Painless Transposition of Reproducible Distributed Environments with NixOS Compose

Quentin Guilloteau, Jonathan Bleuzen, Millian Poquet, Olivier Richard

Painless Transposition of Reproducible Distributed Environments with NixOS Compose

Quentin Guilloteau, Jonathan Bleuzen, Millian Poquet, Olivier Richard
Univ. Grenoble Alpes, Inria, CNRS, LIG
38000 Grenoble France
Firstname.Lastname@inria.fr

***Abstract*—Development of environments for distributed systems is a tedious and time-consuming iterative process. The reproducibility of such environments is a crucial factor for rigorous scientific contributions. We think that being able to smoothly test environments both locally and on a target distributed platform makes development cycles faster and reduces the friction to adopt better experimental practices. To address this issue, this paper introduces the notion of environment transposition and implements it in *NixOS Compose*, a tool that generates reproducible distributed environments. It enables users to deploy their environments on virtualized (docker, QEMU) or physical (*Grid'5000*) platforms with the same unique description of the environment. We show that *NixOS Compose* enables to build reproducible environments without overhead by comparing it to state-of-the-art solutions for the generation of distributed environments (*EnOSlib* and *Kameleon*). *NixOS Compose* actually enables substantial performance improvements on image building time over *Kameleon* (up to 11x faster for initial builds and up to 19x faster when building a variation of an existing environment).**

Keywords: Reproducibility; Distributed Systems; System Image; Deployment; Nix

I. Introduction

The scientific community as a whole has been traversing a reproducibility crisis for the last decade. Computer science does not make an exception. While scientists can accomplish reproducibility in different manners, it requires additional work and discipline. In 2015, Collberg et al. [8] studied the reproducibility of 402 articles published in journals and conferences on computer systems. Each of those articles linked the source code used to produce the results. Out of these 402 papers, Collberg et al. could not reproduce 46 %. The causes were: *(i)* the source code was unavailable, *(ii)* the source code did not compile or run, and *(iii)* the experiments required specific hardware.

In this article, we tackle problem *(ii)* in the context of distributed systems. We aim to make the **entire** software stack involved in a experiment of distributed systems reproducible. This implies making the compilation and the deployment of this stack reproducible, allowing one to rerun the experiment on an identical environment in one week or ten years.

To reach this goal, we exploit the declarative approach for system configuration provided by the *NixOS* [12] Linux distribution. *NixOS* makes use of a configuration that describes the environment of the entire system, from user-space to the kernel. The distribution is itself based on the purely functional *Nix* package manager [11]. The definition of packages (or system configuration in the case of *NixOS*) are functions without side effects, which enables *Nix* and *NixOS* to reproduce the exact same software when the same inputs are given.

We extend this notion of system configuration for distributed systems in a new tool named *NixOS Compose*. *NixOS Compose* enables to define distributed environments and to deploy them on various platforms that can either be physical (*e.g.,* on the *Grid'5000* testbed [1]) or virtualized (Docker or QEMU [2]). *NixOS Compose* exposes the **exact same user interface** to define and deploy environments regardless of the targeted platform. We think that this functionality paired with fast rebuild time of environments improves user experience, and we hope that it will help the adoption of experimental practices that foster reproducibility.

The paper is structured as follows. Section II gives the state-of-the-art on tools related to the reproducible deployment of distributed systems, and positions *NixOS Compose* in it. Section III presents *NixOS Compose*, its main concepts, the external concepts it relies on, and the users' utilization workflow for setting up a complete reproducible system and software stack. Section IV gives technical details on how *NixOS Compose* works. Section V presents how *NixOS Compose* can be used on a complex example that combines several distributed middlewares. Section VI offers experimental performance results of *NixOS Compose* against standard state-of-the-art tools using multiple metrics. Finally, Section VII concludes the paper with final remarks and perspectives on reproducible works and environments.

II. State of the Art

Several terminologies exist on the reproducibility of experiments. In this article we use the terms defined by Feitelson in [15]. In particular, an experiment is said to be *repeatable* if one can rerun it and obtain the same results by using the original artifacts (scripts and configuration files needed to run the experiment). If the same can be achieved without using the original artifacts but by recreating them, the experiment is said to be *replicable*. *Variation* is an orthogonal concept that can be applied to repeat or replicate an experiment with a modification of some parameters. Mercier et al. emphasize in [21] that the *production* environment (the final one used to execute the experiment) is not enough to achieve variation

in an experiment, as the *development* environments contain valuable information for the experiment and are also required.

A. Reproducibility of a local software environment

Several approaches exist to encapsulate a software environment. A first solution is to provide virtual machine (VM) images or containers of the environment. VMs are portable but induce a virtualization cost at runtime which hinders some experiments. Containers are much more lightweight but limit the possible usages (*e.g.,* Docker containers do not encapsulate the version/configuration of the Linux kernel, while this may impact how software behaves). Building VMs and containers in a reproducible fashion can be hard, but tools like Packer [25] can recreate identical virtualized images (either virtual machines or containers) from a unique description.

Another approach is to rely on the properties of package managers. Spack [16] is a package manager used on HPC platforms to share software environments. Spack does not control all the dependencies by design as it can reuse software already present and configured on the system, which is detrimental for reproducibility. Courtes et al. [10] presents purely functional package managers (*Nix* [11], *Guix* [9]) as good candidates to share complex software environments between users. Both approaches are very similar but *Nix* has a more complete ecosystem as we write these lines, so we only consider *Nix* in the remainder of this article.

Each *Nix* package is defined as a function without side effects, which means *Nix* should produce the exact same package if the function inputs are the same (source code version, build options, build and runtime dependencies and their versions, build toolchain and their versions, target architecture). A software environment is defined as a list of *Nix* packages. As *Nix* can reproduce each package, the entire environment is reproducible. This partially solves the well known issue of *Dependency Hell* as *Nix* demands to explicit every dependency and their version to build the package. *Nix* is limited to user-space environments, but the environment on the kernel side matters as well. The *NixOS* Linux distribution, based on *Nix*, solves this issue by describing the entire operating system (*i.e.,* kernel, softwares, configuration files, services) with a descriptive *Nix* expression. The descriptive approach taken by *NixOS* greatly simplifies the work of a reproducer wanting to perform variations of the environment [21].

Perenniality of environments can be hard to achieve on methods based on the reconstruction of a target software and every of their dependencies, as some source code can disappear over time. To solve this issue, reconstruction approaches can use source code from initiatives such as Software Heritage [29], whose goal is to collect, preserve and share publicly available software in source code form.

B. Reproducibility of a distributed software environment

Achieving reproducibility in a distributed setting brings new challenges, as the environments must be deployed on all the involved machines. A common approach is to somehow generate full system *images*, then to deploy them on the nodes with tools such as *Kadeploy* [17]. Creating such images manually can be cumbersome. Tools such as *Kameleon* [27] take an *imperative* approach to build images, as they execute a pipeline of scripts to create images that contain the desired environment. Such images are not enough to achieve reproducibility, as scripts must often be executed after the image has been deployed to set up relations between nodes (*e.g.,* to mount a custom NFS server among the deployed nodes). These scripts are often written manually with bash or Python, and sometimes use helper libraries such as *Execo* [18]. They may also be only valid for a single target platform and are often fragile and difficult to maintain. One solution to avoid this scripted configuration phase would be to encapsulate part of it in the images themselves.

