Compensation-Based vs. Convergent Deployment Automation for Services Operated in the Cloud

Johannes Wettinger, Uwe Breitenbücher, and Frank Leymann

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

Abstract. Leading paradigms to develop and operate applications such as continuous delivery, configuration management, and the merge of development and operations (DevOps) are the foundation for various techniques and tools to implement automated deployment. To expose such applications as services (SaaS) to users and customers these approaches are typically used in conjunction with Cloud computing to automatically provision and manage underlying resources such as storage or virtual machines. A major class of these automation approaches follows the idea of converging toward a desired state of a resource (e.g., a middleware component deployed on a virtual machine). This is achieved by repeatedly executing idempotent scripts until the desired state is reached. Because of major drawbacks of this approach, we present an alternative deployment automation approach based on compensation and fine-grained snapshots using container virtualization. We further perform an evaluation comparing both approaches in terms of difficulties at design time and performance at runtime.

Keywords: Compensation, Snapshot, Convergence, Deployment Automation, DevOps, Cloud Computing.

1 Introduction

Cloud computing [10,21,7] can be used in different setups such as public, private, and hybrid Cloud environments to efficiently run a variety of kinds of applications, exposed as services (SaaS). Prominent examples are Web applications, back-ends for mobile applications, and applications in the field of the "internet of things", e.g., to process large amounts of sensor data. Users of such services based on Cloud applications expect high availability and low latency when interacting with a service. Consequently, the applications need to scale rapidly and dynamically to serve thousands or even millions of users properly. To implement scaling in a cost-efficient way the application has to be elastic, which means that application instances are provisioned and decommissioned rapidly and automatically based on the current load. Cloud providers offer on-demand self-service capabilities, e.g., by providing corresponding APIs to provision and manage resources such as virtual machines, databases, and runtime environments. These

X. Franch et al. (Eds.): ICSOC 2014, LNCS 8831, pp. 336–350, 2014.

capabilities are the foundation for scaling applications and implementing elasticity mechanisms to run them efficiently in terms of costs. Moreover, users of services operated in the Cloud expect fast responses to their changing and growing requirements as well as fixes of issues that occur. Thus, underlying applications need to be redeployed frequently to production, e.g., several times a week. Development and operations need to be tightly coupled to enable such frequent redeployments. *DevOps* [5,3] aims to eliminate the split between developers and operations to automate the complete deployment process from the source code in version control to the production environment. Today, the DevOps community follows a leading paradigm to automate the deployment, namely to implement *idempotent* scripts to *converge* resources toward a desired state. Because this approach has some major drawbacks we propose an alternative approach based on compensation. Our major contributions are presented in this paper:

- We present the fundamentals of state-of-the-art deployment automation approaches and point out existing deficiencies and difficulties
- We propose an alternative approach to implement deployment automation based on compensation on different levels of granularity to improve the efficiency and robustness of script execution
- We further show how compensation actions can be automatically derived at runtime to ease the implementation of compensation based on snapshots
- We evaluate the compensation-based deployment automation approach based on different kinds of applications operated in the Cloud and exposed as services

The remainder of this paper is structured as follows: based on the fundamentals showing state-of-the-art deployment automation approaches (Sect. 2), focusing on convergent deployment automation, we present the problem statement in Sect. 3. To tackle the resulting challenges, Sect. 4 presents approaches to implement compensation-based deployment automation. Our evaluation of compensation-based deployment automation is presented and discussed in Sect. 5 and Sect. 6. Finally, Sect. 7 presents related work and Sect. 8 concludes this paper.

2 Fundamentals

The automated deployment of middleware and application components can be implemented using general-purpose scripting languages such as Perl, Python, or Unix shell scripts. This is what system administrators and operations personnel were primarily using before the advent of DevOps tools providing domain-specific languages [2] to create scripts for deployment automation purposes. We stick to the following definition 1 for a script to be used for automating operations, especially considering deployment automation:

Definition 1 (Operations Script). *An* operations script *(in short* script*) is an arbitrary executable to deploy and operate middleware and application components by modifying the state of resources such as virtual machines. Such a state*

modification could be the installation of a software package, the configuration of a middleware component, etc. A script consists of a sequence of actions such as command statements that implement state modifications.

Technically, a script can be implemented imperatively (e.g., using general-purpose scripting languages) or declaratively (e.g., using domain-specific languages [2]). In case of using a declarative language, the concrete imperative command statements and their sequential ordering has to be derived in a preprocessing step before the actual execution. As an alternative to scripts, compiled programs could be used, based on portable general-purpose programming languages such as Java. However, this would decrease the flexibility and may have performance impact, because the source code has to be compiled after each change.

