Everything you should know about Microservices

IBM **Developer**
Workshops

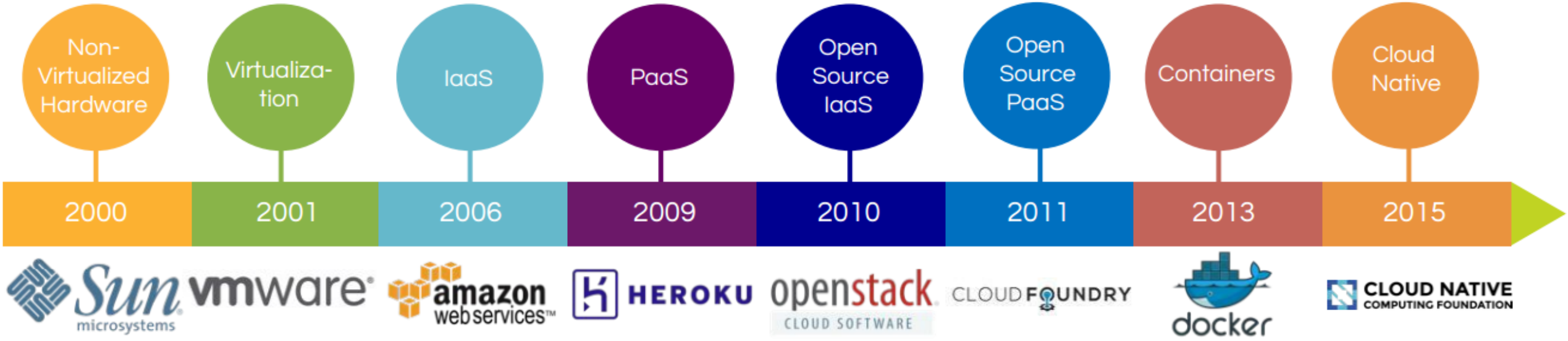
Tal Neeman
Developer Advocate, IBM Alpha Zone
talne@il.ibm.com

Agenda

- CNCF – What happens in the cloud
- Maturity for cloud
- We are ready for Microservices
- Some challenges
- Best practices
- Architecture Pattern
- Q&A

A long line of game changers – **Cloud Native Computing Foundation (CNCF)**

- Founded Dec. 2015
- Part of the Linux Foundation
- Major projects – Kuberenetes, Prometheus



CNCF members

Hybrid Cloud Applications

Monolithic Apps

- Not designed for the Cloud (pre-Cloud era)
- Application and infrastructure tightly coupled,

Cloud Ready Apps

- Modular components (SOA) allowing decoupling of application and infrastructure.
- Components can be scaled separately.
- Can be deployed as standard images or patterns, benefiting from automation.

Cloud Native Apps

- Highly independent self-contained micro modules.
- Highly portable, unaware of infrastructure.
- Event driven and resource aware.
- Scaled horizontally based on events.

Customization; higher costs; slower time to value

Standardization; lower costs; faster time to value

Cloud native maturity model

Cloud Native

- Micro
- API

Cloud Resilient

- Faul
- Metr
- Proa

Cloud Friendly

- 12 F
- Horiz
- High

Cloud Ready

- No p
- Self-
- Platf
- Cons

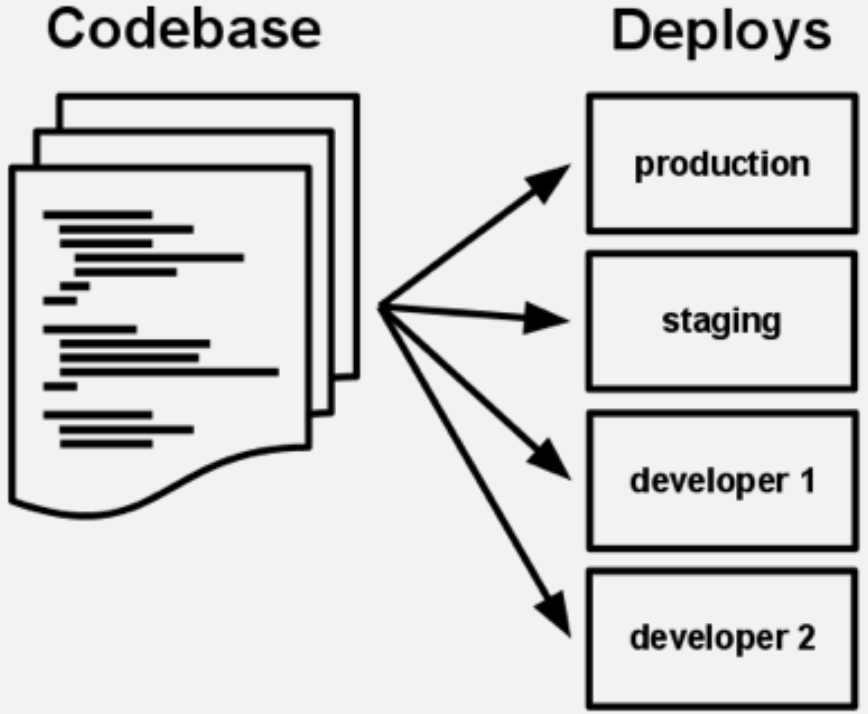
Twelve-Factor App methodology elements

1. **Codebase**: Track one codebase in revision control and many deployments.
2. **Dependencies**: Explicitly declare and isolate dependencies.
3. **Configuration:** Store the configuration in the environment.
4. **Backing services**: Treat backing services as attached resources.
5. **Build, release, and run:** Strictly separate build and run stages.
6. **Processes:** Execute an app as one or more stateless processes.
7. **Port binding**: Export services through port binding.
8. **Concurrency**: Scale out through the process model.
9. **Disposability**: Maximize robustness with fast startup and graceful shutdown.
10. **Development and production parity**: Keep development, staging, and production as similar as possible.
11. **Logs**: Treat logs as event streams.
12. **Administrative processes:** Run administrative and management tasks as one-off processes.

