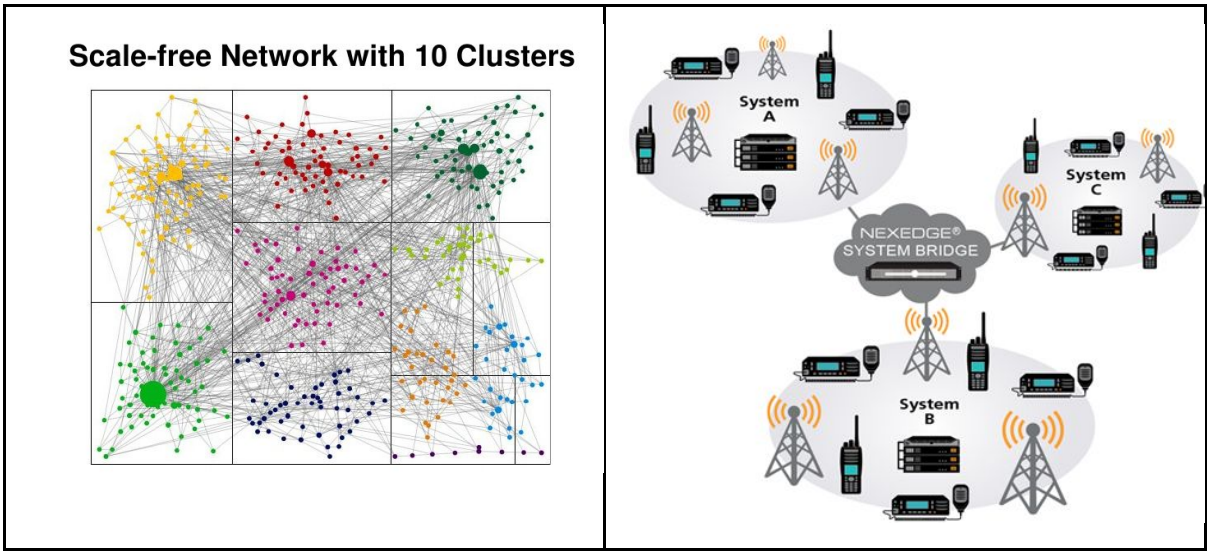
Building Large Scale Services with Microservices

