# Karamel: A System for Timely Provisioning Large-Scale Software Across IaaS Clouds

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*—Cloud-native systems and application software platforms are becoming increasingly complex, and, ideally, they are expected to be quick to launch, elastic, portable across different cloud environments and easily managed. However, as cloud applications increase in complexity, so do the resultant challenges in configuring such applications, and orchestrating the deployment of their constituent services on potentially different cloud operating systems and environments.**

**This paper presents a new orchestration system called Karamel that addresses these challenges by providing a cloud-independent orchestration service for deploying and configuring cloud applications and platforms across different environments. In Karamel, we model configuration routines with their dependencies in composable modules, and we achieve a high level of configuration/deployment parallelism by using techniques such as DAG traversal control logic, dataflow variable binding, and parallel actuation. In Karamel, complex distributed deployments are specified declaratively in a compact YAML syntax, and cluster definitions can be validated using an external artifact repository (GitHub).**

***Index Terms*—cloud; software deployment; orchestration;**

## I. Introduction

The migration of applications and platforms to the Cloud, has increased the complexity of their software deployment processes to include infrastructure provisioning, orchestrated configuration and deployment [20], and support for runtime configuration changes to services.

Current solutions in the DevOps paradigm suffer from limited models, with container management systems, such as Kubernetes [17] and Docker-Swarm [3], using static container images that can be configured using environment variables. Static models enforce the configuration of software and launching of services to happen at the same time. In contrast, legacy configuration management (CM) systems, such as Chef [2], Ansible [1], and Puppet [4], are host-centric, limiting the support for configuration steps that bounce between hosts ( orchestration steps) before services are launched.

To this end, we present our cloud-independent software orchestration system for provisioning distributed systems, Karamel, that follows the classical *feedback control loop* mechanism for provisioning the software (see Figure 1). The controller generates an orchestration plan per deployment and it executes the plan by using *parallel actuation* of the target system on many machines residing on different cloud platforms. According to the feedback it receives from orchestration/configuration steps, it either proceeds with the plan or it only repeats the tasks that have failed – without restarting the plan from the beginning.

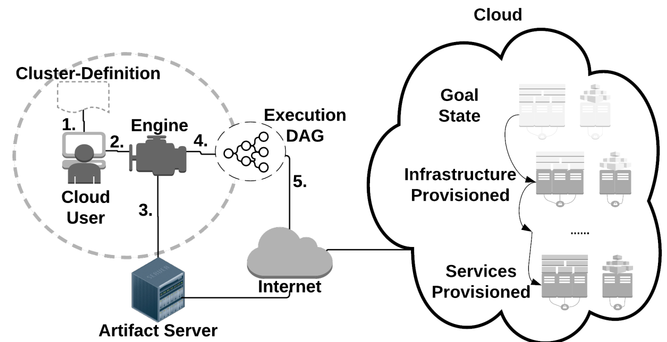


Fig. 1: Interaction of a cloud user with an IaaS cloud through Karamel as the engine transforms a cluster-definition (CD) to the desired cluster state by (1) The user provides a CD; (2) the engine extracts the list of provisioning modules from the CD, (3) the engine downloads the metadata of the modules, (4) the engine generates an execution DAG and runs it, (5) along the execution, many feedback loops are generated between target systems and the engine; the engine handles failures and signals the plan further inside the loops.

We model a configuration routine as a *composable module* such that the dependencies of the module to other modules are also embedded inside the definition of the module. As such, we design modules only ***once*** but we reuse them *many times* in deployments that have different orchestration plans. At runtime, a module is replicated in many similar actuation tasks and the actuator runs each task on a remote host. The controller uses *dataflow* variable binding - making downstream tasks wait for their required input data items to be built by upstream tasks.

