Table of Contents:

[About the Project](#_dfdp5jfft0zn)

[The System](#_sal02jgu7uo4)

[Built With](#_iwjymke4ayxy)

[Getting Started](#_8m9hen7oggyk)

[Prerequisites](#_v1gn6ic02b4p)

[AWS EC2 Set Up](#_rigczamg6quq)

[Installation and Setup](#_4rsyx5ci48gi)

[Implementation](#_ubo4u6sg7dlg)

[Part 1: Clone the repository](#_r3fsd4hxx0ho)

[Part 2: Copy PEM private key on to EC2 Instance](#_92oiofdzukec)

[Part 3: Update Terraform variables](#_gwono0o9gvun)

[Part 4: Provision the instance via Terraform](#_rmk0bgd9fqv)

[Part 5: Update Ansible hosts](#_7fufeklmgo1o)

[Part 6: Run your Ansible Playbook](#_ufy96hnoaqpl)

[Monitoring](#_pie26faz4sdw)

[Tailing the node log file](#_ymtdvxrofq6p)

[Utilising Amazon CloudWatch](#_7z57yjnhz9v3)

[Security](#_9en3vjv2sq0q)

[Access and Private Keys](#_q7kpk3pao8xa)

[Security Groups](#_cd8lyws2q62u)

[Security package update](#_cup4ww77be29)

[Upgrade software deployed and minimise downtime](#_20w17lyd5kst)

[Issues faced](#_nnzmxrbz7nhv)

[Improvements/Limitations](#_41jvu7if6mo)

[Setting up a Full Node on Crypto.org Mainnet](#_rgk3s1qcj7p)

[Improvements to proposed system](#_o6bqqkdizxsp)

Please Note: The below guide is a detailed documentation which covers all steps on how to set up a Full Node on Crypto.org mainnet and reproduce the deployment with IAC and Automation tools. If you are an advanced user who just requires access to the Ansible and Terraform files, then these are available in the following public Git repository [armaanPy/DxJSH237891z (github.com)](https://github.com/armaanPy/DxJSH237891z). The files have also been added to this Drive folder under folder DxJSH237891z. If you just wish to make use of those files separately then please be aware that the files private key variables, aws region, ansible host information will need to be changed to match your own - this has been documented in the guide.

## About the Project

This project contains detailed documentation for setting up a Full Node on Crypto.org mainnet. The full node is packaged and is deployed through Ansible on to AWS hosts that you will be provisioning using Terraform. By following this documentation, you will be able to provision multiple hosts and deploy as many Crypto.org Mainnet Full Nodes as you please.

### The System

The system in itself is packaged code that allows for the ability to provision cloud based Linux servers on-demand, and then with the use of automation, automate a deployment on to those provisioned servers. All in all this system reduces the overhead and complexity that comes with attempting to set up a Full Node on Crypto.org mainnet.

Before creating this system, I ran through a manual installation and set up of an EC2 Linux server which hosted a Full Node on Crypto.org mainnet. This showed me the limitations of the installation, and paved the way for how I was going to implement my system. I understood that as the process was quite mundane and tedious, that automation was going to be a key factor in my system. Therefore in my system I made sure to implement automation at every step; provisioning the Linux server, installing all required packages, and deploying the installation. The use of automation makes it very easy for users to reproduce this system. Please see below to see what this system was built with.

### Built With

The system is built using:

* Crypto.org Chain Mainnet
* AWS
* Terraform
* Ansible
* Bash scripts

## Getting Started

### Prerequisites

* Spin up an EC2 Instance on AWS.
* Install AWS CLI and configure access keys on EC2.
* Install Git on EC2.
* Install Terraform on EC2.
* Install Ansible on EC2.
* Install WinSCP on local machine.

### AWS EC2 Set Up

Please note: An EC2 instance is not explicitly needed for the deployment, you can also follow the guide on your own Linux servers, however the packages that are to be installed are explicitly for Amazon Linux 2, therefore please find alternatives.