Model-Driven Design and Open Orchestration with TOSCA and Cloudify

hello@cloudify.co | cloudify.co

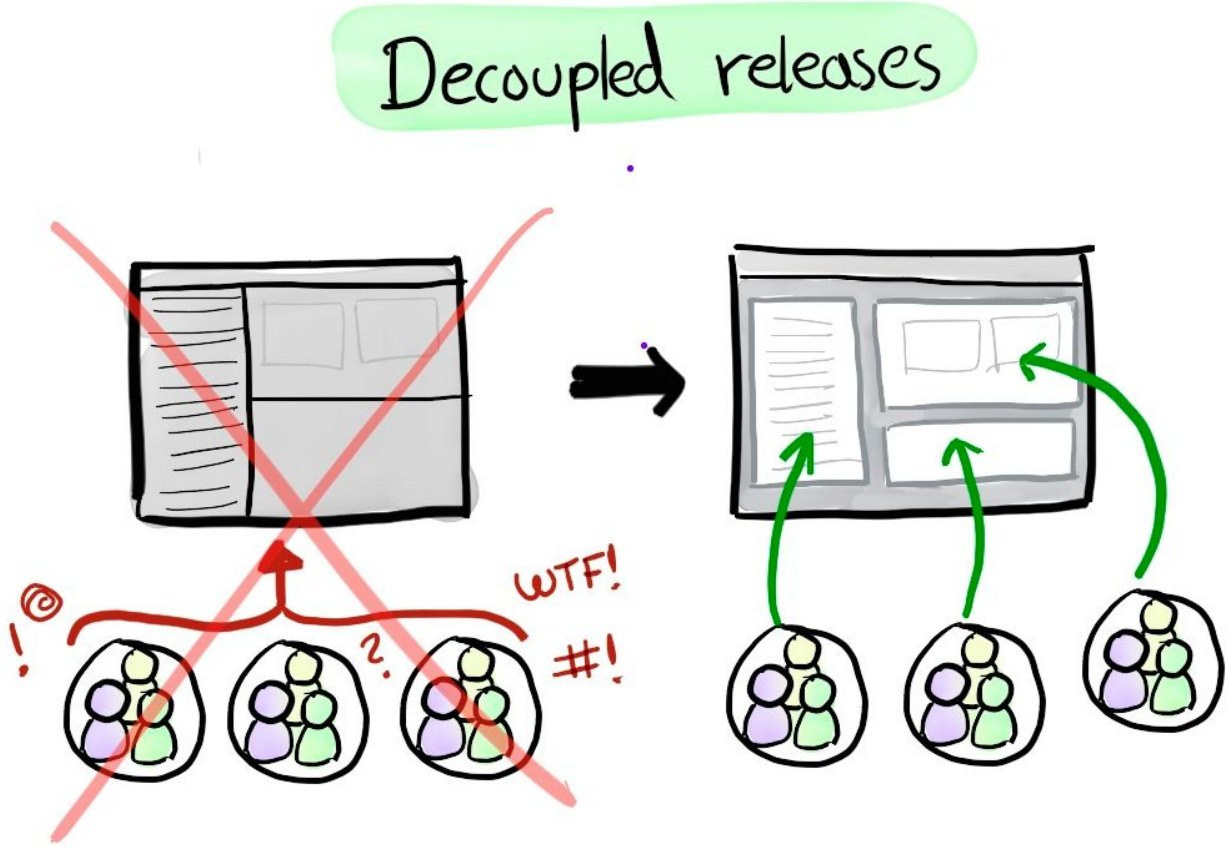
Introduction

Large scale systems used to be an edge case and were often designed as a one-off systems. Social networks, Networking services, Big Data, and IoT brought large scale systems to where they have become the norm rather than the exception. That meant we needed to come up with a more systematic model on how to design and implement large scale systems, or as they are often referred to today - web-scale.



Patterns for designing large scale systems

I first encountered the scalability challenge back in 2007, and got fully into it when it had really matured in 2010. At that time, it was the financial services industry that first picked up the scalability challenge with a new class of high frequency and algorithmic trading systems. The challenge with trading systems wasn't just about scalability - it was doing so with extremely low-latency and in a stateful transactional system. It was clear that the classic tier-based architecture, which was highly popular at the time, with J2EE platforms and later Ruby on Rails, wasn't going to fly. That led me to write one of the first alternative architecture patterns for building large scale systems - The Scalability Revolution from Dead End to Open Road which is also known as Space Based Architecture. This pattern was inspired by previous work known as Tuple Space developed originally at Yale University by Professor David Gelernter.



The Space Based approach was designed for real-time systems. Web-scale services, on the other hand, don't have the same strict requirements for latency and consistency as financial services, as they allow for more relaxation in the way we address consistency and performance. The most notable evidence of this is the use of "eventual consistency" and NOSQL databases in which we traded (no pun intended) consistency for scalability.

Those changes in assumption led to a simpler architectural pattern known today as Microservices. This architecture is described in more detail in Martin Fowler's article on the subject. See also this list of useful Microservices patterns by Chris Richardson.

This paper will focus more specifically on how we design large scale systems based on Microservices patterns using TOSCA and Cloudify.

Demystifying common Microservices misconceptions

Before continuing, let's demystify a few common misconceptions that often arise when discussing this topic.

Misconception 1: Microservices are tied to containers
Nope. Containers lend themselves well to componentized architectures and therefore lend themselves well to the Microservices architecture. In addition, Kubernetes, which was one of the first popular platforms for delivering Microservices, is based on containers. With that in mind it is only natural that people would associate microservices with containers. Having said that, it's important to note that the pattern itself, as described above, isn't specific to containers and could apply to any software architecture with or without containers.

Misconception 2: TOSCA is designed for monolithic architectures
Wrong. TOSCA is a standard application modeling language which describes an application's topology as a graph of nodes. Many of the original examples for TOSCA were modeled around tier-based applications and therefore many people associate TOSCA with monolithic architecture and not Microservices.

TOSCA is, above all, a modeling language which is used to describe any application topology regardless of whether the application architecture is tier-based or Microservice-based.

Misconception 3: TOSCA is a configuration template
It's actually much more. TOSCA is often used as a templating language, and as such, earlier implementations of TOSCA-based orchestration used TOSCA simply as an input file, similar to the way people use configuration files.

TOSCA is a language, not a configuration template. It has many of the characteristics of an object-oriented language and defines Interfaces, Inheritance as well as new concepts that are specific to the application orchestration domain such as Relationships, Operations, Policies, Workflows etc. This makes TOSCA a fairly rich extensible language. TOSCA is good at describing application topologies in a simple, understandable way.

