Ethereum Deposit Tracker Documentation

VIT SDE Hiring task Luganodes

Completed by: S. Swaraj

TABLE OF CONTENTS

INTRODUCTION

SYSTEM ARCHITECTURE

COMPONENTS

MAIN SERVICE

RABBITMQ

INFLUXDB WORKER

TELEGRAM NOTIFICATION WORKER

INFLUXDB

GRAFANA

SETUP AND INSTALLATION

1. INTRODUCTION

The Ethereum Deposit Tracker is an advanced monitoring system designed to efficiently track and record Ethereum (ETH) deposits on the Beacon Deposit Contract. This project, undertaken as part of the VIT SDE Hiring task, provides a robust and scalable solution for real-time deposit data management from the Ethereum blockchain.
The system is built to interface with Ethereum's blockchain through Alchemy's RPC methods, establishing a seamless connection for retrieving deposit information. It actively monitors the Beacon Deposit Contract address **0x00000000219ab540356cBB839Cbe05303d7705Fa** for incoming deposits, capturing crucial details such as deposit amount, sender address, public key, and transaction hash.

Key features of the Ethereum Deposit Tracker include:

- Real-Time Data Fetching: Utilizes Alchemy to efficiently retrieve and process deposit information from the blockchain, ensuring timely updates.
- Comprehensive Recording: Logs detailed information about each deposit, including block number, timestamp, and transaction fee. The data is stored in InfluxDB for efficient querying and long-term storage.
- Error Handling and Logging: Implements robust error handling and logging mechanisms using Winston, facilitating transparent and reliable system operations.
- Optional Alerting and Visualization: Integrated Grafana for visualizing deposit data and system metrics, and integrated Telegram notifications to provide real-time alerts for new deposits.

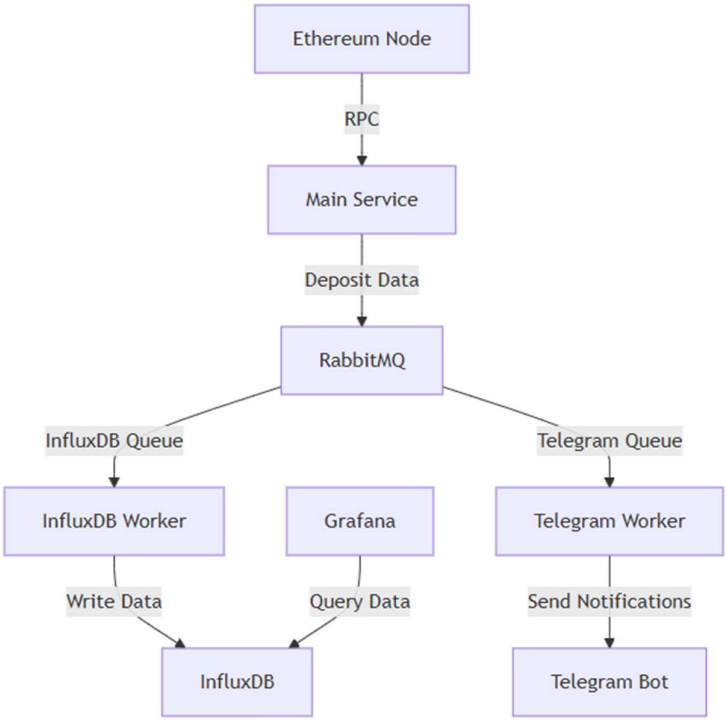
Additionally, the entire system has been containerized using Docker to ensure consistent deployment across different environments. Adjustments were made to the RabbitMQ URL configuration to align with the Dockerized setup, ensuring smooth communication between services.

This report details the architecture, implementation, and functionality of the Ethereum Deposit Tracker, offering a comprehensive overview of how the project meets the specified requirements and delivers an effective solution for monitoring Ethereum deposits.

2. System Architecture

The Ethereum Deposit Tracker employs a microservices architecture to ensure scalability, resilience, and efficient data processing.

Here's an overview of the system architecture:



Key components of the architecture:

- **Main Service:** Interacts with the Ethereum blockchain and processes deposit transactions.
- **RabbitMQ:** Message queue for distributing deposit data to workers.
- **InfluxDB Worker**: Processes deposit data and stores it in InfluxDB.
- **Telegram Worker:** Sends notifications about new deposits via Telegram.
- **InfluxDB:** Time-series database for storing deposit data.
- **Grafana:** Visualization platform for creating dashboards and monitoring the system.

3. Components

3.1. Main Service (index.js)

The main service of the Ethereum Deposit Tracker is responsible for monitoring the Ethereum blockchain for deposit transactions, extracting key data, and relaying this information to downstream systems for further processing. This service is built using Node.js and utilizes the Alchemy API to connect to the Ethereum network and RabbitMQ for message queuing.

Component Responsibilities & Features:

3.1.1 Connecting to the Ethereum Network using Alchemy API:

The service establishes a connection with the Ethereum network via Alchemy, utilizing ethers.js for interaction. The provider is configured to use the Alchemy API key from the environment variables.

```
// Connect to Ethereum using Alchemy
const provider = new ethers.providers.AlchemyProvider("homestead", process.env.ALCHEMY_API_KEY);
```

3.1.2 Monitoring New Blocks for Deposit Transactions:

The service listens for new blocks being mined in real-time using the provider.on("block") event listener.

It processes each transaction within the block and checks if it is directed towards the Beacon Deposit Contract.

```
provider.on("block", async (blockNumber) => {
  try {
    logger.info(`New block detected: ${blockNumber}`);

    // Get block data with transactions
    const block = await provider.getBlockWithTransactions(blockNumber);
    const blockTimestamp = block.timestamp; // Get the block timestamp
    block.transactions.forEach(async (tx) => {
      if (!processedTransactions.has(tx.hash)) {
        // Check if it's a deposit to the Beacon Deposit Contract
        if (tx.to && tx.to.toLowerCase() === beaconContractAddress) {
          logger.info("Deposit transaction detected", { txHash: tx.hash });
          const pubkey = extractPublicKey(tx);
          sendToQueues({ ...tx, blockNumber ,blockTimestamp,pubkey});
          processedTransactions.add(tx.hash);
        }
```

In the initial implementation, the service aimed to handle internal transactions (those triggered within a contract) using the trace_transaction Ethereum RPC method. This would have allowed the service to detect ETH deposits that are not part of the main transaction but occur as internal transfers within the Beacon Deposit Contract.

