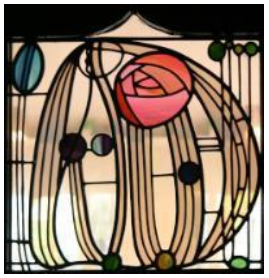
Cloud Systems
Chapter 4: Cloud Infrastructure Management

Dr Lauritz Thamsen
lauritz.thamsen@glasgow.ac.uk
School of Computing Science
University of Glasgow



First Half: Lecture Chapters

Cloud Resource Management:

1. Cloud Computing Intro
2. Virtual Machines
3. Containers
4. **Cloud Infrastructure Management**
5. Cloud Sustainability

Outline of Chapter 4: Cloud Infrastructure Management

4.1 Intro to Cloud Infrastructure Management

4.2 Cloud Operating Systems (like OpenStack)

4.3 VM Management (including Migration)

– short break (10-15 minutes) –

4.4 Infrastructure as Code and IaC Tools (like Ansible)

4.5 DevOps and CI/CD on Clouds

Outline of Chapter 4: Cloud Infrastructure Management

4.1 Intro to Cloud Infrastructure Management

4.2 Cloud Operating Systems (like OpenStack)

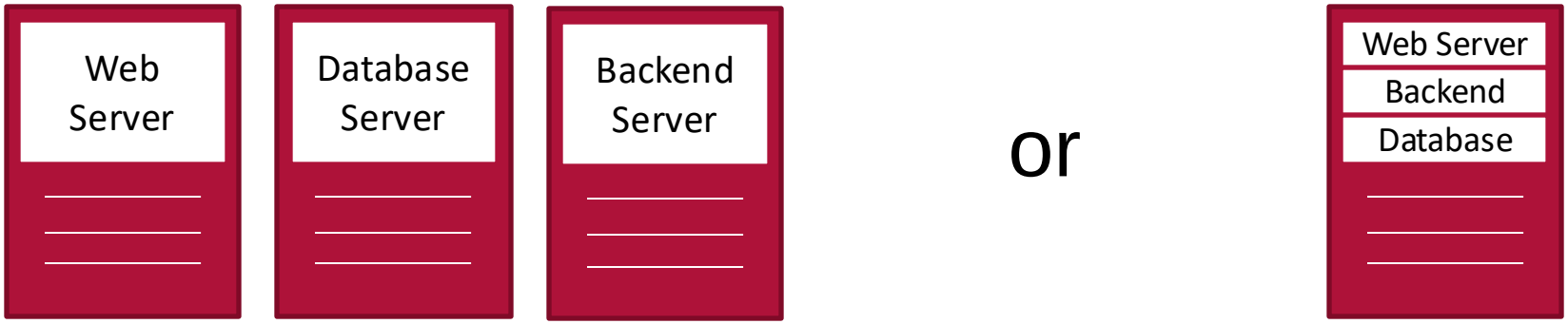
4.3 VM Management (including Migration)

– short break (10-15 minutes) –

4.4 Infrastructure as Code and IaC Tools (like Ansible)

4.5 DevOps and CI/CD on Clouds

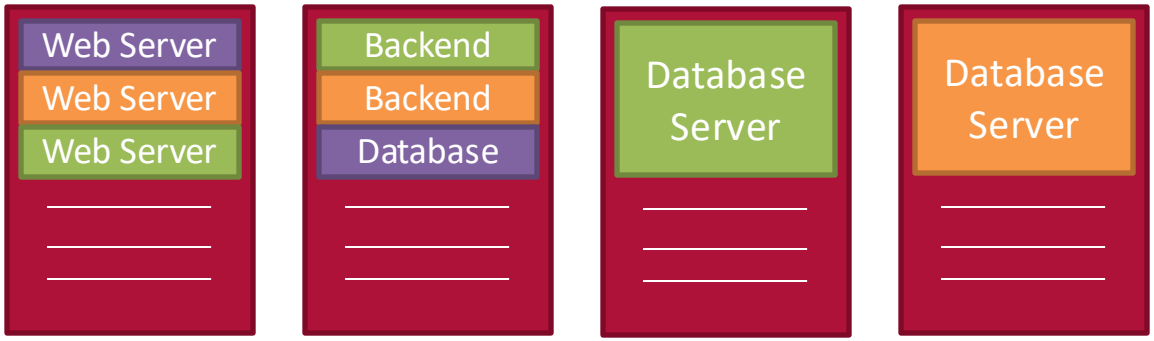
“Iron Age”: Bare-Metal Servers



- Running components of an application on one or multiple physical servers:
  - Costs and agility: Purchase, housing, maintenance
  - Fault tolerance: Servers are single points of failure
  - Resource utilization: Servers are often underutilized, but sometimes also bottlenecks (with dynamic load)

Bare-Metal Servers in One Organization

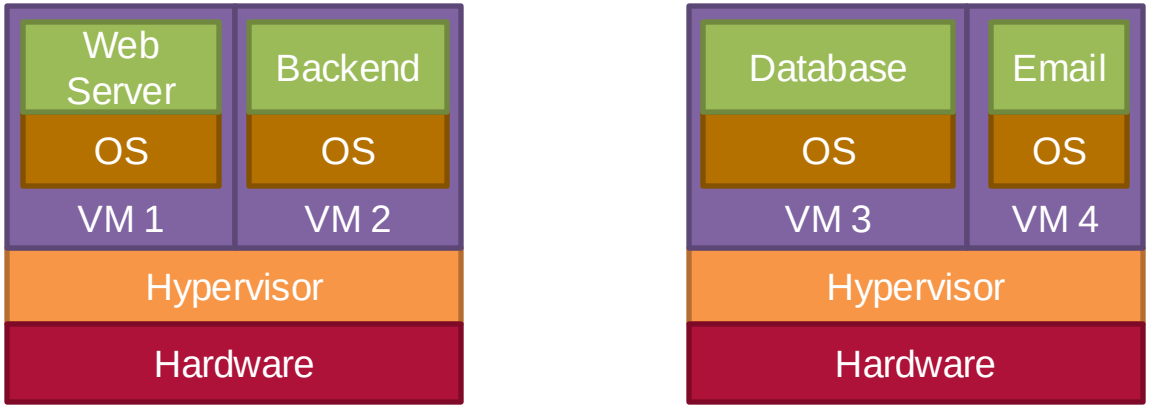
- Server consolidation: (Manually) assign components from different applications to shared physical nodes



- Only possible within one organization without isolation
- Static assignment (while loads will vary dynamically)

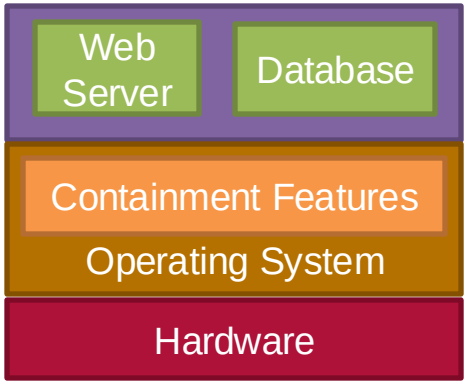
Recap: Virtual Machines

- System virtualization: Same ISA, but possibly different OS and resource configuration (#cores, RAM, storage) and isolation of applications
- Same physical hosts run multiple virtual machines
- Live migration and snapshots of VMs allow for load balancing and fault tolerance



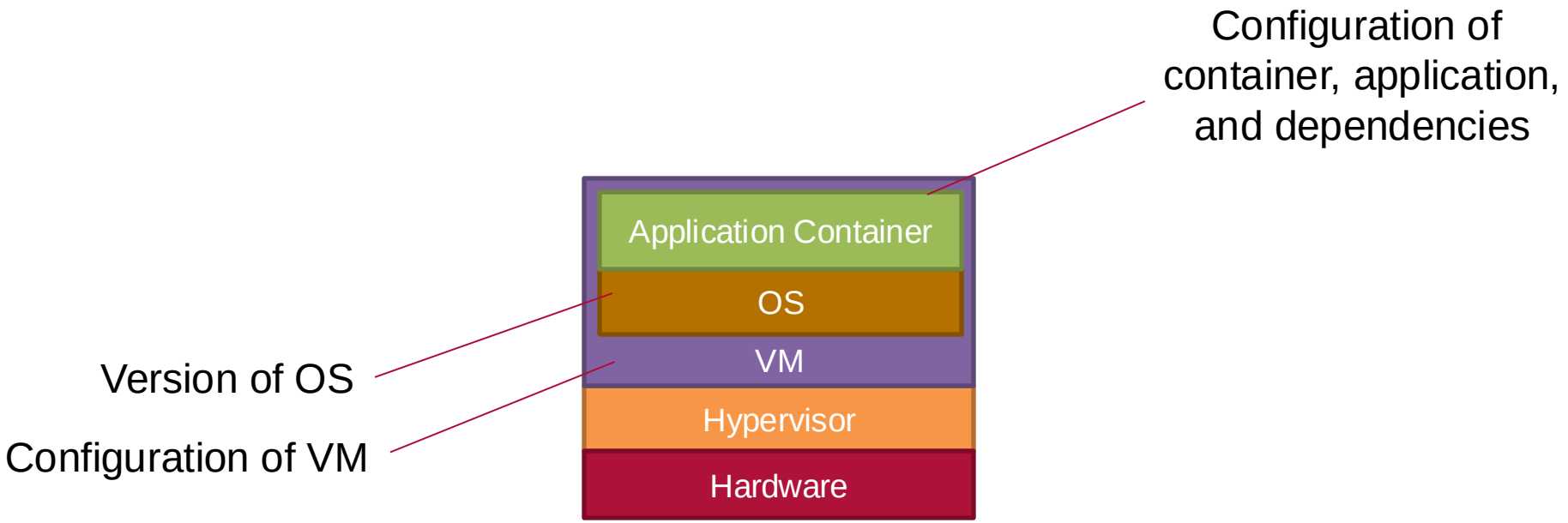
Recap: Containers

- Motivation: VM images are large (contain OS) and starting VMs takes longer than processes (~10x)
- Containers: Same OS kernel, but different resource configuration and isolation of container processes
- Containment using Linux kernel features such as chroot, cgroups, and namespaces



Management of Cloud Resources

- Major remaining problems in the “Cloud Age”: server sprawl, configuration drift, snowflake servers… → technical debt!



