Zurich University
of Applied Sciences

School of
Engineering
InIT Institute of Applied
Information Technology

CNA1 - Cloud Native Applications

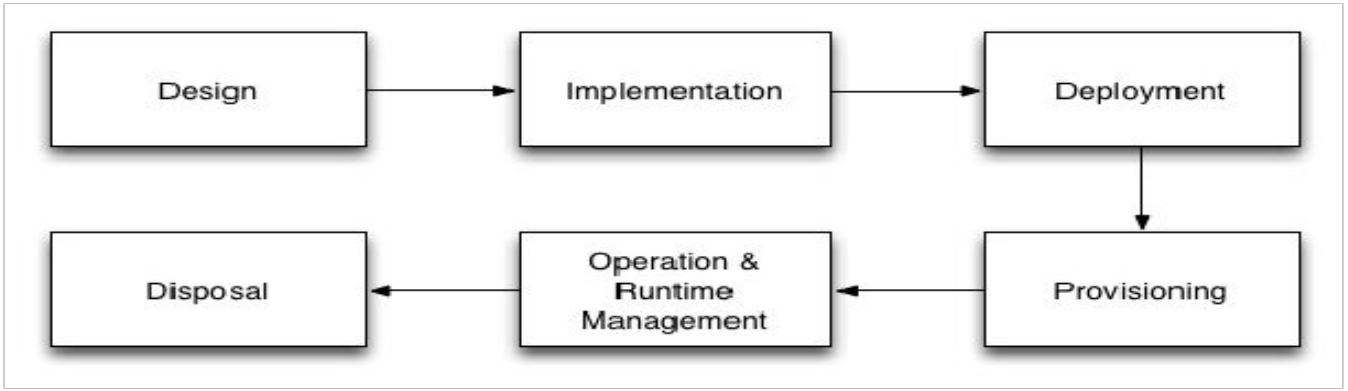
Prof. Dr. Thomas M. Bohnert
Christof Marti

Content

- Cloud-native application
- Service Oriented Architecture / Microservice Architecture
- CNA principles & best practices

Life-cycle of a cloud-application

- Design of the service architecture
- Implementation of the designed solution
- Deployment of the implemented solution & elements



- Destroy service instances
- Activities such as scaling, reconfiguration of Service
- Activation of the service such that the user can actually use it.

Characteristics of cloud-native applications

A cloud-native application is an application **optimized** for running in the cloud (IaaS or PaaS). A cloud-native application exploits all cloud computing principles.

The principal goal of a cloud-native application is to exploit the economic value proposition of cloud computing.

Each phase in the application life-cycle has to be adopted and optimized for running in a cloud-environment.

A cloud-native application is typically designed as a **distributed application**.

It's also possible to get there by **migrating** (re-designing) an already existing application.
→ It's just semantics but cannot be called "cloud-native" in that case

Cloud-native application implementation

- To adopt CNA, the following topics need to be addressed
  - **Architecture**: Applications need to be designed for scalability and resilience
    - → move towards Service Oriented Architecture / Microservices Architecture
    - → use of CNA specific Patterns
  - **Organization**: Teams need to be (re)organized to
    - → Agile teams organized around Business Capabilities
    - → Incorporating DevOps principles and methodologies
  - **Process**: Tools & Technologies needs to be adapted/extended
    - → Automated Software Development / Deployment / Management Pipeline

- Methodologies and guidelines to support Cloud-native application development
  - Service Oriented Architecture (SOA) / Microservices Architecture
  - CNA Principles
  - CNA Patterns

