National College of Ireland

# Configuration Manual

MSc Research Project
Cloud Computing


## Ashwin Sabu
Student ID: 23196505

School of Computing
National College of Ireland


Supervisor: Prof. Diego Lugones

**National College of Ireland**
**Project Submission Sheet**
**School of Computing**

National College of Ireland

| | |
|---|---|
| **Student Name:** | Ashwin Sabu |
| **Student ID:** | 23196505 |
| **Programme:** | Cloud Computing |
| **Year:** | 2024-2025 |
| **Module:** | MSc Research Project |
| **Supervisor:** | Prof. Diego Lugones |
| **Submission Due Date:** | 12/12/2018 |
| **Project Title:** | Configuration Manual |
| **Word Count:** | 5501 |
| **Page Count:** | 18 |

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

**ALL** internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

| | |
|---|---|
| **Signature:** | Ashwin Sabu |
| **Date:** | 11th December 2024 |

## PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

| | |
|---|---|
| Attach a completed copy of this sheet to each project (including multiple copies). | ☐ |
| **Attach a Moodle submission receipt of the online project submission**, to each project (including multiple copies). | ☐ |
| **You must ensure that you retain a HARD COPY of the project**, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer. | ☐ |

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

| **Office Use Only** | |
|---|---|
| Signature: | |
| Date: | |
| Penalty Applied (if applicable): | |

# Configuration Manual

Ashwin Sabu
23196505

# 1 Install and Configure Hyperledger Fabric

## 1.1 EC2 Installation

The first step would be to create an EC2 instance on which Hyperledger Fabric would be installed. For that create an EC2 instance in eu-west-1 region from the AWS Console following the below configuration. Make sure to create a Key Pair if does not have as it would be needed in the next step to login to the instance through command prompt.

| Configuration Parameter | Value |
|---|---|
| Instance Type | t2.small |
| AMI | Ubuntu 22.04 |
| Storage | 30 GB |
| Security Group | Allow All |

Table 1: EC2 Instance Configuration

On creating the EC2 instance log into the instance using SSH from the location where the Key Pair file is located following the below command :

```
ssh -i "file_name.pem" ubuntu@Public_IPv4_DNS
```

## 1.2 Install Prerequisites

Updating the exising packages and installing necessary packages would be the first step for the installation of HLF in the EC2 instance (Makers (2014)).

Update the package manager by running the following command.

```
$ sudo apt-get update
$ sudo apt-get upgrade -y
```

Install Docker by runing the following commands :

```
$ sudo apt-get install -y docker.io
$ sudo systemctl start docker
$ sudo systemctl enable docker
```

Install Docker Compose by running the following command

```
$ sudo apt-get install -y docker-compose
```

Once Docker installation is successful, run the following command command to install GO with version 1.20 and also set the path to the required location.

```
$ wget https://go.dev/dl/go1.20.5.linux-amd64.tar.gz
$ sudo tar -xvf go1.20.5.linux-amd64.tar.gz -C /usr/local
$ export PATH=$PATH:/usr/local/go/bin
```

Last step would be to install Node JS and NPM by running the following command (*Install Node.js on Amazon Linux 2 :: INTRODUCTION TO AMAZON EC2.* (n.d.)).

```
$ sudo apt install nodejs
```

## 1.3 Install Hyperledger Fabric Components

Run the following command to install HLF in the EC2 instance.

```
$ curl -sSL https://bit.ly/2ysbOFE | bash -s
```

Ensure that the installation is successful by running the following command. If successfull there would be different components in the format hyperledger/

```
$ docker images
```

## 1.4 Configure the network

Move to test-network directory location and start the network by running the following command.

```
$ cd fabric-samples/test-network
$ ./network.sh up
```

