Reconstructable Software Appliances with *Kameleon*

Cristian Ruiz, Salem Harrache, Michael Mercier and Olivier Richard
firstname.lastname@inria.fr
INRIA/Univ. of Grenoble, Grenoble, France

ABSTRACT

A *software appliance builder* bundles together an application with its needed middleware and an operating system to allow easy deployment on Infrastructure as a Service (IaaS) providers. These builders have the potential to address a key need in our community: the ability to reproduce an experiment. This paper reports the experiences on developing a software appliance builder called *Kameleon* that leverages popular and well tested tools. *Kameleon* simplifies the creation of complex software appliances that are targeted at research on operating systems, HPC and distributed computing. It does so by proposing a highly modular description format that encourages collaboration and reuse of procedures. Moreover, it provides debugging mechanisms for improving experimenter's productivity. To justify that our appliance builder stands above others, we compare it with the most known tools used by developers and researchers to automate the construction of software environments for virtual machines and IaaS infrastructures.

Categories and Subject Descriptors

D.2.4 [**Distributed Systems**]: Miscellaneous

Keywords

Reproducible Research, Testbed, Virtual Appliances, Cloud Computing, Experiment Methodology.

1. INTRODUCTION

Large testbed infrastructures for experimentation in networks and large scale systems such as Grid'5000 [3], FutureGrid [9], etc. are available, which enable the deployment of complex software stacks either on bare metal or using an IaaS provider. These infrastructures' high degree of software stack customizability appeal to researchers who want to test their ideas in real settings. However, the management of these software stacks is not always trivial, their setup is a tedious and time consuming task that should be automated whenever possible. The lack of automation can be attributed to the low expertise, lack of the proper tools and the long learning path for researchers. The lack of automation leads to the inability to reproduce an experiment, since it is not even possible to build or set the experimental setup under the exact same conditions where an experiment took place. A recent study [5], where the buildability of artifacts was evaluated, found that only 24% of publications in ACM conferences and journals can be built. To preserve the experimental setup some works are relying on *software appliances* technology. Industry has been using *software appliances* for provisioning software in a more resilient way, they have developed *software appliance builders* which automate the construction of software appliances.

1.1 Motivations

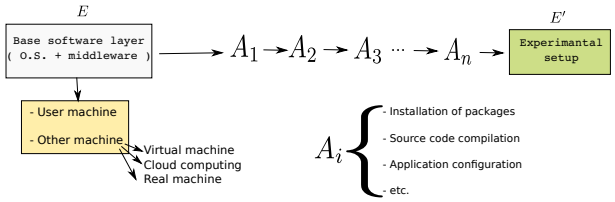

Figure 1: Creation process of an experimental setup.

Figure 1 illustrates the process to derive an experimental setup. Experimenters start from a base setup which includes an operating system plus a middleware. This base setup could be located in the same machine of the experimenter, in a virtual machine, in an IaaS provider as Amazon EC2[1], OpenStack[2], etc; or in a real machine that belongs to a computing cluster. The experimenter will apply a sequence of actions $\langle A_i \rangle$ which consists in, for instance: installation of software packages, source code compilation, software configuration, etc. Applying these actions $\langle A_i \rangle$ produce an experimental setup E', which is then used for the evaluation of a given implementation, algorithm, etc. Due to space limitations in research papers the composition of E' is not properly described, nor are the sequence of actions $\langle A_i \rangle$ that were taken to derive E'. In domains such as High Performance Computing, Distributed Systems and Operating Systems research, experimental setup configuration, which includes the operating system, version of libraries and compilers, compilation flags, etc, are crucial requirements to be able to repeat an experiment [4].

1.2 Reconstructability

To improve experimentation, we claim that an experimenter needs to know the exact process that led to the creation of a particular experimental setup, E', as well as to be able to replay and modify this process to arrive at the same and alternative experimental setups. We introduce the concept of reconstructability of an experimental setup

[1] `http://aws.amazon.com/ec2/`
[2] `http://www.openstack.org/`

to formally capture this process. An experimental setup E' is reconstructable if the following three facts hold:

- Experimenters have access to the original base experimental setup E.
- Experimenters know exactly the sequence of actions $\langle A_1, A_2, A_3, ..., A_n \rangle$ that produced E'.
- Experimenters are able to change some action A_i and successfully re-construct an experimental setup E''.

Reconstructability can be expressed functionally as $E' = f(E, \langle A_i \rangle)$, where f applies $\langle A_i \rangle$ to E to derive the experimental setup E'. Thus, if reconstructability holds, we are guaranteed to be able to derive E' no matter when $\langle A_i \rangle$ is applied to E. Reconstructability does not hold when:

- An action A_i is composed of sub-tasks that are executed concurrently making the process not deterministic. For example: compilation of software using `Makefiles` with the option `-j` that runs parallel compilation process. This provokes compilation rules to run in any order if they are not connected by dependencies.
- Packages with the latest release of Debian (*Debian 8*) have a time of expiration. Therefore, old packages can not be installed.

Reconstructability also does not hold when either the base setup, E, or the specific software used in an action, A_i, is no longer available. The availability of software becomes an issue when reconstructability depends on package managers and configuration management tools [6]. For example, there is no guarantee that a git repository which is used by an action will be available at a later point in time.

1.3 Contributions

This paper identifies the necessary ingredients for a software appliance builder to be a viable solution for the preservation and packaging of experimental setups. The contributions of this paper and what makes it different from our two previous publications [8, 13] are:

1. In Section 1.2, we introduced the concept of reconstructability, which identifies the process to build an experimental setup so that the setup can be rebuilt and can be built with variations.
2. In Section 3, we evaluate existing software appliance builders against the criteria needed to improve user productivity.
3. In Section 4, we refine the *Kameleon* syntax and concepts, and we extend the persistent cache mechanism so that it supports new concepts.
4. In Section 5, we demonstrate that *Kameleon* is modular, enables the reuse of code, and builds on proven technology.
5. Section 5.2, we identify the container requirements for different types of software appliances.

