Ops-Scale: Scalable and Elastic Cloud Operations by a Functional Abstraction and Feedback Loops

Kamal Hakimzadeh[1] and Jim Dowling[1,2]

[1]KTH - Royal Institute of Technology, Stockholm, Sweden
Email: mahh@kth.se

[2]Logical Clocks AB, Stockholm, Sweden
Email: jim@logicalclocks.com

***Abstract*—Recent research has proposed new techniques to streamline the autoscaling of cloud applications, but little effort has been made to advance configuration management (CM) systems for such elastic operations. Existing practices use CM systems, from the DevOps paradigm, to automate operations. However, these practices still require human intervention to program ad hoc procedures to fully automate reconfiguration. Moreover, even after careful programming of cloud operations, the backing models are insufficient for re-running such programs unchanged in other platforms—which implies an overhead in rewriting the programs.**

We argue that CM programs can be designed to be deployment-agnostic and highly elastic with well-defined abstractions. In this paper, we introduce our abstraction based on *declarative functional programming*, and we demonstrate it using a *feedback loop control* mechanism. Our proposal, called Ops-Scale, is a family of cloud operations that are derived by making a functional abstraction over existing configuration programs. The hypothesis in this paper is twofold: 1) it should be possible to make a highly declarative CM system rich enough to capture fine-grained reconfigurations of autoscaling automatically, and; 2) that a program written for a specific deployment can be re-used in other deployments. To test this hypothesis, we have implemented an open source configuration engine called Karamel that is already used in industry for large-scale cluster deployments. Results show that at scale Ops-Scale can capture a polynomial order of reconfiguration growth in a fully automated manner. In practice, recent deployments have demonstrated that Karamel can provision clusters of 100 virtual machines consisting of many-layers distributed services on Google's IaaS Cloud in '*less than* 10 *minutes*'.

***Index Terms*—Cloud Computing; Functional Programming; Elasticity; Auto-Scaling; Feedback Control Loop;**

I. Introduction

Many applications in cloud computing, e.g., IoT and 5G, are designed to be ephemeral and dynamic by exploiting cloud elasticity [8]. While much research has conducted on devising advanced techniques for autoscaling cloud applications to deliver acceptable quality of service (QoS) [25], [36], relatively less effort has been expended on adding dynamism and agility to configuration management (CM) solutions. Modern (micro-service) applications in the cloud have n-layer architectures with many more branches than the classical 3-tier architecture. The increased architectural complexity is reflected in an increasing number of tunable configuration parameters, for example, for tuning resource allocation, when to scale out/in, and how to handle faulty services. Typically, these reconfigurations are done with a human *'in the loop'*, which limits the frequency at which a production system can be adapted to such stack changes. Human experts develop ad hoc automation programs that work for specific setups, but that process needs to be adapted for the various combinations of applications, services, and platforms.

Currently, cloud applications are configured using CM systems, based on the DevOps paradigm [12], [39]. DevOps systems for automating IT application setup are unable to meet the need of modern applications, which may frequently request to add/remove services in a n-layer architecture. Even though layers in an n-layer architecture may be autoscaled horizontally; services may be added/removed inside the same layer and reconfigurations may cross-cut horizontal layers. For example, an autoscaled layer may affect the configuration of other layers as well, a *vertical pattern*, or a reconfiguration of an affected layer may affect the layer that initiated the process, a *circular pattern*. One group of CM systems rely on configuration servers (e.g., service registries, distributed datastores) for planning and propagating the changes of configurations. Existing server-based configuration models are non-elastic because they lose the track of the producers and consumers of configuration items (this is needed for reconfigurations), and existing models rely on ad hoc and non-elastic naming conventions.

The challenge has its root in the trade-off that CM systems make; declarative programming for agility at design time versus imperative programming for better expressing configuration in low-level detail (as a procedural set of steps). Declarative approaches usually have weaker support for advanced configuration patterns whereas imperative approaches easily become non-elastic, e.g., the use of hard-coded names in imperative programming (early binding).

To this end, we propose a new family of CM systems, named Ops-Scale, that can bring the advantages of both declarative and imperative approaches together, being highly agile and expressing low-level configurations. This is done by modeling cloud operations as functions, *operation function*, and making functions highly elastic by employing higher order programming. For example, dependencies between operations, how functions exchange information, and the placement of

X/x	Service/service-instance
λ	Operation: service or cloud.
$\prec_y$	Precedence order (dependency) w.r.t. y
$\prec^1/\prec^*$	One-to-one/all-to-all cardinality.
σ	Target (host) group.
Φ	Second-order function.
ϕ	First-order function.
$\phi\chi$	Data transformer function.
Δ	DAG (directed acyclic graph)
α^x	Action x

TABLE I: **Symbols used in the paper.**

functions are modeled as higher-order functions in our approach. Our approach divides the domain of application deployment into the two concerns of planning and actuation. An operation function generates two second-generation functions: a *planning function* (key function) and an *actuation function* (value function). Then, we make a feedback loop controller, named *Operation Controller*, which contains graph traversal control logic. The controller generates an ordered plan as a DAG, then it traverses the DAG and produces deployment actions. We design an actuator that can execute actions at multiple targets in parallel. The controller binds the values of variables from the results sent from the actuator; using a dataflow variable binding model [38]. Further, the operation controller treats an autoscaling action that is produced by an external controller (e.g., add/remove service instances) as a new deployment plan that has to be implemented by calculating the difference between the current plan and the new one. The contributions of this paper are the following:

- A novel approach for cloud operations based on declarative functional programming [38]. The model supports advanced configuration patterns and it is more elastic compared to existing solutions.
- An implementation of Ops-Scale in an open source project called Karamel [20], [21]. Karamel has been successfully used by organizations from both academia and industry to set up thousands of clusters [10], [15], [31], [19], [33].
- Comparison of our functional model with other models used in DevOps systems.

The rest of the paper is organized as follows, section II provides background terminology and motivations behind Ops-Scale. Section III introduces our functional abstraction while section IV provides details on the implementation of our control loop. Results are presented in section V, while section VI discusses limitations and next steps. Section VII elaborates on related work, and conclusions are drawn in section VIII.

II. Motivation and Background

In this section, we describe the challenges of modeling cloud operations, and we introduce declarative modeling in designing the operations. Throughout the paper, we use the symbols listed in Table I.

Service Operation (λ). We motivate Ops-Scale as a service-oriented architecture by using the term *service operation* in the DevOps sense. A service operation is a list of configuration steps that must be executed to start and run a service. In other words, service artifacts are given as a black box - we do not inspect or change its internal logic or the input/output data the service consumes/produces. For example, the service operation λDB for a distributed database service DB contains steps for installing, generating configuration files, opening ports, launching the service, and connecting the service to the rest of the system. The internal logic of services, such as bootstrapping, data replication, data migration, and state management of the data for DB, is not dealt with by λDB.

