Chapter 1:
Kubernetes (K8s) Foundations
Microservices Patterns

Table of Content

- Advantage of Containers
- Need of an Orchestration Tool
- Introduction to Microservices
- Microservices Patterns
  - Data management
  - Communication
  - Deployment
  - Reliability

Containers

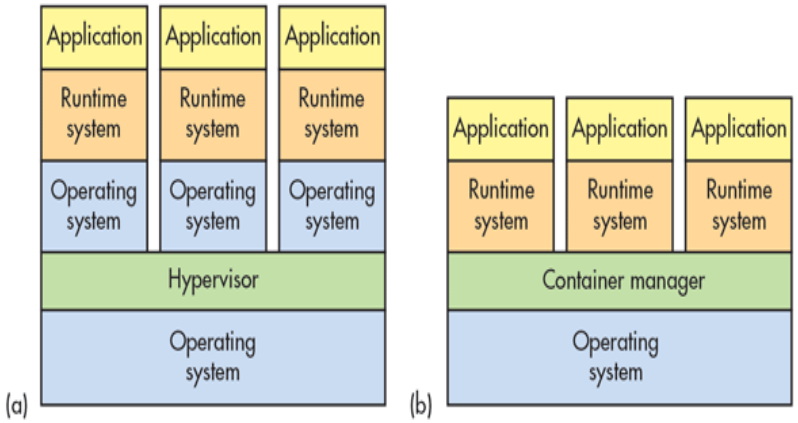
- Containers are becoming the standard unit of deployment
- Each container image has
  - Code
  - Binaries
  - Configuration
  - Libraries
  - Frameworks
  - Runtime
- Developers and Operators love containers

Containers

- Docker has solved the problem of packaging, deploying and running containerized applications
- Docker is great for managing a few containers running on a fewer machines
- Production applications deal with dozens of containers running on hundreds of machines

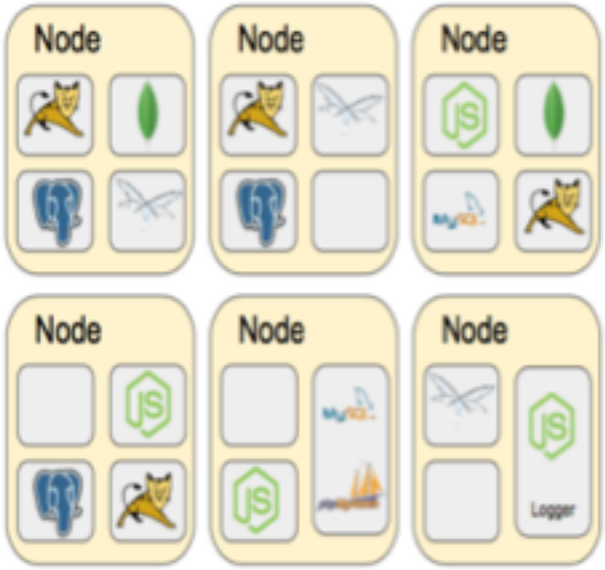
Container Advantages

- Portable
- Isolated
- Lighter footprint & overhead (vs VMs)
- Simplify DevOps practices
- Speed up Continuous Integration
- Empower Microservice architectures and adoption



2.1 Why need container orchestration

Challenges with multiple containers



- How to scale?
- Once I scale, where are they?
- How do my containers find each other?
- How should I manage port conflicts?
- What if a host fails?
- How to update them? Health checks?
- How will I track their logs?

Container Orchestration Tools

- The three most popular are: Kubernetes, Docker Swam & Mesos
- Kubernetes has become the unofficial standard

