Fast Choreography of Cross-DevOps Reconfiguration with Ballet: A Multi-Site OpenStack Case Study

Jolan Philippe[1], Antoine Omond[1,2], Hélène Coullon[1], Charles Prud'Homme[1], Issam Raïs[2]
[1]IMT Atlantique, Nantes Université, Ecole Centrale Nantes, CNRS, INRIA, LS2N, UMR 6004, F-44000 Nantes, France
{firstname.lastname}@imt-atlantique.fr
[2]Department of Computer Science, UiT The Arctic University of Norway, Tromsø, Norway
{firstname.lastname}@uit.no

***Abstract*—In the context of Edge Computing or Cyber-Physical Systems, cross-functional, and cross-geographical DevOps teams are in charge of automating deployments, configuration, and management (*i.e.*, reconfiguration) of complex, large-scale, highly dynamic, and geo-distributed service-oriented software systems. In this context, DevOps teams cannot reasonably manually coordinate their reconfiguration operations in a global manner. Furthermore, as disconnection is the norm in these paradigms, a central entity responsible for reconfiguration should be avoided, and the set of changes to apply should be as fast as possible. This paper presents Ballet, a fast tool to automate decentralized choreographies (*i.e.*, coordination) of cross-DevOps reconfiguration. We show a gain of 42.6% for a deployment scenario and 24% for an update scenario on an OpenStack case study.**

***Index Terms*—DevOps, Infrastructure-as-Code, Decentralized Reconfiguration, Constraint Programming**

I. Introduction

In recent years, mainly because of the advent of distributed paradigms such as Cloud computing, service-oriented (SO) software architectures have become the norm and have evolved towards microservices applications and systems, sometimes made of thousands of small components. At the same time, the DevOps profession has gradually taken shape. The DevOps concept fills the gap between development and operation concerns, which often conflict. One targets very fast integration of new software features, while the other targets reliability and stability. DevOps are in practice responsible for automating and accelerating every possible procedure between the development and the operation. One important aspect of this job is the process of deploying and configuring (initial commissioning), and then reconfiguring (updating, managing) long-running services while avoiding their interruptions, which is made easier through Infrastructure-as-Code (IaC) techniques. In IaC, procedures are defined as well-structured and easy-to-read codes. In particular, declarative approaches have become widely used for their interesting abstraction level: DevOps users specify what is required, not the imperative program (or plan) to get it. This program is instead automatically inferred by a *planner*, which avoids errors.

With the growing complexity and scale of SO distributed software systems, DevOps teams have also evolved into collaborating cross-functional teams, denoted cross-DevOps in this paper. Furthermore, in massively geo-distributed infrastructures (*e.g.*, Edge Computing and Cyber-Physical Systems paradigms), centralizing information (*e.g.*, the state of the system) of all DevOps teams is not possible due to the scale, geo-distribution, and dynamic nature of systems and applications. Moreover, in this environment network disconnection has become the norm [1] which has led to avoiding a central entity responsible for coordinating deployments and reconfigurations. Hence, in this context, cross-geographical and cross-functional DevOps teams are naturally formed. In this context, a DevOps team is responsible for the deployment and reconfiguration of its own services [2], [3]. Most of the existing DevOps declarative tools are designed in a centralized manner: (1) a unique declarative file has to be specified, thus requiring a global knowledge of all services; (2) the inference of the reconfiguration plan and its execution are handled by a central entity that must store the state of all services and that must be accessible by all services. In practice, instead of having a single declarative file and a central tool, cross-DevOps teams use local instances of IaC tools that cannot collaborate (as being designed in a centralized fashion).

However, a change in one service often depends on the availability of other services. Hence, if dependencies span across DevOps teams, their reconfiguration has to be coordinated to ensure the global correctness of the procedure. In practice, this coordination is often done manually (*i.e.*, phone, email) [4]. We may refer in this paper to *decentralized reconfiguration*, or to the *choreography* of cross-DevOps operations [5]–[7].

On the one hand, offering a decentralized declarative IaC tool is important to reduce errors and manual coordination between DevOps teams. As far as we know (see Section VI) Muse [4] (implemented on top of Pulumi [8]) is the only tool that automates the decentralization of provisioning. With Muse, each DevOps team can push cre-

ation, destruction, or update operations on their services. The required actions for other DevOps teams are then automatically inferred in a decentralized manner through automated coordination.

On the other hand, reducing the duration of the reconfiguration in massively geo-distributed infrastructures is a crucial issue that needs to be addressed. The dynamic nature of both the applications and the underlying infrastructures (i.e., mobility, frequent disconnections, new nodes, etc.) necessitates fast deployments and reconfiguration of hosted services. Concerto [9], [10] and its declarative tool [11] is, as far as we know, the tool that offers the best execution time for reconfiguration operations in the literature thanks to parallelism and asynchrony in its definition of services' life cycles and its reconfiguration language.

In this paper, we present a new decentralized declarative reconfiguration tool called *Ballet* which combines the ideas of both Muse and Concerto to improve the following aspects: offering more flexibility in cross-DevOps operations (*i.e.*, not limited to `deploy`, `update`, and `destroy` as in Muse); reducing the execution time of cross-DevOps operations by leveraging parallelism and asynchrony. The contributions of Ballet are a new simple declarative goal language for DevOps; a decentralized version of Concerto; a decentralized planner on top of Concerto; and an evaluation based on a real case study.

In particular, by using the motivating case study of a multi-site OpenStack (see Section II), we seek to answer the following research questions:

- RQ1: Are we able to generate an efficient distributed plan in a decentralized manner for cross-DevOps operations while keeping a strictly local information level?
- RQ2: Can we reduce the execution time of DevOps operations compared to Muse [4] without adding complexity to the input provided by DevOps?

The rest of this paper is organized as follows. Section II presents our motivating use case that will be used throughout the paper. Section III presents the usage of Ballet from developers and DevOps perspectives. In Section IV, we present Ballet's engine to infer reconfiguration plans and execute them in a decentralized manner. Section V evaluates Ballet in comparison with Muse [4], and Section VI studies the related work. Finally, Section VII concludes this work and opens to some perspectives.