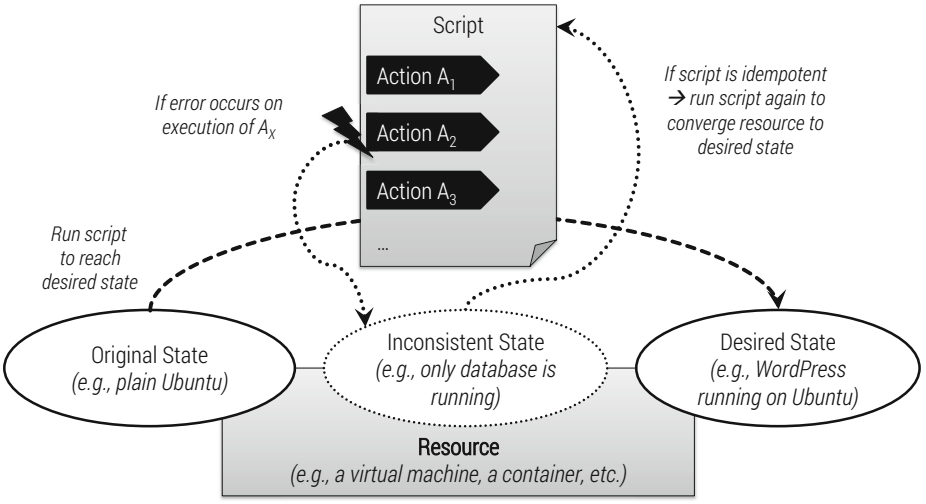

Fig. 1. Script to transfer or converge a resource toward a desired state

Fig. 1 shows the basic usage of scripts: several actions A_x are specified in the script that are command statements (install package "mysql", create directory "cache", etc.) to transfer a particular resource such as a virtual machine (VM) or a container [15,17] into a desired state. For instance, the original state of the virtual machine could be a plain Ubuntu operating system (OS) that is installed, whereas the desired state is a VM that runs a WordPress blog[1] on the Ubuntu OS. Consequently, a script needs to execute commands required to install and configure all components (Apache HTTP server, MySQL database server, etc.) that are necessary to run WordPress on the VM.

This is a straightforward implementation of deployment automation. However, this approach has a major drawback: in case an error occurs during the execution

[1] WordPress: `http://www.wordpress.org`

of the script, the resource is in an unknown, most probably inconsistent state. For instance, the MySQL database server is installed and running, but the installation of Apache HTTP server broke, so the application is not usable. Thus, either manual intervention is required or the whole resource has to be dropped and a new resource has to be provisioned (e.g., create a new instance of a VM image) to execute the script again. This is even more difficult in case the *original state* is not captured, e.g., using a VM snapshot. In this case manual intervention is required to restore the original state. This is error-prone, time-consuming, costly, and most importantly even impossible in cases where the original state is not documented or captured. Since errors definitely occur in Cloud environments, e.g., if the network connection breaks during the retrieval of a software package, it is a serious challenge to implement full and robust deployment automation.

This is why the DevOps community provides techniques and tools to implement *convergent deployment automation:* its foundation is the implementation of *idempotent* scripts [4], meaning the script execution on a particular resource such as a VM can be repeated arbitrarily, always leading to the same result if no error occurs; if an error occurs during execution and the desired state is not reached (i.e., resource is in an unknown state) the script is executed again and again until the desired state is reached. Thus, idempotent scripts can be used to *converge* a particular resource toward a desired state without dropping the resource as shown in Fig. 1. With this approach the resource does not get stuck in an inconsistent state. DevOps tools such as Chef [11] provide a declarative domain-specific language to define idempotent actions (e.g., Chef resources[2]) that are translated to imperative command statements at runtime, depending on the underlying operating system. For instance, the declarative statement "ensure that package `apache2` is installed" is translated to the following command on an Ubuntu OS: `apt-get -y install apache2`; on a Red Hat OS, the same declarative statement is translated to `yum -y install apache2`. Imperative command statements can also be expressed in an idempotent manner. For instance, a simple command to install the Apache HTTP server on Ubuntu (`apt-get -y install apache2`) is automatically idempotent because if the package `apache2` is already installed, the command will still complete successfully without doing anything. Other command statements need to be adapted such as a command to retrieve the content of a remote Git[3] repository: `git clone http://gitserver/my_repo`. This command would fail when executing it for a second time because the directory `my_repo` already exists. To make the command statement idempotent a minor extension is required that preventively deletes the `my_repo` directory: `rm -rf my_repo && git clone http://gitserver/my_repo`.

