DevOpSlang – Bridging the Gap between Development and Operations

Johannes Wettinger, Uwe Breitenbücher, and Frank Leymann

Institute of Architecture of Application Systems, University of Stuttgart
{wettinger,breitenbuecher,leymann}@iaas.uni-stuttgart.de

Abstract DevOps is an emerging paradigm to eliminate the split and barrier between developers and operations personnel that traditionally exists in many enterprises today. The main promise of DevOps is to enable continuous delivery of software in order to enable fast and frequent releases. This enables quick responses to changing requirements of customers and thus may be a critical competitive advantage. In this work we propose a language called *DevOpSlang* in conjunction with a methodology to implement DevOps as an efficient means for collaboration and automation purposes. Efficient collaboration and automation are the key enablers to implement continuous delivery and thus to react to changing customer requirements quickly.

Keywords: DevOps, DevOps Specification, Devopsfile, Deployment Automation, Application Evolution, Cloud Computing.

1 Introduction

Today, many enterprises face a common, major challenge in terms of software delivery: customers and users expect fast responses to their constantly changing requirements, concerning both functional and non-functional properties of a software [6]. Frequent releases are vital to satisfy such expectations, which is indeed a crucial competitive advantage. However, technical and non-technical challenges have to be addressed in order to implement short release cycles. Cloud computing [10,2] introduced key enablers such as on-demand provisioning of resources (virtual machines, storage, network, platform-level services, etc.) and the pay-per-use model to tackle some of the major technical challenges. With these properties Cloud computing provides a means to support different service models such as infrastructure as a service (IaaS) and platform as a service (PaaS) combined with different deployment models (public, private, or hybrid Cloud) [10]. Beside these technical enablers, further conditions are required to enable fast and continuous delivery of software. The DevOps paradigm [6,7,17] addresses another major challenge, namely the split and barrier between developers and operations personnel. To overcome such a split that is predominant in many organizations today, organizational changes, cultural changes, and technical frameworks are required. In terms of organizational changes teams consisting

M. Villari et al. (Eds.) : ESOCC 2014, LNCS 8745, pp. 108–122, 2014.

of both developers and operations people may be established. Moreover, 'DevOps' may be introduced as a new role for people mainly working on coordinating the collaboration between both. Major companies such as Facebook[1], Yahoo[2], and others[3] are seriously implementing DevOps.

The DevOps paradigm is not bound to Cloud computing. Although combining these two makes a lot of sense as outlined before, DevOps could also be implemented in conjunction with other computing paradigms. In this work we mainly focus on enabling DevOps in combination with Cloud computing to reveal the full potential of DevOps. Our major contributions in this context are:

- We define a methodology to implement the DevOps paradigm in practice with a high degree of automation.
- We propose a language to be used to support the aforementioned methodology for collaboration and automation purposes.
- Based on the requirements stated in the motivating scenario we implement and evaluate DevOps-centric artifacts to deploy and operate an application, following our methodology and using the language we introduce.

The remaining of this paper is structured as follows: Section 2 refines the problem statement based on the introduction presented here. Moreover, a motivating scenario is introduced. Derived from this scenario and the problem statement in general, Sect. 3 defines a DevOps-centric methodology to deploy and operate applications in an automated manner. Section 4 introduces a language to practically support our proposed methodology. The evaluation of both the methodology and the language is described in Sect. 5. Finally, Sect. 6 and Sect. 7 present related work, conclusions, and future work.

2 Problem Statement and Motivating Scenario

In the previous Sect. 1 we briefly introduced the DevOps paradigm, aiming to eliminate the traditional split and barrier between developers and operations personnel. This split causes long release cycles for applications in many enterprises, very often several months. However, most users and customers today expect much faster responses to their changing and growing requirements. Thus, it becomes a critical competitive advantage to deliver software continuously [6], incorporating users' feedback and requirements as fast as possible. One major precondition to implement continuous delivery of software is to automate the whole deployment process in a repeatable way [6], including steps such as:

- Retrieve sources from version control
- Build binaries using build scripts
- Verify correctness of built binaries and run unit tests

[1] Facebook uses Chef (DevOps tool): `http://www.getchef.com/customers/facebook`
[2] DevOps at Flickr (Yahoo): `http://goo.gl/XBKq`
[3] Companies moving to DevOps: `http://www.getchef.com/solutions/devops`

– Provision infrastructure resources using provisioning scripts
– Deploy middleware and application components using deployment scripts

Ideally, the implementation of such an overarching automated process takes place in parallel to the development of the application itself, always taking into account changing and growing requirements of the application. The necessary constant collaboration between developers and operations is enabled by implementing the DevOps paradigm. Optionally, the automation of the deployment process can also be implemented afterwards, e.g., for legacy applications that still need to be maintained. In this paper we consider applications that are continuously delivered based on a fully automated deployment process. We assume that an application always consists of two major building blocks:

1. Business functionality such as the business logic, user interfaces, APIs, etc.
2. Supporting functionality such as the *operations logic,* tests, etc.