Introduction to Microservices and design patterns in Microservices

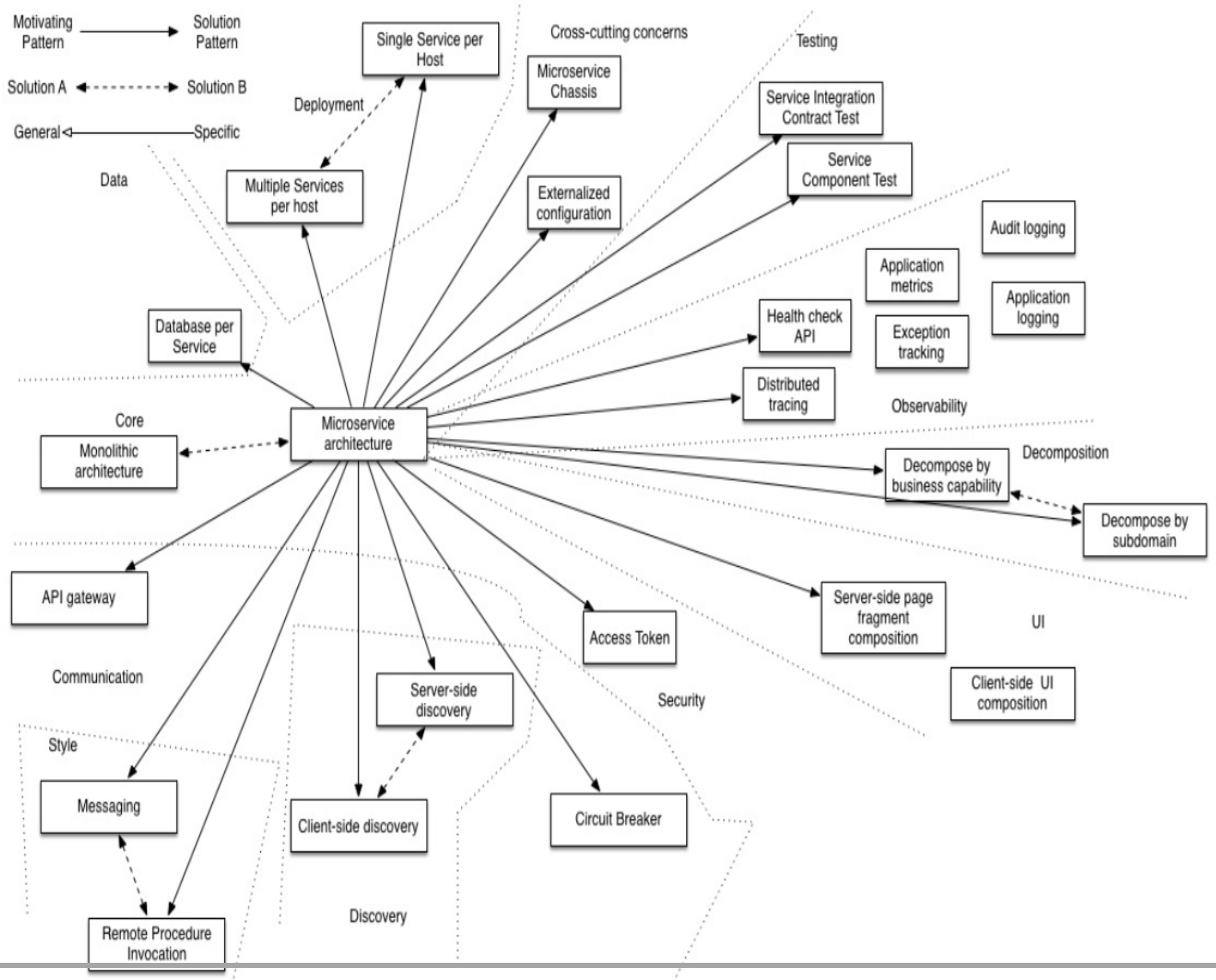
Microservices Overview

Microservices is defined as a loosely-coupled, service-oriented architecture with bounded context.

Microservices – Why do we need

- Context Switching
- Dependency management
- Multiple hand-offs
- Less Frequent Updates
- Slow Velocity
- App becomes complex

Microservices – Patterns



Pattern – Categories

- **Data Management**
- Communication
- Deployment
- Discovery
- Reliability
- Observability
- Testing

Data Management Patterns

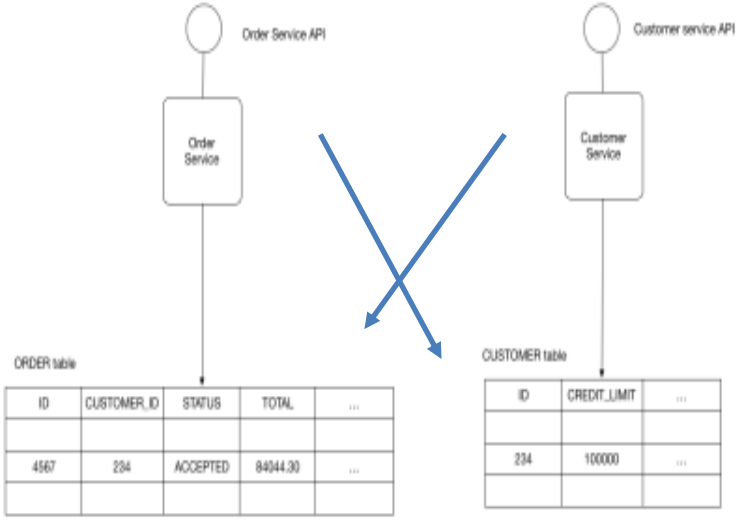
- Shared database
- Database per Service
- Saga
- Event Sourcing
- CQRS (Command Query Responsibility Segregation)
- API Composition

Shared Database

Shared Database
Use a (single) database shared by multiple services. Local ACID transactions.

Benefits:
- Straightforward ACID transactions to enforce data consistency
- Single database is simpler to operate

Drawbacks:
- Since all services access the same database, interference and transaction locks on the table.
- Single database might not satisfy the data storage and requirements of all services.



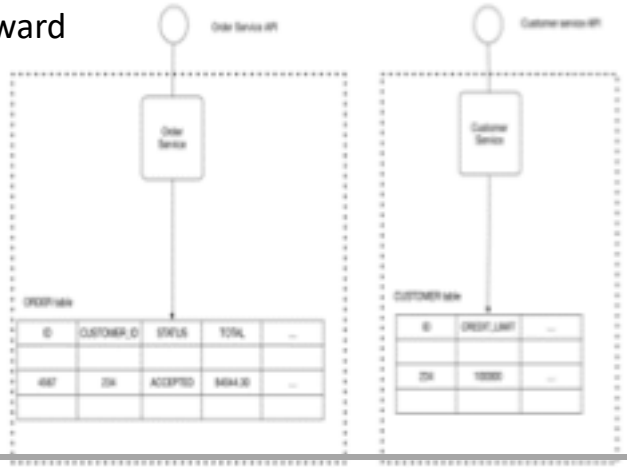
Database Per Service

Database Per Service
Every Microservice has its own persistent data and accessible only via its API
E.g. Private-tables-per-service; Schema-per-service; Database-server-per-service

Benefits:
- Ensures loose coupling
- Each service can use the type of database that is best suited to its needs. E.g., text searches could use ElasticSearch; social graph could use Neo4j.

