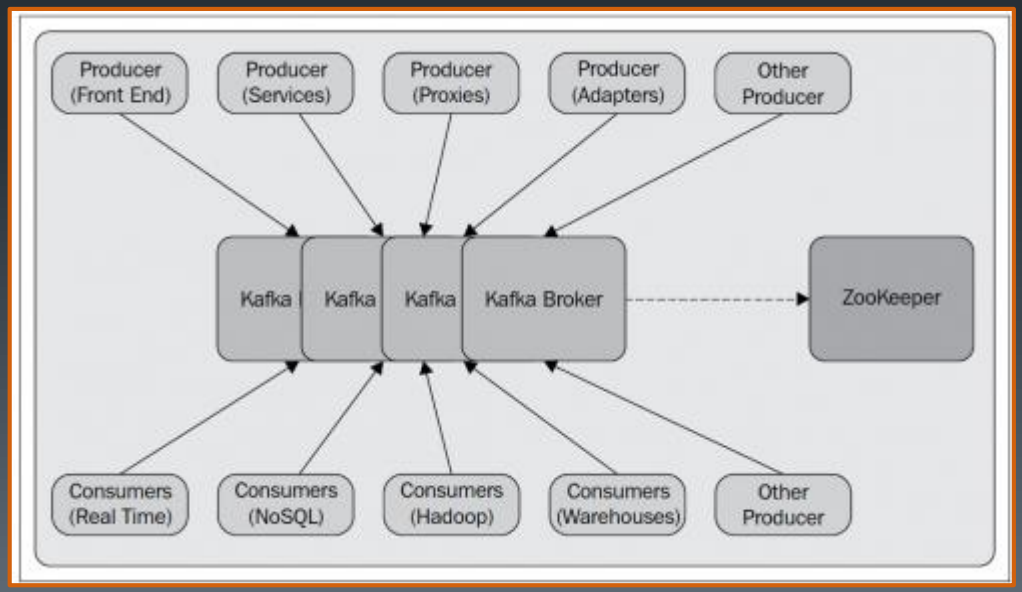




Distributed Messaging

- Apache Kafka
  - Open source, publish-subscribe message broker
  - Amazon Kinesis Streams, managed Kafka
- Implemented as a distributed commit log
- Written in Scala
- Unified, high-throughput, low-latency platform for handling real-time data feeds
- Developed at LinkedIn, open sourced in 2011
- Designed to allow a single cluster to serve as the central data backbone for an organization
- Can be elastically and transparently expanded without downtime
- A single Kafka broker can handle hundreds of megabytes of reads and writes per second from thousands of clients
- Data streams are partitioned and replicated over a cluster of machines
  - Allows data streams larger than the capability of any single machine and to allow clusters of coordinated consumers



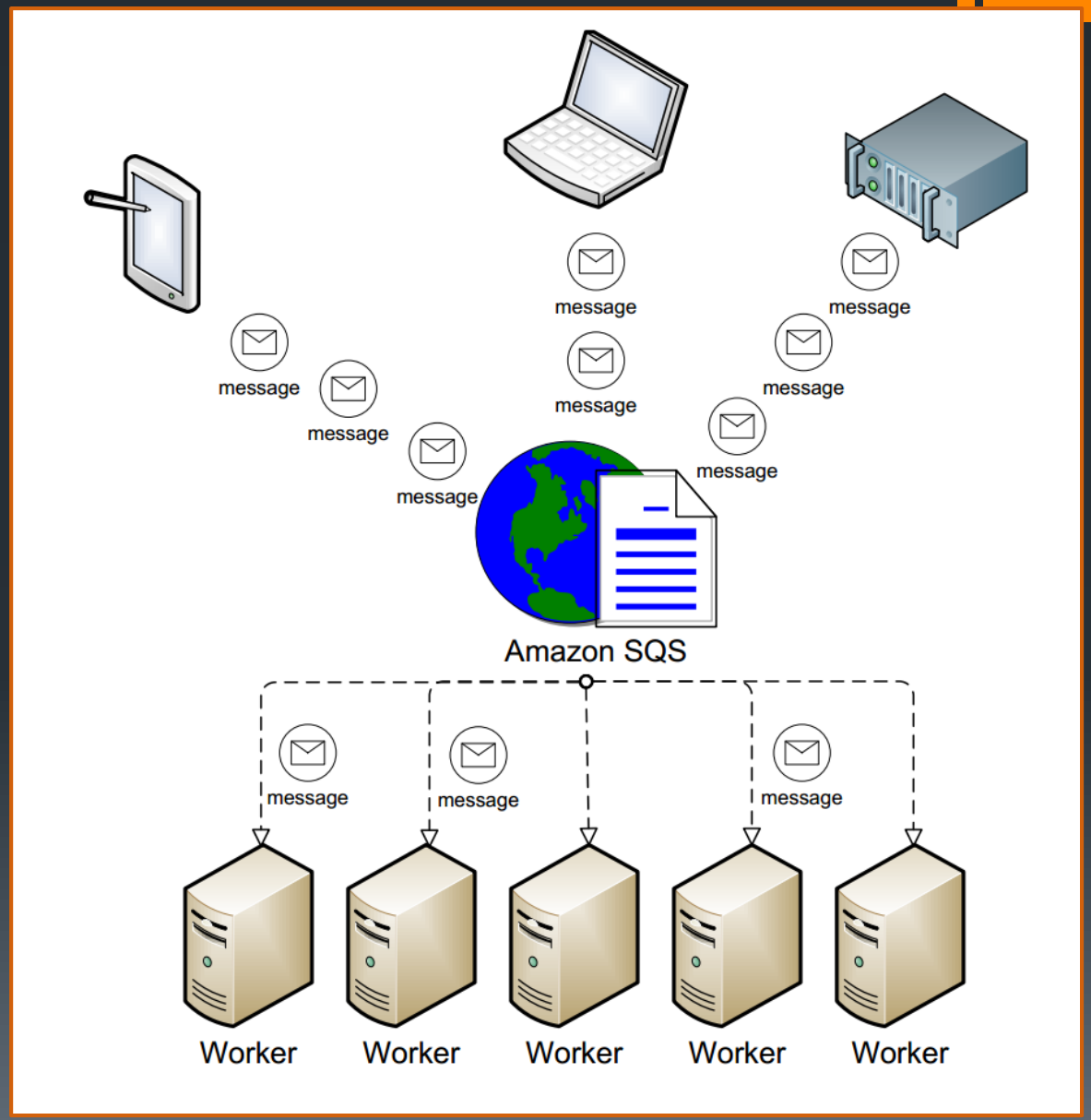


Cloud Based Messaging

- Amazon Simple Queue Service (SQS) is a fast, reliable, scalable, fully managed message queuing service
  - SQS is simple and can transmit large volumes of data, at any level of throughput, without losing messages or requiring other services to be available
- Azure Queue
- Google Cloud Pub/Sub
- Rackspace offers CloudQueues
- Heroku offers RabbitMQ
- EngineYard offers IronMQ
- Etc.

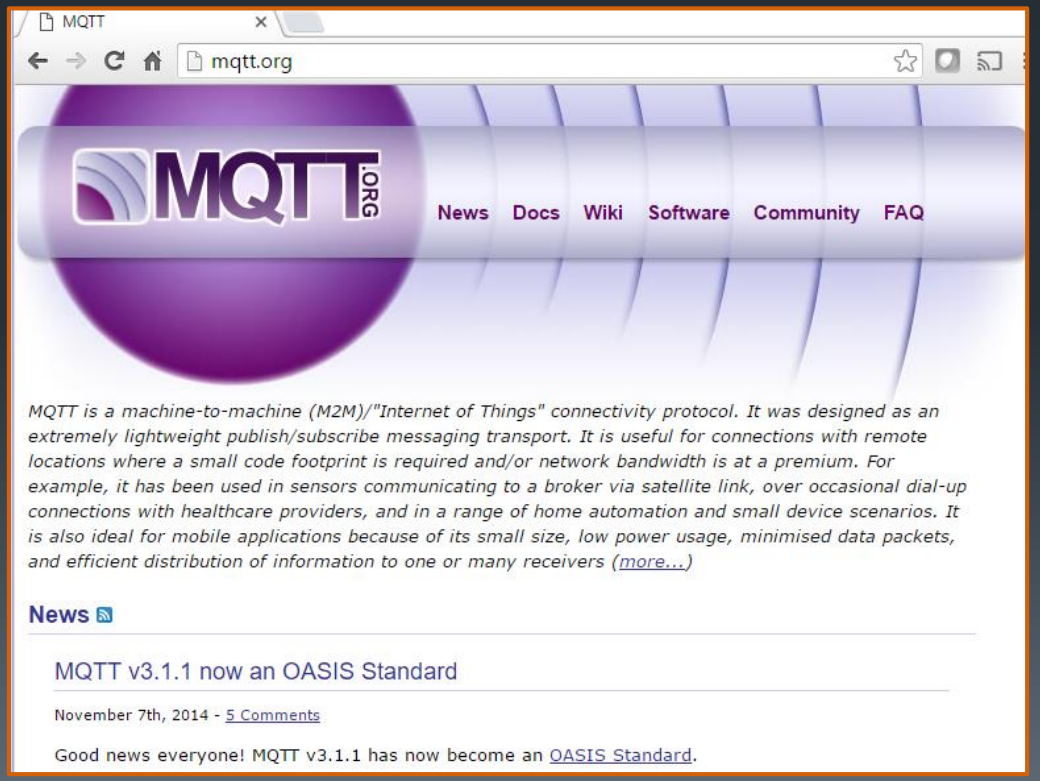
Amazon SQS

- Amazon Simple Queue Service (SQS) is a reliable distributed messaging system
  - A reasonable choice for fault-tolerant applications
  - At least once or at most once delivery
- Messages are stored in user defined queues
  - Each queue is accessed by URL
  - Available to the Internet
  - An Access Control List (ACL) determines access to the queue
- Any messages that you send to a queue are retained for up to four days (or until they are read and deleted by an application)
  - Messages read are hidden for the hide window time
  - Message delivery can have a preconfigured delay
- Amazon Kinesis, a fully managed real-time streaming data service
  - Hosted Apache Kafka
  - Designed for streaming big data
  - Multiple applications can receive the same data
  - Data is delivered in order and cached for up to 24 hours
    - i.e. A client can replay data from the prior 24 hours



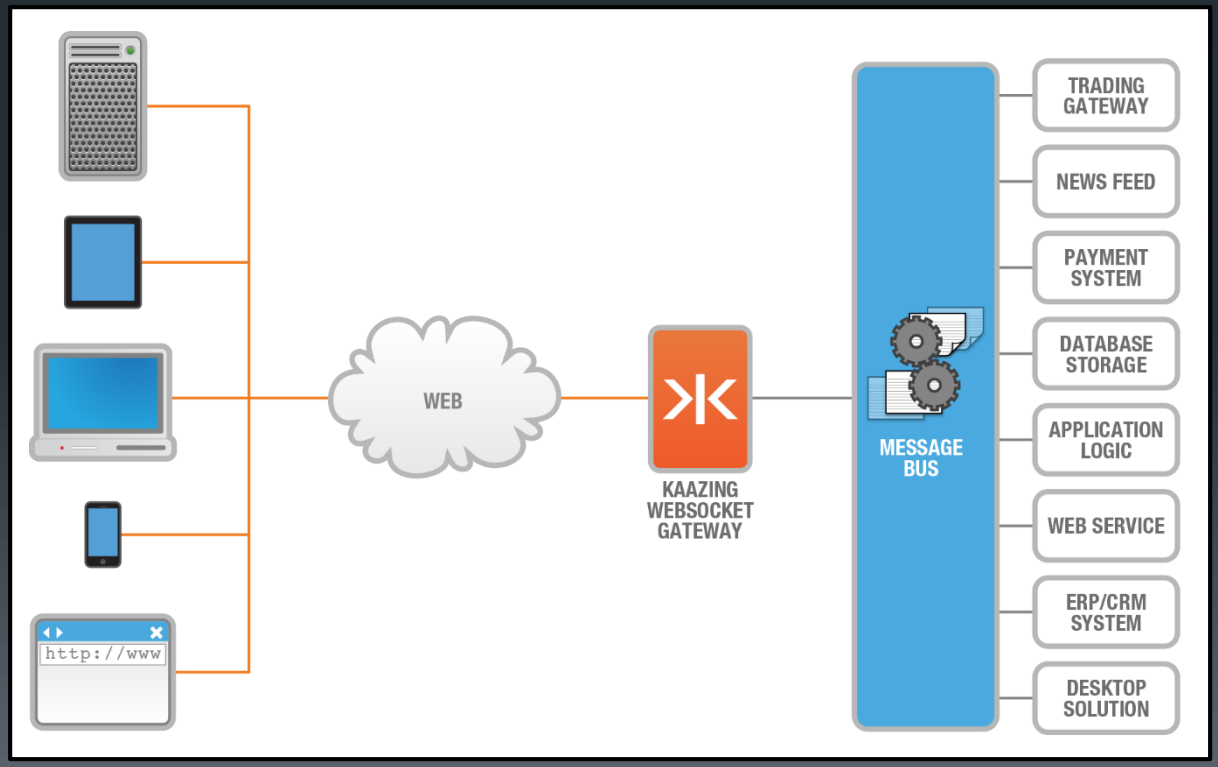
IoT Messaging

- MQTT (formerly MQ Telemetry Transport)
  - ISO standard PRF 20922
  - Publish-subscribe-based "lightweight" messaging protocol for use on top of TCP/IP
  - Designed for connections with remote locations where a "small code footprint" is required or the network bandwidth is limited
- Powers Facebook Messanger
- The publish-subscribe messaging pattern requires a message broker
  - The broker is responsible for distributing messages to interested clients based on the topic of a message
  - MQTT can also work without a broker
- 1999: Cirrus Link Solutions Stanford-Clark/Nipper develop MQTT
- 2013: IBM submitted MQTT v3.1 to OASIS
  - The "MQ" in "MQTT" came from IBM's MQ Series message queuing product
- MQTT-SN is a variation of the main protocol aimed at embedded devices on non-TCP/IP networks, such as ZigBee
- Alternative protocols include:
  - Advanced Message Queuing Protocol
  - IETF Constrained Application Protocol
  - XMPP
  - Web Application Messaging Protocol (WAMP)



Web Socket

- Web Socket brings high performance messaging to the Web
  - Suitable for Messaging and Streaming
- Products like Kaazing and empowering richer JavaScript applications in the browser
  - Kaazing provides a messaging protocol which operates over Web Sockets
- The PaaS EngineYard offers Pusher, a messaging solution over Web Sockets



Summary

- Loosely coupled systems promote service independence
- Asynchronous systems do not wait for operations to complete before returning
- Event based systems allow subscribers to come and go without disturbing the sender
- A wide range of messaging platforms with several key target markets exist, many of which are useful in microservice based applications

Lab 4

- Building an RPC service

5: Cloud Native Transactions and Event Sourcing

Objectives

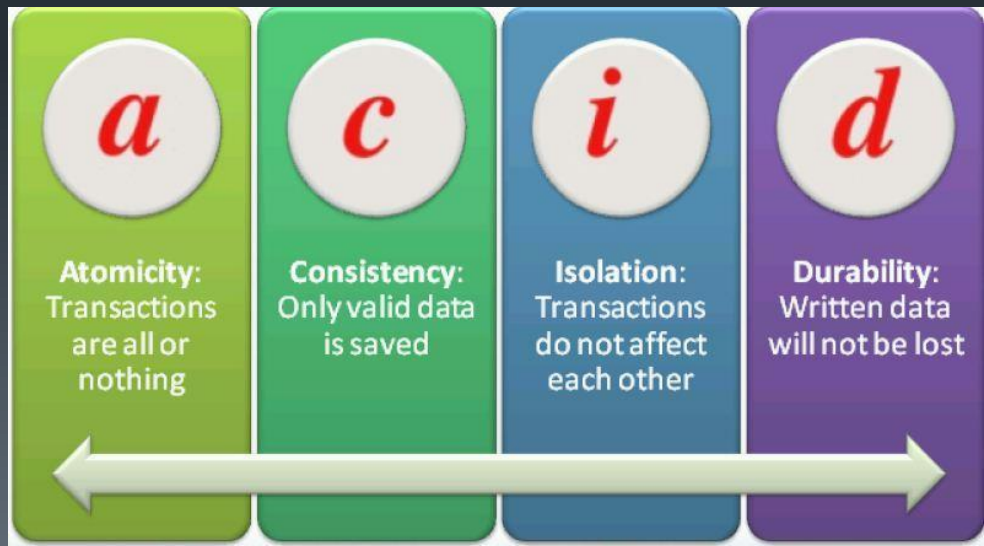
- Define transaction
- Discuss the challenges facing distributed transactions
- Contrast ACID transactions with eventual consistency
- Explain the CAP theorem
- Describe the value of Etags, Event Sourcing and CQRS

State

- State def:
  - All the stored information, at a given instant in time, to which the program has access
- The output of a computer program at any time is completely determined by its current inputs and its state
  - If your state goes bad, your application breaks
- State management presents the single most challenging facet of distributed application design

ACID

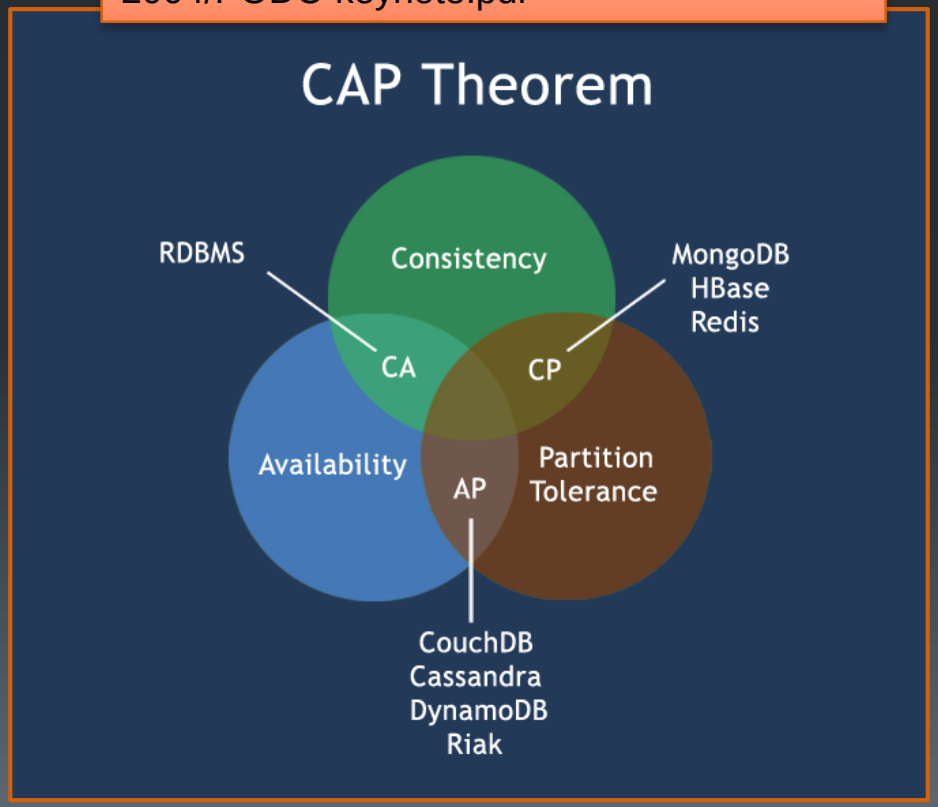
- ACID is a set of properties that guarantee that database transactions are processed reliably
  - Atomicity - requires that each transaction is "all or nothing"
  - Consistency - ensures that any transaction will bring the database from one valid state to another
  - Isolation - ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially
  - Durability - means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors
- Transactions - In the context of databases, a single logical operation on the data is called a transaction
- Jim Gray defined the properties of a reliable transaction system in the late 1970s and developed technologies to achieve them automatically
- In 1983, Andreas Reuter and Theo Härder coined the acronym ACID to describe them



The CAP Theorem

- The CAP theorem states that it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:
  - Consistency - all nodes see the same data at the same time
  - Availability - guarantees every request receives a response whether successful or failed
  - Partition tolerance - the system operates despite message loss or failure of part of the system
- CAP consistency means that any data item has a value reached by applying all the prior updates in some agreed-upon order
  - A consistent service must never forget an update once it has been accepted and the client has been sent a reply
- Availability is a mixture of performance and fault-tolerance
  - A service should keep running and offer rapid responses even if a few replicas have crashed or are unresponsive, and even if some of the data sources it needs are inaccessible
  - No client is ever left waiting, even if we cannot get the needed data
- Partition tolerance means that a system should be able to keep running even if the network itself fails, cutting off some nodes from the others
  - Partitioning is not an issue within modern data centers only across data centers

http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf



The CAP impact on the Cloud

- The CAP Principle was developed by Berkeley Professor Eric Brewer (Brewer 2000)
  - A CAP Theorem was thereafter proved by MIT researchers Seth Gilbert and Nancy Lynch (Gilbert and Lynch 2002)
  - The theorem addresses a strict ACID view of consistency
  - Brewer has since made three key points
    1. Because partitions are rare, there is little reason to forfeit C or A when the system is not partitioned
    2. The choice between C and A can occur many times within the same system at very fine granularity
       - Not only can subsystems make different choices, but the choice can change according to the operation or even the specific data or user involved
    3. All three properties are more continuous than binary
- Brewer argues that the value of quick responses is so high that even a response based on stale data is often preferable to leaving the external client waiting

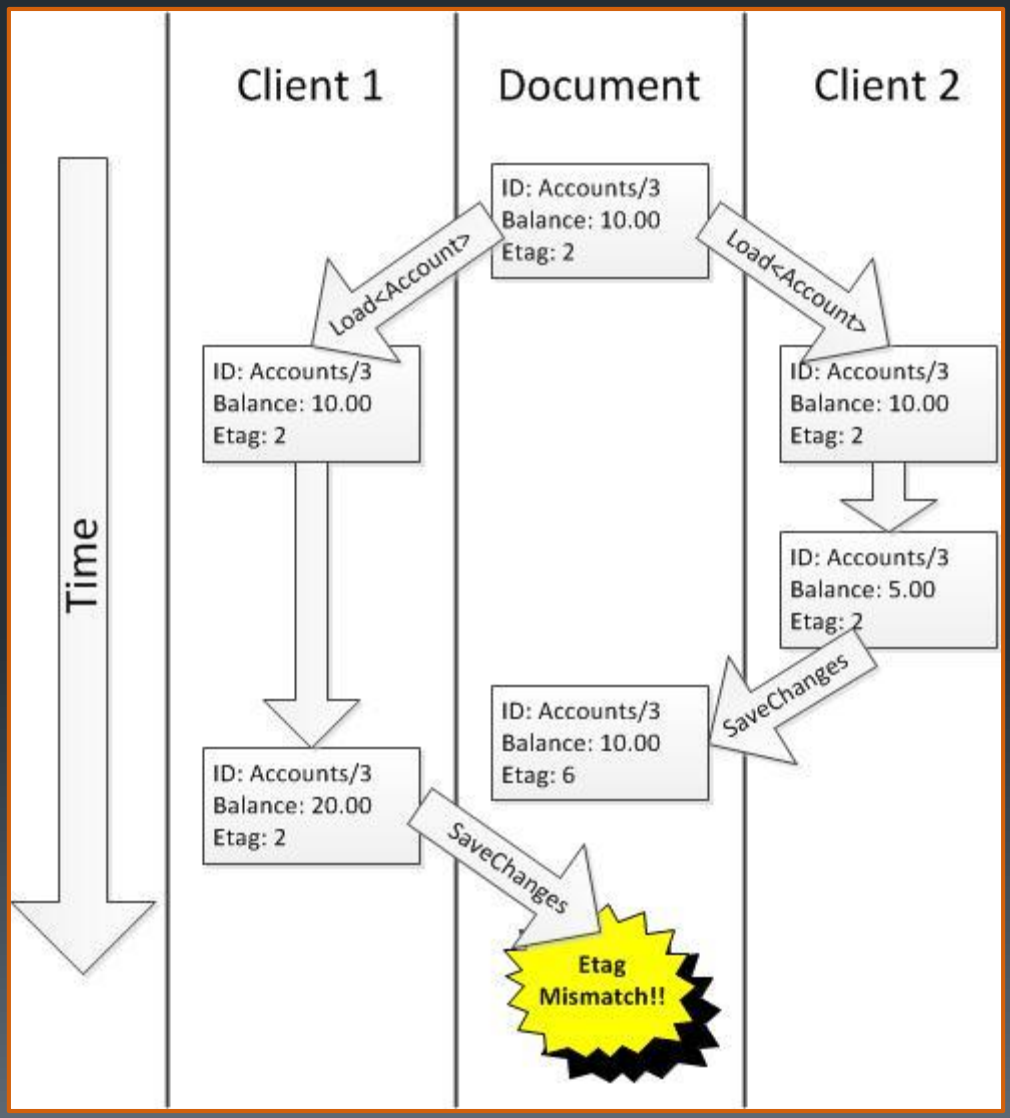
When a web page renders some content with a “broken” icon to designate missing or unavailable content, we are seeing CAP in action

How CAP plays out in application design

- Brewer argues, that it is better to initially run in an optimistic mode when designing large scale applications
  - E.g. Booking the sale without checking the remaining inventory stock
  - A question of priorities
    - scale and performance versus absolute accuracy
- The CAP theorem suggests that sometimes a system should run acceptable risks if by doing so it can offer better responsiveness and scalability
- Consistency in CAP is really a short-hand for two properties
  - An order-based consistency property
    - Updates to data replicated in the system will be applied in the same order at all replicas
  - A durability property
    - Once the system has committed to do an update, it will not be lost or rolled back
- This conflation of consistency with durability is important
  - Durability is expensive and turns out to be of limited value in the first tier of the cloud where services do not keep permanent state

Distributed State

- The web works so well because it is based on:
  - Representations of resources
    - You never have the real resource state, only a representation
    - Your representation may be old the moment you receive it
  - Caching
    - Many web servers would be crushed if they had to directly support all of the clients using pages that they are responsible for
    - The fabric of caches on the web is massive and offloads many servers by one or more orders of magnitude
- The idea that the data you receive might be stale as soon as you receive it is tied up with eventual consistency
- Etags can be used as a state versioning mechanism to ensure that state changes are only applied if you are changing the most recent state

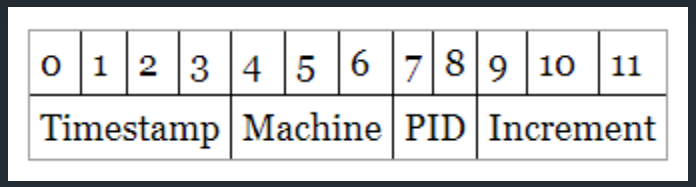


Key Generation

- Keys are a critical feature of distributed data systems
- Keys allow objects to be uniquely identified
  - This is usually a prerequisite for Idempotence
- For this reason keys are best generated at the time of datum/entity creation or ingestion
- Potential Key Algorithm Input Factors:
  - Explicitly assigned prefix
  - Machine ID
    - Process ID
      - Thread ID
  - Time
    - Perhaps highly granular (microsecond, nanosecond, etc.)
  - Monotonically increasing counters
  - Foreign/External Keys
- Unique keys require a set of factors sufficient to eliminate all orthogonal vectors of replication
- Critical to many Idempotent interface implementations

Considerations:

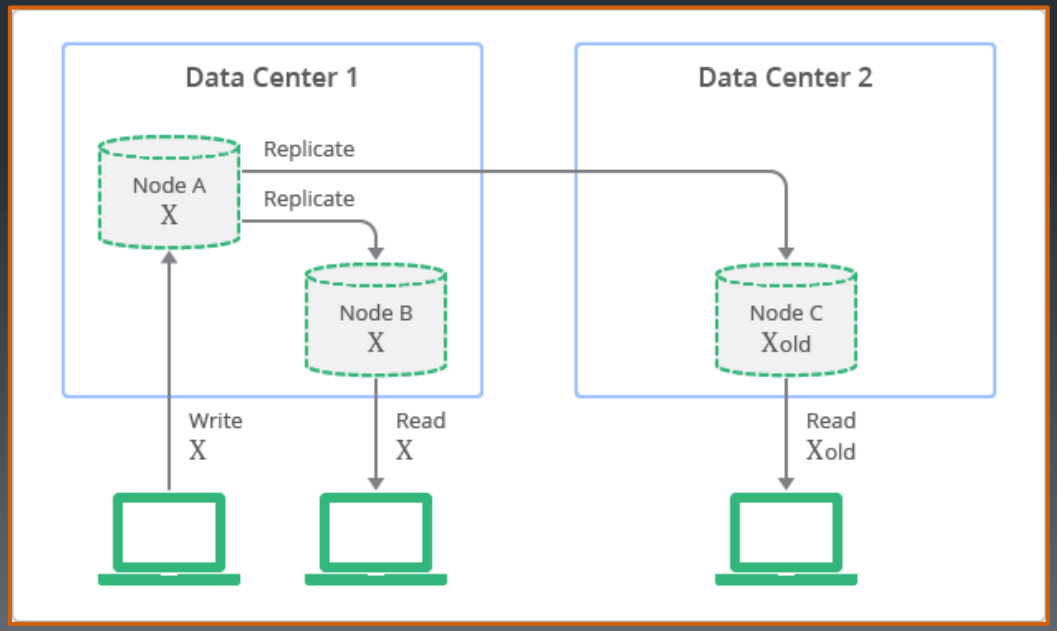
- Key compare time/rate
- Key generation time/rate
- Key storage (data & index)
- Key clustering
- Composite keys and implicit foreign keys



