Giannakopoulos *et al. Journal of Cloud Computing: Advances, Systems and Applications* (2018) 7:11
https://doi.org/10.1186/s13677-018-0112-9

Journal of Cloud Computing: Advances, Systems and Applications

RESEARCH Open Access

Cloud application deployment with transient failure recovery

Ioannis Giannakopoulos[1*], Ioannis Konstantinou[1], Dimitrios Tsoumakos[2] and Nectarios Koziris[1]

Abstract

Application deployment is a crucial operation for modern cloud providers. The ability to dynamically allocate resources and deploy a new application instance based on a user-provided description in a fully automated manner is of great importance for the cloud users as it facilitates the generation of fully reproducible application environments with minimum effort. However, most modern deployment solutions do not consider the error-prone nature of the cloud: Network glitches, bad synchronization between different services and other software or infrastructure related failures with transient characteristics are frequently encountered. Even if these failures may be tolerable during an application's lifetime, during the deployment phase they can cause severe errors and lead it to failure. In order to tackle this challenge, in this work we propose AURA, an open source system that enables cloud application deployment with transient failure recovery capabilities. AURA formulates the application deployment as a Directed Acyclic Graph. Whenever a transient failure occurs, it traverses the graph, identifies the parts of it that failed and re-executes the respective scripts, based on the fact that when the transient failure disappears the script execution will succeed. Moreover, in order to guarantee that each script execution is idempotent, AURA adopts a lightweight filesystem snapshot mechanism that aims at canceling the side effects of the failed scripts. Our thorough evaluation indicated that AURA is capable of deploying diverse real-world applications to environments exhibiting high error probabilities, introducing a minimal time overhead, proportional to the failure probability of the deployment scripts.

Keywords: Cloud application deployment, Resource configuration, Transient failure, Error-recovery, Filesystem snapshot

Introduction

The advent of the Cloud computing [1] era has been a key enabler for the migration of many applications from traditional, on-premise servers to public clouds, in order to fully exploit the advantages of the latter: Seemingly infinite resources, billed in a pay-as-you-go manner allow the cloud users to not only scale their applications, but also do so in a cost-effective way. The ability to dynamically allocate and utilize new virtualized resources has liberated the cloud users from the burden of administering the physical infrastructure, since the cloud provider itself is responsible for maintenance and any administrative tasks. The dynamic nature of the resources, though, gave birth to new requirements: Applications need to be deployed and utilize the resources in an automated manner, without requiring human intervention. This concept is the cornerstone behind *automation*, i.e., the ability to run complex tasks that entail resource provisioning and software configuration in a fully automated manner. Automation is an integral part of the Cloud, that helped traditional system administration principles to evolve in order to consider dynamic infrastructures and virtualized resources [2, 3].

Especially during the application deployment phase, automation is essential in order to guarantee that different software components such as the cloud's software stack, the Virtual Machines (VMs), external services, etc., will cooperate in a synchronous manner so as to successfully deploy a given application to the cloud. This challenging task has been both an active research field [4–11] and the objective of many production systems, operated by modern cloud providers [12–15]. These approaches differ in various aspects: Some of them specialize to specific applications (e.g., Openstack Sahara [13] focuses on deploying data processing systems to Openstack) whereas

*Correspondence: ggian@cslab.ece.ntua.gr
[1]Computing Systems Laboratory, School of ECE, National Technical University of Athens, Iroon Polytechniou 9, 15780 Zografou, Greece
Full list of author information is available at the end of the article

others [7, 16] support an application description language and allow the user to define the application structure. Moreover, some tools operate on specific providers (e.g., [14]), whereas other retain a cloud-agnostic profile and abstract the provider specific details from the application description (e.g., [17]). In spite of their differences, these systems share the same objective: They expect an application description along with any other runtime configuration option and they are responsible to allocate new resources, provision them and configure a new application instance in a fully automated manner.

However, achieving a fully automated application deployment assumes that all the components participating to it (e.g., cloud compute, storage and metadata services, Virtual Machines (VMs), external services, etc.) function in a coordinated and failure-free manner. Even if coordination between different modules is achievable, failures are not rare: Network glitches, request timeouts due to delays in VM bootstrapping, etc., are commonly encountered. Moreover, even the most popular cloud providers, often encounter severe infrastructure failures [18, 19] that, according to their severity, may lead to service failures for considerable amount of time (spanning from seconds up to several minutes). A key characteristic of this type of errors is their *transient* nature: They appear for a short time period and vanish without any obvious, from the application's viewpoint, reason. In most cases, the appearance of such errors may be tolerable. For example, during a network glitch, the application may lose some requests, but after the glitch disappears the requests can be easily repeated and the application will be completely functional again. However, during the *sensitive* application deployment phase, such glitches may lead an entire deployment to failure, requiring human intervention either to terminate the allocated VM instances or to manually fix the failed deployment parts.

To tackle this challenge, a crucial observation can be made: When such a *transient* failure occurs during the deployment phase, in many cases, one only needs to repeat the deployment scripts that failed in order to overcome it. For example, take the case of installing new software packages to a newly allocated VM: If one wants to download a software package (e.g., through `apt`) and a network glitch occurs, all one has to do in order to overcome this error is to repeat the download command until the glitch vanishes. Evidently, the transient nature implies that the cloud user does not have control over it. Hence, an optimistic error-recovery policy would aim at repeating the script execution until the error disappears.

Given this crucial observation, in this work we propose a cloud application deployment methodology that aims at identifying the parts of an application deployment that failed due to a transient failure and repeat them until the deployment is successfully accomplished. Specifically, building upon previous work [20, 21], we propose a deployment model, perfectly suited for distributed applications with multiple software components deployed to different VMs. The application description is serialized to a Directed Acyclic Graph (DAG) that describes the dependencies between different application modules. The application deployment effectively leads to the traversal of the dependency DAG in a specific order, in order to satisfy the module dependencies. In case of *transient* errors, our methodology first examines the DAG and isolates the part of it that failed and, then, executes the failed scripts until the transient errors vanish. In order to ensure that the re-executed deployment parts (i.e., *deployment scripts*) always have the same effects (i.e., they are *idempotent*), we adopt a lightweight filesystem snapshot mechanism that ensures that each configuration script can be re-executed as many times as required, until the error vanishes.

The contributions of this work can be, thus, summarized as follows:

- We propose a powerful deployment model, which is capable of efficiently expressing the configuration actions that need to take place between different application modules, in the form of a Directed Acyclic Graph.
- Using this model, we formulate the application deployment problem as a DAG traversal problem and suggest an efficient algorithm for identifying the scripts that failed, along with their respective dependencies.
- We suggest a lightweight filesystem snapshot mechanism in order to ensure that the configuration scripts are idempotent and, hence, can be re-executed as many times as needed.
- We offer AURA, an open-source Python prototype of the proposed methodology [22], which is capable of issuing application deployments to Openstack instances.
- Finally, we provide illustrative examples of our deployment model for various popular real-world applications and thoroughly evaluate the performance of our prototype through deploying them in a private Openstack cluster.

Our evaluation demonstrated that the proposed approach is capable of deploying diverse applications with different structures in cloud environments that exhibit high error probabilities (reaching up to 0.8), while inserting minimal performance overhead for ensuring the deployment script idempotency. Moreover, the efficiency of the proposed approach is showcased to be increasing with the application structure complexity, as more complex applications are more susceptible to transient failures since they occupy more resources.

Related work

Since the emergence of the cloud computing era, the challenge of fully automated software configuration and resource provisioning has attracted a lot of interest, both by the academia and the industry. The importance of automation was diagnosed early and, thus, many solutions have been proposed and are currently offered to the cloud users. We now briefly discuss the distinct approaches and outline their properties.

Industrial systems: There exist several tools and systems offered by modern cloud providers to their users that aim at providing fully automated application deployment. In the Openstack ecosystem, Heat [12] is an orchestration system which aims at managing an application throughout its lifecycle. The users submit application descriptions (named HOT, i.e., Heat Orchestration Template), where they define: The application modules, the resources each module will occupy, the dependencies between different modules (if any) and any other runtime parameter (e.g., name of the SSH key to use, flavor id, etc.). In order to maximize reusability, Heat templates can be parameterized, abstracting the installation-specific details (e.g., image/flavor IDs, key names, etc.) from the application description. Sahara [13] is a different deployment tool for Openstack that specializes in provisioning Data Processing clusters, i.e., Hadoop and Spark clusters. Sahara uses Heat as the deployment engine and differs from it as it enables the users to provide Hadoop-specific deployment parameters (e.g., HDFS replication size, number of slaves, etc.) and applies them to the cluster. The AWS counterpart of Openstack Heat is the AWS CloudFormation [15]. Similar to Heat, CloudFormation receives templates that describe what resources will be utilized and by which components. The cloud user can then define a set of parameters (keys, image ids, etc.) and launch the deployment. AWS Elastic Beanstalk [23] specializes in deploying web applications to the AWS cloud, hiding the infrastructure internals and automating the provisioning of load balancers, application-level monitoring, etc.

The aforementioned systems are implemented and offered as components of the cloud software stacks they operate on. However, there exist many systems that operate *outside* the target cloud and are capable of deploying to different providers. Vagrant [17] is an open source tool for building and maintaining portable virtual software development environments. It is mainly used during the development phase of a system which will be deployed to a cloud provider and its main objective is to simplify the software configuration process. Juju [14] is another open source system, developed and maintained by Canonical that works on top of Ubuntu images. According to Juju, each application comprise of building blocks named *Charms*. When deploying an application, each Charm is deployed independently and, finally, the different Charms are unified through passing parameters to each other. Finally, CloudFoundry [24] and Heroku [25] are two systems that focus on providing a more platform-oriented view of the cloud, i.e., Platform-as-a-Service (PaaS) semantics on top of Infrastructure-as-a-Service (IaaS) clouds. The difference between them and the previous solutions is that both CloudFoundry and Heroku deploy applications to different providers but their objective is to provide access to the platform level, rather than the VM level, decoupling this way the underlying virtualized hardware from the deployed application.