However, due to limitations in the current Alchemy plan, the trace_transaction functionality was not available, leading to the removal of this feature in the final implementation. The disabled code is still documented in the service for future reference and potential re-implementation if the RPC plan is upgraded.

```
//Removed the trace_transaction functionality, as it's not available on my current Alchemy plan.

// Check internal transactions for deposits to the contract
// const receipt = await provider.getTransactionReceipt(tx.hash);
// if (receipt) {
//   const traces = await provider.send("trace_transaction", [tx.hash]);
//   traces.forEach((trace) => {
//     if (trace.action.to && trace.action.to.toLowerCase() === beaconContractAddress) {
//       logger.info("Internal ETH deposit detected in trace", { txHash: trace.transactionHash });
//       const pubkey = extractPublicKey(tx);
//       sendToQueues({ ...trace, blockNumber,blockTimestamp ,pubkey});
//       processedTransactions.add(tx.hash);
//     }
//   });
// }
```

3.1.3 Extracting Relevant Data from Transactions:

Each detected deposit transaction is processed to extract necessary details such as:

- Transaction hash
- Public key (from the transaction's signature)
- Block number and timestamp

The extractPublicKey() function recovers the public key from the transaction signature In the development of the Ethereum Deposit Tracker, I implemented a function to extract the public key from a transaction using the ethers.js library. This function leverages the ethers.utils methods for serializing the transaction, computing its hash, and recovering the public key from the transaction's signature. This approach follows the guidelines provided in the official ethers.js documentation, ensuring accurate and efficient extraction of public key data.

```
// Function to extract public key from transaction
function extractPublicKey(tx) {
  try {
    const expandedSig = {
      r: tx.r,
      s: tx.s,
      v: tx.v
    };

    // Reconstruct the transaction data
    const txData = {
      chainId: tx.chainId,
      nonce: tx.nonce,
      gasPrice: tx.gasPrice,
      gasLimit: tx.gasLimit,
      to: tx.to,
      value: tx.value,
      data: tx.data
    };

    // Serialize the transaction
    const serializedTx = ethers.utils.serializeTransaction(txData);

    // Get the transaction hash
    const txHash = ethers.utils.keccak256(serializedTx);

    // Recover the public key
    const publicKey = ethers.utils.recoverPublicKey(txHash, expandedSig);

    return publicKey;
  } catch (error) {
    logger.error('Error extracting public key', { error, txHash: tx.hash });
    return null;
  }
}
```

3.1.4 Sending Deposit Data to RabbitMQ Queues:

Once the relevant data is extracted, the service sends the transaction details to two separate RabbitMQ queues:

- One queue for saving the data into InfluxDB.
- Another queue for sending Telegram notifications.

```
// Send transaction data to two queues: InfluxDB and Telegram Notification
function sendToQueues(txData) {
  rabbitConn.createChannel((error, channel) => {
    if (error) {
      logger.error('Error creating RabbitMQ channel', { error });
      return;
    }

    const msg = JSON.stringify(txData);

    // Send to InfluxDB queue
    channel.assertQueue(QUEUE_INFLUXDB, { durable: true });
    channel.sendToQueue(QUEUE_INFLUXDB, Buffer.from(msg), { persistent: true });
    logger.info(`Transaction sent to InfluxDB queue: ${txData.hash}`);

    // Send to Telegram queue
    channel.assertQueue(QUEUE_TELEGRAM, { durable: true });
    channel.sendToQueue(QUEUE_TELEGRAM, Buffer.from(msg), { persistent: true });
    logger.info(`Transaction sent to Telegram queue: ${txData.hash}`);
  });
}
```

3.1.5 Error Handling and Logging:

Comprehensive error handling and logging via winston ensure that errors during the block monitoring, transaction processing, or RabbitMQ messaging do not go unnoticed. Logs will be saved in logs/app.logs .

```
// Setup Winston Logger
const logger = winston.createLogger({
  level: 'info',
  format: winston.format.combine(
    winston.format.timestamp(),
    winston.format.json()
  ),
  transports: [
    new winston.transports.Console(),
    new winston.transports.File({ filename: 'logs/app.log' })
  ]
});
```

3.2 RabbitMQ

RabbitMQ serves as the message broker, facilitating communication between the main service and workers. It ensures:

- Reliable message delivery
- Scalable message distribution
- Decoupling of services for better fault tolerance

3.3 InfluxDB Worker (influxdb_worker.js)

This component is responsible for consuming messages from RabbitMQ, processing Ethereum deposit transactions, and saving them into an InfluxDB database for further analysis and visualization.

Component Responsibilities & Features:

3.3.1 Winston Logging Integration:

A robust logging system is set up using **Winston**, which logs all important events, including successful deposit storage and any errors encountered during processing. Logs are saved in both the console and a file (influxdb_worker.log).

```
new winston.transports.File({ filename: 'logs/influxdb_worker.log' })
```

3.3.2 InfluxDB Configuration and Point Writing:

The service connects to InfluxDB using credentials from environment variables, which includes the database URL, organization, and bucket.

It prepares and writes each transaction to InfluxDB by creating a **Point** object that contains relevant transaction data (such as transaction hash, sender address, deposit value, gas price, fee, and block number).

```
// InfluxDB configurations
const influxDB = new InfluxDB({
  url: process.env.INFLUXDB_URL,
  token: process.env.INFLUXDB_TOKEN
});
const writeApi = influxDB.getWriteApi(process.env.INFLUXDB_ORG, process.env.INFLUXDB_BUCKET);
writeApi.useDefaultTags({ app: 'eth-deposit-tracker' });
```

3.3.3 RabbitMQ Consumer Setup:

The worker connects to a RabbitMQ server and listens to the **InfluxDB queue** .When a message is received, it processes the transaction and acknowledges the message.

```
// RabbitMQ consumer setup
amqp.connect(RABBITMQ_URL, function (error, connection) {
  if (error) {
    logger.error('Failed to connect to RabbitMQ', { error });
    throw error;
  }
  connection.createChannel(function (error, channel) {
    if (error) {
      logger.error('Error creating RabbitMQ channel', { error });
      return;
    }
    channel.assertQueue(QUEUE_INFLUXDB, { durable: true });
    channel.consume(QUEUE_INFLUXDB, (msg) => {
      processTransaction(msg);
      channel.ack(msg);
    });
    logger.info('InfluxDB worker is waiting for messages');
  });
});
```

3.3.4 Ethereum Transaction Processing:

The deposit transaction data sent by the main service (via RabbitMQ) is parsed and processed. Key fields such as value and gasPrice are converted back to their appropriate formats using ethers.js utilities.