The rest of this paper is structured as follows: Section 2 presents related work. Section 3 presents a qualitative comparison of the most widely used software appliance builders. Section 4 presents a complete description of *Kameleon* architecture, concepts and features. Section 5 presents use cases that validate our approach. Section 6 presents future work. Section 7 concludes.

2. RELATED WORK

Re-running an experiment with the original software artifacts could be achieved by using virtual appliances and virtual machine snapshots [10, 7]. Brammer et. al [2] present a system to create executable papers, which relies on the use of virtual machines and aims at improving the interactions between authors, reviewers and readers with reproducilibity purposes. *Kameleon* differs in that it allows the re-execution of an experiment with the original software artifacts and the ability to modify the experimental setup cleanly and easily.

Widely used tools such as Vagrant[3], provide reproducible environments for development. Vagrant uses pre-built images which hinders understanding of the operating system layer and makes modifications to this layer difficult. *Kameleon* differs in that the construction of the operating system layer is part of the software appliance generation. This fact makes its recipes less complex than the recipes used by popular configuration management tools such as Puppet[4] and Chef[5].

From the traceability point of view, *Kameleon* can be compared to interactive notebooks such as IPython[6] where the goal is to track every step that leads to a given result. *Kameleon* keeps a trace of all the steps that led to the creation of a given software stack, it does so by providing a structured, modular and understandable language. *Kameleon* makes reconstructability of software appliances possible, experimenters are able to explore all the actions, modify and repeat the environment generation.

In Section 3.3, we discuss software appliance builders.

3. COMPARISON

We describe and evaluate the most widely used software appliance builders in cloud infrastructures and development environments. The evaluation uses as criteria: 1) how well they support the software appliance build cycle and 2) whether they meet the criteria for improving experimenters' productivity to build an experimental setup.

3.1 Software Appliance Build Cycle

All the analyzed tools follow the same pattern in the process of building a software appliance. The tool takes as input a *Description File* that details all the requirements that the software appliance should meet. Then, it initializes a *Container*. A container is the environment that it is used for building the software appliance. This term container encompasses: system level virtualization techniques (e.g., chroot, openVZ, Linux Containers), full virtualization technologies (e.g., VirtualBox, KVM, Xen, VMware) and real physical machines. Once the container is initialized, the tool parses the description and starts to carry out the *bootstrap*, *setup* and *export* procedures. The output of this process is a software appliance formatted for the infrastructure that will finally host it. The main steps in the software appliance build cycle are explained below:

- **Bootstrap**: This refers to the process of getting a bootable operating system. This bootable image can be either built from scratch or it can be retrieved from some external source. The normal procedure is to get an ISO image from the target operating system and

[3] `http://www.vagrantup.com/`
[4] `http://puppetlabs.com/`
[5] `https://www.getchef.com/chef/`
[6] `http://ipython.org/notebook.html`

follow the installation procedure. Another option is to download and load a software appliance already created.

- **Setup**: In this step, users apply several procedures to customize the base system and make it meet their needs. These procedures include mainly the installation and configuration of software. There are many possible ways to customize, by using shell scripts or configuration management tools such as Salt, Chef, Puppet, Ansible, etc.

- **Export**: This step creates the final format for the software appliance. The final format ranges form the available virtual disk formats (e.g., VDI[7], VMDK[8] ,QCOW2[9]) to more simple formats based on *tarballs*[10].

3.2 Criteria for Improving User Productivity

The evaluation is driven by the question: *What makes an experimenter more productive when building a complex software appliance?* The following criteria will be used for the evaluation:

- **Easiness**: The tool has a low learning curve. Specially, a low learning curve is supported by providing a simple language to describe the appliance across the different levels of the software appliance's software stack (e.g., O.S. level, middleware or application).

- **Support during the build process**: Long compilation times are commonplace when building these kinds of software stacks, for instance the compilation of operating system kernels, modules, scientific libraries. Because this process is frequently error prone, a mechanism for debugging or checkpointing the process makes the experimenter more productive. Validation of the correct functioning of the software appliance is required as well.

- **Containers diversity**: The tool should support a variety of container types. This enables hassle-free transportation of an experimental setup from one infrastructure to another, because experimenters are more comfortable with working in specific environments. Additionally, it should be easy to integrate new types of containers that meet the requirements of the experimenter. For example, libraries such as ATLAS[11] which gets its speed by specializing itself for the underlying architecture, needs to be compiled on the target machine where it will finally run. Certain Linux modules need direct access to real hardware. Therefore, they could not run on virtualize systems. That is the case for Dune [1] and CControl [12].

- **Shareability**: Instructions for building a software appliance must be organized and stored in a modular way to enable the reuse of procedures and collaborate within a community.

- **Reconstructability**: One important shortcoming is the reproduciblity of experiments in computer science. It has been demonstrated that one of the causes is the impossibility to build the same software artifacts[12] used in a publication [5]. Thus a requirement is to be able to reconstruct a software appliance from its definitions, which will at the same time enable later customization as defined in Section 1.2.

3.3 Software Appliance Builders

In this section, we describe and evaluate the most widely used software appliance builders according to our criteria for improving user productivity. Table 1 shows the evaluation.

3.3.1 Docker

Docker[13] offers a powerful and lightweight way to build software appliances that are packed in Linux Containers (LXC). Docker manages and tracks changes and dependencies, making it easier for users to understand how the final appliance was built. It relies on repositories for enabling users to share their artifacts with other collaborators. The most appealing feature of Docker is that it makes applications portable across many infrastructures. As a downside, however, applications are run inside Linux Containers which could be not suitable for certain uses (e.g., run an application that uses *cgroups*[14]). The description of the building process is done using a simple syntax based on few constructs that help customize the containers.