- MongoDB ObjectID (OID) example
- 12 bytes
- 0-3: timestamp in seconds since epoch
  - Provides uniqueness at the granularity of a second
  - The timestamp comes first so OIDs will sort in roughly insertion order, making OIDs efficient to index
  - Implicit creation timestamp
- 4-6: unique machine identifier (usually a hash of the machine's hostname)
  - Statistically ensures that OIDs on different machines will not collide
- 7-8: Process ID
  - Ensures concurrent processes generate unique IDs on the same machine
- 9-11: Increment responsible for uniqueness within a second in a single process
  - Allows up to 256^3 (16,777,216) unique OIDs to be generated per process in a single second
  - Must be synchronized in a multithreaded process

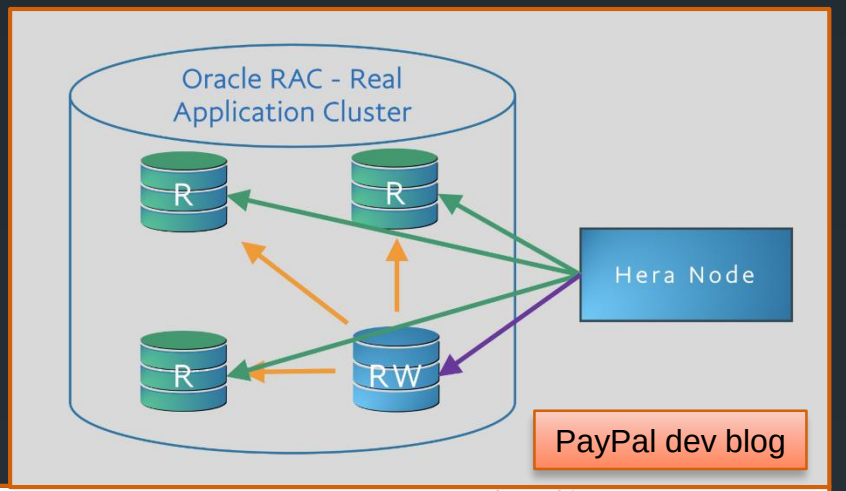
Eventual consistency

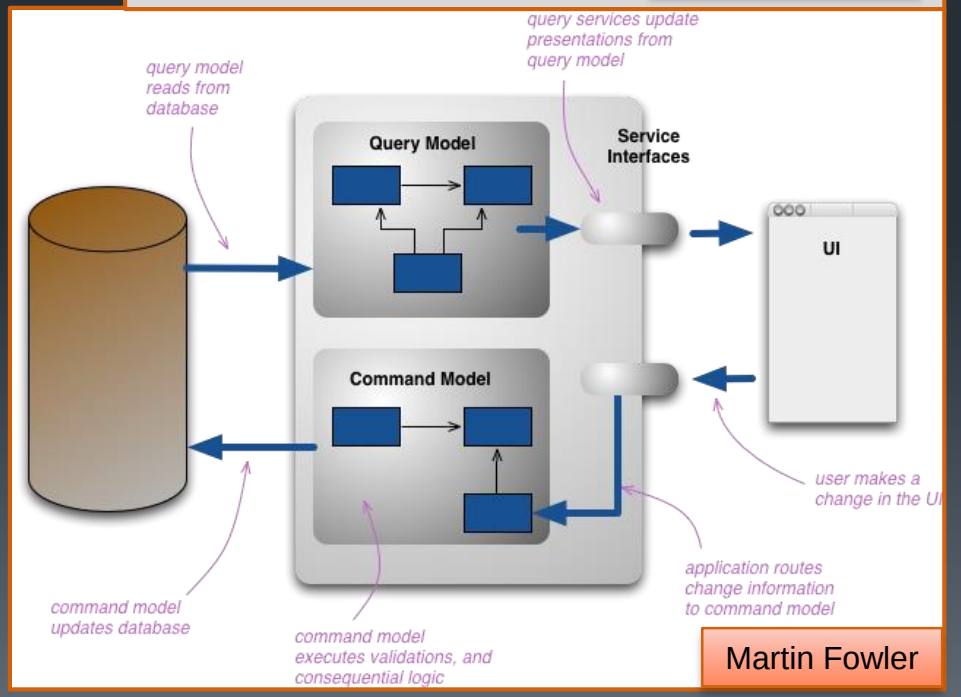
- Eventual Consistency
  - A consistency model used in distributed computing
  - Achieves high availability that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value
- Quorum based systems can ensure strong consistency
  - Consul, etcd, zookeeper, etc.
  - Do not scale well due to the high communications overhead between nodes
  - Good for special purposes
    - Leader election
    - Small key/value bits of cluster state
  - Bad for large data storage
- Many times you can trade things you do not need for things you do
  - Read your own writes
    - Trading global consistency for local consistency
    - Others may not see your writes immediately but you will



CQRS

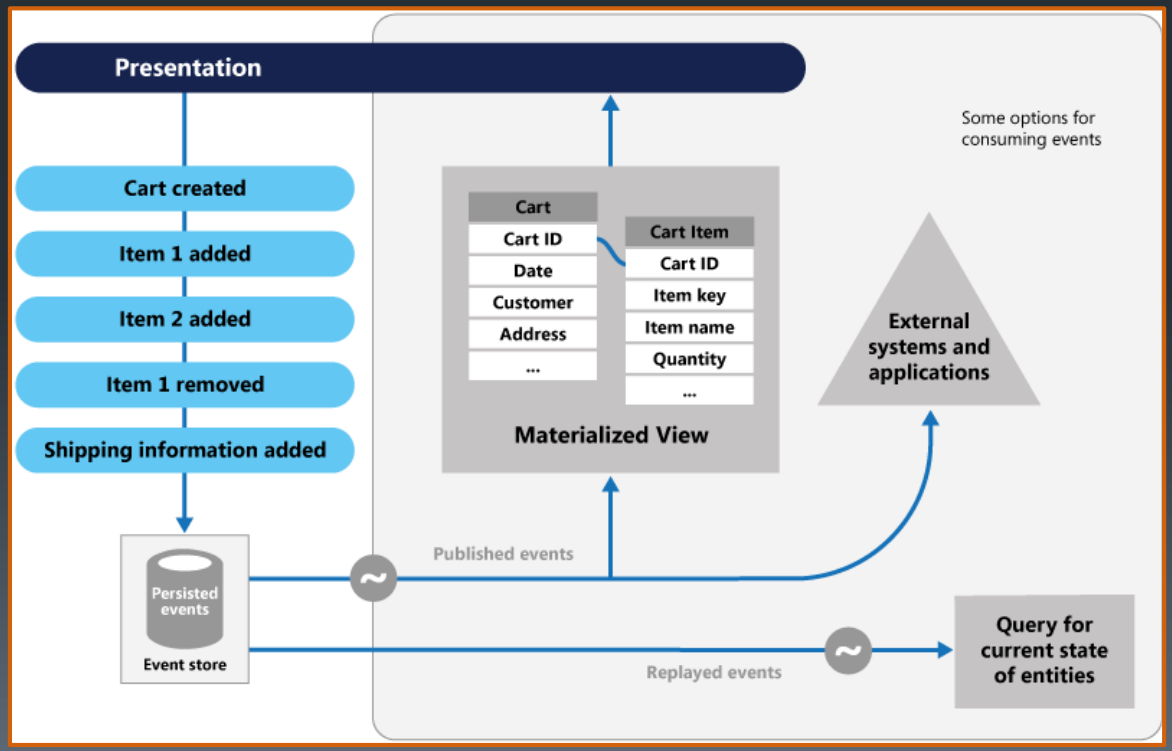
- Command Query Responsibility Segregation
  - A pattern using one model to modify state and another to retrieve it
- Can solve intractable problems or make things intractable
  - Should only be used when appropriate (tax on incorrect usage is high than with most patterns)
- CQRS differs from CRUD, which uses a single (typically synchronous) model for all operations
  - CRUD = create/read/update/delete on (typically) records in a database
- CQRS Commands
  - Modify state
  - Queued and asynchronous
  - Perform more expensive operations
    - Often need to be atomic and/or serialized, perhaps involving some locking
- CQRS Queries
  - Retrieve data
  - Typically synchronous in CQRS
  - Cheap and fast
    - Can often be performed lock less
  - Must tolerate eventual consistency
    - Read your own writes becomes difficult





Event sourcing

- Event Sourcing is the process of capturing all inputs as events
- Often combined with eager read derivation
  - Reads don't touch the main database
  - Reporting databases service most reads and are structured to accept all state change events and derive the typical read data model in advance of any user requests for data
  - This makes Queries even faster
- The state of an entity is established by looking at its event history
  - Each event is immutable
  - Balances and other "summary" type states can be computed and optionally cached if fast access is required
    - However the event history is always canonical

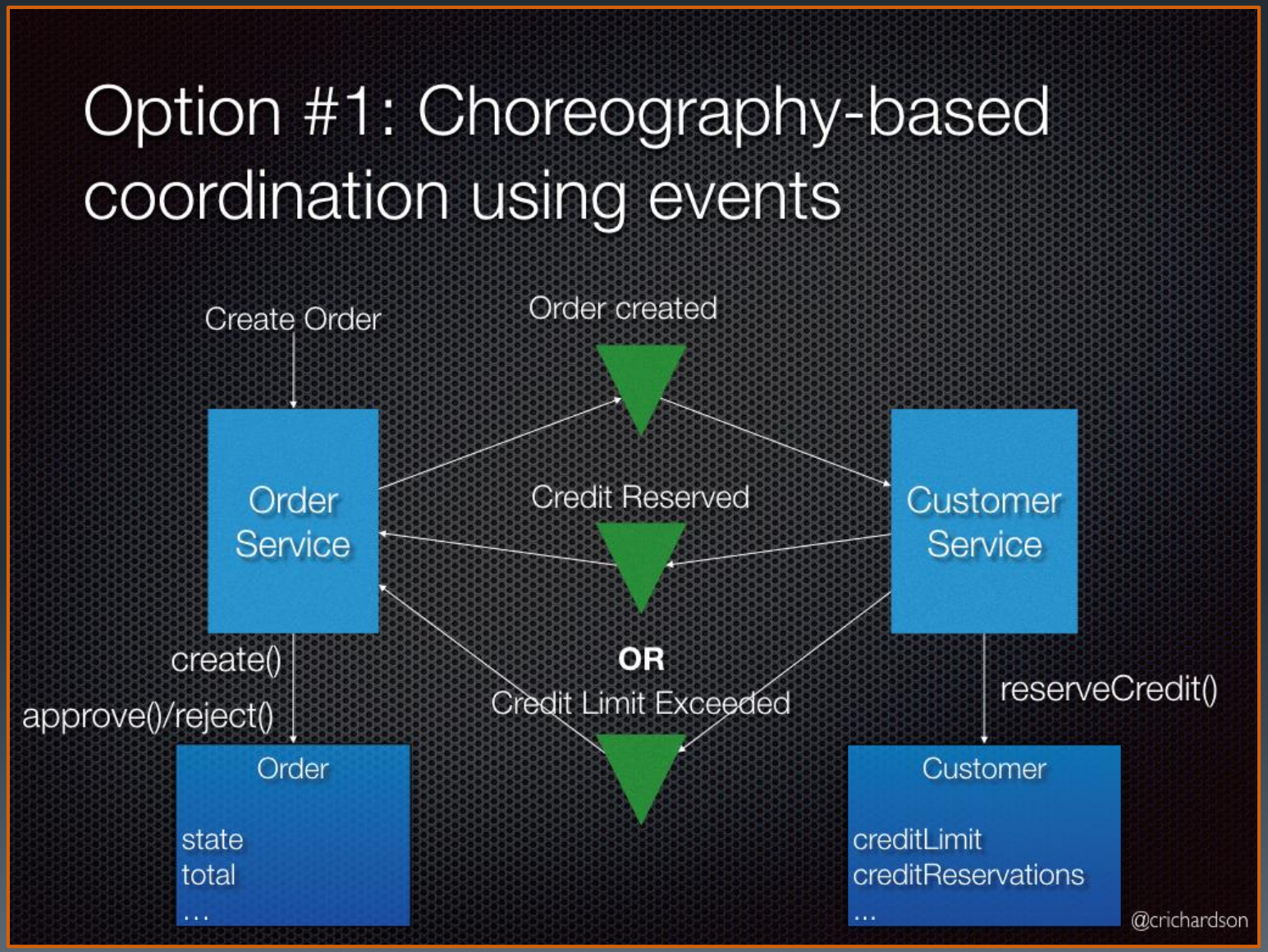


The Saga Pattern

- Microservices are typically designed such that each service has its own database if it is stateful
- Some business transactions span multiple service
- A mechanism may be needed to ensure data consistency across services
  - E.g. an e-commerce store where customers have a credit limit, the application must ensure that a new order will not exceed the customer's credit limit, requiring coordination between the Order and Customer services
- A saga is a sequence of local transactions
  - Each local transaction updates the local state and publishes a message or event to trigger the next local transaction in the saga
  - If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions
- There are two common ways to coordinate sagas:
  - Choreography - each local transaction publishes domain events that trigger local transactions in other services
  - Orchestration - an orchestrator (object) tells the participants what local transactions to execute

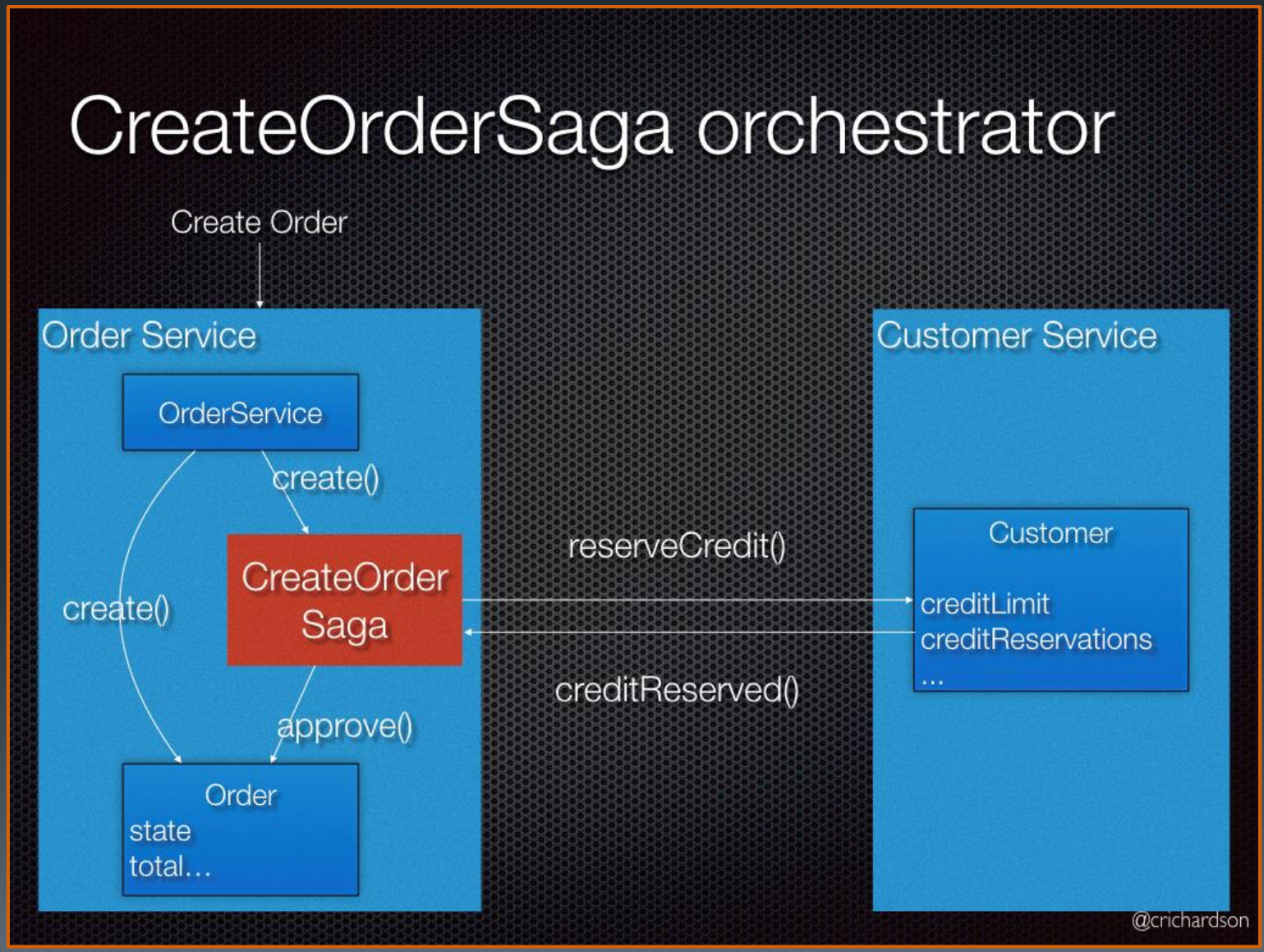
Choreography

1. The Order Service creates an Order in a pending state and publishes an OrderCreated event
2. The Customer Service receives the event attempts to reserve credit for that Order. It publishes either a Credit Reserved event or a CreditLimitExceeded event.
3. The Order Service receives the event and changes the state of the order to either approved or cancelled



Orchestration

1. The Order Service creates an Order in a pending state and creates a CreateOrderSaga
2. The CreateOrderSaga sends a ReserveCredit command to the Customer Service
3. The Customer Service attempts to reserve credit for that Order and sends back a reply
4. The CreateOrderSaga receives the reply and sends either an ApproveOrder or RejectOrder command to the Order Service
5. The Order Service changes the state of the order to either approved or cancelled



Summary

- A transaction is a sequence of operations performed as a single logical unit of work
- Per the CAP theorem, distributed systems must balance consistency with availability when partitioned
- Eventual consistency implies that the state of an object at any given moment may be viewed differently in different parts of a distributed system but will eventually converge
- Etags are a web centric means of ensuring that one only updates a remote object if the object is in the expected pre-state
- Event Sourcing is a means to decouple state managers by generating broadcast events associated with all state changes
- CQRS – Command Query Responsibility Separation is a process where expensive state change commands are queued and fast/cheap state queries are synchronous

Lab 5

- Create a message driven loosely coupled service

6: Stateless Services and Polyglot Persistence

Objectives

- Describe the range of state managers are available for use in microservice based systems
- Explain the differences between various database types
  - Relational
  - Graph
  - Key/Value
  - Document
  - Column
- Explore the importance of Schemas

Relational Databases

- Relational databases still manage most of the world's critical data
- ACID Transactions
- Perfect for shared database integration [Hohpe and Woolf]
  - One database hosting data for a range of applications
- Single Source of Truth
- Years of refinement and evolution
  - No data platform type is more widely supported
- One system to manage
  - Concurrency is notoriously difficult to get right
  - Errors can trap even the most careful programmers
  - Relational systems scale vertically, sidestepping many concurrency concerns and managing most of the rest for you

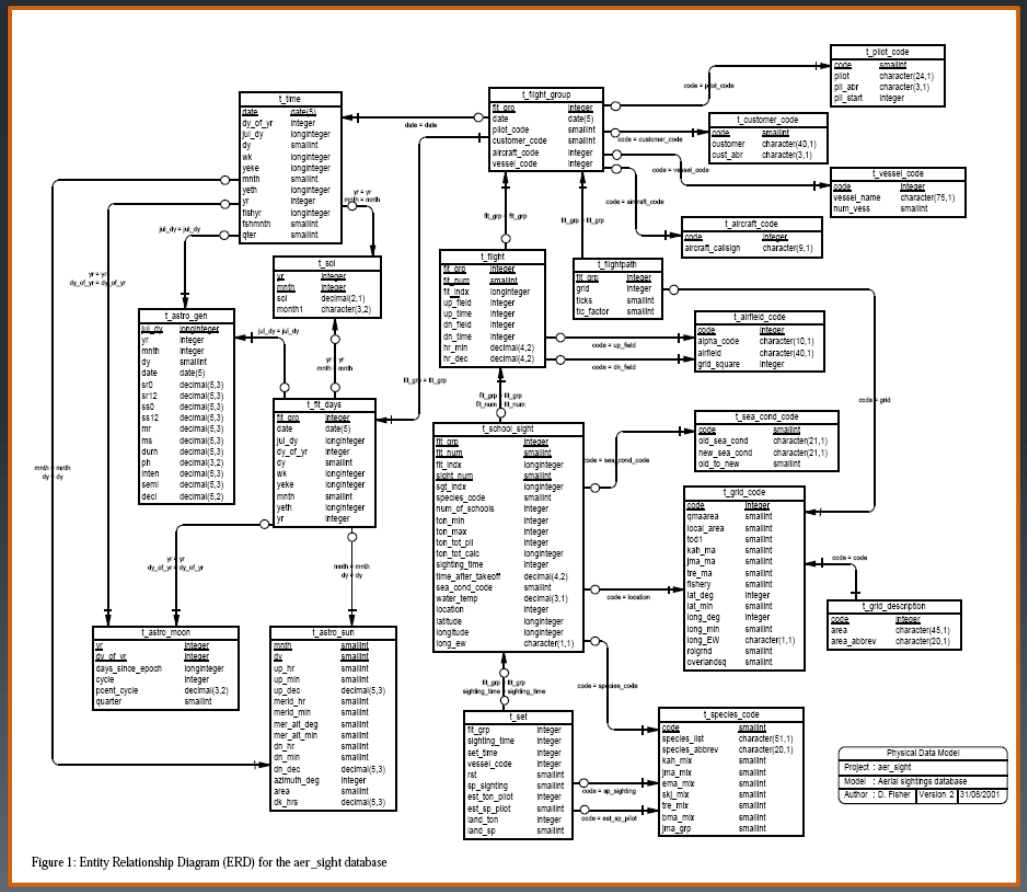
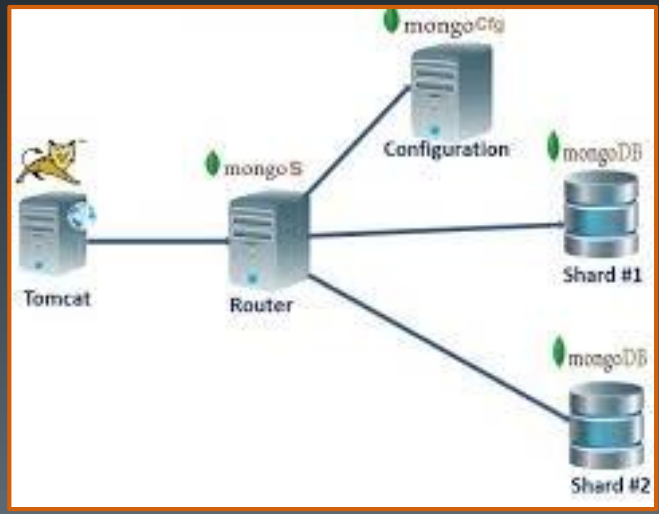


Figure 1: Entity Relationship Diagram (ERD) for the aer_sight database

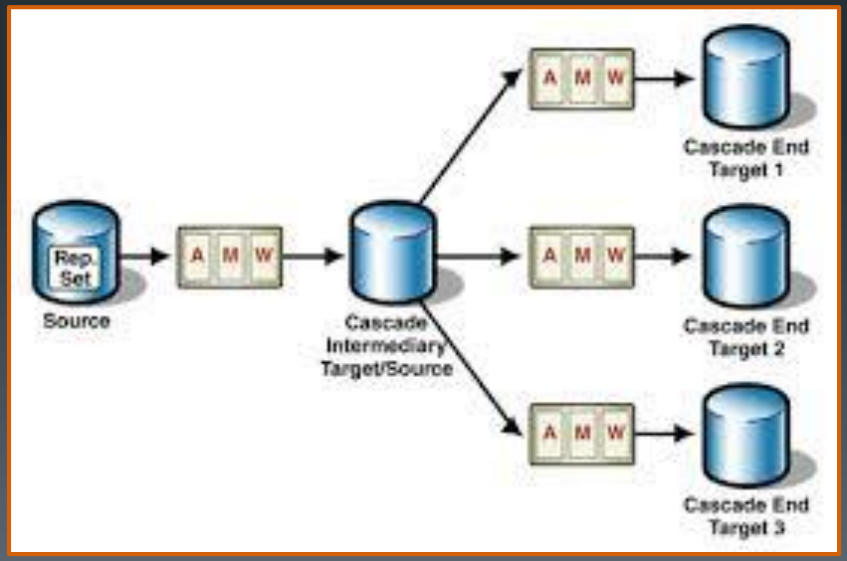
Database Scale Out

- There are two basic means used to distributed data
  - Replication
    - Copies the same data to multiple nodes
  - Partitioning
    - Vertical – sets of columns are placed on different nodes
    - Horizontal (aka Sharding) – sets of rows are placed on different nodes
- Replication and Partitioning can be combined to create an array of data distribution patterns
- Single server databases are the easiest to manage
  - Not everyone is Google
  - If a quality server and a fast database can handle your data you may be much better off keeping things simple
    - Single server solutions run on reliable hardware (not commodity)
    - Redundant power supplies and network interfaces, RAID disks, monitoring hardware, robust manufacturing QA, etc.
  - In memory cache and hot standby solutions allow single server models to run fast and reliably
  - Distributed systems create unwanted overhead, don't do it if you don't have to!
- If you plan to shard do so early and during a slow time (when you have plenty of head room)
  - Enabling partitioning on many systems will cause heavy data rebalancing and replication traffic



Replication Benefits

- Read Scale: replication enables scale out architectures to be designed involving a master server and many replicas sharing application read load
- Availability: system redundancy imparted by replication allows for highly available solutions
- Geographic Distribution: Replicas can be placed in geographically diverse locations, close to their user
- Task Offloading: Replicas present excellent platforms for backups, analytics, application development and testing environments



Why NoSQL

- Impedance mismatch
  - the difference between the relational model and the in-memory data structures is often substantial
  - This requires data translation
  - Relational data is organized into tables and rows (relations and tuples)
    - A tuple is a set of name -value pairs and a relation is a set of tuples
  - SOA services use richer data structures with nested records and lists
    - Represented as documents in XML, JSON, etc.
  - Reducing the number of server round trips makes a rich structure desirable
- Relational systems (due to joins and ACID) are difficult to scale horizontally
  - Join performance and highly structured data also makes them ill suited for today's "big data" environments
- A cluster of small machines can use commodity hardware
  - Scaling out
  - Cheaper
  - More resilient
- NoSQL systems process aggregates not tuples and are typically designed for clustering and unstructured data
  - Good support for nonuniform data
- SQL is a powerful and well understood language
  - Requires significant complexity in the underlying platform
  - Not present in NoSQL platforms
  - SQL like solutions are present on many NoSQL solutions
    - Hive
    - Cassandra CQL

NoSQL Data Models

NoSQL Attributes:

- Not using the relational model
- Running well on clusters
- Open-source
- Built for the 21st century web estates
- Schemaless

KeyValue

- Redis
- Memcached
- Riak
- BerkeleyDB
- Hazelcast
- LevelDB

Document

- AWS DynamoDB
- MongoDB
- CouchDB
- Couchbase
- MarkLogic
- OrientDB
- ArangoDB
- RavenDB

Column

- Cassandra
- AWS SimpleDB
- HBase
- Accumulo
- Hypertable

Graph

- Neo4J
- FlockDB
- Virtuoso
- Giraph
- Neptune
- Titan (can use DynamoDB as backend)
- OrientDB
- ArangoDB

AWS Solutions in orange

NoSQL Considerations

- NoSQL databases have no explicit schema
  - Schemaless databases have implicit schemas
  - Changing the structure of data during the life of an application has an impact on the application
- Many NoSQL databases operate on aggregates
  - An aggregate may be a document, column set, block, etc.
- NoSQL databases work best as application databases
  - Application databases serve a single application or application component
  - Application databases are often wrapped in SOA services
  - A strength of NoSQL solutions is their ability to map directly to application constructs
  - NoSQL databases do not always work well in multitenant (application) scenarios where more general solutions are more broadly applicable
  - Polyglot Persistence using different data stores in different circumstances
- Choosing the best NoSQL solution involves:
  - Selecting a programming model and choosing a well aligned data storage model
  - Selecting the simplest storage solution that provides the scale, performance and growth potential required

