Cloudify 3.2 Training

Prerequisites

Required:

- Basic Linux skills
- Basic understanding of Cloud concepts
- YAML
- Python

Recommended:

- Cloud architecture
- Virtualization
- Configuration management tools (Docker, Chef, Puppet)

Agenda

- Overview
- Bootstrapping a Manager
  - **Lab 1**: Configuring Vagrant and Installing CLI
  - **Lab 2**: Manager Bootstrapping Using the Simple Manager Blueprint
- Blueprints and Deployments
  - Introduction to Blueprints and Their Syntax
  - **Lab 3**: Uploading and Deploying a Sample Blueprint
- Using Scripts in Lifecycle Events
  - **Lab 4**: Tweaking a Simple Blueprint

Agenda (cont'd)

- Introduction to Plugins
- Official Plugins: Script, Fabric, Docker
  - **Lab 5**: Deploying Docker Containers
- Introduction to Workflows
  - Built-in workflows (**`install`**, **`uninstall`**, **`execute_operation`**, **`heal`**, **`scale`**)
  - **Lab 6**: Workflows
- Introduction to Monitoring
  - **Lab 7**: Deploying Collectors and Using the Grafana Dashboard
- Introduction to Security
  - **Lab 8**: Security

Agenda (cont'd)

- Working with Additional Clouds
  - OpenStack
    - **Lab 9**: OpenStack Bootstrapping and Deployment

Agenda: Optional Material

- Cloudify Manager Architecture
- Administration Tasks
  - Logs & Events
- More Official Plugins: Chef, Puppet
  - Lab X1: Chef Integration
  - Lab X2: Puppet Integration
- Working with Additional Clouds
  - AWS
    - Lab X3: AWS Bootstrapping and Deployment
  - CloudStack
    - Lab X4: CloudStack Bootstrapping and Deployment

OVERVIEW

Main Functions

- Deployment modeling using TOSCA
- Automated resource creation, placement and configuration
- Built-in integration with Docker, Chef, Puppet, Fabric, SaltStack and others
- Support for healing and scaling
- Metric and log collection and visualization
- Security integration
- Extensible workflow engine for supporting complex automation scenarios

Main Flows

- Bootstrapping
- Blueprint upload & deployment creation
- Workflow execution
  - Scaling
  - Healing
- Log gathering
- Monitoring

Basic Building Blocks

- *Blueprint*: describes a topology, its workflows, and its policies
- *Deployment*: An instance of a blueprint, with an actual execution context and resources
- *Execution*: a single run of a workflow for a given deployment

Basic Building Blocks (cont'd)

Upload blueprint

Blueprint

Create deployment

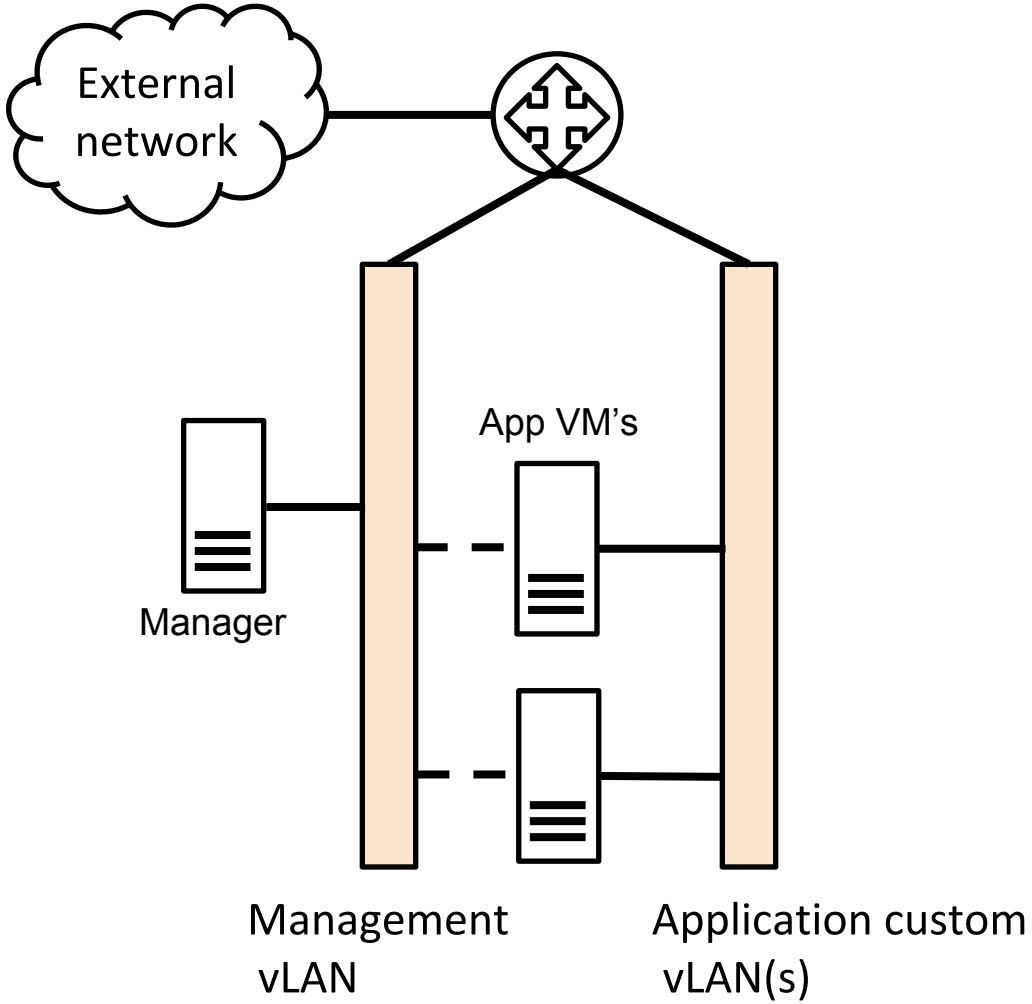
Deployment 1 **Deployment 2** **Deployment 3**

Execute workflows (i.e. "install")

Execution **Execution** **Execution**

BOOTSTRAPPING A MANAGER

Network Architecture



What Is a Manager Blueprint?

- A blueprint for installing all of the manager's components
  - Infrastructure + software
  - Blueprints will be covered shortly
- Allows users to customize the Cloudify Manager's topology

Manager Blueprints Available Out-of-the-Box

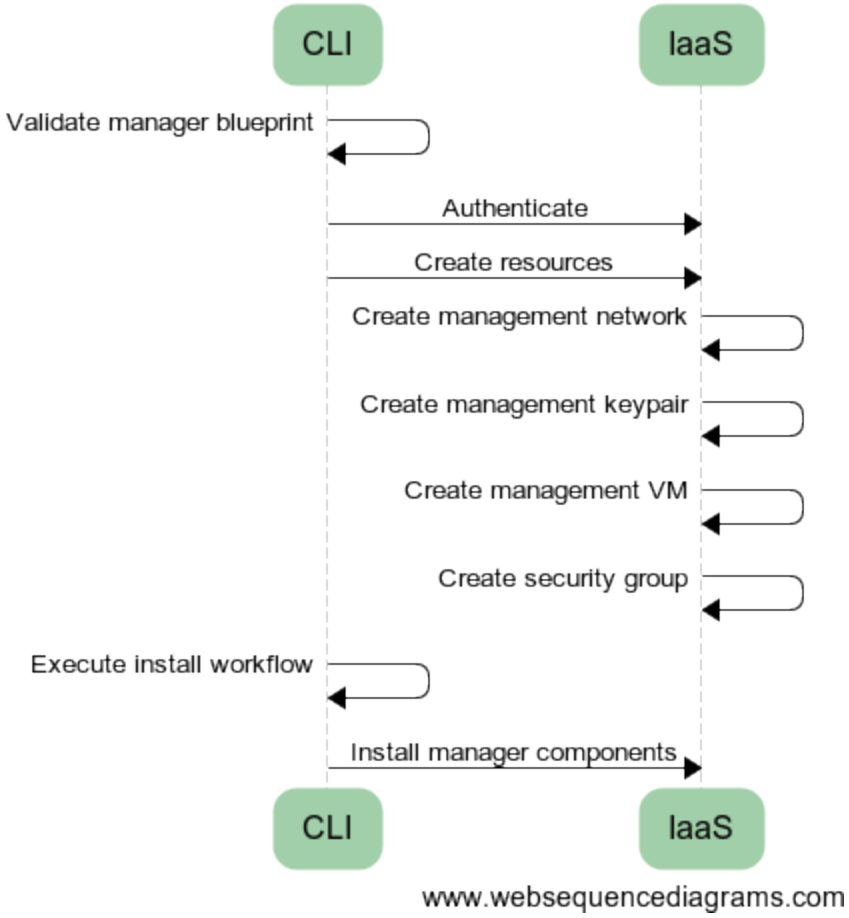
`simple`	Does not create infrastructure, only installs the manager on a given host
`openstack`	Installs the manager in an OpenStack environment. Also creates a network, a subnet, and a number of other infrastructure components to host the manager
`openstack-nova-net`	Same as the regular OpenStack blueprint, but uses Nova network instead of Neutron for the networking infrastructure it creates
`cloudstack`	Installs the manager in an Apache CloudStack environment. Also creates a network, and a number of other infrastructure components to host the manager
`softlayer`	Installs the manager on Softlayer

The Bootstrap Process

- Manager blueprint is validated
- Authentication against IaaS
- IaaS resource creation:
  - Management network
  - Management keypair
  - Virtual machine
  - Security group
  - Possibly other/different resources, depending on the cloud
- Installation of Cloudify manager components

Requires the CLI to run some of the orchestration modules locally on the CLI machine

The Bootstrap Process (cont'd)



LAB 1: CONFIGURING VAGRANT AND INSTALLING CLI

LAB 2: MANAGER BOOTSTRAPPING USING THE SIMPLE MANAGER BLUEPRINT

BLUEPRINTS AND DEPLOYMENTS

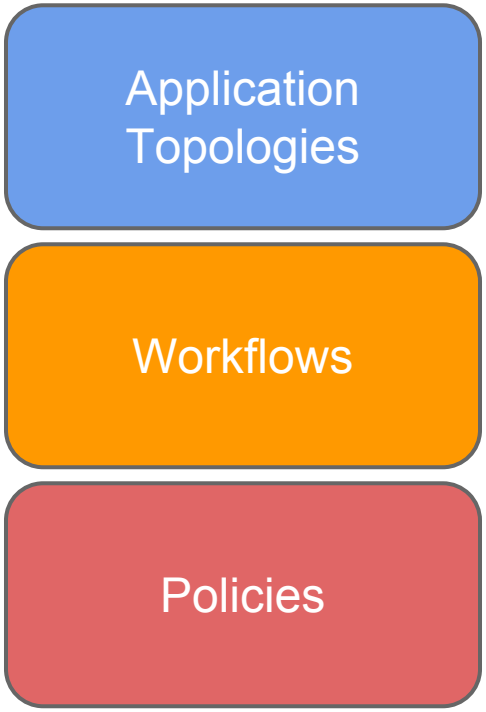
Cloudify Blueprints

- A Blueprint is an orchestration plan for a single application
- Consists of the following parts:
  - *Topology*: Declarative, written in YAML
  - *Workflows*:
    - Declared in YAML
    - Implemented in Python
  - *Policies*: Declarative, written mostly in YAML
- Conforms to TOSCA

What is TOSCA?

- **T**opology & **O**rchestration **S**pecification for **C**loud **A**pplications
- By OASIS
  - Sponsored by IBM, CA, Red Hat, Huawei and others
- Goal: allow for a cross-Cloud, cross-tools orchestration of applications on the Cloud
- Version 1.1, AKA “Simple Profile”, is YAML-based

TOSCA-Inspired Application Blueprints



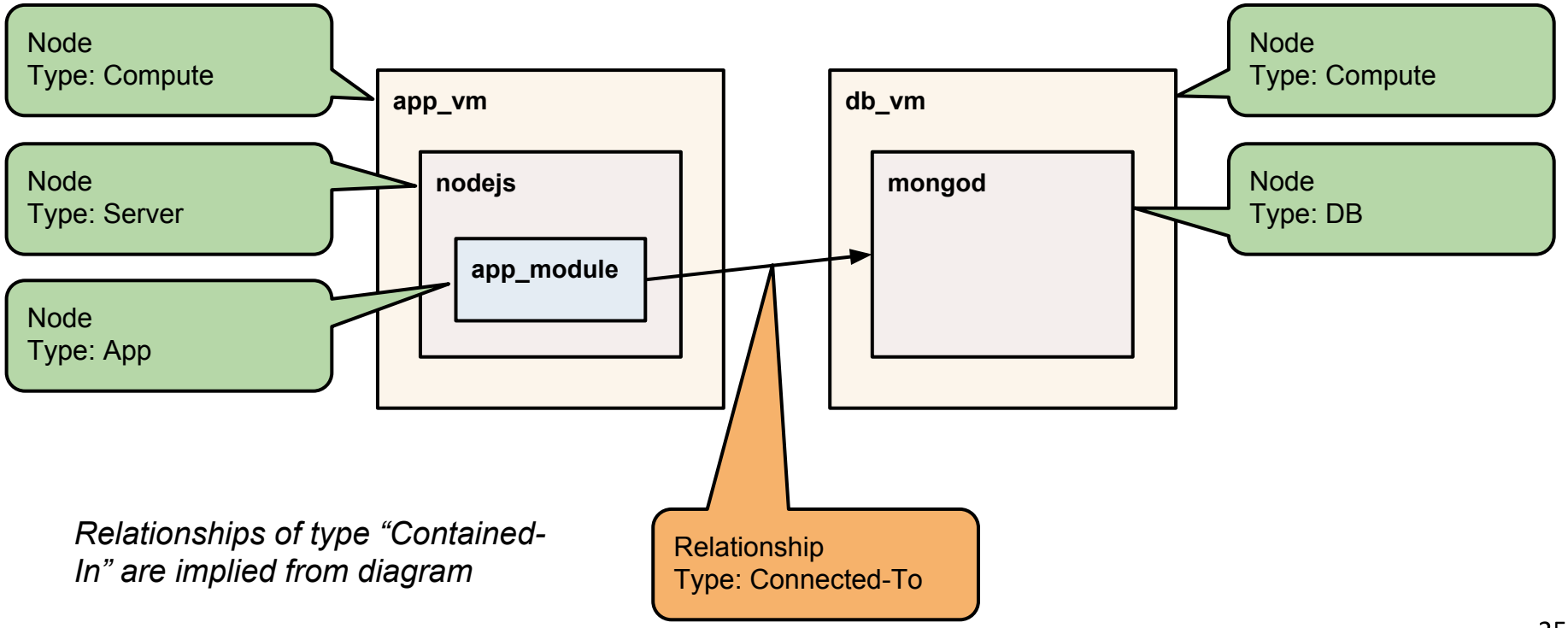
Why TOSCA?

- Open standard
- Can describe any topology and any automation process
- Portable between Cloud implementations and tools

A Topology

Simple example application topology:

- An application VM, containing:
  - NodeJS, containing the application module to run
- A database VM, containing MongoDB



Relationships of type "Contained-In" are implied from diagram

Blueprint: Outline

- Imports
- Plugin declarations
- Inputs and outputs
- Types, nodes and relationships
- Intrinsic functions
- Workflows
- Policies

Blueprint Structure

```
tosca_definitions_version: ...
...
imports:
  ...
plugins:
  ...
inputs:
  ...
node_types:
  ...
relationships:
  ...
node_templates:
  ...
workflows:
  ...
groups:
  ...
outputs:
  ...
```

Imports

- Imports are a way to use reusable plugins and types in your blueprint
- You will always import the default **`types.yaml`** file that contains all major Cloudify types and plugins (more on this later)

Imports: Example

```
imports:
  - http://www.getcloudify.org/spec/cloudify/3.2/types.yaml
  - http://www.getcloudify.org/spec/openstack-plugin/1.2/plugin.yaml
```

- This will import:
  - The basic **`types.yaml`** file which contains all of Cloudify's built-in types
  - The OpenStack plugin and associated types
- By convention, every plugin repository has a **`plugin.yaml`** file in its root that contains all its plugins and type declarations

Plugin Declarations

- Usually imported as part of a **`plugin.yaml`** file
- Plugins are simply python packages (more later)
- Usually imported from a zip file at a URL, but can also be included in the blueprint as a directory and referenced from there

```
plugins:
  docker:
    executor: host_agent
    # Obtain directly from a URL
    source: https://github.com/cloudify-cosmo/cloudify-docker-plugin/archive/1.2.zip
  custom-embedded:
    executor: host_agent
    # Plugin sources must exist in a directory named plugins/some-directory, relative
    # to the blueprint's root
    source: some-directory
```

Cloudify Types

- A type defines:
  - Operations
  - Properties
- Two kinds of types:
  - *Node types*: describe components in the topology
  - *Relationship types*: describe relationships between topology nodes
- Can be extended
- Defined using YAML
- Implemented in Python
- Cloudify provides a set of built-in types
  - Usually, built-in types don't define any implementation
  - The implementation is defined when extending a built-in type, or when using a built-in type in a blueprint

Creating a Cloudify Type

- You need to declare a new type only if:
  - You want to implement a new functionality
  - You want to extend another type for enforcing a schema or assigning default value(s)
- A type can be either:
  - Declared inside the blueprint; or
  - Imported from another file (typically **`plugin.yaml`** of a certain plugin)

Nodes and Node Types

- A node is a realization of a node type
- In Cloudify's terminology, a *node* is a component in the topology. Examples:
  - A VM
  - A floating IP address
  - A Docker container
  - A web server
  - A WAR file
- The node definition specifies the node's type, its configuration values (properties) and its relationships with other nodes

Nodes and Node Types (cont'd)

- All node types are derived, directly or indirectly, from **`cloudify.nodes.Root`**
- The Cloudify "Root" type declares three lifecycle interfaces
- Each interface declares *events* that can be bound to *operations*
  - By default, all events are unbound

```
cloudify.nodes.Root:
    interfaces:
      cloudify.interfaces.lifecycle:
        create: {}
        configure: {}
        start: {}
        stop: {}
        delete: {}
      cloudify.interfaces.validation:
        creation: {}
        deletion: {}
      cloudify.interfaces.monitoring:
        start: {}
        stop: {}
```

Example: Using a Built-In Node Type

```
node_templates:
  ...
  mongodb:
    type: cloudify.nodes.DBMS
    interfaces:
      cloudify.interfaces.lifecycle:
        create: scripts/mongo/install-mongo.sh
        start: scripts/mongo/start-mongo.sh
        stop: script/mongo/stop-mongo.sh
```

Example: Defining and Using a Custom Type

Definition:

```
node_types:
  nodecellar.nodes.MongoDatabase:
    derived_from: cloudify.nodes.DBMS
    properties:
      port:
        description: MongoDB port
        type: integer
        default: 20000
    interfaces:
      cloudify.interfaces.lifecycle:
        create: scripts/mongo/install-mongo.sh
        start: scripts/mongo/start-mongo.sh
        stop: scripts/mongo/stop-mongo.sh
```

With custom types we can define a property schema. If a default value isn't specified, then the property is mandatory.