A. Challenges of service operations

Operation Dependencies($\prec$). Assume a provider of cloud services offers a platform consisting of three internal services: a database DB, a web server WEB, and a load balancer LB. The DB has no dependency on other services but the WEB needs the DB's endpoint, that is, the DB has to be launched before the WEB is launched. The LB is an optional service that can be used to balance the client requests on WEB instances. The LB should know the WEB endpoints and the WEB instances need to know the LB's endpoint in order to advertise the endpoint to their clients. As there is a precedence relationship $\prec$ between λDB and λWEB w.r.t. the endpoint $DB.ep$, DB and WEB can be launched correctly when the precedence $\lambda DB \prec_{DB.ep} \lambda WEB$ is followed. However, the dependency between λWEB and λLB is *bidirectional* because $\lambda LB \prec_{LB.ep} \lambda WEB$ and $\lambda WEB \prec_{Web.ep} \lambda LB$. Thus, if we encapsulate operations at the service level, there is no solution for correctly launching services with a bidirectional dependency.

Dependency Cardinality ($\prec^1/\prec^*$). In a distributed service environment, services are replicated to more than one instance; hence, incorporating service replication in our model. If we reconsider the precedence $\lambda DB \prec_{DB.ep} \lambda WEB$ w.r.t. instances db and web, there are two possibilities. Either all dbs should proceed all webs or each db only needs to proceed one web. For instance, the case where the webs are responsible for load balancing queries to dbs versus the case where the query routing done in the DB layer. We call the two dependency types *dependency cardinality* and we show them as: $\lambda db \prec^1_{db.ep} \lambda web$ versus $\lambda db \prec^*_{db.ep} \lambda web$. The notation $\prec^1$ shows a *one-to-one* cardinality and $\prec^*$ represents a *all-to-all* cardinality. Further, the dependency cardinality between two services may be different w.r.t. different data-items. For instance, if the db has two endpoints $ep1$ and $ep2$, then $\lambda db \prec^1_{db.ep1} \lambda web$ while $\lambda db \prec^*_{db.ep2} \lambda web$. Service discovery model that is used in service oriented architectures comes short for handling the cardinalities (to be discussed in § II-B).

Reconfiguration. Assume a cloud provider launches a new horizontal autoscaling feature for a cloud orchestrator service, which can autoscale offered services (DB, WEB, and LB) for cost saving purposes. Horizontal autoscaling is a resource optimization technique and it refers to adding new instances or removing some instances from a running service in order to adapt the service capacity to varying workloads. Horizontal autoscaling requires running services to be correctly reconfigured without interruption to clients. For removals, the cloud orchestrator service should, first, account for disconnecting the candidate instances from the rest of the system, then, it should terminate the candidate instances. For additions, the orchestrator service should launch new instances first and then connect them to the rest of the system by adhering to operation dependencies and cardinalities.

B. Shortcomings of current models.

Coarse-grained Operation Models. The problem with the presented model for service operation is that, when service operations are encapsulated as atomic units without intermediate interactions between the units, then, the bidirectional, or in a more generic sense *circular*, dependencies cannot be modeled or handled automatically. In circular scenarios, *updating exchanged configuration information*, e.g., updating service endpoints, among services becomes a bottleneck at scale. For instance, in a cluster with a controller host that has to have SSH access into other hosts, some steps should be taken in order: (i) launch the controller host, (ii) generate an SSH key-pair in the controller, (iii) launch the other hosts, (iv) copy the SSH public-key of the controller into the other hosts (the authorized-host file in Linux), (v) launch SSH service on the other hosts, and (vi) run the controller service. This order of operations has a loop between the controller host and the other hosts. The coarse-grain modeling of service operations, e.g., at the host level in the example, has no solution for circular scenarios.

Current orchestrators such as Docker Swarm [28] or Kubernetes [3] rely on a *service registry and discovery* model. In this model, providers or orchestrators have to register the producers' endpoints where consumers can find the producers using service names. The model has two shortcomings: (i) keeping the consumer notified as soon as a change happened, and (ii) distinguishing between service instances for separate consumers (the one-to-one dependency cardinality in § II-A). Consumers can *query the registry before calling the provider* every time. This places too much load on the registry and imposes an unnecessary performance impact. Instead, orchestrators offer an extra *router service* to a well-known location and clients make requests via the router. The router keeps updated with changes happening in the registry, and the router can act as a load balancer for routing client's calls to service instances. Although the solutions may solve the notification problem the specific mapping between consumer instances and producer instances must be programmed imperatively, which is laborious and error-prone.

Non-Elastic Naming Schemes. The naming problem in service discovery model exists for any configuration that we exchange between services. The programmable naming schemes, names that programmer decides instead of automatically generated, are not flexible to be used in deployments with arbitrary service combinations. We should make sure that names do not clash and we should know how to map configuration instances (e.g., service instances) between services.

From Service Operation to IaaS Cloud Operation (λ). So far, we utilized the domain of service operations and assumed that infrastructures, e.g., network and virtual hosts, are provisioned. Service provisioning may go beyond the domain of service operation if the cloud model is IaaS or PaaS. Therefore, we extend the notion of service operation to *cloud operation*; cloud operation is all the steps for making distributed services operational on IaaS cloud model. Examples of such operations are: (i) make a connection to the interface of an IaaS cloud, (ii) to provision virtual networks, (iii) fork virtual machines, (iv) perform service operations, (v) reconfigure the system when a change happened. Cloud operations have similar challenges as in service operations. Each successive pair of cloud operations have an order dependency on each other, the dependencies have cardinalities, and bidirectional dependencies exist.

C. Ops-Scale's Approach

Fine-grained Operation Model. If we split coarse-grained operations into a sequence of finer-grained operations it solves the problem of bidirectional dependencies. For example, we can split λWEB into a sequence of *micro-operations*: $\lambda WEB = \langle\ \lambda WEB.install, \lambda WEB.port, \lambda WEB.launch\ \rangle$ s.t. $\lambda WEB.install \prec \lambda WEB.port \prec \lambda WEB.launch$. Assuming that λLB contains the similar micro-operations, then the dependencies between the two service operations is reduced to: $\lambda WEB.port \prec \lambda LB.launch$ and $\lambda LB.port \prec \lambda WEB.launch$, which has no bidirectional dependency. The result essentially means that launching services must be delayed until the service ports are known and they are exchanged between the services. This should be done through *dynamic port reservation*; static port reservation suffers from the problems of non-elastic naming schemes. Although the fine-grained operation model is flexible enough for handling circular dependencies, it can become cumbersome for programmers and *adversely effect the agility* of designing service deployments. We argue that if fine-grained cloud operations are *formalized with an appropriate abstraction*, they can be *reused* in many deployments and they can become *fully automated* without the need for human intervention.

