MinIO

SCALABLE OBJECT STORAGE

MinIO Intro

MinIO is a high-performance, scalable object storage solution compatible with Amazon S3 APIs. It is designed for large-scale deployments and offers features like horizontal scaling and low latency.

MinIO Key Features

Scalable

Supports petabytes of data. Scales horizontally with ease.

High Performance

Optimized for high throughput and low latency.

S3 Compatibility

Seamless integration with S3 APIs. Supports common S3 operations.

Open Source

Free and open-source software.Active community and regular updates.

Deployment Modes

On Premise

Deploy on local infrastructure.such as windows, linux

On Cloud

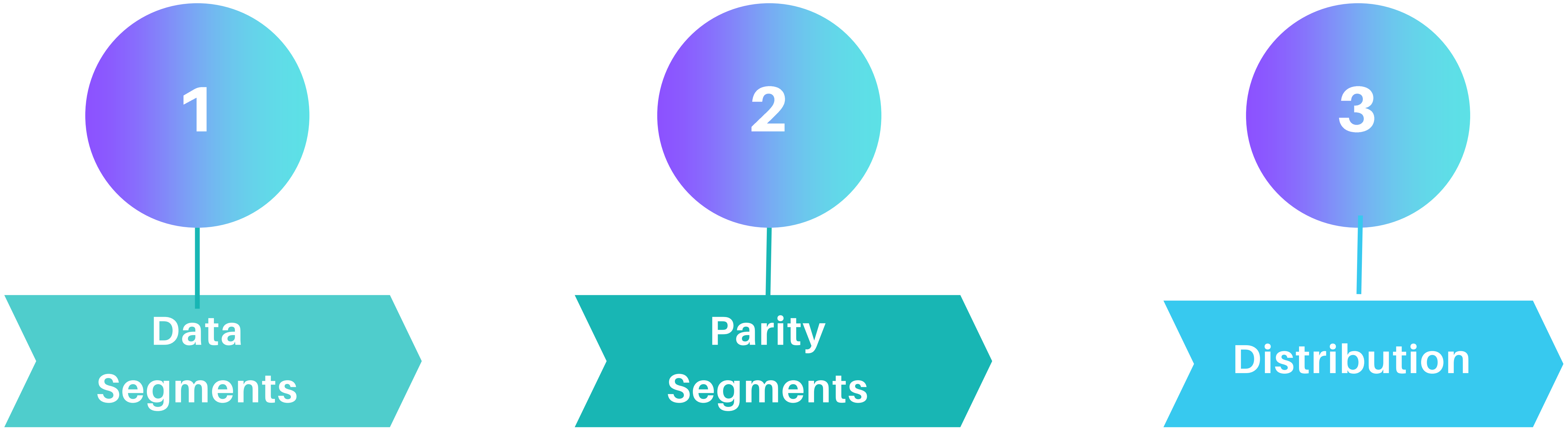
Deploy on cloud platforms (AWS, Azure, GCP).

Kubernetes

Deploy as a containerized application in Kubernetes.

Erasure Coding in MinIO

Erasure Coding is a method for data protection by dividing data into segments and adding redundancy.



1

Data Segments

Original data split into smaller chunks.

2

Parity Segments

Additional segments for redundancy.

3

Distribution

Data and parity segments are spread across drives or nodes.

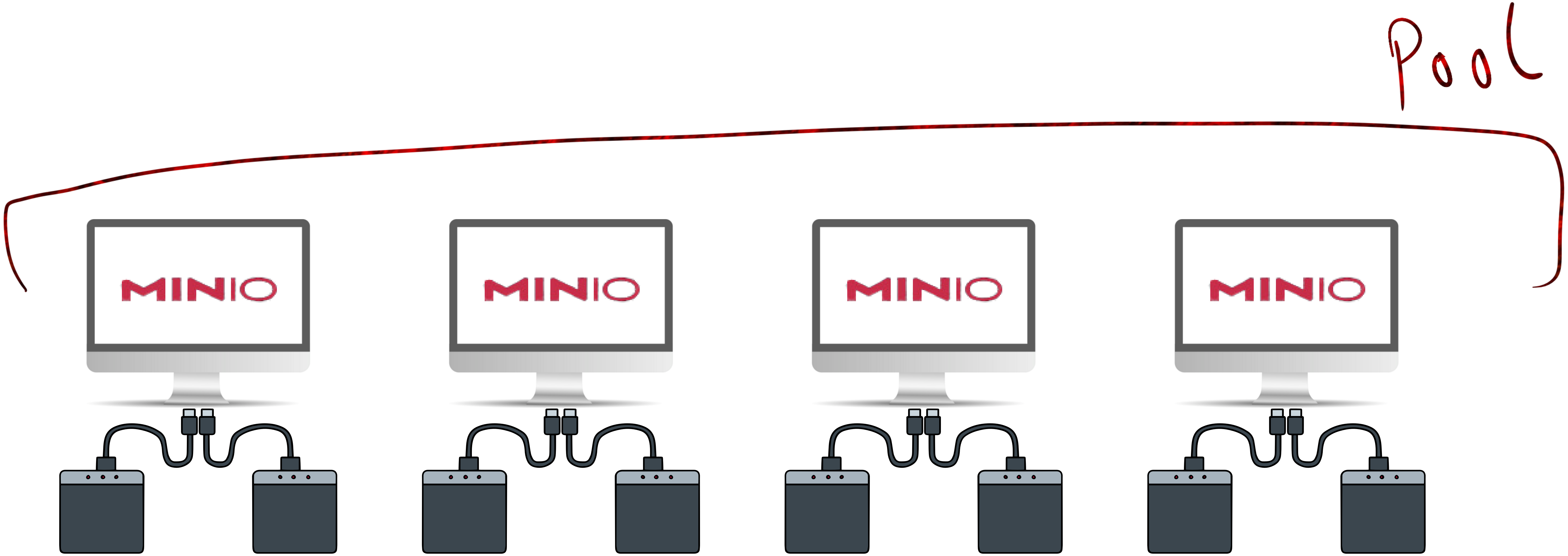
MinIO Server Pool

Components:

Drives: Storage devices within each server.

Nodes: Individual MinIO servers.

Erasure Sets: Logical groups of drives using erasure coding.



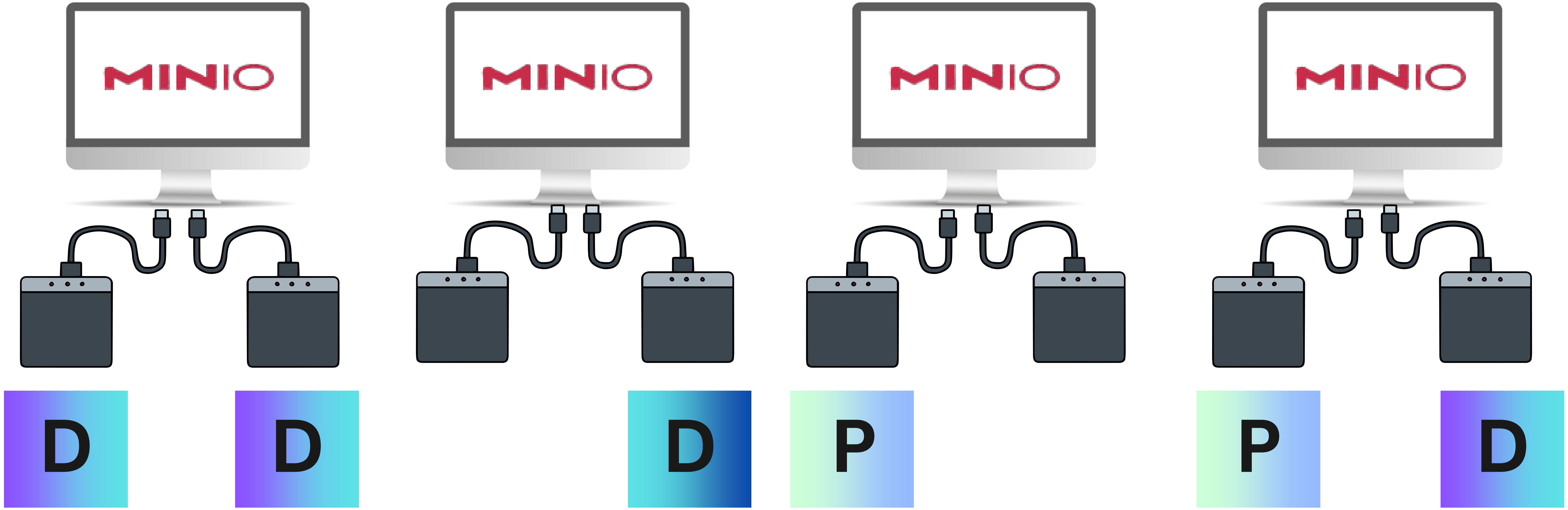
Erasure Set

Configuration within a server pool for distributing and protecting data.