Factor 1: Codebase

I. **Codebase**
II. Dependencies
III. Configuration
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

- One codebase tracked in source code management (SCM) with versioning
  - git & github most commonly used
- Multiple deployments from the same codebase



Factor 2: Dependencies

I. Codebase
II. **Dependencies**
III. Configuration
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

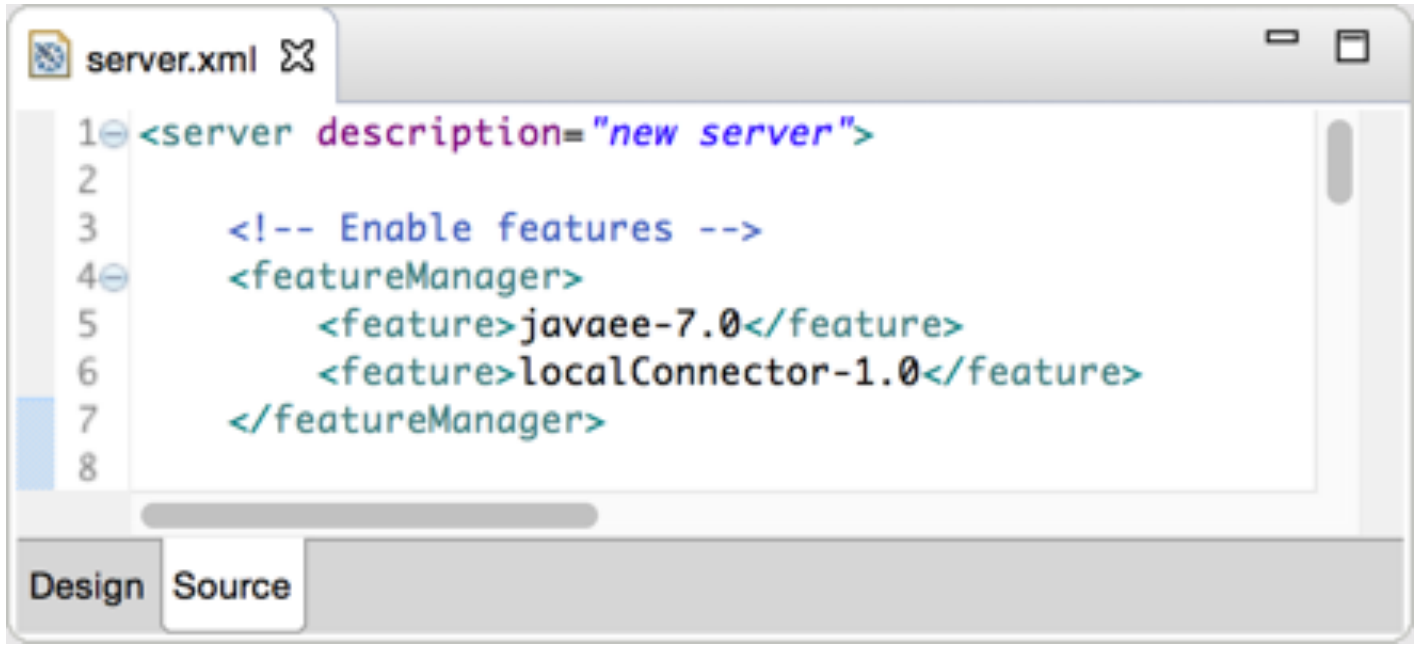
- Explicitly declare and isolate dependencies
- Typically language-specific
  - Node.js: Node Package Manager (NPM)
  - Liberty: Feature manager
  - Ruby: Bundler
  - Java EE: Application resources
- Never rely on system-wide dependencies
- Uses dependency isolation during execution
  - Maven or similar

Dependencies: npm package file

- Application configuration is part of the application
- You manage it in source control

Dependencies: Liberty feature manager

- Server configuration is part of deploying the application
- You manage it in source control



Factor 3: Configuration

I. Codebase
II. Dependencies
III. **Configuration**
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

- Store configuration in the environment
- Separate configuration from source
- Enables the same code to be deployed to different environments
- Configuration includes:
  - Resource handles to database, Memcached, other backing services
  - Credentials to external services such as Amazon S3 or Twitter
  - Per-deploy values such as canonical host name for deploy

Kubernetes configuration

- Kubernetes uses ConfigMap and Secret resources

```
kubectl create secret generic apikey --from-literal=API_KEY=123–456
```

```
kubectl create configmap language --from-literal=LANGUAGE=English
```

Factor 4: Backing services

I. Codebase
II. Dependencies
III. Configuration
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

- Treat backing services as attached resources:
  - Databases
  - Messaging systems
  - LDAP servers
  - Others
- Local and remote resources should be treated identically

  Possible locations for run time and resources:
  - In the same process
  - On the same host
  - On different hosts in the same data center
  - In different data centers

Backing services: Kubernetes bindings

- Bind a container in Kubernetes to an IBM Cloud service
  - Add a volume to the pod definition that stores the binding in a secret
  - By mounting the Kubernetes secret as a volume to your deployment, you make the IBM Cloud service credentials available to the container that is running in your pod.

- Example: Bind to an instance of Watson Tone Analyzer named mytoneanalyzer

```
volumes:
      - name: service-bind-volume
        secret:
            defaultMode: 420
            secretName: binding-mytoneanalyzer
```

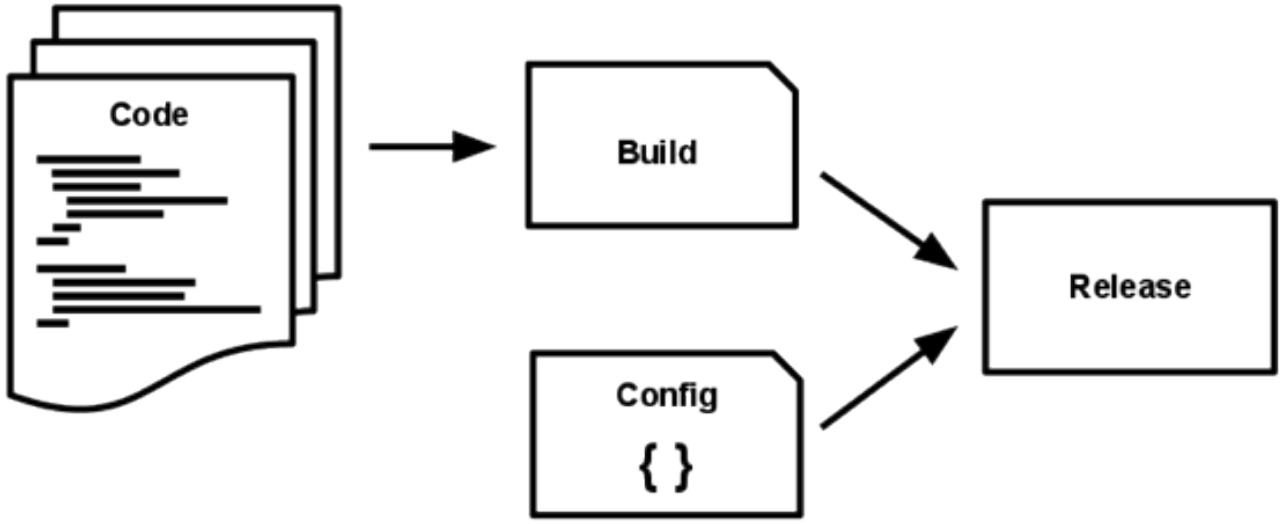
Factor 5: Build, release, run

I. Codebase
II. Dependencies
III. Configuration
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

- Strictly separate build, release and run stages

Build, release, run

- Build stage
  - converts a code repo into an executable bundle known as a build
  - fetches dependencies and compiles binaries and assets
- Release stage
  - takes the build produced by the build stage and combines it with the deploy's current config
  - ready for immediate execution in the execution environment
- Run stage
  - runs the app in the execution environment



