Developing Cloud Services: Background, Concepts & Best Practices

Dr. Graeme Dixon
(graeme@twosigmaiq.com)

Oct 2017

About Two Sigma

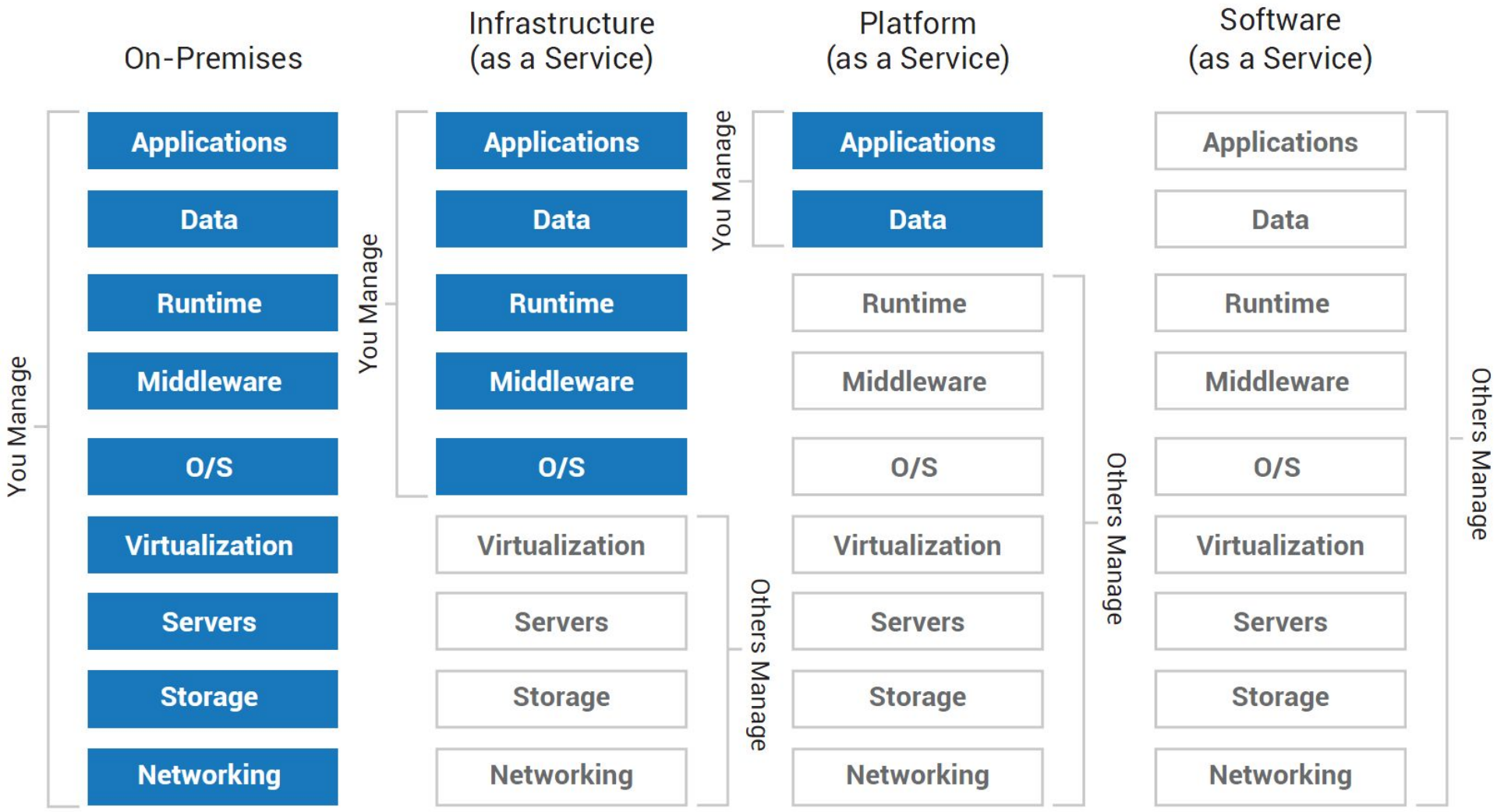
Important Legal Information

The views expressed herein are not necessarily the views of Two Sigma Investments, LP or any of its affiliates (collectively, “Two Sigma”). The information presented herein is only for informational and educational purposes and is not an offer to sell or the solicitation of an offer to buy any securities or other instruments. Additionally, the information is not intended to provide, and should not be relied upon for investment, accounting, legal or tax advice. Two Sigma makes no representations, express or implied, regarding the accuracy or completeness of this information, and you accept all risks in relying on the above information for any purpose whatsoever.

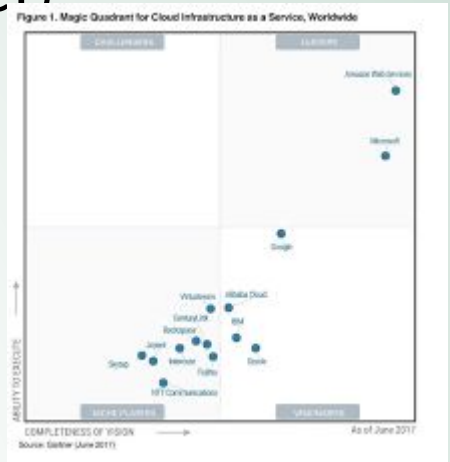
What is Cloud Computing? And where did it come from?

Essential characteristics from NIST(2011):

- *On-demand self-service.*
- *Broad network access.*
- *Resource pooling.*
- *Rapid elasticity.*
- *Measured service.*


On-Premises
Infrastructure (as a Service)
Platform (as a Service)
Software (as a Service)
Applications
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking
You Manage
Others Manage

AWS Dominance of the Cloud (“IaaS” market)



AWS represents ~40% of the total IaaS/PaaS market
- The next ten competitors combine for the next 40%

AWS Origin story
- The fable of excess capacity is not true
- Traditional tech companies remained skeptical beyond 2010

IBM and Microsoft saw the Cloud as a threat to their existing enterprise businesses for way too long
- Margins on the Cloud are a paltry 25% compared to 80% on Enterprise software
- But margins on the Cloud are a massive 25% compared to the paltry 2% on online book sales!