Declarative Functional Programming. *Declarative programming* describes a family of computer programs defining the *'what'* without explaining the *'how'* (see [38] for a canonical reference). More precisely, any programming component that is *independent*, *stateless*, and *deterministic* is declarative. Moreover, any program that is a composite of declarative components is also declarative. *Functional programming* is an important technique used to design declarative models – using functions that adhere the declarative properties[1] (*pure functions*). A declarative function can be run independent of other functions, it doesn't need any sort of long term memory for learning from its past, and it generates the same results every time it is called with the same input. A *second-order*

[1]We call this class of programming *'declarative functional programming'*.

function ΦF accepts other functions (first-order ϕD or second-order ΦD) as arguments. The higher order programming makes functions more generic as it relieves programmers from writing new versions for a function.

Modelling Cloud Operations as Functions. Inspired by declarative programming, we propose that cloud operations can be modeled in a fully functional fashion. Leveraging higher-order programming, a big stack of function dependencies can be realized declaratively. Moreover, higher-order programming gives a high degree of freedom in permuting cloud operations. For instance, when we program λWEB we do not need to bind `WEB` to only one specific type of database system, but a λDB is passed as a higher order function to the λWEB. With a similar programming style, service operations can be programmed independently of any specific IaaS cloud (e.g., cloud provider, host, and network). In other words, we write functions only ***once*** but we design them to be highly *adaptive* using higher-order programming. In § III, we present our functional modeling for cloud operations.

III. Functional Modeling

Our model has two objectives: a highly flexible declarative modeling for mixing services in deployments, and highly automated reconfiguration for elastic actions – to *add* or to *remove* service instances.

Functional Model with Declarative Properties. Making cloud operations declarative (independent, stateless, and deterministic) is challenging for the following reasons: cloud operations have state, e.g., operations fail in the middle and leave the system in a dirty state, cloud operations have dependencies, e.g., the launch operation of a service is dependent on the install operation of the service, and, finally, cloud operations are non-deterministic, e.g., re-running an operation for opening a port returns different port numbers each time. To overcome this, we use two tricks. First, we split cloud operations into the separate concerns of *plan* and *action*. A plan is the description of the results of an operation, e.g., the components and the number of instances of them that have to be built. Action is the making routine/code of operation. As such, we only model the planning part of cloud operations with declarative functions because a plan is independent, stateless, and deterministic. Second, we relate each plan with the corresponding action by introducing a *key-value* model for system state. In other words, a plan function generates globally unique keys for the expected result of operation whereas the action assigns values to the keys after actuating the system. We encapsulate the plan and the action of operation inside a function, called *operation function*, and the function is used by our controller (to be explained in § IV). For functions, we use the notation $f : \langle x_1..x_n\rangle \Rightarrow y$; that is, function f maps the arguments $x_1..x_n$ to the result y [2].

Operation Function (ΦF) is second-order function as below:

$$\Phi F : \langle(\Phi D_i, \phi\chi_i)\rangle_{i=1}^{n} \Rightarrow \langle(label_j^{\Phi F}, \phi Key_j^{\Phi F}, \phi Val_j^{\Phi F})\rangle_{j=1}^{m}$$

[2]While we could express our model in other formal systems such as lambda calculus, we chose our functional notation for the sake of readability; functional programming lifts the level of abstraction above lambda calculus.

The input is a list of tuples $(\Phi D, \phi\chi)$, where, ΦD is another operation function that ΦF is dependent on, and $\phi\chi$ defines how to transform data from ΦD to ΦF.

Key Function ($\phi Key_j^{\Phi F}$) or an instance of key function is a first-order function produced by ΦF and it is modeled as:

$$\phi Key_j^{\Phi F} : \langle\overrightarrow{key}^{\Phi D_i}\rangle_{i=1}^{n} \Rightarrow \overrightarrow{key}^{\Phi F}$$

Key Function $\phi Key_j^{\Phi F}$ receives an equal number of input arguments as in ΦF, each argument $\overrightarrow{key}^{\Phi D_i}$ is a *key vector* produced by the dependency ΦD_i. A key function generates a key vector too.

Value Function ($\phi Val_j^{\Phi F}$) or an instance of Value Function is a first-order function produced by ΦF and it is modeled as:

$$\phi Val_j^{\Phi F} : \langle\overrightarrow{(key, val)}^{\Phi D_i}\rangle_{i=1}^{n} \Rightarrow \overrightarrow{(key, val)}^{\Phi F}$$

Value Function $\phi Val_j^{\Phi F}$ receives an equal number of inputs arguments as in ΦF, each argument $\overrightarrow{(key, val)}^{\Phi D_i}$ is a *key-value vector* produced by the dependency ΦD_i. A value function produces the values of its own keys, and it binds the keys to values as results.

Grouping ΦF and Instantiating ϕKey and ϕVal. We incorporate the notion of *group* (σ) in our model for two purposes: to associate related operations together, and to reference operations by group in defining operation dependencies. For example, to run λDB at a host group σ_1 ($\lambda DB \in \sigma_1$) or to depend $\lambda WEB \in \sigma_2$ on $\lambda DB \in \sigma_1$. In an operation function ΦF, group is given as an argument (not shown in the formula for simple). A group σ has a variable size $|\sigma| = m$ and the size is changed by auto-scaling actions (to be shown in § IV). The produced functions ϕKey and ϕVal have m instances and the number of instances is changed as m is changed.

Data Transformers ($\phi\chi$). We realize dependency cardinalities between operation functions ΦF and ΦD with data transformers. Data transformers are also functions for mapping the results from upstream functions to the arguments of downstream functions. We transform data from the instance of $\phi Key^{\Phi D}$ to the instances of $\phi Key^{\Phi F}$ and from the instances of $\phi Val^{\Phi D}$ to the instances of $\phi Val^{\Phi F}$. Transformer $\phi\chi Select$ implements an one-to-one dependency by mapping function instances by index. Transformer $\phi\chi Collect$ implements an all-to-all dependency by placing result of function instances in a vector sorted by index. For $\phi Key_j^{\Phi F}$ (and similarly for $\phi Val_j^{\Phi F}$):

$$\phi\chi Select : \langle\overrightarrow{key}^{\Phi D_i}, j\rangle \Rightarrow \overrightarrow{key}_j^{\Phi D_i}$$
$$\phi\chi Collect : \langle\overrightarrow{key}^{\Phi D_i}\rangle \Rightarrow \langle\overrightarrow{key}_0^{\Phi D_i}, \overrightarrow{key}_1^{\Phi D_i} ..., \overrightarrow{key}_m^{\Phi D_i}\rangle.$$

Intuition of the Declarativeness. In our model, ΦF and ϕKey have a pure functional model but ϕVal doesn't because the results of value function are dependent on the state of the system under operation (c.f. to our earlier discussion about the declarative properties). Intuitively, assume we want to run a chain of dependent operations $\lambda S1 \prec^{c1} \lambda S2 \prec^{c2} \lambda S3$ where we have two implementation alternatives for $\lambda S1$ are: $\lambda S1_1$

