Patterns for Software Orchestration on the Cloud

Tiago Boldt Sousa, University of Porto and ShiftForward
Filipe Figueiredo Correia, University of Porto and ParadigmaXis
Hugo Sereno Ferreira, University of Porto and ShiftForward

Software businesses are redirecting their expansion towards service-oriented businesses models, highly supported by cloud computing. While cloud computing is not a new research subject, there's a clear lack of documented best practices on how to orchestrate cloud environments, either public, private or hybrid. This paper is targeted at DevOps practitioners and explores solutions for cloud orchestration, describing them as three patterns: a) SOFTWARE CONTAINERIZATION, providing resource sharing with minimal virtualization overhead, b) LOCAL REVERSE PROXY, allowing applications to access any service in a cluster abstracting its placement and c) ORCHESTRATION BY RESOURCE OFFERING, ensuring applications get orchestrated in a machine with the required resources to run it. The authors believe that these three DevOps patterns will help researchers and newcomers to cloud orchestration to identify and adopt existing best practices earlier, hence, simplifying software life cycle management.

Categories and Subject Descriptors: D.2.11 [**Software Architectures**] Patterns

Additional Key Words and Phrases: Design-Patterns, Cloud Computing, DevOps Patterns

ACM Reference Format:

Sousa, T.B., Correia, F.F. and Ferreira, H.S. 2015. DevOps Patterns for Software Orchestration on Public and Private Clouds. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 22 (October 2015), 12 pages.

1. INTRODUCTION

As software businesses move towards the cloud, new practices are required to orchestrate whole of its life cycle. The DevOps mindset was introduced as a natural evolution from traditional operation team's processes, facilitating scalable cloud orchestration. Orchestration in the cloud consists on the process of creating and maintaining the required infrastructure and services deployed to it, with DevOps achieving these goals programatically. Modern systems tend to rely of clusters of servers, increasing the orchestration complexity. Loukides describes DevOps as the enabling process for having infrastructure automatically managed with code instead of requiring human effort[Loukides 2012]. Such automation-focused approach empowers development teams to become independent while managing the life cycle of their software. Smaller companies can have part of the development team working on DevOps, while large companies can have a dedicated team implementing the required tools shared by individual teams. In both cases, each development team adopts the responsibility of building and running their applications [Erich et al. 2014; Saugatuck Technology 2014].

Being an early topic in Software Engineering, there's a general lack of documented knowledge on how to practice DevOps. This paper is targeted at DevOps practitioners and enthusiasts and it is the initial work towards formalizing current software orchestration knowledge using patterns. Figure 1 shows an early version of our proposed pattern map for software orchestration on the cloud. The same pattern map applies at both the application and infrastructure

Author's emails: Tiago Boldt Sousa: tbs@fe.up.pt; Filipe Figueiredo Correia: filipe.correia@fe.up.pt; Hugo Sereno Ferreira: hugosf@fe.up.pt.

levels, meaning that each pattern can be instantiated at the infrastructure or service levels. Such allow us to have a similar approach at orchestrating the required virtual hardware required and the applications that are deployed into it. Six categories are represented in this map:

Development. Those patterns that still apply on the development side and provide the bridge between software development and cloud orchestration.

Deployment. Takes information and instantiates the infrastructure and the applications that need it.

Supervision. Responsible for detecting anomalies in the running entities, executing the proper actions to recover them in case of failure.

Monitoring. Responsible for capturing information regarding the system state and providing it as real time analytics, alarms or for third party services.

Discovery & Application Support. Supports services to discover each-other, simplifying Master-Slave elections or message-driven work distribution.

Isolated Execution. Introduces isolated environments as hosts for running software.

From the pattern map, this paper elaborates on three patterns intended to help DevOps-aware teams and traditional operations teams by elaborating on a) SOFTWARE CONTAINERIZATION (Section 2), which identifies how services can be executed in isolated environments within a shared machine, without requiring full stack virtualization; b) LOCAL REVERSE PROXY (Section 3), which describes how to enable applications to discover each-other in a network by abstracting services using a local port and a reverse proxy and c) ORCHESTRATION BY RESOURCE OFFERING (Section 4), which introduces a technique to evaluate where services should be placed in a cluster, considering its required resources and those available in each machine in the cluster.