All the aforementioned systems generate a dynamic landscape of deployment tools, with diverse characteristics and different strengths. Nevertheless, none of these tools considers the dynamic and, frequently, error-prone nature of the cloud, as none of them provides a mechanism of overcoming transient errors in a fully automated manner, as they require human intervention either to resume the deployment or trigger a new one. For example, CloudFormation's official documentation suggests to manually continue rolling back an update when a Resource is not stabilized due to an exceeded timeout period [26]. A similar suggestion is also made for the Elastic Load Balancing component, extensively utilized both by CloudFormation and Beanstalk instances: In case where a delay in certificate propagation is encountered, the users are advised to wait for some minutes and retry to setup a new load balancer [27].

Research approaches: The complexity on the structure of modern applications comprising different software modules has, inevitably, complicated their deployment to cloud infrastructures and has been the center of research from different works. NPACI Rocks [8] attempts to automate the provisioning of high-performance computing clusters in order to simplify software installation and version tracking. Although this approach was suggested prior to the wide adoption of the cloud and focuses on the HPC world, it is one of the first works that discusses the problem of resource provisioning and software configuration at a big scale and proposes a tool set in order to automate common tasks. Wrangler [7] is a generic deployment system that receives application descriptions in XML format and deploys them to different cloud providers (Amazon EC2, OpenNebula and Eucalyptus). Each description comprises different *plugins*, i.e., deployment scripts, executed on different VMs in order to install a particular software component, e.g., a monitoring plugin, a database plugin, etc. Antonescu et al. in [11] propose a novel specification language and architecture for dynamically managing distributed software and cloud infrastructure. The application is now expressed as a set of services each of which is adjusted (dynamically started and stopped)

according to the achieved SLAs and the user-defined constraints. Katsuno et al. in [6] study the problem of deployment parallelization in multi-module applications. Specifically, the authors attempt to detect the dependencies between different modules through the extension of the Chef [28] configuration tool and execute the deployment scripts in a parallel fashion. Finally, Pegasus [9] is another deployment system that specializes to deploying scientific workloads. All the discussed systems, attempt to either achieve complete automation or speedup application deployment, ignoring the frequently unstable nature of the cloud resources, in contrast to our work that aims both at accelerating deployment and overcoming transient cloud failures.

Interestingly, the problem of failure overcoming has been the center of interest for many research works lately. Liu et al. in [29] provide an interesting formulation where an application deployment is viewed as a database transaction and the target is to implement the necessary mechanisms to achieve the ACID properties of the latter. To this end, the authors assume that each deployment script must be accompanied by an *undo* script that reverts the changes of the former. This way, in case of errors, any unwanted side effects are nullified. Note that, although this is an interesting formulation, the hypothesis that each script is accompanied by another script that executes undo actions is rather strong and, in many cases, impossible. Yamato et al in [10] diagnosed some insufficiencies of the state of the art Heat and CloudFormation deployment tools and proposed a methodology through which Heat Templates can be shared among users, extracted from existing deployments and trigger resource updates. The authors of this work also diagnosed the problem of partial deployments due to transient errors and describe a rollback mechanism in order to delete resource generated due to failed deployments. Although deletion of stale resources is a positive step, there exists much room for an automated failure overcoming mechanism. Hummer et al. in [5] highlight the necessity of achieving *idempotency* in production systems, for cases where a new software version is released. In cases where one needs to return to a previous, stable version it is crucial for the deployment system to rapidly do the rollback and converge to a healthy system state. To this end, this paper focuses on the theoretical model that guarantees theoretical convergence to a healthy deployment state. Rather than wandering to the – possibly enormous – state space, our approach adopts a lightweight filesystem snapshot mechanism that guarantees fast convergence. Finally, the work in [4] shares a similar solution and extends the previous work using a different configuration management language.

Generic deployment systems: Although the prevalence of the Cloud paradigm accentuated the importance of fully automated software configuration, the problem is long discussed prior to the wide adoption of the Cloud and many solutions that operate on the resources per se (either virtualized or not) have been proposed. These solutions do not consider resource allocation. Nevertheless, the problems of synchronization and dependency resolution are also addressed by them and resemble the problem addressed by our work.

CFEngine [30] is one of the first systems that was proposed to deal with automated software configuration. It introduced the idea of *convergent operators*: Each operation should have a fixed-point nature: Instead of explaining what needs to be done, a user explains the *desired state*. CFEngine is, then, responsible to perform the necessary actions to reach it. Chef [28] is a popular software configuration system, adopted by Facebook. The deployment scripts (*recipes*) are written in a Ruby-based domain-specific language. Multiple recipes are organized in *cookbooks* which describe the desired state of a set of resources. Puppet [31] is another widely used software configuration system used by Twitter and Mozilla. Similarly to CFEngine, Puppet adopts a declarative syntax and users describe the resources and the *state* that should be reached when the deployment is over. It is released in two flavors: Enterprise Puppet, that supports coordination between different hosts and Open Source Puppet that comes with a limited number of features.

Unlike the rest of the systems proposed so far, Ansible [32] follows an *agentless* architecture. Instead of running daemon services inside the machines, Ansible utilizes plain SSH connections in order to run the deployment scripts, configure software, run services, etc. Although its architecture supports coordination between different machines, Ansible users need to carefully design the deployment scripts (i.e., *playbooks*) in order to achieve idempotency. Finally, Salt [33] (or SaltStack Platform) is a software configuration and remote execution platform written in Python. Similar to Ansible, Salt also uses an agentless architecture; recent Salt versions also support daemon processes inside the target machines in order to accelerate deployment. It is highly modular as a user can customize the behavior of a deployment workflow through extending the default functionality using Python.

Although these tools are extensively utilized in modern data centers and cloud infrastructures, none of them offers a built-in support of coordination between software modules that span to multiple hosts, as our work. This is supported either on premium versions of them or through adopting community-based plugins. Moreover, although these tools attempt to repeat the execution of configuration scripts in order to reach convergence, none of them guarantees idempotence to the extent our work does, i.e., full idempotence to file system related resources. As all of these tools do support the execution of possibly

non-idempotent calls (e.g., through the `execute-` calls in Chef) that cannot be undone or leave the burden of writing idempotent scripts to the users (Ansible), our work decouples idempotency from the ability of undoing actions and achieves a greater level of decoupling scripts from their side effects. Finally, since our work is based on executing simple bash scripts in the correct order to different machines, it is much easier to exploit existing deployment scripts written in bash in order to compile new application descriptions and eschew the learning curve of a new language that reflects the target deployment states.

Application deployment

In this section, a thorough description of the proposed methodology is provided. We, first, provide the architecture of the system we implemented. Next, we discuss an efficient deployment model, through which one can describe the deployment of any arbitrary application. We, then, examine how this model is utilized for the identification and recovery from transient cloud failures and, finally, we examine the mechanism through which our prototype guarantees the idempotency of each deployment part.

Architecture

We begin our discussion through introducing the architecture of the system that implements the deployment methodology this work proposes. Our prototype is named *AURA*[1] and Fig. 1 depicts its architecture. AURA comprises two components: The *AURA Master* and the *AURA Executor(s)*. AURA Master is deployed to a dedicated host (VM or physical host) and consists of a set of submodules that coordinate the deployment process and orchestrate error recovery. Specifically, the *Web UI* and *REST API* submodules export AURA's functionality to its users. Through them, an authenticated AURA user can submit new application descriptions, launch new deployments, query a deployment's state, obtain monitoring metrics and terminate a deployment. The *Provisioner* is responsible to contact the underlying Openstack cluster in order to allocate the necessary resources. The *Scheduler* is the core component that coordinates the deployment process and orchestrates failure overcoming. The application descriptions, the deployment instances, the user configuration options and any other relative information is persisted to a database inside the AURA Master host and it is accessible by the Scheduler.

AURA Executors are lightweight processes, deployed inside the VMs which are allocated to host the application and they are responsible to execute the deployment scripts of the respective modules they are deployed to. The Executors communicate with the AURA Master through a *Queue* that acts as the communication channel between the different components. In case of a transient error, the Scheduler identifies which deployment parts need to be replayed and transmits messages to the Executors that, in turn, cancel any unwanted side effect a previous script execution left. Note that, although the Queue is depicted as an external module in order to increase the figure's legibility, it also belongs to AURA Master, in the sense that it is statically deployed in the same AURA Master host and Executors from different deployments utilize the same queue to exchange messages.

Before analyzing AURA's functionality is detail, let us provide the key assumptions that drove AURA's design. First of all, the errors that emerge have a *transient* nature. This means that they are only present for a short period of time and vanish without requiring any manual intervention. Second, we assume that the communication channel between the Executors and the Master is *reliable*, i.e., in case where an Executor sends a message to the Queue, this message always reaches its destination. Third, we assume that the AURA Master is always *available*, i.e., it may not fail. Given the above, we now proceed with describing AURA's deployment model and mechanisms through

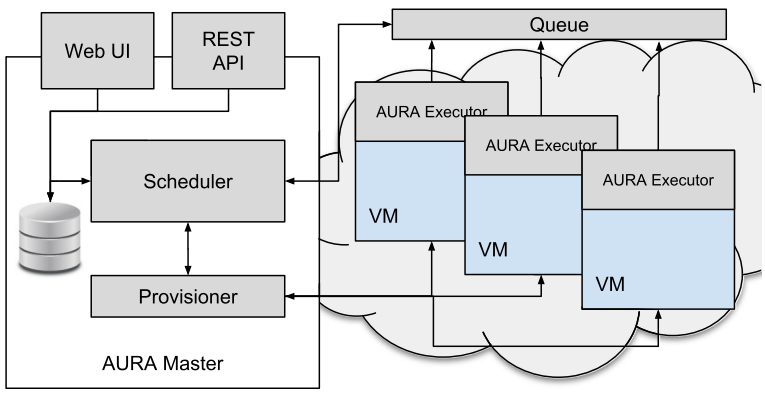

Fig. 1 AURA Architecture

which it achieves error recovery and deployment script idempotence.

Deployment model

Assume a typical three-tier application consisting of the following modules: A Web Server (rendering and serving Web Pages), an Application Server (implementing the business logic) and a Database Server (that persists the application's data). For simplicity's sake, we assume that each module runs in a dedicated server and the application will be deployed in a cloud provider. If the application deployment occurred manually, one should create three VMs and connect (e.g., via SSH) to them in order to execute scripts that take care of the configuration of the software modules. In many cases, the scripts need input that is not available prior to the resource allocation. For example, the Application Server needs to know the IP address and the credentials of the Database Server in order to be able to establish a connection to it and function properly. In such cases, the administrator should manually provide this *dynamic* information.