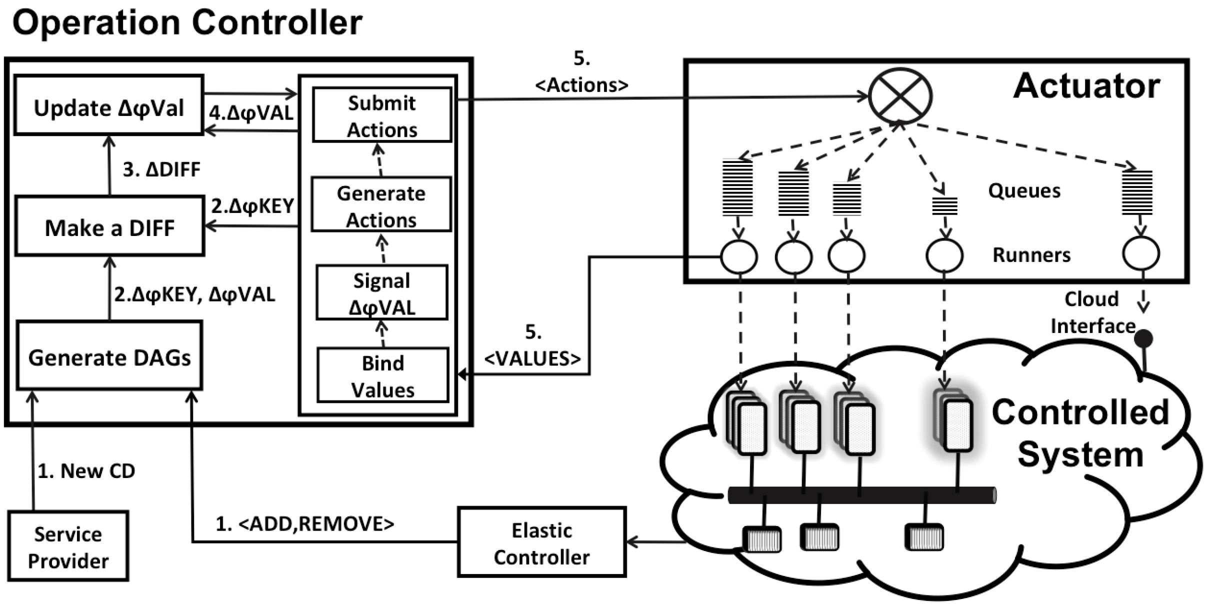


Fig. 1: **Ops-Scale's Control Loop.** 1) A service provider submits a cluster definition (CD) / an elastic controller requests to *add* or *remove* instances. In both cases, the controller generates a new $\Delta\phi Key$ and a new $\Delta\phi Val$ for the new desired state. 2) The controller generates a $\Delta DIFF$ by comparing the current $\Delta\phi Key$ and the new one. 3) The planner uses $\Delta DIFF$ to find the corresponding ϕVals and adjust the new $\Delta\phi Val$ for ultimately generating proper actions (i.e., α^{build}, α^{check}, $\alpha^{reconcile}$, and α^{purge}). 4) The planner submits the new $\Delta\phi Val$ for execution. 5) The control mechanism continues in loops between the DAG executor and the actuator until the execution is done. 5.1) The executor generates the actions of the ready operations. 5.2) The actuator dispatches actions to the associated target queues. 5.3) The runners make the actuation, collect the result, and callback the DAG executor and submit the values. 5.4) The executor binds values for the waiting operations, signals the finished operations, and make a list of ready operations.

and $\lambda S1_2$. Moreover, $c_1 = 1$ if $\lambda S1{=}\lambda S1_1$ and it is $c_1 = *$ if $\lambda S1{=}\lambda S1_2$. Further, the placement of operations and group sizes are not fixed – depending on deployments. The baseline for a deployment is to call the $\Phi S3$ with default values. This call will make a chain of function calls to the corresponding operation functions ($\Phi S3$, $\Phi S2$, $\Phi S1$) but in the reverse order. The chain of dependencies, and the generation of key functions and value functions are done declaratively. For changing the choice of $\lambda S1$ to $\lambda S1_2$, we should pass the corresponding operation function and transformer with higher order programming: $\Phi S3(f_2 = \Phi S2(f_1 = \Phi S1_2, tx_1 = \phi\chi Collect))$. With a similar approach we can assign the operation functions to various groups. As such, the system can automatically emit an ordered chain of actions (α^x encapsulated in ϕVal where x, the type of the action, is a variable depending on the scenario) by traversing operations in the chain. The execution of the model and the usages of ϕKey and ϕVal will be shown in § IV.

IV. Operation Controller & Implementation

In this section, we cover the execution of our functional model and run-time system considerations. We elaborate on the abstraction of the value function as we demonstrate our solution in a *feedback loop* architecture (see Figure 1).

Value Functions and Actions. A value function encapsulates an operation, and it binds values to keys based on the results of the operation. Operations, however, have to be run at remote targets (IaaS API, host, etc.). As such, we partition the logic of the ϕVal into three phases of: *action generation*, *action execution*, and *value binding*. The action generation and value binding are done centrally, but actions are sent to remote targets for execution. Considering the non-deterministic nature of actions and other sources of failures in systems, the execution of actions can produce many feedback loops. We design a controller, called *operation controller*, in order to handle the loops (shown in the Figure 1). Holding onto the same abstraction for ϕVal and feedback loop controller, a range of actions that are related to the same operation can be produced and executed. For instance, for an install operation, α^{build} is used for performing the installation, α^{check} is used for checking if the software is already installed, $\alpha^{reconcile}$ is used to re-install the software, and α^{purge} is used for uninstalling.

DAG-Ordered Planning. The control mechanism is initiated by receiving a *cluster definition (*`CD`*)* as input, where `CD` is a set of operation function calls $\texttt{CD} = \{\Phi F_k()\}_{k=1}^{K}$. Controller builds a directed acyclic graph of operation functions ($\Delta\Phi F$) by recursively fetching all the dependencies (both direct and transitive). Then, it traverses $\Delta\Phi F$ from root to leaves and it generates two other DAGs: one for key functions ($\Delta\phi Key$) and one for value functions ($\Delta\phi Val$). Similarly, a running controller can receive input from external elastic controllers too. The operation controller treats both types of inputs as new cluster definitions; therefore, it needs to make a diff between the currently running cluster definition and the new one. The comparison is between the two $\Delta\phi Key$s; that is, it produces a union of the two DAGs where the nodes are labeled with added, removed, and non-changed. We make the comparison in the key-space using $\Delta\phi Key$s rather than the value-space because the former is deterministic and the latter is not. Therefore, the $\Delta\phi Key$ comparison produces exact information about the state-items that should be added removed/added. The result of the comparison is a $\Delta DIFF$ that is used for updating the $\Delta\phi Val$. For the update, we traverse the $\Delta DIFF$ from root nodes to leaves and we find the corresponding value function from $\Delta\phi Val$ to each modified key in the key function. We find the functions of each key by using the hierarchical name scheme for of keys that have the function label as the prefix, and search for the function label (refer § III). This pre-processing phase is done by the planner module of the controller (left-hand side boxes in the figure).