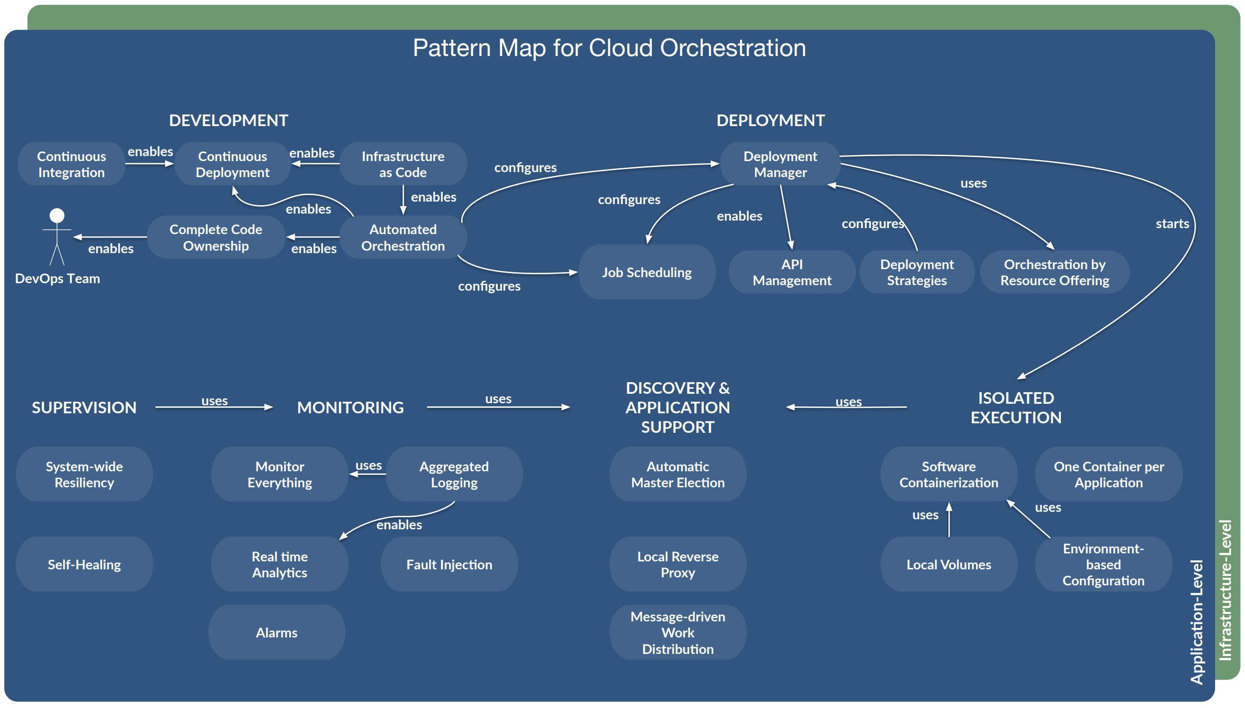

Fig. 1. Pattern Map for Software Orchestration on the Cloud.

While these apply to monolithic systems, the patterns are most useful when adopted with a micro-service based architecture[Namiot and Sneps-Sneppe 2014] and in distributed environments. Future work elaborate on the remaining pattern from Figure 1.

2. SOFTWARE CONTAINERIZATION PATTERN

This pattern addresses the need for a portable, efficient and secure environment for executing applications. The issue is most troublesome when deploying multiple applications in the same host, which a) can result in extremely complex dependency management scenarios and clutter the system or b) requires virtualization which requires part of the host's resources.

2.1 Context

Today's hardware, with multi-core and multi-CPU architectures, is built to execute multiple applications concurrently. Cloud computing often exploits resource sharing for executing multiple applications in a single host. Sharing the host's operative system with the hosted applications might introduce software incompatibilities between the applications or quickly clutter the host, as it must mutate its file system to accommodate the hosted applications' dependencies. Such introduced the need for isolated environments. Full stack virtualization quickly became the *de facto* standard approach to enabling resource sharing, allowing services to be executed in a dedicated installation of the operative system. Paravirtualization further improved that approach by exposing hardware resources directly to the virtualized environment. Still, isolation is achieved with an increased cost of hardware usage required to virtualize the operative system stack on each hosted environment.