The development phase of the images for a distributed environment is an iterative and time-consuming process. Reducing the duration of the development cycles goes towards improving the user experience, and we think that this is very important to reduce the friction to adopt reproducible experimental practices. Tools such as *EnOSlib* [7] or Vagrant [31] go in this direction as they enable users to describe the configuration of the environment while abstracting the target platform. *EnOSlib* takes an *imperative* approach by defining the environment with a pipeline of scripts that describes the provisioning phase and is executed at the beginning of an experiment to set up the desired software environment. The target abstraction makes it easier to test the environment locally before deploying it at full scale. However, reproducibility of the software stack is not a focus for these solutions, and they mostly just inherit reproducibility properties of the underlying platform and technologies. Typically, one probably needs to combine *EnOSlib* with tools such as *Kameleon* to make sure the kernel is reproducible, and the end user would be responsible for the reproducibility of the provisioning scripts.

Projects such as *NixOps* [22], *deploy-rs* [28] or *Disnix(OS)* [3, 4] enable to activate new *NixOS* configuration on target machines, but they are limited to machines that already run *NixOS*. These technologies only change the running configuration on the machines without rebooting them, which keeps a state on the machines and is detrimental for reproducibility.

C. Research Gap, Positioning

We believe that there is a necessity to propose a solution to **deploy reproducible environments** for a distributed system with **fast development cycles**. Figure 1 summarizes the motivation that has led us to create *NixOS Compose*. To generate reproducible environments with the current solutions, users most often need a configuration file for every platform they target. *NixOS Compose* aims at having a **single description** of the distributed environment that can be deployed to **several platforms**. *NixOS Compose* relies on *Nix* and *NixOS* and thus inherits their properties to make the environment **completely reproducible**. We fully embraced the *descriptive* approach for reproducibility reasons and decided to do most of the configuration at the image build time. Some part of the configuration must however still be done at runtime via a

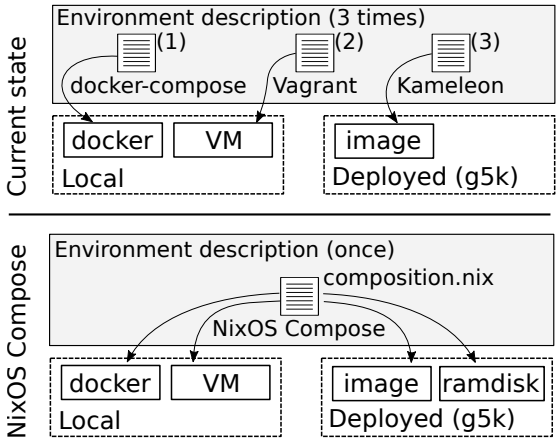

Fig. 1. Motivation of *NixOS Compose*. Currently, to produce a reproducible environment for each platform, users must maintain a description file for each target platform. We want *NixOS Compose* to only use a **single description** file (called a composition) that can build reproducible distributed environments and deploy them to **several platforms**.

provisioning phase, typically to synchronize services between different machines.

III. Presentation of *NixOS Compose*

This section gives the main concepts and terminology of *NixOS Compose*. A **software environment** is a set of applications, libraries, configurations and services. We define the **notion of *Transposition*** as the capacity to deploy a *uniquely defined environment* on several platforms of different natures. For example one may want to deploy an environment on local virtual machines to test and develop, and then want to deploy onto real machines on a distributed platform with the *exact same description*.

A. Concepts of Nix and NixOS

This section presents concepts from *Nix* and *NixOS* that are required for a better understanding of the *NixOS Compose* tool.

A *Nix* package is defined by a **function** taking as input the sources and dependencies, as well as how to construct it, and returning the package. *Nix* fetches or rebuilds the dependencies and then executes *in isolation* the commands to build the package. The packages are then stored and isolated in a special directory called the *Nix Store*. The *Nix Store* is read-only, which by design makes impossible the alteration of packages after the build. Figure 2 summarizes the differences between traditional package manager and *Nix*. A hash code of a package inputs (sources, build script, *Nix* expression) are associated with each package. This enables the cohabitation of multiple versions of the same package on the same machine.

A ***Nix* system profile** defines the configuration of the system (packages, `initrd`, etc.). Among many features, a profile can define filesystems such as NFS and mount them automatically at boot time. Figure 3 depicts an example of user profile containing the Batsim application [14], which requires the SimGrid [6] library at runtime. A *NixOS* image can contain

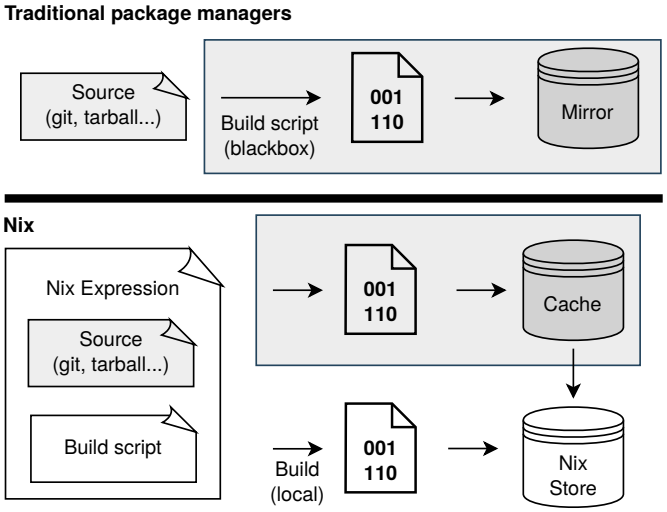

Fig. 2. Comparison between traditional package managers and *Nix*. Traditional package managers fetch a built version of the package from a mirror, but information on how they have been built is unknown. In the case of *Nix*, the package is described as a *Nix* function that takes as input the source code and how to build it. If the package with these inputs has already been built and is available in the *Nix* caches (equivalents of mirrors) it is simply downloaded to the *Nix Store*. Otherwise, it is built locally and added to the *Nix Store*.

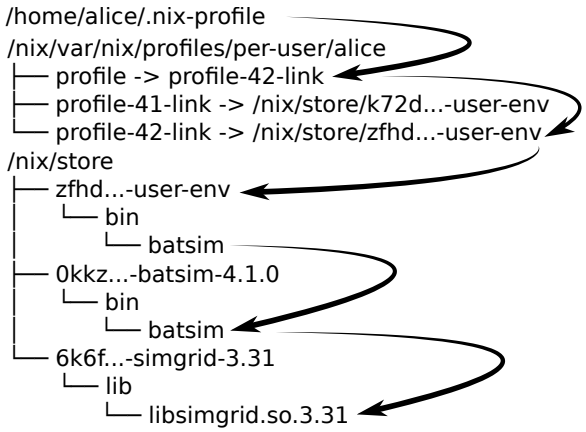

Fig. 3. Figuration of the *Nix Store* content when the `alice` user has installed a Batsim [14] binary in her profile. As Batsim requires the SimGrid [6] library at runtime, SimGrid must also be in the store. Packages are stored in their own subdirectory, but common dependencies are not duplicated as symbolic links and shared libraries are used.

several profiles and *Nix* can switch between them by modifying symbolic links and restarting services via `systemd`.