Inorder to initialise the chaincode, edit the file smartcontract.go with the following code. This ensures that the smart contract aaccepst two parameter ID and SHA256 value, which then is used to store in HLF.

```
$ vi  /fabric-samples/asset-transfer-basic/chaincode-go/chaincode/smartcontract.go
```

```go
package chaincode

import (
    "encoding/json"
    "fmt"

    "github.com/hyperledger/fabric-contract-api-go/v2/contractapi"
)

// SmartContract provides functions for managing an Asset
type SmartContract struct {
    contractapi.Contract
}
// Asset describes basic details of what makes up a simple asset
type Asset struct {
    ID        string `json:"ID"`
    SHAValue string `json:"SHAValue"`
}
```

```
19
20 // InitLedger adds a base set of assets to the ledger
21 func (s *SmartContract) InitLedger(ctx contractapi.
       TransactionContextInterface) error {
22     assets := []Asset{
23         {ID: "asset1", SHAValue: "abc123"},
24         {ID: "asset2", SHAValue: "def456"},
25     }
26
27     for _, asset := range assets {
28         assetJSON, err := json.Marshal(asset)
29         if err != nil {
30             return err
31         }
32
33         err = ctx.GetStub().PutState(asset.ID, assetJSON)
34         if err != nil {
35             return fmt.Errorf("failed to put to world state. %v", err)
36         }
37     }
38
39     return nil
40 }
41
42 // CreateAsset issues a new asset to the world state with the given ID
       and SHA_Value.
43 func (s *SmartContract) CreateAsset(ctx contractapi.
       TransactionContextInterface, id string, shaValue string) error {
44     asset := Asset{
45         ID:       id,
46         SHAValue: shaValue,
47     }
48
49     assetJSON, err := json.Marshal(asset)
50     if err != nil {
51         return err
52     }
53
54     return ctx.GetStub().PutState(id, assetJSON)
55 }
```

Deploy the chaincode created now by running the following command.
`$ ./network.sh deployCC -ccn assettransfer -ccp ../asset-transfer-basic/chaincode-go -ccl go`

## 1.5 Create API to Store Data

Next action item would be to set up a connection to the outside network to store data to the blockchain and for that express framework is used to connect and store the data. Here the environment is set into the location of `$ /home/ubuntu/fabric-samples/test-network` by creating a folder and installing express framework.

```
$ mkdir hlf-api
$ cd hlf-api
$ npm init -y
$ npm install express fabric-network
```

```
$ vi server.js
```

JS file inorder to establish the connection.

```
package chaincode

import (
    "encoding/json"
    "fmt"

    "github.com/hyperledger/fabric-contract-api-go/v2/contractapi"
)

// SmartContract provides functions for managing an Asset
type SmartContract struct {
    contractapi.Contract
}

// Asset describes basic details of what makes up a simple asset
type Asset struct {
    ID        string `json:"ID"`
    SHAValue string `json:"SHAValue"`
}

// InitLedger adds a base set of assets to the ledger
func (s *SmartContract) InitLedger(ctx contractapi.
   TransactionContextInterface) error {
    assets := []Asset{
        {ID: "asset1", SHAValue: "abc123"},
        {ID: "asset2", SHAValue: "def456"},
    }

    for _, asset := range assets {
        assetJSON, err := json.Marshal(asset)
        if err != nil {
            return err
        }

        err = ctx.GetStub().PutState(asset.ID, assetJSON)
        if err != nil {
            return fmt.Errorf("failed to put to world state. %v", err)
        }
    }

    return nil
}

// CreateAsset issues a new asset to the world state with the given ID
   and SHA_Value.
func (s *SmartContract) CreateAsset(ctx contractapi.
   TransactionContextInterface, id string, shaValue string) error const
    express = require('express');
const { Gateway, Wallets } = require('fabric-network');
const path = require('path');
const fs = require('fs');