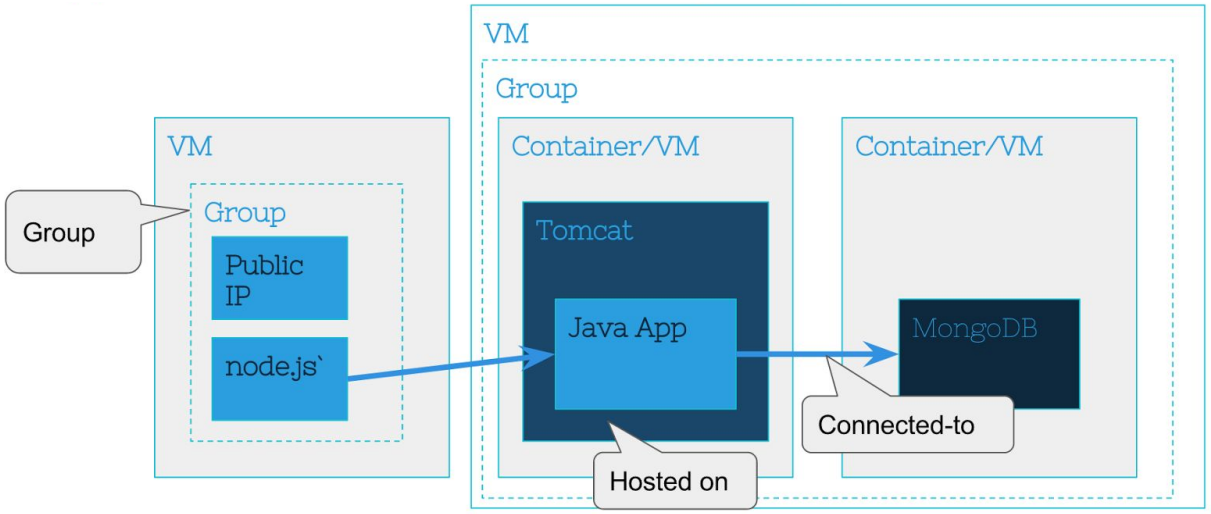
Model-Driven Microservices with TOSCA

The key concept behind Microservices architecture is to break large systems into a set of loosely coupled autonomous services.

TOSCA is a DSL that provides a way in which we can describe the data model behind each individual service, regardless of whether it's based on containerized or non-containerized components.

Model-Driven Design refers to an architectural concept in which the application, or service domain model, is defined first and the rest of the interactions with that service are driven from that domain model.

The following example provides a graphical illustration of a Model-Driven Microservice topology using TOSCA.



Now let's see what are the building blocks that are provided by TOSCA to model this service.

Node, Interfaces

A ***Node*** in TOSCA is analogous to a ***Class*** in other object-oriented languages. A node can define specific interfaces and can inherit from other interfaces. TOSCA comes with a set of built-in nodes.

Cloudify and ARIA add to that a fairly rich set of built-in node libraries that cover most of the popular cloud infrastructures such as AWS, OpenStack, VMware, GCP, Azure etc.

Relationship

Relationship defines the relationship between nodes. ***Contained-in*** means that a node runs within the related node and ***connected-to*** means that the node is connected to the related node but runs independently of that node.

Group

Group packages a certain group of nodes under a single unit (a.k.a "pod" in Microservices lingo). ***Group*** is often used to manage the lifecycle of the group as a single unit rather than by each individual node. For example, in the case of node.js and a Public IP node, in the example above, a scale policy on the group level will create another instance of node.js and assign it with a new instance of a Public IP node.

Adding custom resources as node types

As with other languages, we need to assume that the resources that come with the language or the platform will only cover a portion of the resources that need to be included in a specific application. The ability to add custom nodes is done through imports of other libraries as with other similar languages. The concept of creating extensions with Cloudify and ARIA is done through plugins.

Plugins allow the definition of new TOSCA node types and their implementation. In general, every resource with an API endpoint can be mapped as a node type in this way. A resource can be as simple as a public IP address or as complex as an entire platform such as Kubernetes.

Remote execution through workflow

Workflow defines the interaction and business logic operation between the nodes. TOSCA defines implicit and explicit workflows, and both assume access to the service domain model. Implicit workflows (such as the install workflow) are driven from the graph and relationship definition and responsible for executing the node's lifecycle operations according to their dependencies.

Explicit workflows, as their name suggests, are executed explicitly through a direct call and provide a way to interact with the service nodes in an abstracted manner. An example of such a workflow could be an upgrade workflow.