3.3.2 Packer

Packer[15] helps users to create identical software appliances targeted at multiple platforms. The process is composed of: builders, responsible for creating machines and generating images from them for various platforms; provisioners, used to install and configure software (many options are available from simple shell scripts to high-end configuration management tools) and postprocessors, that help manage the final produced image. Packer supports a variety of container types and it strives to make descriptions portable across different containers. Thus the burden of changing from one development environment to another is reduced. However, a different language is used to describe the operating system layer, which makes difficult to add modifications to this layer. Additionally, the tool do not provide any mechanism for organizing the instructions which hampers *shareability*.

3.3.3 BoxGrinder

BoxGrinder[16] creates appliances from simple plain text descriptions for various platforms. It utilizes the host system to perform the image creation using the *guestfs*[17] library which results in a faster process. Then, the newly created software appliance can be exported locally to be used for a virtualization technology or it can be moved outside to be used in IaaS providers. Software appliance descriptions are simple and easy to understand and can be composed for reuse. BoxGrinder does not offer any mechanism for supporting the build process and it is tied to build the software

[7] https://www.virtualbox.org/manual/ch05.html
[8] http://www.vmware.com/app/vmdk/?src=vmdk
[9] http://www.linux-kvm.org/page/Qcow2
[10] It refers to a computer file format that can combine multiple files into a single file.
[11] http://math-atlas.sourceforge.net/
[12] It refers to source code compiled for testing.
[13] https://www.docker.io/
[14] https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt
[15] http://www.packer.io/
[16] http://boxgrinder.org/
[17] http://libguestfs.org/

Table 1: Comparison of widely used appliance builders based on criteria that would make an experimenter more productive.

Tool	**Kameleon**	**Docker**	**Packer**	**BoxGrinder**	**Veewee**	**Oz**
Easiness	Yes	Yes	No	Yes	No	No
Support in the building process	Yes	Yes	Yes	No	No	No
Container diversity	Yes	No	Yes	No	Yes	No
Shareability	Yes	Yes	No	Yes	No	No
Reconstructability	Yes	Yes	No	No	No	Yes

appliance using the host system which could be problematic when some isolation is needed.

3.3.4 *Veewee*

Veewee[18] is a tool for automating the creation of custom virtual machine images. It is able to interact with several virtual machine hypervisors. It offers to the user the possibility of validating the generated software appliance through the execution of behavioral tests. The capacities of the tool for customizing a software appliance are very limited. Description files are written in Ruby restricting the interaction with shell scripts.

3.3.5 *OZ*

Oz[19] was created to ease the automatic installation of operating systems. It uses QEMU as a container and uses the native operating system tools to install software. The cycle of building a software appliance includes the generation of metadata about the packages installed. Software appliances are created using an XML-based language. Even though the language allows almost any operation of customization, the descriptions rapidly become complex and difficult to maintain.

3.3.6 *Kameleon*

Kameleon achieves easiness by proposing a structured language based on few constructs and which relies on shell commands. The hierarchical structure of recipes and the extend mechanism allow shareability. *Kameleon* supports the build process by providing debugging mechanisms such as interactive shell sessions, break-points and checkpointing. Containers diversity is achieved by allowing the easy integration of new containers using the same language for the recipes. Furthermore, persistent cache makes possible reconstructability. In Section 4, we present *Kameleon* in detailed.

3.4 Discussion

We found that many software appliance builders rely on archive files (e.g. ISO images) to bootstrap a software appliance. However, if the archive files is no longer available in a repository, then reconstructability is impossible. We found that 30% of Veewee definition files[20] point to repositories that either no longer exist or have some packages missing. Furthermore, management of containers is implemented either in the core of the tool or as plugins. This makes integration of new containers for non-advanced users difficult. Most of the tools support a wide variety of containers, however, because they are tied to virtualization, real hardware is not taken into account. Shareability which implies modularity and collaboration is not available. Docker is the only tool, at the moment, which implements a collaborative model for building software appliances. These tools do not support debugging or check pointing in the build process.

[18] `https://github.com/jedi4ever/veewee`

[19] `http://www.aeolusproject.org/oz.html`

[20] This was tested with the version of veewee 0.3.7 by trying to build all templates during the period of 02/12/2013 and 20/12/2013.

Finally, the way tools support the build cycle has an important impact on the recontructability given that some actions would be out of the user's control. When the language used in the tool's *Description file* is based on less human-readable languages, such as XML, or on complex recipes, such as the ones used by Chef and Puppet, that tool ranks lower in the easiness criteria.

4. KAMELEON APPLIANCE BUILDER

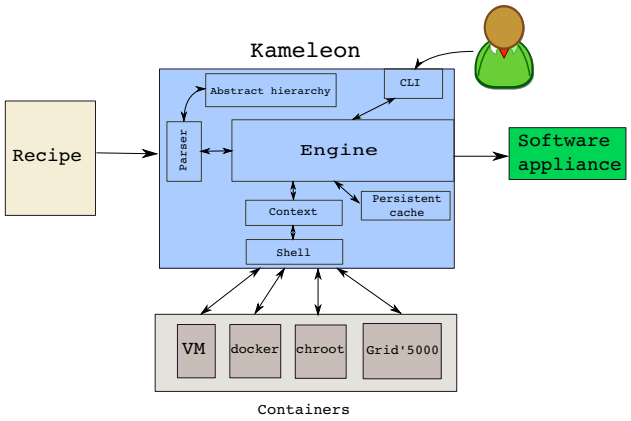

Figure 2: Kameleon architecture.