1. If you do not already have one, create an AWS account. <b>If you already have an AWS account with an EC2 instance already running, then please skip to the Installation and Setup section</b>.

[How do I create and activate a new AWS account?](https://aws.amazon.com/premiumsupport/knowledge-center/create-and-activate-aws-account/)

2. Once you have created your AWS account, please proceed to launch an Amazon Linux 2 AMI (HVM) - Kernel 5.10 EC2 instance.

[How to launch your instance.](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/LaunchingAndUsingInstances.html)

3. You will need a key pair to securely connect to your instance, you must also ensure that you have access to the selected key pair before your launch the instance. Alternatively, you can connect using EC2 Instance Connect. Please refer to the link below on how to connect to your Linux instance once it has been provisioned.

[Connect to your Linux instance.](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AccessingInstances.html)

### Installation and Setup

* Once you have connected to your Linux EC2 instance, you will need to install Git, Terraform and Ansible in order to carry out the steps in this documentation. Please refer to the following vendor documentation on how to install and set up the applications on your Linux instance.
* I have created a Bash script that will automate the installation of all the packages listed below and also will set up your git config (Step 2). If you do not wish to manually install the packages, then please copy over the install.sh script which is located in both the GitHub repository linked at the start of the guide and also the DxJSH237891z folder. It may be easier to simply create an install.sh file on your instance and copy over the contents manually from Git or from the Drive folder (as this script will install and set up Git for you).
* To run the install.sh script, as mentioned above, copy it over to your EC2 instance and run the following commands to make it executable and then run the script.

| $ chmod +x install.sh $ ./install.sh |
| --- |

* After you have run the install.sh script, you may proceed to “Step 6. Create access keys for Terraform”.
* If you wish to install and set up the packages manually, ignore the above step and commence with “Step 1. Install Git on EC2 instance” below.

1. [Install Git on EC2 instance](https://www.atlassian.com/git/tutorials/install-git#linux) (If you spun up an Amazon Linux 2 AMI then follow the Fedora (dnf/yum) section.)

2. After you have followed the above documentation to install Git, you will need to configure your user name and email address. This is important because every Git interaction uses this information, and you will need to set this up before you can clone a repository. To do so enter the following commands replacing the pseudocode with your own Git username and email:

| $ git config --global user.name "John Doe" $ git config --global user.email johndoe@example.com |
| --- |

3. [Install Terraform on EC2 instance](https://www.terraform.io/downloads.html)

4. [Install Ansible on EC2 instance](https://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html)

* If you spun up an Amazon Linux 2 EC2 Instance, then the command you will have to use to install Ansible is as follows, however refer to the vendor documentation for in depth information:

| $ sudo amazon-linux-extras install ansible2 |
| --- |

5. Install the AWS CLI version 2 on Linux.

* Follow these steps from the command line on your EC2 instance to install the AWS CLI on Linux.

| $ curl "https://awscli.amazonaws.com/awscli-exe-linux-x86\_64.zip" -o "awscliv2.zip"  $ sudo yum install unzip -y $ sudo unzip awscliv2.zip $ sudo ./aws/install $ aws --version |
| --- |

6. Create access keys for Terraform.

* To do so, log on to the AWS Management Console, click on your username in the top right hand corner and go to "My Security Credentials".
* Then click on "Access keys" and click on "Create New Access Key".
* This will have created an access key, be sure to download the access key to your local machine as we will be needing it later.

7. Configure your access keys on your EC2 instance.

* On your EC2 command line, enter:

| $ aws configure |
| --- |

* You will then be prompted to enter your AWS Access Key ID, AWS Secret Access Key and Default region name.
* You can refer to the Access Key you downloaded in step 6 to fill out the first 2 entries, and for the default region name, please use the region you are currently based in (and where you have spun up the EC2 instance). You can find the region name from the top right hand corner i.e. "ap-east-1".
* You can leave the Default output format as blank.