[2] Chef resources: `http://docs.opscode.com/resource.html`
[3] Git: `http://git-scm.com`

3 Problem Statement

As discussed in Sect. 2, convergent deployment automation makes the execution of scripts more robust. However, it may not be the most efficient approach to repeatedly execute the whole script in case of errors until the desired state is reached. Furthermore, this approach only works in conjunction with idempotent scripts. While in most cases it is possible to implement idempotent actions, it can be challenging and sophisticated to implement fully idempotent scripts without holding specific state information for each action that was executed. Typical examples include:

- An action to create a database or another external entity by name, so the second execution results in an error such as "the database already exists".
- An action that sends a non-idempotent request to an external service (e.g., a POST request to a RESTful API), so the second request most probably produces a different result.
- An action to clone a Git repository, so the second execution fails because the directory for the repository already exists in the local filesystem.

Consequently, major efforts need to be invested to create and test idempotent scripts to ensure their robustness. Moreover, issues may occur, preventing a resource from converging toward the desired state, so the resource hangs in an unknown state. As an example, Ubuntu's apt package manager[4] may crash during the installation of software packages (e.g., in case of a dropped network connection or a memory bottleneck), so the lock file (ensuring that apt is not running multiple times in parallel) was not removed. In this case the lock file needs to be removed manually; otherwise all subsequent executions of apt fail. Sophisticated monitoring is required to detect such issues at runtime.

In the following Sect. 4 we present *compensation-based deployment automation* on the level of scripts and actions as an alternative to the leading convergent deployment automation approach. Our goal is to increase efficiency and robustness without additional overhead. Moreover, our approach aims to reduce the complexity of creating scripts by allowing arbitrary non-idempotent actions in it.

4 Compensation-Based Deployment Automation

The main idea of *compensation* is to implement an *undo* strategy that is run in case an error occurs during the execution of a particular script or action. Depending on the level of implementing compensation, either *compensation scripts* can be implemented to roll back the work performed by a particular script or *compensation actions* can be implemented to undo a single action. In the following Sect. 4.1 and Sect. 4.2, we discuss how compensation can be implemented on these two different levels. Moreover, Sect. 4.3 presents an approach to automatically derive compensation actions at runtime based on fine-grained snapshots.

[4] Ubuntu's apt package manager: `http://packages.ubuntu.com/trusty/apt`

4.1 Compensation on the Level of Scripts

To implement compensation on the level of scripts, a *compensation script* has to be implemented for each script to compensate the work performed by the script itself. For instance, if the script has installed parts of the WordPress application, the compensation script needs to uninstall these parts in case an error occurs during the installation. Then, the script runs again. Obviously, a proper retry strategy needs to be implemented such as defining the maximum number of retries to deal with situations where a certain issue persists. If the maximum number of retries is reached the compensation script is executed for the last time and the error gets escalated to the invoker of the script, e.g., a deployment plan implemented as a workflow [20].

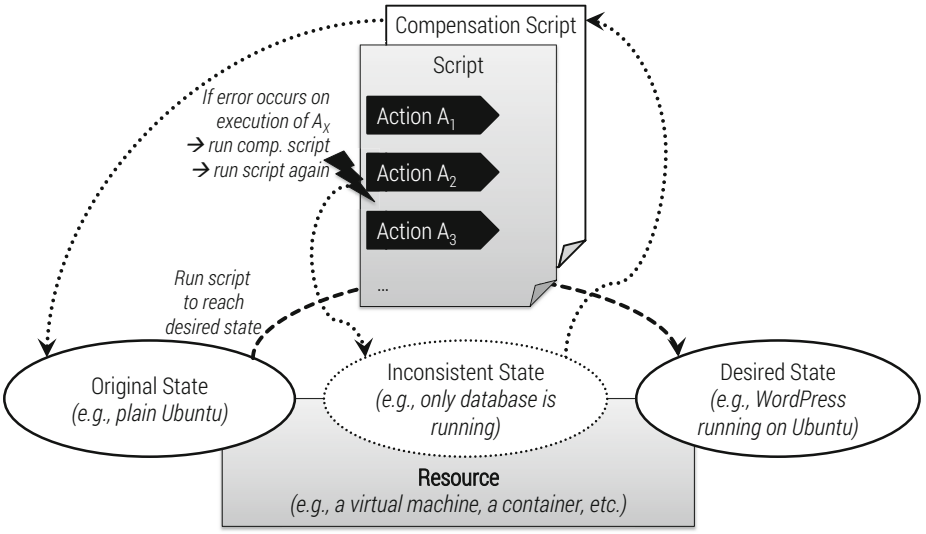

Fig. 2. Compensation script to undo the work performed by a script

Fig. 2 outlines how this coarse-grained way of implementing compensation works, considering a script as an atomic entity in terms of compensation. As an example, the following listing shows an extract of a Unix shell script to create a database and to retrieve the content of a Git repository:

```
1 #!/bin/sh
2
3 ...
4
5 echo "CREATE DATABASE $DBNAME" | mysql -u $USER
6
7 git clone http://gitserver/my_repo
```