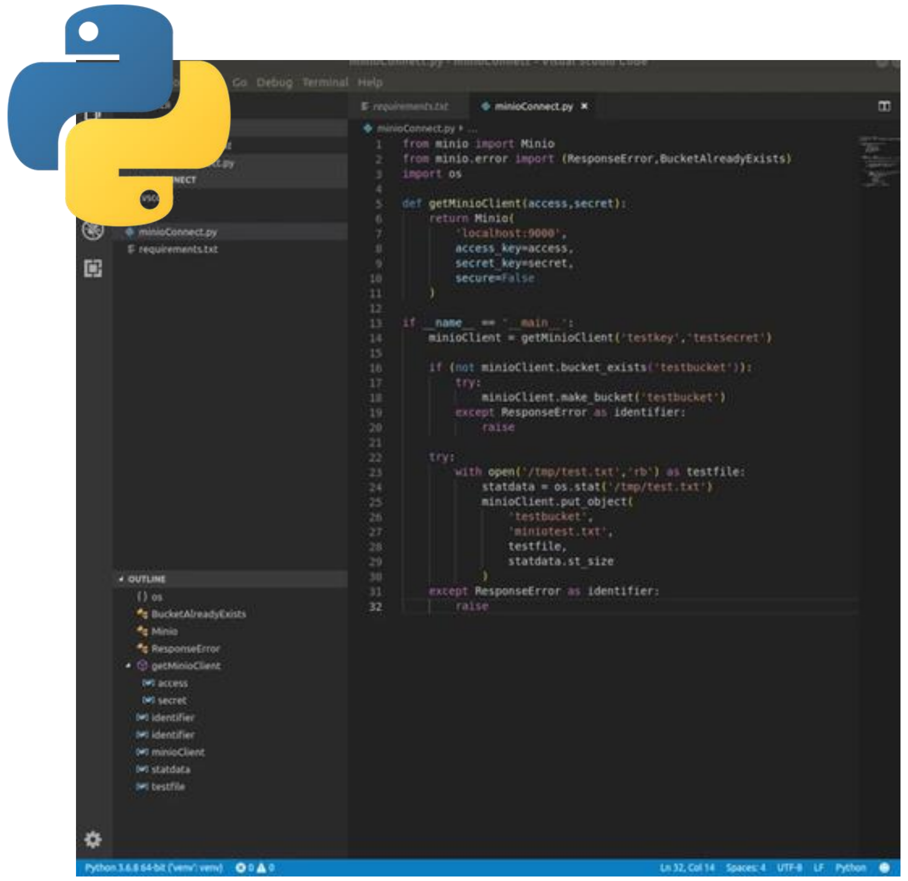
4+2 Erasure Set

Data is divided into 4 segments, with 2 parity segments for redundancy.