Drawbacks:
- Transactions involving multiple services not straight forward

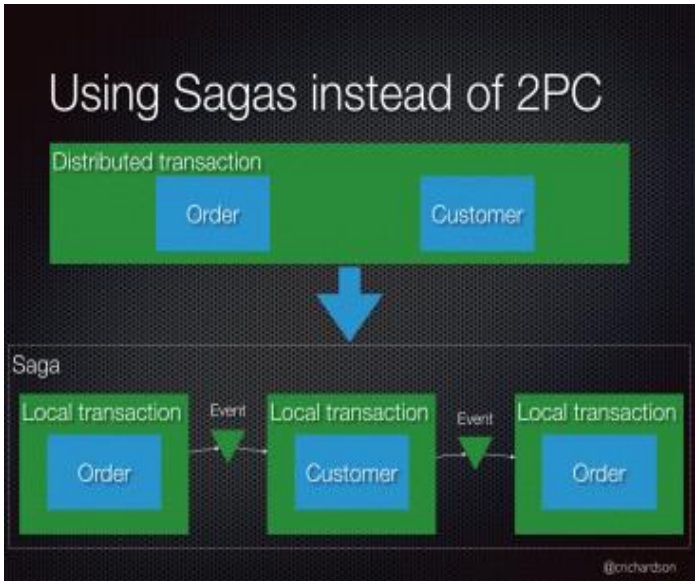
- ❑ For consistency; may need to implement capability to roll back (**Saga** pattern)
- ❑ Interaction is achieved by message/event brokers



Saga Pattern – For Data Consistency

A **Saga** represents a high-level business process (such as booking a trip) that consists of several low-level **Requests** that each update data within a single service.

Each Request has a **Compensating Request** that is executed when the Request fails or the saga is aborted.

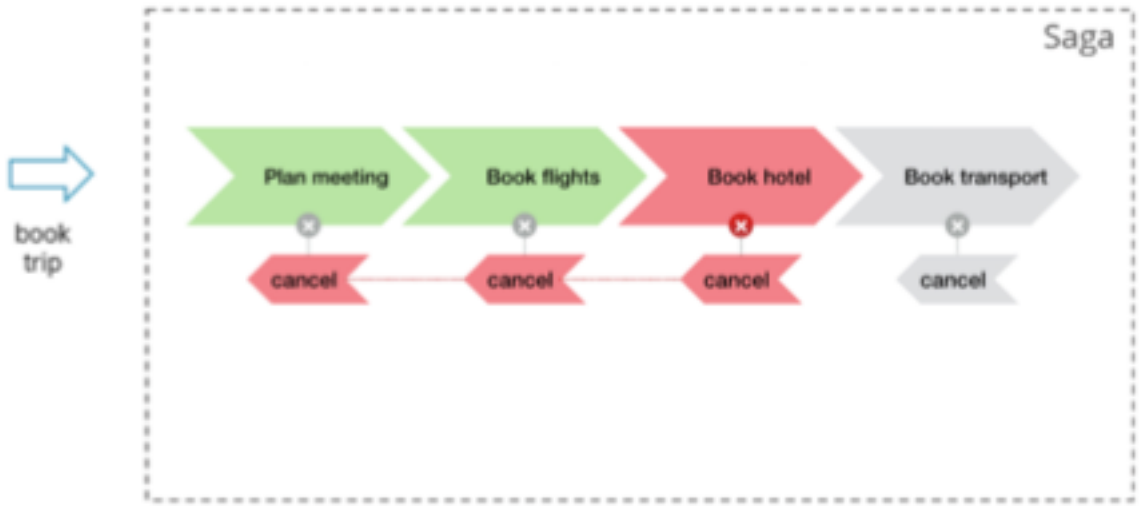


*** How to reliably/atomically publish events whenever state changes? Solution can be **Event Sourcing***

Saga Pattern – For Data Consistency

Saga Pattern is designed for failures and consist of 2 request group

- Collection of request: e.g. {Book Flight}, {Book Hotel}, {Book Transport}
- Collection of compensating request. E.g. {XX, cancel flight, cancel hotel }



Asynchronous request: It may happen that the compensating request arrives before its corresponding request, thus bookhotel arrives later than cancelhotel.