II. Motivating Use-Case

As a motivating example, we use a decentralized version of OpenStack on multiple sites. OpenStack is the standard open-source solution to address the IaaS level of the Cloud paradigm. When handling large-scale geographically distributed Cloud infrastructures (*e.g.*, Edge computing), it is required to handle multiple OpenStack sites in a collaborative manner to support the load, and to be relatively resilient to network disconnections [1]. One possible way is to decentralize the MariaDB database that handles data

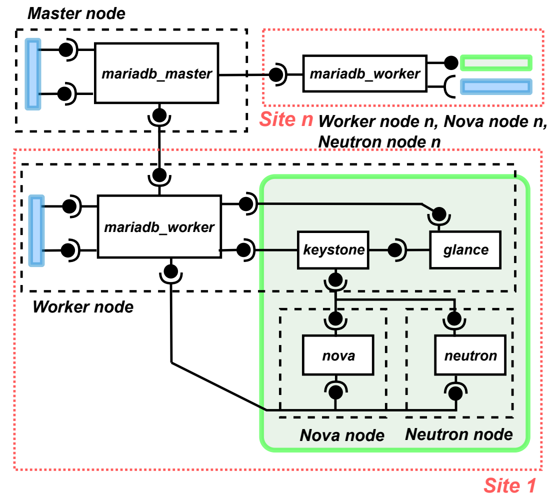


Fig. 1: Assembly of a multi-site OpenStack with a Galera cluster of distributed MariaDB databases. One node holds a Galera master and is connected to n sites. Each site is composed of three nodes. The content of the blue rectangles is not detailed, as not impacted by our choreography scenarios [12]. The content of the green rectangle is hidden in additional sites for the sake of readability.

required by the authentication service (*i.e.*, `Keystone`) and to replicate other services over the sites. In our case, a Galera cluster is used. Galera Cluster is an overlay for MySQL DBMS engines that enables replication between databases [13]. This use-case was showcased during the 2018 Vancouver OpenStack summit[1,2].

Figure 1 gives an overview of the assembly of services corresponding to this use-case with the different sites and nodes, and using the usual use/provide UML notation. Deploying and managing such a large-scale distributed software system is notably difficult [12], [14]. Multiple DevOps teams (typically one per site, and one for the master database) would be deployed to manage and operate such multi-site OpenStack. Our reconfiguration scenarios are to deploy and update the master database, which induces many changes to be triggered in other services as there is a chain of dependencies. Indeed, by following the topology classification of [11], this case study follows a stratified topology that mixes central users, central providers, and linear topologies. With n as the number of sites, a total of $1 + 3 \times n$ reconfiguration programs are required to handle this update in a decentralized manner, and a total of $3 + 20 \times n$ changes are required in $1 + 5 \times n$ services.

First, this use case illustrates why a single DevOps team cannot manually handle, without errors, such a complex set of changes with, for ten sites, more than 200 reconfiguration instructions to coordinate and write, thus requiring declarative IaC tools. Second, analyzing the state

[1] OpenStack summit video (https://youtu.be/AUzaJ8rBvEg)
[2] blog article

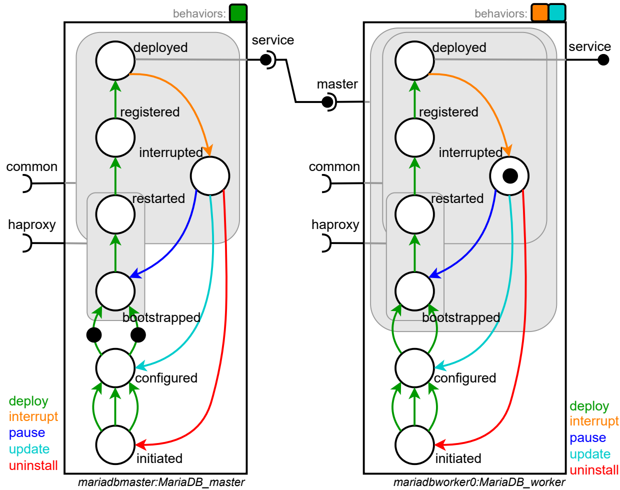


Fig. 2: Control components of `MariaDB master` and `MariaDB_worker`. State example when applying the reconfiguration plans of Fig. 4.

of the system among ten sites, and making appropriate reconfiguration decisions would be very difficult, which is why multiple DevOps teams would typically handle such complex cases. Third, it would also be hard and error-prone for more than ten DevOps teams (*i.e.*, at least one per site) to manually coordinate more than 200 changes in their local areas, *e.g.*, via phone, chat, or email, thus requiring a decentralized tool. Indeed, using a centralized declarative DevOps tool in this scenario (such as Puppet [15], Pulumi [8], or Terraform [16]), the DevOps team responsible for the master node would initiate an update on its MariaDB master service with its local instance of the tool. Local changes would be automatically generated by the local tool. However, to spread these changes to other DevOps teams, human exchanges would be required. Indeed, each team concerned by these changes (in this case all the other teams) would have to be informed and would have to manually write the new target state of their local services so that their own local declarative tool calculates the associated changes. This could once again lead to the need for additional human exchanges, etc. In contrast, a decentralized declarative tool such as Muse or Ballet automates this coordination process.

III. Usage of Ballet

From a user perspective, Ballet is a simple declarative tool to use in Python and YAML, where developers statically specify their component's life cycles and interfaces, and where DevOps teams submit reconfiguration goals on services. Then, Ballet automatically infers and executes in a decentralized manner the required instructions for each DevOps team (each node). Hence, the usage of Ballet is divided into two main concerns: the services developers' concern; and the DevOps concern.

A. Developers' concern

Developers in Ballet have to define the control components (components, for short) associated with their services as defined in Concerto [9], [10]. Components are not intended to represent the functional aspects of services but instead to pilot their life cycle. In other words, a Ballet component is a proxy around a new or legacy piece of software, written by its developer, that can be considered as a replacement for usual control scripts (installation, maintenance, suspension of service, etc.).

The topological interface of a component is specified by its *provide ports* and *use ports*. *Provide ports* denote services or data provided by a component when ports are *active*. *Use ports* denote requirements that the component has when ports are active. Internally, the components are characterized by *places* representing milestones in the life cycle, and *transitions* between places, mapped to concrete actions (*e.g.*, starting a virtual machine, downloading a Docker image, etc.). Each port is bound to a set of places (namely a group), representing the subpart of the life cycle where the port is active. The last characteristic attribute of a component is its set of *behaviors*, its operational interface. A behavior is a subset of transitions in a component. At execution, a component instance is associated with a behaviors queue, and the component can be requested to execute a behavior that is pushed in the queue. Then, similarly (but different) to Petri nets [10] tokens evolve through places and transitions associated to the current behavior, activating and deactivating ports when entering or leaving groups of places.

Figure 2 gives a graphical representation of two control component instances, `mdbmaster` for a MariaDB master, and `mdbworker0` for a MariaDB worker. Seven places (*e.g.*, `configured`, `deployed`) are specified in the control component type `MariaDB_master`, and ten transitions (arrows between places) that belong to five different behaviors, each represented by one color of transition and listed in the bottom right corner of the component (*e.g.*, `deploy`, `interrupt`). The provide port `service` is bound to the place `deployed`, while use ports are bound to groups of places (gray rectangle). Listing 1 gives a subpart of the Python code corresponding to the MariaDB master control component of Figure 2. In this listing only the transitions `configure0`, `configure1`, and `configure2`, are specified. They correspond to the three initial parallel transitions of the `deploy` behavior. Hence, each one is defined by the source place `initiated`, the destination place `configured`, the behavior `deploy`, and a callback function to write concrete actions associated with this transition (*e.g.*, `self.configure0`). This code is written by the developer of the service. Writing this code can be compared to writing usual installation and update scripts for a service in a more structured and composable way.