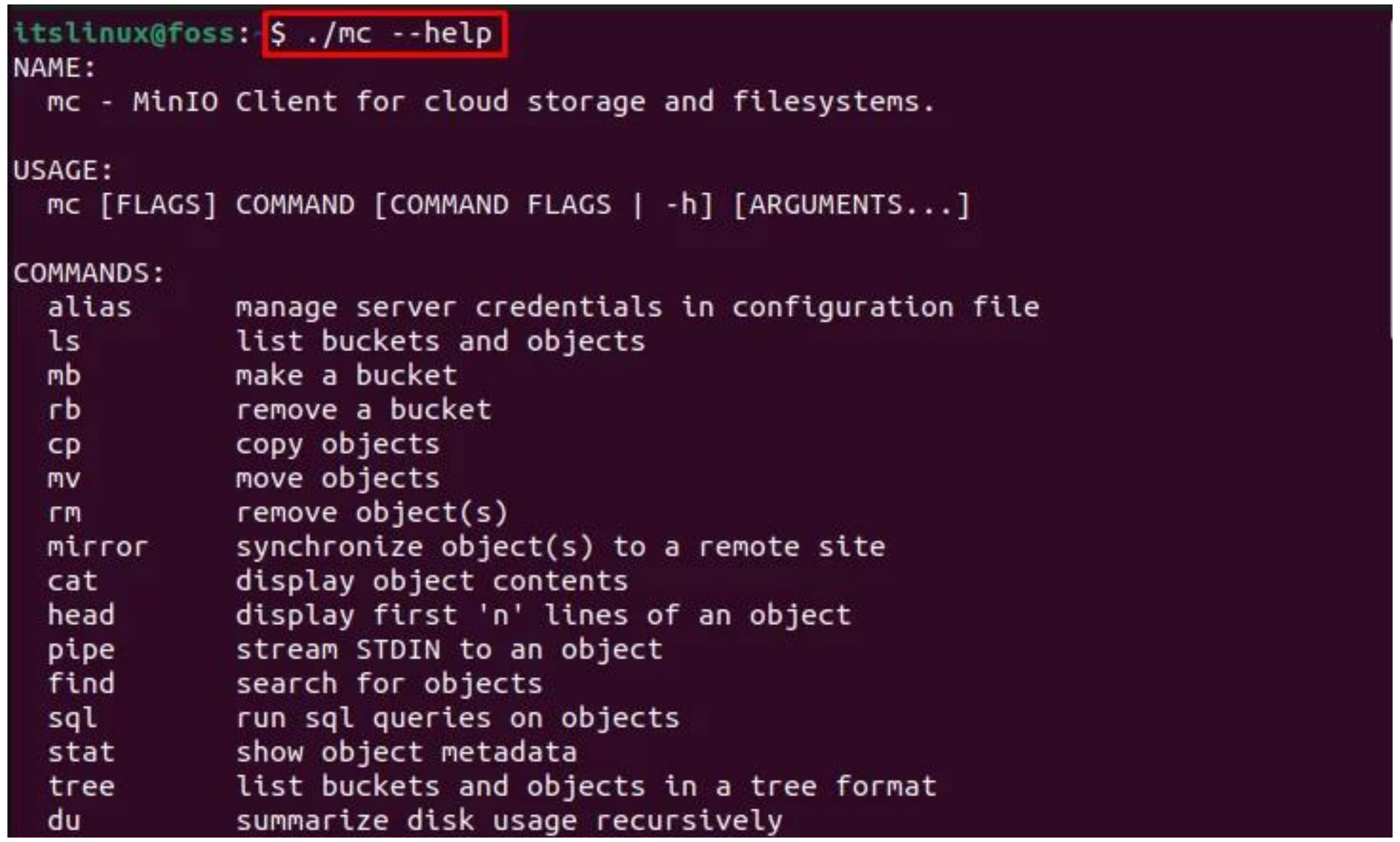
Tolerates up to 2 drive failures,



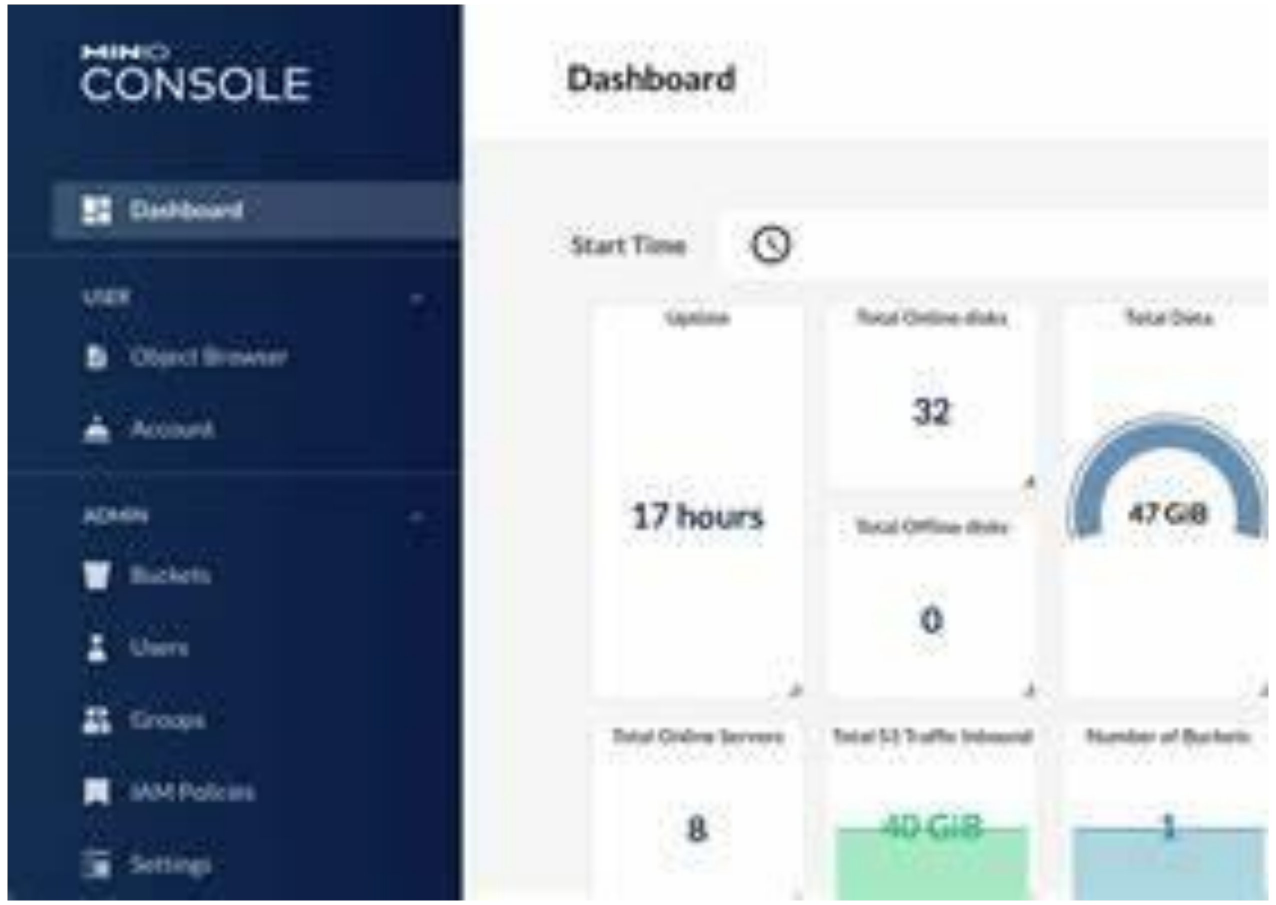
MinIO Clients



```
from minio import Minio
from minio.error import (ResponseError,BucketAlreadyExists)
import os

def getMinioClient(access,secret):
    return Minio(
        'localhost:9000',
        access_key=access,
        secret_key=secret,
        secure=False
    )

if __name__ == '__main__':
    minioClient = getMinioClient('testkey','testsecret')

    if (not minioClient.bucket_exists('testbucket')):
        try:
            minioClient.make_bucket('testbucket')
        except ResponseError as identifier:
            raise

    try:
        with open('/tmp/test.txt','rb') as testfile:
            statdata = os.stat('/tmp/test.txt')
            minioClient.put_object(
                'testbucket',
                'miniotest.txt',
                testfile,
                statdata.st_size
            )
    except ResponseError as identifier:
        raise
```



```
itslinux@foss:$ ./mc --help
NAME:
  mc - MinIO Client for cloud storage and filesystems.

USAGE:
  mc [FLAGS] COMMAND [COMMAND FLAGS | -h] [ARGUMENTS...]

COMMANDS:
  alias      manage server credentials in configuration file
  ls         list buckets and objects
  mb         make a bucket
  rb         remove a bucket
  cp         copy objects
  mv         move objects
  rm         remove object(s)
  mirror     synchronize object(s) to a remote site
  cat        display object contents
  head       display first 'n' lines of an object
  pipe       stream STDIN to an object
  find       search for objects
  sql        run sql queries on objects
  stat       show object metadata
  tree       list buckets and objects in a tree format
  du         summarize disk usage recursively
```



MINIO KEY COMPONENTS

Buckets

Containers for storing objects.

Objects

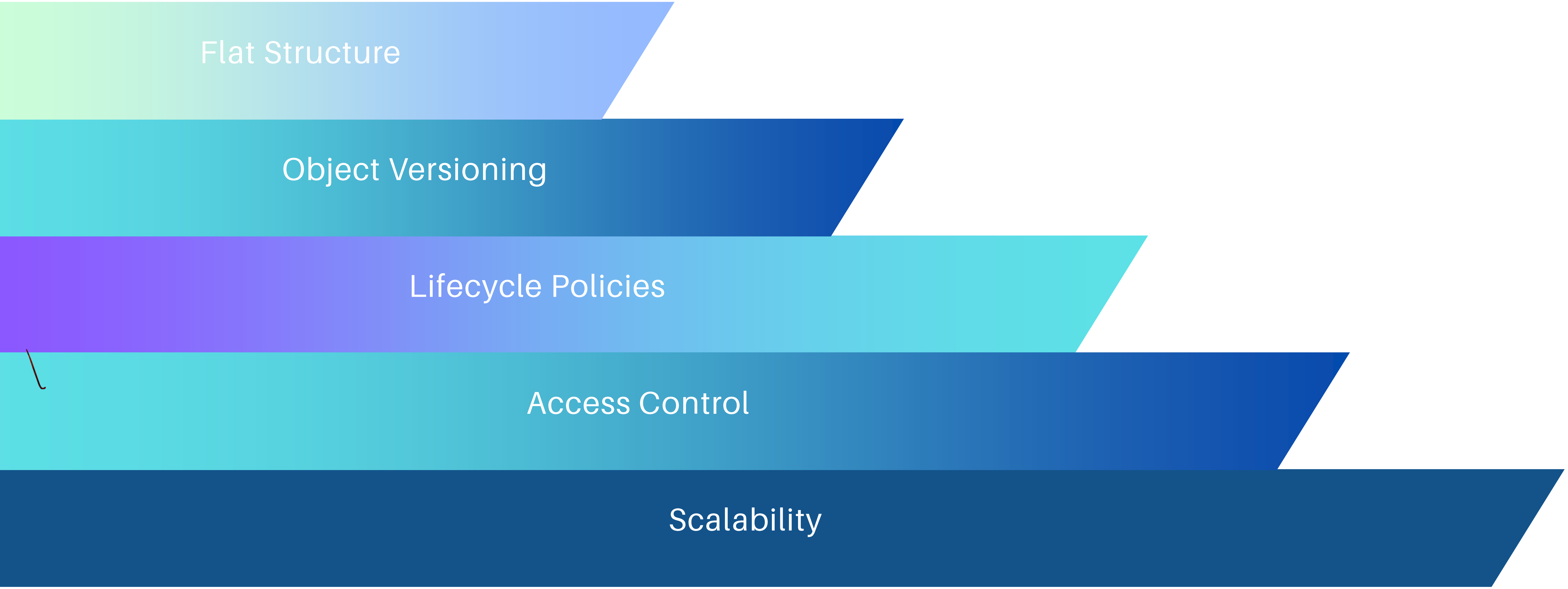
Data stored in buckets (files, images, etc.).

Policies

Control access and manage data lifecycle.

UNDERSTANDING MINIO BUCKETS

A bucket in MinIO is a container for storing objects, similar to folders in a file system.



Flat Structure

Object Versioning

Lifecycle Policies

Access Control

Scalability

BUCKET OPERATIONS

CREATE BUCKET

Instantiate a new bucket to store objects.

LIST BUCKETS

Retrieve a list of all buckets in the MinIO server.

DELETE BUCKET

Remove an existing bucket, including all its objects.