*** How to reliably/atomically publish events whenever state changes?*

*Solution can be **Event Sourcing***

Event Sourcing

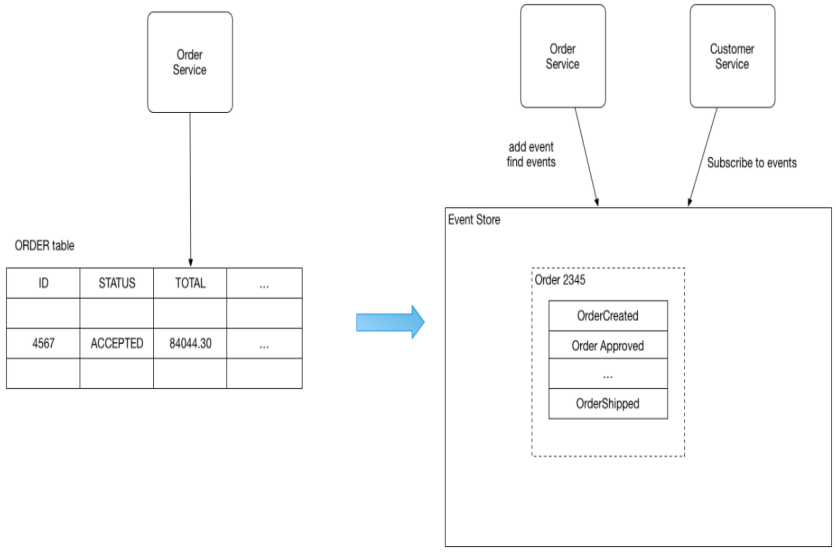
- Persists the state of a business entity as a sequence of state-changing events – in Event Store
- The store has APIs for adding, retrieving an entity's events and subscribe to events.
- The store is like a message broker; A new event is delivered to all interested subscribers.
- E.g. the Order is saved as sequence of events, and customerService subscribe to those events

Benefits:

- reliably publish events whenever state changes

Drawbacks:

- Difficult to design (unconventional)
- The event store is difficult to query since it requires typical queries to reconstruct the state of the business entities - complex and inefficient --- **use CQRS to query**

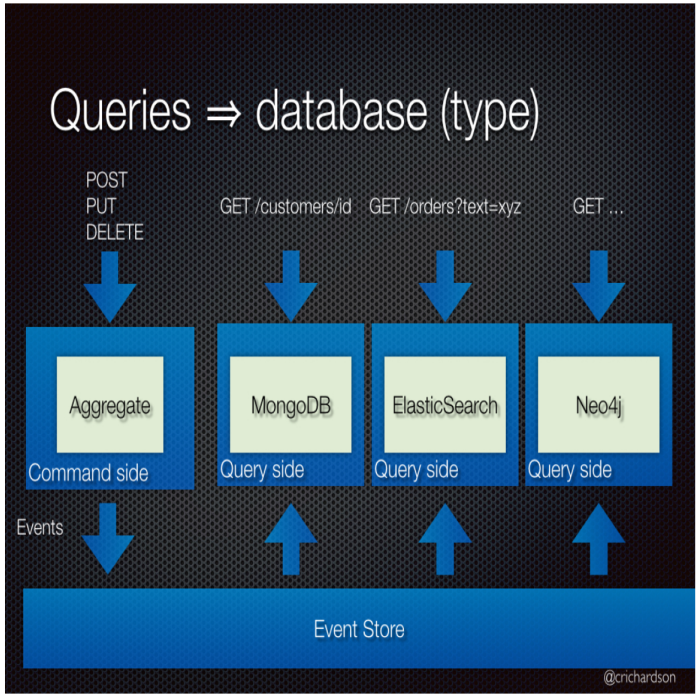


CQRS –
Command Query Responsibility Segregation

Split the application into two parts:

- **Command-side**: handles create, update, and delete requests and emits events when data changes.
- **Query-side:** handles queries by executing them against one or more materialized views that are kept up to date by subscribing to the stream of events emitted when data changes.

A Command cannot return data and
a Query cannot change the data.



Use Case

- Get data from multiple different Aggregates
- Application has an imbalance in responsibility (read/writes)

How to

- Two models talking to same date store (applications that require synchronous processing and immediate results.)
- Two models talking to different data store (Eventual Consistency)

API Composition

Perform queries in Microservices architecture

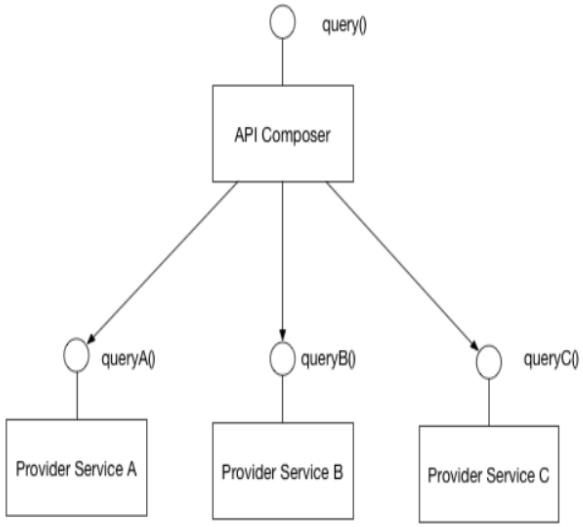
- Implement a query by defining an *API Composer*, which invoking the services that own the data and performs an in-memory join of the results.

Benefits:

- It a simple way to query data in a Microservice architecture

Drawbacks:

- Some queries would result in inefficient, in-memory joins of large datasets.



