DevOps Patterns for Software Orchestration on Public and Private Clouds

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

The software business is directing its growth 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 to orchestrate cloud environments, either public, private or hybrid. This paper explores existing solutions for cloud orchestration and describes them as three patterns: I) ISOLATION BY CONTAINERIZATION, providing resource sharing with minimal virtualization overhead, II) ORCHESTRATION BY RESOURCE OFFERING, ensuring applications get orchestrated in a machine with the required resources to run it and III) DISCOVERY BY LOCAL REVERSE PROXY, allowing applications to access any service in a cluster abstracting its placement. 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. and Ferreira, H.S. 2015. DevOps Patterns for Software Orchestration on Public and Private Clouds. jn 0, 0, Article 0 (October 2015), 11 pages.

1. INTRODUCTION

DevOps is introduced as a natural evolution towards cloud computing over traditional operations teams, 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. Loukides describes DevOps as the enabling process of having infrastructure automatically managed with code instead of people[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; 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 the initial work towards formalizing current DevOps knowledge in the form of patterns. Figure 1 shows an early version of our proposed pattern map for DevOps. The same pattern map applies at both the application and infrastructure levels, meaning that each pattern can be instantiated at the infrastructure

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

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:. Responsible for receiving information and instantiating both infrastructure and applications on top of 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 that pattern map, this paper elaborates on three patterns, that are intended to help DevOps-aware teams and traditional operations teams by explaining how services can be isolated, placed in a specific node and discovered in a cluster. The patterns will have increased value when adopted with an architecture based on micro-services[Namiot and Sneps-Sneppe 2014], but they apply to monolithic systems as well. The first pattern, ISOLATION BY CONTAINERIZATION, identifies how services can be executed in a shared machine in a completely isolated environment, without requiring full stack virtualization. ORCHESTRATION BY RESOURCE OFFERING introduces a technique to evaluate where a new service instance should be placed in the cluster, considering the required and available resources. Finally, DISCOVERY BY LOCAL REVERSE PROXY introduces a technique for service discovery which abstracts where a service is being executed in a cluster by always forwarding traffic from a local port to it.

2. THE ISOLATION BY CONTAINERIZATION PATTERN

This pattern addresses the need for a portable, efficient and isolated environment for executing applications.

2.1 Context

Cloud computing exploits resource sharing for executing multiple applications in a single server, leveraging those resources in order to fully use the server's capabilities. Sharing the server's operative system with the hosted applications might introduce software incompatibilities between the applications or quickly clutter the system, as the host must mutate 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 it by exposing hardware resources directly to the virtualized environment. Still, isolation is achieved with an increased cost of hardware, required to virtualize the operating 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 multitude of Linux distributions for development but production environments are to use a specific distribution. All three services should be deployed on a temporary server 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.

