BUILDING SOFTWARE-DEFINED SYSTEMS WITH REPRODUCIBLE ENVIRONMENTS

Hyungro Lee

Submitted to the faculty of the University Graduate School

in partial fulfillment of the requirements

for the degree

Doctor of Philosophy

in the Department of Computer Science,

Indiana University

May 2019

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Doctoral Committee

Geoffrey Charles Fox, PhD

Martin Swany, PhD

Haixu Tang, PhD

Lei Jiang, PhD

May , 2019

Acknowledgments

First of all, I would like to thank Dr. Geoffrey Fox for his supervision and generous support in exploring my research idea over the course of my academic career. I would like to thank Dr. Martin Swany, Dr. Haixu Tang, and Dr. Lei Jiang for serving on my committee and for their academic advice to complete my PhD journey. I would also like to thank Dr. Dennis Gannon for offering me the research opportunity at Microsoft. I would like to thank Dr. Gregor von Laszewski for his guidance and technical advice during several projects throughout my graduate program at Indiana University.

To my family.

Hyungro Lee
BUILDING SOFTWARE-DEFINED SYSTEMS WITH REPRODUCIBLE ENVIRONMENTS

Modern distributed server applications are often built with a rich set of libraries and packages that present dependency problems for setting up environments across different places. The issues are not only complicated set of libraries but also the particular configuration of infrastructure to fulfill performance requirements. With multi-tenant systems, one can ensure building manageable systems for their applications. The typical interface that we are using to build software environment is a platform dependent which is difficult to extend and migrate to new systems. Building environment has to be thoroughly planned and tested otherwise, requirements are not going to be fulfilled in which users, developers, administrators, and providers have individual specifications on software versions, operating systems and hardware support. In addition, dynamic resource provisioning is a common issue for improving resource utilization, especially for data-intensive applications with continuously growing storage space needs. The main area of this dissertation is to improve the management of software environments with performance-aware resource provisioning by automated techniques, resulting in efficiently built, but the minimal operational effort of application deployment. The contribution of this dissertation presents techniques to enable a software-defined sub systems using recent technologies such as DevOps and containers. In detail, the contributions include 1) scripting application deployment on multi-clouds platforms, 2) sharing common core components for efficient software defined systems, and 3) performance evaluation of deployed environments. Developments of this work are publicly available and verified with experiments on existing systems including production computing environments.

Geoffrey Charles Fox, PhD

Martin Swany, PhD

Haixu Tang, PhD

Lei Jiang, PhD

Contents

Chapter 1

Introduction

From Infrastructure-as-a-Service to Functions-as-a-Service, many efforts have been made to provide computing resources in virtualized environments but with less complication of building infrastructure and preparing environments. Lightweight Linux containers are widely adopted in supporting interdisciplinary field of research and collaboration because of its kernel level of an isolated environment. IaaS is a still best approach to operate fine-grained resource provisioning regarding to CPU, memory, storage and network. This dissertation will explore the rapid evolution of virtualization technologies from IaaS to serverless computing with container technologies to find optimized configuration of systems with a general software deployment. The next generation system must utilize DevOps tools for deploying software stacks on a cluster of virtual machines and infrastructure provisioning for supporting various applications. In recent years, building big data clusters have become an inevitable task for performing analysis with the increased computational requirements for large dataset and the anticipated systems will be perfect for these situations to every dimension of infrastructure provisioning and software deployment.

This dissertation presents techniques and improvements of building dynamic computing environments as a sub-system of software-defined systems that are designed to process computations on resource optimized software environment while dedicated software stacks are deployed. The subsystem deploys user applications with configurations and input data sources to prepare equivalent software environment at once. The virtual clusters are provisioned with desired compute resources and aim to ensure scalability as data size and computation requirement change over time.

Thesis statement: A system with automated scripting and programmable infrastructure provisioning proposes an effective way of enabling software defined systems on virtual clusters with

latest technologies.

Challenges in Integrating Applications and Compute Resources

Supporting big data analytic, for example, becomes difficult for a few reasons: (1) big data applications run with large amounts of data and a collection of software, (2) building and managing big data deployments on clusters require expertise of infrastructure and (3) Apache Hadoop based big data software stacks are not suitable for HPC. In an effort to resolve the first two issues, public dataset and data warehouse on the cloud have offered to ensure instant data access with SQL support and enterprise big data solutions have hosted by cloud providers e.g. Amazon, Google, and Microsoft to save time on deploying multiple software stacks without facing installation errors. These services, however, are only available to their customers and make hard to switch to another when applications and pipelines are built on top of the services. Pre-developed infrastructure for big data stacks are only suitable for particular use cases and are unable to customize or re-define by users. There are efforts to simplify a deployment with a specification such as automated deployment engines using TOSCA [90], but they do not integrate multiple clouds with a cluster deployment or a workload execution. The bigdata deployment on clusters require more than single software to install and configure with different roles i.e. masters and workers. If it is built on virtualized resources, scaling up or down is necessary to maximize resource utilization but with optimal performance.

These issues can be resolved using a template deployments for infrastructure and software which uses YAML or JSON documents to describe installing tools and allocating resources. For example, Amazon OpsWorks uses Chef to deploy software stacks and Cloudformation uses YAML document to deploy Amazon resources. Similarly, Microsoft Resource Manager Templates uses JSON document to deploy Azure resources and Google Cloud deployment Manager uses python or Jinja2 templating language to deploy Google Cloud platform resources. OpenStack heat is originated

from AWS Cloudformation to deploy resources but extended with the integration among openstack services e.g. Telemetry for auto-scaling. These templates have been used for infrastructure deployment and software installation with input parameters which enables repeatable provisioning on virtual environments. We extend the use of a template with workload execution to align resource requirement of a workflow and ensure performance while identical results are generated on virtual clusters. This will be beneficial to share complicated and long-running pipelines which are often failed to replicate. Our approach in the dissertation is to integrate software stacks and infrastructure provisioning by proposing a software-defined system in terms of deploying software environment on flexible virtual clusters. This allow one can replicate the workflow on different platforms but guarantee an equivalent computing environment at any time.

This dissertation consists of the following sections. First, background in Chapter 2 introduces a template deployment with big data software stacks, template use cases for public clouds infrastructure provisioning and reproducible software environment using containers. A literature review of the related work is also addressed in the chapter. Next, the chapter 3 provides a implementation of a template deployment with NIST big data use cases running on virtual clusters. The software defined systems using containers is presented in the chapter 4 with the evaluation of Docker storage drivers for building big data software stacks. Chapter 5,6 provide evaluation of software defined systems using experiments on event-driven computing and bare metal clouds. Last, the chapter 7 outlines this dissertation with future research topics.

Chapter 2

Background

We explore provisioning infrastructure and deploying software environments using automated tools compared to manual administration. Software defined system is designed for data analytic on virtual clusters with access to local disks and is optimized for data-parallel tasks. We examine the system on event-driven computing and bare metal clouds in which we performed scaling experiments on up to 10,000 tasks, identifying bottlenecks, and offering optimization for best performance in this dissertation. This chapter provides a background on an existing technology good for dependency control and software environment management such as containers along with the necessity of performance analysis towards building a software defined sub system. Related work containing the literature review is followed by the contribution as well.

Data analytic frameworks have been widely used to provide insights of massive amounts of data generated by simulations and equipment from industries and academic communities. Large data sets i.e. terabyte or petabyte scale, are generated by molecular dynamics (MD) simulations [78], and genomic data sets have been stored in multiple institutions for analysis conducted by bioinformaticists and biologists [54] but the limited number of frameworks have been implemented to ensure high-throughput analysis on cloud computing and HPC systems.

Infrastructure support for the large data analysis has made substantial improvements in terms of data movement and dynamic resource provisioning.Hadoop and spark for these analysis as those are proven industry applications of scalable computing interfaces. Leverage of these tools would be beneficial but the common libraries are not generally available across different platforms due to the nature of open source software development. Therefore, automated software deployment at scale becomes important for processing large data sets on various platforms.

Deploying a software environment on the virtual clusters is an effort to offer automated management for distributed applications that impose flexible systems ensuring reproducibility and availability of software environments. Big data ecosystems, for example, expect to manage hundreds of software packages on heterogeneous compute resources in an automated fashion repeating software stack deployment with frequent changes. Software installation and configuration can be cumbersome in a typical system which requires higher privileges for managing system libraries and packages and dealing with compatibility issues. It, thus, creates additional barriers for moving unique software stacks to a new platform with different system configurations e.g. operating system and runtime system version. Building reproducible software stacks on various compute platforms is non trivial but it may reduce technical difficulties of deploying application environments for collaborative activities and improve efficiency of the management as further updates and changes are often necessary.

The number of application libraries and the user software have been increased rapidly as many people need new functionality for complex data processing. According to the National Institute of Standards and Technology (NIST) report, 51 use cases across 26 fields are described in which the extensive developments of software stacks are addressed including Hadoop, Spark, Hive, Mahout, Storm, Cassandra and so on in 2018. The fast growth of data size is another challenge as parallel software need to extract value from huge amounts of data across distributed nodes. Various use cases from broad areas like health care, earth, environmental and polar science have built custom software environments to solve particular problems with a rich set of data analysis tools, and the following section describes existing issues on managing shared packages and libraries.

2.1 Dependency Control

Open source software is made by many packages and libraries which support various functionality and establish reliability of requirements e.g. secured connections and consistent performance instead of developing every components from the ground up. While there are several advantages of importing packages or libraries for software development, dependency issue has aroused with incompatibility across different versions of imported tools. Running software on different operating systems or platforms makes more difficult to manage software environments as package managers e.g. apt, yum have individual set of repositories. New technologies have been introduced to mitigate these problems by user defined, independent and portable software environments, for example, Docker provides reproducible and shareable container images, and Conda creates self-contained environments using hard links to system libraries, and python's Pipenv, virtualenv, Node.js' npm or Ruby's bundler simplifies dependency management for each programming language.

Managing reproducible scientific experiments is another challenge to maintain with manual configurations and special settings. End-users are willing to combine multi-language tools and compile software manually with extra skills of Linux systems as long as they are allowed to build executable environments for jobs, e.g. simulations and data analysis, while meta-data of installed software and dependencies are difficult to share equivalent environment on different platforms and systems and to preserve for further development. Installation and configuration using shell scripts is traditional means of software distribution and maintenance which prevents prolonged software stack management as they do not store dependencies and meta data like system package manager does.

The system space managed by administrators prohibits users to have control over the libraries required for software environments in which external tools and packages are relied on them. In share computing systems, the separated privileges between administrators and users reduce problems

on conflicts and incompatibilities but applying rapidly growing packages e.g. community driven packages, would be impractical. Ideal software environment has to be independent to an operating system, programming language, and transparent to manage dependencies. Considering millions of software and complicated dependency relations, schematic and simplistic software environment is preferred to understand and replicate with minimum efforts by users. Only concern of having individual software environment is using outdated insecure software tools, in which administrators are not able to force them to update, and expose vulnerabilities.

2.1.1 Scripting Deployment for Repeatability

Software development has evolved with rich libraries and building a new computing environment (or execution environment) requires set of packages to be successfully installed with minimal efforts. The environment preparation on different infrastructure and platforms is a challenging task because each preparation have individual instructions which build a similar environment, not identical environment. Traditional method of software deployment is using shell scripts to define installation steps with a system package manager command such as apt, yum, dpkg, dnf and make but it is not suitable to deal with large number of packages actively updated and added to community in a universal way. Python Package Index (PyPI) has almost 95,490 packages (as of 12/26/2016) with 40+ daily new packages and github.com where most software packages, libraries and tools are stored has 5,776,767 repositories available with about 20,000 daily added repositories. DevOps tools i.e. Configuration management software supports automated installation with repeatable executions and better error handling compared to bash scripts but there is no industry standards for script formats and executions. Puppet, Ansible, Chef, CFEngine and Salt provide community contributed repositories to automate software installation, for example, Ansible Galaxy has 9329 roles available, Chef Supermarket has 3,135 cookbooks available although there are many duplicates.

We call this is (automated) software deployment and building dynamic computing environments on virtual clusters is the main objective of this dissertation. Software defined systems (or virtual clusters) has discussed [39] to connect distributed big data and computing resources of Cloud and HPC, which will result in developing a suite of software deployment tools at scale. Note that this effort is mainly inspired by the previous research activities [40–43, 46, 76].

2.1.2 Programmable Template Code for Automation

A template has been used to describe a deployment of software packages and infrastructure on virtual environments across multiple cloud providers because of its simple instructions as well as content shareability. YAML (superset of JSON) is a popular language to serialize data delivery especially as for configuration files and object persistence along with the template deployment. As an example of infrastructure deployments, Amazon Cloudformation, a template deployment service, uses YAML or JSON specification to describe a collection of Amazon virtual resources, Google Compute Cloud uses YAML with Jinja2 or Python languages to define a set of google compute resources whereas Microsoft Azure Resource Manager uses JSON to deploy Azure resources and Topology and Orchestration Specification for Cloud Applications (TOSCA) uses XML [89] to define a topology and a relationship. Saltstack and Ansible, a software deployment tool written in Python, use YAML to manage configuration and software installation from instructions defined in YAML text files.

Listing 2.1: AWS CloudFormation Example

```
Resources:
  EC2Instance:
    Type: AWS::EC2::Instance
    Properties:
```

```
      InstanceType:
        Ref: InstanceType
      SecurityGroups:
      - Ref: InstanceSecurityGroup
      KeyName:
        Ref: KeyName
```

The code example in Listing 2.1 is a plain text to deploy a Amazon EC2 instance written in a YAML format which includes a nested data structure by indentations and key value pairs for lists (starts with dash) and dictionaries.

Listing 2.2: Ansible Example

```
---
- hosts: opencv

  tasks:
  - name: compiler package
    apt: name=build-essential state=present update_cache=yes
  ...
```

Ansible, automation tool, uses YAML syntax with Jinja2 template to describe instructions of software installation and the code example in Listing 2.2 shows a code snippet of Ubuntu's APT (advanced packaging tool) installing build-essential Debian package during the OpenCV software installation.

There are several reasons to use a template for a deployment. First, installing software and building infrastructure typically demand lots of commands to run and additional configurations to

setup and a template is suitable for these tasks with its data structures using key-value pairs, lists and dictionaries to contain all instructions to reproduce a same environment and to replicate an identical software installation on different locations at another time. In addition, with the advent of devops, a template deployment enables cooperation between a template developer and a template operator because a complicated set of resources and services is simplified by a single template file and delivered to an operator as an automated means of provisioning a same environment. Moreover, YAML or JSON is a simple text format for storing data which is easy to share and modify with anyone who interested in a template. There are still plenty of benefits that we can find when a template deployment is used.

Big Data applications typically require efforts on deploying all of the software prerequisites and preparing necessary compute resources. A template deployment reduced these efforts by offering an automated management on both tasks; software deployment and infrastructure provisioning, therefore we can focus on big data applications to develop.

The concept of serverless computing also applies to deploy applications with templates e.g. Listing 2.1. For instance, Amazon serverless compute, AWS Lambda, invokes serverless application code (also called function) based on the description of the template but uses a specific model e.g. Listing 2.3 for components of serverless applications. In detail, there is a main function (Handler), runtime environment (Runtime), and an actual code in a compressed format (CordUri).

Listing 2.3: AWS Serverless Application Model (SAM) Example

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Resources:
  MyFunction:
    Type: 'AWS::Serverless::Function'
```

```
Properties:
  Handler: hello_python.handler
  Runtime: python2.7
  CodeUri: 's3://my-bucket/function.zip'
```

2.1.3 Container Technologies for Reproducibility

Container technology has brought a lightweight virtualization with a Linux kernel support to enable a portable and reproducible environment across laptops and HPC systems. Container runtime toolkit such as Docker [71], rkt [3] and LXD [4] has been offered since 2014 which uses an image file to initiate a container including necessary software packages and libraries without an hypervisor which creates an isolated environment using a virtual instance but with an isolated namespace on a same host operating system using the Linux kernel features such as namespaces, cgroups, seccomp, chroot and apparmor. Recent research [33] shows that containers outperform traditional virtual machine deployments yet running containers on HPC systems is still an undeveloped area. Shifter [57] and Singularity [64] have introduced to support containers on HPC with a portability and MPI support along with docker images. These efforts will be beneficial to scientific applications to conduct CPU or GPU intensive computations with easy access of container images. For example, a neuroimaging pipelines, BIDS Apps [53], is applied to HPCs using Singularity with existing 20 BIDS application images and Apache Spark on HPC Cray systems [21] is demonstrated by National Energy Research Scientific Computing Center (NERSC) using shifter with a performance data of big data benchmark. Both researches indicate that scientific and big data workloads are supported by container technologies on HPC systems for reproducibility and portability.

Listing 2.4: Dockerfile Example

```
FROM ubuntu:14.04

MAINTAINER Hyungro Lee <lee212@indiana.edu>

RUN apt-get update && apt-get install -y build-essential
...
```

Dockerfile uses a custom template to describe installation steps of building docker images in a bash like simple format. There are certain directives to indicate particular objective of the commands, for example, FROM indicates a base image to use and RUN indicates actual commands to run.

2.2 Infrastructure Control using Template Orchestration

Template deployment is a means of installing software and building infrastructure by reading a file written in a templating language such as YAML, JSON, Jinja2 or Python. The goal of a template deployment is to offer easy installation, repeatable configuration, shareability of instructions for software and infrastructure on various platforms and operating systems. A template engine or an invoke tool is to read a template and run actions defined in a template towards target machines. Actions such as installing software package and setting configurations are described in a template file using its own syntax. For example, YAML uses spaces as indentation to describe a depth of a dataset along with a dash as a list and a key-value pair with a colon as a dictionary and JSON uses a curly bracket to enclose various data types such as number, string, boolean, list, dictionary and null. In a DevOps environment, the separation between a template writing and an execution helps Continuous Integration (CI) because a software developer writes deployment instructions in a template file while a system operations professional executes the template as a cooperative effort. Ansible, SaltStack, Chef or Puppet is one of popular tools to install software using its own templating language. Common features for those tools are installing and configuring software based on definitions but with different strategies and frameworks. One observation is that the choice of implementation languages for those tools influences the use of a template language. The tools written by Python such as Ansible and SaltStack use YAML and Jinja which are friendly to a Python language with its library support whereas the tools written by Ruby such as Chef and Puppet use Embedded Ruby (ERB) templating language. In scientific community, a template has been used to describe data and processes of pipelines and workflow because a template contains detailed information of them in writing and assists sharing and connecting between different layers and tools. Parallel execution on distributed environments is also supported in many tools yet enabling computations in a scalable manner needs expertise to prepare and build the environments.

We propose a template orchestration to encourage scientists in using distributed compute resources from HPC and cloud computing systems in which provisioning infrastructure is documented in a template and complicated pipelines and workflows are packaged by container technologies for reproducibility.

2.2.1 Template deployment for Big Data Applications

Software installations and configurations for particular domains have become hard to maintain because of an increased number of software packages and complexity of configurations between them to connect. Template deployment for installing and provisioning systems across from a single machine to large number of compute nodes is proposed to achieve consistent and reliable software deployment and system provisioning.

First, we plan to implement a deployment tool with default components for big data software such as Apache Hadoop, Spark, Storm, Zookeeper, etc. therefore a software deployment can be achieved by loading existing templates instead of starting from scratch. The software deployment intends to support various linux distribution with different versions, therefore the software stacks are operational state in many environments without a failure.

Listing 2.5: Template Deployment for Big Data

```
stacks:
  - software A
  - software B
  - ...
```

Each item i.e. software indicates a single template file to look up deployment instructions. Dependencies indicates that related items to complete a deployment and the environment variables are shared while dependencies are deployed. If container image is available on the web, container

image deployment is expected using the URI location to save compile time.

Listing 2.6: Sample of software template

```
instruction:
  - install package A
  - download data B
location:
  <URI>
dependency:
  - software A
  - library B
environment_variables:
  - HOME_DIR=/opt/software_a
```

Infrastructure deployment is provisioning of cloud computing which includes virtual machine images, server types, network groups, etc. in preparation of virtual resources for the software stacks. Infrastructure deployment for multiple cloud platforms includes Microsoft Azure Resource Manager Templates, Amazon CloudFormation Templates, and Google Compute Instance Templates. Each cloud provider owns individual models for their services therefore a template of the deployment is solely executable in each provider although similar infrastructure is necessary for the software stacks.

Listing 2.7: Support for cloud providers

```
infrastructure:
  - default: aws
  - options:
```

```
    - aws
    - gce
    - azure
    - openstack
aws:
  services:
    image:
      - image A
      - image B
      - image B version 2
    server:
      - server type A
    network:
      - network interface a
      - network ip address a
```

We plan to integrate container based deployments with popular tools such as Docker therefore image based software deployment is also supported to enhance reproducibility and mobility on different environments.

Listing 2.8: Template Deployment with Containers

```
format:
  - default: docker
  - options:
    - docker
```

- **ansible**
- **shell**
- **rkt**

Template has been used to document instructions for particular tasks such as software installation and configuration or infrastructure provisioning on cloud computing, however, shareability of templates is not improved which requires for better productivity and reusability. We plan to design a template hub to collect, share, search and reuse well written templates with a common language e.g. yaml or json, therefore building software stacks and provisioning infrastructure both are repeatable in any place at any time.

In addition, provenance data and process state will be reserved.

2.2.2 Infrastructure Provisioning on Clouds

Infrastructure provisioning has supported with templates in many cloud platforms i.e. Amazon Cloudformation, Microsoft Azure Resource Manager, OpenStack Heat and Google Compute Instance Templates. Infrastructure described in a template will be created for simple tasks running in a standalone machine or multiple tasks in clusters.

Simple Azure - Python Library for Template Deployment on Windows Azure

Implementation of infrastructure provisioning is provided with Azure use case. Simple Azure is a Python library for deploying Microsoft Azure Services using a Template. Your application is deployed on Microsoft Azure infrastructure by Azure Resource Manager (ARM) Templates which provides a way of building environments for your software stacks on Microsoft Azure cloud platform. Simple Azure includes 407 community templates from Azure QuickStart Templates to deploy software and infrastructure ranging from a simple linux VM deployment (i.e. 101-vm-simple-linux)

to Azure Container Service cluster with a DC/OS orchestrator (i.e. 101-acs-dcos). It supports to import, export, search, modify, review and deploy these templates using the Simple Azure library and retrieve information about deployed services in resource groups. Initial scripts or automation tools can be triggered after a completion of deployements therefore your software stacks and applications are installed and configured to run your jobs or start your services. Starting a single Linux VM with SSH key from Azure QuickStart Template is described in listing 2.9:

Listing 2.9: Simple Azure

```
>>> from simpleazure import SimpleAzure
>>> saz = SimpleAzure()

# aqst is for Azure QuickStart Templates
>>> vm_sshkey_template = saz.aqst.get_template('101-vm-sshkey')

# arm is for Azure Resource Manager
>>> saz.arm.set_template(vm_sshkey_template)
>>> saz.arm.set_parameter("sshKeyData", "ssh-rsa AAAB... hrlee@quickstart")
>>> saz.arm.deploy()
```

2.2.3 Semantics

Advances in big data ecosystem will require to connect scattered data sources, applications and software in meaningful semantics. It is necessary to develop structured semantics as an effort of support in discovering big data tools, datasets and applications all connected because semantics is more understandable to both human and machine with a standard syntax for expressing contents in RDF (Resource Description Framework) model or JSON-LD (Linked Data using JSON) [15,

65, 79]. It also provides a guideline to construct big data software stacks to community in which preparing development environments is complicated with newly introduced software and datasets. This is particularly useful given the increasing number of tools, libraries and packages for further development of big data software stacks. One example in the listing 2.10 shows two applications, C++ Parser for MNIST Dataset and a Python package to convert IDX file format provided by Yann LeCun's dataset, are available for MNIST database of handwritten digits on github. There are couple of tasks to implement semantics for template deployment:

1. collect big data software, applications, and datasets
2. produce JSON-LD documents
3. derive Rest API to search, list and register
4. implement a library to explore documents about big data ecosystem

Listing 2.10: Sample of linked data between dataset and software