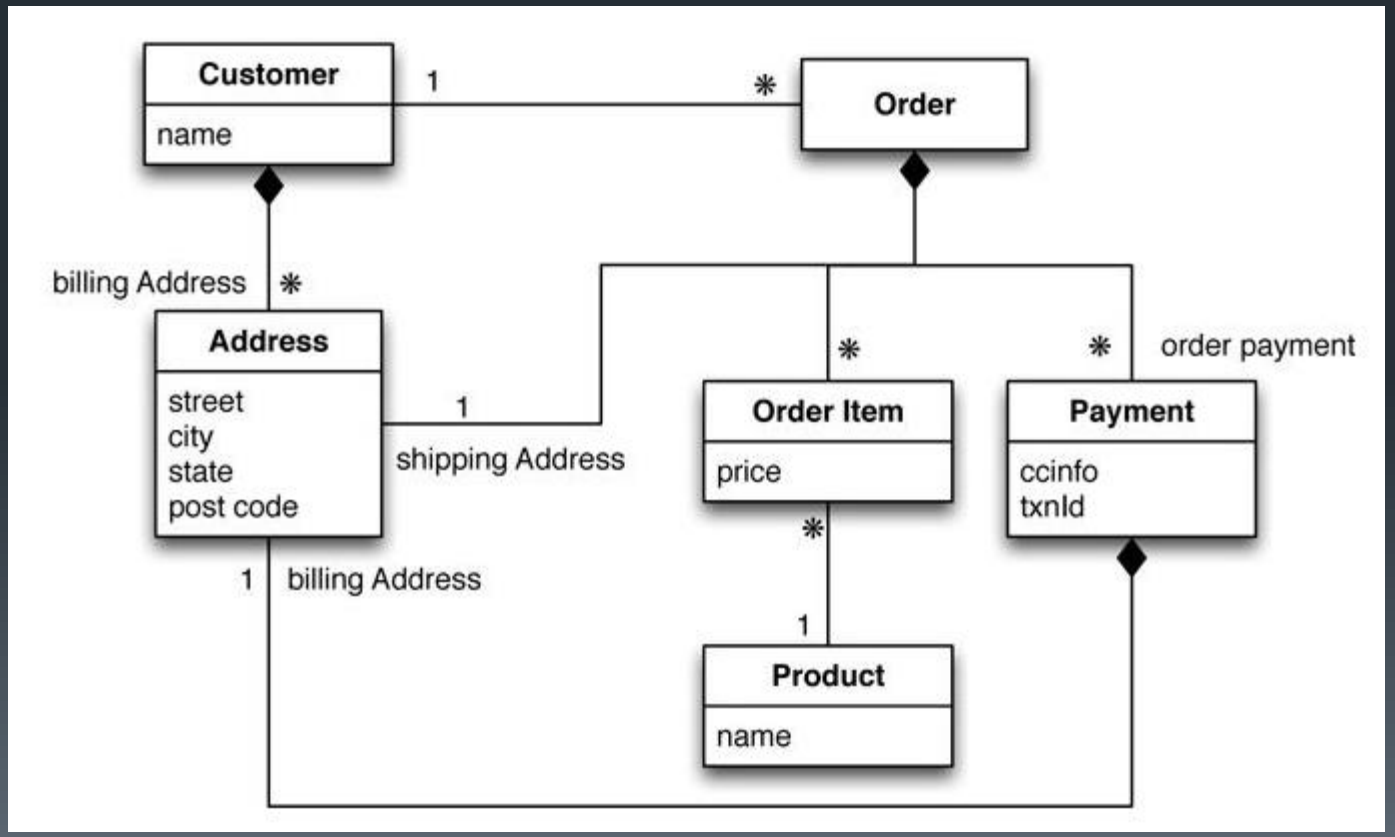
There are two primary reasons for considering NoSQL

- To handle data access with sizes and performance that demand a cluster
- To improve the productivity of application development by using a more convenient data interaction style

--Fowler 2012

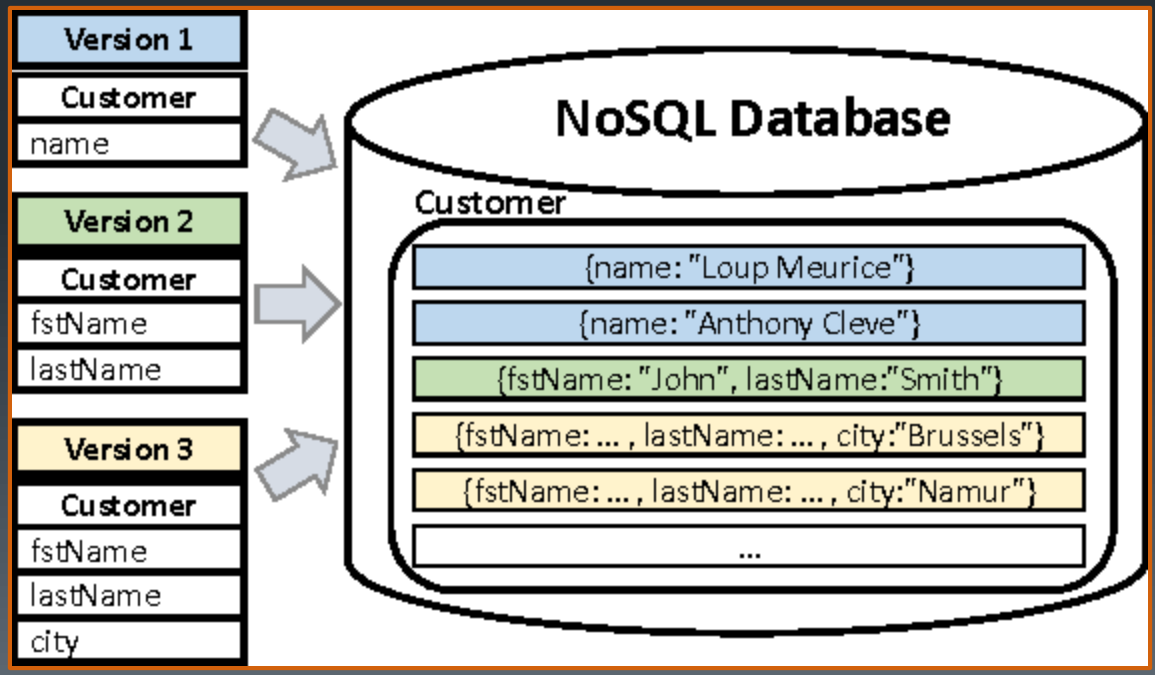
Aggregates

- In Domain-Driven Design [Evans], an aggregate is a collection of related objects that we wish to treat as a unit
- Choosing aggregates is an important part of the design process in distributed applications
- Where are the aggregate boundaries?
- Which datum will be repeated if any?
- How can aggregates be structured to minimize server round trips?



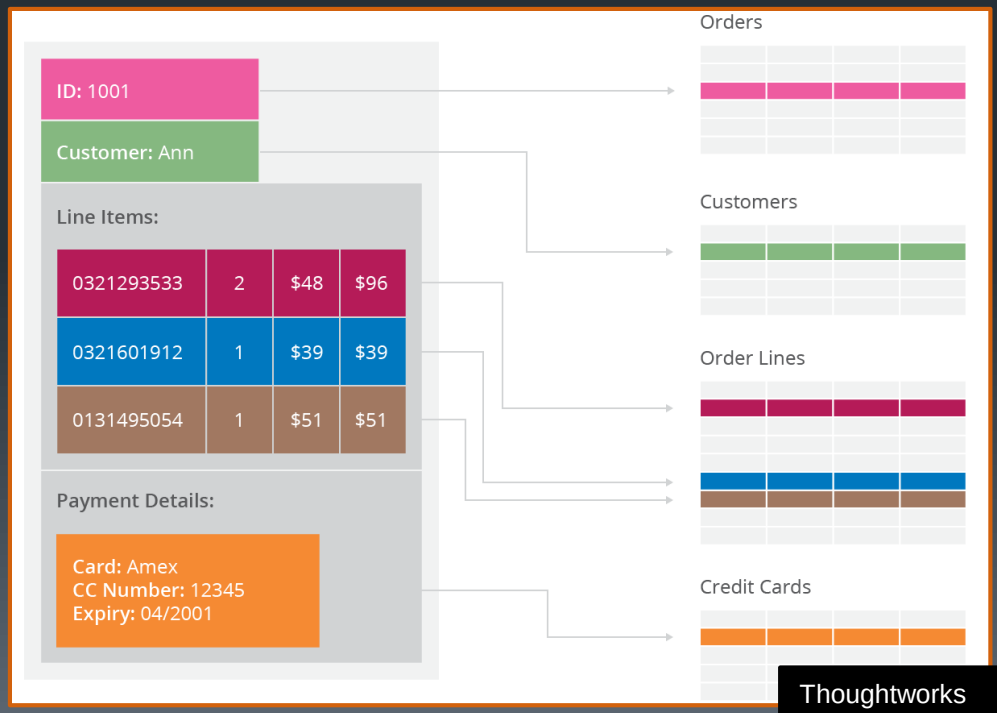
Schemaless Systems

- NoSQL databases often have no explicit schema
- Programs that accesses data in such a system use an implicit schema
  - Schemaless databases shift the schema into the application code that accesses it
- If multiple applications access the same database the implicit schema must be coordinated
  - Error prone and difficult to scale/maintain
  - Microservice systems encapsulate all database interaction for a given aggregate within a single service
- Schemaless data stores allow data to evolve over time



NoSQL Transactions

- ACID transactions offer the pinnacle of consistency
- Most NoSQL database offer no transaction support
  - Graph databases are the exception
- However, transaction-less NoSQL systems offer atomic aggregate updates
- If aggregates equate to joined rows in SQL similar transaction semantics can be achieved



Consistency

- Highly distributed clusters introduce new consistency concerns as compared to Relational ACID type systems
  - **write-write conflict**: two people updating the same data item at the same time
    - Can produce **Lost Updates**
    - We both increment 6 and instead of going to 8 it goes to 7
  - **sequential consistency:** ensuring that all nodes apply operations in the same order
- Systems can take a pessimistic or optimistic approach to consistency
  - pessimistic prevents conflicts from occurring
  - optimistic lets conflicts occur, but detects them and takes action to sort them out (hopefully?)
- Conditional updates test the value just before updating it to see if it's changed since his last read
- Some systems save all write-write conflict updates allowing users (or application code) to resolve the conflict
- Any update that affects multiple aggregates leaves an inconsistency window
  - Amazon SimpleDB reports an inconsistency windows usually less than 1 second
- replication consistency is the process of ensuring that the same data item has the same value when read from different replicas
  - Quorums
  - Eventual Consistency
- read-your-writes consistency means that, once you've made an update, you're guaranteed to continue seeing that update
- session consistency a user's session ensures read-your-writes consistency

Sticky Session

- A session tied to one node (aka. session affinity)
- ensures that as long as you keep read-your-writes consistency on a node, you'll get it for sessions too
- sticky sessions reduce the ability of the load balancer to do its job

Quorums

- write quorum
  - W > N/2
  - The number of nodes participating in the write (W) must be more than the half the number of nodes involved in replication (N)
  - The number of replicas is often called the replication factor
- read quorum
  - How many nodes you need to contact to be sure you have the most up-to-date change
  - You can have a strongly consistent read if R + W > N
- These inequalities are written with a peer-to-peer distribution model in mind
- Most suggest a replication factor of 3
  - This allows a single node to fail while still maintaining quora for reads and writes
  - With automatic rebalancing the cluster will create a third replica quickly
- The number of nodes participating in an operation vary with the operation
  - When writing a quorum may be required for some types of updates but not others, depending on the value of consistency and availability

Key Value Model

- Each aggregate has a key or ID that's used to get the data
- The aggregate is opaque to the database
  - Usually seen as a blob
  - Aggregates can be anything
  - Usually size limited
- Some dbs support meta data to define relationships, expiration times, etc.
- These solutions are very simple, and thus tend to be very fast
- Sharding is based on key hash

Use:
- Storing Session Info
- User profiles/preferences
- Shopping carts

Don't Use:
- Data Relationships
- Multioperation Transactions
- Query by Data
- Operations on sets

Memcached

- A general-purpose distributed memory K/V caching system
- Used to speed up dynamic database/API driven websites
- Runs on Unix, Linux, Windows and Mac OS X
- Keys are up to 250 bytes long and values can be at most 1 megabyte
- The client library knows all servers
  - Servers do not communicate with each other
  - Clients read/set values by computing a hash of the key to determine the server to use
- When the table is full, subsequent inserts cause older data to be purged in least recently used (LRU) order
- Applications using Memcached typically fall back to a database request on cache miss
- Originally developed by Danga Interactive for LiveJournal, now used by YouTube, Reddit, Zynga, Facebook, Twitter, Tumblr, Wikipedia, etc.
- Engine Yard and Jelastic are using Memcached as the part of their platform as a service technology stack
- Heroku offers a managed Memcached service built on Couchbase Server as part of their platform as a service
- Google App Engine, AppScale, Windows Azure and Amazon Web Services also offer Memcached

```
function get_foo(int userid) {
   /* first try the cache */
   data = memcached_fetch("userrow:" + userid);
   if (!data) {
      /* not found : request database */
      data = db_select("SELECT * FROM users WHERE userid = ?", userid);
      /* then store in cache until next get */
      memcached_add("userrow:" + userid, data);
   }
   return data;
}
```

Document Model

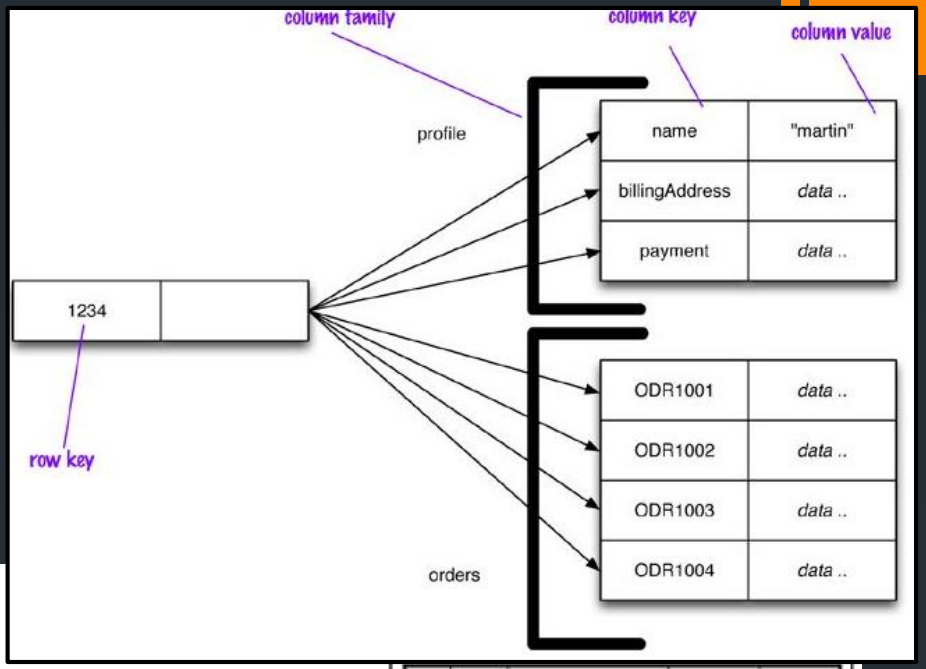
- Each aggregate has a key or ID that's used to get the data
- The aggregate is visible to the database
  - Usually a JSON or XML document
  - Aggregates can have any structure support by the document type
- Document dbs support limited searches on document data
- These solutions are simple, fast but supply useful aggregate search features
  - You can look up data by something other than aggregate key
- Sharding is based on key hash
- In practice the distinction between KeyValue and Document dbs blurs heavily in some implementations

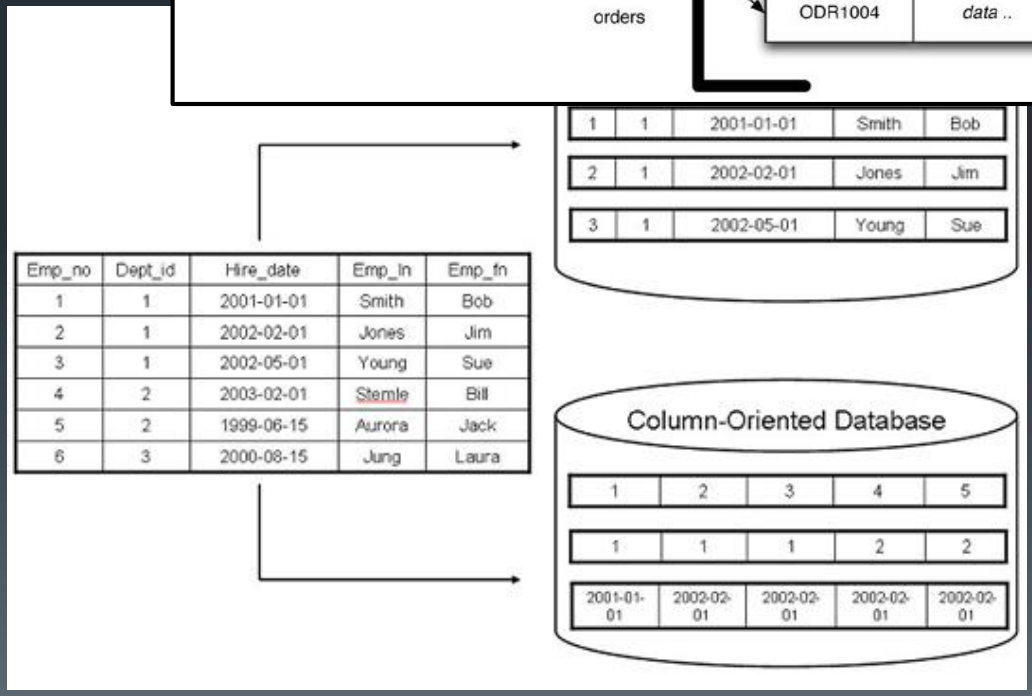
Use:
- Event Logging
- CMS/Blogging
- Web Analytics
- E-Commerce

Don't Use:
- Complex transactions
- Queries against varying aggregate structure

Column

- Google documented BigTable in a famous white paper [Chang etc.]
  - Their core data store at the time (previously used for search)
  - Ues sparse columns and no schema
  - A two-level map
  - Influenced HBase and Cassandra
- Column Families
  - Stores groups of columns together
  - Each column has to be part of a single column family
  - Columns are the unit for access
  - Column stores make summing columns very fast due to column layout on disc versus row layout
  - Column families can be treated like tables
- Columns can be added freely
- Skinny Rows
  - Few columns
  - Same columns used across the many rows
  - The column family defines a record type
  - Each row is a record, and each column is a field
- Wide Rows
  - Have many columns (perhaps thousands), with rows having very different columns
  - A wide column family models a list, with each column being one element in that list
  - A consequence of wide column families is that a column family may define a sort order for its columns





Use:
- Event Logging
- CMS/Blogging
- Counters
- Expiring usage (columns)

Don't Use:
- Situations where the column families change frequently

Cassandra Case Study

- Invented at Facebook by the author of Amazon's DynamoDB, Apache Cassandra is an open source distributed database management system designed to handle large amounts of data across many commodity servers
  - Cassandra's distributed architecture is specifically tailored for multiple-data center deployment, for redundancy, for failover and disaster recovery
- Cassandra is implemented as a DHT producing high availability with no single point of failure
- Cassandra implements asynchronous masterless replication
- University of Toronto researchers studying NoSQL systems report that Cassandra achieved the highest throughput for the maximum number of nodes in all of their experiments
- Based on the Chord DHT and uses an MD5 128 bit hash key
- Cassandra's data model is a partitioned row store with tunable consistency
  - Partition keys can be hashed randomly or in order for systems benefitting from partition key locality
  - Rows are organized into tables
  - The first component of a table's primary key is the partition key
  - Within a partition, rows are clustered by the remaining columns of the key
  - Other columns may be indexed separately from the primary key
  - Tables may be created, dropped, and altered at runtime without blocking updates and queries
- Cassandra does not support joins or subqueries, except for batch analysis via Hadoop
- Cassandra emphasizes denormalization through features like collections
- Memcached competitive performance without a two level architecture

Facebook statistics for a >50TB, 150 node cluster

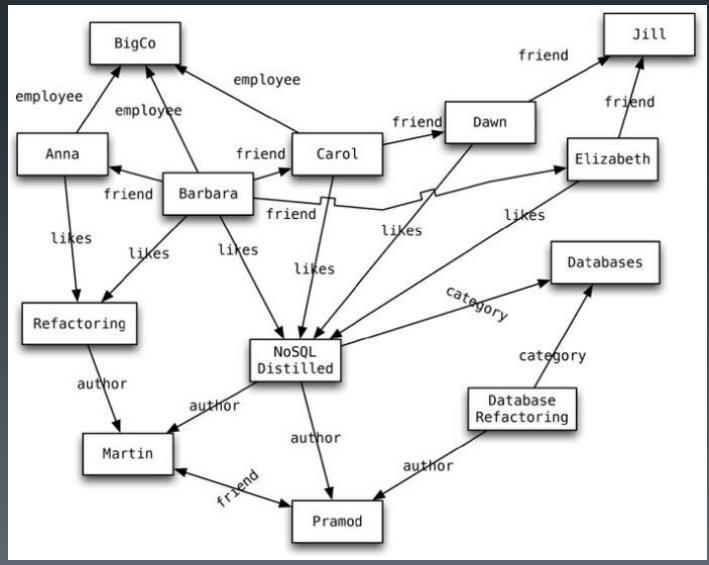
Latency Stat	Search Interactions	Term Search
Min	7.69ms	7.78ms
Median	15.69ms	18.27ms
Max	26.13ms	44.41ms

Graph

- Graph databases are motivated by a different frustration with relational databases and thus have an opposite model
  - small records with complex interconnections
- Nodes & Edges (aka. Arcs)
- Queries like:
  - Select books in the Databases category written by someone whom a friend of mine likes
- Edge traversal is cheap
  - Graph databases shift most of the work of navigating relationships from query time to insert time
  - Good only if querying performance is more important than insert speed
- Usually single server based
- Examples
  - FlockDB - nodes and edges with no mechanism for additional attributes
  - Neo4J - attach Java objects as properties to nodes and edges
  - Infinite Graph - stores Java objects, which are subclasses of its built-in types, as nodes and edges

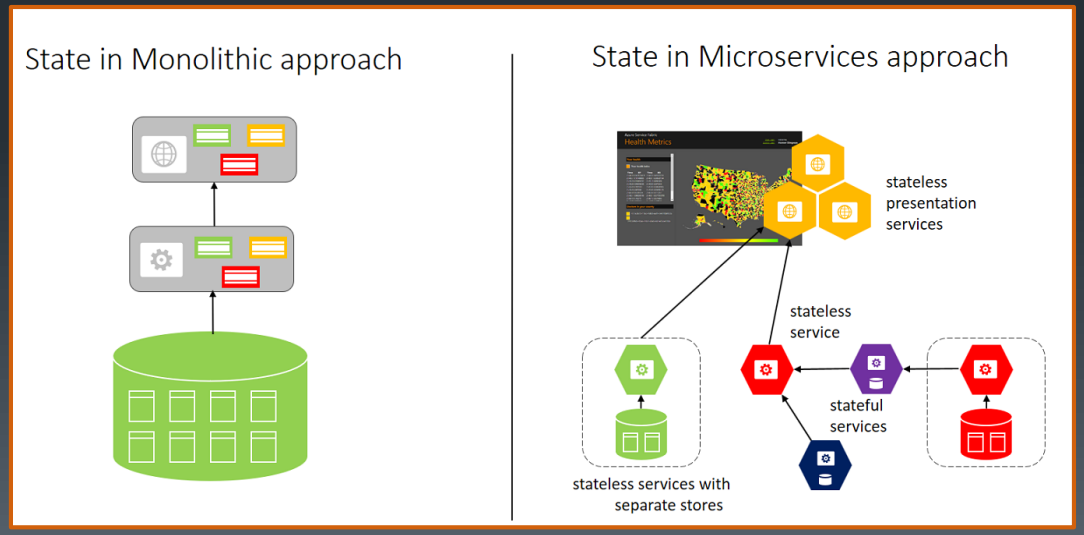
Use:
- Connected Data (social graphs)
- Routing/Dispatch (location based systems
- Recommendation Engines

Don't Use:
- Situations where multiple nodes must be updated



Cloud native state management, patterns and practices

- Cloud native applications divided microservices into 2 camps
  - Stateless
  - Stateful
- Stateful services use volumes to store durable data
- State managers are typically cloud native cluster aware systems
  - Redis, MongoDB, Cassandra, etc.
- State managers are owned by a single service and live inside the bounded context of that service



Summary

- A range of state managers are available for use in microservice based systems
  - Relational
  - Graph
  - Key/Value
  - Document
  - Column
- Schemas are critical components of most data stores and should be documented
- Aggregates describe the elements of storage in NoSQL systems
- A range of consistency tuning features are available with different storage solutions

Lab 6

- Building a stateful service

Day 3 – Microservice in Production

- Orchestration
- Serverless
- Gateways and Meshes

7: Orchestration

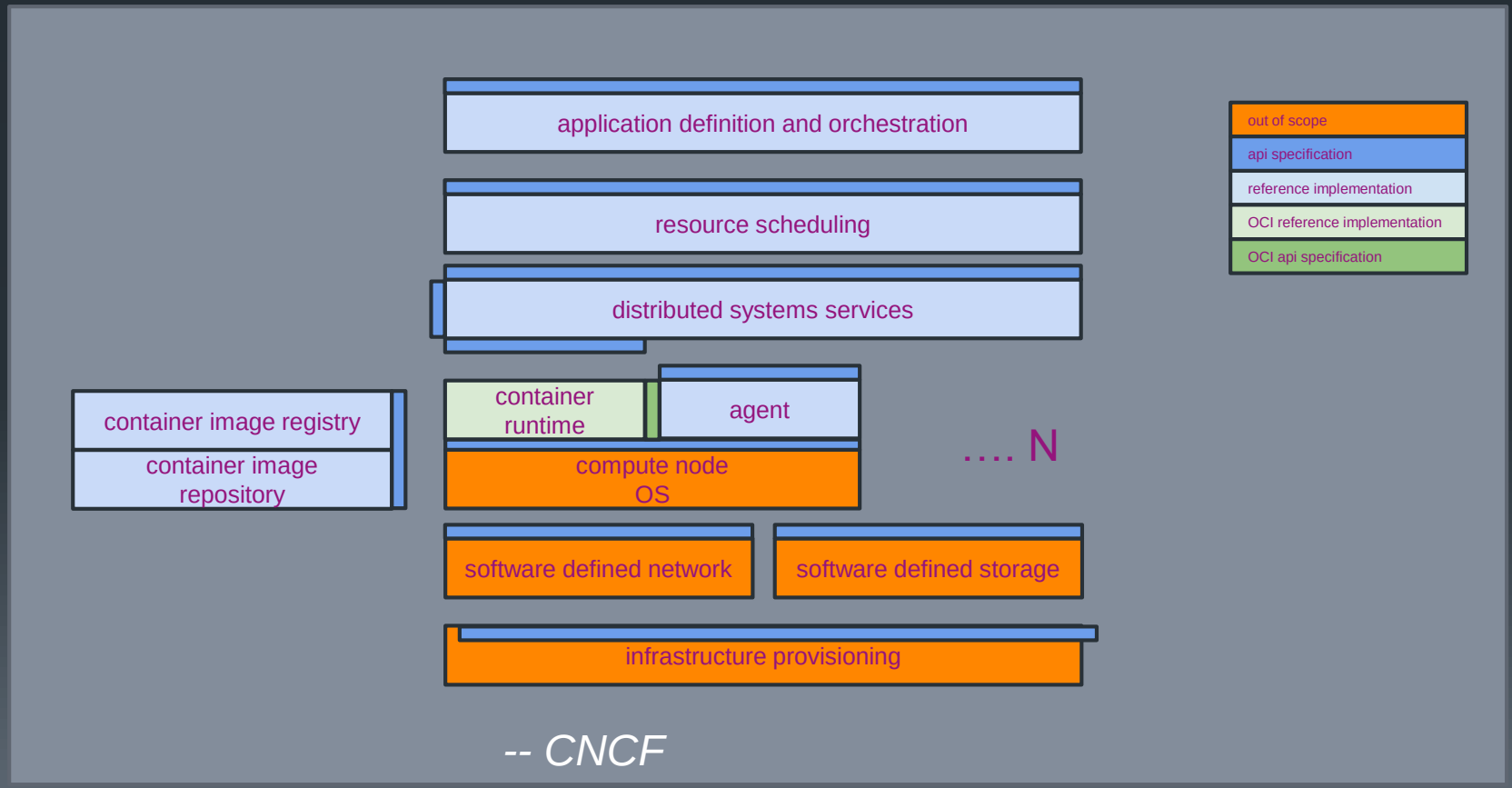
Objectives

- Define Orchestration
- Examine the CNCF cloud native platform reference model working draft
- Describe Mesos, Kubernetes, Docker Swarm and ECS
- Explain the role of the service
- Explore load balancing and service discovery

After Containers, what's left?