The automation of the deployment and configuration process requires the transmission of such dynamic information in a fully automated manner. For example, upon the completion of the configuration of the Database Server, a message can be sent to the Application Server containing the IP address and the credentials of the former, through a *communication channel*. Such a channel can be implemented via a simple queueing mechanism. Each module publishes information needed by the rest of the modules and subscribes to queues, consuming messages produced by other modules. The deployment scripts are executed up to a point where they expect input from another module. At these points, they block until the expected input is received, i.e., a message is sent from another module and it is successfully transmitted to the blocked module. The message transmission is equivalent to posting a new message into the queue (from the sender's perspective) and consuming this message from the queue (from the receiver's perspective). In cases that no input is received (after the expiration of a fixed time threshold), the error recovery mechanism is triggered.

Albeit the content of the messages exchanged between two modules depends on the executed deployment scripts and the type of the modules (e.g., passwords, SSH keys, static information regarding the environment a module is deployed to, etc.), each message belongs to one of the following three categories: *Static*, *Dynamic* and *ACK* messages. *Static* messages are considered the ones that contain information that does not change when a deployment script is re-executed (e.g., VM's IP address, amount of memory, number of cores, etc.). *Dynamic* messages contain information that change any time a deployment script is re-executed. For example, if a script running on a Database Server generates a random password and transmits it to the Web Server, in case of a script re-execution a new password will be generated and a new message will be sent (containing a different password). Finally, *ACK* messages assist synchronization of the deployment between different software modules and are used to enforce a script execution in a particular order as they do not contain any useful information.

To provide a better illustration of the deployment process, consider Fig. 2. In this Figure, a deployment process between two software modules named (1) and (2) is depicted. The vertical direction represents the elapsed time and the horizontal arrows represent message exchange[2]. At first, both (1) and (2) begin the deployment process until points A and A' are reached respectively. When (1) reaches A, it sends a message to (2) and proceeds. On the other side, (2) blocks at point A' until the

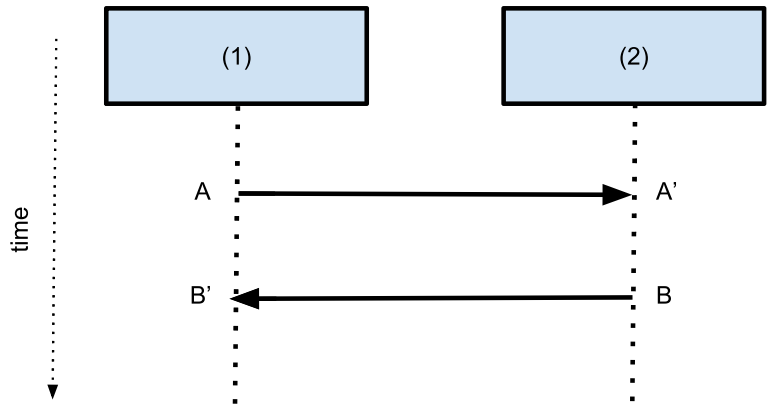

Fig. 2 Message exchange between different software modules and circular dependencies

message sent from (1) arrives and, upon arrival, consumes it and continues with the rest of the script. If the message does not arrive, then the recovery procedure is triggered, as described in the next section.

The functionality of the above message exchange mechanism resembles the functionality of a UNIX pipe: Message receiving is blocking for the receiver end, whereas the sender module posts something to the channel and proceeds instantly. In some cases, though, blocking message transmissions may be desired. For example, take the case of two modules negotiating about a value, e.g., a randomly generated password assigned to the root account of the Database Server. Assume that module (1) (the Application Server) decides on the value and sends it to module (2) (the Database Server). Module (1) must ensure that the password is set, prior to trying to connect to the database. To this end, (2) can send an acknowledgment as depicted between points B and B'. In this context, the message exchange protocol can also function as a synchronization mechanism. This scheme represents a dependency graph between the application's modules, since each incoming horizontal edge (e.g., the one entering at point A') declares that the execution of a configuration script depends on the correct execution of another.

Note that, schemes like the one depicted in Fig. 2 present a circular dependency, since both modules (1) and (2) depend on each other, but on different time steps. This feature enhances the expressiveness of the proposed deployment model, as one can easily describe extremely complex operations between different modules, which depend on each other in a circular manner. Various state-of-the-art deployment mechanisms do not handle this circularity (e.g., Heat [12], CloudFormation [15]) since they lack the concept of time during the configuration process and, hence, forbid their users to declare dependencies that create loops. In our view, this circularity naturally occurs to a wealth of modern cloud applications, that comprise many modules deployed to different VMs and, hence, the prohibition of such loops leads to application descriptions that rely on "hacks" to work. On the other side, the allowance of circular dependencies leaves room for application descriptions that contain deadlocks, i.e., pathological cases where each module waits for another and the deployment blocks forever. This problem is outside the scope of our work; it is the user's responsibility to generate correct, deadlock-free application descriptions that, if no failures occur, reach to termination.

Error recovery

We now describe the mechanism through which the deployment errors are identified. During the deployment process, a module instance may send a message to another module. This message may contain information needed by the latter module in order to proceed with its deployment, or it could just be a synchronization point. In any case, the second module blocks the deployment execution until the message arrives, whereas the first module sends its message and proceeds with the execution of its deployment script. In the case of an error, a module will not be able to send a message and the receiver will remain blocked. To this end, we set a timeout period that, if exceeded, the waiting module sends a message to the AURA Master informing it that a message has been missed and a possible error might have occurred. From that point on, the error recovery mechanism takes control, evaluates whether an error has, indeed, occurred or the running scripts need more time to finish. In the former case, the error is evaluated, as we will shortly describe, and the necessary actions are performed in order for the deployment to be restored. The selection of an appropriate timeout period is not trivial. While smaller thresholds may trigger an unnecessary large number of health checks in cases where the deployment scripts need much time to complete, larger ones make our approach reacting slower to errors. The default timeout period used by AURA equals to 1 min. Nevertheless, if the users are knowledgeable about the real deployment time of their application, they can further optimize it. In the experimental evaluation, we provide a rule of thumb that facilitates with this choice. Finally, when a module's configuration successfully terminates, it reports its terminal state to AURA Master; the entire deployment is considered successfully finished when all software modules have successfully been deployed.

Upon the identification of a (possible) failure, the error recovery mechanism is triggered. The error recovery process aims at overcoming the transient cloud failure encountered during the deployment and is based on the repetition of the execution of scripts/actions which led the system to failure. In order to identify the parts of the deployment that need to be re-executed, AURA needs to be aware of the dependencies that exist between different deployment scripts. These dependencies can be easily resolved, when considering that the executions of the scripts and the message exchange between different modules (as presented in Fig. 2) constitute a *graph* (referred as the *deployment graph* henceforth) that express the order with which different actions must take place. For example, consider the graph presented in Fig. 3, that presents the deployment graph of a Wordpress application comprising a Web Server and a Database Server. The graph nodes represent the states of the deployment, the solid edges represent a script execution and the dotted edges represent a message exchange. The examination of the deployment graph indicates that the execution of, e.g., the `web-server/3` script depends on the execution of two other scripts: `web-server/2` and

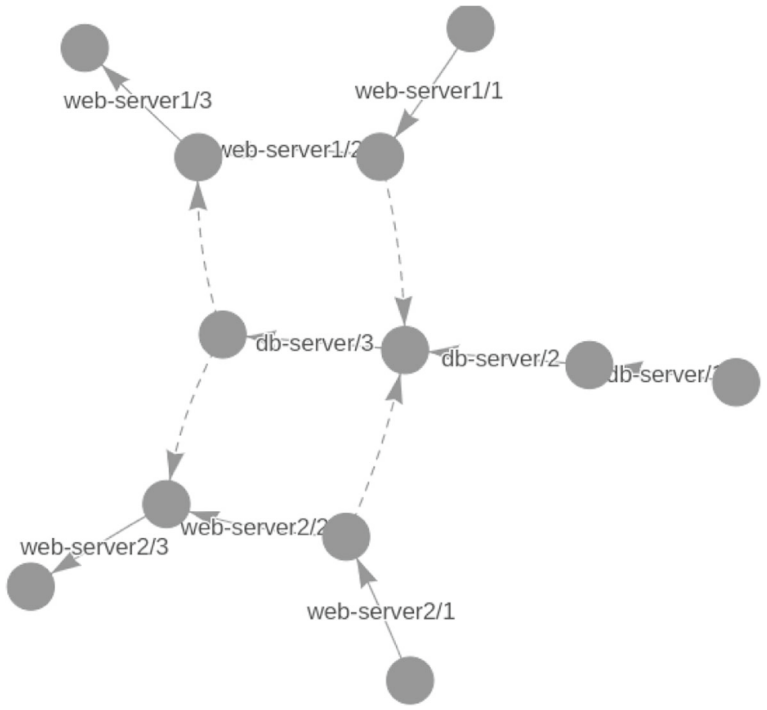

Fig. 3 Wordpress deployment graph with 2 Web Servers and 1 Database Server

`db-server/3` (that sends an *ACK* message). In the general case, if one wants to identify the dependencies of any script, one needs to traverse the graph in reverse edge direction.

Having defined the methodology used to identify the dependencies between different deployment scripts, we can now formally describe the Error Recovery procedure, as presented in Algorithm 1. The algorithm accepts two parameters: A deployment graph T (as the one depicted in Fig. 3) and the id of a failed node n. The algorithm traverses T in a Breadth-First order beginning from the failed node and traversing the graph in opposite edge direction. The goal of the algorithm is to identify a list of nodes that have been successfully reached by the deployment. Starting from the node that triggered the Error Recovery, the algorithm retrieves all the node's dependencies (`depends` function). For example, if the failed node is C', the `depends` function returns the nodes B' and D. The `failed` function checks the respective Executor's state and returns *true* if the Executor reports that the intermediate state has not been reached. For example, if D has not been reached then `failed(D)` returns *true* and D is placed to the `failed` list in order to be later examined. If B' is successfully reached, then B' is placed in the `healthy` list and its dependencies are not examined. This procedure is iterated until all failed nodes are identified. Using this *healthy* list, the AURA Master contacts the respective Executor for each node and instructs them to resume their execution from their last successfully reached healthy state. In the previous example, if we assume that C was successful, then the *db-server* Executor is instructed to repeat `db-server/3`.