Listing 1: Control component MariaDB master in Python

```python
class MariaDB_master(Component):
    def create(self):
        self.places = [ "initiated", "configured", "
             bootstrapped", "restarted", "registered", "
             deployed", "interrupted"]

        self.transitions = {
            "configure0": ("initiated", "configured",
                           "deploy", self.configure0),
            "configure1": ("initiated", "configured",
                           "deploy", self.configure1),
            "configure2": ("initiated", "configured",
                           "deploy", self.configure2),
            ...
        }

        self.dependencies = {
            "service": (DepType.PROVIDE, ["deployed"]),
            "haproxy": (DepType.USE, ["bootstrapped","
                restarted"]),
            ...
        }

        self.initial_place = 'initiated'
        self.running_place = 'deployed'

    def configure0(self):
      # concrete actions
```

B. *DevOps' concern*

In Ballet, DevOps teams only use the interfaces of the control components, *i.e.*, ports and behaviors, and do not have to be aware of the internal life cycle of each component. The work of the DevOps is purely declarative. First, DevOps are responsible for specifying the new targeted assembly of components (*i.e.*, required component instances and their connections). The process of assembling components is a standard practice in component-based software engineering [17], thus, because of space limitations, we do not detail this process. Second, determining an inventory of addresses and ports to contact components is required. This can be done by DevOps, or system operators, or can be automatically generated through placement and scheduling algorithms [18]. Third, DevOps teams are responsible for giving reconfiguration goals. To this purpose, Ballet offers a novel declarative *goals language* in which the DevOps manipulates: behaviors, statuses of ports, and the two specific places representing the initial place and the running place of components.

Listing 2 gives the grammar of this goal language. In Ballet, a goal contains (1) the definition of one or more desired behaviors to execute on control components; (2) optionally the required state of the ports (*i.e.*, `active` `inactive`) after the reconfiguration; or (3) if components should reach their `initial` or `running` state after the reconfiguration. Ballet also offers a `forall` quantifier in the goal language. In this language, the order of statements is not important and `component` statements override the goals that are defined using `forall`.

Listing 3 gives the goals used in the *update* scenario of our case study in YAML (Fig. 1): the behavior `update` should be applied on `mdbmaster`); all components should be `running` after the reconfiguration.

Listing 2: Language to define reconfiguration goals for DevOps usage

```
<goals> ::= behaviors: <bhvr_list>
            ports: <port_list>
            components: <comp_list>

<bhvr_list> ::= <bhvr_item>
              | <bhvr_item> <bhvr_list>
<bhvr_item> ::= - forall: <bhvr_name>
              | - component: <comp_name>
                  behavior: <bhvr_name>

<port_list> ::= <port_item>
              | <port_item> <port_list>
<port_item> ::= - forall: <port_status>
              | - component: <comp_name>
                  port: <port_name>
                  status: <port_status>

<comp_list> ::= <comp_item>
              | <comp_item> <comp_list>
<comp_item> ::= - forall: <comp_status>
              | - component: <comp_name>
                  status: <comp_status>
```

Listing 3: Example of goals for updating mdbmaster in Fig. 1 in Yaml

```yaml
behaviors:
  - component : mdbmaster
    behavior : update
components:
  - forall : running
```

IV. Choreography Engine of Ballet

The architecture of Ballet is depicted in Figure 3. Ballet is composed of three pieces of software to solve choreographies: (1) the gateways are responsible for parsing the set of inputs of Ballet submitted to a front interface by DevOps (*i.e.*, assembly, goals, inventory) and collaborate to exchange information needed to initiate the choreography; (2) the planners are responsible for inferring the n reconfiguration programs associated with the inputs in a decentralized manner; and (3) the executors are responsible for executing in a distributed manner the n reconfiguration programs, hence terminating the reconfiguration. Note that, as depicted in the figure, there is one gateway, one planner, and one executor on each node, and a DevOps team can be responsible for more than one node, thus submitting its inputs to a front-end.

Since the gateway does not present a significant scientific challenge, the rest of this section only focuses on Ballet's planner and executor. While DevOps inputs are declarative, the role of planners is to generate imperative programs (i.e. a reconfiguration plan) for the executors. So, to facilitate understanding, we begin by detailing the imperative language used by executors, followed by an explanation of planners.

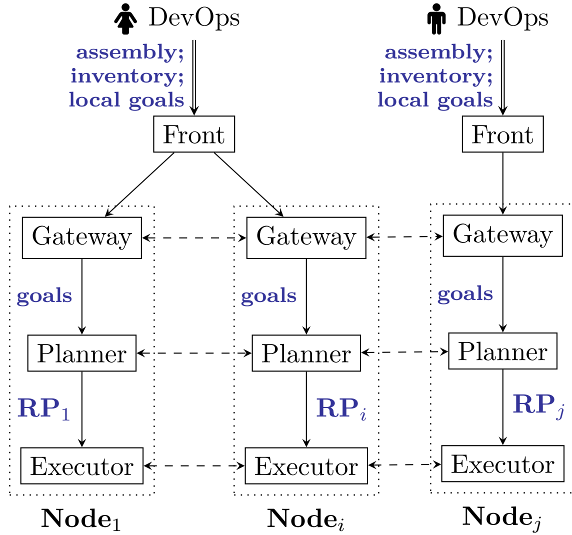


Fig. 3: Overview of Ballet and its three components: gateway, planner, executor. The dashed arrows represent collaborations, while the plain arrows represent the inputs and outputs of each component.

A. Executor

For its executor, Ballet extends the reconfiguration language Concerto [9], [10] that offers very efficient reconfiguration execution times thanks to the fine-grain granularity of components' life cycles dependencies, and to the offered level of parallelism.

In contrast to Concerto, where a single Reconfiguration Program (RP) is executed by a single central entity, and where transitions are executed remotely on distant nodes, the decentralized executor of Ballet handles n local subprograms directly executed on distant nodes with necessary communications for coordination.

The executor imperative reconfiguration language offers four commonly used topological instructions to create or modify an existing assembly of components [17]: $add(id, t)$ creates a new component instance id of type t; $del(id)$ instruction deletes a component instance id; for establishing connections between components $con(id1, p, id2, u)$ connects two components via the provide port p of the component instance $id1$ and the use port u of the component instance $id2$; finally, $dcon(id1, u, id2, p)$ disconnects previously connected components.