B. Concepts of NixOS Compose

NixOS Compose is based on *Nix* and *NixOS* to apply the notion of *Transposition* to distributed systems. As depicted on Figure 1, it enables users to have a **single definition** of their environment and to **deploy it to different platforms**. For the sake of clarity, *NixOS Compose* concepts will be illustrated on an example environment that contains *k3s* [19], a lightweight version of Kubernetes for the orchestration of containers.

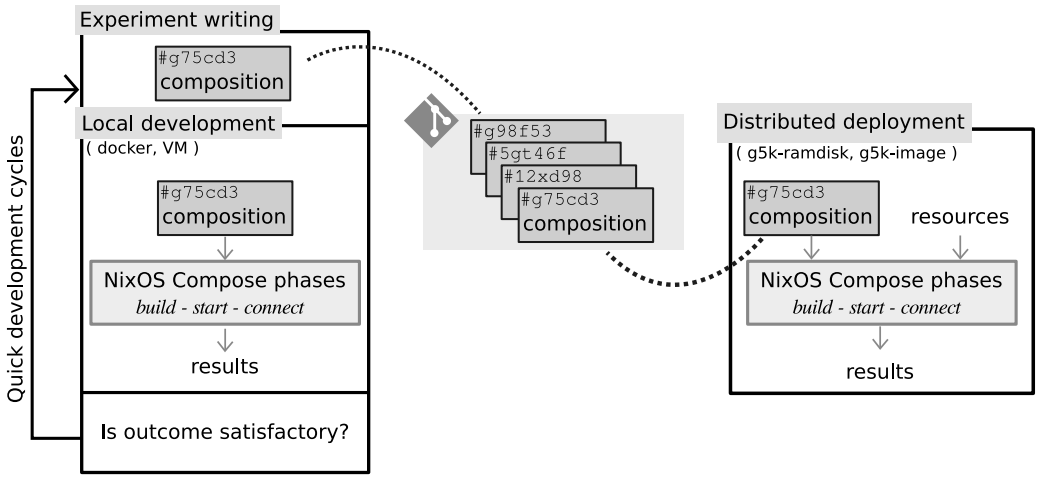

Fig. 4. Workflow of *NixOS Compose*. Local development of the environment is done using light and fast development with containers and virtual machines. Once the description of the environment (composition) has been tested, deployments on a distributed platform can be done with the **exact same interface**.

```
1  { pkgs, ... }:
2  let k3sToken = "df54383b5659b9280aa1e73e60ef78fc";
3  in {
4    nodes = {
5      # Configuration of the server
6      server = { pkgs, ... }: {
7        environment.systemPackages = with pkgs; [
8          k3s gzip # List of the packages to include
9        ];
10       networking.firewall.allowedTCPPorts = [
11         6443 # Ports to open
12       ];
13       services.k3s = {
14         # Definition of the k3s service
15         enable = true;
16         role = "server";
17         package = pkgs.k3s;
18         extraFlags = "--agent-token ${k3sToken}";
19       };
20     };
21     # Configuration of the agent
22     agent = { pkgs, ... }: {
23       environment.systemPackages = with pkgs; [
24         k3s gzip
25       ];
26       services.k3s = {
27         enable = true;
28         role = "agent";
29         serverAddr = "https://server:6443";
30         token = k3sToken;
31       };
32     };
33   };
34 }
```

Listing 1. *NixOS Compose* composition file example for *k3s* [19].

1) Role: A **role** is a type of configuration associated with the mission of a node. *k3s* is a client-server application where clients are named *agents*. There would therefore be 2 roles: one for the `server` and another for the `agents`. As all the agents have the same configuration they can use the same role. Note that in cases where there is one node per role, the notion of role and the notion of node overlap.

2) Composition: A **composition** is a *Nix* expression describing the *NixOS* configuration of every role in the environment. Listing 1 shows an example of composition for *k3s*. Please note that the composition in Listing 1 contains the `nodes` keyword on line 4 to define the various roles (instead of the `roles` keyword). This is done on purpose to make *NixOS Compose*'s syntax retro-compatible with the syntax used in *NixOS tests*[1] (these tests are part of *NixOS* and enable to start virtual machines with QEMU with the defined environment, and execute a Python script to test the application inside the environment). The composition in Listing 1 defines the open port of the `server` (line 10-12), the available packages (lines 7-9 and 23-25, packages are `k3s` and `gzip`), as well as the `systemd` services (lines 13-19 and 26-31). The `k3s` service is not defined explicitly in this example, as we reuse an existing definition that is available in the collection of *Nix* expressions called *nixpkgs*[2]. For personal applications users might have to define their own `systemd` services, which can be done declaratively via a *Nix* expression. *Nix* variables can be used to avoid information duplication, as seen for the *k3s* token (used to manage authentication) defined on line 2 and used to configure both the `server` (line 18) and the `agents` (line 30).

3) Deployment: A **deployment** assigns a role to every node. *NixOS Compose* proposes two ways to define deployment as YAML files, as illustrated on Listings 2 and 3. The first way is to define the number of nodes that should be used for each role (Listing 2). This is convenient when working on homogeneous nodes for simple deployments. For the sake of reproducibility, *NixOS Compose* generates a deterministic assignment – if the same nodes are reserved and the same deployment is used, the role of each node remains the same. The second way to define a deployment is to directly define the role that each node should take (Listing 3). This is more suited for complex scenarios, as it enables users to generate their deployment file depending on their needs.

[1]https://nixos.org/guides/integration-testing-using-virtual-machines.html
[2]https://github.com/NixOS/nixpkgs

```
# Users can define the number of nodes per role
server: 1
agent: 3
```

Listing 2. Deployment file example for the *k3s* example with 1 server and 3 agents where the users defined the quantity of nodes per role. The hostnames are generated by *NixOS Compose*. In this example there would be 3 agents with the hostnames `agent1`, `agent2` and `agent3`.

```
# Users can also define the hostnames per role
server: 1
agent:
    - agent1
    - agent2
    - agent3
```

Listing 3. Deployment file example for the *k3s* example with 1 server and 3 agents where the users defined the hostnames of every node per role.

4) Flavours: A **flavour** is a target for the deployment of the environment. This notion includes the (virtual or physical) platform onto which the deployment should be done, and also the *deployment method* that should be used (*e.g.,* full system image or ramdisk). As we write these lines *NixOS Compose* supports the following flavours:

- `docker` for `docker-compose` [24] configurations.
- `vm-ramdisk` for in-memory QEMU virtual machines.
- `g5k-image` for full system tarball images that can be deployed on *Grid'5000* [1] via *Kadeploy* [17].
- `g5k-ramdisk` for `initrd`s that can be quickly deployed in memory without the need to reboot the host machine on *Grid'5000* (via the `kexec` syscall).

During the development phase of the environment, users can deploy *locally, lightly and quickly* with the `docker` and `vm-ramdisk` flavours. At a later stage, users can test their environment on real nodes from the *Grid'5000* testbed with the `g5k-ramdisk`, which is convenient for trial-and-error operations thanks to its fast boot time. Finally, the environment can be deployed at real scale on *Grid'5000* with the `g5k-image` flavour. Please note that some flavours have reproducibility limitations due to the underlying technologies. For example, controlling the version of the Linux kernel is impossible when using the `docker` flavour.