DAG Traversing Control Logic and Parallel Actuation. The execution module of the controller receives a $\Delta\phi Val$. The execution module traverses the $\Delta\phi Val$ from its roots, it emits actions, and it submits the actions to the actuator. The execution module waits until the results of running actions are returned as feedback to the controller. The controller binds the values of the successfully executed actions in the $\Delta\phi Val$, and it traverses down the $\Delta\phi Val$ one step further. Our actuator runs actions in parallel. For instance, all the instances of a ϕVal are visited together and their actions are produced at the same time. The actuator has an action queue per target (host); serializing the actions of the same target. The submitted actions by the controller are placed at the queue of their corresponding target. The actuator runs exactly one action per non-empty queue. That is, the controller implements a dataflow variable binding model [38] for delivering the results of upstream value functions to the downstream value functions in the $\Delta\phi Val$.

Karamel, Implementation as an Operation Engine. Our presented abstraction Ops-Scale is inspired by the lessons that we learned from our open source operation engine, called Karamel [21], [20]. Karamel has an observational view over functional programming [38]; that is, operations behave like functions instead of being implemented as functions. Moreover, we have a test prototype in Python as the proof of portability between Karamel and Ops-Scale; the full functional Karamel is planned for coming releases. Karamel implements the control logic and the actuator and we leverage on Chef [35] for coding the cloud operations (mostly reusing the already developed code from the large community of Chef developers). We support both JAVA API and YAML DSL for writing cluster definitions. Karamel can seamlessly provision on bare-metal servers [32] and public IaaS cloud offerings [7], [18].

Alias	Deployment Detail
D_1	$\lambda S_1 \prec^1 \lambda S_2 \prec^1 \lambda S_3$ $S_1, S_2, S_3 \in HG_1$
D_2	$\lambda S_1 \prec^* \lambda S_2 \prec^* \lambda S_3$ $S_1, S_2, S_3 \in HG_1$
D_3	$\lambda S_1 \prec^1 \lambda S_2 \prec^1 \lambda S_3$ $S_1 \in HG_1$,$S_2 \in HG_2$,$S_3 \in HG_3$
D_4	$\lambda S_1 \prec^* \lambda S_2 \prec^* \lambda S_3$ $S_1 \in HG_1$,$S_2 \in HG_2$,$S_3 \in HG_3$

TABLE II: **Service deployment** examples. All the deployments consists of three services `S1`, `S2`, and `S3`. In D_1 and D_2 instances of each service are collocated on one host group (HG_1) whereas in D_3 and D_4 services are located on separate hosts groups (HG_1, HG_2, HG_3). In D_1 and D_3 have one-to-one dependencies between services whereas D_2 and D_4 have all-to-all dependencies.

V. Evaluation

We evaluate Ops-Scale in this section and present comparisons over synthetic deployments representing real-world setups. We answer following questions: (i) Can Ops-Scale boost *agility* in programming cluster specific cloud operations (§ V-A)?; (ii) Can Ops-Scale boost *automation* of elasticity beyond existing solutions (§ V-B)?

A. Can Ops-Scale boost agility in programming cluster specific cloud operations?

For the answer, we assume four service deployments shown in Table II, where we compare the programming effort to write Ops-Scale's functions compared to three other models. The other models are presented in Table III,IV, and V. Even though we borrow the terminology of one specific system in writing the pseudocode, the presented model of each table is used in a number of other widely used DevOps systems, too. Table VI shows pseudocode for Ops-Scale.

Pre-constructed image encapsulation with data discovery model. The pre-constructed images are used both for virtual machines and Linux containers. As can be seen in Table III, per host-service placement and dependency type, a new image has to be programmed. Moreover, the service discovery in the one-to-one service dependency in D_3 is dependent on the static naming of service instances; the `x` in **`reg`** `s2-x` has to be named by programmers. In this model, the *service-host placement* and *static naming* reduce the reusability of images across various deployments even with similar services – needing a new image to be programmed per specific configuration.

Functional encapsulation with distributed data sharing model. Many of the widely used configuration management systems for IT products (e.g., Chef, Puppet, Ansible, SaltStack) have a functional programming encapsulation approach for operations. In Table IV, we show the script for SaltStack. The systems in this class usually use local and distributed

Alias	Container
D_1	**Image-1:**
	...ops_S1
	...ops_S2
	...ops_S3
D_2	**Image-2:**
	...ops_S1 ...**reg** S1
	...**disc** S1 ...ops_S2 ...**reg** S2
	...**disc** S2 ...ops_S3
D_3	**Image-3:**
	...ops_S1 ...**reg** s1-x
	Image-4:
	...**disc** s1-x ...ops_S2 ...**reg** s2-x
	Image-5:
	...**disc** s2-x ...ops_S3
D_4	**Image-7:**
	...ops_S1 ...**reg** S1
	Image-8:
	...**disc** S1 ...ops_S2 ...**reg** S2
	Image-9:
	...**disc** S2 ...ops_S3

TABLE III: A *pre-constructed image encapsulation* with *data discovery* model. We use the **Container-based** scripting model in systems like Docker or Kubernetes for the pseudocode and programming the deployments of Table II. **reg:** registering a service endpoint in the registry, **disc:** discovering services' endpoint from service registry.

stores for data sharing among the functions. In SaltStack *formula* is a function implementing an operation, *pillar* is a local datastore, and *mine* is a global datasotre. SaltStack has a flexible composition of formulas in *topfiles*. Each topfile is targeted for a group of hosts (minions), ids of which are queried through a pattern matching function.

The functions of the model are reused across varied host-service placements as long as the dependencies between services remain fixed: formulas of D_2 are reused in D_4. The local versus global storage and static naming weaken the reusability. For instance, as reading and writing to local storage (pillar) is used in D_1, the formulas of D_1 are unusable in D_3. Moreover, as the names of service instances are decided statically (x in s1−x versus S), formulas cannot be reused when dependencies between services are changed.

Entity-based encapsulation with pub-sub data sharing model. Apache Brooklyn models the operations of applications and services by an entity model (Table V). An entity contains operations (effectors) and attributes (sensors). Sensors are topics and inter-entity dependencies are defined through data transformers (enrichers such as joiner, aggregator, etc.). Effectors are bound to sensors based on a global naming resolution. A group of entities can supply each sensor (e.g., instances of the entity at scale more than one), but a specific enricher (first element in a collection) can be used to select the desired entity. Brooklyn uses the dataflow model for variables but this is not the default model; the variables that have the dataflow model have to be imperatively specified through latches.