In addition to these four instructions, two additional instructions are possible in Ballet and Concerto. The instruction $pushB(id, bhv)$ pushes the request of executing the behavior bhv in the queue of the component instance id, while $wait(id, bhv)$ ensures synchronization by waiting for the completion of the behavior bhv on the component instance id. As the $pushB$ instruction is a non-blocking instruction that introduces concurrency of execution, the $wait$ instruction is a synchronization that ensures proper coordination and the appropriate order among component instances during the overall reconfiguration process.

When a control component instance executes a behavior, transitions associated with this behavior are triggered while respecting port synchronization, with semantics close (but different) to Petri nets [10]. Transitions are triggered until the component reaches a state where no further transitions can be fired for the current behavior.

At this point, the behavior request is considered complete and is popped from the behavior queue. The control component then proceeds to execute the next behavior in the queue.

Concurrency between component instances when executing behaviors leads to a need for coordination between components when connected through their ports. Indeed, a use port cannot be activated (*i.e.*, entering the group bound to the port) unless connected to an active provide port (*i.e.*, at least one token is present within the group bound to the port), and a provide port cannot be deactivated while connected to an active use port. Then, in Ballet, a component provides synchronization information to distant components in the following cases: (1) when the current *con* or *dcon* action is locally finished and involves another node; (2) when an expected behavior (i.e., *wait* instruction) is finished; (3) when a provide or use port within a local component is activated; (4) when a provide or use port within a local component is deactivated. Contrary to Concerto, from which our execution engine is inspired, synchronization information is shared through communications on the network.

Recall that this language does not have to be manipulated by either the developers or the DevOps in Ballet and is strictly used by the planner in an automated way.

Reconfiguration example: Figures 4a and 4b provide the reconfiguration programs for a master node and a worker node within a Galera cluster of databases. In this specific scenario, four messages will be exchanged: first, when the worker node stops using the master's service (triggered by worker's `interrupt` behavior); second, when the master node disables its service port (triggered by master's `interrupt` behavior); third, when the master node starts providing its service (`deploy` behavior); and finally, when the worker node resumes using the service (`deploy` behavior). Figure 2 illustrates a possible scenario when executing these programs with associated tokens: `mdbmaster` has finished and popped the behaviors `interrupt` and `update` and is executing its `deploy` behavior; and concurrently `mdbworker` is finishing the behavior `interrupt`, reaching the ending place of this behavior.

B. Planner

From the assembly given by a DevOps team, the planner is responsible for building the reconfiguration plan of each node. To this purpose, the current local assembly (*i.e.*, state) is compared to the one submitted by the DevOps team. If a component is missing or should not exist in the assembly, a `add` or `del` instruction is added for the

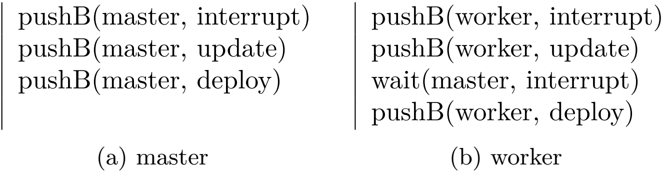
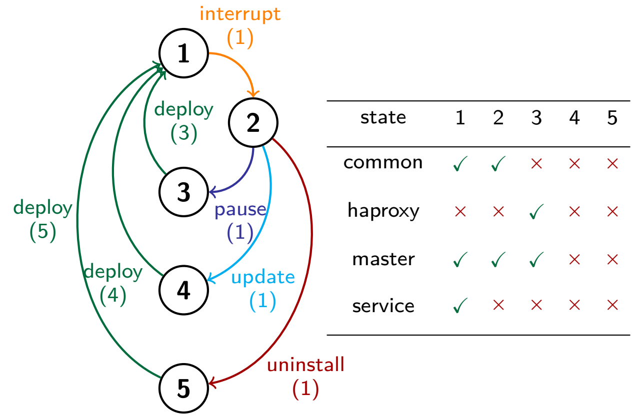


Fig. 4: Example of reconfiguration plans of MariaDB master and one MariaDB worker



state	1	2	3	4	5
common	✓	✓	×	×	×
haproxy	×	×	✓	×	×
master	✓	✓	✓	×	×
service	✓	×	×	×	×

Fig. 5: Automaton representation of $Mariadb_worker$ component's life cycle with its associated incidence matrix for ports statuses.

concerned node (inventory). If a connection is missing or should not exist in the assembly, a `con` or `dcon` instruction is added to the two concerned nodes. Then, Ballet adopts a classical structure of reconfiguration program [17] with three blocks: the first block contains `add` and `con` instructions; the middle block contains the `pushB` and `wait` instructions (see below for details on this block); the third block contains `dcon` and `del` instructions. By following this three-block structure, topological changes in the assembly cannot interfere with the result of the decentralized planners. Generating the correct ordered set of `pushB` and `wait` instructions (*i.e.*, middle block) is the most calling part of the planner. Unexpected emergent behaviors or deadlocks may arise due to interactions between components and behaviors.

Example of failing reconfiguration: To exemplify this difficulty, let us consider the same programs of Figure 4 containing *pushB* instructions but with the `wait` instruction after the `pushB(worker,deploy)` in the worker. Although the execution might seem to follow the same order, the non-deterministic order of parallelism makes the same output impossible to guarantee in this second case. Indeed, in the worst case, the worker could complete its entire reconfiguration, before the master begins its process. Hence, the efficiency introduced by the level of parallelism, and concurrency induces more complexity when inferring RPs.

To infer the middle block, Ballet's planner uses Constraint Programming (CP). CP is a declarative paradigm for solving combinatorial problems [19]. It allows a user to state a problem by describing the constraints (relations) holding on variables (unknowns). A domain of allowed values is defined for each variable by the user, and each constraint is equipped with a filtering algorithm (given by the CP community) that removes impossible values from the variable's domain. The purpose is to find a solution as a mapping of values to variables that satisfy specified constraints.

1) Local Resolution: The local resolution consists of inferring a local valid sequence of behaviors for each component to satisfy local reconfiguration goals. For each control component, the planner builds a labeled automaton in which it is going to find a valid word according to some constraints given by the goals. In this automaton, each state corresponds either to a starting or ending place of a control component behavior, and each transition corresponds to the associated behavior to move from one state to another. Each transition is also associated with a cost representing the number of transitions in the control component to apply this behavior. In other words the automaton models from which to which behavior it is possible to move and at which cost. As a result, a valid word in this automaton is a valid sequence of behaviors in the control component of Ballet. Each state in the automaton also indicates the state of the port (active or inactive) at the beginning or the end of a behavior, hence the automaton also permits to meet specific port constraints. Figure 5 gives an example of the automaton associated with the mariadb control component presented in Figure 2. Each transition is labeled by its corresponding behavior and weight. The table on the right side of the figure represents the port statuses associated with each state of the automaton.