Outline of Chapter 4: Cloud Infrastructure Management

4.1 Intro to Cloud Infrastructure Management

4.2 Cloud Operating Systems (like OpenStack)

4.3 VM Management (including Migration)

– short break (10-15 minutes) –

4.4 Infrastructure as Code and IaC Tools (like Ansible)

4.5 DevOps and CI/CD on Clouds

Cloud Operating Systems (1/3)

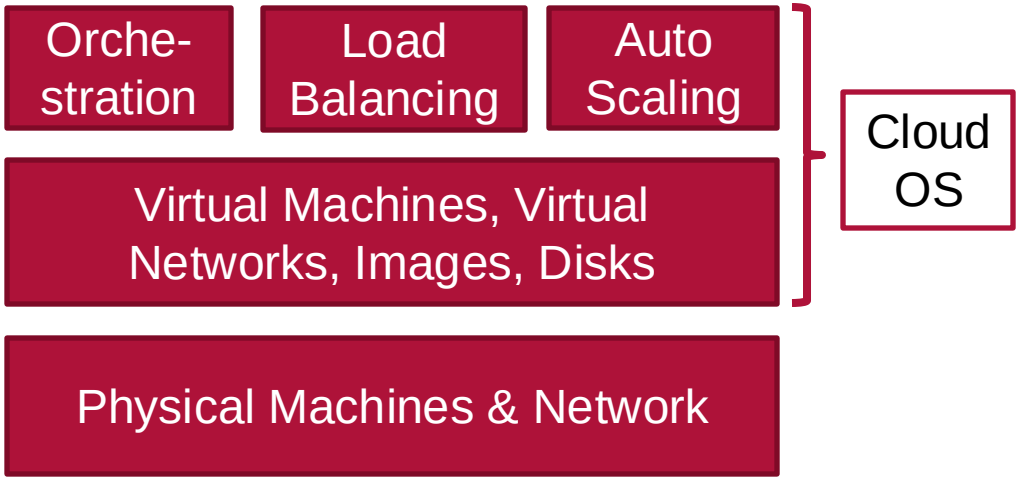
- Virtualization is immensely useful, but managing many virtual machines manually is impractical
- Therefore: Cloud Operating Systems
  - Manage large pools of compute, storage, and networking resources
  - Provides dashboards and APIs for datacenter operators (administration) and users (provisioning) that abstracts infrastructures (e.g. the different hypervisors)

- Open-source systems: OpenStack, OpenNebula
- Commercial public clouds: e.g. EC2, GCE, Azure, Alibaba Cloud, Digital Ocean

Cloud Operating Systems (2/3)

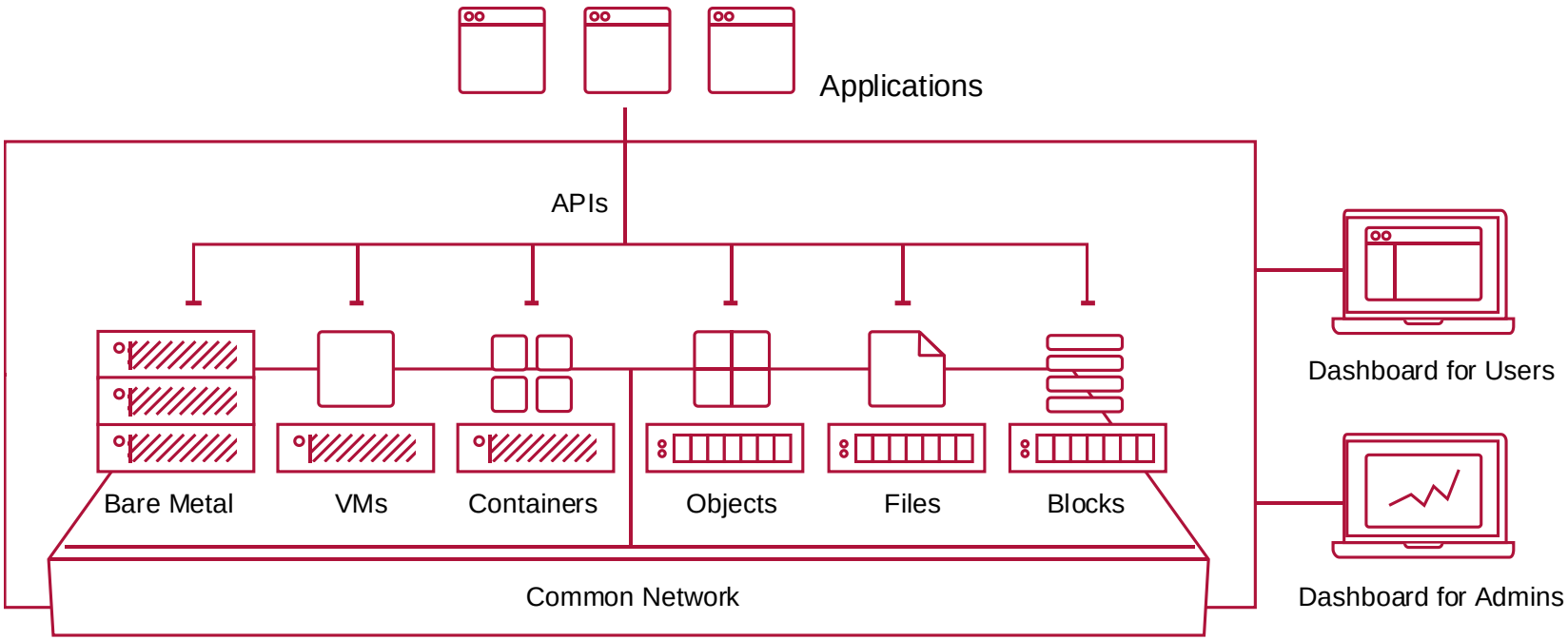
- Basic functions: User Interface and APIs for
  - spawning and maintaining VMs,
  - virtual networking,
  - managing images and virtual disks

- Advanced functions:
  - orchestration,
  - load balancing,
  - and auto scaling



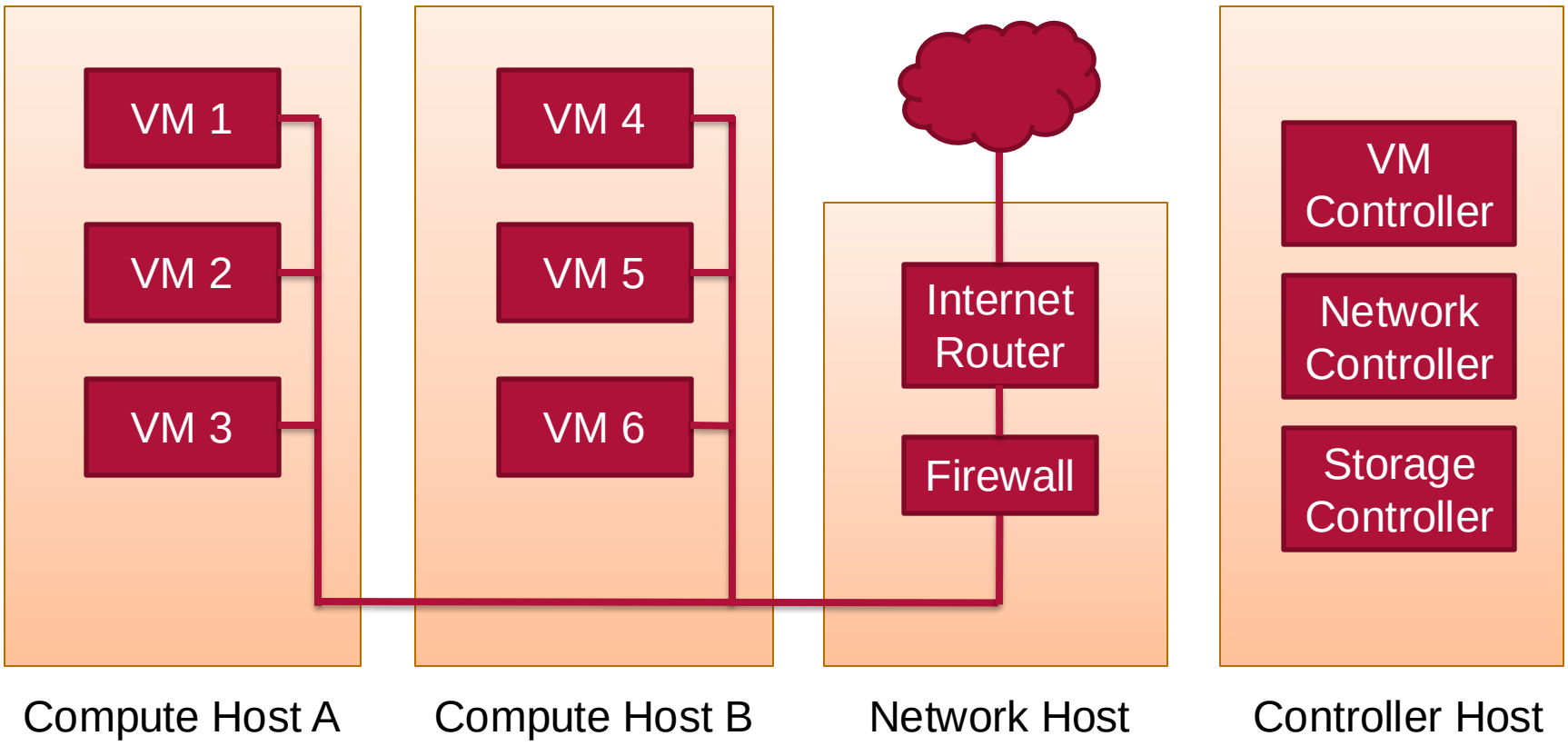
Cloud Operating Systems (3/3)

- Interfaces to manage and provision virtual resources



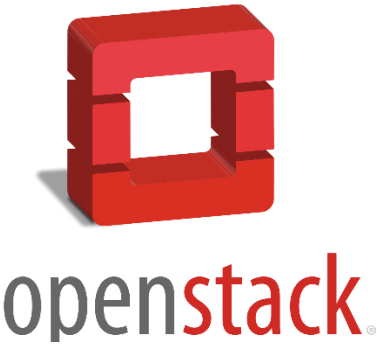
Managed Cloud Environment

- Different roles for physical hosts: compute hosts, storage hosts, network hosts, and controller host



OpenStack

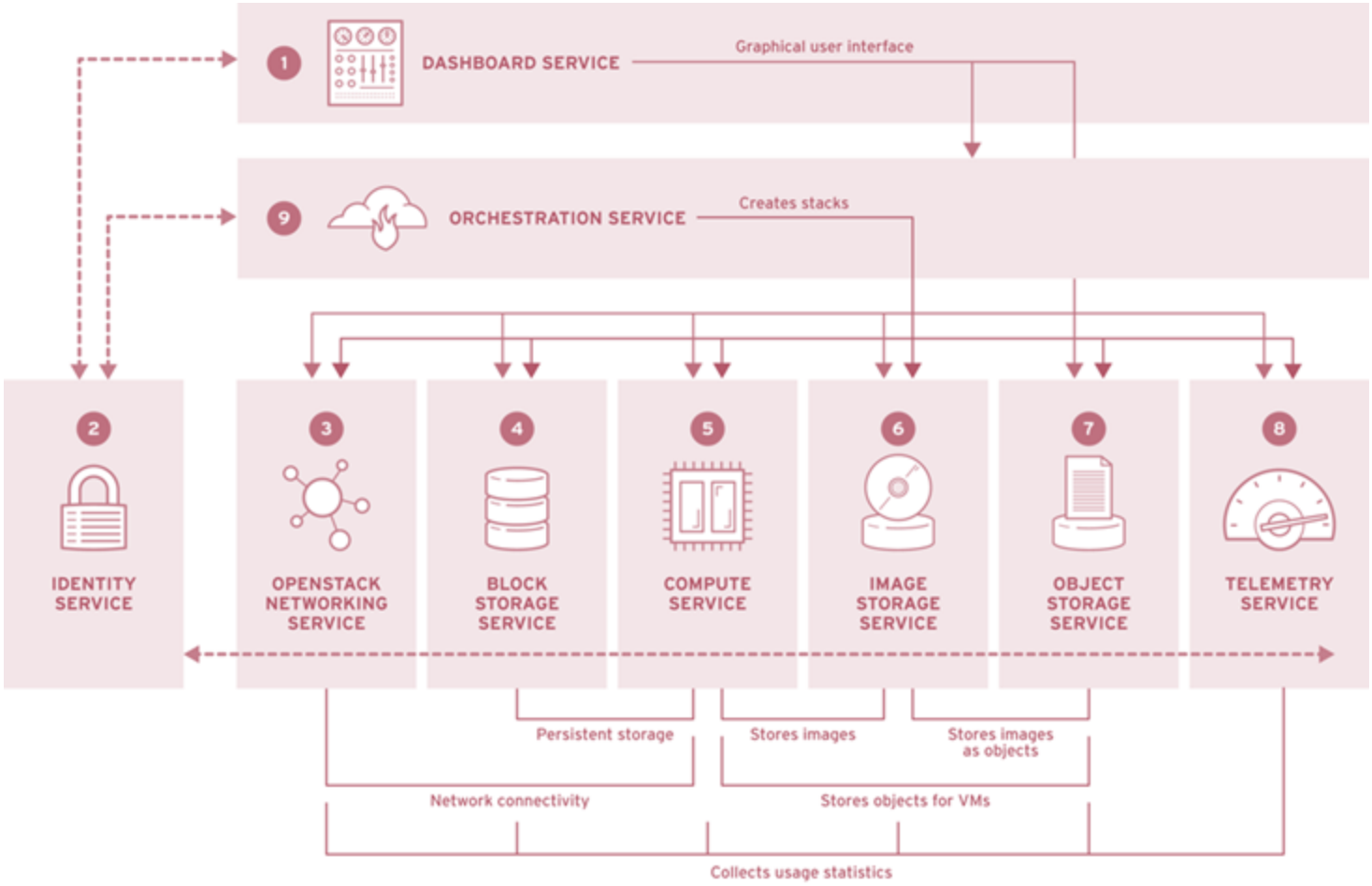
- Open source cloud operating system
- Mostly deployed as IaaS, providing virtual machines and other resources to users