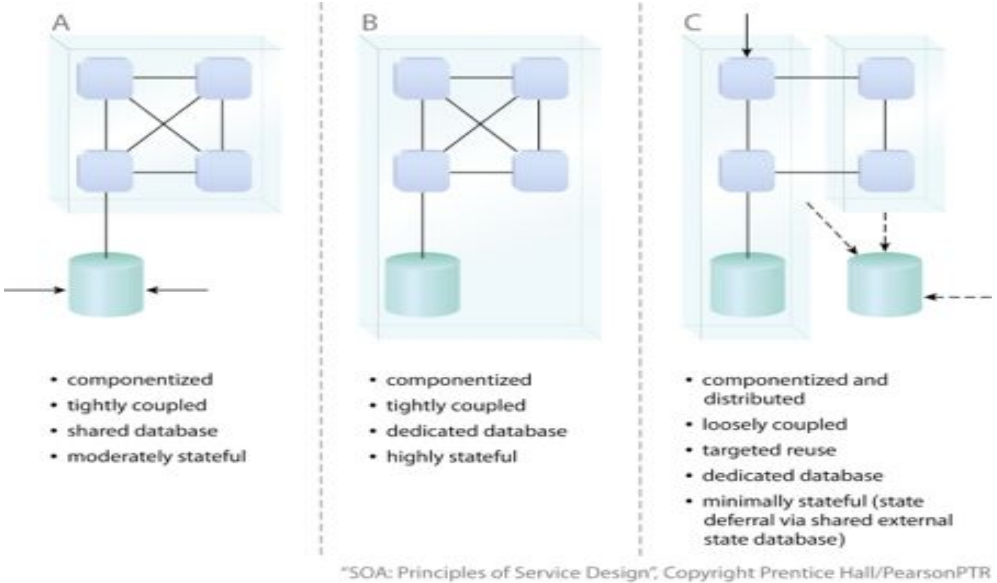
Service Oriented Architecture (SOA)

Open group definition: "Service-Oriented Architecture (SOA) is an architectural style that supports service-orientation. Service-orientation is a way of thinking in the sense of **services** and service-based development and the outcomes of services."

Service: a self-contained unit of functionality that can be accessed in a remote, standardized, technology-independent fashion

SOA Principles:

- Standardized protocols (e.g., SOAP, REST)
- Abstraction (from service implementation)
- Loose coupling
- Reusability
- Composability
- Stateless services
- Discoverable services



"SOA: Principles of Service Design", Copyright Prentice Hall/PearsonPTR

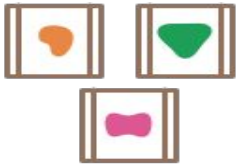
Microservices

- Microservices architecture is an SOA architectural style to develop applications
  - as a **suite of "small" services**,
  - each **running self composed in its own process**
  - and communicating with **lightweight** mechanisms (REST APIs or Messaging).
  - They are **built around business capabilities** following the "do one thing well" principle.
  - They are **independently deployable** in a fully automatic way.
  - There is some **minimal centralized** management of the services.
- The term was coined in 2011 at a software architect's workshop and formalized by Martin Fowler and others in 2014. But emerged much earlier among cloud-oriented companies:
  - Amazon in the early 00s moved from the monolithic Obidos application to a service-oriented architecture.
  - Netflix (whose infrastructure is completely cloud-based) did a major redesign of their system along a microservice architecture.
  - Today any major framework supports the development of microservices and tooling is built with microservices in mind

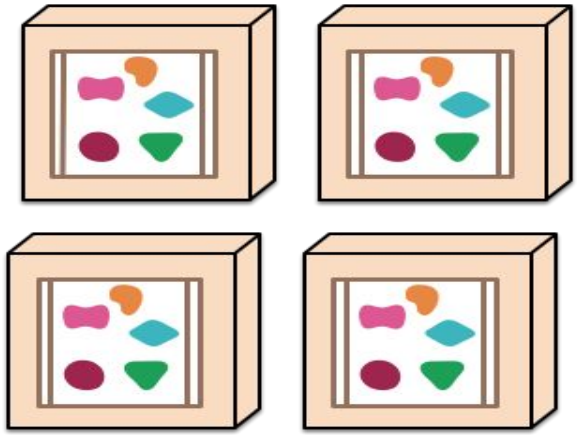
Monolithic vs. Microservices architecture

A monolithic application puts all its functionality into a single process...