Source: https://12factor.net

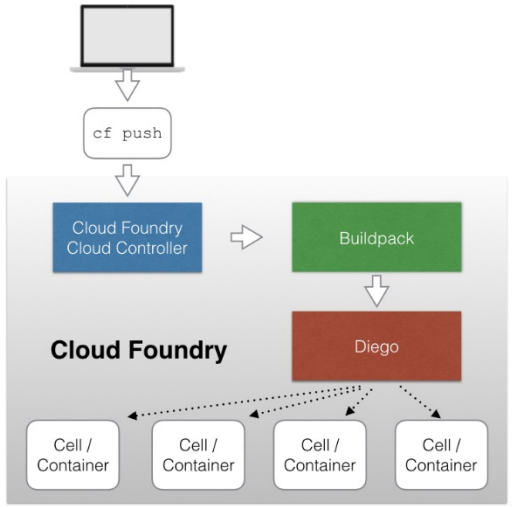
Build & release produce an Immutable image

- An immutable image does not get changed – only used for deploying instances
- If it needs to change, you delete it and create a new one

Examples

- Docker containers
  - A container image is created from a Dockerfile
  - After that, it is only deployed as a container, not changed
  - If you need to make changes, make a new container image by creating a new build and running the Dockerfile again

- Cloud Foundry
  - When an app is pushed to Cloud Foundry, the buildpack creates a droplet – from that CF creates Garden container instances that run the app
  - If you need to make changes, create a new build and push it



Factor 6: Processes

I. Codebase
II. Dependencies
III. Configuration
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

- Run the application as one or more stateless processes
- Do not rely on session affinity, also called sticky sessions

Processes: Stateless application

Data for history of a user session
Scope: Session
Example: HTTP session

Session State

Service State

Data passed into the stateless service
Scope: Transaction or unit of work
Example: Service operation parameters

Runtime 1

Data available to multiple or all services
Scope: Global
Example: Enterprise database of record

Domain State

Factor 7: Port binding

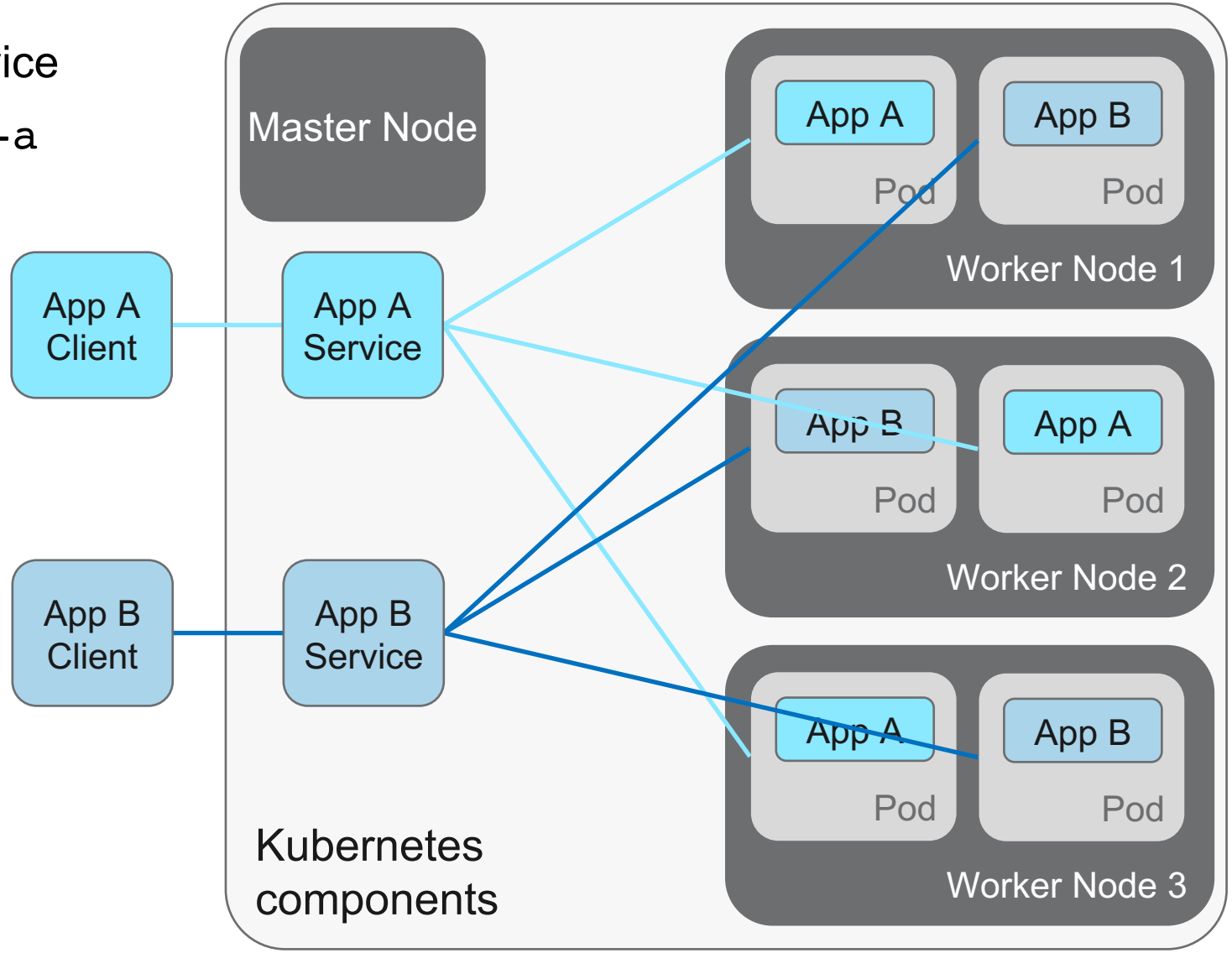
I. Codebase
II. Dependencies
III. Configuration
IV. Backing services
V. Build, release, run
VI. Processes
VII. Port binding
VIII. Concurrency
IX. Disposability
X. Dev/prod parity
XI. Logs
XII. Admin processes

- Export services with port binding
- Web app binds to an HTTP port and listens for requests coming in on that port

Port binding: Kubernetes service

- Expose a set of pod replicas as a service

```
kubectl expose deployment/app-a
        --type=LoadBalancer
        --port=8080
        --name=app-a-service
        --target-port=8080
```