The *operations logic* is the fundamental enabler to implement a fully automated deployment process because it provides the necessary artifacts such as build scripts to create the application's binaries and deployment scripts to repeatable deploy the application to different environments (development, test, production, etc.). Most of today's enterprises and Web applications that aim for fast and frequent releases fall into this category of applications[4]. However, there are other kinds of applications such as legacy applications running in production that are maintained using highly manual processes without any means to deploy or re-deploy the application in an automated and repeatable manner. Our research does not focus on such applications without full deployment automation.

We put emphasis on creating and operating applications that have an evolutionary emerging and changing architecture, mostly by following agile software development practices [1]. This is a common way to create new applications in these days because a huge variety of IaaS and PaaS [10] offerings such as Amazon Web Services[5], Google Cloud Platform[6], and Heroku[7] with many add-on services[8] are easy to use and fast to integrate with each other. Thus, application developers start with some basic offerings such as a simple virtual machine (VM) or a Ruby runtime for an initial version of their application and add or remove additional services such as data stores, caching, queues, and monitoring services as they require it. This results in an evolutionary emerging and changing application architecture according to the requirements of the application's stakeholders.

Figure 1 shows an example for the architecture evolution of a simple chat application. Initially, the application is simply running in a Node.js runtime environment. Then, a database based on MongoDB is added to store some chat

[4] DevOps at Flickr (Yahoo): `http://goo.gl/XBKq`
[5] Amazon Web Services: `http://aws.amazon.com`
[6] Google Cloud Platform: `http://cloud.google.com`
[7] Heroku: `http://www.heroku.com`
[8] Heroku add-ons: `http://addons.heroku.com`

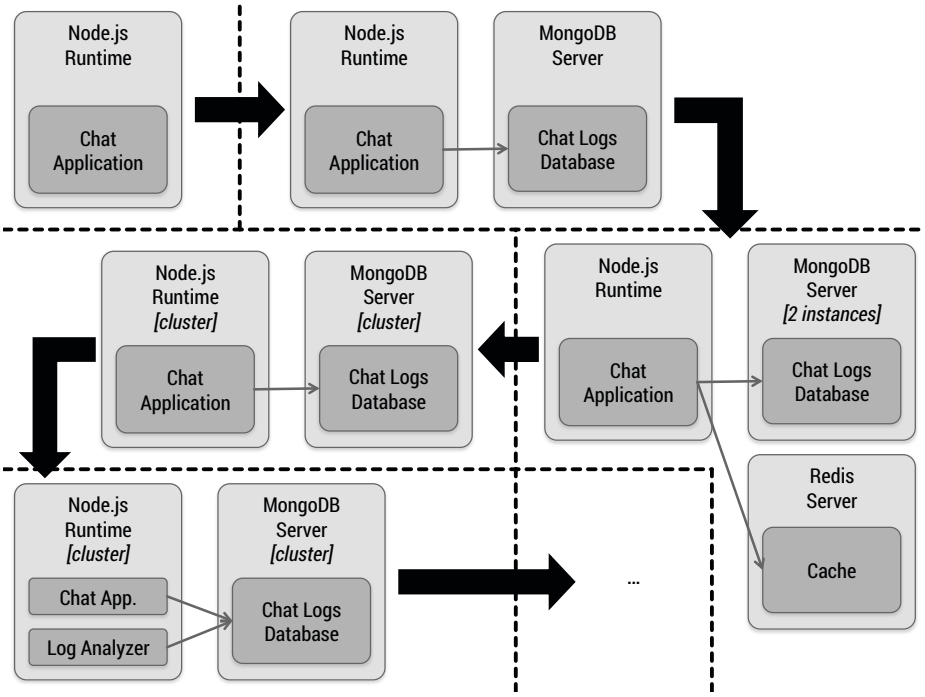

Fig. 1. Evolution of chat application architecture

logs. As the application needs to get more scalable, two instance of MongoDB are run. Moreover, a Redis server is used for caching purposes. However, this architecture does not seem to scale. Thus, the MongoDB server and the Node.js runtime environment are both run as clusters. The Redis server gets removed. In the next iteration, a log analyzer gets introduced as another application component to extract valuable information from the chat logs. The evolution may continue in this fashion including further aspects such as changing the underlying infrastructure. Whereas the first iterations may be hosted on local VMs or VMs provided by an IaaS provider such as Amazon, later versions of the applications might be hosted on PaaS offerings such as Heroku and MongoHQ[9] to address scalability issues.

