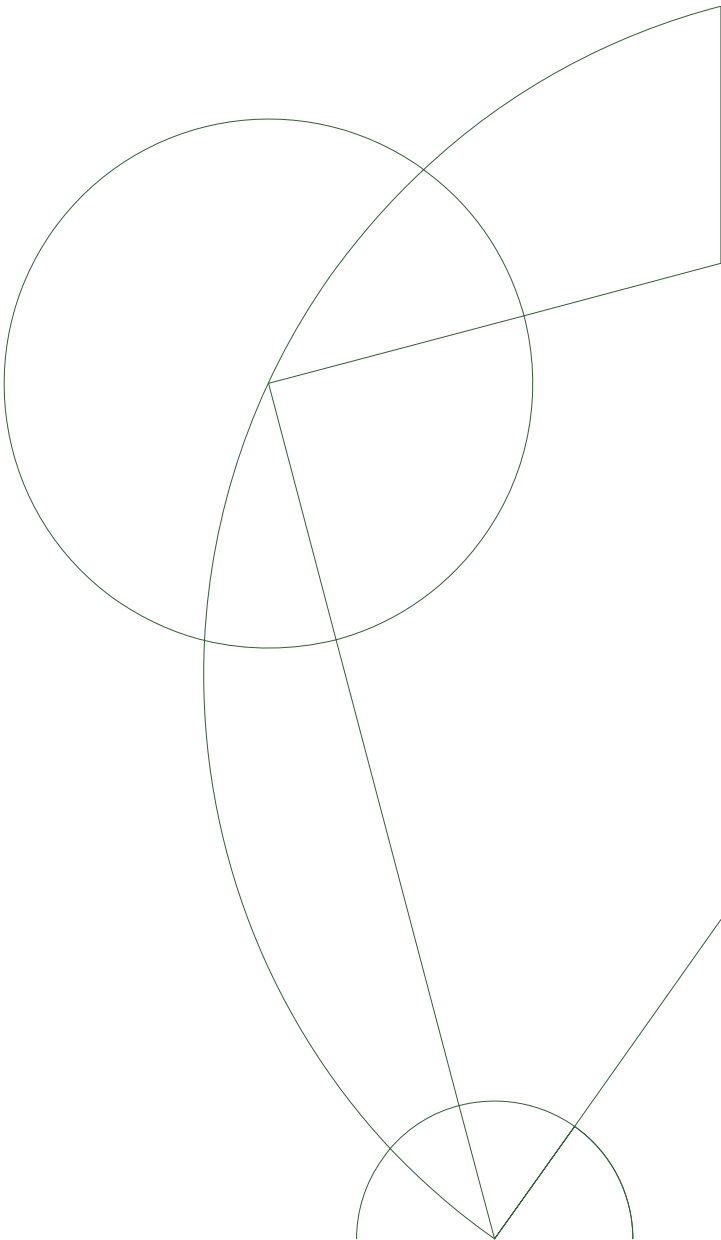
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

PoC Report

Marcus Friis Klausen - QGT728

Nix Derivations and NixOS Configurations for Deployment of Apache Spark Clusters



Academic advisor: Michael Kirkedal Thomsen

Submitted: August 29, 2025

Nix Derivations and NixOS Configurations for Deployment of Apache Spark Clusters

Marcus Friis Klausen - QGT728
DIKU, Department of Computer Science,
University of Copenhagen, Denmark

August 29, 2025

PoC Report

Author:	Marcus Friis Klausen - QGT728
Affiliation:	DIKU, Department of Computer Science, University of Copenhagen, Denmark
Title:	Nix Derivations and NixOS Configurations for Deployment of Apache Spark Clusters /
Academic advisor:	Michael Kirkedal Thomsen
Submitted:	August 29, 2025
GitHub:	https://github.com/MarcusFriisKlausen/nix-spark

Contents

Introduction

1

In recent years, the field of computing has experienced a significant surge in interest in machine learning and data science across both industry and academia. These disciplines require the processing of large volumes of data, which, at scale, can become highly compute-intensive and time-consuming. As a result, optimization in this area is of great importance. Two critical methods for addressing these challenges are **distributed data analytics** and **resource management**.

Among the most prevalent engines for large-scale data analytics lies **Apache Spark**. With Apache Spark, users can distribute data processing across a cluster of nodes, drastically reducing time used on processing. However, deploying and managing such clusters in a reliable, reproducible, and maintainable way remains a complex task. This is especially true in environments where configuration drift, dependency conflicts, or platform-specific inconsistencies can hinder effective development and deployment.

To address the challenge of deploying and managing such clusters in a safe and effective manner, functional package management systems such as **Nix** offer a promising solution. Nix allows for declarative specification of system dependencies, reproducible builds, and isolated environments, making it particularly well-suited for complex software stacks such as those required for Apache Spark clusters.

This project investigates the process of creating Nix and NixOS derivations for deploying and managing Apache Spark clusters. The goal is to explore how Nix can be used to automate and simplify the setup of scalable Spark environments, ensure reproducibility, and support efficient resource management. Specifically, the project will do this, by creating a deployable Apache Spark cluster, scoped to virtual machines running on a single host. Through this, the aim is to contribute to the broader effort of bringing the benefits of functional package management to the domain of big data processing.

