



UNIVERSITÀ DEGLI STUDI DI URBINO CARLO BO

Dipartimento di Scienze di Base e Fondamenti
Scuola di Scienze e Tecnologie dell'Informazione

Tesi di Laurea

CONTAINERIZING THE CLOUD FOR MANAGING APPLICATIONS AT SCALE

Relatore:
Chiar.mo Prof. Alessandro Bogliolo

Candidato:
Antonio Esposito

Correlatori:
Dott. Luca Ferroni
Dott. Andrea Seraghiti
Dott. Saverio Delpriori

Corso di Laurea in Informatica Applicata
Anno Accademico 2014-2015

To my parents

Contents

1	Introduction	2
1.1	Organization	3
2	Background	4
2.1	Manageability in IT Operations	4
2.2	Velocity in Development Workflow	6
2.3	Scalability in Software Architecture	8
2.4	Flexibility in Application Environments	9
2.5	Key Factors: Free Software and Sharing Economy	10
2.6	Case of Study: Gasista Felice by beFair	12
2.7	A Solution for Current State of Art	12
3	Containerized Applications	15
3.1	Container Runtime	15
3.2	Gasista Felice's Architecture	17
3.3	Container Images	18
3.4	Container Orchestration	19
4	IaaS and PaaS bootstrapping	22
4.1	IaaS Providers	23
4.2	Provisioning of IaaS	24
4.3	Provisioning of PaaS	27
5	PaaS Management	30
5.1	The Container Cluster's Core	30
5.2	Microservices Scaling and Orchestration	31
5.3	PaaS for Deploying Applications	33
5.4	Benchmarks	37
6	Monitoring and Logging	39
6.1	Time-Series Storage	40
6.2	Resource Gathering	40
6.3	Log Processing	42
6.4	Data Visualization	42

7	Conclusions	45
7.1	Future Developments	45
A	Source code	47
A.1	Gasista Felice	47
A.1.1	Dockerfile of NGiNX base image	47
A.1.2	NGiNX base configuration file	47
A.1.3	Proxy's Dockerfile	48
A.1.4	NGiNX configuration file	48
A.1.5	Dockerfile of uWSGI/Python2 base image	50
A.1.6	Backend's Dockerfile	51
A.1.7	Frontend's Dockerfile	52
A.1.8	Docker Compose file	52
A.1.9	OpenShift template	53
A.2	IaaS and PaaS bootstrapping	59
A.2.1	Makefile	59
A.2.2	Terraform template for Digital Ocean	61
A.2.3	Master Cloud Config template	64
A.2.4	Node Cloud Config template	67
A.3	Monitoring Stack	69
A.3.1	OpenShift template	69
A.3.2	Heka's configuration file	76
	Bibliography	78
	Acknowledgments	81

List of Figures

2.1	Evolution in IT Operations	6
2.2	Evolution of Development Workflow	8
2.3	Evolution of Software Architectures	9
2.4	Evolution of Application Environments	10
2.5	From current state of art to a modern solution	14
3.1	Performance and Isolation in Multiapps Deployment	16
3.2	Overview of Docker Engine	17
3.3	Container Images Dependency Tree	19
3.4	Gasista Felice before containerization	20
3.5	Gasista Felice after containerization	21
4.1	Cluster variants	23
4.2	Infrastructure Provisioning with Terraform	26
4.3	Infrastructure Dependency Graph	26
4.4	Provisioning of the PaaS	29
5.1	Cluster Overview	30
5.2	Pods and Replication Controllers	32
5.3	Replicas and Scheduling	33
5.4	Scaling Pods with Kubernetes services	34
5.5	OpenShift adds Route and Template objects to Kubernetes	35
5.6	Gasista Felice template	35
5.7	OpenShift dashboard	36
6.1	Overview of monitoring system	40
6.2	The Heka Data Flow	42
6.3	InfluxDB logs dashboard	43
6.4	Grafana dashboard	44

List of Tables

2.1	Summary of Evolution in IT Operations	6
2.2	Summary of Evolution in Development Workflow	8
2.3	Summary of Evolution in Software Architecture	9
2.4	Summary of Evolution in Application Environments	10
2.5	Summary of Evolution in IT	12
5.1	Benchmark results	38

This work is licensed under a *Creative Commons Attribution-ShareAlike 4.0 International License* <https://creativecommons.org/licenses/by-sa/4.0/legalcode>

Chapter 1

Introduction

When the Internet and the Web started to spread all over the world, a first era of cloud computing[1] emerged in order to offer services like mail exchange and search engines. In the last decade, the exponential increment of connections, connected devices and people, led to the need of services for collaborative work and data synchronization between more devices, introducing new challenges.

Today cloud computing technologies are driving the evolution of Information Technology in numerous ways, leading to innovative approaches of managing software development and deployment processes, but also of the application architectures their own.

Since usually the evolution proceeds in gradual steps, most of current solutions are generally too bounded to past paradigms, and do not fit current necessities in an clear and elegant way, making the management a complicated task and reducing the scaling potential of applications. Other solutions are proprietary, therefore they are not publicly available and do not contribute, at least directly, to global progress.

In early 2013, with the announce of Docker¹, cloud computing saw a big step forward thanks to containers as the atomic unit of cloud platforms. Containers provide an isolated environment for running applications, or a part of them, and add a lightweight layer that decouples the application itself from the underling system. In June 2014, Google, with around 10 years of experience in managing containers in production, embraced Docker as emerging industrial standard, releasing Kubernetes², a container orchestration, as core tool for managing whole applications at scale.

The aim of this work is to propose a radical step forward, reusing the growing ecosystem of free software built around containers, in order to provide a platform[2] (PaaS) on top of an existing cloud infrastructure[3] (IaaS), with the goal of provide a suitable environment for running applications[4] (SaaS) in a modern and scalable way. The proposed solution aims to provide a lightweight

¹<https://www.docker.com/docker-engine>

²<http://kubernetes.io/>

approach for a small and modern company, using Docker and Kubernetes, including projects like CoreOS³ by the namesake company, RedHat's OpenShift⁴ and HashiCorp's Terraform⁵, but also InfluxDB⁶, Heka⁷ and Grafana⁸ as monitoring stack, and other minor projects. All these tools are quite young since they are around 1-3 years old.

*Gasista Felice*⁹ has been used as case of study, so it has been containerized for running on top of our platform, providing scalability analysis and some benchmarks.

This project proved that a clear and modern solution could be used for deploying common applications in a small environment, providing all the necessary for management, scaling and guaranteeing the governance through monitoring. Finally, the project has been developed as multiple sub-projects publicly released as free software on the GitHub¹⁰ portal.

1.1 Organization

Chapter 2 introduces the working environment in which this thesis was born and the relative technological context.

Chapter 3 introduces some container core concepts, then exemplified their application in a specific use case.

Chapter 4 describes how to bootstrap an IaaS/PaaS in order to provide a production environment.

Chapter 5 describes the stack of PaaS showing how applications could take advantage of it.

Chapter 6 shows how to control, monitor, analyze, and visualize data about applications.

Chapter 7 tries to draw conclusions about the entire work, and presents some potential further developments of this project.

³<https://coreos.com/>

⁴<http://www.openshift.org/>

⁵<https://terraform.io/>

⁶<https://influxdb.com/>

⁷<http://heka.readthedocs.org/>

⁸<http://grafana.org/>

⁹<https://befair.github.io/gasistafelice/>

¹⁰<https://github.com/>

Chapter 2

Background

This chapter introduces the technological context in which this thesis was born with some references to the working environment in which it is implemented.

Since this document covers the workflow from designing application architecture in containers to deploying it in a real production environment, as well as using the application in development and QA environments, the context is presented from multiple points of view:

- the automation of the IT operations;
- the evolution of the development workflow;
- the scalability of architectures in the cloud;
- the flexibility of multiple environments;
- the key factors of Free Software and Sharing Economy that has led useful tools and practices used to develop the project.

2.1 Manageability in IT Operations

Common GNU/Linux distributions comes with a built-in package manager that provides a powerful entrypoint for installing and managing packages, from libraries to graphical applications. Using this feature, historically system administrators managed servers (IaaS) manually directly via shell, upgrading the system, installing and configuring the necessary packages. This approach introduces human errors and non reproducible operations, so it's unsuitable when scaling the numbers of engineers, hosts, applications and tasks. For example, it's difficult updating all the configurations, because there is no formal way to do it.

Even if the manual management is immediate and doesn't need preplanning, it has several limits. A first step derives from other industrial revolutions, minimizing the human intervention and *automating* all operations as much as

possible, via scripting or dedicated tools for *application deployment* and *configuration management* (e.g.: Chef¹, Puppet², SaltStack³, Ansible⁴). So humans doesn't do things directly, but instruct machines to do it in a more efficient, reproducible and reliable way. As described in *Pets vs Cattle*[5], while traditionally the server hostnames were chosen creatively as if they were cats, in an industrial approach these have to be chosen in a more pragmatic way as if they were cattle, for example with a semantic short name and a counter.

Even if this approach worked for years at scale, the application level remains coupled to underlying infrastructure. In *PaaS*, instead, there is a clear separation of infrastructure and application levels.

From Wikipedia:

In the PaaS models, cloud providers deliver a computing platform, typically including operating system, programming-language execution environment, database, and web server. Application developers can develop and run their software solutions on a cloud platform without the cost and complexity of buying and managing the underlying hardware and software layers. [...]

In 2009 *Ian Murdock*, the Debian founder, in its article *Do operating systems still matter?*[6] concluded that today developers using a PaaS are not more concerned about underlying hardware or operating system related features, but also about platform-level exposed features, so operating systems should be rethought in order to expose a better, more integrated higher level experience.

Also RedHat in *The Platform Abstraction Advantage of PaaS*[7] concludes with the following:

Clearly, platform abstraction solves more problems for IT than an IaaS+ approach. Enterprises embrace cloud for agility benefits and having an abstraction at the platform layer maximizes these benefits. DevOps is more attractive for enterprises because their IT can now deliver at the speed of business and the standardization of application development platform through abstraction at higher layers of stack and self service option for developers enables IT to meet the business needs.

Table 2.1 and figure 2.1 show the evolution of IT Operations, from manual IaaS approach to a modern automated PaaS.

¹<https://www.chef.io/chef/>

²<https://puppetlabs.com/>

³<https://saltstack.com/community/>

⁴<http://www.ansible.com/>

Table 2.1: Summary of Evolution in IT Operations

Model	Operations	IaaS/SaaS strong separation
IaaS	manually	no
IaaS+	automated	no
PaaS	automated	yes

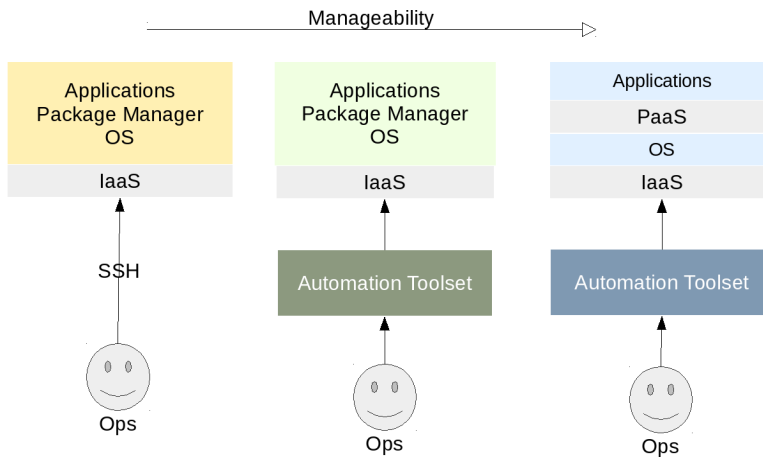


Figure 2.1: Evolution in IT Operations

2.2 Velocity in Development Workflow

In 2001 was published the *Agile Manifesto*⁵, a set of 12 principles that represent a first step towards a faster and more efficient approach to developers needs, going beyond the traditional *waterfall* development. From the development workflow point of view, agile practices introduced the *Continuous Integration* (CI), a system in order to run tests automatically and notify to users when tests fail.

While Agile methodologies made the development faster, the users will not receive this since delivery of applications remains slow. That create the necessity for a more collaboration/interaction between development and operations/delivery teams in order to automating the application deployment as soon as it passed the tests. This is called Continuous Delivery[9], extending the CI concept to production.

Consequently, also development and production environment should getting close, aim to the so called *dev/prod parity*. This way to build applications was formerly by Heroku, popular PaaS, with *The Twelve-Factor App*[8], a 12 good practices to apply at SaaS:

⁵<http://www.agilemanifesto.org/>

1. Codebase: One codebase tracked in revision control, many deploys
2. Dependencies: Explicitly declare and isolate dependencies
3. Config: Store config in the environment
4. Backing Services: Treat backing services as attached resources
5. Build, release, run: Strictly separate build and run stages
6. Processes: Execute the app as one or more stateless processes
7. Port binding: Export services via port binding
8. Concurrency: Scale out via the process model
9. Disposability: Maximize robustness with fast startup and graceful shut-down
10. Dev/prod parity: Keep development, staging, and production as similar as possible
11. Logs: Treat logs as event streams
12. Admin processes: Run admin/management tasks as one-off processes

12factor separate clearly development and operations through a contract, and represent the beginning of DevOps[11] era. Other PaaS shared and was born on top of these principles, such as Deis⁶ or OpenShift.

In a DevOps context there are usually at least these components:

- *Source Code Management* provides repositories hosting for Git (or other source code versioning system) projects (e.g.: Gogs⁷, GitLab⁸);
- *Continuous Integration/Continuous Delivery* execute automated tests and send somewhere else when software is ready to deploy in staging, QA or production environment (e.g.: Travis CI, GitLab CI⁹, Drone¹⁰, Jenkins¹¹);
- *Platform as a Service* is the core of infrastructure and automatically manage the applications, including the upgrade, some monitoring features (e.g.: Deis, OpenShift).

Table 2.2 and figure 2.2 show the evolution in development workflow, from the traditional waterfall to the modern DevOps.

⁶<http://deis.io/>

⁷<http://gogs.io/>

⁸<https://about.gitlab.com/>

⁹<https://about.gitlab.com/gitlab-ci/>

¹⁰<https://drone.io/>

¹¹<https://jenkins-ci.org/>

Table 2.2: Summary of Evolution in Development Workflow

Workflow	Development cycles	Delivery cycles
Waterfall	long	long
Agile	short (CI)	long
DevOps	short (CI)	short (CD)

Velocity in Development Workflow

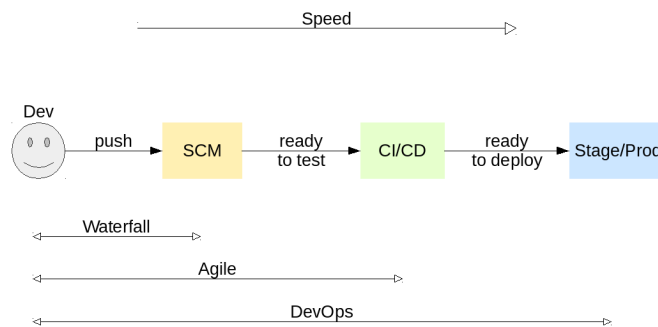


Figure 2.2: Evolution of Development Workflow

2.3 Scalability in Software Architecture

The most common paradigm today is developing *layered* applications thanks to frameworks, like Django and Rails, that enables developers to separate the various components, like data, business logic, routing, templating, caching, etc. This approach has permitted for a better manageability of spaghetti-code applications since it's possible to find where is a particular piece of the application and enables scaling through the layers, usually web/caching, application and database server components.

While this lead several advantage, with increasing the sizing of application, it could be to big making hard developing and scaling. The *microservices* pattern make the modularization another step, and proposes loosely coupled, but independent applications that communicate via APIs. This make possible use the best technology (language, framework, database) for every specific

need. Some of them could also be stateless for data processing only, without persistent storage capability.

Table 2.3 and figure 2.3 show the evolution of software architecture, from monolith application to a set of microservices.