2.2 Example

A web application has three services: an HTTP server, a database and an object caching service. These applications share some core libraries, but each depend on different versions. The development team uses a few different Linux distributions for development but production environments are to use a specific distribution. All three services should be deployed on a temporary host for testing purposes and afterwards deployed in the production environment. It becomes a complex task to develop and deploy each service such that it is easily executed by each team member, as well as quickly installed in the development and production, despite existing configurations or the adopted distribution.

2.3 Problem

Deploying a service to a host creates, mutates and pollutes the hosting environment, possibly making it unable to host incompatible applications.

Software deployments tend to couple the application with its host environment, modifying it according to their needs. When hosting multiple applications that share resources, namely file-system, CPU, memory and network availability, unexpected behavior might be observed as they compete for those resources. Furthermore, situations exist where two applications cannot coexist in the same environment due to incompatible dependencies, either virtual or physical.

2.4 Forces

Resource Allocation. We want to be able to allocate a set of resources to the environment, preventing those limits from being exceeded and granting the application its precise requirements for successful execution.

Resource Optimization. We want to be able to execute multiple applications with the minimum possible hardware for maximizing the host's resources.

Performance. We want to isolate only the required part of the software in order to prevent computation redundancy.

Supervision. We want our application to be monitored and, in case of failure, restarted with original environment.

Portability. We want to deploy the application without polluting the host with dependencies or having to manage incompatibilities with other applications.

Configurability. We expect applications be easily configured on execution time.

Security. We want applications to only expose the resources that should be available outside the execution environment.

Solipsism. Each running environment should only manage itself, communicating with external applications resiliently.

Persistency. We want data to persist in the host beyond the applications' execution lifetime, possibly being reused in future executions in the same or in a different server.

2.5 Solution

Use a container to package the service's binary files and its dependencies and use the more portable container to deploy the service in as an isolated environment.

The problem is solved recurring to application isolation by keeping the host's operative system virtually separated from hosted applications. Full stack virtualization provides complete isolation between hosted environments at the cost of virtualizing the operative system for each environment, reducing performance and adaptability, being far from an optimal solution.

Using operative system level virtualization, or containers, each application is packaged with its dependencies. A container is a self-contained isolated environment with a virtual file-system, network and resources allocation which is executed within an host operative system[Soltesz et al. 2007]. The container can be created and started programatically, with strict resource allocation, it can have configurations injected into its execution environment and be easily ported between hosts. When the container is deleted from the host, the whole environment is removed as well. On failure, it can restart itself with the same configurations and a clean environment.

2.6 Example Resolved

Consider the example previously described. Each application would be packaged into a separate container. Each container would include all the necessary files to run an application, including dependencies, operative system configurations and the application itself. Support for running containers must be introduced at hosting operative system level to enable resource isolation. In a development environment, three containers could be started in the same host. These could then be ported each to its independent host machine without changes. With large applications being separated into smaller parts, each in its own container, it is also possible to scale each component of the application independently from the others by increasing the number of instances for that specific container.

2.7 Resulting Context

This pattern introduces the following benefits:

—Resource use is optimized, with overheads being decreased when compared to full stack virtualization, as only a thin layer needs to be virtualized, improving the performance achievable by a host.

—Resources can be allocated to the container, leveraging the available host's resources between multiple containers, as well as what is exposed from the container to the host and vice-versa.

—Arguments can be provided to the container on execution to configure the application running inside it. Due to its immutability, in case of failure the container can restart with the original configuration.

—Isolated environment can be easily ported between development and production as the image size only packages the application and its dependencies, leaving out all operative system's components.

The pattern also introduces the following liabilities:

—Paravirtualization is a virtualization technique that exposes part of the host's hardware directly to the virtual machine. In some low-level hardware access scenarios, paravirtualization might provide increased performance.

—Packaging applications as containers will still introduce overheads when compared to installing applications directly in the host.

2.8 Related Patterns

Configuration might be required for a container to be adaptable to multiple hosts and scenarios. Using the ENVIRONMENT-BASED CONFIGURATION pattern it is possible to use environment variables to configure running application at execution time.