The following extract of a Unix shell script shows how a corresponding compensation script could be implemented:

```
#!/bin/sh

echo "DROP DATABASE $DBNAME" | mysql -u $USER

rm -rf my_repo

...
```

The challenge of implementing a compensation script is that the current state of the corresponding resource is unknown, depending on the point in time the error occurred during script execution. Consequently, the compensation script has to consider a lot of potential problems that may occur. This makes a compensation script hard to implement and to maintain. Thus, the following Sect. 4.2 presents a more fine-grained approach to implement compensation on the level of actions.

4.2 Compensation on the Level of Actions

In contrast to script-level compensation as discussed before, a *compensation action* is implemented and attached to each action defined in the script: if an error occurs during the execution of action A_x, the corresponding compensation action CA_x is run. Then, A_x is executed again to eventually continue with the following actions. Similar to the script-level compensation a proper retry strategy needs to be implemented. For instance, the maximum number of retries for rerunning a particular action A_x needs to be defined. Once this number is reached all previous actions need to be compensated by running CA_x, CA_{x-1}, ..., CA_1. Then, the error gets escalated to the invoker of the script, e.g., a workflow. The invoker may perform some clean-up work, e.g., removing VMs that are in an unknown state. Compared to script-level compensation (Sect. 4.1) and the convergent approach (Sect. 2) this behavior is more efficient in terms of execution time because the script is not compensated and rerun as a whole; only the affected action gets compensated and is then executed again.

Fig. 3 outlines how this fine-grained compensation approach works. Technically, compensation actions CA_x can be defined and attached on the level of command statements. For instance, the compensation action CA_3 attached to action A_3 that clones a Git repository (`git clone http://gitserver/my_repo`) could be the following to remove the cloned repository from the filesystem: `rm -rf my_repo`. Another example is sending a PUT request to a RESTful API to create a resource. For instance, the compensation action may have to send one or several DELETE requests to the API to remove the created resource and maybe other resources that were created as a result of the original PUT request. The following extract of an extended Dockerfile[5] (sequence of Unix shell commands) shows how compensation actions (`COMPENSATE` statements) can be defined and attached to actions (`RUN` statements) using cURL[6], a simple command-line HTTP client.

[5] Dockerfile reference: `http://docs.docker.io/reference/builder`
[6] cURL: http://curl.haxx.se

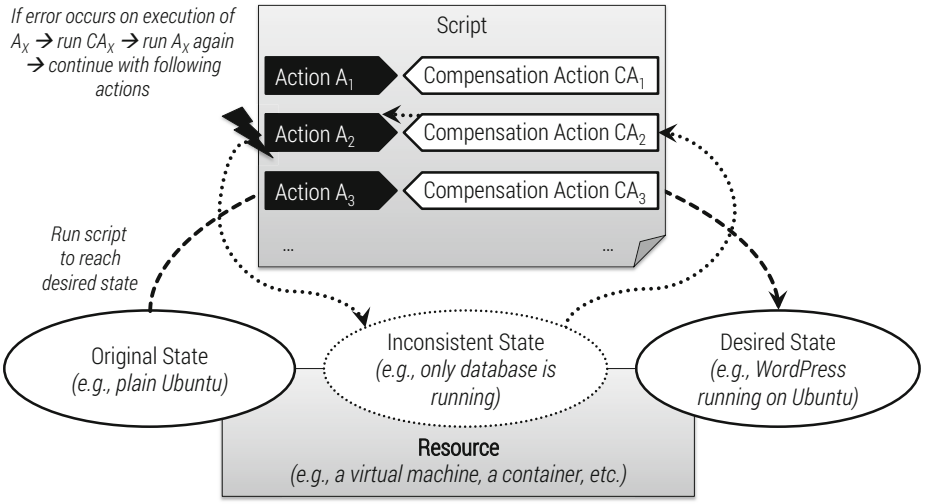

Fig. 3. Compensation actions to undo the work of individual actions

```
1 ...
2
3 RUN curl -H "Content-Type: application/json" -X PUT --data "@$ID.json"
      -u $USER:$PASSWORD http://.../entries/$ID
4
5     COMPENSATE curl -X DELETE -u $USER:$PASSWORD http://.../entries/
          $ID
6
7 RUN ...
8     COMPENSATE ...
```

Compared to compensation scripts, compensation actions are easier to implement because only the scope of one particular command statement needs to be considered. However, it may be tedious to manually implement compensation actions for each particular action defined in a script. Thus, the following Sect. 4.3 presents a compensation approach to dynamically generate compensation actions at runtime based on fine-grained snapshots.