The upgrade workflow exposes a simple operation to the outside world on how to push a new upgrade to a particular service. The workflow is responsible for abstraction of the underlying interaction with all the related service nodes to execute that task. Such a workflow could include taking a snapshot from the database, pushing the new version of the service implementation, and transferring traffic from the load balancer to the new service, or rolling back in case the new version failed.

Discovery through relationship

Relationships provide a way in which one node can discover other related nodes. There are two types of relationships:

1) Explicit - in which a node points to a specific related node
2) Associative - in which a node discovers the related node through matching between requirements and capabilities properties.

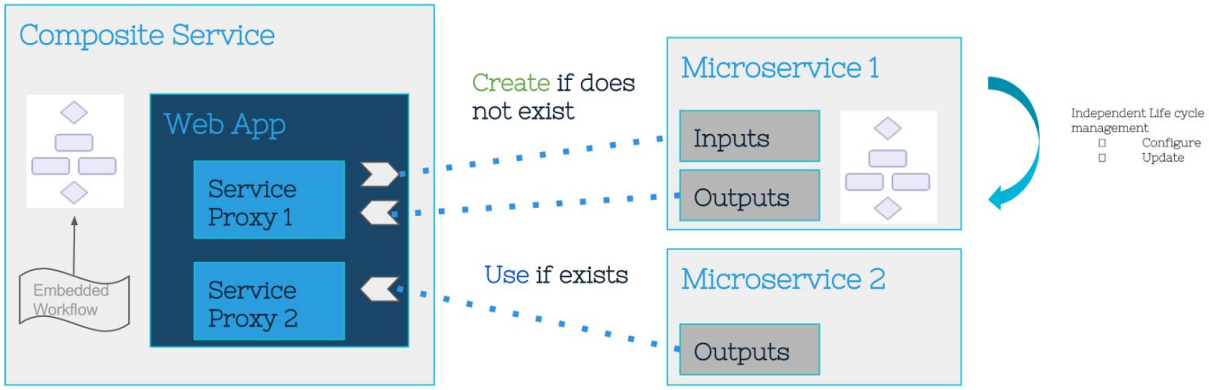
Service Composition Pattern with Cloudify

So far, we discussed how we can use the TOSCA language to describe the domain model of a particular Microservice.

Cloudify introduces a new pattern known as the ***Service-Proxy*** in which a user can describe an entire service as a single node type to other related services.

The ***Service-Proxy*** pattern uses the service input/output as a generic way in which a service interfaces with the outside world.

In the example below, we define a new Composite Service which is comprised of a set of node types, as with any service, but, in addition to that, includes two special service-proxy node types. The service-proxy node type is responsible for representing the output properties of the external service as local properties of that specific node type. It is also responsible for either discovering or launching the related services (Microservice 1 and Microservice 2 in the example below).



In this manner, we can leverage many of the existing TOSCA features for modeling the internal components of a specific service in order to model the relationship with external services as well.

Continuous Service Deployment Pattern

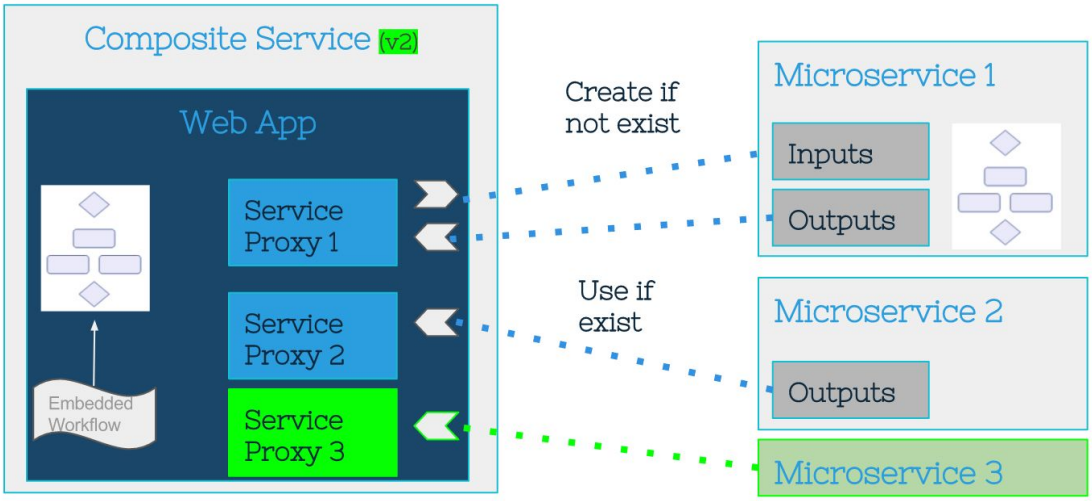
Updating a specific microservice

Cloudify provides a way in which users can update a service topology after it has been deployed through a feature that is called “deployment-update”. This enables users to update an existing service after it has been deployed with new nodes or changed existing nodes.