Table 2.3: Summary of Evolution in Software Architecture

Architecture	Vertical Scaling	Horizontal Scaling	Data partitioning
Monolith	yes	no	no
Layered Monolith	yes	yes	no
Microservices	yes	yes	yes

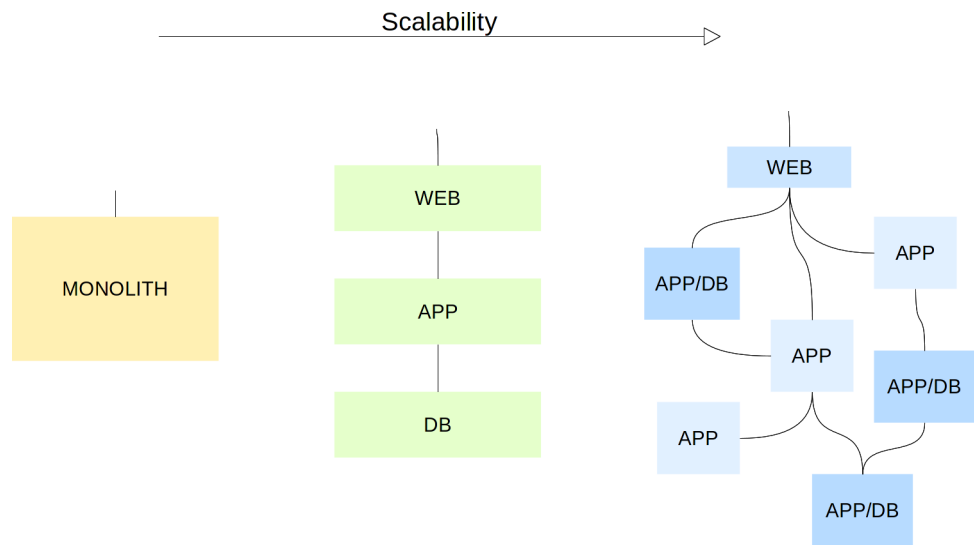


Figure 2.3: Evolution of Software Architectures

2.4 Flexibility in Application Environments

The final destination of every application is to be deployed somewhere for producing an added-value for users/customers. Thanks to advantage in terms of isolation and flexibility than *physical machines*, today *virtual machines* (VM) represent the atom unit on which applications are deployed. In fact, working on VMs is logically equivalent, with a little of overhead.

At the beginning of 2013, was announced Docker, a *container* runtime[10] that aims to be the unit of launching application components, running reproducible environments, and conceived for developers friendly via Git-like CLI. Containers doesn't offer all the isolation of VMs, but reduces the overhead and permits to optimize resources.

Container technologies already exists, but historically were thought as lightweight VMs in order to serve functional-complete operating systems. Instead, Docker focused on deploying only single processes, separating the OS from the application layers. Today containers, and Docker in particular, are in full diffusion and it has been adopted by large industry.

Table 2.4 and figure 2.4 show the evolution in application environments, from physical machines to containers.

Table 2.4: Summary of Evolution in Application Environments

Environment	Hardware abstraction	Flexibility
Physical Machines	no	no
Virtual Machines	yes	low
Containers	yes	high

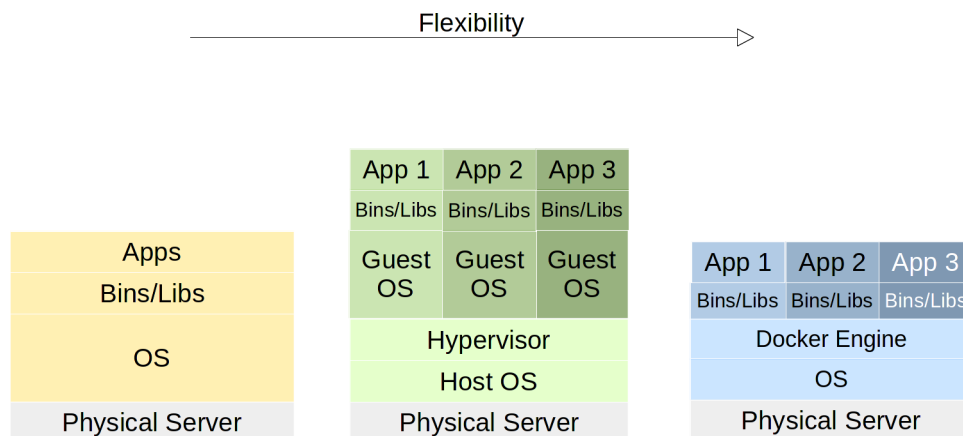


Figure 2.4: Evolution of Application Environments

2.5 Key Factors: Free Software and Sharing Economy

In February 1986 Richard Matthew Stallman¹² formalized¹³ the first formal definition of Free Software in Free Software Foundation¹⁴ (FSF). Then, the definition has been updated several times, and it's still maintained from FSF.

¹²<https://www.stallman.org/>

¹³<https://www.gnu.org/bulletins/bull1.txt>

¹⁴<https://fsf.org/>

Shortly, depends on that Free Software definition[12], a program is free software if the program's users have the four essential freedoms:

- The freedom to run the program as you wish, for any purpose (freedom 0).
- The freedom to study how the program works, and change it so it does your computing as you wish (freedom 1). Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help your neighbor (freedom 2).
- The freedom to distribute copies of your modified versions to others (freedom 3). By doing this you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this.

Free Software enables the opportunities of:

- keeping control over the software;
- learning from other design choices and code implementation;
- improving the software with bug-fixing and feature integration;
- optimizing costs with working on shared projects;
- obtain higher quality.

Free Software enables reuse and leverages the solutions, and enables rapid feedbacks lowering the "time to release". In the web and cloud age, Free Software gained a significant role and is everywhere. It's a key on which many companies and services rely their trust.

The Sharing Economy¹⁵ focuses on relations (a major interaction between providers and consumers), enhances the community, creates discussion places, and makes easier direct feedbacks. The Sharing Economy identifies a collaborative economy, based on cooperation. The Free Software fits very well in the Sharing Economy for the Information Technology topics since it avoid lock-in and other restrictive/bad practices of traditional economy, that aim to centralization instead of distribution and collaboration.

¹⁵<http://www.thepeoplewhoshare.com/blog/what-is-the-sharing-economy>

2.6 Case of Study: Gasista Felice by beFair

Gasista Felice is the main project developed by *beFair*, a small team involved in the *Free Software* and *Sharing Economy* networks. *beFair* is the working environment thanks to which it has been possible this thesis project.

Gasista Felice is a web platform to manage economy for solidarity-based purchasing groups (GAS) and suppliers involved in Solidarity-based economy districts (DES). They all practice Sharing Economy.

The currently production version of Gasista Felice is a web application built on MVC framework with Python¹⁶ 2.7, Django¹⁷ 1.3 and PostgreSQL¹⁸, with non-REST APIs and a frontend in JS/jQuery¹⁹. The work is on a development version with Python and Django 1.7, PostgreSQL, REST APIs and frontend in AngularJS²⁰ and Bootstrap²¹.

2.7 A Solution for Current State of Art

The figure 2.5 represents a possible summary of the IT evolution in last decades from multiple point of views.

Table 2.5: Summary of Evolution in IT

	1990's	2000's	2010's
<i>Development Workflow</i>	Waterfall	Agile	DevOps
<i>IT Operations</i>	IaaS	IaaS+	PaaS
<i>Software Architecture</i>	Monolith	Layered Monolith	Microservices
<i>Application Environment</i>	Physical Machines	Virtual Machines	Containers

The solution provided is a *Proof of Concept* (PoC), thought beyond academic purpose for future real use, of a pre-configured free/libre toolset that enables IT operators to build a PaaS on top of a IaaS/cloud infrastructure, who aims to be optimal for container as the atomic unit, scaling microservices, and DevOps-oriented via CD. In addition, the context regards a small company with reduced engineering resources.

Beyond these core considerations, additional guidelines has been followed:

- elasticity;
- lightweight: should not occupy too much resources, compiled languages over interpreted ones;

¹⁶<https://www.python.org/>

¹⁷<https://www.djangoproject.com/>

¹⁸<http://www.postgresql.org/>

¹⁹<https://jquery.com/>

²⁰<https://angularjs.org/>

²¹<http://getbootstrap.com/>

- reuse pre-existent components: don't reinvent the wheel;
- reuse pre-existent network layers: avoid dedicated transport layer in order to minimize further management and reducing the surface attack;
- microservices architecture: composable pieces with simple interface over monolith;
- convention over configuration: by default it should work out-of-the-box;
- automated bootstrapping for lowering the entry wall;
- avoid tools requires programming language or custom syntax for configuration;
- stay on shoulders of giants: avoid too small community and risk to adopt tools without a solid/active contributor community;
- minimize external dependencies at runtime;
- low dependency from IaaS for (future) private cloud.

With these goals in mind, the following toolset has been chosen:

- Terraform: IaaS provisioning, covering the lower layer of this stack
- CoreOS, the operating system optimized for running containers
- Docker Engine, the container runtime for components of applications
- Kubernetes extends CoreOS and Docker adding cluster scheduling and orchestration, and providing a cluster abstraction to upper level
- OpenShift v3, on top of this stack, reaches the PaaS abstraction level providing container services and workflow for developers

At the end, there is Gasista Felice as use case application running on top of this platform.

Figure 2.5 shows the current state of art on left side, and the proposed solution on the right side, covering from IaaS to SaaS abstraction levels.

In last months have been published interesting and similar projects for larger use cases, that should be quoted for completeness:

- Apollo²² from Capgemini, an open-source platform for apps, microservices and big data. It consists of Mesos cluster provisioning and orchestration using Packer²³ and Terraform;

²²<https://github.com/Capgemini/Apollo>

²³<https://packer.io/>

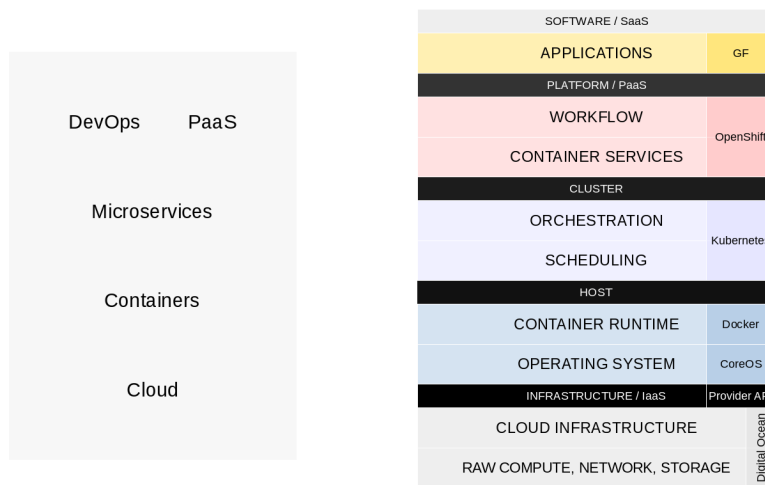


Figure 2.5: From current state of art to a modern solution

- MANTL²⁴ from Cisco Cloud, formerly *Microservices Infrastructure*, is a modern platform for rapidly deploying globally distributed services.

²⁴<https://github.com/CiscoCloud/microservices-infrastructure>

Chapter 3

Containerized Applications

This chapter introduces container core concepts, differences from previous virtualization and container technologies, and how the use case application could be containerized.

3.1 Container Runtime

Docker Engine was released at the beginning of 2013 from a PaaS company, as a lightweight alternative written in Go¹ to VMs to running and develop components of applications. Since then, it gained consent by large part of IT industry and today it's the de-facto standard for container technologies.

While traditional container technologies like OpenVZ³ aimed to emulate a full functional operating system, Docker focused only on running single processes, separating the application from the OS.

Some of the advantages in using Docker are:

- reproducible and standardized environments between development, staging, QA and production

¹The Go Programming Language²<https://golang.org/>), as known as *Golang*, covers a relevant role since it's the core of the vast majority of the software adopted, and in some ways, it could be considered a forerunner which let to containers.

Golang is a recent language developed at Google in 2007, it's atypical since it join traditional low-level language, like compiled and static typing, with high-level language, such as garbage collector and high productivity, adding modern primitives such as concurrency and vast standard library for web related topics. The result of the Go compiler generate is a binary without external dependencies, relatively fast in running, so once a software has been compiled for an *linux/amd64* operating system/architecture, it could be launched on every (modern) GNU/Linux distributions for desktop or server where a Go environment is totally absent, making very straightforward deploying applications written in Go.

In addition it's strong opinionated on conventions, standardization and built-in tools in order to improve team productivity, including testing, linting, trans-compiling and other tools, so it feel good in companies.

³<https://openvz.org/>

- reduce overhead respect to VMs used for deploy single application (as shown in figure 3.1)
- improve isolation and security respect to OS that hosts more applications (as shown in figure 3.1)

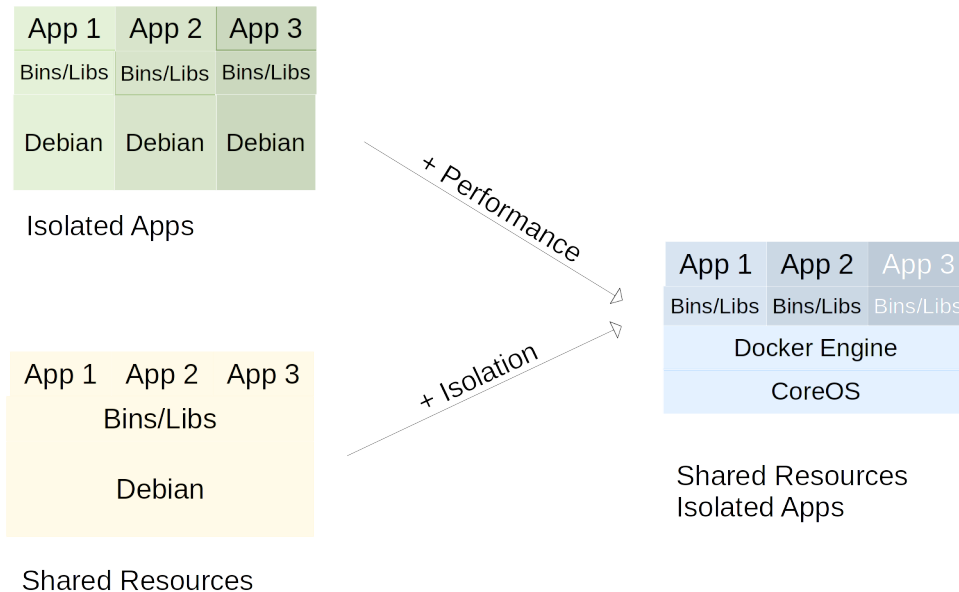


Figure 3.1: Performance and Isolation in Multiapps Deployment

Under the hood, Docker Engine is composed mainly by a Git⁴-like command line interface, a daemon and *libcontainer*, a library that uses Linux kernel capabilities such as *namespaces* and *cgroups*, as shown in figure 3.2.

Someone sees containers more as “lightweight VMs”, so has been prepared an alternative base image⁵. In this case there is a canonical init daemon as PID 0, instead of the application process itself. This was particularly useful in the early period since it was needed for some components that depend on specific system features, such as syslog, and other purposes. But today Docker includes directly that kind of features, so this need is decreased.

There are also alternative container runtimes, such as rkt⁶ (read “rocket”) that focuses on reliability and security in production environment.

On June 22, 2015 was announced the *Open Container Project*[13] from Linux Foundation⁷ and major companies with the target of an industry-wide standard, taking Docker as the starting point.