Image source: Gartner

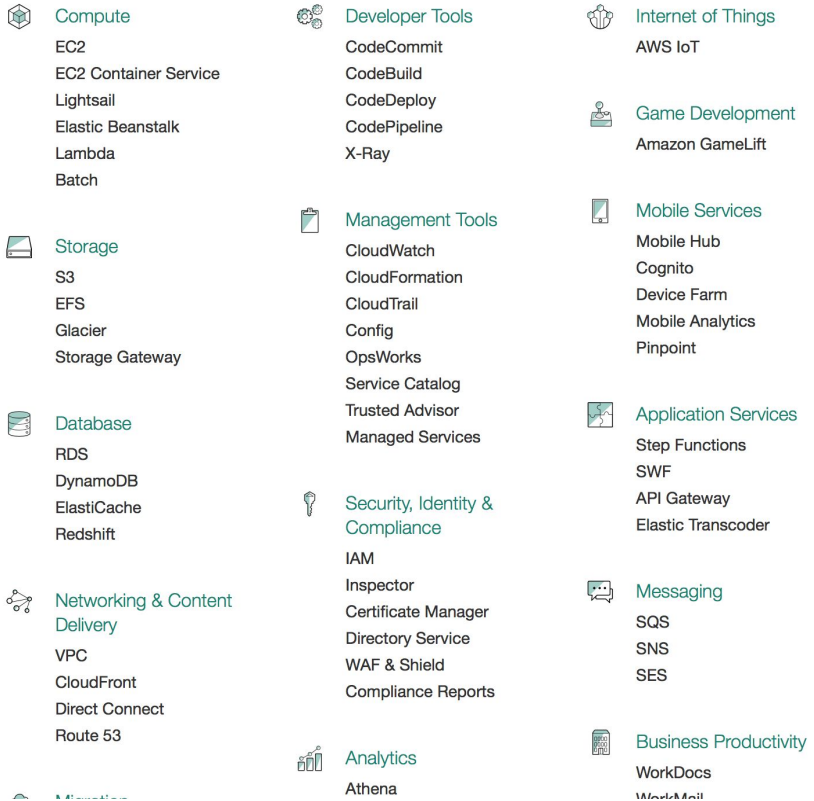
AWS Cloud Scope

Infrastructure Services

Amazon Elastic Compute Cloud
Amazon SimpleDB
Amazon Simple Storage Service
Amazon Simple Queue Service

AWS Premium Support

AWS “Console”
Circa 2007



AWS Console
today

Cloud Native?

- What is cloud native?
  - Cloud native is an approach to building and running applications that fully exploit the advantages of the cloud computing delivery model
  - Typically, using services provided by the public cloud vendors
    - Less to manage, so enables focus on the business problem rather than the plumbing to enable

- But, not everyone has the luxury of developing cloud native solutions
  - Scott Rich next week will describe the challenges and work moving existing services into the public cloud and hybrid cloud solutions

Cloud Native Applications

- Typically mean use of :
  - DevOps - culture and environment where building, testing and releasing software happens rapidly, frequently, and more consistently.
  - CI/CD - is all about shipping small batches of software to production constantly, through automation
  - Microservices - architectural approach to developing an application as a collection of small services; each service implements business capabilities, runs in its own process and communicates via HTTP APIs or messaging
  - Containers - offer both efficiency and speed compared with standard virtual machines (VMs).

- Leverage hosted cloud vendor services, e.g:
  - RDBMS -- AWS Relational Database Service (RDS - "six familiar database engines to choose from, including Amazon Aurora, PostgreSQL, MySQL, MariaDB, Oracle, and Microsoft SQL Server")

Cloud native defn: https://pivotal.io/cloud-native

Advantages of developing in the cloud and...

- Can develop meaningful applications quickly
  - On demand compute, storage and network services along with many key services (databases, messaging etc)

- But, still significant work needed to make robust and supportable (ie. dependable) cloud native applications

- Remainder of this talk covers some best practices to ensure dependable delivery of cloud native applications

Why should you care about Cloud Native and “Best Practices”

- Startups are no longer 2-year hackathons
  - Cloud computing lets you scale quickly but only if designed right
- Evaluate -- a company is only as good as its cloud deployment
  - Know what makes the difference between success and failure
- How to improve
  - Cannot innovate without knowing what is really being done

Some best practices for **dependable** production services delivered in the Cloud

- Design assuming everything can fail
  - And test by inducing failures
- Design for scalability
- Build to/Follow the “Twelve Factor Apps” methodology
- Deploy for availability and use services that support elasticity
  - Exploit Geographic redundancy
- Use an API Gateway

More best practices...

- Services not servers
- Use hosted services were possible
- Deploy with Infrastructure as code
- Implement a Telemetry architecture
- Apply a Security architecture/common isolation techniques

(Most of following examples use AWS -- because this is what we are using and have most experience with. Other cloud vendors provide similar, and in many cases better, alternatives. Not intended to be an endorsement or otherwise of AWS or other cloud vendors.)

Pets vs Cattle

- In a cloud environment, systems should be deployed as cattle rather than pets

- Pets:

  Servers or server pairs that are treated as indispensable or unique systems that can never be down. Typically they are manually built, managed, and "hand fed". Examples include mainframes, solitary servers, HA loadbalancers/firewalls (active/active or active/passive), database systems designed as master/slave (active/passive), and so on.

- Cattle:

  Arrays of more than two servers, that are built using automated tools, and are designed for failure, where no one, two, or even three servers are irreplaceable. Typically, during failure events no human intervention is required as the array exhibits attributes of "routing around failures" by restarting failed servers or replicating data through strategies like triple replication or erasure coding. Examples include web server arrays, multi-master datastores such as Cassandra clusters, multiple racks of gear put together in clusters, and just about anything that is load-balanced and multi-master.