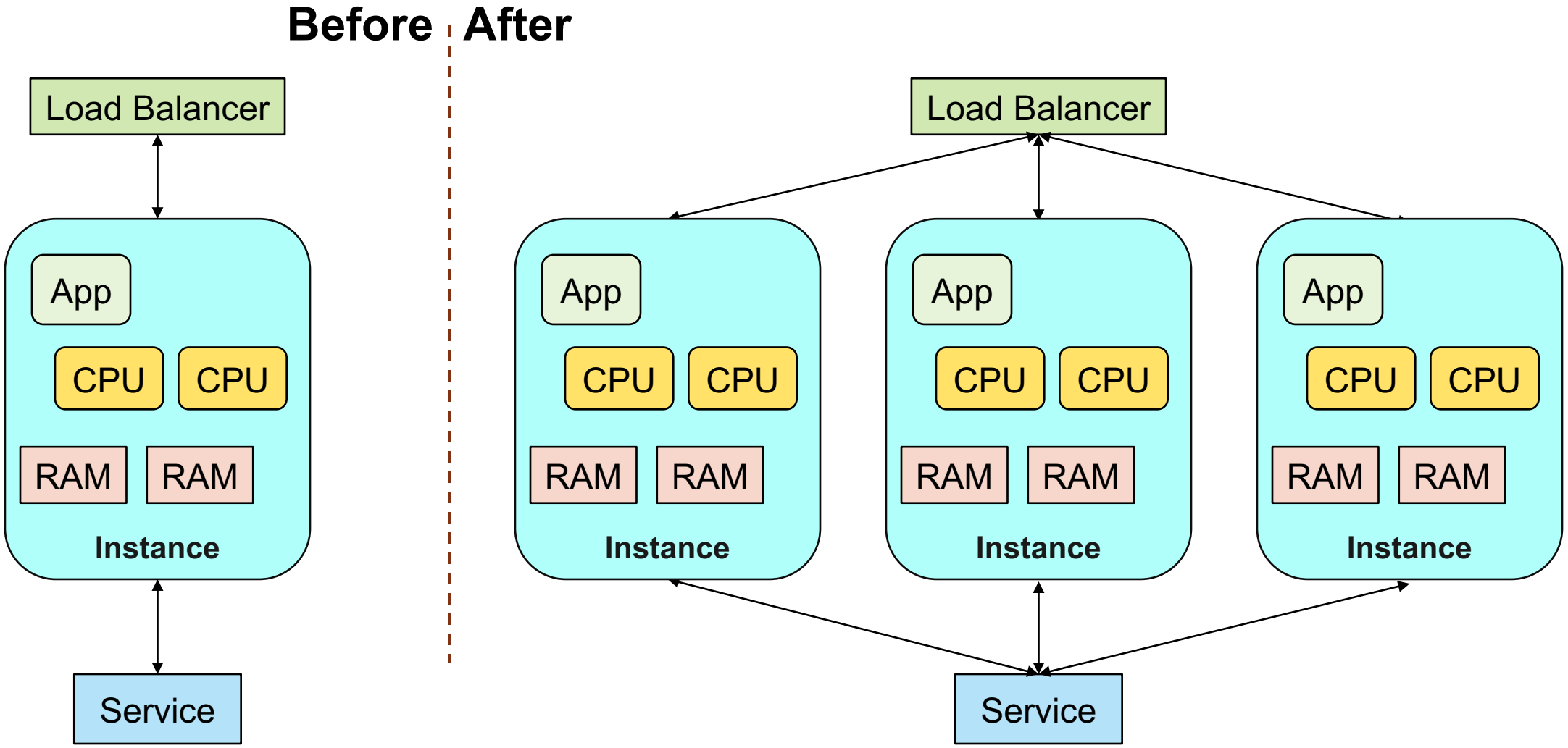


Factor 8: Concurrency

I. Codebase
II. Dependencies
III. Configuration
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

- Scale out using the process model
  To add capacity, run more instances
- There are limits to how far an individual process can scale
- Stateless applications make scaling simple

Concurrency: Runtime instances



Factor 9: Disposability

I. Codebase
II. Dependencies
III. Configuration
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

- Maximize robustness with fast startup and efficient shutdown
- Application instances are disposable
- Application should handle shutdown signal or hardware failure with crash-only design

Processes: Stateless application

Service State

Load Balancer

Session State

Runtime 1

Runtime 2

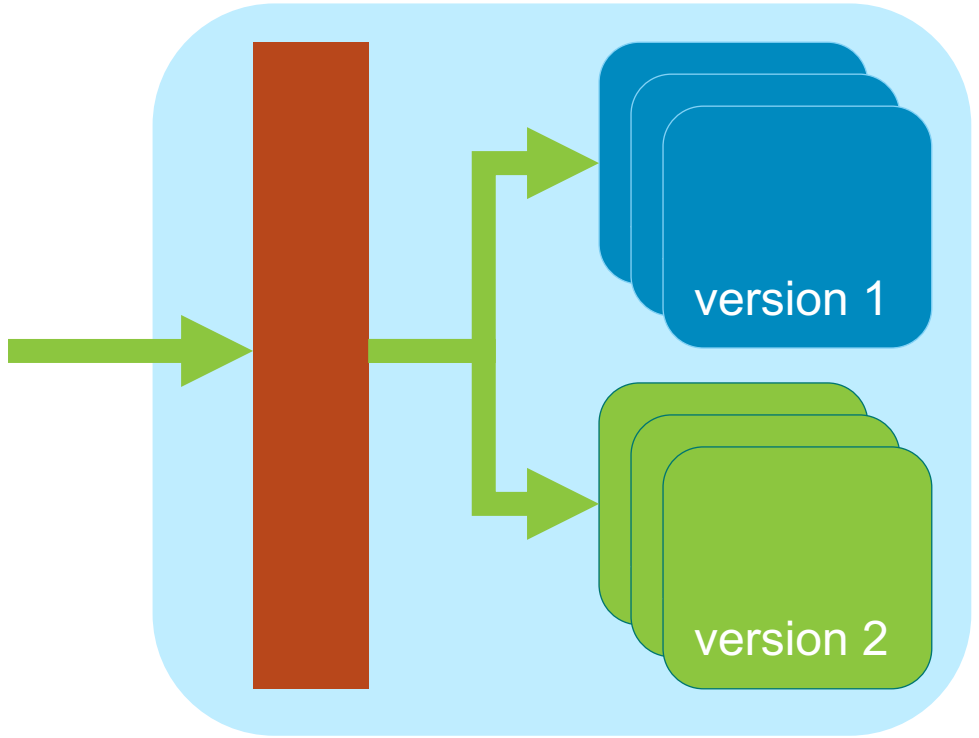
Domain State

NOT

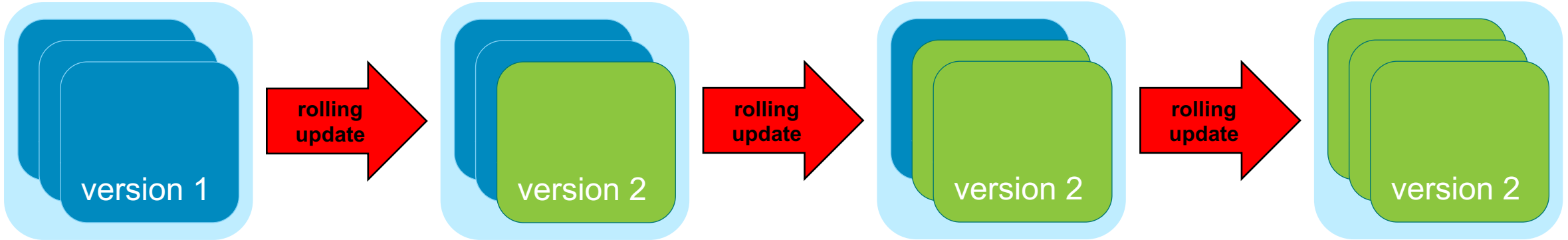
Zero-downtime Deployment

- Deploy a new version without causing an outage
  - Cloud makes this much easier to achieve
  - Two approaches
- Blue/green deployment
  - Deploy v2 alongside v1
  - Shift user load from v1 to v2
  - Requires 2x capacity
- Rolling deployment
  - Replace instances of v1 with v2
  - Requires 1x capacity