⁴<https://www.git-scm.com/>

⁵<https://github.com/phusion/baseimage-docker>

⁶<https://coreos.com/rkt/docs/latest/>

⁷<http://www.linuxfoundation.org/>

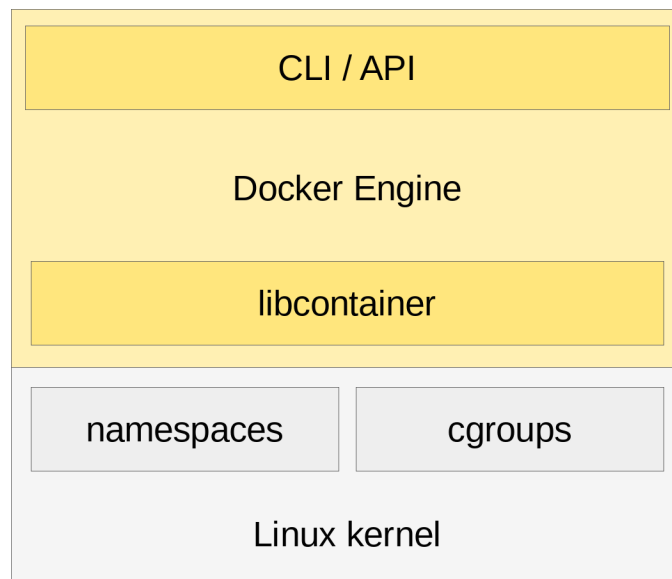


Figure 3.2: Overview of Docker Engine

3.2 Gasista Felice's Architecture

To apply this to the use case Gasista Felice, the first step is splitting the application in well-defined components, then containerize one by one.

In a traditional way, one would have said that the components are AngularJS for frontend, and Django and PostgreSQL for backend. Instead, in DevOps-oriented context with dev/prod parity, components should be the same used in production:

- *NGINX*⁸, web/caching server and reverse proxy;
- *harp*⁹, frontend server with built-in preprocessors;
- *uWSGI*¹⁰, web application server;
- *PostgreSQL*, relational database/

In addition, following the *12factor* guidelines, a convention is to use:

- environment variables for configuration;
- TCP sockets for inter-container communication.

So Django settings should be based on *environment variables* with some defaults. Has been chosen `APP_ENV` as main variable that determines a different

⁸<http://nginx.org/>

⁹<http://harpjs.com/>

¹⁰<https://uwsgi-docs.readthedocs.org/>

behavior of backend. For example, when in development environment, there is the need of reloading app as soon as possible with code changes, while in production environment there will need of more workers and a periodical offloading task.

Another convention is to minimize the mutating components, so the file system should be read-only by default (including binaries and libraries), and only well defined directories should be writable (e.g. files uploaded by application users).

3.3 Container Images

As of virtual machines consists of a base disk image and one or more instances, so containers providers base images and one or more instances (“real” containers) running on. In particular, Docker applies the concept of *Copy on Write* (CoW) that consist of, starting from a well-defined file system status, write only the differences from it, in order to minimize the disk usage.

There are base images of common operating system (Debian¹¹, Ubuntu¹², CentOS¹³) but also language-specific runtimes (Python, NodeJS, Java). Starting from here it's possible build custom images contains the application components. In addition, every image has one or more *tags* in order to provide different versions: e.g. in `debian:8` the `8` is the tags part that identifies *Debian Jessie*, but it's also possible define the Debian testing image with `debian:9`.

The main way to build Docker images is via Dockerfiles¹⁴, files that include:

- the base image via the **FROM** directive;
- a **MAINTAINER** reference;
- **ENV**ironment variables declaration;
- **COPY**ing data inside the image;
- **RUN** commands;
- **TCP** ports to expose via **EXPOSE** directive;
- **CMD** as the default command to run when starting the container;
- others as defined in Docker builder reference.

Following the Docker best practices¹⁵ has been developed some images as shown in the following picture:

¹¹<https://www.debian.org/>

¹²<http://www.ubuntu.com/>

¹³<https://www.centos.org/>

¹⁴<https://docs.docker.com/reference/builder/>

¹⁵https://docs.docker.com/articles/dockerfile_best-practices/

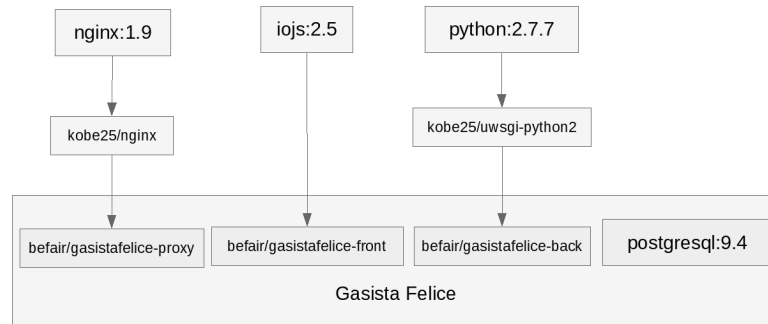


Figure 3.3: Container Images Dependency Tree

In order to follow the DRY principle, has been developed *nginx* and *uwsgi-python2* base images that provides a common environment for potential other containerized applications.

The backend of Gasista Felice is an application written in Python with Django web framework, and served by uWSGI application server. In the base image has been set default values for uWSGI, Python, Debian and PostgreSQL client, were added an *app* Unix user/group and were installed uWSGI and Python core packages. Note that since uWSGI is the default process, when a real container will be started, uWSGI take all the configuration from these defaults and eventual future additions of overriding.

Then the Dockerfile of Gasista Felice backend, including installation of dependencies, and copying of application code.

The NGINX/proxy component also has a base image that move logging to stdout/stderr and other small customizations.

Instead the Gasista Felice proxy Dockerfile provide a production-ready configuration, with static files caching.

Another components consist of the Harp server that served static processed content of frontend. In future this component could be eliminated, since once the files are generated they could be served directly from NGINX.

The PostgreSQL image instead is ready as it is, so it's not be customized.

3.4 Container Orchestration

Since an application is splitted in more well-defined components, there is also the need of a container orchestrator in order to run a multi-container application.

For this target has been chosen Docker Compose¹⁶, a basic tool to deploy applications in development, similar to Vagrant for VMs. Docker Compose will resolve the dependency graph of components and linked them together. While

¹⁶<https://docs.docker.com/compose/>

Docker Engine provides the core capabilities, Docker Compose represent the tool developers use the majority of time.

Following the `docker-compose.yml` reference¹⁷, the containers was orchestrated (only for development) with Docker Compose. So running the whole application is simple as run `docker-compose up` command!

Figures 3.4 and 3.5 show respectively the legacy version of Gasista Felice, a de-facto monolith application running inside Debian, and the refreshed ones with a strict separation between components.

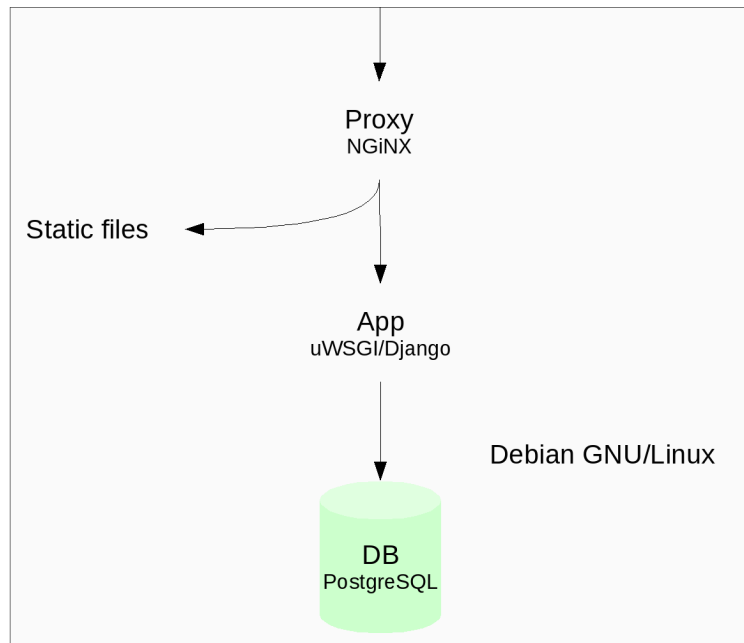


Figure 3.4: Gasista Felice before containerization

¹⁷<https://docs.docker.com/compose/yml/>

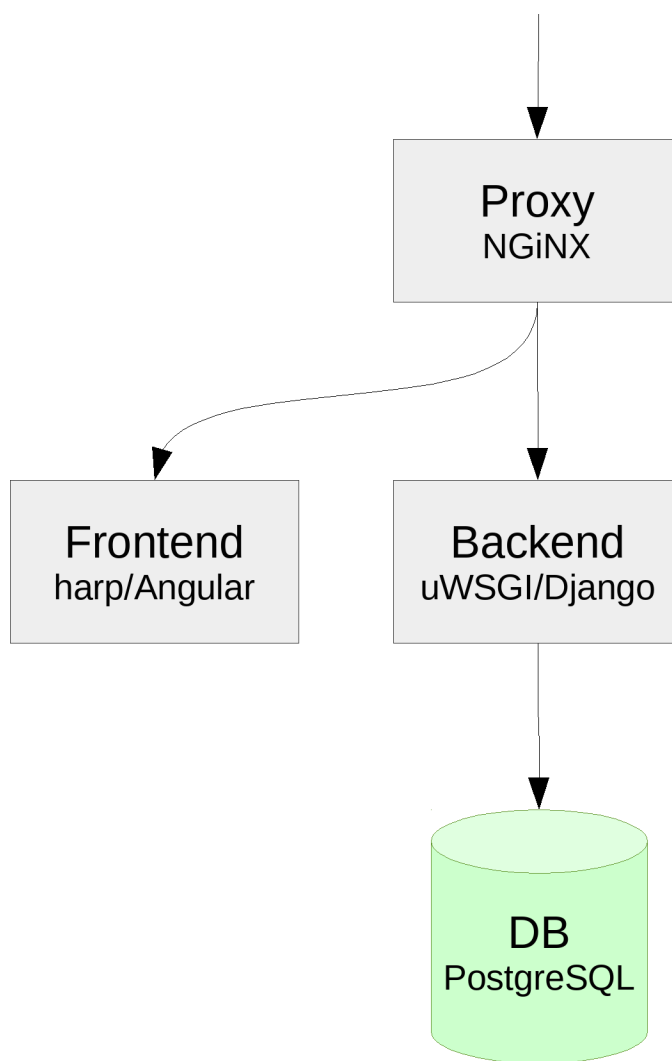


Figure 3.5: Gasista Felice after containerization

Chapter 4

IaaS and PaaS bootstrapping

In order to deploy a containerized application in staging or production, first there is the need of a suitable environment. While a development environment only need of launching one application with limited usage (only the developer itself) exposing only at localhost, a production environment has much more prerequisites. The application should be always available, supporting many concurrent users from a public DNS, and lastly many instances of the same application could run at the same time.

This chapter will cover the process of launching the platform (in the PaaS way) on top of an IaaS from a cloud provider, delegating to the next one the internals of this PaaS and how running applications on top of it.

Usually a non trivial production environment is not composed by a unique machine but it's distributed on more hosts collaborate each other to enable further possibilities. In particular CoreOS, Kubernetes and OpenShift were born with native concept of cluster, in which it's possible classify the hosts in 2 roles:

- *master* (M), a control host of the life of the whole cluster;
- *node* (N), a worker host on which run applications.

Then, looking at the possible clustered architectures provided, as shown in figure 4.1 has been divided 3 kind of possible cluster architectures depends of configuration complexity level:

- a single host with both master and node roles;
- 1 master and multiple nodes;
- multiple masters and nodes.

While the first one represents only a very basic limit case, and the last one an advanced configuration for medium to big environments, the second one fits the needs of a small environment, with a balance between roles isolation, low cluster overhead and configuration complexity. In particular will be used the simplest sub-case with a total of 3 hosts: 1 master and 2 nodes.

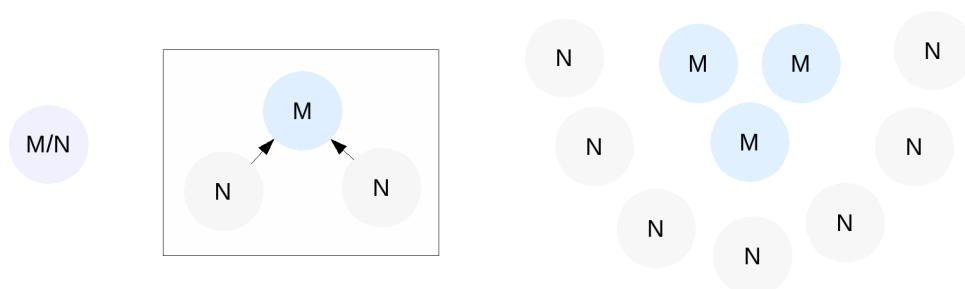


Figure 4.1: Cluster variants

4.1 IaaS Providers

The first step is choice an IaaS provider for machines (compute and storage) and networking. Even if there is no theoretical need of virtualization, there is a pragmatical one since the resources needed for every host is only a fraction of a whole modern physical server. It's relevant note that instead of using virtualization for separating applications, in this case it will used for take only a part of the resource of a physical server, and containers represent the level for isolating the applications. In conclusion, in order to pull up the cluster, there will take 3 VMs from a cloud provider.

In summary, the needed features of the cloud provider are:

- CoreOS native support, with additional possibility of customizing images;
- integration with existing tools (*Terraform* in particular);
- simple plans avoiding proprietary or provider-specific services;
- relatively economic.

Then will be useful some cluster-oriented features like:

- private network;
- low-cost for intra-cluster traffic.

In additional extra features could be the integration of DNS Management in order to cover all the infrastructure-level services.

Digital Ocean¹ (DO), built on top of free software *QEMU* and *KVM*, matches all the above points.

The master host doesn't run applications, and its sizing depends on number of nodes and containers to manage, while the sizing of nodes depends on resources needed for applications. After looking at Digital Ocean offer², the following plans were chosen:

¹<https://www.digitalocean.com/>

²<https://www.digitalocean.com/pricing/>

- 1 master with 1 core, 1GB memory and 30GB SSD;
- 2 nodes with 2 cores, 2GB memory and 40GB SSD each one.

For a total of 5 cores, 5GB memory and 110GB SSD. Finally, the cluster has to be composed by hosts in the same data center in order to keeping the latency in intra-cluster communication as low as possible.

4.2 Provisioning of IaaS

Using a PaaS means a lot of automation and good practices applied in deploying applications. For keeping a similar level of automation in provisioning all the lower stack than applications, there is need of additional tools.

In particular the tasks should be:

- describe the resource of infrastructure, possibly in a declarative way;
- resolve the dependencies graph of infrastructure resources;
- communicate with the provider through APIs (and auth token) doing the necessary operations;
- connecting via SSH to the hosts for bootstrapping;
- setting up the public DNS in order to point to a specific node of the cluster.

Ansible and *SaltStack* are popular choices today since includes several modules for lower-level operations (cloud provisioning) but also higher-level ones (application deployment and configuration management). Since the applications are managed entirely by the PaaS as will be shown in next chapter, these tools are over sized.

Terraform is a more minimal CLI-based tool, focuses only on cloud infrastructure provisioning on which results more productive and powerful. Terraform applies the changes based on:

- actual state: description of current resource states;
- desired state: description of resource states.

With these in mind, Terraform processes the data doing necessary operations aiming to keep from "actual state" to "desired state". A basic example could be:

- actual state: there is no machines;
- desired state: there is need of 3 machines.

So Terraform will communicate to provider via APIs in order to pull up 3 machines, then after the creation of these machines, their data (public IP, etc) will be stored as "actual state". Then, if there is no need of a VMs, it's only necessary change the desired state and re-apply Terraform.

This approach of Terraform enables the possibility of so-called *Infrastructure as Code* (IaC), that consists in versioning this files contains the states in order to tracks the infrastructure evolution, just as it is a common software project. This represent a more structured, deterministic and reproducible way to infrastructure management.

In addition, Terraform could be used for enabling powerful features like horizontal host-level auto-scaling, varying the cardinality of the cluster in response to global load changes.

The resources (in execution order) are:

1. access token for Digital Ocean APIs (previously generated by provider Web UI);
2. upload temporary public SSH key (used for connection to hosts);
3. PaaS bootstrapping, regarding VMs and configuration (will be covered in the next sub-chapter);
4. DNS management for *befaircloud.me*, a dedicated domain for this project, through Digital Ocean DNS service;
5. pointing **.befaircloud.me* A-type record to the first node.