MinIO Object

An object is the fundamental unit of data storage, consisting of the data itself, metadata, and a unique identifier (key).

01 *Immutable*

Once an object is uploaded, it cannot be modified. To "update" an object, you must upload a new version.

02 *Unique Identifier*

Every object is identified by a unique key within a bucket, allowing for efficient data retrieval.

03 *Scalability*

Designed to store billions of objects across distributed systems.

04 *Support for Large Files*

MinIO supports multipart uploads, enabling the storage of large objects by breaking them into smaller part

05 *Data and Metadata*

Each object contains both the actual data and metadata (key-value pairs) that describe the object.

OBJECT OPERATIONS

UPLOAD

Store a new object in a bucket, assigning it a unique key.

RETRIEVE

Download the object using its key.

DELETE

Remove an object from a bucket.

VERSIONING

Retain multiple versions of an object.

VERSIONING IN MINIO

Versioning allows multiple versions of the same object to be stored in a MinIO bucket.

MULTIPLE VERSIONS

EVERY UPDATE TO AN OBJECT CREATES A NEW VERSION, ALLOWING YOU TO RETAIN A HISTORY OF CHANGES.

NON-DESTRUCTIVE OPERATIONS

Deleting an object does not permanently remove it; instead, it marks it as a delete marker, preserving older versions.

RESTORATION

EASILY RESTORE ANY PREVIOUS VERSION OF AN OBJECT, ENSURING YOU CAN RECOVER FROM ACCIDENTAL DELETIONS OR OVERWRIT

VERSION CONTROL

ENABLE OR SUSPEND VERSIONING ON BUCKETS AS NEEDED, OFFERING FLEXIBILITY IN DATA MANAGEMENT.

How Versioning Works

]

Object Upload

When an object is uploaded, it is stored as the latest version

Object Deletion

Deleting an object does not remove it; instead, a delete marker is added, and previous versions remain

No In-Place Modification

Objects are immutable. To change an object, you have to upload a new version of it.

No Versioning

If versioning is not enabled, uploading an object with the same key will overwrite the existing object without retaining the previous version

TLS – Transport Layer Protocol

TLS (Transport Layer Security): A cryptographic protocol designed to provide secure communication over a computer network.

Confidentiality

Data is encrypted and cannot be easily intercepted.

Integrity

Data cannot be altered or tampered with during transit.

Authentication

Verifies the identity of communicating parties

How TLS Works?

Client-Server Communication:

MinIO supports TLS to secure HTTP communications. It uses certificates to authenticate and encrypt the data between MinIO and the client.

Self-Signed or CA-Certificates:

MinIO can be configured with either self-signed certificates or certificates signed by a Certificate Authority (CA).

TLS Setup

1 *Generate Certificates*

Generate a private key and certificate using tools like OpenSSL.

2 *Place Certificates*

Store the private.key and public.crt files in the appropriate directories on the MinIO server. /root/.minio/certs/

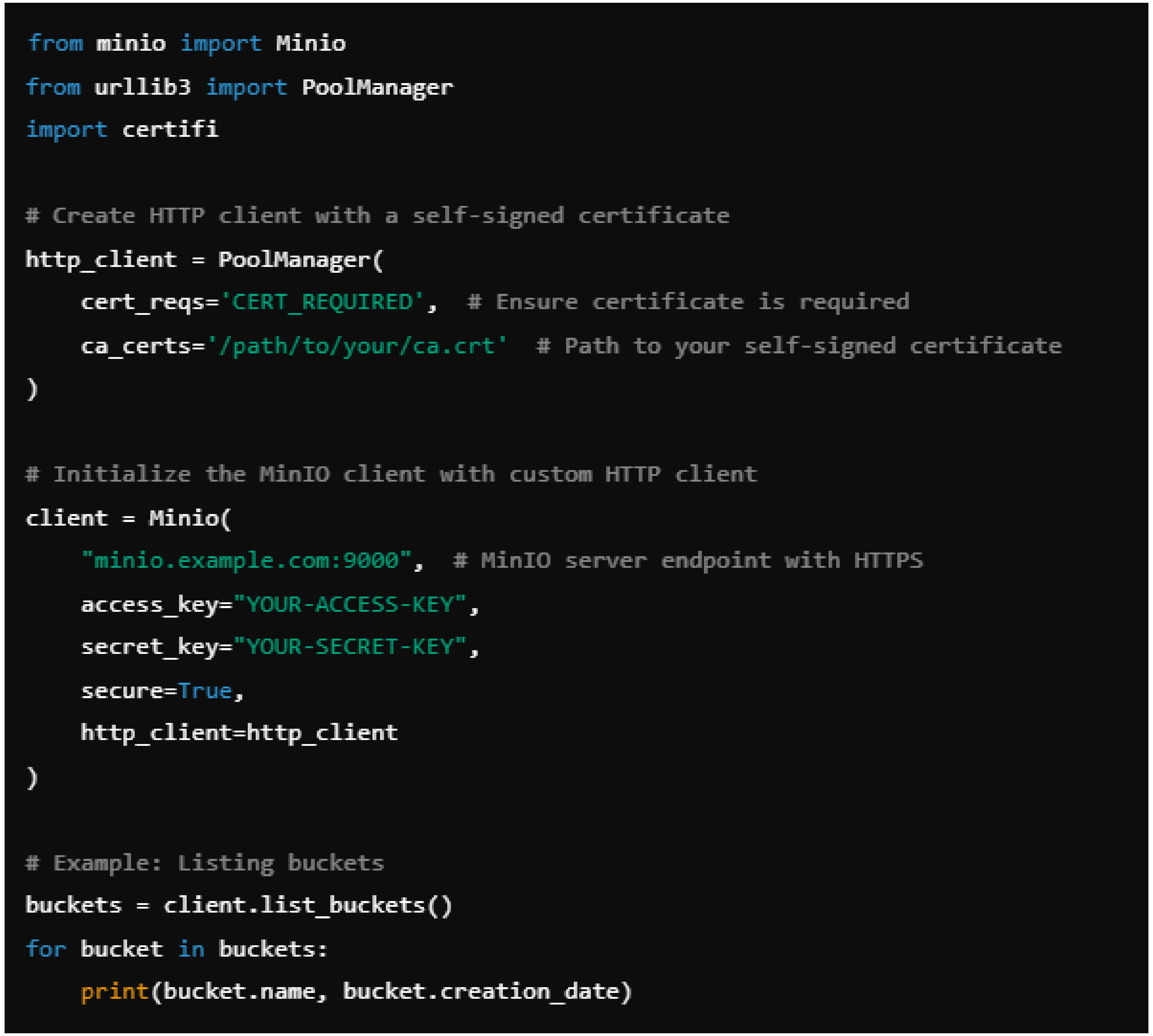
3 *Restart Minio*

Restart the MinIO service to apply TLS configuration.

4 Verify

Verify the connection using HTTPS (https://).