Kameleon is a small and flexible software appliance builder, which eases the construction and reconstruction of custom software stacks for research in HPC, Grid or Cloud computing and Distributed Systems. *Kameleon* version 2.2.4 is written in 2278 lines of Ruby[21] and has few dependencies. *Kameleon* achieves ease of use by structuring the specification (recipes) for the construction of software appliances into a hierarchy. The hierarchy's structure is composed of sections that allow a separation of customization and low level tasks. This structure separates out the customization tasks that can be easily performed by non-expert users from the low level tasks, such as setting up a complete operating system or exporting the whole file system, which are more difficult. These sections are divided into steps that represent actions $\langle A_i \rangle$ such as: installation and configuration of a certain scientific library, kernel patching, configuration of a base system. Steps are composed of microsteps that enable the customization and re-utilization of the same step in different recipes. Finally, the last level of the hierarchy wraps shell commands and *Kameleon* defined commands. All the aforementioned hierarchy is written using YAML, which encourages more human readable shell scripts[22].

An advantage of *Kameleon*, and what distinguished it from the existing appliance builders, is that it serves simply as a recipe parser and orchestrator of shell commands, which means that all the logic for the creation of a software appliance resides entirely in the recipes. *Kameleon* recipes enable four advantages for experimenters: 1) it helps to understand how the software appliance was created (all the

[21] Measured with SLOCCount `http://www.dwheeler.com/sloccount/`

[22] `http://yaml.org/spec/1.2/spec.pdf`

details are embedded in the same language); 2) it gives a total control over the whole process, which reduces the burden of integrating new containers, new operating systems, or new export formats; 3) it enables the easy customization of software appliances at any level (e.g. O.S., middleware, applications, etc.); 4) it encourages a collaboration model where researchers can reuse code and given that all details are in the hierarchy of recipes and steps (text files) they can be easily versioned.

Figure 2 shows the architecture of the system and the interaction between the different modules. First, the parser, with the help of the abstract hierarchy, parses the recipe and creates as output the internal data structures that are input to the engine module. The engine orchestrates the workflow of execution. The workflow is executed sequentially. The context module helps to abstract the access to a given container. All the low level operations (e.g., execution of shell commands, I/O and file management) are performed by the shell module. The engine integrates three important mechanism for debugging: checkpoints, breakpoints and interactive shell sessions. The persistent cache captures all the data used during the process of building a software appliance, which is archived to allow the software appliance to be reconstructed at a later time. Finally, the CLI module implements the user interface.

4.1 Syntax

Figure 3 shows an example of a *Kameleon* recipe. We can highlight three different elements: sections, steps and variables. Four sections are proposed by *Kameleon* but more can be created. One section, called `global`, is dedicated to the declaration of global variables that can be used through out the recipe. The other sections correspond to the main steps in the software appliance build cycle (bootstrap, setup and export). Different sections in a *Kameleon* recipe allow a high degree of customizability, reuse of code, and total control of software appliance creation process by the experimenter. In Figure 3, the based system is built from scratch using the package manager of the Debian distribution as specified in the `bootstrap` section.

Alternatively, it is possible to use existing images (e.g., Grid'5000 base environments, cloud images for different Linux distributions, or software appliances market places[23]). The `setup` section installs packages, configures the O.S., etc. Within a section, users can execute shell commands, read and write files, or perform other commands that are necessary to carry out the desired customization. The options in the `export` section depend on the disk formats that the container supports. At the moment we have implemented recipes for exporting to the most popular virtual disk formats, tarballs and specific Grid'5000 format.

Listing 1 shows the definition of a step file. Each step file is loaded automatically by *Kameleon* after parsing the recipe. A step is divided into microsteps (e.g., `create_group`) which are in turn divided into commands. The goal of dividing steps into microsteps is the possibility of activating certain actions within a step. For example, from Listing 1 we have the possibility of executing only the microstep `create_group` without executing the rest of the microsteps. There are two types of variables: user defined variables that are provided in the recipe such as: Linux distribution (`distrib`), architecture (`kernel_arch`), etc., and *Kameleon* variables such as `$$kameleon_cwd` (*Kameleon* work directory) that interact with the engine. Contexts are mapped to special variables (`out_context` and `in_context`) in the global section. They indicate the necessary actions to set a shell in the respective context (the concept of context is explained in the next section). In the example, the recipe creates a Debian Wheezy appliance with some base configuration, which is specified as the `distrib` and `release` variables in the global section, and exports the appliance in QCOW2 format, which is specified in the export section as the step "- `save_as-qcow2`". The *Kameleon* recipe illustrates that sections are composed of steps that can be customized using variables. Table 2 illustrates `exec_*` commands, which are the minimal building blocks of microsteps. An `exec_*` command wraps a shell command to add error handling and interactiveness in case of a problem.

[23] `http://www.turnkeylinux.org`

```
global:
  ## User varibales : used by the recipe
  user_name: kameleon
  user_password: $$user_name
  # Distribution
  distrib: debian
  release: wheezy
  kernel_arch: $$arch
  hostname: kameleon-$$distrib
  ## Disk options
  nbd_device: /dev/nbd1
  image_disk: $$kameleon_cwd/base_$$kameleon_recipe_name.qcow2
  image_size: 10G
  filesystem_type: ext4
  # rootfs options
  rootfs: $$kameleon_cwd/rootfs

out_context:
  cmd: bash
  workdir: $$kameleon_cwd
  proxy_cache: 127.0.0.1

in_context:
  cmd: USER=root chroot $$kameleon_cwd/rootfs bash
  workdir: /root/kameleon_workdir
  proxy_cache: 127.0.0.1

bootstrap:
  - initialize_disk_chroot
  - debootstrap:
    - repository: http://ftp.debian.org/debian/
  - start_chroot

setup:
  - install_software:
    - packages: >
      debian-keyring sudo less vim acpid linux-image-$$kernel_arch
  - configure_kernel
  - install_bootloader
  - configure_network
  - create_group:
    - name: admin
  - create_user:
    - name: $$user_name
    - groups: sudo admin
    - password: $$user_password
export:
  - qemu_save_appliance:
    - input: $$image_disk
    - output: $$kameleon_cwd/$$kameleon_recipe_name
    - save_as_qcow2
# - save_as_vdi
```


Figure 3: In the example, the section headers illustrate contexts (`out_context` and `in_context`), declarations (`global`) and sections (`bootstrap`, `setup` and `export`). This example uses a chroot jail as a container for building a software appliance based on Debian Wheezy.