Failure -- To succeed, expect components to crash

- The key to a happy life, always expect the worst and prepare for it…
  - But "Always Look on The Bright Side of Life"

- Cloud computing environments, and delivering services in the cloud, are all failure prone
  - Independently failing components can result in unpredictable behavior, hence system needs to be designed to:
    - Detect
      - Use monitoring tools
    - Tolerate
      - Replicate to mask
    - Throttle
      - No response is better than an incorrect or misleading response

Testing for failures -- Part of design; not an afterthought

- To ensure a cloud implementation delivers predictable service in the presence of failures, the system should be stressed by inducing different flavors of failure
- Good approach is to follow the work of Netflix with their Simian Army:
  - **Chaos Monkey**, a tool that randomly disables our production instances to make sure we can survive this common type of failure without any customer impact
  - **Latency Monkey** induces artificial delays in our RESTful client-server communication layer to simulate service degradation and measures if upstream services respond appropriately
  - **Conformity Monkey** finds instances that don't adhere to best-practices and shuts them down
  - **Doctor Monkey** taps into health checks that run on each instance as well as monitors other external signs of health (e.g. CPU load) to detect unhealthy instances
  - **Janitor Monkey** ensures that our cloud environment is running free of clutter and waste. It searches for unused resources and disposes of them.
  - **Security Monkey** is an extension of Conformity Monkey. It finds security violations or vulnerabilities, such as improperly configured AWS security groups, and terminates the offending instances. It also ensures that all our SSL and DRM certificates are valid and are not coming up for renewal.
  - **10-18 Monkey** (short for Localization-Internationalization, or l10n-i18n) detects configuration and run time problems in instances serving customers in multiple geographic regions, using different languages and character sets.
  - **Chaos Gorilla** is similar to Chaos Monkey, but simulates an outage of an entire Amazon availability zone. We want to verify that our services automatically re-balance to the functional availability zones without user-visible impact or manual intervention.

Anticipating Failures

- Support the ability to use Release Canaries
- With replicated services, system should be developed to allow new versions of a service to be introduced and a subset of the incoming requests routed to test the updated service
- Rollback, rollback, **rollback!**
  Have a plan and support to rollback updates

Scalability

- Not unique to cloud, but critical to success in the cloud
  - Scale out not up
  - Deploy stateless servers (see 12-factor apps)

- Replicate services
  - Route and distribute load across replicas (push model)
    - Good support in cloud for load balancing
    - E.g. AWS Elastic load balancer (ELB), ngnix
  - Queue work and replicas retrieve (pull model)
    - Similar support in cloud for queuing incoming work
    - E.g. AWS Simple Queue Service (SQS)