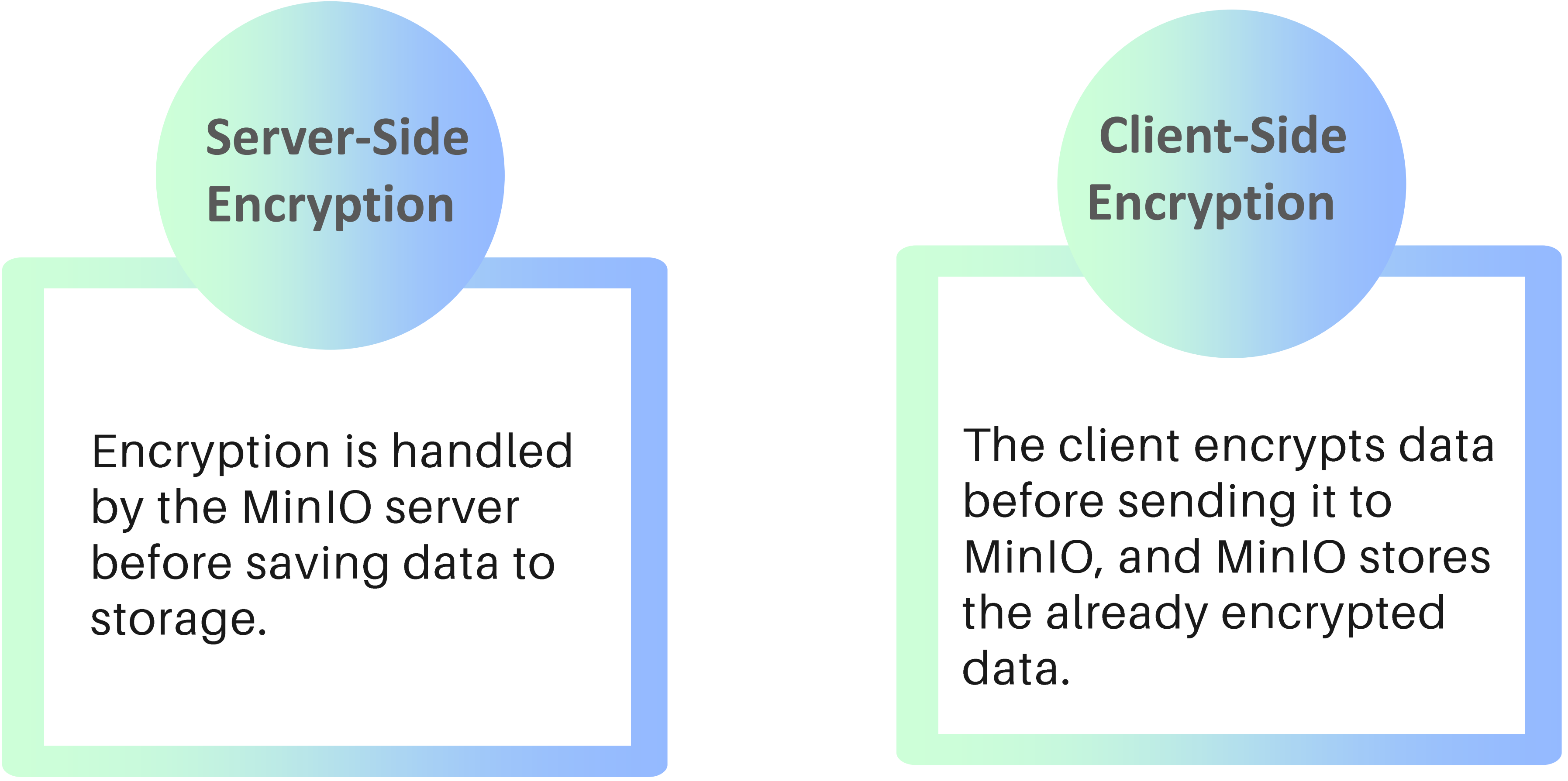
Connecting with Secured Cluster



```python
from minio import Minio
from urllib3 import PoolManager
import certifi

# Create HTTP client with a self-signed certificate
http_client = PoolManager(
    cert_reqs='CERT_REQUIRED',  # Ensure certificate is required
    ca_certs='/path/to/your/ca.crt'  # Path to your self-signed certificate
)

# Initialize the MinIO client with custom HTTP client
client = Minio(
    "minio.example.com:9000",  # MinIO server endpoint with HTTPS
    access_key="YOUR-ACCESS-KEY",
    secret_key="YOUR-SECRET-KEY",
    secure=True,
    http_client=http_client
)

# Example: Listing buckets
buckets = client.list_buckets()
for bucket in buckets:
    print(bucket.name, bucket.creation_date)
```

What is Encryption

Encryption in MinIO is the process of converting readable data (plaintext) into unreadable form (ciphertext) to protect it from unauthorized access.



Server-Side Encryption

Encryption is handled by the MinIO server before saving data to storage.

Client-Side Encryption

The client encrypts data before sending it to MinIO, and MinIO stores the already encrypted data.

Types of Encryptions

1

Server-Side Encryption (SSE)

1. **SSE-S3**: MinIO automatically encrypts objects using server-managed keys.
2. **SSE-C**: Server-Side Encryption with Client-Provided Keys, where clients manage their encryption keys.
3. **SSE-KMS**: Server-Side Encryption using Key Management Service (KMS) for key generation and management.

2

Client-Side Encryption (SSE)

1. Encryption happens on the client side before data is sent to MinIO.
2. Users manage their own encryption keys.
3. MinIO stores the encrypted data without knowledge of the encryption keys.

Benefits of KMS

Centralized Key Management

•Simplifies managing and securing encryption keys by providing a unified platform for creating, storing, and rotating keys.

Automated Key Rotation

Many KMS solutions offer automatic key rotation to maintain high security

Compliance

•KMS integrations help MinIO meet various regulatory compliance standards (e.g., GDPR, HIPAA) that require encryption and key management controls.

Access Control

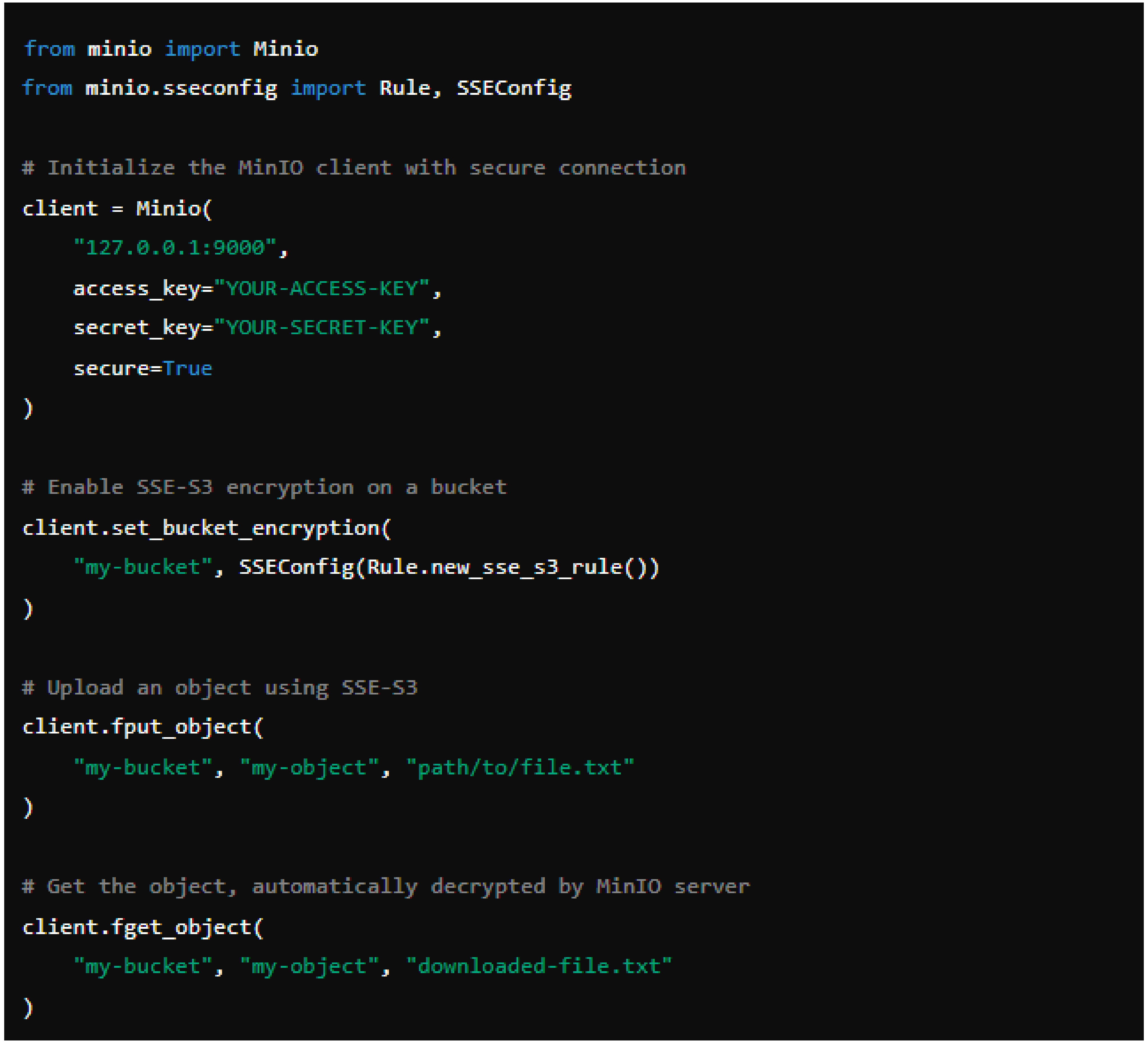
3

KMS solutions enable fine-grained access control, ensuring only authorized entities can manage and access the encryption keys