In a conventional setup with development and operations split across different departments it would be hard or even impossible to catch up with such constantly changing operations requirements of an application. The DevOps paradigm aims to improve the situation for such scenarios by moving together development and operations. However, repeatable and fast processes can only be achieved with comprehensive automation, reducing manual intervention as much as possible. Manual processes to deploy and operate applications are error-prone, slow, and costly [14]. To implement such automated processes not only integrated tool

[9] MongoHQ: `http://www.mongohq.com`

support is required. However, in today's discussions this seems to be the focus beside the cultural change that is necessary to implement DevOps. Thus, in the following Sect. 3 we describe a comprehensive DevOps-centric methodology to support the evolutionary process of creating and operating applications, aiming for a maximum degree of automation.

3 DevOps-centric Methodology to Operate Applications

This section describes a methodology to implement the DevOps paradigm in practice with a high degree of automation. Our goal is to support DevOps scenarios such as the one outlined in Sect. 2 by automating the processes involved as much as possible. Figure 2 provides an overview of our proposed methodology, consisting of two major building blocks: (i) the upper part focuses on developing the application and preparing its operation in a tightly integrated manner; (ii) the lower part describes the actual deployment and operation of the application.

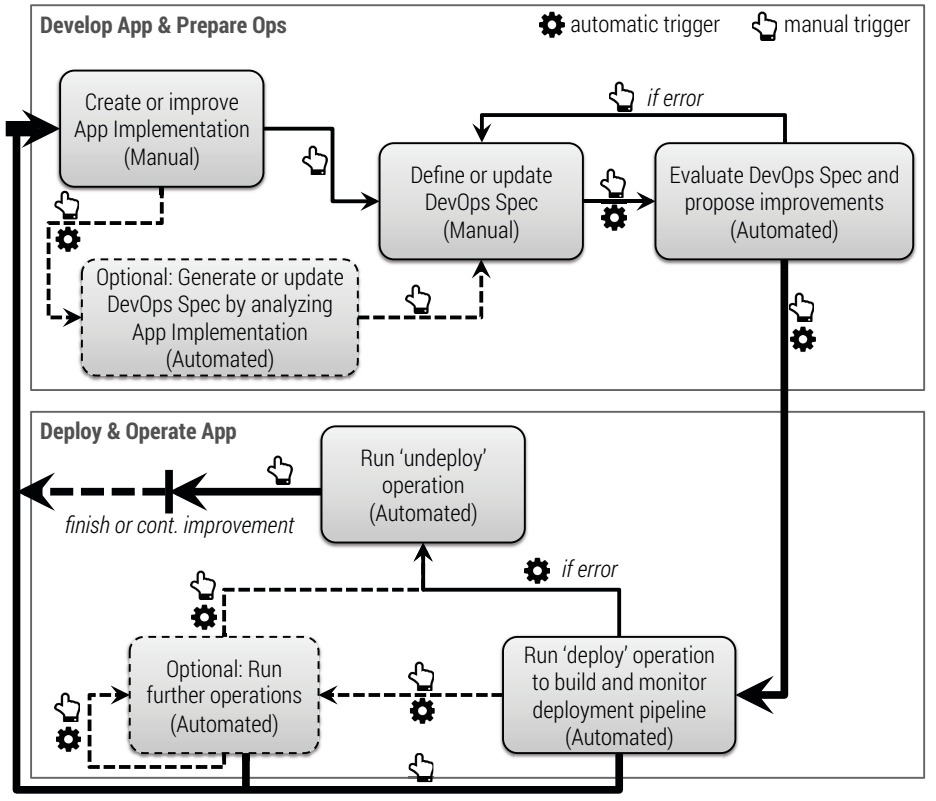

Fig. 2. Overview of our DevOps-centric methodology

Syntactically, Fig. 2 describes our methodology as a cyclic, directed graph. Each node represents a step of the methodology; the edges define the order of the steps. Dashed lines denote optional steps and paths. An edge can be seen as a trigger of the step to which it points. Depending on the annotation of the edge, the following step may be triggered automatically or manually. The entry point of the whole methodology is the *creation of the application implementation.* Obviously, we consider this step as a mainly manual process. As implied in Sect. 2 it does not only cover the implementation of the business functionality such as the business logic and the user interfaces; supporting functionality such as the operations logic are part of the implementation, too. Then, a central artifact in our methodology comes into play, namely the *DevOps Spec,* i.e., the DevOps specification:

Definition 1 (DevOps Spec). *A* DevOps Spec *specifies all developer- and operations-related aspects of a particular application to deploy it fully automated. For this purpose, an executable 'deploy' operation is defined in the* DevOps Spec. *This operation may utilize developer-centric operations such as 'build', 'test', and 'start' defined in the* DevOps Spec, *too. Moreover, operations to manage the application (e.g., 'scale', 'backup-database', 'undeploy', etc.) are specified to be triggered either automatically or manually after a successful deployment.*