The **fee** is calculated by multiplying the value and gasPrice, and the transaction data is saved as a data point into InfluxDB.

```
const tx = JSON.parse(msg.content.toString());
const { hash, from, value, gasPrice, pubkey, blockNumber, blockTimestamp} = tx;

// Convert value and gasPrice back to BigNumber
const valueBN = ethers.BigNumber.from(value);
const gasPriceBN = ethers.BigNumber.from(gasPrice);

const formattedValue = ethers.utils.formatEther(valueBN);
const formattedGasPrice = ethers.utils.formatUnits(gasPriceBN, 'gwei');

// Calculate the fee
const fee = valueBN.mul(gasPriceBN).div(ethers.utils.parseUnits('1', 'gwei'));
const formattedFee = ethers.utils.formatEther(fee);
```

A **Point** object is created with tags (**transaction hash, sender address, public key**) and fields (**value, gas price, fee, block number, and block timestamp)** before being written to the database.

```
const point = new Point('deposits')
  .tag('tx_hash', hash)
  .tag('from_address', from)
  .tag('pubkey', pubkey || 'unknown')
  .floatField('value_eth', parseFloat(formattedValue))
  .floatField('gas_price_gwei', parseFloat(formattedGasPrice))
  .floatField('fee_eth', parseFloat(formattedFee))
  .floatField('block_number', blockNumber)
  .timestamp(blockTimestamp ? new Date(blockTimestamp * 1000) : new Date());

// Write the data point to InfluxDB
try {
  writeApi.writePoint(point);
  logger.info(`Saved deposit transaction to InfluxDB: ${hash}`);
} catch (error) {
  logger.error('Error writing transaction to InfluxDB', { error });
}
```

3.4 Telegram Notification Worker Service (telegram_worker.js)

This component is responsible for consuming Ethereum deposit transaction messages from RabbitMQ and sending notifications via Telegram to a predefined chat or group.

Component Responsibilities & Features:

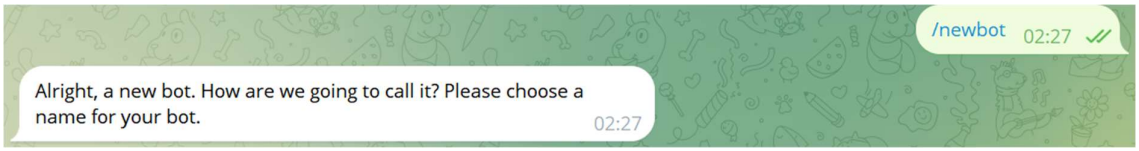
3.4.1 Winston Logging Integration:

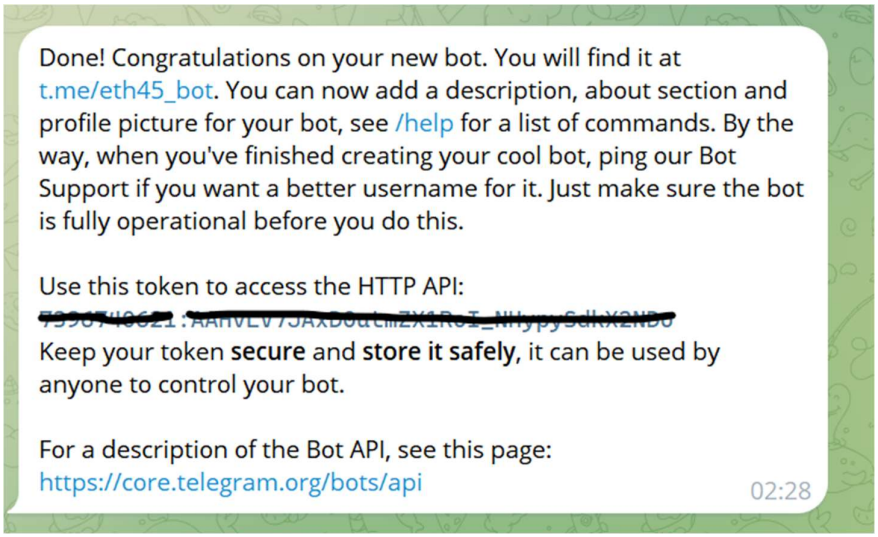
The service uses **Winston** for structured logging, with logs written both to the console and to a file (telegram_worker.log), ensuring that all significant events (like message sending or errors) are logged and available for troubleshooting or auditing.

```
transports: [
  new winston.transports.Console(),
  new winston.transports.File({ filename: 'logs/telegram_worker.log' })
]
```