Blue/Green Deployment



Rolling Deployment



Blue-green deployment

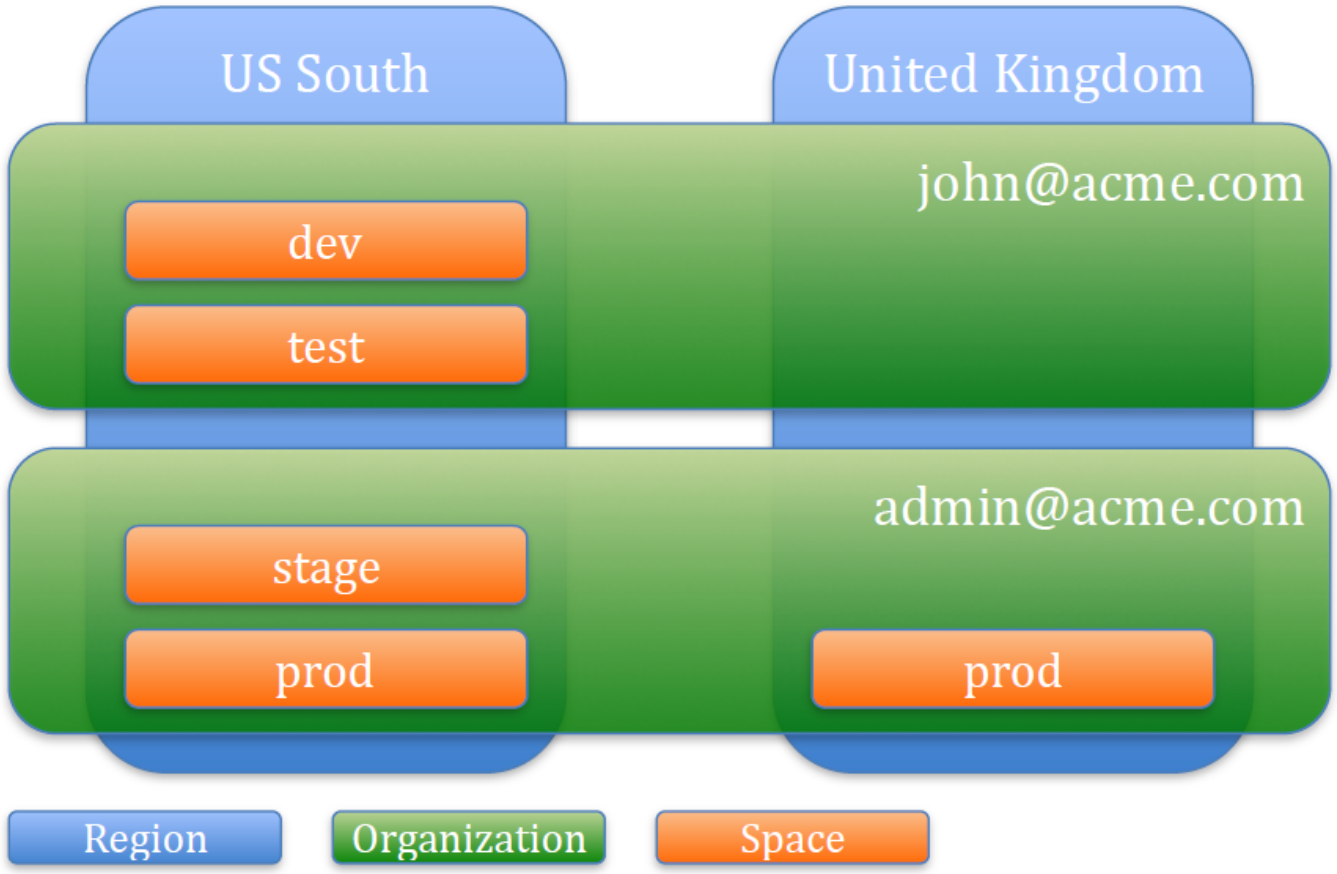
- A zero-downtime deployment technique
  - Also known as red-black Deployment
- Two nearly identical production environments, called Blue and Green
  1. Deploy v1 (blue environment), which users access
  2. Deploy v2 (green environment)
  3. Shift client load from Blue to Green
  4. Delete Blue

Factor 10: Development and production parity

I. Codebase
II. Dependencies
III. Configuration
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

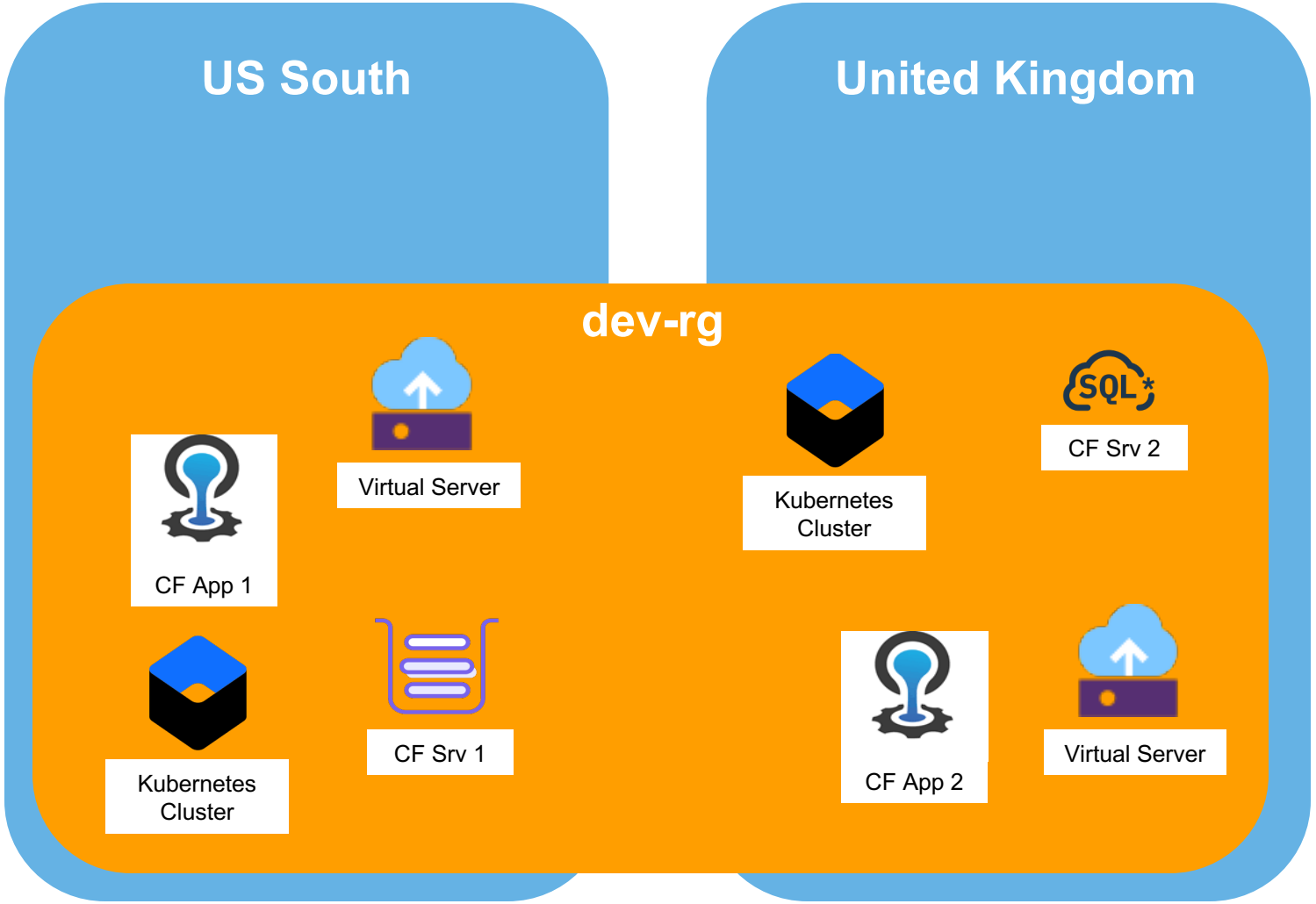
- Keep development, staging, and production environments as similar as possible
- Use the same backing services in each environment

Regions, organizations, spaces, users in Cloud Foundry



- **Region**: A *region* is a specific installation of IBM Cloud. IBM Cloud public has regions like US-South, UK-South, and AP-South.
- **Organization**: An *organization* has a quota of resources available for deploying and running applications. It can span regions. It can have one or more users.
- **Space**: A *space* is a group of runtime artifacts, like applications or services, hosted in a region and belonging to an organization. Can be used in a variety of ways to group artifacts (dev, test, production).
- **User:** A *user* will have assigned roles and permissions within an organization or space.

A Resource Group can contain any kind of resource



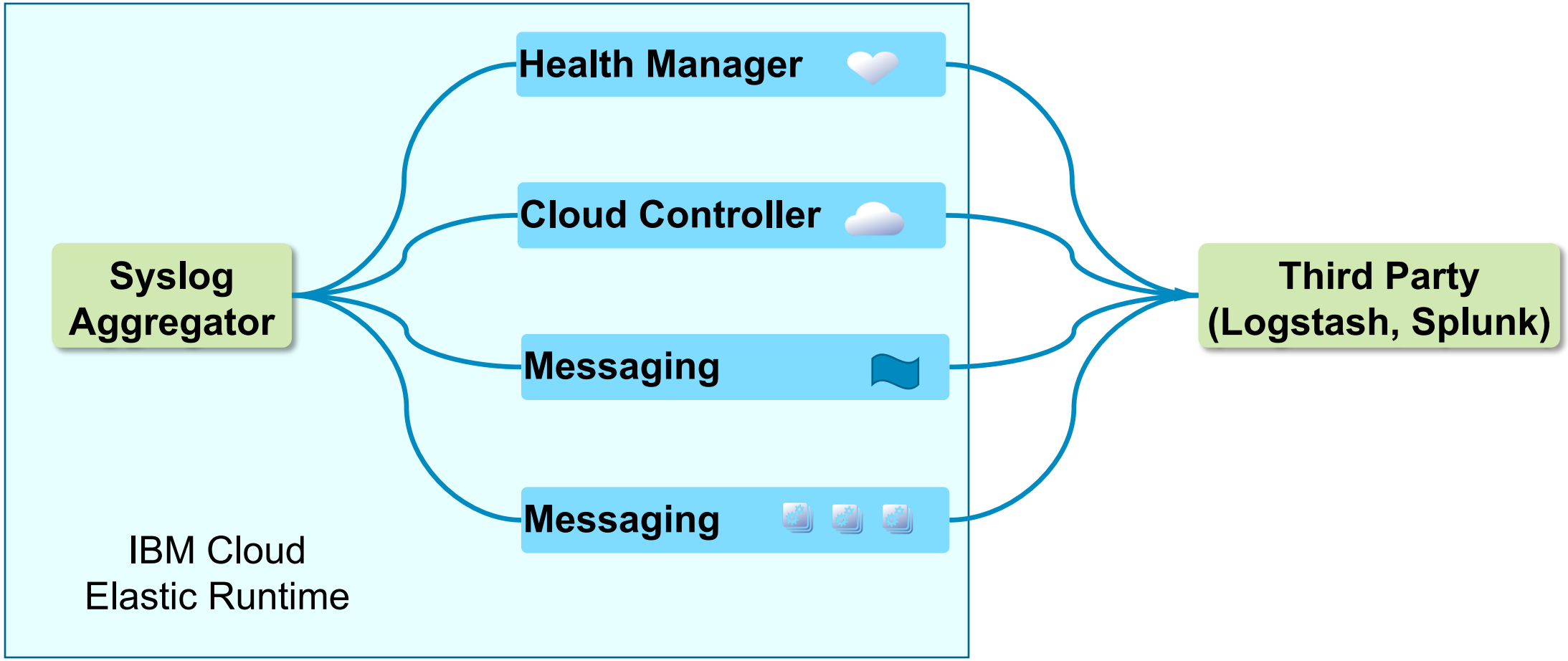
- **Can Span regions**
- **Users provided different levels of access**

Factor 11: Logs

I. Codebase
II. Dependencies
III. Configuration
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

- Treat logs as event streams
- Each process writes to stdout
  - Application should not write to specialized log files
  - Environment decides how to gather, aggregate, and persist stdout output

Logs: IBM Cloud syslog



Factor 12: Administrative processes

I. Codebase
II. Dependencies
III. Configuration
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

Run administrative and management tasks as single processes, such as these examples:

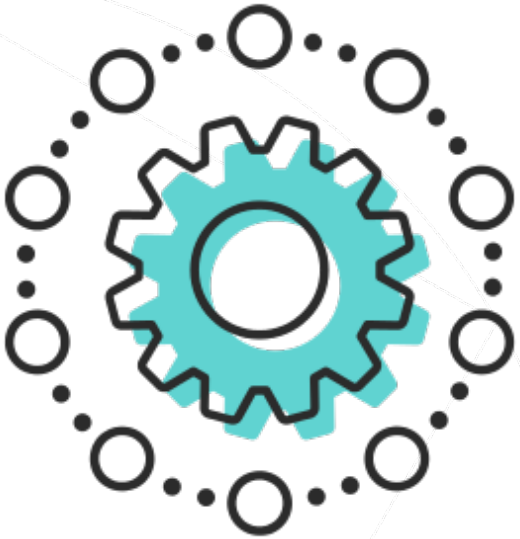
- Tasks for performing database migrations
- Debugging

Administrative processes: Scripting

- Create an SQL database
  - DDL script that creates the schema
  - SQL script that populates initial data
  - Script that runs these scripts as part of creating the database
- Migrate data to a new schema
- Software-defined data center (SDDC)
- Deployment scripts as part of a CI/CD pipeline
  - Creates service instances, deploys runtimes, and binds them

- Scripts are repeatable
  - Test scripts in stage environment
  - Scripts run the same in production
- Store scripts in software configuration management (SCM) system

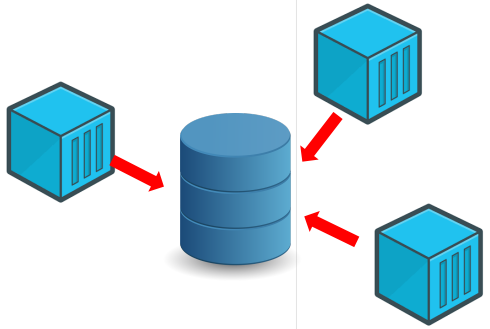
What are Microservices?



Microservices are a software development technique - a variant of the service-oriented architecture (SOA) architectural style that structures an application as a collection of loosely coupled services.

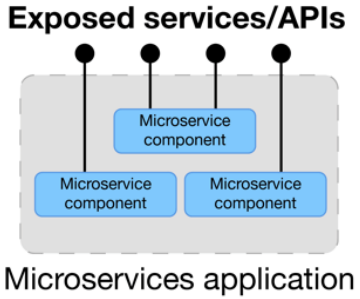
A Microservice

- Serves one purpose



- One primary data source

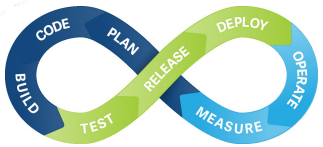
- Has a tightly controlled interface



Microservices application

- Is built by a small unified team - one XL pizza can feed the whole team.

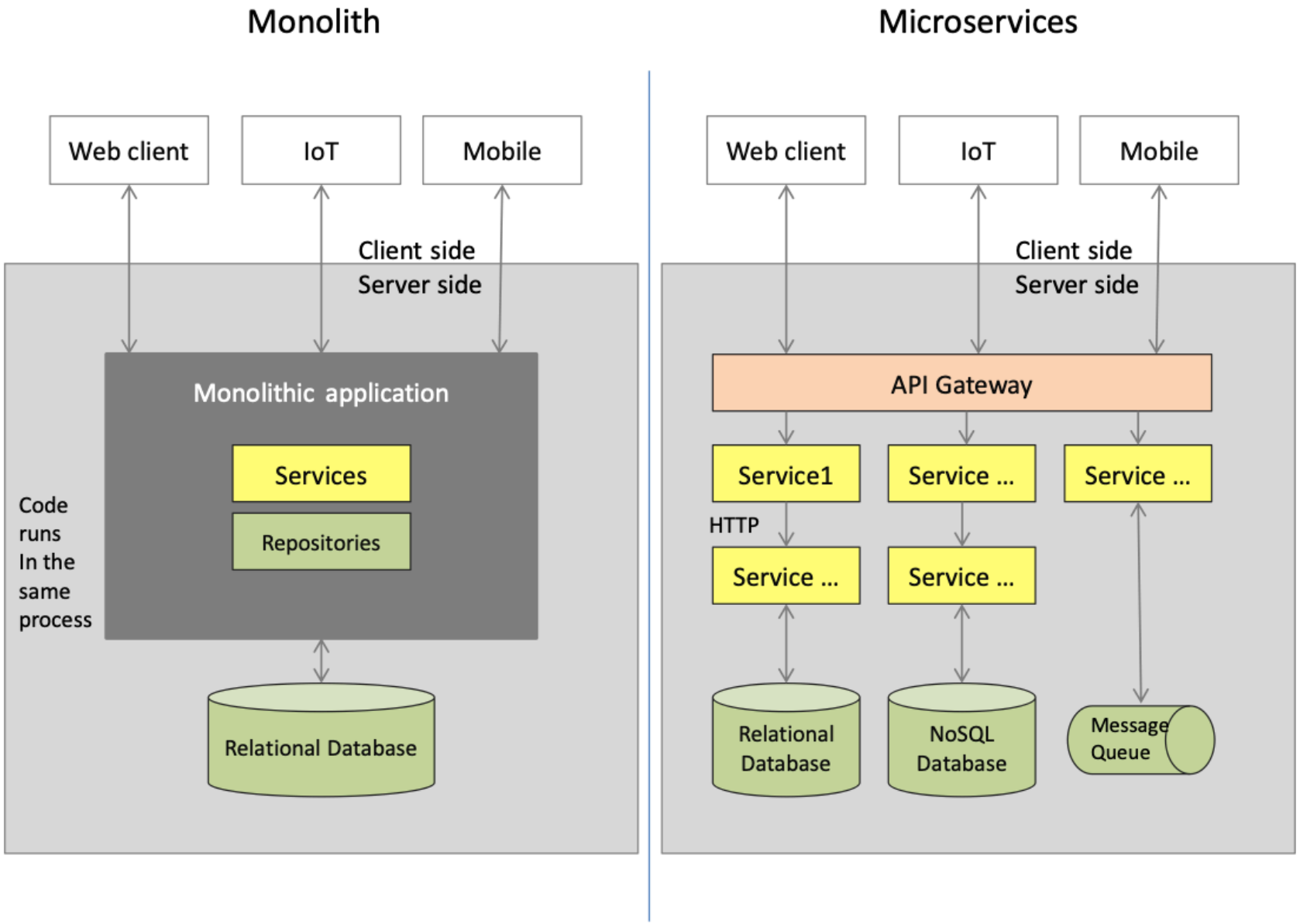
- Is built and deployed as a unit - one devops pipeline, one release schedule (CI/CD).