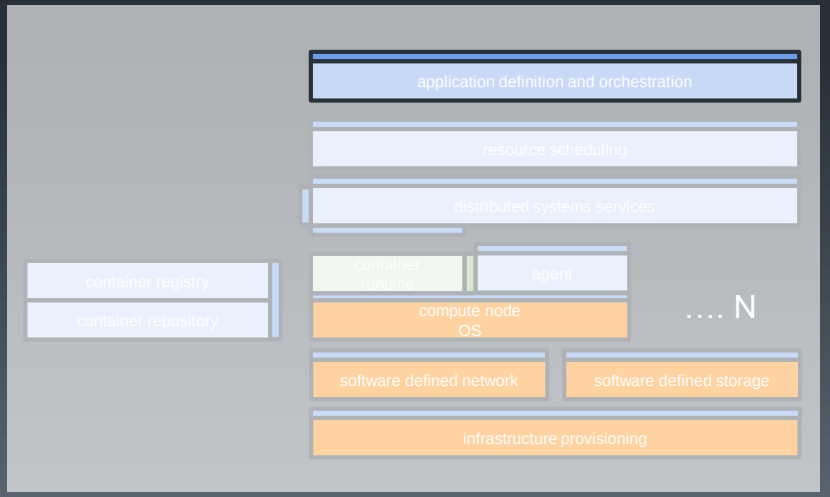
- Orchestration!
- Real cloud native applications are delivered in 10s of images and require 100s - 1000s of running containers
- Registries manage image distribution
- Orchestration manages:
  - Container Scheduling
    - Distributing containers to appropriate hosts
    - Host resource leveling
    - Availability zone diversity
    - Related container packaging and co-deployment
  - Container Management
    - Monitoring and recovery
    - Image upgrade rollout
    - Scaling
    - Logging
  - Service Endpoints
    - Discovery
    - HA
    - Load balancers
    - Auto scaling
  - External Services
    - Network configuration and management
    - Durable volume management

Cloud Native Parts



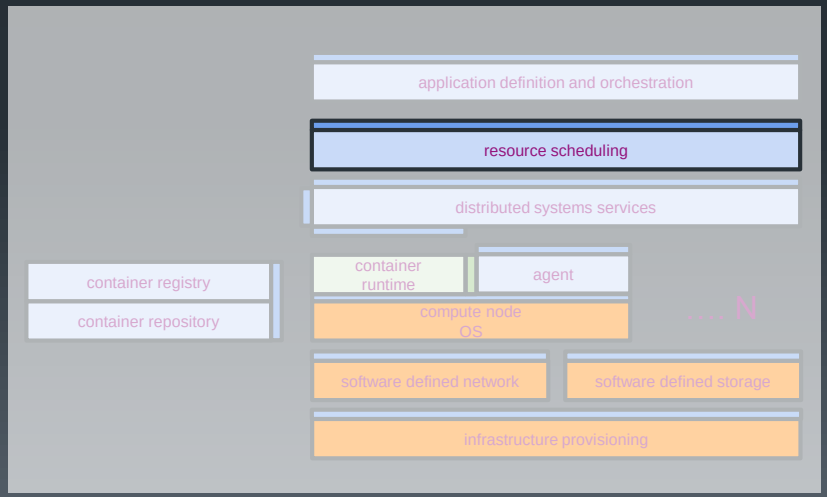
Application definition and orchestration

- standard service definition
  - define a standard distributed system service that can be deployed
- composite application definition
  - a standard model for packaging and shipping an app of many parts
- orchestrator
  - a standard framework to deploy and manage an app of many parts
  - providing workload/job specific orchestration and control



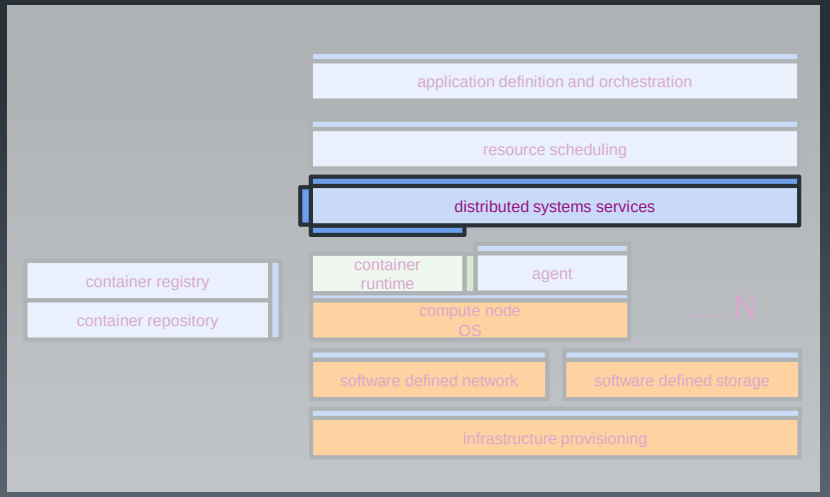
Resource scheduling

- map containers to nodes
- actively manage the health of a specific container
- handle active scaling of containers
- interface with infrastructure provisioning to request more resources
- interface with SDN and SDS to map to storage/network



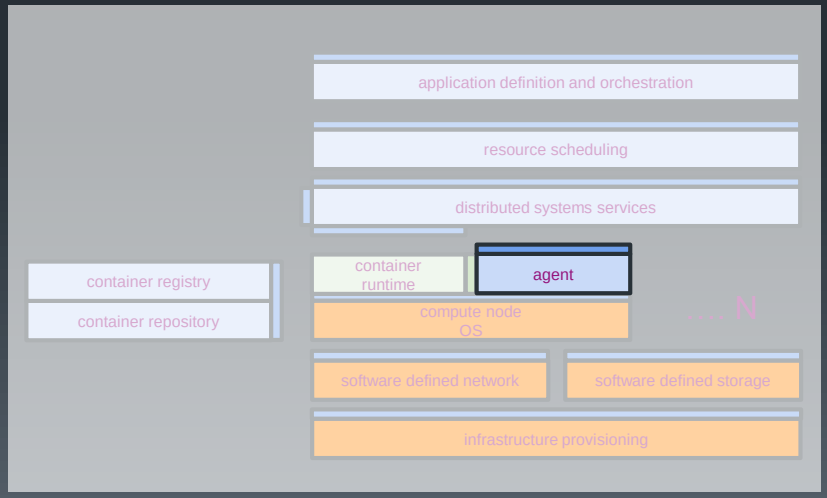
Distributed systems services

- a standard set of services that are not bound to a single node
  - supporting application use cases
    - naming/discovery
    - locking/quorum
    - state management/sharding
    - logging/monitoring
  - supporting cluster use cases
    - distributed state management
    - distributed control plane
    - logging/auditing
- a minimum atom of consumption for software
  - within the cluster
    - between clusters
    - from outside the cluster



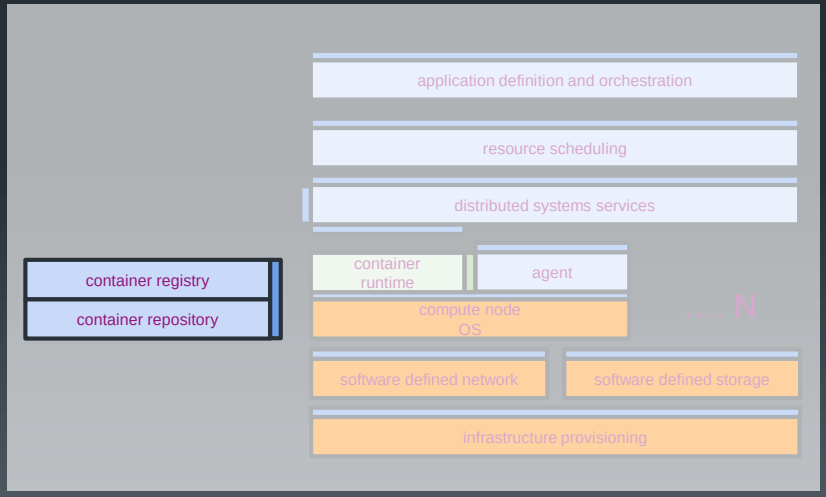
Local agent

- maintains the lifecycle of containers on a node
- monitors the health of container on a node



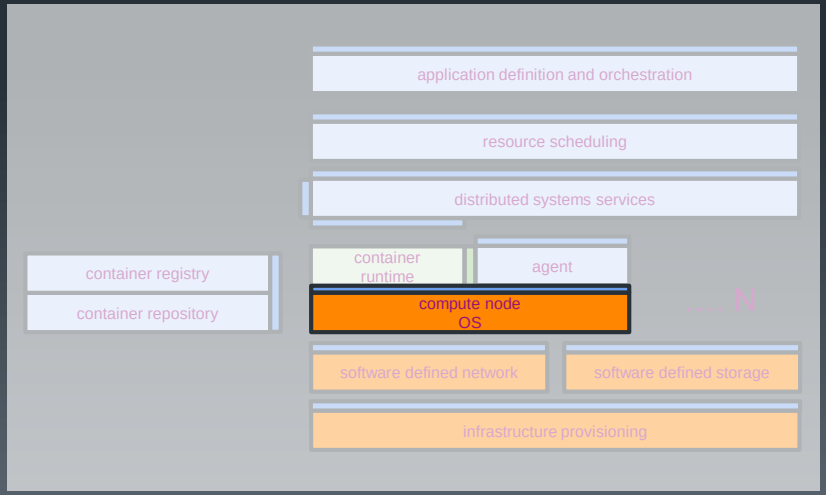
Container registry and repository

- a standard way to find a container
- a standard place to start and distribute containers



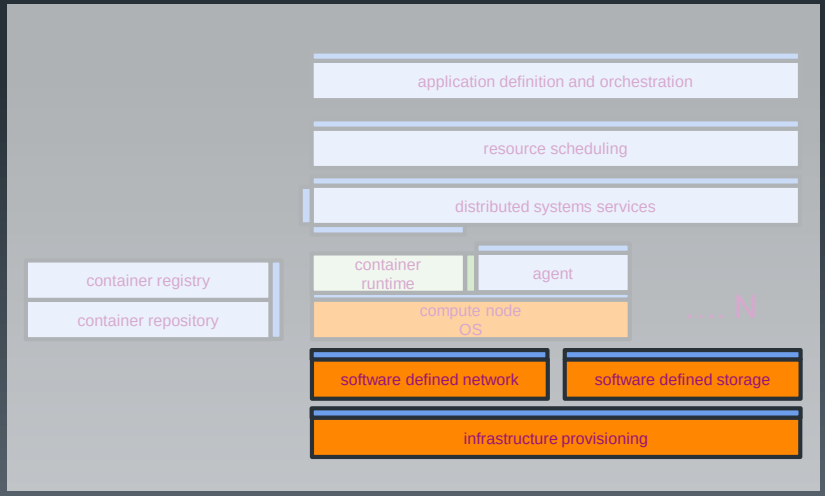
Compute node

- a standard definition for what the minimum set of services on a compute node are
- the actual node distribution is out of scope

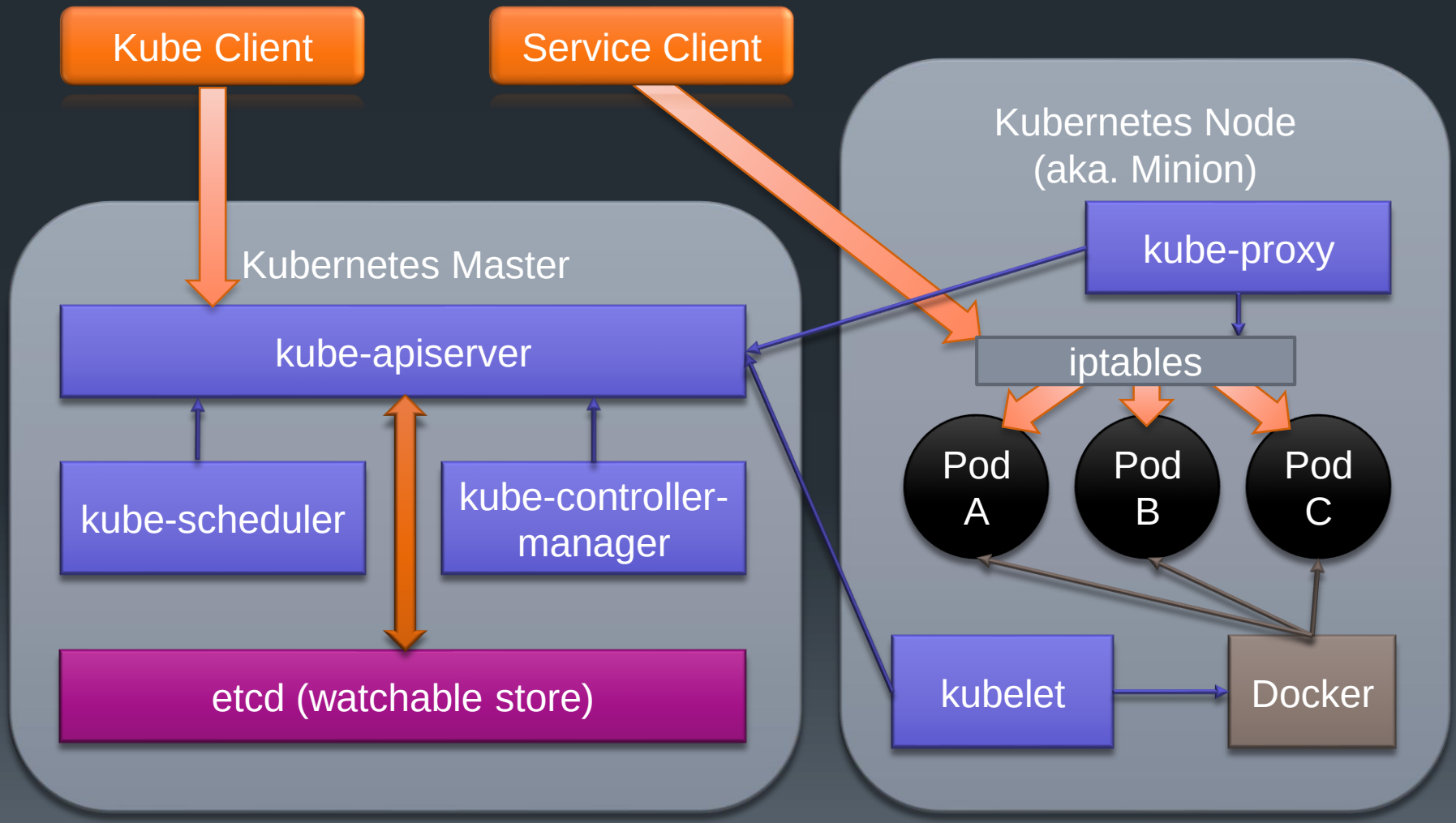


Infrastructure provisioning and integration

- infrastructure provisioning
  - provide a standard interface and plugin model to request additional infrastructure
- software defined datacenter integration
  - provide a standard interface and plugin model for network and storage integration with clusters

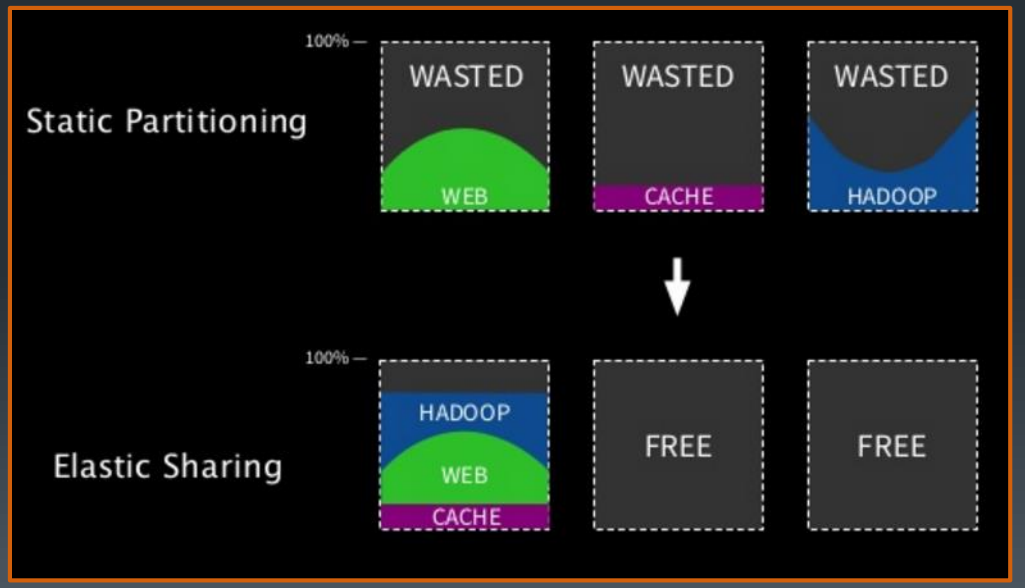


Kubernetes Architecture



Apache Mesos

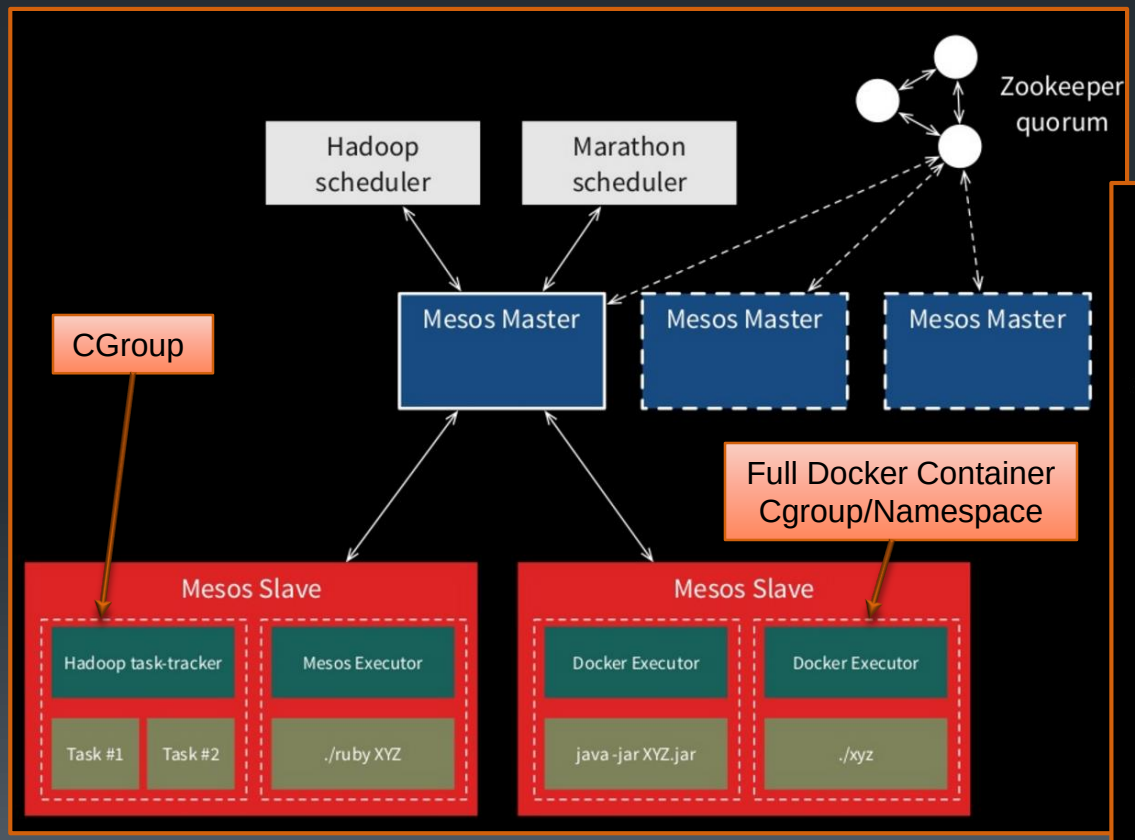
- Apache Mesos lets you program against your datacenter like it's a single pool of resources
- Apache Mesos abstracts CPU, memory, storage, and other compute resources away from machines (physical or virtual)
- Enables fault-tolerant and elastic distributed systems to easily be built and run effectively
- Allows for the sharing a single cluster of commodity computer hardware between many different applications
  - Initially MPI and Hadoop
  - Later Spark, Cassandra, Jenkins and others
- Solves the reprovisioning/static partitioning problem with clusters
- Developed at the UC Berkeley AMP (Algorithms Machine People) Lab
  - Twitter hires Ben Hindman in 2010, the lead author and architect Mesos
    - Over several years, Mesos grows from a research project into the core infrastructure powering tens of thousands of servers at Twitter and many other companies
    - Produces Apache Aurora
  - Ben Hindman and Florian Leibert from Twitter start Mesosphere in 2013
    - Producer of Marathon

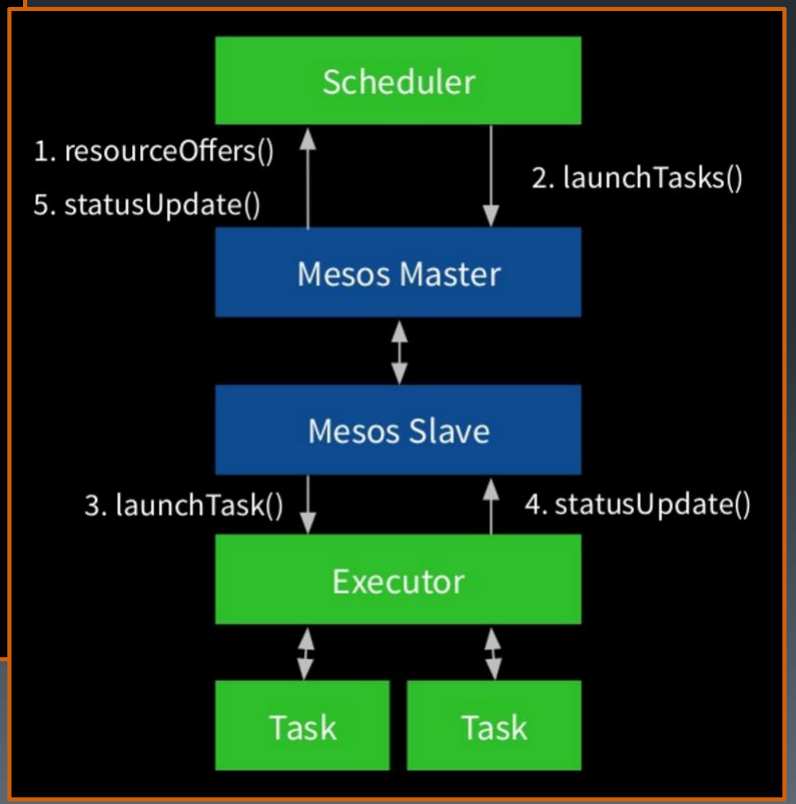


-- Florian Leibert

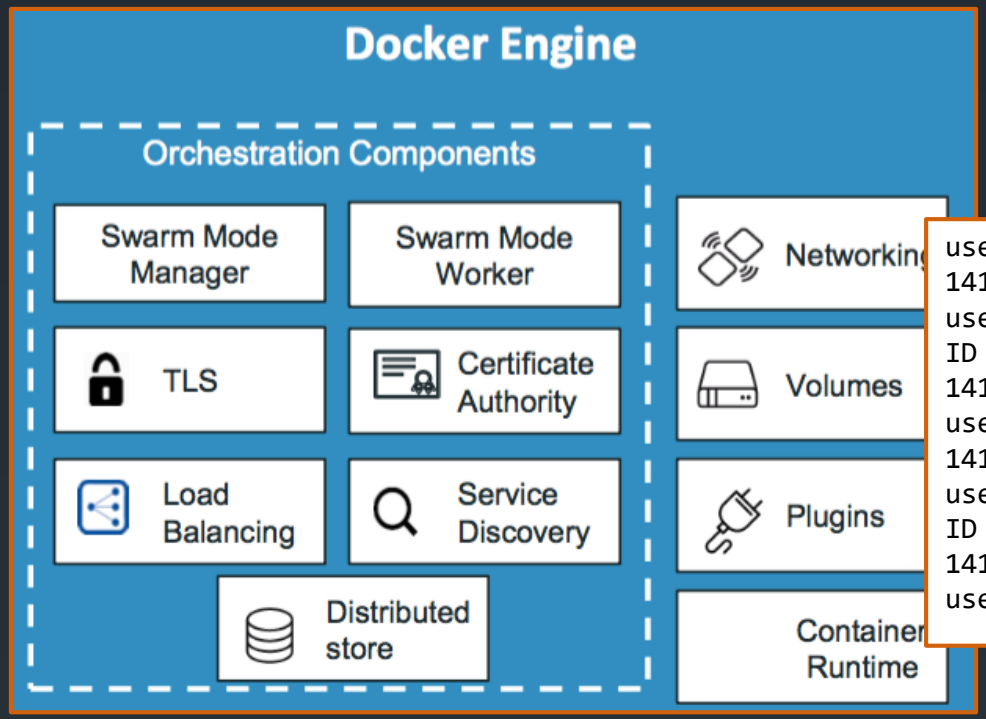
Mesos Architecture

- A Mesos cluster has two components:
  - Masters - coordinates the cluster
  - Slaves - execute code in containers
- Mirrors the organization of frameworks
  - master/slave pattern is simple, flexible, and effective
  - The cluster software is however separate to ensure frameworks remain isolated and secure from one another





Docker Swarm

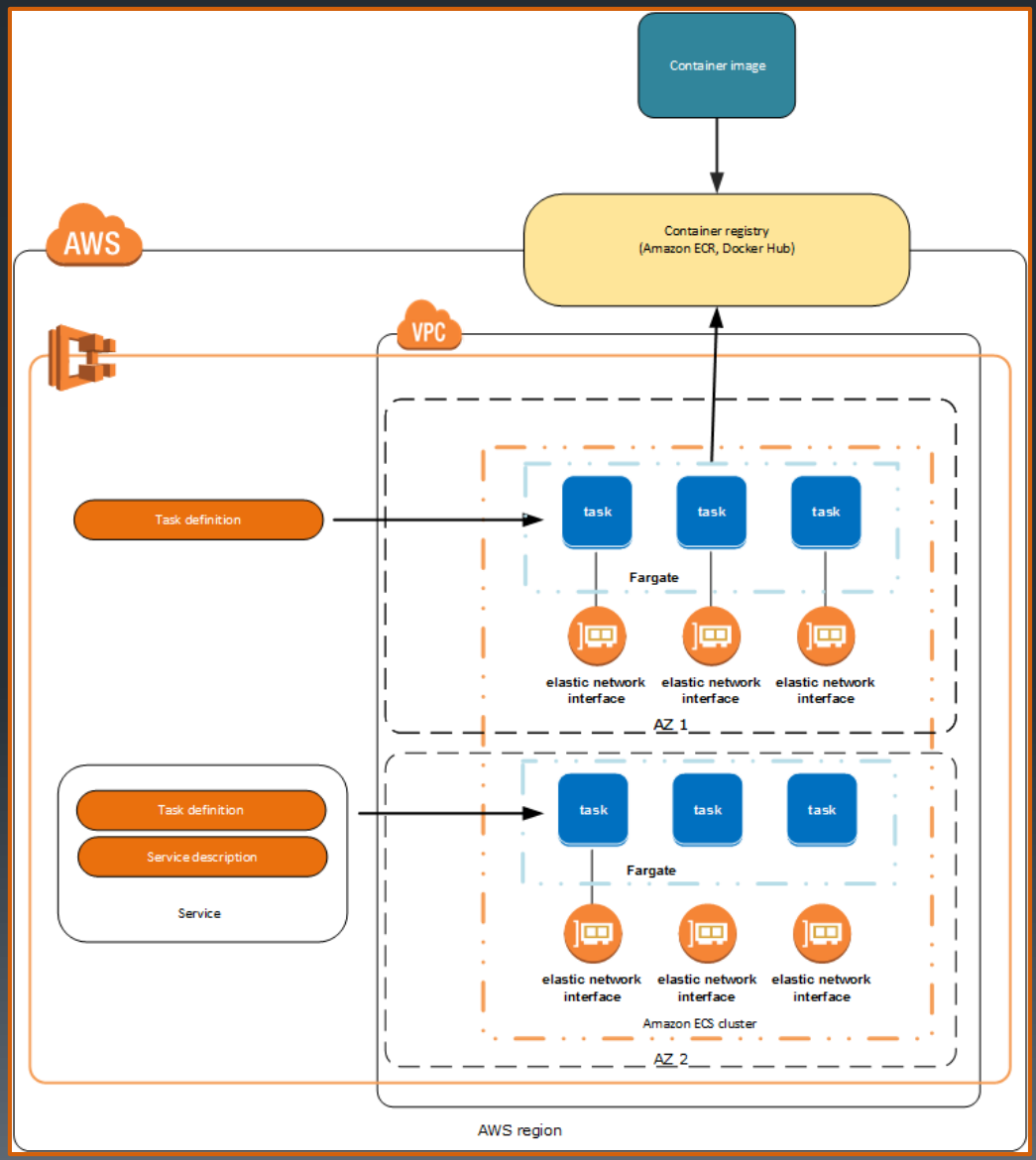



```
user@ubuntu:~$ docker service create nginx
141z2ky40mq8zxv7d7soynfxp
user@ubuntu:~$ docker service ls
ID            NAME             REPLICAS  IMAGE  COMMAND
141z2ky40mq8  awesome_babbage  1/1       nginx
user@ubuntu:~$ docker service scale 141z=2
141z scaled to 2
user@ubuntu:~$ docker service ls
ID            NAME             REPLICAS  IMAGE  COMMAND
141z2ky40mq8  awesome_babbage  2/2       nginx
user@ubuntu:~$
```

```
user@ubuntu:~$ docker swarm init --advertise-addr=10.0.0.220
Swarm initialized: current node (e7n1dpk4axdiyoo2mx4r2tb1r) is now a manager.

To add a worker to this swarm, run the following command:
    docker swarm join \
    --token SWMTKN-1-0wpinu43yhdm5zkwbtoagqvhgpcekuyl4s2bmzeevksf0tx3fi-9k6q6w4osrj4s6000p50l8q1n \
    10.0.0.220:2377

To add a manager to this swarm, run the following command:
    docker swarm join \
    --token SWMTKN-1-0wpinu43yhdm5zkwbtoagqvhgpcekuyl4s2bmzeevksf0tx3fi-0nf4ddcnixz0rbv4ebjl25p3k \
    10.0.0.220:2377
```