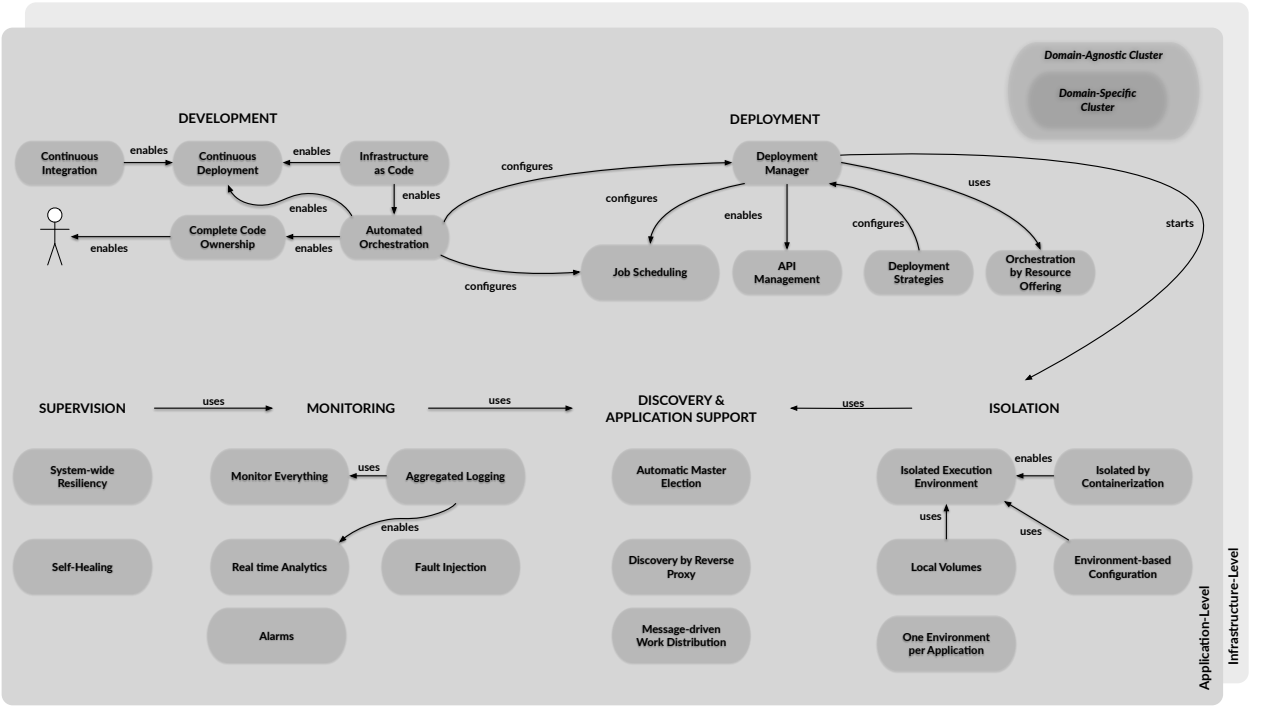

Fig. 1. DevOps Pattern Map.

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 co-exist 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 server'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.

Discoverability. We want applications to be able to discover and communicate with others in the cluster while still network isolated.

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 applications and its dependencies and deploy the service within it.

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

Using operating-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 programatically, started with strict resource allocation, have configurations injected into its internal 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 machine. 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:

—Overheads are decreased when compared to full stack virtualization, as the whole operative system lives outside the isolated environment.

—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:

—As the operating system is not being virtualized, interaction with hardware is proxied through it, possibly reducing performance when compared fully virtualized environments using paravirtualization.

—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 configure variables in the environment using environment variables which can be used to configure running application on execution time.

Some containers might have the need to persist information between executions in the host. That is the case of isolated databases that might not lose their data if the machine reboots. For that, the LOCAL VOLUMES pattern might be applied 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[1]. 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[2].

3. THE DISCOVERY BY LOCAL REVERSE PROXY PATTERN

This pattern deals with service discovery when multiple applications must communicate in dynamically provisioned hardware.

3.1 Context

Cloud applications are commonly composed by a multitude of services. These are possibly spread physically between multiple 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 is being executed. The second is not trivial on dynamically-provisioned hardware. 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 system has no information about where the database servers are running. Figure 2 represent a possible distribution of services into the existing servers in such system.

3.3 Problem

Services in the cloud might lack the connection details of other 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

Location decoupling. We want to enable services to cooperate with their peers as long as they have an active internet connection.

[1]Both cloud providers have native support for running docker containers. Details at `https://aws.amazon.com/docker/` and `https://cloud.google.com/container-engine/`.

[2]Details at `http://www.opencontainers.org/`.

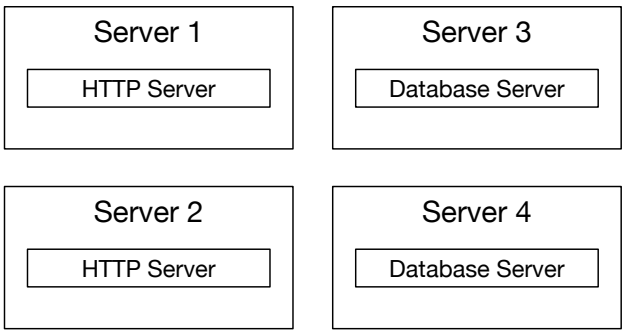

Fig. 2. The four members of a cluster, each hosting a service.

Automatic Discovery. We want the system to be self-aware and provide a mechanism for services to communicate with each-other.

Protocol Agnostic. We want a mechanism that will be agnostic of the protocol used by the services to communicate.

Load Balancing. We want to be able distribute traffic evenly between all instances of the same service.

Tranparency. We want to minimize changes to existing services so that they can discover other services without a complex discovery system.

Immidiate discovery. We want to have discovery information immediately available to the service, once started.