The Twelve Factors

I. Codebase
One codebase tracked in revision control, many deploys

II. Dependencies
Explicitly declare and isolate dependencies

III. Config
Store config in the environment

IV. Backing services
Treat backing services as attached resources

V. Build, release, run
Strictly separate build and run stages

VI. Processes
Execute the app as one or more stateless processes

VII. Port binding
Export services via port binding

VIII. Concurrency
Scale out via the process model

IX. Disposability
Maximize robustness with fast startup and graceful shutdown

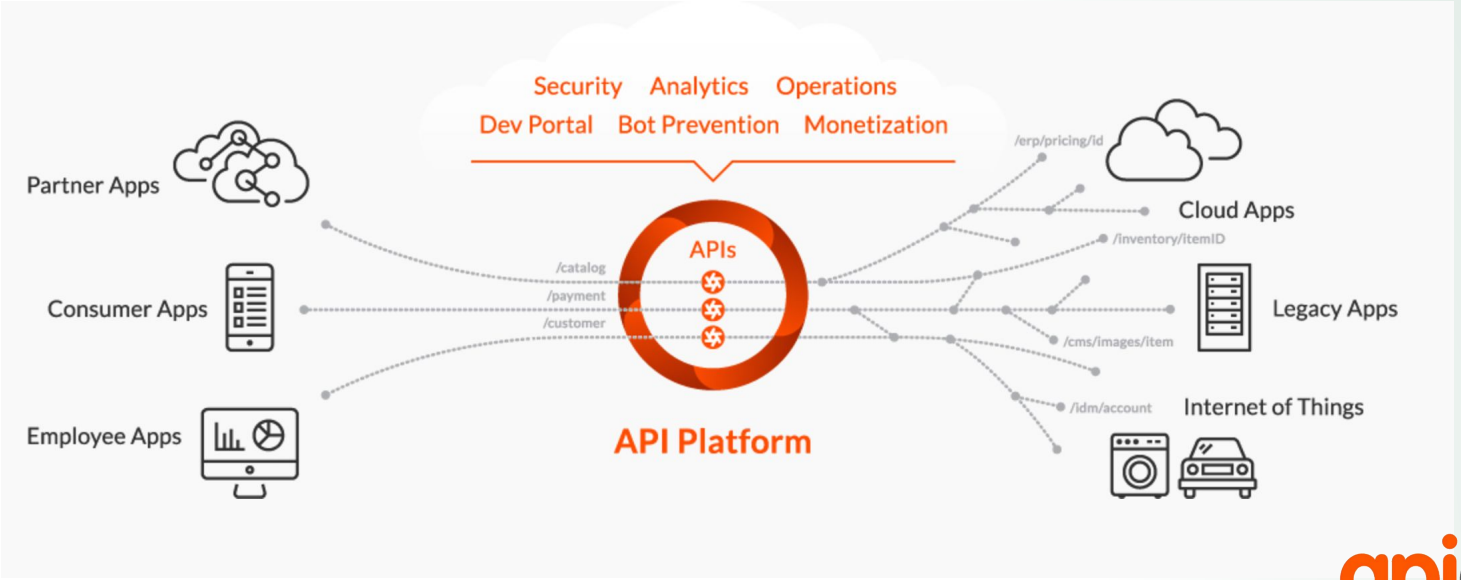
X. Dev/prod parity
Keep development, staging, and production as similar as possible

XI. Logs
Treat logs as event streams

XII. Admin processes
Run admin/management tasks as one-off processes

API Gateways

Use API Gateways



apigee

Elasticity

- **Elasticity** is defined as "the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible".

- Example -- AWS EC2 auto-scaling
  - Ability to define “auto-scaling groups” and with load balancer add new EC2 instances in response to demand

- Interesting challenges are when to add/remove resources
  - Ideally need semantic knowledge of workload/service type
    - AWS released Blox for EC2 Container Service

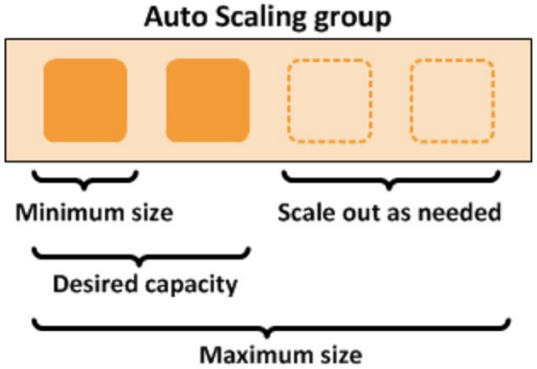


Image source: AWS

Redundancy

- For high-availability, services should be deployed geographically
- Within AWS, each “Region” has multiple “Availability Zones”
  - Availability Zones consist of one or more discrete data centers, each with redundant power, networking and connectivity, housed in separate facilities
- Also possible to deploy in multiple regions and use Route53 (AWS DNS service) to route based in latency of requests