3.4.2 Telegram Bot API Integration:

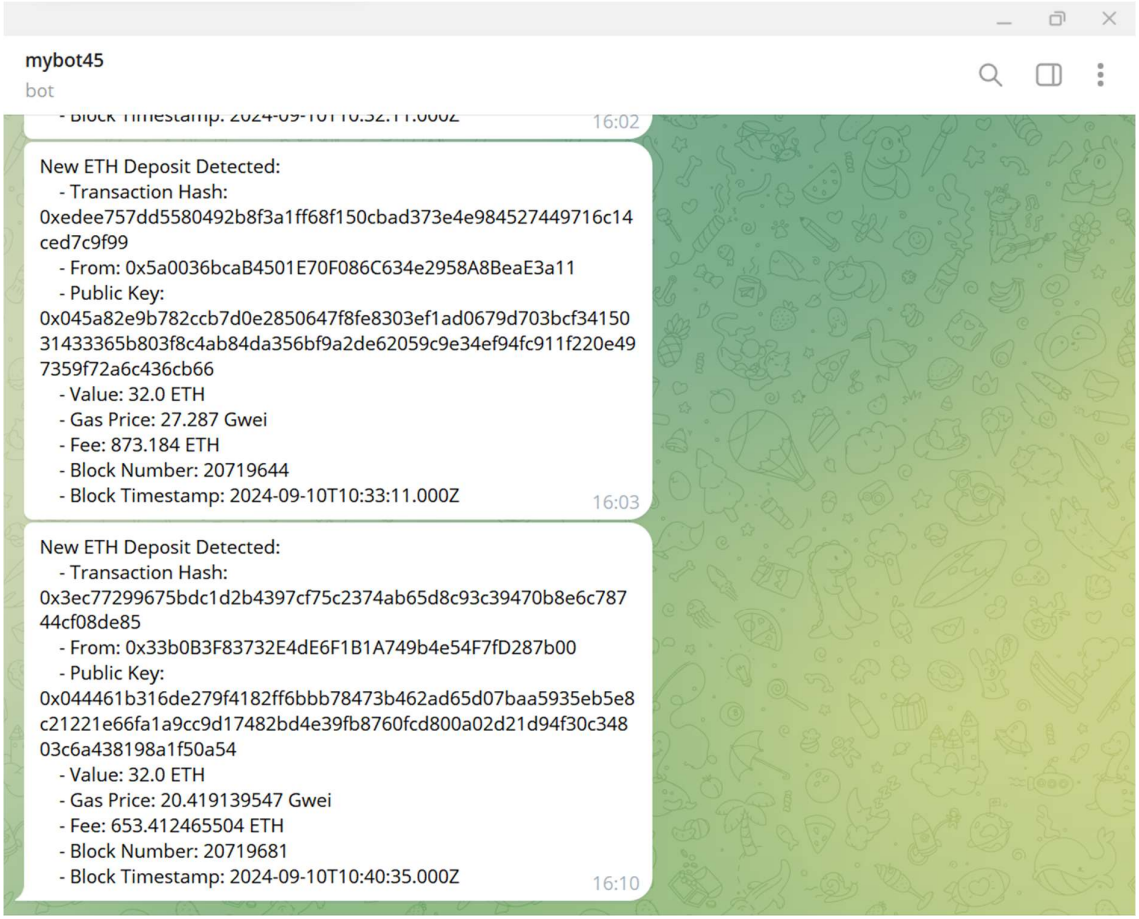
A **Telegram bot** is initialized using the **node-telegram-bot-api** package, with the bot token and chat ID obtained from environment variables. This bot is responsible for sending notifications whenever a new deposit is detected. You can use BotFather to Get API key






```
const bot = new TelegramBot(process.env.TELEGRAM_BOT_TOKEN);
const chatId = process.env.TELEGRAM_CHAT_ID;
```

Alert Screenshots:



3.4.3 RabbitMQ Consumer Setup:

The service establishes a connection to the RabbitMQ server and listens for messages in the **Telegram queue** (telegram_queue). Whenever a new message is received, it processes the transaction and sends a corresponding Telegram notification.

```
// RabbitMQ consumer setup
amqp.connect(RABBITMQ_URL, function (error, connection) {
  if (error) {
    logger.error('Failed to connect to RabbitMQ', { error });
    throw error;
  }
  connection.createChannel(function (error, channel) {
    if (error) {
      logger.error('Error creating RabbitMQ channel', { error });
      return;
    }

    // Assert the Telegram queue
    channel.assertQueue(QUEUE_TELEGRAM, { durable: true });

    // Consume messages from the Telegram queue
    channel.consume(QUEUE_TELEGRAM, (msg) => {
      processTransaction(msg);
      channel.ack(msg);
    });

    logger.info('Telegram worker is waiting for messages');
  });
});
```

3.4.4 Ethereum Transaction Processing and Notification:

Similar to the InfluxDb woker When a message is received from RabbitMQ, the deposit transaction is parsed. Important details such as the **transaction hash**, **sender address**, **public key**, **value in ETH**, **gas price**, **transaction fee**, **block number**, and **block timestamp** are extracted and formatted for the notification message.

The formatted message is then sent as a notification to the predefined Telegram chat using the bot.

```
const message = `New ETH Deposit Detected:
  - Transaction Hash: ${hash}
  - From: ${from}
  - Public Key: ${pubkey || 'Unknown'}
  - Value: ${formattedValue} ETH
  - Gas Price: ${formattedGasPrice} Gwei
  - Fee: ${formattedFee} ETH
  - Block Number: ${blockNumber}
  - Block Timestamp: ${formattedTimestamp}`;

bot.sendMessage(chatId, message)
  .then(() => {
    logger.info(`Telegram notification sent for transaction ${hash}`);
  })
  .catch((error) => {
    logger.error('Error sending Telegram message', { error });
  });
```

3.5 InfluxDB

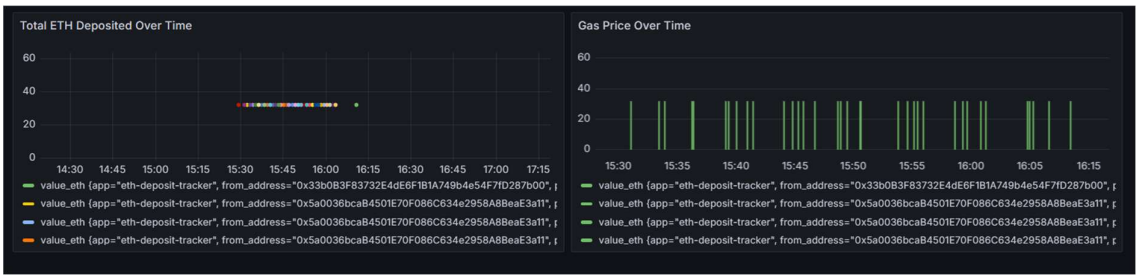
InfluxDB is utilized in this project as a time-series database to store Ethereum deposit transaction data, providing a scalable and efficient solution for capturing and querying the real-time data associated with each transaction. The data stored includes transaction hash, sender address, value in ETH, gas price in Gwei, public key, and block details. InfluxDB's Data Explorer offers a powerful interface for querying and analyzing the stored deposit data, enabling a granular view of individual transactions and historical deposit trends.

3.6 Grafana

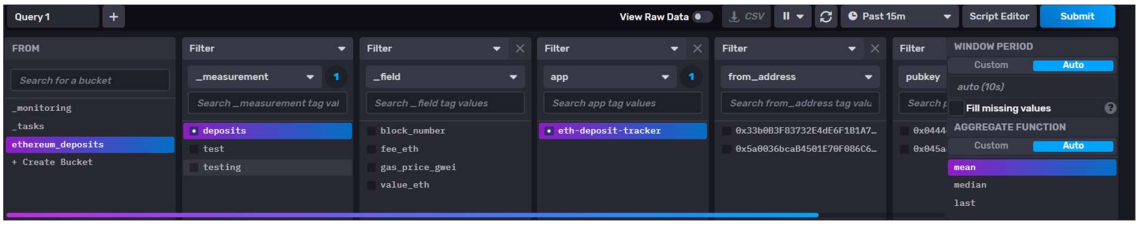
Grafana is used to visualize the data stored in InfluxDB by setting up customizable dashboards. These dashboards provide a visual representation of deposit activity, showcasing metrics such as transaction frequency, ETH values, gas prices, and other key deposit details. By integrating Grafana with InfluxDB, the dashboard enables real-time monitoring of Ethereum deposits on the Beacon Contract, making it easier to track trends, monitor key metrics, and set up alerts.