Background

2

2.1 Apache Spark

Apache Spark, described in [11], is a widespread data analytics engine, which enables distributed computing, making it a powerful tool for machine learning and data science. Spark distributes computing jobs over a cluster, parallelizing the workload. It does so by the user either manually configuring a cluster, called *Standalone Deploy Mode*, or running over an external cluster manager such as *Kubernetes* or *Hadoop YARN*.

The engine also provides high-level APIs for multiple languages including Python and Java, as well as tools like Spark SQL for working with SQL and MLlib for machine learning.

2.2 Nix

The Nix ecosystem comprises a collection of tools, libraries, and practices for constructing fully declarative and reproducible systems, ranging from isolated development environments to complete operating systems. At its core is the **Nix package manager**, a purely functional system that employs the **Nix Expression Language** to define *derivations* — formal build instructions for software packages. Each derivation produces outputs stored in an immutable store, uniquely identified by a cryptographic hash derived from its inputs. This approach ensures that packages are specified declaratively, with new artifacts generated as deterministic transformations of existing files and dependencies [9] [10].

2.2.1 Nix Expressions

In Nix, derivations are descriptions of build tasks created from an attribute set. Packages are built using such derivations, which consist of dependencies, which are first installed recursively (as they themselves can be derivations), where after a build script is ran using specified environment variables [2].

Packages are declared as functions using Nix expressions. Let's take a look at an example from the official Nix documentation:

```
{ lib, stdenv, fetchurl }:

stdenv.mkDerivation rec {

  pname = "hello";

  version = "2.12";

  src = fetchurl {
    url = "mirror://gnu/${pname}/${pname}-${version}.tar.gz";
    sha256 = "1
    ayhp9v4m4rdhjmnl2bq3cibrbqqkgjbl3s7yk2nhlh8vj3ay16g";
  };

  meta = with lib; {
    license = licenses.gpl3Plus;
  };

}
```

Listing 2.1: Example of simplified Nix packaging of the GNU Hello package from nix.dev [7].

The first line declares that this is a function taking an attribute set as input with the attributes **lib**, **stdenv**, and **fetchurl**. The rest of the code is purely function definition. **mkDerivation** is a function from the attribute set **stdenv**, which takes a recursive attribute set as input and outputs a derivation. It is the evaluation of this function call, that is the final return value of the function that this file defines. The attributes given as inputs to mkDerivation are as follows:

pname: The name of the package.

version: The package's version number.

src: Source code, in this case for the GNU Hello package. It is fetched using the **fetchurl** function declared in the function inputs.

meta: Metadata for the package, which is expressed as an attribute set. In this case a license is declared as **gp13Plus** from the attribute set **licenses**, which itself is from the attribute set **lib**, as denoted by the `with lib;`.

Among other mkDerivation input attributes are **buildInputs**, which is a list of dependency derivations to be kept in scope during build-time and **buildPhase** and **installPhase** which are shell scripts that are run in the build phase and install phase [6].

Build scripts install packages as immutable directories in the nix store, located at `/nix/store`, identified by unique hashes, such as:

`/nix/store/veryUniqueHash-hello-2.12`,

where the hash is computed from the inputs of the derivation [2].

The declarative nature of Nix, is what makes the next topic of discussion **NixOS** shine, due to the fact, that if a very crucial part of the system is updated, say a core library of C, that many other packages depend on, there is no risk that the system breaks due to package incompatibility, as versions of packages are locked unless the user declares a new version.

2.2.2 NixOS

Building on the principles of Nix, **NixOS** is a Linux-based operating system designed to be fully declarative and reproducible. In NixOS, not only are packages managed through the Nix package manager, but the entire **system configuration**, including system services, networking, and user accounts, can be specified in a single declarative **configuration file**, which in turn is used to populate the Nix store. This approach practically treats the operating system itself as just another Nix derivation.

The declarative model of NixOS has several implications. First, it enables **atomic upgrades** and **rollbacks**. Since all configurations are built as immutable derivations, users can safely switch between system states, with the guarantee that rollbacks are always possible [2]. Second, system administration becomes more transparent and reproducible; for example, a system can be rebuilt on another machine simply by copying its configuration file and executing a single rebuild command. This makes NixOS particularly well-suited for environments where infrastructure as code and reproducibility are critical, such as cluster deployment.

As seen on Figure 2.1, Nix has a directory for **profiles** with path `/nix/var/nix/profiles`. These are what's called **generations** for the user(s). They are **symlinks** to user environments, such as point 11 on the figure, which are generated each time the system configuration is updated, for example when a new package is installed [8]. This is what allows for the aforementioned system rollbacks.

2.2.3 Nix Flakes

Flakes introduce a more structured and modular framework for managing Nix-based configurations. Unlike traditional Nix expressions, which often rely on implicit dependencies, flakes formalize inputs and outputs in a build specification, named `flake.nix`. Here is a simple example:

Figure 2.1: Tree of the Nix store and Nix profiles directory. Dotted arrows refer to symlink targets, while *italic* names refer to symlinks [2].

```
{
  description = "An example NixOS configuration";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-24.05";
    nixvirt.url = "https://flakehub.com/f/AshleyYakeley/NixVirt/*
    .tar.gz";
    nixvirt.inputs.nixpkgs.follows = "nixpkgs";
  };

  outputs = { self, nixpkgs, nixvirt }: {
    nixosConfigurations.nixos-desktop = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      modules = [
         nixvirt.nixosModules.default
        ./configuration.nix
        ./nixvirt-setup.nix
      ];
    };
  };
}
```

Listing 2.2: Example of simple Nix flake that declares a configuration for a NixOS system using a configuration file and a file that declares virtual networking setup using NixVirt.