C. Workflow of NixOS Compose

This section presents the workflow of *NixOS Compose* and how it enables users to simply transpose their environment from one platform to another.

1) Local Testing: When developing an environment, users can work with the `docker` and `vm-ramdisk` flavours with the following workflow:

1. Building the image: `nxc build -f docker` or `nxc build -f vm-ramdisk`
2. Deploy the environment: `nxc start`. By default, *NixOS Compose* takes the last composition built.
3. Connect to the nodes: `nxc connect [node name]`. This opens a connection to the desired node. If name is omitted, a terminal multiplexer[3] opens with one pane per node, which is convenient to run commands on the nodes.

[3] `tmux`: https://github.com/tmux/tmux

2) Distributed Deployment: Once the environment has been tested with local flavours, it can be tested in a distributed system.

1. Building the image: `nxc build -f g5k-ramdisk` or `nxc build -f g5k-image`
2. Reservation of the nodes to use for the deployment: depends on your platform resource manager. For example `salloc` for *Slurm* [32] or `oarsub` for *OAR* [5].
3. Deploy the environment: `nxc start`.
4. Connect to the nodes: `nxc connect [node name]`.

Figure 4 summarizes the *NixOS Compose* workflow. *NixOS Compose* aims at making the transition between platforms as seamless as possible. Thus, the workflow in a distributed setting (Section III-C2) **is identical** to the workflow in a local one (Section III-C1). The only difference is that in a distributed setting, users need to first reserve the resources before deploying.

IV. Technical Details of *NixOS Compose*

Nix can generate a *NixOS* configuration from a *Nix* expression, including the `boot` and `init` phases required to start the kernel. *Nix* stores those phases in the *Nix Store*, which enables *NixOS Compose* to call them later on. *NixOS Compose* works in two steps: *Building* and *Deployment*. The building phase is done using *Nix* tools wrapped with Python for the command-line interface. The deployment is fully done with Python and mostly consists in the interaction with the different deployment tools (*Kadeploy*, *docker-compose*, *QEMU*). The *Nix* part is around 2000 lines of code, and the Python part around 4000.

The following section (IV-A) details how *NixOS Compose* manages `g5k-ramdisk`, the flavour that enables quick in-memory deployment without the need to reboot host machines on *Grid'5000*. Details are omitted for the other flavours, but please refer to Table I for a summary of the difference in the building and deployment phases for all supported flavours.

A. Details on the `g5k-ramdisk` Flavour

1) Construction: NixOS Compose uses *Nix* to evaluate the configuration of every role in the composition. *Nix* then generates the kernel and the `initrd` of the profiles.

2) Deployment: NixOS Compose relies on the `kexec` Linux system call for this flavour. `kexec` enables to boot a new kernel from the currently running one. This skips the initialization of the hardware usually done by the BIOS, which avoids an entire reboot of the machines and greatly reduces the time to boot the new kernel. The `kexec` commands takes as input the desired kernel, the kernel parameters and the `initrd`. *NixOS Compose* passes the kernel and `initrd` generated in the construction phase to `kexec`.

As *NixOS Compose* produces a single image containing the profiles of all the roles, *NixOS Compose* needs at deployment time to tell each node the role it should take. To achieve this, we pass this information using the kernel parameters to set up environment variables based on the role. *NixOS Compose* also uses the kernel parameters to pass ssh keys and information about the other hosts (*e.g.,* the `/etc/hosts`).

TABLE I
TABLE SUMMARIZING THE DIFFERENT FLAVOURS WITH THEIR BUILDING AND DEPLOYMENT PHASES.

Flavour	Phase		Comments
	Building	Deployment	
`docker`	Generate a `docker-compose` configuration and `docker` containers.	Call the `docker-compose` application with the right arguments.	Fastest and light but limited in application due to virtualization.
`vm-ramdisk`	Generate the kernel and `initrd` for the roles of the composition.	Create a virtual network with Virtual Distributed Ethernet (VDE) and starts the Virtual Machines with QEMU.	Fast but takes a lot of memory. Limited to a couple of VMs on a laptop.
`g5k-ramdisk`	Generate the kernel and `initrd` for the roles of the composition.	Use `kexec` to quicky start the new kernel without rebooting. Send the deployment information through the kernel parameters.	Long to build but fast to deploy. `kexec` has reproducibility limitations and consumes a lot of memory which can be limiting for large images.
`g5k-image`	Generate a tarball of the image of the composition.	Use *Kadeploy* to deploy the image to the nodes. Send the deployment information through the kernel parameters.	Longer to build and deploy, but it has the best reproducibility properties.

There is however a size limit of 4096 bytes on the kernel parameters, which prevents us to use this method to send the deployment information to nodes when users want to deploy a lot of nodes. To deal with this, *NixOS Compose* starts a light HTTP server on the cluster frontend for the duration of the deployment. We pass the URL of this server using the kernel parameters. Then, the nodes query this server with `wget` to retrieve the information associated with their roles. Note that the deployed images do not include a HTTP server but only the `wget` application to fetch the data. Figure 5 represents how the nodes get the deployment information based on the quantity of nodes involved.

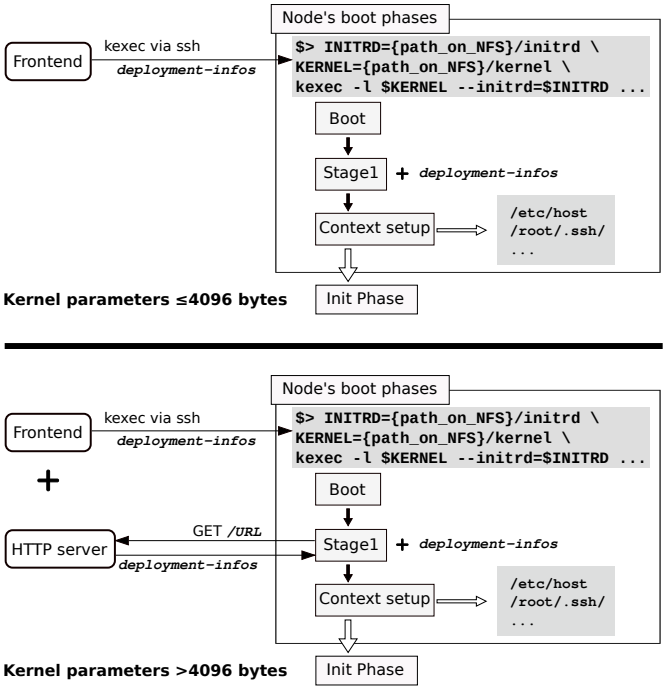

Fig. 5. Mechanism for the nodes to get the deployment information. For a small number of nodes, the information is passed via the kernel parameters. For a higher number of nodes, this is not possible due to the size limit on the kernel parameters (4096 bytes). In this case *NixOS Compose* starts a light HTTP server on the frontend and passes its URL to the nodes via the kernel parameters. The nodes then query this server to retrieve the deployment information.

V. A COMPLEX EXAMPLE: MELISSA

This section shows how complex distributed environments can be developed with *NixOS Compose* by taking the *Melissa* [30] framework as an example.