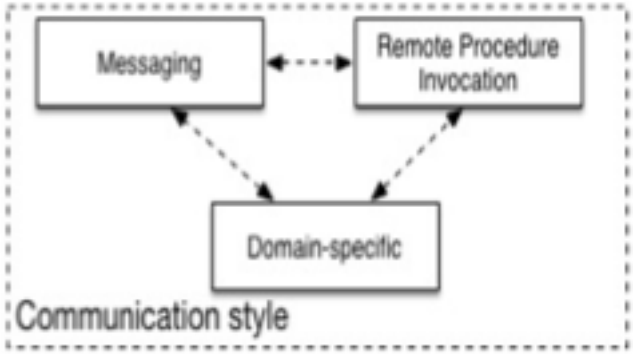
Pattern – Categories

- Data Management
- **Communication**
- Deployment
- Discovery
- Reliability
- Observability
- Testing

Communication Style

Messaging

- Services communicate by sending asynchronous messages
- Decouples client & service
- Must also install and maintain a broker (Kafka, RabbitMQ, etc)
- Complex and should be highly available



Remote Procedure Invocation

- Simple and familiar
- Both client and service must be available
- Only simple request and reply. Notifications, publish/subscribe, async communication not supported
- E.g. Rest, Thrift

Pattern – Categories

- Data Management
- Communication
- **Deployment**
- Discovery
- Reliability
- Observability
- Testing

Deployment Patterns

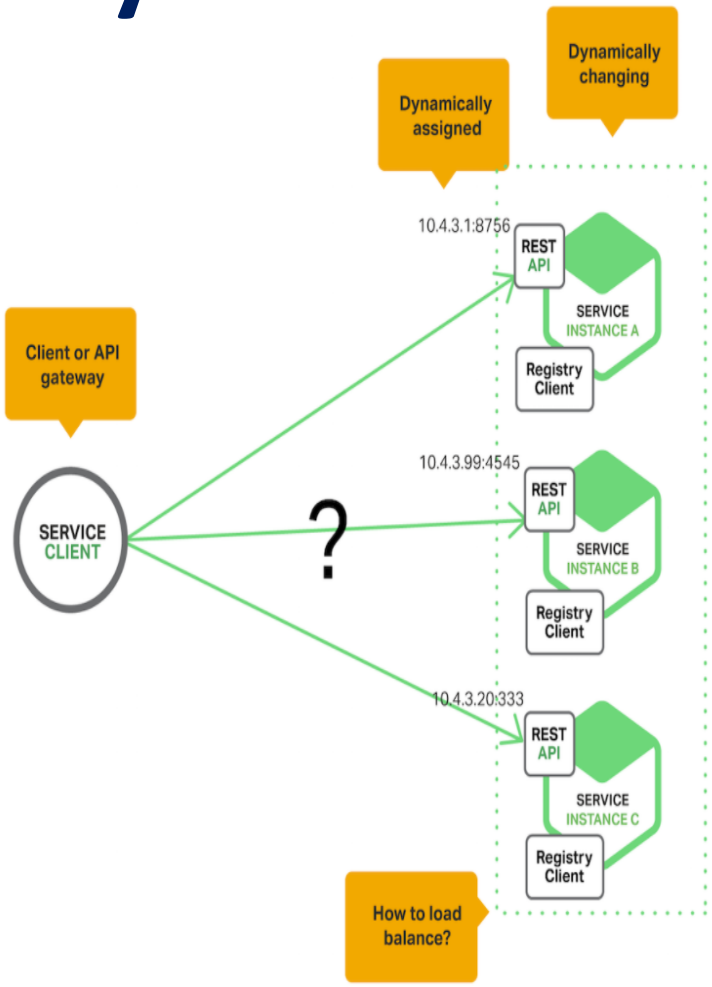
- Service instance per host
- Multiple Service instance per host
- Service instance per VM
- Service instance per container
- Server-less

Pattern – Categories

- Data Management
- Communication
- Deployment
- **Discovery**
- Reliability
- Observability
- Testing

Service Discovery

- Service instances have dynamically assigned network locations. Moreover, the set of service instances changes dynamically because of autoscaling, failures, and upgrades. Consequently, the client code needs to use a more elaborate service discovery mechanism.



Service Registry

- Key of service discovery
  - Database containing the network locations of service instances.
  - Needs to be highly available and up to date.
  - Clients can cache network locations obtained from the service registry -- that information eventually becomes out of date
  - A service registry consists of a cluster of servers that use a replication protocol to maintain consistency.