At the top there are declared inputs, in this case using URLs. These inputs

are then used to declare the outputs. A system configuration, `nixosConfigurations.nixos-desktop` is defined by declaring the system architecture and the modules that define the configuration of the system. In this case there is a generic configuration file, `configuration.nix` and a user defined module that declares a virtual networking configuration that makes use of NixVirt, which itself is a flake. This flake in particular can be used to configure ones own system, but there are further possibilities. As an example it is also possible to define shell environments using **devShells**, but as that won't be used in this project, it will not be described.

The inclusion of a **lock file**, flake.lock which is generated on first build, ensures version control, thus preventing divergence between deployments caused by changes in upstream repositories. Furthermore, flakes promote **modularity** as seen in code snippet Listing 2.2. Configurations and environments can be packaged as reusable components and combined with others, making it easier to maintain consistency in complex distributed setups.

By providing both reproducibility and modularity, flakes strengthen the guarantees already offered by NixOS, ensuring that even large, **multi-machine environments** can be defined and reproduced while minimizing risk of inconsistency.

2.3 libvirt

libvirt is a tool for managing virtualization platforms, providing an interface to create, configure, and control virtual machines and their networking. In the context of reproducible environments, libvirt allows users to define **domains** — virtual machine instances with specific CPU, memory, and storage configurations — and **networks**, which specify the connectivity between domains.

Domains and networks can be created declaratively using XML, which enables consistent deployment of multiple virtual machines with predefined resource allocations and network topologies. This ensures that environments can be reproduced reliably across different hosts, supporting scenarios where multiple machines must interact in a controlled and predictable manner.

For this project, the specifics of the XML formats for domains and networks are not important, as Section 2.3.1 shall explain.

2.3.1 NAT Forwarding Network in libvirt

NAT, short for **N**etwork **A**ddress **T**ranslation, is the process of translating a set of IP addresses to another. It is usually used to map one or more private IPs to a public IP, such that all traffic from the private addresses appear as if

it was coming from the public address [3].

libvirt allows for defining NAT forwarding virtual networks that virtual machines can connect to, to be able to connect with each other, the host server and the internet (see Figure 2.2). Virtual machines connect to a virtual switch. On the network stack of the host server, the virtual network switch sits a a network interface — virbr0 on the diagram, which is the default bridge in libvirt — that directs the virtual machines' traffic through. The virtual networks allows for the use of **DHCP** (Dynamic Host Configuration Protocol), where the network can hand out IP addresses to connecting machines dynamically from a given range of addresses.

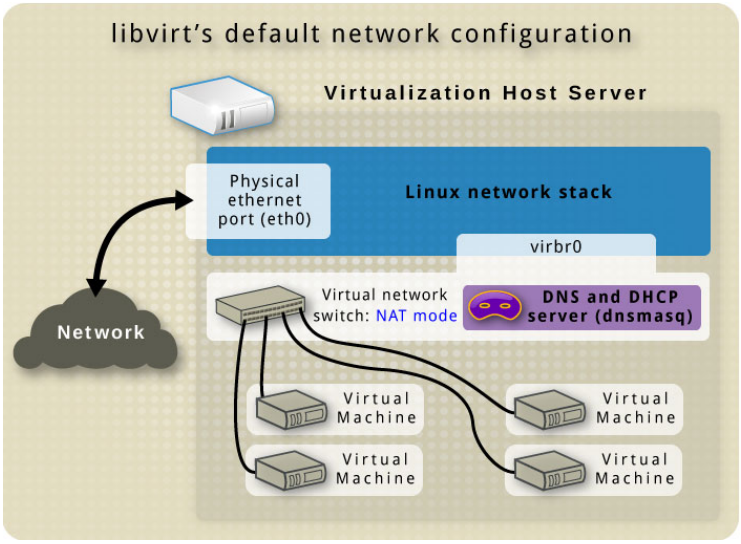


Figure 2.2: Diagram of the architecture of an example NAT forwarding libvirt network [5].

2.3.2 NixVirt

NixVirt is a Nix flake, which enables the configuration of libvirt services to be declared using the Nix Expression Language in flakes. The existence of NixVirt allows for a more consistent method of declaration of virtual environments by swapping out the need for XML with Nix. Among other elements, domains and network are declared as attributes sets. They are listed in **virtualisation.libvirt.connections.<connection>.domains** and **virtualisation.libvirt.connections.<connection>.networks**, which are lists of sets. These sets is constructed by 3 attributes, **definition**, **active**, and **restart**. Here the attribute **definition** the path to an XML declaring a domain or network. As said, no XML is necessary when using NixVirt

and that is because the user can generate XML from Nix using the function **lib.domain.writeXML**, which takes a Nix attribute set and translates it to the XML formats expected from libvirt [1].

Design

3

This chapter will outline the design choices for the Nix Spark cluster setup. The goal of the design is to provide a declarative and reproducible multi-node Spark cluster, that is easily deployable.

The environment is constructed on a single host, which is configured through a Nix flake that specifies both the host system packages and services, as well as a libvirt virtual network to interconnect virtual machines. The cluster itself is composed of multiple virtual machines (VMs) connected to this network, each acting as a Spark node in Standalone Deploy Mode. This approach ensures that the entire environment, including host configuration, VM specifications, and network, can be deployed consistently on any compatible host machine, that is, a machine running NixOS.