4.2 Kameleon Contexts

By dividing the building process into independent parts, contexts provide a way for a user to structure the software appliance creation process so that it is independent from

```
# Create User
- create_group:
  - exec_in: groupadd $$group

- add_user:
  - exec_in: useradd --create-home -s /bin/bash $$name
  - exec_in: adduser $$name $$group
  - exec_in: echo -n '$$name:$$password' | chpasswd
  - on_export_init:
    - exec_in: chown '$$user_name:' -R /home/$$user_name

- add_group_to_sudoers:
  - append_in:
    - /etc/sudoers
    - |
      %admin ALL=(ALL:ALL) ALL
```

Listing 1: Example of a step file. The prefix '$$' is used for variables.

Exec: executes a command in a given context	`- exec_in: echo "Hello!" > hello.txt` `- exec_in: apt-get -y update`

Pipe: it works as Unix pipelines but between contexts	`- pipe:`
	`  - exec_out: cat tlm_code.tar`
	`  - exec_in: cat > ./tlm_code.tar`

Write: allows to write a file in a context	`- write_in:`
	`  - /root/.ssh/config`
	`  - |`
	`  Host *`
	`  StrictHostKeyChecking no`
	`  UserKnownHostsFile=/dev/null`

Hooks: defers some initialization or clean actions.	`- on_setup_clean:` `  - exec_in: rm -rf /tmp/mytemp`

Table 2: *Kameleon* commands.

the final target platform. When an appliance is built with *Kameleon* it is necessary to deal with 3 different contexts (more can be defined if required). The objective of all these contexts is to have a contextualized shell session. Contexts are as follows:

- *Local context*: It refers to the location where *Kameleon* is executed. Normally, it is the user's machine.
- *OUT context*: It is where the process of bootstraping will take place. Some procedures have to be carried out in order to create the place where the software appliance is built (IN context). This could be: the same user's machine using *chroot*. Thus, this context is where the setup of the chroot takes place. Other examples of *OUT context* are: setting up a virtual machine, access to an infrastructure in order to get an instance and be able to deploy, setting up a Docker container. This context also allows the appliance's base file system layout to be setup.
- *IN context*: It makes reference to inside the container created by the *OUT context*. This context can be mapped to a *chroot*, virtual machine, physical machine, Linux container, etc. This context is frequently used for customizing the software appliance.

The relation between the possible contexts used and the section execution is shown in Table 3.

Section	Context used	Description
Bootstrap	Local context and OUT context	Two possibilities: (1) build a file system layout form scratch. (2) start form an already created software appliance.
Setup	Mostly IN context	The commands run on the chosen container: chroot, Docker, Linux container, virtual machine and real machine
Export	Local context and OUT context	Use of the container supported tools for creating the final format for the software appliance.

Table 3: *Kameleon* concepts, interrelation between contexts and sections.

4.3 Checkpoint Mechanism

The construction of a software appliance is a trial and error process. *Kameleon* provides a modular checkpoint mechanism that saves time when debugging the software appliance construction process. Time consuming tasks such as the installation of an operating system from scratch are not repeated during the debugging process. Thus, a checkpoint mechanism encourages the automation of software appliance building as it makes the construction of software appliances less time consuming. We have integrated different checkpointing mechanisms for each container supported by *Kameleon*. They are based on snapshots of virtual machines (QEMU, VirtualBox) and based on snapshots of *QCOW2* disk images for the chroot container. Another checkpoint mechanism use *Docker* commits to preserve the state of a Docker image.

4.4 Extend Mechanism

```
extend: qemu/debian7.yaml

global:

bootstrap:
  - "@base"
setup:
  - "@base"
  - install_software:
    - packages: g++ make openssh openmpi build-essential fort77
  - install_atlas:
    - repository: http://sourceforge.net/math-atlas/Stable/
    - version: "3.10.1"
  - install_hpl:
    - repository: "http://www.netlib.org/benchmark/hpl/"
    - version: "2.1"
    - hpl_makefile: "$$kameleon_recipe_dir/data/Make.Linux"

export:
  - "@base"
```

Listing 2: Extend mechanism.

Listing 2 shows a *Kameleon* recipe that builds a software appliance for the *hpl* benchmark. This recipe adds steps to the setup section and reuse steps from the recipe shown in Figure 3. This is done by using the `extend:` and `"@base"` keywords. Recipes are provided as templates, which enable a user to write a new recipe based on another existing recipe by overwriting certain sections and variables. The main purpose of this mechanism is to reduce the entry barrier for non-expert users by encouraging the reuse of recipes. This allows *Kameleon*'s users to take advantage from the recipes already developed by the community.

4.5 Persistent Cache Mechanism

The persistent cache is the mechanism *Kameleon* uses to enable reconstructability and preservation of environments for experimentation. Our persistent cache captures all the data and instructions (*recipe* and *step* files) used during the construction of a software appliance. Data is captured in two ways: 1) Polipo[24] a web proxy cache for caching all the packages coming from the network, and 2) custom procedures for caching data coming from other sources. This mechanism is detailed in [13]. The persistent cache archive is structured by step (*Kameleon* hierarchy) and it contains files, control version repositories and mainly cache files generated by Polipo. A hash is associated with both a step file and its generated persistent cache directory. This association enables *Kameleon* to ensure the coherency between instructions and data used to build a software appliance. *Kameleon* persistent cache mechanism enables any software appliance to be rebuilt from its persistent cache. The only requirement is that the software appliance has to be built successfully at least once.