8. [Install WinSCP on your local machine](https://winscp.net/eng/download.php)

## Implementation

### Part 1: Clone the repository

* Log on to your EC2 instance and enter the following command to clone this repository:

| $ git clone https://github.com/armaanPy/DxJSH237891z |
| --- |

### Part 2: Copy PEM private key on to EC2 Instance

* In order for Terraform and Ansible to be able to communicate with our EC2 Instance we need to generate a key pair and copy that over to our EC2 instance.

* To do so, first we will need to generate a key pair. To do this, navigate to EC2 on the AWS Management Console and on the menu bar navigate to "Network & Security" -> "Key Pairs".
* In the Key Pairs page, click on "Create Key Pair" on the top right hand corner of the page.
* Under the Name section, enter: crypt0def1x
* Please note: you can name the private key whatever you wish, however for the purpose of this demo I have instructed to name it as crypt0def1x as this is the variable configured in the Terraform variables.tf file. If you wish to choose a different name please do so, and I will advise on the changes needed later on in Step 3.
* For Key pair type, select "RSA".
* Finally for the Private key file format, select ".pem". Once you click create, the key file will be downloaded onto your local machine - keep this safe.
* Now using WinSCP, connect to your EC2 instance (using the initial key pair you created to access the EC2 in the first section of the guide).
* Once you are in WinSCP, copy the crypt0def1x.pem file to the ~/.ssh/ location. Note: In order to see the hidden .ssh file you will need to go to WinSCP -> Options -> Preferences -> Panels and tick "Show hidden files"
* Once the crypt0def1x.pem file has been transferred over to ~/.ssh/ on your EC2 instance, you will need to make sure it is not publicly viewable, to do so run the following commands.

| $ cd ~/.ssh $ chmod 400 crypt0def1x.pem |
| --- |

### Part 3: Update Terraform variables

* Within the ~/DxJSH237891z/Zjdh377SZx/variables.tf file, you will need to update the following aws\_region default variable to whichever AWS region you wish to provision the EC2 node which will host the Blockchain installation.

| $ vi ~/DxJSH237891z/Zjdh377SZx/variables.tf |
| --- |

Edit:

| variable "aws\_region" {  description = "Hong Kong"  default = "ap-east-1" } |
| --- |

* i.e. If you wish to provision the node in North Virginia (us-east-1), then change the description and default variables as intended.
* If you can recall in Step 2, you had the ability to choose your own private key name, if you did not proceed with the placeholder name and created a private key with a different name then you will need to rename this variable within the ~/DxJSH237891z/Zjdh377SZx/variables.tf file:

| $ vi ~/DxJSH237891z/Zjdh377SZx/variables.tf |
| --- |

Edit:

| variable "key\_name" {  description = " SSH keys to connect to ec2 instance"  default = "crypt0def1" } |
| --- |

### Part 4: Provision the instance via Terraform

* Now it's time to provision your instance, to do so:

| $ cd ~/DxJSH237891z/Zjdh377SZx/ $ terraform init $ terraform plan $ terraform apply |
| --- |

* Once your resources have been created you will notice an output detailing the "vm\_public\_ip" - note this down as we will need it for our Ansible step. Alternatively, you can find the public IP of the newly provisioned instance from the EC2 page.
* Now if you navigate to the EC2 page, you will see that your newly provisioned instance is now visible with the same public IP that was output.

### Part 5: Update Ansible hosts

* Now you have to update the Ansible inventory.yml with the public IP of the new instance you just provisioned. To do so:

| $ cd ~/DxJSH237891z/Yh73973jS/ $ vi inventory.yml |
| --- |

* Replace "[Enter Public IP]" with the public IP that was output after your Terraform resource was created (or alternatively get the public IP of your host from the EC2 Instances page.

### Part 6: Run your Ansible Playbook

* Run your Ansible Playbook using the following commands:

| $ cd ~/DxJSH237891z/Yh73973jS/ $ ansible-playbook -i inventory.yml application.yml -v |
| --- |

* The ansible-playbook will create a fully fledged Crypto.org Chain Mainnet installation and run it in the background whilst outputting to a log file.