Adding a new service to an existing composite service

In the example below, we use a combination of the service-proxy pattern in conjunction with the deployment update to add a new service to an existing composite service.

Here we see the deployment update feature being utilized to add a new service-proxy node and point that node to the new Microservice.



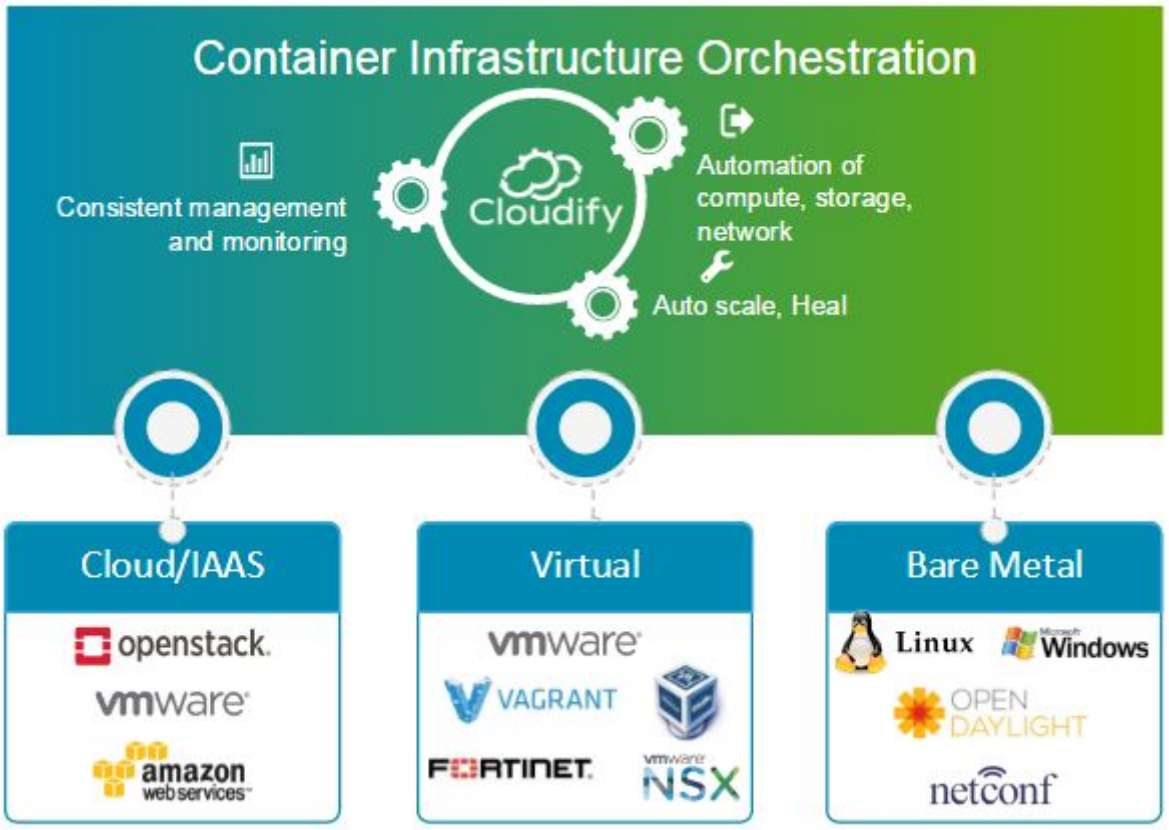
Container support for Kubernetes, Docker, Docker Swarm, and Mesos

Cloudify supports integrations with Docker and Docker-based container platforms, including Docker Swarm, Docker Compose, Kubernetes, and Apache Mesos. Cloudify can both manage container infrastructure and/or orchestrate the services that run on container platforms.

When orchestrating container orchestrators such as Kubernetes, Docker Swarm, and Mesos, Cloudify provides infrastructure management capabilities such as installation, auto-healing, and auto-scaling. When orchestrating services on these platforms, Cloudify integrates seamlessly with native descriptors to not only support container cluster service deployment, but also enable orchestration that encompasses systems beyond the edges of the container cluster.