```

```
const app = express();
app.use(express.json());

// Path for connection profile and credentials
const ccpPath = path.resolve(__dirname, '..', 'organizations', 'peerOrganizations', 'org1.example.com', 'connection-org1.json');
const credPath = path.resolve(__dirname, '..', 'organizations', 'peerOrganizations', 'org1.example.com', 'users', 'Admin@org1.example.com', 'msp');
const certPath = path.join(credPath, 'signcerts', 'Admin@org1.example.com-cert.pem');
const keyPath = path.join(credPath, 'keystore', fs.readdirSync(path.join(credPath, 'keystore'))[0]);

// Read the connection profile and credentials only once
let ccp, cert, key;

const loadConfig = () => {
  return new Promise((resolve, reject) => {
    try {
      ccp = JSON.parse(fs.readFileSync(ccpPath, 'utf8'));
      cert = fs.readFileSync(certPath).toString();
      key = fs.readFileSync(keyPath).toString();
      resolve();
    } catch (err) {
      reject(err);
    }
  });
};

// Initialize wallet and Fabric Gateway for reuse
let wallet, gateway;
const initializeFabricConnection = async () => {
  wallet = await Wallets.newInMemoryWallet();
  const identity = {
    credentials: { certificate: cert, privateKey: key },
    mspId: 'Org1MSP',
    type: 'X.509',
  };
  await wallet.put('admin', identity);

  gateway = new Gateway();
  await gateway.connect(ccp, { wallet, identity: 'admin', discovery: { enabled: true, asLocalhost: true } });
};

// API to create an asset
app.post('/storeData', async (req, res) => {
  const { id, shaValue } = req.body;

  if (!ccp || !cert || !key) {
    // Load configuration if it's not already loaded
    try {
      await loadConfig();
      await initializeFabricConnection();
    } catch (err) {
      return res.status(500).send({ error: `Configuration loading failed: ${err.message}` });
```

```
    }
  }

  try {
    const network = await gateway.getNetwork('mychannel');
    const contract = network.getContract('assettransfer');

    // Submit the transaction
    await contract.submitTransaction('CreateAsset', id, shaValue);

    res.status(200).send({ message: 'Data stored successfully' });
  } catch (error) {
    console.error(`Failed to submit transaction: ${error}`);
    res.status(500).send({ error: error.message });
  }
});

const PORT = 3000;
app.listen(PORT, () => {
  console.log(`API listening on port ${PORT}`);
});
```

Run the following command and a successful message would be displayed showing that API is listening on port 3000.
`node server.js`

Now in order to test this, postman can be used by passing the ID and details to the public_ip:3000/storeData.

# 2 Setting Up AWS Lambda and DynamoDB Integration

In this section Dynamo DB and Lambda would be created and configured. Here Dynamo DB would only have ID as the primary key.

## 2.1 Create DynamoDB

1. Select **DynamoDB Service** from the list of AWS Services.
2. Name the database as `ScalingActions`.
3. Set the Partition Key to be of type `String` and name it as `ID`.
4. Keep the remaining settings as default and click on **Create Table**.

## 2.2 Create Lambda

1. Select **Lambda** from the list of AWS Services.
2. Name the function `StoreSystem`.
3. Select the Runtime to be `Node.js 20.x`.

4. Set the Architecture to `x86_64`.

5. In **Permissions**, select `Create a new role from AWS policy templates` and name it appropriately.

6. Click on **Create Function**.

7. Once the function is created, upload the zip file in the Artifact folder which contains the node modules required to execute the function.

8. Create a file named index.js if it does not exist and paste the code given.

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const {
  DynamoDBDocumentClient,
  PutCommand,
} = require("@aws-sdk/lib-dynamodb");
const crypto = require('crypto');
const axios = require('axios');

const client = new DynamoDBClient({});
const dynamo = DynamoDBDocumentClient.from(client);

const tableName = "ScalingActions";
const apiUrl = "http://34.245.228.5:3000/storeData"; //replace the
    public_ip with the EC2 instance created earlier for HLF

exports.handler = async (event) => {
  let body;
  let statusCode = 200;
  const headers = {
    "Content-Type": "application/json",
  };
  const eventBody = typeof event.body === 'string' ? JSON.parse(
      event.body) : event;