5. USE CASES

In this section, we demonstrate how *Kameleon* was used to build different software appliances. These software appliances illustrate a variety of software stacks (Table 4) with different requirements. Specially, they are taken from different domains (high performance computing, operating system and distributed system); they use different container technologies (chroot, Docker, VirtualBox, QEMU and real machine in Grid'5000); and they use different container isolation (lightweight, service, kernel module, and hardware dependent).

5.1 Software Appliance Complexity

We start by describing different basic software appliances that can be used as a base experimental environment. Then we describe more complex software appliances used in research papers.

- **Basic software appliances**: These software appliances include several Linux flavors, for example: Fedora, CentOS, Debian, Archlinux. Different configurations were built from the very basic console mode to the complete desktop configuration. This shows that complete computer environments for researchers can be built.

- **Complex software appliances**: These software appliances were used in different research papers: an application for controlling cache utilization [12], a safe user-level access to privileged CPU features [1], a formal specification of a JavaScript module system [11]. Other appliances provide widely used computing frameworks such as *MapReduce*[25], benckmarks such as *hpl*[26] and batch schedulers such as *OAR*[27]

5.2 Container isolation

Because software appliances require different levels of isolation at build time, a software appliance builder needs to provide isolation mechanisms. *Kameleon* provides isolation with its notion of context. Below are examples of the isolation requirements by different types of software appliances.

5.2.1 Lightweight.

Lightweight software appliances do not need any kind of isolation, thus they can run inside a *chroot*. This kind of software appliances can be exported to any format and run in any infrastructure. Examples of lightweight software appliances include: MPI + TLM[28] (electromagnetic simulation code), Map Reduce framework. Formal Java [11], *hpl* benchmark, Debian Wheezy basic system.

5.2.2 Service.

Service software appliances run a service (e.g. databases). Since the appliance's service may conflict with services running on the build machine, *Kameleon* allows the experimenter to use container isolation to isolate appliance services from build machine services.

5.2.3 Kernel modules.

When the installation of a kernel module is part of the software appliance creation, isolation at the level of operating system calls is needed, because the target kernel has to be running. Therefore, the IN context has to take place inside either a virtual or real machine. Sometimes a real machine is required, for example: 1) installation of CControl library for cache coloring[29], 2) installation of Dune[30], a kernel module that provides ordinary user programs with safe and efficient access to privileged CPU features, which are normally hidden when using a virtual machine.

5.2.4 Hardware dependent.

In contrast to the previous types of software appliances, which can be built and deployed on different machines, a hardware dependent software appliance must be built and deployed on the same machine. An example of hardware dependent software appliance is the *hpl* benchmark. This benchmark is based on the linear algebra library ATLAS, which must be optimized at built time for the deployment machine.

5.3 Results and Discussion

Table 5 shows the building time of some of the software appliances described above. The purpose of this data is to show the different steps that compose the build process and the time using various container technologies. For experimenters the process of generating an experimental environment could be perceived as a time consuming process. However, we observe that the built time of each of the software appliances is less than 30 minutes, which could encourage users to generate their custom experimental setups.

5.3.1 Hardware dependent software appliance evaluation

In this section, we use the *hpl* benchmark to evaluate hardware dependence container isolation. *hpl* benchmark requires the installation of multiple software packages whose parameters need to be configured, for performance, to the

[24] http://www.pps.univ-paris-diderot.fr/~jch/software/polipo/
[25] https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html
[26] http://www.netlib.org/benchmark/hpl/
[27] http://oar.imag.fr
[28] http://www.petr-lorenz.com/emgine/
[29] https://github.com/perarnau/ccontrol
[30] http://dune.scs.stanford.edu/

Table 4: Software appliances built with *Kameleon*

Name	**Description**	**Software stack**	**Containers used**	**Container isolation**	**Domain**
Debian basic	Debian console mode	Debian Wheezy	chroot, Docker, VirtualBox, QEMU, Grid'5000	Lightweight	Operating systems.
Debian Desktop	Debian GNOME Desktop environment	Debian Wheezy, GNOME	QEMU, VirtualBox	Service	Operating systems.
CentOS	CentOS console mode	CentOS 6.5	VirtualBox, QEMU	Lightweight	Operating systems.
Dune	Dune library which provides safe and efficient access to privileged CPU features	Ubuntu Precise, Linux headers, Git, make, GCC	Grid'5000	kernel module	Operating systems
Formal java	A JavaScript module system	Debian Wheezy, Haskell, JavaScript modules	Chroot, Docker	Lightweight	Operating systems
CControl	Kernel Module to control the amount of cache available to an application	Debian wheezy, make, Git, build tools, CControl libraries, PAPI	QEMU, VirtualBox	kernel module	High performance computing.
hpl benchmark	LinPACK benchmark	Debian Wheezy, OpenMPI, OpenSSH, C++, make, Fortran, ATLAS library, *hpl* benchmark	chroot, Docker, VirtualBox, Grid'5000	Hardware dependent	High performance computing.
Hadoop	Framework for storage and large-scale processing	Ubuntu Lucid, Python, OpenSSH, Java 6, Hadoop.	chroot	Lightweight	Distributed computing.
TLM stack	Large scale electromagnetic simulations	Debian Wheezy, OpenMPI, OpenSSH, TLM application.	chroot	Lightweight	High performance computing.
OAR	Resource and task manager for HPC clusters and other computing infrastructures.	Debian wheezy, Git, Perl, Postgresql, OAR server packages	QEMU, VirtualBox	Service	High performance computing.