```
{
    "@context": "http://schema.org/",
    "@type": "Dataset",
    "distribution": "http://yann.lecun.com/exdb/mnist/",
    "workExample": [
      {
        "@type": "SoftwareSourceCode",
        "codeRepository": "https://github.com/ht4n/CPPMNISTParser",
        "description": "C++ Parser for MNIST Dataset",
        "dateModified": "Sep 1, 2014",
        "programmingLanguage": "C++"
      },
```

```
13      {
14        "@type": "SoftwareSourceCode",
15        "codeRepository": "https://github.com/ivanyu/idx2numpy",
16        "description": "A Python package which provides tools to
              convert files to and from IDX format",
17        "dateModified": "Sep 16, 2016",
18        "programmingLanguage": "Python"
19      }
20    ]
21  }
```

2.3 Environment Control using Container Technology

With the increased attention of Docker container software and reproducibility, the use of virtualization has been moved from the hypervisor to a linux container technology which shares kernel features but in a separated name space on a host machine with a near native performance [33]. The recent researches [10] indicate that the HPC community takes account of container technologies to engage scientists in solving domain problems with less complication of deploying workflows or pipelines on multiple nodes as new implementations have been introduced [57, 64, 74]. Container technology with HPC, however, is focused on supporting compute-intensive applications i.e. Message Passing Interface (MPI) although many scientific problems are evaluated with big data software and applications. Investigation on container technology with big data ecosystem is necessary to nurture the data-intensive software development on HPC with a rich set of data analysis applications.

Modern container software run with container images to create isolated user space based on preconfigured environments. Authoring container image definition is a first step to prepare custom environments via containers and to share with others. Dockerfile is a text file to create a docker container image with instructions for package installation, command executions, and environment variable settings. Definition File of Singularity also contains similar instructions to build container images. Application Container Image (ACI) of CoreOS rkt is generated by a shell script and acbuild command line tool but building container images is similar to docker. The main objective of using these container image definitions (formats?) is to reveal user commands and settings explicitly therefore the development environment can be shared easily and conversion between other platforms is doable. The initial goal of using container technology in this dissertation is building a container-based big data ecosystem by offering a template-based deployment for container images. It would also enable a concise and descriptive way to launch complex and sophisticated scientific

pipelines using existing container images or deployment scripts. Performance tests are followed to demonstrate efficiency of the deployments with big data applications on modern container technologies. We desire to measure overhead introduced by container software i.e. shifter, singularity on HPC environments with comparison of CPU, memory, filesystem, and network usages.

Template based deployment is adopted in container technologies, for example, Singularity uses a custom syntax, SpecFile to describe the creation of a container image with directives which are similar to Dockerfile. Listing 2.11 shows an example of Caffe Deep Learning Framework Singularity image creation.

Listing 2.11: Singularity Example

```
DistType "debian"
MirrorURL "http://us.archive.ubuntu.com/ubuntu/"
OSVersion "trusty"

Setup
Bootstrap

...(suppressed)...
RunCmd git clone -b master --depth 1 https://github.com/BVLC/caffe.git
RunCmd sh -c "cd caffe && mkdir build && cd build && cmake -DCPU_ONLY=1 ..."
RunCmd sh -c "cd caffe/build && make -j1"

RunCmd ln -s /caffe /opt/caffe
RunScript python
```

2.3.1 Package Manager for Common Libraries

One of the benefits of using containers is that required software packages are included in the build instruction, therefore common package names are revealed for particular collections. Table 2.1 is an example of debian packages described in Dockerfiles related to NIST collection and dpkg, debian package command, has been used to collect package information.

2.3.2 Evaluation

Performance evaluation of container technologies has completed with big data applications from NIST Collection. There are six applications in the collection: Fingerprint Matching, Human and Face Detection, Twitter Live Analysis, Data Warehousing, Healthcare Information, and Geospatial information. Performance data on CPU, memory, storage and network will be measured on HPC and cloud computing with container software i.e. docker, rkt, singularity and shifter.

Preloading common packages shows possible optimization for the template deployment according to the figure 2.1. With a considerable reduce on network traffic for downloading packages, 10x speedup is approximately observed over multiple access to Debian software package mirror sites. Statistics for the cache reuse (Table 2.2) indicates that the most benefit of the speedup is gained from the cached packages. In addition, standard deviation for download speed is higher in using remote mirrors than cached proxy server in which network consistency and reliability are ensured with low standard deviation for download speed.

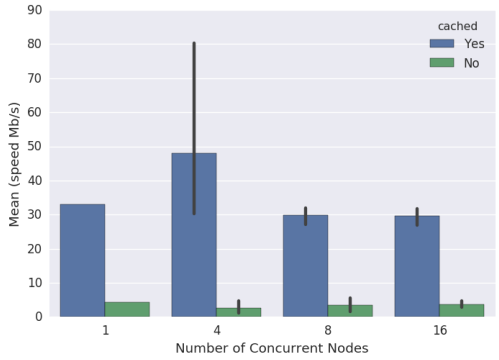
Name	Description	Dependencies	Size (Kb)	Priority
python-dev	header files and a static library for Python (default)	python, python2.7-dev, libpython-dev	45 (1024)	optional
python	interactive high-level object-oriented language (default version)	libpython-stdlib, python2.7	680 (384)	standard
zlib1g-dev	compression library - development	libc6-dev, zlib1g	416 (12516)	optional
apt-utils	package management related utility programs	libgcc1, libapt-inst1.7, libstdc++6, apt, libdb5.3, libc6, libapt-pkg4.16	688 (21070)	important
python-numpy	Numerical Python adds a fast array facility to the Python language	python, python2.7:any, libblas3, liblapack3, libc6	8667 (17873)	optional
nodejs	evented I/O for V8 javascript	libssl1.0.0, libc6, libstdc++6, zlib1g, libv8-3.14.5, libc-ares2	3043 (20625)	extra
python-imaging	Python Imaging Library compatibility layer	python-pil, python:any	45 (1248)	optional

Table 2.1: Top Debian-based Packages used in Dockerfiles for the NIST collection on Github (size with parenthesis indicates total size including dependency packages)

Type	Hits	Misses	Total
Requests	104993 (95.26%)	5220 (4.74%)	110213
Data	12627.32 MiB (99.78%)	27.95 MiB(0.22%)	12655.27 MiB

Table 2.2: Cache Efficiency for Software Package Installation measured by apt-cacher-ng

Figure 2.1: Accelerated Common Package Installation using Software Package Proxy



2.4 Performance Analysis

The performance analysis of a user application is often overlooked when we make resource reservation in a shared environment and submit provisioning request on a virtualized infrastructure. We discuss performance analysis in this section to explain the benefit of using performance data and find out possible opportunities for better resource provisioning as a process of building software defined systems.

People in industry and academia involved in distributed and parallel systems had little concerns of resource provisioning and utilization while many of them suffer from batch job queue waiting time on HPC systems and from extra charges on commercial cloud computing. There are multiple factors contributing this problem such as inaccuracy of resource estimation, imprecision of resource reservation or just human error predicting task runtime but the impact of the problems and solutions was not discussed intensively. The understanding of the problems has to be improved since every user application becomes resource intensive while computing resources are limited in a shared environment, no matter how big and powerful systems are available. Administrators and researchers had worked on resource utilization manually to save allocation and collected profiling for code optimization as they gain technical skills to identify bottlenecks and apply improvement. In order to find out proper resource amounts to allocate, however, automated profiling is necessary to provide an estimate of required resources in advance. This applies not only large scale distributed systems but also cloud computing which users have various options for building infrastructure in terms of cost efficiency. Certainly, there is an awareness of the inaccuracy problems with the estimator which can create over or under provisioning due to imprecise calculation. Interactive adjustment would be considered to mitigate the problem on the fly when it is useful.

2.5 Related Work

2.5.1 DevOps Scripting Tools

In the DevOps phase, configuration management tools automates software deployment to provide fast delivery process between development and operations [31]. Instructions to manage systems and deploy software are written in scripts although different formats i.e. YAML, JSON, and Ruby DSL and various terminologies i.e. recipes, manifests, and playbooks are used. There are notable tools available to achieve automated software deployment. Puppet and Chef are identified configuration management tools written in Ruby and these tools manage software on target machines regarding to installation, execution in a different state e.g. running, stopping or restarting, and configuration through the client/server mode (also called master/agent). Ansible is also recognized as a configuration management tool but more focusing on software deployment using SSH and no necessity of agents on target machines. With the experience from class projects and NIST use cases, a few challenging tasks are identified in DevOps tools, a) offering standard specification of scripts to ease script development with different tools, and b) integrating container technologies towards microservices.

2.5.2 Template Deployment

Several infrastructure provisioning tools have emerged to offer transparent and simple management of cloud computing resources over the last few years. Templates which are structured documents in a YAML or JSON format define infrastructure with required resources to build and ensure identical systems to create over time. A collection of Amazon cloud services are provisioned through Cloudformation [1] templates which is an Amazon infrastructure deployment service. OpenStack Heat [2] was started with similar template models to Amazon but has extended with other OpenStack services e.g. Telemetry, monitoring and autoscaling service to build multiple resources aa a single

unit. The Topology and Orchestration Specification for Cloud Applications (TOSCA) [12, 89] proposes standardization over different cloud platforms with XML-based language and several studies have been made with TOSCA [20, 62, 75]. These tools have been addressed with issues in a few studies [46, 91] and one of identified issues is that individual specification of supported resources, functions, type names, and parameters prevents building and sharing infrastructure blueprints across cloud platforms.

2.5.3 Container Technology

While existing container software, e.g. docker, rkt, lxd, offers various features with outstanding performance there are number of new tools recently developed with the support on HPC. Shifter from NERSC on Cray XC30 with GPU [10] has introduced and singularity from LBNL [64] as well. These new implementations are typically for heavy workloads which requires checkpoint/restart for long running applications and easy deployment of required software stacks in a user space.

2.5.4 Topology and Orchestration Specification for Cloud Applications (TOSCA)

TOSCA is a standardized management of cloud services with applications using workflow technologies and the specification [72] to ensure reproducibility.

One of goals that TOSCA aims is to provide portability of cloud service management along with their environments [13], There are a few terminologies in this context. A service template contains all information about operation and management including a topology of cloud services at a top level of abstraction. Plans, Nodes and Relations are included in the service template. A service topology is a description of service components (nodes) and its relations to others therefore the structure of systems to build is represented. Plans have instructions about operations and managements through workflow technology. Orchestration of service operation and management is described in plans with WSDL, REST or scripts. In addition verification (inspection) of the topology and retrieval or modification of service instance information are supported by plans. With BPMN and BPEL workflow languages, TOSCA plans are portable in differement management envrionments to adopt.

OpenTOSCA is a runtime supporting imperative processing of TOSCA-based cloud applications [11]. The core components of OpenTOSCA are implementation artifact engine, plan engine, container API and plan portability API where build plan conducts management operations and deployment of applications with OASIS TOSCA packaging format CSAR.

Eclipse Winery is a graph based modeling tool for TOSCA-based cloud applications using HTML and Eclipse environment [62]. The frontend components with GUI of Winery are divided by the DevOps paradigm to ease collaboration between developers and operators. Topology Modeler provides visual topology modeling to operators with seven elements; relationship template, relationship constraint, node template, deployment artifact, requirement, capability and policy. Element Manager provides controls of technical details to system experts such as types, implemen-

tations, policy templates and configurations. BPMN4TOSCA Modeler is added later to support in creating BPMN elements and structures used in TOSCA plans through web-based graphical user interface. Winery uses databases (called repository) to store TOSCAL models in CSAR format which is a TOSCA Cloud Service ARchive application package.

Visual notation for TOSCA (named Vino4TOSCA) [19] has introduced with explicit design principles and requirements. Nine requirements for desigining effective visual notations are defined as: R1 Semiotic Clarity, R2 Perceptual Discriminability, R3 Semantic Transparency, R4 Complexity Management, R5 Cognitive Integration, R6 Visual Expressiveness, R7 Dual Coding, R8 Graphic Economy, and R9 Cognitive Fit. The requirements for constructing TOSCA-specific notations are: R10 Completeness, R11 Semantic Correctness, R12 Extensibility, and R13 Compact Representation. The requirements for usability and use experience are: R14 Suitability for the Task, R15 Self-descriptiveness, R16 Simplicity, and R17 User Satisfaction.

Automated provisioning of cloud infrastructure is described with the TOSCA topology template and the plan where the structure of cloud applications is defined in the template and an executable provisioning workflow (called plan) is generated based on the template [17]. In practical terms, Winery, topology modeling GUI tool, creates a service template with nodes and relationships to depict a system structure in CSAR format and OpenTOSCA, a TOSCA runtime environment, executes the plans after the process of generating provisioning order graph, provisioning plan skeleton and executable provisioning plan in workflow languages i.e. BPEL and BPMN.

There are additional tools supporting the TOSCA ecosystem. Vinothek [18] is a web interface of application manager on the TOSCA runtimes using Java Server Pages and HTML5. It accepts user inputs for launching applications on the web such as input parameters and runtime-specific information. TOSCAMART (TOSCA-based Method for Adapting and Reusing application Topologies) [80] offers a method to build desired environments on any cloud provider by assembling

Title	Description	Function	Language	Repository	Extensibility
OpenTOSCA	TOSCA Runtime Environment	Runtime system	Java	github.com /OpenTOSCA/container	BPMN, BPEL
Winery	Web-based environment for modeling TOSCA topologies	Front-end GUI	Java	github.com /eclipse/winery	BPMN
BPMN4TOSCA	Extension for TOSCA management plans	Extensions	Javascript	github.com /winery/BPMN4TOSCAModeler	BPMN
Vinothek	Cloud application management	Front-end GUI	Java	github.com /OpenTOSCA/vinothek	CSAR
TOSCA-MART	Methods for adapting and reusing TOSCA cloud applications	Extensions	Java	github.com /jacopogiallo/TOSCA-MART	CSAR

Table 2.3: Components of TOSCA Ecosystem

- BPMN - Business Process Model and Notation
- BPEL - Business Process Execution Language
- CSAR - TOSCA Cloud Service ARchive

fragments of existing TOSCA topologies. This approach includes finding reusable fragments of the topology from repositories, choosing candidates by rates and filters and adapting final candidate fragments through ratings as a process of building desired environments.

2.6 Contribution

The main contributions of this dissertation are listed below:

- **Scripting Deployment of NIST Use Cases** presents a big data ecosystem analysis based on the survey from classes and online communities i.e. github.com and gitlab.com; a public version control repository along with the six use cases of big data applications using Ansible scripts (called Roles).
- **Efficient Software Defined Systems using Common Core Components** describes stacked image layer optimizations on union file systems using common core components i.e. shared libraries and system tools. HPC-ABDS layers are examined to generate an application-specific collection of common core components in building big data software stacks.
- **Implementing Software Defined Sub Systems with DevOps Tools and Template-based Provisioning** will connect infrastructure provisioning and application deployment to provide optimal performance with minimal underutilization of dynamic computing resources.
- **Performance Evaluation on Event-driven Computing** provides metrics of invoking concurrent tasks using dynamic software environments on commercial clouds Our goal is to measure system performance and limitation in exploring new opportunities for big data analytics and scientific HPC applications.
- **Evaluation of Bare Metal Clouds for Big Data** contains benchmark results of launching big data workloads across a computing environment on HPC equivalent to other platforms using containers. All dependencies and configurations needed to run applications are included in container images, and HPC container tools such as Singularity from Lawrence Berkeley National Lab and Shifter from National Energy Research Scientific Computing Center will support this idea.

Chapter 3

Software Defined Provisioning for Big Data Software Environment

3.1 Introduction

Building compute environments needs to ensure reproducibility and constant deployment over time [50, 52]. Most applications these days run with dependencies and setting up compute environments for these applications require to install exact version of software and configure systems with same options. Ansible is a DevOps tool and one of the main features is software deployment using a structured format, YAML syntax. Writing Ansible code is to describe action items in achieving desired end state, typically through an independent single unit. Ansible offers self-contained abstractions, named Roles, by assembling necessary variables, files and tasks in a single directory and an individual assignment (e.g., installing software A, configuring system B) is described as a role. Compute environments are usually supplied with several software packages and libraries and selectively combined roles conduct a software deployment where new systems require environments with needed software packages and libraries installed and configured. Although the comprehensive roles have instructions stacked with tasks to successfully finish a software deployment with dependencies, the execution of applications still need to be verified. In consequence, to preserve identical results from the re-execution of applications, it is necessary to determine whether environments are fit for the original applications. In certain situations, Ansible fails in building same environments due to following reasons. **First, a variety of operating systems and diverse source of packages are not able to construct equivalent environments across different platforms.** According to the Gnu/linux distribution timeline 12.10 [67], 480 linux distributions exist and each distribution has more than ten thousands packages e.g. Ubuntu Xenial 16.04 has

69154 and Debian Jessie 8 has 57062. Many Unix-like variant systems offer universal package manager e.g. apt on Debian, yum on CentOS, dnf on Fedora and pkg on FreeBSD to ease software installation, upgrading or removal through a central repository package.Ansible Conditionals can handle these multiple package managers with different package names e.g. Apache 2 is listed in RedHat as 'httpd' and in Debian as 'apache2' (see example Listing 3.1) but version differences are not considered. For example, Linux distribution has different version of default packages e.g. glibc, therefore ansible roles may not install same version of packages when it is executed on various Linux distributions. Besides, rapidly developing software are typically available from third-party repositories while stable software packages are provided in official repositories in which compatibility is verified. Software defined system with scripting aims to address this problem using a snapshot of required libraries and tools and therefore ensure reproducible environments. **Second is conflicts (also known as software rot) among packages including existing software and libraries.** When Ansible runs towards target machines, software installation may fail due to conflicts between packages or with already installed software and libraries. For example, 25 CUDA packages for deep neural networks have dependencies and reverse dependencies (Figure 3.1). This may abort further installation or upgrade because of incompatibility issues and missing built-in rollback in Ansible makes failure handling more difficult. **Compile time is also inevitable** which takes a considerable amount of time to complete, especially when source code files are many and complex to compile. There are several techniques to minimize the compilation build time but the optimization is limited and CPU and memories are consumed by compilers.

Listing 3.1: Example of Apache Installation using Ansible Conditionals

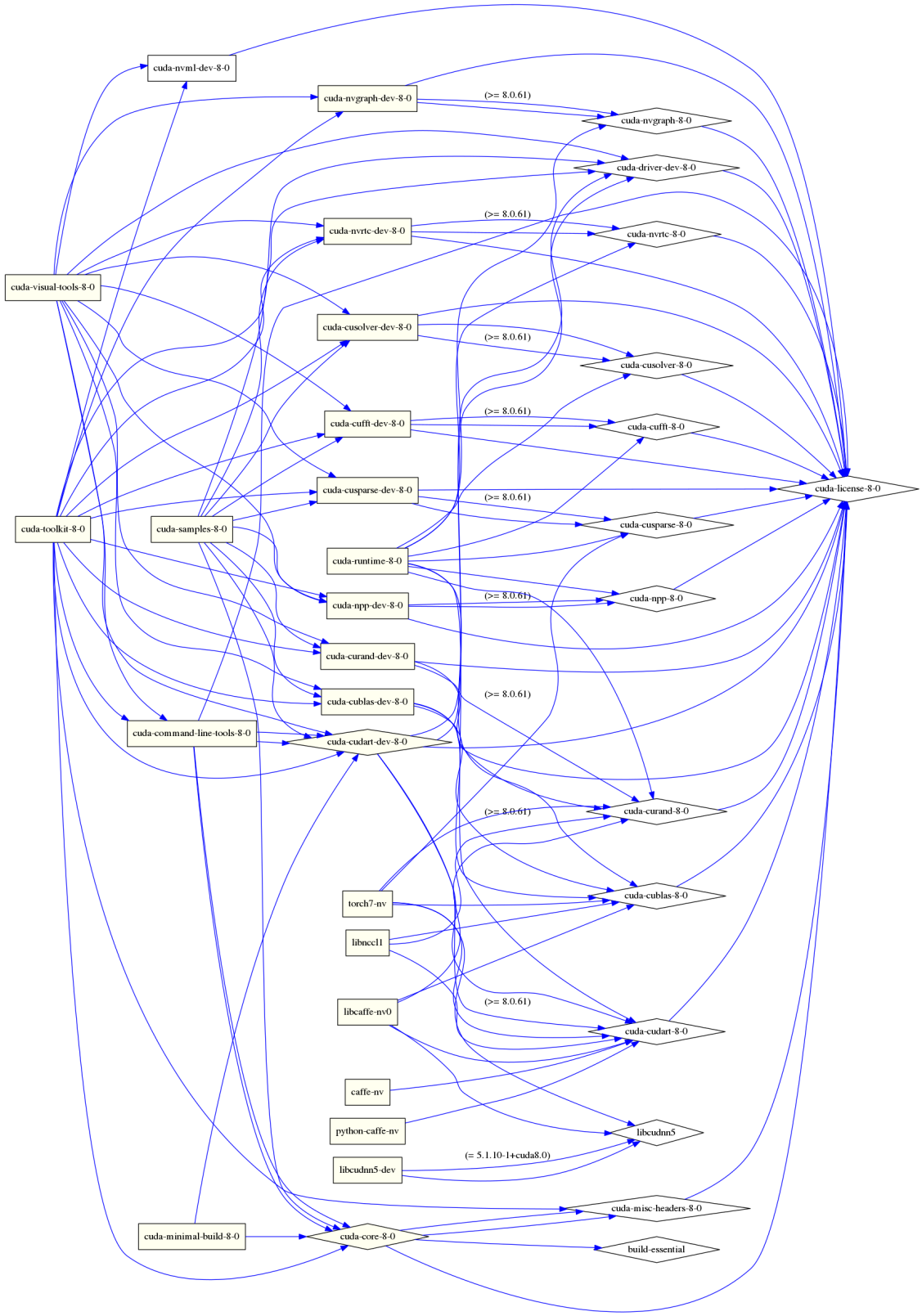
```
- name: Apache Installation
  yum: name=httpd state=installed
  when: ansible_os_family == RedHat
```