In addition, some variables for parametrization are used for a more DRY configuration.

Figure 4.2 shows an overview of automated infrastructure provisioning via Terraform.

Figure 4.3 shows the dependency graph of infrastructure resources, automatically generated by Terraform itself.

All the Terraform configuration could be seen in the Appendix B. Note that while Terraform supports custom language, the *Hashicorp Configuration Language* (HCL) and JSON, the choice is to use a standardized, human-readable and clean YAML.

For launching all the infrastructure, it's so simple as running `make up`, that will do the following tasks:

1. cleaning previous temporary files;
2. compile YAML configurations into JSON;
3. creating a temporary SSH RSA 4096bit public/private key;
4. create an archive with necessary binaries and configuration files to upload on all hosts;

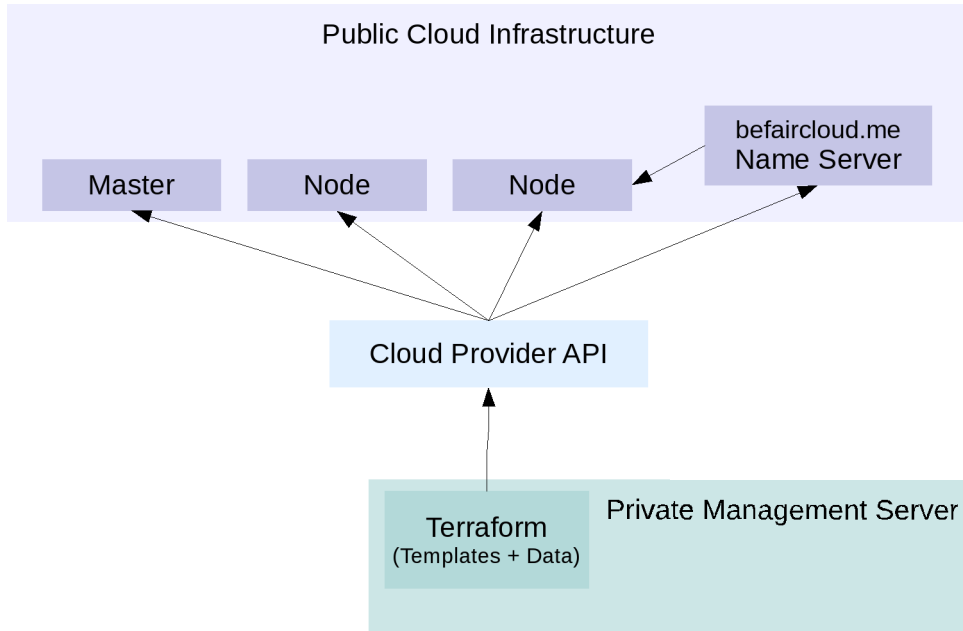


Figure 4.2: Infrastructure Provisioning with Terraform

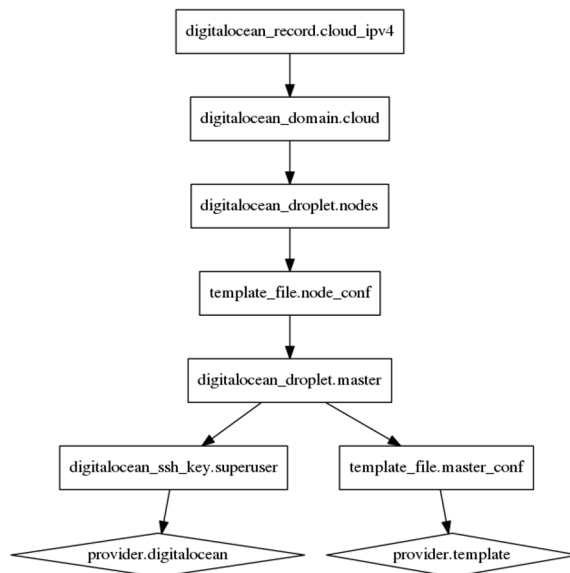


Figure 4.3: Infrastructure Dependency Graph

5. launching Terraform's `plan` in order to plans the operations to do;
6. launching Terraform's `apply` in order to apply the planned operations.

At the end, there is some output with commands for complete the bootstrapping and log via SSH to master:

Outputs:

```
step1 = scp -r core@46.101.133.104:./master .cache/; \
scp -r .cache/master core@46.101.138.191:./; \
scp -r .cache/master core@46.101.200.100:./

step2 = log to master -- ssh core@46.101.133.104
log to node-0 -- ssh core@46.101.138.191
log to node-1 -- ssh core@46.101.200.100
dashboard -- https://46.101.133.104:8443
entrypoint -- http://befaircloud.me
```

Now the whole cluster is up and running!

For enabling some basic services and monitoring stack, it's needed connecting via SSH to master, and running the `openshift/base` script, present on the home of the `core` user.

For deleting all resources (from VMs to DNS records), it's needed running `make destroy` that consists of:

1. `terraform plan -destroy` for planning deletion of all resources;
2. `terraform destroy` for deleting the planned destruction.

4.3 Provisioning of PaaS

Another core aspect to cover regards the PaaS bootstrapping, the process from vanilla CoreOS to a complete functioning clustered PaaS running Kubernetes and OpenShift. There internals of cluster is so postponed to the next chapter.

The key technologies used is a combination of Terraform's template feature and Cloud Config³, an emerging standard for operating system configuration at boot time via a declarative way. Cloud Config consists of a simple YAML file that an OS (CoreOS, Ubuntu, etc.) could read, with information about hostname, ssh keys and user permissions, files and other settings. CoreOS added additional features⁴ for SystemD units and cluster upgrade.

Since master and node have a different configuration needs, there are been developed two templates. Terraform use these templates and passing them

³<https://cloudinit.readthedocs.org/>

⁴<https://coreos.com/os/docs/latest/cloud-config.html>

when requesting the machines. The main dependency is provide the master private IPv4 address to nodes, so they can connect to the master.

The detailed steps for provisioning a master are:

1. Terraform resolves the Cloud Config template of master and resolves variables in the form `${VAR_NAME}`;
2. Terraform calls the API to pull up a new VM with vanilla CoreOS, passing the Cloud Config master, and uploading an archive (with necessary binaries and OpenShift configurations);
3. Digital Ocean parses this template replacing `$public_ipv4` and `$private_ipv4` variables;
4. Digital Ocean pulls up the master passing Cloud Config template to CoreOS;
5. CoreOS reads the full-rendered Cloud Config template, and apply the described configurations.

Once the master has been correctly deployed, the nodes could be deployed:

1. Terraform resolves the Cloud Config template of node, passing the *master IP*. So this template could be used for every node for that specific master;
2. the same steps as for the master.

The configuration described in Cloud Config is necessary for:

- enable SSH log-in to `core` user with provided key;
- provide a SystemD unit for bootstrapping the host and write the OpenShift/Kubernetes configuration (different between master and node);
- provide a SystemD unit for running the main OpenShift/Kubernetes binary (different between master and node).

Thanks to this capabilities, this entire process is (quite) totally automated, except for a single command, that is anyway automatically provided by Terraform at the end.

Figure 4.4 shows all the dependencies of PaaS bootstrapping, managed with a combination of Terraform, Cloud Config and SystemD units.

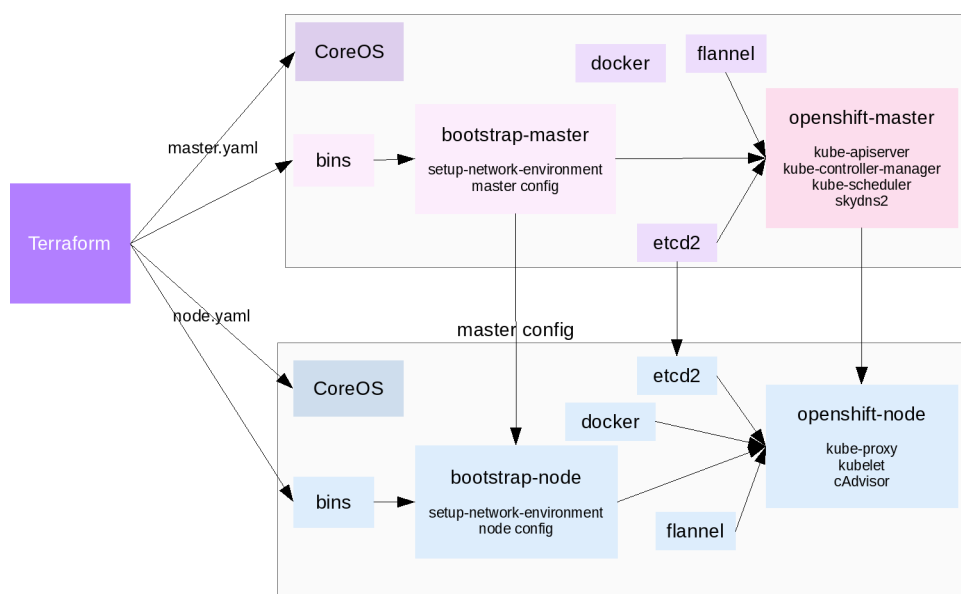


Figure 4.4: Provisioning of the PaaS

Chapter 5

PaaS Management

This chapter describes the stack around a container-based production environment, and how applications could take advantage on it. Figure 5.1 shows an overview of the whole clustered PaaS.

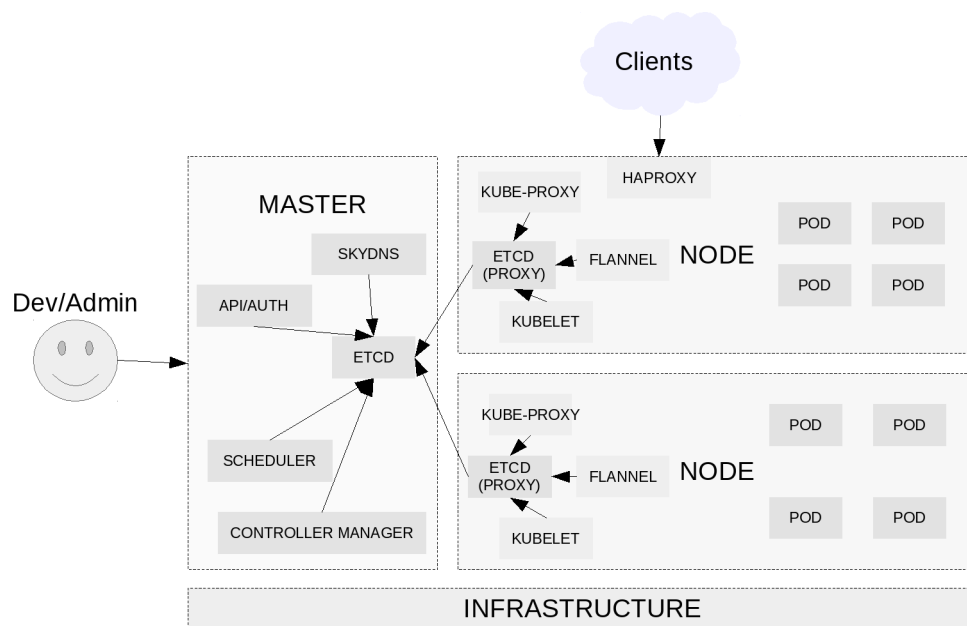


Figure 5.1: Cluster Overview

5.1 The Container Cluster's Core

In the container world, the operating system role should only be concerned in providing a minimal base for running containers. In fact, since all the application dependencies are demanded to containers, there is only need of basic

functionality, without a package manager and other features of a traditional OS.

The first and currently the most popular project that take into that direction has been *CoreOS*, announced in summer 2013 as a Gentoo-based GNU/Linux distribution for deploying containers in a clustered environment. CoreOS occupies few memory and comes with SystemD¹, Docker and additional tools for distributed cluster environments like Etcd².

SystemD is the de-facto standard in GNU/Linux distribution as init daemon. SystemD made really easy to demonize processes and has advanced features such as parallelization and socket activation. For example, the SSH daemon is stopped by default in order to save memory, and it will be started only after an incoming request on 22 TCP socket. SystemD is used for running directly the OpenShift/Kubernetes binaries, outside containers, and the other core tools.

Etcd is a key-value store which represent the database of the cluster since it has been used for inter-host communication by all components. It could be used as a single master as in this project, or in high availability and distributed mode, useful for multi-master cluster configuration.

Fleet³ provides a distributed init system, extending SystemD functionality from host-level to cluster-level. Fleet has not been used in this case, but represent an alternative to Cloud Config in settings up the SystemD units. In fact, it permits to bootstrap the OpenShift/Kubernetes stack from the master to all nodes.

Flannel⁴ has been development as a requisites of the Kubernetes networking model, providing a subnet mask to every host in order to enable an overlay network for container inter-host communication. At boot time, Flannel set the network data into Etcd under `/coreos.io` key.

CoreOS releases are organized in 3 channels: alpha, beta and stable. Alpha is generally release once a week and includes new software, while stable it's more suitable for production.

Most popular alternatives to CoreOS are Rancher OS⁵, Atomic⁶ by Red Hat and Ubuntu Snappy⁷ by Canonical.

5.2 Microservices Scaling and Orchestration

Kubernetes has been announced at Google I/O in June 2014, it's a cluster management system, a framework for container scheduling and orchestration

¹<https://wiki.freedesktop.org/www/Software/systemd/>

²<https://coreos.com/etcd/>

³<https://coreos.com/fleet/docs/latest/>

⁴<https://coreos.com/flannel/docs/>

⁵<http://rancher.com/rancher-os/>

⁶<http://www.projectatomic.io/>

⁷<http://www.ubuntu.com/cloud/tools/snappy>

in a clustered environment. Kubernetes has been developed by the same engineers who developed the current Google infrastructure, and takes advantage of that experience. While CoreOS provides the basic for running containers, Kubernetes enables an higher level through the container scheduling and orchestration across all the cluster.

From the experience at Google, Kubernetes introduces some additional concepts on top of containers:

- *pod*, a set of one or more containers that run together on a node, represents a single logic unit;
- *replicationController* is concerned about guarantee the correct numbers of replicas of a pod;
- *service* groups pods and provides a common DNS name and a load-balanced IP address to access them.

While containers could be seen as *atoms*, pods could be considered *molecules*. Replication controllers and services are both concepts built around pods, as shown in figure 5.2.

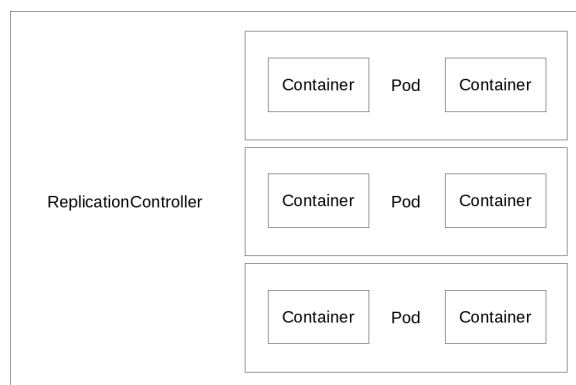


Figure 5.2: Pods and Replication Controllers

A basic example of pod composed by 2 containers could be an application server and a Redis⁸ instance for caching from database.

In Gasista Felice every component is composed by a pod, a replicationController and a service for providing access to it. Since there are 4 components, a minimal instance of the application includes 4 service, 4 replication controller and 4 pods. The number of the pods could be changed dynamically.

Kubernetes stores the data about pods, replication controllers and services into Etcd under the `/kubernetes.io` key.

In order to add these features, Kubernetes build on top of Docker, Etcd and Flannel, adding 3 master components:

⁸<http://redis.io/>

- *API Server* validates and configures the data for pods, services, and replication controllers;
- *Controller Manager* watches Etcd for changes to replication controller and then uses the API to enforce the desired state;
- *Scheduler* schedule the pods into nodes depends on resources availability or other parameters.

And 2 more components in nodes:

- *Kubelet* applies changes to the node as specified in local Etcd, and updated the data on Etcd depending on local state of containers, as shown in figure 5.3;
- *Proxy* manages the services defined in Etcd on that node, as shown in figure 5.4.