Table 5: Building time of some software appliances. The time is presented in seconds (some steps have been omitted).

Steps	AP1[1]	AP2[2]	AP3[3]	AP4[4]	AP5[5]	AP6[6]	AP7[7]	AP8[8]	AP9[9]	AP10[10]
start-virtualbox					21	12	15	21	20	20
g5k-reserv			177							
start-qemu				10						
install-requirements				11	11	11	12	41	13	36
debootstrap	70	77		73		76	73		187	
yum-bootstrap								279		141
switch-context-virtualbox						10	10	105	93	32
switch-context-qemu				7						
Boostrap	**70**	**77**	**177**	**101**	**32**	**109**	**110**	**446**	**313**	**229**
install-software	25	81	339	18	15	209	22	61	38	264
configure-system	6	6		6	17	6	6	8	11	10
configure-apt	13	9	37	9		9	9		12	
install-atlas						497				
install-hpl						12				
install-ccontrol							18			
install-gnome									821	
oar-prereq-install				89						
install-lambdajs		78								
upgrade-system			212							
install-kameleon			76							
oar-git-install				53						
tlm-installation	16									
Setup	**130**	**251**	**841**	**276**	**64**	**842**	**165**	**515**	**1195**	**503**
save-qemu-appliance	83			88						
save-virtualbox-appliance					47	75	34	71	150	89
save-docker-appliance		6								
save-appliance-from-g5k			157							
Export	**83**	**6**	**157**	**88**	**47**	**75**	**34**	**71**	**150**	**89**
Total	**213**	**257**	**998**	**364**	**111**	**917**	**199**	**586**	**1345**	**592**

[1] TLM stack
[2] Formal java using docker
[3] Ubuntu using Grid'5000
[4] OAR using QEMU
[5] Archlinux using VirtualBox
[6] hpl benchmark using VirtualBox
[7] CControl using VirtualBox
[8] CentOS using VirtualBox
[9] Debian Desktop using VirtualBox
[10] Fedora minimal system using VirtualBox

Table 6: Containers comparison machine M1.

Container	Build Time[Secs]	Image Size [Mbytes]	hpl result [MFLOPS]
VirtualBox	2722	1100	3.3
QEMU	1826	1200	109.1
Docker	2293	1600	110.1
Grid'5000	1782	638	113.3

Table 7: Containers comparison machine M2.

Container	Build Time[Secs]	Image Size [Mbytes]	hpl result [MFLOPS]
VirtualBox	1004	1100	8.1
QEMU	971	1200	189.7
Docker	1066	1600	222.3

hardware that the appliance is running on. The parameter configuration requires significant compilation time. The evaluation was performed using two different machines.

- M1: Machine available in Grid'5000 in the cluster genepi. Intel Xeon E5420 QC CPU 2.5 Ghz with 8GB of RAM and HDD SATA disk.
- M2: Local machine. Intel Core i7-2760QM CPU 2.4 GHz with 8GB of RAM and SSD disk.

The machine descriptions indicate that the machines differ only in their disk technology. Table 6 shows the results for machine M1. Table 7 shows the results for machine M2. The tables illustrate the time to build the software appliance (Build Time[Secs]), the software appliance size (Image Size[MBytes]) and the time to execute the benchmark *hpl* (hpl result[MFLOPS]). In the worst case scenario, the build time never exceeds one hour (or 3,600 seconds). All the elements necessary for reproducing these results are available in our repository[31].

Additionally, both tables show the millions of floating-point operations per second (MFLOPS) obtained by deploying the generated appliance and executing the benchmark. This is illustrative for a hypothetical experiment with goal would be to evaluate for example, the performance of virtual machine monitors. From this simple experiment, we can see that the virtualization provide by VirtualBox significantly impacts hpl benchmark performance: a factor of 34 times for M1 (from 113 Mflops to 3.3) and a factor of 27 times for M2 (222.3 to 8.1). In addition, the difference in performance is minimal for the other containers on a particular machine. Finally, across machines, the difference in disk technology make a significant difference in both build and execute time.

Table 8 illustrates the correlation between the image size of a software appliance and the cache size needed to store the data used to build the appliance. We are using the image size from Table 7: building *hpl* benchmark on machine M1. Finally, the total archive space to build all three appliances is illustrated on the last row. We can observe that storage requirements is reduced in a factor of 5.

5.3.2 Experiment packaging example

[31] This paper was written using Org mode which enables to embed all the analysis presented. This is available along with persistent cache archives, *Kameleon* recipes and some additional scripts at `http://exptools.gforge.inria.fr/kameleon/`

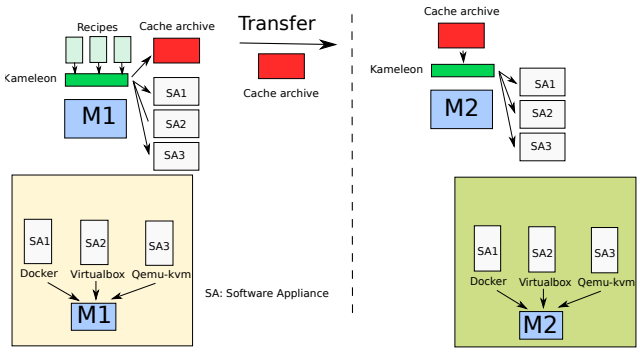

Figure 4: Example of experiment packaging with *Kameleon*.