Algorithm 1 Error Recovery Algorithm

Require: deployment graph T, failed node n
Ensure: list of *healthy* nodes *healthy*

```
failed ← {n}
healthy ← ∅
while failed ≠ ∅ do
    v = pop(failed)
    for t ∈depends(T, v) do
        if failed(t) then
            failed ← failed ∪ {t}
        else
            healthy ← healthy ∪ {t}
        end if
    end for
end while
return healthy
```

It should not be overlooked, that when certain parts of the deployment graph are re-executed, the replayed scripts may produce new or require to consume older messages previously transmitted by other modules. In

case where a replayed script needs to consume older messages previously sent by other modules, AURA keeps track of all the messages and forwards the ones required by a replayed script. What happens, though, when a replayed script produces new messages? In these cases, AURA's behavior is affected by the type of the message: If the new message is of type *Static* or *ACK*, the message is sent by the replayed script and ignored by AURA's Queue module. Since both message types are not affected by a script re-execution, there is no need to propagate the new messages to other, healthy deployment parts. However, when a *Dynamic* message is sent, the recipient module need to repeat the configuration since the receiving information changed. For example, if a Database Server script re-generated a new password and sent it to a Web Server, the latter needs to be reconfigured in order to reflect the updates. It is, now, evident, why AURA implements different message types and how these affect error-recovery. Through this mechanism, AURA's user has the liberty to define the message types of the deployment scripts and affect the algorithm's behavior.

Finally, before proceeding to the examination of the idempotency enforcement mechanism, we should not overlook that since different deployment parts run in parallel, *race conditions* between different modules may be encountered. For example, take the case where one module broadcasts a health-check request, and while the master runs Algorithm 1, another module broadcasts a new request. Since parallel algorithm executions may lead to contradicting actions (i.e., different *healthy* sets), we need to ensure that the algorithm executions are properly synchronized. To this end, each health-check request is placed to a FIFO queue and examined one at a time: The first health-check request is examined by the master, which runs Algorithm 1 and informs the failed modules about the necessary actions. Subsequently, the next health-check request is examined, Algorithm 1 runs again, but now the previously failed modules are in an "Executing" state. If the new health-check request was issued for another module (i.e., not one of the modules that failed in the previous Algorithm execution), then this module is resumed. In any other case, the failing module is already in an "Executing" state and, hence, the health-check request is ignored. In essence, the serialization of both the health-check requests and the algorithm's executions, ensure that the actions taking place for resuming the failed deployment parts leave the deployment in consistent states and eschew race conditions.

Idempotency

The idea of script re-execution when a deployment failure occurs, is only effective when two important preconditions are met: (a) the failures are transient and (b) if a script is executed multiple times it always leads the deployment into the same state, i.e., it is *idempotent*. In this, section we discuss these preconditions and describe how are these enforced through our approach.

First, a failure is called "transient" when caused by a cloud service and was encountered for a short period of time. Network glitches, routers unavailability due to a host reboot and network packet loss are typical examples of such failures caused by the cloud platform but, in most cases, they are only observed for a short period of time (seconds or a few minutes) and when disappeared the infrastructure is completely functional. Various works study those types of failures (e.g., [34]) and attribute them to the complexity of the cloud software and hardware stacks which presents strong correlations between seemingly independent cloud components [35]. Although most cloud providers protect their customers from such failures through their SLAs [36], they openly discuss them and provide instruction for failures caused by sporadic host maintenance tasks [37] and various other reasons [38]. Since the cloud environment is so dynamic and the automation demanded by the cloud orchestration tools requires the coordination of different parties, script re-execution is suggested on the basis that such transient failures will eventually disappear and the script will become effective.

However, in the general case, if a script is executed multiple times it will not always have the same effect. For example, take the case of a simple script that reads a specific file from the filesystem and deletes it. The script can only be executed exactly once: The second time it will be executed it will fail, since the file is no longer available. This failure is caused by the side effects (the file deletion) of the script execution, which lead the deployment to a state in which the same execution cannot be repeated.

In order to overcome this challenge, we employ a lightweight filesystem snapshot mechanism, that aims at persisting the filesystem state *prior* to each script execution and, in case of failure, revert to it, in order to cancel any changes committed by the failed script. Filesystem snapshotting is a widely researched topic in the Operating Systems field [39, 40], and it is commonly used for backups, versioning, etc. The mechanics behind the snapshot implementation differs among various filesystems: A straightforward implementation requires exhaustively copying the entire VM filesystem to a secure place (and copying it back during the *revert* action), whereas more sufficient approaches rely on layering or copy-on-write techniques. The huge overhead of copying the entire filesystem lead us to favor the latter techniques. To this end, we implemented two different snapshot mechanisms, using two different widely popular filesystems: AUFS [41],

that relies on layering and BTRFS [39] that relies on copy-on-write.

AUFS is one of the most popular layered filesystems that rely on union mounting. The main idea behind union mount is that a set of directories are mounted with a specific order and the final filesystem is the union of the files contained to each distinct directory. Whenever one wants to *read* a file, the uppermost layers are scanned first and, if the file is not found, the search is continued to the bottom layers. Each layer may be mounted with different permissions; usually, the topmost layer is mounted with read-write access and the rest of the layers are mounted with read-only access. When a *write* operation takes place, the updates are only written to the topmost, read-write layer, keeping the rest of the layers intact. The same happens when deleting a file: Instead of physically removing the file from its layer, a new *special file* is generated, demonstrating that the target file is not available. This mechanism inherently supports filesystem snapshots: Prior to each script execution, a new *layer* is mounted on top of the others. If the script fails, then the newly allocated layer is removed and, hence, all the scripts' side-effects are canceled. In different case, new layers are appended on top of the existing ones.

BTRFS, on the other hand, is a much newer and promising filesystem that relies on copy-on-write for snapshotting. It supports the concept of *volume*, which acts as an endpoint used by the OS to mount it to the root filesystem. Snapshotting occurs on a per-volume basis: One can create a snapshot, which represents a state of a volume. When something changes (e.g., new files are created/updated/deleted), BTRFS only creates the inodes that are updated, minimizing the number of filesystem structures which are updated. The main difference between BTRFS and AUFS is that the latter works on a file level, i.e., when a file is updated, the entire file is replicated to the topmost layer, whereas the former works on a block level, i.e., only the filesystem blocks which are affected are replicated. On the other side, AUFS is more generic, since it can be implemented on top of other filesystems, whereas BTRFS is more restrictive since the application data must be persisted to a BTRFS partition. In the experimental evaluation, we continue the discussion for their differences and evaluate their impact to the efficiency of our approach.

Finally, we should note that the discussion around idempotency, is only limited to filesystem related resources because the configuration process of most applications usually relates to modifying configuration files. We shall not overlook, though, that modern applications may demand idempotency to other resources, as well: The memory state of a module, for example, cannot be reset when a script need to be re-executed. This is an interesting extension of our approach which will be addressed in the future.

Implementation aspects

Upon presenting AURA's architecture and describing its main functionality, we now wish to extend the discussion to specific implementation aspects of the proposed system. Our analysis comprises two dimensions. We first introduce two optimizations that boost AURA's applicability and, subsequently, discuss the technical means through which the proposed system approaches the concepts of *Portability* and *Reusability*.

Optimizations

We now provide some optimizations that were implemented to enhance AURA's applicability. So far, we have made the assumption that there exists a one-to-one mapping between application modules and VMs, i.e., each module (e.g., (1) of Fig. 2) is deployed to a dedicated VM. Bearing in mind, though, that modern distributed applications, inherently deployed to cloud environments, rely on deploying the same software modules to more than one VMs (e.g., HDFS clusters [42] comprise many datanodes), this assumption is rather restrictive. To this end, AURA's implementation supports augmenting application description with an extra *deployment parameter* that refers to each module's `multiplicity`. This means that prior to deploying, a user can define a (strictly positive) module multiplicity and this, practically, leads to replicating the same module description as many times as required. In this case, the Queue module modifies message exchange accordingly: During message transmission, messages from VMs that host the same software modules are merged in a predefined order (ascending Executor UUID) and offered to the recipient(s), whereas during message reception, the Queue replicates the message as many times as needed. Fig. 4 graphically represents this process, being a variation of Fig. 2, where (2) is deployed to two VMs. Note that, the fact that the Queue replicates and merges messages, abstracts the concept of multiplicity from application descriptions and maximize their reusability.

Furthermore, apart from the initial deployment phase, the suggested workflow can be extremely helpful during an application's lifetime, for different occasions. For example, take a two-tier application comprising a Web and a Database Server and assume that an administrator wants to run a maintenance task, e.g., update the Database Server software. This task can easily be described in the form of a DAG, where one would, first, gracefully shutdown the Web Server (closing any stale DB connections), then shutdown the DBMS, update it and, finally, start the services in reverse order. Note that, this chain of events

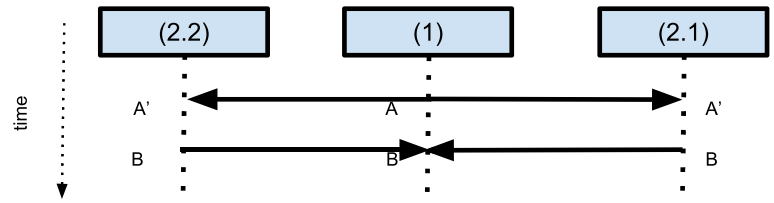

Fig. 4 Message exchange between modules of different multiplicity

can be much more complex and may span to more than two modules (e.g., assume that there exist more than one Web Servers). In order to support these actions, AURA supports the definition of a second *deployment parameter* which defines the `address` of a VM: If set, the Provisioner submodule is short-circuited and no resource allocation takes place. In this case, the (already deployed) AURA Executor, receives a new set of scripts that need to be executed according to the proposed methodology. It must be emphasized that this is a first step towards dynamic application *scaling*, i.e., resource allocation and software configuration in order for an application to utilize the new resources, and it is a very interesting future direction that will be explored.

Portability and reusability

Two dimensions that highly affected AURA's design choices and have not been examined this far, are *Portability* and *Reusability*. Portability is related to AURA's ability to be able to deploy new application instances in different environments. Reusability is the ability to utilize existing deployment scripts in order to compose new application descriptions compliant with AURA. Although these aspects are not considered *Functional Requirements* in the narrow sense, they are both highly desirable and, if achieved, they can drastically increase AURA's utility.