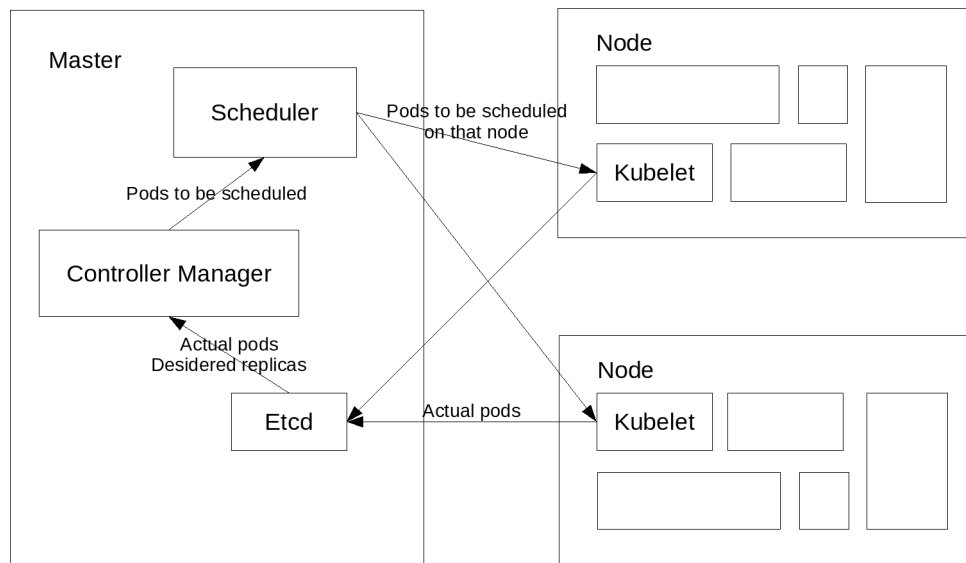


Figure 5.3: Replicas and Scheduling

On July 20, 2015 the *Linux Foundation* announced the *Cloud Native Computing Foundation*[14] in order to standardize the orchestration of components in a cloud environment, taking Kubernetes as one of the starting point.

5.3 PaaS for Deploying Applications

Even if Kubernetes represents a great solution for scaling services, it lacks the concept of application and the development workflow, so it's more a tool for

Services

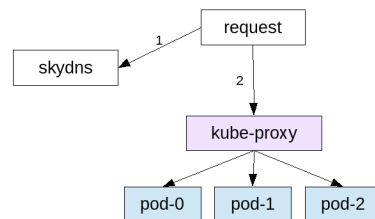


Figure 5.4: Scaling Pods with Kubernetes services

system specialists. In order to reach a PaaS abstraction and deploying real applications, there is needed another level on top of Kubernetes.

The 3rd version of OpenShift is a PaaS built on top of Kubernetes and Docker, providing a solution covers continuous delivery for modern containers and microservices world.

As shown in figure 5.5, OpenShift extends Kubernetes APIs with additional concepts:

- *route* configure a public DNS to point to a specific service;
- *template* providing a way to define an application as a set of components (pods, replication controllers, services and routes), in a similar way as happens with Docker Compose.

As shown in figure 5.6, Gasista Felice template is composed by a total of:

- 4 pods (variable);
- 4 replication controllers;
- 4 services;
- 1 route, *gf.befaircloud.me* that points to proxy service.

OpenShift stores the own objects in Etcd under `/openshift.io` key.

In order to create an instance of Gasista Felice, it needed running with `openshift/gasistafelice` script from the master.

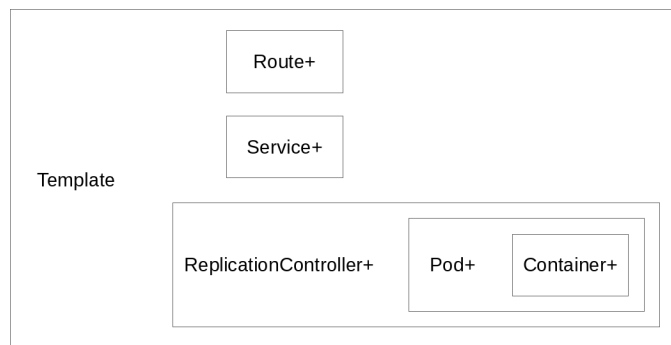


Figure 5.5: OpenShift adds Route and Template objects to Kubernetes

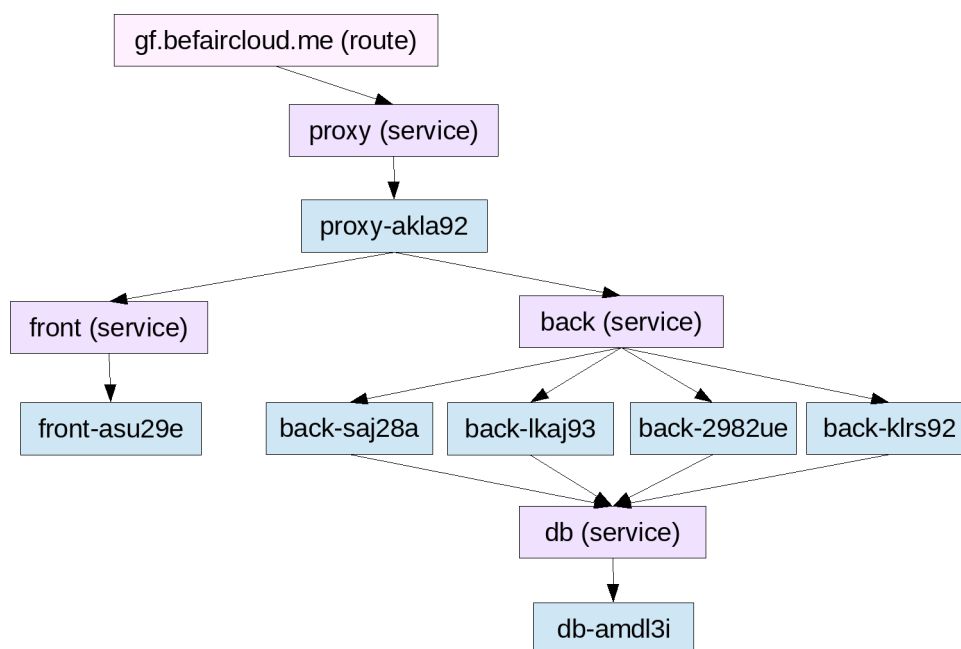


Figure 5.6: Gasista Felice template

In Figure 5.7 the OpenShift dashboard shows the main objects like replicationController, pods, services and routes in a clean UI.

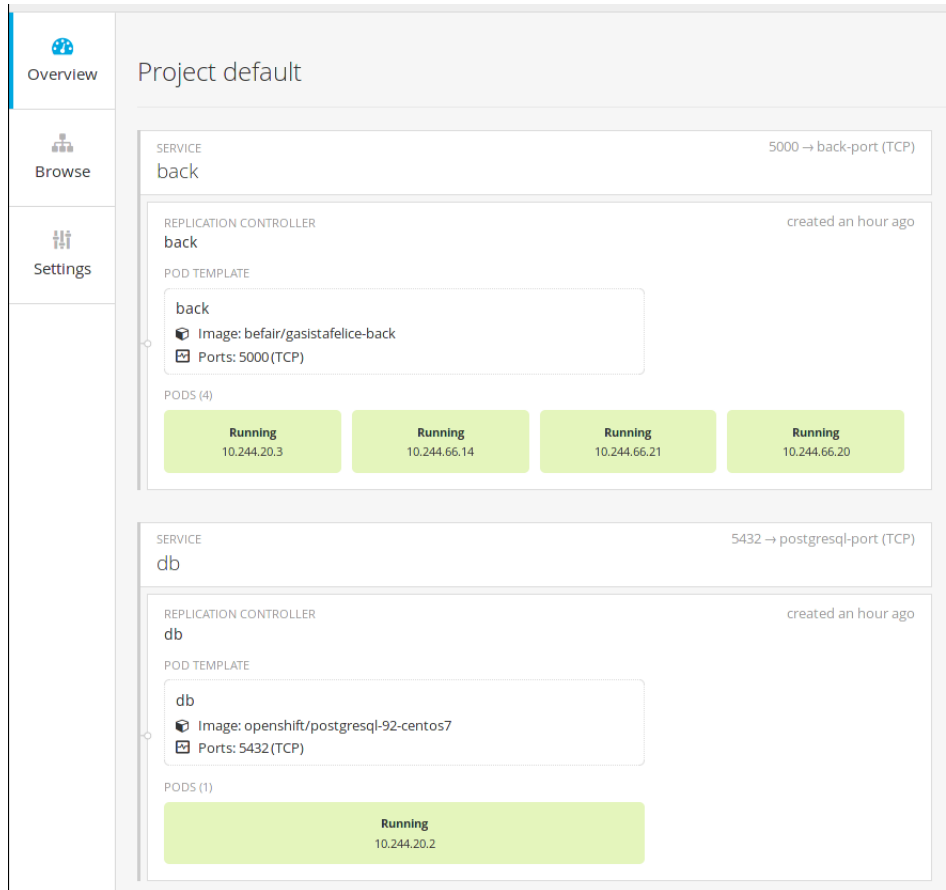


Figure 5.7: OpenShift dashboard

A small note derived on experience in deploying Gasista Felice on top of OpenShift: while in development the default `postgresql:9.4` image worked fine, on OpenShift it had some problems probably due to permissions. So it has been used a dedicated image for OpenShift, providing PostgreSQL 9.2, also compatible with Gasista Felice.

OpenShift comes with a set of components:

- an *HAProxy*⁹-based external router for external reachability;
- *skydns*¹⁰ server in order to respond to DNS queries for services.

All incoming requests come to HAProxy first, than reaches the application web server, like NGINX.

⁹<http://www.haproxy.org/>

¹⁰<https://github.com/skynetservices/skydns>

Future development will include a more extensively use of OpenShift features, from built-in container image *registry*, to concepts for enabling Continuous Delivery such as *build*, *imageStream* and *deploymentConfig*.

5.4 Benchmarks

Since Kubernetes provides the horizontal scaling of pods, that consist in adding components of the same size, will be first analyzed the Gasista Felice components from this perspective, then done some benchmarks.

The *database* is the unique component that stores data. As of other RDBMS, PostgreSQL by design is composed by a unique master, and for major scalability in reads, could be set up some slaves for future advances needs, or adding some Redis instances for caching.

The *backend* there is an uWSGI application server running 4 processes of Python-based Gasista Felice backend. This is the main pod on which it's possible act.

The *frontend* is mainly useful in development, but in production once it has generated the files, it could be cached from NGINX. In further optimization, this component will be deleted at all, and the static files will be served directly. In this case will be use 1 replica since there is no need of scaling this component.

NGINX represent the entry point of the application, so it manages caching of static content, demanding to the **front** and **back** non-cached requests. Since NGINX is generally enough fast thanks to asynchronous loops, then should not be necessary adding further pods for low to medium loads. In a small benchmark on <http://gf.befaircloud.me/> path, with 50 concurrent request for a total of 1000 requests, the result is: medium of 0.62s, 75.3 requests/s for a total of 13.2s.

The benchmark consist in stressing the backend, varying the number of pods.

The test consists in using *boom* for 50 concurrent requests for a total of 1000 requests, that consist in a *GET* at `/gasistafelice` path of the application, so a *GET* <http://gf.befaircloud.me/gasistafelice> in this case. The test could be run with `make test` command.

Even if this should be a simple request, instead involves several work and represent a simple but significant test case. The values registered consist in:

- number of pods, and related uWSGI worker processes (4 uWSGI workers/pod);
- *total time*: interval from the beginning of the first request, to the end of the last requests;
- *requests per second*: represent the medium of requests served per second;
- *medium*: medium of time in responding to a single request.

Table 5.1: Benchmark results

Backend pods/uWSGI workers	total time	reqs/s	medium
1/4	647s	1.34reqs/s	32.0s
2/8	410s	2.44reqs/s	19.8s
4/16	297s	3.34reqs/s	14.3s
6/24	202s	4.94reqs/s	9.71s
8/32	185s	5.37reqs/s	8.85s
10/40	175s	5.68reqs/s	8.35s
12/48	186s	5.27reqs/s	8.69s
16/64	194s	5.00reqs/s	9.09s

Table 5.1 shows the results as the time decreasing from 1 to 10 pods. Beyond that number there is no advantage, probably because of cluster resource limit.

Chapter 6

Monitoring and Logging

This chapter shows how to control, monitor, analyze, and visualize metrics about applications, doing some benchmark to test the scalability of the apps on this stack.

Monitoring the system resources and logging of applications should be a first-class task in IT Operations, because it's not possible to control what it's not possible to measure. Traditional tools are generally hard to configure and it needs to create a dedicated transport layer (with encryption/authentication) of various components between hosts. Luckily, in this situation it's possible to take advantage of the cluster primitives to make it an easier (and efficient) task.

While *InfluxDB* and *Grafana* are a popular choice respectively for storage and visualize resource metrics, in the logging world the most popular one is the *ELK* stack, where ELK stands for Elasticsearch¹ (storage), Logstash² (log processing) and Kibana³ (visualization). Since a guideline is the lightweight, instead of using an additional stack for logging, it has been reused both InfluxDB and Grafana, adding *Heka* for logs processing, so the stack could be called *IHG*: InfluxDB, Heka and Grafana.

In addition, Kubernetes' kubelet comes with built-in cAdvisor⁴ for resource monitoring, while Heapster⁵ is the main plug-in for resource cluster manager. At the end, Logspout (built-in in recent versions of Heka) will be used for gathering logs from all running containers.

Figure 6.1 shows an overview of the whole monitoring system.

The monitoring stack is deployed as a unique application, and includes **ReplicationControllers**, **Services** but also **Routes** for external reachability:

¹<https://www.elastic.co/products/elasticsearch>

²<https://www.elastic.co/products/logstash>

³<https://www.elastic.co/products/kibana>

⁴<https://github.com/google/cadvisor>

⁵<https://github.com/kubernetes/heapster>

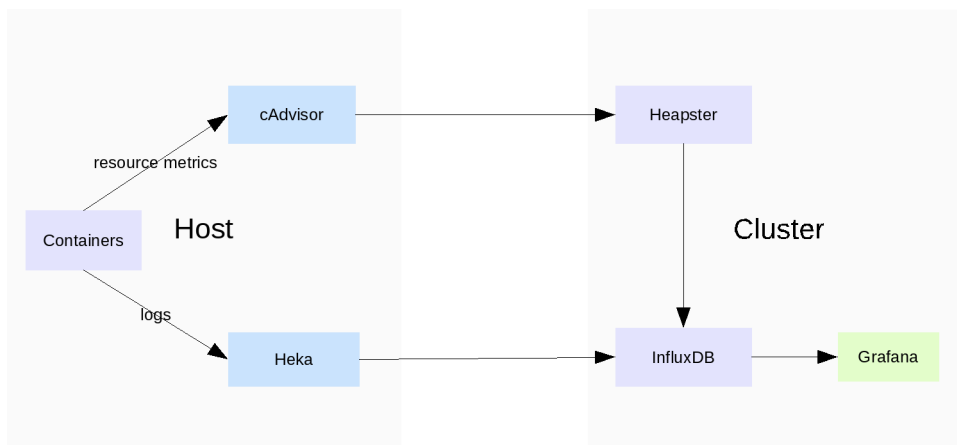


Figure 6.1: Overview of monitoring system

6.1 Time-Series Storage

InfluxDB is a distributed time-series database that fits well since it doesn't need a predefined schema for data, it's horizontally scalable in the NoSQL way, support native data replication and include a familiar SQL-like language, InfluxQL⁶.

A single metric of a resource could be stored as (in YAML format):

```
-
name: load_avg
columns: [ time, app, value ]
points: [[ 1400425342091, pod_name, 0.8 ]]
```

Instead a line of log could be stored as:

```
-
name: log_lines
columns: [ time, app, line ]
points: [[ 1400425947368, pod_name, "some useful log info" ]]
```

This is only a very basic example, of course additional columns could increase the usefulness of archived data.