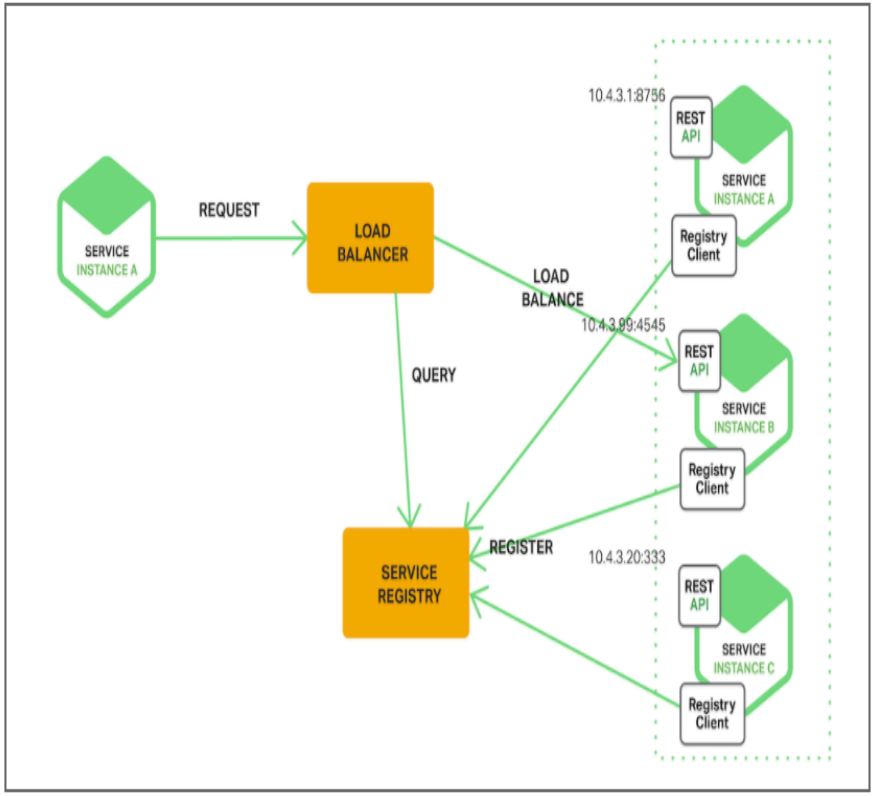
  E.g. Netflix Eureka, consul, Zookeeper, etcd

  https://github.com/hashicorp/consul-template -- dynamically reconfigure NGINX nginx.conf

Service Discovery Pattern

The client makes a request to a service via a load balancer. The load balancer queries the service registry and routes each request to an available service instance. As with client-side discovery, service instances are registered and deregistered with the service registry.

E.g. Kubernetes, Mesos, AWS Elastic LoadBalancer



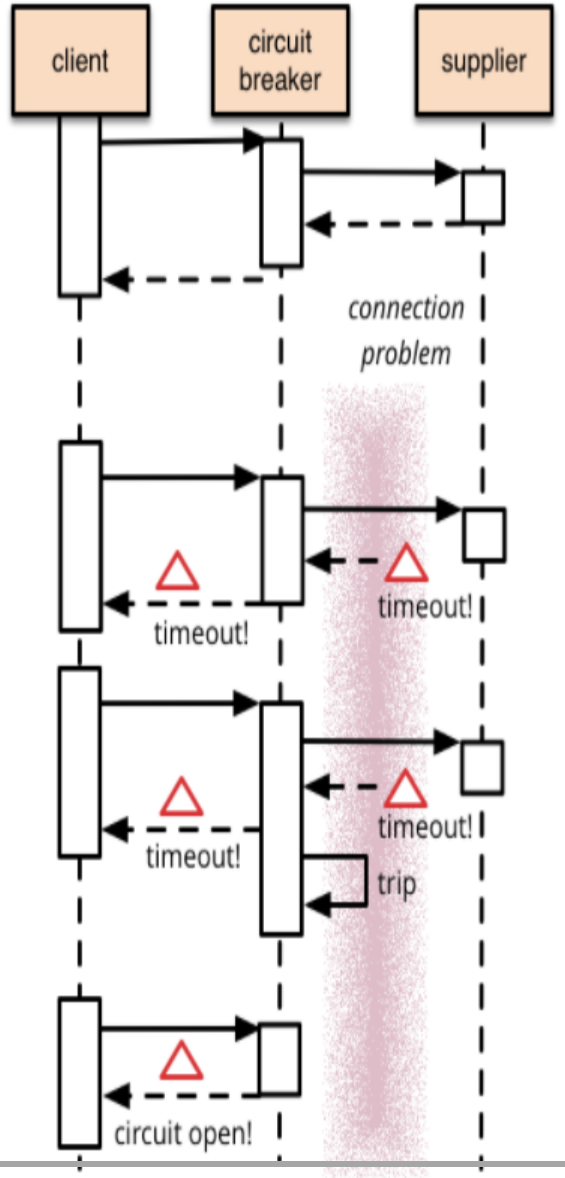
Pattern – Categories

- Data Management
- Communication
- Deployment
- Discovery
- **Reliability**
- Observability
- Testing

Circuit Breaker

Reason:
Remote calls can fail, reach timeout due to unresponsive supplier.
Increased number of calls may lead to cascading failures across multiple systems

What is it:
Wrap a protected function call in a circuit breaker object, which monitors for failures. Once the failures reach a certain threshold, the circuit breaker trips, and all further calls to the circuit breaker return with an error, without the protected call being made at all.



Pattern – Categories

- Data Management
- Communication
- Deployment
- Discovery
- Reliability
- **Observability**
- Testing

Observability Patterns

- Application Metrics
- Health check API
- Distributed Tracing
- Exception Tracking
- Log aggregation
- Audit Logging

Observability Metrics

- **USE** (Utilization, Saturation, Error) method (Brendan Gregg)
- For every resource, check
  - Utilization: average time that the resource was busy servicing work
    - Example: One disk is running at 90%
  - Saturation: the degree to which the resource has extra work which it can't service, often queued
    - Example: The CPUs have an average run queue length of four
  - Error count (rate): the count of error events
    - Example: This network interface has had fifty late collisions
- Useful for resources such as queues, CPU's, memory, interconnects, etc.