4.3 Snapshot-Based Compensation

Action-level compensation as discussed before provides some advantages over script-level compensation because only the scope of a single action has to be considered when implementing the compensation logic. However, for scripts with a huge number of actions, many individual compensation actions have to implemented and attached to the script. Because their creation is time-consuming and error-prone, plus they are hard to maintain, we need a means to automatically generate compensation actions. Fig. 4 shows how fine-grained snapshots can be used to capture and restore an arbitrary state of a resource. This technique can be used to create a snapshot S_0 of the original state and an additional snapshot S_1, S_2, ... for each action A_1, A_2, ... that was executed successfully. Moreover,

a compensation action CA_x for each action A_x gets generated automatically at runtime to restore the snapshot S_{x-1} that was created after the previous action has been executed successfully.

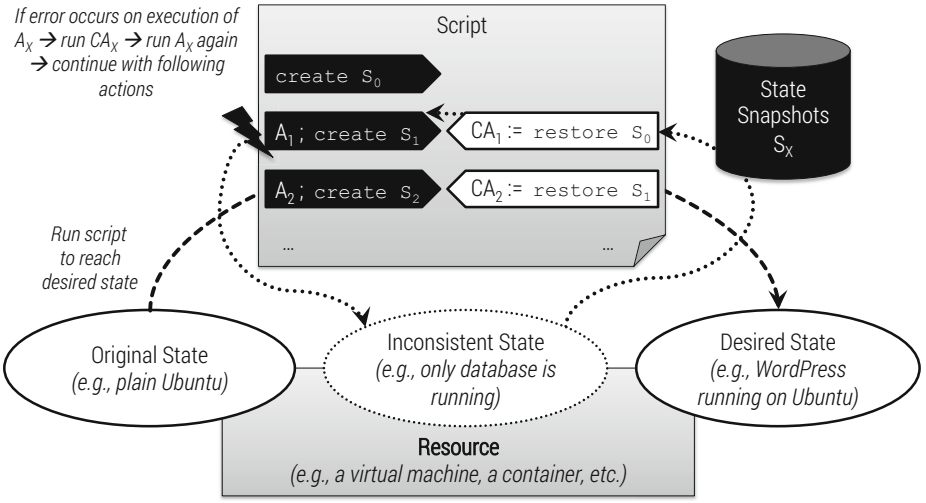

Fig. 4. Snapshot-based compensation of individual actions

Of course, the snapshot-based compensation approach can also be implemented on the level of scripts as discussed in Sect. 4.1. However, this is only feasible if all actions of a script can be compensated using snapshots and do not need custom compensation logic such as sending specific requests to external resources. In this case custom compensation actions have to be attached to the affected actions. Consequently, the snapshot-based compensation approach can be used as a fallback to generate compensation actions at runtime for all actions that do not have a custom compensation action attached. This speeds up the development of scripts because compensation actions have to be implemented only for actions that cannot rely on the snapshot-based approach to compensate their work.

5 Evaluation

Conceptually, we discussed multiple variants of compensation-based deployment automation in Sect. 4. Our evaluation compares the compensation-based approach with convergent deployment automation in terms of performance impact at runtime and difficulties at design time. We implemented the automated deployment of three different kinds of open-source Web applications, covering a set of wide-spread technologies and middleware to implement such applications.

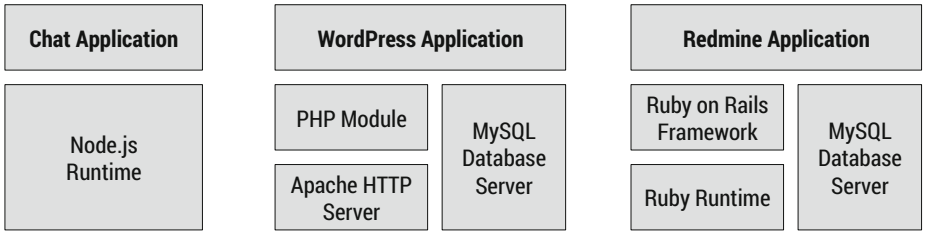

Fig. 5. Architectures of three Web applications

Fig. 5 outlines the architectures of the applications, namely a simple Chat Application[7] based on Node.js, the Ruby-based project management and bug-tracking tool Redmine[8], and WordPress[9] to run blogs based on PHP. Each application is deployed on a clean VM (1 virtual CPU clocked at 2.8 GHz, 2 GB of memory) on top of the VirtualBox hypervisor[10], running a minimalist installation of the Ubuntu OS, version 14.04.