3.1 Host

The foundation of the cluster is the host, as it is here, that the cluster resides. The host is simple, and does not require much functionality. The most important tools it should have is a SSH Protocol connectivity , such as OpenSSH, and a virtualization manager, such as libvirt. SSH is necessary to remotely connect to the master node of the cluster to start Spark jobs (Note: it is also possible to start Spark jobs from the host using what's called *client mode*, but using client mode has the Spark driver running on the host and not the cluster, which isn't what we want). libvirt is of course needed to create the virtual network to connect the cluster. Lastly, internet connectivity is needed to build the VMs, and therefore a network configuration tool such as NetworkManager is needed.

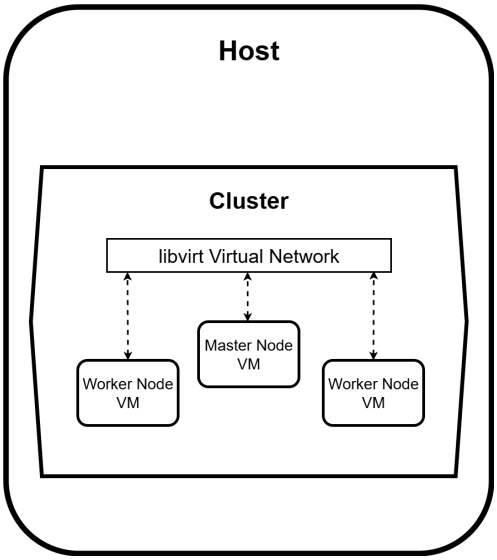


Figure 3.1: High level architecture of Nix Spark cluster design. The cluster is inside the host machine. VM nodes communicate through the virtual network as indicated by the arrows.

3.2 Cluster

The cluster can be divided into two sections: the nodes and the virtual network. The nodes are VMs, that are connected through the virtual network. The network is integral, as the master node needs to distribute jobs to executors on the worker nodes, and thus, the VMs have to be able to communicate. Let's first dive into the design of the nodes.

3.2.1 Nodes

First things first, the nodes trivially must include Apache Spark. To be able to run jobs, the nodes must also have installed the languages and libraries used in the Spark jobs, as the jobs will fail otherwise. The master node specifically must enable SSH connectivity from the host.

To make the processes of connecting worker nodes to the master node smooth, the nodes should have predefined IP addresses. The specific addresses are not important, provided that they are share subnet with the default gateway of the virtual network.

3.2.2 Virtual Network

To allow for seamless communication between individual nodes, as well as between host and cluster, a libvirt virtual network must be defined. Node access

to the internet is not strictly necessary, but still a NAT-based network should serve well for the use case, as a user might want to temporarily install a package or a library on nodes without having to redeploy the whole environment.

The network should have a fixed subnet and gateway, matching the subnet of the nodes. DHCP is not necessary, but might be a neat feature, if a user wants to quickly add a new worker node without redeploying the full environment.

3.3 Testing

To ensure the correctness and reproducibility of the designed environment, an integration testing strategy should be incorporated into the design. Since the system is composed of multiple interacting components, it is essential to validate that these elements function cohesively once deployed.

The testing should focus on the following objectives:

Host Validation: Confirm that the host is correctly provisioned with libvirt services running and the virtual network instantiated as declared in the flake.

VM Connectivity: Verify that all virtual machines are successfully instantiated, obtain addresses on the libvirt network, and can communicate with each other.

Cluster Formation: Ensure that the Spark master node is reachable and that all worker nodes successfully register with it over the private network.

Cluster Functionality: Execute a representative Spark job across the cluster to confirm that distributed execution operates as expected.

Implementation

4

As described in the last chapter, the implementation takes form in the shape of a Nix flake. This flake, along with separate custom Nix modules, define the host and cluster through configuration files for each system. Up until this point, it has only been implied that there are several worker nodes in the cluster, but to be precise, for simplicity's sake (and for the sake of the storage of my laptop) there are a total of three virtual machines used for cluster nodes: one for the master node and two for worker nodes. This is the simplest cluster possible, but can be easily scaled up to many more, by small changes in the implementation, as will be discussed later.

4.1 Flake

The center of the implementation is `flake.nix`. It is very simple, as it mostly serves as file where inputs and modules are imported to define the host. Inputs match those of the example in Listing 2.2, that is `nixpkgs` and `NixVirt`, with the only difference being the version of `nixpkgs`. The choice of version 24.11 is not particularly well thought out, it is merely the version that my own machine was running during development. The host system is defined in Listing 4.1.

```
nixosConfigurations.cluster = nixpkgs.lib.nixosSystem {
  specialArgs = { inherit inputs; };
  modules = [
    nixvirt.nixosModules.default
    ./hosts/configuration-cluster.nix
    ./modules/virtual-network.nix
    (import ./modules/user.nix)
  ];
};
```

Listing 4.1: Snippet from flake.nix showing the definition of the cluster host system.

Nothing fancy is going on here. Four modules are imported, where one of them, `nixvirt.nixosModules.default`, is required by NixVirt. The others three are declarations of the host system and the libvirt virtual network, which are described in the next section.