Developers and operations people work closely together when maintaining the DevOps Spec, so the DevOps Spec serves as an important means to enable efficient collaboration between the two parties. As shown in Fig. 2 the *DevOps Spec* for an application may be *created* manually, either after a first iteration of the application implementation has been created or in parallel to creating the implementation. Optionally, an initial version or a skeleton of the *DevOps Spec* can be *generated* by analyzing the application implementation to find out some initial deployment requirements of the application. Such an analysis may be based on common conventions for application components such as the existence and the content of certain descriptor files [18]. For instance, a Node.js application typically owns a `package.json` file specifying its dependencies and the command to start the application.

The next step, triggered automatically or manually, is the automated *evaluation of the DevOps Spec.* The goal of this step is to use a set of rules to find possible conflicts, errors, missing parts, or weaknesses. Based on these findings, improvements are proposed to refine the DevOps Spec. As an example, a set of platform-bound commands to deploy a particular middleware component may be better replaced by a portable, tested, open-source artifact such as a Chef cookbook[10] maintained by the DevOps community. Based on these improvement suggestions the DevOps Spec may be updated accordingly.

Once the DevOps Spec is declared to be in a condition to be ready for deployment (by addressing the reported issues or by ignoring them), we switch to the second part of our methodology to deploy and operate the application. The *'deploy' operation* is run to *build the deployment pipeline:*

[10] Chef cookbooks: `http://community.opscode.com/cookbooks`

Definition 2 (Deployment Pipeline). *A* deployment pipeline *is an automated manifestation of the process for getting software from its sources (e.g., from version control) to be deployed to the target environment (e.g., development, test, production, etc.) [6]. The 'deploy' operation defined in the DevOps Spec prescribes how to build the deployment pipeline.*

The *deployment pipeline,* i.e., the execution of the 'deploy' operation is *monitored.* If an error occurs, the *'undeploy' operation is run* automatically. All error logs are stored for later analysis. Optionally, further operations such as 'scale' or 'backup-database' may run to manage the application. These runs may be triggered manually or automatically. In any case after the 'undeploy' operation has been run, it depends on a manual decision to go back to the first part to continuously improve the application implementation, update the DevOps Spec, and eventually re-deploy the application. Alternatively, the application is not targeted for re-deployment, e.g., in case the application is decommissioned completely. In this case, running the 'undeploy' operation is the final step. Moreover, we may also go back to the first part improving the application implementation and updating the DevOps Spec while the application is already deployed and operated. We could, for instance, deploy an updated version of the application in parallel and decommission older versions once the updated version is considered to run correctly. In the context of this paper we focus on the first part of our methodology *(Develop App & Prepare Ops),* especially on the automated and manual steps to define, generate, update, and evaluate the DevOps Spec. For this purpose, the next Sect. 4 proposes a language to be used to create and maintain a DevOps Spec.

4 DevOpSlang – A Language to Bridge the Gap

Based on the need to implement DevOps in practice (Sect. 1 and Sect. 2) we introduced a DevOps-centric methodology to deploy and operate applications (Sect. 3). We defined the notion of a *DevOps Spec* (Definition 1) as a key artifact to enable the implementation of our methodology. However, in the methodology's context we do not define how such a DevOps Spec is structured technically. This is absolutely necessary to implement the methodology in practice and to implement automated processes in particular. In this section we propose *DevOpSlang,* a new domain-specific language to be used for implementing DevOps Specs. The most important goal of DevOpSlang in conjunction with our methodology (Sect. 3) is to enable and support efficient collaboration between developers and operations, leading to automated processes as much as possible. Technically, DevOpSlang is a domain-specific language based on JavaScript Object Notation (JSON) [4]. We use JSON Schema [8] to define a formal schema for DevOpSlang that may be used for validation purposes. `Devopsfiles` are the technical artifacts rendered using DevOpSlang:

Definition 3 (`Devopsfile`). *A* `Devopsfile` *is the technical implementation of a DevOps Spec using DevOpSlang. A* `Devopsfile` *orchestrates arbitrary artifacts (Unix shell commands, Chef scripts, etc.) to implement operations.*