Table 1. Measurements in clean environment and their standard deviation σ

Application	Average Duration (in sec.)	Average Memory Usage (in MB)
Clean Deployments Using Chef:		
WordPress	211 ($\sigma = 114$)	333 ($\sigma = 2$)
Chat App	265 ($\sigma = 37$)	248 ($\sigma = 1$)
Redmine	1756 ($\sigma = 191$)	1479 ($\sigma = 4$)
Clean Deployments Using Docker:		
WordPress	71 ($\sigma = 10$)	548 ($\sigma = 1$)
Chat App	249 ($\sigma = 7$)	478 ($\sigma = 2$)
Redmine	741 ($\sigma = 17$)	583 ($\sigma = 5$)

Technically, we use Chef solo[11] version 11.12.4 as a configuration management solution to implement convergent deployment automation for all three applications based on idempotent scripts (Chef cookbooks). Furthermore, we use Docker[12] version 0.9.1 as a container virtualization solution to implement action-level compensation based on fine-grained container snapshots. Consequently, we

[7] Chat Application: `http://github.com/geekuillaume/Node.js-Chat`
[8] Redmine: `http://www.redmine.org`
[9] WordPress: `http://www.wordpress.org`
[10] VirtualBox: `http://www.virtualbox.org`
[11] Chef solo: `http://docs.opscode.com/chef_solo.html`
[12] Docker: `http://www.docker.io`

Table 2. Measurements in disturbed environment and their standard deviation σ

Application	Average Duration (in sec.)	Average Memory Usage (in MB)
Disturbed Deployments Using Chef:		
WordPress	182 ($\sigma = 84$)	334 ($\sigma = 3$)
Chat App	394 ($\sigma = 78$)	237 ($\sigma = 5$)
Redmine	1948 ($\sigma = 262$)	1479 ($\sigma = 2$)
Disturbed Deployments Using Docker:		
WordPress	74 ($\sigma = 6$)	779 ($\sigma = 1$)
Chat App	258 ($\sigma = 36$)	576 ($\sigma = 59$)
Redmine	991 ($\sigma = 120$)	1260 ($\sigma = 268$)

implemented scripts as Dockerfiles (sequence of Unix shell commands) that do exactly the same as the Chef cookbooks created before, but without being idempotent. Based on these implementations we run the deployment process of each application using both Chef and Docker in two different environments: the *clean* environment allows the deployment process to run without any errors; the *disturbed* environment emulates networking issues and memory bottlenecks by blocking TCP connections and killing system processes. We run each of the 24 combinations five times, so table 1 and table 2 present the average duration, the average memory usage, and their standard deviation. Each run is triggered using the same setup without any pre-cached container images or beneficial preparations. In the following Sect. 6 we discuss the results of our evaluation based on the measurements presented in table 1 and table 2 as well as the experience we gained during the implementation of the scripts following different deployment automation approaches.

6 Discussion

By analyzing the measurements presented in Sect. 5 we see that the compensation-based deployment automation approach with snapshots on the level of actions based on Docker consistently has a better performance in terms of deployment duration than the convergent approach based on Chef. This shows that repetitively executing an idempotent script to reach the desired state is more time-consuming than using a compensation-based approach on the level of actions. Moreover, the convergent approach may require more resources because declarative configuration definitions such as Chef cookbooks need to be compiled to imperative command statements at runtime. However, especially for deployment processes that have a shorter duration the memory consumption for convergent deployment automation is less compared to the compensation-based approach. This manifests the overhead of a snapshot-based approach where fine-grained,

incremental snapshots are cached to quickly restore the state captured after the last successfully executed action. This happens preventively, even in case the snapshots are not used, e.g., if no error occurs (clean environment). For longer-running deployment processes with more memory consumption in general such as the one of Redmine this overhead becomes less relevant, so in some cases such as the Docker-based deployment of Redmine the memory usage is even less compared to the corresponding Chef-based deployment.