```
- name: Apache Installation
  apt: name=apache2 state=installed
  when: ansible_os_family == Debian
```

Configuration drift, which creates divergent in server configuration makes building consistent and identical environments difficult as time goes on. In practice, software version and repository location may vary at install time although deployments are made from a same template. For instance, software provides a downloadable link to the latest release without a specific version in the link therefore an unique link is used to download. However, if a link to the latest release is defined in scripts to download software package, an actual version of the release may not be same when software downloading occurred at a different time, especially where frequent releases are applied to software development. This will increase the likelihood of building another environments or getting failure of deployment.

In addition, **provisioning proper computing resources for an application is not feasible** because virtual server provisioning by Ansible is not bound by the application deployment. Decoupled infrastructure and applications may cause an execution failure of the applications or poor performance due to insufficient compute resources. Applications deployed on virtual environments need to run with particular computing resources such as GPU support, InfiniteBand options, and Solid State Drive (SSD) with TRIM support to satisfy performance requirements and ensure same results. Figure 3.2 shows that a linear performance increment from small to xlarge instance type for Hadoop and the garbage collection overhead is observed in the small instance type. Ansible Roles are tailored to application deployment but it is not for allocating proper hardware resources. Manual provisioning plans are necessary to meet the application requirements. **Vendor lock-in problem in deploying Ansible roles prevents building a same environment across multi-**

Figure 3.1: Example of Package Dependencies for CUDA Libraries (1-level depth)



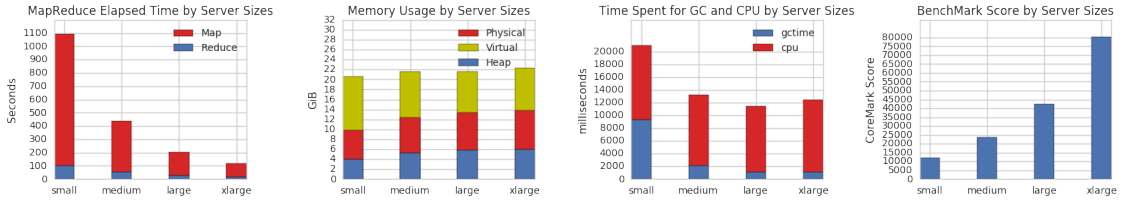


Figure 3.2: Hadoop Comparison by Server Sizes

(where small has 1vCPU and 2GiB memories, medium has 2vCPUs and 4GiB, large has 4vCPUs and 8GiB and xlarge has 8vCPUs and 16GiB. GC stands for a garbage collection.)

ple cloud providers although Ansible provides multi-cloud functions (called modules in Ansible) to favor portability and agility. For example, Amazon CloudFormation, Google Compute Instance Templates, Microsoft Azure Templates and OpenStack Heat have an individual specification that defines properties and resources of the infrastructure. Inter-cloud standard specification needs to be defined, thereby building a similar environment with vendor free templates. **Software deployment using Ansible may not work in the HPC clusters due to the restriction of root or superuser privileges which Ansible invokes package managers e.g. apt-get, yum, dnf or pkg with sudo command**. As a workaround, many HPC systems provide a user environment by modules, the software environment management, or virtualenv, the isolated directory for Python or RVM for Ruby but system software packages still need admin privileges to setup required libraries and software globally.

NIST Big Data Public Working Group (NBD-PWG) [40, 84] reported 51 use cases across nine application domains including Government Operation, commercial, Defense, Healthcare and Life Sciences, Deep Learning and Social Media, The Ecosystem for Research, Astronomy and Physics, Earth, Environmental and Polar Science and Energy to understand Big Data requirements and advance the development of big data framework. We ought to keep up the same effort to support scientific community in regard to analyzing data with modern technologies and the part of this

dissertation is gathering more use cases and requirements by reviewing publicly available big data applications.

3.2 Use Cases

3.2.1 Fingerprint Recognition

Fingerprint matching software [35, 36] has been developed by National Institute of Standards and Technology (NIST) with special databases to identify patterns of fingerprint. NIST Biometric Image Software (NBIS) includes MINDTCT, a fingerprint minutiae detector and BOZORTH3, a minutiae based fingerprint matching program to process biometric analysis. MINDTCT program extracts the features of fingerprint such as ridge ending, bifurcation, and short ridge from the FBI's Wavelet Scalar Quantization (WSQ) images and BOZORTH3 runs fingerprint matching algorithm with the images generated by MINDTCT as part of fingerprint identification processing [88]. In this use case, Apache Spark runs fingerprint matching on the Hadoop cluster with NIST Fingerprint Special Database 4 [87] and stores results in HBase with the support of NoSQL database, Apache Drill. Additional dataset from FVC2004 can be used as well with 1440 fingerprint impressions [69]. Individual software represents a stack or a role in the context in which a set of commands is listed to complete a software deployment. Suggested software stacks for Fingerprint matching are currently including: Apache Hadoop, Spark, HBase, Drill, and Scala.

3.2.2 Human and Face Detection with OpenCV

Human and face detection have been studied during the last several years and models for them have improved along with Histograms of Oriented Gradients (HOG) for Human Detection [28]. OpenCV is a Computer Vision library including the SVM classifier and the HOG object detector for pedestrian detection and INRIA Person Dataset [29] is one of popular samples for both training

and testing purposes. In this use case, Apache Spark on Mesos clusters are deployed to train and apply detection models from OpenCV using Python API. Individual software represents a stack or a role in this context in which a set of tasks to complete a software deployment is included. Suggested software stacks (Roles) for human and face detection with OpenCV are currently including: Apache Mesos, Spark, Zookeeper, OpenCV for HOG and Haar Cascades, and INRIA Person image files.

3.2.3 Twitter Live Analysis

Social messages generated by Twitter have been used with various applications such as opinion mining, sentiment analysis [73], stock market prediction [16], and public opinion polling [25] with the support of natural language toolkits e.g. nltk [14], coreNLP [70] and deep learning systems [61]. Services for streaming data processing are important in this category. Apache Storm is widely used with the example of twitter sentiment analysis, and Twitter Heron, Google Millwheel, LinkedIn Samza, and Facebook Puma, Swift, and Stylus are available as well [22]. Suggested software stacks (roles) for Twitter Live Analysis are currently including: Apache Hadoop, Storm, Flume, Twitter Heron, and Natural Language Toolkit (NLTK).

3.2.4 Big Data Analytics for Healthcare Data and Health Informatics

Several attempts have been made to apply Big Data framework and analytics in health care with various use cases. Medical image processing, signal analytics and genome wide analysis are addressed to provide efficient diagnostic tools and reduce healthcare costs [9] with big data software such as Hadoop, GPUs, and MongoDB. Open source big data ecosystem in healthcare is introduced [77] with examples and challenges to satisfy big data characteristics; volume, velocity, and variety [93]. Cloud computing framework in healthcare for security is also discussed with concerns about privacy [83]. Suggested software stacks (roles) for Big Data Analytics for Healthcare Data

and Health Informatics are currently including: Apache Hadoop, Spark, Mahout, Lucene/Solr and MLlib.

3.2.5 Spatial Big Data, Spatial Statistics and Geographic Information Systems

The broad use of geographic information system (GIS) has been increased over commercial and scientific communities with the support of computing resources and data storages. For example, Hadoop-GIS [6], a high performance spatial data warehousing system with Apache Hive and Hadoop, offers spatial query processing in parallel with MapReduce, and HadoopViz [32], a MapReduce framework for visualizing big spatial data, supports various visualization types of data from satellite data to countries borders. Suggested software stacks (roles) for Spatial Big Data, Spatial Statistics and Geographic Information Systems are currently including: Apache Hadoop, Spark, Mahout, MLlib and GIS-tools.

3.2.6 Data Warehousing and Data Mining

Researches in data warehousing, data mining and OLAP have investigated current challenges and future directions over big data software and applications [27] due to the rapid increase of data size and complexity of data models. Apache Hive, a warehousing solution over a hadoop [85], has introduced to deal with large volume of data processing with the other research studies [23, 56] and NoSQL platforms [24] have discussed with data warehouse ETL pipeline [51]. Suggested software stacks (roles) for Data Warehousing and Data Mining are currently including: Apache Hadoop, Spark, Mahout, Lucene/Solr, MLlib, MongoDB, Hive, and Pig.

3.3 Ecosystem Analysis

We believe that big data ecosystem consists of various software, applications and datasets on different platforms. To understand current activities on big data projects and provide recommended software components (roles) in big data, we conduct analysis on big data projects 1) from community (i.e. github), 2) and academia (i.e. Indiana University) regarding to the following entities: development language preference, library/package/tool dependencies, and sectors of public dataset source.

This effort will result in building recommended software components (roles) which supports most of functionality in a given big data applications.

3.3.1 Analysis on Big Data Projects from Community

Github.com has been used to provide version control and manage source code development along with diverse collaborators across countries. The popularity of github as a collaboration tool has been significantly increased and about 4 million repositories exist in 2016 with thousands of daily added repositories. To understand trends on big data software development from community, we conducted a survey of github repositories regarding to big data applications and tools. Every github repository has a description of a project and we searched them using topic keywords. For example, we collected github repositories for Face Detection with search keywords; face detection, face recognition, human detection, and person detection to conduct a survey on a series of questions regarding to 1) A development language distribution, 2) dependency of libraries and packages, and 3) sectors of public dataset. Actual source code of public github repositories are evaluated with the survey query data available on `https://github.com/lee212/bd_stats_from_github`. There are six topics of NIST Collection used in this analysis where N1: Fingerprint Matching, N2: Face Detection, N3: Twitter Analysis, N4: Data Warehousing, N5: Geographic Information

Topic	C++	Python	Java	Matlab	JS	C#	C	R	Ruby	Scala	Count*
Fingerprint (3.2.1)	15%	11%	13%	20%	3%	16%	8%	0%	1%	5%	43
Face (3.2.2)	26%	21%	12%	9%	7%	5%	2%	2%	1%	.02%	538
Twitter (3.2.3)	2%	35%	15%	.6%	9%	2%	1%	10%	3%	1%	1429
Warehousing (3.2.6)	3%	27%	18%	2%	10%	3%	1%	10%	4%	1%	3435
Geographic (3.2.5)	5%	15%	27%	4%	15%	3%	5%	7%	3%	16%	6487
Healthcare (3.2.4)	2%	13%	19%	2%	14%	5%	1%	10%	6%	2%	132

Table 3.1: Language Distribution of Topics related to those in the NIST collection on Github

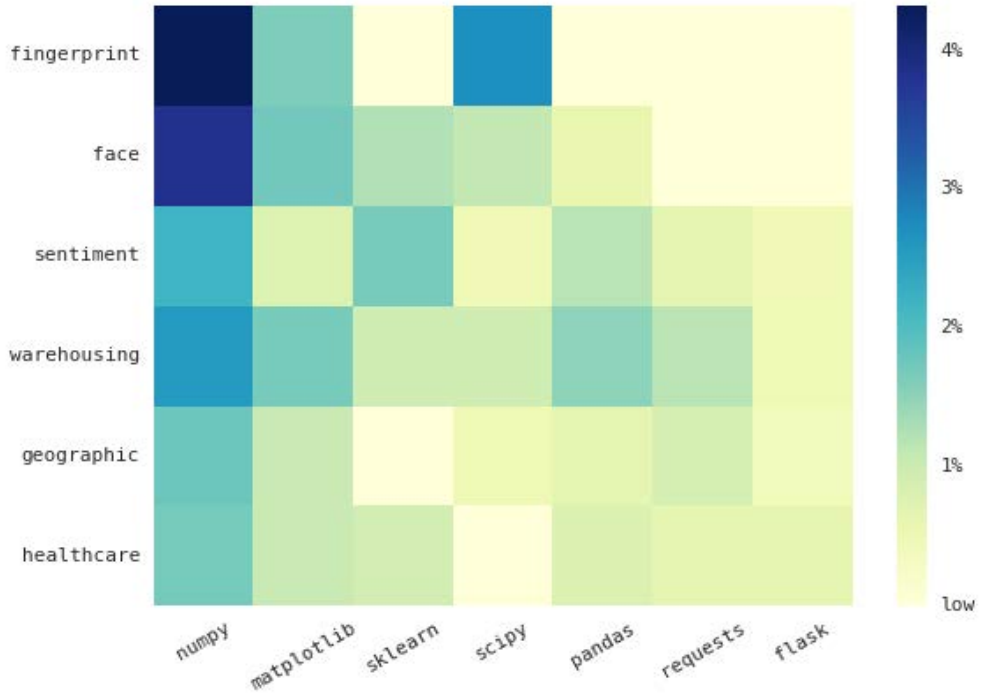
* Count: average number of github.com repositories.

Systems, and N6: Healthcare Data. In addition, a list of recommended components (roles) is created based on the survey results. Python and Java are most common languages among the six NIST projects (Table 3.1), although matlab is popular in the fingerprint project. We also noticed that scientific python packages are commonly used to enable numerical computation, data analysis and visualization for these big data applications (Figure 3.3). There are extra packages for each project (Table 3.2) that are not required in a default set of big data ecosystem but it is useful to indicate the dependency with a particular application. Tweepy, twitter API, is used in the twitter live analysis cases with NLTK, the natural language processing toolkit to complete sentiment analysis with tweets. Similarly, GIS projects use particular libraries for spatial analysis such as geopy and shapely. Each github project has different language preferences with various libraries and packages which shows recent activities, for example, deep learning software such as Keras, Theano, mxnet and Caffe is adopted among multiple projects.

Python Package	Description	Fingerprint	Face	Twitter	Warehousing	Geographic	Healthcare
cv2	OpenCV	✓	✓				
skimage	Image Processing		✓				
PIL	Python Imaging Library		✓				
caffe	Deep Learning		✓				
nltk	Natural Language Toolkit			✓			
tweepy	Twitter for Python			✓			
BeautifulSoup	Screen-scraping library			✓	✓		
gensim	Topic Modelling			✓	✓		
geopy	Geocoding library					✓	
shapely	Geometric Analysis					✓	
django	Web framework				✓		✓

Table 3.2: Additional Python packages found in NIST Collection

Figure 3.3: Scientific Python Packages used in NIST Projects (collected from Github)



3.3.2 Analysis on Big Data Projects from Academia

At Indiana University, two big data courses have been offered, Big Data Analytics and Applications and Big Data Open Source Software Projects. The class had been asked to develop a project with open source software, tools and datasets. Several implementations have been made and the survey of the programming languages, tools, and dataset are completed. Table 3.3 shows the language distribution for the big data Classes.

Table 3.3: Language Distribution for Big Data Classes from Indiana University

Class	Java	Python	R	C#	Projects Count
Fall '15	6	29	10	1	49
Spring '16	11	16	1	0	37
Fall '16	1	73	3	0	77

Package Dependencies

We had 37 final projects from Big Data Open Source Project Spring 2016 and Table 3.4 shows that tools used in the projects. Apache Hadoop is mostly used in conducting data analysis with a database support from HBase, Hive and HDFS. Python was the most preferred language in the course projects which resulted in high use of Spark in data processing with the python library, pyspark. One another observation is that Ansible, software deployment tool, had offered as a method of project deployment to ensure reproduceability.

Datasets

There were 49 class project in Big Data Analytics and applications Fall 2015, and use of 27 dataset are observed. Public dataset from industry was mainly used (44%) due to the interest on analytics from kaggle and twitter and availability e.g. amazon reviews and yelp reviews.

3.4 Discussion

Based on the analysis from community and academia, we observed that there are crucial software, dataset and analytics in big data ecosystem. We, therefore, offered deployable first-class roles which enable major functionality on big data processing and analysis. The software deployment is accommodated with Cloudmesh, Ansible, Chef, Puppet, Salt, OpenStack Heat, Microsoft Azure Template, and Amazon Cloudformation.

Finding relevant datasets for particular applications is another challenge for the big data ecosystem because of its difficulty of collecting data from different sources [60], complexity and diversity [55]. Community contributed lists of public datasets [26] provide structured information with a specific location to access data and a category to describe itself. We intend to generate linked json data for datasets and applications in big data ecosystem based on these lists because it connects

scattered data and software in an organized way. The whole categories of the surveyed data are available online: `https://github.com/lee212/bd_datasets` for the further discussion.

3.5 Conclusion

With the complexity of Big Data ecosystem, we demonstrated scripting software deployment using NIST Big Data use cases and implemented software stacks to deploy, configure, and run big data applications. We also describe resource provisioning by allocating and managing virtual clusters. A survey of big data ecosystem from open source projects available at GitHub and academic class projects via Indiana University provides an insight of building software stacks with latest tools. Our implementation of deploying Big Data software stacks proves that automated scripting deployment is capable of building Hadoop-based virtual clusters on clouds e.g. Microsoft Azure and Amazon EC2. DevOps tools e.g. Ansible show that it is flexible and easy to control software dependency along with infrastructure provisioning as an effort of implementing a software-defined system in terms of an environment management sub system.

Table 3.4: List of Common Packages used in Big Data Class Spring 2016

Package	Type	Use	Language	Count*
Hadoop	framework	parallel processing	Java	31
Ansible	tool	deployment	Python	26
Spark	framework	in-memory processing	Scala	14
HBase	database	NoSQL	Java	12
Pig	language	data abstraction	Java	11
Hive	database	SQL	Java	7
MongoDB	database	NoSQL	C++	7
Mahout	library	machine learning, data mining	Java	4
MLLib	library	machine learning	Java	4
OpenCV	library	computer vision	C++	3
Zookeeper	framework	directory service	Java	3
Tableau	tool	visualization	C++	3
D3.js	tool	visualization	Javascript	2
MySQL	database	SQL	C++	2
HDFS	database	distributed filesystem	Java	2

* Count: a number of class projects in a given tool/library/package

Chapter 4

Efficient Software Defined Systems using Containers

4.1 Introduction

Deployment for modern applications requires frequent changes for new features and security updates with a different set of libraries on various platforms. DevOps tools and containers are used to complete application deployments with scripts but there are problems to reuse and share scripts when a large number of software packages are required. For example, Ansible Galaxy - a public repository provides over ten thousand scripts (called roles) and Docker Hub has at least fourteen thousand container images but most of them are individualized and common libraries and tools are barely shared. This might be acceptable if a system runs only one or two applications without multi tenants but most systems in production need to consider how to run applications efficiently. Container technology i.e. Docker permits a repeatable build of an application environment using container image layers but redundant images with unnecessary layers are observed because of a stacked file system. In this chapter, we introduce two approaches about building Common Core Components (3C) in containers therefore building application environments is optimized and contents are visible in detail for further developments.

Common Core Components is a collection of libraries and tools which aims to share dependencies at one place thereby minimizing storage usage for container images. Docker stores container images efficiently and gains speedup in launching a new container instance because images on union mounts reuse same contents i.e. identical image layers over multiple containers without creating copies. The copy-on-write technique adds a new layer to store changes and keeps the original unchanged. In practice, however, many duplicates of package dependencies are observed between old and new

container images with version updates as well as containers in similar application groups. Docker images represent contents of image layers using a directed tree, and duplicates in child image layers can occur when a parent image layer is different although contents in child layers are same. This is normal in version control systems and the goal of 3C is to resolve these issues using dependency analysis and revision control functions. We notice that the build of Docker images is transparent through Dockerfile, a script for building a Docker image and Docker history metadata, therefore 3C is able to be established based on these information. The process of building 3C is following. First, installed packages are collected and then dependencies are analyzed. The two functions that we have chosen from version control systems; submodules and merge typically support unifying two separate repositories and branches. If there are containers that change software versions frequently but use same dependencies, the 3C with submodules provides an individual image layer to share dependencies but no changes in existing images. If there are containers that have similar interests but created by different users, the 3C with merge provides a new parent image layer to suggest common dependencies. The effectiveness and implementation of 3C are described in detail at the section 6.3.

We demonstrate that 3C optimizes both consuming disk space and detecting security vulnerability by determining shared components of containers and analyzing dependencies. 3C also suggests a collection of the most commonly required dependencies from High Performance Computing Enhanced Apache Big Data Stack (HPC-ABDS) [46] and survey data, where sampling is done from public Dockerfiles and project repositories. Performance comparison is presented to show efficiency regarding to disk space usage against existing container images. 3C achieves improvements in storing Nginx container images by 37.3% and detects 109 duplicate dependencies out of 429 from survey data of the HPC-ABDS streams layer with 50% of overlaps. We illustrate security vulnerabilities for Ubuntu 16.04 according to system packages and libraries.

4.2 Background

Reproducibility is ensured with container images which are stored in a stackable union filesystem, and "off the shelf" software deployment is offered through scripts e.g. Dockerfile to build an equivalent software environment across various platforms. Each command line of scripts creates a directory (called an image layer) to store results of commands separately. Container runs an application on a root filesystem merged by these image layers while a writable layer is added on top and other layers beneath it are kept as readable only, known as copy-on-write. The problem is that system-wide shared libraries and tools are placed on an isolated directory and it prevents building environments efficiently over multiple versions of software and among various applications that may use the same libraries and tools. We use collections of HPC-ABDS (Apache Big Data Stack) [46] and Github API to present surveyed data in different fields about automated software deployments. In this case, we collected public Dockerfiles and container images from Docker Hub and github.com and analyzed tool dependencies using Debian package information.

4.2.1 Software Deployment for dynamic computing environments

Software development has evolved with rich libraries and building a new computing environment (or execution environment) requires a set of dependencies to be successfully installed with minimal efforts. The environment preparation on different infrastructures and platforms is a challenging task because each preparation has individual instructions to build a similar environment, not an identical environment. The traditional method of software deployment is using shell scripts. A system package manager such as apt, yum, dpkg, dnf and make help to automate installing, compiling, updating and removing tools but shell scripts can be easily difficult to understand once it handles more systems and various configurations. A large number of packages are actively updated and added to communities and proper managing in a universal way is necessary to deal with them

sustainably. Python Package Index (PyPI) has 107,430 packages in 2017 with 40+ new packages on a daily basis. Public version control repository, Github.com has 5,649,489 repositories with about 20,000 daily added repositories. Most software packages, libraries and tools can be found on their website and using their API. DevOps tools i.e. Configuration management software supports automated installation with repeatable executions and better error handling compared to bash scripts but there is no industry standards for script formats and executions. Puppet, Ansible, Chef, CFEngine and Salt provide community contributed repositories to automate software installation, for example, Ansible Galaxy has 11,353 roles available, and Chef Supermarket has 3,261 cookbooks available although there are duplicated and inoperative scripts for software installation and configuration. Building dynamic computing environments on virtual environments is driven by these DevOps tools and container technologies during the last few years for its simplicity, openness, and shareability. Note that this effort is mainly inspired by the previous research activities [40–43, 46, 76].

4.2.2 Scripts

Building compute environments needs to ensure reproducibility and constant deployment consistently [50, 52]. Most applications these days run with dependencies and setting up compute environments for these applications requires an exact version of software and configure systems with same options. Ansible is a DevOps tool and one of the main features is software deployment using a structured format, YAML syntax. Writing Ansible code is to describe action items in achieving desired end state, typically through an independent single unit. Ansible offers self-contained abstractions, named Roles, by assembling necessary variables, files and tasks in a single directory. For example, installing software A or configuring system B can be described as a single role. Compute environments are supplied with several software packages and libraries and selectively combined

roles build compute environments where new systems require software packages and libraries installed and configured. Although the comprehensive roles have instructions stacked with tasks to complete a software deployment with dependencies, the execution of applications still need to be verified. In consequence, to preserve an identical results from the re-execution of applications, it is necessary to determine whether environments are fit for the original applications.

4.2.3 Containers with Dockerfile

Container technology has brought a lightweight virtualization with a Linux kernel support to enable a portable and reproducible environment across laptops and HPC systems. Container runtime toolkit such as Docker [71], rkt [3] and LXD [4] uses an image file to initiate a virtualized environment including necessary software packages and libraries without a hypervisor. These tools create an isolated environment on a same host operating system using the Linux kernel features such as namespaces, cgroups, seccomp, chroot and apparmor. Recent research [33] shows that containers outperform the traditional virtual machine deployments but running containers on HPC systems is still an undeveloped area. Shifter [57] and Singularity [64] have introduced to support containers on HPC with a portability and MPI support along with Docker images. These efforts will be beneficial to scientific applications to conduct CPU or GPU intensive computations with easy access of container images. For example, a neuroimaging pipelines, BIDS Apps [53], is applied to HPCs using Singularity with existing 20 BIDS application images and Apache Spark on HPC Cray systems [21] is demonstrated by National Energy Research Scientific Computing Center (NERSC) using shifter with a performance data of big data benchmark. Both researches indicate that workloads for scientific applications and big data are manageable by container technologies on HPC systems with a reproducibility and portability.

Listing 4.1: Dockerfile Example