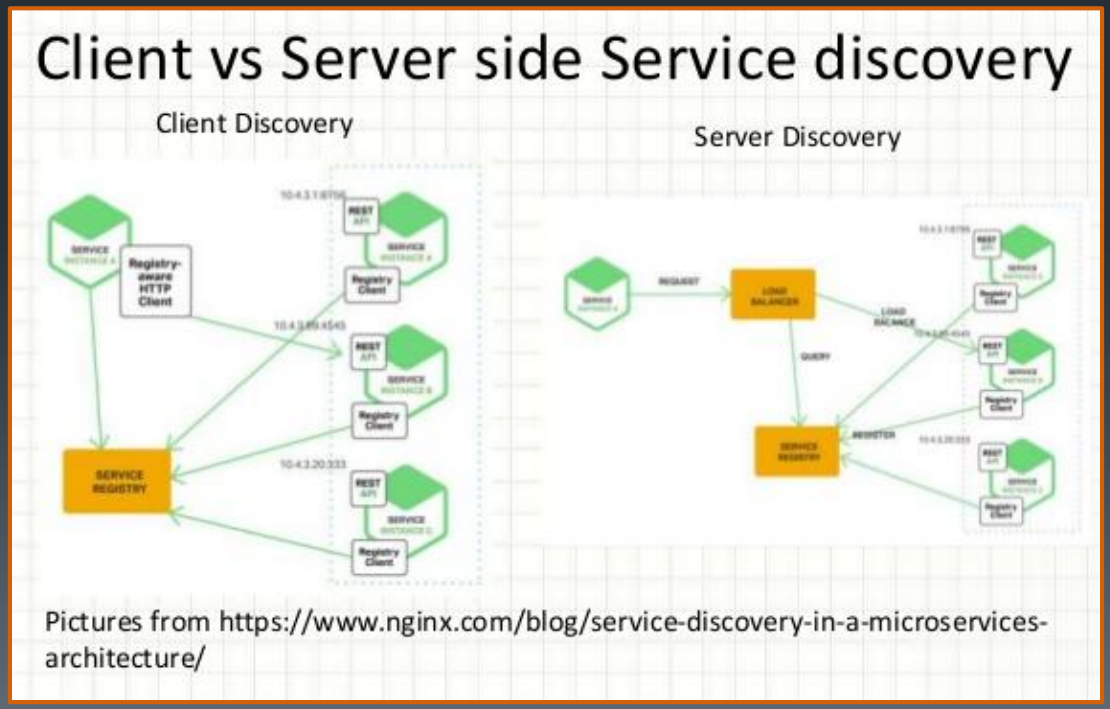
Amazon AWS ECS

- Amazon Elastic Container Service (Amazon ECS)
- Features
  - Scalable
  - High-performance
  - Container orchestration
- Supports Docker/OCI containers
- Eliminates the need for you to install and operate container orchestration software
- Cluster of virtual machines
- API based
- Integrates with AWS:
  - IAM
  - security groups
  - load balancers
  - Amazon CloudWatch Events
  - AWS CloudFormation templates
  - AWS CloudTrail logs



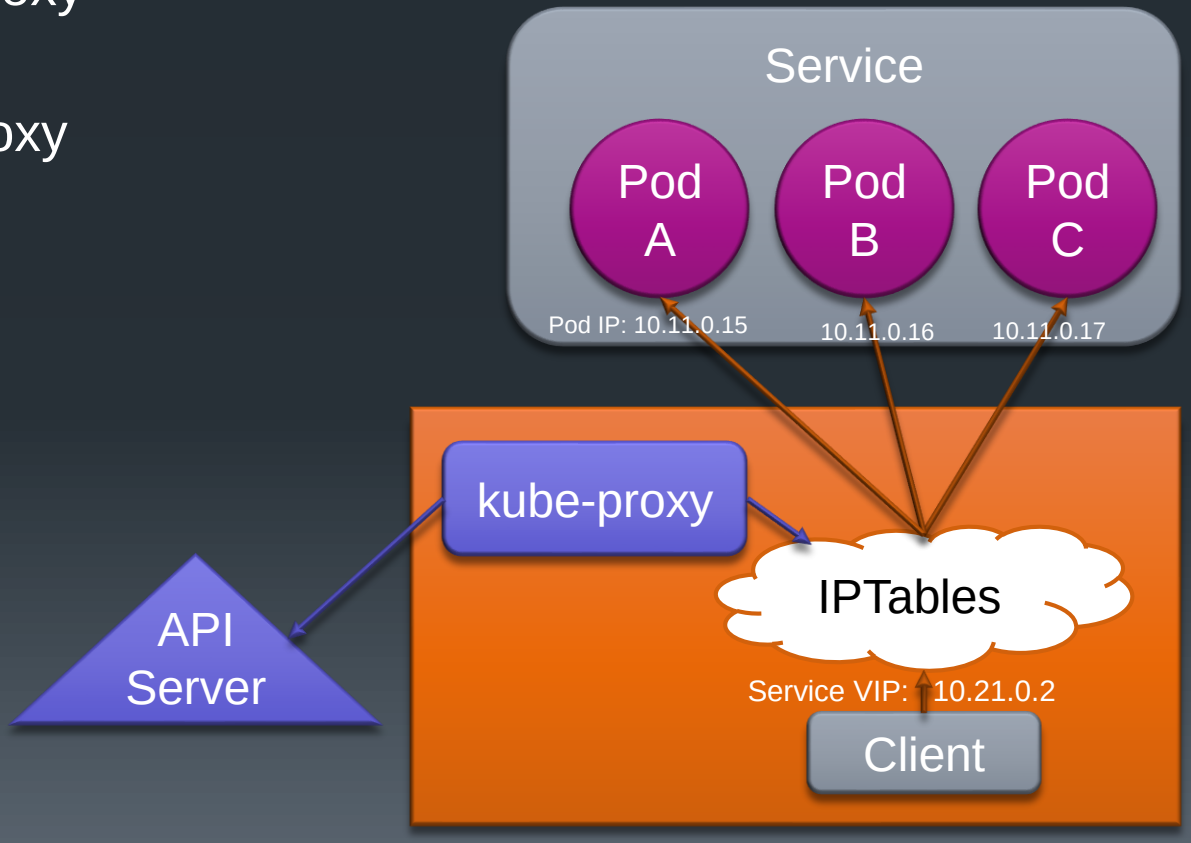
Service discovery schemes

- Service discovery is the process by which a client locates a desired service
- Most common approach involves a client with a hardcoded or operationally supplied hostname using the resolver to acquire an IP address
  - /etc/hosts
  - DNS
- Resolver host records only supply an IP
  - SRV records can be used to acquire port numbers as well
- Dedicated services can also be accessed directly or wired into DNS/resolver systems
  - HashiCorp Consul
  - Apache Zookeeper
  - CoreOS Etcd
  - Netflix Eureka
  - Kubernetes SkyDNS
  - Docker Container Name/Alias/Link
  - Spring Cloud Config



Routing Meshes

- Docker Swarm Mode – integrated with service functionality
  - Uses iptables
- Kubernetes – kube-proxy
  - Uses iptables
- Linkerd – software proxy
  - Uses Finagle dtabs
- Others…



Summary

- Orchestration in cloud native systems is the process of managing the deployment and ongoing operation of containerized microservices
- The CNCF cloud native platform reference model is working toward the creation of a common language which can be used to discussion the facets of a cloud native platform and the integrations points they expose
- There are several popular cloud native orchestration platforms available in the open source world
  - Mesos/Marathon
  - Mesos/Aurora
  - Kubernetes
  - Docker Swarm
- Services define and endpoint for a defined interface
- Load balancing schemes are used to distribute connections to service implementations
- Service discovery is the process used by service client to discover the service end point

Lab 7

- Explore Docker swarm orchestration

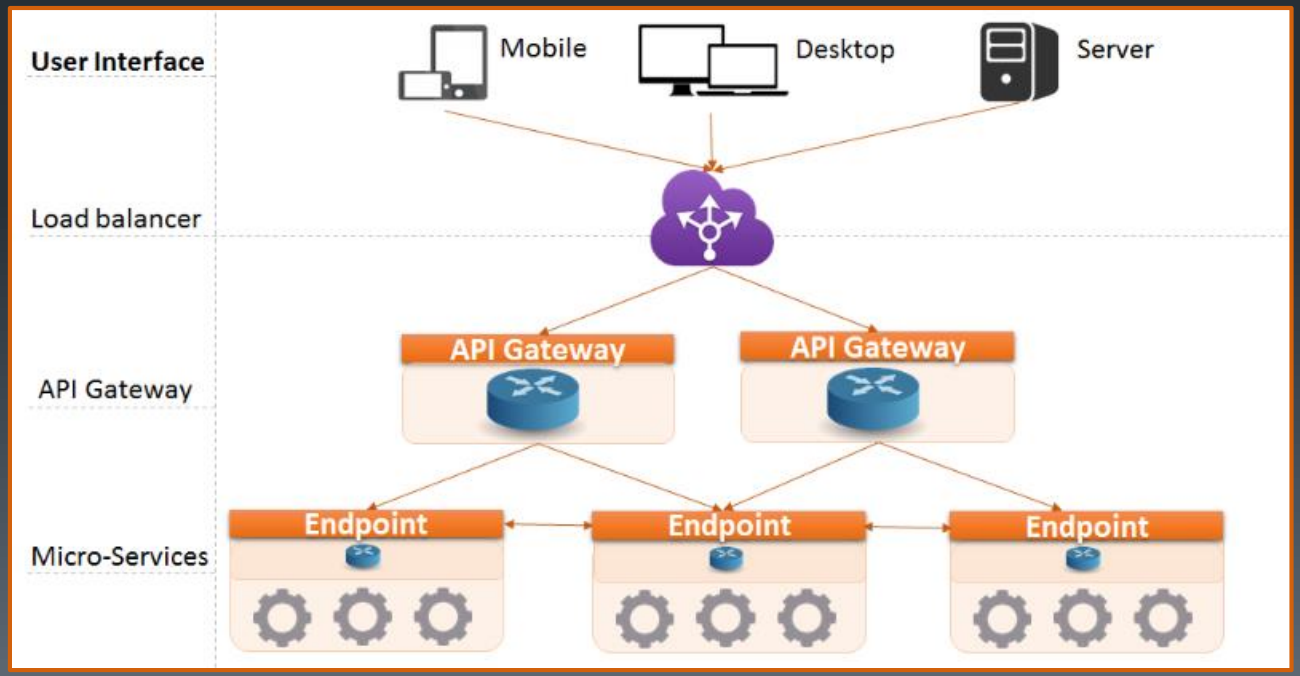
8: Gateways and Meshes

Objectives

- Understand the role of an API gateway
- Explain the importance of bounded context and how API Gateways can enforce isolation
- List key benefits and responsibilities of API Gateways

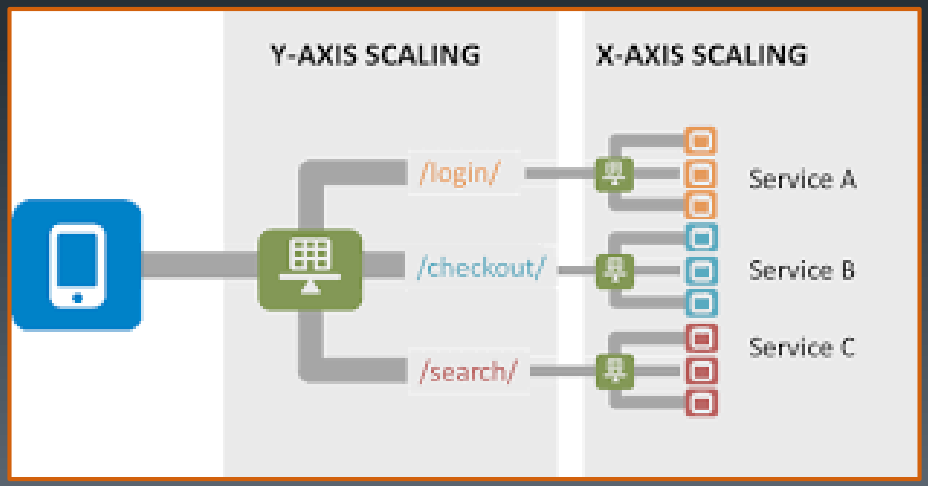
The role of gateways

- Microservice based applications must define explicit means for clients to interact with the application
- With a monolithic application there is just one set of (typically replicated, load-balanced) endpoints
- In a microservices architecture each microservice exposes a set of fine-grained endpoints
- API Gateways expose a perimeter API isolating the microservices within the bounded context of the application
- Client use the API Gateway perimeter API to consume services



Service constructs and end points

- Services are conceptual in cloud native systems
  - Containers implement services
  - Orchestration systems start and stop additional containers for a service as load dictates
  - Load balancing mechanisms distribute clients across the containers available
  - Endpoints define the connection target for clients
    - End points are manifested by real or virtual Load Balancers
      - VIP
      - Per Node Ports
      - Netflix Ribbon
      - AWS Elastic/Application Load Balancer
      - Google Cloud Load Balancer
      - Nginx
      - HA Proxy
      - Traefik
      - Etc.

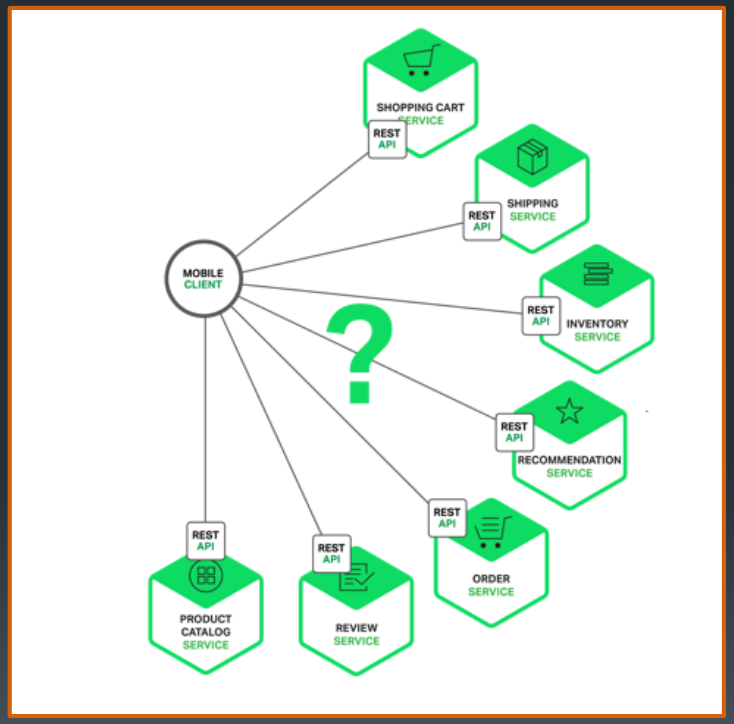


Auto Scalling

- Cluster scaling – infrastructure scaling
  - Adding/removing worker nodes based on cluster utilization
- Application scaling – pod/container scaling
  - Based on CPU utilization, HTTP requests per second, etc.
  - Horizontal Pod Autoscalers (HPAs) – adding/removing pod replicas depending on certain metrics
  - Vertical Pod Autoscalers (VPAs) – increases/decrease resource requirements of containers/pods

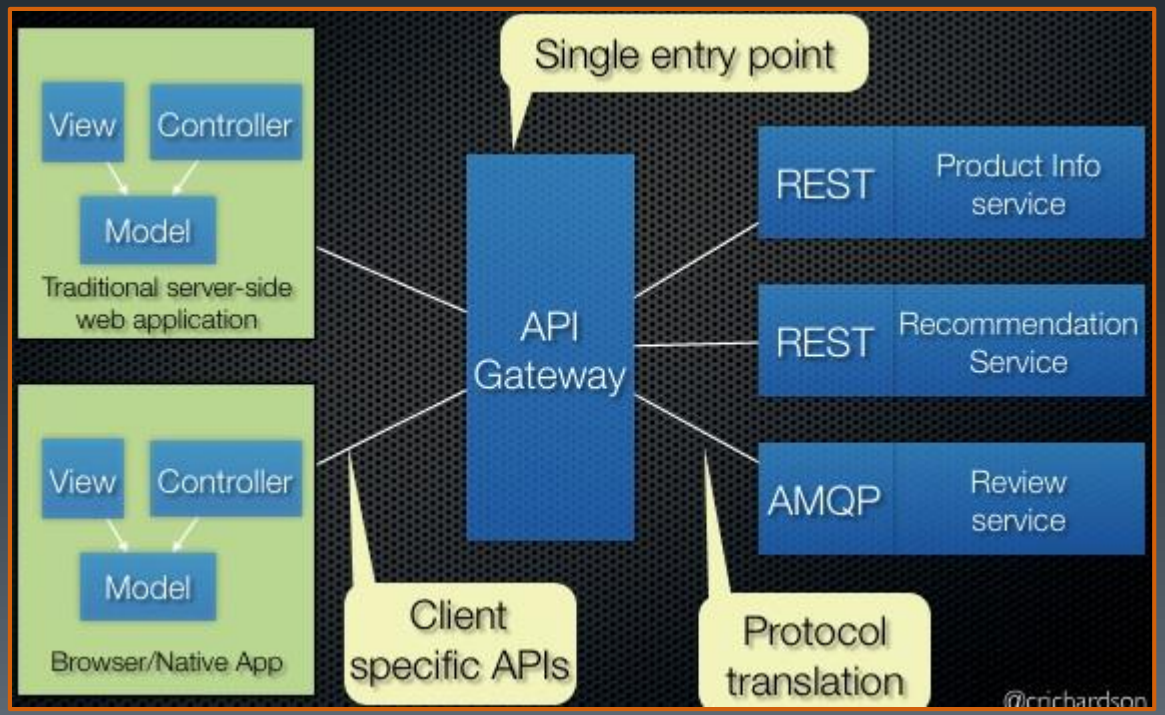
Ingres integration anti-patterns

- Direct Client-to-Microservice Communication
  - A client could make requests to each microservice directly
  - Each microservice would have a public endpoint (https://serviceName.api.company.name) mapped to the microservice's load balancer
- There are challenges and limitations with this option
  - The mismatch between the needs of the client and the fine-grained APIs exposed by each of the microservices
    - The client has to make many separate requests to perform an operation
  - Chatty services in a 10Gb data center may work well, but they almost always produce problems over the slower, less reliable Internet
  - Microservices may use protocols that are not web-friendly (Thrift, AMQP messaging protocol, etc.)
- A huge drawback with this approach is that it makes it difficult to refactor the microservices
  - How the system is partitioned is now the client's concern (!)
  - Information about the implementation of services should never escape the enclosing bounded context
- It rarely makes sense for clients to talk directly to microservices



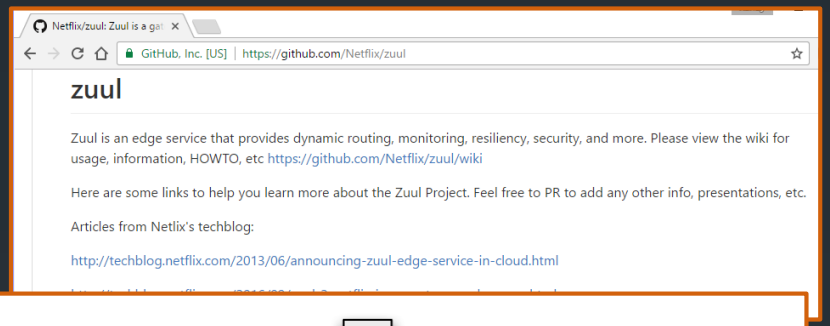
Gateway Features

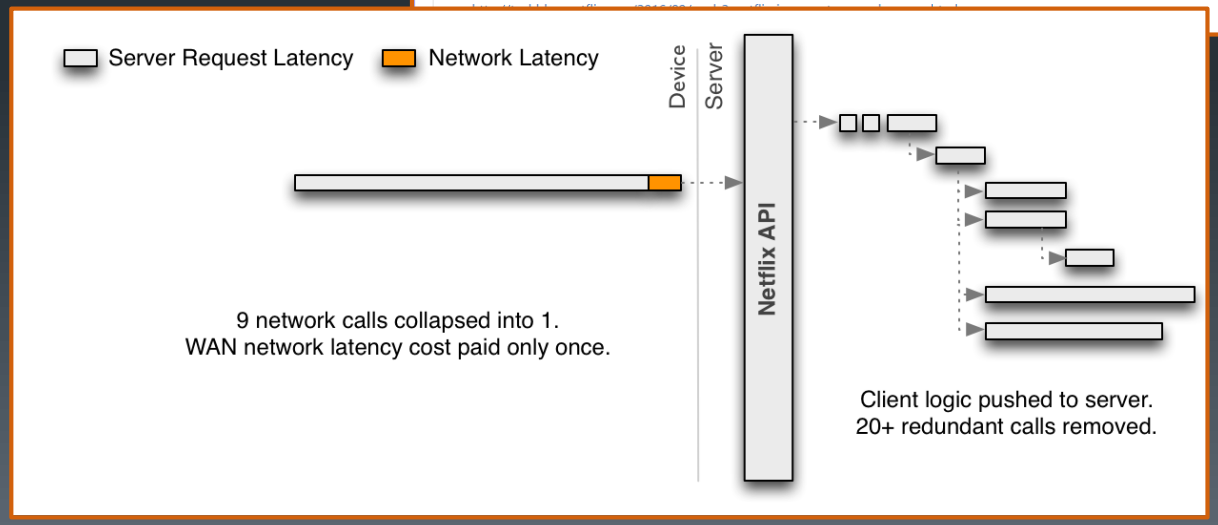
- API Gateways should allow systems to rapidly change behavior in order to react to changing circumstances
- Key functions:
  - Authentication and Security - identifying authentication requirements for each resource and rejecting requests that do not satisfy them
  - Insights and Monitoring - tracking meaningful data and statistics at the edge in order to give an accurate view of production
  - Dynamic Routing - dynamically routing requests to different backend clusters as needed
  - Stress Testing - gradually increasing the traffic to a cluster in order to gauge performance
  - Circuit Breakers - allocating capacity for each type of request and dropping requests that go over the limit
  - Caching - building some responses directly at the edge instead of forwarding them to an internal cluster
  - Multiregion Resiliency – routing requests across regions in order to diversify usage and move the edge closer to clients



Netflix Gateway

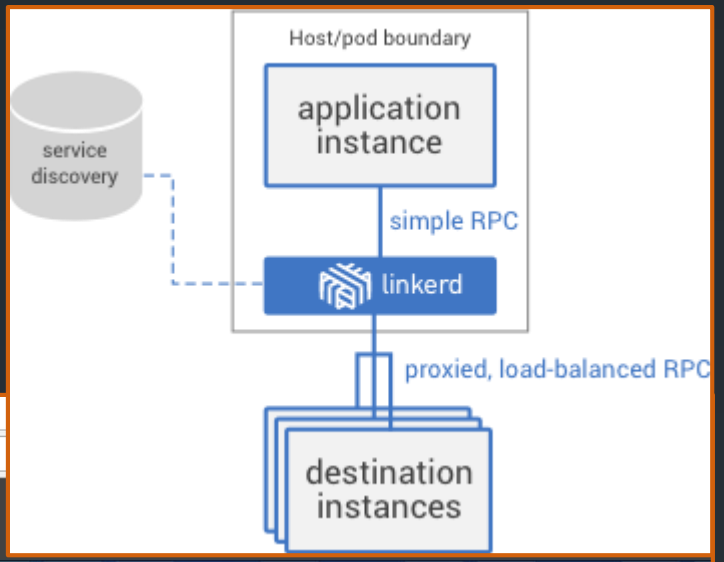
- API Gateways are responsible for:
  - Request routing
  - Composition
  - Protocol translation
- Typically requests from external clients go through an API Gateway
  - Often request invoke multiple microservices requiring data aggregation
- An example API Gateway is the Netflix API Gateway Zuul
  - Netflix streaming service is available on hundreds of different kinds of devices including televisions, set-top boxes, tablets, phones, etc.
    - Handles billions of requests per day
  - Netflix attempted to provide a one-size-fits-all API for their streaming service, discovered it didn't work because device diversity and their unique needs
  - The Zuul API Gateway provides an API tailored for each device by running device-specific adapter code
  - An adapter typically handles each request by invoking on average six to seven backend services

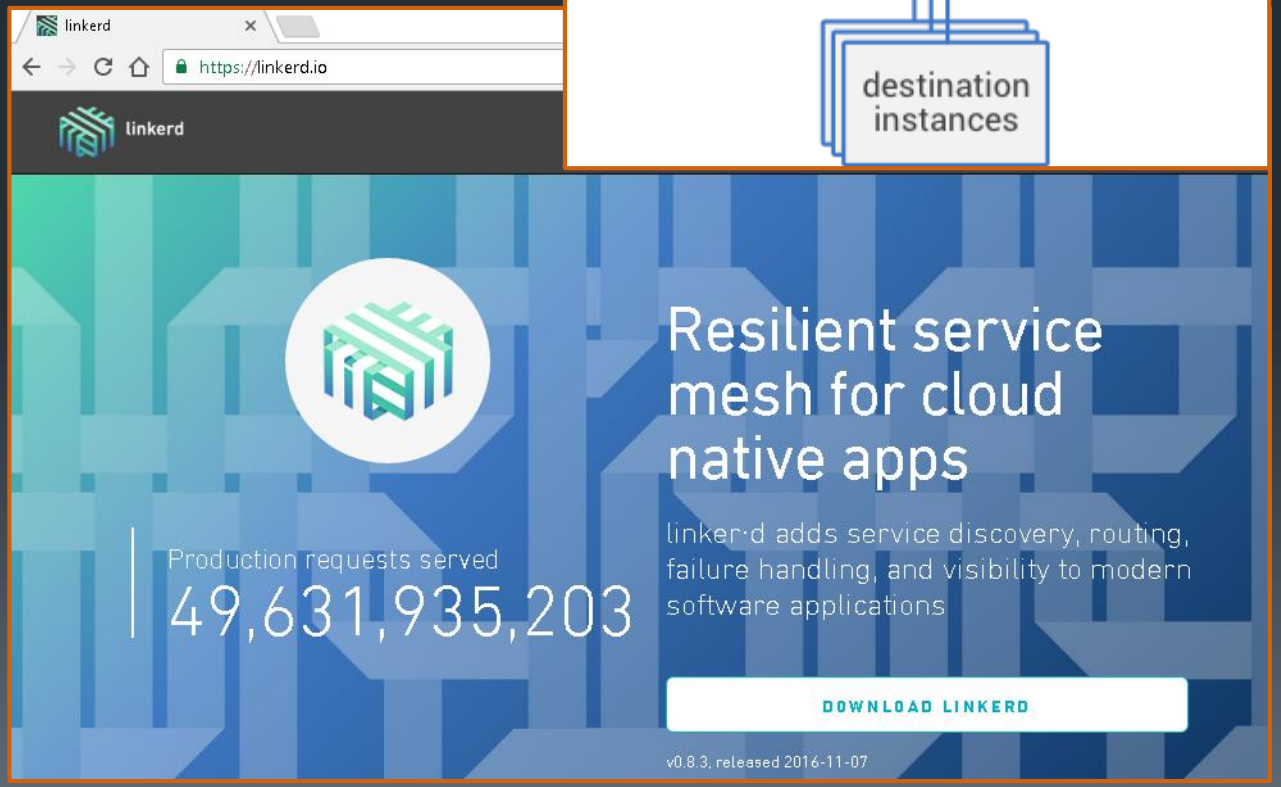




Linkerd

- An open source, scalable service mesh for cloud-native applications
  - Backed by Bouyant
- Built to improve the communication between services
  - 1.0 – Java
  - 2.0 - Rust
- Features:
  - Load balancing
  - Circuit breaking
  - Service discovery
  - Dynamic request routing
  - HTTP proxy integration
  - TLS
  - gRPC integration
  - Instrumentation