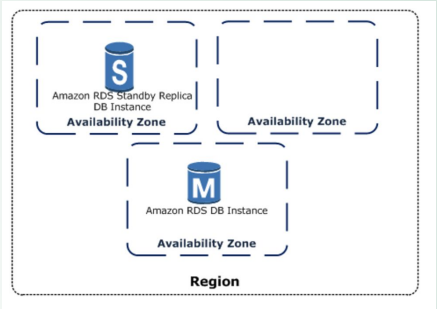


Image source: AWS

Services not Servers

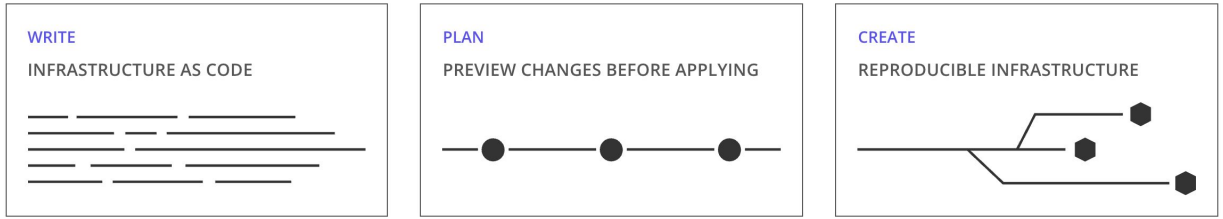
- Again, not cloud-specific, but it is generally good practice to build a “solution” as a collection of cooperating services rather than a large monolithic server
  - Simplifies the lifecycle management
    - The ability to update a service independently simplifies lifecycle management
- Cloud vendors are providing good support for hosting services
  - Serverless application spun up on demand in response to events:
    - Google Cloud Platform Cloud Functions
    - Amazon Web Service Lambda

Hosted Services

- Perhaps obvious….
- With Cloud you can deploy a service within a cloud server and manage yourself, or you can leverage the hosted services that provide that functionality from the cloud vendor