4.2 Host Configuration

Just as the flake, `hosts/configuration-cluster.nix` — the host system configuration file — is very simple. The least trivial parts are shown in Listing 4.2:

```
imports = [
    ../hardware/hardware-configuration.nix
  ];

  ....

  environment.systemPackages = with pkgs;  [
    spark
    python3
    python3Packages.pandas
    python3Packages.numpy
    python3Packages.pyspark
  ];
```

Listing 4.2: Snippet from hosts/configuration-cluster.nix showing the declaration of packages and hardware configuration.

Most important are the package declarations. None of the packages are actually strictly necessary for the host to function, but they give access to useful tools. `spark`, which of course is a Nix packaging for Apache Spark, is useful if the user does not wish to start Spark jobs using SSH, but would rather use Sparks client mode. `python` and the related libraries are there, should the user wish to create scripts on the host for jobs.

At the start, a file called `../hardware/hardware-configuration.nix` is imported. I will not go into details of this file, as it is a file automatically generated by Nix with relevant hardware information of the system machine. Users of this deployment are to put their own file there in replacement of the one placed in the directory by default.

4.2.1 Module: Virtual Network

The module `modules/virtual-network.nix` declares the configuration of libvirt on the host. It also declares the virtual network using NixVirt, which is added to libvirt as a network on build. A large snippet of the module can be seen in Listing 4.3.

```
let
  libvirt_nat_bridge = "virbr1";
  nixvirt = inputs.nixvirt;
in
{
  programs.virt-manager.enable = true;
  virtualisation.libvirtd = {
    enable = true;
```

```
  ...
};

virtualisation.libvirt = {
  enable = true;
  connections."qemu:///system".networks = [
    {
      definition = nixvirt.lib.network.writeXML {
        name = "sparknet";
        uuid = "6d405544-f162-4965-acdd-f3c4909db6e8";
        forward = {
          mode = "nat";
          nat = {
            port = { start = 49152; end = 65535; };
          };
        };
        bridge = {
          name = libvirt_nat_bridge;
        };
        ip = {
          address = "192.168.123.1";
          netmask = "255.255.255.0";
          dhcp = {
            range = { start = "192.168.123.100"; end = "
  192.168.123.200"; };
          };
        };
      };
      active = true;
    }
  ];
};
```

Listing 4.3: Snippet from modules/virtual-network.nix with libvirt configuration and the declaration of the virtual network.

At the top virt-manager and libvirtd — the libvirt **daemon** — are enabled, along with some settings of libvirtd being declared. The bulk and core part of the file is the configuration of `virtualisation.libvirt`, which is the NixVirt declaration of libvirt, specifically here the virtual network, aptly named `sparknet` and given a randomly generated UUID. As described in the Design chapter, the forward mode chosen is `NAT`, to allow for the virtual machines to reach the internet if necessary. The port range chosen is [49152; 65535] as recommended by IANA [4]. The IP address `192.168.123.1` along with the netmask `255.255.255.0` are as they match the `192.168.123.0/24` subnet, which is regularly used for private IPs. It will be important that the virtual machines have IPs on the same subnet, when their configurations are defined. DHCP is enabled, but isn't strictly necessary in this case due the fact, as will be seen later, that the VMs will get static IP addresses. Still it might be worth adding, which will be discussed in the Discussion chapter. A last important detail is the fact, that attribute `active` is set to true, making

sure that the network is active after being setup.

4.2.2 Module: User

The user module, as seen in Listing 4.4 declaratively defines the user profile of the host configuration.

```
{ config, lib, ... }:
{ userName ? "marcus" }: {
  users.users.${userName} = {
    isNormalUser = true;
    extraGroups = [ "wheel" "libvirtd" "networkmanager" ];
  };
}
```

Listing 4.4: User module modules/user.nix that declares the user of the host.

It is quite simple: the expression is a function, which takes the attribute set with attribute `userName` as input and outputs a user configuration. This username defaults to `"marcus"`, which was merely preferable for my machine. Users of this deployment can issue the preferred name of their user. It is implemented this way, so that if this is deployed on a system, where the host should have access to directories an already existing user has, one can set `userName` equal to that users username, and the new hosts user will have access automatically.

These 3 files make out the full configuration of the host system. Now let's look at the configuration of the virtual machines.

4.3 VM Configurations

This section dives into the configurations for the virtual machines. There are three files, each corresponding to an individual system configuration: `vm/master/configuration-master.nix`, `vm/master/configuration-worker1.nix`, and `vm/master/configuration-worker2.nix`. They are all very similar, and the section will therefore focus on the master configuration and thereafter pinpoint the differences between it and the worker configurations.