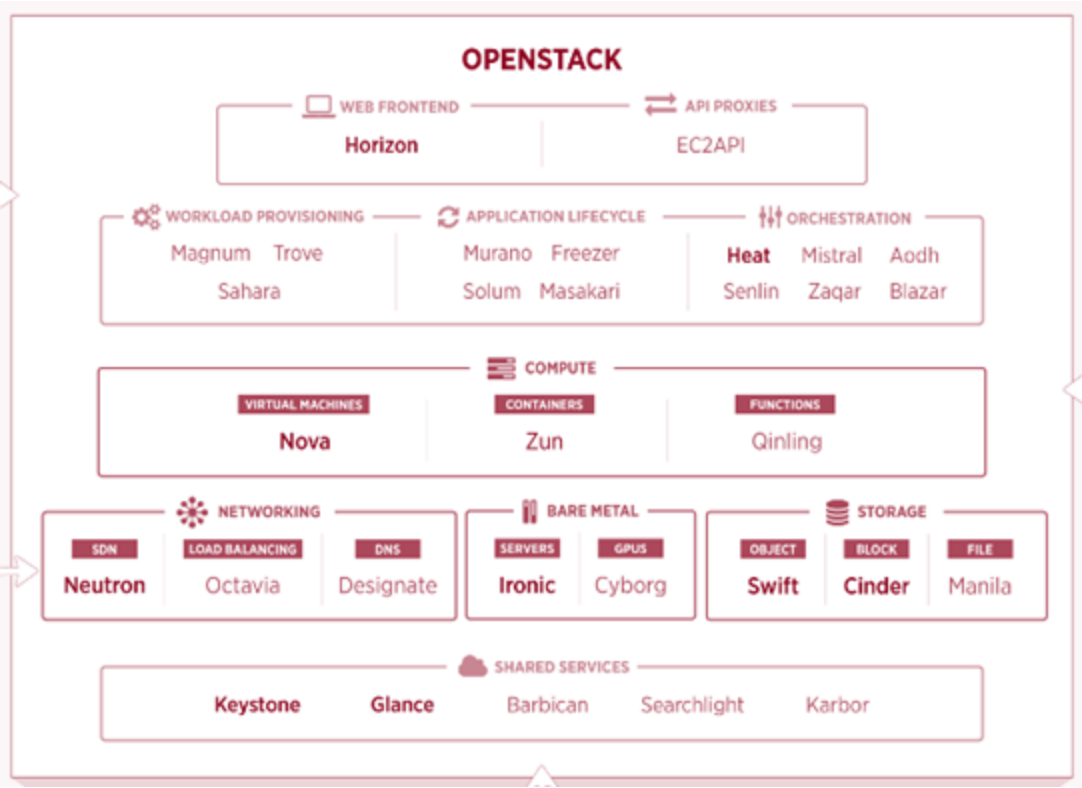
- Started in 2010 by Rackspace Hosting and NASA
- Contributions from IBM, Red Hat, HP, Cisco, Google, Oracle, EMC, VMware

- Most popular open cloud platform in both industry and academia

Core Components of OpenStack (1/2)



Core Components of OpenStack (2/2)



Compute Service: *Nova*

- Allows to create and manage virtual machines on demand from images
- Schedules virtual machines to run on physical nodes
- Defines drivers that interact with underlying virtualization mechanisms

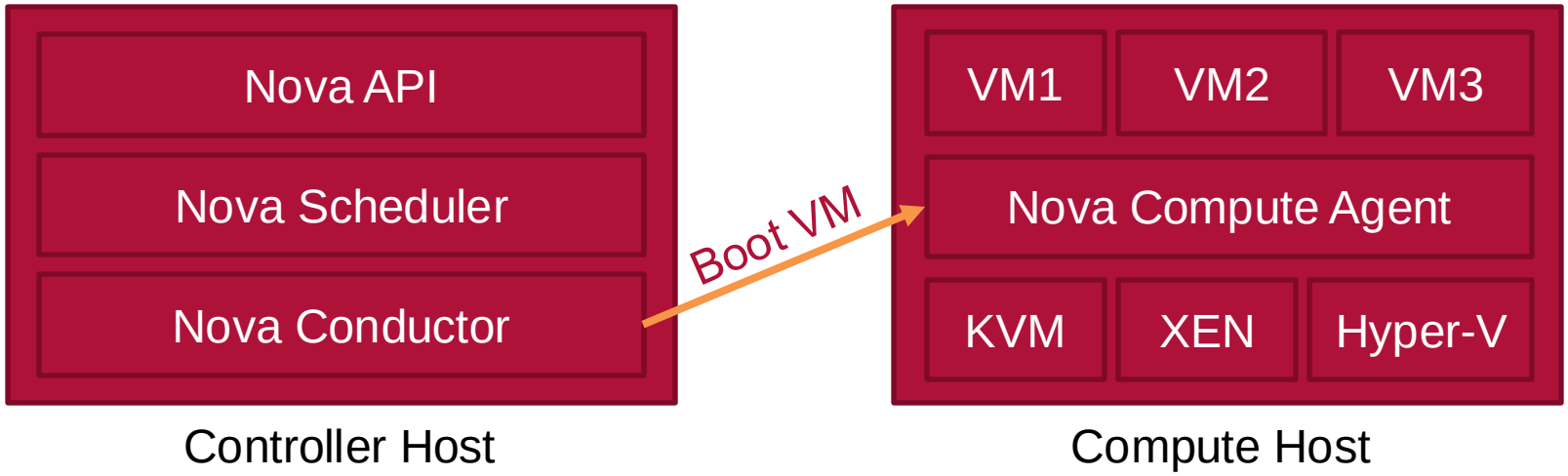
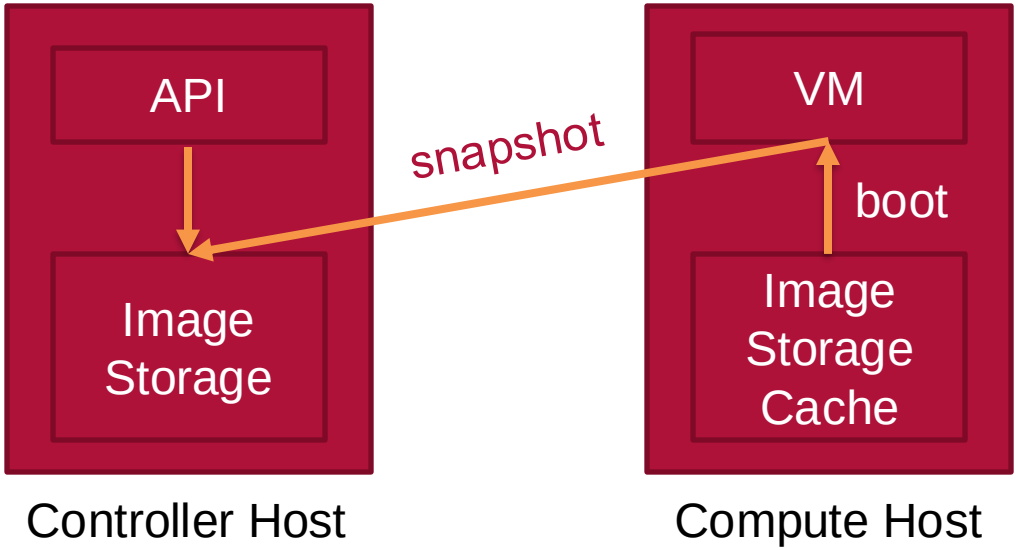


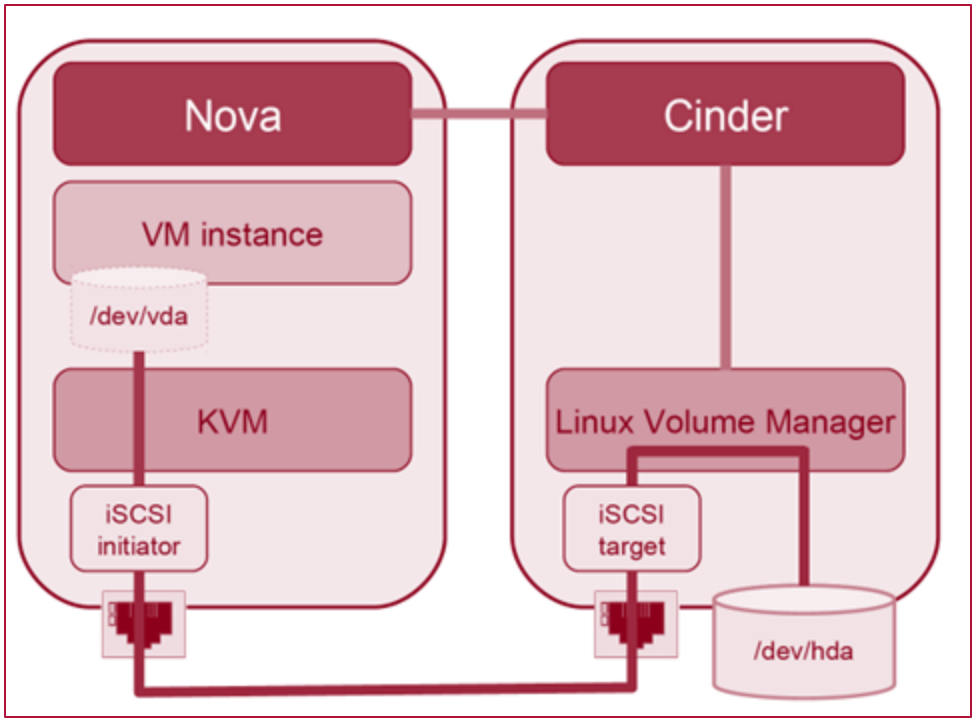
Image Service: *Glance*

- Registry for virtual disk images
- Users can add new images or take snapshots of existing VMs
- Snapshots can be used as templates for new servers



Block Storage Service: *Cinder*

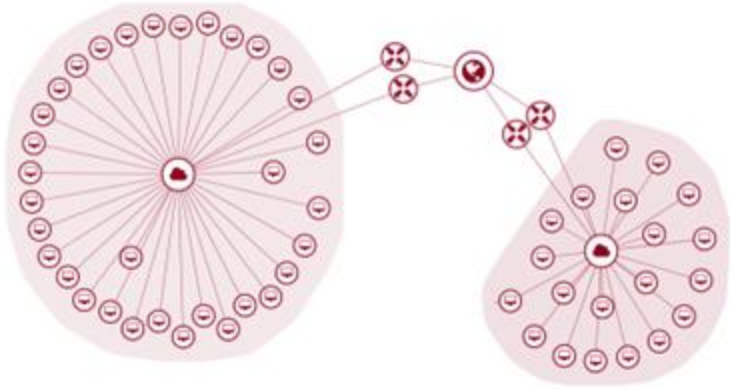
- Provides network-backed virtual block storage volumes
- Enables live migration of VMs and supports replication