KMS

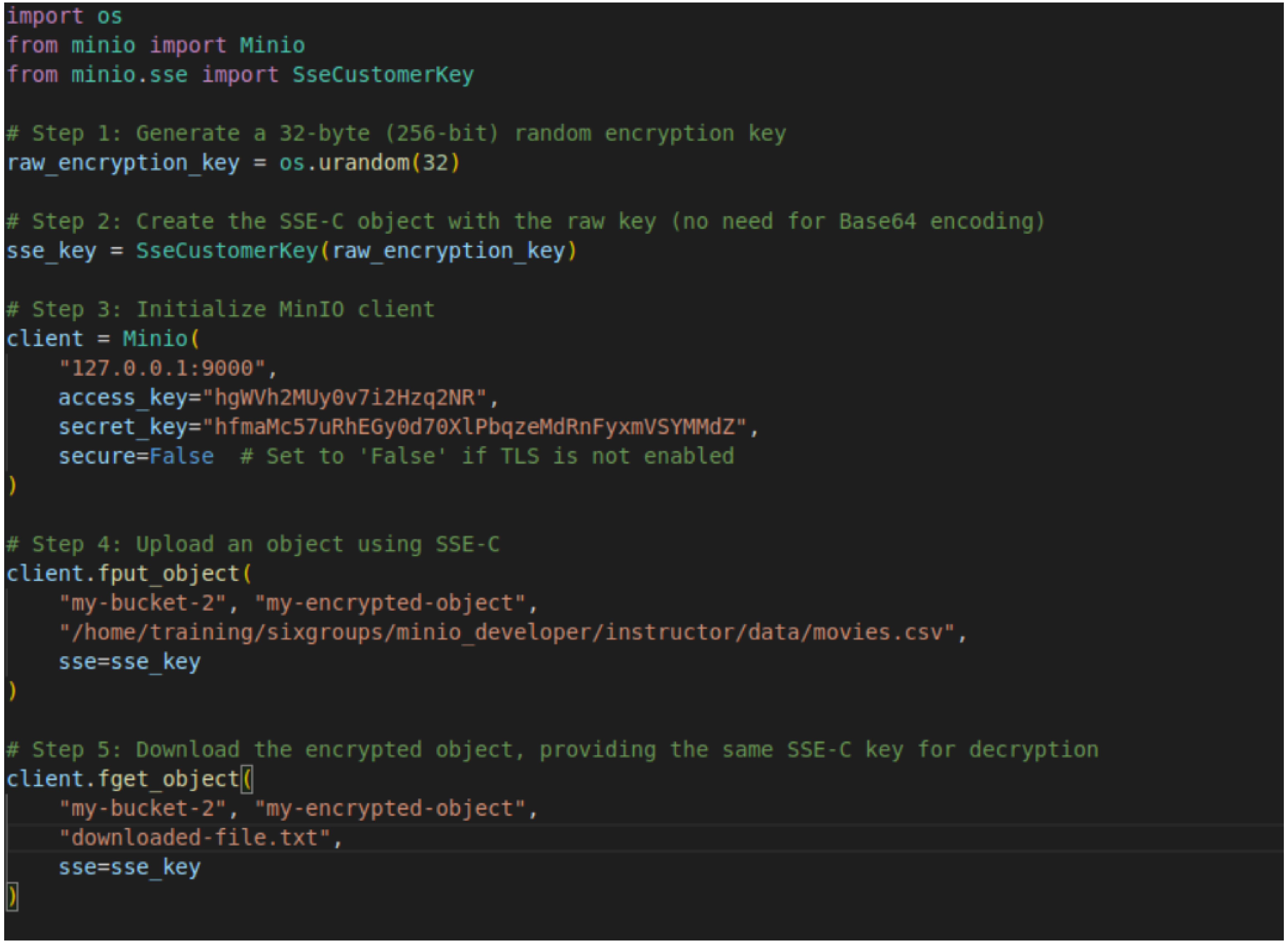
1. HashiVault Corp
2. AWS Secret Manager
3. Google Cloud Secret Manager
4. Azure Vault
5. Entrust Key Control
6. Fortanix
7. Thales CipherTrust Manager

SSE-S3 (Server-Side Encryption with S3-Managed Key)



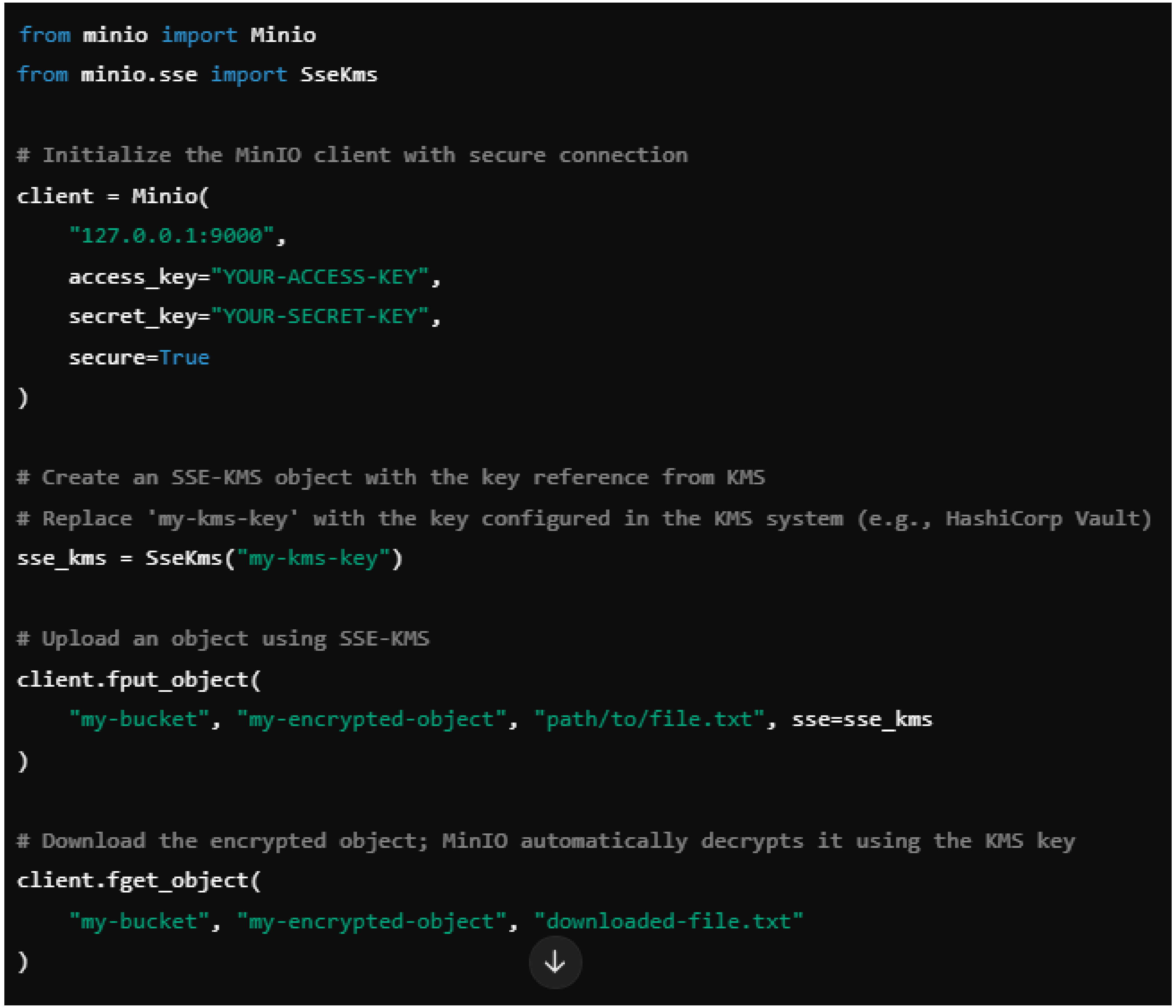
```python
from minio import Minio
from minio.sseconfig import Rule, SSEConfig

# Initialize the MinIO client with secure connection
client = Minio(
    "127.0.0.1:9000",
    access_key="YOUR-ACCESS-KEY",
    secret_key="YOUR-SECRET-KEY",
    secure=True
)

# Enable SSE-S3 encryption on a bucket
client.set_bucket_encryption(
    "my-bucket", SSEConfig(Rule.new_sse_s3_rule())
)

# Upload an object using SSE-S3
client.fput_object(
    "my-bucket", "my-object", "path/to/file.txt"
)

# Get the object, automatically decrypted by MinIO server
client.fget_object(
    "my-bucket", "my-object", "downloaded-file.txt"
)
```