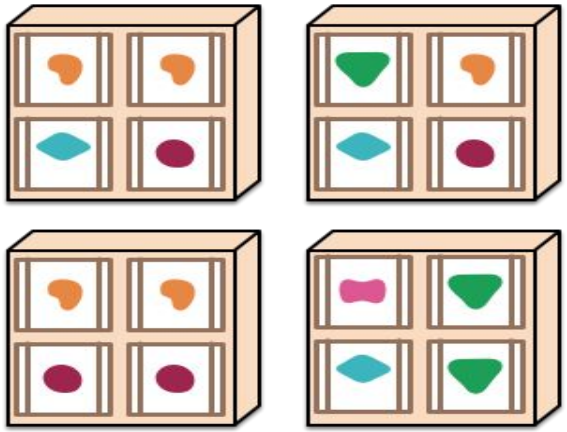
A microservices architecture puts each element of functionality into a separate service...



... and scales by replicating the monolith on multiple servers



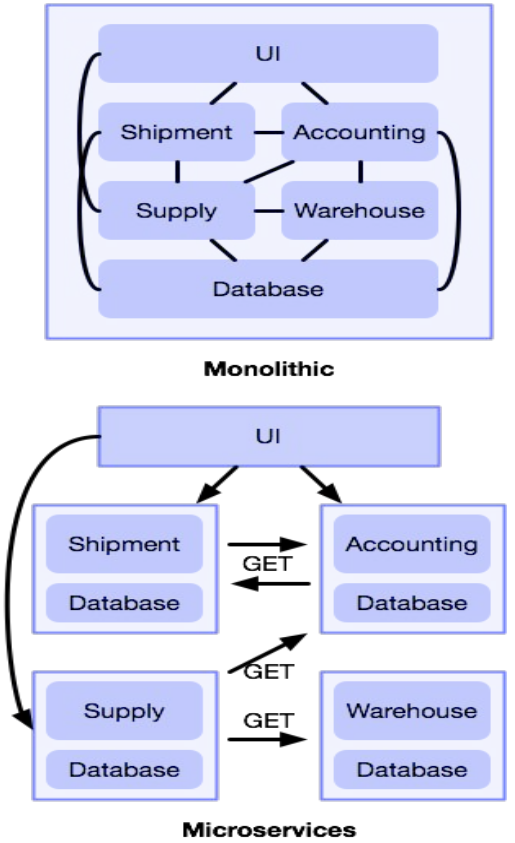
... and scales by distributing these services across servers, replicating as needed.



Source: http://www.martinfowler.com/articles/microservices.html

Componentization around services

- Any well-architected system is based on modular components.
  - Traditionally, components have been encapsulated into libraries.
  - Interfaces are defined using programming language mechanisms.
- (Micro)Service components consist of processes.
  - Services/Processes are accessed over the network.
  - Interface defined by APIs using RPC or messaging mechanisms (e.g. REST APIs, gRPC, AMQP, MQTT).
  - Services are independently deployable (no need to deploy the whole monolith when a component changes).
  - Services evolve independently (Bug fixing, new functionality)
  - Services are independently scalable (able to accommodate individual processing load)
  - Generally, services have individual life-cycles



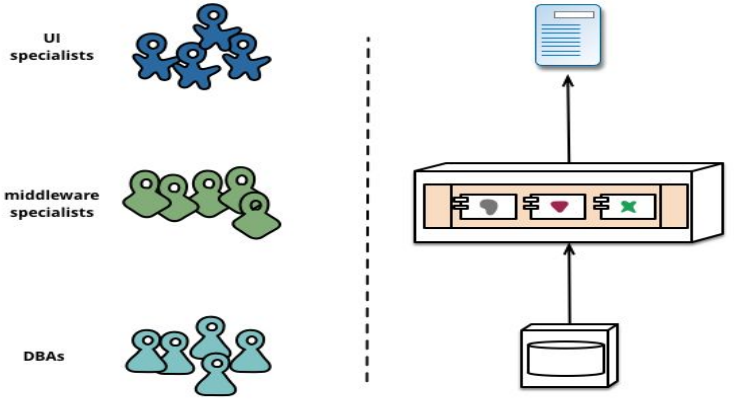
Organized around Business Capabilities

- Conway's law:

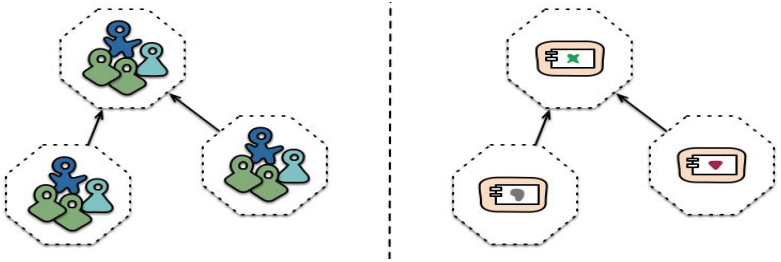
  Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

- Common to see in large organizations, teams split along UI, server-side logic and database (silos).
  - Because simple changes require cross-team approval which is burdensome, each team starts to develop code workarounds.

- In the (micro)service approach, teams are **split along (business) capabilities**.
  - Teams are cross-functional: skills across the whole stack.
  - Team size follows "two pizza" principle: the whole team can be fed by two pizzas.



Siloed functional teams lead to a siloed application architecture.



Cross-functional teams organized around business capabilities

How to design a Cloud-Native Application?

Principles? ⇒ CNA1 (This lecture)

Architecture patterns? ⇒ CNA2 (Next lecture)

CNA Principles: Twelve-Factor Applications

- Collection of principles and best-practices for cloud-native application architectures, originally developed by engineers at Heroku.
- Methodology for building software-as-a-service apps that
  - are suitable for deployment on modern cloud platforms, obviating the need for servers and systems administration
    → support development / deployment workflow
  - can scale without significant changes to tooling, architecture, or development practices.
    → scalability
- Published in 2011
  - http://12factor.net/
  - Old? Established? Proven foundations in Computer Science do not age.

CNA-Principles – Twelve Factor Application

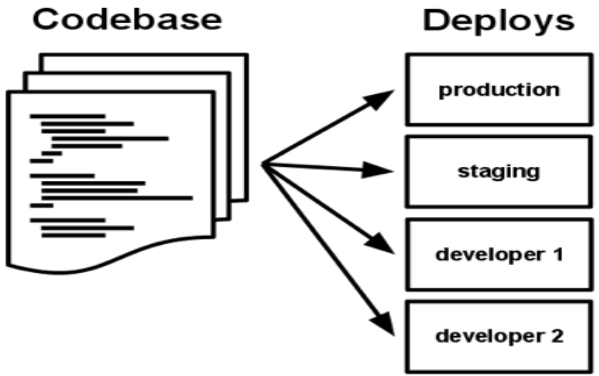
I	Codebase
II	Dependencies
III	Config
IV	Backing services
V	Build, release, run
VI	Processes
VII	Port binding
VIII	Concurrency
IX	Disposability
X	Dev/prod parity
XI	Logs
XII	Admin processes

See blackboard

Twelve Factors – Codebase

One codebase tracked in revision control, many deploys

- A twelve-factor app is always tracked in a version control system (Git, Mercurial, Subversion, …)
- There is only one codebase per app, but there will be many deploys of the app.
- The codebase is the same across all deploys, although different versions may be active in each deploy.



same codebase, but config may be different for each deployment; see Factor III – Config

Twelve Factors – Dependencies

Explicitly declare and isolate dependencies

- A twelve-factor app never relies on implicit existence of system-wide packages.
- It declares all dependencies, completely and exactly, via a **dependency declaration** manifest.
- It uses a **dependency isolation** tool to ensure that no implicit dependencies “leak in” from the surrounding system.
- Advantage of explicit dependency declaration: it simplifies setup for developers new to the app. They just run a build command which will set up everything deterministically.

Twelve Factors – Dependencies

Examples for declaration and isolation dependencies

Programming Languages

- Java: maven pom.xml, build.gradle (dep) & JVM (isolation)
- node.js: npm packages (dep) & npm bundle or nodeenv (isolation)
- Ruby Gemfile (dep) & bundle exec (isolation)
- Python: pip (dep) & virtualenv (isolation)
- C: autoconf (dep) & static linking (isolation)

Generic

- Build automation (Heat, Ansible, ...) & VM images
- Dockerfiles (dep) & Container-Image

⇒ dependency mgmt AND isolation are required (one is not sufficient)

⇒ Do not use wildcard dependencies !!