  // Extract the instance ID and action from the EventBridge event
  const instanceId = eventBody.detail?.EC2InstanceId;
  const action = eventBody["detail-type"];
  if (!instanceId || !action) throw new Error("Missing
      EC2InstanceId or detail-type");

  // Construct the details string
  const details = `Instance ${instanceId} ${action}`;

  // Generate SHA value
  const shaValue = crypto.createHash('sha256').update(`${instanceId
      }${details}`).digest('hex');

  try {
    // Store in DynamoDB
    await dynamo.send(
      new PutCommand({
        TableName: tableName,
        Item: {
          ID: instanceId,
```

```
            Details: details,
            SHA_Value: shaValue
          },
        })
    );

    // Send data to the external API
    await axios.post(apiUrl, {
      id: instanceId,
      shaValue: shaValue
    });

    body = `Data stored in DynamoDB and sent to API successfully
       for Instance ID: ${instanceId}`;
  } catch (error) {
    statusCode = 500;
    body = `Error: ${error.message}`;
  } finally {
    body = JSON.stringify(body);
  }

  return {
    statusCode,
    body,
    headers,
  };
};
```

# 3 Configure EC2 instance for Application

In this section, EC2 instance would be having an application where using the node exporter the metrics are send to lambda using API Gateway. Following that Autoscaling group would be created which would create new instances as the load demands. Eventbridge would be configured to send the autoscaling events to lambda.

## 3.1 Create EC2 Instance

1. Select **EC2** instance from the list of AWS Services available.
2. Name the instance as `Application2024`.
3. Select the AMI as `Amazon Linux 2023`.
4. Set the instance type to `t2.micro`.
5. Select the security group created earlier for **HLF**.
6. Click on **Launch Instance**, keeping the default settings.

## 3.2 Deploy the Application

1. Initialize the project by running the following commands:

```
$ sudo mkdir application_scale
$ cd application_scale
$ npm init -y
```

2. Install the `express` package:

```
$ npm install express
```

3. Create the express API:

```
$ vi server.js
```

```
const express = require('express');
const AWS = require('aws-sdk');

const app = express();
const dynamoDB = new AWS.DynamoDB.DocumentClient({ region: 'eu-west-1' });

// Serve static files
app.use(express.static('public'));

app.get('/records', async (req, res) => {
    const params = {
        TableName: 'ScalingActions',
        Limit: 10,
        ScanIndexForward: false
    };

    try {
        const data = await dynamoDB.scan(params).promise();
        res.json(data.Items);
    } catch (error) {
        console.error(error);
        res.status(500).send('Error fetching records');
    }
});

const PORT = process.env.PORT || 8080;
app.listen(PORT, () => {
    console.log(`Server is running on port ${PORT}`);
});
```

4. Create public directory to host the static file

```
$ mkdir public
$ vi index.html
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Last 10 Records</title>
    <style>
        body { font-family: Arial, sans-serif; }
        table { width: 100%; border-collapse: collapse; margin-top: 20px; }
        th, td { padding: 10px; border: 1px solid #ddd; text-align: left; }
    </style>
</head>
<body>
    <h1>Last 10 Records from DynamoDB</h1>
    <table id="recordsTable">
        <thead>
            <tr>
                <th>ID</th>
                <th>Details</th>
                <th>SHA Value</th>
            </tr>
        </thead>
        <tbody></tbody>
    </table>

    <script>
        async function fetchRecords() {
            const response = await fetch('/records');
            const data = await response.json();
            const tableBody = document.querySelector('#recordsTable tbody');

            data.forEach(record => {
                const row = document.createElement('tr');
                row.innerHTML = `<td>${record.ID}</td><td>${record.Details}</td><td>${record.SHA_Value}</td>`;
                tableBody.appendChild(row);
            });
        }

        // Fetch records when the page loads
        fetchRecords();
    </script>
</body>
</html>
```

## 3.3 Install and Configure Node Exporter

1. Run the following commands to download and extract the Node Exporter installation package:

```
$ cd /tmp
```

```
$ curl -LO https://github.com/prometheus/node_exporter/releases/download/
          v1.5.0/node_exporter-1.5.0.linux-amd64.tar.gz
```

2. Change the path:

```
$ sudo mv node_exporter-1.5.0.linux-amd64/node_exporter /usr/local/bin/
```

3. Create a user for running Node Exporter:

```
$ sudo useradd -rs /bin/false node_exporter
```

4. Create a SystemD service for Node Exporter:

```
$ sudo nano /etc/systemd/system/node_exporter.service
```

   Add the following configuration:

```
[Unit]
Description=Node Exporter
After=network.target

[Service]
User=node_exporter
Group=node_exporter
ExecStart=/usr/local/bin/node_exporter

[Install]
WantedBy=multi-user.target
```

5. Reload the SystemD manager configuration:

```
$ sudo systemctl daemon-reload
```

6. Now, the system metrics will be available at the following URL:

```
http://instance_publicip:9100/metrics
```

7. In order to send the metrics to Lambda, create a python file and extract the metrics from node exporter.

```
$ vi collect_metrics.py
```

```
    import requests
    import uuid
    from datetime import datetime

    # API Gateway URL for Lambda function
    API_GATEWAY_URL = 'https://cao3wzniz2.execute-api.eu-west-1.amazonaws.com/test/scalingaction/'

    # Node Exporter metrics URL. Place the EC2 instance public_ip
    NODE_EXPORTER_URL = 'http://54.74.9.79:9100/metrics'

    def collect_and_send_metrics():
        # Fetch metrics from Node Exporter
        response = requests.get(NODE_EXPORTER_URL)
        metrics = response.text

        # Initialize variables to calculate CPU utilization
        idle_cpu_seconds = 0
        total_cpu_seconds = 0

        # Parse CPU utilization from metrics
        for line in metrics.splitlines():
            if line.startswith('node_cpu_seconds_total'):
                parts = line.split()
                # Parse mode and CPU seconds
                mode = parts[0].split('{')[1].split('}')[0]  # Extract mode (e.g., idle, user)
                cpu_seconds = float(parts[1])  # Keep as float for calculations

                # Accumulate total CPU seconds for all modes
                total_cpu_seconds += cpu_seconds

                if 'mode="idle"' in mode:
                    idle_cpu_seconds += cpu_seconds

        # Calculate CPU utilization percentage
        if total_cpu_seconds > 0:
            cpu_utilization_percent = (1 - (idle_cpu_seconds / total_cpu_seconds)) * 100
        else:
            cpu_utilization_percent = 0

        timestamp = datetime.utcnow().isoformat()
        unique_id = str(uuid.uuid4())  # Creates a random unique ID

        # Prepare data in the format Lambda expects
        data = {
            "detail": {
                "EC2InstanceId": unique_id
            },
            "detail-type": f"CPU utilization recorded at {timestamp}: {cpu_utilization_percent:.2f}%"
        }

        # Send data to Lambda through API Gateway
        try:
```

```
            response = requests.post(API_GATEWAY_URL, json=data)
            response.raise_for_status()
            print(f'Successfully sent data to Lambda for ID {
                unique_id}: {response.text}')
        except requests.exceptions.RequestException as e:
            print(f'Error sending data to Lambda: {e}')

if __name__ == '__main__':
    collect_and_send_metrics()
```

8. Run the script using the following command. Install Python3 if not installed.