Usage:

```
node_templates:
  mongod:
    type: nodecellar.nodes.MongoDatabase
    properties:
      port: 27017
```

Relationships

- Relationships are types
- Cloudify provides three built-in relationships (through **`types.yaml`**):
  - **`depends_on`**: the base type of all relationships
    - **`contained_in`**: a component is hosted / contained / deployed within another component
    - **`connected_to`**: a component needs to establish a connection to another
- Can be extended to create new relationship types

Relationships (cont'd)

- The basic **`depends_on`** type defines operations to be applied to the source and the target instances of a relationship

```
cloudify.relationships.depends_on:
  source_interfaces:
    cloudify.interfaces.relationship_lifecycle:
      preconfigure: {}
      postconfigure: {}
      establish: {}
      unlink: {}
  target_interfaces:
    cloudify.interfaces.relationship_lifecycle:
      preconfigure: {}
      postconfigure: {}
      establish: {}
      unlink: {}
  properties:
    connection_type:
      default: all_to_all
```

Relationships (cont'd)

- Operations can optionally be implemented for source and target instances:
  - **`preconfigure`**: before the node's **`configure`** life cycle operation is called
  - **`postconfigure`**: after the node's **`configure`** is called but before start
  - **`establish`**: after start, when a connection needs to be established
  - **`unlink`**: before stop, when a connection needs to be removed

Example: Using a Built-In Relationship Type

```
relationships:
  - type: cloudify.relationships.contained_in
    target: vm
  - type: cloudify.relationships.connected_to
    target: db
    source_interfaces:
      cloudify.interfaces.relationship_lifecycle:
        preconfigure: some-scripts/preconfigure.sh
```

Creating a Relationship Type

- Create a custom Relationship Type:
  - If you need a plugin to implement the gluing of components together
  - To avoid repeating relationship lifecycle definitions for multiple nodes of the same relationship type
- The declaration is similar to a node type declaration, but it resides under **`relationships`** and not under **`node_types`**

```
cloudify.openstack.subnet_connected_to_router:
  derived_from: cloudify.relationships.connected_to
  target_interfaces:
    cloudify.interfaces.relationship_lifecycle:
      - establish: neutron_plugin.router.connect_subnet
      - unlink: neutron_plugin.router.disconnect_subnet
```

Code called when establishing and unlinking the relationship.

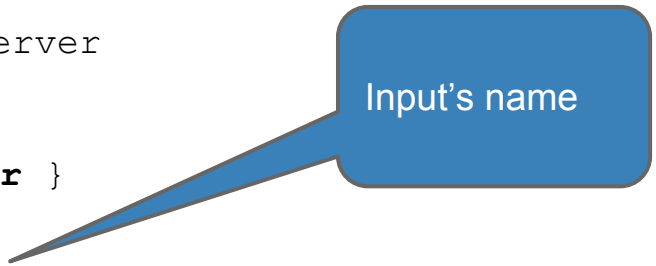
Intrinsic Functions

- Intrinsic functions are special keywords in the blueprint's YAML
- Used to place inputs, properties, and runtime properties into elements in the blueprint (nodes, outputs)

Intrinsic Functions: `get_input`

Returns the value of a blueprint's input.

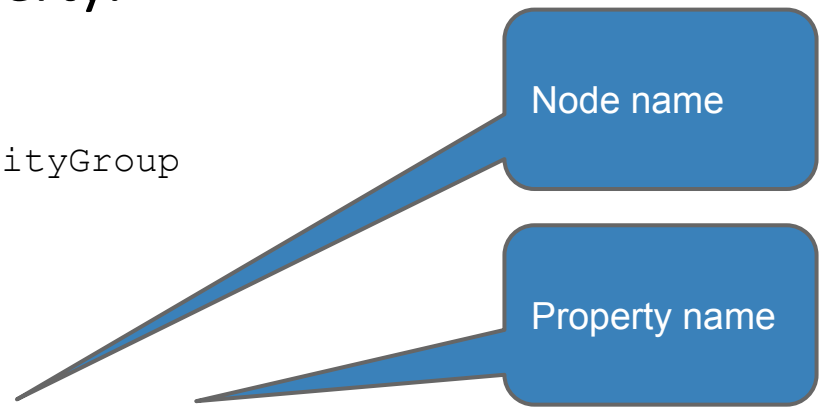
```
inputs:
  agent_user: {}
  image: {}
  flavor: {}

...

vm:
  type: cloudify.openstack.nodes.Server
  properties:
    cloudify_agent:
      user: { get_input: agent_user }
    server:
      image: { get_input: image }
      flavor: { get_input: flavor }
```



Intrinsic Functions: `get_property`

Returns the value of a node's property.

```
nodecellar_security_group:
    type: cloudify.openstack.nodes.SecurityGroup
    properties:
      security_group:
        name: nodecellar_security_group
      rules:
        - remote_ip_prefix: 0.0.0.0/0
          port: { get_property: [  nodecellar, port ] }
```



Intrinsic Functions: `get_attribute`

Returns the value of a node's runtime property.

```
outputs:
  endpoint:
    description: Web application endpoint
    value:
      ip_address: { get_attribute: [  floatingip, floating_ip_address ] }
      port: { get_property: [ nodecellar, port ] }
```

Node's name

Runtime property's name

Inputs

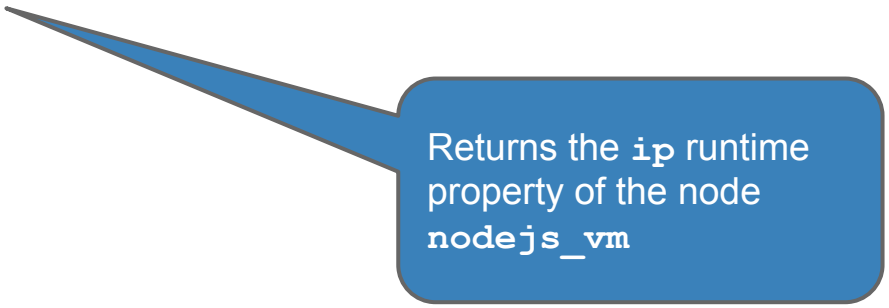
- A way to parameterize blueprints and let them declare runtime computed values (URLs, passwords, etc.)
- An input's value can be retrieved via the **`get_input`** intrinsic function

```
inputs:
  mongodb_port:
    default: ''
    type: string

...
...

node_templates:
  database:
    type: cloudify.nodes.DBMS
    properties:
      port: { get_input: mongodb_port }
```

Outputs

- A way to provide runtime information about the deployment
  - Dedicated API call to retrieve outputs
- Typically, uses intrinsic functions to obtain deployment-specific information

```
outputs:
  endpoint:
    description: Web application endpoint
    value:
      ip_address: { get_attribute: [ nodejs_vm, ip ] }
```



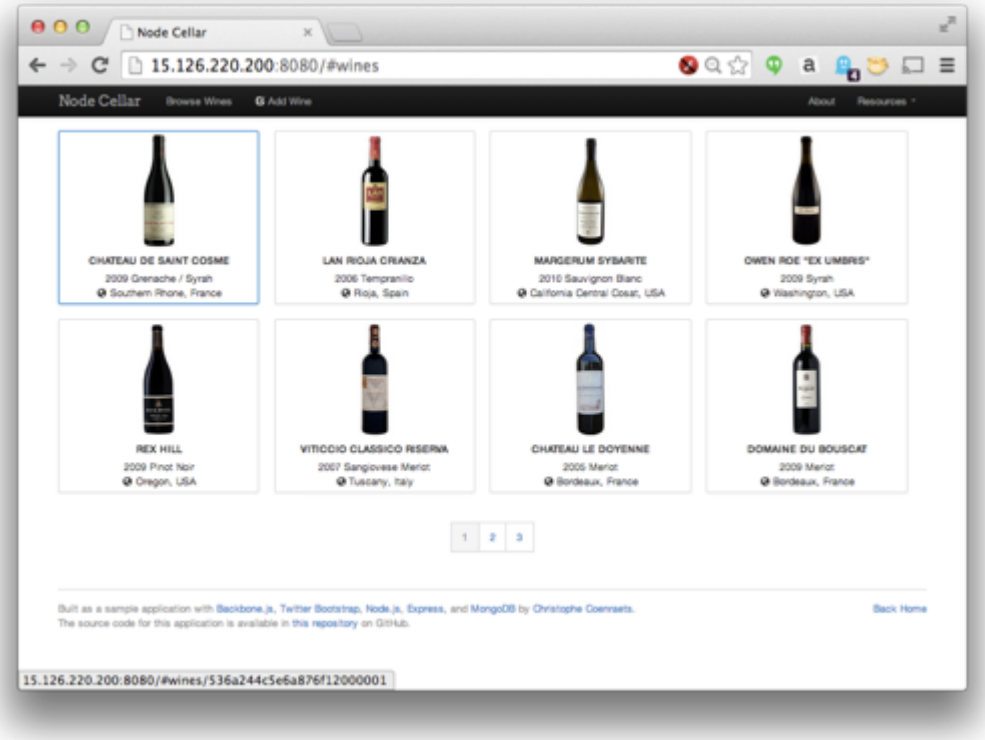
Workflows

- A workflow is a special type of plugin
- A workflow plugin is declared as any other plugin, and then specified in the **`workflows`** section of the blueprint
- More later

Putting It All Together

- TOSCA Template ("Blueprint" in Cloudify's terminology) contains:
  - Application topology
    - Nodes
    - Interfaces and operations
    - Properties
    - Relationships
  - Workflows (**`install`**, **`uninstall`**, **`scale`**, **`heal`**)

Review a Full Blown Blueprint



https://github.com/cloudify-cosmo/cloudify-nodecellar-docker-example

LAB 3: UPLOADING AND DEPLOYING A SAMPLE BLUEPRINT

USING SCRIPTS IN LIFECYCLE EVENTS

Invoking Scripts: Introduction

- Functionality is provided by the built-in Script plugin
- For example: instructing Cloudify to run a shell script called **`my-install.sh`** during the **`install`** lifecycle operation
- Scripts can be written in python, bash, ruby… or any other language
- Scripts can access Cloudify resources, topology information and more
- Will be covered in-depth when we discuss the Script plugin later

Invoking Scripts: Example

blueprint.yaml

```
imports:
  - http://www.getcloudify.org/spec/cloudify/3.2/types.yaml

node_templates:
  example_web_server:
    # The web server type is only used as an example. The type used
    # could be any valid Cloudify type.
    type: cloudify.nodes.WebServer
    interfaces:
      cloudify.interfaces.lifecycle:
        start: scripts/start.sh
```

scripts/start.sh

Location: relative to blueprint's root

"**ctx**" will be discussed later

```
#!/bin/bash -e
ctx logger info "Hello to this world"
```

LAB 4: TWEAKING A SIMPLE BLUEPRINT

INTRODUCTION TO PLUGINS

Introduction to Plugins

- Cloudify is extensible: it supports custom types, operations, workflows, …
- A plugin packages Python operations, tasks and workflow implementations, plus optional custom types
- A plugin extends the functionality of a Cloudify agent
- Plugins make up the implementation for interface operations
- Plugins implementations describe *how* a type (node or relationship) behaves
- There are two types of plugins:
  - **Agent-host plugins**: run on agent machines
  - **Central-deployment-agent plugins**: run on a central location (currently the Cloudify Manager)
- In this section, the term “plugin” will refer to *operation plugins* (as opposed to *workflow plugins*, discussed later)

Introduction to Plugins (cont'd)

- What a plugin implementation actually looks like:

```
from cloudify import ctx

@operation
def create(**kwargs):
  node_type = ctx.node.type
  ctx.logger.info('node type is {0}'.format(node_type))
```

- Plugin operations should be simple and focus on discrete, well-defined operations

Using Plugins: Plugin Declaration

- Using a plugin requires declaring it in the blueprint:

```
plugins:
  openstack:
    executor: central_deployment_agent
    source: https://github.com/cloudify-cosmo/cloudify-openstack-
plugin/archive/1.2.zip
```

- Alternatively, blueprints can import a YAML file that contains the plugin's declaration:

```
imports:
  - http://www.getcloudify.org/spec/openstack-plugin/1.2/plugin.yaml
```

- By convention, plugin repositories will contain such **`plugin.yaml`** file at the repository's root

Using Plugins: Plugin Declaration (cont'd)

- The **`executor`** field specifies where the plugin will execute (AKA "plugin type")
  - Overridable on a per-operation basis
- The **`source`** field points to the plugin implementation:
  - A URL, pointing to a pip-installable archive; or
  - A directory path, for bundled plugins
    - Bundled plugins must be located in a directory named **`plugins`**, which is on the same level as the main blueprint file

```
| (blueprint's root)
|___ plugins
|___ | ___ custom_plugin
|___ | ___ | ___ plugin_file.py
```

Using Plugins: Operation Mapping

- Once the plugin is declared, operations may be mapped to it accordingly:

```
node_templates:
  example_node:
    type: my_type
    interfaces:
      cloudify.interfaces.lifecycle:
        start:
          implementation: openstack.nova_plugin.server.start
          inputs:
            my_param: my_value
```

- Mapping explanation:
  - **`openstack`**: name of the plugin as declared before
  - **`nova_plugin`**: name of the Python package
  - **`server`**: name of the Python module
  - **`start`**: name of the Python method

Plugin Authoring

- Authoring plugins is comprised of the following steps:
  - Coding
  - **`plugin.yaml`**
  - Testing
  - Packaging
- A plugin-template is available to ease the process of writing custom plugins
  - Contains the basic building blocks for a working plugin
  - **`plugins/tests/test_plugin.py`**: a working integration test (will fail unless the plugin is implemented correctly)

Plugin Benefits over Scripts

- **Ease of Debugging & Testing**:
  - Ability to set breakpoints and step through code when using local workflows
  - Ability to use a mock for the Cloudify context object and test operations directly
- **Reusability**: Scripts must be provided along with each blueprint that uses them, whereas plugin provisioning is automatic

Plugin Benefits over Scripts (cont'd)

- **Less overhead**: No need for external ctx-proxy service and ctx-proxy program; no dependency on the Script plugin
- **Readability**: The context API is designed for Python code; the script plugin offers the full API as well, yet its syntax and usage may be less clear to understand
- **Platform independence**

Plugin Coding

- A plugin task is a Python function decorated with Cloudify's **`@operation`** decorator
  - An operation is currently required to receive a **`**kwargs`** parameter
- Several such functions may run in parallel
  - Order and parallelism of tasks depends on the workflow being executed
- The plugin's author can interact with Cloudify via the operation context object, available through importing **`ctx`** from the **`cloudify`** package

Plugin Coding: `ctx` Object

- The ctx object holds all information for the current operation context:
  - General information about the operation: **`execution-id`**, **`workflow-id`**, plugin name, operation name, …
- Grants access to other objects relevant for the operation:
  - In a node operation context: **`ctx.instance`** and **`ctx.node`**
  - In a relationship operation context:
    - **`ctx.source.node`**
    - **`ctx.source.instance`**
    - **`ctx.target.node`**
    - **`ctx.target.instance`**

Plugin Coding: `ctx` Object (cont'd)

- A **`ctx.node`** object holds:
  - The node's properties, as set in the blueprint
  - The node's general information, such as **`id`**, **`type`**, **`type-hierarchy`**, etc.
- A **`ctx.instance`** object holds:
  - The node instance's **`runtime-properties`**
  - The node instance's **`host-ip`**
- **`ctx.logger`**: provides Cloudify logging
- **`ctx.get_resource`** and **`ctx.download_resource`**: retrieve resources from the blueprint archive
- **`ctx.bootstrap_context`**: access to Manager settings configured during bootstrapping

Examples: Using the `ctx` Object

```
# Gets the “application_name” property from the node.
ctx.node.properties['application_name']

# Gets a dict showing the node’s type hierarchy
ctx.node.type_hierarchy

# Sets a runtime property for a node instance
ctx.instance.runtime_properties['my_property'] = 'my_value‘

# Returns the node instance’s host IP
ctx.instance.host_ip

# Logs
ctx.logger.info('Some logging')

# Downloads a resource provided in the blueprint archive
ctx.download_resource('images/hello.png', **{'target_path': '/tmp/hello.
png'})

# Retrieves bootstrapping information
ctx.bootstrap_context.cloudify_agent.agent_key_path
```

Plugin Coding: Error Handling

- Two types of errors raised from plugins:
  - **`RecoverableError`**
  - **`NonRecoverableError`**
- Recoverable errors will be automatically retried
  - Set **`retry-after`** when raising a **`RecoverableError`**, to specify the interval between attempts
- Other errors (non-recoverable, unhandled or built-in) will cause the workflow to fail immediately
- When executing a workflow, the **`retry_attempts`** and **`timeout_seconds`** parameters can be provided

Blueprint Declarations

Plugins typically include a **`plugin.yaml`** file, which may include:

- The plugin declaration itself, including its source and type (executor):

```
plugins
  openstack:
    executor: central_deployment_agent
    source: https://github.com/cloudify-cosmo/cloudify-openstack-
plugin/archive/1.2.zip
```

- Custom node and relationship types
- Operation mappings for lifecycle events
- Imports required by the plugin (including other plugins)

Plugin Testing

- Plugin testing strategies:
  - Using local workflows (will be discussed later)
  - Unit testing, using a mock for the context object
- The strategy to use depends on what is being tested
  - See the plugin-template for local workflows example
  - Import **`MockCloudifyContext`** from the **`cloudify`** package for mocking the context object
- Plugin testing is important — without it, finding bugs may be a lengthy process, as testing cycles require VMs, agents installations, etc.

Plugin Packaging

- Standard Python packages containing a **`setup.py`** file
  - Will be installed by *pip*, on their agent(s)
  - *pip* Installation arguments can be provided
- See the **`plugin-template`** structure for packaging example

Additional Information

- Cloudify Documentation:
  - Plugins authoring guide
  - Plugins-common API documentation
- Other Resources:
  - Plugin template
  - Example: Openstack plugin

OFFICIAL PLUGINS

Official Plugins

- **IAAS plugins**:
  - Openstack
  - AWS
  - vSphere
  - Softlayer
  - Cloudstack
  - …
- **Deployment plugins**:
  - Docker
  - Chef
  - Puppet
  - Saltstack
  - …

Official Plugins (cont'd)

- **Implementation plugins**:
  - Script
  - Fabric
  - ...
- **Monitoring plugins**:
  - Diamond

Bundled Plugins

- Most of these plugins are optional add-ons to Cloudify, and are not required for Cloudify to function properly
- The **Script plugin**, however, is built into Cloudify, as it is very commonly used, and is also used internally by Cloudify itself
  - This plugin's definitions are located in **`types.yaml`**
  - Automatically installed and ready for use by the default agent packages
- The **Fabric plugin** is also bundled with Cloudify

Script Plugin

- The Script plugin is used to map operations and workflows to scripts
- Scripts can be written in **any language**
- Script files may be additional resources to the blueprint or be retrieved via URL
- The Script plugin is defined as an **`agent_host`** plugin, meaning that by default, the script it runs and the plugin itself run on the host of the node whose operation was mapped.