AURA's modular architecture, as depicted in Fig. 1, allows different modules (e.g., Provisioner, Scheduler, etc.) to communicate in a transparent way, without relying on the design of each other. For example, when the Scheduler module issues a new request to the Provisioner module to allocate new VMs, the latter communicates with Openstack without exposing Openstack-specific semantics to the former. This means that if one wants to utilize AURA in a different cloud environment that adheres to a different API (e.g., AWS, CloudStack [43]) or utilizes a different virtualization technology (e.g., Kubernetes [44] utilizing Linux Containers), one could provide a custom Provisioner module that contacts the respective cloud platform without altering anything else to AURA's functionality. The only requirement for a custom Provisioner module is to respect the `CloudOrchestrator` API, as seen from AURA's source code [22], and implement the `create_vm` and `delete_vms` functions. In a similar manner, the user may also override AURA's default behavior regarding VM creation that in case where a VM allocation request fails, the whole deployment is aborted. One could easily change this behavior through adding a loop that eagerly spawns new VM allocation requests in case where the underlying Openstack calls fail.

Finally, we should note that the design choice of supporting DAGs for AURA's application descriptions was favored against other options (e.g., describing the state of the different VMs) for the following reason: The users can easily utilize existing deployment scripts, written in `bash` or any other language, in order to compose more complex application descriptions. One can easily transform a set of existing deployment scripts to a DAG-based description, as the one depicted in Fig. 5, following two steps:

1. "Break" a larger deployment script in smaller parts, identifying which ones should be executed on which VMs and what input is anticipated for each module,
2. Generate a `description.json` file that encodes the order of execution inside a module and the input/output messages each script anticipates.

The syntax of the description file is simple, and contains the bare minimum information required to generate the deployment graph. Listing 1 provides an example description file for the Wordpress application. Initially, the users define the application modules of their application (e.g., `db-server` and `web-server`). For each module, the user provides a list of scripts that contain its sequence number (`seq` parameter) that indicates its order of execution, the path of the deployment script (`file` parameter) that may be any executable file, and (if applicable) a list of script names in which the current script depends to or the scripts to which the current script will send messages to (i.e., `input` and `output` parameters respectively).

Listing 1 Wordpress description file

```
{
 "name": "Wordpress",
 "description": "Simple Wordpress installation",
 "modules": [{
 "name": "db-server",
 "scripts": [{
 "seq": 1,
 "file": "db_server/install.sh"
 },{
 "seq": 2,
 "file": "db_server/configure.sh"
```

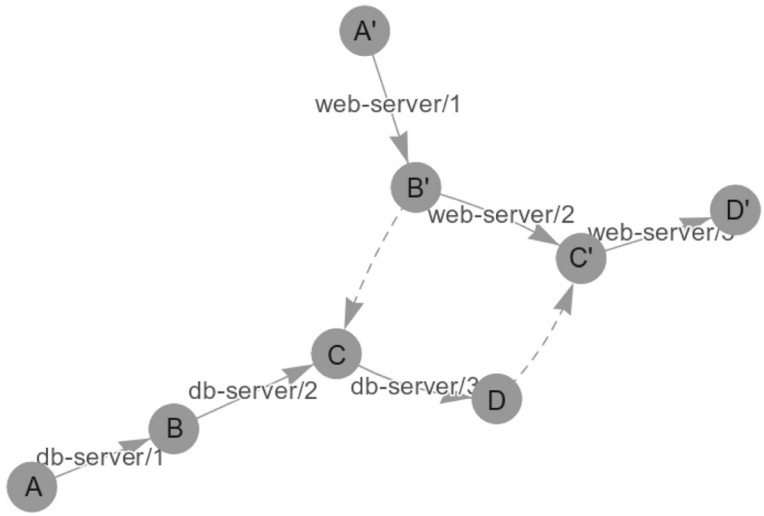

Fig. 5 Wordpress deployment graph with 1 Web Server and 1 Database Server

```
},{
"seq": 3,
"file": "db_server/create_user.sh",
"input" : ["web-server/1"],
"output": ["web-server/3"]
}]
},{
"name": "web-server",
"scripts": [{
"seq": 1,
"file": "web_server/send_ip.sh",
"output":["db-server/3"]
},{
"seq": 2,
"file": "web_server/install.sh"
},{
"seq": 3,
"file": "web_server/configure.sh",
"input": ["db-server/3"]
}]
}]
}
```

Note that the actual deployment scripts (e.g., `web_server/send_ip.sh`) need not be modified in order to publish messages to the AURA Queue. Instead, the AURA Executors that run the deployment scripts, collect their output (everything a script has produced in its standard output) and send it through the AURA Queue to their recipients. On the other end, the Executor that runs a script awaiting for a message, serializes this output and stores it to a temporary file. The path of this temporary file is, then, given as the first argument to the deployment script that opens it, parses it and utilizes its content as it wishes. For example, the script `web_server/send_ip.sh` echoes the VM's IP address. The Web Server Executor collects it and sends it to the Database Server Executor through the Queue module. When the script `db_server/create_user.sh` need to be executed, the Database Server Executor collects the IP address and places it to a file; the script is then launched with this path as the first argument. Subsequently, the script reads the IP address and utilizes it to grant access to Wordpress' database so that the Web Server can access it.

It should be stressed that the choice of not exporting the Queue's semantics to the deployment scripts, liberate the users from writing "AURA-compliant" application descriptions and allow them to write simple scripts that produce and consume information in a traditional way, i.e., through files. In our view, this option greatly simplifies the application description generation process and makes AURA more user-friendly and maximize the reusability of existing deployment scripts.

Experimental evaluation

We now provide a thorough experimental evaluation of the proposed approach that attempts to quantify AURA's efficiency and suitability for deploying real-world application in unstable environments.

Experimental setup: All experiments were conducted on a private Openstack cluster installation, that comprises 8 nodes, each of which has 2 × Intel Xeon E5-2630 v4 (2.20GHz) and 256G RAM (totaling 320 HW threads and 2TB of RAM). The nodes are connected with 10G network interfaces and the VM block devices are stored over a CEPH cluster that consists of 8 OSDs (4 × 3TB 3.5" HDDs on RAID-5 setup each) and 98TB storage. The cluster runs the latest stable Openstack version (Pike) and the hosts run Ubuntu 16.04 with Linux kernel 4.4.0-97.

Applications: In order to evaluate the efficiency of our approach, we opted for popular, real-world applications, commonly encountered to cloud environments:

- *Wordpress* is a popular Content Management System, used to run and manage Web Applications for different purposes. It requires two components: A Web Server, which renders the user interface and a Database Server, which persists the application's data. Therefore, the application description comprise these modules: One module that installs the Apache Web Server along with any other dependency and one module that hosts MariaDB, i.e., the application's database backend. Each module is installed to a dedicated VM and the Web Server may be replicated to more than one VMs, i.e., the Web Server's multiplicity may be greater than one.
- *Hadoop* [45] is a popular data processing system, commonly deployed to cloud infrastructures in order to persist and process Big Data. It also comprises two modules: A Master node that acts as the coordinator of the cluster and the Slave node(s) that are responsible both to persist the cluster's data and to run tasks from MapReduce jobs. In each deployment, there only exists one Hadoop Master node and a set of Slave nodes, determined by the module's multiplicity. We should emphasize that Hadoop is a typical application deployed to the Cloud; for this reason, the most popular cloud providers offer tools to their users that automates the provisioning of Hadoop clusters and also consider their elastic scaling (e.g., Amazon EMR[3]).

Both application descriptions contain: (a) the appropriate scripts in order to install the necessary software components (e.g., Web/Database servers, Hadoop daemons, etc.) along with any other software (e.g., Java, PHP, etc.) or other requirements (e.g., SSH keys, setting up the hosts file, etc.) and (b) the deployment DAG that describes the order of the script execution. For brevity, we omit a detailed description of each configuration script[4]; Figs. 3 and 6 depict the structure of the deployment graphs for Wordpress (that consists of 1 Web Server and 1 Database Server) and Hadoop (that consists of 1 Hadoop Master and 2 Hadoop Slaves) respectively.

The graph nodes represent the states of the modules, the solid edges represent script executions and the dotted edges represent message exchanges between different modules/VMs. Each script execution is labeled in the form `<module>/<sequence>`. When the module multiplicity is greater than 1, the module name in the label also contains the VM's serial number as a suffix, e.g., `hadoop-slave2/3` denotes the third configuration script for the second hadoop slave node.

Methodology: In order to quantify AURA's efficiency, we deployed the previously described applications, using different deployment parameters (e.g., different module multiplicity, filesystem snapshot methodology, etc.) and studied the deployments' behavior. In order to eliminate the unavoidable noise attributed to the randomness of our setup, we executed each deployment 10 times and provide the mean of our results. Our evaluation unfolds in four dimensions. First, we test the deployment behavior for varying module multiplicity, measuring the *scalability* of the proposed deployment scheme, i.e., the ability to deploy

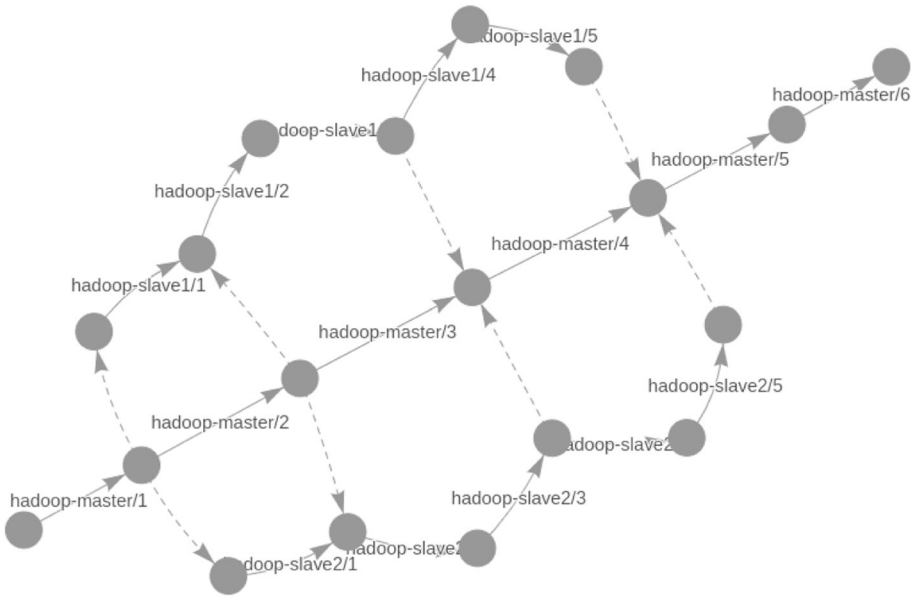

Fig. 6 Hadoop deployment graph with 1 master and 2 slave nodes

an application comprising more nodes with minimal execution overhead. Second, we study the deployment behavior when transient errors appear with *varying frequency*, measuring not only the total execution time but also the overhead introduced due to our filesystem snapshot mechanism. Third, we outline the differences between the implemented *snapshot mechanisms*, i.e., *AUFS* and *BTRFS*. Finally, we compare AURA to Openstack Heat, a popular, state-of-the-art deployment system.