In a disturbed environment that may be similar to an error-prone Cloud environment, where network issues appear and memory bottlenecks occur, the gap between the compensation-based and the convergent approach is significantly larger in terms of deployment duration. In this case compensation clearly outperforms convergence. Considering the design and implementation of scripts the compensation-based scripts and actions are easier to implement because they do not have to be idempotent as in the convergent approach. Moreover, most compensation actions can be automatically generated at runtime based on snapshots, so the implementation of custom compensation actions is not necessary for most actions. Fine-grained snapshots are also a convenient tool when developing, testing, and debugging scripts: snapshots can be created at any point in time to capture a working state and build upon this state, always having the possibility to quickly restore this state. Without using snapshots the whole script has to be executed for each test run. This can be time-consuming in case of more complex scripts that do not terminate after a few seconds already.

7 Related Work

Today, compensation techniques for deploying and managing infrastructure resources, middleware, and application components are mainly used by workflows on the orchestration level: workflows or plans based on standardized languages such as BPMN [14] or BPEL [12] are used on a higher level to coordinate the execution of scripts, API calls etc. [1,6,20]. Fig. 6 provides an overview of a possible interrelation between higher-level plans (e.g., BPEL workflows) defining the overarching flow of activities and the scripts SCR_y that actually manage the states ST_n of the underlying resources R_m that are involved. Compensation activities can be defined to compensate the work of another activity in case an error occurs [9,18,8]. In this example the *install* activity triggers the execution of script SCR_1 on R_1. If an error occurs, e.g., during the execution of the *install* activity, the attached *compensate* activity is triggered to run script SCR_2, which could be some kind of compensation script.

As an alternative to workflows, model-based approaches such as application topology models can be used to orchestrate scripts in a declarative manner. The Topology and Orchestration Specification for Cloud Applications (TOSCA) [13] is an emerging standard to specify such models. Moreover, there are provider- and tooling-specific approaches to build topology templates such as Amazon's

CloudFormation[13], OpenStack Heat[14], and Juju bundles[15]. All these approaches utilize scripts for lower-level tasks such as installing and configuring packages on VMs. Thus, the compensation-based deployment automation approaches presented in this paper can be combined with any of these higher-level approaches to ease the development of underlying scripts and to enhance the overall efficiency and robustness at runtime. Previous work [19,20] shows how to implement separation of concerns for plans that invoke and orchestrate scripts and services.

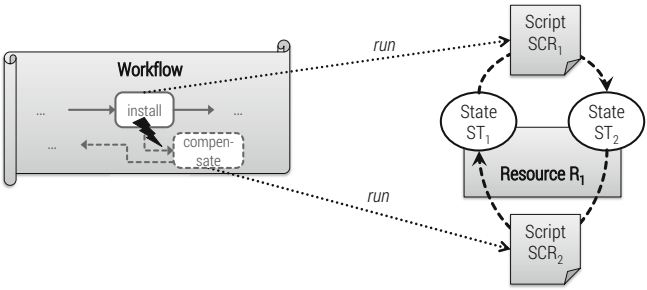

Fig. 6. Activities and compensation activities in workflows

8 Conclusions

Convergent deployment automation based on idempotent scripts is the leading paradigm implemented by wide-spread DevOps tools such as Puppet [16] and Chef [11]. We discussed the issues and deficiencies of this approach that occur at design time and at runtime: idempotent scripts are hard to test and to implement; due to their repetitive execution to converge a resource toward a desired state, the scripts' efficiency and robustness is not ideal. Based on these deficiencies we presented compensation-based deployment automation as an alternative to the convergent approach. We discussed how to implement compensation on the level of scripts and on the level of individual actions. Moreover, we showed how action-level compensation can be implemented using fine-grained snapshots to minimize the effort of implementing custom compensation actions. Our evaluation of compensation-based deployment automation compared to the convergent approach showed:

– Compensation is more robust, preventing a resource such as a VM from hanging in an inconsistent state without converging toward the desired state anymore

[13] Amazon CloudFormation: `http://aws.amazon.com/cloudformation`
[14] OpenStack Heat: `http://wiki.openstack.org/wiki/Heat`
[15] Juju bundles: `http://juju.ubuntu.com/docs/charms-bundles.html`

- Action-level compensation is always more efficient in terms of deployment duration
- Compensation may consume slightly more memory in some cases
- Compensation is easier to implement because scripts and actions do not have to be idempotent
- Snapshot-based compensation eases the development of scripts because compensation actions such as *restore snapshot* S_x can be automatically generated at runtime for most actions defined in a script