A. *Presentation of* Melissa

Melissa is a framework to run large-scale sensitivity analyses. Its main specificity is the online processing of data to limit usage of intermediate file storage, contrary to postmortem approaches. It can be used in several HPC environments as it is compatible with two resource managers (RM): Slurm [32] and OAR [5]. *Melissa* implements a client/server model where the clients are simulations that generate and send data to a server that runs the statistics algorithms.

NixOS Compose enables to deploy a resource manager (including all the components it requires, *e.g.,* a database), *Melissa*, and all the components required by *Melissa* at runtime (*e.g.,* a distributed file system).

In the following example we deploy *Melissa* with the Slurm resource manager. Four roles are needed to define the environment.

- `server`: RM server and file system server
- `dbd`: MariaDB database (accounting for the RM)
- `computeNode`: worker node
- `frontend`: node from where initial jobs are submitted

```
server = { pkgs, ... }: {
  imports = [ slurmconfig nfsConfigs.server ];
  services.slurm.server.enable = true;
  systemd.services.slurmctld.serviceConfig = {
    Restart = "on-failure";
    RestartSec = 3;
  };
};
computeNode = { pkgs, ... }: {
  imports = [ slurmconfig nfsConfigs.client ];
  environment.systemPackages = [
    melissa melissa-heat-pde
  ];
  services.slurm.client.enable = true;
  systemd.services.slurmd.serviceConfig = {
    Restart = "on-failure";
    RestartSec = 3;
  };
};
```

Listing 4. Configuration of the `server` and `computeNode` roles.

Some roles share parts of their configuration, like `server` and `computeNode`. They both use Slurm but their configuration differs in terms of services, as they respectively enable the `slurm.server` and `slurm.client` services.

Melissa itself also needs to be part of the environment. Unlike the *k3s* example shown on Listing 1, *Melissa* is not available in *nixpkgs* and thus needs to be packaged. Listing 5 is a snippet of *Melissa*'s package definition that notably defines which source code should be used (lines 5-9) and which build dependencies should be used (lines 10-13). The build commands (line 14) are omitted for the sake of readability.

```
1  { pkgs, ... }:
2  pkgs.stdenv.mkDerivation rec {
3    pname = "melissa-${version}";
4    version = "0.7.1";
5    src = pkgs.fetchgit {
6      url="https://gitlab.inria.fr/melissa/melissa";
7      rev="e6d09...";
8      sha256="sha256-IiJad...";
9    };
10   buildInputs = with pkgs; [
11     cmake gfortran python3 openmpi
12     zeromq pkg-config libsodium
13   ];
14   # Build phases are omitted
15 }
```

Listing 5. Snippet of the package definition for *Melissa*

Similarly, the `melissa-heat-pde` simulation application must also be in the environment (line 12 of Listing 4). Finally, as a distributed filesystem is necessary in this environment, our composition imports a NFS module for the roles that need it.

B. Key difficulties

This section emphasizes the advantages of using *NixOS Compose* to deploy the *Melissa* distributed environment.

1) NFS Server: Setting up a NFS server with tools like *Kameleon* or *EnOSlib* is cumbersome. The users would first need to install the `nfs` tools on the nodes and define the NFS server. Then, the users would have to mount the newly defined server on every client node. These steps can be automated with scripts but that would be fragile.

```
# Configuration of the NFS server
nfsServer = {
  services.nfs.server.enable = true;
  services.nfs.server.exports =
    "/srv/shared *(rw,no_subtree_check,fsid=0,
    no_root_squash)";
  services.nfs.server.createMountPoints = true;
};
```

Listing 6. Definition of the NFS server. It exposes the `/srv/shared` folder.

```
# Mouting of the NFS server
nfsClient = {
  fileSystems = {
    "/data" = {
      device = "server:/";
      fsType = "nfs";
    };
  };
};
```

Listing 7. Mounting of the NFS server for the compute nodes. The local mounting point is `/data`.

The declarative definition of NFS with *NixOS* is based on `systemd` services (see Listings 6 and 7). This makes them easier to define and more robust as they can be restarted until they perform the mount succesfully in the case of the NFS server starting after the clients.

2) Resource Manager: *Melissa* runs on production clusters managed by resource managers such as Slurm and OAR. However, experimenting on *Melissa*'s behavior in different scenarios requires controlling the resource manager part of the environment. The installation of such systems is far from trivial as they include several distributed services that must interact, and each one of these services require configuration. A composition can be made modular so that users can descriptively change the resource manager. The same benefit is achievable for comparison studies of versions of the *Melissa* framework, as the *Nix* function that defines *Melissa* can be written in such a way that it takes *Melissa*'s source code as a function input.

We have deployed the *Melissa* composition we have written with *NixOS Compose* on 13 nodes with the experimental setup described in Section VI-A. The deployment took approximately 2 minutes with the `g5k-ramdisk` flavour.

C. Melissa *Images Content Comparison*

NixOS Compose aims to provide the same environment on different target platforms. This section analyses the content of the *Nix Store* in the *Melissa* images generated for every flavour, as the *Nix Store* content represents the software environment available on the node. The `docker` flavour is omitted here as the containers do not have a well-defined specific *Nix Store* but mount the *Nix Store* of the host machine instead.

Figure 6 presents the content of the *Nix Store* of the *Melissa* image for the different flavours. The smaller packages are gathered under the `others-*` name. We can see that the vast majority of the software stack is shared by several flavours. There is a common 2 GiB set of packages common to every flavour containing the *NixOS* definition and the dependencies of *Melissa*. The two flavours targeting the *Grid'5000* platform need more packages, for example the

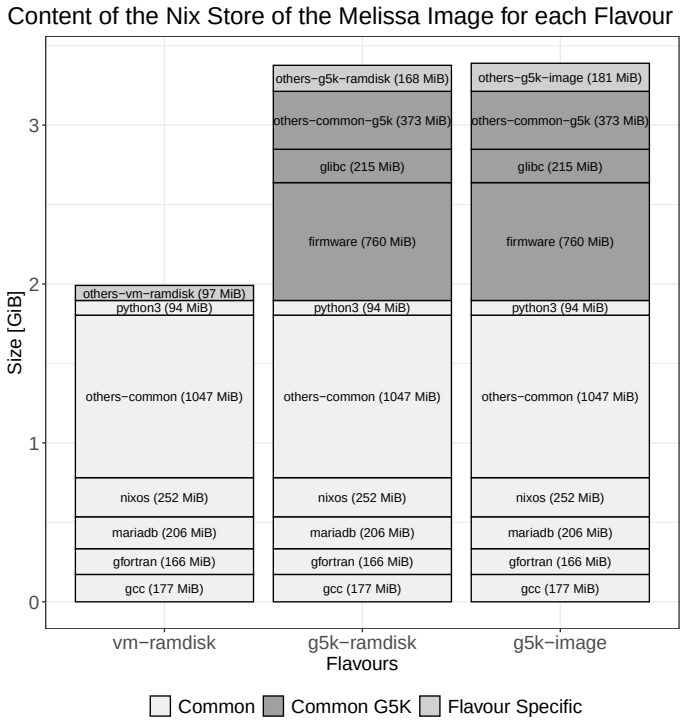

Fig. 6. Packages present in the *Nix Store* of the *Melissa* image for the different flavours. The colors represent the packages common to the flavours. The smaller packages are gathered under the `others-*` name. The `docker` flavour is omitted as it mounts the *Nix Store* of the host machine.

firmware to use the nodes' hardware. Then for each of the flavour, there is about 5 % of the total *Nix Store* size for packages specific to the flavour. For example, deploying a `g5k-image` image requires a complete reboot of the host and to go through the boot loader, hence the presence of `grub` in this image.