The problem of finding a sequence of behaviors constrained by reconfiguration goals can be modeled and solved using CP techniques. The formulation of the problem includes several variables and is given by Model 1: (i) B denotes a sequence of behaviors of size m (m being an integer with a fixed maximal size where all behaviors are applied), $b_i \in B$ is the behavior of the control component at step i; (ii) S denotes an array of size $m+1$, $s_i \in S$ (resp. $s_{i+1} \in S$) the automaton state at step i, before (resp. after) executing b_i; and (iii) C denotes a global cost for executing all behaviors in B.

In the following, s_{init} and S_{goal} respectively denote the current state of the modeled component, and the accepted states at the end of the reconfiguration. The constraint (1) ensures that the sequence of behaviors represented by B is accepted by the given finite automaton Π [20], starting from state s_{init}. To specify the state goals, the set S_{goal} is reduced, according to the provided goals. By default, S_{goal} contains all the states of the automaton. Equation (2) (which is also considered as a constraint in CP) is used to

Minimize C subject to

$$\textsc{Regular}(B, \Pi, s_{init}, S_{goal}) \quad (1)$$
$$s_{i+1} = inc_{\Pi}[s_i][b_i], \;\; \forall i \in 1..m \quad (2)$$
$$\textsc{Count}(b, B, >, 0) \quad (3)$$
$$status(p, s_{m+1}) = \Gamma_p \quad (4)$$

where $\Gamma_p \in \{\texttt{active}, \texttt{inactive}\}$
$c_i = cost(s_i, b_i), \;\; \forall i \in 1..m$
$C = \textsc{Sum}([c_i \mid i \in 1..m])$

Model 1: A CP model for finding a sequence of behavior to execute from an automaton.

define the state after applying a behavior b_i from a state s_i by using the incidence matrix of the automaton. The behavior goals are expressed using the $\textsc{Count}(v, V, r, L)$ constraint [21]. This constraint holds if the number N of variables in V assigned to the value v verifies NrL, with r being a relational operator and L a limit. For instance, Constraint (3) states that the behavior b must be executed at least once. Furthermore, Constraint (4) models objectives on the status of ports, specifying the status a port p should have after running the last behavior of the reconfiguration sequence. The cost of running each behavior is represented by the variable c_i, taking into account the current state s_i, captured in S, from which the behavior is executed. The cost function estimates the cost by considering the transitions of the component model. (e.g. in Fig. 5 the associated cost for running `deploy` from `state 5` is greater compared to executing it from `state 1`). The objective function aims at minimizing the variable C, which represents the sum of costs for executing all behaviors in the resulting sequence.

The above model is purely local, thus it does not include some constraints related to external ports. These missing constraints will be addressed thanks to a distributed protocol, described below.

2) Constraint propagation: Intuitively, when handling the planning problem in a decentralized fashion, applying behaviors in one component on a given node may lead to the deactivation of a provide port, which results in the need for another component (hosted on another node) to execute additional behaviors, typically to disable their using ports. Consequently, Ballet adopts a distributed protocol inspired by gossip [22] to propagate constraints throughout the system, enabling concerned components to adjust their behaviors based on port statuses.

When a behavior changes the status of a control component's port, a message is sent to the connected neighbor components. A message contains the identifier of the source component, the port that is affected and how (i.e., active or inactive), and the name of the behavior responsible for this change. These messages are interpreted as additional constraints. The new constraints are used to extend the local automaton and the CP model to ensure that the locally connected ports also change their statuses during the local resolution. In other words, the locally connected ports have a count greater than zero in the sequence of states obtained during local resolution. This extension is achieved using the following constraint, where s_p can take values of either `active` or `inactive`:

$$\textsc{Count}(s_p, [status(p, s_i) \mid i \in 1..m+1], >, 0)$$

This propagation mechanism has to reach a convergence and ending point. To this purpose, an acknowledgment protocol is employed during this phase. It is inspired by distributed consensus algorithms such as Paxos [23] or Raft [24].

3) Final plan inference: When all constraints have been propagated (*i.e.*, acknowledgment received), a final resolution has to be applied. The previously received messages are used once again in the final resolution to enrich the automaton with synchronization instructions (*i.e.*, wait). Intuitively, when a constraint is received from a connected component, meaning that the status of a connected external port will change during a given behavior, the external behavior responsible for this change has to be waited before the local application of a behavior that also changes the local connected port status. To ensure this synchronization, all local automaton states related to the concerned port are enriched with a new transition labeled as `wait_cpnt_bhv`, with a cost of 0. The domain of each b_i (behavior) now includes new `wait` instructions in the CP model. Furthermore, an additional Count constraint is declared on the sequence. This constraint ensures that at least one occurrence of `wait_cpnt_bhv` appears in the sequence. This last constructed model is used to locally find a planning solution.

V. Evaluation

Our implementation of Ballet, the experimental scenarios, and the results are all accessible at the following Zenodo link https://doi.org/10.5281/zenodo.10472116.

Ballet is implemented in 8400 LoC using Python 3.9.2 with packages `grpcio` (version 1.47.0) for communications between the planners, and `flask` (version 2.2.2) for the executor's coordination. To solve our CP problems we have chosen the high-level solver-independent constraint modeling language MiniZinc [25] (version 2.7.6), and the Python package `minizinc` (version 0.9.0) as an interface for it. Among all possible solvers available with MiniZinc, the open-source C++ constraint solver Gecode [26] (version 6.3.0) has shown better performances in our case and is used in the following.

A. Experimental setup

We conducted a series of experiments on Ballet and Muse by using the multi-site OpenStack use-case detailed in Section II. We have evaluated two choreography scenarios, one to *deploy* a full multi-site OpenStack and another to *update* the MariaDB master node of an already deployed

multi-site OpenStack. For the running scenario, the definition of the goals is shown in Listing 3. Our experiments range from 1 to 10 sites following the assembly of Figure 1 (*i.e.*, maximum of 31 nodes for 10 sites).

For the sake of simplicity and reproducibility, we have decided to use traces of third-party previous real experiments on OpenStack and Galera [10], [12]. Traces are available online[3]. Our experiments, while using past traces, are performed on a real infrastructure offered by the experimental platform Grid'5000, thus facing real distribution and communications between nodes. Evaluations were conducted on the *Gros* and *Paravance* clusters[4]: *Gros* is composed of 124 hosts equipped with one 18-core Intel Xeon Gold 5220 CPUs, 96GB RAM, 480 GB + 960 GB SSD, and a network interface with a transfer rate of 2 × 25 Gbps (SR-IOV); *Paravance* has 72 hosts equipped with two 8-core Intel Xeon E5-2630 with 128 GiB of memory, 2×600 GB HDD, and a transfer rate of 2 x 10 Gbps (SR-IOV) on the network. EnosLib [27] scripts have been used as a front interface for Ballet's nodes.