Infrastructure Orchestration

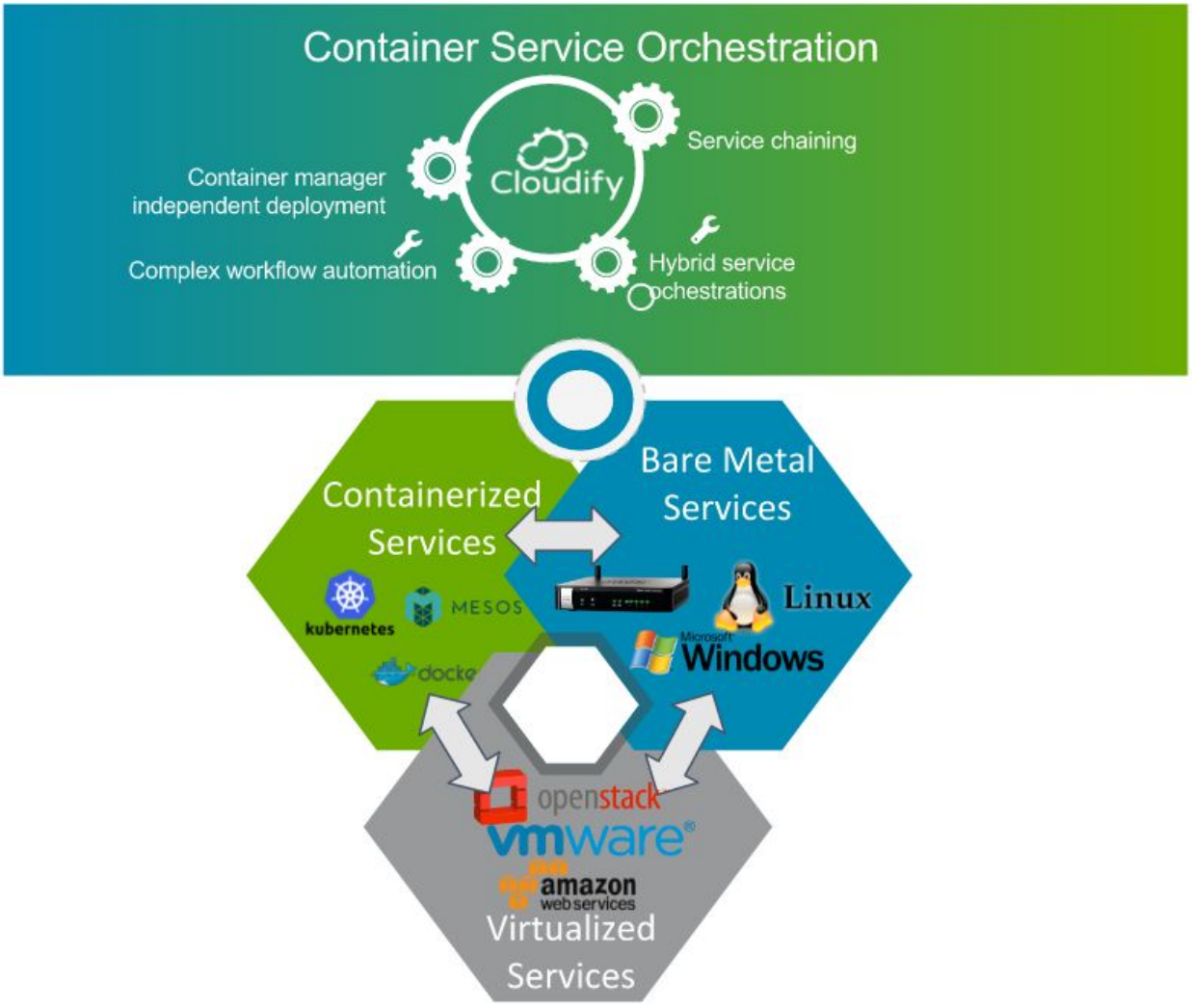
Cloudify can be used to create, heal, scale, and tear down container clusters. This capability is key in providing a scalable and highly available infrastructure on which container managers can run.



Cloudify can also orchestrate related infrastructure on bare metal, virtualized, and cloud platforms. This can include networking and storage infrastructure, both virtual and physical.

Service Orchestration

Independent of the infrastructure orchestration, Cloudify provides the ability to orchestrate heterogenous services across platforms. By leveraging the strength of TOSCA modeling, Cloudify can manage the instantiation and configuration of service chains regardless of the target platform. This ranges from containerized to virtualized to "bare metal" OS to physical hardware.