Deployment model scalability

We begin our discussion through evaluating our scheme's scalability when an increasing number of modules/VMs is deployed to an error-free environment, i.e., no transient failures occur. We deploy the two considered applications and increase the `multiplicity` parameter for one of their modules, i.e., the *Web Server* and *Hadoop Slave* modules for Wordpress and Hadoop respectively and measure the time needed to complete each deployment phase. We consider three deployment phases: (a) The resource allocation (*Alloc*) phase, in which Openstack allocates the necessary resources, (b) the VM booting phase (*Boot*), in which the guest OS boots and (c) the software configuration phase (*Conf*), where the deployment scripts are executed. Note that, when multiple VMs/modules are considered, each phase is concurrently executed on each one separately. The presented time equals the *real* execution time, i.e., the time between the first VM entering a phase until the last VM leaving this phase. Figure 7 presents our results (mean times of 10 runs).

Figure 7 demonstrates that both applications present similar behavior. The resource allocation time presents a marginally increasing trend when more VMs are deployed, whereas the boot time remains constant. In both cases, the configuration phase presents the largest increase, which is, in fact, linear to the number of deployed VMs. The reason behind this rapid increase, though, is not attributed to the deployment model, but to the resource contention introduced when multiple VMs compete for the same resources. Specifically, the configuration scripts for both toy applications assume that they operate on a vanilla VM image and try to configure the entire environment from the beginning, download many software packages from the Web and increasing the deployment's network requirements. This means that an increase in module multiplicity results in a linear increase to the size of files needed to be downloaded and, hence, linear increase in the configuration time. In order to eschew this misleading behavior and avoid the network bottleneck, we repeat the same experiment, but this time we utilize a *prebaked* image that contains the raw software packages (though unconfigured). Figure 8 demonstrates our findings.

Figure 8 depicts that now that the network bottleneck is removed, our deployment scheme achieves to execute the configuration phase in practically constant time, regardless of the number of deployed VMs. This behavior drastically changes the relationships between the times of the deployment phases, making the resource allocation phase dominant of the entire deployment, whereas, again, the booting time is constant and the configuration phase presents marginal increase with the number of deployed VMs. Moreover, note that the absolute times remain extremely low: AURA achieved to deploy a Hadoop cluster of 1 Master and 8 Slaves in less that 200 s, a time that can be further decreased if AURA operates on an enterprise cluster with a faster storage medium that accelerates resource allocation.

Transient error frequency

We now evaluate our deployment model's behavior when transient errors appear. In order to produce such transient errors in a controllable and reproducible way, we inserted code snippets inside all application configuration scripts that lead them to failure with a given probability. Specifically, every time a deployment script is executed, a random number is drawn from a uniform distribution between $[0, 1]$ and if it is lower than p, where $0 < p < 1$,

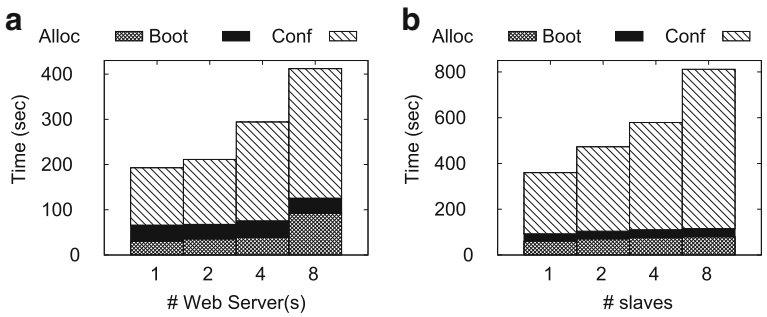

Fig. 7 Execution times of different deployment phases for varying module multiplicity using vanilla images. **a** Wordpress. **b** Hadoop

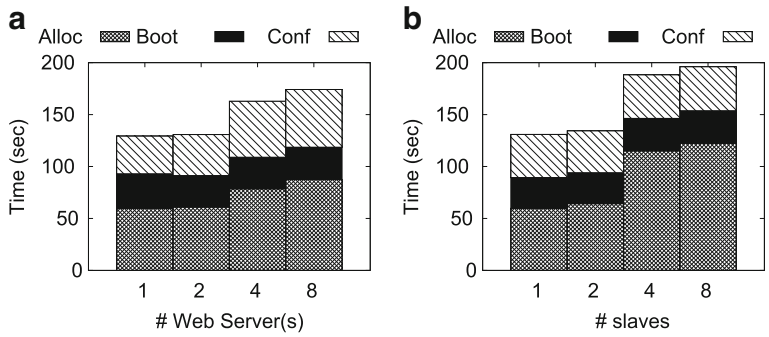

Fig. 8 Execution times of different deployment phases for varying module multiplicity using prebaked images. **a** Wordpress. **b** Hadoop

the script is terminated with a non-zero exit code, leading AURA to interpret this as a failure. The code snippet that generates this behavior is introduced *in the beginning* of each deployment script, hence, each failing script has minimal running time.

One option that highly impacts AURA's behavior for error identification is the timeout period used for detecting errors. As described previously, the default value equals 1 min. However, when comparing this interval with the actual configuration time for both applications depicted in Fig. 7, one can notice that 60 s is a sizeable portion of the entire configuration time. Hence, it can produce slow reactions to errors. To this end, and after experimenting with different values, we concluded that a good approximation of the best timeout interval equals $25\% \times \frac{real\ deployment\ time}{\max\ no.\ scripts/module}$. Intuitively, the fractional part represents the "mean deployment time per script", as if the deployment time was uniformly distributed to all scripts of the module with the most scripts. The percentage indicates the portion of this "mean deployment time" that AURA should wait before checking for errors. In our case, 25% indicates that AURA will approximately do $3 - -4$ health-checks before the script terminates (in an error-free case). This allows both fast reactions and a minimal number of unnecessary health-checks. With this rule of thumb in mind, we calculate the timeout interval for both applications as follows: $t_{Wordpress} = 25\% \times \frac{127}{3} \approx 10\ sec$ and $t_{Hadoop} = 25\% \times \frac{257}{6} \approx 10\ sec$. The real deployment time for each application is obtained by Fig. 7 for the 1 Slave/Web Server case and the maximum number of scripts/module is obtained by Figs. 3 and 6: 3 (for `db-server`) and 6 (for `hadoop-master`) respectively.

Given this, we deploy Wordpress (2 Web Servers and 1 Database Server) and Hadoop (1 Master and 2 Slaves) 10 times for varying p values from 0.05 to 0.8, measuring the total number of script executions and the real time of the configuration deployment phase. In Fig. 9 we provide the mean values and the respective deviation. We utilized AUFS as the filesystem snapshot mechanism.

Both Fig. 9a and b present very similar patterns: When the error probability is low (e.g., $0.05 - 0.20$), a minimal number of script executions occur and the configuration time remains very close to the configuration time witnessed to the error-free (i.e., $p = 0$) case. However, when p increases one can observe that both the mean and the standard deviation values rapidly increase. This is attributed to the fact that achieving error-free execution becomes exponentially more difficult, since the execution of a deployment script requires the successful execution of all the scripts it depends on and, hence, much more script executions and time is needed in order to achieve this. Moreover, the two plots also showcase that an increase

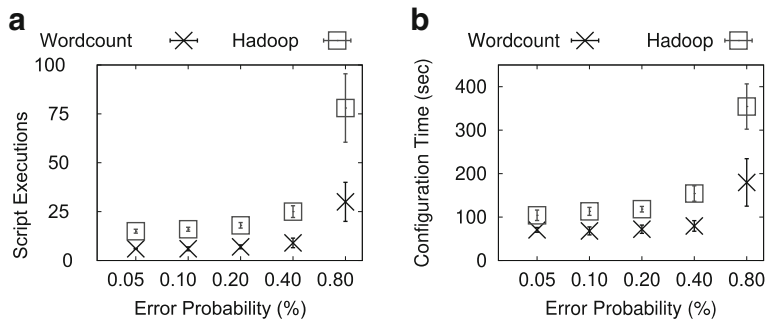

Fig. 9 Number of script executions and configuration times for varying error probability. **a** Script Executions. **b** Configuration Time

in p has a greater impact on more complex deployment graphs: Indeed, Hadoop presents a faster increase both in terms of the number of Script Executions and the respective configuration time it requires to be deployed, since it presents more dependencies (Fig. 6) than Wordpress (Fig. 3) and, hence, is more susceptible to script re-executions when errors appear more frequently.

A subtle point of the above discussion, is that this experiment implies that transient failures are independent of each other, in the sense that the emergence of a transient failure in a certain deployment script does not affect another, concurrently executed deployment script. In reality, different transient errors may be strongly correlated: If a network glitch occurred due to a failed switch, chances are that VMs belonging to the same host or rack will equally be affected, hence, their error probabilities are not independent. Although we do recognize that such correlated failures may prolong deployment times, we opted for a simpler transient failure generator in order to simplify the evaluation and obtain a better understanding of AURA's ability to overcome random failures. Furthermore, we should note that the p values used for this discussion are extremely large and only used in order to investigate what happens even in the most unstable cloud environments. It is interesting, though, that even when $p = 0.2$, AURA achieves to overcome any transient error and lead the deployment to successful termination with marginal delays, i.e., less than 10% when compared to the error-free case, both for Wordpress and Hadoop.