B. Results

1) Inference of the reconfiguration plans: For both our deployment and update scenarios, from 1 to 10 sites, Ballet can correctly generate the reconfiguration programs of each node in a decentralized manner. The programs generated for each node are strictly local to the concerned nodes. The order of instructions is correct (checked manually) and the execution of the generated plans has successfully been performed without any deadlock or unwilling emergent behavior. Table I gives an overview of the work achieved by the planner on the *update* scenario. The number of messages, that are responsible for propagating constraints, reflects the number of synchronizations that would be needed between DevOps teams to complete the design of the reconfiguration plan. Considering n sites, the reconfiguration goal leads to $n \times 9$ required messages, and $8 + 11 \times n$ inferred constraints. This scenario also leads to a total of $3 + 20 \times n$ instructions distributed among $1 + 3 \times n$ reconfiguration programs, all inferred by the planners from a few lines of YAML goals (Listing 3) with a strict local information level. Then, thanks to the inference of constraints and a communication protocol, we can generate an efficient distributed plan.

2) Execution time: For both the *deploy* and *update* scenarios, Ballet consistently outperformed Muse. Table II illustrates the average execution times for 10 runs for each configuration, *i.e.*, from 1 to 10 sites and one extra master database node, with a standard deviation lower than 0.1s. Note that both Ballet and Muse automatically infer the set of reconfiguration plans, however, in Muse this inference is computed progressively when executing changes, while Ballet calculates all the plans in a decentralized manner

#Sites	#Messages	#Constraints	#Instructions
1	9	19	23
2	18	30	43
5	45	63	103
10	90	118	203

TABLE I: Results of the planning phase for the *update* scenario when varying the number of `Mariadb_workers` in a Galera cluster. The number of messages sent in the gossip algorithm, the associated number of constraints, and reconfiguration instructions inferred by the planners are given.

Sc.	# Sites	Ballet Planning	Ballet Execution	Total	Muse	Gain
Deploy	1	1.69s	306.02s	307.71s	536.57s	42.7%
	2	1.78s	306.09s	307.86s	536.69s	42.6%
	5	1.77s	306.19s	307.97s	537.09s	42.7%
	10	2.02s	306.14s	308.19s	538.13s	42.7%
Update	1	3.36s	416.84s	420.20s	555.56s	24.4%
	2	4.39s	416.92s	421.31s	555.70s	24.2%
	5	6.05s	417.17s	423.22s	556.08s	24.0%
	10	5.97s	417.46s	423.43s	556.77s	24.0%

TABLE II: Comparison of durations (seconds) for planning and executing a deployment of a multi-site OpenStack (from 1 to 10 sites) and an update of the `MariaDB_master` instance with Ballet and Muse. Results have shown a very low standard deviation (< 0.1s) for running any scenarios with both engines.

before execution which is why planning and execution phases are not dissociated for Muse. On average, Ballet takes about 307 seconds to plan and perform a deployment for any number of sites, and 424 seconds to plan and perform the update. It represents a gain of 42.6% (resp. 24%) for the *deploy* (resp. *update*) scenario, compared to Muse for all number of sites.

Unlike the *deploy* scenario, the *update* does not offer a lot of opportunity for parallelism which partly explains the differences in gain. Furthermore, in contrast to Ballet, Muse adopts an immutable approach in the update case where components are recreated before old versions are destroyed. Indeed, the granularity of the life cycles in Muse (on-off) does not offer as much flexibility as Ballet to interrupt services, update them, or restart them. Hence, the set of instructions for Muse and Ballet in the *update* scenario are not strictly identical. In particular, some update commands on databases, performed by Ballet only (extracted from real traces), are long compared to others. In other words, this case is not very favorable to Ballet, but still Ballet shows a performance improvement of 24% thanks to its concurrency and parallelism.

As illustrated in Table II, the computation time of Ballet is driven by the time to execute the reconfiguration plan. Contrary to the *deploy* scenario, planning an update needs communications between nodes. Table III provides average times, and its standard deviation, for inferring plans for the *update* scenario. On the one hand, the *Solving* column includes the time required for behavior inference, and for

[3]https://doi.org/10.5281/zenodo.10472116
[4]https://www.grid5000.fr/w/Hardware

#Sites	Solving	Communications	Total
1	1.58 (0.06)	1.78 (0.44)	3.36 (0.43)
2	1.53 (0.13)	2.85 (1.62)	4.39 (1.72)
5	1.59 (0.06)	4.47 (0.92)	6.05 (0.91)
10	2.61 (0.17)	0.26 (0.01)	5.97 (0.63)

TABLE III: Average duration in seconds (and standard deviation) to calculate the plans for the *update* scenario.

solving the CP model to generate the final plan. On the other hand, the *Communications* column reflects the time taken for data exchanges between nodes in our distributed protocol. The results reveal stable values for solving, but some variations in the *Communications* phases. As the proportion of work involved in the planning phase is low compared to execution, these variations have little impact on the overall results.

Overall, our experimental results show that the granularity adopted by Ballet (through the decentralization of Concerto) to control the component's life cycle, and the concurrency introduced by its reconfiguration language, improves the performance of a decentralized reconfiguration compared to Muse, both for the *deploy* and *update* scenarios. This was not obvious because the granularity and concurrency introduced by Concerto also increase the plan inference complexity compared to Muse.

3) Expressiveness and flexibility: While both Muse and Ballet aim to facilitate DevOps operations, they differ in their usage. First, from the developer perspective, Muse offers a fixed life cycle of resources and services with three functions *create*, *destroy*, and *update* coupled with internal parameters. Muse also requires an additional function from the developer (*i.e.*, *diff*) to indicate which function to trigger in which cases when the parameters of the service or resource are changed. In contrast, in Ballet, the life cycle is fully programmable, and a clear model and programming support are offered to developers to express in which case (*i.e.*, behavior) a set of actions (*i.e.*, transitions associated to the behavior) have to be executed. Ballet also automatically handles parallelism of actions in the life cycle. Thus, the expressiveness at the developer level is improved with Ballet as offering a programmable life cycle with associated models and semantics.

Second, from a DevOps viewpoint, Muse requires DevOps to give a new assembly of services/resources and a set of modified parameters in each of those services and resources. In contrast, with Ballet, DevOps also gives a new assembly but, instead of changing the values of internal parameters, a goal language with clear semantics that manipulates the interfaces exposed by control components (*i.e.*, ports, behaviors) is used. It is difficult to claim that the expressiveness is improved by Ballet at the DevOps level as the same set of actions seems possible through parameters and behavior requests. But we can claim that the complexity is not increased by Ballet.

The above results give elements to answer our research questions on our multi-site OpenStack case study. First, (RQ1) Ballet correctly generates an efficient distributed plan in a decentralized manner for cross-DevOps operations while asking the DevOps a strictly local information level. Second, (RQ2) Ballet reduces the execution time of cross-DevOps operations compared to Muse, thanks to programmable life cycles, without adding complexity to the input provided by DevOps, thanks to its goal language and associated planner.