Screenshots of the Grafana dashboard and InfluxDB Data Explorer can be added to provide a visual reference for the project's monitoring capabilities.

Grafana Dashboard



InfluxDb Data Explorer



4 Setup and Installation

1.Clone the repository:

git clone "https://github.com/Swaraj4507/eth-deposit-tracker"

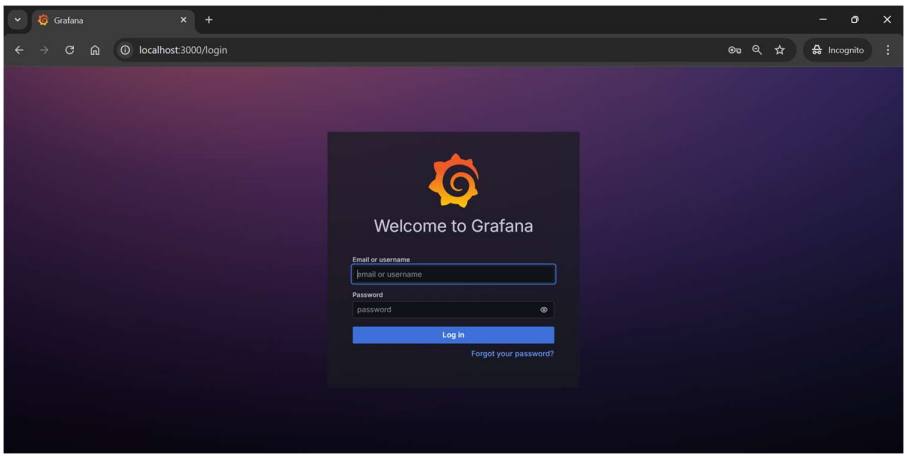
2.Run Docker Compose for Grafana and InfluxDB:

Note: You may need to update the .env configuration files for Grafana and InfluxDB if necessary. Ensure that these settings are also updated in the backend .env file. Follow the existing configuration format for consistency.

Navigate to the **grafana_influxdb folder** in your project directory and run the following command:

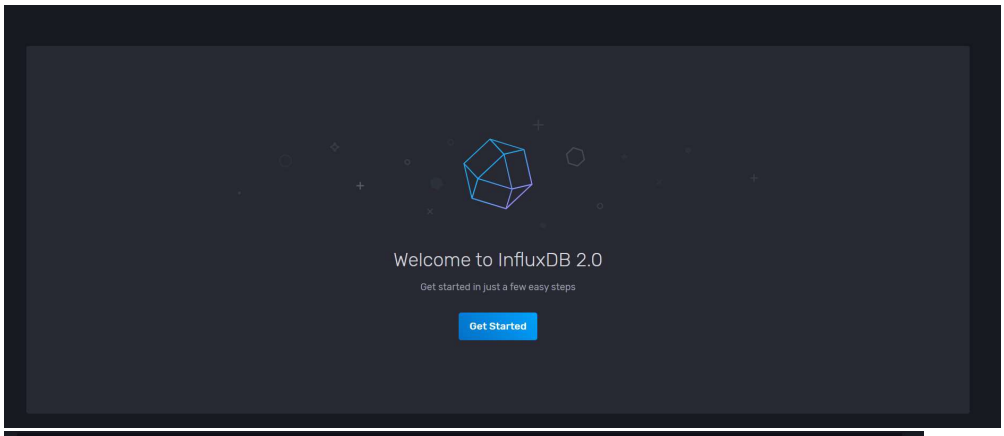
docker-compose up -d

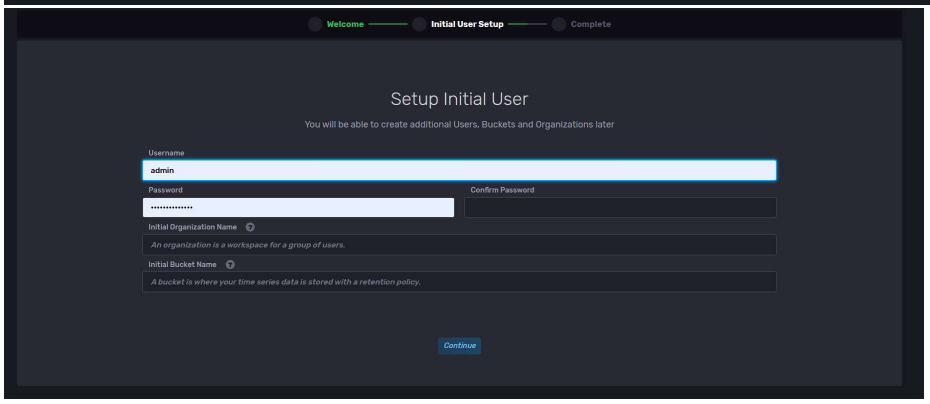
Now you should be able to see Grafana UI at http://localhost:3000/



Login with username:admin
Password: password in Grafana.env file

Influxdb UI at http://localhost:8086/





Fill them by looking at influxdb.env in grafana_influxdb/ folder

(Skip these if you can access both services)

If you are unable to see them

To check whether the containers are running, use the command:
docker ps

```
hp@SwarajPC MINGW64 ~
$ docker ps
CONTAINER ID   IMAGE                  COMMAND                  CREATED       STATUS       PORTS
                                  NAMES
b0341111546f   rabbitmq:management   "docker-entrypoint.s…"   6 hours ago   Up 2 hours   4369/tcp, 5671/tcp, 0.0.0.0:5672->5672/tcp, 15671/tcp, 15691-15692/tcp, 25672/tcp,
0.0.0.0:15672->15672/tcp   rabbitmq
a360b93df6b5   influxdb:2.0           "/entrypoint.sh infl…"   8 hours ago   Up 2 hours   0.0.0.0:8086->8086/tcp
                                  influxdb
34b64dbeae41   grafana/grafana        "/run.sh"                8 hours ago   Up 2 hours   0.0.0.0:3000->3000/tcp
                                  grafana
```

If you see nothing, create the network influxdb_nk with the following command:
docker network create influxdb_nt

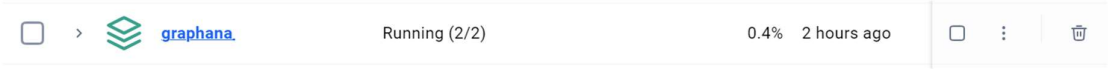
Then Inspect the Network:
docker network inspect influxdb_nt

This command will output JSON data that includes details about the network, including the containers connected to it. Look for the Containers section in the output.