We implement a domain specific language (DSL) for expressing software deployments (cluster definitions). Leveraging on the properties provided by our composable design, our DSL is *expressive* but *concise*. The DSL allows fine-grained parameter passing to the modules, letting users customize the cluster definitions. Parameters defined in cluster definitions are

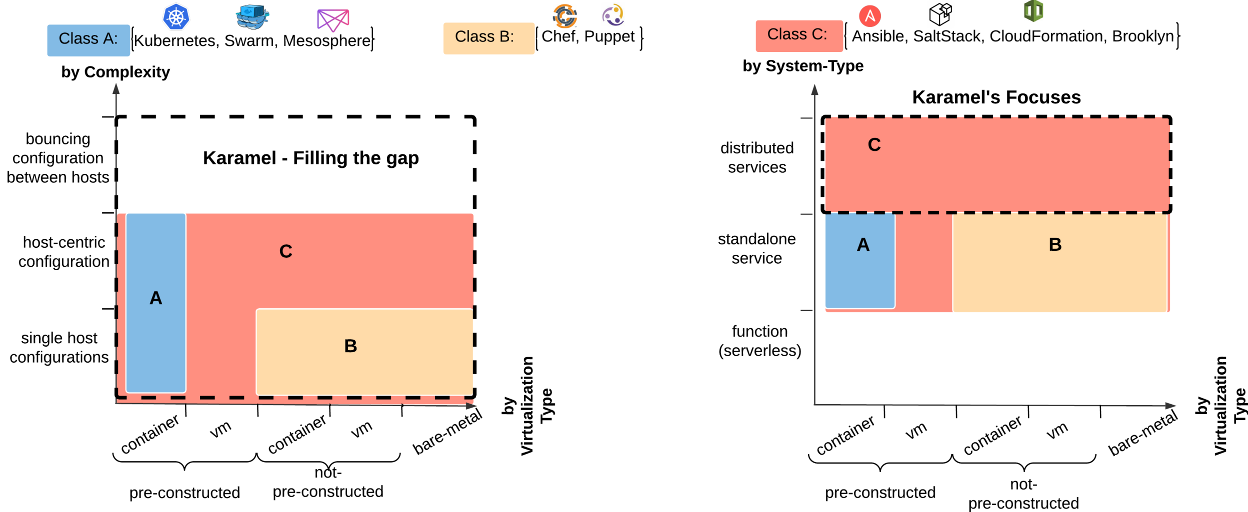


Fig. 2: Visualization of different system classes by the complexity of configurations. Karamel supports distributed systems that require more inter-component configuration and it introduces advanced configuration patterns that are not bound to hosts.

validated against their interface definition, typically hosted in a GitHub repository, and specified using Chef [2], a popular configuration management platform. That is, we reuse for software configuration, while Karamel provides orchestration capabilities.

The contributions of this work are:

**(C1)** An orchestration system for deploying software systems on cloud platforms at scale. Deployments in Karamel are portable between different cloud providers (it currently supports Amazon [6], Google Cloud [10], Open-Stack [15], and bare-metal servers) and virtualization technologies [16]. The system is open source [13], [12] and it is used by organizations from academia and industry [7], [9], [11], [14].

**(C2)** A composable model for encapsulating orchestration routines along with their dependencies inside them. The model allows the declarative design of many layers of software with complex orchestration dependencies at scale.

**(C3)** A level of parallelism higher than existing solutions. Karamel achieves this by its systematic solution for deployments based on a DAG traversal control logic. Our task abstraction allows us to support heterogeneous types of orchestration routines (e.g., forking machines, installing software, etc.) in a single coherent plan. Further, parallel actuation and dataflow variable binding increase support for concurrency.

## II. Motivation and Related Work

We present related work in the DevOps paradigm to motivate Karamel. Firstly, we group systems into three classes: A, B, and C. We synthesize this information by analyzing the systems' documentation and, where necessary, their source code. Class A follows the microservice architecture [18] using container images for fast launch. Class B supports deployments on many hosts but they can only configure hosts individually (host-centric). Class C supports orchestration for deploying distributed systems but they are limited in supporting advanced configuration patterns.

**Configuration Complexity.** In Karamel, we push these boundaries further. As Figure 2 shows, we target distributed services, like class C (the stronger class), but we want to support advanced configuration patterns. The advanced patterns are not limited to a single host, like in class B, and they are not host-centric, like in class A and C. Besides, the model is portable and it supports the combinations of bare-metal servers, virtual technologies, and virtual machine images.

| Classes ↓ | Features →<br>Levels →<br>DevOps ↓ | **Encapsulation (definition)**<br>1-Predefined Commands<br>2-Predefined Functions<br>3-Custom Scripts<br>4-Custom Functions | **Composition (at usage)**<br>0-None<br>1-Imperative<br>2-Declarative Local Dep.<br>3-Declarative Global Dep. |
|---|---|---|---|
| A | **Kubernetes** | 1 | 0 |
| | **Swarm** | 1 | 0 |
| | **Marathon** | 1 | 3 |
| B | **Chef** | 3 | 2 |
| | **Puppet** | 3 | 2 |
| C | **Ansible** | 3 | 1 |
| | **CloudFormation** | 3 | 1 |
| | **SaltStack** | 4 | 2 |
| | **Brooklyn** | 4 | 1 |
| | **Karamel** | 4 | 3 |

TABLE I: Modular design of provisioning routines. 1-4 denote *'simple to advanced'* and 0 denotes *'not supported'*.