  - E.g. databases, deploy DB server in EC2 instance or use AWS RDS
- The best approach is almost always to use the hosted services from the cloud vendors

Infrastructure as Code

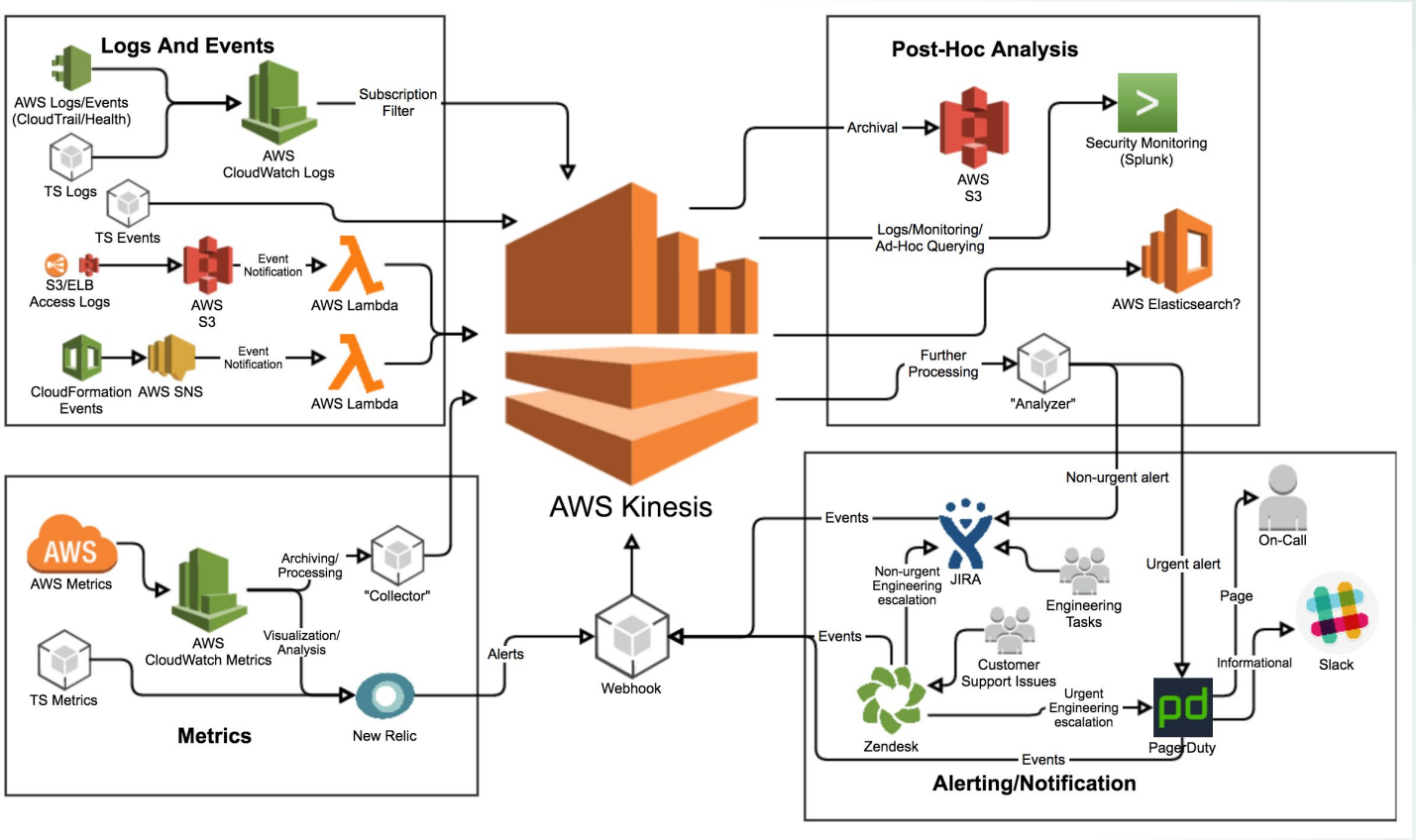
- The ability to be able to quickly reproduce or reconfigure a (likely complex) cloud environment is key and should lead to the use of “infrastructure as code”
- Infrastructure as code means writing code to manage configurations and automate provisioning of infrastructure in addition to deployments.
- This is not simply writing scripts, but involves using tested and proven software development practices that are already being used in application development.
- Good example: Hashicorp Terraform