Basic Usage: Writing the Script

- The following short script executes two simple bash commands — one uses echo while the other uses the Cloudify **`ctx-proxy`**, which will be described in detail later on
- Note that the script begins with **`/bin/bash`** — this tells the script plugin that this should be run as a bash script
- To run scripts in other languages, use a different "**`#!`**" line, or use the **`command_prefix`** property which will be described later

```
#! /bin/bash –e
echo “this simply executes an echo command”
ctx logger info “this will create a cloudify log message"
```

Basic Usage: Blueprint Mapping

```
node_templates:
  example_web_server:
    type: cloudify.nodes.WebServer
    interfaces:
      cloudify.interfaces.lifecycle:
        start: scripts/write-stuff.sh
        stop: myplugin.stopper.stop_server
```

Can be a script or a Python `@task` implementation in a plugin

- The **`cloudify.interface.lifecycle.start`** operation is mapped to the **`write-stuff.sh`** script
- The **`write-stuff.sh`** script should appear under a directory named **`scripts`**
- The **`scripts`** directory is expected to exist on the same level as the main blueprint file

Basic Usage: Blueprint Mapping (cont'd)

- Mapping operations to script plugin tasks may use a special shortened syntax, as was seen in the previous slide
- The DSL parser translates this specialized syntax to the standard way for mapping operations
- The previous example is equivalent to this full-syntax example:

```
node_templates:
  example_web_server:
    type: cloudify.nodes.WebServer
    interfaces:
      cloudify.interfaces.lifecycle:
        start:
          implementation: script.script_runner.tasks.run
          inputs:
            script_path: scripts/write-stuff.sh
```

Process Configuration

The script plugin offers various process configuration properties. Notable examples:

- **`cwd`**: sets the working directory for the script
- **`env`**: sets environment variables for the script process
- **`args`**: arguments to pass to the script
- **`command_prefix`**: prefix to add before the script path
  - This may be used instead of the shebang line to run scripts in different languages; for example, when mapping an operation to a PowerShell script, set **`command_prefix`** to be “powershell”

Process Configuration (cont'd)

blueprint.yaml

```
start:
  implementation: scripts/start.sh
  inputs:
    process:
      # This directory should already exist
      cwd: /tmp/workdir
      args: [arg1_value, arg2_value]
      env:
        MY_ENV_VARIABLE: MY_ENV_VARIABLE_VALUE
```

scripts/start.sh

```
#!/bin/bash -e

# will log "current working directory is: /tmp/workdir"
ctx logger info "current working directory is: ${PWD}"

# will log "first arg is: arg1_value"
ctx logger info "first arg is: $1"

# will log "my env variable is: MY_ENV_VARIABLE_VALUE"
ctx logger info "my env variable is: ${MY_ENV_VARIABLE}"
```

Invoking Scripts: Python

blueprint.yaml

```
imports:
  - http://www.getcloudify.org/spec/cloudify/3.2/types.yaml

node_templates:
  example_web_server:
    type: cloudify.nodes.WebServer
    interfaces:
      cloudify.interfaces.lifecycle:
        start: scripts/start.py
```

Invoking Scripts: Ruby

blueprint.yaml

```
imports:
  - http://www.getcloudify.org/spec/cloudify/3.2/types.yaml

node_templates:
  example_web_server:
    type: cloudify.nodes.WebServer
    interfaces:
      cloudify.interfaces.lifecycle:
        start:
          implementation: scripts/start.rb
          inputs:
            process:
              command_prefix: /opt/ruby/bin/ruby
```

Invoking Scripts: PowerShell

blueprint.yaml

```
imports:
  - http://www.getcloudify.org/spec/cloudify/3.2/types.yaml

node_templates:
  example_web_server:
    type: cloudify.nodes.WebServer
    interfaces:
      cloudify.interfaces.lifecycle:
        start:
          implementation: scripts/start.ps1
          inputs:
            process:
              command_prefix: powershell
```

Operation Inputs

- It is possible to pass parameters to the script via the standard mechanism of **operation inputs**
- These inputs will be available in the script runtime environment as shell variables
- Complex data structures will be JSON-encoded
- This is the recommended way of passing parameters to scripts, as opposed to using the process configuration facilities described earlier

Operation Inputs: Example

- Passing the parameter via the operation inputs section:

```
...
cloudify.interfaces.lifecycle:
  start:
    implementation: scripts/server/start-server.sh
    inputs:
      port: 8080
      items:
        - cat
        - dog
      data:
        key1: value1
```

- Using the input in the script via a shell variable:

```
echo "Starting web server..."
nohup python -m SimpleHTTPServer ${port} > /dev/null 2>&1 &
# The list (“items”) and dict (“data”) can be accessed through
# the ctx-proxy, as will be demonstrated shortly.
```

`ctx-proxy`

- The **`ctx-proxy`** is a stand-alone process which enables access to the Cloudify operation context
- It is started automatically by the script plugin when non-Python scripts are executed (unless instructed otherwise)
- Scripts in any language can access the Cloudify context almost seamlessly
- The **`ctx-proxy`** reads commands executed on it, and executes equivalent commands on the real Cloudify context

ctx-proxy: Usage

The following examples are for bash script usage:

- Accessing an attribute:

```
ctx bootstrap-context cloudify-agent agent-key-path
->
ctx.bootstrap_context.cloudify_agent.agent_key_path
```

- Accessing a dictionary/list:

```
ctx instance runtime-properties endpoint.urls[2]
->
ctx.instance.runtime_properties['endpoint']['urls'][2]
```

- Setting a value:

```
ctx instance runtime-properties my_property my_value
->
ctx.instance.runtime_properties['my_property'] = 'my_value'
```

- Method invocation:

```
ctx download-resource images/hello.png
->
ctx.download_resource('images/hello.png')
```

`ctx-proxy`: Usage (cont'd)

- Working with non-string arguments is done by signaling the proxy that a certain argument should be parsed as JSON, by prefixing it with `@`:

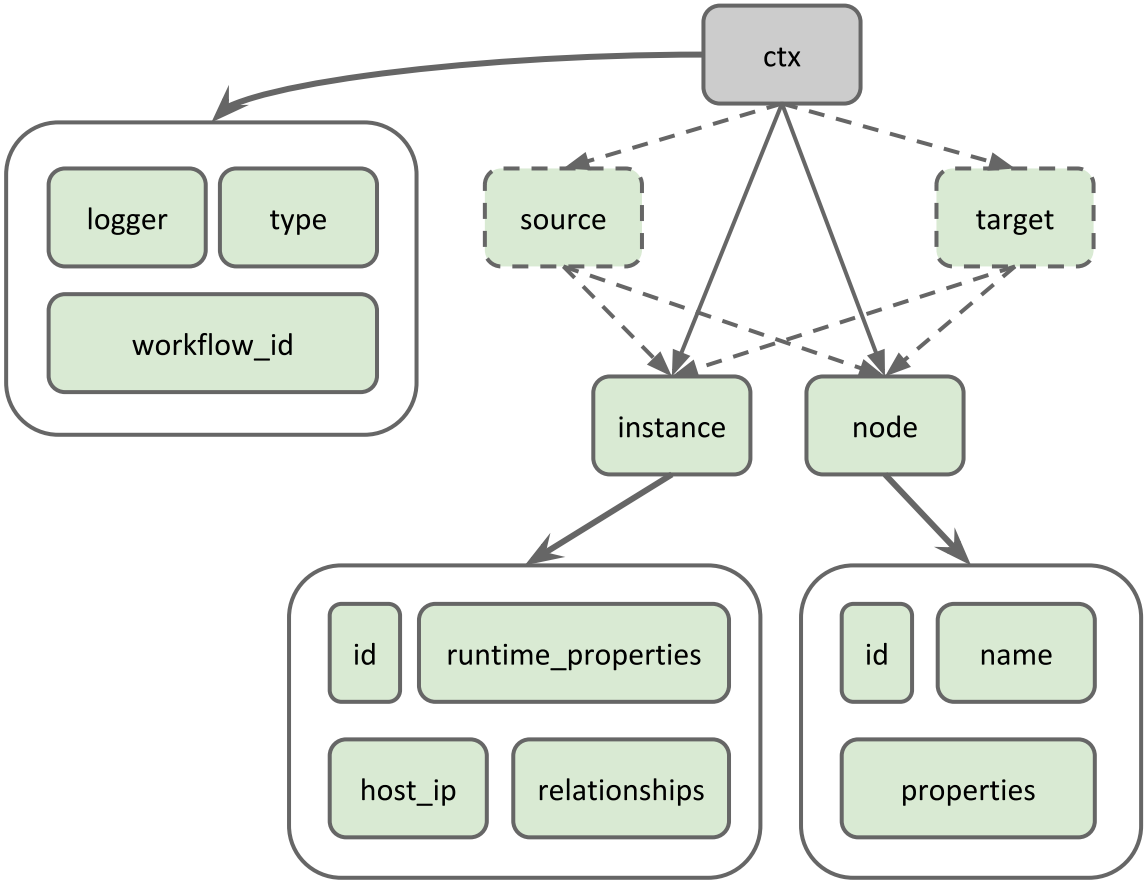
```
ctx instance runtime-properties number_of_clients @14
->
ctx.instance.runtime_properties['number_of_clients'] = 14
```

- Returning a value from the script is possible like so:

```
ctx returns my_value
```

Note that this command won't cause the termination of the script, so it's recommended for it to be the last command in the script.

Operation Context: Available Data



Operation Context: Available Data (cont'd)

- **`instance`**: the node instance the operation is executed for
  - **`id`**: the node instance's ID
  - **`runtime_properties`**: the node instance's runtime properties as a read-only dictionary
  - **`host_ip`**: the node instance host ip address
  - **`relationships`**: list of this instance relationships
- **`node`**: the node that the operation is executed for
  - **`id`**: the node's ID
  - **`name`**: the node's name
  - **`properties`**: the node's properties as a read-only dictionary; these properties are the properties specified in the blueprint

Operation Context: Available Data (cont'd)

- **`source`**: provides access to the relationship's source node and node instance
- **`target`**: provides access to the relationship's target node and node instance
- **`logger`**: Cloudify's context-aware logger
- **`type`**: the context's type
  - **`DEPLOYMENT`**
  - **`NODE_INSTANCE`**
  - **`RELATIONSHIP_INSTANCE`**
- **`workflow_id`**: the workflow ID that the plugin was invoked from (install, uninstall, etc.)

Script Plugin: Python Scripts

- Python scripts are handled differently by the script plugin — instead of being launched as a different process, they get evaluated within the script plugin's operation
- This allows for script writers to use the full standard plugin API, without having to write a full blown plugin, and without having to use the **`ctx-proxy`**
- Script example (this is an actual script, not an excerpt):

```
from cloudify import ctx
ctx.logger.info('Just logging the web server port: {0}'.format(ctx.
node.properties['port']))
```

Script Plugin: Misc.

- Relationship operations may also be mapped using the Script plugin
  - The script plugin is **`agent_host`**
  - Will run on the source's host if mapped through **`source_interface`**
  - Will run on the target's host if mapped through **`target_interface`**
- The script plugin also supports mapping workflows scripts
  - Must be Python

Script Plugin: Misc. (cont'd)

When using **`nohup`** in your scripts, don't forget to redirect **`stdout`** and **`stderr`** to **`/dev/null`** and run the operation in the background using **`&`**, e.g.:

```
nohup python -m SimpleHTTPServer > /dev/null 2>&1 &
```

Failing to do so will cause the process which was run (**`SimpleHTTPServer`** in this case) to terminate when the script plugin does.

Fabric Plugin

- Similar to the Script plugin: can be used to map operations to scripts
- Difference: Fabric plugin can run remote commands via SSH
- Several modes of operation

Fabric Plugin: Scripts Runner Mode

- Main mode of the Fabric plugin
- Acts very similarly to the Script plugin, except that the script will run completely remotely
- The plugin code itself runs on the Cloudify Manager (as opposed to the Script plugin), whereas the script itself runs on the host of the node whose operation was mapped

Fabric Plugin: Scripts Runner Mode (cont'd)

- Mapping of the operation is done to a script at a relative path or a URL, like in the Script plugin
- Supports a process configuration input which is similar to the one supported by the Script plugin
- Operation inputs are also passed and used in the same way as they are used in the Script plugin
- This mode does not support special handling of Python scripts
  - To evaluate Python scripts within the plugin operation, see the other modes of the Fabric plugin

Fabric Plugin: Scripts Runner Mode (cont'd)

- When using the Fabric plugin in this mode rather than the Script plugin, there is no need for a Cloudify agent to be installed on the host VM
- If running scripts is the only usage made with the given agent, it might be possible to forego the agent completely on that specific VM
- Note that port 22 (SSH) must still be open

Fabric Plugin: Commands Runner Mode

- This is the simplest mode, fitting mostly to very simple tasks
- The Fabric plugin will simply run each command from a list of commands on the remote host
- Note that in this mode, the **`ctx-proxy`** is not available
- Mapping example:

```
node_templates:
  example_node:
    type: cloudify.nodes.WebServer
    interfaces:
      cloudify.interfaces.lifecycle:
          start:
            implementation: fabric.fabric_plugin.tasks.run_commands
            inputs:
              commands:
                - echo "source ~/myfile" >> ~/.bashrc
                - apt-get install -y python-dev git
                - pip install my_module
```

Fabric Plugin: Tasks Runner Mode

- In this mode, Fabric runs a Python method from a script in a relative path to the main blueprint file
- Mapping example:

```
node_templates:
  example_node:
    type: cloudify.nodes.WebServer
    interfaces:
      cloudify.interfaces.lifecycle:
        start:
          implementation: fabric.fabric_plugin.tasks.run_task
          inputs:
            tasks_file: my_tasks/tasks.py
            task_name: install_nginx
            task_properties:
              important_prop1: very_important
```

Fabric Plugin: Tasks Runner Mode (cont'd)

- The script may make normal use of the **`ctx`** object, as well as various constructs from the Fabric API — whose environment is automatically configured to target the remote machine
- Script example:

```
from fabric.api import run, put
from cloudify import ctx

def install_nginx(important_prop1, important_prop2):
    ctx.logger.info('Installing nginx‘)
    run('sudo apt-get install nginx')


def configure_nginx(config_file_path):
    conf_file = ctx.download_resource(config_file_path)
    put(conf_file, '/etc/nginx/conf.d/')
```

Fabric Plugin: Module Task Runner Mode

- Very similar to the Task Runner mode, except that the script's module is referenced by Python path rather than by file system path
- Mapping example:

```
node_templates:
  example_node:
    type: cloudify.nodes.WebServer
    interfaces:
      cloudify.interfaces.lifecycle:
        start:
          implementation: fabric.fabric_plugin.tasks.run_module_task
          inputs:
            task_mapping: some_package.some_module.install_nginx
            task_properties:
              important_prop1: very_important
```

SSH Configuration

- All of the Fabric plugin modes require SSH configuration
- The configuration will default to the following:
  - **`user`**: the agents user as defined during bootstrap
  - **`key_filename`**: the agents private key as defined during bootstrap
  - **`host_string`**: the IP of the host of the node whose operation was mapped. The IP is taken from either the node's properties or from the node instance's runtime properties.

SSH Configuration (cont'd)

These configurations, as well as any other configuration available in the underlying Fabric library's environment configuration, may be set as operation inputs, such as:

```
...
start:
  implementation: fabric.fabric_plugin.tasks.run_commands
  inputs:
    commands: [touch ~/my_file]
    fabric_env:
      host_string: 192.168.10.13
      user: some_username
      key_filename: /path/to/key/file
```

Script Plugin → Fabric Plugin Conversion

- Simple conversion, as the two plugins are similar
- Differences:
  - Operation mappings
  - No sugared mapping syntax in Fabric
  - Fabric plugin must be imported in the blueprint

Script Plugin → Fabric Plugin Conversion (cont'd)

Importing the Fabric plugin in the blueprint:

```
imports:
    - http://www.getcloudify.org/spec/fabric-plugin/1.2/plugin.yaml
```

This is the **`http_web_server`** node before any changes (mapped to the Script plugin)

```
http_web_server:
    type: cloudify.nodes.WebServer
    properties:
      port: { get_input: webserver_port }
    relationships:
      - type: cloudify.relationships.contained_in
        target: vm
    interfaces:
      cloudify.interfaces.lifecycle:
        configure: scripts/configure.sh
        start: scripts/start.sh
        stop: scripts/stop.sh
```

Script Plugin → Fabric Plugin Conversion (cont'd)

Converted **`http_web_server`** node — operations are mapped to the Fabric plugin

```
http_web_server:
    type: cloudify.nodes.WebServer
    properties:
      port: { get_input: webserver_port }
    relationships:
      - type: cloudify.relationships.contained_in
        target: vm
    interfaces:
      cloudify.interfaces.lifecycle:
        configure:
          implementation: fabric.fabric_plugin.tasks.run_script
          inputs:
            script_path: scripts/configure.sh
        start:
          implementation: fabric.fabric_plugin.tasks.run_script
          inputs:
            script_path: scripts/start.sh
        stop:
          implementation: fabric.fabric_plugin.tasks.run_script
          inputs:
            script_path: scripts/stop.sh
```

Script Plugin → Fabric Plugin Conversion (cont'd)

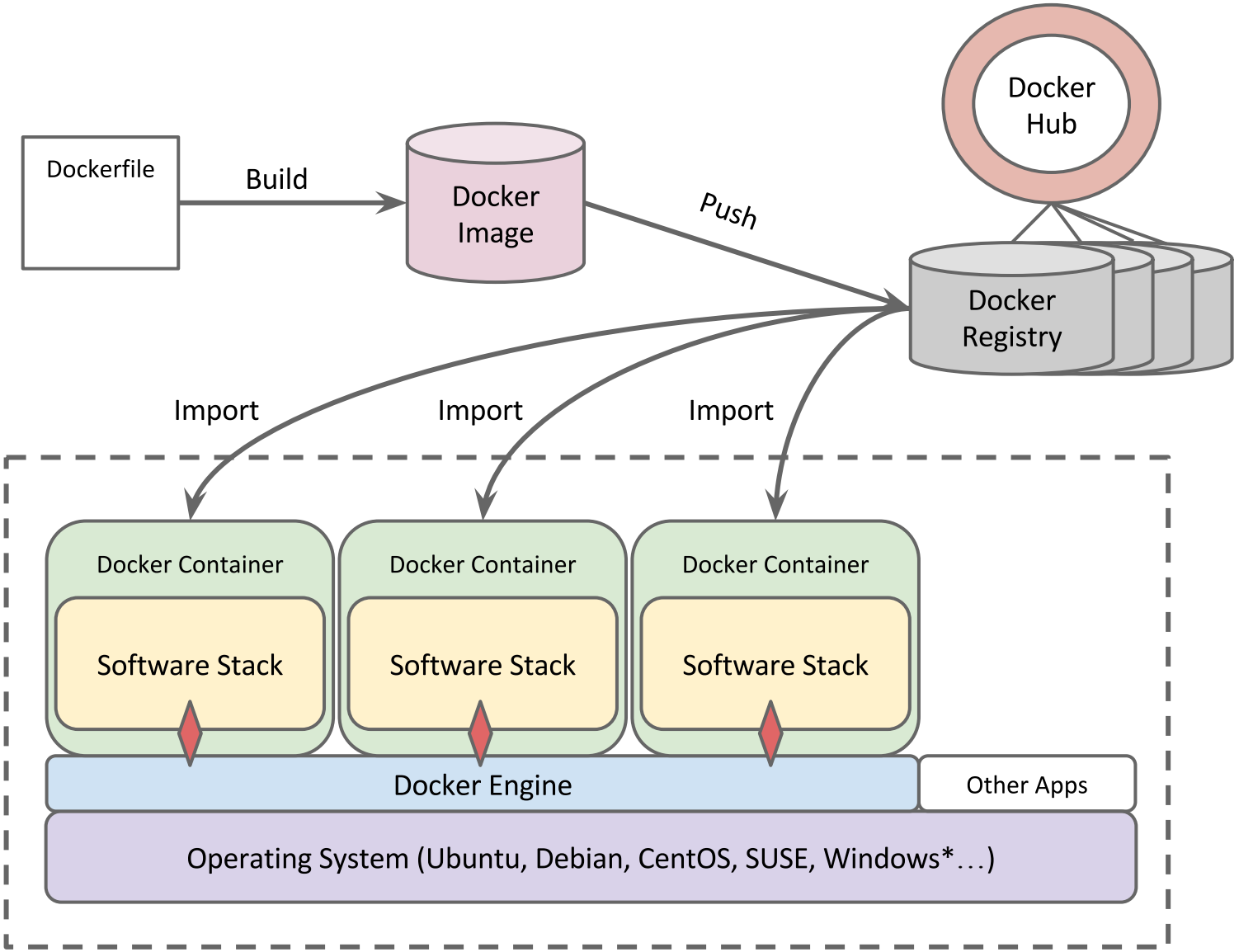
Finally, it is now possible to completely skip the Cloudify agent installation on that Compute node:

```
vm:
  type: cloudify.openstack.nodes.Server
  properties:
    install_agent: false
    cloudify_agent:
      user: { get_input: agent_user }
    server:
      image_name: { get_input: image_name }
      flavor_name: { get_input: flavor_name }
  relationships:
    - type: cloudify.openstack.server_connected_to_floating_ip
      target: virtual_ip
    - type: cloudify.openstack.server_connected_to_security_group
      target: security_group
```