Twelve Factors – Config

Store config in the environment

- An app's config is everything that is likely to vary between deploys (staging, production, developer environments, etc). This includes:
  - Resource handles to the database, Memcached, and other backing services
  - Credentials to external services such as Amazon S3 or Twitter
  - Per-deploy values such as the canonical hostname for the deploy
- Twelve-factor requires strict separation of config from code and stores config in environment variables.
  - Env vars are easy to change between deploys without changing any code.
  - Unlike config files, there is little chance of them being checked into the code repo accidentally.

Twelve Factors – Config

How and where to set environment

- VM: cloud-init (set when creating the VM)
- docker / docker-compose:
  - Dockerfiles ENV entries
  - Runtime:
    - docker run -e POSTGRES_USER='dbuser' …
    - docker run -env-file prod.env …
- Kubernetes
  - Image (Dockerfile ENV)
  - Deployment / Pod

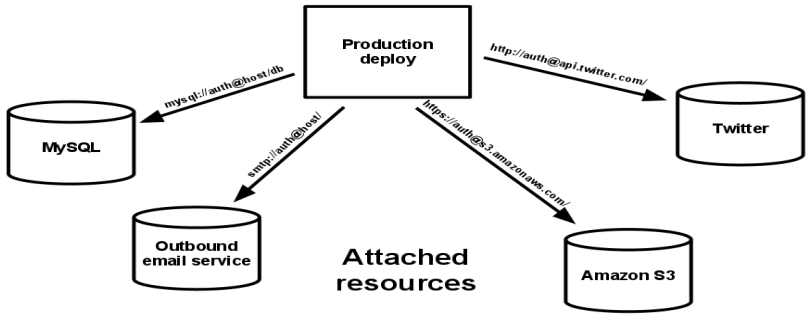
```
spec.template.spec.containers.env:
 - name: POSTGRES_USER
   value: "dbuser"
```

  - configMap & secrets

Twelve Factors – Backing Services

Treat backing services as attached resources

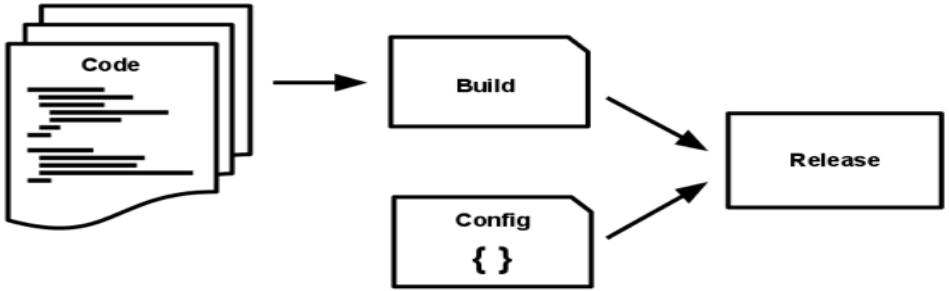
- A backing service is any service the app consumes over the network as part of its normal operation (databases / datastores, messaging / queueing systems, SMTP services, caching, …)
- The code for a twelve-factor app makes no distinction between local and third party services.
  - To the app, both are attached resources, accessed via a URL or other locator/credentials stored in the config.
  - At runtime resources can be attached and detached to deploys at will, without any code changes.



Twelve Factors – Build, release, run

Strictly separate build and run stages

- The twelve-factor app uses strict separation between the build, release, and run stages.
  - For example, it is impossible to make changes to the code at runtime, since there is no way to propagate those changes back to the build stage.
  - Builds are initiated by the app's developers whenever new code is deployed. Runtime execution, by contrast, can happen automatically in cases such as a server reboot, or a crashed process being restarted by the process manager.