```
FROM ubuntu:14.04

MAINTAINER Hyungro Lee <lee212@indiana.edu>

RUN apt-get update && apt-get install -y \\
      build-essential wget git
...
```

Dockerfile (See Listing 4.1) uses a custom template to describe installation steps of building Docker images in a bash-like simple format. There are certain directives to indicate particular objectives of the commands, for example, FROM indicates a base image to use and RUN indicates actual commands to run. When an image is being generated, each directive of Dockerfile creates a single directory to store execution results of commands. Meta-data of these directories is recorded in a final image to provide a unified logical view by merging them. The tag for an image is a reference for stacked image layers. For example in Listing 4.1, *ubuntu:14.04* is a tag to import stacked image layers of Ubuntu 14.04 distribution and the following directives i.e. *MAINTAINER* and *RUN*, will be added. This allows users to import other image layers and start building own images.

4.2.4 Environment Setup

Preparing environment is installing all necessary software, changing settings and configuring variables to make your application executable on target machines. Container technology simplifies these tasks using a container image which provides a repeatable and pre-configured environment to your application therefore you can spend more time on an application development rather than software installation and configuration. One of the challenges we found from container technologies in preparing environment is managing dependencies for applications. Container users who want to

Name	PCT1	PCT2	PCT3	PCT4	Description	Section	CT1	CT2	Dependencies	Size	Important
software-properties-common	0.01	0.06	0.02	0.03	manage the repositories that you install software from (common)	admin	8	4	python3-dbus, python-apt-common, python3-software-properties, gir1.2-glib-2.0, ca-certificates, python3:any, python3-gi, python3	9418 (630404)	optional
groovy	-	-	**0.01**	-	Agile dynamic language for the Java Virtual Machine	universe/devel	14	10	libbsf-java, libservlet2.5-java, antlr, libxstream-java, libcommons-logging-java, libjline-java, libasm3-java, libjansi-java, libregexp-java, libmockobjects-java, junit4, default-jre-headless, ivy, libcommons-cli-java	9729202 (3257906)	optional
libatlas-base-dev	-	**0.06**	-	-	Automatically Tuned Linear Algebra Software, generic static	universe/devel	2	8	libatlas-dev, libatlas3-base	3337570 (2690424)	optional
liblapack-dev	-	**0.03**	-	-	Library of linear algebra routines 3 - static version	devel	2	22	liblapack3, libblas-dev	1874498 (2000176)	optional
ruby	-	0.01	0.01	0.01	Interpreter of object-oriented scripting language Ruby (default version)	interpreters	1	987	ruby2.1	6026 (73880)	optional
libffi-dev	-	0.03	-	**0.01**	Foreign Function Interface library (development files)	libdevel	2	11	libffi6, dpkg	162456 (2101914)	extra
libssl-dev	0.12	0.07	0.01	0.03	Secure Sockets Layer toolkit - development files	libdevel	2	70	libssl1.0.0, zlib1g-dev	1347070 (1258956)	optional
net-tools	0.01	0.02	0.03	0.05	NET-3 networking toolkit	net	1	51	libc6	174894 (4788234)	important
nodejs	0.01	0.04	-	0.02	evented I/O for V8 javascript	universe/web	6	287	libssl1.0.0, libc6, libstdc++6, zlib1g, libv8-3.14.5, libc-ares2	683742 (7551922)	extra

Table 4.1: Common Debian Packages from Survey Data

(PCT1: Percentage by General Software, PCT2: Percentage by Analytics Layer, PCT3: Percentage by Data processing Layer, PCT4: Nosql Layer, CT1: Count of Dependencies, CT2: Count of Reverse Dependencies)

run applications with particular libraries have to find relevant container images otherwise they have to create a new image from scratch thereby brining all required tools and libraries. One possible solution for this problem is to offer a common core component (3C) when environment is being built. We noticed that there is a common list of libraries for particular type of applications based on the survey from Docker images and Dockerfile scripts. The idea is to offer curated collection of libraries for domain-specific applications and use the list of libraries surveyed from communities. For example, libraries for linear algebra calculation i.e. liblapack-dev and libopenblas-dev are commonly used for applications in the analytics layer of HPC-ABDS according to the survey (shown in Table 4.1). Additional package installation might be required if a suggested list of dependencies does not satisfy all requirements of an application.

4.2.5 Package Dependencies

Software packages have many dependencies especially if the packages are large and complex. Package management software e.g. apt on Debian, yum on CentOS, dnf on Fedora and pkg on FreeBSD automates dependency installation, upgrading or removal through a central repository package and a package database. The information of package dependencies along with version numbers controls a whole process of software installation and avoids version conflicts and breaks. In addition, reverse dependencies show which packages will be affected if the current package is removed or changed. *nodejs* and *ruby* in Table 4.1 have a few dependencies but a large number of reverse dependencies exist . Software incompatibility can easily occur to other packages if these packages are broken or missing. Figure 4.1 shows relations between dependencies (depends), reverse dependencies (rdepends), package size (size) and package size including dependencies (total_size) among six different sections. Interestingly the size of package itself does not increase when the number of dependencies are incremented but it shows positive correlation between the number of dependencies

and the total package size including dependencies. It explains that shared libraries are common for most packages to manage required files efficiently on a system. This is based on the survey of Docker scripts i.e. Dockerfile from public software repositories on github.com. Note that there are several package managers available on Linux distributions, such as dpkg, apt, yum and dnf.

4.2.6 Application Domains

Debian packages are categorized in 57 sections ranging from administration utilities (abbreviation is *admin*) to X window system software (abbreviation is *x11*) and it helps us to better understand the purpose of a package. An application typically requires several packages installed and a certain choice of packages is found in common according to interests of applications. SciPy [**?**] is a collection of python packages for scientific computing, for example, and the dependencies include a math library i.e. libquadmath0 - GCC Quad-Precision Math Library and basic linear algebra packages i.e. libblas3 - shared library of BLAS (Basic Linear Algebra Subroutines) and liblapack3 - Library of linear algebra routines 3. The classification of Big Data and HPC applications is well established in the HPC and Apache Big Data Stack (HPC-ABDS) layers [46]. Figure 4.2 shows six dependency sections for selected HPC-ABDS layers such as Layer 6) Application and Analytics - (Analytics with green dot), Layer 11B) NoSQL - (Nosql with purple dot) and Layer 14B) Streams - (Stream with beige dot). Library dependencies (4.2a) including development tools, utilities and compilers are observed in most layers as well as reverse dependencies (4.2b), especially in the analytics layer and the machine learning layer. Note that, the machine learning layer is not a part of HPC-ABDS but is manually added to demonstrate other interesting collections as an example. Sub groups of the library section will be necessary to identify a common collection of dependencies for the particular application domains in detail.

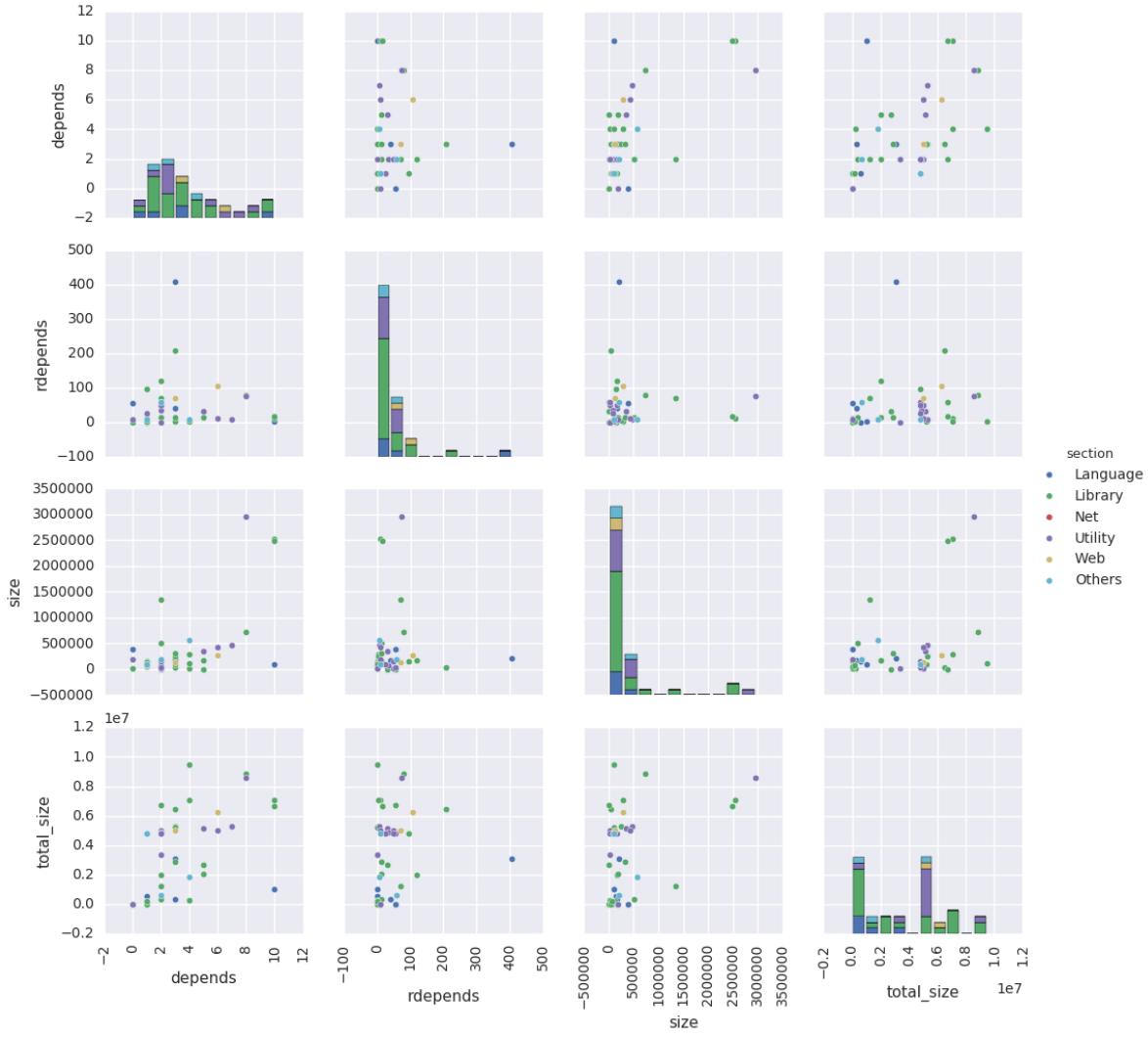
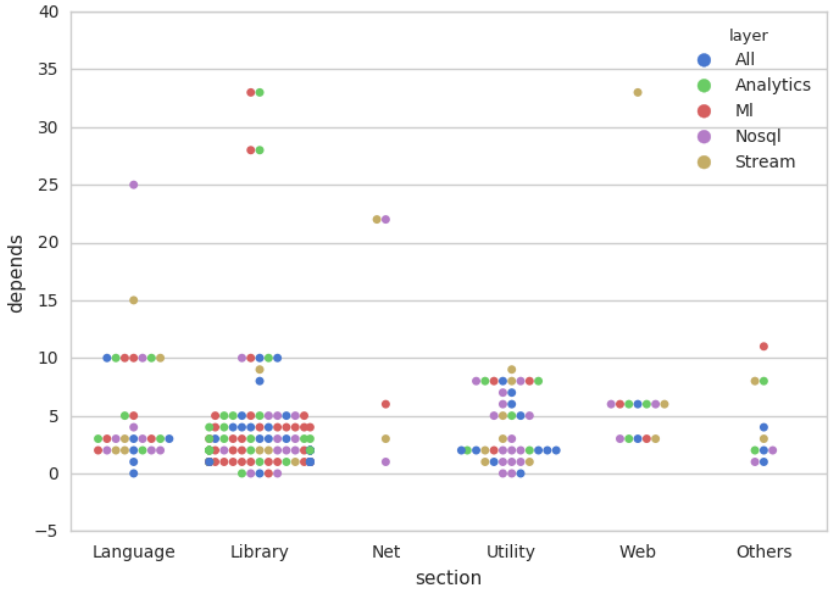
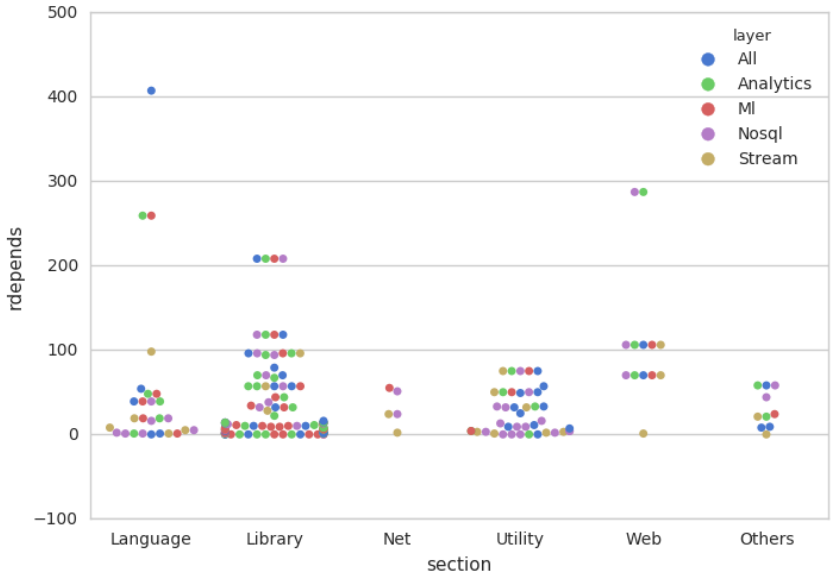


Figure 4.1: Debian Package Relations between Dependencies, Reverse Dependencies and Package Sizes (itself and including dependencies) for Popular Docker Images



(a) Dependencies



(b) Reverse Dependencies

Figure 4.2: Debian Package Dependencies for HPC-ABDS Layers

4.2.7 Docker Images on Union Mounting

Union mount implementations e.g. aufs and overlayfs enable Docker containers to have stackable image layers thereby ensuring storage efficiency for images where a base image layer includes common contents. Additional image layers only carry changes made to the base image while multiple containers share a same base image. This enables containers to reduce storage and booting up time when a new container starts. From a practical point of view, a base image is a set of image layers built from *scratch*, for a linux distribution with a version e.g. *ubuntu:latest, xenial, or 16.04* or *centos:latest or 7* which is a starting point of most images. Common tools or special packages can be added and declared as an another base image for a particular purpose, for example, NVIDIA's CUDA and cuDNN packages are defined as a base image for GPU-enabled computation including deep neural network on top of an Ubuntu or CentOS image. This approach is widely adopted because of the following reasons. First, "off the shelf" container images provide application environments to normal users and a standard collection of required software packages is built for communities of interest. It is also more convenient to update a single base image rather than multiple images, if there are changes to apply. Note that using a same base image reduces storage in its database and avoids duplicates. We see a flattened view of Docker images from Figure 4.3. Base images start from *scratch* as a first image layer and most applications are diverged out from base images.

4.3 Results

While there are advantages of using layered file systems for containers, we noticed that redundancy in storing Docker container images exists. The duplication of image contents occurs when an identical software installation completes with different parents of an image layer. As shown in Figure 4.4, tree structure is preserved to represent container images, for example, two images (#1 and #2)

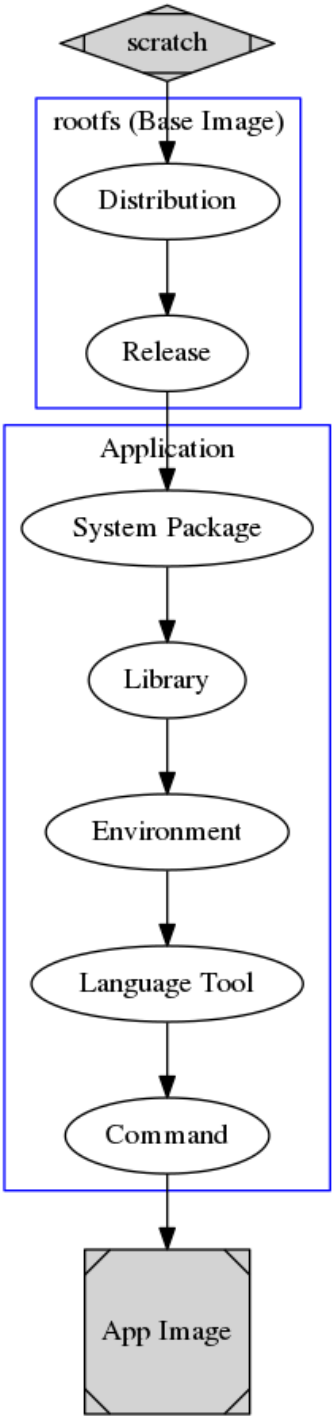


Figure 4.3: Dockerfile Workflow

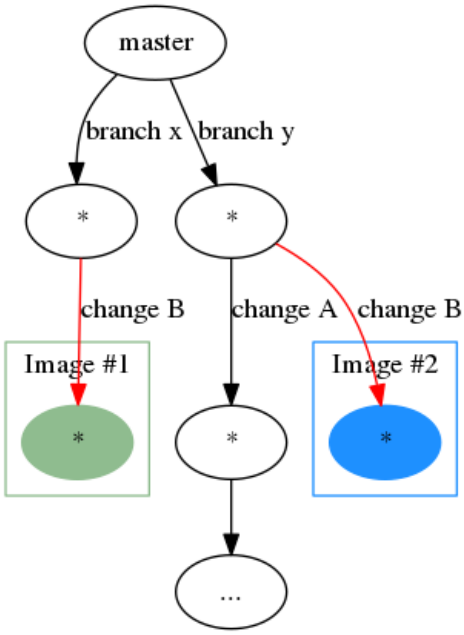


Figure 4.4: Union File System Tree

are identified as a distinct image although the change applied to them is indistinguishable. In this section, we demonstrate two approaches of reducing these duplicates using package dependencies.

4.3.1 Common Core Components

The general software deployment retrieves and installs dependencies to complete installation and ensure proper execution of software on a target machine. As shown in Figure 4.5, Nginx, a lightweight

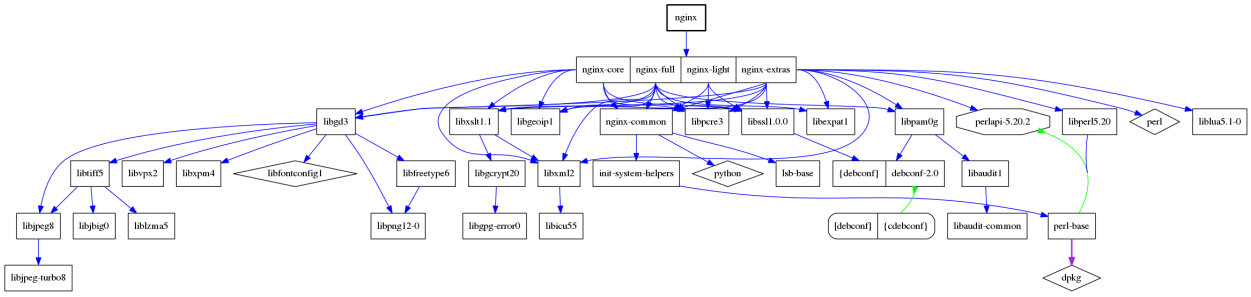


Figure 4.5: Nginx Debian Package Dependencies

HTTP server, requires 40 more libraries and tools installed, although Nginx itself is only about 3MB of an installed size. We identify these package dependencies and define them as common Core Components (3C).

4.3.2 Approach I: Common Core Components by Submodules

In version control systems, submodules keep repository commits separate but allow cloning other repositories in a sub directory. With submodules, common core components (3C) can be dispatched but in a separated image layer (See Figure 4.6). This approach lets you include an image layer without concerning a parent image layer and reduces duplicates without creating a new base image. 3C is supposed to contain dependencies for application and we can find out the dependency information after reviewing current Docker images with Dockerfiles and searching package manager databases. Dockerfile is a text file and a blueprint of building an image and installation commands are recorded to replicate an image anytime. Dockerfile has *RUN* directives to execute commands and package manager commands i.e. *apt-get* and *yum* are executed with *RUN* to install libraries and tools. Dependencies of these libraries and tools are described in a package manager cache file (Packages) and stored in its internal database. 3C is built by looking up dependency information from the database with package keywords obtained from Dockerfile. Docker history can be used to examine image construction or composition if Dockerfile is not available. In Figure 4.7, we created Nginx-3C (about 59.1MB) and re-generated Nginx Docker images including the new 3C. The current Nginx Docker has 9 individual images (in total 1191.5MB) among various versions of Nginx ranging from 1.9 to 1.13. The base image also varies from Debian 9 (in a slim package) to Debian Jessie 8.4 and 8.5. Whereas the size of new images including Nginx-3C increase about 2.9MB per each version change. The accumulated size of new images is 747.1MB in total to provide 9 individual Nginx images from version 1.9 to 1.13. 37.3% improvements regarding to storing

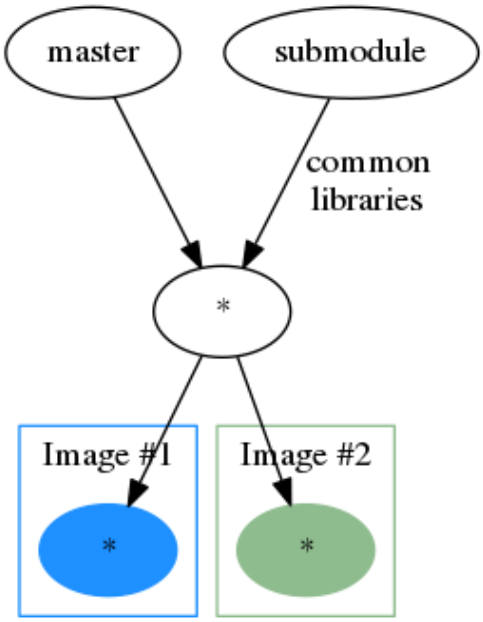


Figure 4.6: Common Core Components by submodules

Docker images is observed compared to the current Nginx Docker images. We notice that 3C by submodules reduce duplicates of contents, especially if software changes its versions but uses equivalent libraries. Nginx 1.9.0 and 1.13.0 have similar constraints of dependencies including version numbers. According to the Debian package information, the similar constraints are following: C library greater than or equal to 2.14 (libc6), Perl 5 Compatible Regular Expression Library greater than or equal to 1:8.35 (libpcre3), Secure Sockets Layer toolkit greater than or equal to 1.0.1 and zlib compression library greater than or equal 1:1.2.0 (zlib1g). Backward compatibility of libraries is ensured for general packages therefore 3C with the latest version of dependencies may cover most cases.

4.3.3 Approach II: Common Core Components by Merge

The goal of this approach is preparing compute environments on the premises with domain specific common core components merged into a base image. New base images are offered with the common

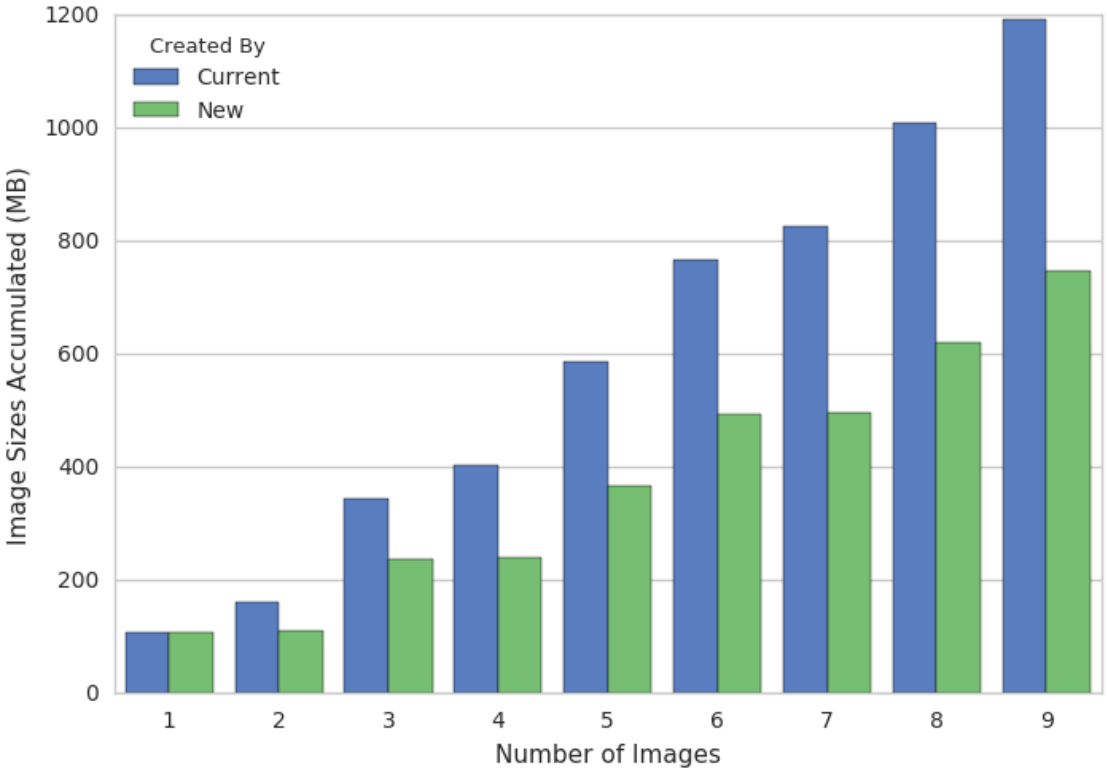


Figure 4.7: Comparison of Container Images for Nginx Version Changes

(Current: Built by Official Dockerfiles, New: Built by Common Core Components)

core components of applications. Similar application images (such as Image #1 and #2) in Figure 4.8 branched out from a same parent image layer. The storage might not be saved if not many images refer a same master image. One of the benefits of this approach is updating base images. Newly discovered components or vulnerable packages are updated and included through updates. Once a number of images sharing a same base image incremented, an additional survey can be conducted to follow trends of development tools and applications. In addition to that, outdated packages can be removed from the 3C. Docker offers 'bring-your-own-environment' (BYOE) using Dockerfile to create images and users can have individual images by writing Dockerfile. We observe that developers and researchers store Dockerfile on a source code version control repository i.e. github.com along with their applications. Luckily, GitHub API offers an advanced keyword search for various use and Dockerfile in particular domains is collected using API tools. Besides, we did a survey of package dependencies for application domains using the collection of HPC-ABDS and built 3C according to the survey data. To construct suitable environments with minimal use of storage, finding an optimal set of dependencies per each domain is critical. As shown in Figure 4.9, we found that relations between the size of components and the number of components as well as the percentage of common components among images. The first subplot for the streams layer shows that the size of most common components (between 40% and 100%) is increased slowly compared to the least common components. Based on the sample data, 109 out of 429 packages are appeared 50% of Docker images in the streams layer. Other layers of HPC-ABDS are also examined.

4.4 Discussion

We achieved application deployments using Ansible in our previous work [5]. In the DevOps phase, configuration management tool i.e. Ansible automates software deployment to provide fast delivery process between development and operations [31] but preserving environments for applications is

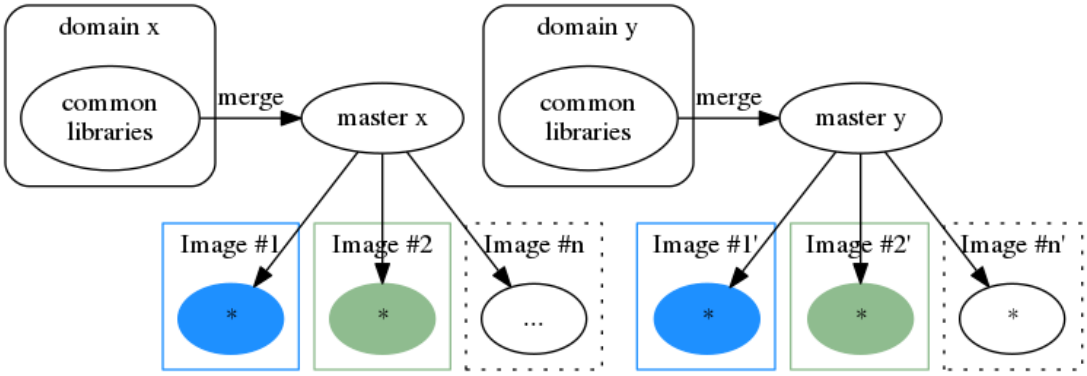


Figure 4.8: Common Core Components by merge

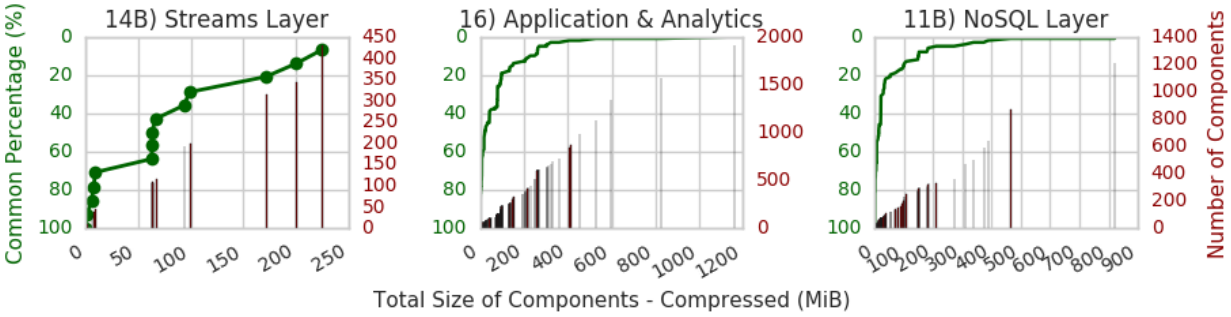


Figure 4.9: Common Core Components for HPC-ABDS

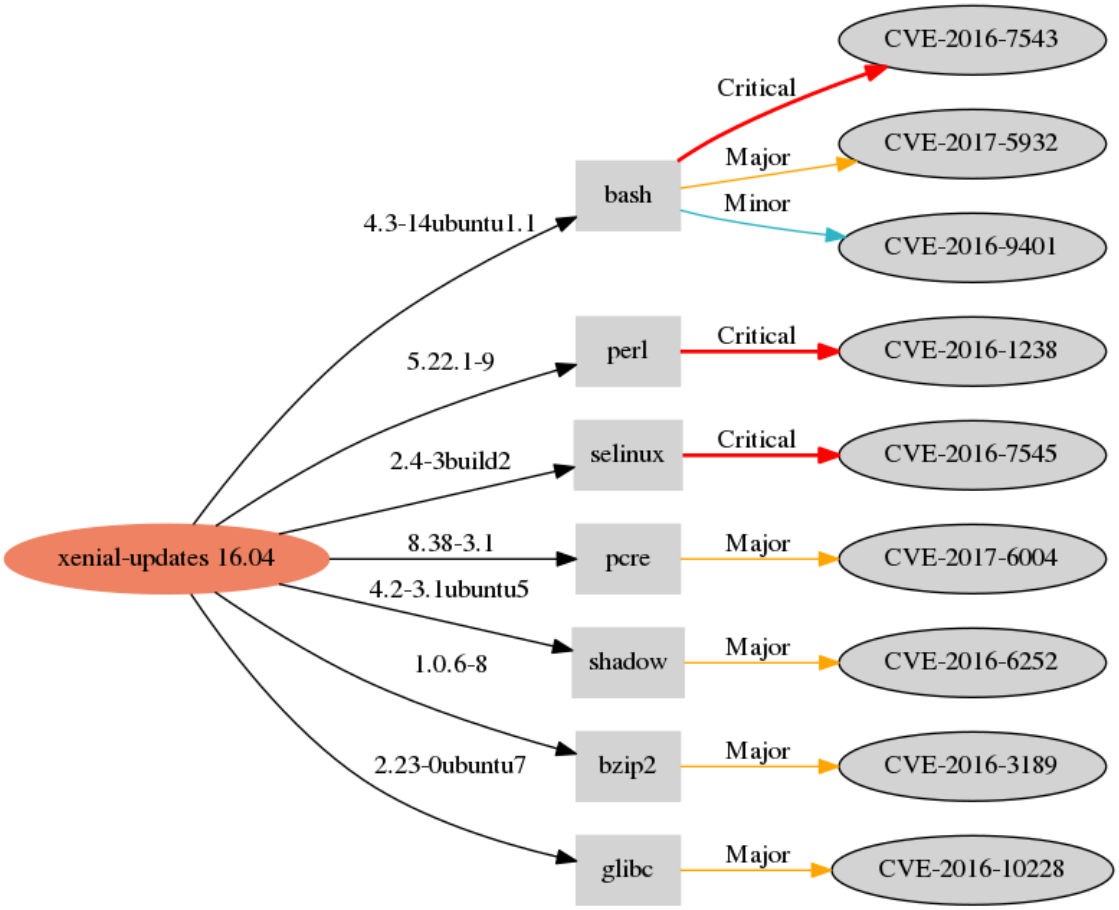


Figure 4.10: Example of Security Vulnerabilities for Ubuntu 16.04 based on Libraries

Table 4.2: NIST Big Data Projects

ID	Title
N_1	Fingerprint Matching
N_2	Human and Face Detection
N_3	Twitter Analysis
N_4	Analytics for Healtcare Data / Health Informatics
N_5	Spatial Big Data/Spatial Statistics/Geographic Information Systems
N_6	Data Warehousing and Data Mining

not ensured unless all required repositories are preserved. Linux containers resolve this problem but their scripts are not organized like DevOps tools. Instructions of DevOps tools are written in structured document formats i.e. YAML, JSON, and Ruby DSL, and there are benefits of using DevOps scripts like Ansible Roles that we wrote in our previous work. Various terminologies i.e. recipes, manifests, and playbooks are used to manage systems and deploy software but all of them have similar concepts and abstract levels. We also notice that these scripts can be converted to build container images, and vice versa if any of DevOps scripts need to be called in building compute environments. For example, Table 4.3 shows that 27 Ansible roles are created to deploy software components among six NIST use cases in Table 4.2. Some of the roles such as Apache Hadoop and Spark are shared frequently and we intend to provide more roles in building Big Data applications. With the experience from NIST projects [5], a few challenging tasks are identified in DevOps tools, a) offering standard Ansible Roles to ease application development with different tools, and b) integrating container technologies towards application-centric deployments. Infrastructure provisioning need to be integrated to avoid resource underutilization. We defer these considerations to future work.

Table 4.3: Technology used in a subset of NIST Use Cases. A ✓ indicates that the technology is used in the given project. See Table 4.2 for details on a specific project. The final row aggregates ✓ across projects.

ID	Hadoop	Mesos	Spark	Storm	Pig	Hive	Drill	HBase	Mysql	MongoDB	Mahout	D3 and Tableau	nltk	MLlib	Lucene/Solr	OpenCV	Python	Java	Ganglia	Nagios	zookeeper	AlchemyAPI	R
N_1	✓		✓			✓	✓	✓	✓									✓	✓	✓	✓		
N_2		✓	✓													✓	✓				✓		
N_3				✓				✓		✓		✓	✓				✓	✓			✓	✓	✓
N_4	✓		✓					✓			✓	✓		✓	✓			✓			✓		
N_5	✓		✓								✓	✓		✓				✓					
N_6	✓		✓		✓	✓		✓		✓	✓	✓		✓	✓			✓			✓		
count	4	1	5	1	1	2	1	4	1	2	3	4	1	3	2	1	2	5	1	1	5	1	1

4.5 Related Work

4.5.1 Template-Based Software Deployment

Template deployment is a means of installing software and building infrastructure by reading instructions written in a templating language such as YAML, JSON, Jinja2 or Python. The goal of a template deployment is to offer easy installation, repeatable configuration, shareability of instructions for software and infrastructure on various platforms and operating systems. A template engine or an invoke tool is to read a template and run actions defined in a template towards target machines. Actions such as installing software package and setting configurations are described in a template with its own syntax. For example, YAML uses spaces as indentation to describe a depth of a dataset along with a dash as a list and a key-value pair with a colon as a dictionary and JSON uses a curly bracket to enclose various data types such as number, string, boolean, list, dictionary and null. In a DevOps environment, the separation between a template writing and an execution helps Continuous Integration (CI) because a software developer writes deployment instructions in a template file while a system operations professional executes the template as a cooperative effort. Ansible, SaltStack, Chef or Puppet is one of popular tools to install software using its own templating language. Common features of those tools are installing and configuring software based on definitions but with different strategies and frameworks. One observation is that the choice of implementation languages for those tools influences the use of a template language. The tools written by Python such as Ansible and SaltStack use YAML and Jinja which are friendly with a Python language with its library support whereas the tools written by Ruby such as Chef and Puppet use Embedded Ruby (ERB) templating language.

4.5.2 Linux Containers on HPC

The researches [30, 47, 49] indicate the difficulty of software deployments on High Performance Computing (HPC). Linux containers is adopted on HPC with the benefits of a union file system, i.e. Copy-on-write (COW) and a namespace isolation and is used to build an application environment by importing an existing container image [57, 64, 74]. The container runtime tools on HPC e.g. Singularity, Shifter and chroot import Docker container images and wish to provide an identical environment on HPC as one on other platforms.

4.6 Conclusion

We presented two approaches to minimize image duplicates using package dependencies, named Common Core Components (3C). The current stacked Docker images create redundancies of storing contents in several directories when software packages are installed with different parent image layers and we build dependency packages that mostly shared with other images and provide where it needs. First approach is building 3C based on the analysis of current Docker images and scripts i.e. Dockerfile and combines with a master image using submodules. This is useful where software is updated frequently with new versions but equivalent dependencies are shared. In our experiment, Nginx with 3C shows 37.3% improvements in saving image layers compared to the current Docker images. The other approach is building 3C based on the surveyed data for application domains and provides a certain set of dependencies to provide a common collection for various applications.