**Composable Modules.** Our modular design brings benefits such as logical separation of configuration boundaries, independent development, and building large-scale and robust configuration management (CM) systems from smaller and more stable modules. Table I shows encapsulation methods used by systems versus their flexibility of composing the modules in different deployments.

Containers are bounded to pre-defined commands (e.g., start, stop, etc.) - only Marathon supports distributed dependencies but they have to be defined imperatively and per deployment. Chef and Puppet rely on scripting languages by making dependencies declarative inside modules: they are

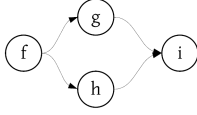


Fig. 3: An example of Logical DAG that is generated from the dependencies defined between functions $f$, $g$, $h$, and $i$ .

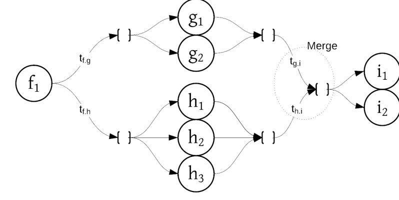


Fig. 4: An execution DAG for an application of the logical DAG in Figure 3. In this example, the target-groups of the functions $f$, $g$, $h$, and $i$ have respectively 1, 2, 3, and 2 hosts.

highly composable but they are single host configuration systems. Class C support customized functions for encapsulation but they suffer from defining compositions imperatively. In Karamel, we seek to achieve high flexibility for composing pre-defined modules but without repeating the definitions of the modules and their dependencies for each deployment.

| Classes ↓ | Features → | Parallel Paradigm | Execution Plan | Data Sharing |
|---|---|---|---|---|
| | Levels →<br>DevOps ↓ | 0-None<br>1-Parallel Launch Rep. Apps<br>2-Parallel Run Rep. Tasks<br>3-Task Parallel<br>4-Data Parallel | 0-None<br>1-Imperative<br>2-Declarative | 1-Host Limited<br>2-Global Store<br>3-Pull (from Host)<br>4-Push (to Consumer) |
| A | **Kubernetes** | 1 | 0 | 2 |
| | **Swarm** | 1 | 0 | 2 |
| | **Marathon** | 1 | 1 | 2 |
| B | **Chef** | 0 | 1 | 1 |
| | **Puppet** | 0 | 1 | 1 |
| C | **Ansible** | 2 | 1 | 3 |
| | **CloudFormation** | 0 | 1 | 2 |
| | **SaltStack** | 3 | 1 | 2 |
| | **Brooklyn** | 4 | 2 | 4 |
| | **Karamel** | 4 | 2 | 4 |

TABLE II: Shortcomings of systems in maximizing parallel execution. 1-4: *'simple to advanced'* and 0: *'not supported'*.

**Parallel Execution.** As quick launching is very crucial in large-scale deployments, we investigate the pros and cons of the supported execution models from a parallelism standpoint. As Table II presents, class B are task-serial as they run their recipes in an imperatively defined order. Class A uses parallel replication of identical services as parallelism but cross different applications commands run serially. Marathon repeats the definition of execution plans per deployment. Ansible can run replications of the same tasks in parallel and it allows each host to proceed without synchronizing hosts for similar tasks. However, Ansible does not have parallelism between target hosts of different tasks. SaltStack has higher task parallelism compared to Ansible, as it handles task dependencies between all target-groups. Brooklyn only supports data parallel model with its sensor-effector model. Inspired by learning from the limitations, we choose to capture execution plans in declarative fashion as we select data parallel execution by dataflow variable binding model to maximize the parallelism.

## III. MODEL

**State.** A collection of key-value pairs (state-items) form the final state for deployment. Provisioning modules produce State-items. For instance: machines, containers, storage, files, and services are typical state-items.