Twelve Factors – Processes

Execute the app as one or more stateless processes

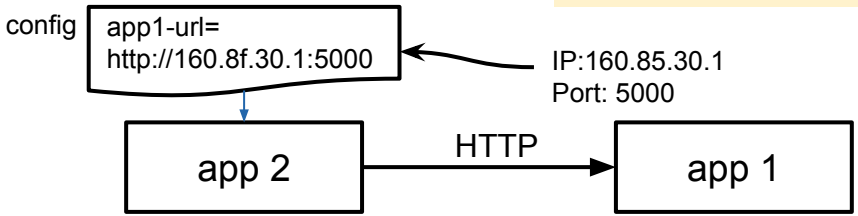
- Twelve-factor processes are *stateless* and *share-nothing*.
  - The twelve-factor app never assumes that anything cached in memory or on disk will be available on a future request.
  - Any data that needs to persist must be stored in a stateful backing service, typically a database.
  - Sticky sessions are a violation of twelve-factor and should never be used or relied upon. Session state data is a good candidate for a datastore that offers time-expiration, such as Memcached or Redis.

Twelve Factors – Port binding

Export services via port binding

- The twelve-factor app is completely *self-contained* and does *not rely on runtime injection of a webserver* into the execution environment to create a web-facing service. The web app exports HTTP*) as a service by binding to a port, and listening to requests coming in on that port.
- One app can become the backing service for another app, by providing the URL to the backing app as a resource handle in the config for the consuming app.

→ Service concept in Kubernetes

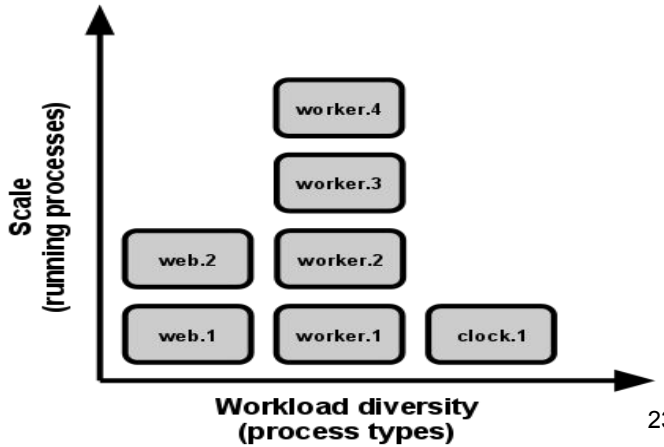


*) Beside HTTP many other protocols can be bound to a port (e.g. AMQP, XMPP or Redis)

Twelve Factors – Concurrency

Scale out via the process model

- Apps handle diverse workloads by assigning each type of work to a process type. For example, HTTP requests may be handled by a web process, and long-running background tasks handled by a worker process.
- Scale out app processes *horizontally and independently* for each process type.
  - The array of process types and number of processes of each type is known as the *process formation*.



Twelve Factors – Disposability

Maximize robustness with fast startup and graceful shutdown

- The twelve-factor app's processes are disposable, meaning they can be started or stopped at a moment's notice.
  - Minimize startup time
  - Shut down gracefully when receiving a termination signal (e.g. SIGTERM)
    - Free resources
    - Unsubscribe from message channel
    - Release Locks, etc.
  - This facilitates fast elastic scaling, rapid deployment of code or config changes, and robustness of production deploys.
- Processes should also be robust against sudden death, in the case of a failure in the underlying hardware.

Twelve Factors – Dev/Prod parity

Keep development, staging, and production as similar as possible

- Continuous delivery and deployment are enabled by keeping development, staging, and production environments as similar as possible.
- Avoid gaps between development and production
  - *Avoid time gap*: Write code and have it deployed hours or even just minutes later
  - *Avoid personnel gap*: developers who wrote code are closely involved in deploying it and watching its behavior in production
  - *Avoid tools gap*: use the same tools in development and production

Twelve Factors – Logs

Treat logs as event streams

- All running processes and backing services produce logs, which are commonly written to a file on disk.
- A twelve-factor app never concerns itself with routing or storage of its log output stream.
  - In staging or production deploys, each process' log stream will be captured by the execution environment, collated together with all other streams from the app, and routed to one or more final destinations for viewing and long-term archival.
- The stream can be sent to a log indexing and analysis system such as Splunk. This allows for great power and flexibility for introspecting an app's behavior over time, including:
  - Finding specific events in the past.
  - Large-scale graphing of trends (such as requests per minute).
  - Active alerting according to user-defined heuristics (such as an alert when the quantity of errors per minute exceeds a certain threshold).

