Fachhochschule Nordwestschweiz
Hochschule für Architektur, Bau und Geomatik

PCLS – Autoscaling Nextcloud in AWS

Skalierung von Nextcloud in der AWS-Umgebung durch Autoscaling

Viktor Weilenmann, Damjan Mlinar

Brugg, 07.01.2024

Viktor.Weilenmann@fhnw.ch
Damjan.Mlinar@fhnw.ch

Inhaltsverzeichnis

1	Beschreibung Use Case	3
2	Vorgehensweise für Umsetzung	4
3	Gewählte Architektur	5
4	Beschreibung der Implementation	7
4.1	RDS DB	7
4.2	S3 Bucket (Data Storage)	8
4.3	Nicht umgesetzt: ElastiCache for Redis	8
4.4	EC2	9
4.5	Load Balancer	11
4.6	Auto Scaling	12
4.7	Virtual Private Cloud	13
5	Erkenntnisse und Fazit	15
5.1	Mögliche zukünftige Anpassungen	15

1 Beschreibung Use Case

Dieses Projekt wurde im Rahmen des PCLS Moduls, während des Herbstsemesters 2023, an der Fachhochschule Nordwestschweiz umgesetzt.

Für das Abschlussprojekt haben wir uns das Ziel gesetzt, eine gehostete Version von Nextcloud bereitzustellen. Nextcloud ist eine Open-Source-Software, die es Nutzern ermöglicht, Daten wie Files, Kalender und Kontakte in der Cloud zu speichern. Der Zugriff auf diese Daten ist sowohl über eine Weboberfläche als auch über Client-Applikationen für Smartphones und Desktops möglich. Die Synchronisation zwischen Server und Clients gewährleistet einen konsistenten Datenbestand, der von verschiedenen Endgeräten aus abgerufen werden kann. Zusätzlich bietet Nextcloud Funktionen für Videokonferenzen sowie verschiedene Office-Applikationen über die Weboberfläche.

Die Besonderheit von Nextcloud liegt darin, dass es auf einem privaten Server oder Webspace betrieben werden kann. Dies ermöglicht es dem Nutzer, die Kontrolle über die verwalteten Daten zu behalten und potenzielle Risiken für Datenmissbrauch durch Diensteanbieter zu vermeiden. Insbesondere die kommerzielle Verwertung von Daten mit oft unklaren Datenschutzrichtlinien kann durch den Betrieb eines eigenen Nextcloud-Servers umgangen werden.

In unserem Projekt stand die Herausforderung, die optimale Konfiguration für das Hosting von Nextcloud auf Amazon Web Services (AWS) zu ermitteln. Unser Hauptziel bestand darin, dem Kunden eine detaillierte Anleitung bereitzustellen, mit der er die erforderliche Infrastruktur eigenständig und ohne umfassende AWS-Kenntnisse aufbauen kann. Das Ergebnis sollte ein reibungsloser und schneller Einsatz von Nextcloud auf der AWS-Plattform sein.

2 Vorgehensweise für Umsetzung

Wir hatten beide noch nie mit AWS gearbeitet, die einzigen Inputs bekamen wir aus dem PCLS Modul während dem Semester. Wir stellten uns die Frage, was wir von diesem Projekt mitnehmen wollten. Beiden war es wichtig erste Erfahrungen mit dem Infrastructure-as-Code Tool Terraform zu machen. Um einen guten ersten Eindruck mit AWS zu bekommen wollten wir auch unbedingt den Infrastructure-as-a-Service (IaaS) Ansatz verwenden. Von den Services von AWS her wollten wir uns auf drei Dinge fokusieren: Wie verwendet man Virtual Private Cloud mit den Features, wie verbinden sich EC2 Instanzen mit RDS Datenbanken und wie konfiguriert man Autoscaling.