Virtual Networking: *Neutron*

- Provides API for networking between virtual machines and physical network

- Interconnects virtual machines
  - within one hypervisor
  - between several hypervisors
  - with the Internet



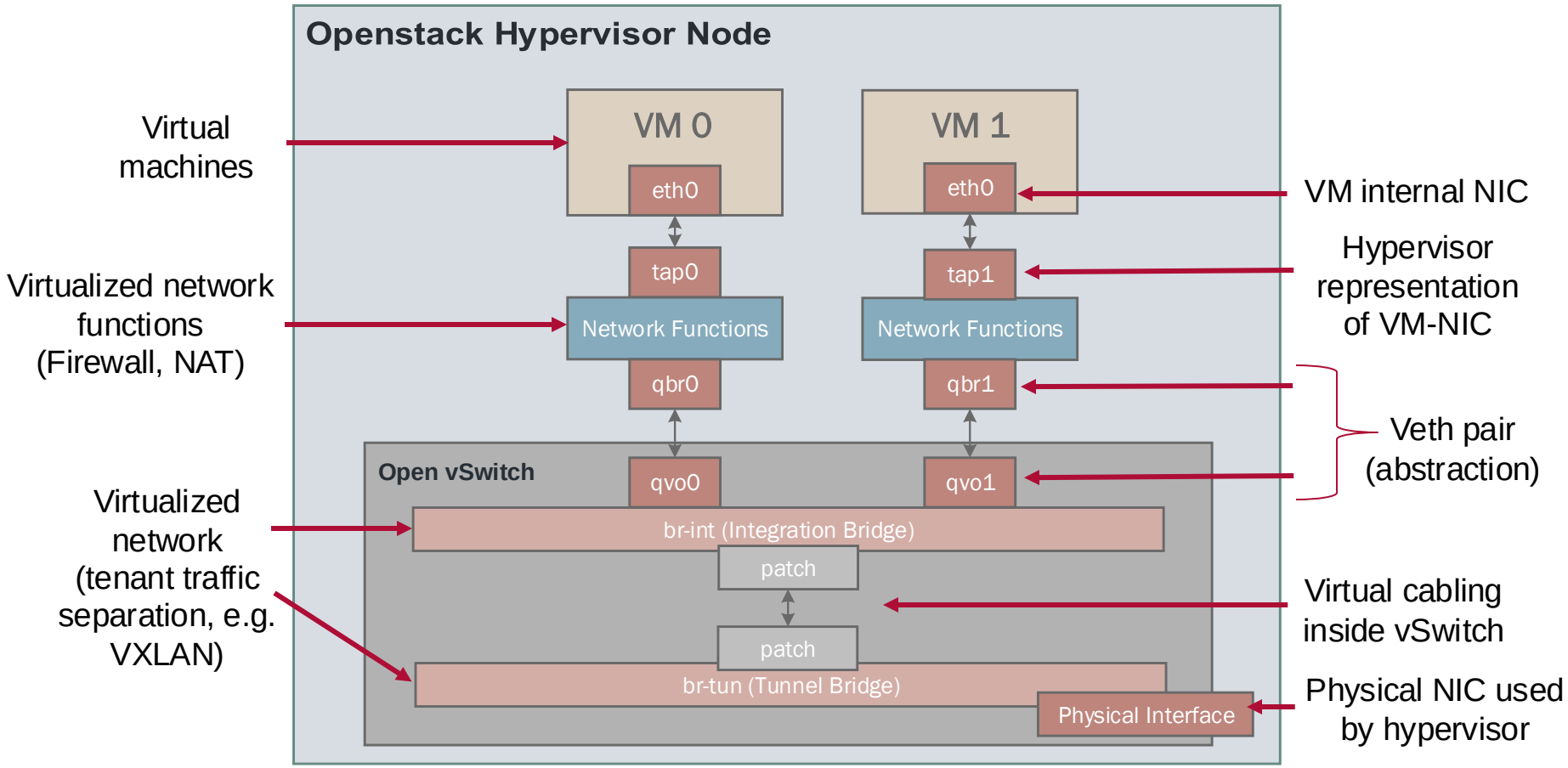
- Provides additional network functions
  - Like firewalls, VPN, public IPs …

Virtual Switches & SDN

- Virtual switches: Virtual switching stack that enables network automation through programmatic extensions and standard protocols (e.g. OpenFlow)

- Software-Defined Networking (SDN)
  - Goal: Simplify network administration
  - Separation of control plane from data plane
  - Allows on the fly configuration of switches (controller-based decision making)

Virtual Networking in OpenStack



Outline of Chapter 4: Cloud Infrastructure Management

4.1 Intro to Cloud Infrastructure Management

4.2 Cloud Operating Systems (like OpenStack)

4.3 VM Management (including Migration)

– short break (10-15 minutes) –

4.4 Infrastructure as Code and IaC Tools (like Ansible)

4.5 DevOps and CI/CD on Clouds

VM Snapshots

- VM snapshots
  - Point-in-time copy that preserves memory and disk state in a VM image
  - Useful for roll-back or migration (and potentially for backups)
- Virtual snapshots
  - Reduce overhead for taking snapshots
  - *Copy-on-write* when changing the original, allowing to store just deltas (for one VM)
  - Not ideal for backups (as stored on the same host)

VM Suspending vs. Pausing

- Suspending:
  - Full VM state is written to disk, so only disk resources and networking resources remain required
  - Resuming takes little time (way less than booting)

- Pausing:
  - Only the CPU activity is halted, so the VM does not run but does require main memory (and other resources)
  - Resuming takes very little time (less than resuming a suspended VM)

VM Migration

- Migration: Move VM from one physical host to another
- Motivation:
  - Fault mgmt.: Host reports HW errors, must be shut down
  - Maintenance: Update of BIOS, hypervisor, …
  - Load balancing: Move workload to another physical host

- **Cold migration**: Start a VM based on image elsewhere
- **Warm migration**: Suspend VM to disk; copy image and suspended state; resume execution elsewhere
- **Live migration**: Copy full state while VM executes

Live VM Migration

- Desired property: Live migration
  - No shutdown of the virtual machine
  - No disruption of the service, no impact for the user
  - → Minimize *downtime* and *total migration time*

- What needs to be migrated?
  - **Memory pages**: Ensure consistency between the memory state of the source and destination VMs
  - **Network resources**: Maintain open connections
  - **Storage resources**: Storage must be accessible both at the source and destination VM

Strategies for Memory Migration

1. Push
   - Source VM continues running, sends pages to destination
   - Memory must potentially be sent multiple times

   → Minimum downtime, potentially long migration time
2. Stop-and-copy
   - Source VM stopped, pages are copied to destination VM
   - Destination VM is started after having received all pages

   → Short overall migration time, long downtime
3. Pull
   - Execute new VM, pull accessed pages from source

   → Performance depends on number of page faults

Strategy for Memory Migration in XEN[15]

- XEN pursues *pre-copy* strategy for memory migration
  - Usage of a *push* and a *stop-and-copy* phase
  - Balances short downtime with short total migration time
- Iterative approach: Multiple rounds of push phase, then short stop-and-copy phase in the end
- Similar approach in VMware vMotion: vMotion also capable of slowing down source VM in case memory changes too fast for network transfer

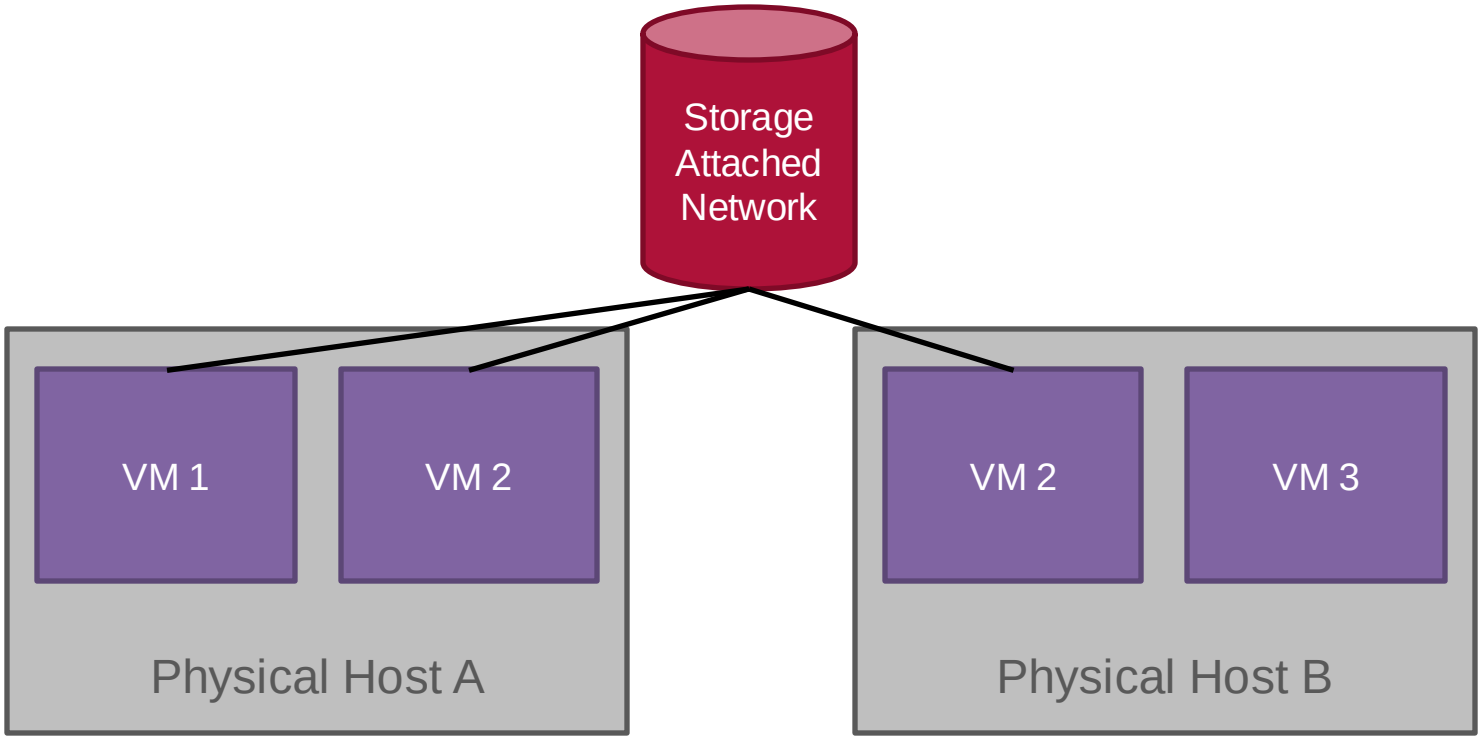
Strategy for Network Migration in XEN[15]

- Assumption: Source and destination VM are on same IP subnet (no IP-level routers involved)