4.3.1 Master Configuration

There are several important points in the master configuration, of which a snippet can be seen in Listing 4.5. The snippet mostly skips the packages installed, which match those of the host configuration. First of all there is the networking declaration. DHCP is opted out of in trade for a static IP address `192.168.123.101`, sharing subnet with the virtual network.

```
{
  ...
```

```
  networking.useDHCP = false;
  networking.interfaces.eth0 = {
    ipv4.addresses = [
      {
        address = "192.168.123.101";
        prefixLength = 24;
      }
    ];
  };
  ...
  users.users.node = {
    ...
    openssh.authorizedKeys.keyFiles = [
      ../../keys/ssh-key.pub
    ];
  };
  ...
  networking.firewall.allowedTCPPorts = [
    22
    7077
    7079
    8080
    50051
  ];
  ...
  systemd.services.spark-master = {
    description = "Spark Master";
    after = [ "network.target" ];
    wantedBy = [ "multi-user.target" ];
    serviceConfig = {
      Type = "forking";
      Environment = [
        "SPARK_LOG_DIR=/var/log/spark"
        "SPARK_MASTER_HOST=192.168.123.101"
        "SPARK_MASTER_PORT=7077"
        "SPARK_DRIVER_HOST=192.168.123.101"
        "SPARK_DRIVER_PORT=7079"
        "PATH=/run/current-system/sw/bin:${pkgs.spark}/bin:${pkgs
    .spark}/sbin"
      ];
      ExecStart = "${pkgs.spark}/sbin/start-master.sh";
      Restart = "on-failure";
      RestartSec = "5";
      User = "root";
    };
  };
  ...
}
```

Listing 4.5: Snippet of the master configuration vm/master/configuration-master.nix.

In the user configuration, the user is called node for the sake of simplicity, and

then more importantly an SSH key is authorized from the file `../../keys/ssh-key.pub`. It is expected of the deployer, that they generate their own key and place it here, to be able to access the master node via SSH from the host machine. TCP port 22 is opened, as it is used by SSH, and 7077 is used by Spark for the web UI, should one be interested in using that (though that would require port forwarding of the port to a host port, which is not implemented).

The most interesting part of the implementation is the systemd service `spark-master`, which is declared at the bottom. Declaring this runs the `ExecStart` shell script when the virtual machines powers on, and the `ExecStop` before it powers off. `ExecStart` here runs a spark script `start-master.sh`, which sets up the machine as the master node of a cluster, while `ExecStop` stops Spark.

4.3.2 Worker Configurations

As mentioned, the worker configurations are very similar to the master, but there are a few key differences apart from just the host name. Just as the master, their static IPs are defined on the same subnet as `192.168.123.102` and `192.168.123.103`. As will be discussed later, this is not necessary, but helped in the early stages of development with testing that the nodes could communicate over the network. No SSH keys are authorized, as there is no need for connecting to them manually. Lastly, the most important distinction is, that the systemd service defined does not start them as master nodes, but as worker nodes, which use the host name of the master to connect to it to form the cluster.

4.4 Build & Deployment

In the directory you will find the subdirectory `scripts`, which contains two shell scripts: `build-vms.sh` (Listing 4.6) and `deploy-vms.sh` (Listing 4.7).

```
#!/usr/bin/env bash

project_dir=$(dirname "$(dirname "$(realpath "$0")")")
hosts_dir="$project_dir/hosts"
vms_dir="$project_dir/vms"
filename="nixos.qcow2"

for dir in "$vms_dir"/*; do
  config=$(find "$dir" -maxdepth 1 -type f -name "*.nix" -print -
    quit)

  qemu-img create -f qcow2 "$dir/$filename" 8G

  nix-build "<nixpkgs/nixos>" \
    -A config.system.build.isoImage \
    -I nixos-config="$config" \
```

```
    --out-link "$dir/iso"
done
```

Listing 4.6: Shell script scripts/build-vms.sh for building the virtual machines.

The build script handles building the VM disk files and installation media files. It does this in a single for-loop, first using `qemu-img` to create a QEMU Copy-On-Write Version 2 file (.qcow2) in each of the VM directories. These are the storage files for the virtual machines — used by libvirt to manage them — which all allow for 8 GB of storage, as specified in the command, which should be more than sufficient for the packages needed. Next NixOS Live CD files (.iso) are created from the VM configurations using the `nix-build` command setting the attribute flag to `config.system.build.isoImage`. The installation media files are placed in `vm/<vm-name>/iso` as symlinks to files packaged in the Nix store.

With these files built, the VMs can now be added as domains in libvirt. This is where `deploy-vms.sh` comes in.

```
#!/usr/bin/env bash

project_dir=$(dirname "$(dirname "$(realpath "$0")")")
vms_dir="$project_dir/vms"

for dir in "$vms_dir"/*; do
  name=$(basename "$dir")
  iso=$(find "$dir/iso/iso" -maxdepth 1 -type f -name "*.iso" -
    print -quit)
  qcow=$(find "$dir" -maxdepth 1 -type f -name "*.qcow2" -print -
    quit)

  sudo virt-install \
    --name=$name \
    --memory=1024 \
    --vcpus=2 \
    --disk $qcow,device=disk,bus=virtio,size=8 \
    --cdrom=$iso \
    --os-variant=generic \
    --boot=uefi \
    --nographics \
    --console pty,target_type=virtio \
    --network network=sparknet
done
```

Listing 4.7: Shell script scripts/deploy-vms.sh for deploying the virtual machines.

It runs a for-loop over the directories of the VMs and finds the files created in the build script. Then, for each VM, the `sudo virt-install` command is run using the disk and CD files to add the VMs as libvirt domains. Their libvirt names are equivalent to the <name> in the `vm/<name>` of their individual directory. Virtual memory and CPU cores are declared as 1 GiB of memory

and 2 cores, but a user could change this to fit their needs. Of course most importantly they are attached to `sparknet`.

When running the script, the user is prompted to type the `sudo` password and then for each VM an CLI installation process is started. When prompted the user should choose the first installer by pressing **enter**, after which the NixOS is installed onto the disk file, taking a few seconds. After installation the user must press **Ctrl+]** to escape the console of the VM and go on to the next installation. After the last installation, the cluster is official up and running, and the user can start Spark jobs.