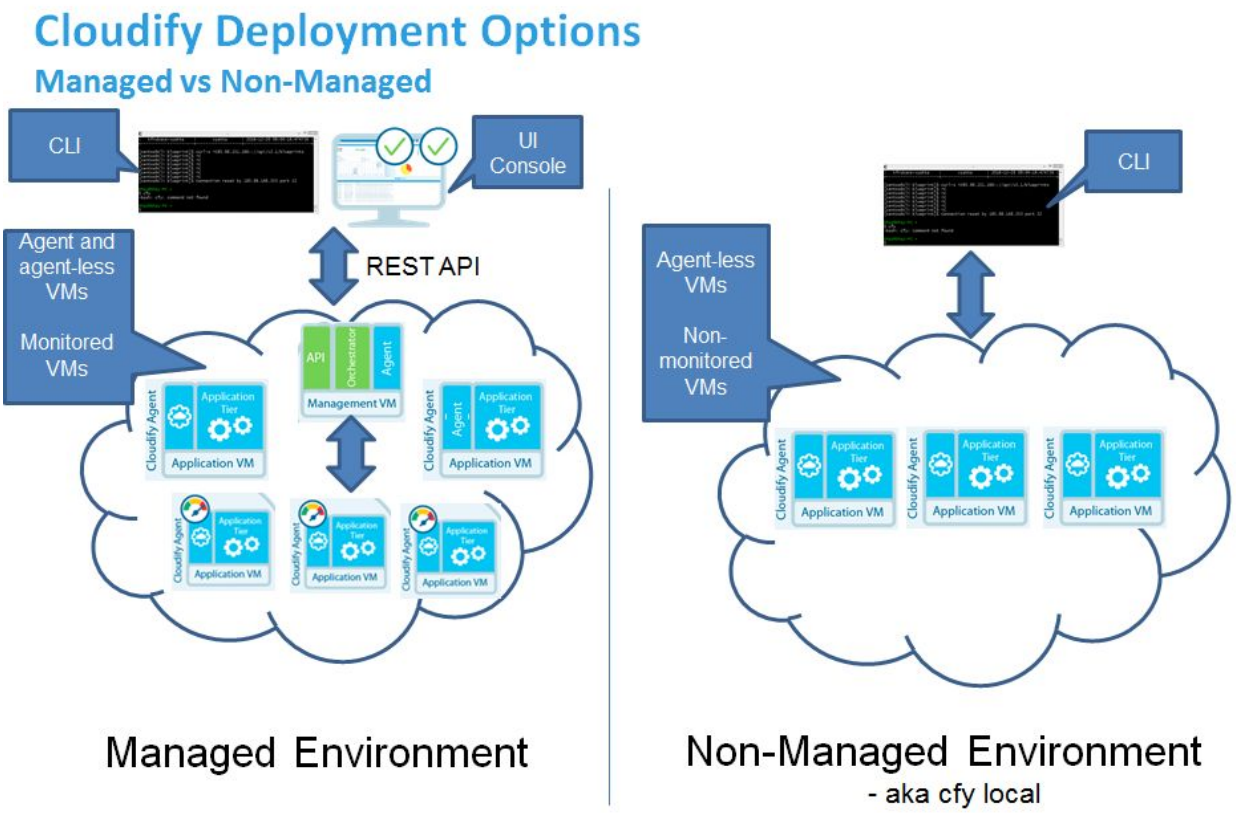


For more information on this see our Container Support documentation.