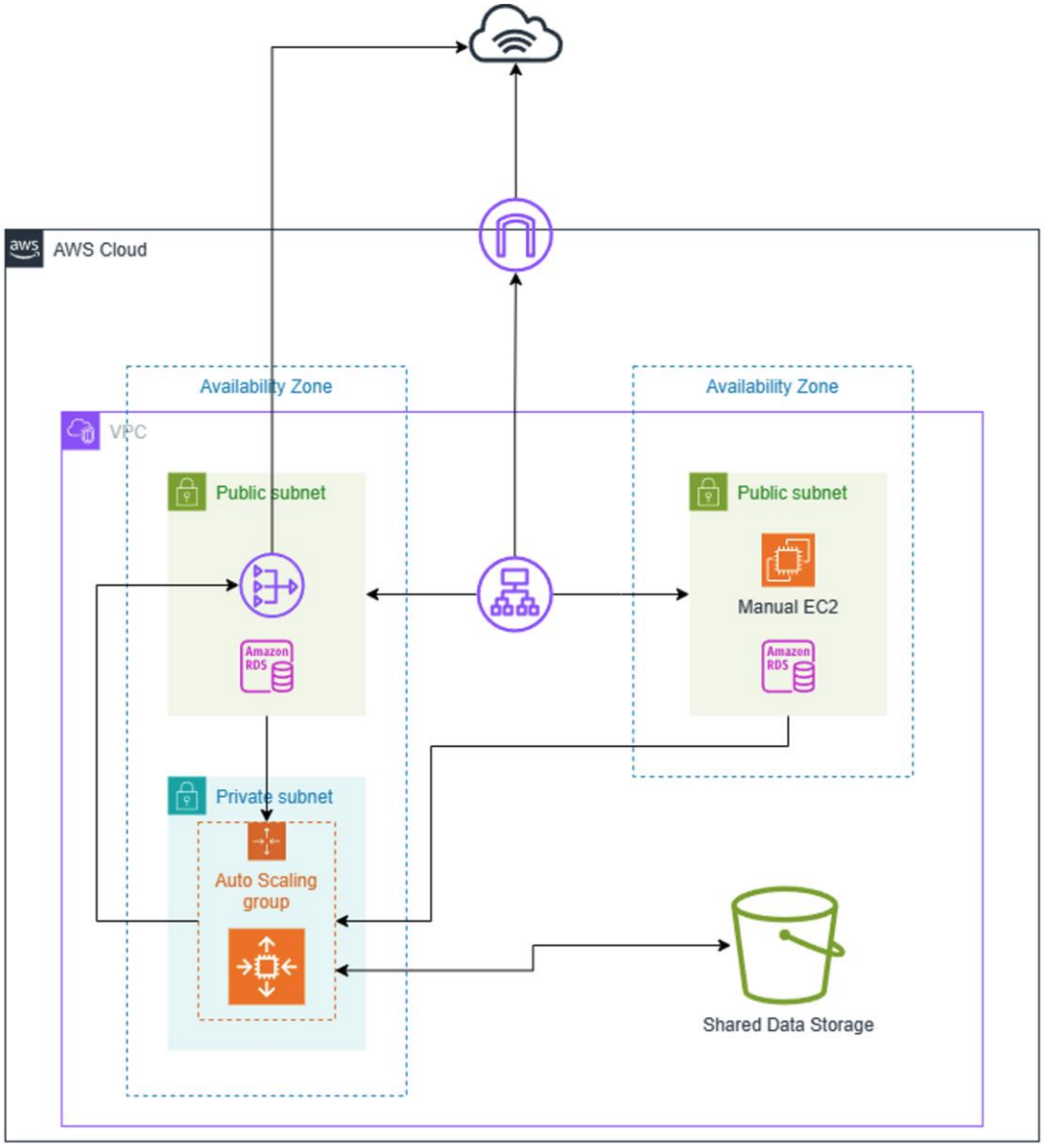
Die Wahl von Terraform als Infrastruktur-as-Code (IaC) Tool bietet uns eine flexible und konsistente Möglichkeit, unsere AWS-Ressourcen zu verwalten. Terraform ermöglicht es uns, die gesamte Infrastruktur, einschliesslich EC2-Instanzen, Datenbanken und Netzwerkkomponenten, in einem einzigen Konfigurationscode zu definieren. Dies fördert die Wiederverwendbarkeit, Konsistenz und Skalierbarkeit unserer Infrastrukturdefinitionen.

Die Entscheidung für AWS als unseren bevorzugten Cloud-Anbieter basiert massgeblich auf seiner Position als unangefochtener Marktführer und dem damit verbundenen hohen Stellenwert auf dem Cloud Engineer-Jobmarkt. Zudem trug der Fokus auf AWS anstelle von Azure während des Unterrichts dazu bei, dass wir uns mit den spezifischen Services dieser Plattform vertrauter fühlten als mit Azure.

Mit dem Ziel, das Beste aus unserem Projekt herauszuholen, haben wir uns für Infrastructure-as a-Service (IaaS) entschieden. In Anlehnung an das Konzept von «New Pizza as a Service» möchten wir nicht nur die fertige Pizza (Pizza-as-a-Service) erhalten, sondern auch den gesamten Herstellungsprozess nach unseren individuellen Anforderungen gestalten (Kitchen-as-a-Service). Dabei minimiert IaaS den Vendor Lock-in, indem es uns ermöglicht, die Kontrolle über die Infrastruktur zu behalten und gegebenfalls den Cloud Provider in Zukunft zu wechseln.