You should see


```
"Containers": {
    "34b64dbeae41613bf6dece134e9d6300cb08dfa2fa3e95474ea696ed59a38e75": {
        "Name": "grafana",
        "EndpointID": "d8831d04fcf7b9d808179b51429e6fe0ed2d1b0d48e902e2de6c640522e295c3",
        "MacAddress": "02:42:ac:14:00:03",
        "IPv4Address": "172.20.0.3/16",
        "IPv6Address": ""
    },
    "a360b93df6b5eb8fc7f7ff43ffdf8051baba1e5399d3ca73bb754af98b45dd44": {
        "Name": "influxdb",
        "EndpointID": "11e836b01223947dc280fd630f9a35215c292dffeaa3a7295bc5daf6bae12423",
        "MacAddress": "02:42:ac:14:00:02",
        "IPv4Address": "172.20.0.2/16",
        "IPv6Address": ""
    }
```


You can also use Docker Desktop to check

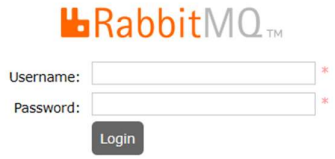


Make sure you enabled virtualisation in your machine to run docker engine.

3.Run Docker Compose for RabbitMQ:

Navigate to the rabbitmq/ folder in your project directory and run the following command:
docker-compose up -d

Now you should be able to see Grafana UI at http://localhost:15672/



Login with username:guest , password:guest.
Now navigate to /backend folder

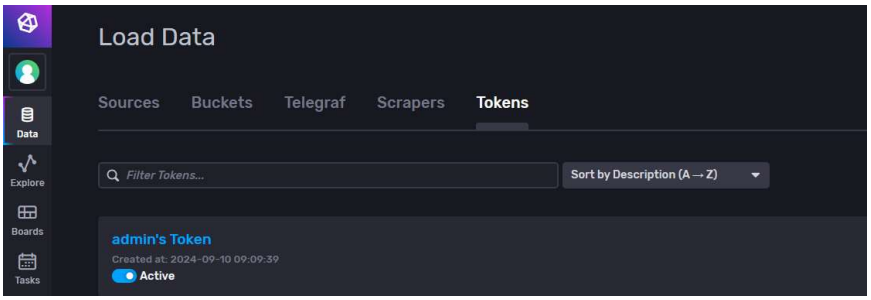
Install dependencies:

npm install

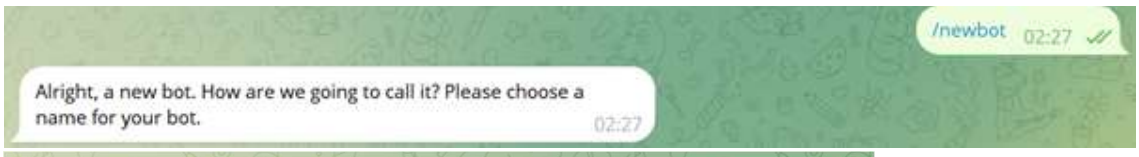
Set up environment variables: Create a .env file in the root directory and add the following:

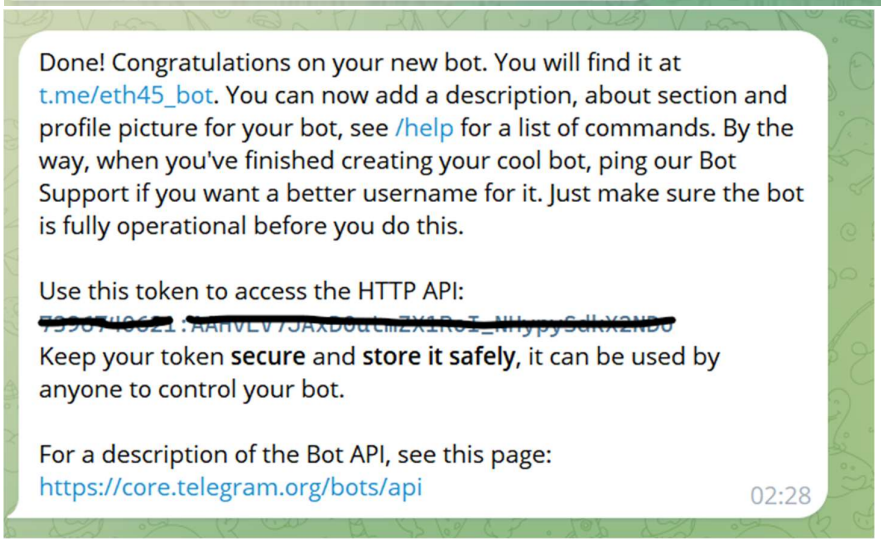
```
ALCHEMY_API_KEY=your_alchemy_api_key
BEACON_CONTRACT_ADDRESS=0x00000000219ab540356cBB839Cbe05303d7705Fa
RABBITMQ_URL=amqp://localhost
INFLUXDB_URL=http://localhost:8086
INFLUXDB_TOKEN=your_influxdb_token
INFLUXDB_ORG=your_org
INFLUXDB_BUCKET=your_bucket
TELEGRAM_BOT_TOKEN=your_telegram_bot_token
TELEGRAM_CHAT_ID=your_telegram_chat_id
```

Alchemy API key from Alchemy site.
Influxdb_Token will be found at http://localhost:8086/
Navigate to Data -> Token Copy Token of Admin user .