VI. Evaluation

NixOS Compose brings reproducibility guarantees to distributed environments contrary to state-of-the-art solutions. The overall goal of this evaluation section is to determine whether this is done with a significant overhead or not.

A. Experimental Setup

The following experiments have been carried out on the `dahu` cluster of the *Grid'5000* testbed. This cluster has machines with 2 Intel Xeon Gold 6130 CPUs with 16 cores per CPU and 192 Gib of memory. The nodes of this cluster have SSD SATA Samsung MZ7KM240HMHQ0D3 disks with a capacity of 240 GB formatted in ext4.

The experiments conducted in this article are repeatable with variation. Data and analysis scripts are available on Zenodo[4] with the link to the experiments repository.

[4]https://zenodo.org/record/6568218

B. Comparison to Kameleon

Grid'5000 provides base images for several Linux distributions and versions. Users need to build their own images if they want to use more complete images. This is usually done with *Kameleon* on *Grid'5000* — in fact all the images provided by *Grid'5000* are generated by *Kameleon* recipes.

In this study, we want to compare the performance of *NixOS Compose* and *Kameleon* to build images. We will focus on the image build time, as well as the size of the generated images. We also want to evaluate whether caching the *Nix Store* enables an interesting build time speedup.

1) Protocol: The following steps are executed in this order.

1. Construction. Build an image from a recipe.
2. Modification. Change the recipe slightly.
3. Reconstruction. Build an image from the new recipe.

We first build a `base` image with *NixOS Compose* and *Kameleon*, measuring its build time and the size of the generated images. `base` contains the basic software needed to conduct a distributed experiment: `grid5000/debian11-x64-nfs` for *Kameleon* as this is the most convenient and common image for distributed experiments on *Grid'5000*, and all the packages required by the flavour for *NixOS Compose*. Then we add the `hello` package to the recipes and build a new image (`base + hello`) while measuring the same metrics.

This experiment has been executed using *Grid'5000*'s NFS (mounted on `/home`) or without it (using local disks on machines mounted on `/tmp`), in order to compare the performance of the tools depending on the filesystem setup used.

We clear the *Nix Store* before building the `base` image, but not before building `base + hello`, in order to evaluate the impact of cached builds on *NixOS Compose*. *Kameleon* has an indirect caching mechanism via the HTTP proxy Polipo [26]. However, we did not manage to make it work with *Kameleon*, and Polipo is no longer maintained as its utility has become arguable – most of today's traffic is encrypted, including fetching packages from mirrors.

2) Results and Comments: As seen on Figure 7, *NixOS Compose* substantially outperforms *Kameleon* in terms of image build time. When building from an empty cache on local disks, *NixOS Compose* is 11x faster than *Kameleon*. Moreover, *NixOS Compose* uses its local cache efficiently, which enables it to build the image variation 1.7x faster than the initial image build time when the filesystem is used efficiently (local disks).

Figure 7 also shows that *NixOS Compose* produces bigger images than *Kameleon*. This is mainly because we have not optimized the content size of the images as we write these lines — *e.g.,* many firmwares are kept in the images instead of only the ones needed on *Grid'5000* (see Figure 6). Another reason for this image size comes from our design choice to prioritize compression speed over compression quality. This is important for *NixOS Compose* as image variations should be built as fast as possible to improve user experience. For information, we have measured that *NixOS Compose* takes

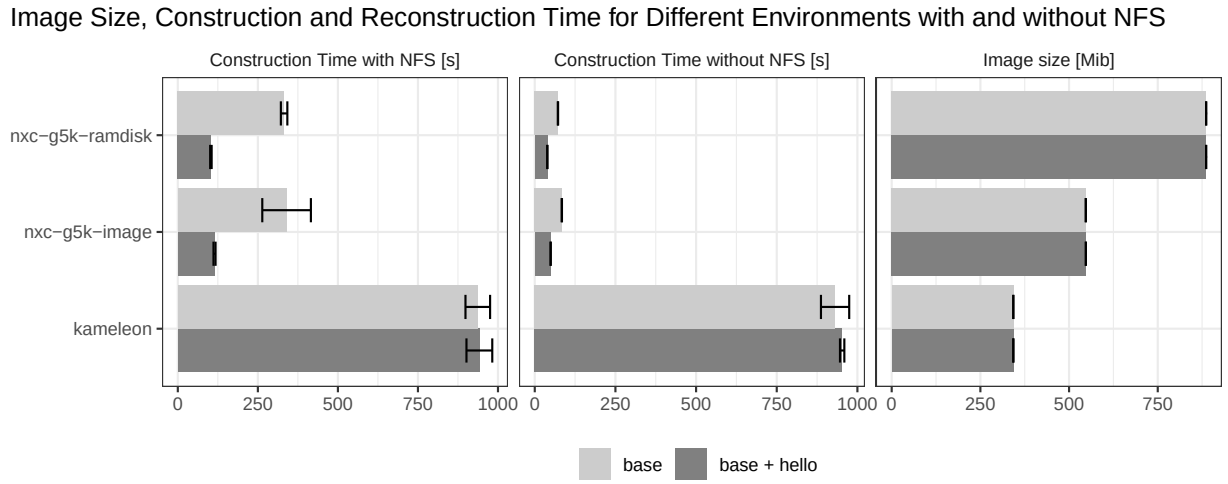

Fig. 7. Performance comparison between *Kameleon* and *NixOS Compose* (`nxc`). A `base` image is first built, then a new image (`base + hello`) that contains the additional `hello` package is built. As *Grid'5000* is the targeted platform of this experiment, images are built with the `g5k-ramdisk` and `g5k-image` flavours. Shown values are averages over 5 repetitions. Error bars represent 99 % confidence intervals.

about 25 s to compress each image, which is a non-negligible portion of a variation build time (30 - 35 %).

Finally, Figure 7 shows how the filesystem setup impacts the build time. Here, using an efficient filesystem setup (local disks) greatly benefits to *NixOS Compose* as it makes it 4x faster. This is caused by the many smalls writes done by *Nix* in the *Nix Store*. Filesystem setup has little impact on *Kameleon* as it uses local disks by default to generate the whole image, that is later on copied to the NFS.

C. Comparison to EnOSlib

EnOSlib is a state-of-the-art solution to conduct distributed experiments. *EnOSlib* does not claim to be reproducible, but it inherits the reproducibility of its underlying components. In this section, our goal is to compare the performance of *NixOS Compose* and *EnOSlib* to set up *fully reproducible* distributed environments. In particular, we want to know how much time is taken for each phase of a deployment.

1) Case Studies: We chose to study the two following distributed applications that are already implemented in *EnOSlib*.

- *k3s* [19] is a lightweight version of Kubernetes.
- *flent* [23] is a network benchmarking tool.

Our *k3s* environment consists in two nodes: one *k3s* server and one *k3s* agent. The agent deploys a `nginx` web server to the agents. The *run* part of the experiment simply consists in querying the webserver to retrieve the `nginx` web page.