Twelve Factors – Admin processes

Run admin/management tasks as one-off 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.
  - Same release, codebase and config
  - Same dependencies / versions and isolation
- Application should never do such admin tasks in startup phase

Kubernetes - Config Map

- Config-Map is a key-value Kubernetes object, which can be deployed to environments and referenced from other objects (e.g. pods, deployments)
- It can be deployed from a manifest file
  `kubctl apply -f game-config`
- Or it can be created from CLI

  `kubectl create configmap game-demo --from-literal=player_lives="3"`

  `--from-file=config/game.properties`
- Read the config map to file:
  `kubectl get configmap game-demo -o yaml`

game-config.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: game-demo
data:
  # property-like keys; each key maps to a simple value
  player_lives: "3"
  properties_file: "ui.properties"

  # file-like keys
  game.properties: |
    enemy.types=aliens,monsters
    player.maximum-lives=5
```

Kubernetes - Using Config Map

- Config maps can be referenced from deployments or pods as
  - environment variables
  - read only mounted files

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-demo-pod
spec:
  containers:
  - name: demo
    image: alpine
    env: # Define the environment variable
    - name: PLAYER_LIVES
      valueFrom:
        configMapKeyRef:
          name: game-demo    # The ConfigMap name
          key: player_lives  # The key to fetch.
    volumeMounts:  # Will mount /config/game.properties
    - name: config
      mountPath: "/config"
  volumes:         # from game-demo config-map
  - name: config
    configMap:
      name: game-demo
```

Kubernetes - Secrets

- Secrets are very similar to configMaps, but intended for confidential data
  - credentials (passwords, usernames)
  - certificates
- It can be deployed from a manifest file
  `kubctl apply -f db-creds.yaml`
- Or it can be created from CLI

  ```
  kubectl create secret generic db-creds
  --from-literal=db-user=mydbuser
  --from-literal=db-passwd=supersecret
  ```
- Read and decode the secret:

  ```
  kubectl get secret db-creds -o
  jsonpath="{.data.db-user}" | base64 --decode
  ```

db-creds.yaml

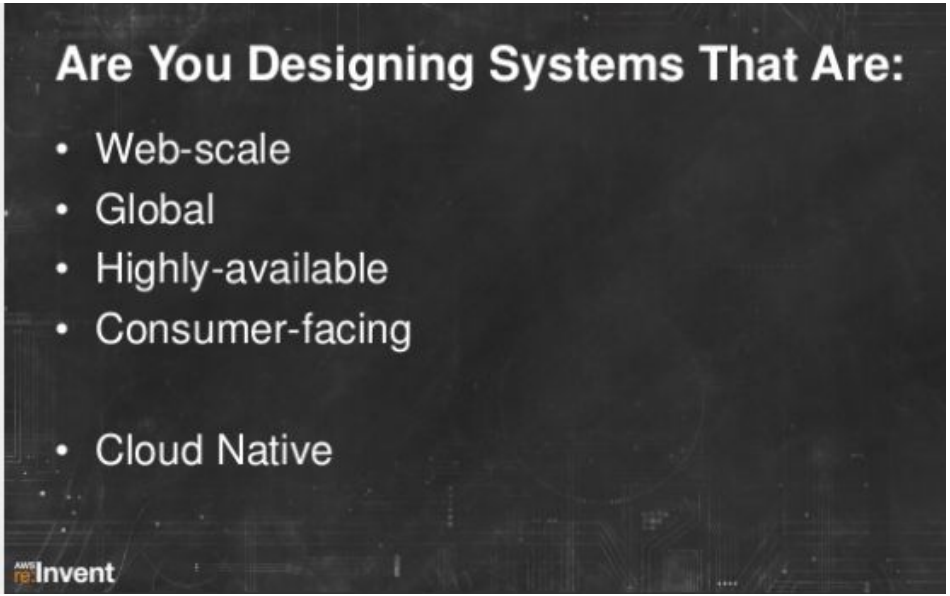
```
apiVersion: v1
kind: Secret
metadata:
  name: db-cred
type: Opaque
data:
  db-user: ZGJ1c2VyCg==
  db-passwd: ZGJwYXNzCg==
```