## Monitoring

### Tailing the node log file

* Now that your node is up and running you can monitor the status of the node and view logs which will output its real-time state. To do so, you will need to connect to the newly provisioned instance via SSH on the initial host where you transferred over the crypt0def1x.pem.
* To do this, first you will need to obtain the Public DNS of the newly provisioned EC2 instance, you can find this from the AWS EC2 Instances page. Once you have that you can run the following command to ssh on to the box and then tail the log file which we generated in the ansible-playbook:

| $ ssh -i "~/.ssh/crypt0def1x.pem" ec2-user@[Public DNS] $ tail -f node.log |
| --- |

* Here's an example of the ssh command (this will differ based on your Public DNS):

| $ ssh -i "~/.ssh/crypt0def1x.pem" ec2-user@ec2-21-153-296-123.ap-east-1.compute.amazonaws.com |
| --- |

### Utilising Amazon CloudWatch

* You can use Amazon CloudWatch Logs to monitor, store, and access your log files from Amazon Elastic Compute Cloud (Amazon EC2) instances - which makes it an ideal tool to monitor the logs/health of your node.
* [How to Install and configure the CloudWatch Logs agent on a running EC2 Linux instance](https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/QuickStartEC2Instance.html)

## Security

### Access and Private Keys

* To ensure the security of this deployment I have made sure to make sure of AWS Access Keys and Secret Access Keys built into the EC2 instance rather than creating a separate Terraform file which will hold those details. This makes sure that your Access and Secret Access Keys are encrypted within the EC2 instance.
* For maximum security, you are in charge of who has access to this deployment. In part 2 you created and copied over a private key on to your EC2 instance, at the time of creating this key, only the creator has access and can distribute it as they please. This is to ensure only you have access over the deployment and other users who can also access it.

### Security Groups

* An EC2 Security Group was also defined in the main.tf file which provisioned the EC2 instance. This security group only allows for ingress traffic from port 22 (which is the SSH port), this means that you can only access the instance via SSH. This was implemented to stop the occurrence of brute force attacks or API floods from the HTTP/HTTPS ports and so on. If you wish to access the deployment by a means other than SSH then you can alter the main.tf file to create a Security Group of your liking.

### Security package update

* In the Ansible-playbook I also implemented a task that would update all the packages on the instance and remove any obsolete packages. This was to ensure the security of the host where the deployment is taking place. To put into context how this step helps with security, this task to update the packages would have updated all applications which use Log4J, therefore you would not have been affected by the Log4Shell exploit.

| 1 package(s) needed for security, out of 8 available Run "sudo yum update" to apply all updates. |
| --- |

## Upgrade software deployed and minimise downtime

* In the ~/DxJSH237891z/Yh73973jS/ location which holds the Ansible files, I have created 2 additional Ansible scripts which can be used to upgrade the software deployed whilst minimising downtime. It does this by first creating all of the directories and configuration needed for the upgrade, once that is done it will run through a Bash script that I created that contains a while loop that will only exit once the current installation prompts that an upgrade is needed. Once this prompts it will start the new chain-maind. TO DO: Figure out how to stop the old chain-maind first before starting upgraded version.
* Steps taken in script to minimise downtime:
  + Keep the current node running.
  + Whilst the current node is running, download and extract the upgraded Chain binaries.
  + Stop the old node only when upgrade files are downloaded and set up accordingly.
  + Start the new node only when the old version prompts that upgrade is needed.
* How to ensure blockchain data is persistent across upgrades and can be backup:
  + Will need to make a backup of the /home/ec2-user/.chain-maind/data directory and copy this over to where the new upgraded node will run. In the case of my current script, we will be upgrading on the same box as where this directory already exists - so blockchain data will be persistent.
  + This can then be copied over to the remote host where the upgrade will take place.

## Issues faced

* Along the way there were many issues that were created with the implementation of this deployment, I have listed these below along with the steps I took to resolve them.