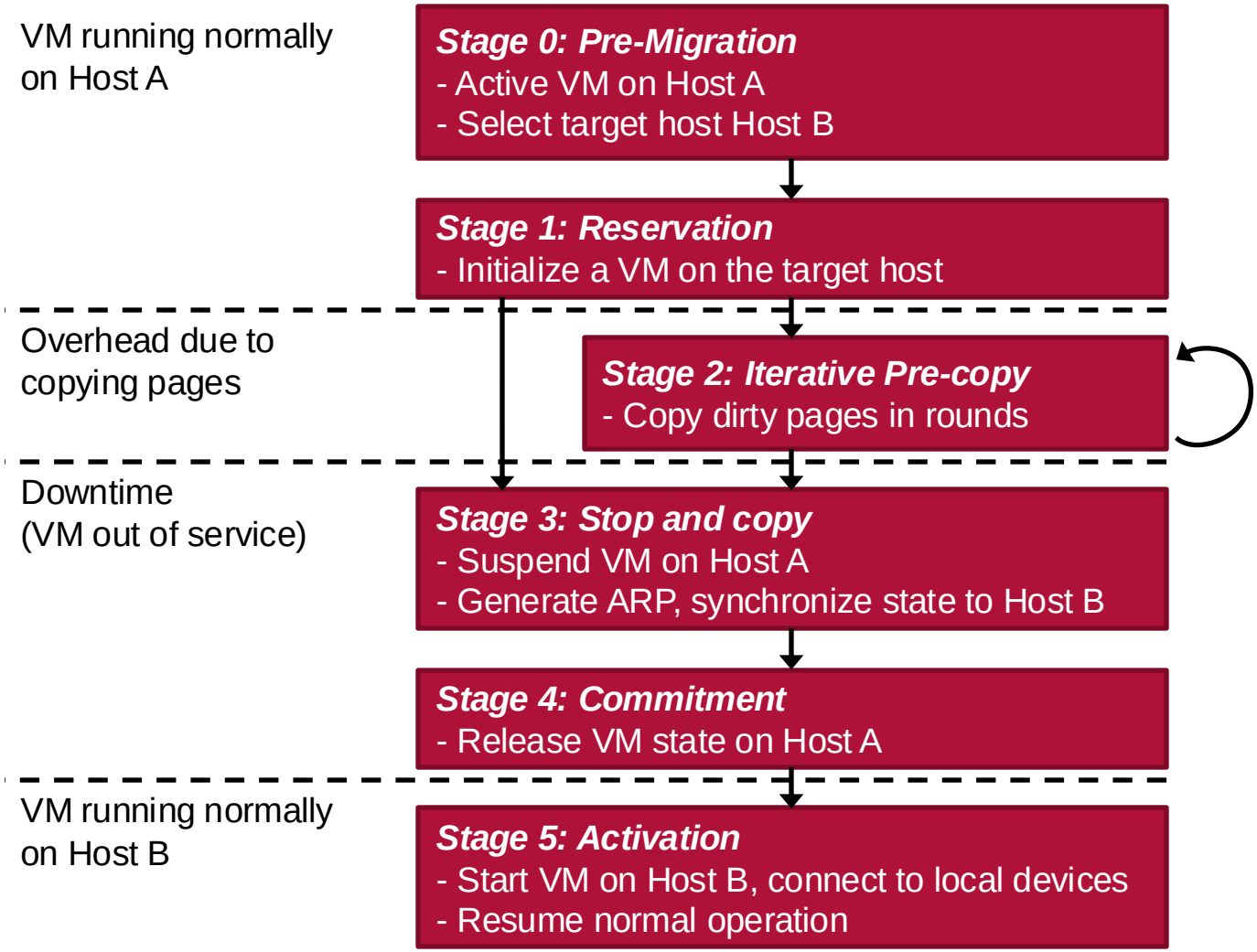
- Approach for migration:
  - Destination VM will have new MAC but old IP address
  - After memory transfer, source host sends unsolicited ARP reply
    - Broadcast message to all hosts on the same network
    - Hosts will remove IP ←→ MAC mapping from caches
  - Upon new ARP request for IP, destination VM will return its MAC address

Strategy for Storage Migration in XEN[15]

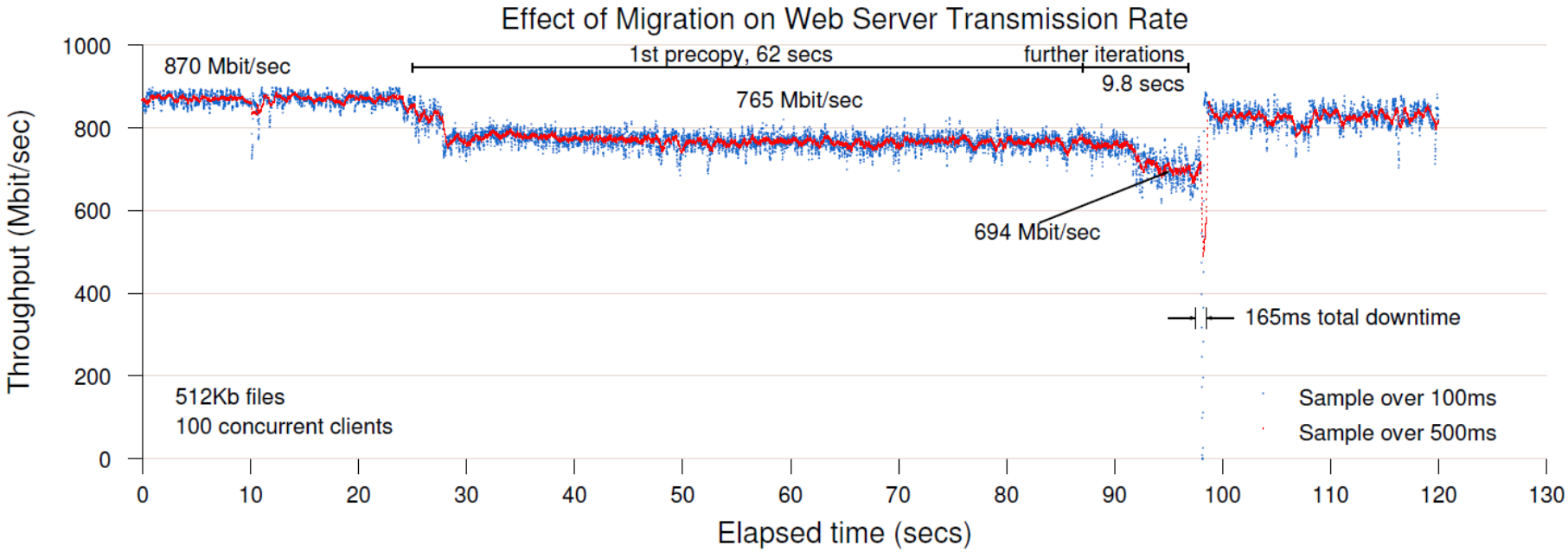
- XEN assumes VMs to reside on storage network (→ virtual / distributed storage)
  - Migration by rerouting network traffic
  - Solved with network migration



XEN Migration Timeline[15]



XEN Migration Performance Figures[15]



- Migration of a running web server VM (800 MB RAM)
  - Web server continuously serves 512 KB file to 100 clients

Outline of Chapter 4: Cloud Infrastructure Management

4.1 Intro to Cloud Infrastructure Management

4.2 Cloud Operating Systems (like OpenStack)

4.3 VM Management (including Migration)

– short break (10-15 minutes) –

4.4 Infrastructure as Code and IaC Tools (like Ansible)

4.5 DevOps and CI/CD on Clouds

Outline of Chapter 4: Cloud Infrastructure Management

4.1 Intro to Cloud Infrastructure Management

4.2 Cloud Operating Systems (like OpenStack)

4.3 VM Management (including Migration)

– short break (10-15 minutes) –

4.4 Infrastructure as Code and IaC Tools (like Ansible)

4.5 DevOps and CI/CD on Clouds

Motivation

- Manual configuration of servers usually leads to
  - *Configuration Drift*: configuration of servers changed manually over time, without documentation
  - and *Snowflake Servers*:
    - Have a unique configuration
    - Are not reproducible when hardware dies
    - Are difficult to mirror for a test environment
    - Deliberate vs. default configuration
    - Host-specific vs. general configuration

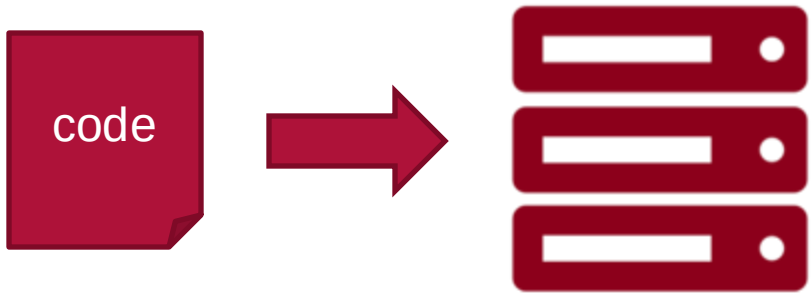
Open Questions

- What if a server fails?
- How to mirror a production environment for testing?
- How to keep a cluster of servers consistent?

- Disk snapshots allow to recover and mirror servers, but
  - still not straightforward to understand the state of servers
  - do not allow managing multiple servers that are mostly similar, but specific in a few configurations

Infrastructure-as-Code (1/2)

- Goal: Make systems easily reproducible, support testing, and aid understanding configurations
- Solution: One single source of truth with a clear declaration of host-specific configuration → infrastructure definition files
  - that can be executed
  - that can be versioned
  - that can be shared



- Only use these definitions and automation tools, without manually updating single servers

Infrastructure-as-Code (2/2)

- Define servers, networking, and other infrastructure elements in source code:
  - Configuration of environments
  - Dependencies with specific versions

- Automation tools built around infrastructure definitions ➔ **immutable infrastructure**: (easily) rebuild servers instead of updating running servers

Ansible

- Automation language and engine

- Resource orchestration
  - i.e. allocate and provision new servers
- Configuration management
  - i.e. configure servers

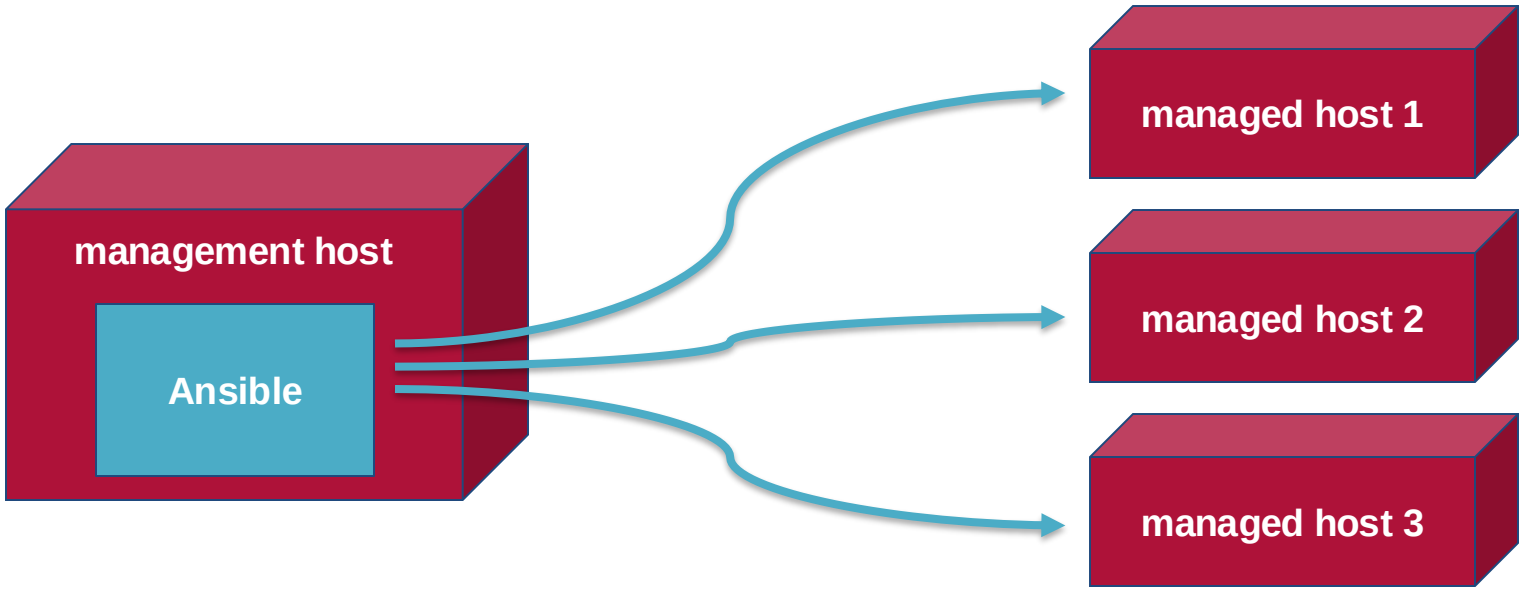
- Started in 2012, acquired by Red Hat in October 2015