base64 encoded

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-demo-pod
spec:
  containers:
  - name: demo
    env:
    - name: DB_USER
      valueFrom:
        secretKeyRef:
          key: db-user
          name: db-cred
    - name: DB_PASSWD...
```

Appendix

Netflix's Definition of Cloud Native (2013)

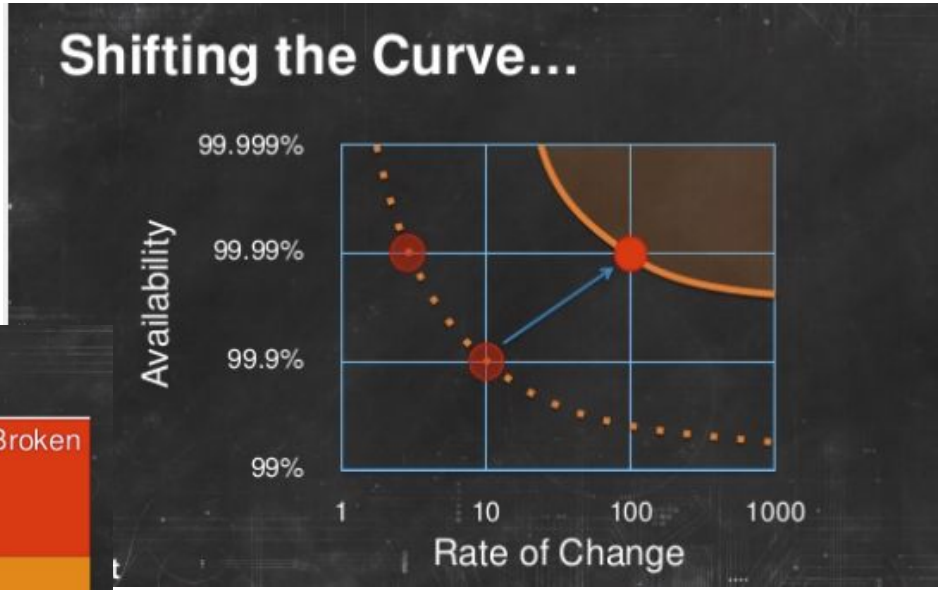


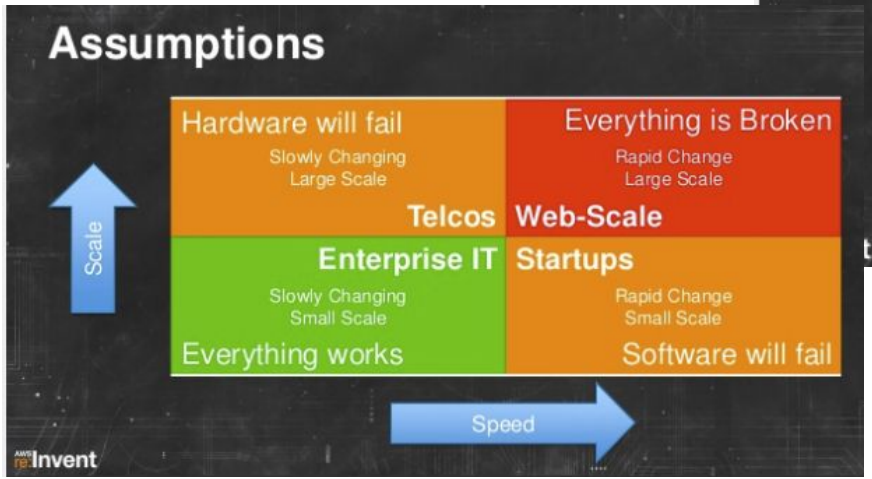
https://www.slideshare.net/AmazonWebServices/dmg206

Netflix's Definition of Cloud Native (2013)

Goal to optimize for:

- Availability
- Scale
- Performance
- **Rate of Change**





https://www.slideshare.net/AmazonWebServices/dmg206