Alias	SaltStack
D_1	**formula-1:**
	...ops_s1 **w-pillar** s1
	formula-2:
	...**r-pillar** s1 ...ops_s2 ...**w-pillar** s2
	formula-3:
	...**r-pillar** s2 ...ops_s3
	topfile-1:
	...{ formula-1,formula-2,formula-3}
	deploy topfile-1 on HG1
D_2	**formula-4:**
	...ops_s1 **reg-mine** S1
	formula-5:
	...**disc-mine** S1 ...ops_s2 ...**reg-mine** S2
	formula-6:
	...**disc-mine** S2 ...ops_s3
	topfile-2:
	...{ formula-4,formula-5,formula-6}
	deploy topfile-2 on HG1
D_3	**formula-7:**
	...ops_s1 **reg-mine** s1-x
	formula-8:
	...**disc-mine** s1-x ...ops_s2
	......**reg-mine** s2-x
	formula-9:
	...**disc-mine** s2-x ...ops_s3
	topfile-3:
	...{ formula-7}
	topfile-4:
	...{ formula-8}
	topfile-5:
	...{ formula-9}
	deploy topfile-3 on HG1
	deploy topfile-4 on HG2
	deploy topfile-5 on HG3
D_4	**topfile-6:**
	...{ formula-4}
	topfile-7:
	...{ formula-5}
	topfile-8:
	...{ formula-6}
	deploy topfile-6 on HG1
	deploy topfile-7 on HG2
	deploy topfile-8 on HG3

TABLE IV: A *functional encapsulation* with a mix of *local and distributed data sharing* model. We use **SaltStack** terminology in the pseudocode for programming the deployments of Table II. **formula:** an operation function, **pillar:** local data store, **mine** distributed data store (**r:** read, **w:** write), **topfile:** list of functions to be run by a host group

As can be seen in Table V, the entity model is abstracted from the host-entity placement, reusing entities of D_1 in D_3 and the entities of D_2 in D_4. The pub-sub sensor model along with data enrichers improves the scalability of the model. However, the problems of static naming and imperative dataflow definitions (latches) make the entities less reusable (the entities of D_1 are unusable in D_2).

Higher order functions with a dataflow model. Table VI shows the deployments in Ops-Scale. There are three functions defined per service operation in D_1 then all the functions are reused in D_2, D_3, and D_4. The enablers for this is the

Alias	Brooklyn

D_1	**entity-1**{
	...**effector** ops_s1 ...**sensor** s1-x.isUp
	}
	entity-2{
	...**latch** s1-x.isUp
	...**effector** ops_s2 **sensor** s2-x.isUp
	}
	entity-3{
	...**latch** s2-x.isUp
	...**effector** ops_s3
	}
	deploy {S1,S2,S3} on HG1

D_2	**entity-4**{
	...**effector** ops_s1 ...**sensor** S1.isUp
	}
	entity-5 {
	...**latch** S1.isUp ...**aggregate** S1
	...**effector** ops_s2 ...**sensor** S2.isUp
	}
	entity-6{
	...**latch** S2.isUp ...**aggregate** S2
	...**effector** ops_s3 }
	depoy {S4,S5,S6} on HG1

D_3	**deploy** entity-1 on HG1 **deploy** entity-2 on HG2 **deploy** entity-3 on HG3
D_4	**deploy** entity-4 on HG1 **deploy** entity-5 on HG2 **deploy** entity-6 on HG3

TABLE V: An *entity-based encapsulation* with a *pub-sub data sharing* model. We use **Apache Brooklyn** syntax in the pseudocode for programming the deployments of Table II. **effector:** implements an operation **sensor:** publishes a topic, **latch:** awaits the entity until a sensor value is ready, **aggregate:** collects values from all the publishers of a sensor.

higher order programmed data transformers, higher ordered dependent functions, and declarative naming. As can be seen, in D_2 we partially executed the fun_2 by assigning a new data transformer and the dependency of fun_3 to fun_2 is updated too. Similar updates are done in D_3 and D_4. As can be observed, the amount of extra programming that is needed is as little as updating the host and dependencies through higher-order programming and service functions are programmed only once.

Even though we show reusability for only a limited number of deployments at the level of service operations, the situation becomes much more complicated when a service provider has to deal with the explosion in permutations that come with more services each with a variety of placements.

B. Can Ops-Scale boost automation *of elasticity beyond existing solutions?*

In § V-A we showed that the higher order functions and dataflow variable model in Ops-Scale can boost declarativeness w.r.t. service placement and dependency cardinality. Now, we show how complex dependent services become at scale and how the declarative model of Ops-Scale can fully automate it.

Alias	Ops-Scale

D_1	**def** fun_1(host_fun=host) {
	...ops_s1
	}
	def fun_2(host_fun=host,
	...s1_tran=select, s1_fun=fun1) {
	...ops_s2
	}
	def fun_3(host_fun=host,
	...s2_tran=select, s2_fun=fun2) {
	...ops_s3
	}
	fun_3(host_fun=HG1)

D_2	fun_2_p=fun_2(s1_tran=collect) fun_3(host_fun=HG1, s2_tran=collect, ...s2_fun=fun_2_p)

D_3	f1=fun_1(host_fun=HG1)
	f2=fun_2(host_fun=HG2, s1_fun=f1)
	f3=fun_2(host_fun=HG3, s2_fun=f2)
	f3()

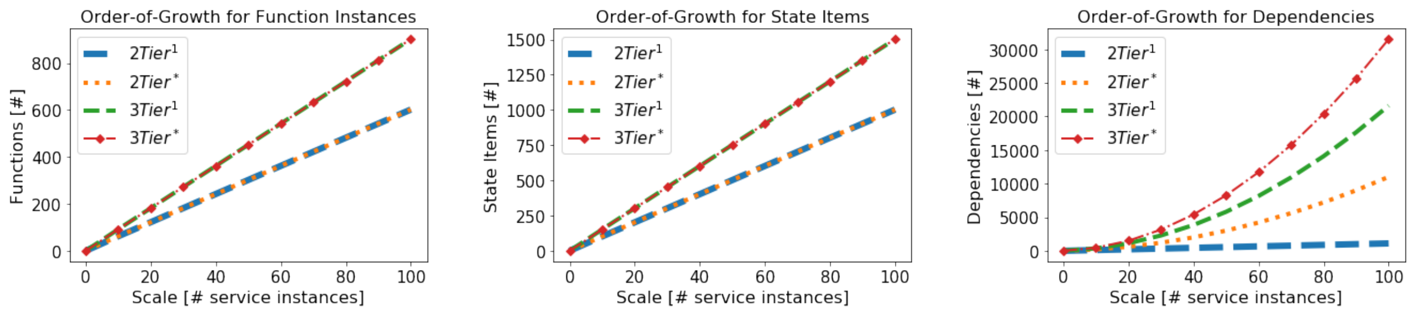
D_4	f1=fun_1(host_fun=HG1)
	f2=fun_2(host_fun=HG2, s1_fun=f1,
	...s1_tran=collect)
	f3=fun_2(host_fun=HG3, s2_fun=f2,
	...s2_tran=collect)
	f3()

TABLE VI: *Higher order functional* with a *dataflow variable* model. The pseudocode represents *Ops-Scale*'s programming for the deployments of Table II. We use partial function calls, which freezes some argument values until the final call made to functions.