Our *flent* environment consists in 2 nodes: one server and one client. The *run* part of the experiment simply runs the *flent* benchmark.

For both case studies, we made sure that *NixOS Compose* and *EnOSlib* set up similar environments, and that they execute the same *run* part of the experiment after the deployment and provisioning phases have been done.

2) Protocol: EnOSlib's approach is mostly based on the provisioning phase, which executes commands to set up a desired environment (software, services…) on a machine that is already available. As this limits reproducibility on the kernel side, we decided to deploy reproducible images on the nodes. We used the `grid5000/debian11-x64-nfs` image that is pre-built by *Grid'5000*. *EnOSlib* deploys the image with *Kadeploy*, then executes the provisioning phase, and then execute the *run* part of the experiment once the environment is ready.

In *NixOS Compose* most of the configuration is done inside the composition and thus in the image. This enables us to completely skip the provisioning phase for *flent*. For *k3s*, our provisioning phase simply consists in waiting that the web server becomes available.

For both tools we measure the time to build the image, to deploy it, to execute the provisioning and to run the experiment script. We use an empty *Nix Store* for a fair build time.

3) Results and Comments: As seen on Figure 8, *NixOS Compose*'s `g5k-ramdisk` flavour is faster to deploy than a full image as it uses `kexec` and does not require a full reboot. As expected, we also can see that the solutions with *NixOS Compose* have a smaller provisioning time than with *EnOSlib*. This is because *NixOS Compose* includes this as part of this provisioning in the build and deployment phase.

Please note that *EnOSlib* does not directly provide the time taken by the deployment phase, but only provides the time between the submission and the first command of the provisioning. That is why the submission and deployment time (Sub + Deploy) are shown together on Figure 8 both for *EnOSlib* and *NixOS Compose*, for the sake of fairness.

From Figure 8, it seems that packing part of the provisioning in the image improves the provisioning time without deteriorating the deployment time. The only drawbacks are the non-negligible build times. However, those times can be improved

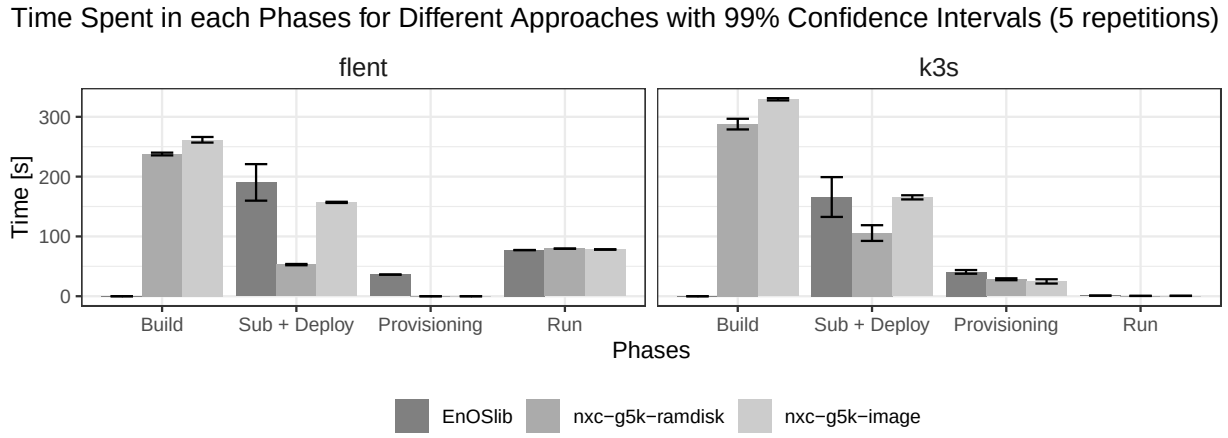

Fig. 8. Time spent in the different phases of the deployment of a distributed experiment (build, submission + deploy, provisioning, run). We compare *EnOSlib* and *NixOS Compose* (with the flavours `g5k-ramdisk` and `g5k-image`) on two examples: a network benchmarking tool (*flent*) and a containers' orchestrator (*k3s*). The errors bars represent the confidence intervals at 99 %.

by utilizing the *Nix Store* as a local cache (see Section VI-B). Note that *NixOS Compose* proposes several flavours that can be executed locally to develop and test the environment. The cost of the construction of these images is thus amortized by the numerous quick and light local deployments. Finally, please also note that the build time of *EnOSlib* is null on Figure 8 as a pre-built image is used. However, depending on their scenario users may need to actually build an image via another technology (*e.g., Kameleon* or *NixOS Compose*).

VII. Conclusion and Future Work

This article has presented *NixOS Compose*, a tool that enables the generation of reproducible distributed environments. We have showed that *NixOS Compose* deploys the exact same software stack on various platforms of different natures, without requiring specific work from users. The software stack is reconstructible by design, as *NixOS Compose* inherits its reproducibility properties from *Nix* and *NixOS*. Our experiments showed that *NixOS Compose*'s reproducibility and platform versatility properties are achieved without deployment performance overhead in comparison to the existing solutions *Kameleon* and *EnOSlib*.

NixOS Compose is a free software released under the MIT license. As we write these lines, we think that its maturity level enables to deploy the distributed environment of many experiments. However, *NixOS Compose* currently only provides first-class support for *Grid'5000*. We would like to support baremetal and virtualized deployments on other experimental testbeds such as CloudLab [13] and Chameleon [20].

NixOS Compose enables to build and deploy reproducible distributed environments. This is crucial for conducting reproducible distributed experiments, but this is only a part of the bigger picture. We plan to explore how *NixOS Compose* can be coupled to other tools that solve other parts of this problem. *EnOSlib* is for example well-suited to control the dynamic part of complex distributed experiments but lacks reproducibility properties, which makes us think that a well-designed coupling may be beneficial for practitioners.

The experiments conducted in this article showed that build caches greatly improves *NixOS Compose*'s build times, and that properly using the filesystem is important for its performance. From a cluster administration perspective, providing a shared *Nix Store* between users would be very interesting to avoid data duplication and to prevent different *NixOS Compose* users to build the same packages over and over. There are many ways to implement a distributed shared *Nix Store* and we think that exploring their trade-offs would provide valuable insights, as reproducibility improvements should not be done at the cost of a higher resource waste on clusters.

User experience is a crucial factor that must be considered for reproducible experimental practices to become the standard. With this in mind, we think that the notion of *Transposition* we have defined in this article and implemented in *NixOS Compose* is very beneficial. *Transposition* reduces the development time of distributed environments, as it enables users to do most of the trial-and-error parts of this iterative process with fast cycles, without any reproducibility penalty on real-scale deployments. However, practitioners that adopt *NixOS Compose* are likely to experience a paradigm shift if they are not already accustomed to *Nix*'s approach. We strongly believe that the reproducibility and serenity gains it brings are worth it.

Acknowledgment

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

This research was partly supported by the EuroHPC EU Regale project (g.a. 956560).

The authors would like to thank Matthieu Simonin for the helpful discussion on *EnOSlib*.

References

[1] Daniel Balouek et al. "Adding Virtualization Capabilities to the Grid'5000 Testbed". In: *Cloud Computing and Services Science*. Ed. by Ivan I. Ivanov et al. Vol. 367. Communications in Computer and Information Science. Springer International Publishing, 2013, pp. 3–20. ISBN: 978-3-319-04518-4. DOI: 10.1007/978-3-319-04519-1_1.