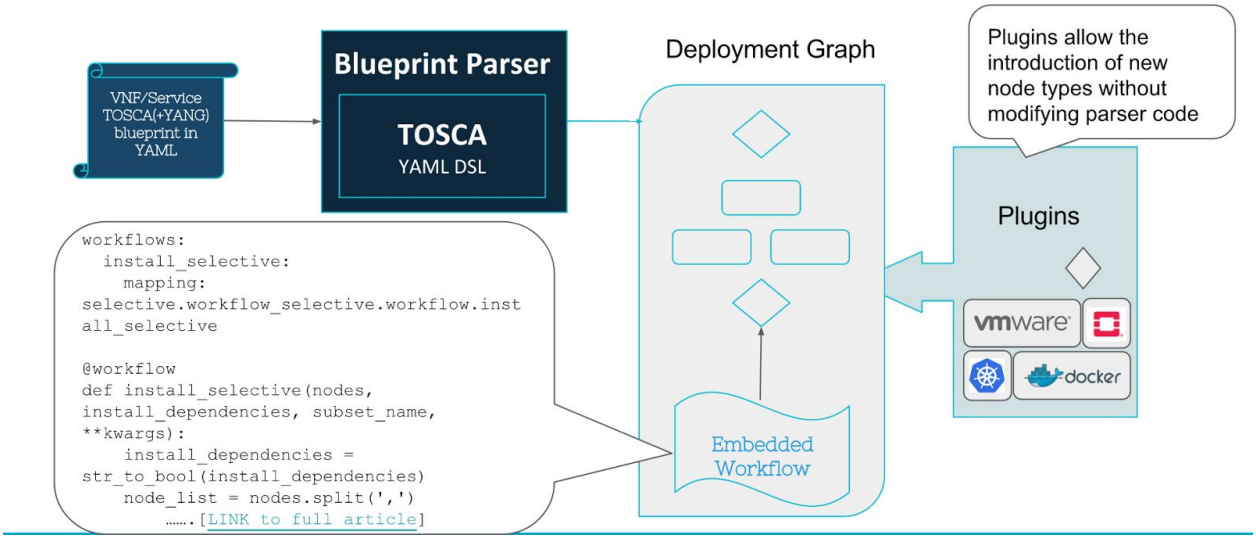
Putting it all together

Now that we covered the core patterns, let's see how the entire framework works with Cloudify. There are generally two modes of operation that are available - managed and non-managed environments.

With the non-managed option, a command line tool is provided. This concept is similar to other DevOps tools such as Ansible, Terraform etc.



The diagram below provides a description of the core components behind this framework.



The first element (top-left of above image) is known as a blueprint. This is basically a TOSCA YAML file that include the definition of a particular service.

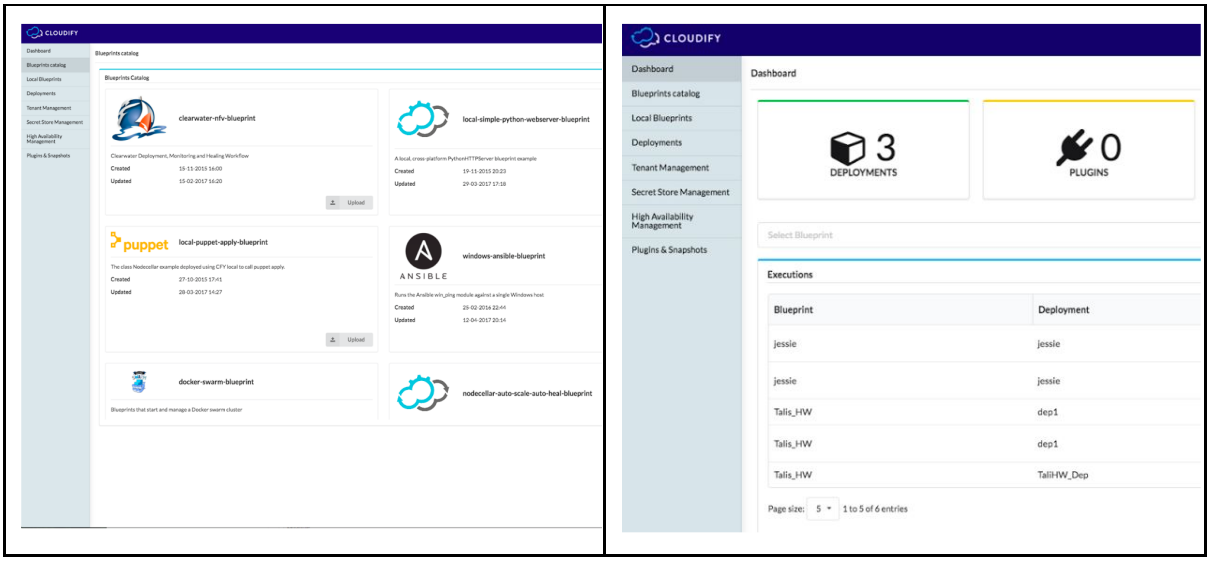
The Cloudify parser takes that definition and parses it into its domain model and execution plan (tasks) based on the definition of the nodes and their relationships.
The plugin library provides the resource implementations behind each node.

Cloudify executes the lifecycle operation of the relevant node in the service definition based on their dependencies and relationships, and the result is a fully deployed service.

Explicit workflows that are also part of the service definition allow continuous interaction with the deployed service. A typical workflow might be an upgrade, heal, or scale workflow.

Microservices deployment management as a service through the Cloudify Manager

The simple command line interface is a useful tool for those who want the “send and forget” experience in which we use the command line mainly to spawn the service but not to continuously manage it.



Cloudify Manager provides a service front end for that framework which allows executing many concurrent deployments in a multi-tenant fashion. In addition, it adds monitoring, logging and management capabilities that make it easier to use as an enterprise service.

Learn more at http://Cloudfy.co.