Observability Metrics

RED method (Tom Wilkie)

For every service, check

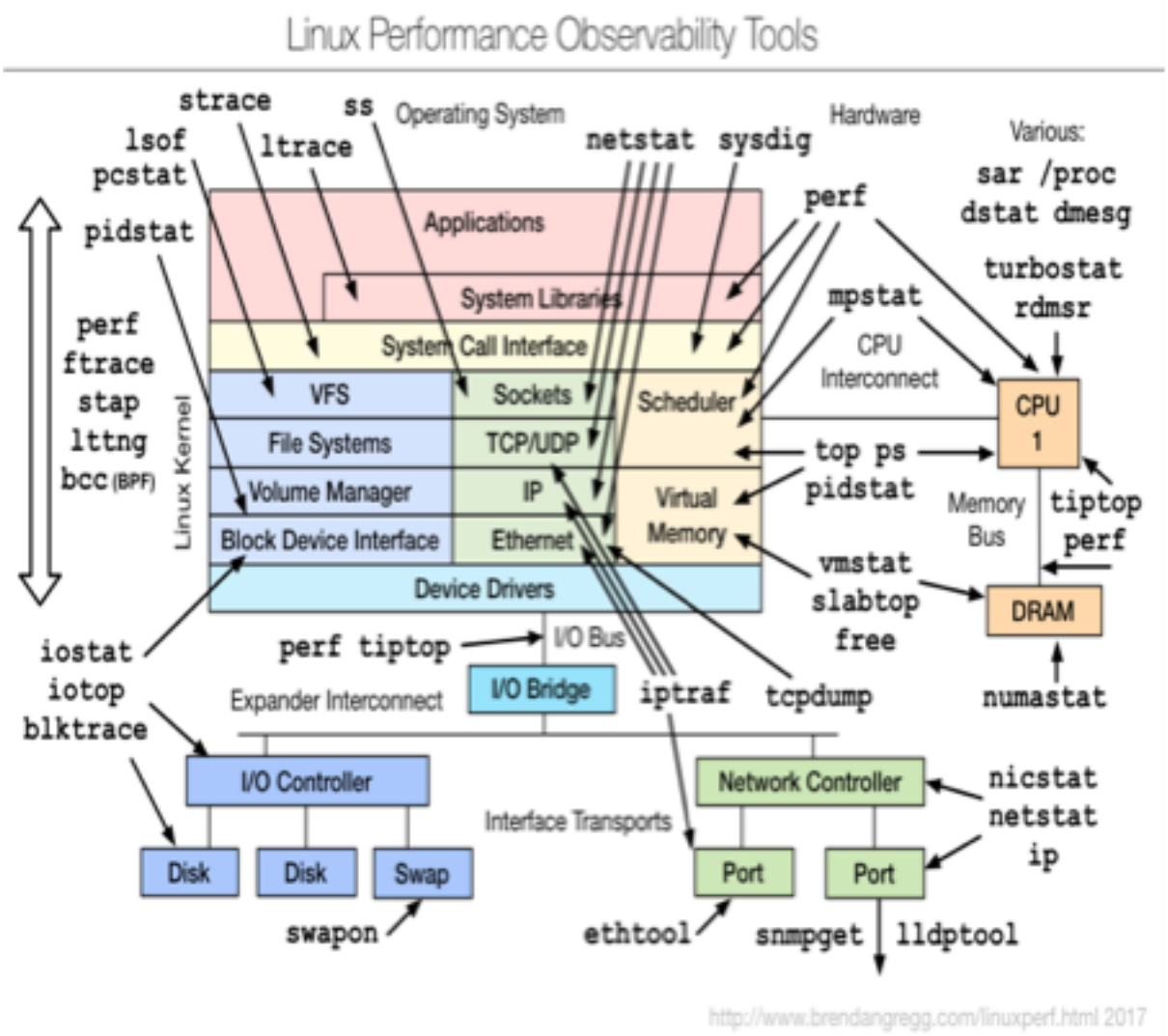
- **Request count** (rate)
- **Error count** (rate)
- **Duration**

Note:

Observability = Logging + Monitoring + Tracing + Visualization

All good instrumentation libraries (Prometheus, Graphite, etc.) have at least three main primitives

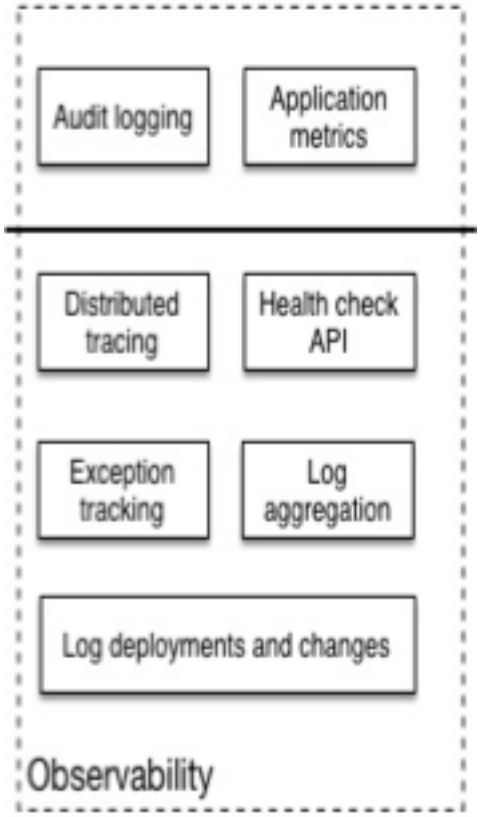
- Counter: records events that happen such as incoming requests, bad requests, errors, etc.
- Gauge: records things that fluctuate over time such as the size of a thread pool
- Histogram: records observations of scalar quantities of events such as request durations