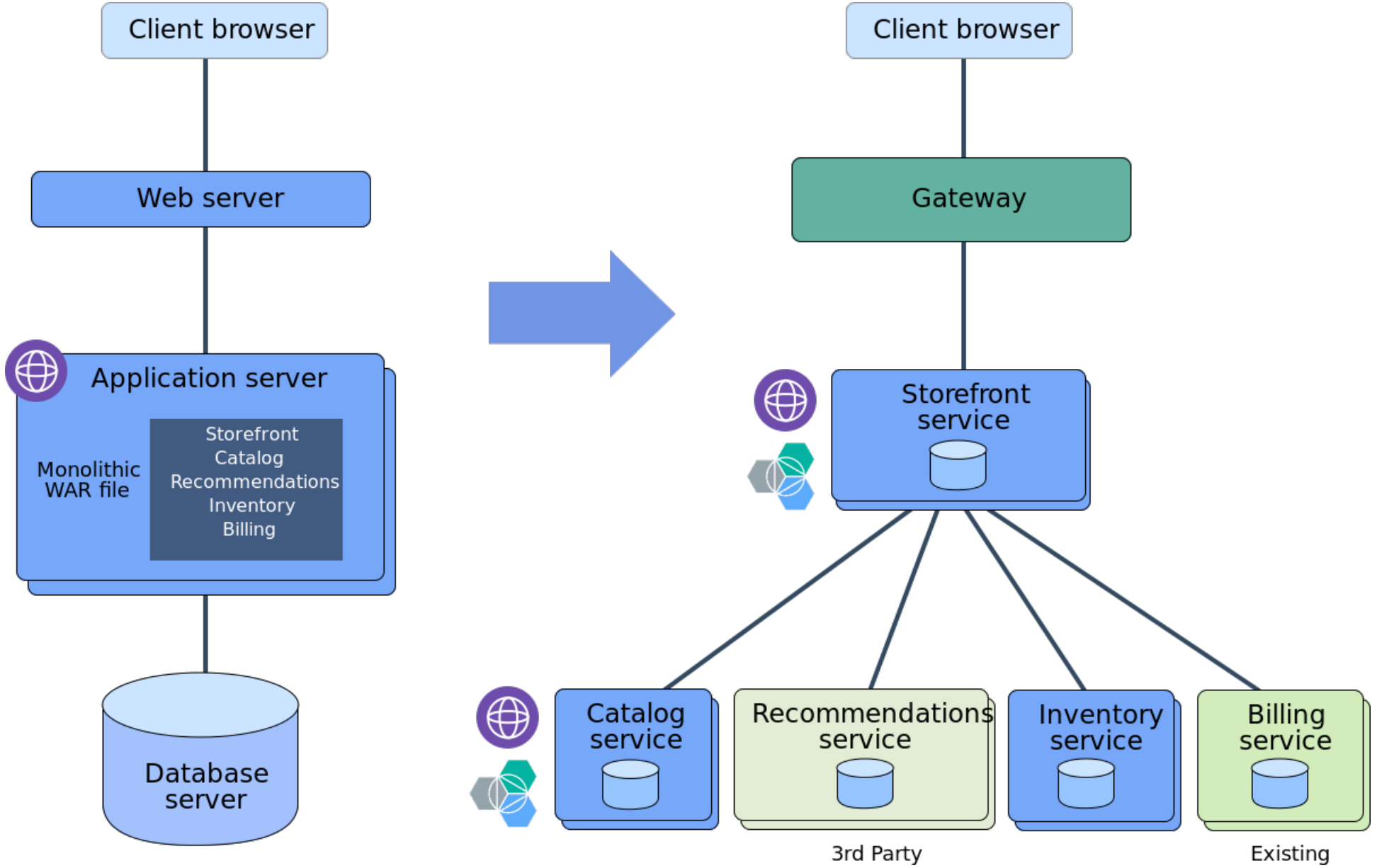
What are the real benefits of microservice architectures compared to monoliths?

- Agility, and faster time-to-market.
- Innovation - using the best technology for each problem.
- Resilience, such as better fault isolation, less impact on other services.
- Better scalability.
- Better reusability.
- Improved Return on Investment (ROI), and better Total Cost of Ownership (TCO).

Monolith vs Microservices

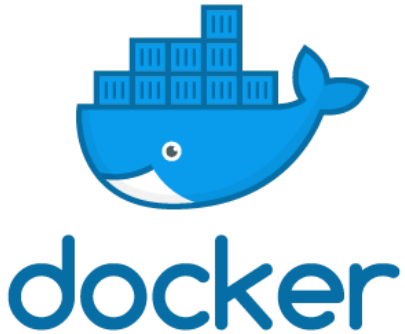


Migrate applications to microservices

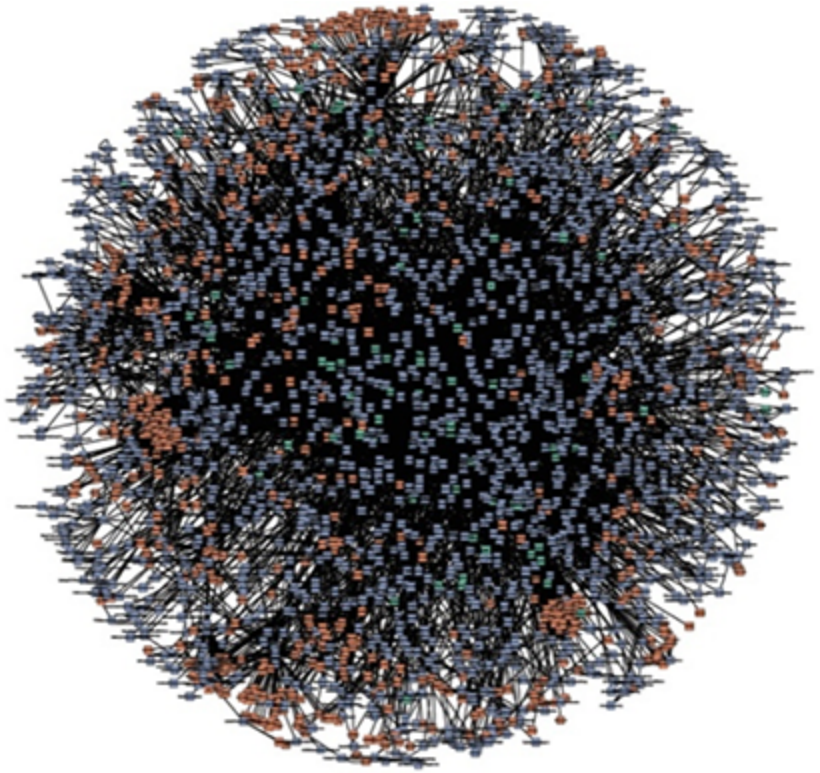


Options for running microservices

- **Docker containers**
- **Cloud Foundry**
- **Serverless**
- **Virtual machines (VMs)**



Challenges of the microservice architectural

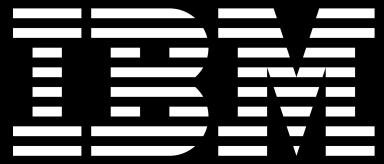


amazon.com

NETFLIX

Best practices use in microservices-style applications

- **Consistency across services**
- **Conflict-free data structures / log-style databases / Single-Writer principle**
- **Versioning support**
- **Resilience**
- **Scalable (No-)SQL database**
- **Autoscaling**
- **Service mesh (Istio)**
- **Security by design**
- **Infrastructure automation**



Tal Neeman
Developer Advocate, IBM Alpha Zone
talne@il.ibm.com