Alias	Deployment Detail
$2Tier^1$	$\lambda S_1 \prec^1 \lambda S_2$ $\lvert S_1\rvert=\lvert S_2\rvert= m\, s.t.\, m \in [1, 100]$
$2Tier^*$	$\lambda S_1 \prec^* \lambda S_2 \prec^* \lambda S_3$ $\lvert S_1\rvert=\lvert S_2\rvert= m\, s.t.\, m \in [1, 100]$
$3Tier^1$	$\lambda S_1 \prec^1 \lambda S_2 \prec^1 \lambda S_3$ $\lvert S_1\rvert=\lvert S_2\rvert=\lvert S_3\rvert= m\, s.t.\, m \in [1, 100]$
$3Tier^*$	$\lambda S_1 \prec^* \lambda S_2 \prec^* \lambda S_3$ $\lvert S_1\rvert=\lvert S_2\rvert=\lvert S_3\rvert= m\, s.t.\, m \in [1, 100]$

TABLE VII: **Service autoscaling** examples. The deployments consists of two-tier applications $2Tier^1$and $2Tier^*$and three-tier applications $3Tier^1$and $3Tier^*$. In $2Tier^1$and $3Tier^1$service dependencies are one-to-one but $2Tier^*$and $3Tier^*$have all-to-all dependencies. We autoscale the tiers of each application together for simplicity, ending up having similar number of service instances across the tiers at each time. The number of service instances is varied between 1 to 100.

The Order-of-Growth of Cloud Operations. We measure the order-of-growth of operations as we autoscale the instances of services. For the experiment, we implement the 4 different cluster configurations shown in Table VII in Ops-Scale. The results are shown in Figure 2. As can be seen, the number of functions and state-items grow linearly w.r.t. the scale. As we



(a) Functions (b) States (c) Dependencies

Fig. 2: Order-of-Growth as the system scale changes from 1 to 100. The measurements are done by implementing the cluster configurations presented in Table VII. The implementations are done in Ops-Scale and the reconfigurations are done *declaratively* by Ops-Scale.

add more tiers to the application, the slope of the growth increases. The dependencies have, however, polynomial growth and the degree of the polynomial is varied w.r.t. scale, the number of tiers, and the cardinality of dependencies. The automatic key generation scheme enables Ops-Scale to capture the required steps declaratively and to perform the steps fully automatically.

VI. Discussion

We have shown that Ops-Scale can autoscale cloud operations efficiently in practice. However, there are still some questions requiring follow up research. In this section, we elaborate on these questions.

Can't cloud operations be performed without a formal model? The new generations of cloud computing applications such as IoT and 5G at edge data centers are much more complex and dynamic than the standard 3-tier cloud applications. Handling cloud operations for such applications without formal modeling does not have the level of agility in design and execution. Ops-Scale is a small step towards that goal.

What are the impacts of reconfigurations in Ops-Scale on the performance of production systems? We build our operation model on the assumptions that: systems under the operation are designed for horizontal auto-scaling, and the operations encapsulated by value functions can safely accommodate configuration changes. These two assumptions are challenging problems on their own but they are different from the problems we address in Ops-Scale, which are declarative designs and rapid executions.

Can Ops-Scale detect state inconsistencies, and, if yes, how soon after they occur? We mainly focused on deploying and auto-scaling deployments in this paper. However, as we mentioned in § IV, the value function can produce other actions such as CHECK for health-check. The abstractions, semantics, and procedures for incorporating new actions in the controller are the same as for the presented actions. The frequent launching of new DAGs for customized scenarios is easily implemented in the current controller of Ops-Scale.

VII. Related Wrok

Solutions for the modeling and orchestration of cloud operations span from low-level scripting CM systems to high-level management techniques. We classify them by their orchestration techniques while considering their models, level of declarativeness, and applicability for elasticity.

Orchestration based on *Topology* or *Desired State* Models. A group based on TOSCA, a cloud-portable provisioning specification[11], [6], rely on separate models for application structure, through topology model, and provisioning plan[29], [13]. Although the topology model is declarative, components and their relationships are typed and deployment specific. [27] has a search based approach for finding actions and it runs actions without orchestration plan. Binding provisioning plan to topology model makes the orchestration less elastic, whereas our proposal is to internalize elasticity in fine-grained cloud operations. Another group [17], [24], [14] uses state into which an application shall be transferred: an orchestration plan consists of relationships between provisioning operations and relationships are generated based on the desired state. AI planning and graph covering techniques are used to analyze dependencies between nodes, relationships, and operations in order to generate workflows. CFEngine [14] has a behavioral model for cloud resources based on promise theory [9]. In Ops-Scale, both topology and desired state are the implicit results of value functions; at the functional level, data is untyped and unstructured—giving a better degree of freedom to functions to encapsulate elastic operations.

Orchestration for *Service Oriented Architecture*. These often have a coarse-grained operation model (at service level) - they rely on server-based data sharing between operations (name servers and distributed data stores), and they use a PaaS cloud model [37], [23]. DisNix [37] uses the purely functional model and the immutable key-value system from [16] but service dependencies must be specifically defined per deployment. Orchestration for microservices is limited to launching all instances [34], [2] or launching before rules [4].

Script Centric DevOps—*Manual Orchestration*. They support CM at the very deep technical level but they are single host IT automation for continues-code-upgrade; they have no support for deployment orchestration for distributed systems [26], [35], [22].

Orchestration for *Distributed DevOps*. Ansible [30] and SaltStack [5] have deployment dependent and imperative orchestration, and they use a data store for data sharing. Brooklyn's [1] model is based on promise theory; orchestration plans are generated from sensor-effector dependencies between deployment entities, and data sharing has a publish-subscribe model. Although flexible, typed entities and static naming are anti-elastic patterns.