3.5 Solution

Configure a service port for each service, which is routed locally from each server to the proper destination using a reverse proxy.

With execution environments being provisioned on demand, an orchestration managing 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.

3.6 Example Resolved

Considering 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. Each Server machine has a local proxy that is periodically updated by an external mechanism in order to forward a known local port to the proper host(s) and port of each service available in the cluster. To enable the proxy to be updated, an external orchestration mechanism must keep meta-information on all the services running in the cluster. 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.

3.7 Resulting Context

This pattern introduces the following benefits:

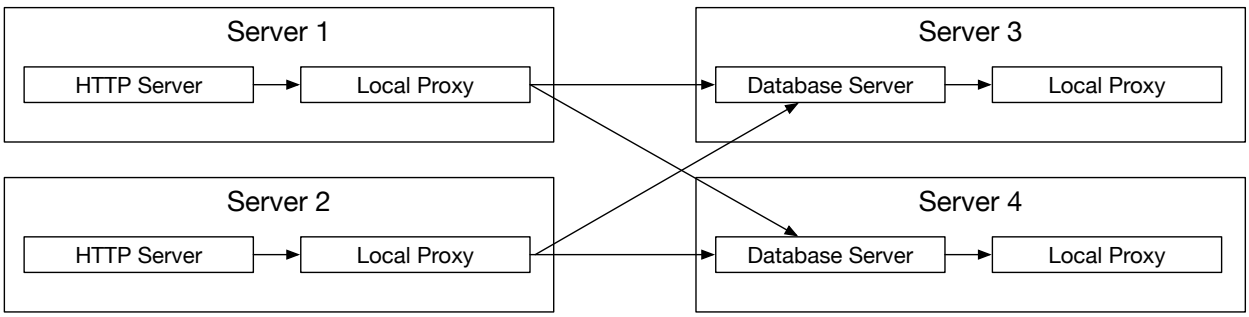

Fig. 3. Local Proxy configuration example.

—Services are decoupled from the orchestration component, requiring only information about the service port for each server they need to connect to, which can be set during development.

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

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].

Vulcanproxy [3], together with Etcd[4] 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[5], and Marathon[6]. 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 Bamboo [7] or a *cron* script that directly configures a proxy, made available by the Marathon team [8].

[3] An open source project, available at `http://www.vulcanproxy.com/`.
[4] Etcd is a distributed Key-value storage, enabling shared configuration and discovery, and is available at `https://github.com/coreos/etcd`.
[5] Apache Mesos allow jobs to be spawned across multiple nodes, managing their allocation to them. Details at `http://mesos.apache.org/`.
[6] Marathon is a cluster-wide init and control system Mesos. Details at `https://mesosphere.github.io/marathon/`.
[7] Bamboo is an HAProxy auto-discovery and configuration tool for Marathon. Details at `https://github.com/QubitProducts/bamboo/`
[8] Details at `https://mesosphere.github.io/marathon/docs/service-discovery-load-balancing.html`.

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. These services 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

Orchestrating services in dynamically-provisioned hardware must guarantee that the service's hardware 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

Service allocation. We expect services to be allocated among existing nodes, guaranteeing that the host servers fulfill the requirements to execute hosted services.

Infrastructure Decoupling. We want our services to be agnostic of the environment where they are being executed, but still enforce requirements needed by them.

Relocation on Failure. We expect services that where being executed on a failing server to be automatically relocated to other servers that still fulfill the services' requirements.

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, with the master being responsible for receiving service requests for the cluster. When the master receives a new request with the required information on how to start the service and its requirements, it broadcasts the request amongst slaves. Those able to accommodate the