Chapter 5

Performance Evaluation of Event-driven Computing

5.1 Introduction

Cloud Computing offers event-driven computing for stateless functions executable on a container with small resource allocation. Containers are lightweight which means that it starts in a second and destroys quickly whereas a software environment for applications is preserved in a container image and distributed over multiple container instances. This elastic provisioning is one of the benefits that cloud computing takes along with its ease of use while traditional virtual machines on Infrastructure as a Service (IaaS) need some time to scale with system settings, i.e., an instance type, a base image, a network configuration and a storage option.

Most services in the cloud computing era, pay-as-you-go is a primary billing method in which charges are made for allocated resources rather than actual usage. Ephemeral computing may provide a dynamic computing environment because it is tuned for the execution time of containers without preparing procured resources that never used. Amazon, for example, recently applied Big Data and Machine Learning platforms to EC2 services as Google Compute and Microsoft Azure have started similar cloud services with dynamic and lightweight software environment.

Temporal cloud computing emphasizes no infrastructure configuration along with the preparation of computing environments. Fox et al [44] defines event-driven computing among other existing solutions, such as Function-as-a-Service (FaaS) and we see computing environments offer various options using microservice in which event is an occurrence generated by other systems and resource and microservice are described as a formal syntax written in a programming function. New record on a database, deletion of object storage, or a notification from the Internet of Things devices is an

example of various events and the event typically contains messages to be processed by single or multiple event handlers. Sometimes an event is generated at a particular interval of time which is predictable, but in many cases, significant numbers of event messages need to be processed at scale instantly. Horizontal scaling for processing concurrent requests is one of the properties of cloud-native applications [48] which have practical approaches and designs to build elastic and scalable systems. Data processing software (ExCamera [38], PyWren [58]) for video rendering and Python program recently show that large loads on the event handlers can be ingested on the computing by using concurrent invocations. We also understand that namespaces and control groups (cgroups) offered by containers power up event-driven computing with resource isolation to process dynamic applications individually, but provisioning a thousand of instances within a few seconds.

A new event message is processed on a function instance isolated by others, and multiple instances are necessary when several event messages are generated at the same time. Event messages generated by mobile applications, for example, are lightweight to process but the quantity of incoming traffic is typically unpredictable so that such applications need to be deployed on a particular platform built with dynamic provisioning and efficient resource management in which event-driven computing aims for [8]. We may observe performance degradation if a single instance has to deal with multiple event messages with a heavy workload in parallel. Unlike IaaS, the cost of instantiating a new instance is relatively small, and an instance for function execution is short-lived on event-driven computing thus it would have demanded to process concurrent function invocations using multiple instances like one instance per request. Some applications that can be partitioned into several small tasks, such as embarrassingly parallel, may take advantage of the concurrent invocations on event-driven computing in which horizontal scaling is applied to achieve the minimal function execution time required to process the distributed tasks.

In this chapter, we evaluate event-driven computing environments invoking functions in parallel

to demonstrate the performance of event-driven computing for distributed data processing on a given software environment. We measure the performance of deployed applications using functions for CPU, Memory and I/O bound tasks in which helps describing bottlenecks and behaviors of the computing model. Continuous deployment and continuous integration (CD/CI) are often referred to address rapid software development issues with dependencies, and our experiment was conducted in the same manner to ensure proper integration between software environment and compute resources. The remaining sections provide a summary of the event-driven model over IaaS by big data experiments and new features are explained.

Event-driven computing environments with concurrent invocations may support distributed data processing with its throughput, latency and compute performance at scale [66]. There are certain restrictions that we must be aware of before implementing a event-driven function, for example, event handler types are a few; HTTP, object storage, and database in common, memory allocation is small; 512MB to 3GB memory allowance per a container, function execution time is allowed only in between 5 minutes and 10 minutes and a 500MB size of a temporary directory is given. In the following sections, we show our results towards Amazon Lambda, Google Functions, Microsoft Functions, and IBM OpenWhisk Functions with its elasticity, concurrency, cost efficiency and, functionality to depict current event-driven environments in production. Big Data Benchmark from AmpLab and TensorFlow ImageNet examples are included as a comparison of costs and computation time between event-driven computing and virtual machines as well.

5.2 Evaluation

We evaluate event-driven computing environments on the throughput of concurrent invocation, CPUs, the response time for dynamic workload, runtime overhead, and I/O performance. We also compare cost-effectiveness, event trigger throughput, and features using a set of functions

written by supported runtimes such as nodeJS, Python, Java, and C#. Each provider has different features, platforms, and limitations, so we tried to address the differences and find similarities among them. Some of the values may not be available because of two reasons, an early stage of the event-driven environments and limited configuration settings. For example, Microsoft Azure runs Python 2.7 on Windows NT as an experimental runtime language thus some libraries and packages for data analysis are not imported, e.g., TensorFlow library with Python 3.5 or higher, and Google Functions is in a beta version which only supports NodeJS, a javascript runtime although Python is internally included in a function instance. 512MB memory limit on IBM OpenWhisk prevents running TensorFlow ImageNet example which requires at least a 600MB size of memory to perform image recognition using trained models. New recent changes are also applied in our tests such as 3008MB memory limits on Amazon Lambda, and Java runtime on Microsoft Azure Functions. All of the evaluations were performed using 1.5GB memory allocation except IBM with 512MB and 5 minutes execution timeout. We use the Boto3 library on Amazon Lambda to specify the size of a concurrent function invocation, and HTTP API for Microsoft Azure and IBM OpenWhisk. We use object storage to invoke Google Functions as well. We completed benchmarks using the set of functions written by NodeJS 6.10, Java, C#, and Python 3 and 2.7.

5.2.1 Concurrent Function Throughput

Function throughput is an indicator of concurrent processing because it tells how many function instances are supplied to deal with extensive requests. Asynchronous, non-blocking invocations are supported by various methods over the providers. Amazon SDK (Boto3) allows to invoke a function up to an account's concurrent limit, and S3 object storage or DynamoDB database is an alternative resource to invoke a function in parallel. Google Functions only allows for concurrent execution by a storage bucket and a rate of processing event messages varies on the length of the

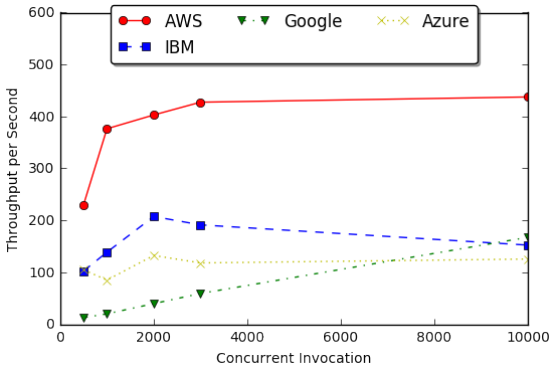


Figure 5.1: Function Throughput on Concurrent Invocations

message and its size. Microsoft Azure Functions also scales out automatically by its heuristic scale controller. IBM OpenWhisk does not seem to provide scalability unless functions are manually invoked as a workaround. We had to create a thousand of functions with an identical logic but a different name and treat them like invoking a single function in parallel. Figure 5.1 is a throughput result per second over the four event-driven providers from 500 to 10000 concurrent invocations. Amazon Lambda generates about 400 throughputs per second in average, and AWS quickly reaches its maximum throughput from a small number of concurrent invocation (1000). IBM OpenWhisk and Microsoft Azure Functions show similar behavior in reaching the best throughput at 2000 invocations and decreasing slowly over increased invocations. Google Functions shows a slow rising but steady progress of throughput over increased invocations. Throughput at ten thousands of invocations on Google Functions is about 168 per second which is better than IBM and Azure.

5.2.2 Concurrency for CPU Intensive Workload

Multiplying two-dimensional array requires mostly compute operations without consuming other resources thus we created the matrix multiplication function written in a JavaScript to stress CPU resources on a function instance with concurrent invocations. Figure 5.2 shows an execution time of the function between 1 and 100 concurrent invocations. The results with 1 concurrent invocation

which is non-parallel invocation are consistent whereas the results with 100 invocations show the overhead of between 28% and 4606% over the total execution time. The results imply that more than one invocation was assigned to a single instance which may have to share allocated compute resources, i.e., two CPU-intensive function invocations may take twice longer by sharing CPU time in half. For instance, the median of the function execution time on Amazon Lambda (3.72 seconds) is about twice the non-concurrent invocation (1.76 seconds).

19.63 gigaflops are detected on AWS Lambda with the 1.5GB size of memory configuration (whereas 40 gigaflops are measured with 3GB memory), but it can reach more than a teraflop when a fleet of containers are provisioned for concurrent invocations. event-driven platform allocates computing resources based on the number of requests which shows to a peak double-precision floating point performance of 66.3 TFLOPs when an Amazon Lambda function is invoked concurrently. Table 5.1 is the result of invoking three thousands of functions on event-driven functions which indicates proportional between the number of functions and the aggregated flops. 66.3 teraflops are relatively good performance. For example, Intel six-core i7-8700K generates 32 gigaflops, and the latest NVIDIA TESLA V100 GPU delivers 7.8 teraflops for a double precision floating point. In the comparison of the total of TFLOPS, AWS Lambda generates 5-7 times faster computing speed than others. Azure Functions, IBM OpenWhisk, and Google Functions are in either a beta service or an early stage of development; therefore, we expect that these results will need to be revisited in the future.

5.2.3 Concurrency for Disk Intensive Workload

A function in event-driven computing has a writable temporary directory with a small size like 500MB, but it can be used for various purposes, such as storing extra libraries, tools, and intermediate data files while a function is running. We created the function which writes and reads

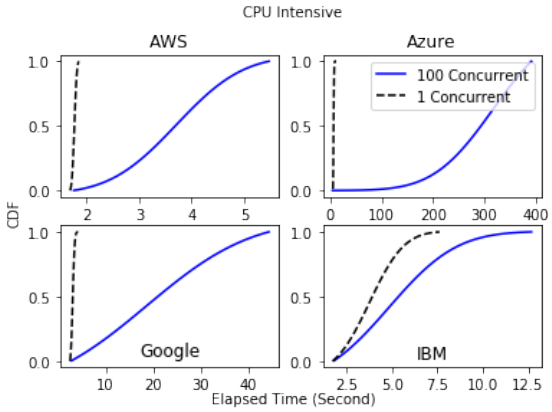


Figure 5.2: Concurrency Overhead with CPU Intensive Function

Table 5.1: CPU Performance

Provider	GFLOPS per function	TFLOPS in total of 3000
AWS	19.63	66.30
Azure	2.15	7.94
Google	4.35	13.04
IBM	3.19	12.30

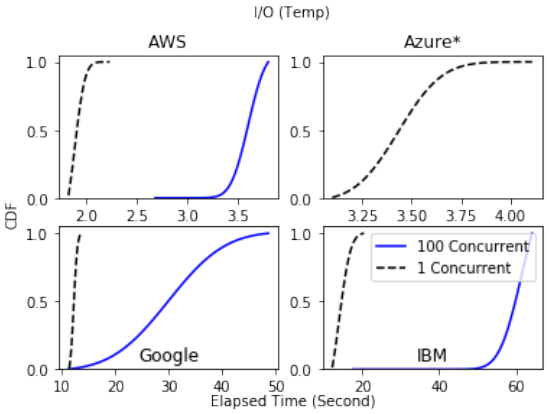


Figure 5.3: Concurrency Overhead with File I/O Intensive Function

files in the temp directory to stress a file I/O. 100MB size of a file is written by a random size of blocks with fsync to ensure all buffered file objects are transferred to the disk. Reading the file is done by random offset blocks with 512 bytes read. We do not have information of the actual device hardware type of the temporary directory we tested. Google claims that the temporary directory exists on memory which consumes allocated memory size of a function but we do not find information from other providers whether they mount it with a persistent storage device like HDD or SSD. The measured I/O performance toward a temporary directory is shown in Figure 5.3 with concurrent invocations. The results with 100 invocations show that Amazon generates an execution time overhead of 91%, Google generates the overhead of 145% and IBM generates the overhead of 338% whereas Microsoft Functions fail to complete function invocations within the execution time limit, 5 minutes. The median speed of the file read and write is available in the Table 5.2. Reading speed on Azure Functions is the most competitive among others although it is not measured on 100 concurrent invocations due to the storage space issue. Writing a file between 1 and 100 concurrent invocations is slightly worse compared to reading, the overhead of 110% on Amazon Lambda, 164% on Google Functions and 1472% on IBM OpenWhisk exist whereas the writing speed on Amazon Lambda is 11 to 78 times faster than Google and IBM when 100 concurrent invocations are made.

Table 5.2: Median Write/Read Speed (MB/s)

Provider	100 Concurrent		1 Concurrent	
	Write	Read	Write	Read
AWS	39.49	92.95	82.98	152.98
Azure	-	-	44.14	423.92
Google	3.57	54.14	9.44	55.88
IBM	0.50	33.89	7.86	68.23

5.2.4 Concurrency for Network Intensive Workload

Processing dataset from dynamic applications such as big data and machine learning often incur significant performance degradation in congested networks due to large transactions of file uploading and downloading. If such activities are distributed at multiple locations, network delays can be easily mitigated. Containers for event-driven functions tend to be deployed at different nodes to ensure resource isolation and efficient resource utilization and this model may help resolve this issue especially when functions are invoked in parallel. We created a function which requests data size of 100 megabytes from object storage on each service provider thus a hundred concurrent invocations create network traffic in a total of 10 gigabytes. Figure 5.4 shows delays in the function execution time between 1 and 100 concurrent invocations. We find that Google Functions has a minimal overhead between 1 and the 100 concurrency level whereas Amazon Lambda is four times faster in loading data from Amazon object storage (S3) than Google object storage. Microsoft Azure Functions fails to get access of data from its blob storage at 100 concurrency level, and we suspect it is caused by the experimental runtime, i.e. Python 2.7 or a single instance for multiple invocations. Default runtime such as C# and F# may support scalability better than the other runtime under development on Microsoft Azure Functions.

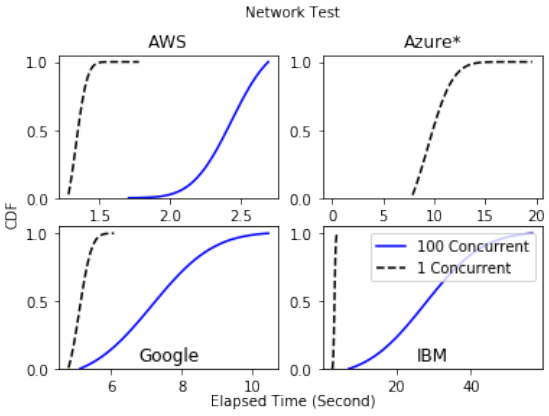


Figure 5.4: Concurrency Overhead with Downloading Objects Function

5.2.5 Elasticity

A dynamic application performing latency-sensitive workloads needs elastic provisioning of function instances otherwise overhead and failure would be observed during the processing of workloads. We assume that event-driven computing scales out dynamically to provide additional compute resources when the number of concurrent invocations is increased rapidly. We created the simple function that takes less than 100 milliseconds, and the function was invoked with different concurrent sizes ranging from 10 to 90 over time resulting in about 10 thousands of the total invocations within a minute. With this setup, we expected to observe two values; delays of instantiating new instances (which also called cold start) and a total number of instances created during this test. We believe that it explains whether elasticity mechanisms on event-driven computing is efficient or not with regarding provisioning and resource utilization. Delays in processing time would be observed when existing function instances are overloaded and new instances are slowly added which may cause performance degradation in the entire invoked functions. Figure 5.5 shows different results among the event-driven providers with the same dynamic workloads over time. The line with a gray color indicates the number of concurrent invocations per 200ms time window which completes ten thousand function executions within a minute and the changes of ± 3 *to* ± 30 concurrent sizes was

made in each invocation to measure horizontal scaling in/out. We observed that new function instances were added when a workload jumps to higher than the point that existing instances can handle and the increased number of function instances stay for a while to process future requests. We find that Amazon and Google support elasticity well in which the 99th percentile of the function execution time is below 100 and 200 milliseconds whereas both IBM and Azure show significant overhead at least two times bigger than others if we compare the 99th percentile of the execution time. The number of instances created for this workload was 54, 10, 101 and 100 in the order of Amazon, Azure, Google, and IBM. If there is a new request coming in while a function is in processing current input data, Amazon provides an additional instance for the new request whereas others decide to increase the number of instances based on other factors, such as CPU load, a queue length, an age of a queued message, which may take some time to determine. The function we tested uses the NodeJS runtime and scalable trigger types, but we would consider other runtimes such as C# and Java and other triggers like databases to see if it performs better in dealing with the dynamic workload. Each event-driven provider uses different operating systems and platforms and it seems certain runtimes and triggers have better support in handling elasticity than others. For example, Azure Queues has the 32 maximum batch size to process in parallel and Azure Event Hubs doubles the limit. Table 5.3 contains actual numbers we measured during this test and the function execution time which is represented by blue dots in the figure would expect to take less than 0.1 second in a single invocation, but there are overhead when the workload is increased in which the standard deviations and 99th percentile indicate in the table. It explains that the increased number of instances should be available instantly with additional amounts of computing resources to provide enough capacity for the future demands.

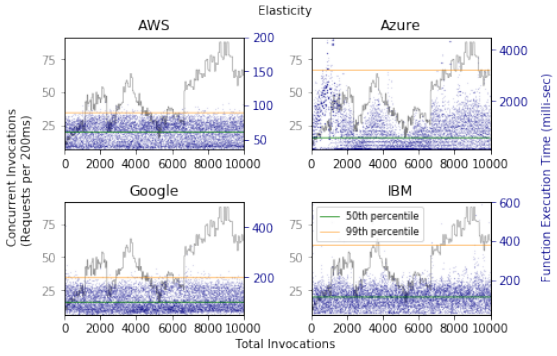


Figure 5.5: Response Time for Dynamic Workload

Table 5.3: Execution Time (milli-sec) on Elasticity

Provider	MED	SD	1st PCTL	90th PCTL	99th PCTL
AWS	61.08	14.94	35.08	78.99	89.41
Azure	574.0	747.33	118.0	1808.30	3202.0
Google	101.0	38.75	57.0	162.0	198.0
IBM	112.0	142.23	31.0	177.0	378.79

MED = Median, SD = Standard Deviation, PCTL = Percentile

5.2.6 Continuous Deployment and Integration

*Dev*elopment and *Op*eration*s* (DevOps) paradigm is applicable to event-driven functions in enabling continuous delivery and integration while functions are in action. Functions ought to be often changed for bug fixes and updates, and a new deployment of functions should not affect existing workloads. In this section, we measure failures and delays of function executions when code blocks are updated and function configurations are changed where the transition to the next version of a function is explained in the context of concurrent invocations. We started with 10 concurrent invocations with a simple NodeJS function which takes a few seconds to complete and made a total of 500 function executions within 10 seconds. Two events were made during this experiment. First, a change of source code was made before the first 200 invocations completed and the second event with new function configurations such as updates on timeout value and the size of memory allocation was made in the next 200 invocations. We also prepared the function with a warmed-up instance by executing the function multiple times before this experiment to ensure that function instances are initialized and ready to process a certain load. Figure 5.6 shows different behaviors in dealing with those events. A gray dot indicates an existing function instance and green plus marker indicates a new function instance whereas a red 'x' marker indicates a failure of a function invocation which we avoid in any level of processing production workloads. It seems that Amazon re-deploys a function instance whenever there is a change in a code or a configuration but it keeps an existing deployment in a certain period to handle incoming requests during the transition. If the function is invoked right after those events, it is likely that the previous version of the function will be processing the new invocation. We also do not find a same number of instances are prepared. For example, we saw that 10 instances were waiting for new invocations, but new deployment started with a single instance along with purging previous instances. If a new function instance requires initializing processes such as downloading necessary files to a temp directory, it creates