1. The sed commands to update minimum gas price to stop transaction spamming and modify the configuration of persistent\_peers and create\_empty\_blocks\_intervals had quotes within them which were breaking the application.yml YAML syntax, I therefore had to use the YAML literal block string syntax to not have to escape any quotes.
2. When trying to reach the Public IP of the Terraform provisioned host I was getting an error that "Permissions 0664 for 'crypt0def1.pem' are too open." Therefore I had to ensure that my key was not publicly viewable by running chmod 400 crypt0def1.pem.
3. On attempt to update all the packages on the Linux box I faced the following error {"changed": false, "msg": "No package matching 'update' found available, installed or updated"} to resolve this I had to make changes to the Ansible-playbook and I instead used "yum: name=\* state=latest" in the task to update, and this resolved it.
4. I was unable to pass or fail the step where I need to verify the sha256sum checksum of the crypto genesis.json is "OK". I tried using an expect response and logging the output under a debug task but to no avail therefore I implemented a shell check in the Ansible script and advised to run the playbook in verbose mode so you can see the status of this verification.
5. The Ansible script would create directories but failed in starting up the node due to the error ""~/.chain-maind/config/genesis.json" does not exist", therefore after thorough investigation I found that the ~/.chain-maind/ directory only generates after you run a command, so I ran the ./bin/chain-maind version command in the script to generate it. This step worked when deploying manually, however when I implemented a task in the Ansible script to run the version command to generate the .chain-maind directory, it would still not generate. Therefore I had to implement a step in the Ansible playbook that would copy the directory from local and shift it to the remote provisioned host.
6. The Ansible job would not end as it was continuously running through the chain-maind script, so I made Ansible start it in the background and output the log to another file.

## Improvements/Limitations

* This section will entail the improvements to setting up a Full Node on Crypto.org Mainnet and also the limitations to the system proposed in this guide with enhancements that could be implemented to make the system more efficient.

### Setting up a Full Node on Crypto.org Mainnet

1. After installing and extracting the Crypto.org Chain binaries (chain-maind) from GitHub onto my local machine, the ~/.chain-maind/ directory only became present after running a ./chain-maind command. The ~/.chain-maind/ is an important directory that the installation cannot function without as it contains the app.toml and config.toml files. I became acquainted with this issue after attempting to install the installation via Ansible. If you are to follow the Crypto.org docs and implement them into an ansible-playbook for automation, then the ~/.chain-maind/ will not generate and the installation will cease to run. This became apparent after my script tried to run the installation and came across a “genesis.json file does not exist error”. To work around this in the automation, I had to copy the ~/.chain-maind/ directory from local to remote as well as then create the data directory within it. An improvement to setting up a Full Node on Crypto.org Mainnet for automation users would be to perhaps make the ~/.chain-maind/ not a hidden file, or ensure that this file is generated when the Chain binaries are extracted from GitHub.
2. Taking into consideration the “Canis Major” network upgrade at the block height of 922,363, I believe another improvement (especially for automation users) would be to also include the v2.1.2 chain binary within the first step of the documentation, rather than installing v1.2.1 first and then manually carrying out the upgrade to v2.1.2 after the block height reaches 922,363. This will make the installation much smoother for automation users, as they will not need to amend their automation scripts to take this into account.

### Improvements to proposed system

1. The first improvement I would make would be to make it a completely automated deployment with no manual intervention. At present, the user will have to manually create access keys and configure them in their EC2 instance.
2. At present the guide has been explicitly created for Amazon Linux 2 instances, although all Linux OS are relatively similar, in the case for Amazon Linux 2 there are different packages available and the method of installing these packages also differs. Therefore a limitation to the current system is that the entire configuration outlined in this guide only works for Amazon Linux 2. Therefore as an improvement, I could add in Bash scripts to take into account every OS package rather than the user manually having to find packages themselves.
3. Instead of using Bash scripts I could implement Python scripts as Python is a programming language as opposed to Bash which is a command-line interpreter. Python is more efficient than Bash and is known for its consistency and readability in contrast to Bash.
4. Another improvement I could make is to utilise Ansible modules more often rather than simply executing shell commands in my playbook. This will increase readability and performance as it will be using Ansible’s own modules.