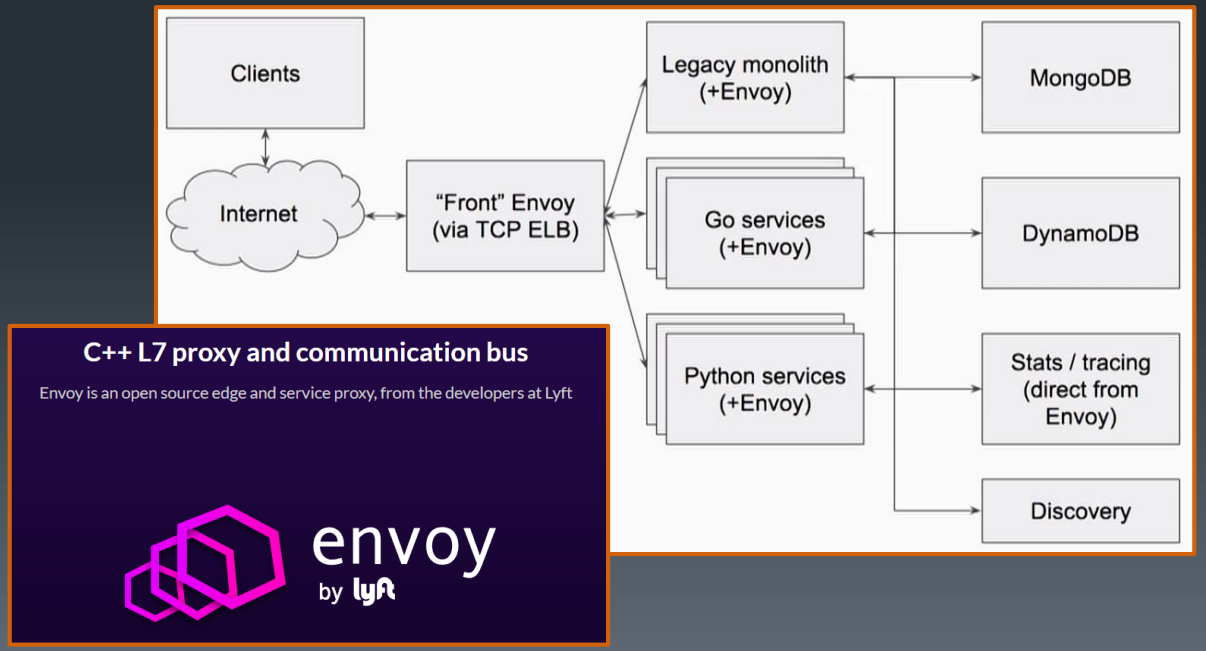




Envoy

- A self contained OSS network service proxy

- HTTP/2
  - First class in/out support for HTTP/2 and transparent HTTP/1.1 to HTTP/2
- Load balancing
  - Automatic retries
  - Circuit breaking
  - Global rate limiting
  - Request shadowing (copy requests to QA)
  - Zone local LB (avoids cross zone fees)
- Filtering
  - Envoy allows HTTP/TCP/IP filtering
  - Filters can be chained
  - New filters can be written
- Coded in C++11
- gRPC support
- HTTP routing
  - Redirection, virtual hosts, virtual clusters, matching on different request parameters
- TLS Support
  - Termination and initiation, client certificate verification, and certificate pinning
- Observability
  - Envoy exposes rich statistics and distributed tracing via third party providers (e.g. light step, zipkin)
- MongoDB
  - Envoy contains a MongoDB wire format parser used to gather statistics about database connections
- DynamoDB
  - Envoy contains a DynamoDB API parser that is used to gather statistics about database requests and responses
- Service discovery
  - Envoy supports asynchronous DNS resolution as well as integration with an external service discovery service
- Health checking
  - Envoy is capable of active health checking of backend servers
  - Active health checking along with service discovery yields eventually consistent and extremely resilient load balancing



Istio

- In a phrase: Kubernetes integrated Envoy
- Istio supports policy based traffic management between microservices
- Istio generates telemetry data
- Istio is transparent to the actual microservices it intermediates
- Features:
  - Automatic load balancing for HTTP, gRPC, and TCP traffic
  - Fine-grained control of traffic behavior with rich routing rules, retries, failovers, and fault injection
  - A pluggable policy layer and configuration API supporting access controls, rate limits and quotas
  - Automatic metrics, logs, and traces for all traffic within a cluster, including cluster ingress and egress
  - Secure service-to-service authentication with strong identity assertions between services in a cluster
- Only supports Kubernetes today
  - Support is planned for additional Cloud Foundry, Mesos, and bare metal
- Components:
  - Envoy Service Proxy
  - Mixer collects telemetry and enforces service mesh policy
  - Pilot is an interface for users to configure Istio via
  - Istio-Auth provides service-to-service and end-user authentication using mutual TLS

AWS API Gateway

- A managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at scale
- In Amazon API Gateway, an API refers to a collection of resources and methods that can be invoked through HTTPS endpoints
- Support for:
  - Amazon Elastic Compute Cloud (Amazon EC2)
  - AWS Lambda
- Handles:
  - traffic management
  - authorization
  - access control
  - Monitoring
  - API version management
- Charges for the API calls received and data transferred out

```
{
  "swagger": "2.0",
  "info": {
    "title": "PetStore",
    "description": "Sample API"
  },
  "schemes": [
    "https"
  ],
  "paths": {
    "/": {
      "get": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "text/html"
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "headers": {
              "Content-Type": {
                "type": "string"
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "responses": {
            "default": {
              "statusCode": "200",
              "responseParameters": {
                "method.response.header.Content-Type": "'text/html'"
              },
              "responseTemplates": {
                "text/html": "<html>\n<head>\n
                ...
                 </body>\n</html>"
              }
            }
          },
          "requestTemplates": {
            "application/json": "{\"statusCode\": 200}"
          },
          "type": "mock"
        }
      },  ...
```

Summary

- API gateway expose a client centric aggregate API, encapsulating the many finer grained services within a given bounded context

Lab 8

- Gateways

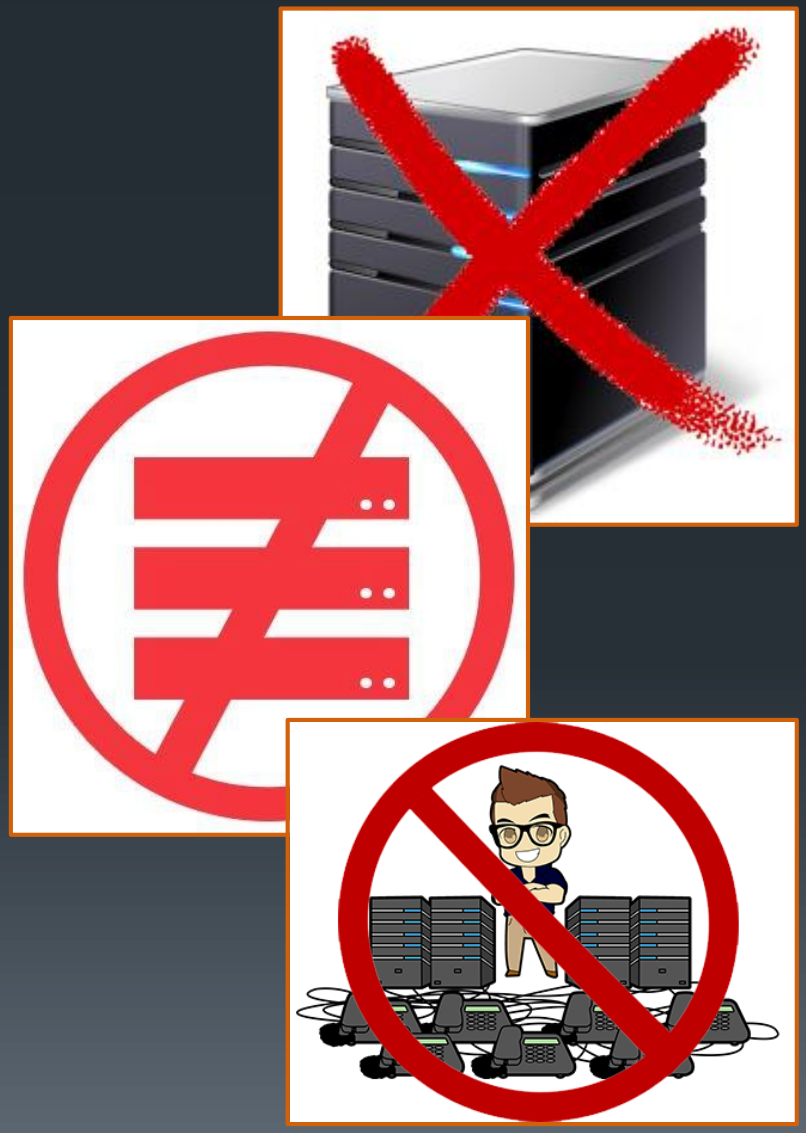
9: Serverless

Objectives

- Define serverless computing
- Understand the nature of functions as a service
- Describe the role of Gateways for functions
- Explain the means of triggering functions in the cloud
- Consider the various cloud function offerings and client side tools

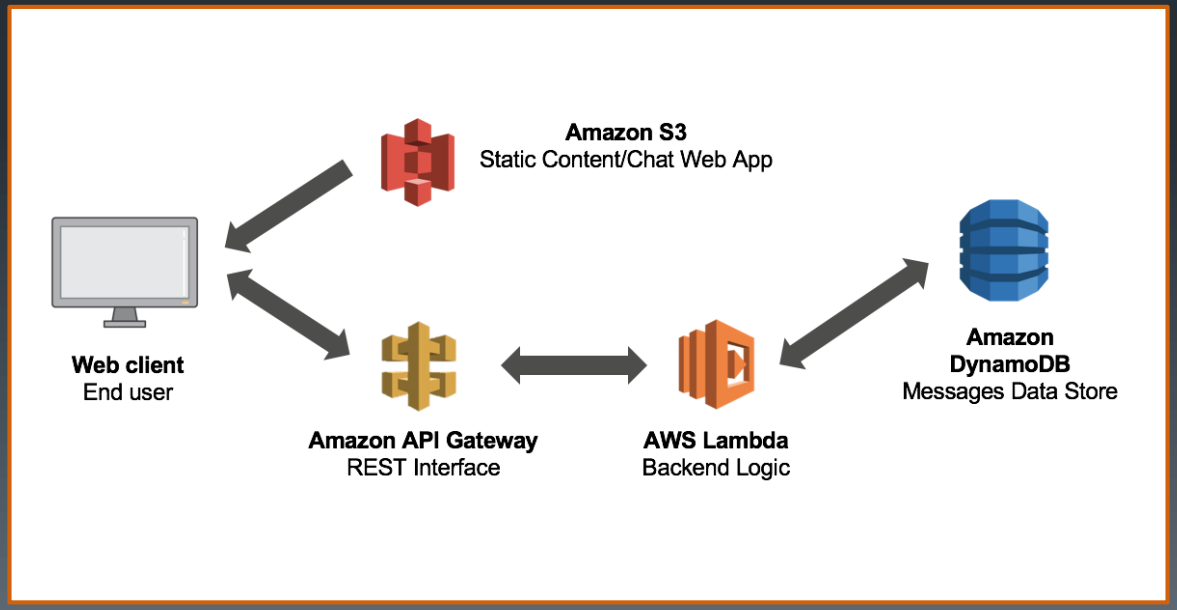
Serverless

- The Cloud has made possible fully "serverless" models of computing
  - Logic can be spun up on-demand in response to events originating from anywhere
  - Applications can be built from these bite-sized bits of business logic
  - Billing occurs only when code is running
  - Can serve users counts from zero to planet-scale without infrastructure management
- "As we move forward, it's becoming increasingly clear (to me at least) that the future will be containerized and those containers will run on serverless infrastructure."
  - Brendan Burns, Microsoft Distinguished Engineer and Kubernetes founder



Serverless computing

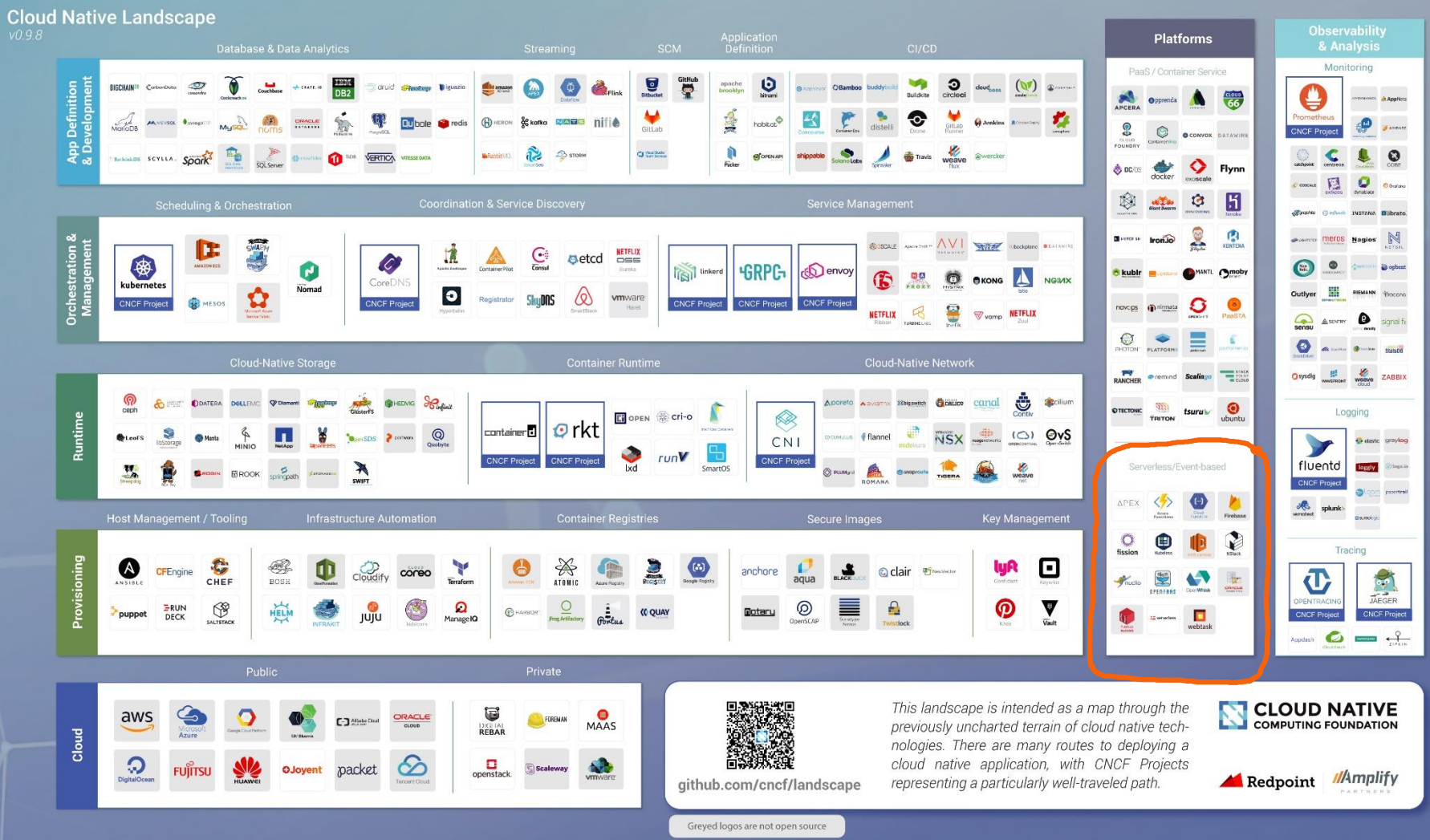
- Serverless computing
  - A cloud service model in which the provider fully manages the invocation and hosting of a function
  - Users provide the function code which must execute under one of the offered runtime environments
  - Users are billed each time the function is invoked, rather than per VM/hour
  - Also known as Function as a Service (FaaS)
- Behind the scenes the cloud provider supplies a PaaS environment for the function to run on (which of course involves servers)
  - "serverless computing" because user does not interact with, purchase, rent or provision servers
- Serverless code can be used in conjunction with normal services, such as microservices
- Serverless code is triggered is various ways
  - By specific events (such as user registration with Amazon Cognito)
  - By API calls to an API management platform exposing functions as REST API endpoints



History of FaaS

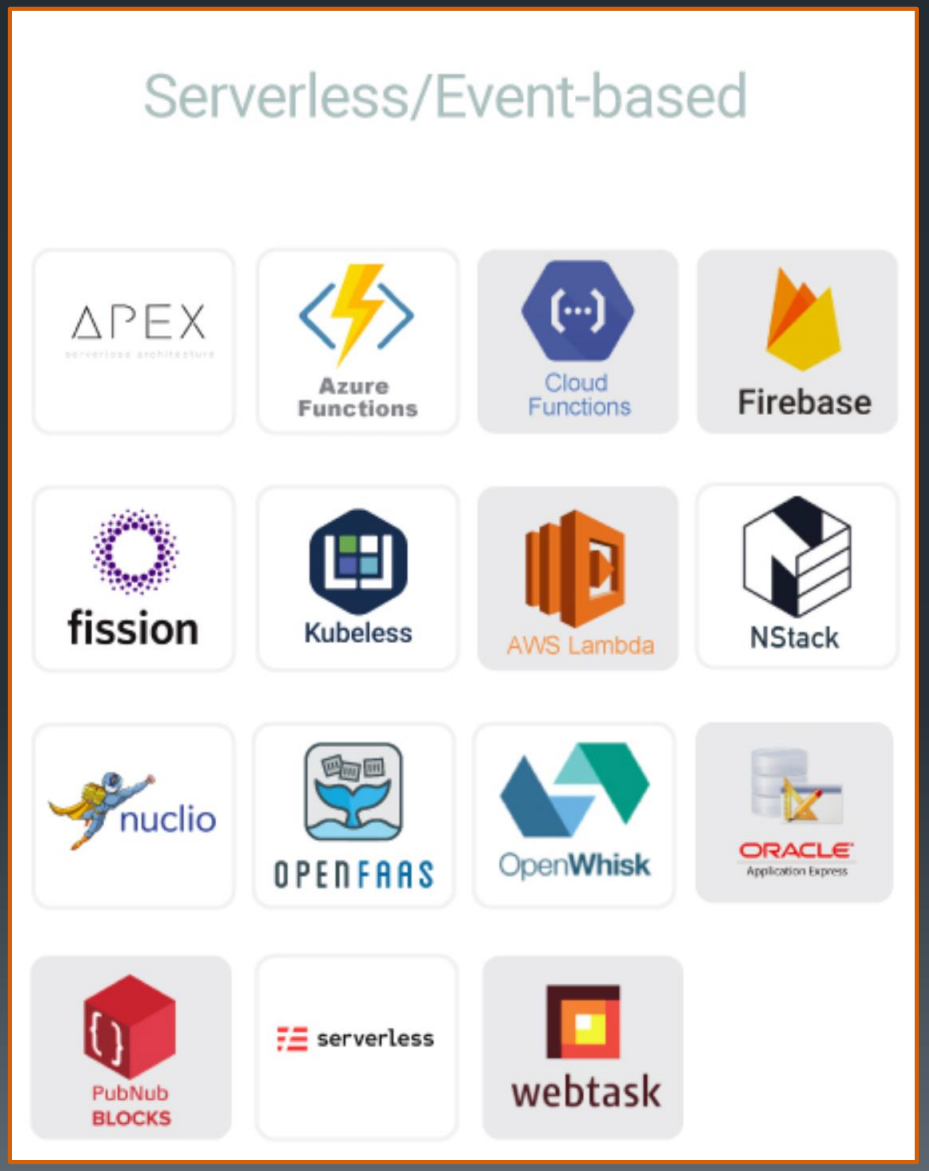
- 2006 – First serverless offering provided by Zimki
- 2007 – First serverless offering shutdown by Zimki
  - CTO quits on stage at OSCON, ouch
- 2014 – AWS Lambda introduced, first major provider offering
  - Launched for Node.js later added Python, Java, C#
  - Others languages supported using Node.js as an invoker
- 2016 January – Google Cloud Functions alpha
  - Node.js support (now in beta)
- 2016 February – IBM OpenWhisk announced
  - Open source FaaS project for OpenStack
    - Now Apache OpenWhisk
  - Runs functions in Docker containers (any language)
  - Also PaaS support for Node and Swift (later added Python and Java)
- 2016 June – Serverless Framework appears on GitHub as the first application framework for FaaS
- 2016 November – Microsoft Azure Functions
  - Open source and usable in Azure and on any other cloud environment, public or private
- 2017 January – OpenFaas arrives on GitHub as the first K8s FaaS framework
- 2018 April – Microsoft ACI (Azure Container Instances) goes GA
- 2018 June – AWS Fargate for ECS and EKS goes GA

Cloud Native Landscape

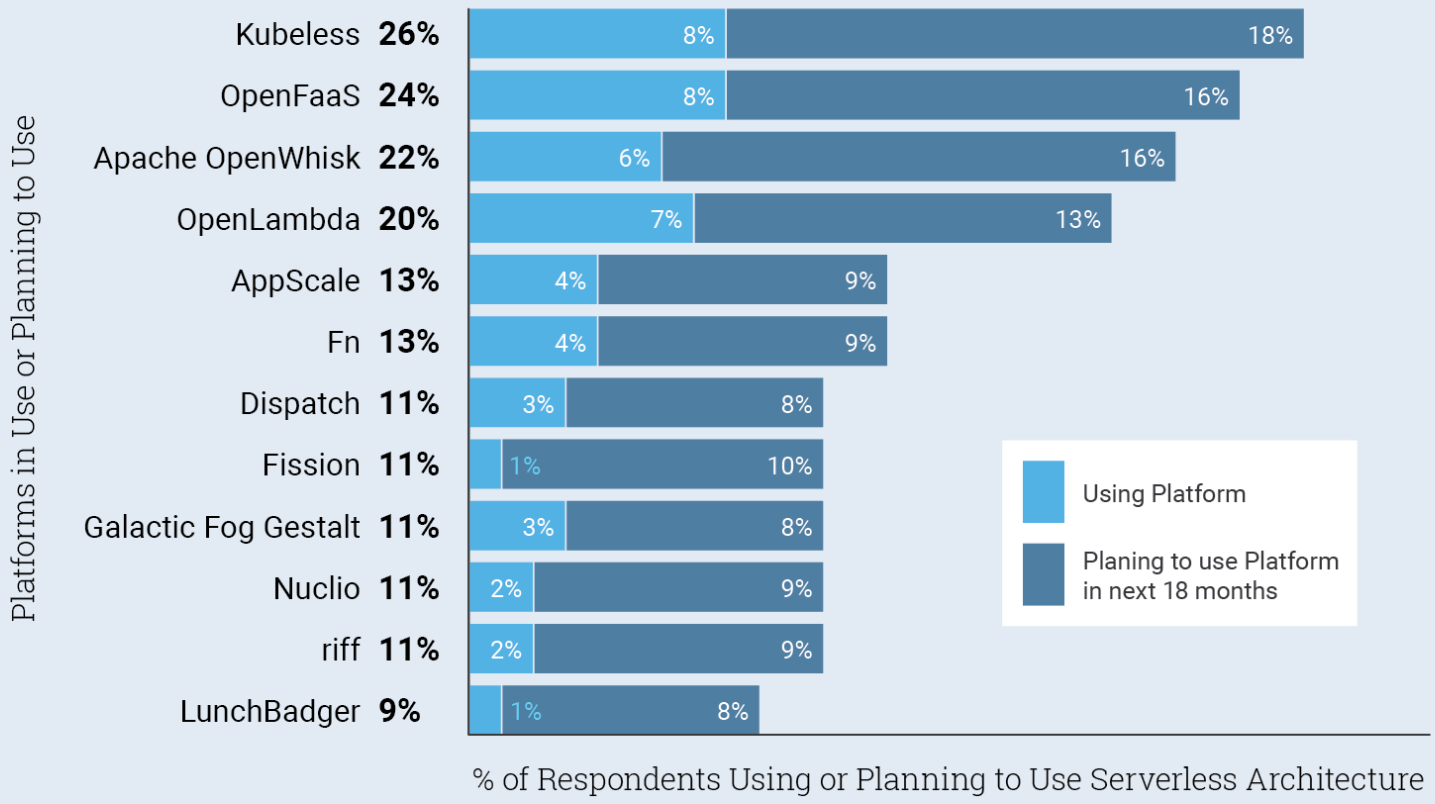


Approaches to FaaS

- Public Cloud
  - AWS Lambda
  - Azure Functions
  - Google Cloud Functions
  - IBM Cloud Functions
  - PubNub Blocks
- Database centric
  - Oracle APEX (Application Express)
    - API/Event integration with Oracle DBs
  - Firebase
    - API/Event integration with GCF
- Frameworks/IDEs
  - APEX
    - Dev front end for AWS Lambda
  - Serverless
    - Multicloud Framework for FaaS applications
  - Webtask
    - Cloud IDE for building task based applications
- Open Source Platforms
  - Fission [targets k8s]
  - Kubeless [targets k8s]
  - Nstack
    - Cross cloud serverless analytics platform
  - Nuclio [targets k8s]
    - Serverless scale-out event processing platform
  - OpenFaaS [targets k8s/swarm]
  - OpenWhisk
  - Knative [targets k8s/istio]

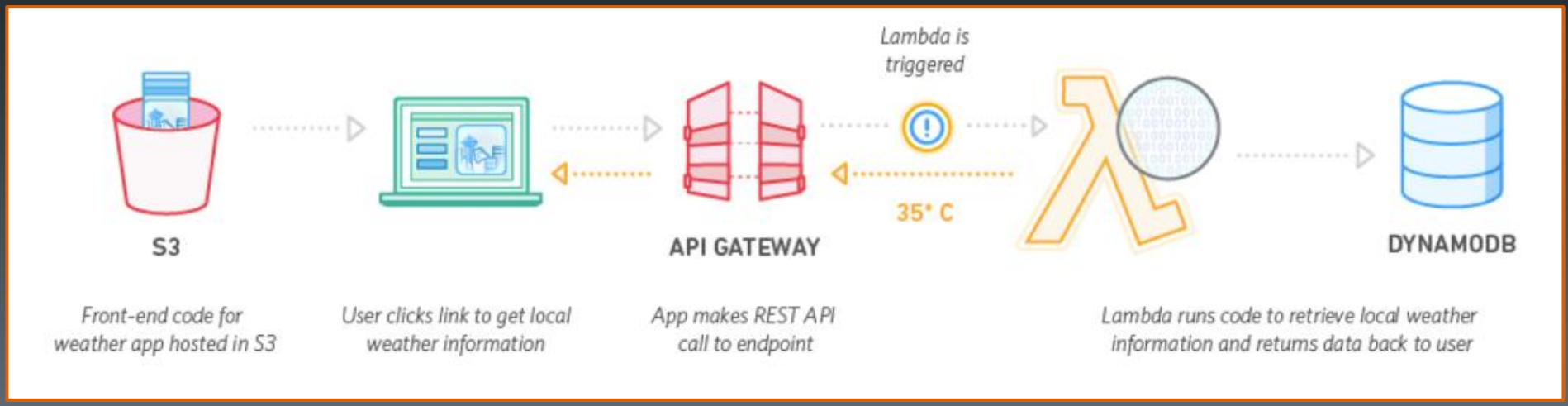


Open Source FaaS Market Share



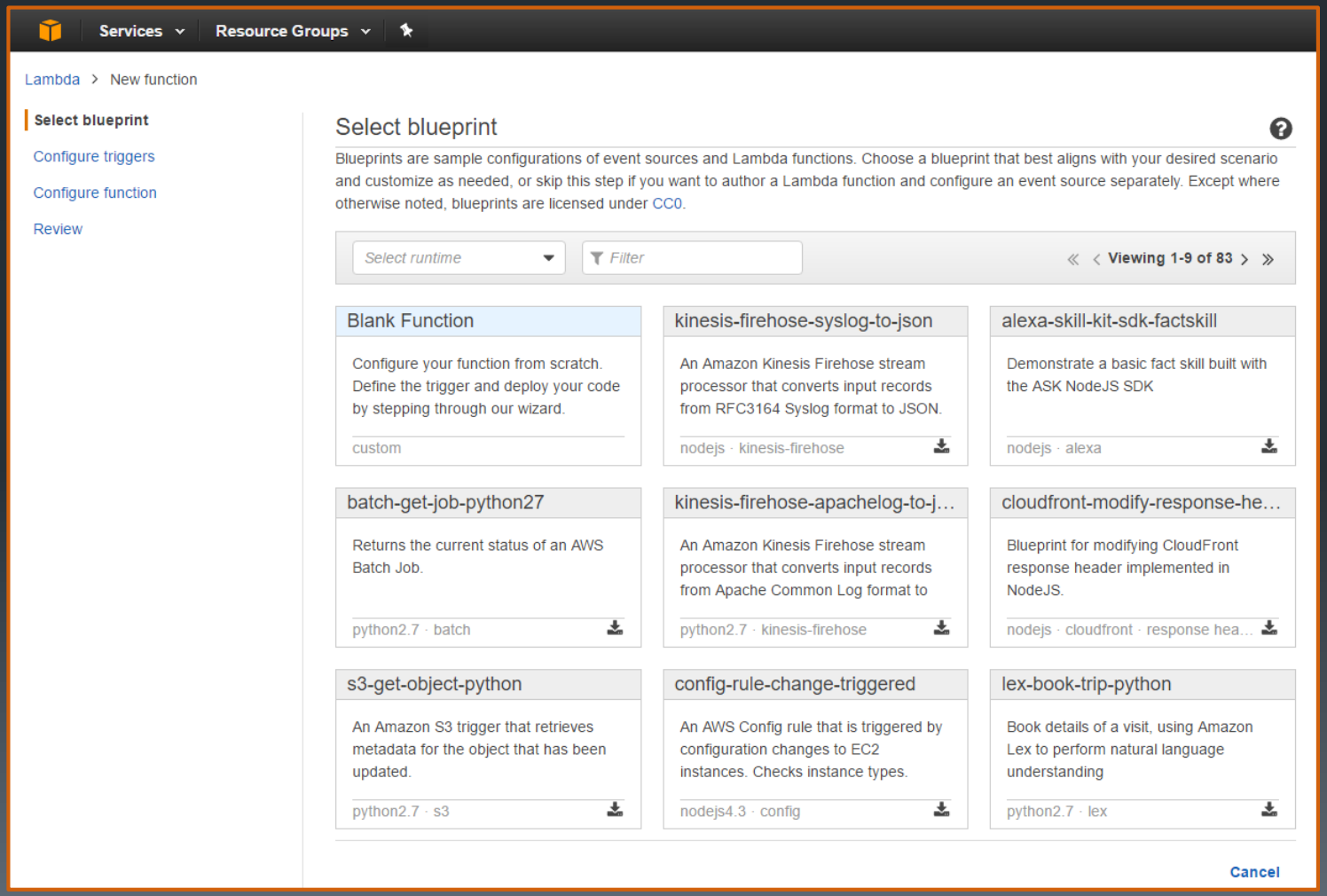
AWS Lambda

- AWS Lambda lets you run code without provisioning or managing servers
- Pay for the compute time you consume
- Lambda can run code for virtually any type of application or backend service with zero administration
- Upload code and Lambda takes care of everything required to run and scale the code with high availability
- You can set up code to automatically trigger from other AWS services or call it directly from any web or mobile app
- AWS Lambda supports Java, Node.js, and Python
  - More expected