Some containers might have the need to persist information between executions in the host. That is the case of isolated databases that cannot lose their data if the machine reboots. With this goal in mind, the LOCAL VOLUMES pattern may be used to expose a folder from the host inside the container.

2.9 Known Uses

Containerization was first introduced in 1982 in the Seventh Edition Unix by Bell Labs, as a tool for testing the installation and build system of the operative system, providing an isolated file-system environment where applications could be executed. By 2008 Linux Containers (LXC) were introduced in Linux Kernel version 2.6.24, reducing the virtualization overhead and increasing efficiency [Felter et al. 2014]. By 2013 Docker was built, based on LXC, in order to make containerization easier for a broader audience. Docker is now the cloud standard for container-based deployment, with native support with multiple cloud providers, such as Amazon Web Services and Google Cloud Platform, both with native support for running docker containers [Amazon 2015; Google 2015a]. A draft is being worked on to create a standard format for containers, with RunC being the reference implementation for it, which can also run Docker-created containers [Open Container Initiative 2015].

3. LOCAL REVERSE PROXY PATTERN

This pattern deals with service discovery when deploying services that should cooperate in dynamically provisioned hardware, who lack each-others' network addresses.

3.1 Context

Cloud applications are commonly composed by a multitude of services, which may be spread over multiple physical servers in different networks. In order for services to cooperate they need to know how to contact each other, which implies the need for configuration or discovery of the hostname or IP and port where the required service can be reached. Furthermore, when a service has multiple instances, required in high availability setups, there might be the need to evenly distribute traffic between existing instances.

3.2 Example

An application server receives HTTP requests and queries a database server to get the required information to process the response. For scalability purposes, the database is distributed and the number of instances varies according to the average system load. The application server will have to query the database but, as it is running on an dynamically provisioned hardware, the application has no information about how the database servers can be reached. Figure 2 represent a possible distribution of services among the existing servers of such system.

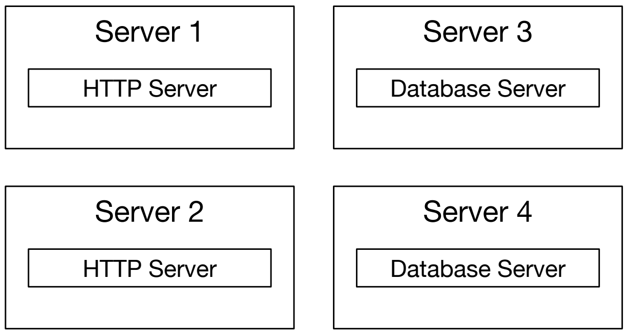

Fig. 2. The four members of a cluster, each hosting a service.

3.3 Problem

Services in the cloud might lack the knowledge of how to connect to dynamically-allocated services.

Service decoupling is required as software gets deployed and scaled automatically in the cloud, enabling the scaling of individual software components using dynamically provisioned hardware. Deploying in these conditions leave the services unaware of where their dependencies are allocated, requiring a discovery method to enable communication between them.

3.4 Forces

Realtime discovery. We want to have updated service information available in real time for proper traffic forwarding.

Location decoupling. We want to services to communicate despite their physical location.

Protocol Agnostic. We want a discovery mechanism agnostic to the protocol adopted by the services.

Load balancing. We want to be able distribute traffic between multiple instances of the same service.

Tranparency. We want discovery to be transparent for service developers.

3.5 Solution

Define a service port for each service, which is available in all servers in a cluster. Have that port routed to where the service is available using a reverse proxy.

With execution environments being provisioned on demand, an orchestration manager component should know the system state in order to understand how to route traffic optimally. Before orchestration, each service is attributed a service port. When a new service instance is started, it registers itself in orchestration component, providing an host address and the execution port. In each cluster, a local job periodically queries this information and updates a local proxy so that the service port will forward requests to the proper host and execution port where the service instance is being executed. When multiple instances are available, it is up to the local proxy to decide the distribution algorithm, possibly distributing requests evenly with a round-robin technique. Whenever a service goes down, the local proxy will forward the request to another instance of the same service or refuse the connection if no instance is available. A requesting system should keep retrying its request until the required service becomes available.