4.5 Switching to the host configuration

Switching to the host configuration is done by running

```
sudo nixos-rebuild switch --flake .#cluster
```

If switching to the host configuration before building the and deploying the VMs, remember to initiate a WIFI connection using networkmanager.

4.6 Integration Test

`test/sparkClusterIntegrationTest.java` is a simple Java script using Spark API that tests whether work is distributed over the cluster. This section will not go into specific details of the code, but just give a simple explanation. The script starts a Spark session and generates a dataset. It then splits this dataset into 4 partitions. A function `partition_info` is then defined to return which task handles which partition and the amount of elements processed in that partition, along with the number of cores used for the task. The function is called on the partitions and the information is printed. Lastly, to involve some simple data processing, the sum of the square of all the elements of the dataset is calculated and printed, and spark is stopped, ending the script.

The file has been compiled using javac, yielding a `test.jar` file. From the host, the `test.jar` file is copied to the master nodes `/home/node` directory using the `scp` command. Submitting the the file as a Spark job is done by running

```
spark-submit --master spark://192.168.123.101:7077
             -- class SparkClusterIntegrationTest
          -- conf spark.driver.host=192.168.123.101
--conf spark.driver.port=7079 --conf spark.blockManager.port=50051
```

Evaluation

5

This short chapter presents the results of the test described in the previous chapter.

5.1 Test Results

The test had the following output:

```
Total sum of squares: 333338333350000
Partition 0 : 25000 elements | taskId=4 | cpus=1 | host=spark-worker1
Partition 1 : 25000 elements | taskId=5 | cpus=1 | host=spark-worker2
Partition 2 : 25000 elements | taskId=6 | cpus=1 | host=spark-worker1
Partition 3 : 25000 elements | taskId=7 | cpus=1 | host=spark-worker2
```

At the top we have the total sum of the squares of 1 to 100000. The rest shows for each partition, how many elements were processed in which task, on how many cpus and on which node.

Discussion

6

This chapter is a discussion of the project as a whole. It will discuss whether Nix and NixOS are fitting tools for creating reproducible Apache Spark environments. It poses answers to the questions: Is it a useful implementation now and in the future? What are possible improvements? But first the results of the test presented in the previous chapter are shortly discussed.

6.1 Discussion of Test

The tests gave the expected answer. The sum of the squares of the numbers 1 to 100000 is indeed 333338333350000, and as we can see, an equal amount of elements were processed on each worker, where each partition was processed on an individual core, meaning the data processing was parallelized, as the workers have 2 cores each.

6.2 NixOS for Reproducible Apache Spark Clusters

Generally, Nix and NixOS serve as a great tools for creating reproducible systems, and this project shows, that it is specifically great for the creation of reproducible clusters. The ability to define machine configurations declaratively in simple Nix expressions allows for complex clusters to be built and deployed within minutes. All it takes is to switch to the host configuration and run two shell scripts and everything is set up and ready to be used. A potential user would only have to add or remove some packages and change a couple of lines of code to fit their needs, i.e. changing the host username or adding an SSH key as described earlier.

Should the libvirt network configuration not suffice the use case of a deployment, adding or configuring networks is made very straight forward with NixVirt.

6.3 Shortcomings

With all the great points of the implementation, there are of course still parts of it that could use some work, which leads to the first shortcoming: It was a struggle to setup the VMs as domains in libvirt using NixVirt, and it was never accomplished, leading to `scripts/deploy-vms.sh`. It is suboptimal having to manually leave the console of each installation process, as were you to scale up the amount of nodes to hundreds or thousands, which some Spark clusters have, it would take a very long time to get through this process. Nix has ways to generate disk images directly from configurations, but through a lot of searching and many tries, it was not achieved to make these disk images work out of the box with libvirt, even though it was possible to spin them up as VMs with Nix's `nix-build '<nixpkgs/nixos>' -A vm` command for creating bootable VMs.

Having the individual worker configurations separate, as the current implementation has, doesn't really make sense. The only attributes they don't share are their hostnames and their IPs. Their hostnames might as well be the same and it is very simple to let the DHCP of the virtual network assign them IPs dynamically, as the deployer does not need their IPs for anything, at least in the scope of this project. Having a single worker configuration would allow for easy scale-ups of more nodes, to fit the clusters size to the users needs. This could easily be implemented by changing the build script slightly to take an input of the amount of worker nodes needed, and in turn it could make a directory for each node and build the files there. Trivially the DHCP range might need to be increased in the NixVirt declaration of the virtual network. This would make the deployment much more powerful.

A third point is the lack of health checks. If one of the nodes suddenly crash or lose connection to the network, there is no way for the user to know other than to manually check. This is obviously a problem as the user wouldn't necessarily be aware that anything was wrong with the cluster, if only some of the nodes weren't working. On top of health checks, automatic re-establishment of lost connection or crashed nodes would make the cluster more robust, leaving less work to the deployer.

Lastly, there is no real way of updating the system configuration, without building all the virtual machines from scratch, which takes a while, especially if scaled up. It is very simple to add or remove packages to the configurations, but without manually adding or removing them on each VM, one has to redeploy the whole system again, which is a shame if all you need is to add one extra important package or library.