Observability (Exception Tracking)

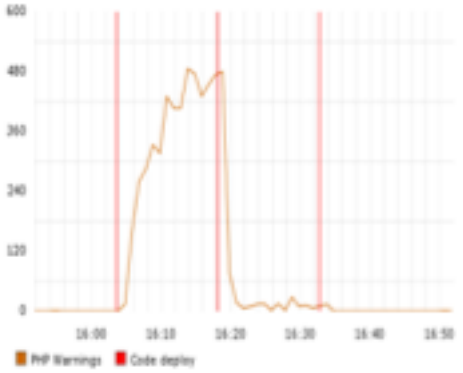
Exception Tracking

- It is crucial to save errors and the corresponding stack trace
- Ideally, exceptions are de-duplicated and recorded for further aggregation and investigation



Log deployments and changes

- Track changes as they are opportunities for failure
- For example, tracking deployments and changes so they can be easily correlated with issues later for faster resolution

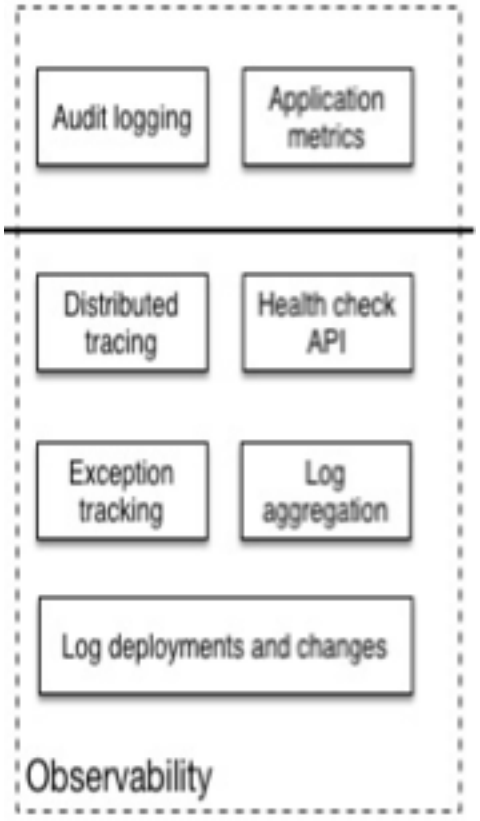


*Etsy graphing error rates against new code deployments

Observability (Aggregation and Audit)

Log Aggregation

- Requests span multiple instances over multiple machines. It is important to gather information in a standardized format
- Log files contain errors, warnings, and debug information
- It is important to aggregate, search and analyze such logs
- Popular solutions for this include the ELK stack (Elastic Search, Logstash and Kibana) and AWS Cloud Watch



Audit Logging

- It is vital to know actions a user has performed.
- For security and compliance, this is often a must have
- Average time to detect intrusions is 98 days for financial services, 197 days for retail, it should/can be much lower

Logging best practices

- Log output as a set of JSON key/value pairs instead of pure text
- Easier for computer to test and aggregate
- Easier to make rules and query
- Comprehensive

- Do this:
  - “Timestamp: 2017-09-01 1:25”, ”caller”:”main.go:5”, “transport”:”HTTP”,” addr”:”8080”
- Not this
  - “2017-09-01 1:25 main.go 5 HTTP 8080”

Pattern – Categories

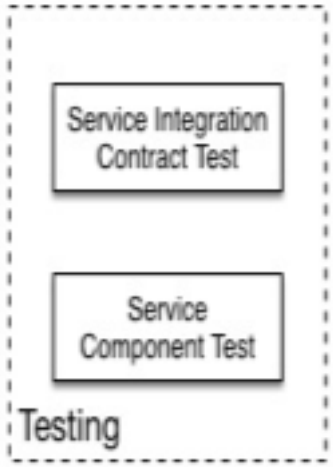
- Data Management
- Communication
- Deployment
- Discovery
- Observability
- **Testing**

Service Integration/Component Testing

Service Integration Tests

- End-to-end testing is slow and expensive
- Test the APIs (functionality) provided by services

- **Black box testing**



Service Component Tests

- Test the components of the service
- A test suite that tests a service in isolation using test doubles for any services that it invokes.

- **A/B testing**

End-To-End tests Vs Unit Test

End-to-end tests: time taking and expensive

- The dev team might have to spend weeks finding all the issues breaking the end-to-end test
- They would be better served with a large base of unit test that find bugs quickly

- ❖ Slow
- ❖ No isolated failures
- ❖ Simulates a real user

Unit Tests: Fast and cheap

- Unit tests should form the majority of tests as they quickly identify bugs and represent a fast feedback loop
- When a Unit test fails, the function and area that fails is much narrower in scope than broad end-to-end tests

- ❖ Fast
- ❖ Isolated Failures
- ❖ But do not simulates a real user