[2] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX annual technical conference, FREENIX Track*. Vol. 41. 46. Califor-nia, USA. 2005, pp. 10–5555.

[3] Sander van der Burg. *Disnix*. original-date: 2013-06-24T12:44:47Z. Apr. 2022. URL: https://github.com/svanderburg/disnix (visited on 04/28/2022).

[4] Sander van der Burg. *DisnixOS*. original-date: 2013-06-24T14:20:16Z. Mar. 2022. URL: https://github.com/svanderburg/disnixos (visited on 04/28/2022).

[5] N. Capit et al. "A batch scheduler with high level components". en. In: *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.* Cardiff, Wales, UK: IEEE, 2005, 776–783 Vol. 2. ISBN: 978-0-7803-9074-4. DOI: 10.1109/CCGRID.2005.1558641. URL: http://ieeexplore.ieee.org/document/1558641/ (visited on 05/25/2020).

[6] Henri Casanova et al. "Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms". In: *Journal of Parallel and Distributed Computing* 74.10 (June 2014), pp. 2899–2917. URL: http://hal.inria.fr/hal-01017319.

[7] Ronan-Alexandre Cherrueau et al. "EnosLib: A Library for Experiment-Driven Research in Distributed Computing". en. In: *IEEE Transactions on Parallel and Distributed Systems* 33.6 (June 2022), pp. 1464–1477. ISSN: 1045-9219, 1558-2183, 2161-9883. DOI: 10.1109/TPDS.2021.3111159. URL: https://ieeexplore.ieee.org/document/9534688/ (visited on 11/22/2021).

[8] Christian Collberg, Todd Proebsting, and Alex M Warren. "Repeatability and Benefaction in Computer Systems Research - A Study and a Modest Proposal". en. In: (2015), p. 68.

[9] Ludovic Courtes and Ricardo Wurmus. "Reproducible and User-Controlled Software Environments in HPC with Guix". en. In: *Euro-Par 2015: Parallel Processing Workshops*. Ed. by Sascha Hunold et al. Vol. 9523. Cham: Springer International Publishing, 2015, pp. 579–591. ISBN: 978-3-319-27307-5 978-3-319-27308-2. DOI: 10.1007/978-3-319-27308-2_47. URL: http://link.springer.com/10.1007/978-3-319-27308-2_47 (visited on 06/13/2020).

[10] Ludovic Courtès. "Functional Package Management with Guix". en. In: *arXiv:1305.4584 [cs]* (May 2013). URL: http://arxiv.org/abs/1305.4584 (visited on 06/13/2020).

[11] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. "Nix: A Safe and Policy-Free System for Software Deployment". en. In: (2004), p. 14.

[12] Eelco Dolstra and Andres Löh. "NixOS: A Purely Functional Linux Distribution". In: *SIGPLAN Not.* 43.9 (Sept. 2008), pp. 367–378. ISSN: 0362-1340. DOI: 10.1145/1411203.1411255. URL: https://doi.org/10.1145/1411203.1411255.

[13] Dmitry Duplyakin et al. "The Design and Operation of CloudLab". In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. July 2019, pp. 1–14. URL: https://www.flux.utah.edu/paper/duplyakin-atc19.

[14] Pierre-François Dutot et al. "Batsim: a Realistic Language-Independent Resources and Jobs Management Systems Simulator". In: *20th Workshop on Job Scheduling Strategies for Parallel Processing*. Chicago, United States, May 2016. URL: https://hal.archives-ouvertes.fr/hal-01333471.

[15] Dror G. Feitelson. "From Repeatability to Reproducibility and Corroboration". en. In: *ACM SIGOPS Operating Systems Review* 49.1 (Jan. 2015), pp. 3–11. ISSN: 0163-5980. DOI: 10.1145/2723872.2723875. URL: https://dl.acm.org/doi/10.1145/2723872.2723875 (visited on 05/21/2020).

[16] Todd Gamblin et al. "The Spack package manager: bringing order to HPC software chaos". en. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Austin Texas: ACM, Nov. 2015, pp. 1–12. ISBN: 978-1-4503-3723-6. DOI: 10.1145/2807591.2807623. URL: https://dl.acm.org/doi/10.1145/2807591.2807623 (visited on 11/22/2021).

[17] Yiannis Georgiou et al. "A tool for environment deployment in clusters and light grids". In: *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE. 2006, 8–pp.

[18] Matthieu Imbert et al. "Using the EXECO toolbox to perform automatic and reproducible cloud experiments". In: *1st International Workshop on UsiNg and building ClOud Testbeds (UNICO, collocated with IEEE CloudCom 2013*. Bristol, United Kingdom: IEEE, Dec. 2013. DOI: 10.1109/CloudCom.2013.119. URL: https://hal.inria.fr/hal-00861886.

[19] *K3s: Lightweight Kubernetes*. URL: https://k3s.io/ (visited on 04/28/2022).

[20] Kate Keahey et al. "Lessons Learned from the Chameleon Testbed". In: *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[21] Michael Mercier, Adrien Faure, and Olivier Richard. "Considering the Development Workflow to Achieve Reproducibility with Variation". In: *SC 2018-Workshop: ResCuE-HPC*. 2018, pp. 1–5.

[22] *NixOps*. original-date: 2011-10-24T15:49:58Z. Apr. 2022. URL: https://github.com/NixOS/nixops (visited on 04/28/2022).

[23] *Overview — Flent: The FLExible Network Tester*. URL: https://flent.org/ (visited on 05/08/2022).

[24] *Overview of Docker Compose*. en. May 2022. URL: https://docs.docker.com/compose/ (visited on 05/17/2022).

[25] *Packer by HashiCorp*. en. URL: https://www.packer.io/ (visited on 05/13/2022).

[26] *Polipo — a caching web proxy*. URL: https://www.irif.fr/~jch/software/polipo/ (visited on 05/13/2022).

[27] Cristian Ruiz et al. "Reconstructable Software Appliances with Kameleon". en. In: *ACM SIGOPS Operating Systems Review* 49.1 (Jan. 2015), pp. 80–89. ISSN: 0163-5980. DOI: 10.1145/2723872.2723883. URL: https://dl.acm.org/doi/10.1145/2723872.2723883 (visited on 06/12/2020).

[28] *serokell/deploy-rs*. original-date: 2020-09-28T17:46:15Z. Apr. 2022. URL: https://github.com/serokell/deploy-rs (visited on 04/28/2022).

[29] *Software Heritage*. en-US. URL: https://www.softwareheritage.org/ (visited on 05/17/2022).

[30] Théophile Terraz et al. "Melissa: Large Scale In Transit Sensitivity Analysis Avoiding Intermediate Files". In: *The International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing)*. Denver, United States, Nov. 2017, pp. 1–14. URL: https://hal.inria.fr/hal-01607479.

[31] *Vagrant by HashiCorp*. en. URL: https://www.vagrantup.com/ (visited on 05/13/2022).

[32] Andy B Yoo, Morris A Jette, and Mark Grondona. "Slurm: Simple linux utility for resource management". In: *Workshop on job scheduling strategies for parallel processing*. Springer. 2003, pp. 44–60.