Currently, one major drawback of the compensation-based approach is its minimalist tooling support. We were using Docker as a container virtualization solution and Dockerfiles (construction plans for Docker containers) as scripts that can be compensated based on fine-grained container snapshots. In terms of future work we plan to extend existing domain-specific languages such as the ones used by Chef, Puppet, and Docker to seamlessly integrate the compensation approaches discussed in this paper. For instance, Chef can be extended to capture and restore fine-grained container snapshots automatically in the background, moving away from the inefficient strategy of running the whole script again and again. Another approach would be to automatically generate Dockerfiles from Chef scripts and then use Docker to execute them based on Docker's compensation and snapshot capabilities. In addition to deployment we plan to extend the scope of our research to cover further lifecycle operations that are relevant after the deployment phase. Existing approaches such as Cloud Foundry[16] centered around the platform-as-a-service model may be the technical foundation to consider these lifecycle operations such as scaling certain application components. Furthermore, we plan to extend our evaluation, including additional measurements such as the disk storage used for storing snapshots.

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

References

1. Breitenbücher, U., Binz, T., Kopp, O., Leymann, F.: Pattern-based Runtime Management of Composite Cloud Applications. In: Proceedings of the 3rd International Conference on Cloud Computing and Services Science. SciTePress (2013)
2. Günther, S., Haupt, M., Splieth, M.: Utilizing Internal Domain-Specific Languages for Deployment and Maintenance of IT Infrastructures. Tech. rep., Very Large Business Applications Lab Magdeburg, Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg (2010)
3. Humble, J., Molesky, J.: Why Enterprises Must Adopt Devops to Enable Continuous Delivery. Cutter IT Journal 24 (2011)
4. Hummer, W., Rosenberg, F., Oliveira, F., Eilam, T.: Testing Idempotence for Infrastructure as Code. In: Eyers, D., Schwan, K. (eds.) Middleware 2013. LNCS, vol. 8275, pp. 368–388. Springer, Heidelberg (2013)

[16] Cloud Foundry: `http://cloudfoundry.org`

5. Hüttermann, M.: DevOps for Developers. Apress (2012)
6. Kopp, O., Binz, T., Breitenbücher, U., Leymann, F.: BPMN4TOSCA: A Domain-Specific Language to Model Management Plans for Composite Applications. In: Mendling, J., Weidlich, M. (eds.) BPMN 2012. LNBIP, vol. 125, pp. 38–52. Springer, Heidelberg (2012)
7. Leymann, F.: Cloud Computing: The Next Revolution in IT. In: Photogrammetric Week 2009. Wichmann Verlag (2009)
8. Liu, F., Danciu, V.A., Kerestey, P.: A Framework for Automated Fault Recovery Planning in Large-Scale Virtualized Infrastructures. In: Brennan, R., Fleck II, J., van der Meer, S. (eds.) MACE 2010. LNCS, vol. 6473, pp. 113–123. Springer, Heidelberg (2010)
9. Machado, G.S., Daitx, F.F., da Costa Cordeiro, W.L., Both, C.B., Gaspary, L.P., Granville, L.Z., Bartolini, C., Sahai, A., Trastour, D., Saikoski, K.: Enabling Rollback Support in IT Change Management Systems. In: IEEE Network Operations and Management Symposium, NOMS 2008, pp. 347–354. IEEE (2008)
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: Web Services Business Process Execution Language (BPEL) Version 2.0 (2007)
13. 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
14. OMG: Business Process Model and Notation (BPMN) Version 2.0 (2011)
15. Soltesz, S., Pötzl, H., Fiuczynski, M.E., Bavier, A., Peterson, L.: Container-based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors. ACM SIGOPS Operating Systems Review 41, 275–287 (2007)
16. Turnbull, J., McCune, J.: Pro Puppet. Apress (2011)
17. Vaughan-Nichols, S.J.: New Approach to Virtualization is a Lightweight. Computer 39(11), 12–14 (2006)
18. Weber, I., Wada, H., Fekete, A., Liu, A., Bass, L.: Automatic Undo for Cloud Management via AI Planning. In: Proceedings of the Workshop on Hot Topics in System Dependability (2012)
19. Wettinger, J., Behrendt, M., Binz, T., Breitenbücher, U., Breiter, G., Leymann, F., Moser, S., Schwertle, I., Spatzier, T.: Integrating Configuration Management with Model-Driven Cloud Management Based on TOSCA. In: Proceedings of the 3rd International Conference on Cloud Computing and Services Science. SciTePress (2013)
20. 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)
21. Wilder, B.: Cloud Architecture Patterns. O'Reilly Media, Inc. (2012)