3.6 Example Resolved

This technique requires an external orchestration mechanism to keep meta-information on the services running in the cluster, regarding hosts and ports. Each host machine has a local proxy that periodically queries the orchestration manager and forwards a known local port to the host(s) and port of where a service available in the cluster. The applications expect a specific port to be available locally that will abstract the exact port and host where the service is actually running.

Consider the example previously described: a web application is deployed with two HTTP Servers receiving external requests, which must communicate with one of the two other Database Servers to create a reply. For the HTTP servers to communicate with the database, instead of establishing a direct connection, they connect to the local known port, leaving for the proxy to forward the request to an available Database server. Scalability is achieved by varying the number of Database or HTTP Servers independently, relying on the Local Proxies on the HTTP side to properly identify available Database Servers and distribute load between them. This example is represented in Figure 3.

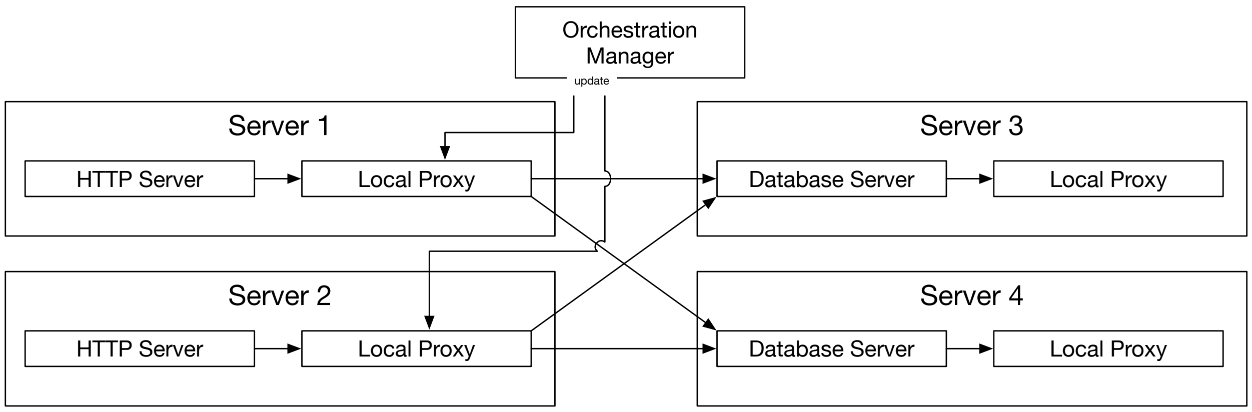

Fig. 3. Local Proxy configuration example.

3.7 Resulting Context

This pattern introduces the following benefits:

—Service development can ignore the actual physical location of other services it is integrating with, relying on the local reverse proxy to forward traffic to where the service is executing.

—Changes to the cluster are immediately identified by the orchestration manager, which will reconfigure the local proxies.

—Proxies work at the transport OSI layer or lower, hence, are protocol agnostic.

—The local proxy can be leveraged to act as load balancer, distributing connections between existing application servers as soon as they are made available.

—The approach follows an eventually consistency approach: whenever server 2 becomes unavailable, server 1 just have to keep retrying to send information to the same local port, which will eventually be forwarded to a viable instance of server 2.

The pattern also introduces the following liabilities:

—A mapping of existing services to their service ports must be maintained for each service. Furthermore, an external system must evaluate running services' status and provide up to date configurations for the reverse proxies.

3.8 Related Patterns

This pattern may be applied when ISOLATION BY CONTAINERIZATION is being used to isolate applications, facilitating communication between containers and different servers, without requiring applications to individually integrate with discovery mechanisms. Information about service ports might be configured using the ENVIRONMENT-BASED CONFIGURATION pattern.

This pattern depends on an external mechanism that keeps track of each service in the cluster. A DEPLOYMENT MANAGER holds this information and could be queried for it.

3.9 Known Uses

A basic approach is presented by Wilder, keeping an Nginx reverse proxy updated according to meta-information extracted from running docker containers in the local machine [Wilder 2015].

The reverse proxy Vulcanproxy [Community 2015b], together with the distributed key-value storage Etcd [Community 2015a] provides a reverse proxy service agnostic to the software using it. By depending on Etcd, it is not an optimal solution as it requires services to register themselves with Etcd.