4) Threats to validity: As in any empirical study, there are threats to the validity of our work. First, Ballet has been experimented on a single real case study, presented above, and additionally some synthetic use cases not presented here for space reasons. To further validate our approach, we intend to evaluate Ballet on additional real use cases in an extended version of this paper. Second, there may be errors in our design and implementation of Ballet. If additional use cases may help detect such problems, we intend to cover a larger set of tests as well as explore a formal study of Ballet (as done with Concerto [10]). Finally, it is clear in the above evaluation that the complexity of using Ballet is difficult to establish. A survey and test campaign with DevOps teams would be needed to evaluate this metric.

VI. Related Work

First, when applying a reconfiguration to a set of interconnected services (*i.e.*, component assembly), the life cycle of the services has to be manipulated (*e.g.*, on, off, updated, interrupted, etc.). The simplest way of handling this life cycle is to consider that a component is either *on* or *off* [17], [28]. However, such a simple model is too limited in many cases (for instance, the stop propagation [29]). The life cycle flexibility also influences the granularity of the dependencies between components. When the life cycle is programmable (*i.e.*, freely coded and specified by the user), more parallelism can be exposed, thus reducing the duration of reconfiguration [10], [30]. In the literature and in DevOps tools, the life cycle modeling can either be fixed or programmable. *The life cycle modeling is our first metric of interest.* Second, as already explained, we target decentralized declarative reconfiguration in this paper. *The decentralization of declarative dynamic reconfiguration is our second metric of interest* split into three parts in Table IV: the fact of offering an automated planner, thus a declarative approach; the fact of offering a cross-DevOps mechanism for both the planner and the execution, hence a decentralized approach. Table IV sums up the presented related work below.

As already mentioned in the introduction, DevOps IaC solutions offer reconfiguration capacities in a declarative manner. Examples of widely used solutions include Ansible [31], where a declarative approach is not strictly possible but where idempotence is possible by using modules; Terraform [16], Puppet [15], or Pulumi [8] that offer

	Ansible [31]	Puppet [15]	Terraform [16]	Pulumi [8]	Kubernetes [32]	Aeolus [30], [33]	Concerto [10], [11]	[5]	[6]	Muse [4]	Ballet
Life cycle	prog	prog	fixed	fixed	fixed	prog	prog	fixed	prog	fixed	prog
Planner	✗	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓
CDO plan	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓
CDO exec	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓

TABLE IV: Comparison of tools in the literature according to the metrics of interest. The life cycle can either be fixed or programmable. *Planner* denotes the fact of having (or not) an automated planning phase within the tool. The *CDO plan* and *CDO exec* denote the fact of having a cross-DevOps decentralized mechanism for either the planner or the executor.

declarative provisioning of various kind of resources, thus where a planning phase exists; orchestration tools, like Kubernetes [32], that offer automatic scalability features, and automatic restart after failure for instance. However, such tools adopt a centralized vision, where both the plan computation (*i.e.*, the imperative program associated with the declarative requirements) and its execution are done by a single central entity.

Dynamic reconfiguration, alongside DevOps tools, is also a domain of component-based software engineering (CBSE) [17]. Aeolus [30] and Concerto [10] are the component models offering the highest degree of flexibility to define life cycles and are exclusively made for this purpose. Both offer a declarative way of handling reconfiguration thanks to their respective planners [11], [33] that automatically infer reconfiguration programs from a specified goal. However, these solutions adopt a centralized approach.

In [5] and [6] solutions to coordinate and deploy an application made of multiple components are presented. Each component of the application expresses its dependencies with the other components in a central plan, distributed to the corresponding nodes, deploying their part of the application. The executions of the deployments are then coordinated between the nodes according to their dependencies. Here, the execution is decentralized. The plan however is manually written in a centralized manner. In [5] the solution uses TOSCA descriptions for services, hence having a non-customizable life cycle [9], [10], while in [6] a generic specific language is used to model the system by creating custom elements, dependencies, and life cycles. Muse [4] allows the deployment; update and destruction of multiple services and resources in a coordinated way. Furthermore, the creation of the plan is automated and decentralized. In Muse, the life cycle modeling is fixed, thus limiting the level of parallelism and concurrency between reconfiguration plans.

A set of contributions in the literature try to decentralize container-based orchestrators, such as Kubernetes. However, the proposed solutions focus on decentralizing the scheduler of Kubernetes that assigns jobs to resources, not how to execute and coordinate in a decentralized manner multiple reconfiguration requirements from DevOps teams (*i.e.*, manifests). For this reason, those contributions are out of our scope [34]–[36].

Finally, many tools exist to automatically decide the set of requirements without the intervention of DevOps teams [17]. In particular, in [37], [38] constraint programming and SMT solvers are leveraged in this purpose. In this paper, we consider this information as input coming from DevOps teams or from such a tool, and we focus on the step after this decision: from a declarative set of requirements, how to compute the required program to reach it, and then how to execute it, both in a decentralized manner. Those approaches are thus complementary to our contribution.

VII. Conclusion

In this paper, we have presented Ballet, a declarative tool to coordinate cross-DevOps reconfiguration procedures decentrally. Ballet takes as inputs a set of declarative goals from DevOps teams. It automatically performs the following actions: (1) computes in a decentralized manner the set of reconfiguration plans on all nodes directly or indirectly affected by the goals from a simple declarative file; (2) executes the set of reconfiguration plans in a decentralized manner, by automating required communications between nodes; (3) speeds up the overall reconfiguration procedure. Ballet has been evaluated on the real use case of a multi-site OpenStack with a Galera cluster of MariaDB databases for two scenarios: the full deployment of the system, and an update of the master database that induces many changes in other services. First, results have shown the interest of the approach from a DevOps perspective: with n the number of sites, from a few lines of goals, $3 + 20 \times n$ reconfiguration instructions have automatically been generated within $1 + 3 \times n$ nodes, and $n \times 9$ messages (originating $8 + 11 \times n$ inferred constraints) have been avoided between DevOps teams. Second, results have shown that Ballet is faster to perform the choreography than Muse (choreography tool of the literature on top of Pulumi) with a gain of 42.6% for the deployment scenario, and 24% for the update scenario. We plan in the future to plug upgrade Ballet with an automated and decentralized way to decide the set of DevOps goals [39]. Furthermore, we plan to formalize both the executor's and planner's semantics and use it to certify Ballet. For instance, we may verify the correctness of the generated plans, the correctness of the generated automaton for CP, etc.

Acknowledgement

The authors acknowledge the support of the French Agence Nationale de la Recherche (ANR), under grant ANR-20-CE25-0017 (SeMaFoR project)

References

[1] R.-A. Cherrueau, A. Lebre, D. Pertin, F. Wuhib, and J. M. Soares, "Edge Computing Resource Management System: a Critical Building Block! Initiating the debate via OpenStack," in *HotEdge 2018 - USENIX Workshop on Hot Topics in Edge Computing*, 2018.