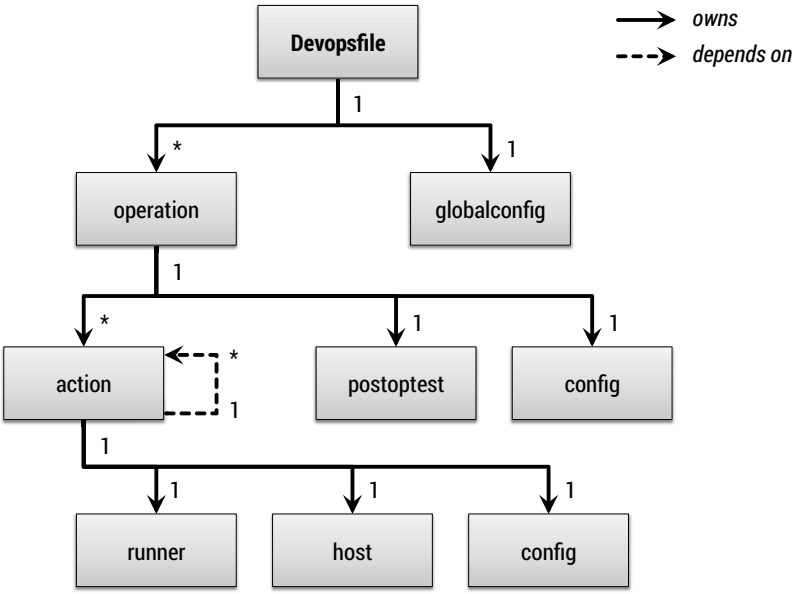

Fig. 3. Structure of a `Devopsfile` based on DevOpSlang

Figure 3 shows the structure of a `Devopsfile` based on DevOpSlang. The complete JSON schema definition of DevOpSlang is publicly available on GitHub[11]. An arbitrary number of operations can be defined. A single operation is implemented by a collection of actions that may depend on each other. Each action may implement an individual step of an operation. To make an action executable a *runner* is used. For instance, an operation may consist of two actions. The first one may be a Ruby script, using a *Ruby runner* to execute the script. The second one may be a single Unix shell command, using a command runner to execute the command. This makes the runners to be the actual workers to execute operations defined in a `Devopsfile`.

Definition 4 (Runner). *A* runner *is an executable enabler in the context of a* runner framework (Fig. 4)*. It enables the execution of a certain action defined in a* `Devopsfile`.

An architecture overview of a runner framework that may be used to run such operations is shown in Fig. 4. Runners that are stored in the *runner repository* are reusable by different actions implementing operations in different `Devopsfiles`. However, highly application-specific runners can be implemented and stored inside the runner repository, too. An operation is run by the *operation executor*. Each action of the operation is executed by the *action executor*, considering the dependencies among actions. To actually execute an action, the action executor retrieves

[11] GitHub project *DevOpSlang:* `http://github.com/jojow/devopslang`

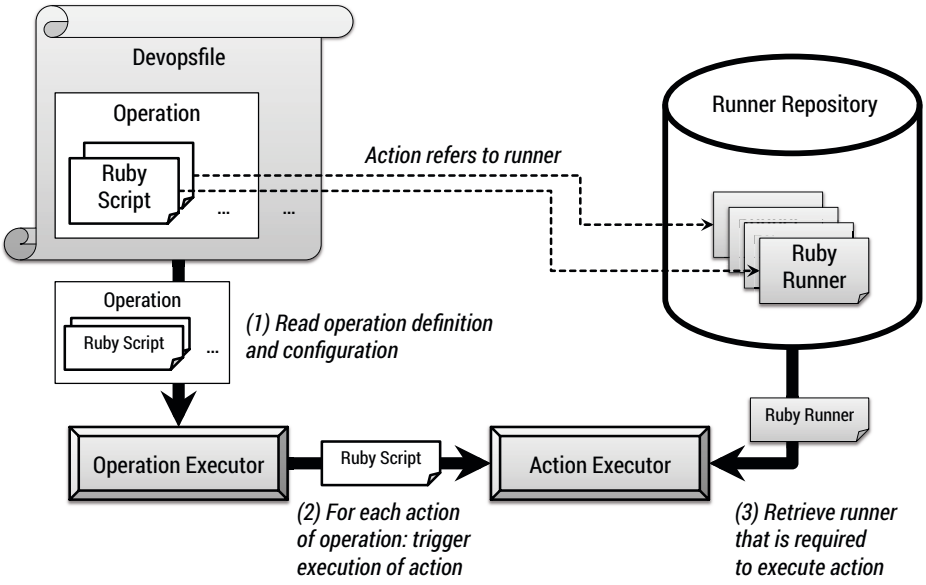

Fig. 4. Architecture overview of runner framework

the corresponding runner from the runner repository. [19] presents a similar architecture that may serve as a foundation for such a runner framework. Hosts such as VMs, containers, or platforms may be defined as a means to run different actions on different hosts. If no host is defined for an action, the invoker of the `Devopsfile` or the operation in particular determines where to run the action (e.g., localhost or a sandbox). In terms of configuration there is a global configuration at the top. Operations may have individual configurations that are merged with the the global configuration and may override parts of the global one. The same is true for configurations on the action level: they are merged with the operation's configuration and then with the global configuration, possibly overriding parts of them.