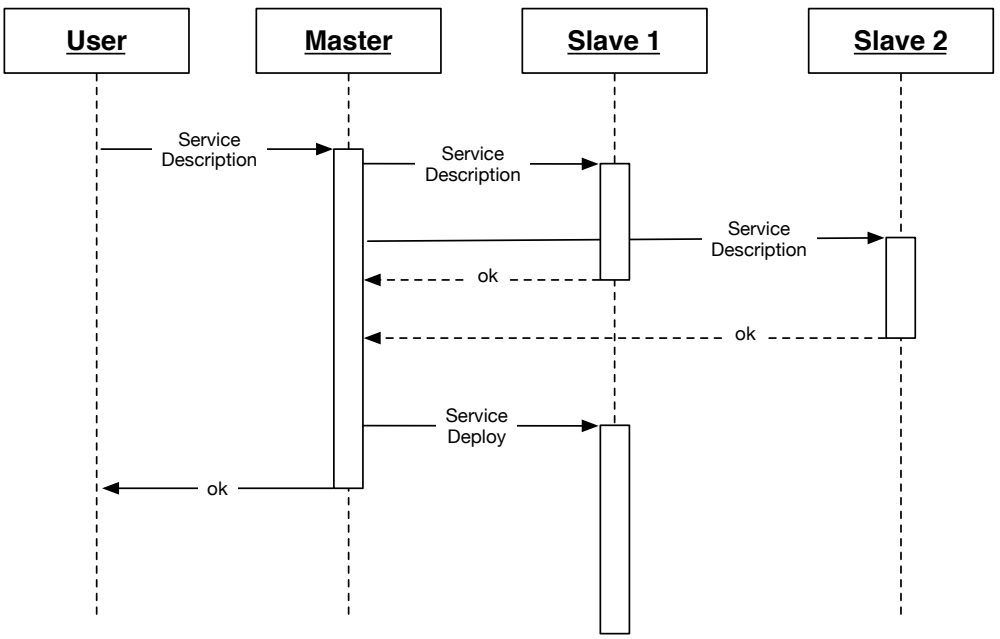

Fig. 4. Sequence diagram representing communication between master and slaves for service allocation.

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

service will notify the master, which then evaluates the received responses and allocates the service in the server with the most available resources. If there isn't a slave capable of hosting the service, the master will periodically retry sending the message, until allocation is successful. Figure 4 represents the described interaction.

4.6 Example Resolved

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

To deploy the services, an 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 not only the availability of resources but also 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.

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 allocation is automatically managed.

—Spare resources on a server can be allocated to smaller services or short-lived scripts.

—Scalability is achieved by increasing the service count and adding slaves to the cluster. Pending services will be immediately forwarded to them.

The pattern also introduces the following liabilities:

—Allocation using a greedy placement algorithm can result in an non-optimal solution.

4.8 Related Patterns

This pattern allow 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. The behavior described enables a master-slave architecture, where the master's behavior is to be described in the DEPLOYMENT MANAGER pattern.

4.9 Known Uses

Kubernetes [9] by Google abstract 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[10] offers similar technology, with a centralized registry made available with Etcd.

Mesos and Marathon provide a better tested and robust solution for the 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

DevOps is playing a key role at 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. The authors believe that capturing information as patterns will ease the adoption of DevOps best practices and tools, 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 DevOps and documenting three of those patterns. In the future we aim at capturing a complete pattern language for DevOps.

[9] Details at urlhttp://kubernetes.io/.
[10] Details at `https://coreos.com/`.

6. ACKNOWLEDGMENTS

We would like to thank José Ruiz for his contributions as shepherd to this paper. We would also like to thank ShiftForward's team for their valuable input.

REFERENCES

ERICH, F., AMRIT, C., AND DANEVA, M. 2014. Report : Devops literature review.

FELTER, W., FERREIRA, A., RAJAMONY, R., AND RUBIO, J. 2014. An updated performance comparison of virtual machines and linux containers. *Technology 25482*.

HINDMAN, B., KONWINSKI, A., AND ZAHARIA, M. 2011. Mesos: A platform for fine-grained resource sharing in the data center. *Nsdi*.

LOUKIDES, M. 2012. *What Is DevOps?* O'Reilly Media.

NAMIOT, D. AND SNEPS-SNEPPE, M. 2014. On micro-services architecture. *International Journal of Open Information Technologies 2,* 9, 24–27.

SOLTESZ, S., SOLTESZ, S., PÖTZL, H., PÖTZL, H., FIUCZYNSKI, M. E., FIUCZYNSKI, M. E., BAVIER, A., BAVIER, A., PETERSON, L., AND PETERSON, L. 2007. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev. 41*, 275–287.

TECHNOLOGY, S. 2014. Why devops matters : Practical insights on managing complex & continuous change.

WILDER, J. 2015. Automated nginx reverse proxy for docker.

WUGGAZER, P. 2015. Evaluation of an architecture for a scaling and self-healing virtualization system. Ph.D. thesis.