Snapshot implementation overhead

We now evaluate the performance of the two snapshot mechanisms we implemented in order to guarantee the idempotent script execution. Specifically, we want to evaluate the overhead that AUFS and BTRFS introduce to the configuration time, i.e., how much time is spent to snapshots and rollbacks against the "useful" deployment time, i.e., the time spent executing the configuration scripts. To this end, we deploy Wordpress and Hadoop using the same multiplicities as before, but now repeat each deployment twice: Once using AUFS for snapshots and once using BTRFS. We launch deployments for varying error frequencies, i.e., different error probabilities, and repeat each deployment 10 times. In Figs. 10 and 11 we provide the mean values of those runs for Wordpress and Hadoop respectively. The left figures depict the total snapshot time (including both snapshots and rollbacks) for all application scripts and the right figures express this time as a percentage of the total execution time. Observe the difference between this time expression and the time expression used so far: The total snapshot time represents the sum of time periods that the snapshot mechanism is triggered for each module without taking into consideration that different modules may run in parallel. For example, if module (1) and (2) required times t_1 and t_2 for snapshotting, the Total Snapshot time equals $t_1 + t_2$ whereas the *real* snapshot time (if running in parallel) would be $\max(t_1, t_2)$[5]. Finally, the Relative Time equals the Total Snapshot Time divided by the Total Running Time, i.e., the sum of execution time for each script of each module.

Both Figures demonstrate that BTRFS outperforms AUFS measured both in terms of Total Snapshot Time and in terms of relative time. In fact, the difference between the two mechanisms is increasing for larger p values, i.e., more snapshots and rollbacks are essential. Interestingly, one can also observe that AUFS achieves similar to BTRFS snapshot times for lower p values for Wordpress but, in the Hadoop case, AUFS requires much more time that BTRFS to snapshot and restore the filesystem layers. This interesting finding can be explained when examining the content that is snapshot: In both cases, AURA snapshots the `/opt` directory which is used as the root directory for all application modules. In the Wordpress case, `/opt` contains fewer files (Wordpress files) that aggregate 30MB, whereas in the Hadoop case `/opt` contains much more files that aggregate 500MB (Hadoop bin and configuration files). This is indicative of BTRFS' ability to handle massive storage in a more efficient way:

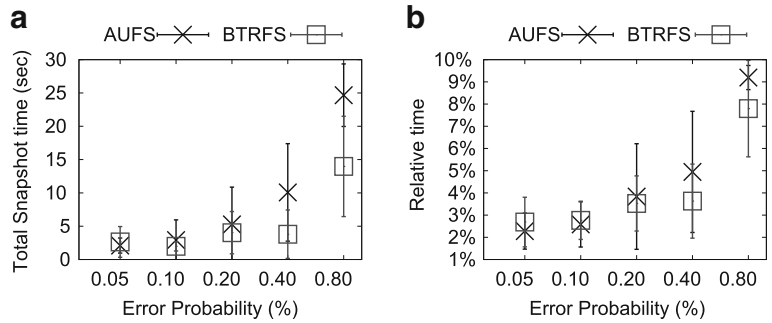

Fig. 10 Overhead of AUFS and BTRFS for Wordpress. **a** Total Snapshot Time. **b** Relative Time

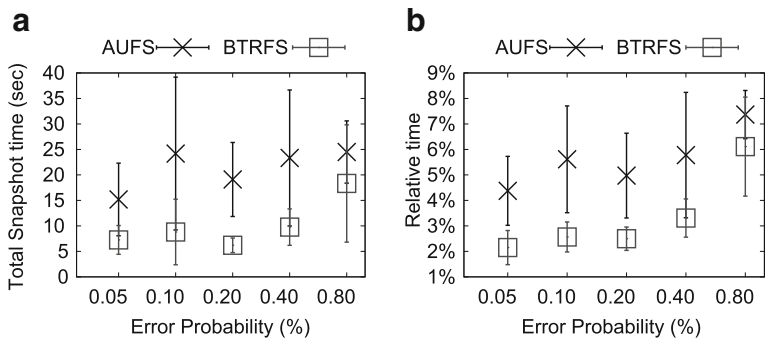

Fig. 11 Overhead of AUFS and BTRFS for Hadoop. **a** Total Snapshot Time. **b** Relative Time

Due to the powerful copy-on-write mechanism that operates on the inode level, BTRFS is able to create snapshots and revert to existing snapshots much faster than AUFS that needs to re-assemble the chain of layers from scratch any time a new layer is added/removed. On the contrary, when fewer data are persisted (as in the Wordpress case), both mechanisms can produce equivalent results.

Despite presenting superior performance, BTRFS has more runtime requirements than AUFS. To begin with, both mechanisms require kernel support. However, AUFS can operate on top of other filesystems and require no special devices (e.g., a dedicated block device) since it only requires a set of directories in order to work, which are located in the VM's filesystem. On the contrary, BTRFS requires a dedicated block device to be formatted and mounted in order to work. Although this does not limit its applicability, it reduces AURA's transparency towards the application to-be-deployed, as the user needs to take into consideration BTRFS' requirements in order to utilize it. In Table 1 we summarize AUFS' and BTRFS' requirements. Given the above, we can conclude that when one wants to snapshot a filesystem containing massive amounts of data, BTRFS must be preferred, despite its higher requirements; in cases where one needs to snapshot filesystems with fewer data, AUFS presents a decent behavior and has much fewer requirements.

End-to-end performance comparison

We now wish to evaluate AURA's end-to-end performance in comparison to Openstack Heat, i.e., a state-of-the-art deployment system that is frequently used in the Openstack ecosystem in order to orchestrate application deployments. Our evaluation's objective is to compare the deployment times of Hadoop and Wordpress for different multiplicities, when these are deployed through AURA and Heat. Since Heat does not support recovery from transient failures, only the error-free cases are considered. Furthermore, since Heat does not support circular dependencies between different software modules, we re-wrote the application descriptions of both applications, constructing two *Heat Orchestration Templates* (*HOTs*) that avoid such loops. Specifically, we conducted the following modifications:

- In the *Hadoop* case, we used predefined SSH keys (hence no negotiation is required between the master and the slaves) and only kept the dependencies from the slaves to the master (sending their IPs and informing the latter that the former ones are ready). Schematically, only the dotted edges beginning from Hadoop slaves towards the Hadoop master are kept, as seen in Fig. 6.
- In the *Wordpress* case, the Database Server does not wait for obtaining the IP address(es) of the Web Server(s), as it is configured to listen to any connection. Schematically, we removed the dotted edges from the output states of `web-server1/1`, `web-server2/2` scripts to the input of `db-server/3` of Fig. 3.

Finally, since Heat does not support an information exchange mechanism as the one supported by AURA [46], messages are exchanged inside the deployment scripts that securely transfer (through `scp`) any piece of information is anticipated by a consumer script. Given the above, Table 2 provides the relative deployment times for each application when a varying number of modules (i.e., slaves for Hadoop and Web Servers for Wordpress) is utilized. The relative deployment time is expressed as the ratio

Table 1 Requirements of AUFS and BTRFS

Requirement	AUFS	BTRFS
Kernel support	Yes	Yes
Block device	No	Yes
Cooperates with other FS	Yes	No

Table 2 Relative Deployment Time for Openstack Heat vs AURA

Application	Deployment phase	Number of modules			
		1	2	4	8
Hadoop	Allocation	1.0010	0.9991	1.0001	1.0020
	Booting	1.0001	0.9999	0.9991	0.9997
	Configuration	1.0000	0.9964	0.9951	0.9889
Wordpress	Allocation	1.0005	1.0030	0.9997	0.9995
	Booting	1.0001	0.9999	1.0003	1.0020
	Configuration	0.9991	0.9965	0.9903	0.9862

between the deployment time in Heat divided by AURA's time, i.e., $T_{relative} = \frac{T_{Heat}}{T_{AURA}}$. Each deployment was repeated 10 times and the mean values are presented.

A closer examination of Table 2 presents some interesting findings. First of all, both Heat and AURA present similar times during the *Allocation* and *Booting* phases of the deployment and this is not affected by the number of employed modules. This is a reasonable finding, since both tools use the same mechanism to instantiate the VMs, i.e., they both contact the `nova` component in order to issue requests for new VMs. Furthermore, the VM boot time is independent of the employed deployment tool, hence no differences in the VM booting time is observed. This is the reason behind obtaining $T_{relative}$ values close to 1 for both applications during these first deployment phases.

Investigating the respective times during the *Configuration* phase, nevertheless, showcases that Heat requires slightly less time when deploying applications of higher multiplicity (i.e., for 8 modules) than AURA; a difference that does not surpass 2% in the worst case. This marginal difference is ascribed to two factors. First, and as described previously, before the execution of any configuration script, AURA keeps a snapshot of the underlying filesystem in order to achieve idempotency. Although this extra time is negligible, it is an extra step that is avoided by Heat. Second, in this experiment Heat deploys slightly modified deployment graphs that present fewer dependencies between different modules. Consequently, the synchronization points for Heat are fewer and the deployment scripts are allowed to be executed without waiting for other scripts to finish.

It should be stressed, though, that Heat's lack of support for dynamic information exchange between different modules leads to slightly reduced deployment times in comparison to AURA, but also limits its expressiveness for describing complex applications. On the other hand, AURA's ability to execute complex deployment graphs and support recovery from transient failures, add a negligible time overhead that is instantly counterbalanced in the emergence of transient errors.

Conclusions and future work

In this work, we revisited the problem of application deployment to cloud infrastructures that present transient failures. The complexity of the architecture of modern data centers makes them susceptible to errors that hinder automation and jeopardize the execution of application deployment, a complex task that requires coordination between multiple different components. To address this challenge, we introduced AURA, a cloud deployment system that attempts to fix transient errors through re-executing the part of the deployment that failed. In order to enforce script idempotency, AURA implements a lightweight filesystem snapshot mechanism and, in case of script failure, cancels any unwanted side effects through reverting the filesystem to a previous, healthy state. Our evaluation, conducted for popular, real-world applications, frequently deployed to cloud environments, indicated that AURA manages to deploy applications even in infrastructures presenting high error probabilities and only introduces a minimal overhead to the deployment time that does not surpass 10% of the total deployment time in the worst case.