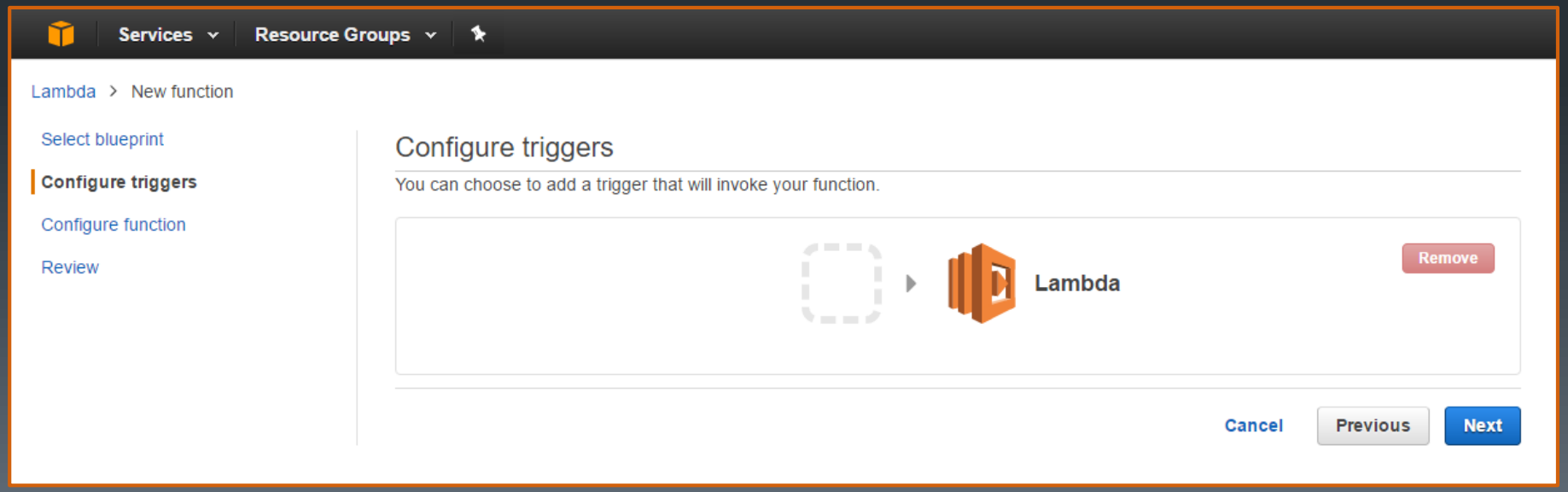
AWS Blueprints

- AWS offers over 80 lambda blue prints to help you quickly wire functions to other AWS services



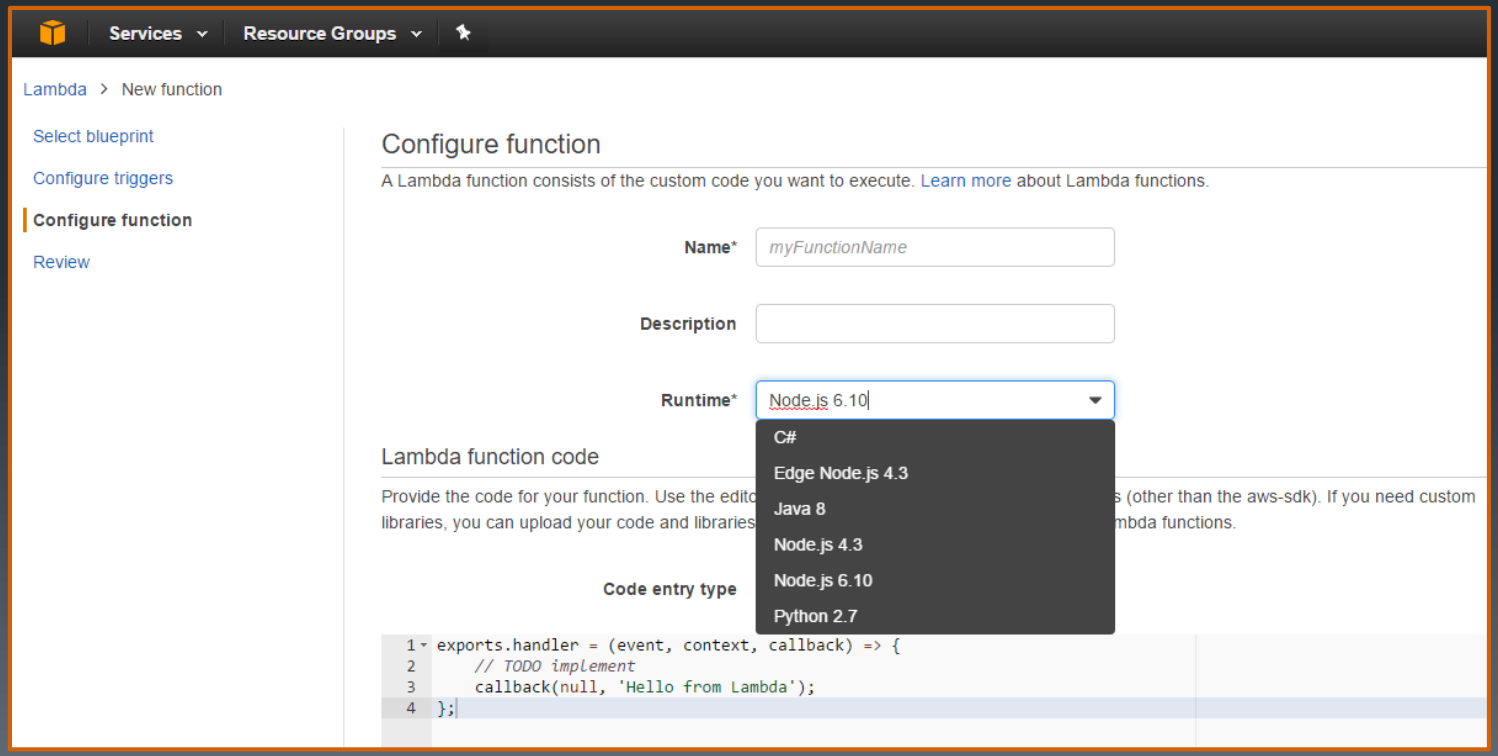
AWS Triggers

- Lambdas can be invoked by one or many events within AWS
  - Amazon S3
  - Amazon DynamoDB
  - Amazon Kinesis Streams
  - Amazon Simple Notification Service
  - Amazon Simple Email Service
  - Amazon Cognito
  - AWS CloudFormation
  - Amazon CloudWatch Logs & Events
  - AWS CodeCommit
  - Scheduled Events (powered by Amazon CloudWatch Events)
  - AWS Config
  - Amazon Echo
  - Amazon Lex
  - Amazon API Gateway
  - Invoked On Demand



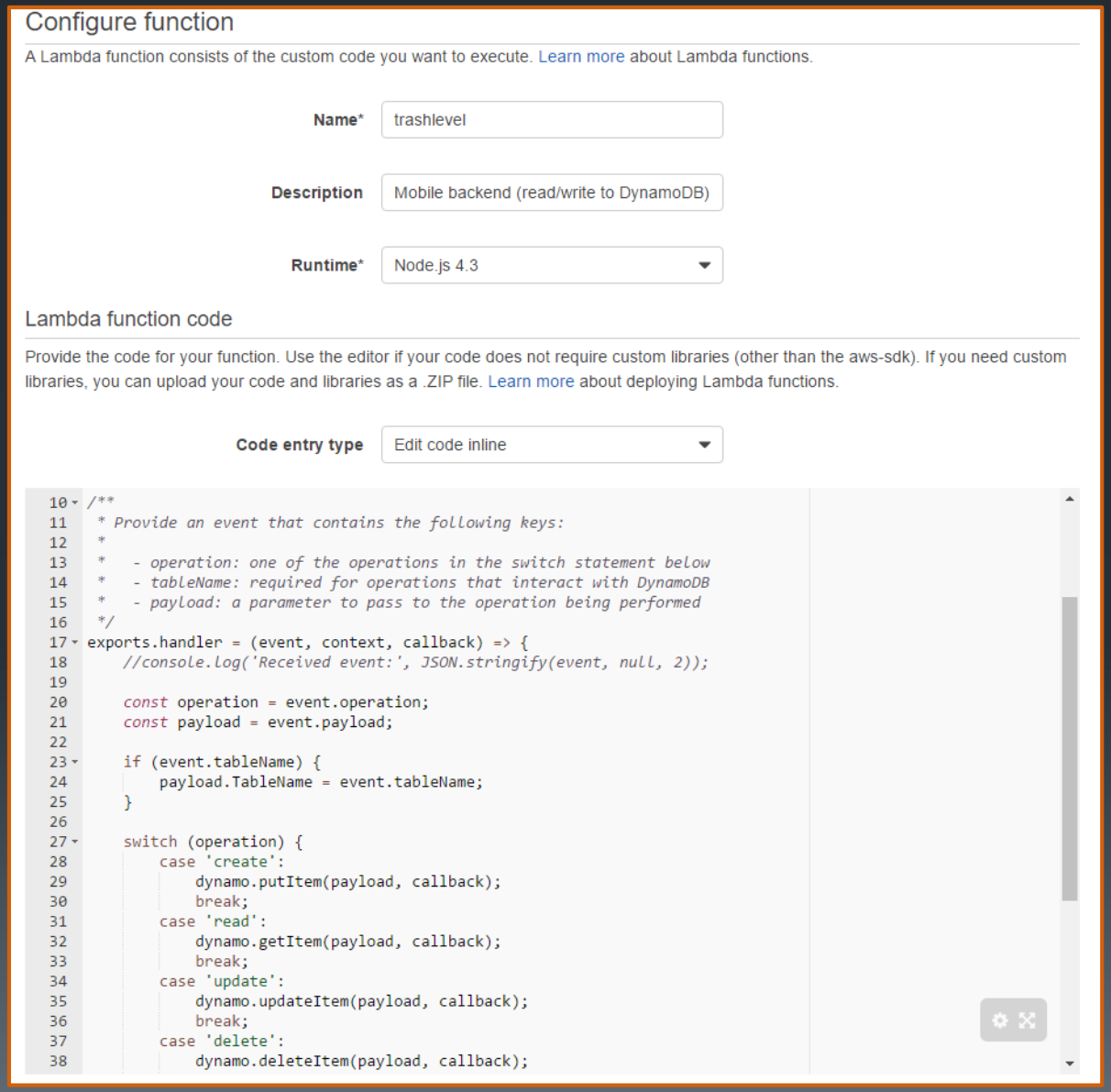
Lambda Language Support

- Lambda functions can be uploaded various ways:
  - raw code
  - Zip (if dependencies other than aws-sdk are required)
  - S3
- Lambdas can be coded in C#, Node.js, Java or Python
  - You can invoke binaries from Node.js to use other languages



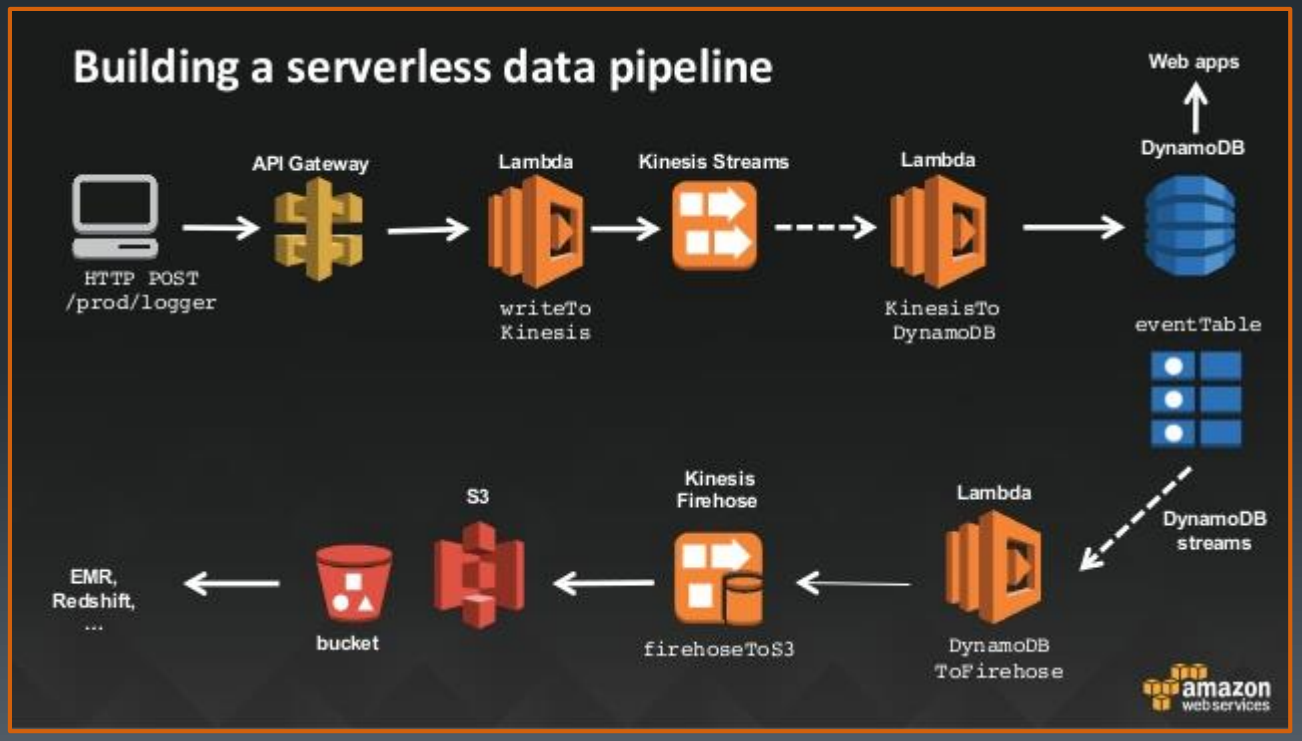
Accessing state from Lambda functions

- Lambdas have built in access to the aws-sdk
- They can use Dynamo, S3, RDS and any other stateful AWS services
- Lambdas are configured with a IAM roles which control their platform permissions
- Lambdas are built to process events
  - Platform events (such as API gateway calls) can then be mapped to lambda events



Serverless Big Data

- Extensive pipelines can be configured through event chains invoking one or more lambdas in the progression of processing

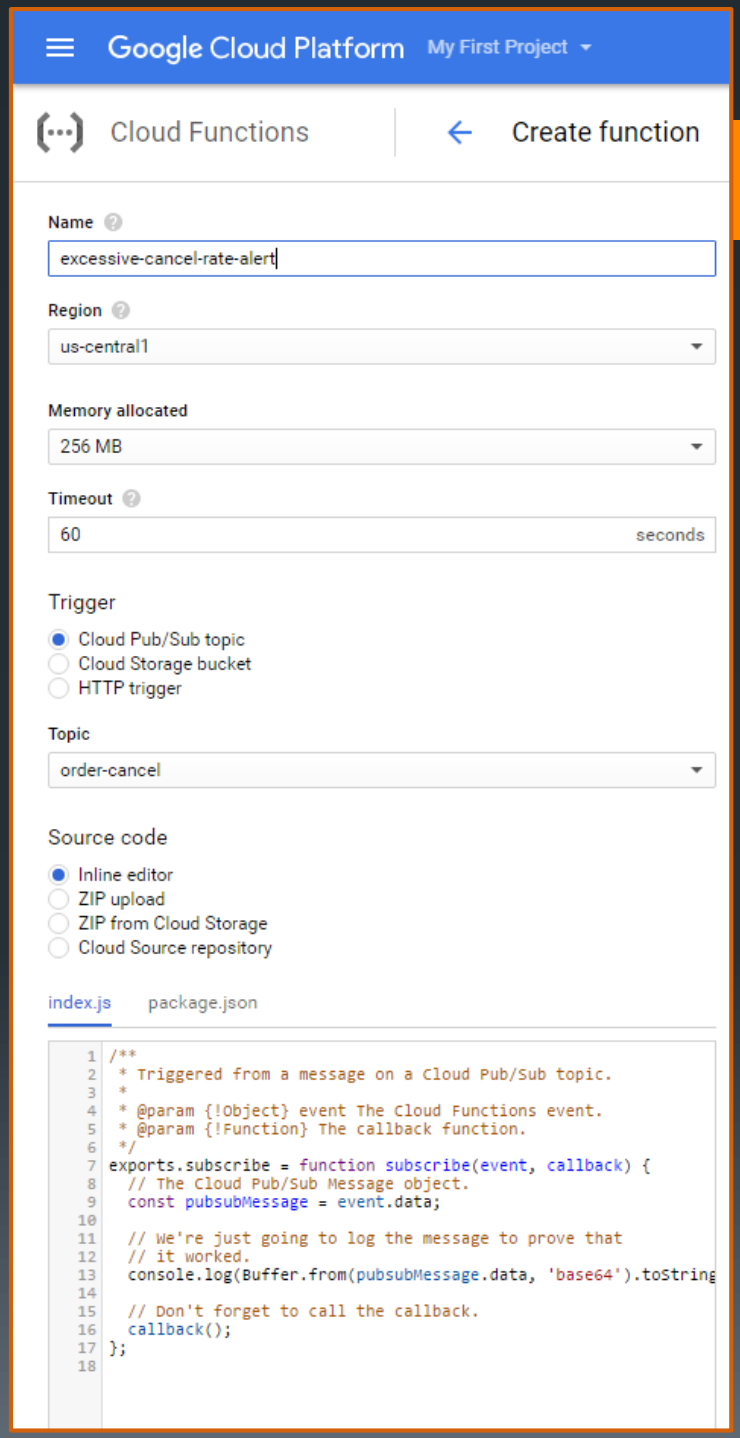


Google Cloud Functions

- Google Cloud Functions is a lightweight compute solution for developers to create single-purpose, stand-alone functions that respond to Cloud events without the need to manage a server or runtime environment
- You can invoke Cloud Functions with an HTTP request using the POST, PUT, GET, DELETE, and OPTIONS HTTP methods
  - To create an HTTP endpoint for your function, you specify --trigger-http as the trigger type when deploying your function
  - From the caller's perspective, HTTP invocations are synchronous, meaning that the result of the function execution will be returned in the response to the HTTP request

- `gcloud beta functions deploy helloHttp --stage-bucket cloud-functions --trigger-http`

- `curl -X POST https://<YOUR_REGION>-<YOUR_PROJECT_ID>.cloudfunctions.net/helloHttp -H "Content-Type:application/json" --data '{"name":"Keyboard Cat"}'`



Economics of FaaS

- Pay for use not while idle!
  - 1 GB-second is 1 second of wallclock time with 1GB of memory provisioned
  - 1 GHz-second is 1 second of wallclock time with a 1GHz CPU provisioned

CLOUD FUNCTIONS PRICING

Google Cloud Functions charges for invocations, compute time, and outbound data. Inbound data, and outbound data to other Google APIs in the same region is free. For detailed pricing information, please view the pricing guide.

	FREE *LIMIT PER MONTH*	PRICE *ABOVE FREE LIMIT (PER UNIT)*	PRICE UNIT
Invocations *	2 million invocations	$0.40	per million invocations
Compute Time	400,000 GB-seconds	$0.0000025	per GB-Second
	200,000 GHz seconds	$0.0000100	per GHz-Second
Outbound Data (Egress)	5GB	$0.12	per GB
Inbound Data (Ingress)	Unlimited	Free	per GB
Outbound Data to Google APIs in same region	Unlimited	Free	per GB

Azure Functions

- Triggers:
  - Timers (cron)
  - Platform Events (object upload, new message in queue, new vm created, etc.)
  - HTTP End points
- Supported Languages:
  - JavaScript – NPM support
  - C# - NuGet support
  - F#
  - Python
  - PHP
  - Bash
  - Batch
  - PowerShell
  - Any binary executable
- Integrates with:
  - Visual Studio Team Services
  - GitHub
  - BitBucket

Open sourced here:
https://github.com/azure/azure-webjobs-sdk-script

As an example, here's a simple Node.js function that receives a queue message and writes that message to Azure Blob storage:

```
module.exports = function (context, workItem) {
    context.log('Node.js queue trigger function processed work item ', workItem.id);
    context.bindings.receipt = workItem;
    context.done();
}
```

And here's the corresponding **function.json** file which includes a trigger **input binding** that instructs the runtime to invoke this function whenever a new queue message is added to the `samples-workitems` queue:

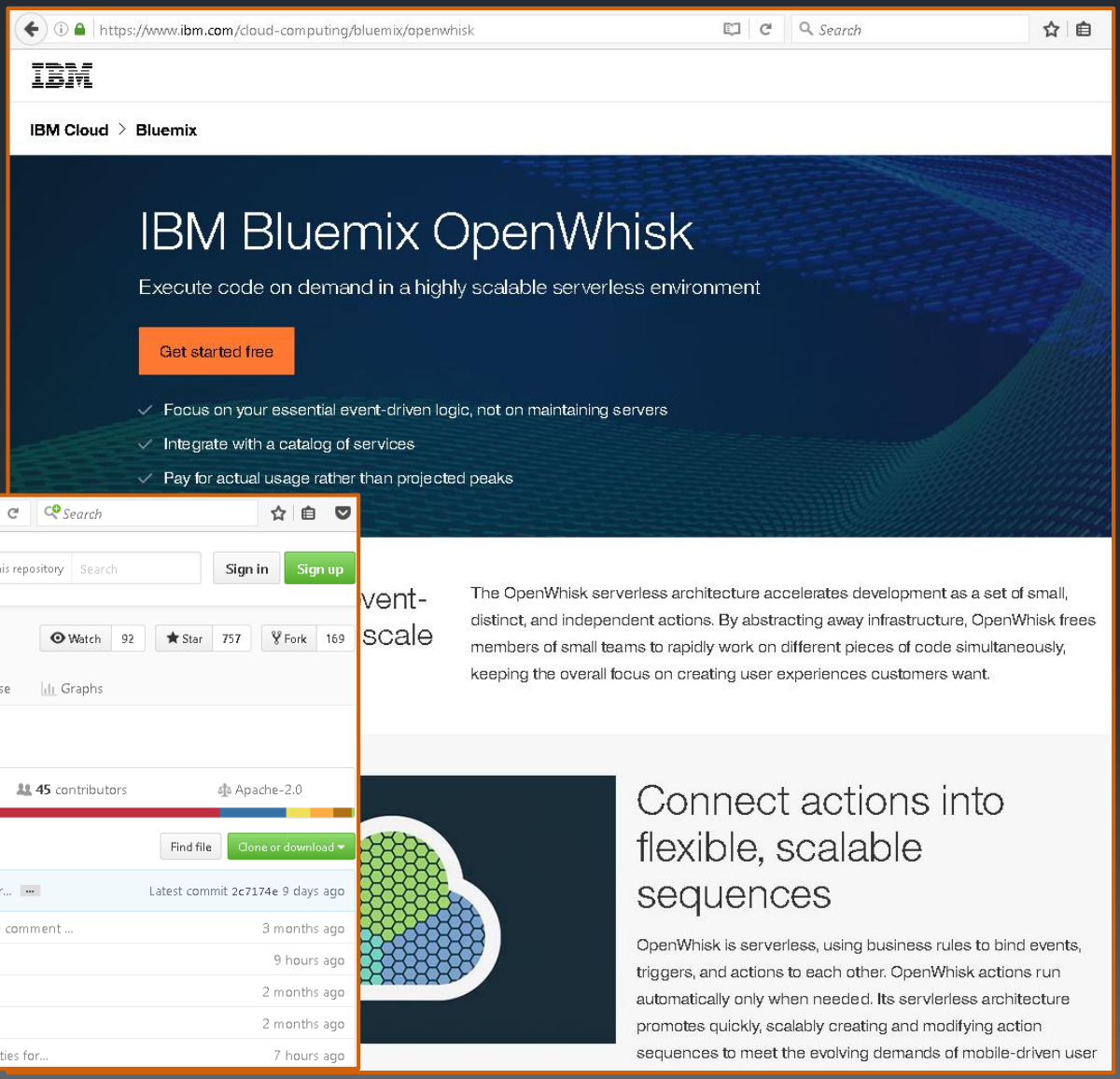
```
{
  "bindings": [
    {
      "type": "queueTrigger",
      "direction": "in",
      "queueName": "samples-workitems"
    },
    {
      "type": "blob",
      "name": "receipt",
      "direction": "out",
      "path": "samples-workitems/{id}"
    }
  ]
}
```

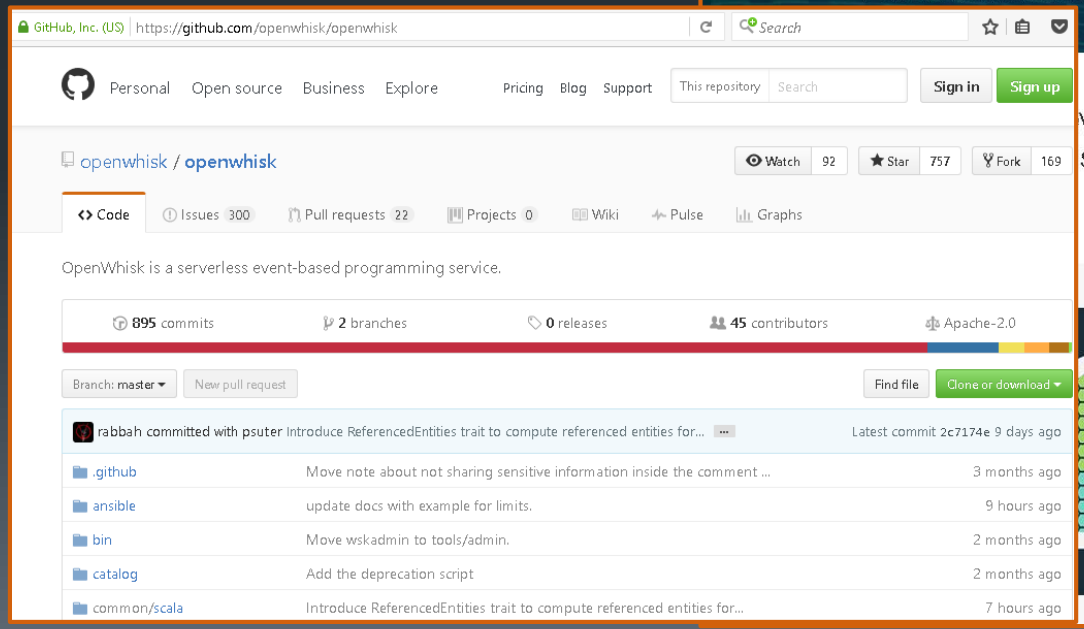
And here's a Python function for the same function definition doing the same thing:

```
import os

# read the queue message and write to stdout
workItem = open(os.environ['input']).read()
message = "Python script processed work item '{0}'".format(workItem)
print(message)

# write to the output binding
f = open(os.environ['receipt'], 'w')
f.write(workItem)
```