Telemetry Flows

- Enable support for various telemetry flows:
  - **Logs/Events** - They basically record that something happened and care must be taken to ensure as much as possible that they aren't lost. Where they mainly differ is that Logs are typically unstructured while Events are structured.
  - **Metrics** - Structured data that can be measured over at least one dimension, time, and are used to indicate the behavior of a system. If a high number of metrics are generated then depending on the technology used it might be advantageous to attach a lower SLA to the data such that it is okay if some amount of messages are lost. Typically each message contains three items: timestamp, key, and value.
  - **Alerts** - A message indicating that some action needs to be taken, e.g. a notification or perhaps triggering of some other business process. In a way these can also be considered implementations of Events, e.g. a message indicating that an alert has been raised.
- Since anything and everything can fail it is critical to collect all telemetry for the services deployed in the cloud and the cloud infrastructure itself
  - To enable post-mortem debugging and gradual improvements

Typical Telemetry Architecture



Security

- Security is often the hardest aspect
  - Security configuration is often obscure, complex and difficult
  - Security “experts” even more so

- Building a provably secure system often results in an unusable system

- Some key approaches however:
  - Manage permissions -- apply the principle of least privilege
  - Isolation of environments (dev, staging, prod)
  - Encrypt everything
  - Log everything

Suggested reading

- A View of Cloud Computing
  - http://cacm.acm.org/magazines/2010/4/81493-a-view-of-cloud-computing/fulltext
- NIST definition of Cloud Computing - 2011
  - http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf
- The origin of Amazon Web Services:
  - https://blog.hackerrank.com/how-amazon-web-services-surged-out-of-nowhere/
- https://aws.amazon.com/whitepapers/architecting-for-the-aws-cloud-best-practices/
- http://techblog.netflix.com/2011/07/netflix-simian-army.html
- https://12factor.net/
- https://cloudplatform.googleblog.com/2017/03/how-release-canaries-can-save-your-bacon-CRE-life-lessons.html
- http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle