Zurich University
of Applied Sciences

School of
Engineering
InIT Institute of Applied
Information Technology

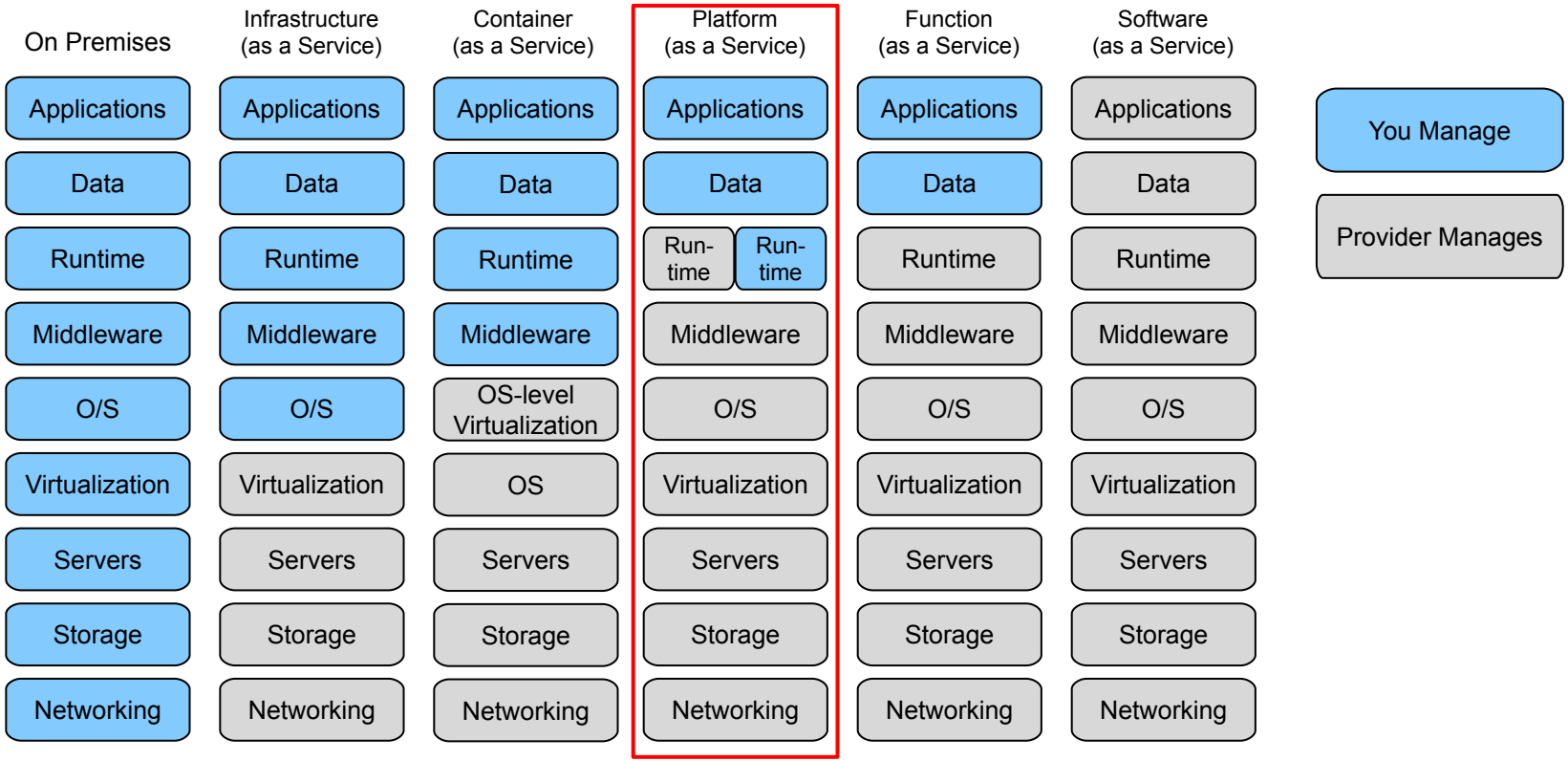
ARCH – PaaS Architecture

Prof. Dr. Thomas M. Bohnert
Christof Marti

Content

- Components of a PaaS System
- Generic PaaS Architecture
- PaaS Distributions (Products)
  - OpenShift / Kubernetes
  - CloudFoundry Application Runtime

Cloud Computing Service Models



PaaS - Functional Separation of Concerns

PaaS Extended Functionalities	app model:	composite application model / management
	app life-cycle:	CI/CD pipelines, deployment tools
	app testing:	complex testing environments
	app analytics:	usage, performance, visualization
PaaS Core Functionalities	application:	stage, run, schedule, scale, health
	backing service:	marketplace, provision, bind, manage
	networking:	routes, domains, load-balancer, dns
	monitoring:	application logs, metrics, events
	multi-tenancy:	users, organization, project/space, quota
PaaS Operation Functionalities	performance, availability:	deploy, provision, monitor, scale PaaS components
Infrastructure	IaaS - Compute, Network, Storage Virtualized or Bare-Metal	

PaaS - Functionality and Architecture

- Actual Application Platforms (aPaaS) focus on PaaS Core Functionalities
- Extended functionalities are usually left to the 3rd parties (e.g. users, providers) E.g.
  - CI/CD (Jenkins, GitLab, GitHub Actions, Concourse, ...),
  - Application Provisioning & Management (e.g. Ansible, Pulumi, …),
  - Monitoring & Analytics (Prometheus, ELK[1)], Jaeger, …)
- Although PaaS distributions / products are using different technologies, they share some common concepts and components
- What are the generic architectural components of a PaaS?

1) ELK: Elastic Search, Logstash, Kibana

Architectural Components known from Infrastructure?

- **Compute**: e.g. OpenStack Nova
  - Scheduling, monitoring & management of VM
  - interacting with Hypervisor
  - Host, Cluster, Cell, Availability Zones, Region,
- **Networking**: e.g. OS Neutron, Designate
  - Virtual Networks (L2/L3), Bridges, Routers, DHCP, FW (Security Groups), DNS
  - interacting with SDN-Service, Physical Network Infrastructure / NMS
  - Between VMs, Hosts, Cluster, Data Centers,…
- **Storage**: e.g. OS Cinder, Swift, Manila
  - Block-Storage, Object-Storage, File-Storage,
  - Pluggable backend (ZFS, Ceph, ...)
- **Image-Store**: e.g. OS Glance
  - Storing, managing VM images, snapshots,

Service- / Data-Plane

- **Controller**: e.g. OpenStack Keystone
  - Users, Projects / Tenants,
  - Authentication, Authorization, Security Tokens
  - Policies & Rules
  - Catalog / Service-Registry
- **Mgmt-APIs**:
  - REST-based API-Endpoints for each Service
- **Clients**:
  - Web UI / Dashboard: OpenStack Horizon
  - Command Line Interface (CLI)
  - Software Development Kit (SDK)
- **Automation / Orchestration**: OS Heat
  - Infrastructure as Code

Control-Plane

What are the matching components on PaaS level?

Managing Virtual Machines
- Scheduling & Placement
- Replication
- VM-state

→ **Managing Applications**
- Scheduling & Placement
- Scaling of Instances
- Health monitoring (process, endpoint)

Network Connectivity
- Layer 2 / 3 (Ethernet / IP)
- Firewall (TCP/UDP Ports)
- Load balancing

→ **Application Communication**
- Layer 4-8 (TCP/UDP, HTTP, AMQP, ...)
- LB, Proxy, API-Gateway
- SSL-encryption, Certificates

Storage access
- Block, File & Object-Storage

→ **Data persistence**
- Storage-Service access (Volumes, DB, KV-Store, …)

Image-Store
- VM-Images & snapshots

→ **Repository**
- Container-Images / -Blobs
- Libraries, Packages

What are the matching components on PaaS level?

Zurich University of Applied Sciences
School of Engineering
InIT Institute of Applied Information Technology

Controller

- Users, Projects/Tenants, AAA[1)], Security
- Mgmt-API-Registry/Gateway (Infrastructure-Mgmt)

Controller

- Users, Projects/Spaces, AAA[1)], Security
- Mgmt-API-Registry/Gateway (Application-/Service-Mgmt)

Mgmt-Clients

- Web UI / Dashboard, CLI, SDK (Horizon, os-client, awscli, …)

Mgmt-Clients

- Web UI / Dashboard, CLI, SDK (oc, kube-dashb, Lens, kubectl, k9s, …)

Automation / Orchestration

- Infrastructure as Code (Heat, CloudFormation, Terraform, …)

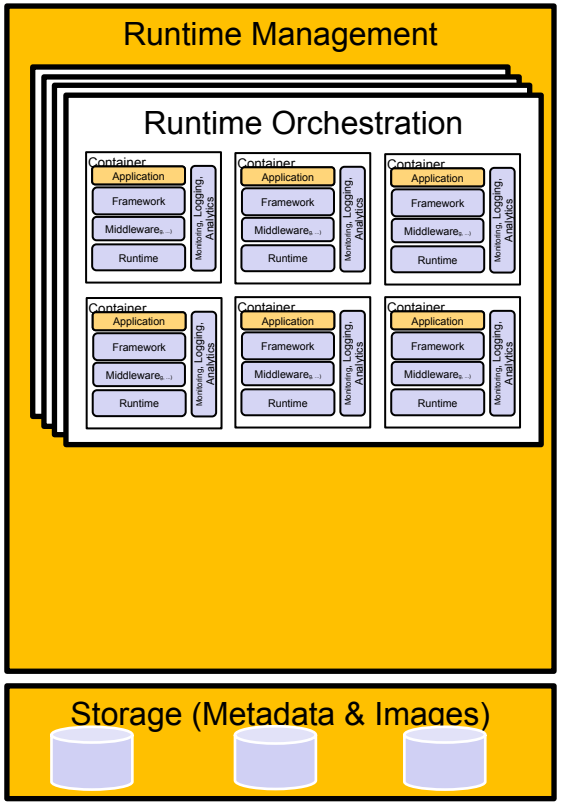
Automation / Orchestration

- Runtime Environments / Image Building (Dockerfile, Buildpacks,Source2Image,...)
- Application Orchestration (DockerCompose, Helm, Kustomize,...)

1) Authentication, Authorisation & Accounting

Runtime Management / Orchestration

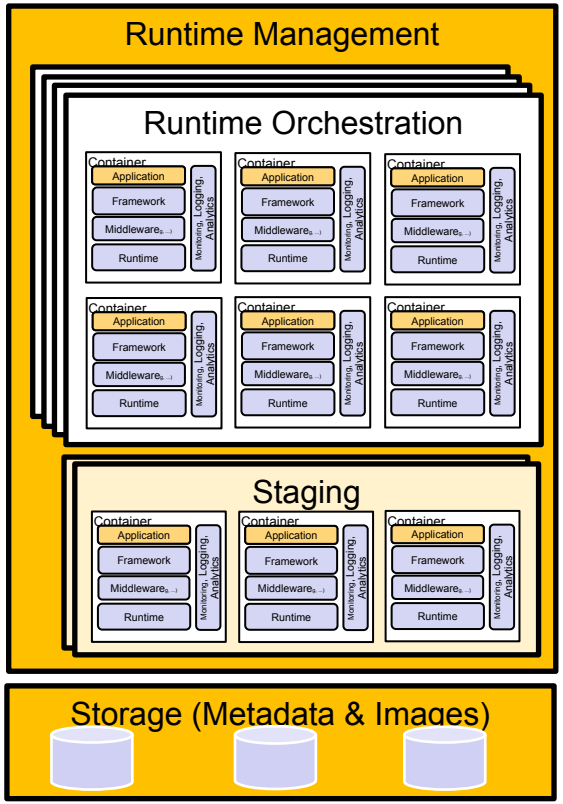
- Manages the creation, caching, scheduling, placement and disposal of application runtimes (VM, Container, Unikernel/MicroVM, …)
  - Distribution over multiple hosts (physical or virtual).
  - Manages individual application instances
    - deploy / dispose
    - starting / stopping
    - scaling
    - track and broadcast state messages



- Metadata and Images of runtime environments are stored in a persistent storage area

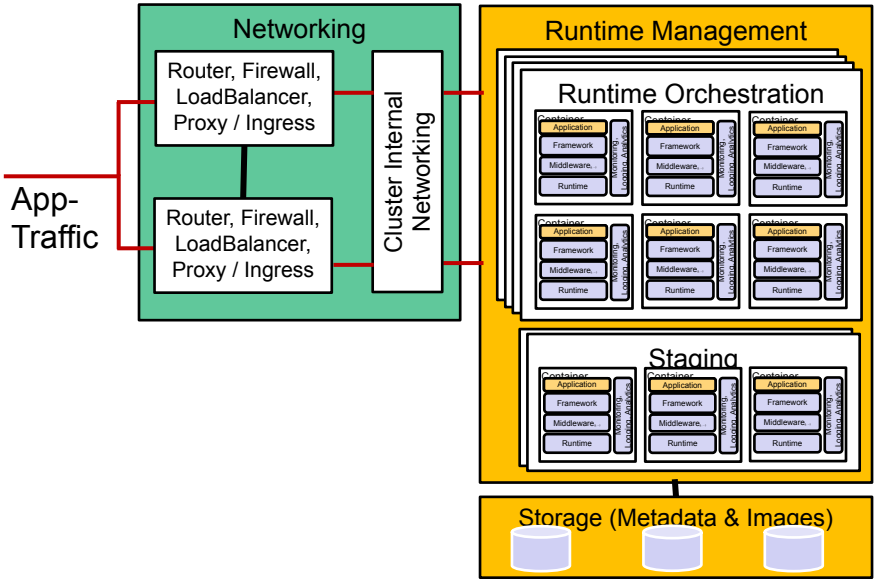
Runtime Staging / Image Building

- Automated building of runtime images
  - Create runtime image from code based on build instructions
    - install runtime, compile app, fetch dependencies, package the application, configuration, startup, …)
  - Popular options: Dockerfiles, Builder images, Buildpacks, …
  - Basic process
    - spins up temporary runtimes (container, VM, …)
    - runs the instructions in the runtime
    - creates runtime image
    - saves image in the store / registry
    - destroys temporary runtimes
- Staging may be done in shared or separate environments



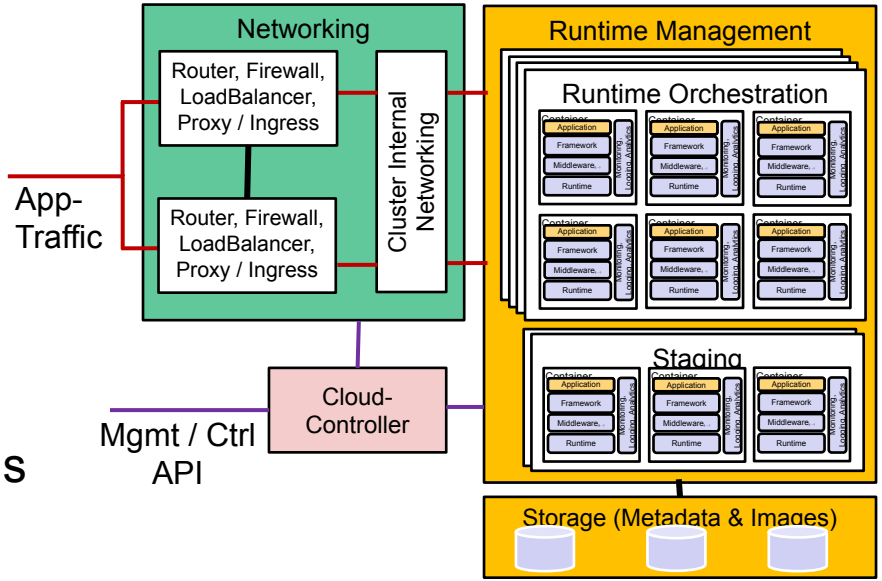
Networking

- Handles all traffic to and between apps / services (data plane)
  IP → TCP/UDP → HTTP(S),AMQP,...
- Cluster Internal Networking
  - Traffic between Apps / Services
  - Using private network addresses
- External gateway
  - Routes public incoming traffic (URL/public IP) to the appropriate runtime (internal IP/port)
  - Provides Load Balancing features
    - multiple runtimes per app
  - Maintains distributed routing state
    - Removes routes, which are stale or unhealthy
    - Real Time updates to routing table of containers
  - Provides Tunnels to Apps & Services
  - May provide SSL-Endpoints



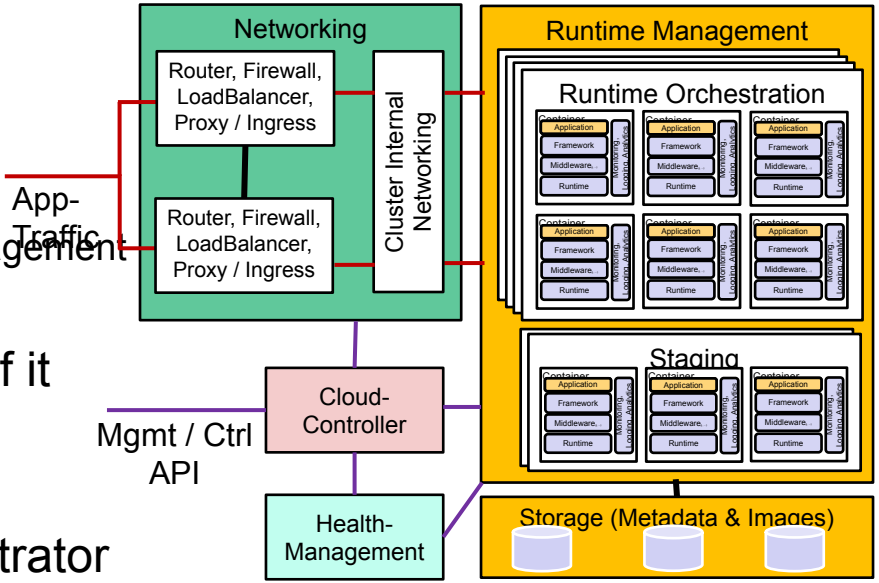
Cloud Controller

- Provides a Management / Control (REST) API to the user
  - Interacts with different subsystems
  - Deals with users, apps and services
  - Management and Control Plane
- Manages the lifecycle of applications
  - Provides runtimes (Container, VM, …)
  - Kicks off staging
  - Binds/Links services to applications
  - Handles all state transitions
- Interface for (external) tooling/tool chains
  - Web UI, CLI, SDK, …



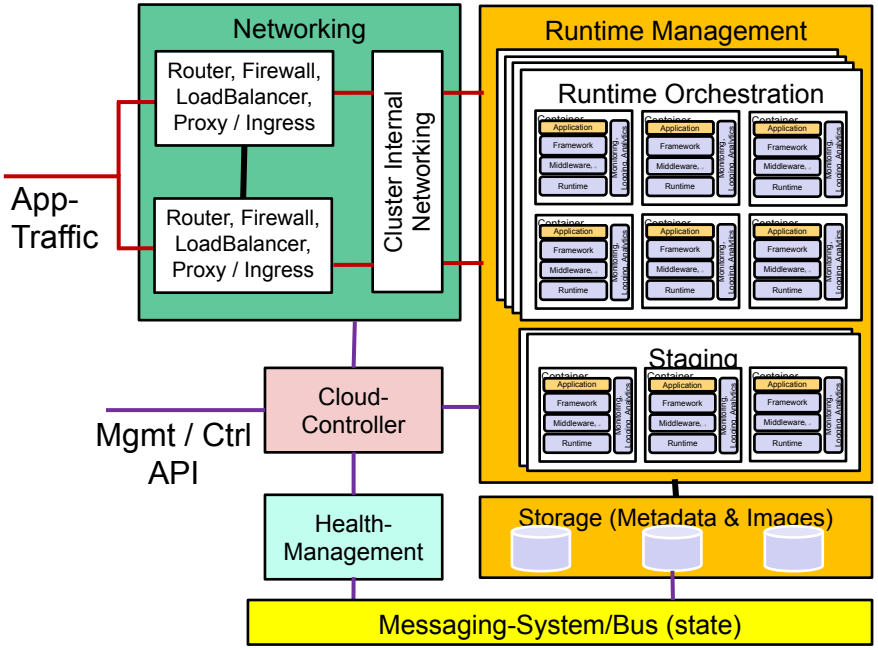
Health-Management

- Monitors the state of the applications
  - IO, Memory, CPU, Threads, Disk, health endpoints
  - Running, stopped, crashed
  - Number of instances, version number
  - Bound services
- Compares “intended” with “actual” state
  - Listening on state messages from Runtime-Management
  - Determines drift from metainfo
- Takes (orchestrates) corrective actions if it detects differences
  - directly (or through) cloud-controller
- May be integrated with Runtime Orchestrator



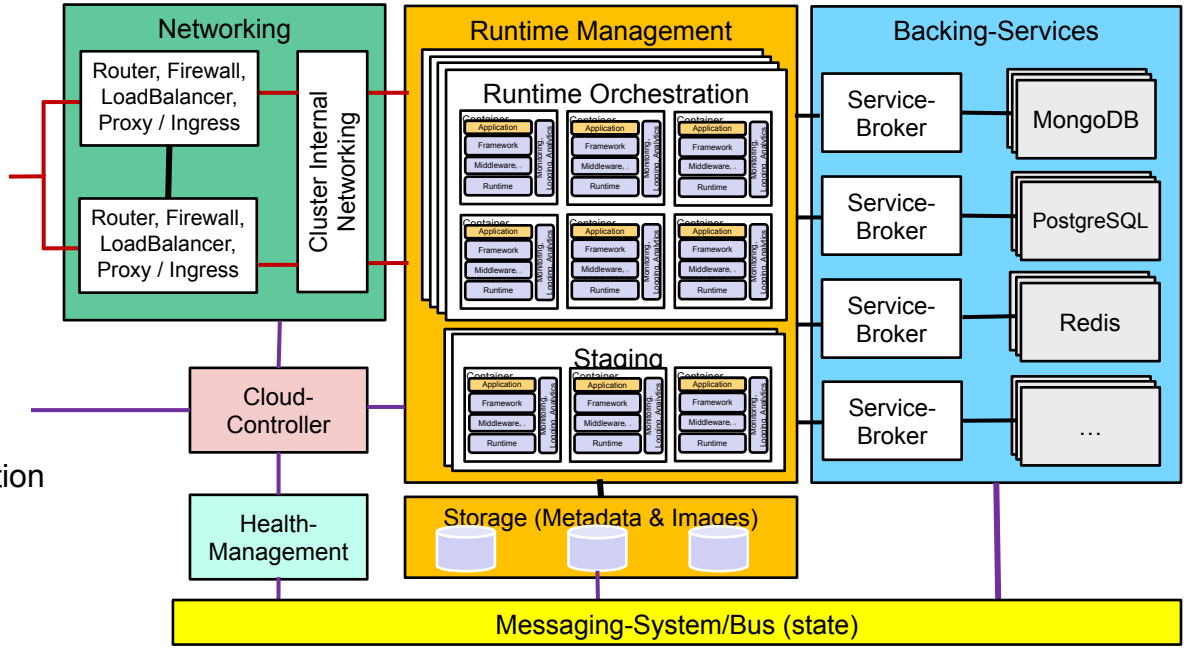
Messaging-System / Bus

- Central and standard communication system
  - Used by components for internal communication
  - Command & control
  - State information
  - Log-messages
- Publish / subscribe based
  - Dial tone, fire and forget (stateless)
  - Auto discovery and addressing
  - Decoupling of components
- Protects “itself” at all cost
  - If it hangs, the whole system breaks



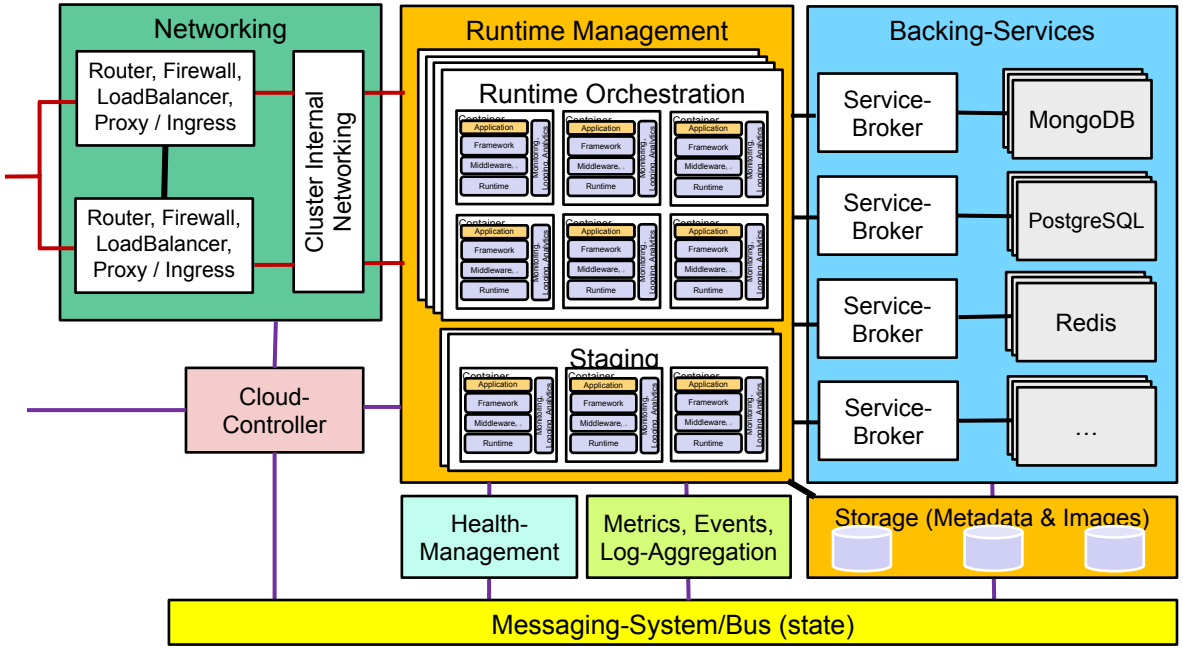
Backing Services / Service Broker

- Backing Service Marketplace
  - Maintains a service catalog (marketplace) & service metadata
  - Service advertising
  - Services are either external or provided within the runtime environment
- Deployment of service instances
  - setup, credentials, …
  - Shared (multiple user accounts), Dedicated (instance per connection)
  - Access control, Single-sign on
- Bind/Unbind service to application
  - Provisioning, providing access to application
  - Configuration of application
- Service broker API
  - Possibility to add new local & 3rd party services



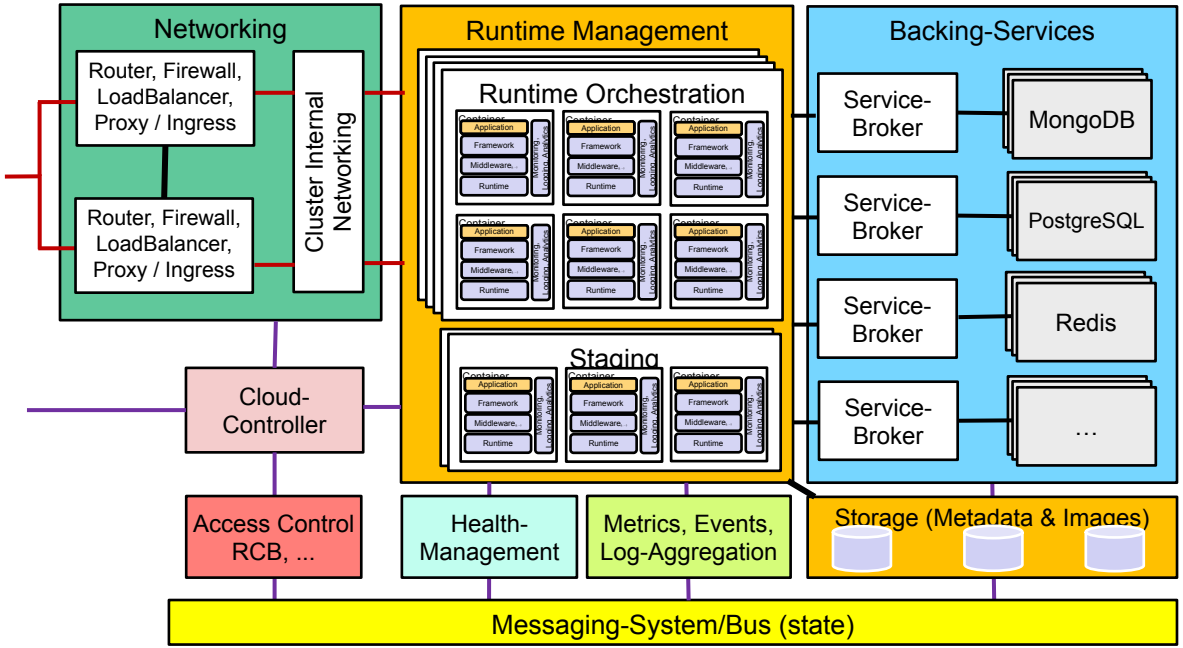
Log-Aggregation, Event/Metrics-Collector

- Aggregate Logs of applications
  - from all application instances
- Emits system events
  - Application started, restarted, stopped
  - Instance crashed, restarted
  - (Auto-)Scaling
- Emits metrics
  - Usage statistics
  - Uptimes
  - Traffic



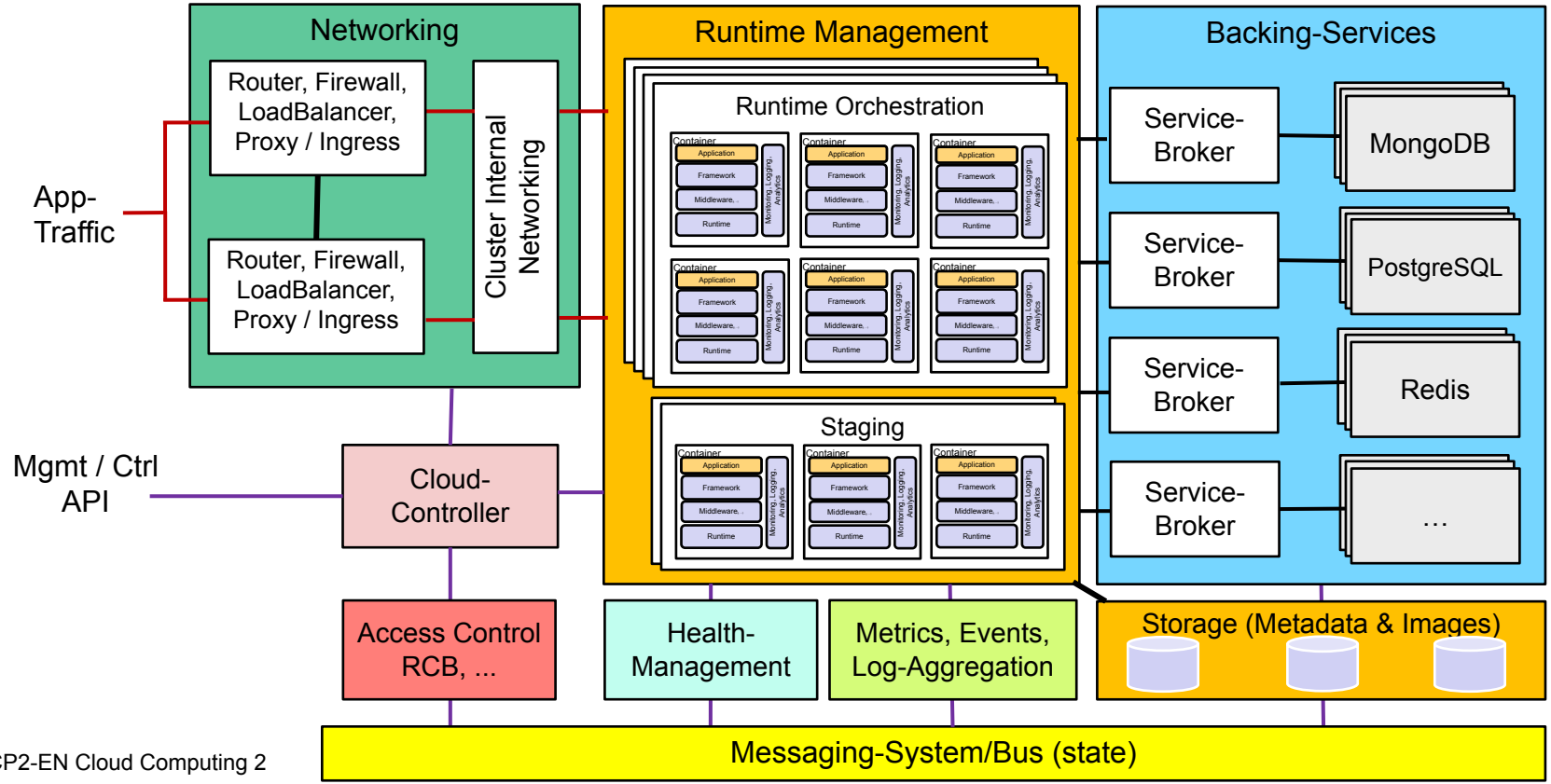
Operations Support and Management Systems

- Access Control
  - Multi-Tenancy
    - Users
    - Organizations
    - Projects / Spaces
  - AAA[1)]
    - Backend to AAA[1)] mechanisms (LDAP, OAuth, SAML)
    - Roles (Admin, Developer, Auditor, …)
- Rating, Charging, Billing
  - Collects usage information from services, containers
  - Aggregation of usage over time, calculation of costs
  - Write bills and charge credit cards



1) Authentication, Authorisation & Accounting

PaaS Core Architecture



PaaS Design and Architecture Principles

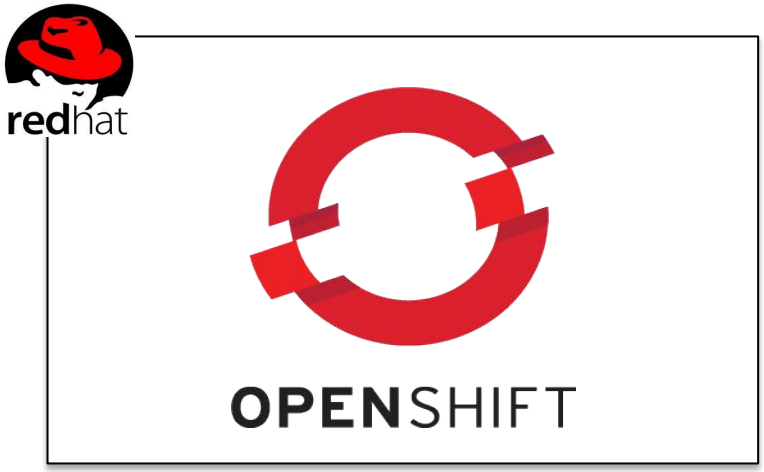
- Follow SOA/CNA Principles: Loose coupling, Event-driven, Idempotent, Asynchronous / non-blocking, Eventually consistent, Language-independent
- Declarative vs. Imperative: State your desired results, let the system actuate
- Control loops: observe, rectify, repeat
- Think big, economies of (world-wide) scale
- Cattle vs. Pets: Manage your workload in bulk
- Open vs. Closed: Open for extension, without modifying the components directly (Open-Source, Open-APIs, interfaces, modularity, plugins)
- Legacy compatible:
  Apps should run without modification (may not use the full potential)

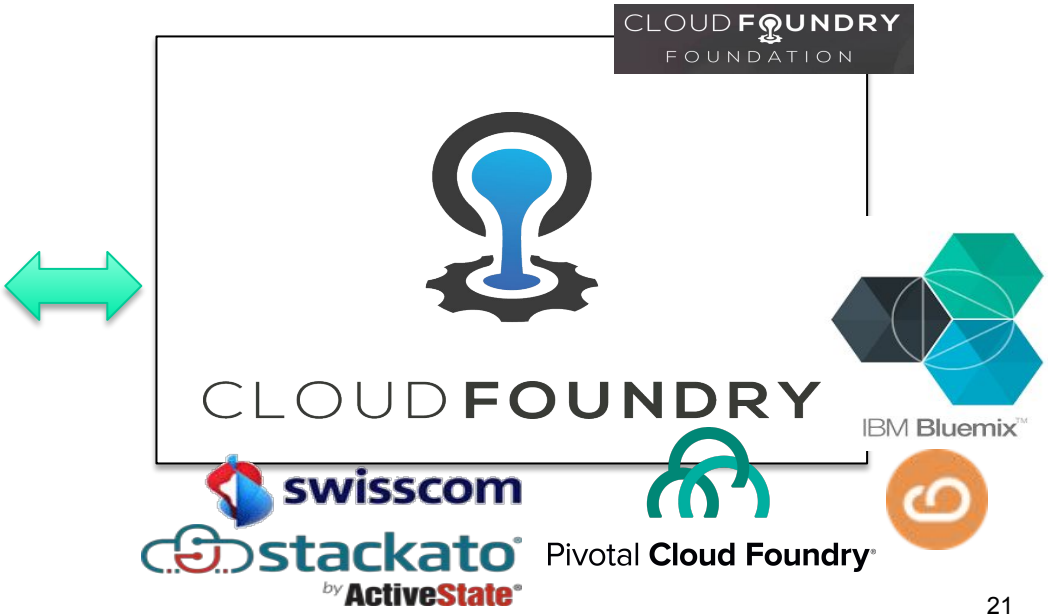
PaaS Architectural Requirements

- No single point of failure → redundant components
- Distributed state → No central database
- Self healing → health manager
- Horizontal scalable → add VMs/Hosts if required
- Dynamically discoverable components
  - Components are announcing themselves on the message-bus
  - Service-Registry
- Loose coupling, distributed components
  - Launch in any order (wait until dependencies are resolved)
  - Scale up and down independently
- Monitor all components using defined end points (e.g. HTTP)

PaaS Open Source Implementations

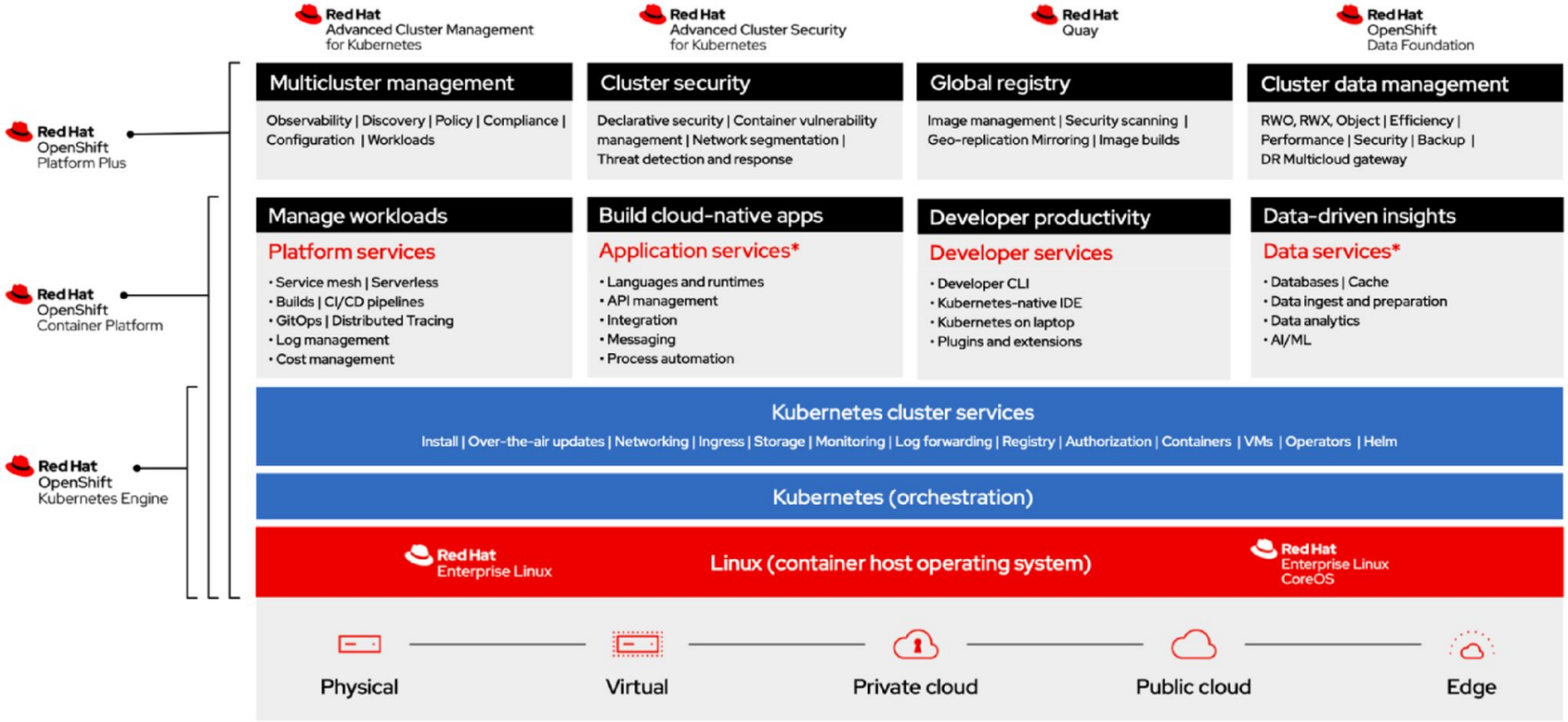
Following we are covering some implementation details of the main Open Source PaaS Frameworks





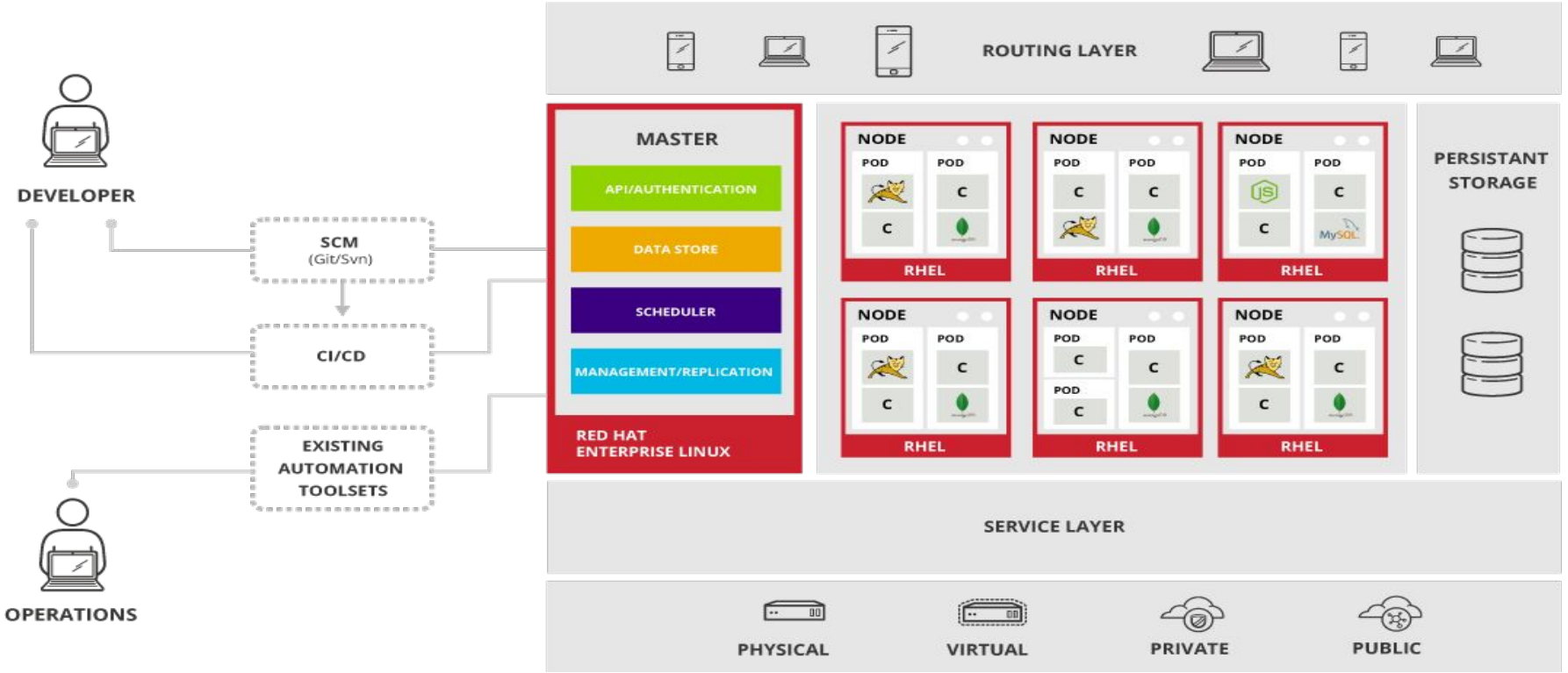
3 Flavors: okd - origin community distribution, Online (Public), Dedicated (Enterprise)

OpenShift 4 Container Platform



source: Red Hat OpenShift Container Platform datasheet
https://www.redhat.com/en/openshift-4/features

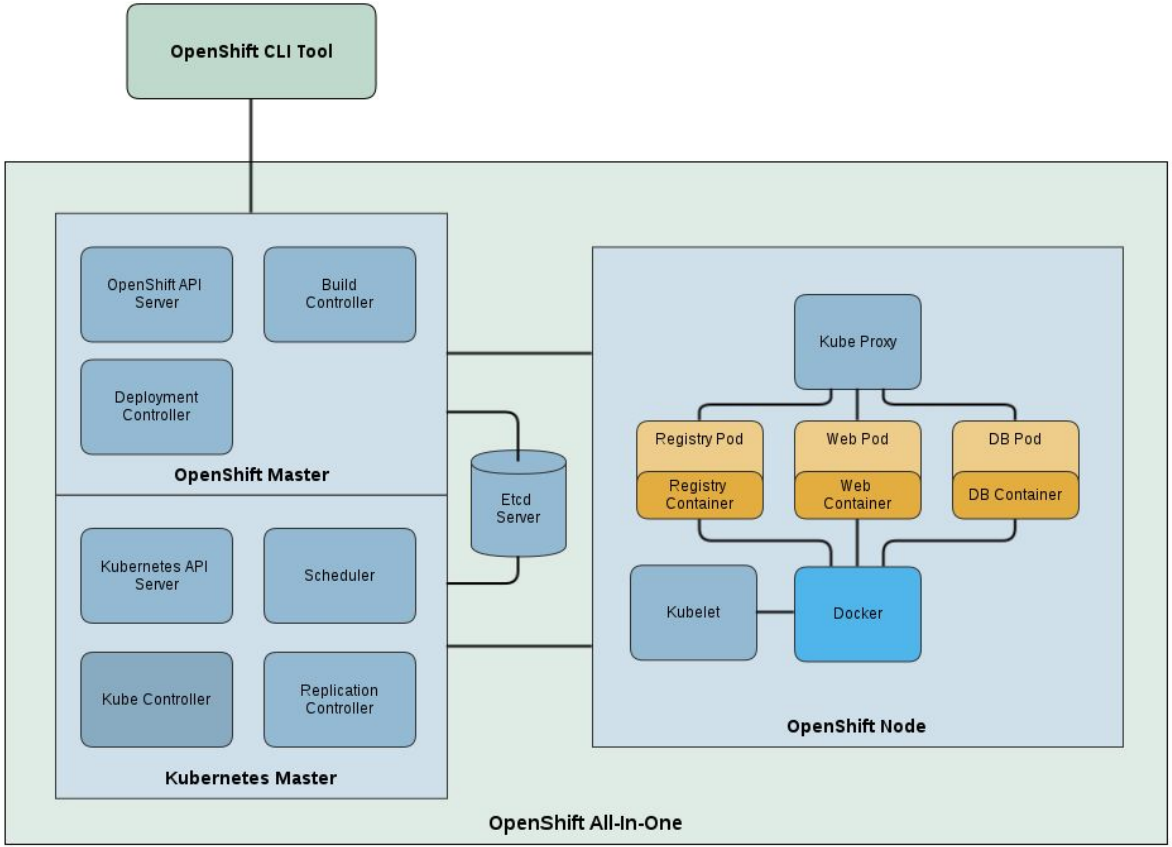
OpenShift - Components



OpenShift - Components

- Runtime based on Kubernetes as the Container Management System
  - Master for management (multiple for HA)
  - Nodes to run applications
- Additional OpenShift Management functionality
  - **Multitenancy**: Authentication, Users, Projects, Organizations
  - **Build-tooling**:
    - Source Control Management integration: Git-only (Github, Bitbucket, …)
    - Staging of applications (Docker Build, Source-To-Image, Builder-Containers, Pipeline Build)
  - **Service-Layer**: Docker Container Registry, Service Catalog, Cluster Security
- Persistent Storage
  - Volumes (block storage) mounted on containers (NFS, iSCSI, Gluster, Ceph)
- Networking
  - Tenant based network isolation
  - Router plugins/strategies (HAProxy integration, Path based routing, secured routes, sharding, …)

OpenShift - Master/Node components

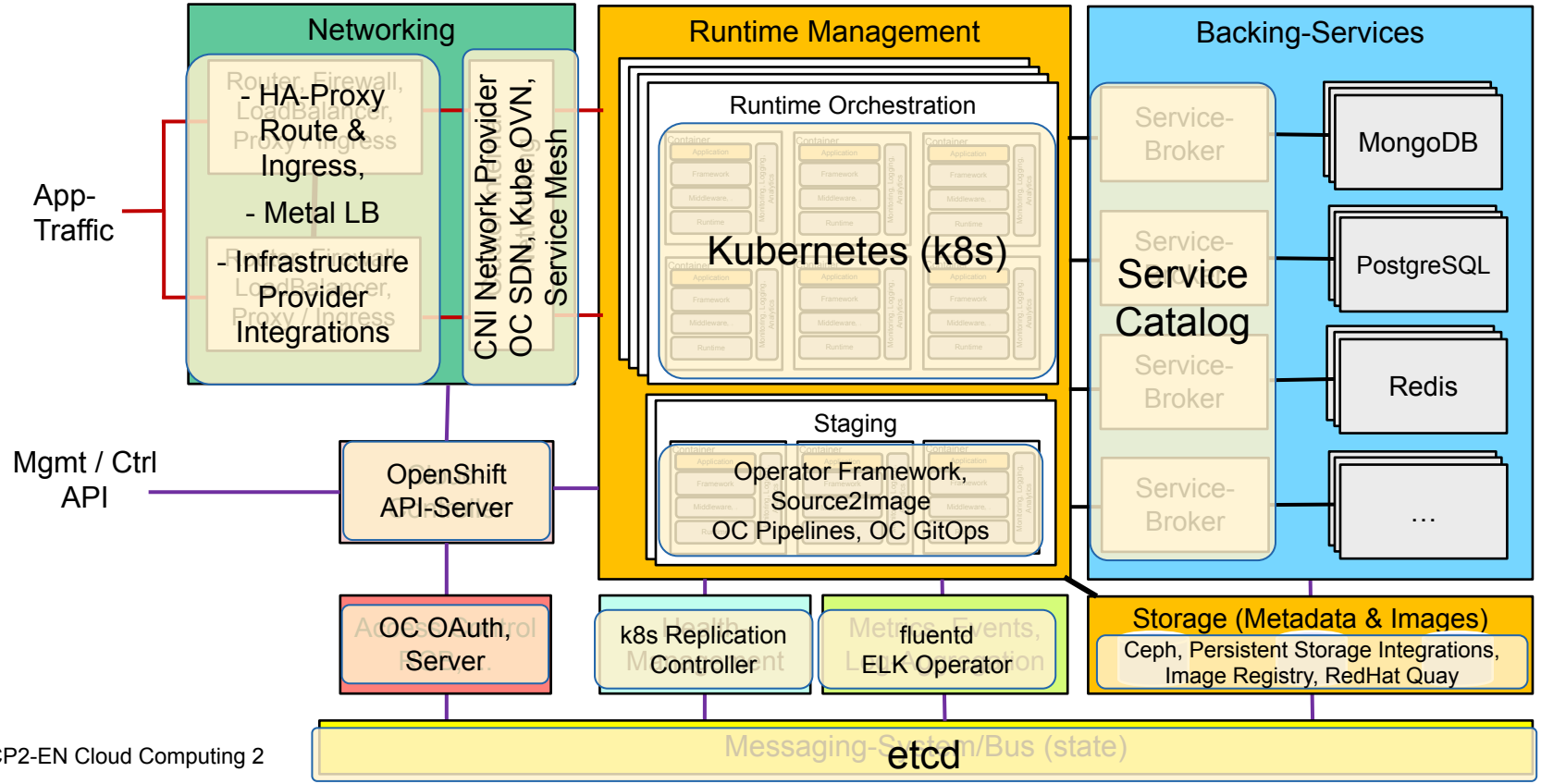


OpenShift - Master/Node components

- **Kubernetes Master**: Responsible for managing the state of the system, ensuring that all containers that should be running, are running, and that other requests (eg builds, deployments) are serviced.
- **Kubelet**: act as agents to control Kubernetes nodes. They handle starting/stopping containers on a node, based on the desired state defined by the master.
- **Kube Proxy**: allows applications running inside containers to access other containers deployed across the system.
- **OpenShift Master**: provides a REST endpoint for interacting with the system.
- **etcd Server**: to store system configuration and state.
- **Controllers**: Run with the masters to make sure the running system matches the desired state as stored in etcd. E.g. a **DeploymentController** watches for new Deployment objects and processes them. Similarly a **BuildController** watches for new Build objects and schedules the build (→ staging).

Kubernetes Details in CCP1 slides

OpenShift Component Mapping



Cloud Foundry Ecosystem

CLOUDFOUNDRY CONTAINER RUNTIME™

Origin Kubernetes Clusters managed by BOSH
- High availability, multi availability zone
- Scaling of cluster
- VM healing
- rolling updates of clusters

Details → Appendix

CLOUDFOUNDRY APPLICATION RUNTIME™

Elastic Runtime Environment for CNA
- Focus on Application
- Using Diego container management
- Integrated Application build/staging
- Open Service Broker integration
- Multitenant
- Foundation for most public CF offerings (e.g. Swisscom Application Cloud, PWS, …)

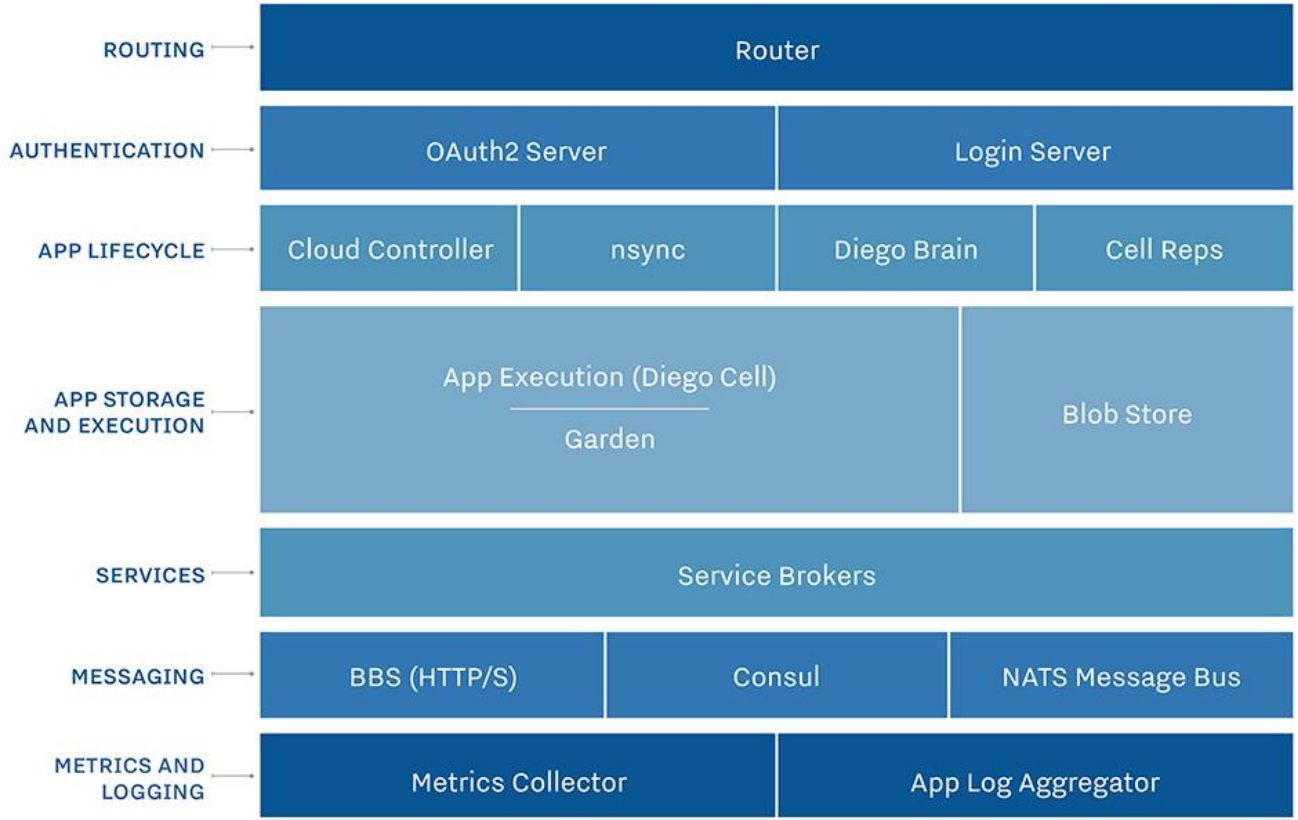
→ following slides

CLOUDFOUNDRY BOSH™

Automation tool to manage lifecycle of complex distributed systems
- Deploy & monitor Software services on IaaS VMs or bare-metal
- IaaS agnostic (AWS, OpenStack, Azure, Google Cloud, VMWare,...)
- Health management of VMs and processes
- Supports scaling and rolling update

CF Application Runtime - Components

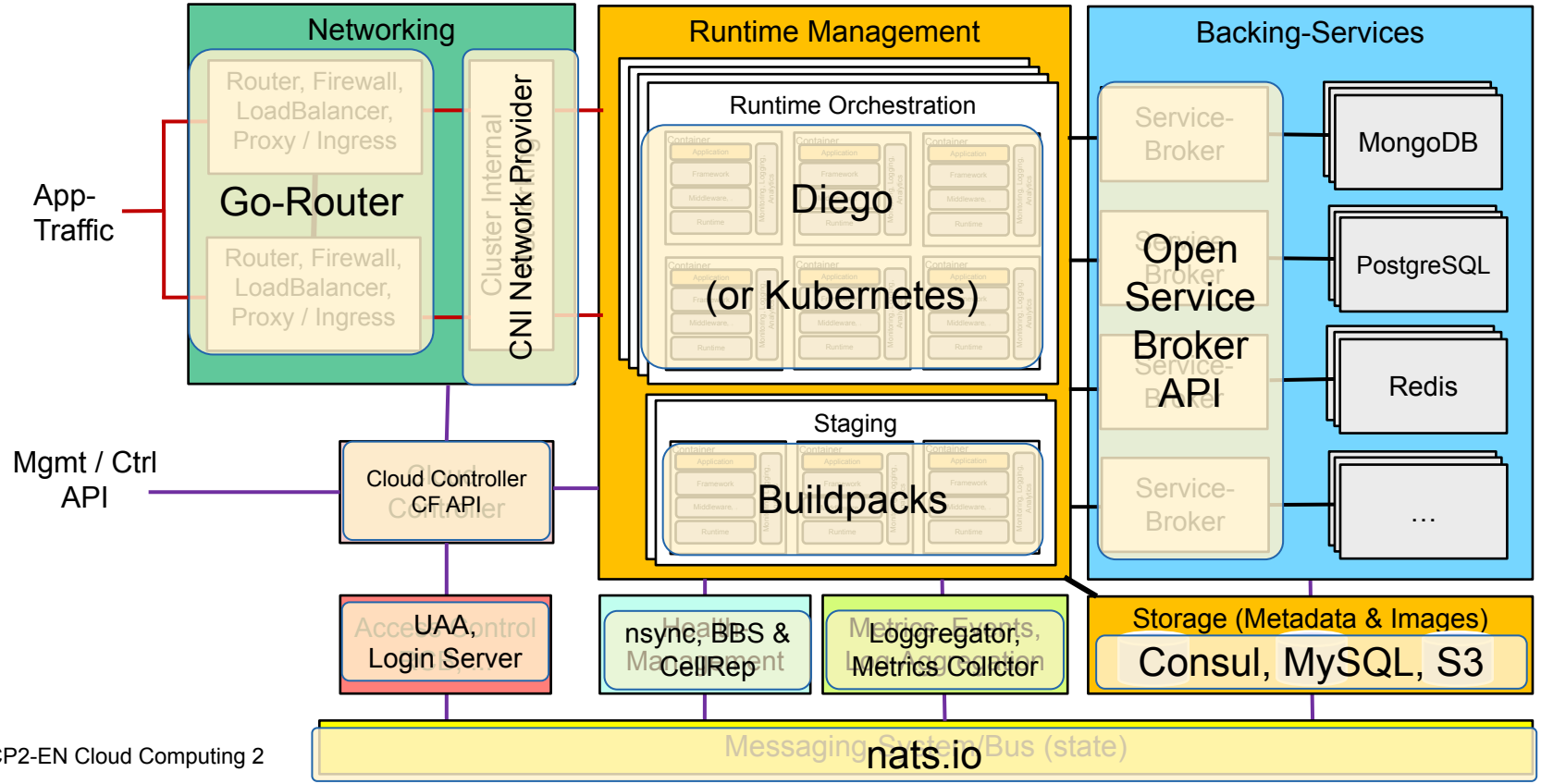
CLOUD FOUNDRY APPLICATION RUNTIME™



CF Application Runtime - Components

- Runtime based on Diego as Container (Runtime) Management System
  - Diego Brain(s) for management
  - Diego Cells for running application (Garden/runC.io based)
- Cloud-Controller
  - Provides API, Controls (ctrl plane) the application lifecycle (with Diego & buildpacks)
  - Manages (management plane) Backing Services (Marketplace /Service Brokers)
  - Maintain Multitenancy (Org, spaces, users, roles, services)
- Storage & Messaging
  - Consul for long living metadata (service-registry, dns, locks)
  - Diego BBS (Bulletin Board System) for real-time state (cluster, processes)
  - BlobStore (Filesystem, S3) to store images, BuildPacks,
  - NATS for lightweight messaging between components
- Services
  - Service Marketplace and Service-Broker to provision, bind and destroy Backing-Services (see Services Lecture)

CF App Runtime - Component Mapping



Diego Container Management System

- Diego is the 2nd generation Container Management System used in Cloud Foundry.
  - Replaces Droplet Execution Agent (DEA) and Health Manager (HM9000) and relieves the Cloud Controller from time consuming tasks.
- Main goals:
  - Independence of Container Technology (Docker, Rkt, Warden, Windows Containers, …)
  - Supporting multiple process types (not only applications)
    - batch-jobs, (reactive-)streaming-services, computational-services,...
  - Distributed Health Management
  - Modern Auction based scheduling mechanism
  - Support of very large systems, managing thousands of containers

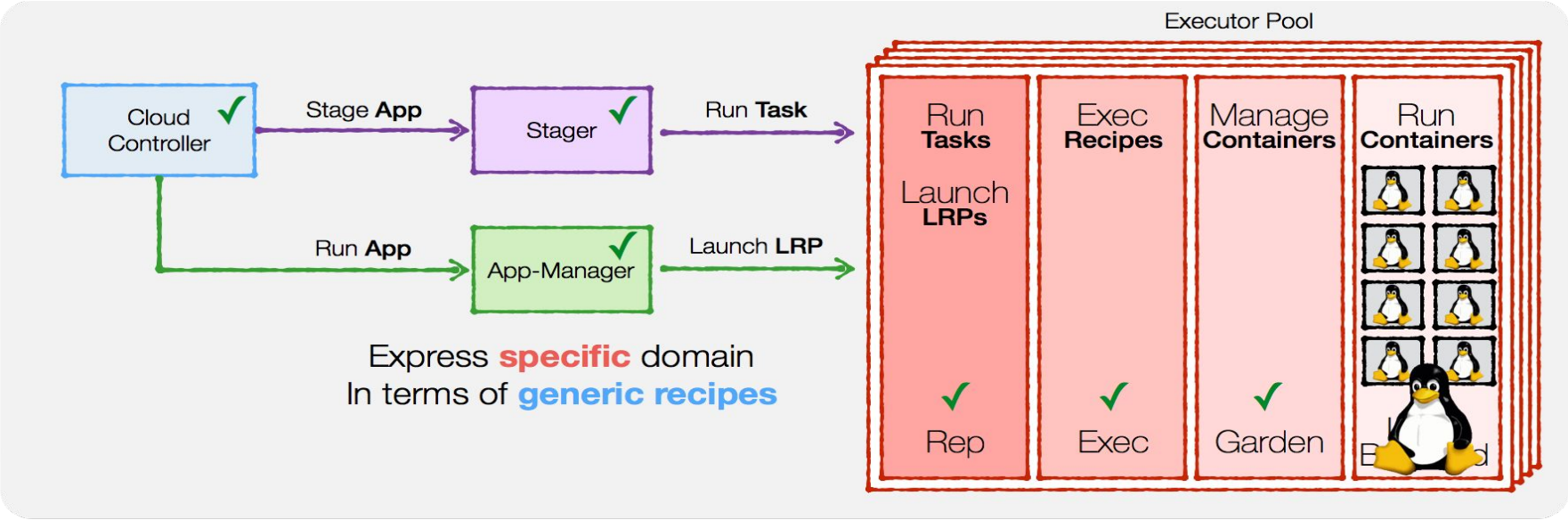
Appendix

Diego – Types of Processes

Two types of processes:

- **Tasks**
  Guaranteed to run at most once with a finite duration. Could be a Batch-Job or Script. (Simplified scheduling, no health management)
  Example: Staging Process to build container image in deployment phase
- **Long Running Process (LRP)**
  Runs continuously and may have multiple instances. Cloud Controller dictates to Diego the desired LRP (incl. # instances), which itself attempts to keep the actual LRP in sync. Example: Webapplication

Diego – Separation of concerns



Platform Independent ✓ →

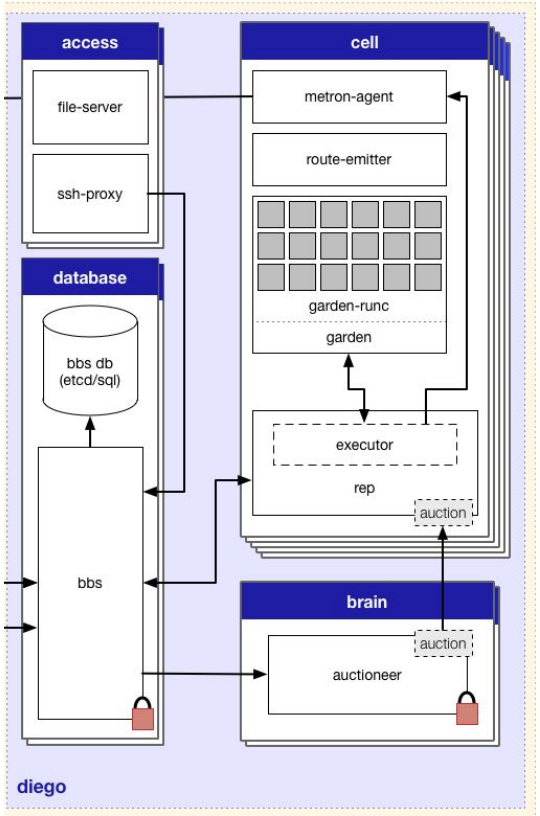
Diego – Process Mgmt - Separation of Concerns

Separation of process management in subcomponents:

- **Rep**: API to Runtime component (cell), Manages Lifecycle of Task resp. LRP, participates in auctions
- **Executor** (Exec): Executor that runs processes based on specific recipes (tree of composable actions) Examples: run staging using buildpack, run batch job, start application, …
- **Garden**: API of the Container runtime, provides platform independent and backend agnostic commands to manage containers
- **Runtime Backend**: Actual Container Runner implementations (e.g. Garden-Linux, Garden-Windows, libcontainer, runC)

Diego – Components

- **Brain**: master node
  - **Auctioneer**: responsible for holding auctions to schedule a task or LRP
  - **Converger**: Responsible to keep workers eventually consistent (health management)
- **Cell**(s): executor nodes running the processes
  - Rep/Executor/Garden/Runtime subcomponents
  - Network and Metrics/Logging components
- **BBS** (Bulletin Board System)
  - Database keeping state / cache of the actual / desires LRP instances
  - uses etcd for short term / (actual) state, sql-db for long term / (desired) state
- **Access** (VM): Provides external access
  - File-Server serves Diego assets like life-cycle binaries to cells
  - SSH-Proxy to access LRP processes



Diego – Auction-based Scheduling and Placement

The Auction process decides where Tasks and actual LRP instances are run

Three types of auctions:

- Task-Auction → Run a one-time-tasks
- LRPStart-Auction → Start additional LRP instances
- LRPStop-Auction → Stop existing LRP instances

Process:

1. BBS is asking the Brain-Auctioneer to start/stop a specific number of Tasks/LRPs
2. Brain-Auctioneer is asking the Cell-Reps about current capacity (and what they run)
3. Cell-Reps reply with a bid to start/stop instances
4. Auctioneer uses Reps response to make a placement decision
5. Reps winning the Auction will start/stop the instances