SSE-C (Server-Side Encryption with Client-Provided Keys)



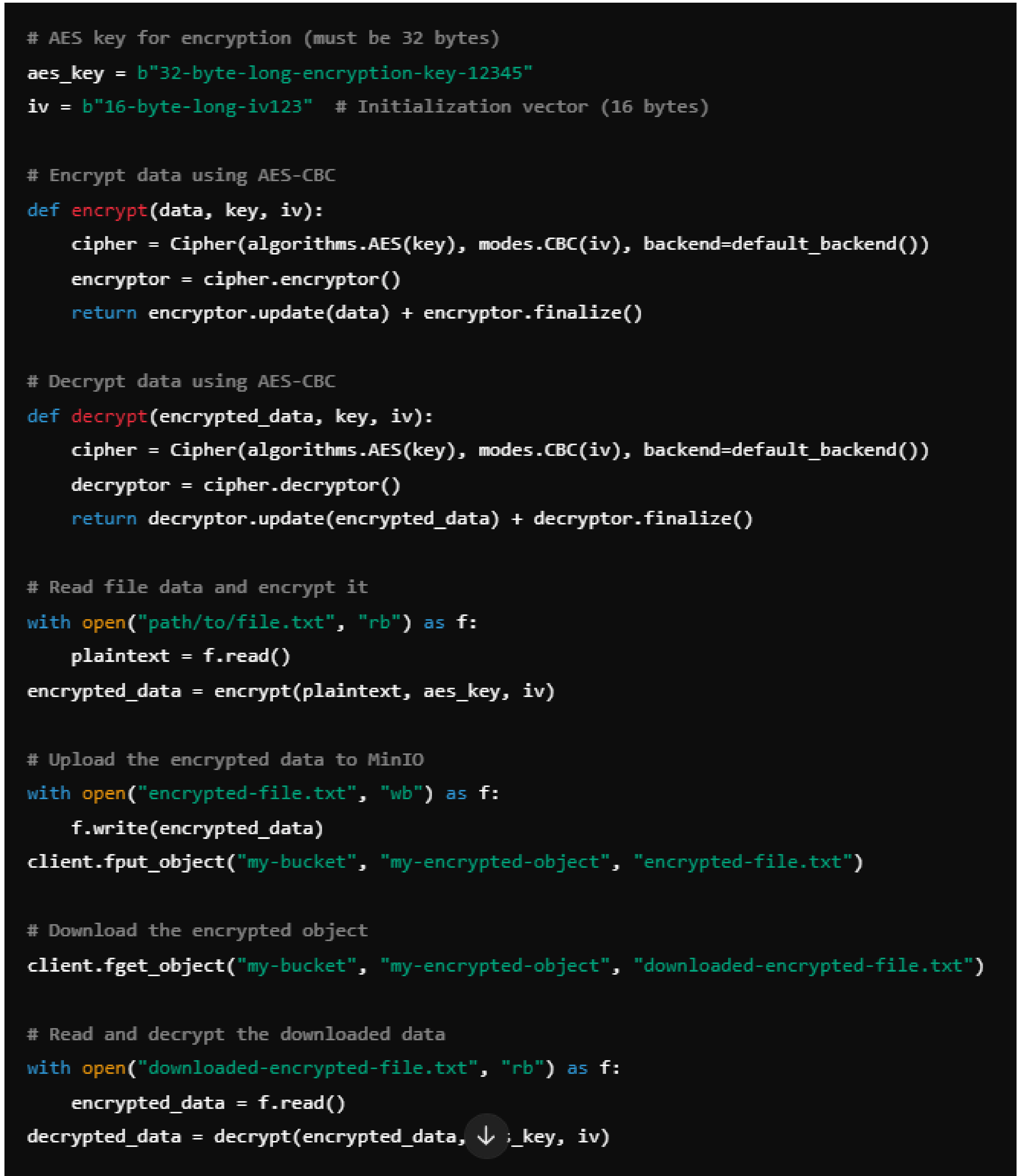
```python
import os
from minio import Minio
from minio.sse import SseCustomerKey

# Step 1: Generate a 32-byte (256-bit) random encryption key
raw_encryption_key = os.urandom(32)

# Step 2: Create the SSE-C object with the raw key (no need for Base64 encoding)
sse_key = SseCustomerKey(raw_encryption_key)

# Step 3: Initialize MinIO client
client = Minio(
    "127.0.0.1:9000",
    access_key="hgWVh2MUy0v7i2Hzq2NR",
    secret_key="hfmaMc57uRhEGy0d70XlPbqzeMdRnFyxmVSYMMdZ",
    secure=False  # Set to 'False' if TLS is not enabled
)

# Step 4: Upload an object using SSE-C
client.fput_object(
    "my-bucket-2", "my-encrypted-object",
    "/home/training/sixgroups/minio_developer/instructor/data/movies.csv",
    sse=sse_key
)

# Step 5: Download the encrypted object, providing the same SSE-C key for decryption
client.fget_object(
    "my-bucket-2", "my-encrypted-object",
    "downloaded-file.txt",
    sse=sse_key
)
```

SSE-KMS (Server-Side Encryption with Key Management System)