VIII. Conclusions

In this paper, we have introduced the concept of declarative functional modeling with higher-order programming for cloud operations, which allows flexible cluster design and automated adaptation of cluster configurations for horizontal auto-scaling. We realized the implementation of the formal model in a feedback loop architecture with a DAG traversal control logic. Parallel actuation along with dataflow variable binding brings a high level of agility for executing deployment plans. To demonstrate the concept, we implemented a system called Karamel which is a cluster provisioner on public IaaS cloud offerings (AWS, GCE, etc.). Karamel implements a functional approach on pre-defined cloud operations implemented in Chef. In our comparison, we demonstrated that cluster programming in Ops-Scale is brief and precise compared to the state of the art DevOps systems. Moreover, Ops-Scale can automate linear state growth and polynomial dependency growth in deployments with the formal model. In the future, we plan to add functional syntax to Karamel, and to include more actions to achieve a complete functional engine for a broader class of cloud operations.

References

[1] Apache Brooklyn v0.12.0. https://goo.gl/1Nrf9Q, [01-03-2019].

[2] Docker v17.09. https://goo.gl/sKQKyQ, [01-03-2019].

[3] Kuberenetes v1.9. https://goo.gl/EX8MrB, [01-03-2019].

[4] Mesosphere Marathon v1.5.0. https://goo.gl/8aDmKM, [01-03-2019].

[5] SaltStack v.2017.7.2. https://goo.gl/sCXoX5, [01-03-2019].

[6] Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0. https://goo.gl/8m5kQK, [01-03-2019].

[7] Amazon Web Services, Inc. Amazon Elastic Computing Cloud. https://aws.amazon.com/ec2/, 2017.

[8] A. Barker, B. Varghese, J. S. Ward, and I. Sommerville. Academic cloud computing research: Five pitfalls and five opportunities. In *HotCloud*, 2014.

[9] J. A. Bergstra and M. Burgess. *Promise Theory: Principles and Applications*. XtAxis Press, 2014.

[10] A. Bessani, J. Brandt, M. Bux, V. Cogo, L. Dimitrova, J. Dowling, A. Gholami, K. Hakimzadeh, M. Hummel, M. Ismail, et al. Biobankcloud: a platform for the secure storage, sharing, and processing of large biomedical data sets. In *the First International Workshop on Data Management and Analytics for Medicine and Healthcare (DMAH 2015)*, 2015.

[11] T. Binz, G. Breiter, F. Leyman, and T. Spatzier. Portable cloud services using tosca. *IEEE Internet Computing*, 16(3):80–85, 2012.

[12] C. Boettiger. An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, 2015.

[13] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, and J. Wettinger. Combining declarative and imperative cloud application provisioning based on tosca. In *2014 IEEE International Conference on Cloud Engineering*, pages 87–96. IEEE, 2014.

[14] M. Burgess et al. Cfengine: a site configuration engine. In *in USENIX Computing systems, Vol*. Citeseer, 1995.

[15] M. Bux, J. Brandt, C. Lipka, K. Hakimzadeh, J. Dowling, and U. Leser. Saasfee: scalable scientific workflow execution engine. *Proceedings of the VLDB Endowment*, 8(12):1892–1895, 2015.

[16] E. Dolstra, A. LÖh, and N. Pierron. Nixos: A purely functional linux distribution. *Journal of Functional Programming*, 20(5-6):577–615, 2010.

[17] K. El Maghraoui, A. Meghranjani, T. Eilam, M. Kalantar, and A. V. Konstantinou. Model driven provisioning: Bridging the gap between declarative object models and procedural provisioning tools. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 404–423. Springer, 2006.

[18] Google. Google Compute Engine. https://cloud.google.com/compute/, 2017.

[19] K. Hakimzadeh, H. P. Sajjad, and J. Dowling. Scaling hdfs with a strongly consistent relational model for metadata. In *Distributed Applications and Interoperable Systems*, pages 38–51. Springer, 2014.

[20] Kamal Hakimzadeh. One Click Installation for Clusters. http://www.karamel.io/, 2017.

[21] Kamal Hakimzadeh. Reproducing Distributed Systems on Cloud. https://github.com/karamelchef/karamel, 2017.

[22] L. Kanies. Puppet: Next-generation configuration management. *; login:: the magazine of USENIX & SAGE*, 31(1):19–25, 2006.

[23] L. Leite, C. E. Moreira, D. Cordeiro, M. A. Gerosa, and F. Kon. Deploying large-scale service compositions on the cloud with the choreos enactment engine. In *Network Computing and Applications (NCA), 2014 IEEE 13th International Symposium on*, pages 121–128. IEEE, 2014.

[24] K. Levanti and A. Ranganathan. Planning-based configuration and management of distributed systems. In *2009 IFIP/IEEE International Symposium on Integrated Network Management*, pages 65–72. IEEE, 2009.

[25] T. Lorido-Botrán, J. Miguel-Alonso, and J. A. Lozano. Auto-scaling techniques for elastic applications in cloud environments. *Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09*, 12:2012, 2012.

[26] C. Ltd. Juju, 2017. https://jujucharms.com, [01-03-19].

[27] H. Lu, M. Shtern, B. Simmons, M. Smit, and M. Litoiu. Pattern-based deployment service for next generation clouds. In *2013 IEEE Ninth World Congress on Services*, pages 464–471. IEEE, 2013.

[28] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[29] R. Mietzner. A method and implementation to define and provision variable composite applications, and its usage in cloud computing. 2010.

[30] M. Mohaan and R. Raithatha. *Learning Ansible*. Packt Publishing Ltd, 2014.

[31] S. Niazi, M. Ismail, S. Grohsschmiedt, M. Ronström, S. Haridi, and J. Dowling. Hopsfs: Scaling hierarchical file system metadata using newsql databases. In *FAST*, pages 89–103, 2017.

[32] OpenStack Org. OpenStack Compute (Nova). https://github.com/openstack/nova, 2017.

[33] S. Perera, A. Perera, and K. Hakimzadeh. Reproducible experiments for comparing apache flink and apache spark on public clouds. *arXiv preprint arXiv:1693949*, 2016.

[34] D. K. Rensin. *Kubernetes - Scheduling the Future at Cloud Scale*. 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015.

[35] N. Sabharwal and M. Wadhwa. *Automation through Chef Opscode: a hands-on approach to Chef*. Apress, 2014.

[36] A. Ullah, J. Li, Y. Shen, and A. Hussain. A control theoretical view of cloud elasticity: taxonomy, survey and challenges. *Cluster Computing*, pages 1–30, 2018.

[37] S. Van Der Burg and E. Dolstra. Automated deployment of a heterogeneous service-oriented system. In *Software Engineering and Advanced Applications (SEAA), 2010 36th EUROMICRO Conference on*, pages 183–190. IEEE, 2010.

[38] P. Van-Roy and S. Haridi. *Concepts, techniques, and models of computer programming*. MIT press, 2004.

[39] D. Weerasiri, M. C. Barukh, B. Benatallah, Q. Z. Sheng, and R. Ranjan. A taxonomy and survey of cloud resource orchestration techniques. *ACM Computing Surveys (CSUR)*, 50(2):26, 2017.