**Functions.** Provisioning modules *behave like functions* in Karamel (observational view over functions as in [19]). A function is responsible for building, controlling, and roll-backing a subset of the state. The input arguments of functions are a mix of definition-items and state-items. The function's body transforms definition-items into new state-items. The results of the functions are new state-items. A function is targeted to run at a *target-group* of hosts. We apply each function once at each host in a target group. In our model, the functions are activated when all their input arguments (definition-items and state-items) are available, similar to dataflow variables from concurrent programming [19]. The production and consumption of state-items define a partial order ($\preceq$) over the modules.

$$f.results \cap g.arguments \neq \emptyset \iff f \preceq g$$

**DAG** Given a set of functions, a logical acyclic graph (LDAG) is constructed based on the partial-order relationship between the functions. For example, for the function set $F = \{f, g, h, i\}$ with the partial-order set $R_{\preceq} = \{f \preceq g, f \preceq h, g \preceq i, h \preceq i\}$ the LDAG in Figure 3 is drawn. An application of an LDAG is an execution DAG (EDAG). We form an execution DAG as we apply all the functions inside the LDAG at their corresponding target-groups (Figure 4) and transform the data between them.

## IV. KARAMEL - SYSTEM COMPONENTS

In this section, we give an overview of the main components of the system and its algorithms.

**Cluster Definition DSL.** Figure 5 shows an example of the DSL in YAML [5]. The DSL can be simply interpreted as a list of functions (e.g., $hadoop :: namenode$) and definition-items. Algorithm 1 summarizes how we build $LDAG$ and $EDAG$ give an cluster definition (CD). First, it parses the $CD$ and loads set of referenced functions $F$ with their definition items $DI$. Then, it loads the metadata $meta$ and the $body$ of functions from our function repository. Next, it loads direct and transitive dependencies from loaded $meta$ in a loop. Finally, it builds $LDAG$ by using dependencies, and it instantiates an $EDAG$ by traversing the $LDAG$ from the root and applying definition items to the functions.

**Orchestration Controller.** Our engine orchestrates deployments by using a control loop mechanism. As Algorithm 2 shows, a control loop starts as a new $EDAG$ given. The controller, always, submits ready tasks to our actuator, then, it waits until feedback arrives from the actuator. If the received task $t$ was successful, the controller collects the set of state-items $SI$ that are built by task $t$. The controller submits all the successors of the task $t$ for actuation as it binds $SI$ to the tasks variables. Otherwise, if the task was unsuccessful,

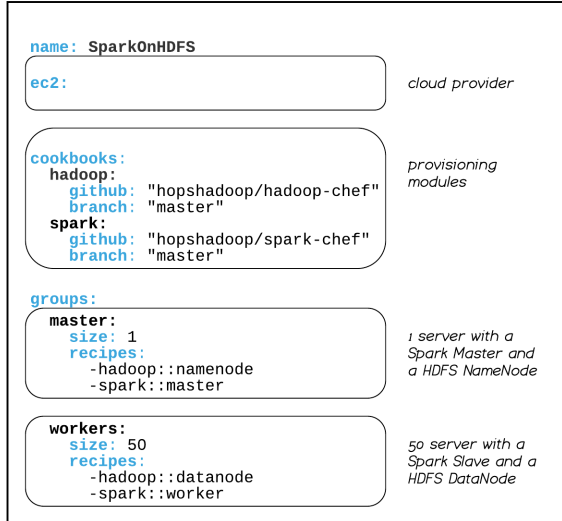


Fig. 5: Minimal Cluster Definition for deploying Apache-Spark [21] and HDFS [8] on Amazon EC2.

**Algorithm 1:** $dag_builder(CD)$

1 $F, DI \leftarrow parse(CD)$
2 $D = \langle meta, body \rangle_{i=1}^{k} \leftarrow load_detail(F)$
3 **while** $\exists f \in F | f.meta.dep \ni F$ **do**
4 $\quad F = F \cup f.meta.dep$
5 $\quad D = D \cup load_detail(f)$
**end**
6 $LDAG = build_ldag(D)$
7 $EDAG = apply(LDAG, DI)$

the controller allows human intervention by escalating the error to our dashboard. When a failed task was fixed, either automatically or by a human, then the controller signals the DAG and continues the deployment.