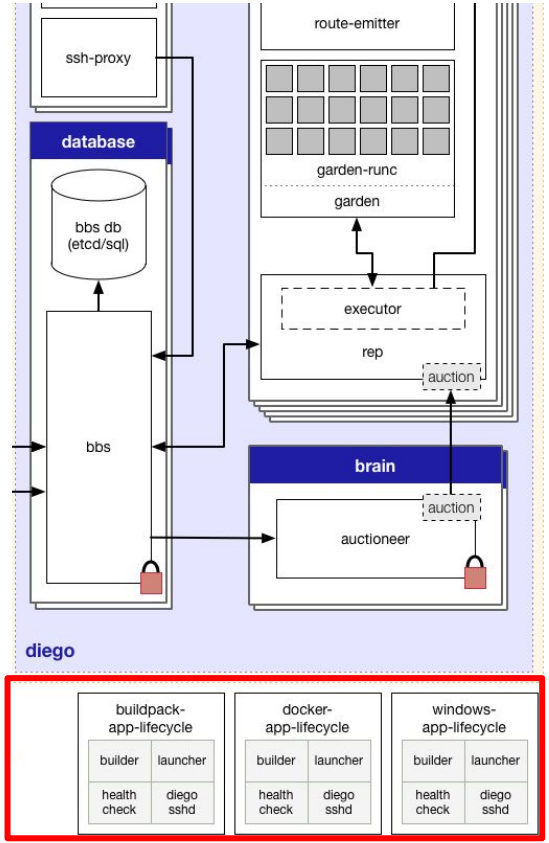
Diego – Cloud Foundry Lifecycle (Binaries)

Available platform specific Lifecycle implementations:

- Buildpack-Application Lifecycle (CF v1 buildpacks)
- Docker-Application Lifecycle (run docker images)
- Windows-Application Lifecycle (run .NET applications on Windows)

Lifecycle implementations consist of 3 components:

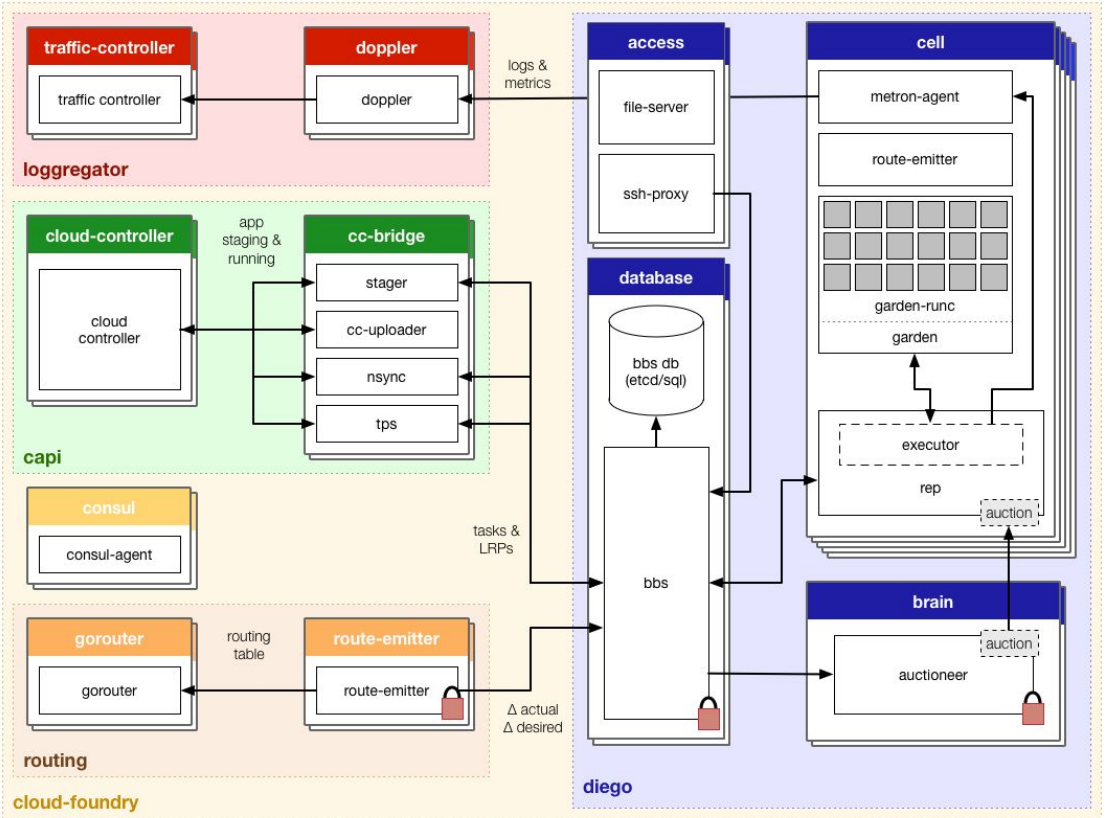
- **Builder**: responsible for the staging process
- **Launcher**: runs a LRP process (CF application)
- **Healthcheck**: performs status check for the running actual LRP



Diego – Integration in Cloud Foundry

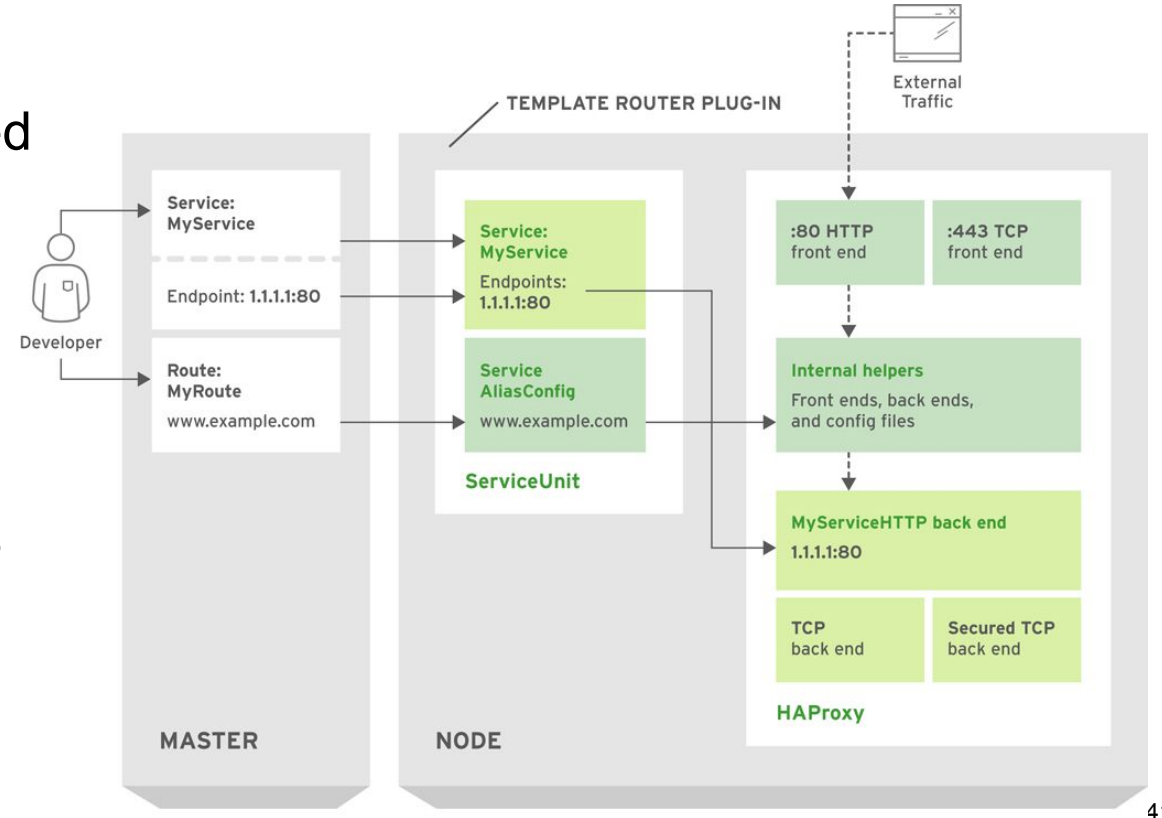
Zurich University of Applied Sciences
School of Engineering
InIT Institute of Applied Information Technology

- **doppler & traffic controller**
  Provide log aggregation & metrics
- **cc-bridge**
  Adapter to interact between Cloud Controller and Diego BBS
  - *stager* (translates staging requests to Diego Tasks)
  - *cc-uploader* (handles assests, droplets, build artifacts etc.)
  - *nsync* (sync cc state to BBS)
  - *tps* (The Process Status) reports Diego status to CC
- **route-emitter**
  Adapter between gorouter and BBS.



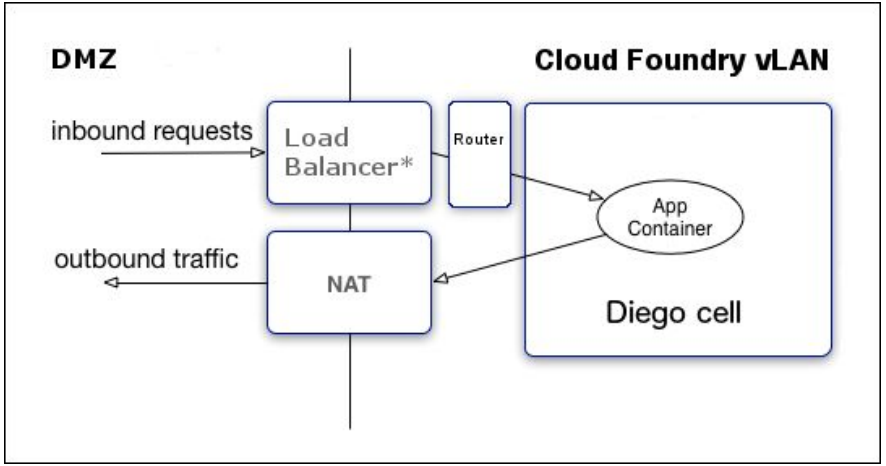
OpenShift v3 - Routing

- Application endpoints (services) are maintained in the etcd database on master node(s)
- Proxy / router config is continuously updated from the etcd database
- HA-Proxy or optional F5 router plugin



CloudFoundry v2 - Routing

- Load Balancer (HA installation)
  - Forwarding incoming traffic to multiple Routers
- Router (go-router)
  - Forwards incoming HTTPS/TCP traffic to App Containers
  - Load-balancing between multiple Containers
- NAT
  - Optional NAT component forwards outgoing traffic to public network



- **Inbound** requests flow from the load balancer through the router to the host cell, then into the application container. The router determines which application instance receives each request.
- **Outbound** traffic flows from the application container to the cell, then to the gateway on the cell's virtual network interface. Depending on your IaaS, this gateway may be a NAT to external networks.

Cloud Foundry - Container Runtime