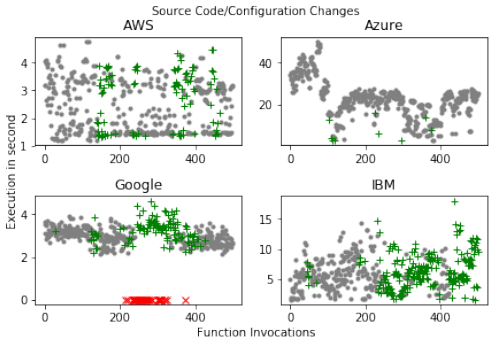


Figure 5.6: Function Behavior over CD/CI

(gray dot: existing instances, green +: new instances, red x: failed instances)

delays in processing new invocations. In the subplot of Microsoft Azure, we observe about a small number of new instances launched during the entire invocations where it either applies any changes through existing deployments or has only a few deployments to swap. It is not visible in the figure due to a small number of instances, and we may need additional tests to determine the behavior of functions towards those events. In the subplot of Google, we observed failures and delays of function executions for those events. We need to revisit Google when the beta release of the event-driven platform is ended. Unlike AWS Lambda, IBM re-deploys a new version of source code and starts to manage incoming messages on new instances that may cause excessive delays on a client-side program if new deployment takes some time to initialize.

We suggest that new deployment of a function need to be prepared with the equivalent size of function instances compared to the current loads which will prevent delaying response time in the context of concurrent invocations. event-driven framework with DevOps may enhance software development and continuous delivery through an agile function deployment and configuration as we will use multiple functions together with frequent changes.

5.2.7 Event-driven Model versus Virtual Machine

Event-driven computing does not offer either high-end computing performance or an inexpensive pricing model compared to virtual machines like Amazon EC2. Virtual machines on cloud computing have offered multiple options to scale compute resources with machine types, network bandwidth and storage performance to meet the expectation of performance requirements of a given workload which requires optimal capacity planning and system management. event-driven computing, however, aims to provide dynamic compute resources for lightweight functions without these administrations and offer cost-efficient solutions in which users pay for the execution time of functions rather than paying for the leased time of machines. Amazon, for example, has an EC2 machine choice optimized for intensive tasks with up to 128 vCPUs and a 3.8TiB size of memory with a limited number of allocations per account. AWS Lambda allows invoking a function at least a thousand of concurrency per region with a small memory allocation up to 2.8GiB (3008MB) which result in a total size of 2.8TiB. We ran an experiment in this section to demonstrate possible use cases of event-driven with the understanding of the differences between these two computing resources. event-driven computing is powered by container technologies which have near zero start-up delay and deleting latency during a function life-cycle. For example, we ran a test of a NodeJS function using Apache OpenWhisk with Kubernetes, and a small Docker container (as a Kubernetes Pod) is deployed and terminated within a few milliseconds for the function invocation. The container instance was running (*warm state*) for a specified period to receive a future event message which merely consumes resources and was changed to a pause state which indicates a terminated process but reserving function data like source code and a temp directory in storage. The paused instance saves time to re-instantiate a function for the future requests without wasting compute resources. Some delays might be observed at first which is called *cold start* but a configuration can be changed to extend the idle time of the running container, or a regular wake-up

invocation can be implemented as a workaround if necessary. On the contrary, virtual machines take at least a few seconds to be ready, and a high-end server type with multiple virtual CPUs and large size of memory and storage with a custom machine image may take 10 to 20 minutes to initialize. Another issue of using virtual machines is that a resource utilization needs to be handled by users to maximize values of leasing machines. If a VM is idle, the utilization rate is decreased, and if more VMs are necessary to support a tremendous amount of traffic immediately, existing VMs are overloaded which may cause performance degradation. Regarding the charge interval of leased VMs, many cloud providers have applied a per-second basis like event-driven computing with some exceptions. Therefore, the particular workload would be deployed on VMs if it requires high performance compute resources for a short amount of time.

We made a cost comparison between event-driven computing and traditional virtual machines because we think it would explain cost-effectiveness for specific workload deployed on these two platforms. The charging unit is different, event-driven computing is based on 100 milliseconds per invocation, and a virtual machine uses either an hour or a second basis charge per instance. When we break down the cost in a second, event-driven is almost ten times expensive compared to a virtual machine regarding the allocated compute resources. We ran two scripts written by a python and a JavaScript building binary trees. Table 5.4 shows the execution time of the creating binary trees and the total cost with the rank ordered by cost-effectiveness. The result indicates that a sequential function on event-driven computing would not be a good choice regarding cost-savings although it is still a simple way of deploying a function as a service. However, dynamic concurrent invocations on event-driven computing will save cost against overloaded virtual machines when many event messages are requested.

Table 5.4: Building Binary Tree with Cost-Awareness

Platform	RAM	Cost/Sec	Elapsed Second	Total Cost (Rank)
AWS Lambda	3008MB	$4.897e-5	20.3	$9.9409e-4 (6)
AWS EC2 (t2.micro)	1GiB	$3.2e-6	29.5	$9.439e-05 (3)
Azure Functions	192MB	$3e-6	71.5	$2.145e-4 (4)
Azure VM	1GiB	$3.05e-6	88.9	$2.71145e-4 (5)
Google Functions	2GB	$2.9e-5	34.5	$0.001 (7)
Google Compute (f1-micro)	600MB	$2.1e-6	19.2	$4.0319e-05 (1)
IBM OpenWhisk	128MB	$2.2125e-6	34.2	$7.5667e-05 (2)

5.3 Use Cases

There are several areas where event-driven can play an important role in research applications as well as in a commercial cloud. Big Data map-reduce application can be executed similarly but a cost-effective way of deployment as we discuss implementations of the big data applications in a series of event-driven functions with cloud object storage and databases [45, 59]. We ran some Big Data Benchmark tests by scanning 100 text files with different queries and aggregating 1000 text files with a group by and sum queries which show that certain applications are executable on event-driven framework relatively easily and fast compared to running on server-based systems. Image processing for CDN is applicable by the event-driven framework to process thumbnails of the images concurrently. Internet Of Things (IoT) is also one of the use cases for event-driven framework because IoT devices typically have a small computing power to process all the information and need to use remote resources by sending event messages which are a good fit for event-driven computing. IoT devices may invoke a function using a policy. For example, in case of a data-center, a cooling facility is essential for proper functioning of servers. When cooling is not working properly, a thermostat invokes a function to calculate live migration of allocated workloads to other data centers and determine shutdown of servers on particular areas. We hope to see several use cases of

event-driven computing as the main type of cloud-native application development soon.

5.4 Discussion

event-driven computing would not be an option for those need high-end computing power with intensive I/O performance and memory bandwidth because of its resource limitation, for example, AWS Lambda only provides 3GB of memory and 2 virtual cores generating 40 flops with 5 minutes execution timeouts. These limitations will be adjusted once event-driven environments are mature and there are more users but bulk synchronous parallel (BSP) and communication-free workloads can be applied to event-driven computing with its concurrent invocations. Additional containers for concurrent function invocations reduce a total execution time with a linear speed up, for example, a function invocation divided into two containers decreases an execution time in half. There are also overlaps and similarities between event-driven and the other existing services, for example, Azure Batch is a job scheduling service with an automated deployment for a computing environment. AWS Beanstalk [7] is deploying a web service with automated resource provisioning.

5.5 Related Work

We have noticed that there were several efforts to utilize existing event-driven computing for parallel data processing using concurrent invocations. PyWren [58] is introduced in achieving about 40 TFLOPs using 2800 Amazon Lambda invocations. The programming language runtime on event-driven computing has been discussed in the recent work [68]. Deploying scientific computing applications has been conducted with experiments to argue the possible use cases of event-driven computing for adopting existing workload [82] with its tool [81]. McGrath et al [**?**] shows event-driven comparison results for function latency among production event-driven computing environments including Microsoft Azure Functions but it was not a comprehensive review, such as testing

CPU, network and a file I/O, and several improvements have been made later such as an increment of memory allocation such as 3GB on Amazon Lambda and additional runtime support such as Java on Azure Functions and Golang on Amazon. OpenLambda [**?**] discusses running a web application on event-driven computing and OpenWhisk is introduced for mobile applications [8]. Since then several offerings on the event-driven framework with new features have been made. Kubeless [63] is a Kubernetes-powered open-source event-driven framework, like Fission [34]. Zappa [92] is a python based event-driven powered on Amazon Lambda with additional features like keeping the warm state of a function by poking at a regular interval. OpenFaaS is event-driven for Docker and Kubernetes with language support for Node.js, Golang, C#, and binaries like ImageMagicK. Oracle also started to support open source event-driven platform, Fn project [37]. In this work, we have investigated four event-driven computing environments in production regarding the CPU performance, network bandwidth, and a file I/O throughput and we believe it is the first evaluation across Amazon Lambda, Azure Functions, Google Functions and IBM OpenWhisk.

5.6 Conclusion

Functions on event-driven computing can process distributed data applications by quickly provisioning additional compute resources on multiple containers. In this chapter, we evaluated concurrent invocations on event-driven computing including Amazon Lambda, Microsoft Azure Functions, Google Cloud Functions and IBM Cloud Functions (Apache OpenWhisk). Our results show that the elasticity of Amazon Lambda exceeds others regarding CPU performance, network bandwidth, and a file I/O throughput when concurrent function invocations are made for dynamic workloads. Overall, event-driven computing scales relatively well to perform distributed data processing if a divided task is small enough to execute on a function instance with 1.5GB to 3GB memory limit and 5 to 10 minute execution time limit. It also indicates that event-driven computing would be

more cost-effective than processing on traditional virtual machines because of the almost zero delay on boot up new instances for additional function invocations and a charging model only for the execution time of functions instead of paying for an idle time of machines. We recommend researchers who have such applications but running on traditional virtual machines to consider migrating on functions because event-driven computing offers ease of deployment and configuration with elastic provisioning on concurrent function invocations. event-driven computing currently utilizes containers with small computing resources for ephemeral workloads, but we believe that more options on computing resources will be available in the future with fewer restrictions on configurations to deal with complex workloads.

Chapter 6

Efficient Software Defined Storage Systems on Bare Metal Cloud

6.1 Introduction

In the Big Data ecosystem, the bare metal server has become a high performance alternative to hypervisor-based virtual machines since it offers advantages of direct access to hardware and isolation from other tenants' workloads. However, benchmark results or comprehensive data of infrastructure options are not generally available. This chapter reports on the results of a big data benchmark of commercial cloud services that have provided bare metal equivalent server types. This work aims to address data processing performance using Hadoop-based workloads including Terasort, and the results would be useful in designing and building infrastructure along with the cost analysis and performance requirements depending on use cases. We perform big data benchmark on production bare metal environments to demonstrate compute performance with local NVMe and block storage are tested as an alternative storage option.

We started using bare metal servers with Hadoop workloads because the previous work indicated the needs of the exascale-like infrastructure for data-intensive applications [**?**], and we wanted to see how these new servers perform differently with additional computing power and large volumes of storage. Oracle Cloud Infrastructure has bare metal server types which offer 104 virtual cores with hyper-threading, 768GB memory and 51.2TB size of local storage per instance which can be better solutions to any existing big data problems in which massive intermediate data are generated rapidly for subsequent analysis with many CPU and memory intensive sub-tasks. Other cloud providers have a different configuration of those resources resulting in a broad range of choices in bare metal environments, for example, Amazon r5.24xlarge instance offers similar resources

compared to Oracle in compute, memory and network except the local NVMe storage or z1d.metal for high CPU clock speeds. Google and Microsoft do not explicitly have bare metal servers but equivalent options are available to compare such as n1-highmem-96 with local scratch volumes on GCE and L64s_v2 on Azure. Furthermore, persistent block storage can mitigate extra storage needs for those who have data-intensive workloads with large volumes. This is particularly helpful when terabyte-scale volume is not enough or data separation from compute is necessary. There is also a lack of evaluation data indicating actual performance optimization and designing efficient clusters with scalability.

Hardware performance data is subject to the actual execution time of applications, high IOPS and low latency storage devices contribute to the performance of I/O intensive jobs and FLOPS is a measure of provisioned computing resources, as well as high network bandwidth for fast communication. In practice, however, the complex workloads have multiple characteristics to detect tuning factors and inspect bottlenecks if exists and therefore performance evaluation with various scenarios is necessary for understanding the environment deployed.

Bare metal servers are widely available with various options to add extra CPU cores, memory, and local NVMe as well as high network bandwidth. Big data users with data intensive application may utilize these configurations when vertically scaled clusters generate better performance than constructing many numbers of commodity servers. In addition, improved performance results in increasing cost efficiency as more resources are quickly returned for further use.

The previous work claimed that Hadoop jobs often run better on scale-up servers than horizontally scaled servers [**?**] so that one can take advantage of an increased number of virtual cores, memory sizes and network bandwidth provided by typical bare metal servers. Fig 6.1 shows distributed I/O benchmark, TestDFSIO, as an example between two different environments, one with 8 worker nodes, 48 virtual cores each and the other with half number of the worker nodes, four, but

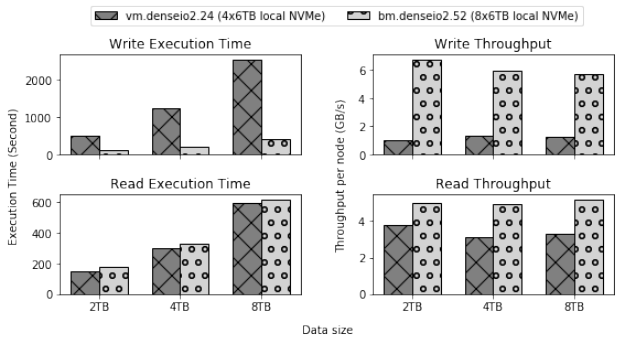


Figure 6.1: TestDFSIO between VM and Bare Metal

increased core count, 104 virtual cores each. For writing task of TestDFSIO between 2TB to 8TB data sizes, about 4 to 6 times reduced execution time and throughput was observed on vertically scaled bare metal servers, i.e. BM.DenseIO2.52. For reading task of TestDFSIO, the throughput is slightly improved as data size increased from 2TB to 8TB for bare metal servers. Note that these two environments have same hardware specifications on local NVMe storage and CPU and memory although the bare metal setup has a few more core counts and memory sizes due to the different CPU/MEM ratio per server type.

With the advent of high performance storage options and reduced hardware costs, public data centers and academic platforms are now equipped with either solid state devices (SSDs) with Serial ATA (SATA) or NVMe SSD with PCI Express bus for better I/O performance than magnetic storage, i.e. hard disk drives (HDDs) and bare metal servers are typically offered with SSDs to support increased performance needs. Production cloud services have two storage types, one is temporal local terabyte-scale NVMe dedicated to a server instance and the other one is persistent remote petabyte-scale block storage. We tend to explore these options for I/O intensive workloads so that one can understand the difference between these two storage options for running big data workloads.

We know that bare metal servers are available on most production cloud environments but it

is difficult to find the analysis of big data workloads across these environments. This chapter is the first evaluation of commercial bare metal environments focusing on various storage options for running big data applications. Amazon EC2, Google Cloud, Oracle Cloud, and Microsoft Azure are considered to run our experiments.

Fast and powerful hardware accelerates large data processing, and we have observed that real world workloads are a mixture of compute intensive, i/o intensive, and memory intensive, in which bare metal clouds can be a solution for these. We use built-in Hadoop benchmark tools which are useful to measure the performance of Hadoop systems by varying workload volumes and configurations. The workloads tested here include WordCount, PageRank, K-Means, TeraSort, and DFSIO and the performance data would be widely applicable to various applications running on similar compute environments.

The block storage is a network storage device which provides an individual mount point to access, and therefore multiple devices with various volume sizes can be attached and detached in a few steps. Block storage also ensures scalability as more attached volumes with increased sizes deliver additional capacity and performance. The service limit per server instance, however, prevents to scale vertically and requires to use additional instances for achieving increased performance. For example, Amazon Elastic Block Store (AWS EBS) generates 64,000 IOPS and 1,000 MB/s throughput per volume but extra I/O operations will be throttled if two or more volumes attached to the same instance. It is caused by the service limit, 80,000 IOPS and 1.7 GiB throughput per server and non-optimized server types for block storage may reduce these caps additionally.

We have tested Hadoop cluster using the block storage to demonstrate these problems. Table 6.1 is an IOPS comparison between block storage and local NVMe per server instance and the performance difference is ranging from 11 times (GCE) to 41 times (AWS). It is trivial that choosing local NVMe against block storage is good for high IOPS required workloads as long as the

limitations do not apply. First, petabyte scale data would not fit into local NVMe as it is currently offered between 3TB to 51.2TB per instance. Amazon's Block storage EBS offers up to 496TiB by attaching 31 volumes and Oracle allows up to 1PB by aggregating 32 attachments with 32TiB volume size each. Handling intermediate and temporal data would work better on local NVMe otherwise there is a cost moving data from/to other permanent storage. Processing data generated during analysis and simulation is a good use case in this context.

Block storage is convenient to use and applicable to Hadoop data nodes but there are several limitations. IOPS in the table cannot be achievable at a small volume size, and each provider has a different ratio such as IOPS per provisioned volume size. For example, Amazon provides 64k IOPS for the volume of 1280GiB or greater with 50:1 ratio. Google cloud provides 60k IOPS for the volume of 2048 GiB or greater with 30:1 ratio, Oracle cloud provides 25k IOPS for the volume of 417 GB or greater with 60:1 ratio. Note that the numbers in the table only indicate reading performance at a 4096 byte block size. It does not include IOPS for write with different block sizes which will be lower than those.