A better solution is based on Apache Mesos [Foundation 2015] which allow jobs to be spawned across multiple nodes, managing their allocation and Marathon, a cluster-wide init and control system for Mesos [Mesosphere 2015a]. Using meta-information available with Marathon, a script can periodically update a local proxy server on each machine in the cluster, forwarding a TCP or UDP port, named the service port, to the actual address where the application is running, despite it being local or in a remote machine [Wuggazer 2015]. There are many implementations available to work with Marathon, including Bambo, an HAProxy auto-discovery and configuration tool for Marathon [Qubit 2015]. There is also a script that can configure a local HA proxy, made available by Marathon's team [Mesosphere 2015b].

4. THE ORCHESTRATION BY RESOURCE OFFERING PATTERN

This pattern deals with service allocation in a cluster when each server can host multiple isolated applications.

4.1 Context

The need for DevOps tools and techniques was mostly introduced on par with the need to scale and orchestrate highly-available applications in multi-server environments. A key requirement is to guarantee that applications are allocated to host machines which fulfill the application's hardware requirements and that this happens without human interaction. Such enables servers to run multiple services while ensuring their execution within the host's resource limits, guaranteeing the expected performance. To do so, whenever a new service is to be instantiated in the cluster, a negotiation between existing nodes should take place in order to identify the machine where the service can be started.

4.2 Example

A software team as created a set of micro-services that together provide a web application. These services are described in Table I. They must be deployed in the available servers, respecting their resource requirements and constraints.

Table I. List of services and their possible configurations for a production environment.

Service Name	CPUs Count	RAM	Disk Space	Instance Count	Other Constraints
HTTP	2	2 GB	5 GB	2	*hostname*=unique; *location*=Europe
Database	2	8 GB	50 GB	2	*hostname*=unique; *SSD*=true; *location*=Europe

4.3 Problem

Placing services in dynamically-provisioned infrastructure must guarantee that the service's infrastructure requirements are fulfilled by the selected host.

Given a cluster and a set of services, a viable deployment setup of the services to the servers needs to be identified and executed. When dealing with dynamic hardware, we want to guarantee service placement and relocation in case of a server failure without human interaction. Considering that servers might host multiple services, we need to ensure that services are always deployed in servers that fulfill their requirements, ensuring the expected performance.

4.4 Forces

Infrastructure Decoupling. We want our services to be agnostic of the environment where they are being executed.

Resource leveraging. We expect allocation to happen in a host that meets all service requirements.

Relocation on Failure. We expect services to be relocated in case of failure by the host.

Restart on Failure. We expect the services to be restarted in case of failure.

4.5 Solution

Orchestrate services in a cluster based on each host's resource-offering announcements.

The clustered machines could adopt a master-slave architecture, being the master responsible for handling service allocation requests for the cluster. When the master receives a new allocation request with the service requirements, it broadcasts the request to slaves. Those able to accommodate the requirements will notify the master, which then chooses the best slave for accommodating the service. If no slave is capable of hosting the service, the master can periodically retry, until allocation is successful. Figure 4 illustrates this interaction.