**Orchestration Actuator.** Algorithm 3 shows the routine of the actuator. The actuator is launched for provisioning software on a set of hosts $H$ (for simplicity, we assume that hosts are provisioned for only show the routine). The actuator creates a set of task queues $Q$ (one queue per host) and a set of executor threads $R$ (one dedicated thread per host) for parallel execution. The actuator waits until it receives a new task from the controller: it places the new task at the tail of the queue. A task runner $r$ removes a task from its queue, or it waits for a new task to arrive if the queue is empty, and it runs it. The serial ordering of tasks at the host level is to avoid potential conflicts of software packages as some packaging systems like *apt* take global (host-level) locks.

Karamel uses agent-less communication over SSH for simplicity of the usage (details are skipped in Algorithm 3 for short). The task runner uses a stubborn retry mechanism when a failure occurs; that is, it retries the execution a number of times $RE$ with a back-off period $BP$ between each attempt.

**Algorithm 2:** $control_loop(EDAG)$

1 $T = EDAG.roots$
2 **for** $t \in T$ **do**
3 $\quad callback(t, this)$
4 $\quad submit_to_actuator(t)$
**end**
5 **repeat**
6 $\quad t \leftarrow wait_for_callback()$
7 $\quad$ **if** $t.succeed$ **then**
8 $\quad\quad SI \leftarrow t.state_items$
9 $\quad\quad T \leftarrow t.successors$
10 $\quad\quad$ **for** $s \in T$ **do**
11 $\quad\quad\quad callback(s, SI, this)$
12 $\quad\quad\quad submit_to_actuator(t)$
$\quad\quad$ **end**
$\quad$ **end**
$\quad$ **else**
13 $\quad\quad submit_for_debugging(t)$
$\quad$ **end**
**until** $EDAG.isdone$

**Algorithm 3:** $actuator(H, RE, BP, \alpha)$

1 $Q \leftarrow allocate_Q_per_host(H)$
2 $R \leftarrow allocate_runner_per_host(H)$
3 $\forall r \in R | r.start()$

\# Main Thread Runs:
4 **repeat**
5 $\quad t \leftarrow await_controller_submission()$
6 $\quad Q[t.host].enq(t)$
**until** $TRUE$

\# Runner Thread $r \in R$ Runs:
7 **repeat**
8 $\quad t \leftarrow wait_dq(Q[r.host])$
9 $\quad run_over_ssh_stubborn(t, RE, BP, \alpha)$
10 $\quad call_controller(t)$
**until** $TRUE$

The back-off period is exponentially increased by a factor $\alpha$. The stubborn mechanism is designed to overcome temporary faults, due to networking or external services, such as the artifact repository.

## V. Evaluation

First, we present some statistics related to some real deployments done by Karamel, then we present results of an experiment measuring the latency of deployments.

**Statistics.** We present some statistics that we have collected over the course of a year in 2015-2016 related to the provisioning of 1660 clusters at a variable scale between 1-100 machines. The data shows that 47 different users used Karamel from 15 countries and 33 cities. Figure 6a shows that more

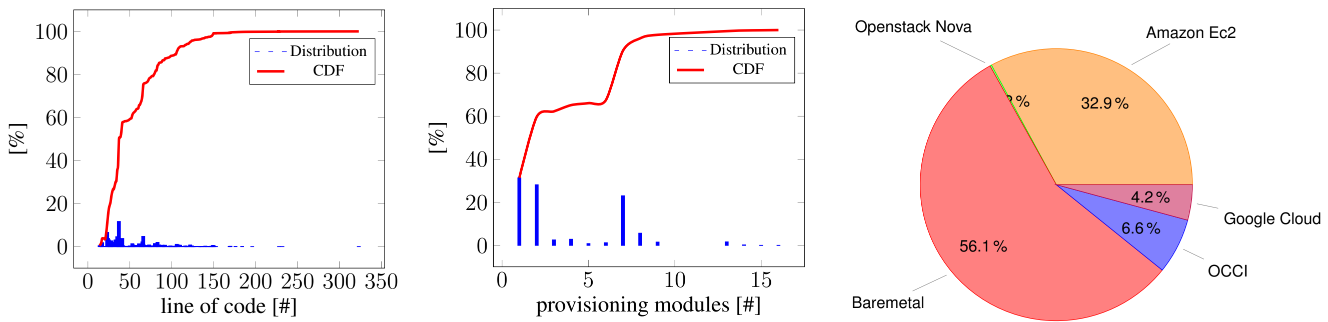


Fig. 6: Statistics collected from real-world clusters run in Karamel over the course of one year in 2016. It shows distributions of: (i) lines of code (LOC), (ii) number of provisioning modules, and (iii) cloud provider used - over 1660 launched clusters.