In OpenShift, InfluxDB runs as component of monitoring application, and is composed by a `replicationController` and a `service`:

6.2 Resource Gathering

Kubernetes comes with built-in cAdvisor in Kubelet, for resource metrics gathering, such as CPU, RAM, I/O, network, etc.

⁶https://influxdb.com/docs/v0.8/api/query_language.html

Heapster is the Kubernetes cluster monitoring system, that gather metrics from nodes and send them to InfluxDB.

The exported metrics are:

- uptime: Number of milliseconds since the container was started;
- cpu/usage: Cumulative CPU usage on all cores;
- cpu/limit: CPU limit in millicores;
- memory/usage: Total memory usage;
- memory/working_set: Total working set usage. Working set is the memory being used and not easily dropped by the kernel;
- memory/limit: Memory limit;
- memory/page_faults: Number of page faults;
- memory/major_page_faults: Number of major page faults;
- network/rx: Cumulative number of bytes received over the network;
- network/rx_errors: Cumulative number of errors while receiving over the network;
- network/tx: Cumulative number of bytes sent over the network;
- network/tx_errors: Cumulative number of errors while sending over the network;
- filesystem/usage: Total number of bytes consumed on a filesystem;
- filesystem/limit: The total size of filesystem in bytes.

While the exported labels are:

- hostname: Hostname where the container ran;
- host_id: Identifier specific to a host. Set by cloud provider or user;
- container_name: User-provided name of the container or full container name for system containers;
- pod_name: The name of the pod;
- pod_id: The unique ID of the pod;
- pod_namespace: The namespace of the pod;
- namespace_id: The UID of namespace of the pod;

- labels: Comma-separated list of user-provided labels;
- resource_id: Identifier(s) specific to a metric.

Heapster is deployed as `replicationController`, a `service` and a `route`.

6.3 Log Processing

Heka is a framework developed by Mozilla for easy data collection and processing. Heka is modular and supports mainly 6 different type of plugins: inputs, splitters, decoders, filters, encoders and outputs. The first 3 are for gather and parse incoming data to *Heka Message* standard, the filters parse all these messages and the later 2 is for encoding and sending data somewhere else, for storing or further processing.

Heka permits to logs parsing, filtering, structuring them and send to InfluxDB. In this example will be used Gasista Felice's NGiNX component for generating logs and *boom* for doing a lot of requests.

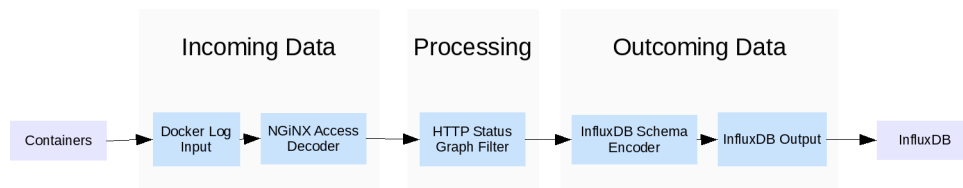


Figure 6.2: The Heka Data Flow

As shown in figure 6.2, the input part is composed by the *Docker Log Input*, build on top of *Logspout* that acquires stdout/stderr of running containers from the Docker Engine Unix socket, and *NGiNX access* and *error decoders*. Filtering consist of *HTTP Status Graph* that filter NGiNX access logs and generate an HTTP response code based graph. Finally the output part is composed by encoding in *InfluxDB schema* and sending data to *InfluxDB HTTP APIs*.

Heka is deployed on top of OpenShift as a simple `replicationController`.

6.4 Data Visualization

Grafana is a metrics dashboard and graph editor, born as fork of Kibana in the beginning of 2014.

Figure 6.3 shows structured logs from InfluxDB dashboard, that have been queried from Grafana dashboard.

The first 2 graphs in Grafana dashboard show data derived from logs processing, while the last 2 from Heapster:

```
# Show the network traffic
```


<div>  Databases Cluster Admins Cluster </div> <div> not influxdb.befaircloud.me:80 Disconnect </div>											
time	sequence_number	remote_user	http_referer	status	http_x_forwarded_for	body_bytes_sent	remote_addr	user_agent_os	user_agent_browser	user_agent_version	request
1440351354936	4220001	-	http://gt.befaircloud.me/	200	-	4504	172.17.0.9	Linux	Firefox	42	GET /gasistafelice/accounts/login/ HTTP/1.1
1440351354248	4210001	-	http://gt.befaircloud.me/	200	-	6105	172.17.0.9	Linux	Firefox	42	GET /components/order/order.html HTTP/1.1
1440351354111	4200001	-	http://gt.befaircloud.me/ui/css/style.css	200	-	23292	172.17.0.9	Linux	Firefox	42	GET /ui/fonts/glyphicons-halflings-regular.woff HTTP/1.1
1440351353933	4190001	-	http://gt.befaircloud.me/	200	-	3287	172.17.0.9	Linux	Firefox	42	GET /ui/js/ngDialog.min.js HTTP/1.1
1440351353932	4180001	-	http://gt.befaircloud.me/ui/css/style.css	404	-	2015	172.17.0.9	Linux	Firefox	42	GET /ui/fonts/glyphicons-halflings-regular.woff2 HTTP/1.1
1440351353984	4170001	-	http://gt.befaircloud.me/	200	-	8621	172.17.0.9	Linux	Firefox	42	GET /ui/img/user_pic.png HTTP/1.1
1440351353753	4160001	-	http://gt.befaircloud.me/	200	-	2924	172.17.0.9	Linux	Firefox	42	GET /ui/components/profile/profile.js HTTP/1.1
1440351353735	4150001	-	http://gt.befaircloud.me/	200	-	1116	172.17.0.9	Linux	Firefox	42	GET /ui/components/cash/cash.js HTTP/1.1
1440351353723	4140001	-	http://gt.befaircloud.me/	200	-	1255	172.17.0.9	Linux	Firefox	42	GET /ui/js/angular-locale_it-it.js HTTP/1.1
1440351353681	4130001	-	http://gt.befaircloud.me/	200	-	2386	172.17.0.9	Linux	Firefox	42	GET /ui/components/basket/basket.js HTTP/1.1
1440351353677	4120001	-	http://gt.befaircloud.me/	200	-	1020	172.17.0.9	Linux	Firefox	42	GET /ui/components/order/order.js HTTP/1.1
1440351353674	4110001	-	http://gt.befaircloud.me/	200	-	18532	172.17.0.9	Linux	Firefox	42	GET /ui/app.js HTTP/1.1
1440351353672	4100001	-	http://gt.befaircloud.me/	200	-	15317	172.17.0.9	Linux	Firefox	42	GET /ui/js/modernizr.js HTTP/1.1
1440351353305	4090001	-	http://gt.befaircloud.me/	200	-	14659	172.17.0.9	Linux	Firefox	42	GET /ui/bower_components/satellizer/satellizer.min.js HTTP/1.1
1440351353249	4080001	-	http://gt.befaircloud.me/	200	-	125410	172.17.0.9	Linux	Firefox	42	GET /ui/bower_components/leaflet/dist/leaflet.js HTTP/1.1
1440351353246	4070001	-	http://gt.befaircloud.me/	200	-	86022	172.17.0.9	Linux	Firefox	42	GET /ui/bower_components/angular-bootstrap/ui-bootstrap-tpls.min.js HTTP/1.1
1440351353199	4060001	-	http://gt.befaircloud.me/	200	-	66061	172.17.0.9	Linux	Firefox	42	GET /ui/bower_components/angular-bootstrap/ui-bootstrap.min.js HTTP/1.1
1440351353192	4050001	-	http://gt.befaircloud.me/	200	-	20511	172.17.0.9	Linux	Firefox	42	GET /ui/bower_components/angular-new-router/dist/router.es5.min.js HTTP/1.1
1440351353190	4040001	-	http://gt.befaircloud.me/	200	-	147059	172.17.0.9	Linux	Firefox	42	GET /ui/bower_components/angular/angular.min.js HTTP/1.1

Figure 6.3: InfluxDB logs dashboard

```

select mean(body_bytes_sent) from "nginx.access"

# Show the success and failure responses
select count(status) from "nginx.access" where status < 400
select count(status) from "nginx.access" where status >= 400

# Show the CPU load of backend
select derivative(value) from "cpu/usage_ns_cumulative" \
    where container_name="back"

# Show the memory usage of backend
select value from "memory/usage_bytes_gauge" \
    where container_name="back"

```

Figure 6.4 shows the graphical results of these queries.



Figure 6.4: Grafana dashboard

Grafana is deployed with a `replicationController`, a `Service` and a `Route` in order to be reached from the external.

Chapter 7

Conclusions

Containers and emerging ecosystem around them are questioning the whole IT industry about the role of the OS today. There are several new opportunities for radically enhance the development and deployment workflow, software architecture and application environments.

Docker is a new and quite immature runtime, but the future seems very prosperous. Kubernetes on the other side, has been adopted as scheduling and orchestration framework by other PaaS beyond OpenShift, like Deis, and from lower-level components, like Mesos¹.

7.1 Future Developments

At this point, there are a large variety of potential improvements, mainly about *high availability*[15].

Applications running on top of the platform that use traditional relational database such as PostgreSQL, could be configured for high availability also for data².

From the monitoring and logging point of view, an alert system could be feasible by an Heka alert encoder plug-in, or via a dedicated software like Sentry³ or Bosun⁴.

From the cluster perspective, there is the possibility of rolling update or auto-scaling of the cluster itself. In fact over time, the stack components should be updated, and the applications deployed on cluster could need globally more resources. A way to do this is monitoring the global load of the cluster, and instruct Terraform for varying the number of nodes accordingly to it.

¹<https://mesos.apache.org/>

²<https://www.compose.io/articles/high-availability-for-postgresql-batteries-not-included/>

³<https://getsentry.com/>

⁴<https://bosun.org/>

Beyond the single cluster, there is an emerging proposal for *cluster federation*, called Ubernetes⁵, that enables high-availability with geographically distributed Kubernetes clusters.

With that global distributed infrastructure, it's possible to build geographically distributed applications on top of databases like Consul⁶ for key-value storage or CockroachDB⁷ for more complex cases.

⁵<http://kubernetes.io/v1.0/docs/proposals/federation.html>

⁶<https://consul.io/>

⁷<http://www.cockroachlabs.com/>

Appendix A

Source code

A.1 Gasista Felice

A.1.1 Dockerfile of NGiNX base image

```
FROM nginx:1.9
```

```
MAINTAINER Antonio Esposito "kobe@befair.it"
```

```
RUN rm -rf /etc/nginx/conf.d/*
```

```
COPY nginx.conf /etc/nginx/nginx.conf
```

A.1.2 NGiNX base configuration file

```
user nginx;
```

```
worker_processes 1;
```

```
error_log stderr warn;
```

```
pid /var/run/nginx.pid;
```

```
events {
```

```
    worker_connections 1024;
```

```
}
```

```
http {
```

```
    include /etc/nginx/mime.types;
```

```
    default_type application/octet-stream;
```

```
    log_format main '$remote_addr - $remote_user [$msec] "$request" '
                    '$status $body_bytes_sent "$http_referer" '
                    '"$http_user_agent" "$http_x_forwarded_for"';
```

```
access_log /dev/stdout main;

sendfile      on;
#tcp_nopush   on;

keepalive_timeout 65;

#gzip on;

server_tokens      off;
server_name_in_redirect off;
port_in_redirect   off;

include /etc/nginx/conf.d/*.conf;
}
```

A.1.3 Proxy's Dockerfile

```
FROM kobe25/nginx:latest

MAINTAINER Antonio Esposito "kobe@befair.it"

COPY site.conf /etc/nginx/conf.d/site.conf
```

A.1.4 NGiNX configuration file

```
server {
    listen 8080 default_server;

    server_name _;
    root /code;

    charset utf-8;
    client_max_body_size 75M;
    client_body_timeout 600s;

    location = /favicon.ico {
        log_not_found off;
        access_log off;
    }

    location = /robots.txt {
        allow all;
```



```
    log_not_found off;
    access_log off;
}

location ~ /\. {
    deny all;
    log_not_found off;
    access_log off;
}

location = / {
    proxy_pass      http://front:5000/ui/index.html;
    expires max;
}

location /components/ {
    proxy_pass      http://front:5000/ui/components/;
    expires max;
}

location /ui/bower_components/ {
    proxy_pass      http://front:5000/libs/bower_components/;
    expires max;
}

location /ui/ {
    proxy_pass      http://front:5000;
    expires max;
}

location /static/ {
    include          uwsgi_params;
    uwsgi_pass       uwsgi://back:5000;
    expires max;
}

location /media/ {
    include          uwsgi_params;
    uwsgi_pass       uwsgi://back:5000;
}

location /api/ {
    gzip             on;
    gzip_types       application/json;
```

```
gzip_min_length 1000;

include uwsgi_params;
uwsgi_pass uwsgi://back:5000;
}

location /gasistafelice/ {
    include uwsgi_params;
    uwsgi_pass uwsgi://back:5000;
}

location / {
    return 404;
}
}
```

A.1.5 Dockerfile of uWSGI/Python2 base image

FROM python:2.7.7

MAINTAINER Antonio Esposito "kobe@befair.it"

```
ENV DEBIAN_FRONTEND noninteractive

ENV PYTHONUNBUFFERED 1
ENV PYTHONPATH /code:/usr/local/lib/python2.7/site-packages
ENV UWSGI_CALLABLE app

ENV UWSGI_MASTER true
ENV UWSGI_MASTER_AS_ROOT true
ENV UWSGI_UID app
ENV UWSGI_GID app
ENV UWSGI_UWSGI_SOCKET 0.0.0.0:5000
ENV UWSGI_NO_ORPHANS true
ENV UWSGI_VACUUM true
ENV UWSGI_LOG_DATE true

ENV UWSGI_WORKERS 4
ENV UWSGI_THREADS 1
ENV UWSGI_ENABLE_THREADS true
ENV UWSGI_BUFFER_SIZE 65536
ENV UWSGI_MAX_REQUESTS 128
ENV UWSGI_HARAKIRI 120
ENV UWSGI_HARAKIRI_VERBOSE true
```

```

ENV UWSGI_THUNDER_LOCK      true

ENV PGDATABASE               app
ENV PGUSER                   app
ENV PGPASSWORD               app
ENV PGHOST                   db
ENV PGPORT                   5432

RUN groupadd -r app && \
    useradd -r -g app -d /code app

RUN apt update && \
    apt install -y \
        build-essential \
        python-dev \
        python-setuptools && \
    rm -rf /var/lib/apt/lists/*
RUN pip install \
    'uWSGI >=2.0, <2.1'

EXPOSE 5000
CMD ["uwsgi"]

```

A.1.6 Backend's Dockerfile

```

FROM kobe25/uwsgi-python2:latest

MAINTAINER Antonio Esposito "kobe@befair.it"

ENV LC_ALL                   it_IT.UTF-8
ENV LANG                     it_IT.UTF-8
ENV LANGUAGE                 it_IT.UTF-8

ENV PYTHONPATH
    /code:/code/gasistafelice:/usr/local/lib/python2.7/site-packages
ENV UWSGI_CHDIR              /code/gasistafelice
ENV UWSGI_WSGI_FILE          /code/gasistafelice/gf/wsgi.py
ENV DJANGO_SETTINGS_MODULE   gf.settings
ENV UWSGI_STATIC_MAP         /static=/code/gasistafelice/static
ENV UWSGI_STATIC_SAFE
    /usr/local/lib/python2.7/site-packages/django/contrib/admin/static/admin

COPY deps/debian /code/gasistafelice/deps/debian
RUN apt update && \

```

```
apt install -y $(cat /code/gasistafelice/deps/debian) && \  
rm -rf /var/lib/apt/lists/*
```

```
COPY deps/locale.gen /etc/locale.gen  
RUN locale-gen
```

```
COPY deps/ /code/gasistafelice/deps/  
RUN pip install -r /code/gasistafelice/deps/dev.txt
```

```
COPY ./ /code/gasistafelice/  
WORKDIR /code/gasistafelice/
```