Let us, now, summarize the takeaways of our work. The suggested deployment model that formulates an application deployment as a DAG of dependencies, provides the necessary building blocks for expressing extremely complex tasks in an efficient way. The introduction of synchronization through message exchange between different modules (or VMs), in particular, increases our model's expressiveness and efficiently addresses the limitation of many real-world deployment systems that assume that two software modules will not depend on each other on the same time, i.e., they only support unidirectional dependencies. Furthermore, in this work we attempted to utilize the well-known technique of filesystem snapshot, in order to give a solution to the problem of idempotent script execution, which is a key requirement for re-executing failed scripts. In our best knowledge, our work is the first that tries to achieve idempotency through it and, according to our evaluation, this technique achieves to nullify any unwanted

script side effects to filesystem related resources in an efficient manner and only introduces marginal overhead. As the existing filesystems improve and new ones emerge, this technique can produce even better results.

Finally, the contributions of this work provide a strong foundation for future work. First, the expressiveness of the suggested deployment model renders it as a suitable candidate for expressing complex tasks not only in the initial application deployment phase, but also during the runtime of an application. Elastic scaling, which is a key concept to the cloud industry [47], requires automated resource provisioning and software configuration, so that resources are utilized appropriately. The adoption of a deployment model as the one suggested in our work would facilitate enforcing complex application resizing actions and increase resource reliability. Second, in this work we studied script idempotency in the light of resources that reside in the filesystem. Even if this hypothesis seems realistic for the deployment phase, as a typical software configuration script handles and modifies files (configuration files, binaries, libraries, etc.), in the general case, idempotency is not achieved to other resources, e.g., the memory state of a process. This is an interesting future direction of investigation that would maximize our work's applicability and allow for considering alternative tasks that exceed the scope of this work. Third, although AURA is capable of addressing concurrent errors in different software modules, it assumes that these errors are independent of each other, as each part of the deployment graph is treated independently. However, as previously discussed, in practice errors can be strongly correlated. This observation provides an interesting foundation for future research as studying the nature of this correlation can contribute to taking smarter decisions during recovery actions and further accelerate deployments.

Endnotes

[1] According to Greek mythology, Aura was the goddess of breeze, commonly encountered to cloudy environments.

[2] Note that message transmission might not be instant (as implied by the Figure) since consumption of a specific message might occur much later than the message post, but the arrows are depicted perpendicular to the time axis for simplicity.

[3] https://aws.amazon.com/emr/

[4] More details, though, can be found online at [48] for Hadoop and [49] for Wordpress.

[5] In the Operating Systems realm, the running time of a process is distinguished to *real*, *user* and *sys*, where the first denotes the actual clock time, the second equals the total time spent at user space for all threads and the last equals the time spent at kernel for all threads. So far, we have used the *real* time; in this experiment we prefer to demonstrate the *user* time in order to better isolate the snapshot behavior.

Abbreviations

DAG: Directed acyclic graph; VM: Virtual machine

Acknowledgements

We wish to thank the anonymous reviewers for their valuable feedback for improving the quality of the manuscript.

Availability of data and materials

The experimental evaluation was based on deployments executed in a private Openstack environment which is not publicly accessible. However, the reader can recreate the results discussed in our work, through downloading AURA [22] and deploying the tested applications located in [48, 49].

Authors' contributions

All authors equally contributed in this work. All authors read and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests.

Publisher's Note

Author details

[1] Computing Systems Laboratory, School of ECE, National Technical University of Athens, Iroon Polytechniou 9, 15780 Zografou, Greece. [2] Department of Informatics, Ionian University, I. Theotoki 72, 49100 Corfu, Greece.

Received: 11 December 2017 Accepted: 7 June 2018
Published online: 26 June 2018

References

1. Armbrust M, Fox A, Griffith R, Joseph AD, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I, et al (2010) A view of Cloud Computing. Commun ACM 53(4):50–58
2. Bass L, Weber I, Zhu L (2015) DevOps: A Software Architect's Perspective. Addison-Wesley Professional
3. Kapur K. C, Pecht M (2014) Reliability Engineering. John Wiley & Sons
4. Hanappi O, Hummer W, Dustdar S (2016) Asserting reliable convergence for configuration management scripts. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. ACM. pp 328–343
5. Hummer W, Rosenberg F, Oliveira F, Eilam T (2013) Testing idempotence for infrastructure as code. In: ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing. Springer. pp 368–388
6. Katsuno Y, Takahashi H (2015) An automated parallel approach for rapid deployment of composite application servers. In: Cloud Engineering (IC2E), 2015 IEEE International Conference On. IEEE. pp 126–134
7. Juve G, Deelman E (2011) Automating application deployment in infrastructure clouds. In: Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference On. IEEE. pp 658–665
8. Papadopoulos PM, Katz MJ, Bruno G (2003) Npaci rocks: Tools and techniques for easily deploying manageable linux clusters. Concurr Comput Pract Experience 15(7-8):707–725
9. Deelman E, Vahi K, Juve G, Rynge M, Callaghan S, Maechling PJ, Mayani R, Chen W, da Silva RF, Livny M, et al (2015) Pegasus, a workflow

management system for science automation. Futur Gener Comput Syst 46:17–35
10. Yamato Y, Muroi M, Tanaka K, Uchimura M (2014) Development of template management technology for easy deployment of virtual resources on openstack. J Cloud Comput 3(1):7
11. Antonescu A-F, Robinson P, Braun T (2012) Dynamic Topology Orchestration for Distributed Cloud-Based Applications. In: Network Cloud Computing and Applications (NCCA), 2012 Second Symposium On. IEEE. pp 116–123
12. Openstack Heat. https://wiki.openstack.org/wiki/Heat. Accessed 11 Dec 2017
13. Openstack Sahara. https://wiki.openstack.org/wiki/Sahara. Accessed 11 Dec 2017
14. Juju. https://juju.ubuntu.com/. Accessed 31 Dec 2017
15. AWS CloudFormation. http://aws.amazon.com/cloudformation/. Accessed 11 Dec 2017
16. Juve G, Deelman E (2011) Wrangler: Virtual cluster provisioning for the cloud. In: Proceedings of the 20th International Symposium on High Performance Distributed Computing. ACM. pp 277–278
17. Vagrant. https://www.vagrantup.com/. Accessed 11 Dec 2017
18. AWS Incident. https://goo.gl/f959fl. Accessed 11 Dec 2017
19. Google App Engine Incident. https://goo.gl/lCl0Mo. Accessed 11 Dec 2017
20. Giannakopoulos I, Konstantinou I, Tsoumakos D, Koziris N (2016) Recovering from cloud application deployment failures through re-execution. In: International Workshop of Algorithmic Aspects of Cloud Computing. Springer. pp 117–130
21. Giannakopoulos I, Konstantinou I, Tsoumakos D, Koziris N (2017) Aura: Recovering from transient failures in cloud deployments. In: Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. IEEE Press. pp 762–765
22. AURA Source Code. https://github.com/giagiannis/aura/. Accessed 11 Dec 2017
23. AWS Elastic BeanStalk. http://aws.amazon.com/elasticbeanstalk/. Accessed 11 Dec 2017
24. CloudFoundry. https://www.cloudfoundry.org/. Accessed 11 Dec 2017
25. Heroku. https://www.heroku.com/. Accessed 11 Dec 2017
26. AWS CloudFormation Documentation. https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/troubleshooting.html. Accessed 31 Mar 2018
27. AWS Elastic Load Balancing Documentation. https://docs.aws.amazon.com/elasticloadbalancing/latest/classic/ts-elb-error-api-response.html. Accessed 31 Mar 2018
28. Chef. https://www.chef.io/chef/. Accessed 11 Dec 2017
29. Liu C, Mao Y, Van der Merwe J, Fernandez M (2011) Cloud resource orchestration: A data-centric approach. In: Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR). cidrdb.org. pp 1–8
30. CFEngine. https://cfengine.com/. Accessed 11 Dec 2017
31. Puppet. https://puppet.com/. Accessed 11 Dec 2017
32. Ansible. https://www.ansible.com/. Accessed 11 Dec 2017
33. SaltStack. https://saltstack.com/. Accessed 31 Mar 2018
34. Potharaju R, Jain N (2013) When the network crumbles: An empirical study of cloud network failures and their impact on services. In: Proceedings of the 4th Annual Symposium on Cloud Computing. ACM. p 15
35. Zhai E, Chen R, Wolinsky DI, Ford B (2014) Heading off correlated failures through independence-as-a-service. In: OSDI. pp 317–334
36. Patel P, Ranabahu AH, Sheth AP (2009) Service level agreement in cloud computing. https://corescholar.libraries.wright.edu/knoesis/78/, hosted by Kno.e.sis Publications (The Ohio Center of Excellence in Knowledge Enabled Computing (Kno.e.sis))
37. AWS Maintenance. https://aws.amazon.com/maintenance-help/. Accessed 11 Dec 2017
38. VMware vCloud Automation Center Documentation Center. http://goo.gl/YkKNic. Accessed 11 Dec 2017
39. Rodeh O, Bacik J, Mason C (2013) Btrfs: The linux b-tree filesystem. ACM Trans Storage (TOS) 9(3):9
40. Heidemann JS, Popek GJ (1994) File-system development with stackable layers. ACM Trans Comput Syst (TOCS) 12(1):58–89
41. AUFS. http://aufs.sourceforge.net/. Accessed 11 Dec 2017
42. Borthakur D, et al. (2008) Hdfs architecture guide. Hadoop Apache Proj 53
43. Kumar R, Jain K, Maharwal H, Jain N, Dadhich A (2014) Apache cloudstack: Open source infrastructure as a service cloud computing platform. In: Proceedings of the International Journal of advancement in Engineering technology, Management and Applied Science. www.IJAETMAS.com. pp 111–116
44. Bernstein D (2014) Containers and cloud: From lxc to docker to kubernetes. IEEE Cloud Comput 1(3):81–84
45. White T (2012) Hadoop: The Definitive Guide. " O'Reilly Media, Inc."
46. Openstack Heat Signaling and Coordination. https://wiki.openstack.org/wiki/Heat/Signaling-And-Coordination. Accessed 31 Mar 2018
47. Giannakopoulos I, Papailiou N, Mantas C, Konstantinou I, Tsoumakos D, Koziris N (2014) Celar: automated application elasticity platform. In: Big Data (Big Data), 2014 IEEE International Conference On. IEEE. pp 23–25
48. Hadoop AURA application description. https://github.com/giagiannis/aura/tree/master/example/hadoop. Accessed 11 Dec 2017
49. Wordpress AURA application description. https://github.com/giagiannis/aura/tree/master/example/wordpress. Accessed 11 Dec 2017