Docker: Introduction

- Provides cross-platform mechanism for packaging and running software stacks
- Software is packaged into *Docker Images* and, optionally, uploaded to a *Docker Repository*
- Images define interfacing points:
  - Ports
  - File systems
- *Docker Images* are imported and *Docker Containers* created off them:
  - A Docker Container contains the delta between the source Docker Image and the underlying OS
  - Containers can be started, stopped and deleted
  - Can run on any OS that has a Docker Engine installed

Docker: High-Level View



Docker Image

- Proprietary packaging structure
- Created based on a Docker File (conventionally named **`Dockerfile`**)
  - **`Dockerfile`** contains maintainer information and instructions how to install the software stack
- Always created off a *base image*
  - Off-the-shelf examples: **`ubuntu`**, **`mongo`**
  - Users can create custom base images

Docker Image (cont'd)

- Uses a Union File System (UnionFS) to achieve layering
  - When updating an image, no need to recreate — simply add a new layer on top of the existing ones, containing the delta
- Serves as a read-only template, off which *Docker Containers* are created
- Stored in one or more Docker Registries

Docker Registry

- Holds Docker images
- Can be private or public
- Public registries delegate authentication to the Docker Hub
- Supports various storage backends
  - Cloud storage (such as S3)
  - Local storage
  - Etc.
- Examples of public Docker registries:
  - Ubuntu
  - MySQL
  - Nginx

The Docker Hub

- http://hub.docker.com
- Central hub for public Docker registries
- Owned and operated by Docker Inc.
- Provides authentication and authorization services for registries
- Provides a UI for searching and managing repositories

Docker Containers

- Conceptually: the instantiation of a Docker Image on a host
- Physically stored as a directory on the host where the Docker Engine runs (under **`/var/lib/containers`**)
- Provide an isolated environment for the contained software stack to operate
- Controlled by the Docker Engine (create, start, stop, remove, and others)
- Exposes ports (declared in the applicable image's Dockerfile)
- Can mount and use external volumes

The Docker Engine

- A daemon, available for virtually any Linux distribution
  - Windows supported as well, but requiring installation of a lightweight VM
- Uses Linux kernel mechanisms (such as namespaces) to provide isolation to Docker Containers
- Manages containers and images on the host
- Stores all data in **`/var/lib/docker`**
- Accessed by through a Docker client
  - Command-line client
  - Python client (*Docker-Py*, also used by the Cloudify Docker Plugin)
  - Other clients available

Docker: Running a Container

```
$ sudo docker run -i -t ubuntu /bin/bash
```

1. Pulls the ubuntu image from the Docker Hub (if it isn't already available locally)
2. Creates a new container
3. Allocates a file-system and mounts a read-write layer
4. Allocates network resources
5. Sets up an IP address
6. Executes the specified process (**`/bin/bash`** in this example)
7. Captures and provides application output

Docker Plugin

- Uses the **`Docker-Py`** package for executing Docker functions
- Defines the **`cloudify.docker.Container`** type
  - Adds properties: **`name`**, **`image`**
  - Adds lifecycle operations for creating, starting, stopping and deleting Docker containers
- Flexible; can adapt to any application topology consisting of arbitrary Docker containers

Docker Plugin (cont'd)

- The plugin does **not** install Docker
  - Must either already exist, or be installed as part of the Compute node's creation process
  - Can be done by adding a script to the application blueprint, via **`userdata`**:

```
vm_with_docker:
  derived_from: cloudify.openstack.nodes.Server
  properties:
    server:
      default:
        userdata: |
          #!/bin/bash
          sudo service ssh stop
          curl -o install.sh -sSL https://get.docker.com/
          sudo sh install.sh
          sudo groupadd docker
          sudo gpasswd -a ubuntu docker
          sudo service docker restart
          sudo service ssh start
```

Docker Plugin: Container Selection

Selection accomplished through the **`image`** dictionary property:

- **`src`**: file or URL to obtain image from
- **`repository`**, **`tag`**:
  - If **`src`** is provided: the name/tag to assign to the newly-created repository
  - Otherwise: the name/tag of the repository to pull from

```
app_server:
  type: cloudify.docker.Container
  properties:
    name: app_server_container
    image:
      repository: library/tomcat
      tag: jre8
```

Docker Plugin: Relationships

- Container nodes have a contained-in relationship with a Compute node.
- Container nodes can have dependency relationships with other Container nodes.

```
vm_1:
  type: cloudify.openstack.nodes.Server

container_1:
  type: cloudify.docker.Container
  relationships:
    - type: cloudify.relationships.contained_in
      target: vm_1

container_2:
  type: cloudify.docker.Container
  relationships:
    - type: cloudify.relationships.contained_in
      target: vm_1
    - type: cloudify.relationships.depends_on
      target: container_1
```

Docker Plugin: Lifecycle Operations

- **`create`**: create the container
- **`start`**: start the container
- **`stop`**: stop the container
- **`delete`**: delete the container

```
interfaces:
  cloudify.interfaces.lifecycle:
    create:
      implementation: docker.docker_plugin.tasks.create_container
    start:
      implementation: docker.docker_plugin.tasks.start
    stop:
      implementation: docker.docker_plugin.tasks.stop
    delete:
      implementation: docker.docker_plugin.tasks.remove_container
```

Docker Plugin: Inputs

- One blueprint input parameter for each lifecycle operation, called **`params`**
  - A dictionary, which maps directly to input parameters of the called function in Docker-Py

blueprint.yaml

```
start:
    implementation: docker.docker_plugin.tasks.start
    inputs:
      params:
        port_bindings:
          27017: 27017
          28017: 28017
```

docker-py/docker/client.py

```
def start(self, container, binds=None,  port_bindings=None, lxc_conf=None,
              publish_all_ports=False, links=None, privileged=False,
              dns=None, dns_search=None, volumes_from=None,
              network_mode=None, restart_policy=None, cap_add=None,
              cap_drop=None, devices=None, extra_hosts=None,
              read_only=None, pid_mode=None):
```

Docker Plugin: Lifecycle Operations

- Map to Docker-Py methods:

  https://docker-py.readthedocs.org/en/latest/api/

Lifecycle Operation	Plugin Operation	Docker-Py Method
`create`	`docker.docker_plugin.tasks.create_container`	`create_container`
`start`	`docker.docker_plugin.tasks.start`	`start`
`stop`	`docker.docker_plugin.tasks.stop`	`stop`
`delete`	`docker.docker_plugin.tasks.remove_container`	`remove_container`

Docker Plugin: `create_container`

Creates a Docker Container.

Main Docker-Py parameters:

- **`command`**: the command to run inside the container
- **`ports`**: list of port numbers to be exposed

```
create:
  implementation: docker.docker_plugin.tasks.create_container
  inputs:
    params:
      ports:
        - 27017
        - 28017
      command: mongod --rest --httpinterface --smallfiles
```

Docker Plugin: `start`

Starts a Docker Container.

Main Docker-Py parameters:

- **`binds`**: list of volumes to bind
- **`port_bindings`**: a dictionary of port bindings

Additional parameters:

- **`processes_to_wait_for`**: list of processes that Cloudify should wait for, before starting the container
- **`retry_interval`**: post-start status check interval

```
start:
    implementation: docker.docker_plugin.tasks.start
    inputs:
      params:
        port_bindings:
          27017: 27017
      processes_to_wait_for:
        - /bin/sh
      retry_interval: 10
```

Docker Plugin: `stop`

Stops a Docker Container.

Main Docker-Py parameters:

- **`timeout`**: time interval to wait for shutdown before **`SIGKILL`** is sent

Additional parameters:

- **`retry_interval`**: post-stop status check interval

```
stop:
    implementation: docker.docker_plugin.tasks.stop
    inputs:
      params:
        timeout: 60
      retry_interval: 10
```

Docker Plugin: `remove_container`

Removes a Docker Container.

Main Docker-Py parameters:

- **`force`**: use **`SIGKILL`** to remove the container

Additional parameters:

- **`retry_interval`**: post-delete status check interval

```
stop:
    implementation: docker.docker_plugin.tasks.remove_container
    inputs:
      params:
        force: true
      retry_interval: 10
```

LAB 5: DEPLOYING DOCKER CONTAINERS

Additional Information

- Cloudify Documentation:
  - Script plugin
  - Fabric plugin
  - Docker Plugin
- Usage Examples:
  - Cloudify hello-world example
  - Cloudify nodecellar example

INTRODUCTION TO WORKFLOWS

Workflows

- Workflow: a unit of orchestration
- Written in Python, using dedicated APIs and framework
- Every deployment has its own set of workflows
  - Declared in the blueprint
  - Workflow executions happen within the context of a deployment
- Executing workflows from the CLI:

```
cfy executions start -w my_workflow -d my_deployment
```

Output:

```
Executing workflow 'my_workflow' on deployment 'my_deployment' at management server
11.0.0.7 [timeout=900 seconds]
2014-12-04T10:02:47 CFY <my_deployment> Starting 'my_workflow' workflow execution
2014-12-04T10:02:47 CFY <my_deployment> 'my_workflow' workflow execution succeeded
Finished executing workflow 'my_workflow' on deployment 'my_deployment'
* Run 'cfy events list --include-logs --execution-id 7cfd8b9c-dcd6-41bc-bc88-
6aa0b00ffa62' for retrieving the execution's events/logs
```

Workflow: Execution Parameters

```
> cfy executions start -d my_deployment -w my_workflow -p my_parameters.yaml
```

my_parameters.yaml

```
mandatory_parameter: mandatory_parameter_value
nested_parameter:
     key1: overridden_value
```

Workflow: Execution Statuses

The workflow *execution status* is stored in the **`status`** field of the **`Execution`** object:

`pending`	execution is waiting for a worker to start it
`started`	execution is currently running
`cancelling`	execution is currently being cancelled
`force_cancelling`	execution is currently being force-cancelled
`cancelled`	execution has been cancelled
`terminated`	the execution has terminated successfully
`failed`	the execution has failed; an execution with this status should have an error message available under the execution object's error field

Built-In Workflows

- Cloudify provides built-in workflows:
  - **`install`**
  - **`uninstall`**
  - **`execute_operation`**: for executing arbitrary logic
  - **`heal`**: for healing a malfunctioning subset of the topology
  - **`scale`**: for scaling (in or out) a subset of the topology
- Built-in workflows described in **`types.yaml`**
- Full documentation available at:

http://getcloudify.org/guide/3.2/reference-builtin-workflows.html

Built-In Workflows: `types.yaml`

```
workflows:
    install: default_workflows.cloudify.plugins.workflows.install
    uninstall: default_workflows.cloudify.plugins.workflows.uninstall
    execute_operation:
        mapping: default_workflows.cloudify.plugins.workflows.execute_operation
        parameters:
            operation: {}
            operation_kwargs:
                default: {}
            ...

    heal:
        mapping: default_workflows.cloudify.plugins.workflows.
auto_heal_reinstall_node_subgraph
        parameters:
            node_id:
            ...

    scale:
        mapping: default_workflows.cloudify.plugins.workflows.scale
        parameters:
            node_id:
            ...
```

Built-In Workflow: `install`

- Installs the application
- The application's blueprint is used to determine:
  - Node instantiation
  - Node instantiation order
  - Relationships between nodes
- Used by the **`heal`** and **`scale`** workflows

Built-In Workflow: `uninstall`

- Semantic opposite of the **`install`** workflow
- Application blueprint determines:
  - Node relationships to unlink
  - Nodes to remove
- Used by the **`heal`** and **`scale`** workflows

Built-In Workflow: `heal`

- Executes **`uninstall`** and then **`install`** for a subset of the topology
- Input:
  - **`node_instance_id`**: the instance of the node to heal
- Process:
  - Finds the enclosing Compute node
  - Uninstalls, and then reinstalls, that Compute node
  - Re-establishes all applicable node relationships

Built-In Workflow: `scale`

- Input:
  - **`node_id`**: ID of the node to scale
  - **`scale_compute`**: whether or not to scale the enclosing Compute node
  - **`delta`**: number of node instances to add or subtract
- Process:
  - delta > 0: run **`install`** against input node
  - delta < 0: run **`uninstall`** against input node
  - **`scale_compute`** == **`true`**: scale the enclosing Compute node

Built-In Workflow: `execute_operation`

- Input:
  - **`operation`**: a Python @operation
  - **`operation_kwargs`**: arguments to operation
  - **`run_by_dependency_order`**: Dependency order / parallel indication
  - Filters:
    - **`type_names`**
    - **`node_ids`**
    - **`node_instance_ids`**
- Process:
  - Calculates all nodes on which to execute operation
  - Executes the operation, either sequentially or in parallel

Custom Workflows

- Implemented as Python functions
- Must be decorated with the **`cloudify.decorators.workflow`** decorator from the module of the **`cloudify-plugins-common`** package
- May import **`cloudify.workflows.ctx`**
  - Context data
  - System services

Custom Workflows (cont'd)

Typical workflow method signature:

```
from cloudify.decorators import workflow
from cloudify.workflows import ctx

@workflow
def my_workflow(**kwargs):
    pass
```

Workflow parameter lists should use named parameters and terminate with **`**kwargs`**

Custom Workflows: Mapping

```
workflows:
  my_workflow:
    mapping: my_workflow_plugin_name.my_workflow_module_name.my_workflow_method_name
    parameters:
      mandatory_parameter:
        description: this parameter is mandatory
      optional_parameter:
        description: this parameter is optional
        default: optional_parameter_default_value
      nested_parameter:
        description: >
          this parameter is also optional,
          it's default value has nested values.
        default:
          key1: value1
          key2: value2
```

Having a **`default`** makes a parameter optional.

Custom Workflows: Workflow Plugins

When packaged, a workflow implementation is considered a *workflow plugin*. Blueprints can import the workflow using the same syntax as for *operation plugins*:

```
plugins:
  my_workflow_plugin_name:
    executor: central_deployment_agent
    source: http://example.com/url/to/plugin.zip
```

Executing Workflows Locally Using `cfy local`

Installs required plugins locally:

```
cfy local init --install-plugins -p my_blueprint.yaml -i inputs.yaml
```

Executes **`install`** workflow:

```
cfy local execute -w install
```

Displays the resulting output:

```
cfy local outputs
```

LAB 6: WORKFLOWS

INTRODUCTION TO MONITORING

Monitoring

- Diamond: a Python daemon that collects system metrics and publishes them to multiple destinations
  - Monitors CPU, memory, network, I/O, load and many other metrics
  - Offers API for custom collectors
  - More information in the documentation
- Cloudify's Diamond plugin installs and configures monitoring agents on hosts

Monitoring: The Diamond Plugin

Two interfaces are involved in setting up a monitoring agent on a machine:

- **`cloudify.interfaces.monitoring_agent`**: the interface for installing, starting, stopping and uninstalling the agent
- **`cloudify.interfaces.monitoring`**: the interface for configuring the monitoring agent

Monitoring: The Diamond Plugin (cont'd)

Monitoring Agent:

```
imports:
  - http://www.getcloudify.org/spec/cloudify/3.2/types.yaml
  - http://www.getcloudify.org/spec/diamond-plugin/1.1/plugin.yaml

node_templates:
  host:
    type: cloudify.nodes.Compute
    ...
    interfaces:
      cloudify.interfaces.monitoring_agent:
          install:
            implementation: diamond.diamond_agent.tasks.install
            inputs:
              diamond_config:
                interval: 1
          start: diamond.diamond_agent.tasks.start
          stop: diamond.diamond_agent.tasks.stop
          uninstall: diamond.diamond_agent.tasks.uninstall
```

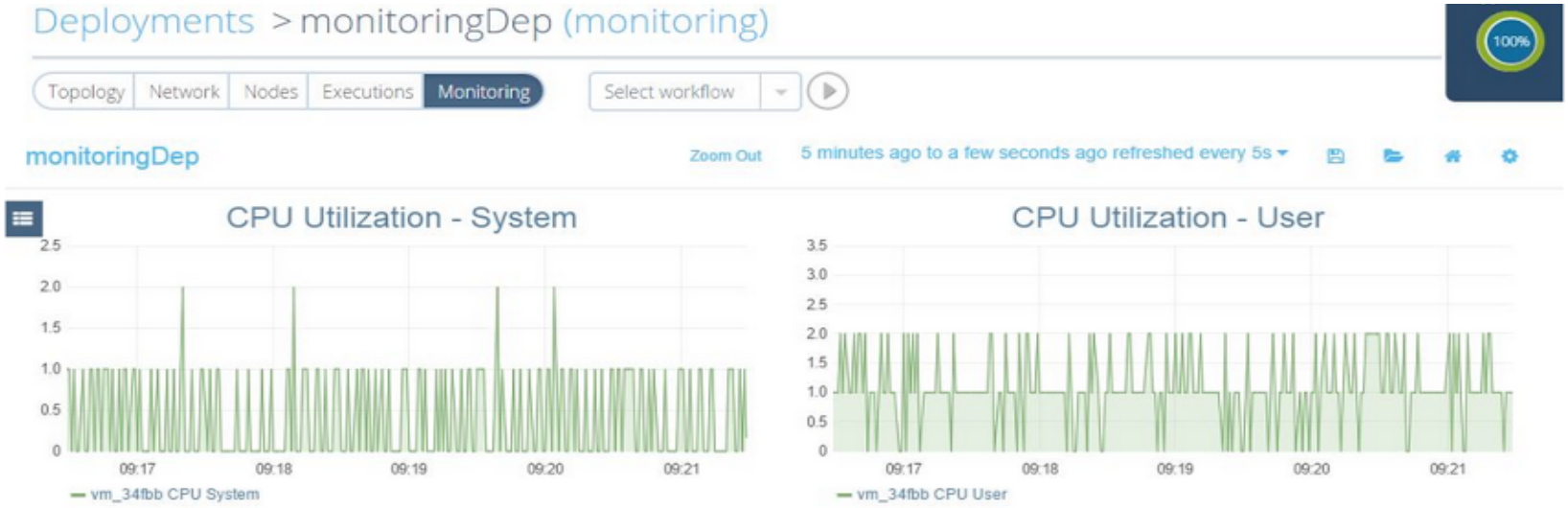
Monitoring: The Diamond Plugin (cont'd)

Monitoring Collectors:

```
mongod:
    type: nodecellar.nodes.MongoDatabase
    ...
    interfaces:
      cloudify.interfaces.lifecycle:
        configure: scripts/mongo/install-pymongo.sh

      cloudify.interfaces.monitoring:
          start:
            implementation: diamond.diamond_agent.tasks.add_collectors
            inputs:
              collectors_config:
                MongoDBCollector:
                  config:
                    hosts: "localhost:27017"
    relationships:
      - type: cloudify.relationships.contained_in
        target: host
```

Monitoring UI: Grafana

- **Grafana** is an open source, feature-rich metrics dashboard and a graph editor.
- Cloudify uses Grafana for displaying system metrics
  - The monitoring section can be found on each deployment's page in the Cloudify UI