A.1.7 Frontend's Dockerfile

```
FROM iojs:2.5
```

```
MAINTAINER Antonio Esposito "kobe@befair.it"
```

```
COPY deps/npm /code/ui/deps/npm  
RUN npm install -g $(cat /code/ui/deps/npm)
```

```
COPY ./bower.json /code/libs/bower.json  
RUN cd /code/libs/ && bower install --allow-root
```

```
EXPOSE 5000
```

```
COPY ./ /code/ui/  
WORKDIR /code/ui/
```

```
CMD ["harp", "server", "-i", "0.0.0.0", "-p", "5000", "/code"]
```

A.1.8 Docker Compose file

```
proxy:  
  image: befair/gasistafelice-proxy:latest  
  volumes:  
    - ./proxy/site.conf.dev:/etc/nginx/conf.d/site.conf:ro  
  ports:  
    - '127.0.0.1:8080:8080'  
  links:  
    - front  
    - back
```

```
front:
```

```
image: befair/gasistafelice-front:latest
volumes:
  - ./ui:/code/ui:rw

back:
  image: befair/gasistafelice-back:latest
  volumes:
    - ./gasistafelice:/code/gasistafelice:ro
    - ./gasistafelice/fixtures:/code/gasistafelice/fixtures:rw
    - /tmp/gf_tracebacker:/tmp/tracebacker:rw
    - /tmp/gf_profiling:/tmp/profiling:rw
  ports:
    - '127.0.0.1:7000:7000'
  links:
    - db
  env_file: ./compose/settings.env

db:
  image: postgres:9.4
  env_file: ./compose/settings.env
```

A.1.9 OpenShift template

```
apiVersion: v1
kind: Template
metadata:
  name: gasistafelice
  annotations:
    description: >
      Gasista Felice is the platform for DES Macerata project
    tags: app, gas, des
labels:
  app: gf
  application: gasistafelice
parameters:
-
  name: ENV
  description: dev/stage/prod environment
  value: prod
-
  name: SERVER_NAME
  description: server name
-
  name: POSTGRES_PASSWORD
```

```
description: Password used for DB authentication
generate: expression
from: '[\w]{12}'
objects:
-
  apiVersion: v1
  kind: ReplicationController
  metadata:
    name: db
    labels:
      name: db
  spec:
    replicas: 1
    selector:
      name: db
    template:
      metadata:
        labels:
          name: db
      spec:
        volumes:
        -
          name: postgres-ps
          emptyDir: {}
        containers:
        -
          name: db
          image: openshift/postgresql-92-centos7
          env:
          -
            name: POSTGRESQL_USER
            value: app
          -
            name: POSTGRESQL_PASSWORD
            value: ${POSTGRES_PASSWORD}
          -
            name: POSTGRESQL_DATABASE
            value: app
          -
            name: POSTGRESQL_ADMIN_PASSWORD
            value: ${POSTGRES_PASSWORD}
        ports:
        -
          name: postgresql-port
```

```
        containerPort: 5432
      volumeMounts:
      -
        name: postgres-ps
        mountPath: /var/lib/pgsql/data
        readOnly: false
-
  apiVersion: v1
  kind: Service
  metadata:
    name: db
    labels:
      name: db
  spec:
    ports:
    -
      port: 5432
      targetPort: postgresql-port
    selector:
      name: db
-
  apiVersion: v1
  kind: ReplicationController
  metadata:
    name: back
  spec:
    replicas: 1
    selector:
      name: back
    template:
      metadata:
        labels:
          name: back
      spec:
        containers:
        -
          name: back
          image: befair/gasistafelice-back
          ports:
          -
            name: back-port
            containerPort: 5000
        env:
        -
```

```

        name: APP_ENV
        value: ${ENV}
      -
        name: APP_SERVER_NAME
        value: ${SERVER_NAME}
      -
        name: POSTGRES_USER
        value: app
      -
        name: POSTGRES_PASSWORD
        value: ${POSTGRES_PASSWORD}
      -
        name: POSTGRESQL_USER
        value: app
      -
        name: POSTGRESQL_PASSWORD
        value: ${POSTGRES_PASSWORD}
      -
        name: POSTGRESQL_DATABASE
        value: app
      -
        name: POSTGRESQL_ADMIN_PASSWORD
        value: ${POSTGRES_PASSWORD}
      -
        name: PGUSER
        value: app
      -
        name: PGPASSWORD
        value: ${POSTGRES_PASSWORD}
    -
      apiVersion: v1
      kind: Service
      metadata:
        name: back
      spec:
        ports:
          -
            port: 5000
            targetPort: back-port
        selector:
          name: back
    -
      apiVersion: v1
      kind: ReplicationController

```



```
metadata:
  name: front
spec:
  replicas: 1
  selector:
    name: front
  template:
    metadata:
      labels:
        name: front
    spec:
      containers:
      -
        name: front
        image: befair/gasistafelice-front
        ports:
        -
          name: front-port
          containerPort: 5000
-
  apiVersion: v1
  kind: Service
  metadata:
    name: front
  spec:
    ports:
    -
      port: 5000
      targetPort: front-port
    selector:
      name: front
-
  apiVersion: v1
  kind: ReplicationController
  metadata:
    name: proxy
  spec:
    replicas: 1
    selector:
      name: nginx
  template:
    metadata:
      labels:
        name: nginx
```

```

spec:
  volumes:
  -
    name: nginx-cache
    emptyDir: {}
  -
    name: nginx-run
    emptyDir: {}
  containers:
  -
    name: proxy
    image: befair/gasistafelice-proxy
    securityContext:
      runAsUser: 0
      privileged: true
    ports:
    -
      name: insecure-port
      containerPort: 8080
    volumeMounts:
    -
      name: nginx-cache
      mountPath: /var/cache/nginx
      readOnly: false
    -
      name: nginx-run
      mountPath: /var/run
      readOnly: false
-
  apiVersion: v1
  kind: Service
  metadata:
    name: proxy
  spec:
    selector:
      name: nginx
    ports:
    -
      port: 8080
      targetPort: insecure-port
-
  apiVersion: v1
  kind: Route
  metadata:

```

```

    name: route
spec:
  host: ${SERVER_NAME}
  to:
    kind: Service
    name: proxy

```

A.2 IaaS and PaaS bootstrapping

A.2.1 Makefile

```

NETENV_V := 1.0.0
TF_V := 0.6.3
OS_V := 1.0.5
OS_COMMIT := 96963b6

BINPATH := ${GOPATH}/bin
export PATH := ${GOPATH}/bin:${PATH}
TF_URL := https://dl.bintray.com/mitchellh/terraform/\
    terraform_${TF_V}_linux_amd64.zip
OS_URL := https://github.com/openshift/origin/releases/download/\
    v${OS_V}/openshift-origin-v${OS_V}-${OS_COMMIT}-linux-amd64.tar.gz
NETENV_URL := https://github.com/kelseyhightower/\
    setup-network-environment/releases/download/\
    v${NETENV_V}/setup-network-environment

help:
    @cat Makefile

install:
    @mkdir -p .cache
    @go get -u github.com/dbohdan/remarshal
    @go get -u github.com/rakyll/boom
    @go install github.com/dbohdan/remarshal
    @go install github.com/rakyll/boom
    @curl -L -o .cache/setup-network-environment \
        -z .cache/setup-network-environment ${NETENV_URL}
    @curl -L -o .cache/terraform.zip \
        -z .cache/terraform.zip ${TF_URL}
    @curl -L -o .cache/openshift-origin.tar.gz \
        -z .cache/openshift-origin.tar.gz ${OS_URL}
    @unzip -o .cache/terraform.zip -d ${BINPATH}/
    @tar -xf .cache/openshift-origin.tar.gz -C .cache/

```

```

@ln -sf      .cache/openshift ${BINPATH}/openshift
@ln -sf      ${BINPATH}/openshift ${BINPATH}/oc
@ln -sf      ${BINPATH}/openshift ${BINPATH}/oadm

clean soft-clean:
    @rm -rf \
        terraform.* \
        .cache/*.{gz,zip,toml} \
        .cache/master/ \
        .cache/id*

full-clean: clean
    @rm -rf \
        ${BINPATH}/terraform* \
        .cache/

compile:
    @mkdir -p .cache
    @remarshal \
        -if yaml -i terraform/digitalocean.yaml \
        -of json -o terraform.tf.json
    @remarshal \
        -if yaml -i vars.yaml \
        -of json -o terraform.tfvars

up: clean compile
    @ssh-keygen -b 4096 -t rsa -f .cache/id -N ''
    @tar -czf .cache/pkg.tar.gz \
        openshift/ \
        .cache/{setup-network-environment,openshift}
    @terraform plan -out terraform.tfplan
    @terraform apply terraform.tfplan

delete destroy: compile
    @terraform plan -destroy
    @terraform destroy

infrastructure-graph:
    @terraform graph | dot -Tsvg > graph.svg

test:
    @./benchmark

```

A.2.2 Terraform template for Digital Ocean

```
variable:
  BASE_DOMAIN: { default: befaircloud.me }
  NODES: { default: 2 }
  OMASTER: { default: /opt/bin/openshift.local.config/master }
  ONODE: { default: /opt/bin/openshift.local.config/node }
  # Provider specific
  DO_TOKEN: {}
  DO_SSH_ID: { default: core }
  DO_IMAGE: { default: coreos-beta }
  DO_REGION: { default: fra1 }
  DO_MASTER_SIZE: { default: 1gb }
  DO_NODE_SIZE: { default: 2gb }

provider:
  digitalocean:
    token: ${var.DO_TOKEN}

resource:
  template_file:
    master_conf:
      filename: terraform/master.yaml
      vars:
        HOME: /home/core
        OMASTER: ${var.OMASTER}
        SSH_PUBLIC_KEY: ${file(".cache/id.pub")}
        BASE_DOMAIN: ${var.BASE_DOMAIN}
        NODES: ${var.NODES}

    node_conf:
      filename: terraform/node.yaml
      vars:
        HOME: /home/core
        OMASTER: ${var.OMASTER}
        ONODE: ${var.ONODE}
        MASTER_IP: >
          ${digitalocean_droplet.master.ipv4_address_private}
        SSH_PUBLIC_KEY: ${file(".cache/id.pub")}
      depends_on:
        - digitalocean_droplet.master

  digitalocean_ssh_key:
    superuser:
```

```

    name: ${var.DO_SSH_ID}
    public_key: ${file(".cache/id.pub")}
digitalocean_droplet:
  master:
    image: ${var.DO_IMAGE}
    name: master
    region: ${var.DO_REGION}
    size: ${var.DO_MASTER_SIZE}
    ipv6: true
    private_networking: true
    user_data: ${template_file.master_conf.rendered}
    ssh_keys:
      - ${digitalocean_ssh_key.superuser.fingerprint}
    depends_on:
      - template_file.master_conf
    connection:
      user: core
      key_file: .cache/id
    provisioner:
      file:
        source: .cache/pkg.tar.gz
        destination: /home/core/pkg.tar.gz
      remote-exec:
        inline:
          - sudo mkdir -p /opt/bin/
          - tar -xf /home/core/pkg.tar.gz -C /home/core
          - sudo mv /home/core/.cache/setup-network-environment \
            /opt/bin/
          - sudo mv /home/core/.cache/openshift /opt/bin/
          - mv /home/core/openshift/* /home/core/
          - rm -rf /home/core/{pkg.tar.gz,.cache,openshift}

  nodes:
    count: ${var.NODES}
    image: ${var.DO_IMAGE}
    name: node-${count.index}
    region: ${var.DO_REGION}
    size: ${var.DO_NODE_SIZE}
    ipv6: true
    private_networking: true
    user_data: ${template_file.node_conf.rendered}
    ssh_keys:
      - ${digitalocean_ssh_key.superuser.fingerprint}
    depends_on:

```

```

- digitalocean_droplet.master
- template_file.node_conf
connection:
  user: core
  key_file: .cache/id
provisioner:
  file:
    source: .cache/pkg.tar.gz
    destination: /home/core/pkg.tar.gz
remote-exec:
  inline:
    - sudo mkdir -p /opt/bin/
    - tar -xf /home/core/pkg.tar.gz -C /home/core
    - sudo mv /home/core/.cache/setup-network-environment \
      /opt/bin/
    - sudo mv /home/core/.cache/openshift /opt/bin/
    - rm -rf /home/core/{pkg.tar.gz,.cache,openshift}

digitalocean_domain:
  cloud:
    name: ${var.BASE_DOMAIN}
    ip_address: ${digitalocean_droplet.nodes.0.ipv4_address}
    depends_on:
      - digitalocean_droplet.nodes

digitalocean_record:
  cloud_ipv4:
    domain: ${digitalocean_domain.cloud.name}
    type: A
    name: '*'
    value: ${digitalocean_droplet.nodes.0.ipv4_address}

output:
  step1:
    value: |
      scp -r core@${digitalocean_droplet.master.ipv4_address}:/master .cache/; \
      scp -r .cache/master core@${digitalocean_droplet.nodes.0.ipv4_address}:/; \
      scp -r .cache/master core@${digitalocean_droplet.nodes.1.ipv4_address}:/
  step2:
    value: |
      log to master -- ssh core@${digitalocean_droplet.master.ipv4_address}
      log to node-0 -- ssh core@${digitalocean_droplet.nodes.0.ipv4_address}
      log to node-1 -- ssh core@${digitalocean_droplet.nodes.1.ipv4_address}
      dashboard -- https://${digitalocean_droplet.master.ipv4_address}:8443

```

```
entrypoint -- http://${var.BASE_DOMAIN}
```

A.2.3 Master Cloud Config template

```
#cloud-config
```

```
---
```

```
hostname: master
```

```
ssh_authorized_keys:
```

```
- ${SSH_PUBLIC_KEY}
```

```
coreos:
```

```
  update:
```

```
    group: alpha
```

```
    reboot-strategy: off
```

```
  etcd2:
```

```
    name: master
```

```
    listen-client-urls: http://0.0.0.0:2379
```

```
    advertise-client-urls: http://$private_ipv4:2379
```

```
    initial-cluster-token: k8s_etcd
```

```
    listen-peer-urls: http://$private_ipv4:2380
```

```
    initial-advertise-peer-urls: http://$private_ipv4:2380
```

```
    initial-cluster: master=http://$private_ipv4:2380
```

```
    initial-cluster-state: new
```

```
units:
```

```
-
```

```
  name: bootstrap-master.service
```

```
  command: start
```

```
  content: |
```

```
    [Unit]
```

```
    Description=Bootstrap the OpenShift master
```

```
    Requires=network-online.target
```

```
    After=network-online.target
```

```
    [Service]
```

```
    # Setup network environment
```

```
    ExecStartPre=/opt/bin/wufae \
```

```
        /opt/bin/setup-network-environment
```

```
    ExecStartPre=/usr/bin/chown root:root \
```

```
        /opt/bin/setup-network-environment
```

```
    ExecStartPre=/usr/bin/chmod +x \
```

```
        /opt/bin/setup-network-environment
```

```
    ExecStartPre=/opt/bin/setup-network-environment
```

```
    # Generate OpenShift master configurations
```

```
    ExecStartPre=/opt/bin/wufae /opt/bin/openshift
```



```

ExecStartPre=/usr/bin/chown root:root /opt/bin/openshift
ExecStartPre=/usr/bin/chmod +x /opt/bin/openshift
ExecStartPre=/usr/bin/ln -sf /opt/bin/openshift \
    /opt/bin/oadm
ExecStartPre=/usr/bin/ln -sf /opt/bin/openshift \
    /opt/bin/oc
ExecStart=/opt/bin/openshift start master \
--write-config=${OMASTER}/ \
--listen=https://0.0.0.0:8443 \
--master=https://$private_ipv4:8443 \
--public-master=https://$public_ipv4:8443 \
--dns=tcp://$private_ipv4:53 \
--etcd=http://$private_ipv4:2379 \
--network-cidr='10.1.0.0/16'
# Copy files for nodes
ExecStartPost=/usr/bin/chmod +r ${OMASTER}/admin.kubeconfig
ExecStartPost=/usr/bin/chmod +r \
    ${OMASTER}/openshift-registry.kubeconfig
ExecStartPost=/usr/bin/sleep 5
ExecStartPost=/usr/bin/cp -R ${OMASTER} ${HOME}/
ExecStartPost=/usr/bin/chown -R core:core ${HOME}/master/
RemainAfterExit=yes
Type=oneshot
-
name: flanneld.service
command: start
drop-ins:
- name: 50-network-config.conf
  content: |
    [Unit]
    Requires=etcd2.service
    [Service]
    ExecStartPre=/usr/bin/etcdctl set /coreos.com/network/config \
        '{"Network":"10.244.0.0/16", "Backend": {"Type": "vxlan"}}'
-
name: docker.service
command: start
-
name: openshift-master.service
command: start
content: |
    [Unit]
    Description=OpenShift Master
    Documentation=https://github.com/openshift/origin