```
$ python3 collect_metrics.py
```

## 3.4 Create Autoscaling Group

1. Select **Autoscaling** from AWS Services → Instances.
2. Name the autoscaling group as `Application_traffic`.
3. Create a Launch Template with the same configuration performed earlier. Ensure to select the AMI from the **Recently launched** section.
4. Set the instance requirement to `t2.micro`.
5. Proceed with the default VPC and Subnets.
6. For testing purposes, configure the scaling to have a maximum of 2 instances.
7. Follow the remaining default configuration and create the scaling group.

## 3.5 Create EventBridge to Send Events to Lambda

1. Select **Rules** from AWS Services → **EventBridge**.
2. Select the event bus as `default`.
3. Set the **Rule type** to `Rule with an event pattern`.
4. Choose the event source as `AWS events or EventBridge partner events`.
5. Under the creation method, select the **Use pattern from** option.
6. Set the event pattern to track `EC2 instance terminated` and `running` states.
7. In the target section, choose the Lambda function that was created earlier.
8. Create the EventBridge rule.

This configuration will track the scaling events occurring in the auto-scaling group.

### 3.6 Attach SQS

As mentioned in the report, to handle the limitation of lambda on more than 10 transactions per second (*Lambda quotas - AWS Lambda* (n.d.)), AWS SQS is used. Here SQS would take the requests from the EC2 instance hosting the application. In this section setting up the SQS into the architecture would be focused on.

1. In the details type, select `Standard Queue` and proceed by giving a name to the queue.

2. Set the remaining values to be default for the time being and click on `Create Queue`.

3. Once the queue is ready, add a trigger in Lambda to accept from the queue and at the same time change the script mentioned above to send metrics to SQS instead of Lambda (*Using Lambda with Amazon SQS - AWS Lambda* (n.d.)).

## 4 Measure the Performance

In this section, the performance would be analyzed using two tools. The first one is Hyperledger Caliper, which would be installed on the instance where HLF is set up. The other load testing tool is Artillery.

### 4.1 Install and Configure Hyperledger Caliper

1. Navigate to the `fabric-samples` directory and create a folder named `caliper`.

```
cd ~/fabric-samples
mkdir caliper
cd caliper
```

2. Initialize a Node.js project.

```
npm init -y
```

3. Install Caliper CLI by running the following command:

```
npm install --only=prod @hyperledger/caliper-cli@0.4.2
```

4. Bind Caliper to Fabric after installation:

```
npx caliper bind --caliper-bind-sut fabric:2.2
```

5. Create the necessary directories for configuration:

```
mkdir networks benchmarks workload
```

6. Create the network configuration file:

```
vi networks/networkConfig.yaml
```

Add the following content:

```
name: Caliper test network
version: "2.0.0"

caliper:
  blockchain: fabric
  sutOptions:
    mutualTls: true

channels:
  - channelName: mychannel
    contracts:
      - id: assettransfer

organizations:
  - mspid: Org1MSP
    identities:
      certificates:
        - name: "Admin"
          clientPrivateKey:
            path: /home/ubuntu/fabric-samples/test-network/
                organizations/peerOrganizations/org1.example.com
                /users/Admin@org1.example.com/msp/keystore/
                priv_sk
          clientSignedCert:
            path: /home/ubuntu/fabric-samples/test-network/
                organizations/peerOrganizations/org1.example.com
                /users/Admin@org1.example.com/msp/signcerts/
                Admin@org1.example.com-cert.pem
      connectionProfile:
        path: /home/ubuntu/fabric-samples/test-network/
            organizations/peerOrganizations/org1.example.com/
            connection-org1.yaml
        discover: true

  - mspid: Org2MSP
    identities:
      certificates:
        - name: "Admin"
          clientPrivateKey:
            path: /home/ubuntu/fabric-samples/test-network/
                organizations/peerOrganizations/org2.example.com
                /users/Admin@org2.example.com/msp/keystore/
                priv_sk
          clientSignedCert:
            path: /home/ubuntu/fabric-samples/test-network/
                organizations/peerOrganizations/org2.example.com
                /users/Admin@org2.example.com/msp/signcerts/
                Admin@org2.example.com-cert.pem
```

```
          connectionProfile:
            path: /home/ubuntu/fabric-samples/test-network/
                organizations/peerOrganizations/org2.example.com/
                connection-org2.yaml
            discover: true
```

7. Create the benchmarking file:

```
vi benchmarks/assetBenchmark.yaml
```

Add the following content:

```
test:
  name: Create Asset Benchmark Test
  description: Performance test for CreateAsset function with
      ID and SHA_Value.
  workers:
    type: local
    number: 2

  rounds:
    - label: createAssetTest
      description: Testing CreateAsset with ID and SHA_Value
      txNumber: 1000
      rateControl:
        type: fixed-rate
        opts:
          tps: 100
      workload:
        module: workload/createAsset.js
        arguments:
          contractId: assettransfer
          contractFunction: CreateAsset
          assetPrefix: asset
          shaPrefix: shaValue
```

8. Define the workloads:

```
vi workload/createAsset.js
```

Add the following content:

```
'use strict';

const { WorkloadModuleBase } = require('@hyperledger/caliper-
    core');

class CreateAssetWorkload extends WorkloadModuleBase {
    constructor() {
        super();
        this.txIndex = 0;
    }

```

```
        async initializeWorkloadModule(workerIndex, totalWorkers,
            roundIndex, roundArguments, sutAdapter, sutContext) {
            await super.initializeWorkloadModule(workerIndex,
                totalWorkers, roundIndex, roundArguments, sutAdapter
                , sutContext);
        }

        async submitTransaction() {
            this.txIndex++;

            const assetID = `asset_${this.workerIndex}_${this.
                txIndex}`;
            const shaValue = `sha_${Date.now()}`;

            const request = {
                contractId: this.roundArguments.contractId,
                contractFunction: 'CreateAsset',
                contractArguments: [assetID, shaValue],
                readOnly: false
            };

            await this.sutAdapter.sendRequests(request);
        }
    }

    function createWorkloadModule() {
        return new CreateAssetWorkload();
    }

    module.exports.createWorkloadModule = createWorkloadModule;
```

9. Run the benchmarking test. Ensure the directory is in `caliper`:

```
npx caliper launch manager \
    --caliper-workspace . \
    --caliper-networkconfig networks/networkConfig.yaml \
    --caliper-benchconfig benchmarks/assetBenchmark.yaml \
    --caliper-flow-only-test \
    --caliper-fabric-gateway-enabled
```

## 4.2 Install and Configure Artillery

Artillery would be used to analyse the overall performance of the system

1. Install Artillery on the EC2 instance where the application is installed.

```
sudo npm install -g artillery
```

2. Create a file called loadtest.yaml and insert the following code.

```
vi loadtest.yaml
```

```
config:
  target: "http://ec2_public_ip:3000" # Replace with your endpoint IP
  phases:
    - duration: 10        # Test duration in seconds
      arrivalRate: 6       # Requests per second (adjust as needed)
  defaults:
    headers:
      Content-Type: "application/json"

scenarios:
  - flow:
      - post:
          url: "/storeData"
          json:
            id: "unique_{{ $uuid }}"     # Generates a unique string for each request
            shaValue: "sha_{{ $randomString }}" # Generates a random string for each request
```

3. Run the testing by following command.

```
artillery run loadtest.yaml
```

Collect the results by changing in the configurations in order to come into a conclusion.

# References


*Install Node.js on Amazon Linux 2 :: INTRODUCTION TO AMAZON EC2.* (n.d.).
**URL:** *https://000004.awsstudygroup.com/6-awsfcjmanagement-linux/6.2-setupnodejsonec2linux/*

*Lambda quotas - AWS Lambda* (n.d.).
**URL:** *https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html*

Makers, D. W. (2014). TechSavvy Coders in Virginia.
**URL:** *https://myhsts.org/tutorial-learn-how-to-install-blockchain-hyperledger-fabric-on-amazon-web-services.php*

*Using Lambda with Amazon SQS - AWS Lambda* (n.d.).
**URL:** *https://docs.aws.amazon.com/lambda/latest/dg/with-sqs.html*