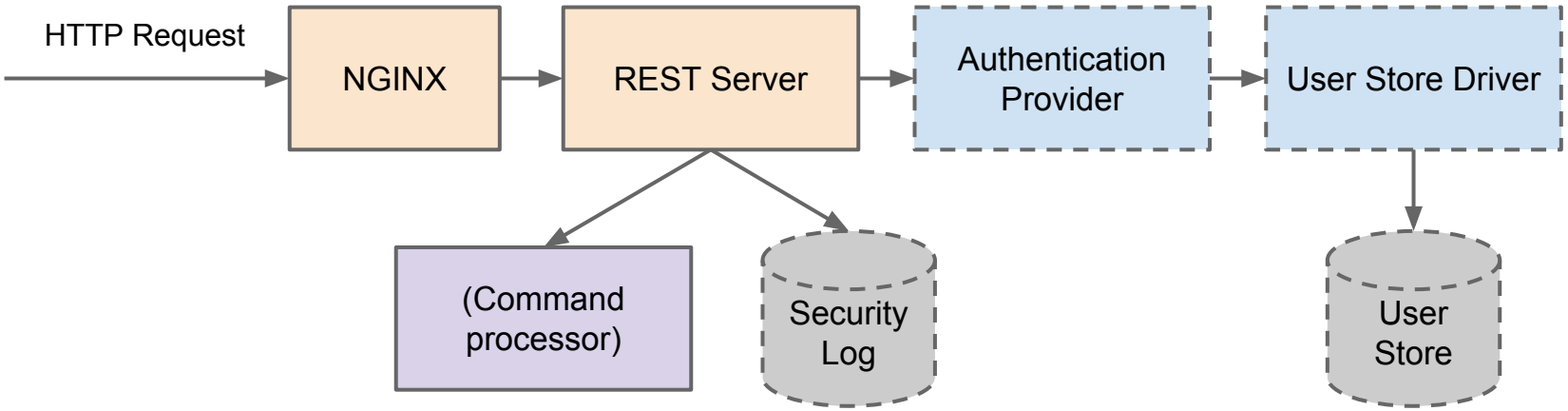


LAB 7: DEPLOYING COLLECTORS AND USING THE GRAFANA DASHBOARD

INTRODUCTION TO SECURITY

Security Architecture

- *Flask-SecuREST*: an in-house developed extension to the Flask-RESTful framework
- Offers a pluggable authentication mechanism
- Handles all requests processed by the REST Server



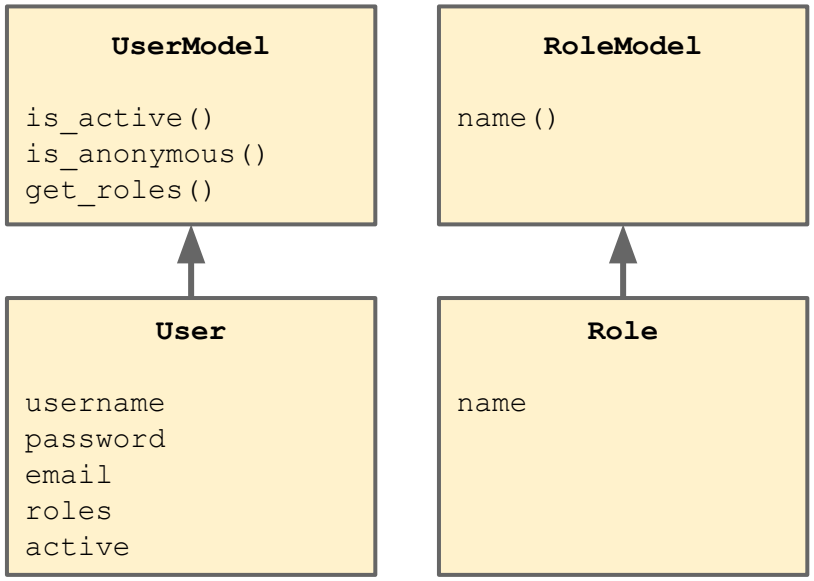
Security Architecture (cont'd)

- Security logging
  - Configurable: target file and logging level
- Support for SSL
- Security configuration: at the manager's blueprint, under **`manager` → `properties` → `security`**

```
security:
  enabled: ...
  log_file: ...
  log_level: ...
  userstore_driver:
    ...
  authentication_providers:
    ...
  auth_token_generator:
    ...
  ssl:
    enabled: false
    certificate_path: ""
    private_key_path: ""
```

User / Role Model

- Abstract API for use by
  - Authentication providers
  - User store drivers
- Defined in **`flask-securest.models`**
- Simple implementations defined as well
  - **`class User(UserModel)`**
  - **`class Role(RoleModel)`**



Security Configuration

- **`enabled`**: whether to enable authentication
  - When disabled, SSL is also disabled
- **`log_file`**: security logging destination
- **`log_level`**: security logging level
  - **`OFF`**, **`CRITICAL`**, **`ERROR`**, **`WARNING`**, **`INFO`**, **`DEBUG`**
- **`ssl`**:
  - **`enabled`**: whether to enable SSL
    - When **`true`**, non-SSL HTTP access is forbidden
  - **`certificate_path`**: location of the certificate to present for connection requests
  - **`private_key_path`**: location of the certificate's private key

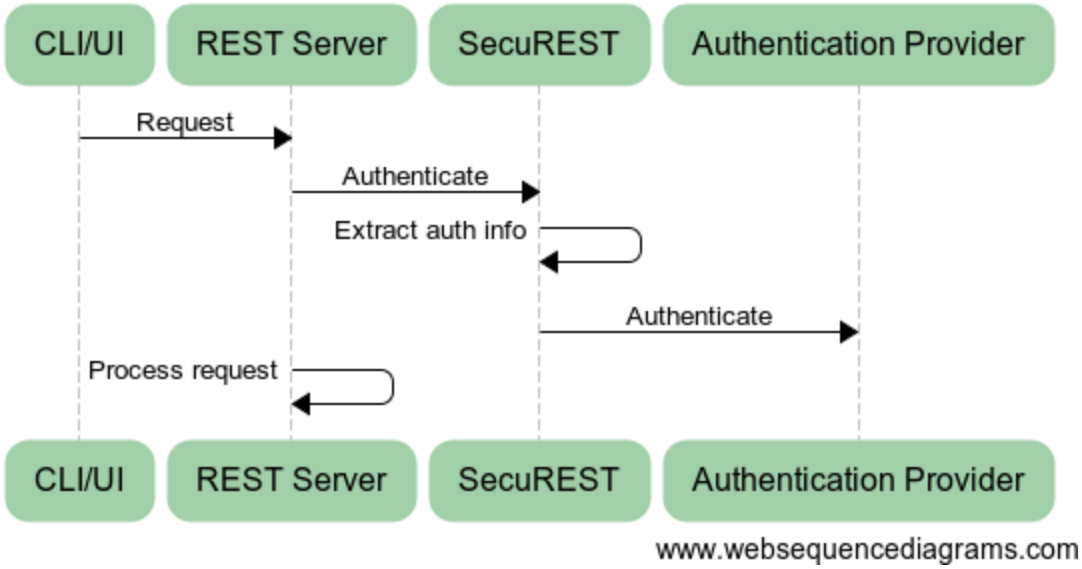
Security Configuration (cont'd)

- **`authentication_providers`**: list of authentication providers
- **`userstore_driver`**: implementation class for user store access
- **`auth_token_generator`**: implementation class for generating authentication tokens

Authentication Providers

- Authenticate users
  - Input: Authentication information from request, provided by SecuREST
  - Output: an instance of a `UserModel` implementation
- Typically, use a User Store Driver to load users' data
- Provided implementations:
  - **Password**: authenticates against password stored in the user store
  - **Token**: authenticates against a predetermined token
- Multiple providers can be configured

Authentication Flow



Authentication Flow (cont'd)

1. On startup, SecuREST instantiates:
   a. Authentication providers
   b. User store driver (optional)
2. REST request processing:
   a. REST server asks SecuREST to authenticate
   b. SecuREST extracts authentication information
   c. All authentication providers, in sequence, authenticate the user
   d. Authentication considered successful upon first success

Token Generation

- Request to **`/tokens`** generates a token for an authenticated client
- A token generator must be configured under **`auth_token_generator`**
  - Class must implement the **`generate_auth_token`** method
- Returned token can later be used for token authentication
- Token generator and token authenticator must agree on token format

Built-In Authentication: Password

```
authentication_providers:
  - name: password
    implementation: flask_securest.authentication_providers.password:
PasswordAuthenticator
    properties:
      password_hash: plaintext
```

- Reads the user's information using the User Store Driver
- Calculates a hash of the provided password, according to the **`password_hash`** property
  - **`plaintext`**: passwords stored as plain text
  - Other options: **`bcrypt`**, **`des_crypt`**, **`pbkdf2_sha256`**, **`pbkdf2_sha512`**, **`sha256_crypt`**, **`sha512_crypt`**
- Compares hashed password to the password read from user store

Built-In Authentication: Token

```
authentication_providers:
  - name: token
    implementation: flask_securest.authentication_providers.token:
TokenAuthenticator
    properties:
      secret_key: my_secret
auth_token_generator:
  implementation: flask_securest.authentication_providers.token:
TokenAuthenticator
  properties:
    secret_key: my_secret
    expires_in_seconds: 600
```

- Same class performs both token generation and authentication
- Uses a predetermined secret key for token generation
  - **`secret_key`** value must be identical for both generation and authentication
- Supports an expiry time property

User Store Drivers

- Responsible for retrieving a user's information by key
  - The returned object is an instance of **`UserModel`**
- Driver provided out-of-the-box:
  - File (YAML, inline with the manager's blueprint)
- Only one driver may be configured
- Optional, but commonly used by authentication providers

Built-In User Store Driver: File

```
userstore_driver:
  implementation: flask_securest.userstores.simple:SimpleUserstore
  properties:
    userstore:
      user1:
        username: example_user
        password: example_password
        email: example_user@your_domain.dom
    identifying_attribute: username
```

- Users defined inline in the manager's blueprint
- **`username`**, **`password`** and **`email`** are required
  - Other arbitrary fields may be defined
- **`identifying_attribute`** designates which attribute to use as a discriminator
  - Usually **`username`**, however can be changed to any field

Built-In User Store Driver: File (cont'd)

Initialization:

1. Read all user store data from the blueprint

Retrieving a user by an identifier:

1. Cycle through user store
2. For each user, compare the discriminator field to the provided identifier
3. Once found, returns a **`User`** object initialized with the user record's **`username`**, **`password`** and **`email`** fields

Customizations

- Authentication providers
  - By extending **`AbstractAuthenticationProvider`**
- User store drivers
  - By extending **`AbstractUserstore`**
- Custom implementation: packaged as a Python package
- Declared in the security configuration section of the manager's blueprint
  - **`node_templates`** → **`manager`** → **`properties`** → **`cloudify`** → **`plugins`**

Customizations (cont'd)

```
node_templates:
  manager:
    properties:
      cloudify:
        plugins:
          my_auth_provider:
            source: my_extensions/simple_authentication_provider
            install_args: '--pre'


          my_userstore:
            source: https://github.com/my_auth_provider/archive/master.zip
```

Implementation is packaged with the blueprint

Implementation is retrieved from a URL

- **`install_args`**: passed to "**`pip install`**"

LAB 8: SECURITY

WORKING WITH ADDITIONAL CLOUDS

WORKING WITH OPENSTACK

What Is OpenStack?

- Started in 2010 by Rackspace and NASA
  - Over 200 companies participate: AT&T, Cisco, EMC, HP, IBM, Oracle, Red Hat, and more...
- Available on Ubuntu since 2011 and Red Hat since 2012
- Non-profit foundation
  - Directed by premium contributors
  - Technical committee drives software development
- Wide adoption
  - eBay, Intel, NASA, AT&T, PayPal, Wikimedia, Yahoo…

What Is OpenStack? (cont'd)

- Allows organizations to use and offer Cloud services
- Examples
  - Public Cloud: vendor owns infrastructure and offers Cloud services via the internet
  - On-premises: installed and managed in-house
  - Hosted Private Cloud: Customer-provided infrastructure, vendor-provided services. Accessible via public or private networks
- Admin functionality accessible via RESTful API

OpenStack Implementation

- Consists of multiple projects, each responsible for a subset of Cloud Computing functionality, including:

Project Name	Purpose
Nova	Compute
Swift	Object storage
Cinder	Block storage
Neutron	Networking
Horizon	Dashboard
Keystone	Identity service
Glance	Image service
Heat	Orchestration

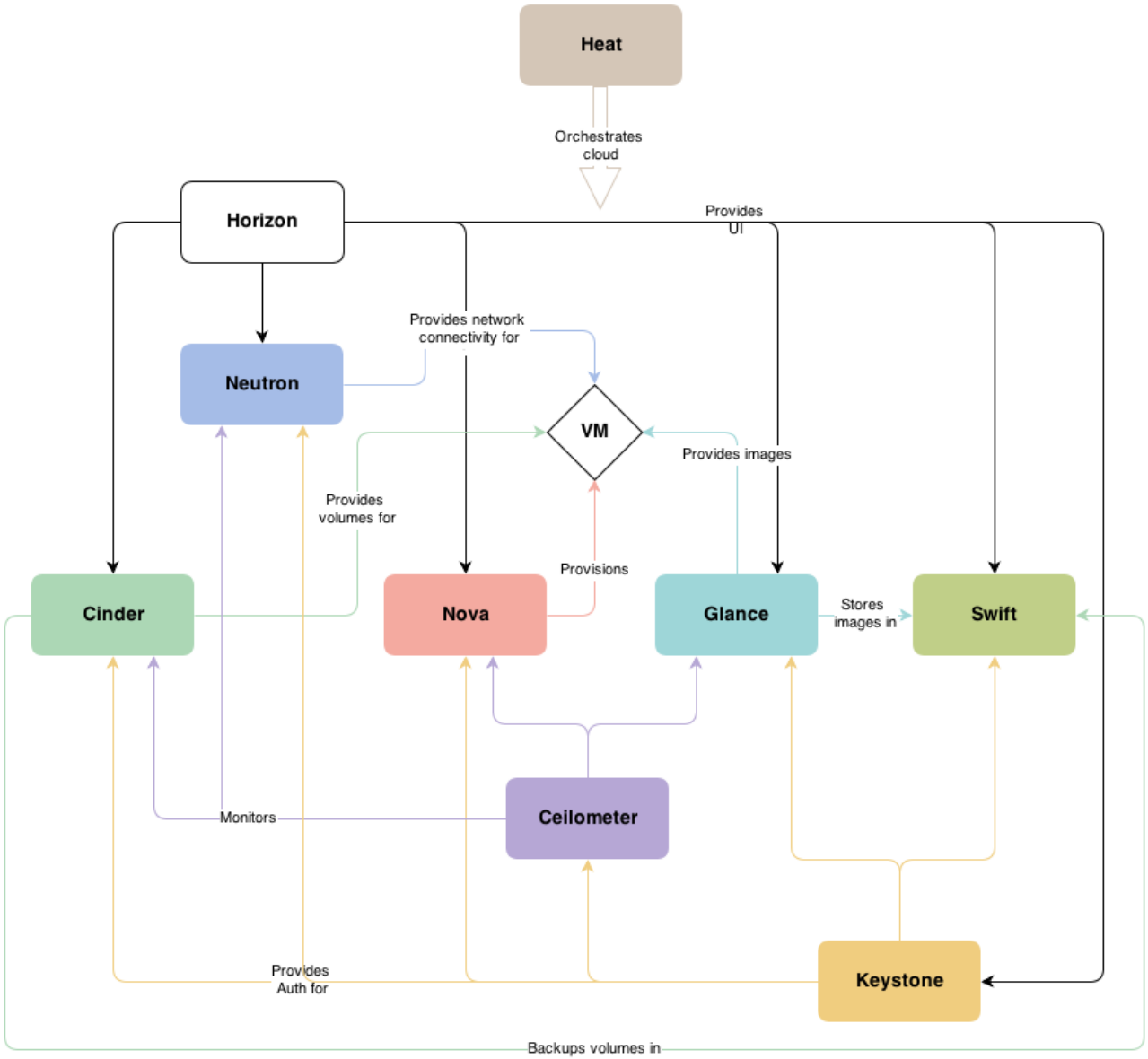
Project Name	Purpose
Murano	Application catalog
Ceilometer	Monitoring
Trove	Database service
Ironic	Hardware provisioning
Zaqar	Messaging
Barbican	Secure storage
Designate	DNS services
Manila	Shared file systems

- All OpenStack projects are released together as an OpenStack release

Who implements OpenStack?

- IaaS providers
  - Install OpenStack on their infrastructure
  - Sell infrastructure services to customers (CPUs, storage devices, network devices, etc.)
- In-house
  - Install OpenStack on own infrastructure
  - Use OpenStack API to allocate resources according to organizational needs

OpenStack: Conceptual Architecture



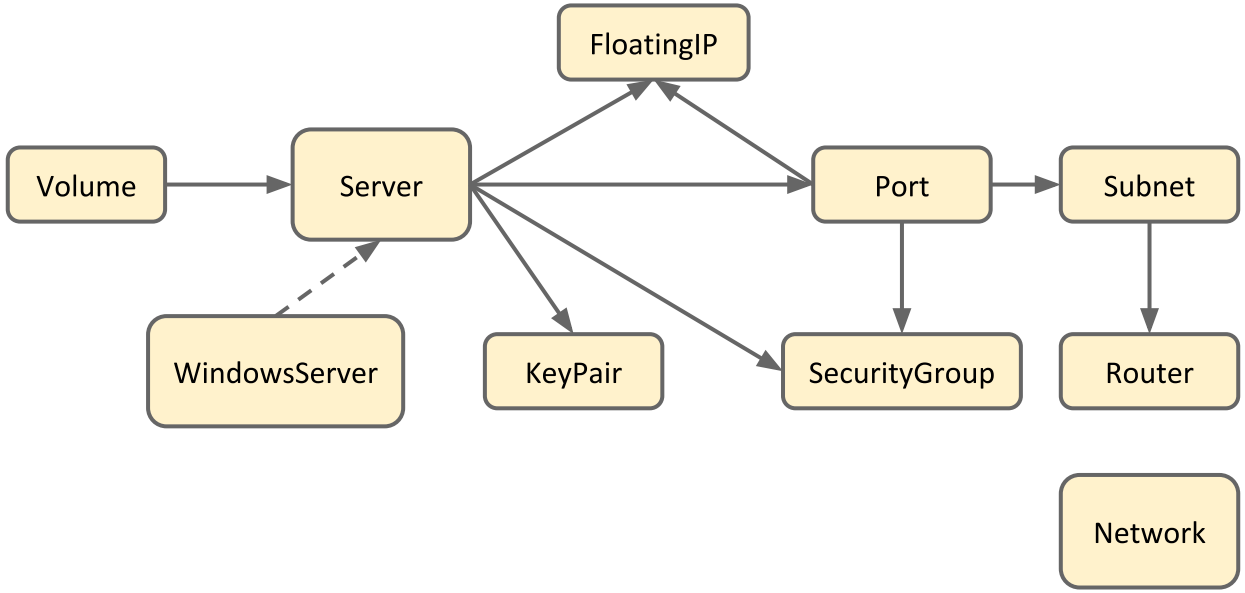
The OpenStack Plugin

- Integration with OpenStack is provided through the OpenStack plugin:

```
imports:
  - http://www.getcloudify.org/spec/openstack-plugin/1.2/plugin.yaml
```

- Defines types in terms of OpenStack elements
  - Types derived from Cloudify built-in types (from **`types.yaml`**)
  - Neutron used for networking
- Defines relationship types for OpenStack-specialized implementations

OpenStack Node & Relationship Types



- Solid arrows denote **`connected_to`** relationships
- Dashed arrows denote type extension
- Type names are prefixed with **`cloudify.openstack.nodes`** in the OpenStack Plugin's **`plugin.yaml`**

The OpenStack Plugin: Configuration

- Communication with OpenStack requires this configuration:
  - API Endpoints
  - Credentials
- Provided as either:
  - System environment variables
  - JSON file, with location specified as either:
    - Default location: **`~/openstack_config.json`**
    - **`OPENSTACK_CONFIG_PATH`** environment variable
    - **`openstack_config`** node property