The following Sect. 5 presents the validation and evaluation of DevOpSlang in conjunction with our methodology (Sect. 3). We stick to the motivating scenario (Sect. 2) to create and refine `Devopsfiles` in an evolutionary manner as proposed by our methodology. This is enabled by the evolution of `Devopsfiles` and related runners.

5 Validation and Evaluation

The evaluation of our methodology in conjunction with DevOpSlang is twofold: the first part shows how `Devopsfiles` can evolve based on the changing architecture of an application and based on the collaboration between developers and operations personnel. The second part outlines the possibility to orchestrate multiple `Devopsfiles` recursively, so `Devopsfiles` remain maintainable even for

large applications. Beside the `Devopsfile` schema all `Devopsfiles` discussed in this section are completely and publicly available on GitHub[12]. We validated all `Devopsfiles` against the `Devopsfile` schema using the JSON Schema Validator[13].

5.1 `Devopsfile` Evolution

Based on the evolutionary developed chat application described in Sect. 2 the following listing shows an initial version of a `Devopsfile` for the application:

```
{
    "name": "chat-app",
    "version": "0.1",
    "author": "Johannes <wettinger@iaas.uni-stuttgart.org>",
    "description": "Automated deployment and operations for chat app
        ",

    "operations": {
        "build": {
            "actions": {
                "install-deps": {
                    "runner": "command-runner",
                    "comment": "Node.js 0.10+ must be installed",
                    "config": { "command": "npm install" }
                }
            }
        },
        "start": {
            "actions": {
                "chatapp": {
                    "runner": "command-runner",
                    "comment": "Node.js 0.10+ must be installed",
                    "config": { "command": "node app.js" }
                }
            }
        },
        "deploy": {
            "actions": {
                "build-app": {
                    "runner": "operation-runner",
                    "config": { "operation": "build" }
                },
                "start-app": {
                    "runner": "operation-runner",
                    "config": { "operation": "start" }
                }
            },
            "dependencies": [
                [ "start-app", "build-app" ]
            ]
        }
    }
}
```

Beside some meta data such as 'version' and 'author' this `Devopsfile` defines a 'start' operation consisting of a single action entitled 'chatapp'. This is a minimalist definition specifying the command to run the Node.js-based application. Similarly, a 'build' operation is defined to install the dependencies required to

[12] `Devopsfile` schema and sample `Devopsfiles`: `http://github.com/jojow/devopslang`

[13] JSON Schema Validator: `http://github.com/fge/json-schema-validator`

run the application. The 'deploy' operation in its initial version points to the operations 'build' and 'start'. Such initial definitions may be automatically derived from existing application descriptor files such as the `package.json`[14] file for Node.js-based applications. For the next iteration of the chat application the Node.js runtime solely is not enough. A database is used to store chat logs. Thus, we need to start MongoDB as an additional component before running the application ('chatapp' action depends on 'mongodb' action):

```
"start": {
    "actions": {
        "chatapp": { ... },
        "mongodb": {
            "runner": "command-runner",
            "comment": "MongoDB 2.6+ must be installed",
            "config": { "command": "mongod" }
        }
    },
    "dependencies": [
        [ "chatapp", "mongodb" ]
    ]
}
```

Up to now, we assume the middleware components such as the Node.js runtime and the MongoDB server are available already when running the 'deploy' operation. This might be true for some developer machines. However, on a freshly provisioned VM, for instance, these components need to be installed, too. Thus, the operation definition may be extended as follows to retrieve and install all components that are involved:

```
"deploy": {
    "actions": {
        "deploy-nodejs": {
            "runner": "command-runner",
            "config": {
                "command": "... && sudo apt-get install nodejs"
            }
        },
        "deploy-mongodb": {
            "runner": "command-runner",
            "config": {
                "command": "... && sudo apt-get install mongodb-org"
            }
        },
        "build-app": { ... },
        "start-app": { ... }
    },
    ...
}
```

Defining actions on the level of commands might be a good starting point because this is what developers typically use for creating the first prototypes and iterations of an application. However, the DevOps community publicly shares and maintains reusable artifacts such as Chef cookbooks to deploy middleware and application components. To increase the portability and reliability of operations it may make sense to reuse these artifacts instead of putting together a few platform-specific commands. The following listing shows how this can be done using a different runner for the 'mongodb' action:

[14] `Package.json` description: `http://www.npmjs.org/doc/json.html`

```
"deploy": {
    "actions": {
        ...
        "deploy-mongodb": {
            "runner": "chef-solo-runner",
            "config": {
                "files": { "mongodb.tgz": "http://.../mongodb.tgz" },
                "runlist": [ "recipe[mongodb::default]" ]
            }
        }
    },
    ...
    "postoptest": {
        "runner": "command-runner",
        "config": {
            "command": "export RESCODE=$(curl -sL -w \"%{http_code}\\
                n\" \"http://localhost:3000\" -o /dev/null) && [[ \"
                $RESCODE\" == \"200\" ]] && true || false"
        }
    }
}
```

Moreover, a 'postoptest' is defined. It implements a test case that is executed directly after the operation execution finished. This is to check whether the operation was executed successfully. In this example we simply send an HTTP request to our application and check if the response code is 200 (OK).

All `Devopsfile` iterations discussed so far assume that the whole application is deployed to a single host such as a VM. To address scalability and performance issues the application needs to be deployed in a distributed manner. As a first step, the Node.js runtime and the MongoDB server are running on two distinct VMs. Moreover, additional actions need to be included in the operation definition to cover the provisioning of these VMs. This further improves the completeness of the `Devopsfile`. The following listing provides a small extract of a more advanced iteration of the `Devopsfile`[15] to provision a new VM:

```
"deploy": {
    "actions": {
        "provision-app-vm": {
            "runner": "js-sandbox-runner",
            "config": {
                "hostname": "app-vm",
                "files": { "ec2-provision.js": "http://ops-artifact-
                    store/aws-management/ec2-provision.js" },
                "include": [ "ec2-provision.js" ]
            }
        },
        ...
        "deploy-nodejs": {
            "host": "app-vm",
            ...
        },
        ...
    },
    ...
}
```

In this iteration of the `Devopsfile` we assume that the application is always deployed to VMs at Amazon's EC2 platform. However, this could be changed

[15] `Devopsfile` v8: `http://goo.gl/mda8c4`

easily by using provider abstraction libraries such as fog[16] to implement more generic provisioning scripts or corresponding runners. In any case, actions of an operation need to be annotated with a host for a distributed deployment, so it is clear where the action should run. Further iterations of the `Devopsfile`[17] may define additional management operations such as an 'expose' operation to explicitly make the application available to the outside world. Technically, this could be a script to configure a security group of an Amazon EC2 VM, opening port 80 for inbound traffic to retrieve HTTP requests.

We have seen that DevOpSlang provides an efficient means to change the level of abstraction implementing operations seamlessly. Moreover, different abstraction levels may be combined consistently such as a 'deploy' operation consisting of actions on the level of Unix shell commands and actions using portable Chef cookbooks.

5.2 Recursive Orchestration of `Devopsfiles`

As an application grows, the `Devopsfile` may get huge and thus more difficult to maintain. To avoid such issues the application may be split into different components that own their individual `Devopsfiles.` The 'operation-runner' may be utilized to transparently invoke operations defined in other `Devopsfiles` as shown in the following listing. This approach enables the recursive orchestration of `Devopsfiles` to keep them maintainable in size and thus enabling separation of concerns.

```
"deploy": {
    "actions": {
        ...
        "deploy-app-core": {
            "runner": "operation-runner",
            "config": {
                "Devopsfile": "./core/Devopsfile",
                "operation": "deploy"
            }
        },
        ...
    },
    ...
}
```

6 Related Work

Our work is related to similar approaches in the field of Cloud computing that introduce a domain-specific language to deploy and operate applications in an automated manner. On the IaaS level approaches such as Amazon CloudFormation[18] or OpenStack Heat[19] are used to orchestrate infrastructure resources (VMs, storage, network, etc.). Moreover, middleware and application components can be

[16] fog library: `http://fog.io`
[17] `Devopsfile` v9: `http://goo.gl/b6Fu0f`
[18] Amazon CloudFormation: `http://aws.amazon.com/cloudformation`
[19] OpenStack Heat: `http://wiki.openstack.org/wiki/Heat`

stacked and orchestrated using application topologies based on Ubuntu Juju[20], Amazon OpsWorks [16], Blueprints [15], or enterprise topology graphs [3]. The Topology and Orchestration Specification for Cloud Applications (TOSCA) [12] is an emerging standard to define portable application topologies. However, some of these approaches are bound to specific providers or tools (CloudFormation, OpsWorks, Juju, etc.); some are focused on defining the higher-level structure of an application (TOSCA, Blueprints, etc.), so implementing automation requires additional imperative logic such as build plans, or conventions for declarative processing have to be defined. Others focus on prescribing fine-grained technical mechanisms how to implement automation, mainly considering operations-related aspects. Thus, they can hardly be used as a means of collaboration to fill the DevOps gap. Furthermore, there are modeling languages such as UML deployment diagrams [13] that may be a nice fit for collaboration purposes, but corresponding models are not executable.