Die Wahl der Services von AWS wird im Kapitel 3 beschrieben.

3 Gewählte Architektur



Unsere Netzwerktopologie ist darauf ausgerichtet, den Load Balancer jeweils in zwei Availability Zonen (eu-north-1a, eu-north-1b) zu platzieren. Der Vorteil der Nutzung von zwei Availability Zonen liegt in der Verbesserung der Ausfallsicherheit. Sollte eine Verfügbarkeitszone ausfallen, kann der Load Balancer den Verkehr auf die andere Zone umleiten, was eine höhere Zuverlässigkeit gewährleistet.

Die eigentlichen Instanzen werden dynamisch nach Abfragen im privaten Subnetz erstellt und durch Auto Scaling verwaltet. Der Vorteil der Platzierung von Instanzen in privaten Subnetzen besteht darin, eine zusätzliche Sicherheitsebene zu schaffen, indem der direkte Zugriff von aussen eingeschränkt wird. Der Load Balancer befindet sich im öffentlichen Subnetz und dient als Proxy um den Verkehr von aussen zu handhaben und auf die privaten Instanzen weiterzuleiten.

Um den Instanzen dennoch den Zugriff auf das Internet zu ermöglichen, wurde ein NAT Gateway erstellt, das private Adressen in öffentliche übersetzt. Dies ermöglicht den Instanzen die Kommunikation mit externen Ressourcen, ohne direkt von aussen erreichbar zu sein.

Die EC2-Instanzen speichern ihre Daten in den Shared Data Storage, während Benutzerkonten in den RDS-Datenbanken abgelegt werden. Der Endbenutzer erreicht die Nextcloud-Anwendung über den Domainnamen des Load Balancers, der den Lastenausgleich auf die Instanzen in den verschiedenen Availability Zonen verteilt.

4 Beschreibung der Implementation

4.1 RDS DB

Für die Datenbankintegration haben wir eine PostgreSQL-Datenbank mithilfe von Terraform erstellt. Die nachfolgende Terraform-Konfiguration zeigt die relevanten Parameter und Einstellungen für die Instanzierung der PostgreSQL-Datenbank:

```
#create database
resource "aws_db_instance" "postgres" {
  allocated_storage    = 20
  storage_type         = "gp2"
  engine               = "postgres"
  engine_version       = "15"
  instance_class       = "db.t3.small"
  db_name              = "postgres"
  username             = "nextcloud"
  password             = "nextcloud"
  skip_final_snapshot  = true
  publicly_accessible = true
  vpc_security_group_ids = [aws_security_group.ec2-securitygroups.id]
  db_subnet_group_name = aws_db_subnet_group.postgres_subnet_group.name
  depends_on = [aws_internet_gateway.gw]
}
```

Wir haben Subnetze in unserer Architektur integriert, um Netzwerkkomponenten effektiv zu organisieren und zu isolieren. Hier sind die relevanten Teile unserer Terraform-Konfiguration:

```
  # subnet linking to database
  resource "aws_db_subnet_group" "postgres_subnet_group" {
    name       = "postgresubgroup"
    subnet_ids = [aws_subnet.pub-sub-1.id,aws_subnet.pub-sub-2.id]
  }
```

Diese Ressource definiert eine Gruppe von Subnetzen, die für die PostgreSQL-Datenbankinstanz verwendet werden. Die Subnetze pub-sub-1 und pub-sub-2 sind hier eingebunden, um eine optimale Verteilung und Verfügbarkeit zu gewährleisten. Diese Subnetze sind jeweils in unterschiedlichen Availability Zones lokalisiert, nämlich "eu-north-1a" für pub-sub-1 und "eu-north-1b" für pub-sub-2.

4.2 S3 Bucket (Data Storage)

Die Einrichtung des Buckets für den gemeinsamen Serverdatenspeicher von Nextcloud gestaltet sich dank Terraform äusserst einfach:

```
#create shared S3 bucket
resource "aws_s3_bucket" "s3bucket" {
  bucket = "nextcloud-aio-bucket-pcls"
}
```

Für die Authentifizierung und den Zugriff auf diesen Bucket verwendet Nextcloud Access Keys und Secrets. Aufgrund dieser Konfiguration benötigen wir keine weiteren spezifischen Einstellungen für den Bucket selbst. Die Integration von Terraform in diesen Prozess vereinfacht nicht nur die Bereitstellung, sondern gewährleistet auch die Konsistenz und Wiederholbarkeit dieser Konfiguration in verschiedenen Umgebungen.