The OpenStack Plugin: Configuration (cont'd)

JSON file example:

```
{
    "username": "os-user",
    "password": "os-password",
    "tenant_name": "tenant1",
    "auth_url": "http://auth-url",
    "region": "region-1",
    "nova_url": "",
    "neutron_url": ""
}
```

Only override if necessary (URLs are usually taken from Keystone)

OpenStack Types: Commons

All OpenStack types exhibit certain common traits:

- Common properties:
  - **`use_external_resource`**
  - **`resource_id`**
  - **`openstack_config`**
- Runtime properties are defined for the node upon creation:
  - **`external_id`**: the OpenStack ID of the resource
  - **`external_type`**: the OpenStack type of the resource
  - **`external_name`**: the OpenStack name of the resource

Using External Resources

All types support the **`use_external_resource`** property

- Defaults to **`false`**, meaning that no attempt is made to use existing resources
  - Quota validation takes place
- If **`true`**, then the resource *must* exist (validation takes place; no creation is attempted)

```
node_templates:
  app_subnet:
    type: cloudify.openstack.nodes.Subnet
    properties:
      use_external_resource: true
```

- External resources are supported regardless of whether they were created by Cloudify

OpenStack Resource ID

- The **`resource_id`** property determines:
  - If **`use_external_resource`** is **`true`**: the ID of the resource to use
  - If **`use_external_resource`** is **`false`**: the ID to assign to the newly-created resource

```
node_templates:
  app_subnet:
    type: cloudify.openstack.nodes.Subnet
    properties:
      use_external_resource: true
      resource_id: my_existing_subnet
```

Overriding OpenStack Configuration

- Each type supports the **`openstack_config`** property
  - Defaults to an empty dictionary
- Can be used to override certain (or all) OpenStack configuration keys, if needed
  - Keys in the dictionary should correspond to the keys used in the **`openstack_config.json`** file

```
node_templates:
  app_subnet:
    type: cloudify.openstack.nodes.Subnet
    properties:
      openstack_config:
        auth_url: http://some-other-url
```

OpenStack Types: Server

- Derived from **`cloudify.nodes.Compute`**
- Main properties:
  - **`image`**: the ID of the image to use to create the server
  - **`flavor`**: the ID of the flavor to use to create the server (specifies CPU count, RAM, etc.)
- Use **`cloudify.nodes.openstack.WindowsServer`** for Windows-based servers

```
node_templates:
  main_server:
    type: cloudify.openstack.nodes.Server
    properties:
      image: 5f3c18e7-d0eb-4440-ac37-0cf6d6d2b687
      flavor: 102
  windows_server:
    type: cloudify.openstack.nodes.WindowsServer
    properties:
      image: ...
      flavor: ...
```

OpenStack Types: Key Pair

- Derived from **`cloudify.nodes.Root`**
- Main properties:
  - **`private_key_path`**: the location where the private key is located, or should be placed after creation (depending on the value of **`use_external_resource`**)
  - **`keypair`**: a dict specifying parameters for key creation, if no existing key was specified
    - Structure documented in OpenStack's API

```
my_keypair:
  type: cloudify.openstack.nodes.KeyPair
  properties:
    private_key_path: /home/user/private.pem
```

OpenStack Types: Subnet

- Derived from **`cloudify.nodes.Subnet`**
- Main properties:
  - **`subnet`**: a dict, structured according to the OpenStack API for subnet creation

```
my_subnet:
  type: cloudify.openstack.nodes.Subnet
  properties:
    subnet:
      ip_version: 4
      cidr: 10.67.79.0/24
```

OpenStack Types: Security Group

- Derived from **`cloudify.nodes.SecurityGroup`**
- Main properties:
  - **`security_group`**: a dict, structured according to the OpenStack API for security group creation

```
my_security_group:
  type: cloudify.openstack.nodes.SecurityGroup
  properties:
    security_group:
      name: security_group_name
      description: Some test security group
```

  - **`rules`**: list of rules, structed according to the OpenStack API for rule creation

OpenStack Types: Router

- Derived from **`cloudify.nodes.Router`**
- Properties:
  - **`router`**: a dict, structured according to the OpenStack API for router creation
  - **`external_network`**: the external network to connect this router to

```
my_router:
  type: cloudify.openstack.nodes.Router
  properties:
    router:
      name: my_router
    external_network: org_external_network
```

OpenStack Types: Port

- Derived from **`cloudify.nodes.Root`**
- Properties:
  - **`port`**: a dict, structured according to the OpenStack API for port creation
    - **`network_id`** must be omitted; use a relationship to a network node instead
  - **`fixed_ip`**: the IP to request; must be available, otherwise an error is raised

```
my_port:
  type: cloudify.openstack.nodes.Port
  properties:
    port:
      name: my_port
      admin_state_up: true
  relationships:
    - target: my_network
      type: cloudify.relationships.depends_on
```

OpenStack Types: Network

- Derived from **`cloudify.nodes.Network`**
- Properties:
  - **`network`**: a dict, structured according to the OpenStack API for network creation

```
my_network:
  type: cloudify.openstack.nodes.Network
  properties:
    network:
      name: my_network
      admin_state_up: true
```

OpenStack Types: Floating IP

- Derived from **`cloudify.nodes.Root`**
- Properties:
  - **`floatingip`**: a dict, structured according to the OpenStack API for floating IP creation
    - **`floating_ip_address`** can be used to specify an existing floating IP address
    - **`floating_network_name`** resolves a network ID by its name

```
my_floating_ip:
  type: cloudify.openstack.nodes.FloatingIP
  properties:
    floatingip:
      floating_network_name: my_network
```

OpenStack Types: Volume

- Derived from **`cloudify.nodes.Volume`**
- Properties:
  - **`volume`**: a dict, structured according to the OpenStack API for volume creation
  - **`device_name`**: the device name to attach this volume to
    - Defaults to **`auto`** (recommended)
    - Explicit device name may be ignored or replaced by OpenStack

```
my_volume:
  type: cloudify.openstack.nodes.Volume
  properties:
    volume:
      size: 20
      name: data_volume
```

OpenStack Relationships

- **`cloudify.openstack.port_connected_to_security_group`**
  - Lifecycle operation: **`establish`**
  - Associates a security group with a port

```
my_port:
  type: cloudify.openstack.nodes.Port
  properties:
    port:
      name: my_port
  relationships:
    - target: test_security_group
      type: cloudify.openstack.port_connected_to_security_group
test_security_group:
  type: cloudify.openstack.nodes.SecurityGroup
  properties:
    security_group:
      name: group_1
    rules:
      port: 22
      remote_group_id: 0.0.0.0/0
```

OpenStack Relationships (cont'd)

- **`cloudify.openstack.subnet_connected_to_router`**
  - Lifecycle operations: **`establish`**, **`unlink`**
  - Associates a subnet with a router

```
my_subnet:
  type: cloudify.openstack.nodes.Subnet
  properties:
    ip_version: 4
    cidr: 10.67.79.0/24
  relationships:
    - target: my_router
      type: cloudify.openstack.subnet_connected_to_router
my_router:
  type: cloudify.openstack.nodes.Router
  properties:
    router:
      name: router_1
    external_network: org_external_network
```

OpenStack Relationships (cont'd)

- **`cloudify.openstack.server_connected_to_floating_ip`**
  - Lifecycle operations: **`establish`**, **`unlink`**
  - Input for "establish"
    - fixed_ip: a specific fixed IP of the server, to attach to the floating IP
  - Associates a server with a floating IP

```
my_server:
  type: cloudify.openstack.nodes.Server
  ..
  relationships:
    - target: my_floating_ip
      type: cloudify.openstack.server_connected_to_floating_ip

my_floating_ip:
  type: cloudify.openstack.nodes.FloatingIP
  ..
```

OpenStack Relationships (cont'd)

- **`cloudify.openstack.server_connected_to_security_group`**
  - Lifecycle operations: **`establish`**, **`unlink`**
  - Associates a server with a security group

```
my_server:
  type: cloudify.openstack.nodes.Server
  ..
  relationships:
    - target: my_security_group
      type: cloudify.openstack.server_connected_to_security_group

my_security_group:
  type: cloudify.openstack.nodes.SecurityGroup
  ..
```

OpenStack Relationships (cont'd)

- **`cloudify.openstack.volume_attached_to_server`**
  - Lifecycle operations: **`establish`**, **`unlink`**
  - Associates a volume with a server

```
my_server:
  type: cloudify.openstack.nodes.Server
  ..

my_volume:
  type: cloudify.openstack.nodes.Volume
  ..
  relationships:
    - target: my_server
      type: cloudify.openstack.volume_attached_to_server
```

OpenStack Relationships (cont'd)

- **`cloudify.openstack.server_connected_to_port`**
  - Associates a server with a port
  - Implements no lifecycle operations; relationship is used for Cloudify to automatically connect the server to the port upon creation

```
my_server:
  type: cloudify.openstack.nodes.Server
  ..
  relationships:
    - target: my_port
      type: cloudify.openstack.server_connected_to_port

my_port:
  type: cloudify.openstack.nodes.Port
  ..
```

OpenStack Relationships (cont'd)

- **`cloudify.openstack.port_connected_to_subnet`**
  - Associates a port with a subnet
  - Useful when a network has multiple subnets, and a port should be associated with specific subnet(s)
  - Implements no lifecycle operations; relationship is used for Cloudify to automatically connect the port to the subnet upon creation

```
my_port:
  type: cloudify.openstack.nodes.Port
  ..
  relationships:
    - target: my_subnet
      type: cloudify.openstack.port_connected_to_subnet

my_subnet:
  type: cloudify.openstack.nodes.Subnet
  ..
```

OpenStack Relationships (cont'd)

- **`cloudify.openstack.port_connected_to_floating_ip`**
  - Lifecycle operations: **`establish`**, **`unlink`**
  - Associates a port with a floating IP

```
my_port:
  type: cloudify.openstack.nodes.Port
  ..
  relationships:
    - target: my_ip
      type: cloudify.openstack.port_connected_to_floating_ip

my_floating_ip:
  type: cloudify.openstack.nodes.FloatingIP
  ..
```

OpenStack Types: Security Group Rules Sugarings

- **`port`**: can be used to specify a single port

```
my_security_group:
  type: cloudify.openstack.nodes.SecurityGroup
  properties:
    security_group:
      name: my_security_group_name
      description: My security group's description
    rules:
      # These two definitions are equivalent
      - port: 22
        remote_ip_prefix: 0.0.0.0/0
      - port_range_min: 22
        port_range_max: 22
        remote_ip_prefix: 0.0.0.0/0
```

OpenStack Types: Security Group Rules Sugaring (cont'd)

- **`remote_group_node`**
  - The value corresponds to another security group node in the blueprint
  - Automatically populates **`remote_group_id`** with the ID of another security group
  - A dependency must be defined

```
my_group:
  type: cloudify.openstack.nodes.SecurityGroup
  properties:
    security_group:
      rules:
        - remote_group_node: my_other_group
  relationships:
    - target: my_other_group
      type: cloudify.relationships.depends_on
      ..
      ..
my_other_group:
  type: cloudify.openstack.nodes.SecurityGroup
  ..
```

OpenStack Types: Security Group Rules Sugaring (cont'd)

- **`remote_group_name`**
  - Resolves the ID from the actual OpenStack name of an existing security group

```
my_group:
  type: cloudify.openstack.nodes.SecurityGroup
  properties:
    security_group:
      rules:
        - remote_group_name: existing_openstack_group
```

Bootstrapping a Manager

- Two bundled manager blueprints exist for bootstrap:
  - **`openstack`**: Uses standard, out-of-the-box OpenStack APIs
  - **`openstack-nova-net`**: Uses Nova Net (instead of Neutron) for networking
- Nova Net is deprecated in OpenStack, therefore it is not covered in the training material

Bootstrapping: The OpenStack Blueprint

- Sample blueprint provides a good starting point:

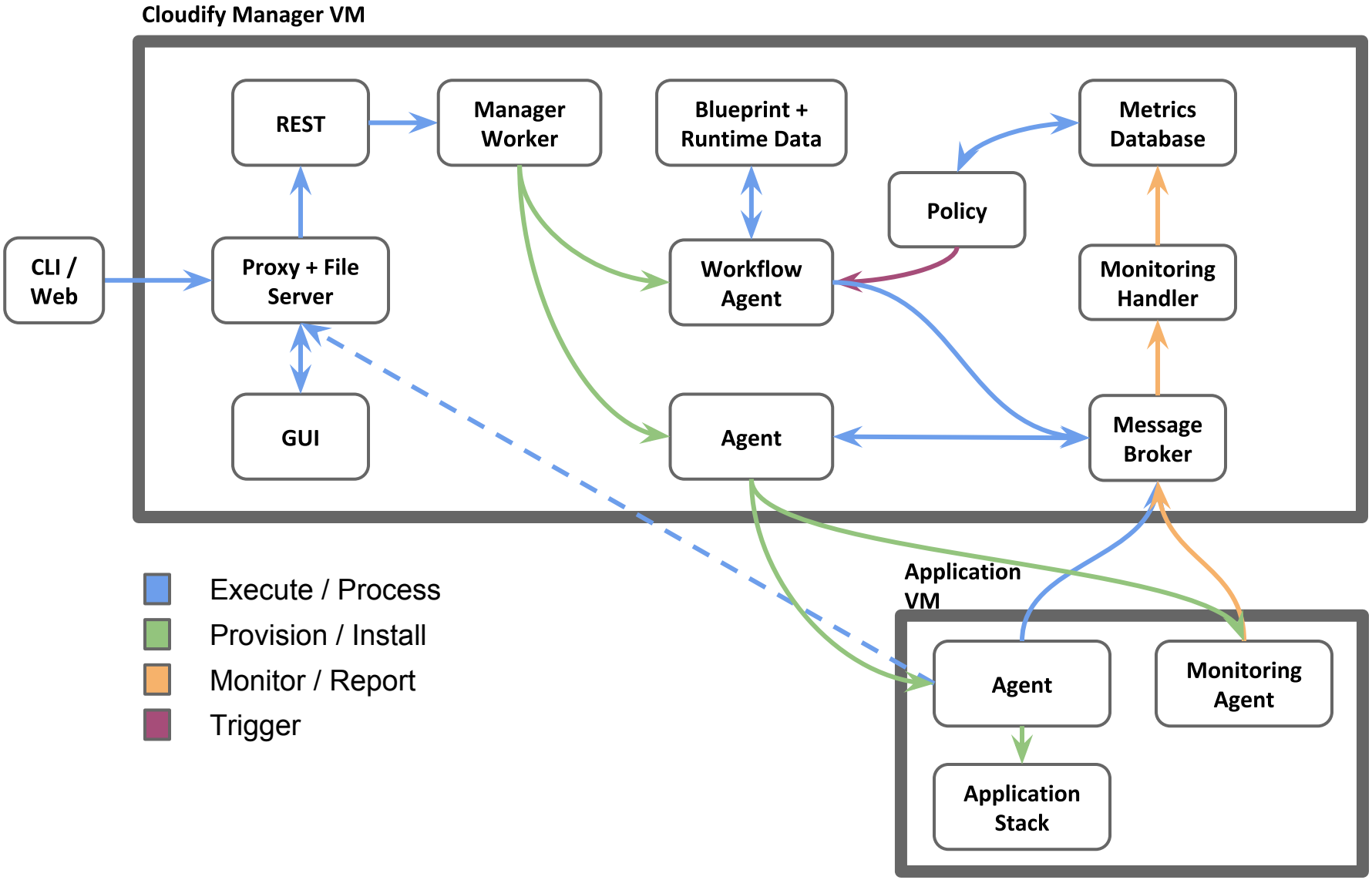
  `openstack/openstack-manager-blueprint.yaml`

- Blueprint inputs are used to customize manager topology:
  - VM provisioning parameters (image, flavor, etc.)
  - Keystone parameters
  - Networking parameters
  - Security groups, keypairs
  - etc.
- **`inputs.yaml.template`** contains a template for the blueprint's inputs

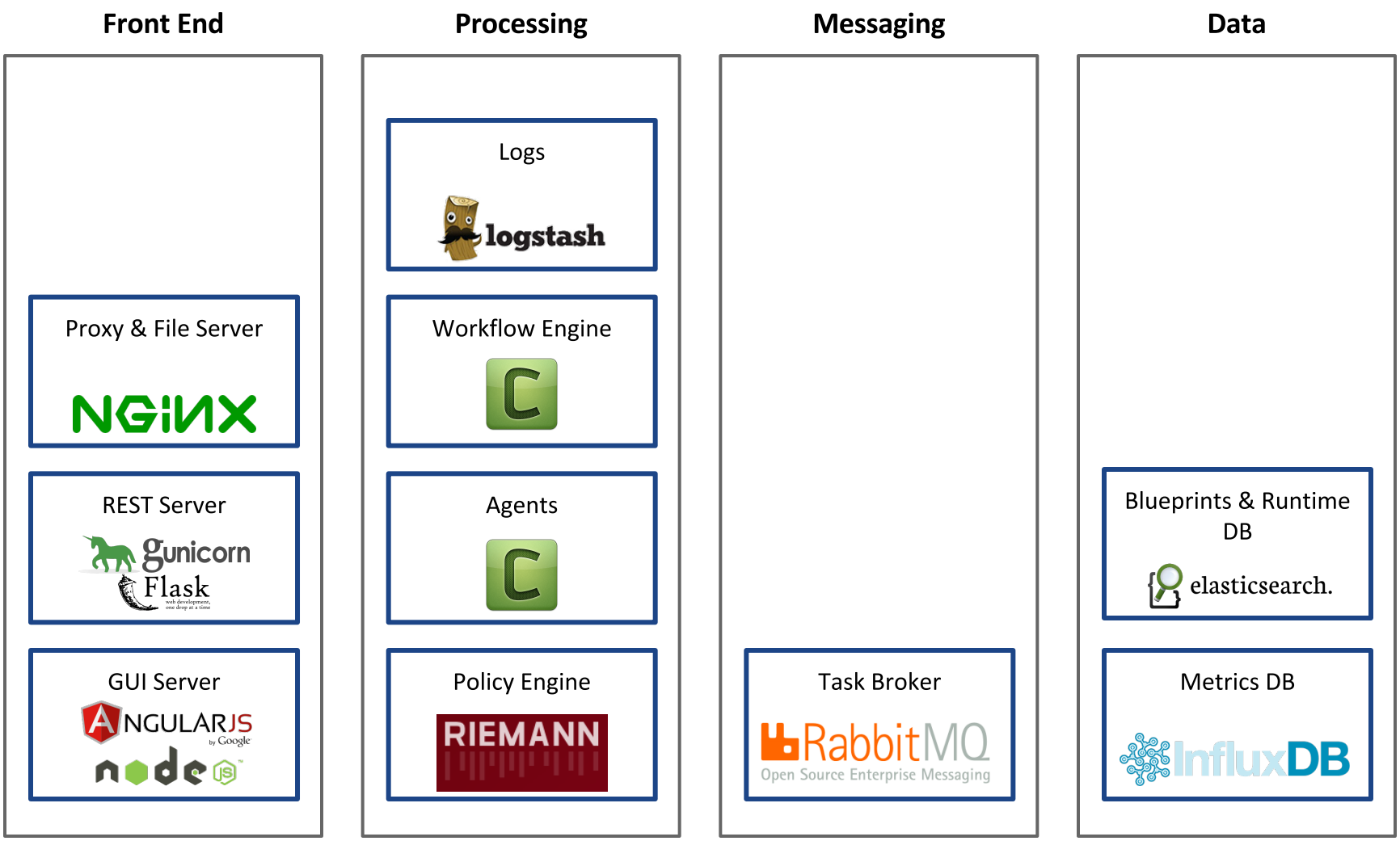
LAB 9: OPENSTACK BOOTSTRAPPING AND DEPLOYMENT