Local NVMe storage has tremendous performance but there are also limitations. Fixed number of volumes are offered with server types and the volume size is not changeable. For example, GCE provides a local NVMe disk in one size, 375GB, and Oracle has two options, 28.8TB and 51.2TB in bare metal server types. To utilize the maximum IOPS, data has to be evenly distributed on attached volumes, i.e. 8 volumes per instance. Server types are limited to the use of local NVMe storage. i3.metal server type on AWS, L64s_v2 on Azure, any server with 32+ vCPUs on GCE and BM.DenseIO2.52 on OCI is available for the IOPS on the table. We believe that new products with improved hardware will replace these performance data and eliminate the limitations anytime soon. For example, Azure Ultra SSD in preview provides 160k IOPS for the block storage which 8 times greater than the current performance, 20k per volume.

Table 6.1: IOPS

Provider	Block Storage IOPS	Local NVMe IOPS
AWS	80,000 (64,000 per volume)	3,300,000
Azure	80,000 (20,000 per volume)	2,700,000
GCE	60,000 (60,000 per volume)	680,000
OCI	400,000 (25,000 per volume)	5,500,000

In this chapter, we emphasize on the performance analysis by evaluating Hadoop benchmark workloads and comparing the results with system metrics e.g. IOPS and FLOPS across the environments. Our results indicate that there is performance benefit of leveraging bare metal servers due to the increased compute resources per node in the Hadoop cluster but system upper limit may prevent fully utilizing provisioned resources when applications become I/O intensive. We also provide cost analysis for those workloads to show the economic benefits of provisioned resources so that one can choose the best option of running their applications with the consideration of the economic value and performance requirements.

The contributions of this work are:

- Comparing the performance of Hadoop workloads on different bare metal platforms
- Understanding the difference between block storage and local NVMe for I/O intensive workloads
- Providing the analysis of cost efficiency potentially reducing storage costs

The rest of the chapter is prepared as follows. In Section 6.2, we describe our experimental configuration and explain the results in the next section 6.3. The section 6.4, we described related work briefly.

6.2 Hardware Specification and Application Layout

6.2.1 Experimental Setup

We built a Hadoop cluster with a various number of workers ranging from 3 to 8 and two master nodes and one gateway node. The deployment was completed by Cloudera 5.16.1.

Amazon EC2 has a memory optimized server type (r5.24xlarge) with Intel Xeon 8175M processors running at 2.5 GHz, with a total of 48 hyper-threaded cores, 96 logical processors and recently added bare metal server type i.e. r5d.metal is excluded in this experiment [**?**] which offers 3.6 TB size of local NVMe divided by four mount points. Microsoft Azure also has an E64s v3 server type which provides 64 virtual cores by Xeon E5-2673 processor and 432 GB size of memory. We noticed that the number of vCPUs offered by Azure is increased in powers of two and 96 and 104 vCPUs are not available in the ESv3 series. M128 server types are excluded because of the pricing ($13.338/hour for 2TiB of memory, and $26.688/hour for 4TiB). n1-highmem-96 server from Google Compute Engine offers 96 virtual cores on Intel Xeon Skylake and 624 GB of memory. Local scratch storage allows us to choose between NVMe and SCSI interface with a maximum volume size of 3TB. BM.Standard2.52 is a standard server type from Oracle with Intel Xeon Platinum 8167M resulting in a total of 104 logical processors. BM.DenseIO2.52 is a server type with eight of local NVMe in a total size of 51.2 TB from Oracle [**?**]. Table 6.2 provides the details of the server types with a hardware specification and we believe that these server choices are comparable although the numbers are not completely the same across different environments. We are aware that the performance gaps of provisioned resources among each other may reduce the consistency of our experiments and we address this limitation when we represent our results in the following sections.

Table 6.2: Cluster Configuration

Item	AWS	Azure	GCE	OCI
Server name	r5.24xlarge	E64s v3	n1-highmem-96	bm.standard2.52
CPU type	Intel Xeon Platinum 8175M	Intel Xeon E5-2673 v4 (Broadwell)	Intel Xeon Scalable Processor (Skylake)	Intel Xeon Platinum 8167M
Clock Speed	2.5 GHz	2.3 GHz	2.0 GHz	2.0 GHz
Turbo Boost	3.1 GHz	3.5 GHz	-	2.4 GHz
Core Count	96	64	96	104
Memory Size	768 GB	432 GB	624 GB	768 GB
Network Bandwidth	25 Gbps	30Gbps	-	2x25 Gbps
OS	CentOS 6	CentOS 6	CentOS 6	CentOS 7
Kernel Version	2.6.32	2.6.32	2.6.32	3.10.0
Storage type attached	General purpose SSD (gp2)	Premium SSD (P30)	Regional SSD	Block Storage SSD (iSCSI)
HDFS Volumes	2048GB x 7	1024GB x 7	834GB x 8	700GB x 6
Max IOPS per volume	6000	5000	25000	25000
Max IOPS per worker	42000	35000	60000 (r) 30000 (w)	150000
Max throughput per volume	250MiB/s	200MB/s	400MB/s	320MB/s
Max throughput per worker	1750MB/s	1400MB/s	1200MB/s (r) 400MB/s (w)	1920MB/s
Package Version	hadoop-2.6.0+cdh5.16			
Number of Workers	3			

6.2.2 I/O Test

We ran flexible I/O tester (fio) on this storage to measure performance data before running our experiments. Our Hadoop clusters mount data nodes by either local NVMe or block storage and storage performance make a big difference for running our HDFS-based jobs. Table 6.3 shows our test results in detail. Note that these are aggregated IOPS by fio's group reporting and randread means random read, randwrite means random write, and rw50 means random read and write in 50/50 distribution. Generally speaking, high IOPS at small block size is good for database systems which have usage patterns of frequent access for handling transaction data, and high IOPS at large block size is good for data intensive jobs including Hadoop which requires high throughput for sequential reading and writing. Also, most high-end SSD devices generate steady performance across different block sizes although we find interesting results from the test. Amazon shows the best storage performance at a small block size, 2 million IOPS at 4k, but Oracle has significant performance at a large block size, 21.6 GB throughput per second. This will affect data intensive workloads of our experiments such as DFSIO, Wordcount, and Terasort. The results also revealed that Google does not offer comparable storage performance per instance in terms of IOPS and volume size and therefore scaling out approach would be appropriate to build a system for data intensive workloads. It is worth to mention that high IOPS for writing is important as frequent writing and deleting are expected. The local NVMe is created as an empty space when a server instance is launched and any data to analyze or permanent data to keep has to be copied from/to other space e.g. block storage. Changing the status of an instance may purge contents in the storage handling like temporal scratch space. With these IOPS, OCI produced the maximum 18 GB/s throughput whereas AWS produced 6.1 GB/s throughput for random write.

Table 6.3: Flexible I/O Tester (fio) Results

(IOPS x 1000)

Block Size, I/O Pattern	AWS i3.metal 8 x 1.9T	Azure L64s_2 8 x 1.9T	GCE highmem96 8 x 375G	OCI BM.DenseIO2.52 8 x 6.4T
4K, randread	2048.9	886.3	275.9	1334
4K, randwrite	1457.5	760.5	269.7	1098
4K, rw50	1528.5	840.9	345.2	1180
16K, randread	891	750.9	161.6	1088
16K, randwrite	378.1	534.4	90.7	713
16K, rw50	427.3	529.9	115.8	850
256K, randread	60.1	58.2	11.1	75.4
256K, randwrite	24.6	38.5	6.2	68.7
256K, rw50	27.6	33.8	7.9	82.6

6.2.3 Scalability

We wanted to evaluate the scalability of our workloads by using the scaling context of HPC systems. Figure 6.2 shows Hadoop benchmark results to describe how our cluster handles terabyte scale data with additional worker nodes. The upper plot in the figure shows reduced execution time when the number of workers is increased. Each benchmark ran with the same data size over 3, 6, and 8 worker nodes which indicate good scaling with more resources. This is not always valid for certain workloads due to shuffling costs. Typical workloads go through data reduction from map to reduce phase which decreases the amount of data exchanged between nodes but in some applications e.g. PageRank may not or increase the amount of data transferred over additional worker nodes. I/O intensive workloads which can be partitioned by the number of mappers generally guarantee performance improvements over an increased number of nodes. Wordcount and DFSIO are identified in this context.

The bottom plot of the Figure 6.2 depicts flat lines for processing an increased amount of data

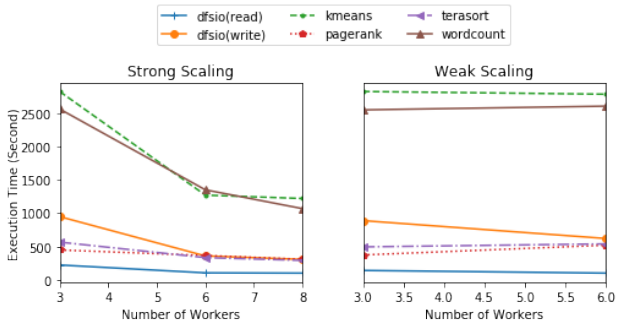


Figure 6.2: Scaling Results

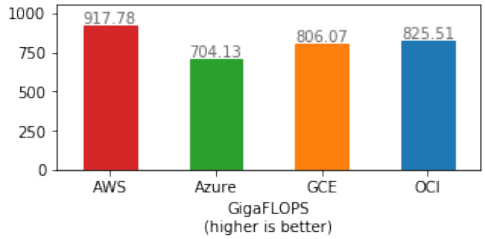


Figure 6.3: CPU Performance (GFLOPS)

by adding more workers which are good for weak scaling. K-Means, Wordcount, and DFSIO are relevant to this interpretation. PageRank and Terasort were slightly worsened as more worker nodes were added and this concludes the same results that we discussed in the strong scaling results, shuffling costs over multiple worker nodes. The tested data size are varied between 1.6TB and 3.2TB for Wordcount, 50 million and 100 million pages for PageRank, 1.2 billion and 2.4 billion samples for K-Means, 600GB and 1.2TB for Terasort and 2TB and 4TB for DFSIO. The worker node consists of 104 hyper-threaded CPU cores, 768 GB memory and a dual port 25GB Ethernet adapter.

6.3 Experimental Results

6.3.1 Compute Performance

Processing power from various server types and platforms are very different from each other which prevents us from building equivalent compute environments. Floating point operations per second (FLOPS) is, however, a reasonable methods to compare CPU performance on target server types. Figure 6.3 shows our results on the four server types in which measured gigaFLOPS in double precision calculations are 917.78 for AWS r5.24xlarge (96 vCores), 825.51 for OCI BM.Standard2.52 (104 vCores), 806.07 for GCE n1-highmem-96 (96 vCores), and 704.13 for Azure e64s_v3 (64 vCores) in a rank order. We believe that these are the closest match we can find for our experiments with negligible gaps of provisioned resources. We were searching for one of the largest server types offering a large number of vCPUs, high amount of memory and increased network bandwidth but around 100 vCores, less than 1TB of memory (10GB of memory per vCPU) and up to 50 gigabit networks by spending under $10 per an hour. The different types of hardware configuration may reduce the consistency of our experiment, therefore, we use the performance data as a reference, not a direct measure of the evaluation.

6.3.2 Storage Performance

We discussed the storage performance in the section 6.1 and evaluated by running TestDFSIO with data nodes on block volumes and local NVMe volumes respectively. With the understanding of the performance gaps, i.e. 652K IOPS vs 1334K IOPS, it is expected to see better results with local NVMe volumes but we still find block storage is useful. Figure 6.4 shows the comparison results between these two storage options by TestDFSIO write (upper subplots) and read (bottom subplots). First, the performance difference is significant as data size increased on both write and read tests. For the writing results, We find that 1.3 times reduced execution time on NVMe for

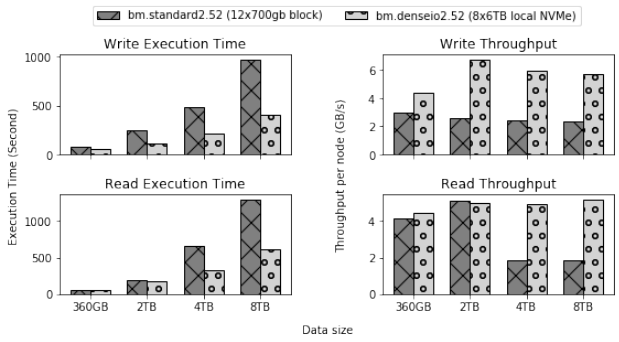


Figure 6.4: TestDFSIO between Block Storage and local NVMe

360GB data and 2.4 times reduced execution time for 8TB data size. The write throughput results (subplot on the top right) explains why the gaps were enlarged. The write throughput on block storage was decreased but one on local NVMe was improved over increased data sizes. For the reading results, it showed similar results, 2 times reduced execution time for 8TB data read. The first two bars, 360GB and 2TB show comparable results because the worker nodes uses memory rather than writing into disks.

Table 6.4 shows performance data between these two storage options and we find higher gaps in this direct performance data than TestDFSIO results. It implies that we may not observe the same performance difference when benchmark is completed with actual applications and other storage options e.g. block storage should not be ignored due to raw performance data as they might offer other benefits e.g. a persistent store and data movement.

6.3.3 Production Comparison

We chose to evaluate production bare-metal environments by HDFS based Hadoop workloads because these benchmark tools are popular and widely used to verify provisioned resources including compute, storage, and network. Figure 6.5 provides a single view for the six different workloads, tested with block storage attachments. As we discussed earlier, block storage generates different

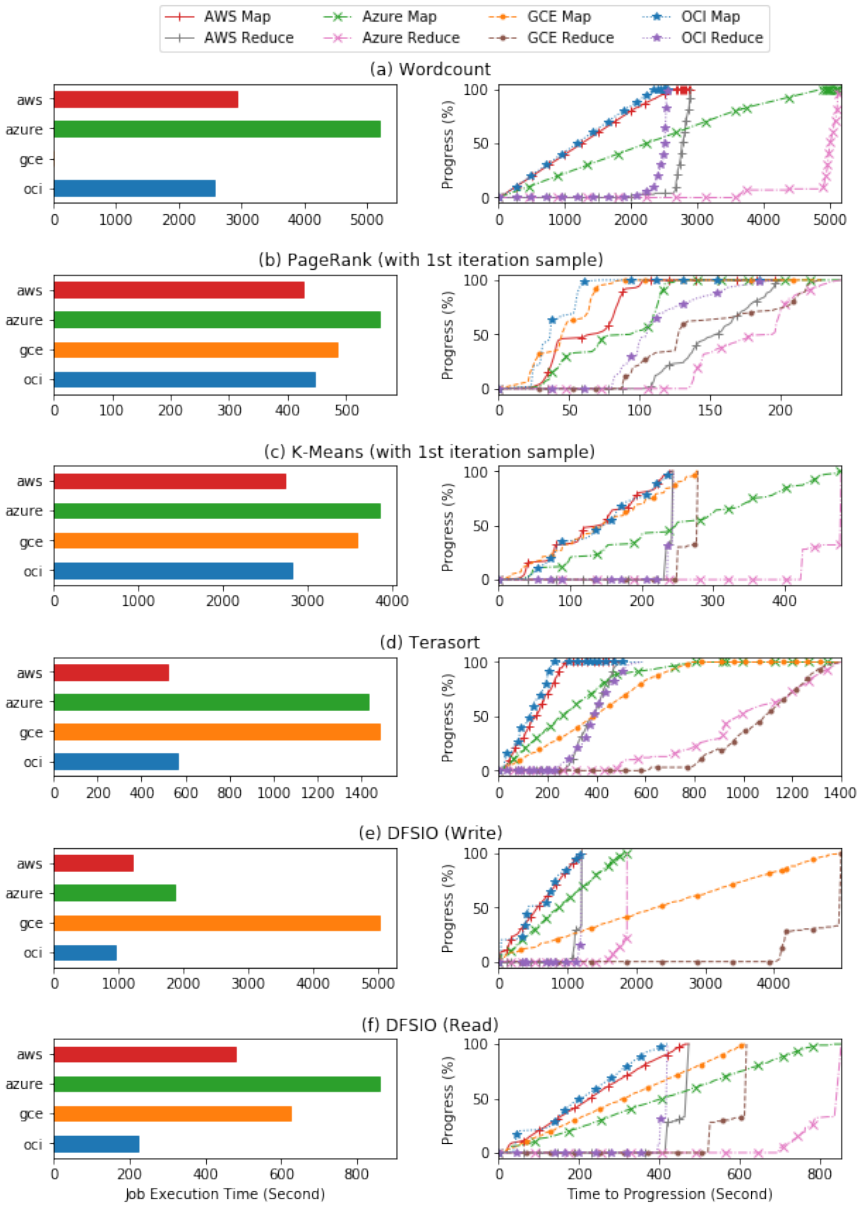


Figure 6.5: Hadoop Workloads on AWS, Azure, GCE and OCI

Table 6.4: Oracle I/O Performance (Per Instance)

Item	12 x 700 GB Block Storage	8 x 6.4 TB Local NVMe	Difference
4K randwrite IOPS	303,000	1,098,000	3.62x ↑
4K randread IOPS	292,000	1,334,000	4.56x ↑
256K randwrite Throughput	3.0 GB/s	18.0 GB/s	6x ↑
256K randread Throughput	3.0 GB/s	19.8 GB/s	6.6x ↑
4k randwrite Latency	7,908μsec	1,455μsec	5.4x ↓
4k randread Latency	8,205μsec	1,198μsec	6.8x ↓

IOPS by varying volume sizes and we configured the storage to meet certain performance level, i.e. 25,000 IOPS and 200MB throughput per volume. Wordcount ran with about 2TB size of text files which is to stress HDFS filesystem with simple computations. OCI completed Wordcount in about 42 minutes whereas Azure took twice, 87 minutes. GCE failed to complete this workload in two hours and we had to cancel it leaving an empty bar in the plot. PageRank updates score with some amount of data exchange between worker nodes and iterative tasks for counting and assigning values to unique records of pages. Our result shows that processing 50 million pages needs less than 10 minutes among all environments. K-Means clustering implementation ran with 10 iterations for processing 1 billion samples. AWS completed the task in 45 minutes while Azure took 64 minutes. However, it does not mean Amazon outperforms Azure as the CPU FLOPS is 1.3 times better on r5.24xlarge. Terasort stresses both CPU and storage and high IOPS are required. We find a similar result compared to PageRank. Both AWS and OCI completed the task in 10 minutes as GCE and Azure took about 24 minutes to finish. It explains that Azure needs more core count and GCE needs additional volumes to show similar results. We have additional experiments for Terasort, see Section 6.3.4. TestDFSIO has two tasks, writing and reading. We reduced the number of a replica to 1 which removes a data exchange task for better results but this will result in increased reading time. OCI completed writing in 15 minutes but GCE took 1 hour and 23 minutes. The cap of

IOPS for write on GCE is 30,000 per instance and we suspect that this is a major contribution to the long execution time.

6.3.4 Workloads

We dedicate this subsection to describe Hadoop workloads. K-Means and Terasort implementation were explained with our experiments.

K-Means

The KMeans clustering method is a well known iterative algorithm and is a common example to examine MapReduce functions. The distance computation between data points with centroids runs in parallel at a Map function step by reading the dataset from HDFS, and representing a new centroid to the subsequent iterations completes a cycle. The intermediate data is stored on HDFS, therefore I/O performance is critical as well as computing requirements for this workload.

Terasort

Terasort is characterized by high I/O bandwidth between each compute and data node of a Hadoop cluster along with CPU intensive work for sorting 10 bytes key of each 100 bytes message. Running Terasort is a common measure of system performance and Figure 6.6 shows our results for handling 10TB size over a different number of workers. As we learned that Hadoop may skip data exchange between nodes if there are enough memory spaces, we increased the data size enough to overfill the available amount of memory in the cluster. Scaling efficiency was decreased after 6 worker nodes but we see linear scaling performance in our results. In addition, we also find that measured system performance data is useful to understand the behavior with the workload, especially if it requires a mixture of CPU and I/O resources. Figure 6.7 is added to show the system behavior for processing 600GB of Terasort data. The plot (a) Write I/O shows that many IOPS were generated

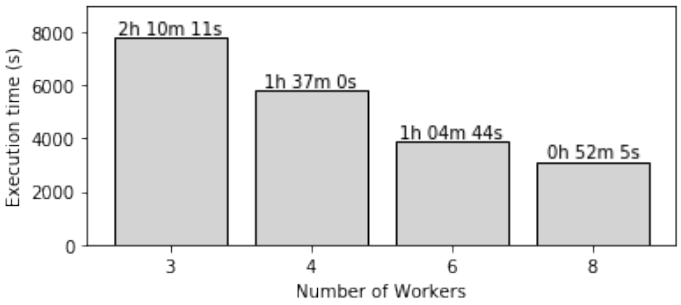


Figure 6.6: Terasort 10TB

during the reducer phase with the maximum of 22,528 IOPS. 'w/s' legend indicates a number write operations per second in the plot. The second plot shows network activity and most traffic was generated during the mapper phase. The last plot in Figure 8 describes CPU utilization and we find that the system was idling slightly during the transition phase between mapper and reducer. The system monitoring can be applied to other workloads identifying bottlenecks occurred by lack of provisioned resources i.e. high iowait with low IOPS storage and poor network speeds with a saturated network adapter so that system performance is ensured without under provisioning.

6.3.5 Cost Efficiency

The evaluation of the cost efficiency needs two sub metrics, one for evaluating the total CPU cost required for workloads and another one for evaluating the total storage cost.

Total Execution Cost (TEC) calculates the expense of the entire virtual cores provisioned by aggregating the individual wall time for completed workloads which is:

$$TEC = \sum_{n=1}^{T} cpu_n + storage_n \tag{6.1}$$

where cpu_n is the total cost of provisioned CPU cores per second, $storage_n$ is the total cost of provisioned storage volumes and T is the execution second of workloads.

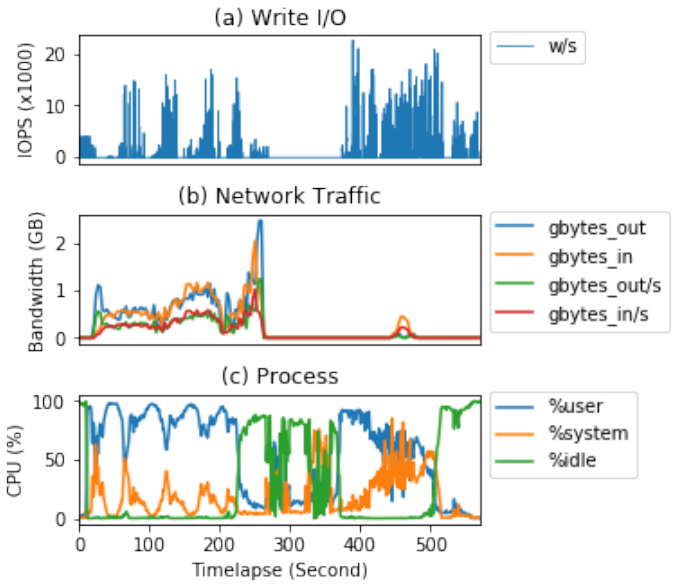


Figure 6.7: System Metrics for Terasort

CPU

The server pricing consists of CPU cost, memory cost and storage cost and core count is the main factor to yield a final value. Cost analysis for running big data workloads on these infrastructures need to verify actual performance on CPU and storage. We already measured FLOPS in the section 6.3.1 and we just need to convert them with pricing so that one can understand how much they actually pay. We are aware that FLOPS is not perfectly accurate as a performance metric, and we seek other methods to compare among different VM server types. We created Figure 6.8 by applying FLOPS to pricing. BM.Standard2.52 is 45% cheaper than AWS r5.24xlarge according to the pricing in the Table 6.5 while the measured FLOPS are similar, 684 and 687 GFLOPS for AWS and OCI respectively.

Table 6.5: Instance Pricing

Provider	Pricing	Type	VCPUs	Memory
AWS	$6.048 per Hour	r5.24xlarge	96	768GB
Azure	$3.629 per Hour	E64s v3	64	432GB
GCE	$5.6832 per Hour	n1-highmem-96	96	624GB
OCI	$3.3176 per Hour	bm.standard2.52	104	768GB

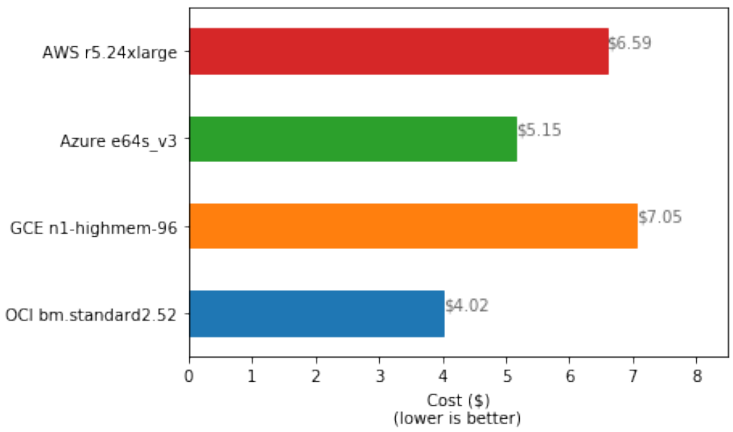


Figure 6.8: Cost for 1TFLOPS

Storage

Storage performance is a good indicator to find reduced execution time of workloads especially if they are I/O intensive. Multiple options are available to reduce costs of provisioning high performance storage and several limitations exist based on the type of storage and the type of instance attached. Table 6.6 indicates maximum IOPS and throughput for SSD based block storage and Google persistent SSD disk and Azure Blob is included as well to show similar storage choices among others. This will help plan a scalable system with performance and to avoid exceeding performance limits where throttling will occur to reject additional requests.

Maximum IOPS and throughput per instance is also an important metric because multiple volumes per instance can easily reach these limits. This also affects cost on provisioning block storage as they may require additional instances to have volumes with high IOPS or high throughput. In other words, a 64k IOPS EBS volume attached to an instance does not have additional space to ensure a maximum IOPS per volume as throttling occurs by instance IOPS limit. OCI produces great performance in this context, 400,000 aggregated IOPS per instance when multiple volumes attached with the maximum IOPS per volume, 25k. This will affect the cost effectiveness of block storage resulting in more instances, i.e. extra cost for the needs of high IOPS volumes.

With the high performance storage options, storage cost can be expensive with the extra charge on IOPS and throughput. The base pricing, however, is simple, OCI has the most inexpensive price tag compared to others. For example, OCI block storage is $0.0425 per GB in a month which is 57.5% cheaper than AWS general purpose SSD (gp2), $0.1. In addition, IOPS and throughput may require additional charges to pay. AWS io1, Azure Ultra SSD and Google SSD persistent disk have an extra cost for provisioned IOPS and throughput. We added instance cap data in the table but these are not applicable to all server types. Many CPU server types are usually applicable with these maximum numbers but the numbers in our table were referenced from the following instance

types: AWS r5.24xlarge, Azure Standard_M128m, OCI bm.standard2.52. IOPS can vary by block sizes and I/O pattern e.g. sequential read or random write, but the numbers in our table were prepared by 16384 bytes block size for reading in AWS, 8096 bytes block size for reading in GCE, and 4096 bytes block size for reading in OCI. Maximum throughput is for 128kilobytes or greater block size as IOPS affects this rate.