IBM OpenWhisk

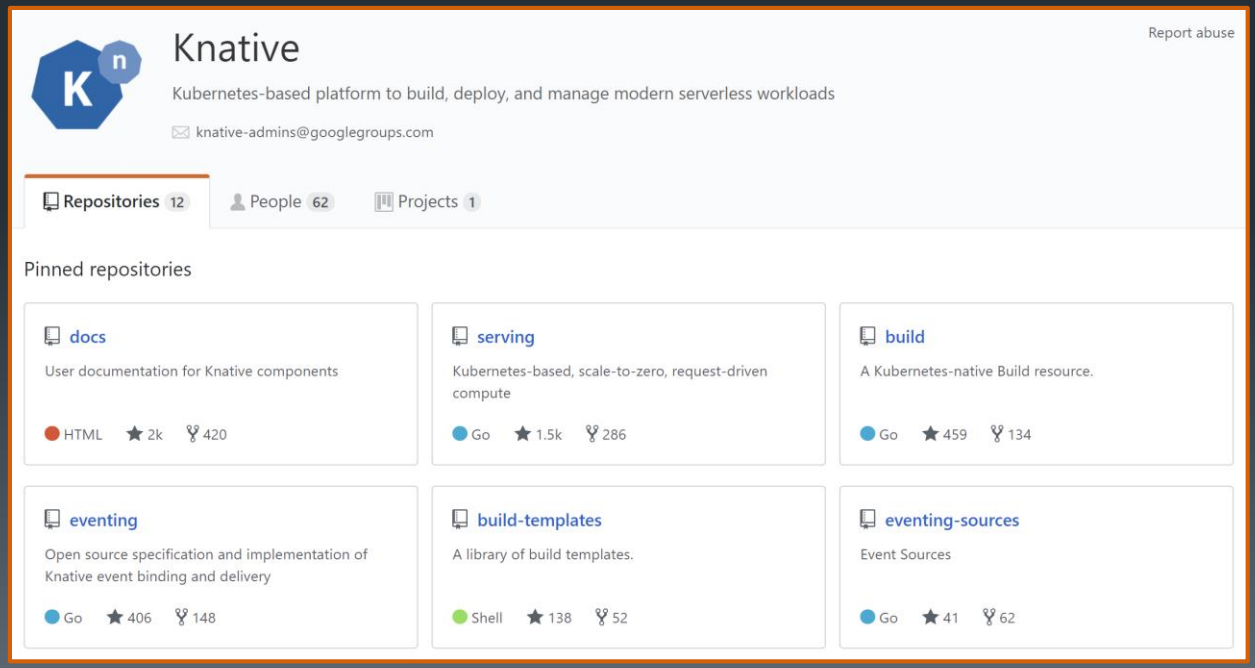
- An open source lambda platform
  - Now Apache OpenWhisk
- Runs on BlueMix, OpenStack, AWS, Vagrant etc.
- Can execute containers
  - No Language restrictions



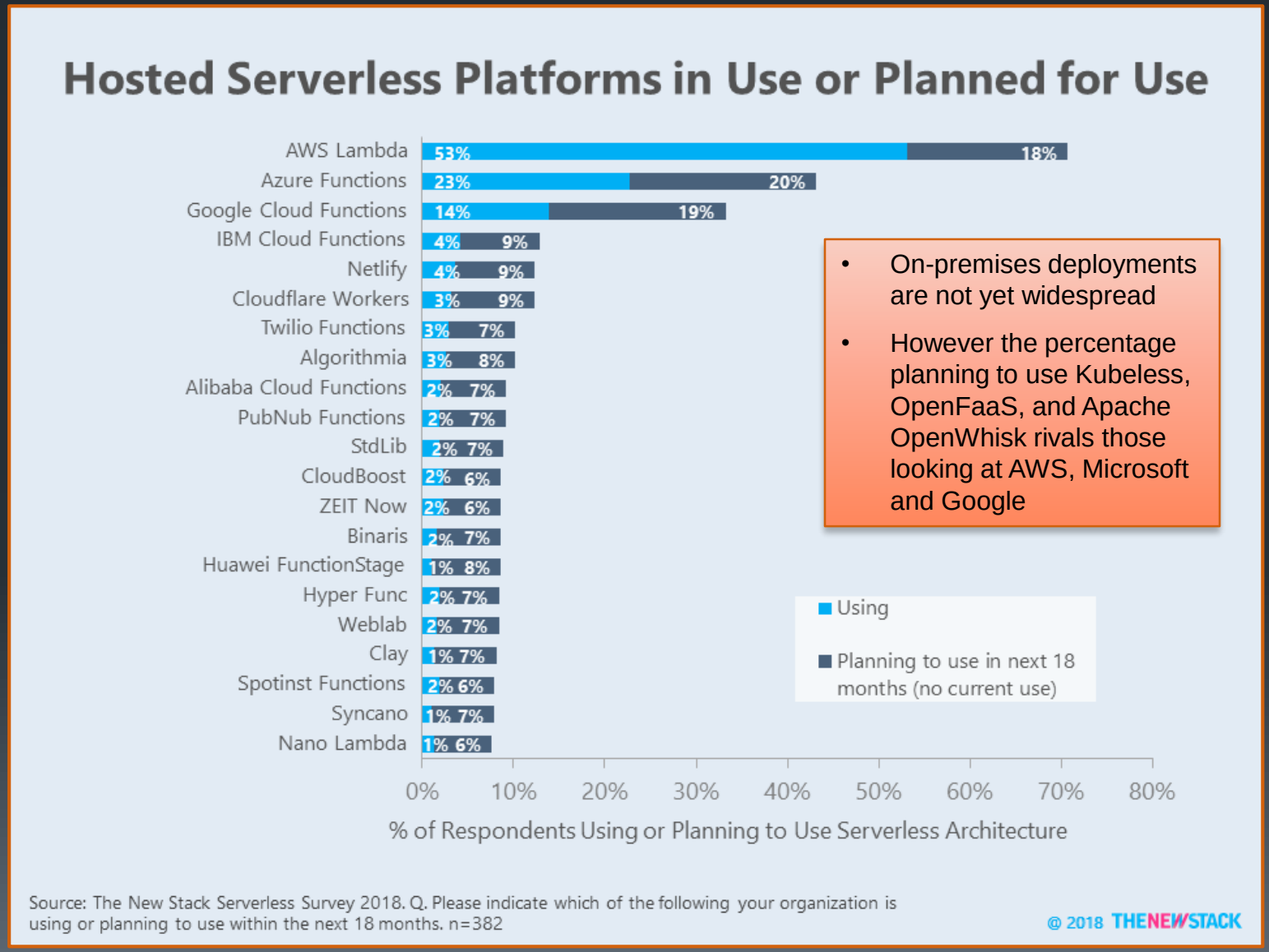


Knative

- Kubernetes-based platform to build, deploy, and manage modern serverless workloads
- Provides middleware components support source-centric and container-based applications
- Orchestrates source-to-container workflows
- Routes and manages traffic during deployment and auto-scaling
  - Using Istio
- Binds running services to eventing ecosystems
- Backed by: Google, Pivotal, SAP, Red Hat/IBM



Hosted FaaS Marketshare



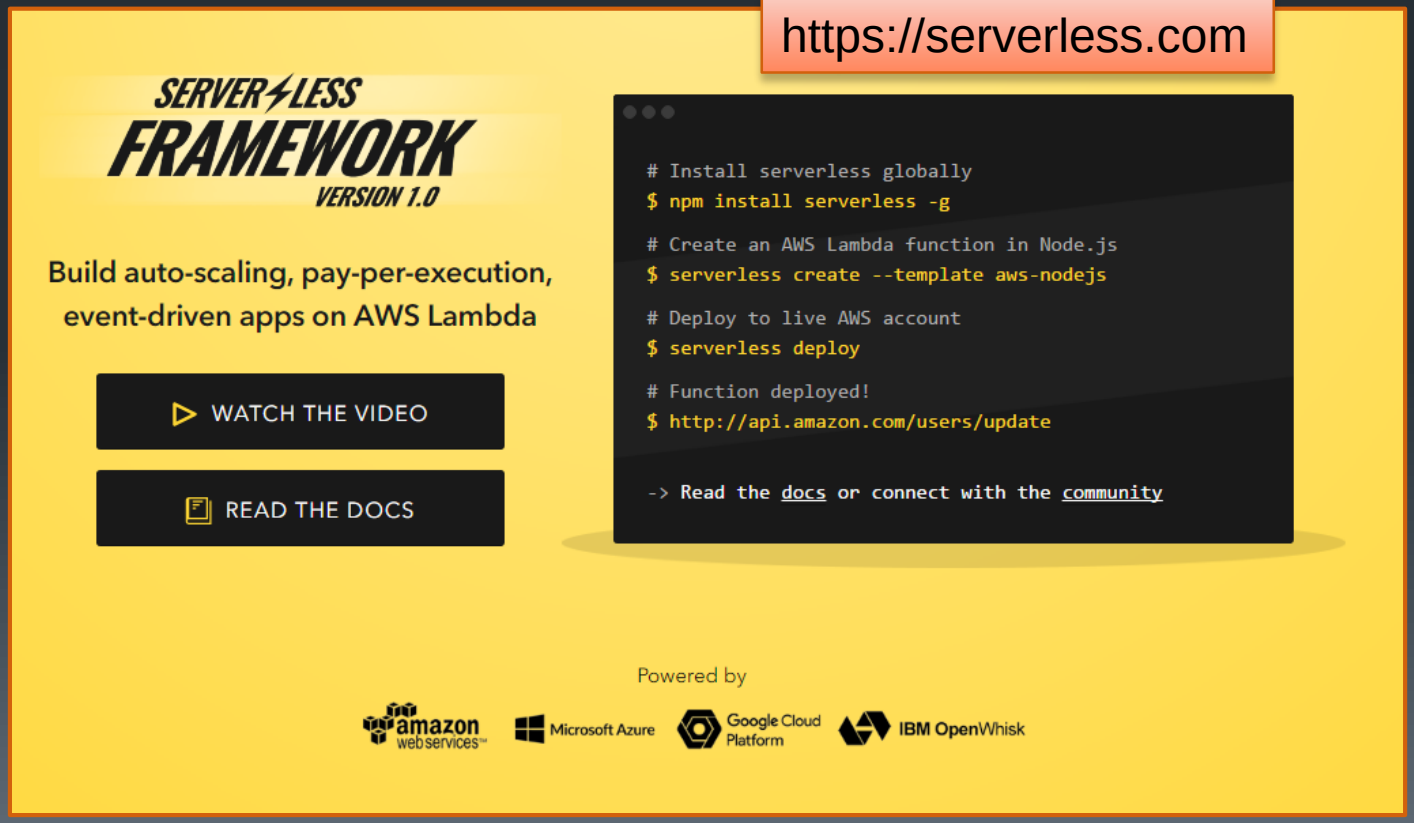
Open Source Serverless Platforms

- Open-source serverless Cons:
  - Not as integrated as cloud provider serverless
- Open-source serverless Pros:
  - Broader event/integration options
  - More parameter choices
  - Local debugging support
  - Generally faster
    - Ability to avoid cold starts

Serverless Frameworks

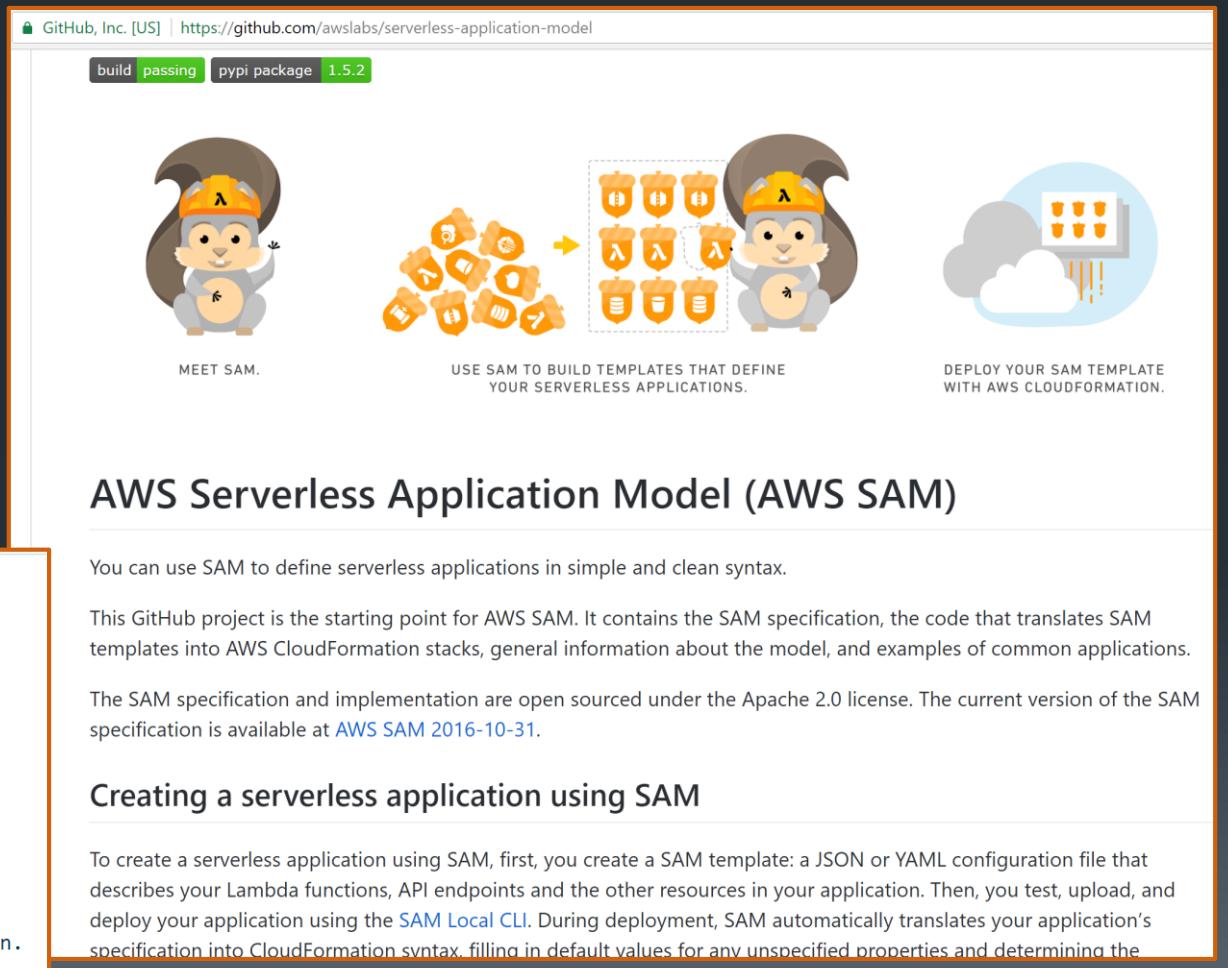
- The Serverless Framework helps you develop and deploy your functions, along with the infrastructure resources they require
- A CLI that offers structure, automation and best practices
- The Serverless Framework is different than other application frameworks because it manages your code as well as your infrastructure, configuring and wiring events to functions

https://serverless.com



AWS Serverless Application Model

- AWS SAM
- Prescribes rules for expressing Serverless applications on AWS
- Apache 2.0 Open Source
  - https://github.com/awslabs/serverless-application-model



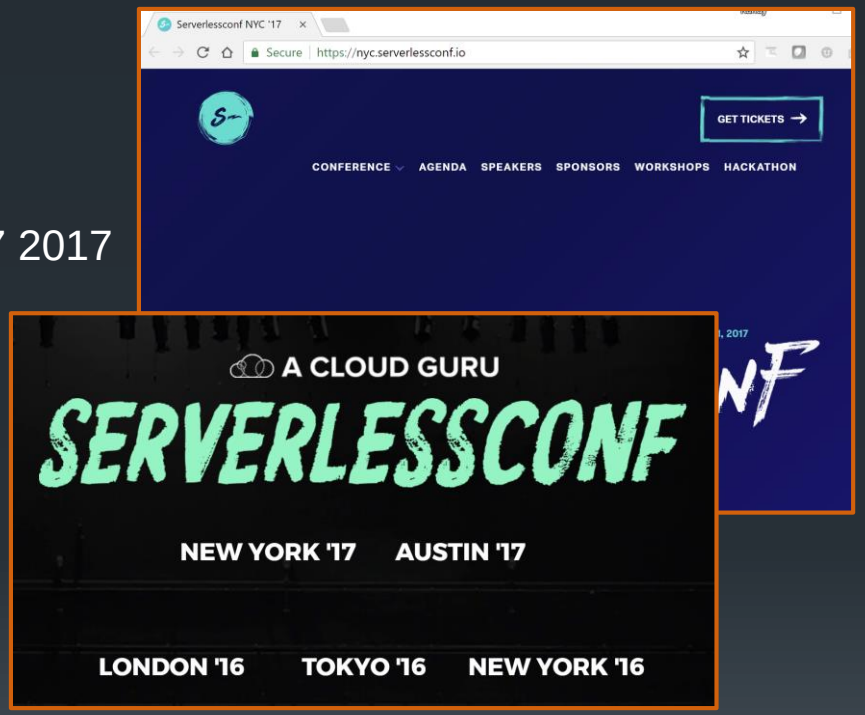

```
1  AWSTemplateFormatVersion: '2010-09-09'
2  Transform: 'AWS::Serverless-2016-10-31'
3  Description: A starter AWS Lambda function.
4  Resources:
5    helloworld:
6      Type: 'AWS::Serverless::Function'
7      Properties:
8        Handler: index.handler
9        Runtime: nodejs6.10
10       CodeUri: .
11       Description: A starter AWS Lambda function.
12       MemorySize: 128
13       Timeout: 3
```

FaaS Use Cases

- Mobile Backend
  - Gateways can map REST routes to FaaS routines which can then use other cloud services such as Analytics, Database, Authentication, and Storage
- APIs & Microservices
  - Functions can be API event-driven or invoked directly from other microservices
- Data Processing / ETL
  - Storage events can trigger FaaS methods, such as when a file is created, changed, or removed, causing image processing, transcoding, validation or transformation
- Webhooks
  - HTTP triggers can cause FaaS methods to respond to events originating from 3rd party systems like GitHub, Slack, Stripe, or from anywhere that can send HTTP/S requests
- IoT
  - Hundreds of thousands of devices streaming data into Cloud Pub/Sub systems can automatically launching FaaS methods to process, transform and store data

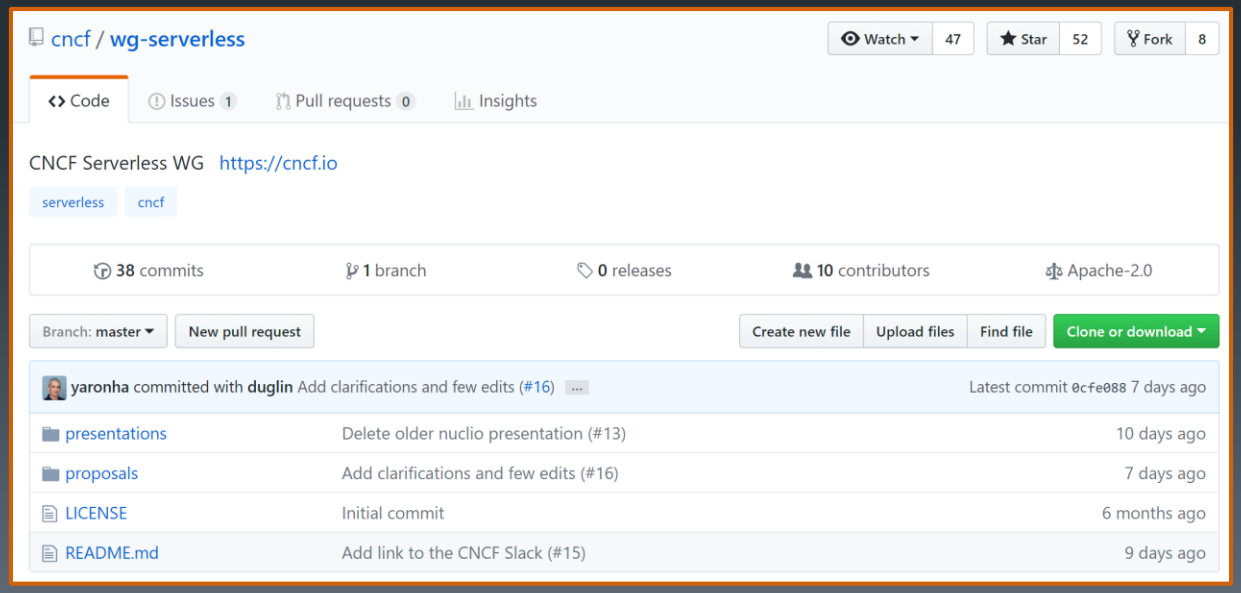
Serverless Conferences

- Serverless conferences are popping up everywhere
  - Serverlessconf.io
    - Hells Kitchen New York, Oct 2017
  - QCon: Learning from Expert Peers
    - New York June 26-30, Shanghai October 19-21 and San Francisco November 13-17 2017
    - All with Serverless tracks
  - AWS Re:Invent
    - Lots of coverage of AWS Lambda and serverless
    - Las Vegas Nov 2017
- Meetups
  - 127 meetups
  - > 30,000 members
  - https://www.meetup.com/topics/serverless-architecture/



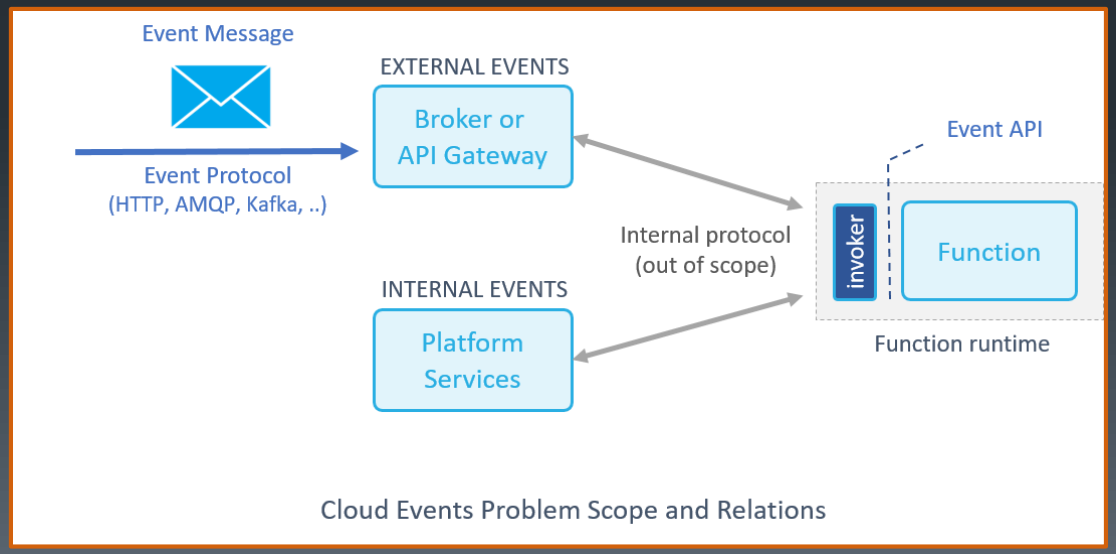
CNCF Serverless WG

- Exploring the intersection of cloud native and serverless
- Coms:
  - Google Group: cncf-wg-serverless
  - Slack #serverless channel: https://slack.cncf.io/
  - GitHub: https://github.com/cncf/wg-serverless
- Goals
  - Provide clarity for ourselves and consumers w.r.t. what this topic is all about
  - Define common terminology
  - Define the scope of the space as it exists today - over time this may change
  - Identify common use cases and patterns between existing implementations
  - Identify where serverless fits in relative to PaaS and container orchestration
  - Identify potential next steps for the community (may or may not happen within CNCF)
  - Identifying areas where we'd like to see some harmonization or interop work
  - Create a whitepaper on Serverless in relation to Cloud Native



Serverless Model

- Serverless communication between services and functions is done through events/messages
- Describing and publishing events is a key to standardization
  - The event message and protocol
  - This enables a common way to consume events (an Event API)
- A common definition will allow function portability across platforms and interactions between services on different platforms



OpenEvents

- Proposals to the CNCF WG for a common event format and API
  - OpenEvents
  - Cloud Auditing Data Federation (CADF) event format as standardized at the Digital Mgmt. Task Force (DMTF). See https://www.dmtf.org/standards/cadf
- A range of event processing models should be considered
  - Most cloud vendors have their own proprietary implementations of eventing and event distribution

openevents

Introduction

OpenEvents is a specification for a common, vendor-neutral format for event data.

All cloud providers (e.g. Microsoft, IBM, Google, Amazon) have their own event payload format. Lack of a consistent format means that developers must constantly re-learn how to receive events within a single provider, or across providers. This specification aims to drive convergence on a common set of attributes (e.g., names and types) for event data.

Our end goal is to offer this specification to the Cloud Native Computing Foundation.

Version

Version 0.3 - 2017/09/20

Schema

- **eventId** - *string* - ID of the event. Can be specified by the producer. The semantics of this string are explicitly undefined to ease the implementation of producers (e.g. a database commit ID).
- **eventType** - *string* - Type of the event (e.g. `customer.created`). Producers can specify the format of this, dependening on their service. This specification does not (yet) enforce a type format (up for discussion). We are also discussing putting the type version in here as well.

timezone Z - Timestamp of event creation generated by the producer.

event payload. Payload depends on the event type. Producer can specify format/ecoding of ema.

ontent type of the data (e.g. `application/json`).

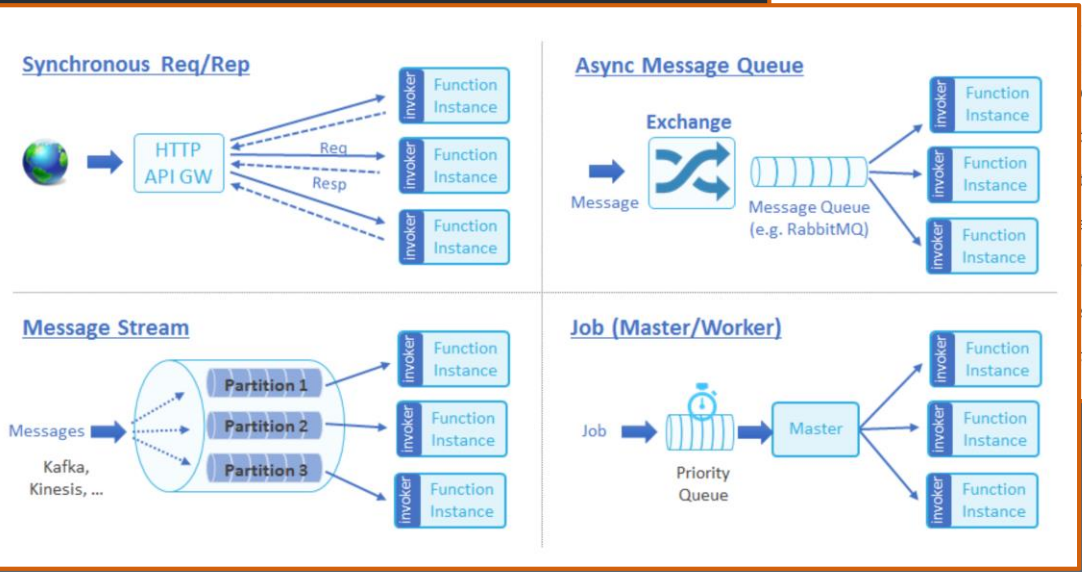
he resource that published the event. This object has the following fields:

event source. Providers define list of event sources.

nt source.

sion of the OpenEvents specification (e.g. `1.0`).

t> - This is for additional metadata and this does not have a required structure. The goals



Problems with Serverless

- Attempting to observe Function-as-a-Service (FaaS) serverless applications can present challenges
  - Nowhere to install monitoring agents
  - No opportunity for background processing
  - Telemetry data this has to be sent during a function invocation (when the user is still waiting on a potentially business critical response)
- AWS Case
  - Deep integration between AWS Lambda and Kinesis has made event-driven architectures easier to implement within AWS
  - Tracing function invocations through asynchronous event sources like AWS Kinesis is not currently supported by existing tools like Amazon X-Ray
- The space is developing quickly but many critical production tools remain to be created/enabled

Summary

- All of the major public cloud providers now offer FaaS
  - Amazon AWS Lambda
  - Google Cloud Platform – Cloud Functions
  - Microsoft Azure Functions
  - IBM OpenWhisk
    - Now Apache OpenWhisk

Lab 9

- Apache OpenWhisk

The End

- Many thanks for attending!

- Bounding Context
  - Models are consistent for ubiquitous languages within the context
  - A bounding context defines a monolith which can be decomposed into microservices
  - Just like a monolith, the set of implementing microservices should have an explicit API (gateway to the context)
- Immutability
  - Event Sourcing
  - Append only logs
  - Compensating transactions
- Idempotence
  - Single Writer for Idempotence and Order Based Consistency
  - “at least once” delivery ➔ “exactly once processing”
- Partitioning to Scale
  - Partial ordering
- Replication for Resilience (availability/durability)
- CAP (any two, continuously variable, only pay for the features you need)
  - Consistency
  - Availability
  - Partition Tolerance
- Eventually Consistency
  - Read your own writes
- Highly Consistent
  - Quorum writes
  - Quorum reads
- Sagas (Distributed Transactions)
  - Orchestration (call based)
  - Choreography (message based)

Key microservice architecture concepts and tools

Books & Pubs

- Domain Driven Design
  - Eric J. Evans, Addison-Wesley Professional
- Building Microservices
  - Sam Newman, O'Reilly
- Accelerate: The Science of Lean Software and DevOps
  - Building and Scaling High Performing Technology Organizations
  - Nicole Forsgren
- Microservices Patterns
  - Chris Richardson
- The New Stack
  - https://thenewstack.io
- Microservices Weekly
  - https://microserviceweekly.com