CLOUDIFY MANAGER ARCHITECTURE

Architecture: Logical View



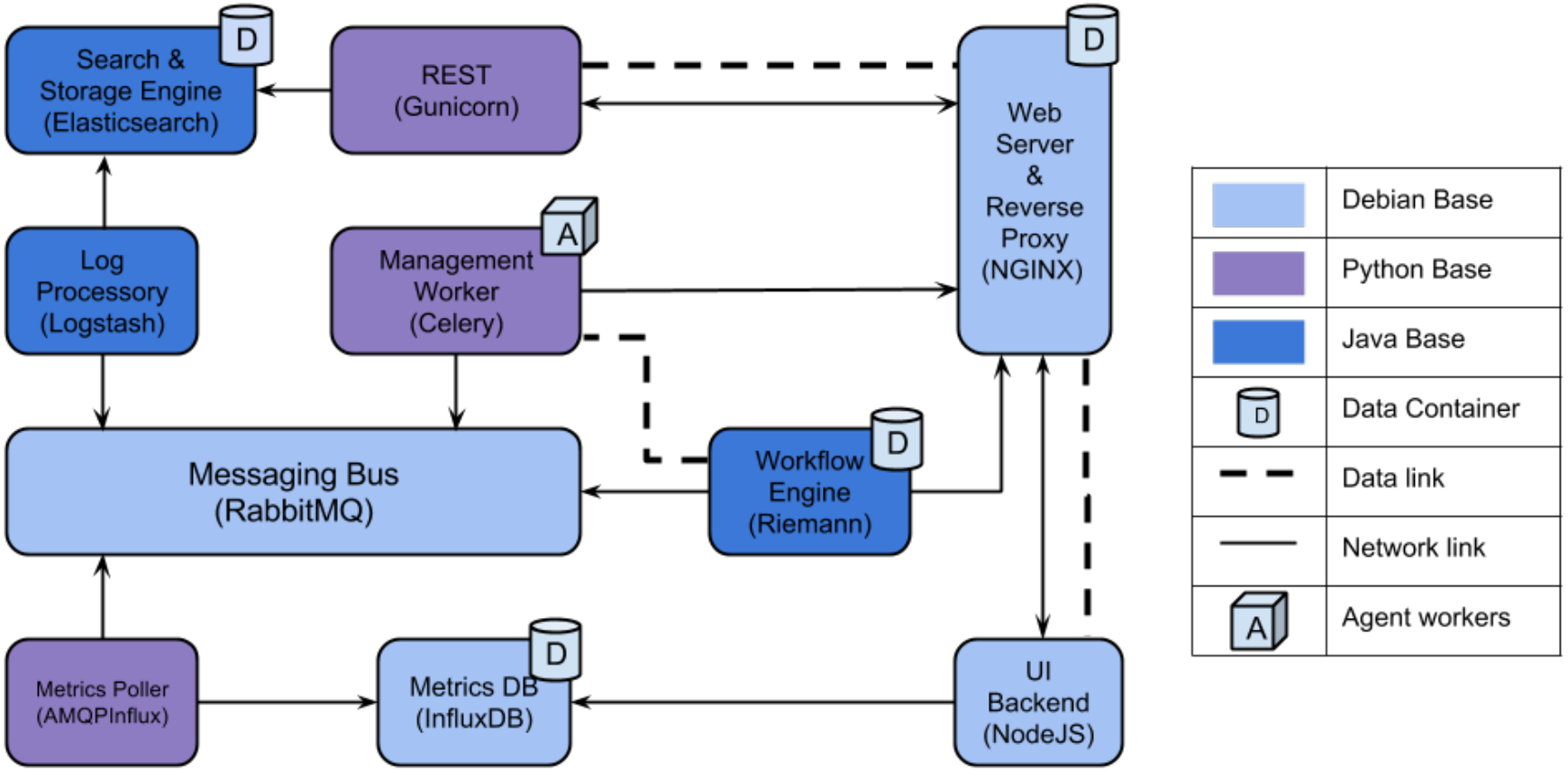
Architecture: Layered View



Cloudify Manager: Docker View

- Docker: provides a cross-platform mechanism for packaging and running software stacks
  - For Linux, Windows and others
  - Will be covered in-depth later
- Provided manager blueprints are Docker-based
- Cloudify may attempt to install Docker if it is not already installed

Cloudify Manager: Docker View (cont'd)

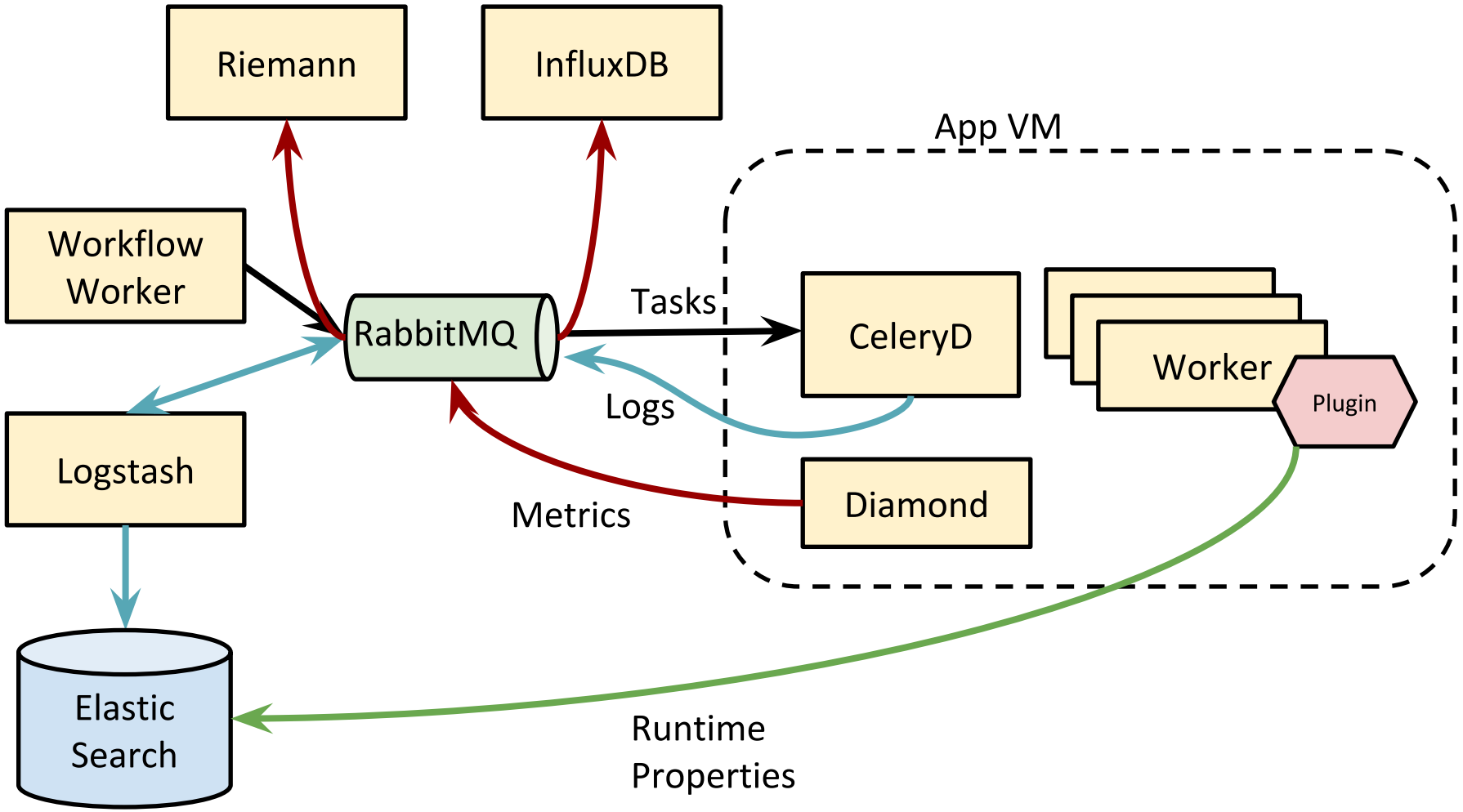


ADMINISTRATION TASKS

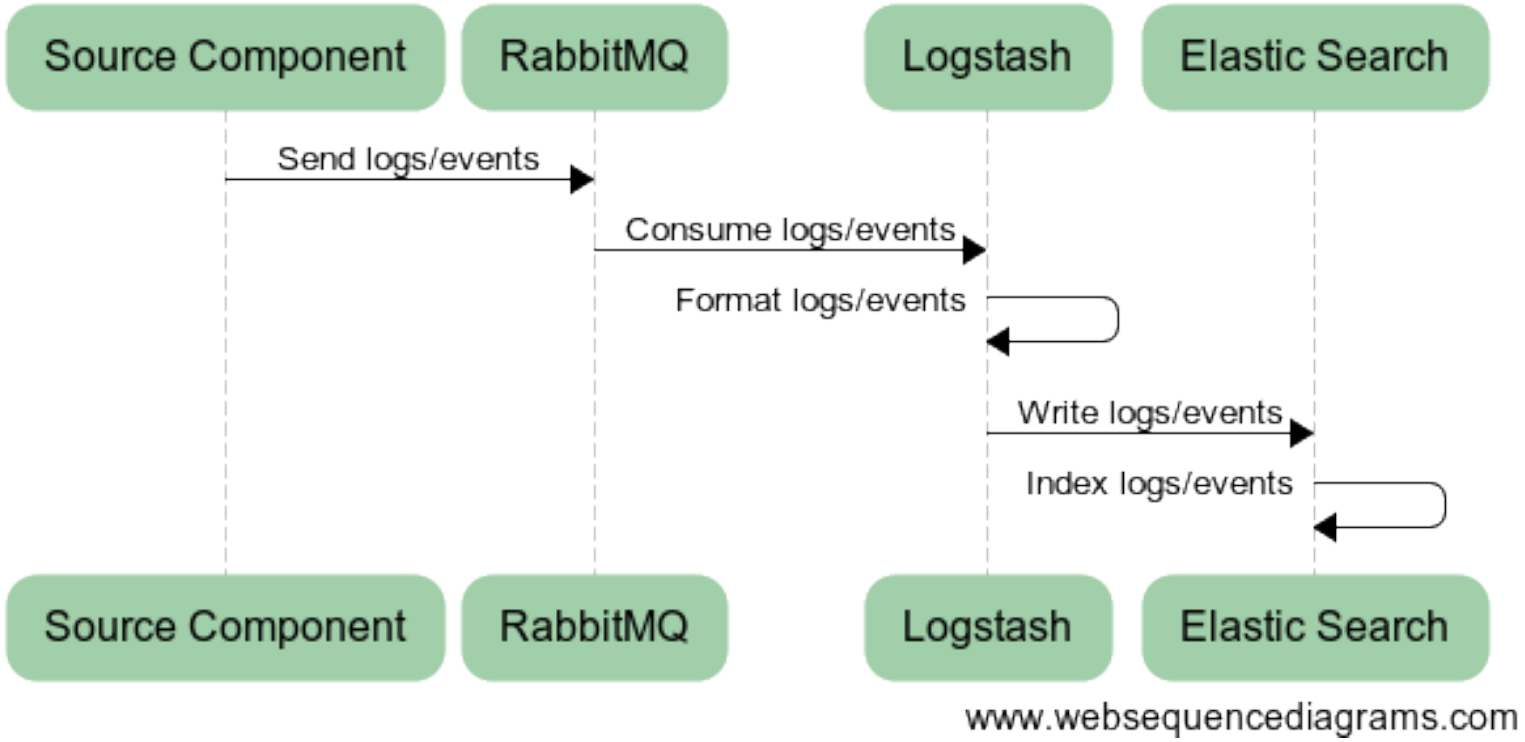
Logs & Events Functionality

- Cloudify components' logs are gathered, indexed and persisted
- Events provide the user with a simple and clear way to trace the progress of a workflow execution
- Available through API, CLI and web GUI

Logs & Events Mechanism



Logs & Events Mechanism (cont'd)

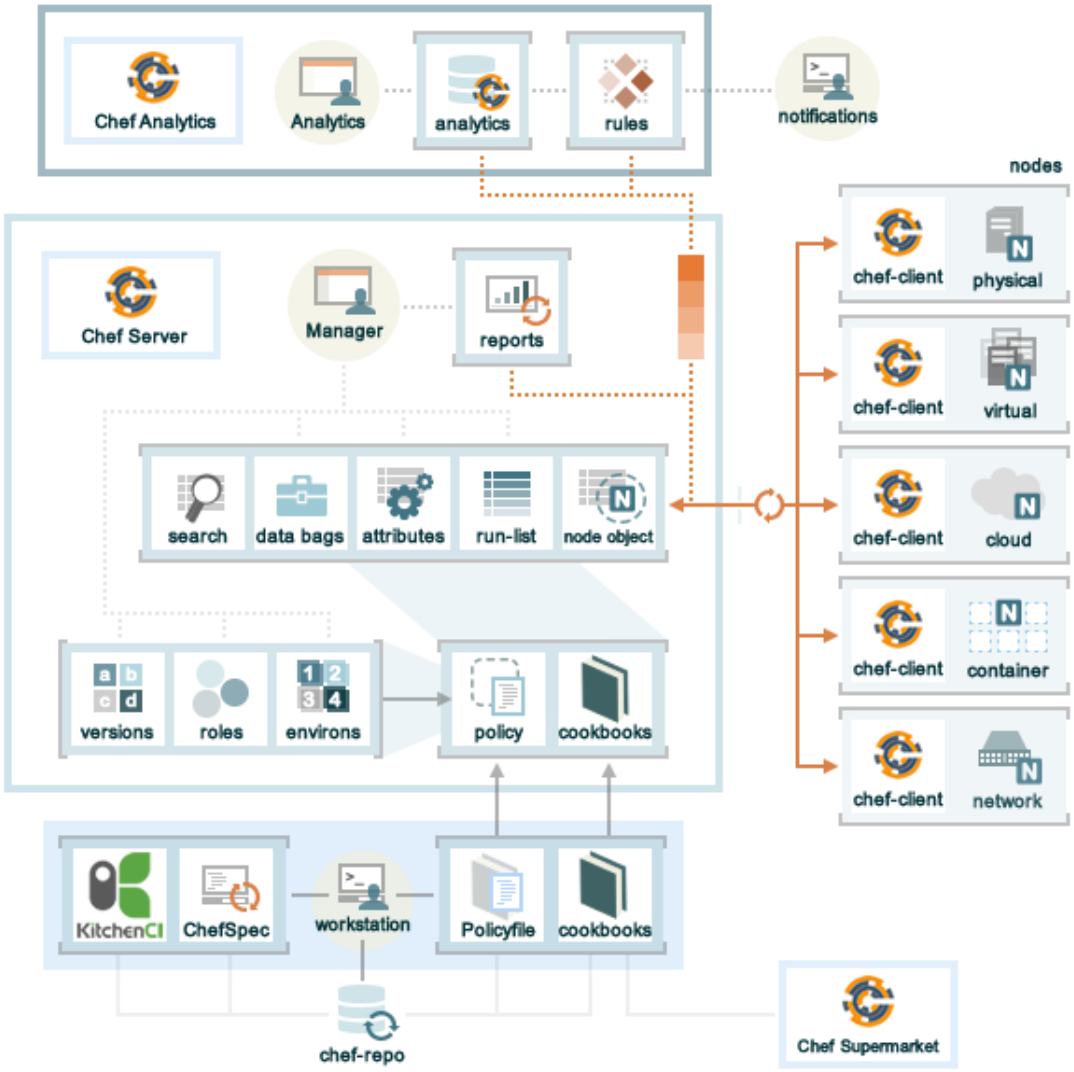


MORE OFFICIAL PLUGINS

Chef: Introduction

- Configuration management software, owned by Chef Software
- System configuration defined in “recipes”, using a Ruby-based DSL
  - Defines products to be installed, files to be created, other configuration adjustments to be made
  - Procedural in nature
- Recipes are uploaded to and managed on a Chef Server
- Chef Clients are installed on nodes, and communicate with Chef Server
  - Chef Server ensures that managed nodes are in line with prescribed configuration

Chef: Architecture



Chef Plugin

- Defines types derived from Cloudify's built-in types
  - **`cloudify.chef.nodes.SoftwareComponent`**
  - **`cloudify.chef.nodes.ApplicationServer`**
  - **`cloudify.chef.nodes.DBMS`**
  - **`cloudify.chef.nodes.ApplicationModule`**
  - **`cloudify.chef.nodes.WebServer`**
- Supports both Chef-Client and Chef-Solo

Chef Plugin (cont'd)

- Added property (**`chef_config`**) containing Chef's configuration for the node
- Method of operation (Chef-Client vs. Chef-Solo) is determined by which set of keys is defined under **`chef_config`**
- Either the Chef-Client set of keys, or the Chef-Solo set of keys, must be present; never both

Chef Plugin (cont'd)

- Keys for Chef-Client:

`chef_server_url`	URL of the Chef server
`environment`	Chef environment to use
`node_name_prefix`	Prefix to apply to the names of created nodes
`node_name_suffix`	Suffix to apply to the names of created nodes
`validation_client_name`	Client name to use for validation against the Chef server
`validation_key`	The key file to use for validation (actual contents — not path)
`version`	Chef version to install

Chef Plugin (cont'd)

- Keys for Chef-Solo:

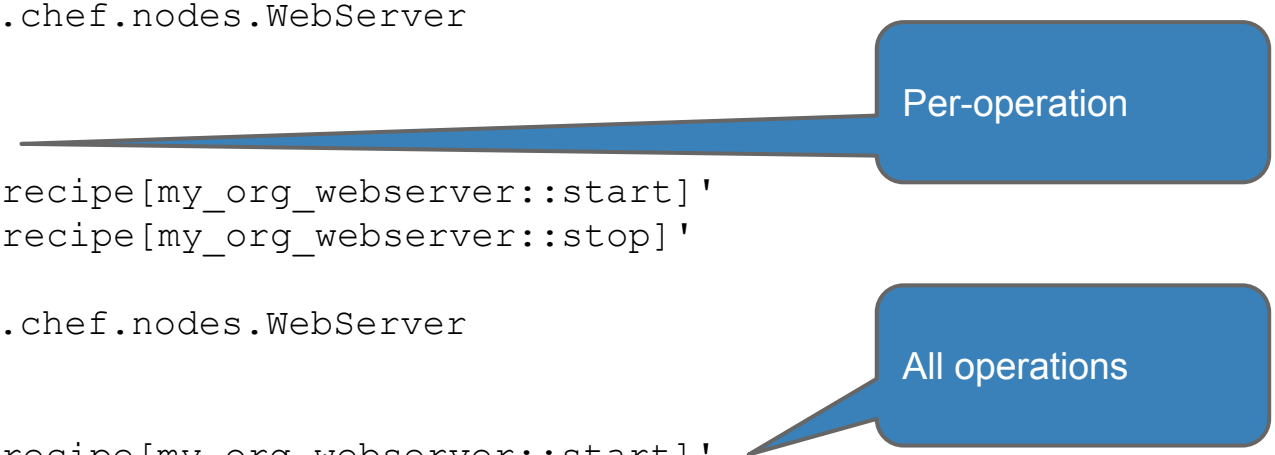
`cookbooks`	URL, or blueprint-relative path, to a **`.tar.gz`** file containing cookbooks
`data_bags`	URL, or blueprint-relative path, to a **`.tar.gz`** file containing data-bags JSON files
`environments`	URL, or blueprint-relative path, to a **`.tar.gz`** file containing environments' JSON files
`roles`	URL, or blueprint-relative path, to a **`.tar.gz`** file containing roles' JSON files
`version`	Chef version to install