Block storage is offered by network storage solutions e.g. NAS or SAN and we find that throughput is more controlled than IOPS based on Table6.6. For example, we see the maximum IOPS per instance is ranging from 60K to 400K but throughput is between 1.2GB/s and 3GB/s. It is mainly by the dedicated network bandwidth. However, we expect to have increased throughput in the near future as big data applications have to process rapidly growing data needs. Oracle, again shows good throughput performance, 3GB/s compared to other providers, although Oracle block storage shares network bandwidth with other traffic and iSCSI block storage. Luckily, Oracle has a dual port 25GB Ethernet adapter for bare metal servers, therefore additional bandwidth can be achieved by adding a new network interface card (NIC). Better throughput will improve the cost effectiveness of block storage, especially for data intensive applications.

High IOPS to volume size ratio is recommended to effectively provide storage devices if applications are sensitive to IOPS. A low ratio may have to provision unnecessary volume sizes to achieve high IOPS, especially in distributed data placements e.g. HDFS data nodes. For example, 3:1 ratio from AWS general purpose SSD (gp2) requires 5334GB volume size to achieve 16000 IOPS whereas 60:1 ratio from OCI ensures the same IOPS from 267GB or greater volume sizes. Table 6.6 indicates that the minimum volume size to provision for maximum IOPS.

Table 6.6: SSD Based Block Storage

Provider	Cost (per GB-month)	Max Throughput per Volume (MiB/s)	Max Throughput per Instance (MiB/s)	Max IOPS per Volume	Max IOPS per Instance	IOPS Ratio to Volume Size (IOPS/GB)	Max Volume Size (TiB)
AWS General Purpose SSD(gp2)	$0.1	250	1750	16000	80000	3:1	16
AWS Provisioned IOPS SSD (io1)	$0.125 + $0.065/IOPS	1000	1750	64000	80000	50:1	16
Azure Premium SSD (p80)	$0.1	900	1600	20000	80000	-	32
Azure Ultra SSD (preview)	$0.05986 + $0.02482/IOPS + $0.5MB/s	2000	2000	160000	160000	-	64
GCE SSD persistent disk	$0.17 + $0.0057/IOPS	write: 400, read: 1200	write: 400, read: 1200	write: 30000 read: 60000	write: 30000 read: 60000	30:1	60
OCI	$0.0425	320	3000	25000	400000	60:1	32

6.4 Related Work

We use this section to describe the previous work related to the evaluation of bare metal servers, performance analysis of storage and address big data benchmark tools.

While there was a significant overhead created by virtualization with a hypervisor, research has been conducted [**?**, **?**, 86] to evaluate cloud environments for seeking performance improvement. In a recent study, Rad et al [**?**] showed promising results on scaling HPC and scientific applications by OpenStack Ironic software and Omote et al [**?**] introduced non-virtualized development for bare-metal servers with a quick startup. These activities are not directly related to our work but their experiments indicated the performance benefits of bare metal servers.

There are several Hadoop benchmark suites available including HiBench and BigDataBench [**?**,**?**] supported by Intel and Institute of Computing Technology, Chinese Academy of Sciences. These tools contain various big data workloads to evaluate the workload performance with low-level system information.

Performance analysis with NVMe disks has been growing with hardware improvements. Several studies [**?**, **?**, **?**, **?**] have focused on the evaluation of storage systems with I/O intensive applications. Their experiments were made to examine the scalability of storage with software developments.

6.5 Conclusion

With the advance of bare metal servers for big data workloads, a significant amount of research have been accomplished with the latest techniques and hardware accelerations. The rapid increasing challenges in big data, however, extend the discussion to the exclusive and consistent compute resource, bare metal clouds which can be embraced by the big data community.

Our experiment results indicate that Hadoop systems provisioned by bare metal servers with powerful storage options can be better options to build a high performance virtual clusters for

processing various workloads with a cost consideration. The result of our experiments delivers a thorough analysis of production environments with extensive research on storage options, i.e. block storage and local NVMe. Our results for Hadoop workloads on Amazon, Google Oracle, and Microsoft expose underlying hardware requirements e.g. IOPS, along with service limitations e.g. throughput allowance per instance.

High storage performance made a significant impact on HDFS based jobs with a large number of virtual cores. JBOD-style ('just a bunch of disks') non-RAID storage attachment shows 6 times better results with additional volume counts and CPUs per server than large numbers of low-end servers according to our result in Fig 6.1. Data intensive workloads, for that reason, may gain better scalability and efficiency on high capacity servers and bare metal servers are suitable for seeking performance improvements and cost savings.

In the future, we plan to extend our work to HPC server types evaluating communication intensive applications, and practical experience will be gained to improve actual performance with high-end network adapters.

Chapter 7

Conclusion

This dissertation described a software-defined system in terms of managing software environment as a sub-system, which is automated and reproducible for big data software stacks. We deployed and provisioned virtual clusters for NIST big data use cases using DevOps scripts and infrastructure provisioning templates. We observed the Hadoop-based software stack deployable on multi-clouds i.e. Amazon EC2, and OpenStack which improves software environment management along with the resource provisioning while recipes complete package installation and system configuration on-the-fly. We also used this approach to study the efficiency of our system using containers, showing that the reproducibility of software environment is improved through container images created by DevOps scripts and that our recipes to virtual clusters can provision scalable compute nodes.

Additionally, we evaluated the software defined system with event-driven computing and bare metal clouds. With the software defined system, one can invoke a user defined function concurrently on a pre-defined compute environment, including Amazon Lambda, Microsoft Azure Functions, IBM OpenWhisk and Google functions. Software defined system makes intermediate adjustments while determining proper resources constantly during the execution which aims to present manageable, dynamic and flexible computing resources. The serverless paradigm is to ensure data processing given environments but meet the performance requirement of computation through instant infrastructure provisioning. The evaluation is extended with bare metal clouds utilizing local NVME storage with high IOPS for HDFS transactions and dedicated and low latency network interface for data exchange in a cluster setup. Our results show that the bare metal cloud provides scale-up capability of data analytic which result in efficient resource utilization and good performance in practice.

In conclusion, we believe that a software defined system with regarding to managing software environments removes a barrier of utilizing compute resources with various software stacks, increasing the shareability and elasticity of user applications. The integration of DevOps scripting and infrastructure provisioning is well suited for the software defined system using virtual clusters, offering good performance and simple programmable sub-system on clouds.

Bibliography

[1] Amazon CloudFormation. `https://aws.amazon.com/cloudformation/`, 2010. [Online; accessed 17-February-2017].

[2] OpenStack Heat. `https://wiki.openstack.org/wiki/Heat`, 2012. [Online; accessed 17-February-2017].

[3] Coreos/rkt: a container engine for linux designed to be composable, secure, and built on standard. `https://github.com/coreos/rkt`, 2016. [Online; accessed 09-November-2016].

[4] Ubuntu lxd: a pure-container hypervisor. `https://github.com/lxc/lxd`, 2016. [Online; accessed 09-November-2016].

[5] B. Abdul-Wahid, H. Lee, G. von Laszewski, and G. Fox. Scripting deployment of nist use cases, 2017.

[6] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz. Hadoop gis: a high performance spatial data warehousing system over mapreduce. *Proceedings of the VLDB Endowment*, 6(11):1009–1020, 2013.

[7] A. Amazon. Elastic beanstalk, 2013.

[8] I. Baldini, P. Castro, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, and P. Suter. Cloud-native, event-based programming for mobile applications. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, pages 287–288. ACM, 2016.

[9] A. Belle, R. Thiagarajan, S. Soroushmehr, F. Navidi, D. A. Beard, and K. Najarian. Big data analytics in healthcare. *BioMed research international*, 2015, 2015.

[10] L. Benedicic, M. Gila, S. Alam, and T. C. Schulthess. Opportunities for container environments on cray xc30 with gpu devices. 2016.

[11] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner. Opentosca–a runtime for tosca-based cloud applications. In *International Conference on Service-Oriented Computing*, pages 692–695. Springer, 2013.

[12] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann. Tosca: portable automated deployment and management of cloud applications. In *Advanced Web Services*, pages 527–549. Springer, 2014.

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

[14] S. Bird. Nltk: the natural language toolkit. In *Proceedings of the COLING/ACL on Interactive presentation sessions*, pages 69–72. Association for Computational Linguistics, 2006.

[15] C. Bizer, P. Boncz, M. L. Brodie, and O. Erling. The meaningful use of big data: four perspectives–four challenges. *ACM SIGMOD Record*, 40(4):56–60, 2012.

[16] J. Bollen, H. Mao, and X. Zeng. Twitter mood predicts the stock market. *Journal of Computational Science*, 2(1):1–8, 2011.

[17] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, and J. Wettinger. Combining declarative and imperative cloud application provisioning based on tosca. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 87–96. IEEE, 2014.

[18] U. Breitenbücher, T. Binz, O. Kopp, and F. Leymann. Vinothek-a self-service portal for tosca. Citeseer, 2014.

[19] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, and D. Schumm. Vino4tosca: A visual notation for application topologies based on tosca. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 416–424. Springer, 2012.

[20] G. Breiter, M. Behrendt, M. Gupta, S. D. Moser, R. Schulze, I. Sippli, and T. Spatzier. Software defined environments based on tosca in ibm cloud implementations. *IBM Journal of Research and Development*, 58(2/3):9–1, 2014.

[21] N. Chaimov, A. Malony, S. Canon, C. Iancu, K. Z. Ibrahim, and J. Srinivasan. Scaling spark on hpc systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 97–110. ACM, 2016.

[22] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz. Realtime data processing at facebook. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1087–1098. ACM, 2016.

[23] S. Chen. Cheetah: a high performance, custom data warehouse on top of mapreduce. *Proceedings of the VLDB Endowment*, 3(1-2):1459–1468, 2010.

[24] M. Chevalier, M. El Malki, A. Kopliku, O. Teste, and R. Tournier. Implementing multidimensional data warehouses into nosql. In *17th International Conference on Enterprise Information Systems (ICEIS15), Spain*, 2015.

[25] E. M. Cody, A. J. Reagan, P. S. Dodds, and C. M. Danforth. Public opinion polling with twitter. *arXiv preprint arXiv:1608.02024*, 2016.

[26] J. P. Cohen and H. Z. Lo. Academic torrents: A community-maintained distributed repository. In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*, page 2. ACM, 2014.

[27] A. Cuzzocrea, L. Bellatreche, and I.-Y. Song. Data warehousing and olap over big data: current challenges and future research directions. In *Proceedings of the sixteenth international workshop on Data warehousing and OLAP*, pages 67–70. ACM, 2013.

[28] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pages 886–893. IEEE, 2005.

[29] N. Dalal and B. Triggs. Inria person dataset, 2005.

[30] A. Devresse, F. Delalondre, and F. Schürmann. Nix based fully automated workflows and ecosystem to guarantee scientific result reproducibility across software environments and systems. In *Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*, pages 25–31. ACM, 2015.

[31] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano. Devops. *IEEE Software*, 33(3):94–100, 2016.

[32] A. Eldawy, M. Mokbel, and C. Jonathan. Hadoopviz: A mapreduce framework for extensible visualization of big spatial data. In *IEEE Intl. Conf. on Data Engineering (ICDE)*, 2016.

[33] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015.

[34] Fission-Platform9. Serverless with kubernetes, 2017.

[35] P. Flanagan. Nist biometric image software (nbis). 2010.

[36] P. Flanagan. Fingerprint minutiae viewer (fpmv). 2014.

[37] Fn-Project. a container native apache 2.0 licensed serverless platform, 2017.

[38] S. Fouladi, R. S. Wahby, B. Shacklett, K. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *NSDI*, pages 363–376, 2017.

[39] G. Fox. Distributed data and software defined systems. 2013.

[40] G. Fox and W. Chang. Big data use cases and requirements. In *1st Big Data Interoperability Framework Workshop: Building Robust Big Data Ecosystem ISO/IEC JTC 1 Study Group on Big Data*, pages 18–21. Citeseer, 2014.

[41] G. Fox, J. Qiu, and S. Jha. High performance high functionality big data software stack. 2014.

[42] G. Fox, J. Qiu, S. Jha, S. Ekanayake, and S. Kamburugamuve. Big data, simulations and hpc convergence. In *Workshop on Big Data Benchmarks*, pages 3–17. Springer, 2015.

[43] G. Fox, J. Qiu, S. Jha, S. Ekanayake, and S. Kamburugamuve. White paper: Big data, simulations and hpc convergence. In *BDEC Frankfurt workshop. June*, volume 16, 2016.

[44] G. C. Fox, V. Ishakian, V. Muthusamy, and A. Slominski. Status of serverless computing and function-as-a-service (faas) in industry and research. *arXiv preprint arXiv:1708.08028*, 2017.

[45] G. C. Fox and S. Jha. A tale of two convergences: Applications and computing platforms.

[46] G. C. Fox, J. Qiu, S. Kamburugamuve, S. Jha, and A. Luckow. Hpc-abds high performance computing enhanced apache big data stack. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 1057–1066. IEEE, 2015.

[47] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral. The spack package manager: Bringing order to hpc software chaos. In *Proceedings*

of the International Conference for High Performance Computing, Networking, Storage and Analysis, page 40. ACM, 2015.

[48] D. Gannon, R. Barga, and N. Sundaresan. Cloud-native applications. *IEEE Cloud Computing*, 4(5):16–21, 2017.

[49] M. Geimer, K. Hoste, and R. McLay. Modern scientific software management using easybuild and lmod. In *Proceedings of the First International Workshop on HPC User Support Tools*, pages 41–51. IEEE Press, 2014.

[50] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers. Examining the challenges of scientific workflows. *Computer*, 40(12), 2007.

[51] K. Goodhope, J. Koshy, J. Kreps, N. Narkhede, R. Park, J. Rao, and V. Y. Ye. Building linkedin's real-time activity data pipeline. *IEEE Data Eng. Bull.*, 35(2):33–45, 2012.

[52] A. Goodman, A. Pepe, A. W. Blocker, C. L. Borgman, K. Cranmer, M. Crosas, R. Di Stefano, Y. Gil, P. Groth, M. Hedstrom, et al. Ten simple rules for the care and feeding of scientific data. *PLoS computational biology*, 10(4):e1003542, 2014.

[53] K. J. Gorgolewski, F. Alfaro-Almagro, T. Auer, P. Bellec, M. Capota, M. M. Chakravarty, N. W. Churchill, R. C. Craddock, G. A. Devenyi, A. Eklund, et al. Bids apps: Improving ease of use, accessibility and reproducibility of neuroimaging data analysis methods. *bioRxiv*, page 079145, 2016.

[54] C. S. Greene, J. Tan, M. Ung, J. H. Moore, and C. Cheng. Big data bioinformatics. *Journal of cellular physiology*, 229(12):1896–1900, 2014.

[55] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan. The rise of big data on cloud computing: Review and open research issues. *Information Systems*, 47:98–115, 2015.

[56] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1199–1208. IEEE, 2011.

[57] D. M. Jacobsen and R. S. Canon. Contain this, unleashing docker for hpc. *Proceedings of the Cray User Group*, 2015.

[58] E. Jonas, S. Venkataraman, I. Stoica, and B. Recht. Occupy the cloud: Distributed computing for the 99%. *arXiv preprint arXiv:1702.04024*, 2017.

[59] S. Kamburugamuve and G. Fox. Designing twister2: Efficient programming environment toolkit for big data.

[60] G.-H. Kim, S. Trimi, and J.-H. Chung. Big-data applications in the government sector. *Communications of the ACM*, 57(3):78–85, 2014.

[61] Y. Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[62] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann. Winery–a modeling tool for tosca-based cloud applications. In *International Conference on Service-Oriented Computing*, pages 700–704. Springer, 2013.

[63] Kubeless. A kubernetes native serverless framework, 2017.

[64] G. M. Kurtzer. Singularity 2.1.2 - Linux application and environment containers for science, Aug. 2016.

[65] A. Labrinidis and H. V. Jagadish. Challenges and opportunities with big data. *Proceedings of the VLDB Endowment*, 5(12):2032–2033, 2012.

[66] H. Lee. Building software defined systems on hpc and clouds. 2017.

[67] A. Lundqvist and D. Rodic. Gnu/linux distribution timeline, 2013.

[68] M. Maas, K. Asanović, and J. Kubiatowicz. Return of the runtimes: Rethinking the language runtime system for the cloud 3.0 era. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 138–143. ACM, 2017.

[69] D. Maio, D. Maltoni, R. Cappelli, J. L. Wayman, and A. K. Jain. Fvc2004: third fingerprint verification competition. In *Biometric Authentication*, pages 1–7. Springer, 2004.

[70] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky. The stanford corenlp natural language processing toolkit. In *ACL (System Demonstrations)*, pages 55–60, 2014.

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

[72] T. OASIS. Orchestration specification for cloud applications version 1.0. *Organization for the Advancement of Structured Information Standards*, 2013.

[73] A. Pak and P. Paroubek. Twitter as a corpus for sentiment analysis and opinion mining. In *LREc*, volume 10, pages 1320–1326, 2010.

[74] R. Priedhorsky and T. Randles. Charliecloud: Unprivileged containers for user-defined software stacks in hpc. Technical report, Los Alamos National Laboratory (LANL), 2016.

[75] R. Qasha, J. Cala, and P. Watson. Towards automated workflow deployment in the cloud using tosca. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 1037–1040. IEEE, 2015.

[76] J. Qiu, S. Jha, A. Luckow, and G. C. Fox. Towards hpc-abds: an initial high-performance big data stack. *Building Robust Big Data Ecosystem ISO/IEC JTC 1 Study Group on Big Data*, pages 18–21, 2014.

[77] W. Raghupathi and V. Raghupathi. Big data analytics in healthcare: promise and potential. *Health Information Science and Systems*, 2(1):1, 2014.

[78] E. E. Schadt, M. D. Linderman, J. Sorenson, L. Lee, and G. P. Nolan. Computational solutions to large-scale data management and analysis. *Nature reviews genetics*, 11(9):647, 2010.

[79] Y. Simmhan, S. Aman, A. Kumbhare, R. Liu, S. Stevens, Q. Zhou, and V. Prasanna. Cloud-based software platform for big data analytics in smart grids. *Computing in Science & Engineering*, 15(4):38–47, 2013.

[80] J. Soldani, T. Binz, U. Breitenbücher, F. Leymann, and A. Brogi. Toscamart: a method for adapting and reusing cloud applications. *Journal of Systems and Software*, 113:395–406, 2016.

[81] J. Spillner. Snafu: Function-as-a-service (faas) runtime design and implementation. *arXiv preprint arXiv:1703.07562*, 2017.

[82] J. Spillner, C. Mateos, and D. A. Monge. Faaster, better, cheaper: The prospect of serverless scientific computing and hpc. In *Latin American High Performance Computing Conference*, pages 154–168. Springer, 2017.

[83] V. Stantchev, R. Colomo-Palacios, and M. Niedermayer. Cloud computing based systems for healthcare. *The Scientific World Journal*, 2014, 2014.

[84] N. I. o. S. Technology., , I. T. Laboratory., N. B. D. P. W. G. (NBD-PWG), N. I. o. S. (U.S.), and Technology. Nist big data interoperability framework : volume 3, use cases and general requirements, 2015.

[85] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[86] G. von Laszewski, H. Lee, J. Diaz, F. Wang, K. Tanaka, S. Karavinkoppa, G. C. Fox, and T. Furlani. Design of an accounting and metric-basedcloud-shifting and cloud-seeding framework for federatedclouds and bare-metal environments. In *Proceedings of the 2012 workshop on Cloud services, federation, and the 8th open cirrus summit*, pages 25–32. ACM, 2012.

[87] C. Watson and C. Wilson. Nist special database 4. *Fingerprint Database, National Institute of Standards and Technology*, 17, 1992.

[88] J. H. Wegstein. *An automated fingerprint identification system.* US Department of Commerce, National Bureau of Standards, 1982.

[89] J. Wettinger, U. Breitenbücher, and F. Leymann. Standards-based devops automation and integration using tosca. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 59–68. IEEE Computer Society, 2014.

[90] J. Wettinger, U. Breitenbücher, and F. Leymann. Dyn tail-dynamically tailored deployment engines for cloud applications. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 421–428. IEEE, 2015.

[91] Y. Yamato, M. Muroi, K. Tanaka, and M. Uchimura. Development of template management technology for easy deployment of virtual resources on openstack. *Journal of Cloud Computing*, 3(1):1, 2014.

[92] Zappa. Serverless python web services, 2017.

[93] P. Zikopoulos, C. Eaton, et al. *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.