Get Telegram API key from telegram BotFather





To get chat id

Go to telegram_worker.js

Uncomment this code and run inside /backend

node telegram_worker.js

```
//Use this script to know your chatid
//Just send  messages to your bot for few times you will see botid

// const bot = new TelegramBot(process.env.TELEGRAM_BOT_TOKEN, { polling: true });

// // When any message is received, log the chatId
// bot.on('message', (msg) => {
//   const chatId = msg.chat.id;

//   // Send a confirmation message with the chatId
//   bot.sendMessage(chatId, `Your chat ID is: ${chatId}`);

//   console.log(`Chat ID: ${chatId}`);

//   // Optionally, you can stop polling after getting the chatId
//   bot.stopPolling();
// });
```

After you get your chatid you can comment this .

Update this ChatId in .env .

Remaining fields

INFLUXDB_ORG=ethereum_org
INFLUXDB_BUCKET=ethereum_deposits

Final .env in /backend folder should look like this

```
ALCHEMY_API_KEY=[redacted]
BEACON_CONTRACT_ADDRESS=0x00000000219ab540356cBB839Cbe05303d7705Fa
INFLUXDB_URL=http://localhost:8086
INFLUXDB_TOKEN=[redacted]
INFLUXDB_ORG=ethereum_org
INFLUXDB_BUCKET=ethereum_deposits
TELEGRAM_BOT_TOKEN=[redacted]
TELEGRAM_CHAT_ID=[redacted]
```

Now you are ready to use application

Open 3 terminals in /backend and run

Terminal 1:
node index.js
Terminal 2:
node influxdb_worker.js
Terminal 3:
node telegram_worker.js

Expected Outputs:

Terminal-1:

```
PS D:\eth-deposit-tracker\backend> node index.js
{"level":"info","message":"Server is running on port 3030","timestamp":"2024-09-10T12:57:18.692Z"}
{"level":"info","message":"Connected to RabbitMQ","timestamp":"2024-09-10T12:57:18.798Z"}
{"level":"info","message":"Latest block number: 20720358","timestamp":"2024-09-10T12:57:25.495Z"}
{"level":"info","message":"New block detected: 20720358","timestamp":"2024-09-10T12:57:25.498Z"}
```

```
PS D:\eth-deposit-tracker\backend> node index.js
{"level":"info","message":"Server is running on port 3030","timestamp":"2024-09-10T08:40:31.075Z"}
{"level":"info","message":"Connected to RabbitMQ","timestamp":"2024-09-10T08:40:31.187Z"}
{"level":"info","message":"Latest block number: 20719081","timestamp":"2024-09-10T08:40:31.801Z"}
{"level":"info","message":"New block detected: 20719081","timestamp":"2024-09-10T08:40:31.804Z"}
{"level":"info","message":"New block detected: 20719082","timestamp":"2024-09-10T08:40:39.457Z"}
{"level":"info","message":"New block detected: 20719083","timestamp":"2024-09-10T08:40:51.638Z"}
{"level":"info","message":"New block detected: 20719084","timestamp":"2024-09-10T08:41:03.633Z"}
{"level":"info","message":"New block detected: 20719085","timestamp":"2024-09-10T08:41:15.627Z"}
{"level":"info","message":"Deposit transaction detected","timestamp":"2024-09-10T08:41:16.144Z","txHash":"0xa2c59939dbafee7b9a889ace511af5bc5da17cadf1c3d3397b5030bf27688545"}
{"level":"info","message":"Transaction sent to InfluxDB queue: 0xa2c59939dbafee7b9a889ace511af5bc5da17cadf1c3d3397b5030bf27688545","timestamp":"2024-09-10T08:41:16.207Z"}
{"level":"info","message":"Transaction sent to Telegram queue: 0xa2c59939dbafee7b9a889ace511af5bc5da17cadf1c3d3397b5030bf27688545","timestamp":"2024-09-10T08:41:16.208Z"}
{"level":"info","message":"New block detected: 20719086","timestamp":"2024-09-10T08:41:27.519Z"}
{"level":"info","message":"New block detected: 20719087","timestamp":"2024-09-10T08:41:39.543Z"}
{"level":"info","message":"New block detected: 20719088","timestamp":"2024-09-10T08:41:51.641Z"}
```

Terminal-2:

```
PS D:\eth-deposit-tracker\backend> node influxdb_worker.js
{"level":"info","message":"InfluxDB worker is waiting for messages","timestamp":"2024-09-10T12:56:44.169Z"}
```

```
PS D:\eth-deposit-tracker\backend> node influxdb_worker.js
{"level":"info","message":"InfluxDB worker is waiting for messages","timestamp":"2024-09-10T08:40:22.144Z"}
{"level":"info","message":"Saved deposit transaction to InfluxDB: 0xa2c59939dbafee7b9a889ace511af5bc5da17cadf1c3d3397b5030bf27688545","timestamp":"2024-09-10T08:41:16.214Z"}
```

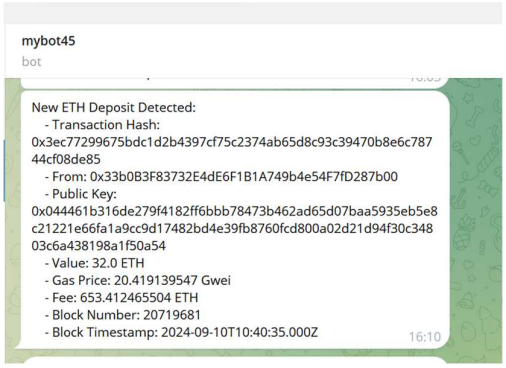
Terminal-3:

```
PS D:\eth-deposit-tracker\backend> node telegram_worker.js
{"level":"info","message":"Telegram worker is waiting for messages","timestamp":"2024-09-10T12:56:13.637Z"}
```

```
PS D:\eth-deposit-tracker\backend> node telegram_worker.js
{"level":"info","message":"Telegram worker is waiting for messages","timestamp":"2024-09-10T08:40:26.396Z"}
{"level":"info","message":"Telegram notification sent for transaction 0xa2c59939dbafee7b9a889ace511af5bc5da17cadf1c3d3397b5030bf27688545","timestamp":"2024-09-10T08:41:17.056Z"}
```