Chef Plugin: Runlists

- Two methods of specifying Chef runlists:
  - Using the **`runlists`** parameter
  - Using the **`runlist`** parameter
  - If both specified: **`runlist`** takes precedence
- Using **`runlists`**
  - Runlists are specified on a per-lifecycle operation basis
    - Sugaring: operation name can be specified by its last component
  - Operations with no runlists: Chef skipped
- Using **`runlist`**
  - Used for all lifecycle operations

Chef Plugin: Runlists (cont'd)

```
imports:
  - http://getcloudify.org/spec/chef-plugin/1.1/plugin.yaml
node_templates:
  example_node_1:
    type: cloudify.chef.nodes.WebServer
    properties:
      chef_config:
        runlists:
          start: 'recipe[my_org_webserver::start]'
          stop:  'recipe[my_org_webserver::stop]'
  example_node_2:
    type: cloudify.chef.nodes.WebServer
    properties:
      chef_config:
        runlist: 'recipe[my_org_webserver::start]'
```



Chef Plugin: Version

- The **`version`** parameter is required for each Chef node
- All Chef nodes, contained within a particular server, must use the same value for the **`version`** parameter
- Best practice: use as blueprint input

```
node_templates:
  example_web_server:
    type: cloudify.chef.nodes.WebServer
    properties:
      chef_config:
        version: 11.10.4-1
```

Chef Plugin: Attributes

- **`attributes`** key under **`chef_config`**
  - Dictionary, passed as-is to Chef
  - Must not contain a key named **`cloudify`**, as it is reserved (see below)

Blueprint

```
node_templates:
  example_web_server:
    type: cloudify.chef.nodes.WebServer
    properties:
      chef_config:
        attributes:
          attr_1: value1
          attr_2: value2
```

Chef

```
node['attr_1'] == 'value1'
node['attr_2'] == 'value2'
```

Chef Plugin: Attributes (cont'd)

- Within Chef: **`node['cloudify']`** contains certain Cloudify-provided keys
  - **`node_id`**: current node ID
  - **`blueprint_id`**: current blueprint ID
  - **`deployment_id`**: current deployment ID
  - **`properties`**: current node's properties
  - **`runtime_properties`**: current node's runtime properties
  - **`capabilities`**: mapping of runtime properties of related nodes

```
node['cloudify']['node_id']
node['cloudify']['blueprint_id']
node['cloudify']['deployment_id']
node['cloudify']['properties']
node['cloudify']['runtime_properties']
node['cloudify']['capabilities']
```

Chef Plugin: Attributes (cont'd)

- Operations that involve two nodes: dictionary **`node['cloudify']['related']`** contains information about the related node
  - **`node_id`**: the related node's ID
  - **`properties`**: the related node's properties
  - **`runtime_properties`**: the related node's runtime properties

Chef Plugin: Attributes (cont'd)

Blueprint

```
node_templates:
  example_web_server:
    type: cloudify.chef.nodes.WebServer
    properties:
      prop1: value1
    relationships:
      - type: cloudify.relationships.connected_to
        target: example_db_server
  example_db_server:
    type: cloudify.chef.nodes.DBMS
      properties:
        prop2: value2
```

(Assume **db_node_id** is the node ID of **example_db_server**)

Chef

```
node['cloudify']['properties']['prop1'] == 'value1'
node['cloudify']['related']['properties']['prop2'] == 'value2'
node['cloudify']['related']['node_id'] = 'db_node_id'
```

Chef Attributes as Runtime Properties

At the end of a Chef run, Chef's attributes are stored as runtime attributes with the node that caused the Chef run.

Chef

```
attr_1 == value_1
attr_2 == value_2
```

Cloudify

```
ctx.instance['runtime_properties']['attr_1'] == 'value_1'
ctx.instance['runtime_properties']['attr_2'] == 'value_2'
```

Referring to Other Node's Attributes

Within a blueprint, it is possible to refer to Chef attributes and Chef runtime properties of a connected node.

```
node_templates:
  example_web_server:
    type: cloudify.chef.nodes.WebServer
    properties:
      chef_config:
        attributes:
          db:
            port: {related_chef_attribute: db_port}
            some_val: {related_runtime_property: some.other.prop}
    relationships:
      - type: cloudify.relationships.connected_to
        target: example_db_server

  example_db_server:
      type: cloudify.chef.nodes.DBMS
      properties:
        chef_config:
          attributes:
            db_port: 1521
```

LAB X1: CHEF INTEGRATION

Puppet: Introduction

- Configuration management software, owned by Puppet Labs
- System configuration defined using Puppet's proprietary language
  - Model-driven: user specifies desired state and dependencies, Puppet ensures the desired state is met
-

LAB X2: PUPPET INTEGRATION

WORKING WITH AWS

The AWS Plugin

- Integration with AWS is provided through the Cloudify official AWS plugin:

```
imports:
  - http://www.getcloudify.org/spec/aws-plugin/1.2/plugin.yaml
```

- Defines types that map to AWS elements
  - Derived from Cloudify's node types (declared in **`types.yaml`**)
- Defines relationship types for AWS-Specialized relationship implementations
- Communication with AWS performed by underlying **`boto`** API

AWS Types: Commons

- Common properties:
  - **`use_external_resource`**
  - **`resource_id`**
  - **`aws_config`**
- Upon creation of a node, the following runtime properties become available:
  - **`aws_resource_id`**: the AWS ID of the resource

The AWS Plugin: Configuration

- All AWS types support a property named **`aws_config`**
  - A dictionary of values to pass to boto's connection client
  - Reference can be found in boto's API documentation
- Common scenario for **`aws_config`** reuse:

```
node_templates:
  aws_configuration:
    derived_from: cloudify.nodes.Root
    properties:
      aws_config:
        key: value
…
node_types:
  server_ip:
    type: cloudify.aws.nodes.ElasticIP
    properties:
      aws_config: { get_property: [aws_configuration, aws_config] }
```

Configuration is defined once...

... And then reused through **`get_property`**

LAB X3: AWS BOOTSTRAPPING AND DEPLOYMENT

WORKING WITH CLOUDSTACK

The CloudStack Plugin

Similar concept as the OpenStack plugin:

- Cloudify official CloudStack plugin:

```
imports:
  - http://www.getcloudify.org/spec/cloudstack-plugin/1.2/plugin.yaml
```

- Defines types that map to CloudStack elements
- Defines relationship types for CloudStack-specialized relationship implementations

Bootstrapping a Manager

The CloudStack Plugin

- Integration with CloudStack is provided through the Cloudify official CloudStack plugin:

```
imports:
  - http://www.getcloudify.org/spec/cloudstack-plugin/1.2/plugin.yaml
```

- Similarly to the OpenStack plugin, the CloudStack plugin defines types that map to CloudStack elements
- Defines relationship types for CloudStack-specialized relationship implementations

Bootstrapping a Manager

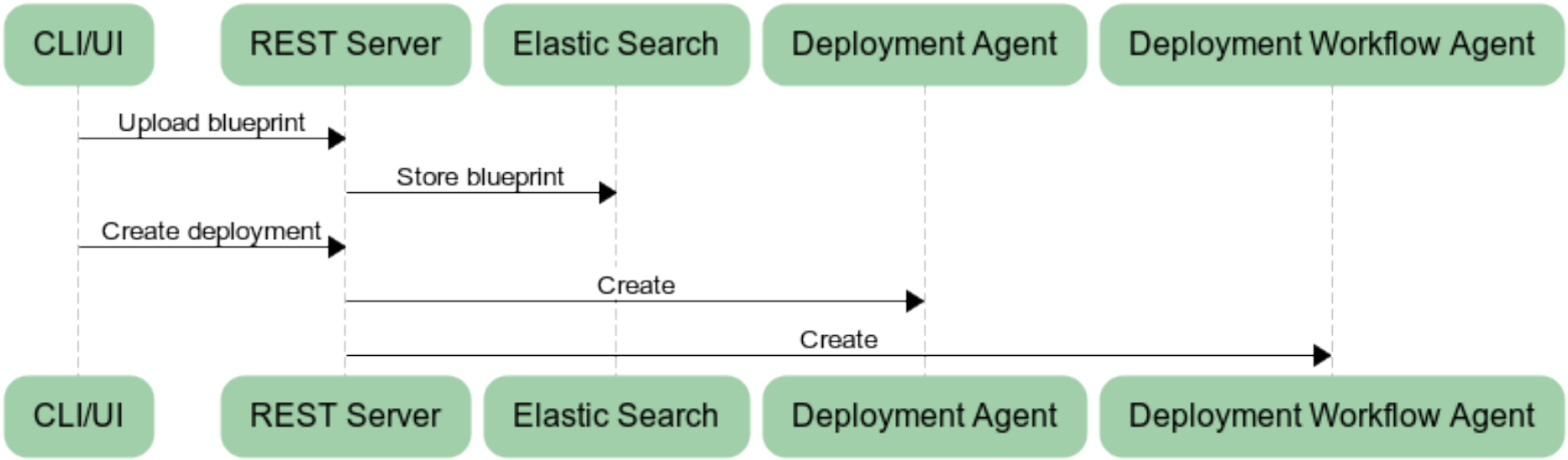
- Bundled manager blueprint for manager bootstrapping:

  https://github.com/cloudify-cosmo/cloudify-manager-blueprints

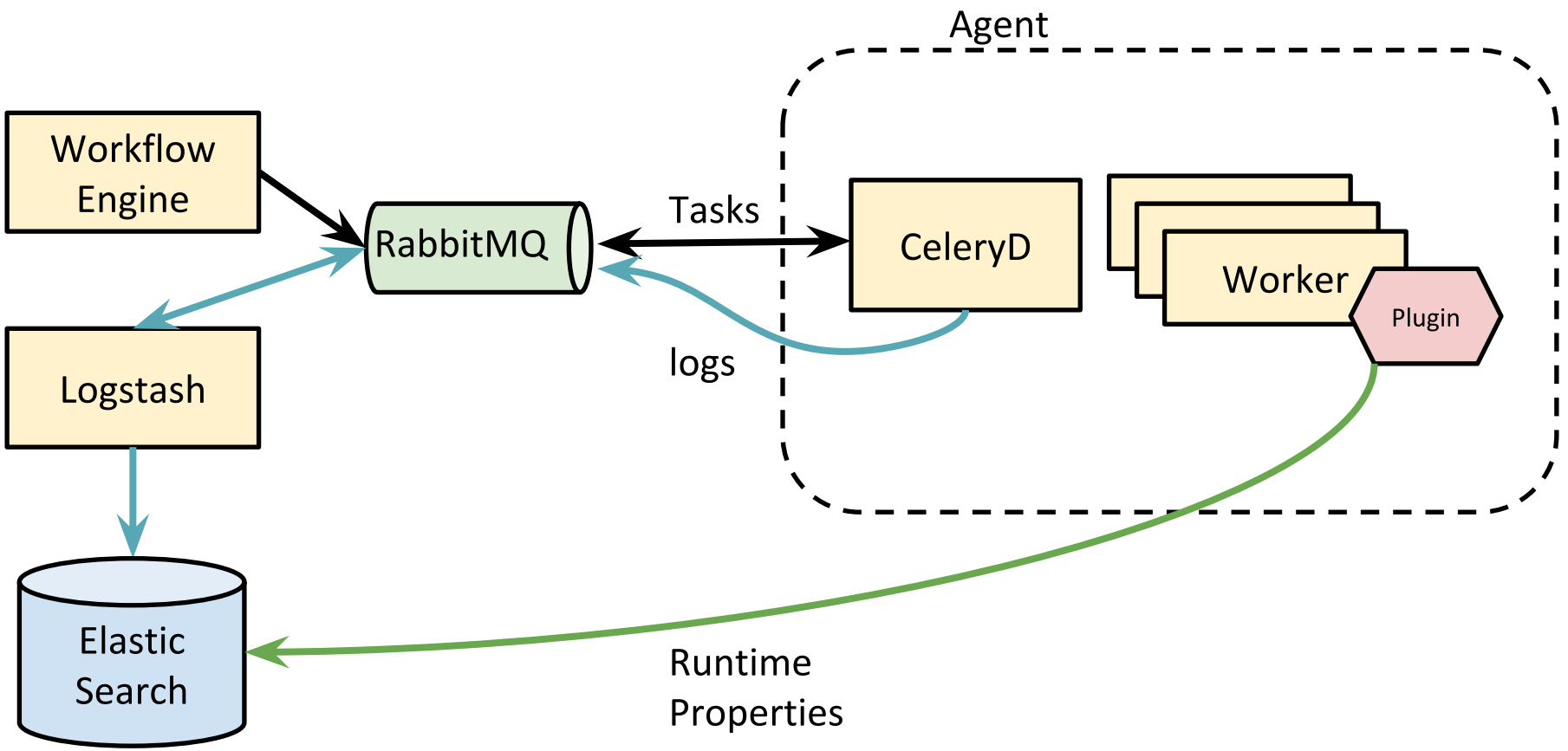
-

LAB X4: CLOUDSTACK BOOTSTRAPPING AND DEPLOYMENT

Blueprint Upload & Deployment Creation



Agent-Host Plugin Architecture



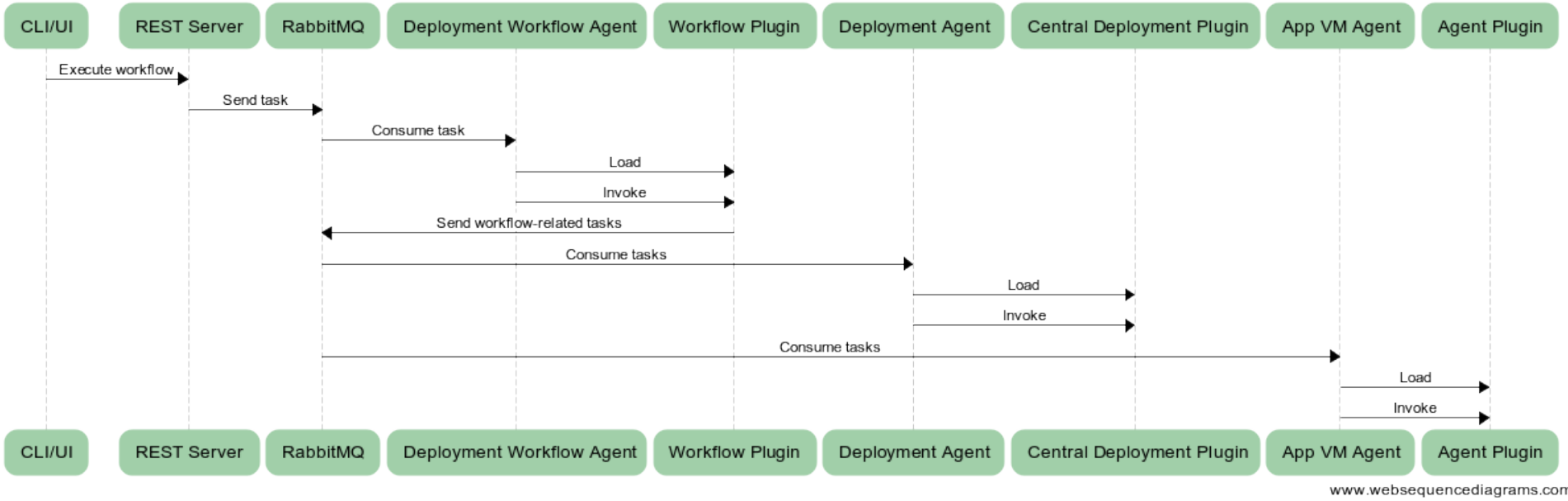
How Do Plugins Work

- Cloudify uses Celery as a task broker to execute local and remote operations
- Cloudify's workflow engine sends a task through Celery; the task passes via RabbitMQ to a Celery agent/worker which will execute the task
- The task then interacts with Cloudify via REST calls to the REST service

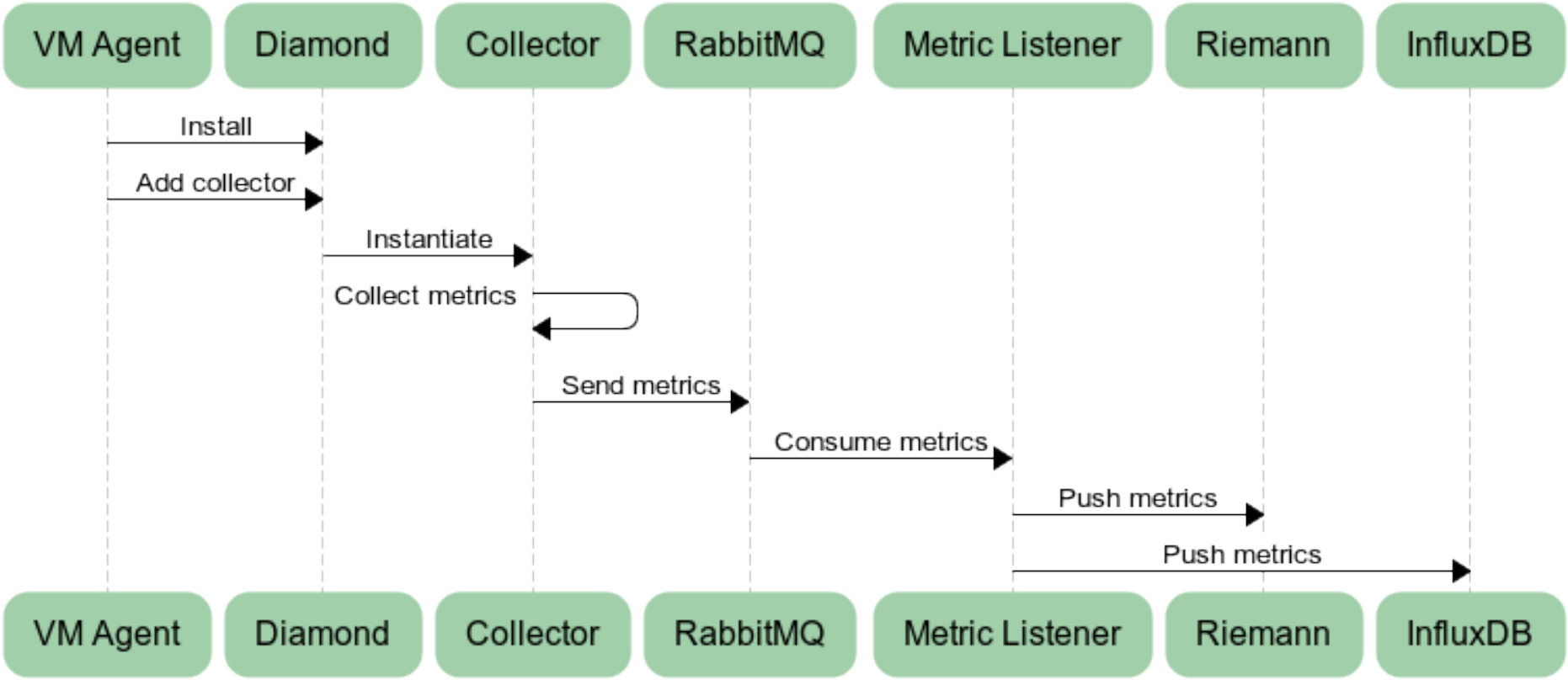
How Do Plugins Work (cont'd)

- The central-deployment agents are installed:
  - On the manager's host
  - When a deployment is created
- The host agents are installed on Compute nodes:
  - During the node's creation operation
  - Optional (default is to install)
- Plugins are installed — on central or host agents — immediately after the agent's installation (eagerly)
- A plugin may be installed on both types of agents if an operation's **`executor`** has been overridden

Workflow Execution



Monitoring (cont'd)



Zoom-In on Monitoring