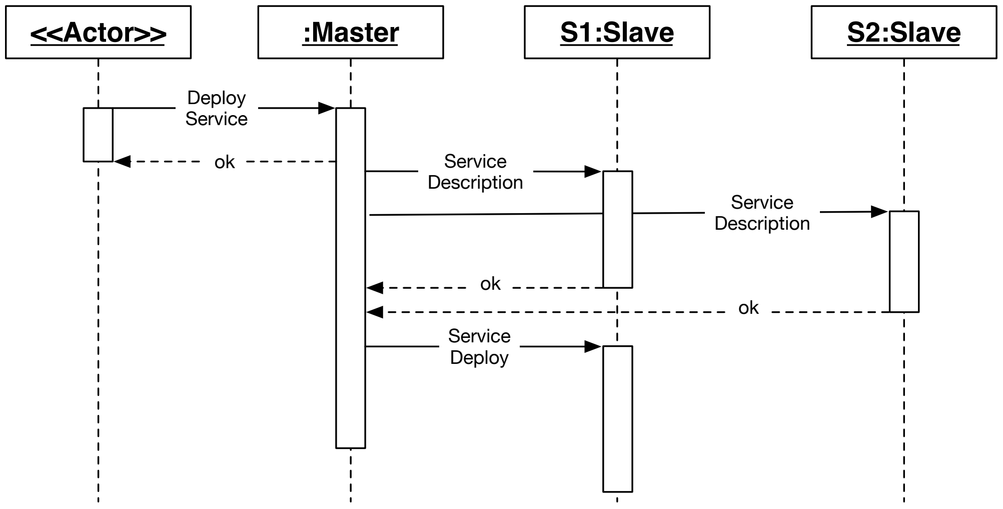

Fig. 4. Sequence diagram representing communication between master and slaves for service allocation.

4.6 Example Resolved

Consider the list of services introduced in Table I and the servers introduced in Table II. Consider also that the servers are clustered in a master-slave architecture, where the master is also a slave, and the elected master is responsible for receiving and orchestrating work between slaves.

To deploy the services, a user sends a request with its specifications to the master, which is propagated to all the slaves. The slaves evaluate if they have the ability to host the service, considering the availability of resources and by checking existing constrains. In the example we can see that the hostname must be unique, meaning that no two HTTP or database servers can be placed in the same host. Also, location for deployment must be in the European zone and for the database service it must be placed in a server with SSD storage available.

Table II. List of available servers in the cluster.

Server Name	CPUs Count	RAM	Disk Space	Constraints
Server1	8	2 GB	5 TB	*location*=Europe
Server2	4	8 GB	50 TB	*SSD*=true; *location*=Europe
Server3	8	2 GB	1 TB	*location*=Asia
Server4	16	32 GB	500 GB	*location*=US

In this example, when a request for deploying the HTTP service arrives, all servers are at full capacity and are able to fulfill the requested resources regarding CPU, RAM and disk space. Still, the constrains further restrict the service placement, since it must be placed in Europe, leaving only Server1 and Server2 to be able to accommodate it. Those servers deploy the master as possible candidates and since there is the intention of deploying two instances of the service, both servers are used for this purpose. When the service is deployed, each host server subtract 2 CPUs, 2GB RAM and 5 GB of storage from their available resources, influencing how they reply to future requests.

When a request is issued to deploy the Database service, only European servers with SSD storage will respond to the request, resulting in Server2 as the only possible host. With one instance deployed and another without allocation, the master periodically queries the cluster, waiting for a new slave to join and provide a positive answer.

Whenever a service is stopped, the allocated resources are released, increase the server's capacity to host new services.

4.7 Resulting Context

This pattern introduces the following benefits:

—Service development can be agnostic of the host where the service is placed, describing only its requirements.

—Services are automatically allocated based on their requirements.

—Service recovery is provided both at the service or host level.

—Resources on the host are allocated in order to allocate as many services possible, without compromising performance.

—Scalability is achieved by adding slaves to the cluster and individually change the number of instances for each service.

The pattern also introduces the following liabilities:

—Allocation using a greedy placement algorithm might result only in a locally-optimal solution.

—Using a single master would result in a single point of failure.

4.8 Related Patterns

This pattern allows slaves to evaluate their resources against the needs described for an application. This concept works better with the ISOLATION BY CONTAINERIZATION pattern, by evaluating the required resources to execute a container. This pattern prescribes a master-slave architecture, where the master's behavior is to be described in the DEPLOYMENT MANAGER pattern.

4.9 Known Uses

Kubernetes [Google 2015b] by Google abstracts a set of machines, receiving requests for allocating CONTAINERS in the cluster. Kubernetes is still in an early phase and support for more granular resource allocation is being worked on. CoreOS [CoreOS Community 2015] offers similar technology, with a centralized registry made available with Etcd.

Mesos and Marathon provide a better tested and robust solution for achieving the same goal. New applications are submitted to the cluster using an HTTP API describing its requirements and constraints. With this information, the master communicates with the slaves, identifying a valid host and issuing the order for placing the service[Hindman et al. 2011].