If these shortcomings, along with getting the Spark Driver to work, were to be

addressed, the project could be a great tool for easy and quick deployments of reproducible virtual Apache Spark Clusters.

Conclusion

7

This project has investigated the process of creating Nix and NixOS derivations for deploying and managing Apache Spark clusters. The primary goal was to explore how Nix could be utilized to automate and simplify the setup of scalable Spark environments, ensure reproducibility, and support efficient resource management. This was specifically demonstrated by creating a deployable Apache Spark cluster using virtual machines on a single host.

Nix and NixOS have proven to be great tools for creating reproducible Apache Spark clusters. The Nix ecosystem's declarative specification of system dependencies, reproducible builds, and isolated environments is particularly well-suited for complex software stacks like those required for Apache Spark clusters. By defining machine configurations declaratively in Nix expressions, it is possible to build and deploy complex clusters within minutes.

The implementation centered around a Nix flake that configured the host system and a libvirt NAT forwarded virtual network using NixVirt. The cluster itself was composed of three virtual machines — one master and two worker nodes — that communicated via this virtual network. All VMs were configured with Apache Spark and the necessary languages and libraries, and the master node was enabled for SSH connectivity from the host. The implementation utilized static IP addresses for the VMs that matched the subnet of the virtual network. An integration testing strategy was incorporated to validate cluster functionality. Tests results confirmed that work was correctly distributed across the cluster, validating the designed environment's functionality.

Despite these successes, apart from the testing, several areas for improvement were identified:

Manual VM Setup: Configuring VMs directly as libvirt domains using NixVirt proved challenging, leading to the use of shell scripts and a manual installation process, which is suboptimal for scaling.

Separate Worker Configurations: The current individual worker configurations are separate, even though only hostnames and IP addresses differentiate them. A single, parameterizable worker configuration with dynamic IP assignment via DHCP would enable easier scaling of the cluster.

Lack of Health Checks: The system currently lacks health checks and automatic re-establishment of lost connections or crashed nodes. This could lead to a lack of awareness regarding cluster issues.

Update Process: Updating the system configuration presently requires rebuilding all virtual machines from scratch, which is time-consuming and inefficient for minor changes.

Addressing these shortcomings would enable the project to serve as a powerful tool for the rapid and straightforward deployment of reproducible virtual Apache Spark clusters. The fundamental approach leveraging Nix and NixOS lays a solid foundation for bringing the benefits of functional package management to the domain of big data processing.

Bibliography

[1] AsheleyYakeley. Nixvirt. https://github.com/AshleyYakeley/NixVirt. Accessed: 2025-08-27.

[2] Dolstra, E., Löh, A., and Pierron, N. Nixos: A purely functional linux distribution. *J. Functional Programming 20*, 5 6 (2011), 577–615.

[3] IBM. Network address translation. https://www.ibm.com/docs/sr/i/7.4.0?topic=ucc-network-address-translation, 2023. Accessed: 2025-08-28.

[4] Internet Assigned Numbers Authority. Service name and transport protocol port number registry. https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml. Accessed: 2025-08-28.

[5] libvirt. Virtual networking. https://wiki.libvirt.org/VirtualNetworking.html. Accessed: 2025-08-28.

[6] nix-docs. mkderivation. https://blog.ielliott.io/nix-docs/mkDerivation.html. Accessed: 2025-08-27.

[7] NixOS Foundation. Nix language basics. https://nix.dev/tutorials/nix-language. Accessed: 2025-08-27.

[8] NixOS Foundation. Profiles. https://nix.dev/manual/nix/2.25/package-management/profiles. Accessed: 2025-08-27.

[9] NixOS Wiki. Nix package manager. https://nixos.wiki/wiki/Nix_package_manager. Accessed: 2025-08-14.

[10] NixOS Wiki. Overview of the nix language. https://nixos.wiki/wiki/Overview_of_the_Nix_Language. Accessed: 2025-08-14.

[11] The Apache Software Foundation. Apache spark - a unified engine for large-scale data analytics. https://spark.apache.org/docs/latest/. Accessed: 2025-08-08.

KU's AI-deklaration

Deklaration for anvendelse af generative AI-værktøjer
☒ **Jeg/vi har benyttet generativ AI som hjælpemiddel/værktøj** *(sæt kryds)* ☐ **Jeg/vi har <u>IKKE</u> benyttet generativ AI som hjælpemiddel/værktøj** *(sæt kryds)* *Hvis brug af generativ AI er tilladt til eksamen, men du ikke har benyttet det i din opgave, skal du blot krydse af, at du ikke har brugt GAI, og behøver ikke at udfylde resten.*
Oplist, hvilke GAI-værktøjer der er benyttet, inkl. link til platformen (hvis muligt): *Gratis version af ChatGPT https://chatgpt.com/*
Beskriv hvordan generativ AI er anvendt i opgaven: *1) Formålet ved brugen var til at lære om brugen af Nix , Spark og Bash, herunder brugbare kommandoer og eksempler på stykker af kode. I starten blev det også brugt til ide-generering.* *2) Det blev brugt gennem hele arbejdsforløbet.* *3) Outputtet blev brugt til at blive klogere på stoffet, og er ikke kopieret ind i rapporten eller brugt som direkte kilde.* *NB.* GAI-genereret indhold brugt som kilde i opgaven kræver korrekt brug af citationstegn og kildehenvisning. Læs retningslinjer fra Københavns Universitetsbibliotek på KUnet