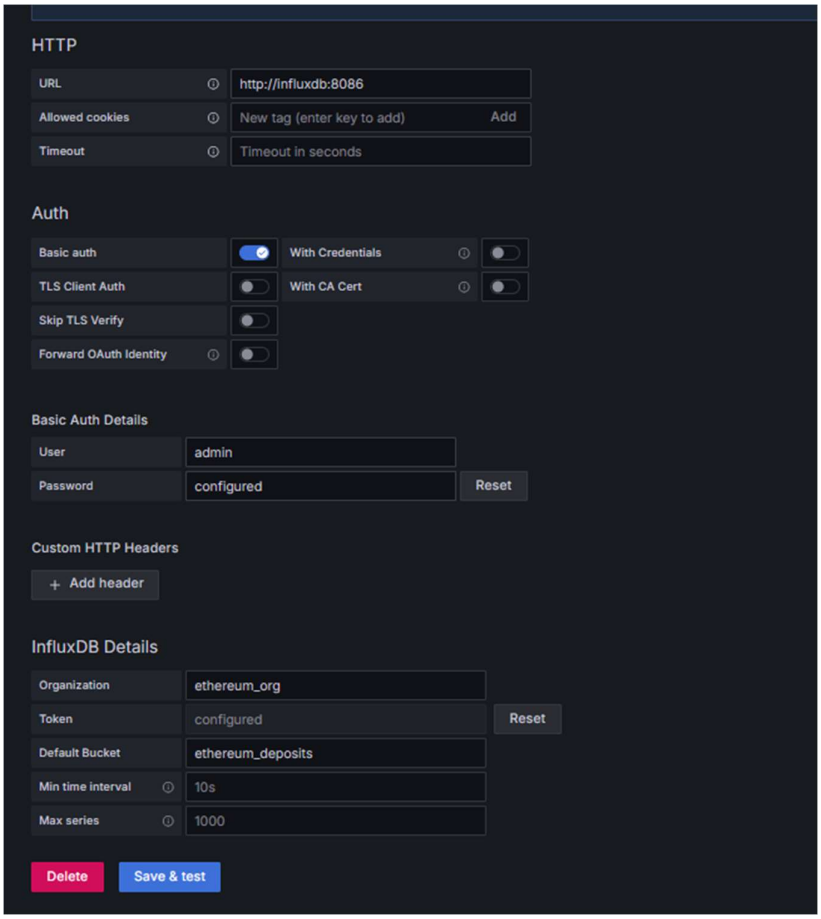
You can also check logs in logs/ folder.

Telegram Alert:



Setting up InfluxDB Data Source in Grafana

Follow these steps to set up the InfluxDB data source in Grafana. Go through this if required[https://youtu.be/Jszd7zrl-_U?si=WQhMbHZXDj5OaBgA]



1. Access Grafana UI

Open your browser and go to http://localhost:3000.

Log in with the following credentials:

- **Username:** admin
- **Password:** (from the Grafana.env file)

2. Add a New Data Source

On the Grafana home page, click the **gear icon (⚙)** on the left sidebar to open **Configuration**.

Select **Data Sources** from the menu.

3. Choose the Data Source Type

Click the **Add data source** button.

Scroll down and choose **InfluxDB** from the list of available data sources.

4. Configure InfluxDB Data Source

In the **HTTP** section, enter the following details:

- **URL:** http://influxdb:8086 or http://influxdb:8086.

In the **Auth** section, leave as default unless your InfluxDB requires additional authentication.

In the **InfluxDB Details** section, fill in:

- **Database:** Your InfluxDB database name (e.g., ethereum_deposits).

- **User:** The username for your InfluxDB.
- **Password:** The password for your InfluxDB.

5. Set Up Access Token (if needed)

If using tokens, provide an **InfluxDB Token**:

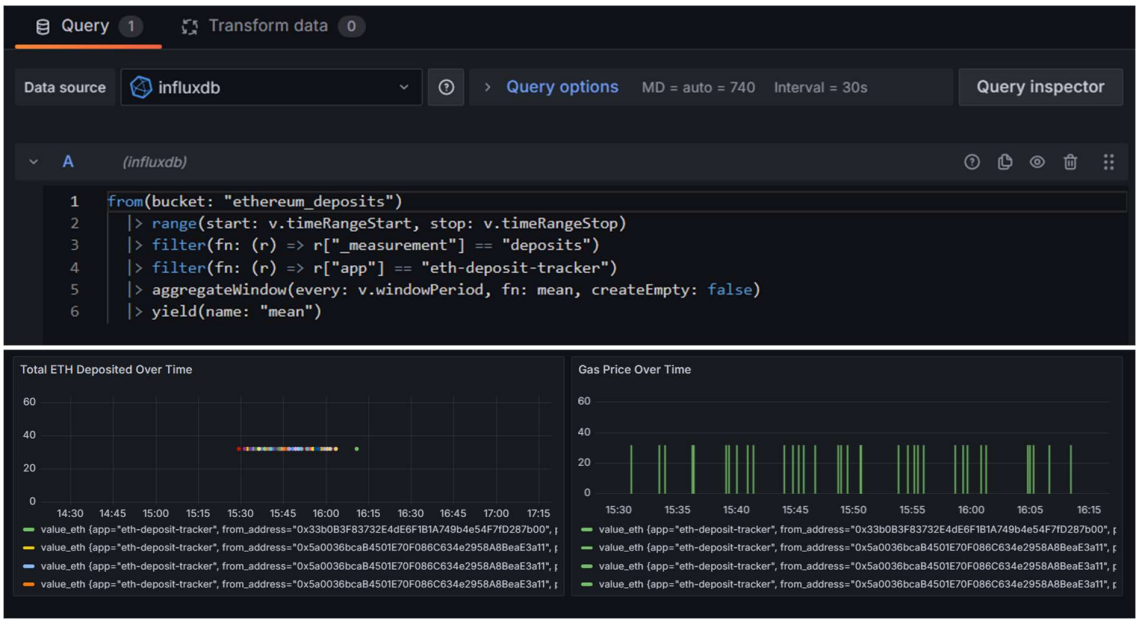
- Navigate to http://localhost:8086.
- Go to **Data -> Tokens** and copy the token for the Admin user.
- Paste this token into the **Auth Token** field in Grafana.

6. Save and Test

Click **Save & Test**.

Grafana will attempt to connect to InfluxDB. If successful, you'll see a confirmation message.

Now your InfluxDB data source is set up in Grafana!



Docker Setup for whole app (optional)

Run this in root directory

docker-compose up -d

Check application logs:

docker-compose logs app

Verify RabbitMQ:

Open a web browser and go to http://localhost:15672 Log in with the credentials (default: guest/guest) Check if the queues influxdb_queue and telegram_queue exist

Check logs for a specific service

docker-compose logs influxdb_worker

docker-compose logs telegram_worker