5. CONCLUSIONS

Cloud Computing plays a key role at enabling businesses to expand their reach. DevOps plays a part, enabling *devs* to do their own *ops*, allowing teams to manage vast amounts of servers without a proportional investment in operations through human resources. With hundreds of tools and practices being introduced, both newcomers and experts might feel uncertain on what choices to make towards DevOps. This paper introduced three patterns that will help software development teams to run their software to the cloud. The authors believe that capturing information as patterns will ease the adoption of these practices, hence, improving the ability of teams to leverage cloud computing to build and orchestrate better software. This paper was the first step towards outlining a pattern map for orchestration software on the cloud. Future work will pursue the evolution and complete description of the pattern map shown in Figure 1.

6. ACKNOWLEDGMENTS

We would like to thank Hironori Washizaki, Eduardo Fernandez, Madiha Syed and Bharatkumar Sharma for their valuable input during the Pirates Session at PloP 2015. We are also thankful to José Ruiz for his contributions as shepherd to this paper ShiftForward's team for their shared experience in the form of so many valuable discussions.

REFERENCES

Amazon. 2015. Amazon EC2 Container Service. (2015). Retrieved 2015-09-01 from https://aws.amazon.com/docker/

CoreOS Community. 2015a. Etcd Project Page. (2015). Retrieved 2015-09-10 from https://github.com/coreos/etcd

Vulcanproxy Community. 2015b. Vulcanproxy Project Page. (2015). Retrieved 2015-09-01 from http://www.vulcanproxy.com/

CoreOS Community. 2015. CoreOS Project Page. (2015). Retrieved 2015-09-10 from https://coreos.com/

Floris Erich, Chintan Amrit, and Maya Daneva. 2014. Report : DevOps Literature Review. (2014).

Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. 2014. An Updated Performance Comparison of Virtual Machines and Linux Containers. *Technology* 25482 (2014).

Apache Foundation. 2015. Mesos Project Page. (2015). Retrieved 2015-09-12 from http://mesos.apache.org/

Google. 2015a. Google Cloud Container Service. (2015). Retrieved 2015-09-01 from https://cloud.google.com/container-engine/

Google. 2015b. Kubernetes Project Page. (2015). Retrieved 2015-09-12 from http://kubernetes.io/

Benjamin Hindman, Andy Konwinski, and Matei Zaharia. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. *Nsdi* (2011). DOI:http://dx.doi.org/10.1109/TIM.2009.2038002

Mike Loukides. 2012. *What Is DevOps?* O'Reilly Media. 15 pages. http://shop.oreilly.com/product/0636920026822.do

Mesosphere. 2015a. Marathon Project Page. (2015). Retrieved 2015-09-12 from https://mesosphere.github.io/marathon

Mesosphere. 2015b. Mesosphere Service Discovery & Load Balancing. (2015). Retrieved 2015-09-10 from https://mesosphere.github.io/marathon/docs/service-discovery-load-balancing.html

Dmitry Namiot and Manfred Sneps-Sneppe. 2014. On Micro-services Architecture. *International Journal of Open Information Technologies* 2, 9 (2014), 24–27. http://injoit.org/index.php/j1/article/view/139

Open Container Initiative. 2015. Open Containers Project Page. (2015). Retrieved 2015-09-01 from http://www.opencontainers.org/

Qubit. 2015. Bamboo Project Page. (2015). Retrieved 2015-09-01 from https://github.com/QubitProducts/bamboo

Saugatuck Technology. 2014. Why DevOps Matters : Practical Insights on Managing Complex & Continuous Change. (2014).

Stephen Soltesz, Stephen Soltesz, Herbert Pötzl, Herbert Pötzl, Marc E Fiuczynski, Marc E Fiuczynski, Andy Bavier, Andy Bavier, Larry Peterson, and Larry Peterson. 2007. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.* 41 (2007), 275–287. `DOI:http://dx.doi.org/10.1145/1272998.1273025`

Jason Wilder. 2015. Automated Nginx Reverse Proxy for Docker. (2015). Retrieved 2015-06-20 from http://jasonwilder.com/blog/2014/03/25/automated-nginx-reverse-proxy-for-docker/

Patrick Wuggazer. 2015. *Evaluation of an Architecture for a Scaling and Self-Healing Virtualization System*. Ph.D. Dissertation. University of Magdeburg.