```

```

Requires=bootstrap-master.service etcd2.service
After=bootstrap-master.service etcd2.service

[Service]
EnvironmentFile=/etc/network-environment
ExecStartPre=/opt/bin/wupiao 127.0.0.1:2379/v2/machines
ExecStart=/opt/bin/openshift start master \
--config=${OMASTER}/master-config.yaml
Restart=always
RestartSec=10
write-files:
-
  path: /etc/conf.d/nfs
  permissions: '0644'
  content: |
    OPTS_RPC_MOUNTD=""
-
  path: /opt/bin/wupiao
  permissions: '0755'
  content: |
    #!/bin/bash
    # [w]ait [u]ntil [p]ort [i]s [a]ctually [o]pen
    [ -n "$1" ] && \
      until curl -o /dev/null -sIf http://$$1; do \
        sleep 1 && echo -n .;
      done;
    exit $?
-
  path: /opt/bin/wufae
  permissions: '0755'
  content: |
    #!/bin/bash
    # [w]ait [u]ntil [f]ile [a]ctually [e]xists
    while [ ! -f "$1" ]; do
      sleep 1 && echo -n .; done
    sleep 5
-
  path: /home/core/.bashrc
  permissions: '0644'
  owner: core
  content: |
    if [[ $- != *i* ]]; then
      return; fi

```

```

export OMASTER=${OMASTER}
export KUBECONFIG=${OMASTER}/admin.kubeconfig
export CURL_CA_BUNDLE=${OMASTER}/ca.crt
export BASE_DOMAIN=${BASE_DOMAIN}
export NODES=${NODES}

```

A.2.4 Node Cloud Config template

```

#cloud-config

---
hostname: node
ssh_authorized_keys:
- ${SSH_PUBLIC_KEY}
coreos:
  update:
    group: alpha
    reboot-strategy: off
  etcd2:
    listen-client-urls: http://0.0.0.0:2379
    advertise-client-urls: http://0.0.0.0:2379
    initial-cluster: master=http://${MASTER_IP}:2380
    proxy: on
  units:
  -
    name: bootstrap-node.service
    command: start
    content: |
      [Unit]
      Description=Bootstrap the OpenShift node
      Requires=network-online.target
      After=network-online.target

      [Service]
      # Setup network environment
      ExecStartPre=/opt/bin/wufae \
        /opt/bin/setup-network-environment
      ExecStartPre=/usr/bin/chown root:root \
        /opt/bin/setup-network-environment
      ExecStartPre=/usr/bin/chmod +x \
        /opt/bin/setup-network-environment
      ExecStart=/opt/bin/setup-network-environment
      # Generate OpenShift node configurations
      WorkingDirectory=/opt/bin

```

```

ExecStartPre=/usr/bin/mkdir -p ${ONODE}
ExecStartPre=/opt/bin/wufae /opt/bin/openshift
ExecStartPre=/usr/bin/chown root:root /opt/bin/openshift
ExecStartPre=/usr/bin/chmod +x /opt/bin/openshift
ExecStartPre=/usr/bin/ln -sf /opt/bin/openshift \
    /opt/bin/oadm
ExecStartPre=/usr/bin/ln -sf /opt/bin/openshift \
    /opt/bin/oc
# wait for master config
ExecStartPre=/opt/bin/wufae ${HOME}/master/master-config.yaml
ExecStartPre=/usr/bin/cp -R ${HOME}/master/ ${OMASTER}/
ExecStartPre=/usr/bin/chown -R root:root ${OMASTER}/
ExecStartPre=/opt/bin/oadm create-node-config \
--master=https://${MASTER_IP}:8443 \
--dns-ip=${MASTER_IP} \
--node-dir=${ONODE} \
--node=$private_ipv4 \
--hostnames=$private_ipv4 \
--listen=https://0.0.0.0:10250 \
--volume-dir=/var/volumes
# wait for kubernetes master to be up and ready
ExecStartPre=/opt/bin/wupiao ${MASTER_IP} 8443
ExecStart=/usr/bin/echo 'Ready for starting OpenShift Node!'
RemainAfterExit=yes
Type=oneshot
-
name: flanneld.service
command: start
drop-ins:
- name: 50-network-config.conf
  content: |
    [Unit]
    Requires=etcd2.service
    [Service]
    ExecStartPre=/usr/bin/etcdctl set /coreos.com/network/config \
        '{"Network":"10.244.0.0/16", "Backend": {"Type": "vxlan"}}'
-
name: docker.service
command: start
-
name: openshift-node.service
command: start
content: |
    [Unit]

```

```

Description=OpenShift Node
Documentation=https://github.com/openshift/origin
Requires=bootstrap-node.service etcd2.service docker.service
After=bootstrap-node.service etcd2.service docker.service

[Service]
EnvironmentFile=/etc/network-environment
WorkingDirectory=/opt/bin
ExecStart=/opt/bin/openshift start node \
--config=${ONODE}/node-config.yaml
Restart=always
RestartSec=10
write-files:
-
  path: /opt/bin/wupiao
  permissions: '0755'
  content: |
    #!/bin/bash
    # [w]ait [u]ntil [p]ort [i]s [a]ctually [o]pen
    [ -n "$1" ] && [ -n "$2" ] && while ! curl --output /dev/null \
      --silent --head --fail \
      http://${1}:${2}; do sleep 1 && echo -n .; done;
    exit $?
-
  path: /opt/bin/wufae
  permissions: '0755'
  content: |
    #!/bin/bash
    # [w]ait [u]ntil [f]ile [a]ctually [e]xists
    while [ ! -f "$1" ]; do
      sleep 1 && echo -n .; done
    sleep 5

```

A.3 Monitoring Stack

A.3.1 OpenShift template

```

apiVersion: v1
kind: Template
metadata:
  name: monitoring
  annotations:
    description: Monitoring stack for OpenShift cluster
    tags: monitoring,influxdb,grafana,heka

```

```
labels:
  app: monit
  application: monitoring
parameters:
-
  name: BASE_DOMAIN
  description: server name
-
  name: NODES
  description: number of nodes
-
  name: INFLUXDB_PASSWORD
  description: Password used for DB authentication
  generate: expression
  from: '[\w]{12}'
objects:
-
  apiVersion: v1
  kind: Service
  metadata:
    name: influxdb
    labels:
      component: influxdb
  spec:
    selector:
      component: influxdb
    ports:
      -
        port: 80
        targetPort: api
-
  apiVersion: v1
  kind: Service
  metadata:
    name: influxdb-ui
    labels:
      component: influxdb
  spec:
    selector:
      component: influxdb
    ports:
      -
        port: 80
        targetPort: ui
```

```
-
  apiVersion: v1
  kind: ReplicationController
  metadata:
    name: influxdb
    labels:
      component: influxdb
  spec:
    replicas: 1
    selector:
      component: influxdb
    template:
      metadata:
        labels:
          component: influxdb
      spec:
        volumes:
          -
            name: influxdb-data
            emptyDir: {}
        containers:
          -
            name: influxdb
            image: tutum/influxdb:0.8.8
            volumeMounts:
              -
                name: influxdb-data
                mountPath: /data
            ports:
              -
                name: ui
                containerPort: 8083
              -
                name: api
                containerPort: 8086
              -
                name: raft
                containerPort: 8090
              -
                name: protobuf
                containerPort: 8099
        env:
          -
            name: ADMIN_USER
```

```

        value: root
      -
        name: INFLUXDB_INIT_PWD
        value: root
      -
        name: PRE_CREATE_DB
        value: logs;k8s
-
  apiVersion: v1
  kind: Service
  metadata:
    name: heapster
    labels:
      component: heapster
  spec:
    selector:
      component: heapster
    ports:
      -
        port: 80
        targetPort: ui
-
  apiVersion: v1
  kind: ReplicationController
  metadata:
    name: heapster
    labels:
      component: heapster
  spec:
    replicas: 1
    selector:
      component: heapster
    template:
      metadata:
        labels:
          component: heapster
      spec:
        serviceAccount: heapster
        containers:
          -
            name: heapster
            image: kubernetes/heapster:v0.17.0
            args:
              - -port=8082

```



```
- -source=kubernetes:\
  https://openshift.default.svc.cluster.local?\
  auth=&insecure=true&useServiceAccount=true
- -sink=influxdb:http://influxdb.default.svc.cluster.local
- -logtostderr=true
ports:
-
  name: ui
  containerPort: 8082
-
apiVersion: v1
kind: Service
metadata:
  name: grafana
  labels:
    component: grafana
spec:
  selector:
    component: grafana
  ports:
  -
    port: 80
    targetPort: http
-
apiVersion: v1
kind: ReplicationController
metadata:
  name: grafana
  labels:
    component: grafana
spec:
  replicas: 1
  selector:
    component: grafana
  template:
    metadata:
      labels:
        component: grafana
    spec:
      containers:
      -
        name: grafana
        image: kobe25/grafana:latest
        ports:
```

```

-
  name: http
  containerPort: 3000
env:
-
  name: GF_SERVER_ROOT_URL
  value: http://grafana.${BASE_DOMAIN}
-
  name: GF_SECURITY_ADMIN_PASSWORD
  value: admin
-
apiVersion: v1
kind: ReplicationController
metadata:
  name: heka
  labels:
    component: heka
spec:
  replicas: 2
  selector:
    component: heka
  template:
    metadata:
      labels:
        component: heka
    spec:
      volumes:
      -
        name: heka-cache
        emptyDir: {}
      -
        name: docker-socket
        hostPath:
          path: /var/run/docker.sock
    containers:
    -
      name: heka
      image: kobe25/heka:latest
      volumeMounts:
      -
        name: heka-cache
        mountPath: /var/cache/hekad
      -
        name: docker-socket

```

```
        mountPath: /var/run/docker.sock
      ports:
      -
        name: http
        containerPort: 4352
-
  apiVersion: v1
  kind: Route
  metadata:
    name: heapster
  spec:
    host: heapster.${BASE_DOMAIN}
    to:
      kind: Service
      name: heapster
-
  apiVersion: v1
  kind: Route
  metadata:
    name: influxdb-ui
  spec:
    host: influxdb-ui.${BASE_DOMAIN}
    to:
      kind: Service
      name: influxdb-ui
-
  apiVersion: v1
  kind: Route
  metadata:
    name: influxdb
  spec:
    host: influxdb.${BASE_DOMAIN}
    to:
      kind: Service
      name: influxdb
-
  apiVersion: v1
  kind: Route
  metadata:
    name: grafana
  spec:
    host: grafana.${BASE_DOMAIN}
    to:
      kind: Service
```

```
name: grafana
```

A.3.2 Heka's configuration file

```
[DockerLogInput]
  endpoint = 'unix:///var/run/docker.sock'
  decoder = 'MultiDecoder'

[MultiDecoder]
  cascade_strategy = 'first-wins'
  subs = [
    'NginxAccess',
    'NginxError',
    'PayloadRegexDecoder',
  ]

[NginxAccess]
  type = 'SandboxDecoder'
  filename = 'lua_decoders/nginx_access.lua'
  [NginxAccess.config]
    type = 'nginx.access'
    user_agent_transform = true
    log_format = "$remote_addr - $remote_user [$msec] \
      \"$request\" $status $body_bytes_sent \"$http_referer\" \
      \"$http_user_agent\" \"$http_x_forwarded_for\""

[NginxError]
  type = 'SandboxDecoder'
  filename = 'lua_decoders/nginx_error.lua'
  [NginxError.config]
    type = 'nginx.error'

[PayloadRegexDecoder]
  match_regex = '(...)'
  [PayloadRegexDecoder.message_fields]
    Type = "LogNotManaged"

[HttpStatus]
  type = 'SandboxFilter'
  filename = 'lua_filters/http_status.lua'
  message_matcher = 'Type == "nginx.access"'

[InfluxSchema]
  type = 'SandboxEncoder'
```

```
filename = 'lua_encoders/schema_influx.lua'
[InfluxSchema.config]
    exclude_base_fields = true
    series = "%{Type}"

[InfluxOutput]
    type = 'HttpOutput'
    encoder = 'InfluxSchema'
    message_matcher = 'Type == "nginx.access"'
    address = 'http://influxdb.default.svc.cluster.local/db/logs/series'
    method = 'POST'
    username = 'root'
    password = 'root'
```

Bibliography

- [1] <http://doi.acm.org/10.1145/1496091.1496100> Vaquero, Luis M. and Rodero-Merino, Luis and Caceres, Juan and Lindner, Maik. A Break in the Clouds: Towards a Cloud Definition. ACM, 2009.
- [2] Lawton, G. Developing Software Online With Platform-as-a-Service Technology. 2008
- [3] Prodan, R. and Ostermann, S. A survey and taxonomy of infrastructure as a service and web hosting cloud providers, Grid Computing, 2009 10th IEEE/ACM International Conference on
- [4] Gold, N. and Mohan, A. and Knight, C. and Munro, M. Understanding service-oriented software. IEEE, 2004
- [5] <https://blog.engineyard.com/2014/pets-vs-cattle> Noah Slater. Pets vs. Cattle. Engine Yard, 2014.
- [6] <http://ianmurdock.com/cloud/do-operating-systems-still-matter/> Ian Murdock. Do operating systems still matter? Ian Murdock, 2009.
- [7] <https://blog.openshift.com/platform-abstraction-advantage/> Krishnan Subramanian. The Platform Abstraction Advantage of PaaS. Red Hat, 2015.
- [8] <http://12factor.net/> The Twelve-Factor App. Heroku, 2012.
- [9] <http://ptgmedia.pearsoncmg.com/images/9780321601919/samplepages/0321601912.pdf> Humble, Jez, and David Farley. Continuous delivery: reliable software releases through build, test, and deployment automation. Pearson Education, 2010.
- [10] <http://dl.acm.org/citation.cfm?id=2600239.2600241> Merkel, Dirk. Docker: Lightweight Linux Containers for Consistent Development and Deployment. Belltown Media, 2014.
- [11] <http://doi.acm.org/10.1145/2524713.2524721> Roche, James. Adopting DevOps Practices in Quality Assurance. ACM, 2013.

-
- [12] <https://www.gnu.org/philosophy/free-sw.html> FSF Contributors. Free Software Foundation, 2015.
 - [13] <https://www.opencontainers.org/pressrelease/> Industry Leaders Unite to Create Project for Open Container Standards. Linux Foundation, 2015.
 - [14] <http://www.linuxfoundation.org/news-media/announcements/2015/07/new-cloud-native-computing-foundation-drive-alignment-among/> New Cloud Native Computing Foundation to Drive Alignment Among Container Technologies. Linux Foundation, 2015.
 - [15] <http://doi.acm.org/10.1145/227210.227227> Dolev, Danny and Malki, Dalia. The Transis Approach to High Availability Cluster Communication. ACM, 1996

Acknowledgments

First of all I'd like to thanks my family and friends for supporting me, but also for endurance, during all these years.

In particular I'd like to thanks my parents to whom I dedicated this thesis, for giving me the opportunity to follow this educational path.

I thank Luca Ferroni for giving me the opportunity of doing valuable work experience, and the whole beFair team for chasing wonderful goals every day.

I thank Michele Sorcinelli for being a faithful partner of studies and of first hacks on GNU/Linux systems.

Finally I'd like to thank the individuals, communities and companies actively develop and promote free software projects or adopt and spread sharing economy practices, locally or worldwide.