This section demonstrates how *Kameleon* and its persistent cache allow an experimenter to evaluate the performance of a high performance application using different virtualization techniques on different machines. This section's demonstration approximates the process used in the evaluation of Section 5.3.1. This section demonstrates the advantage of using *Kameleon* and its persistent cache system through an example. Let us suppose an experimenter wants to measure the performance of different techniques of virtualization and implementations of them for the execution of high performance applications. Assume that we have run an experiment that measures execution time for two virtualization techniques: system level virtualization (Docker) and full virtualization (VirtualBox and QEMU-KVM) on a machine M1. Now, suppose a different experimenter wants to run the same experiment in another machine M2. Here are the issues they would face:

- The software appliances are rarely well described and the information of how they are configured is missing.
- Three different images have to be available which will consume space to store them and time to transfer.
- The images are static and introducing changes into them is not always easy and clean.
- Depending on the type of applications or benchmarks run in the experiment, recompilation could be needed in order to re-run the experiment in the same exact conditions. Therefore the images are not directly executable on M2.

The process using *Kameleon* is depicted in Figure 4. *Kameleon* brings the following advantages:

- All the details of composition and configuration resides on the recipes as shown in Section 4.
- In the process of generating the different software appliances, a persistent cache archive will be generated that contains all the data used during the generation of the respective software appliances. This is the only file that has to be stored and, in terms of size it is most of the time smaller than the images generated as shown in Table 8.
- The persistent cache archive contains all the original data used for generating the images. This means that the software appliance can be adapted to new contexts.

Table 8: Some persistent cache archives

Software appliance	Container	Image Size [Mbytes]	Cache Size[MBytes]
hpl benchmark	VirtualBox	1100	581
hpl benchmark	QEMU	1200	582
hpl benchmark	Docker	1600	520
Archive for all appliances		3900	703

6. FUTURE WORK

In future work, we plan to generalize the persistent cache to provide a repository of persistent cache files, and make this repository available to the community. Our vision of this community includes researchers and software developers: anyone who needs to build a particular software stack. This repository will include the instructions (recipes and steps files) and its associated data. Therefore, multiple software appliances can be stored, reducing significantly the storage requirements (as demonstrated in the last row of Table 8). Using this repository and *Kameleon* eliminates the need to store large binary files. *Kameleon* can impact the manage of IT infrastructures as it can be used to manage the deployment and customization of software appliances. Furthermore, we are interested in exploring *Kameleon* as a platform for continuous integration. We believe that Kameleon's automation of software appliance building is well suited for continuous integration. Finally, because the whole environment setup is known, we believe that *Kameleon* can make bug tracking easier.

7. CONCLUSIONS

We introduced the concept of reconstructability which establishes the requirements that a software experimental setup has to meet for improving the reproducibility of experiments in computer science. We proposed *Kameleon* a software appliance builder that supports reconstructability. *Kameleon* provides a modular way to describe the construction of software appliances, which encourages collaboration and reuse of work. Support of reuse lowers the entry barrier for experimenters with low sysadmin skills. *Kameleon* persistent cache makes experimental setups reconstructable at any time.

8. ACKNOWLEDGMENTS

We would like to thank Peter F Sweeney for his insightful, detailed and constructing comments on the paper.

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.

9. REFERENCES

[1] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 335–348, Berkeley, CA, USA, 2012. USENIX Association.

[2] G. R. Brammer, R. W. Crosby, S. Matthews, and T. L. Williams. Paper mâché: Creating dynamic reproducible science. *Procedia CS*, 4:658–667, 2011.

[3] F. Cappello, F. Desprez, M. Dayde, E. Jeannot, Y. Jégou, S. Lanteri, N. Melab, R. Namyst, P. Primet, O. Richard, E. Caron, J. Leduc, and G. Mornet. Grid'5000: a large scale, reconfigurable, controlable and monitorable Grid platform. In *6th IEEE/ACM International Workshop on Grid Computing (Grid)*, pages 99–106, Nov. 2005.

[4] A. Carpen-Amarie, A. Rougier, and F. LÃijbbe. Stepping stones to reproducible research: A study of current practices in parallel computing. In *Euro-Par 2014: Parallel Processing Workshops*, volume 8805 of *Lecture Notes in Computer Science*, pages 499–510. Springer International Publishing, 2014.

[5] C. Christian, P. Todd, M. Gina, S. Akash, S. Zuoming, and W. Alex. Measuring reproducibility in computer systems research. Technical report, Arizona Univeristy, Technical Report, 2013.

[6] E. Dolstra and A. Löh. Nixos: A purely functional linux distribution. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 367–378, New York, NY, USA, 2008. ACM.

[7] J. T. Dudley and A. J. Butte. In silico research in the era of cloud computing. *Nature Biotechnology*, 28(11):1181–1185, Nov. 2010.

[8] J. Emeras, B. Bzeznik, O. Richard, Y. Georgiou, and C. Ruiz. Reconstructing the software environment of an experiment with kameleon. In *Proceedings of the 5th ACM COMPUTE Conference: Intelligent and scalable system technologies*, COMPUTE '12, pages 16:1–16:8, New York, NY, USA, 2012. ACM.

[9] G. Fox, G. von Laszewski, J. Diaz, K. Keahey, J. Fortes, R. Figueiredo, S. Smallen, W. Smith, and A. Grimshaw. *FutureGrid - a reconfigurable testbed for Cloud, HPC, and Grid Computing*. CRC Computational Science. Chapman & Hall, 04/2013 2013.

[10] B. Howe. Virtual appliances, cloud computing, and reproducible research. *Computing in Science and Engg.*, 14(4):36–41, July 2012.

[11] S. Kang and S. Ryu. Formal specification of a javascript module system. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 621–638, New York, NY, USA, 2012. ACM.

[12] S. Perarnau, M. Tchiboukdjian, and G. Huard. Controlling cache utilization of hpc applications. In *International Conference on Supercomputing (ICS)*, 2011.

[13] C. Ruiz, O. Richard, and J. Emeras. Reproducible software appliances for experimentation. In *Proceedings of the 9th International ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (Tridentcom)*, Guangzhou, China, 2014.