4.3 Nicht umgesetzt: ElastiCache for Redis

Die Integration des Redis-Caches, der die Leistung und Reaktionsfähigkeit von Nextcloud verbessert, konnte aufgrund von Schwierigkeiten bei der Inbetriebnahme und Verbindungsproblemen mit dem Redis Cluster leider nicht erfolgreich umgesetzt werden. Der Cache speichert temporäre Daten im Arbeitsspeicher, um häufig verwendete Informationen schneller abrufen zu können. Diese Entscheidung wurde getroffen, um uns auf die erfolgreiche Implementierung anderer Schlüsselelemente des Projekts zu konzentrieren. Für künftige Entwicklungen bleibt die Integration eines Redis-Caches jedoch eine vielversprechende Möglichkeit zur weiteren Optimierung der Leistung von Nextcloud.

4.4 EC2

Unsere Implementierung erfolgte in zwei Teilen: Zuerst erfolgte die manuelle Instanzierung, gefolgt vom Einsatz des Autoscaling-Skripts.

```
# Create first EC2 instance
resource "aws_instance" "server" {
  ami           = "ami-0014ce3e52359afbd"
  instance_type = "t3.micro"
  availability_zone = "eu-north-1a"
  key_name = "pcls-sshkey"
  network_interface {
    device_index = 0
    network_interface_id = aws_network_interface.pcls-networkinterface.id
  }

  user_data = templatefile("setup.sh.tpl", {
access_key = var.access_key
secret_key = var.secret_key
postgres_host = aws_db_instance.postgres.address
postgres_db = aws_db_instance.postgres.db_name
postgres_user = aws_db_instance.postgres.username
postgres_password = aws_db_instance.postgres.password
s3_bucket = aws_s3_bucket.s3bucket.bucket
})
}
```

```
#!/bin/bash
sudo apt update -y
sudo apt upgrade -y
sudo apt install docker.io -y
sudo docker run -d -p 8080:80 \
--name NextCloudContainer  \
-e POSTGRES_HOST=${postgres_host} \
-e POSTGRES_DB=${postgres_db} \
-e POSTGRES_USER=${postgres_user} \
-e POSTGRES_PASSWORD=${postgres_password} \
-e OBJECTSTORE_S3_HOST=s3.eu-north-1.amazonaws.com \
-e OBJECTSTORE_S3_BUCKET=${s3_bucket} \
-e OBJECTSTORE_S3_KEY=${access_key} \
-e OBJECTSTORE_S3_SECRET=${secret_key} \
-e OBJECTSTORE_S3_REGION=eu-north-1 \
nextcloud
```

Zunächst haben wir uns darauf konzentriert, eine einzelne Nextcloud-Instanz zum Laufen zu bringen. Nach einigen Versuchen gelang dies erfolgreich. Innerhalb der EC2-Instanz verwenden wir ein Docker-Image von Nextcloud, das mit bestimmten Umgebungsvariablen konfiguriert ist, um den initialen Setup zu vereinfachen.

Dieser Initiierungsschritt erfordert jedoch erhebliche manuelle Arbeit. Es ist notwendig, die Instanz zuerst zu starten, das Nextcloud-Setup durchzuführen, die Konfigurationsdatei aus dem Container zu extrahieren und sie auf einem separaten Bucket oder FileShare-Dienst hochzuladen. Dies ermöglicht es, die Konfigurationsdatei schnell per Curl bei der Neuinstanzierung der Autoscaling-Gruppe zu verwenden.

Während des Nextcloud-Setups werden alle Datenbankeinträge sowie die Konfiguration für den S3-Bucket und die RDS-Datenbank erstellt. Bedauerlicherweise konnten wir während unserer Recherche keine Methode finden, um diesen Schritt zu automatisieren. Daher erfordert unser Skript zunächst die Erstellung der einzelnen EC2-Instanz, das Abschliessen des Nextcloud-Setups und anschliessend das Kopieren der Konfigurationsdatei auf einen Bucket. Das Autoscaling-Skript kann dann diese Konfigurationsdatei für jede neue Instanz einbinden.