[2] L. Leite, C. Rocha, F. Kon, D. Milojicic, and P. Meirelles, "A survey of devops concepts and challenges," *ACM Comput. Surv.*, 2019.

[3] K. Nybom, J. Smeds, and I. Porres, "On the impact of mixing responsibilities between devs and ops," in *Agile Processes, in Software Engineering, and Extreme Programming*, 2016.

[4] D. Sokolowski, P. Weisenburger, and G. Salvaneschi, "Automating serverless deployments for devops organizations," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[5] K. e. a. Wild, "Decentralized cross-organizational application deployment automation: An approach for generating deployment choreographies based on declarative deployment models," in *Advanced Information Systems Engineering*, 2020.

[6] H. Herry, P. Anderson, and M. Rovatsos, "Choreographing configuration changes," in *Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013)*, 2013.

[7] L. Cruz-Filipe, E. Graversen, L. Lugović, F. Montesi, and M. Peressotti, "Modular Compilation for Higher-Order Functional Choreographies," in *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, 2023.

[8] "Pulumi," https://www.pulumi.com/, 2023, accessed: 2023-24-10.

[9] M. Chardet, H. Coullon, and C. Pérez, "Predictable Efficiency for Reconfiguration of Service-Oriented Systems with Concerto," in *CCGrid 2020 : 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*, 2020.

[10] M. Chardet, H. Coullon, and S. Robillard, "Toward Safe and Efficient Reconfiguration with Concerto," *Science of Computer Programming*, 2021.

[11] S. Robillard and H. Coullon, "SMT-Based Planning Synthesis for Distributed System Reconfigurations," in *FASE 2022 : 25th International Conference on Fundamental Approaches to Software Engineering*, 2022.

[12] M. Chardet, H. Coullon, C. Pérez, D. Pertin, C. Servantie, and S. Robillard, "Enhancing Separation of Concerns, Parallelism, and Formalism in Distributed Software Deployment with Madeus," 2020, working paper or preprint. [Online]. Available: https://inria.hal.science/hal-02737859

[13] "Galera cluster," https://galeracluster.com/, 2023, : 2023-24-10.

[14] H. Coullon, D. Pertin, and C. Pérez, "Production Deployment Tools for IaaSes: an Overall Model and Survey," in *The IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*, 2017.

[15] "What is puppet," http://puppetlabs.com/puppet/what-is-puppet, 2014, : 2023-24-10.

[16] Y. Brikman, *Terraform: Up and Running.* O'Reilly Media, Inc., 2022.

[17] H. Coullon, L. Henrio, F. Loulergue, and S. Robillard, "Component-based distributed software reconfiguration: A verification-oriented survey," *ACM Comput. Surv.*, 2023.

[18] F. A. Salaht, F. Desprez, and A. Lebre, "An overview of service placement problem in fog and edge computing," *ACM Comput. Surv.*, vol. 53, 2020.

[19] F. Rossi, P. van Beek, and T. Walsh, Eds., *Handbook of Constraint Programming*, ser. Foundations of Artificial Intelligence, 2006.

[20] G. Pesant, "A regular language membership constraint for finite sequences of variables," in *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, ser. Lecture Notes in Computer Science, M. Wallace, Ed., vol. 3258, 2004.

[21] "count," https://sofdem.github.io/gccat/gccat/Ccount.html, 2014, accessed: 2023-07-12.

[22] M. Jelasity, A. Montresor, and O. Babaoglu, "Gossip-based aggregation in large dynamic networks," *ACM Trans. Comput. Syst.*, vol. 23, no. 3, p. 219–252, aug 2005. [Online]. Available: https://doi.org/10.1145/1082469.1082470

[23] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, 2001.

[24] D. Huang, X. Ma, and S. Zhang, "Performance analysis of the raft consensus algorithm for private blockchains," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 50, no. 1, pp. 172–181, 2019.

[25] "Minizinc 2.7.6," https://www.minizinc.org/resources.html, 2023, accessed: 2023-07-12.

[26] "Gecode 6.3.0," https://www.gecode.org/index.html, 2021, accessed: 2023-07-12.

[27] R.-A. Cherrueau, M. Delavergne, A. van Kempen, A. Lebre, D. Pertin, J. R. Balderrama, A. Simonet, and M. Simonin, "Enoslib: A library for experiment-driven research in distributed computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 6, pp. 1464–1477, 2022.

[28] A. Basu, B. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis, "Rigorous component-based system design using the bip framework," *IEEE Softw.*, 2011.

[29] L. Henrio and M. Rivera, "Stopping safely hierarchical distributed components: application to gcm," in *CBHPC '08: Proceedings of the 2008 compFrame/HPC-GECO workshop on Component based high performance*, 2008.

[30] R. Di Cosmo, J. Mauro, S. Zacchiroli, and G. Zavattaro, "Aeolus: A component model for the cloud," *Information and Computation*, 2014.

[31] "Ansible," https://www.redhat.com/en/technologies/management/ansible, 2023, : 2023-24-10.

[32] B. Burns, J. Beda, K. Hightower, and L. Evenson, *Kubernetes: up and running.* O'Reilly Media, Inc., 2022.

[33] T. A. Lascu, J. Mauro, and G. Zavattaro, "Automatic deployment of component-based applications," *Science of Computer Programming*, 2015.

[34] M. A. e. a. Tamiru, "mck8s: An orchestration platform for geo-distributed multi-cluster environments," in *2021 International Conference on Computer Communications and Networks (ICCCN)*, 2021.

[35] Y. Xiong, Y. Sun, L. Xing, and Y. Huang, "Extend cloud to edge with kubeedge," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, 2018.

[36] L. L. Jiménez and O. Schelén, "Docma: A decentralized orchestrator for containerized microservice applications," in *2019 IEEE Cloud Summit*, 2019.

[37] J. A. Hewson, P. Anderson, and A. Gordon, "A declarative approach to automated configuration," in *Proceedings of the 26th international conference on Large Installation System Administration: strategies, tools, and techniques*, 2012.

[38] E. Ábrahám, F. Corzilius, E. B. Johnsen, G. Kremer, and J. Mauro, "Zephyrus2: On the fly deployment optimization using smt and cp technologies," in *Dependable Software Engineering: Theories, Tools, and Applications*, 2016.

[39] A. Alidra, H. Bruneliere, H. Coullon, T. Ledoux, C. Prud'Homme, J. Lejeune, P. Sens, J. Sopena, and J. Rivalan, "SeMaFoR - Self-Management of Fog Resources with Collaborative Decentralized Controllers," in *SEAMS 2023 - IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2023. [Online]. Available: https://hal.science/hal-04043471