than 88% of the clusters are defined in less than 100 lines of YAML code whereas the referenced provisioning modules are coded in more than 500 lines in Ruby. Totally 68 modules are referenced in the definitions and maximum 16 modules are used per cluster: 59% of the clusters only use 1 or 2 modules, almost 23% of them use exactly 7 modules (Figure 6b).

Figure 6c shows that clusters are deployed portably on supported clouds: 56.1% are deployed on in-house premises, 39.2% on Amazon, 6.6% on OCCI, and 4.2% on Google.

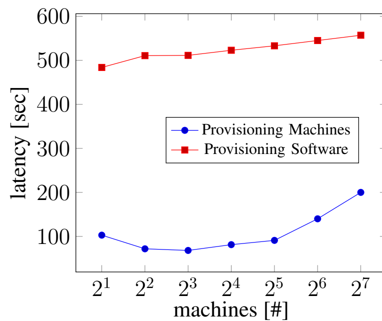


Fig. 7: Provisioning Latency at Scale.

**Latency at Scale.** Moreover, we perform an experiment to deploy the stack of Spark on HDFS in Figure 5 on Amazon Cloud. In each run, we change the number of machines by the factor of 2 and we measure the latency of deployment. Figure 7 shows that latency in the provisioning of infrastructure fluctuates slightly which is related to the changing delays incurred from Amazon. However, the latency for the provisioning of software starts from 488 seconds for 2 machines and it increases to 522 for 16 machines. The slight increase is due to the fact that some machines finish some jobs slower than others and they increase the total time of the provisioning plan at larger scales.

## VI. Conclusion

In this paper, we explained that the orchestration of modern systems should be time-efficient and repeatable as it deals with complex configurations scenarios. We showed that existing systems have limited models for the required agility. We demonstrated that Karamel overcomes the issues by using designing orchestration as composable modules and employing data parallelism for actuation. Karamel is available as an open source project and is used in industry and academia.

## References

[1] Ansible v2.4. https://goo.gl/gYPjkW, [01-03-2019].
[2] Chef Client v12.0. https://goo.gl/ai6Za8, [01-03-2019].
[3] Docker v17.09. https://goo.gl/sKQKyQ, [01-03-2019].
[4] Puppet 5.3. https://goo.gl/HQdkRG, [01-03-2019].
[5] YAML Data Serialization Standard. http://yaml.org
[6] Amazon Web Services, Inc. Amazon Elastic Computing Cloud. https://aws.amazon.com/ec2/, 2017.
[7] 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.
[8] D. Borthakur. Hdfs architecture guide. *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*, page 39, 2008.
[9] 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.
[10] Google. Google Compute Engine. https://cloud.google.com/compute/, 2017.
[11] K. Hakimzadeh, H. P. Sajjad, and J. Dowling. Scaling hdfs with a strongly consistent relational model for metadata. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 38–51. Springer, 2014.
[12] Kamal Hakimzadeh. One Click Installation for Clusters. http://www.karamel.io/, 2017.
[13] Kamal Hakimzadeh. Reproducing Distributed Systems on Cloud. https://github.com/karamelchef/karamel, 2017.
[14] 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.
[15] OpenStack Org. OpenStack Compute (Nova). https://github.com/openstack/nova, 2017.
[16] H. Peiro Sajjad, K. Hakimzadeh, and S. Perera. Reproducible distributed clusters with mutable containers: To minimize cost and provisioning time. In *Proceedings of the 2017 workshop on hot topics in container networking and networked systems*. ACM, 2017.
[17] D. K. Rensin. *Kubernetes - Scheduling the Future at Cloud Scale*. 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015.
[18] J. Thönes. Microservices. *IEEE Software*, 32(1):116–116, 2015.
[19] P. Van-Roy and S. Haridi. *Concepts, techniques, and models of computer programming*. MIT press, 2004.
[20] 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.
[21] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. *HotCloud*, 10:10–10, 2010.