- Well-documented conventions and best practices, but no artificial limits, so users can choose what to ignore

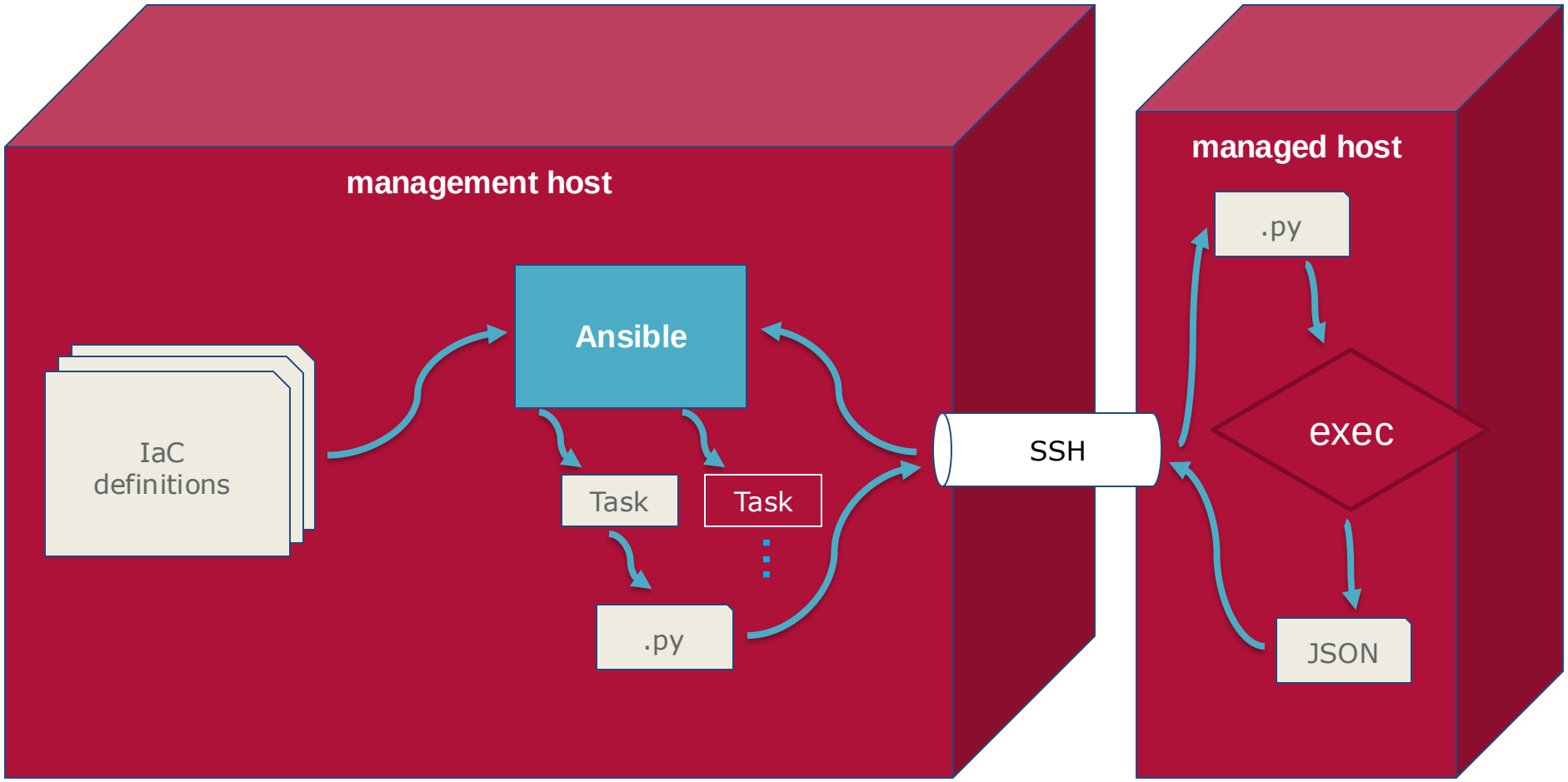
Ansible

- Multi-platform
  - Operating systems: Linux, MacOS, Windows
  - Orchestration-related tools: LXC, libvirt, VMware, …
  - IaaS services: AWS, Azure, OpenStack, …
- Well-established technologies as the foundation
  - Python
  - SSH (PowerShell remoting, respectively)
  - YAML + Jinja templating
- Design: Declarative and agent-less

Agent-Less Architecture (1/2)



Agent-Less Architecture (2/2)



Ansible Tasks (1/2)

- Declarative
  - i.e., define the desired state
- Opposed to imperative
  - i.e., define how to achieve the desired state
- Implies idempotency
  - i.e., applying the configuration again leads to same state
    - # that's the intention at least; you can, of course, create non-idempotent tasks

```
# an example "task":
- name: "enforce permissions for .ssh directory of {{ ansible_user_id }}"
  # ↑ "name" is optional but strongly encouraged
  file: # ← the module name, ↓ followed by module parameters
    dest: "{{ ansible_user_dir }}/.ssh"
    state: directory
    mode: u+rX,go-rwx
    recurse: yes
```

Ansible Tasks (2/2)

But to which hosts should this task be applied to?

```
# an example "task":
- name: "enforce permissions for .ssh directory of {{ ansible_user_id }}"
  # ↑ "name" is optional but strongly encouraged
  file: # ← the module name, ↓ followed by module parameters
    dest: "{{ ansible_user_dir }}/.ssh"
    state: directory
    mode: u+rX,go-rwx
    recurse: yes
```

Ansible Playbooks (1/2)

- Ansible's execution unit: Playbooks map everything to hosts
  - Tasks
  - Roles
  - Variables

```
# file: my_playbook.yml

# a "Play":
- name: configure internal Web servers
  hosts: webservers
  roles:
    - webserver
  tasks:
    - name: forbid external connections
      iptables:
        chain: INPUT
        source: !192.168.1.1/24
        jump: DROP
```

- Further features include:
  - Rolling updates
    - Configure only n hosts at a time
    - Configure only x% of host at a time
  - Tolerate percentage of hosts where configuration fails
  - …

Ansible Playbooks (2/2)

But how does Ansible know the hosts "webservers"?

```
# file: my_playbook.yml

# a "Play":
- name: configure internal Web servers
  hosts: webservers
  roles:
    - webserver
  tasks:
    - name: forbid external connections
      iptables:
        chain: INPUT
        source: !192.168.1.1/24
        jump: DROP
```

Ansible Inventory

```
# no FQDNs required, as long as
# ``ssh <hostname>`` works (~/.ssh/config)
mail

[webservers]
w1.example.com
w2.example.com


[dbservers]
d1.example.com
d2.example.com

[appservers]
# we can override variables:
a01.example.com www_user=web
# we can use a pattern-like syntax:
a[02-12].example.com

# we can define variables for groups:
[appservers:vars]
www_user=www-data

# a group of groups
[rootservers:children]
appservers
dbservers
```

```
# alternatively, we can use YAML:

all:
  hosts:
    mail.example.com:
  children:
    webservers:
      hosts:
        w1.example.com:
        w2.example.com:
    dbservers:
      hosts:
        d1.example.com:
        d2.example.com:
  ...
```

- Inventories can also be obtained from external scripts / services (e.g. IaaS clouds)

Ansible Variables (1/2)

- Variables can be defined in many places
  - Playbooks, Tasks, Inventory, CLI args, roles, facts, …
  - But scattering variables all over the place is discouraged, of course
  - e.g., using role, group and host variables can suffice
- "facts"
  - A heap of auto-detected variables: e.g., Uptime, chroot?, user home, SSH host key, Python version, total and free memory, host name, FQDN, environment variables, block devices, mounts, interfaces, BIOS version, processor specs, OS name and version, …
- Jinja templating also in variables files
  - e.g., app_user: "{{ ansible_env.SUDO_USER|default(ansible_user_id) }}"

Ansible Variables (2/2)

- Host and group variables
  - e.g., in project directory:

```
# e.g., set inventory to ./inventory.ini:
.ansible.cfg
inventory.ini
my_playbook.yml
group_vars/
  # applies to all hosts:
  all.yml
  # applies to webservers:
  webservers.yml
host_vars/
  mail.yml
```

```
# no FQDNs required, as long as
# ``ssh <hostname>`` works (~/
mail

[webservers]
w1.example.com
w2.example.com


[dbservers]
d1.example.com
d2.example.com

[appservers]
# we can override variables:
a01.example.com www_user=web
# we can use a pattern-like syntax
a[02-12].example.com

# we can define variables for group
[appservers:vars]
www_user=www-data

# a group of groups
[rootservers:children]
appservers
dbservers
```

Run a Playbook

$ cat my_playbook.yml

```
- name: test SSH connection
  hosts: pim:static
  tasks:
    - name: test if logging into host and running a module remotely works
      ping: # does not require any arguments
```

$ ansible-playbook my_playbook.yml

```
PLAY [test SSH connection] ****************************************************
TASK [Gathering Facts] ********************************************************
ok: [static]
ok: [pim]

TASK [test if logging in and running a module remotely works] ***************
ok: [pim]
ok: [static]

PLAY RECAP ********************************************************************
pim                        : ok=2    changed=0    unreachable=0    failed=0
static                     : ok=2    changed=0    unreachable=0    failed=0
```