DevOpSlang aims to fill this gap as a language to improve DevOps collaboration and to enable comprehensive automation based on the fact that operations defined in `Devopsfiles` are executable. However, to implement a framework to process `Devopsfiles`, the aforementioned and other existing approaches [19,5] may be used and combined to enable the automated run of operations. Furthermore, the DevOps community proposes several domain-specific languages centered around tools such as Puppet [9], CFEngine [20], and Chef [11]. However, these languages focus on the configuration of lower-level resources such as middleware and application components installed on VMs. Moreover, they are bound to a specific tool such as Chef or Puppet. Consequently, they are less appropriate as a holistic means of collaboration and can hardly be used to automate deployment and operations of applications based on an arbitrary combination of tools and artifacts. However, they may perfectly complement DevOpSlang to implement actions using these lower-level domain-specific languages.

7 Conclusions

In this paper we introduced a new domain-specific language called *DevOpSlang* in conjunction with a methodology to enable the implementation of DevOps. The language serves as an efficient means of collaboration and provides the foundation to automate deployment and operations of an application. We evaluated both DevOpSlang and the methodology based on an evolutionary emerging application described in our motivating scenario. In terms of future work we plan to implement a runner framework to process `Devopsfiles` based on DevOpSlang. We further plan to implement mechanisms to generate `Devopsfile` skeletons based on existing descriptor files, evaluate `Devopsfiles` automatically, and make suggestions how to improve a given `Devopsfile.` Moreover, our goal is to provide alternative renderings of `Devopsfiles` based on XML and YAML.

[20] Ubuntu Juju: `http://juju.ubuntu.com`

Acknowledgments. This work was partially funded by the BMWi project CloudCycle (01MD11023).

References

1. Manifesto for Agile Software Development (2001), http://agilemanifesto.org
2. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A View of Cloud Computing. Communications of the ACM 53(4), 50–58 (2010)
3. Binz, T., Fehling, C., Leymann, F., Nowak, A., Schumm, D.: Formalizing the Cloud through Enterprise Topology Graphs. In: Proceedings of 2012 IEEE International Conference on Cloud Computing. IEEE Computer Society Conference Publishing Services (2012)
4. Ecma International: The JSON Data Interchange Format (2013), http://json.org
5. Fischer, J., Majumdar, R., Esmaeilsabzali, S.: Engage: A Deployment Management System. SIGPLAN Not. 47(6), 263–274 (2012)
6. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Addison-Wesley Professional (2010)
7. Humble, J., Molesky, J.: Why Enterprises Must Adopt Devops to Enable Continuous Delivery. Cutter IT Journal 24 (2011)
8. Internet Engineering Task Force: JSON Schema, http://json-schema.org
9. Loope, J.: Managing Infrastructure with Puppet. O'Reilly Media, Inc. (2011)
10. Mell, P., Grance, T.: The NIST Definition of Cloud Computing. National Institute of Standards and Technology (2011)
11. Nelson-Smith, S.: Test-Driven Infrastructure with Chef. O'Reilly Media, Inc. (2013)
12. OASIS: Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0, Committee Specification 01 (2013), http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html
13. OMG: Unified Modeling Language (UML), Version 2.4.1 (2011)
14. Oppenheimer, D., Ganapathi, A., Patterson, D.A.: Why do internet services fail, and what can be done about it? In: USENIX Symposium on Internet Technologies and Systems, Seattle, WA, vol. 67 (2003)
15. Papazoglou, M., van den Heuvel, W.: Blueprinting the Cloud. IEEE Internet Computing 15(6), 74–79 (2011)
16. Rosner, T.: Learning AWS OpsWorks. Packt Publishing Ltd. (2013)
17. Shamow, E.: Devops at Advance Internet: How We Got in the Door. IT Journal, 14 (2011)
18. Wettinger, J., Andrikopoulos, V., Strauch, S., Leymann, F.: Characterizing and Evaluating Different Deployment Approaches for Cloud Applications. In: Proceedings of the IEEE International Conference on Cloud Engineering (IEEE IC2E 2014), Boston, Massachusetts, USA, March 10-14. IEEE Computer Society (2014)
19. Wettinger, J., Binz, T., Breitenbücher, U., Kopp, O., Leymann, F., Zimmermann, M.: Unified Invocation of Scripts and Services for Provisioning, Deployment, and Management of Cloud Applications Based on TOSCA. In: Proceedings of the 4th International Conference on Cloud Computing and Services Science. SciTePress (2014)
20. Zamboni, D.: Learning CFEngine 3: Automated System Administration for Sites of Any Size. O'Reilly Media, Inc. (2012)