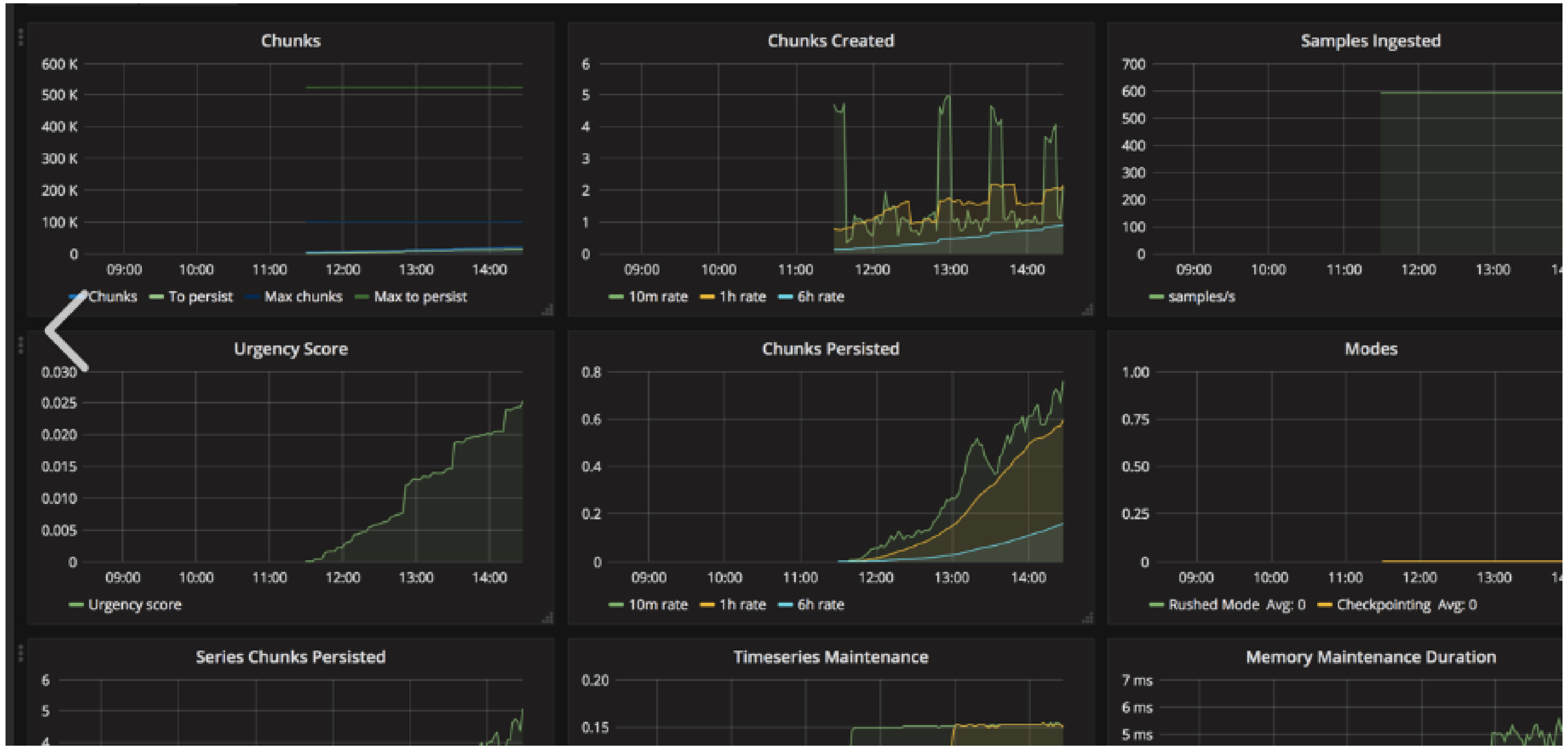
```python
from minio import Minio
from minio.sse import SseKms

# Initialize the MinIO client with secure connection
client = Minio(
    "127.0.0.1:9000",
    access_key="YOUR-ACCESS-KEY",
    secret_key="YOUR-SECRET-KEY",
    secure=True
)

# Create an SSE-KMS object with the key reference from KMS
# Replace 'my-kms-key' with the key configured in the KMS system (e.g., HashiCorp Vault)
sse_kms = SseKms("my-kms-key")

# Upload an object using SSE-KMS
client.fput_object(
    "my-bucket", "my-encrypted-object", "path/to/file.txt", sse=sse_kms
)

# Download the encrypted object; MinIO automatically decrypts it using the KMS key
client.fget_object(
    "my-bucket", "my-encrypted-object", "downloaded-file.txt"
)
```

Client Side Encryption/ Decryption



```
# AES key for encryption (must be 32 bytes)
aes_key = b"32-byte-long-encryption-key-12345"
iv = b"16-byte-long-iv123"  # Initialization vector (16 bytes)

# Encrypt data using AES-CBC
def encrypt(data, key, iv):
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=default_backend())
    encryptor = cipher.encryptor()
    return encryptor.update(data) + encryptor.finalize()

# Decrypt data using AES-CBC
def decrypt(encrypted_data, key, iv):
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=default_backend())
    decryptor = cipher.decryptor()
    return decryptor.update(encrypted_data) + decryptor.finalize()

# Read file data and encrypt it
with open("path/to/file.txt", "rb") as f:
    plaintext = f.read()
encrypted_data = encrypt(plaintext, aes_key, iv)

# Upload the encrypted data to MinIO
with open("encrypted-file.txt", "wb") as f:
    f.write(encrypted_data)
client.fput_object("my-bucket", "my-encrypted-object", "encrypted-file.txt")

# Download the encrypted object
client.fget_object("my-bucket", "my-encrypted-object", "downloaded-encrypted-file.txt")

# Read and decrypt the downloaded data
with open("downloaded-encrypted-file.txt", "rb") as f:
    encrypted_data = f.read()
decrypted_data = decrypt(encrypted_data, aes_key, iv)
```

Monitoring with Prometheus



Open Source

Open source tool
for monitoring

Alert Mechanism

Inbuilt alerting system
For sending notification in
case of surges.

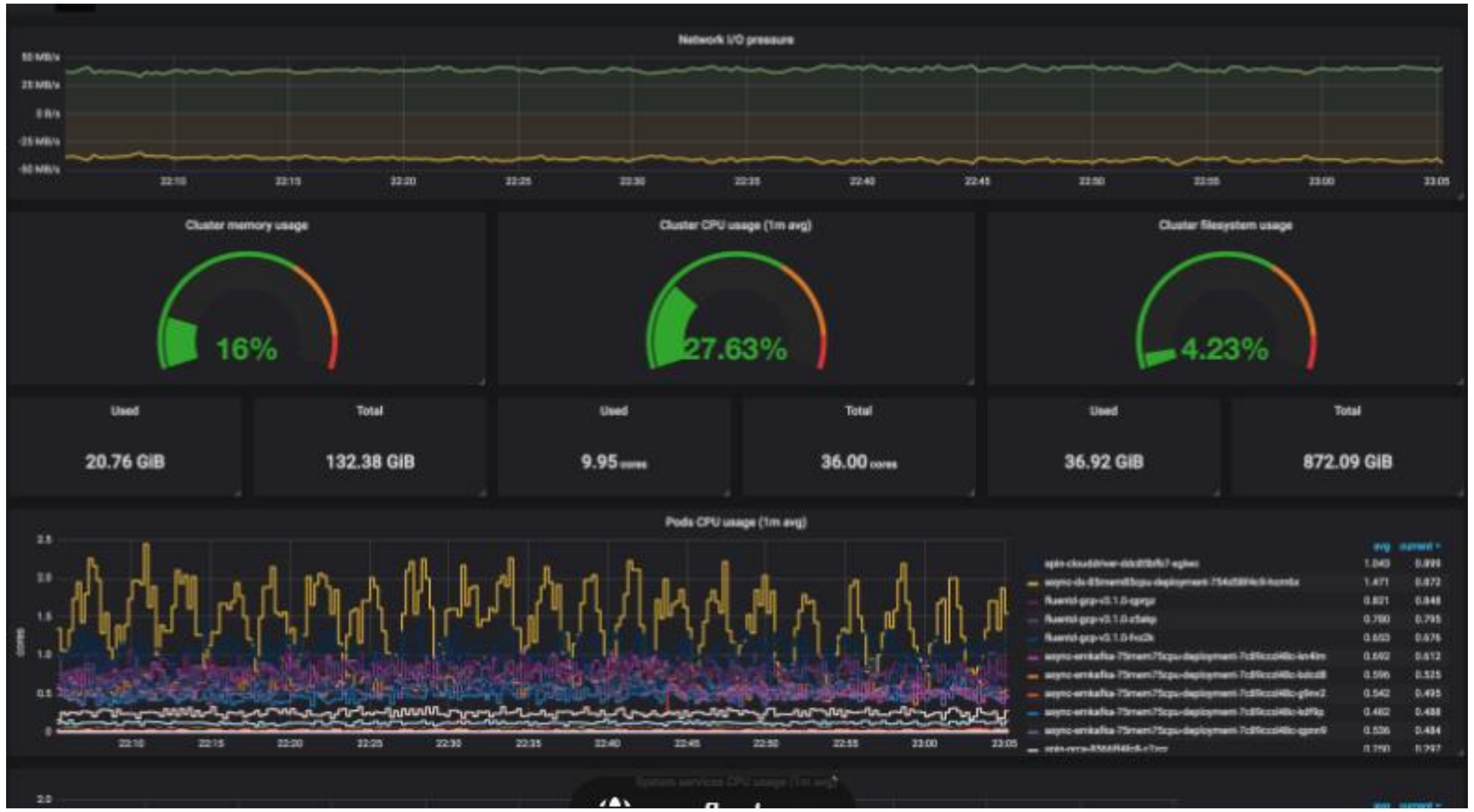
Pull Based

Pull-based data collection
For collecting metrics like
cpu usage, memory
consumption etc.

PromQL

PromQL for querying
metrics..

Visualization with Grafana



Interactive

Using Grafana, you can make Interactive & customizable dashboards

Real Time

Grafana provides you real time data visualization.

Alerts

Using Grafana, we can set up real time alerts to send emails and notifications

Reports

You can also create and share reports.