Outline of Chapter 4: Cloud Infrastructure Management

4.1 Intro to Cloud Infrastructure Management

4.2 Cloud Operating Systems (like OpenStack)

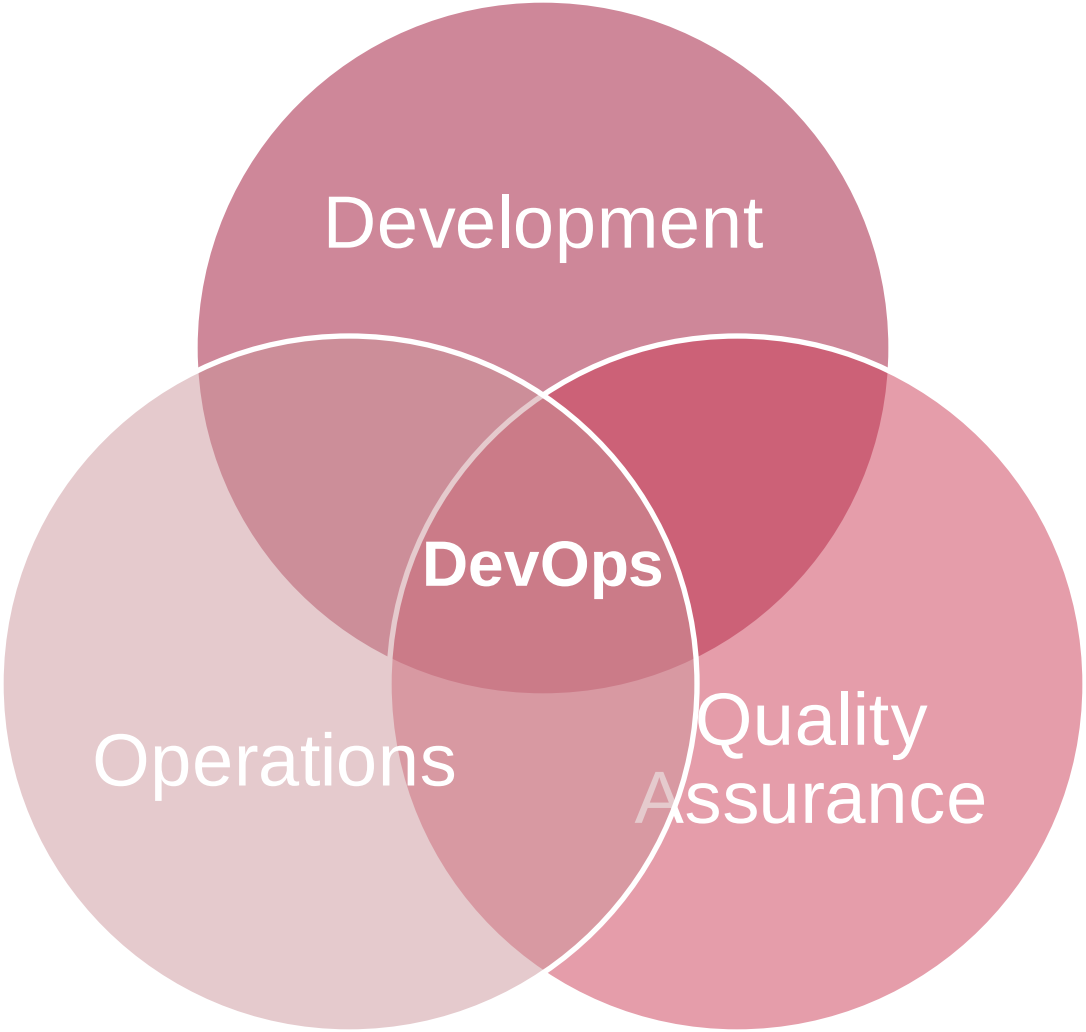
4.3 VM Management (including Migration)

– short break (10-15 minutes) –

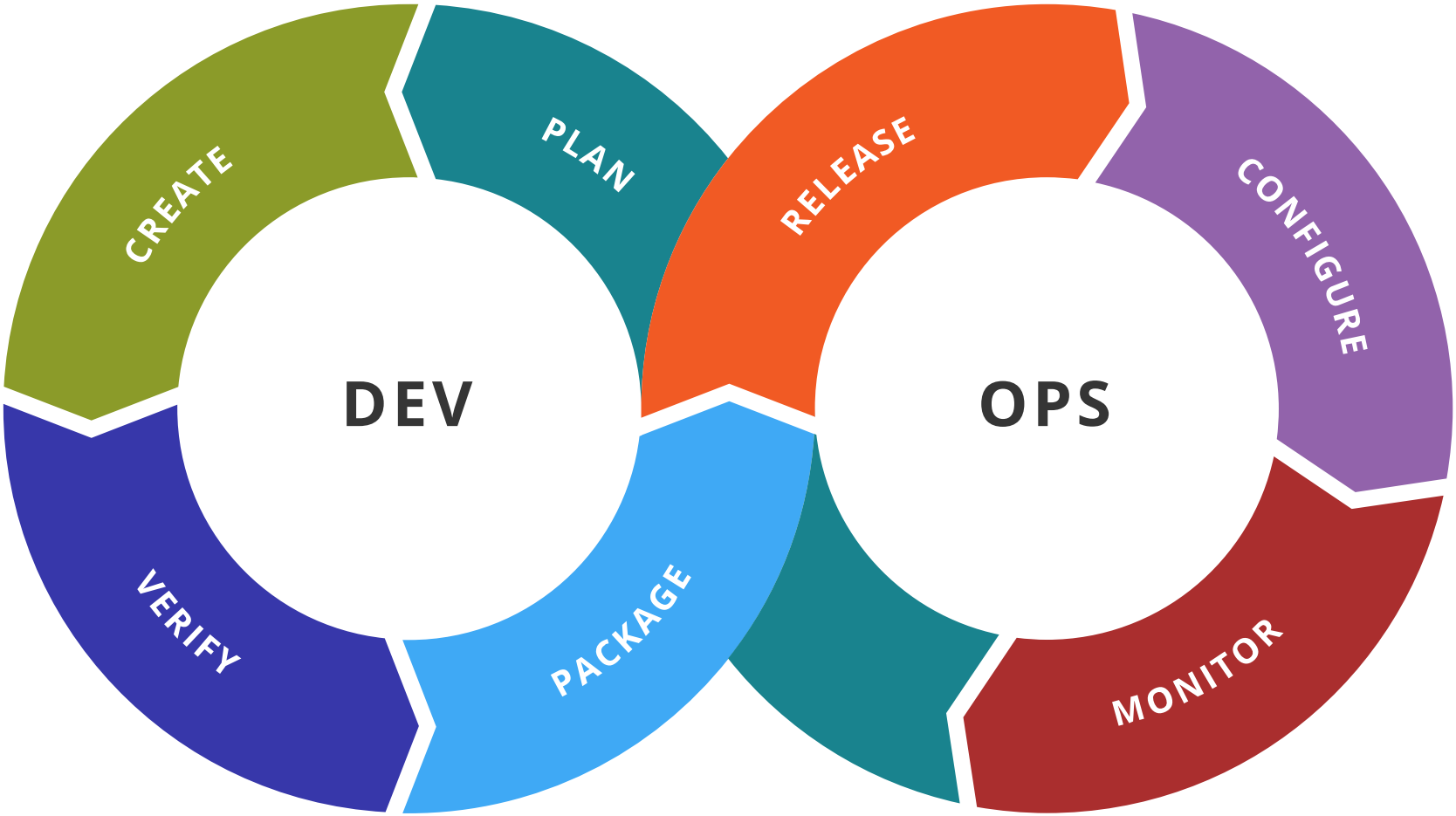
4.4 Infrastructure as Code and IaC Tools (like Ansible)

4.5 DevOps and CI/CD on Clouds

DevOps



DevOps


CREATE
PLAN
RELEASE
CONFIGURE
DEV
OPS
VERIFY
PACKAGE
MONITOR

Continuous Integration

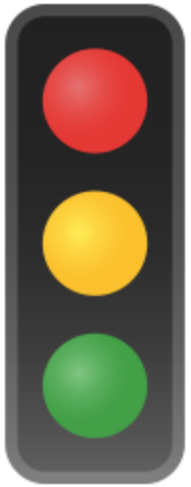
- Automation of *integration and testing* of software

- First proposed by Grady Booch, 1991
- Adopted and popularized as part of XP (Extreme Programming)

- Motivation:
  - Evade “integration hell”: Find bugs while still easy to fix
  - Fast feedback on system-wide impact of local changes
  - Team has a common view of the software project state

Continuous Integration: Practices

- One common (versioned) code repository
- Build automation (and short builds)
- Self-testing builds
- Regular commits
- Building every commit (usually on a CI server)
- Test & production environments as similar as possible
- Test results visible for everyone



Continuous Delivery

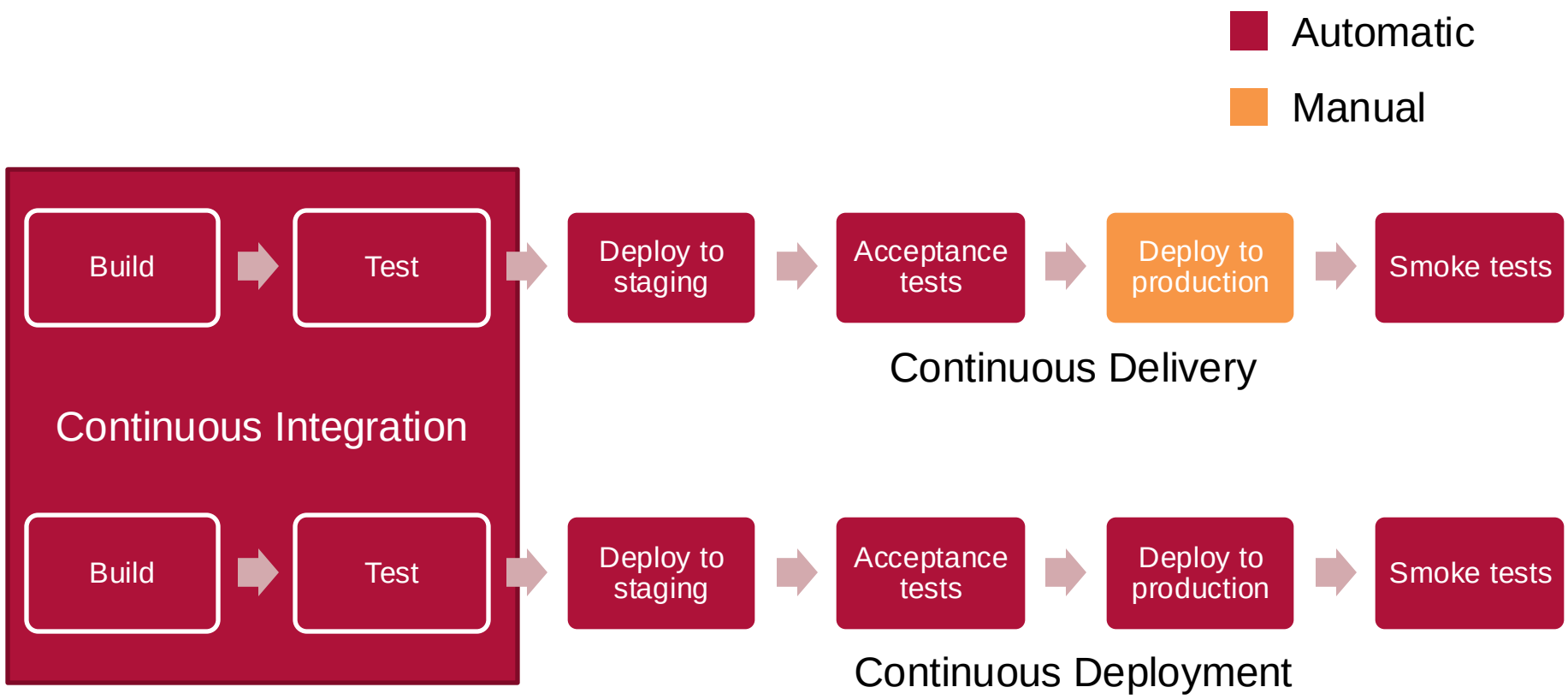
- Fast and reproducible software releases

- From the first principle of the agile manifesto:

  "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software."

- CD Metric: How long does it take to release a single-line-code-change to production? *(cycle time)*
- Business case: Software development as investment → optimizing ROI through small cycle time

Continuous Delivery vs. Continuous Deployment



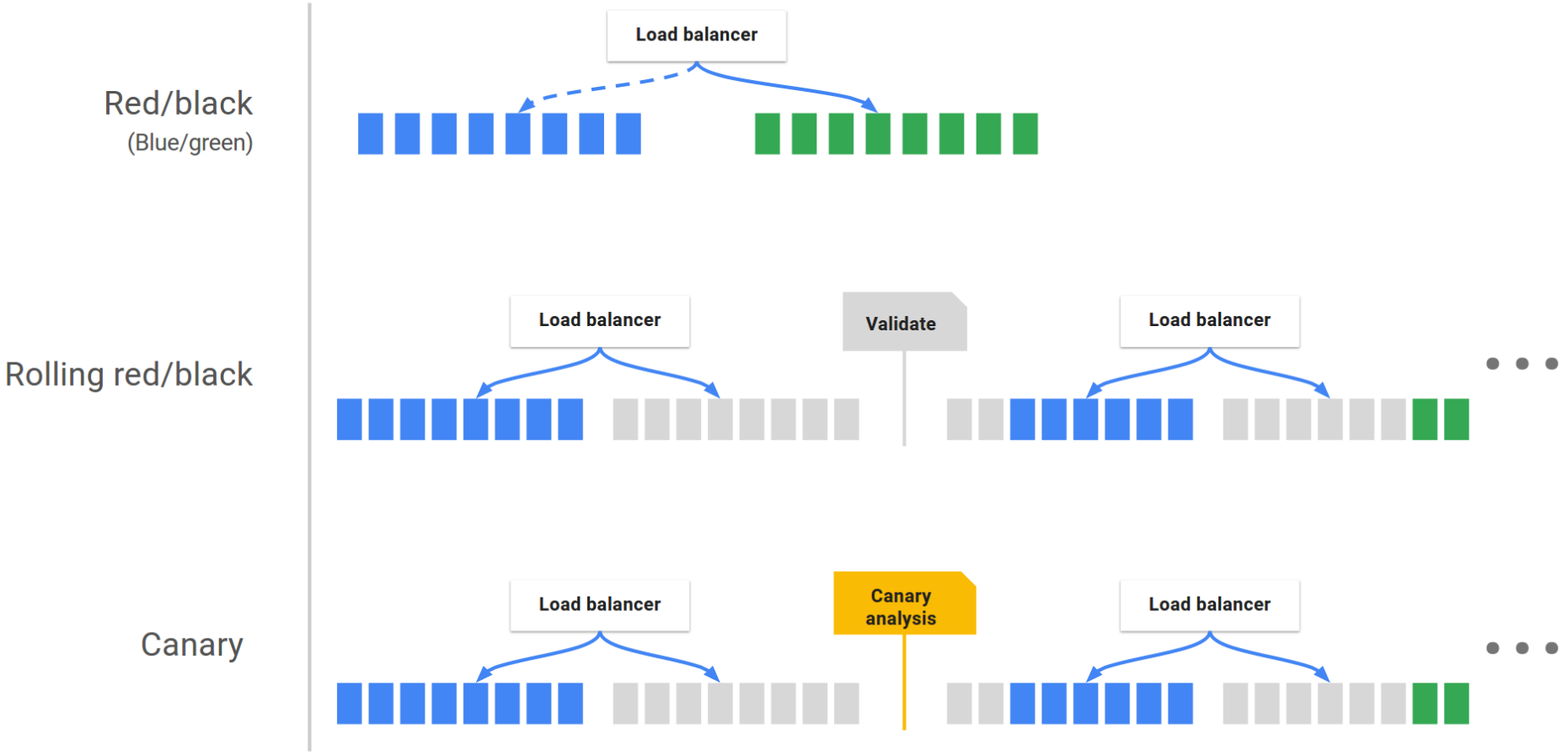
Deployment Strategies: Blue/Green

- Strategy to deploy an upgrade
- Two production environments: Blue and green
- Only one environments is active at a time

1. Deploy to the non-active environment
2. Switch the traffic coming through the load balancer over to the new environment
3. Monitor new version, maybe switch back

- One environment always runs the last version, the other the newest version: easy rollbacks

Deployment Strategies

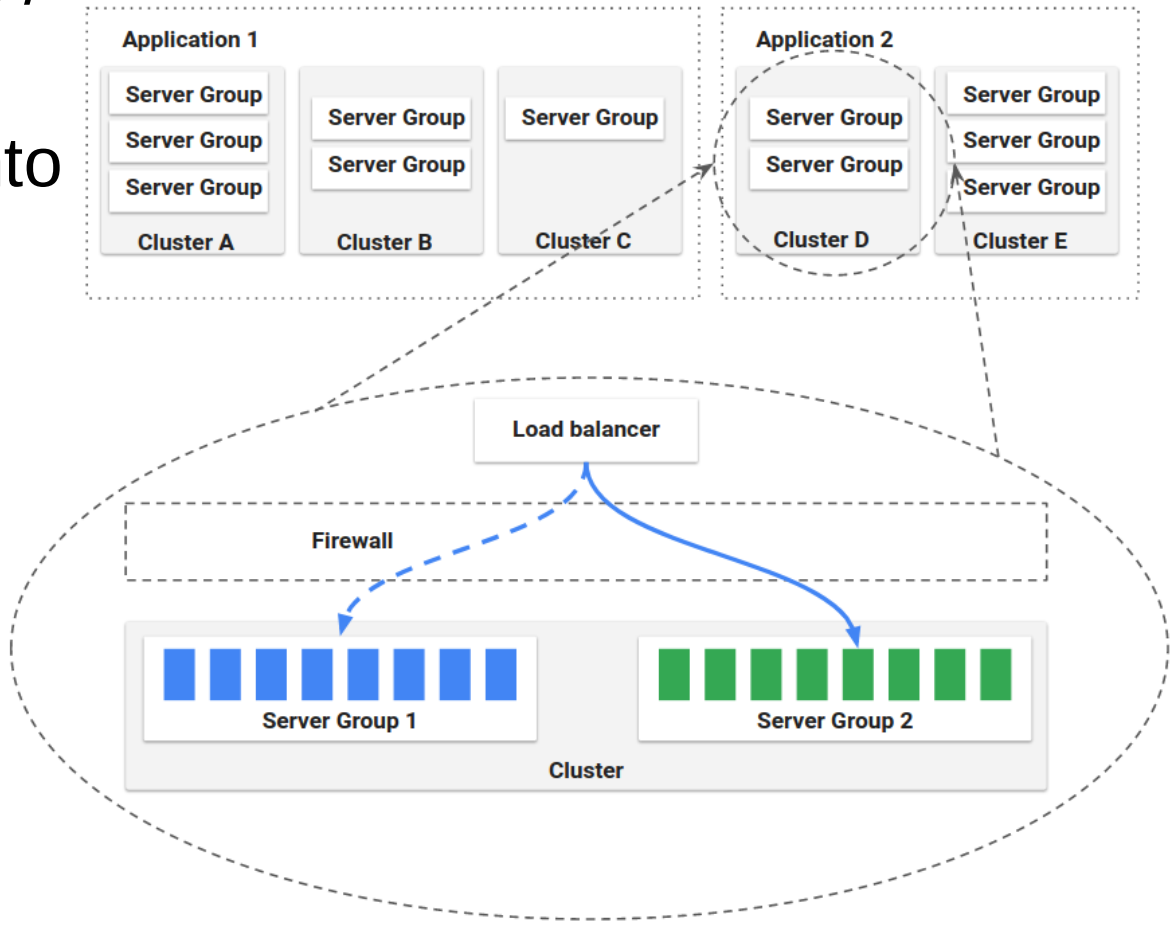


Spinnaker

- Continuous delivery platform
- Started by Netflix, picked up and extended by Google
  - Initial release November 2015
  - Successor to Asgard, a web interface for automated AWS deployments
  - From baking to cloud deployment
- Support and common abstractions for EC2, GCE, Azure, Kubernetes...

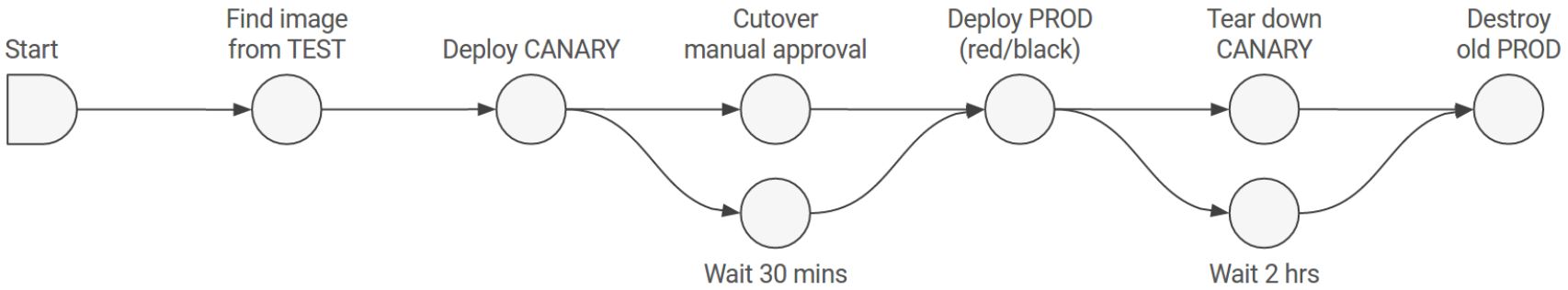
Spinnaker

- Deploy applications / (micro)services
- Organize servers into
  - Clusters
  - Server groups

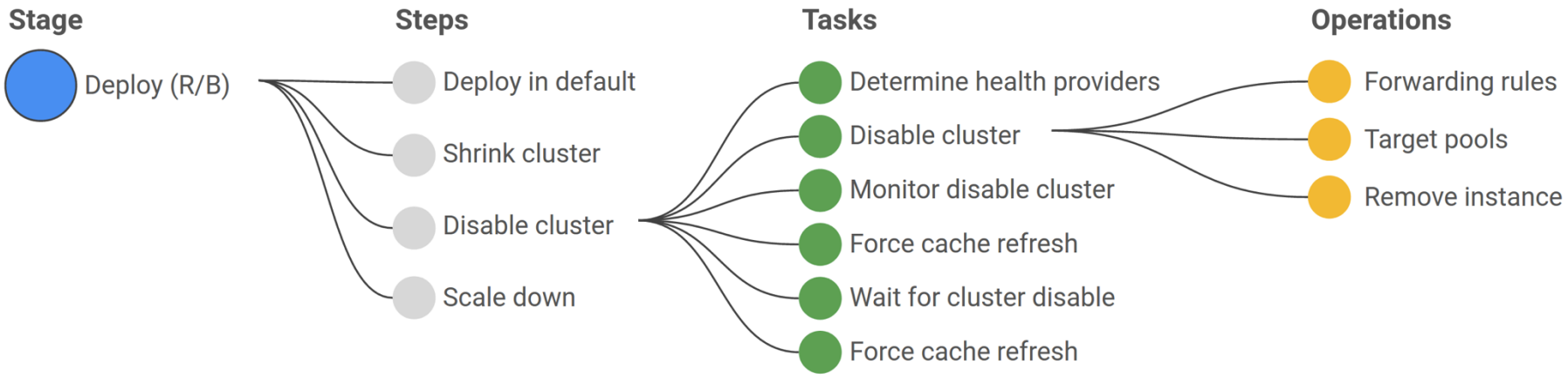


Spinnaker Pipeline

- Deployment happens via pipelines
- Pipelines are built from stages
- Predefined stages exist for many things
  - Bake
  - Manual judgement
  - Clone server group
  - Deploy



Spinnaker Stage Example



Outline of Chapter 4: Cloud Infrastructure Management

4.1 Intro to Cloud Infrastructure Management

4.2 Cloud Operating Systems (like OpenStack)

4.3 VM Management: Pausing and Migration

– short break (10-15 minutes) –

4.4 Infrastructure as Code and IaC tools (like Ansible)

4.5 DevOps and CI/CD on Clouds

Summary

- Server sprawl, configuration drift, and snowflake servers exemplify the complexity of managing cloud infrastructure

- Cloud operating and orchestration systems as well as automation engines help, but development and operating processes need to change as well:
  - IaC: Strictly automate and rebuild infrastructure
  - DevOps: Responsibility extends to operation
  - CI/CD: Integrate and deploy systems continuously

References

- [15] C. Clark, K. Fraser, S. Hand, J.G. Hanseny, E. July, C. Limpach, I. Pratt, A. Wareld: “Live Migration of Virtual Machines”, Proc. of the 2nd Symposium on Networked Systems Design & Implementation, 2005

- Real-world systems discussed:
  - https://www.openstack.org/
  - https://www.xenserver.com/
  - https://www.ansible.com/
  - https://spinnaker.io/