Die erstellte Konfigurationsdatei kann mithilfe des folgenden Befehls kopiert werden:

docker cp NextCloudContainer:/var/www/html/config/config.php ~/Desktop/

Ein Ausschnitt aus unserem Bash-Skript für eine neue Instanz ist wie folgt:

```
#!/bin/bash
  sudo apt update -y
  sudo apt upgrade -y
  sudo apt install docker.io -y
  sudo docker run -d -p 80:80 \
       --name NextCloudContainer  \
       nextcloud
  sleep 20
  sudo docker exec NextCloudContainer bash -c "apt update && apt install curl -y"
  sudo docker exec NextCloudContainer bash -c "curl 'https://onlyforconfig.s3.eu-north-
  sudo docker exec NextCloudContainer  rm /var/www/html/config/CAN_INSTALL
  sudo docker exec NextCloudContainer  chown -R 33:33 /var/
  sudo docker exec NextCloudContainer  touch /var/www/html/data/.ocdata
```

Dieser Abschnitt des Skripts installiert erforderliche Abhängigkeiten, lädt die Konfigurationsdatei von einem S3-Bucket herunter und führt anschliessend Anpassungen an den Dateien in der Nextcloud-Instanz durch, um den automatischen Einrichtungsprozess abzuschliessen. Dieser Schritt hat den grössten Zeitbedarf erfordert. Die Dokumentation von Nextcloud selbst war wenig hilfreich. Erst nach umfangreichen Recherchen in Foren und auf Reddit stiessen wir auf eine Methode, wie wir die Automatisierung für eine neue Instanz realisieren können.

Die zuvor kopierte **config.php**-Datei kann nun mittels eines Curl-Befehls von einem beliebigen File-Hoster abgerufen werden. Hierfür ist eine Anpassung des zweiten "docker exec"-Befehls erforderlich. Ich verwende dazu einen manuell erstellten S3 Bucket der öffentlich zugänglich ist.

Es ist notwendig, die Datei "CAN_INSTALL" zu entfernen, um anzuzeigen, dass das initiale Setup abgeschlossen wurde. Anschliessend wird dem Benutzer, auf dem Nextcloud läuft, die Berechtigung für das Verzeichnis "var" zugewiesen. Abschliessend benötigt Nextcloud noch eine vorhandene ".ocdata"-Datei.

Mit diesen Schritten ist das Skript zur Erstellung neuer Instanzen anhand einer Konfigurationsdatei funktionsfähig.

4.5 Load Balancer

```
resource "aws_lb" "pcls-loadbalancer" {
  name               = "pcls-loadbalancer"
  internal           = false
  load_balancer_type = "application"
  security_groups    = [aws_security_group.loadbalancer-securitygroups.id]
  subnets            = [aws_subnet.pub-sub-1.id, aws_subnet.pub-sub-2.id]
  depends_on         = [aws_internet_gateway.gw]
}

resource "aws_lb_target_group" "pcls-targetgroup" {
  name     = "pcls-targetgroup"
  port     = 80
  protocol = "HTTP"
  vpc_id   = aws_vpc.myvpc.id
  stickiness {
    type            = "lb_cookie"
    cookie_duration = 86400
    enabled = true
  }
}

resource "aws_lb_listener" "pcls-listener" {
  load_balancer_arn = aws_lb.pcls-loadbalancer.arn
  port              = "80"
  protocol          = "HTTP"
  default_action {
    type             = "forward"
    target_group_arn = aws_lb_target_group.pcls-targetgroup.arn
  }
}
```

Der erste Abschnitt des Codes widmet sich der Erstellung des Load Balancers. Die Einstellung Internal ist dabei auf false gesetzt, was bedeutet, dass der Load Balancer von aussen erreichbar sein muss. Der Loadbalancer wird den zwei öffentlichen Subnetzen zugeordnet.

Im zweiten Abschnitt wird eine Target Group erstellt, die sicherstellt, dass die EC2-Instanzen auf den Port 80 lauschen. Sticky Sessions garantieren die Verwaltung anhand von Cookies. Dieser Schritt gewährleistet, dass bei einem Browser-Refresh des Endbenutzers die Verbindung durch den Load Balancer nicht auf eine andere Instanz umgeleitet wird. Dies ist besonders wichtig, um eine konsistente Benutzererfahrung sicherzustellen.

Im letzten Abschnitt wird ein Listener erstellt, der auf Seiten des Internet Gateway auf Anfragen an Port 80 wartet. Die Default Action gibt an, dass Anfragen an die zuvor definierte Target Group weitergeleitet werden sollen.

4.6 Auto Scaling

```
# Launch template for ec2
resource "aws_launch_template" "ec2-template" {
  image_id      = "ami-0014ce3e52359afbd"
  instance_type = "t3.micro"
  user_data     = filebase64("NewInstance.sh")

  network_interfaces {
    associate_public_ip_address = false
    subnet_id                   = aws_subnet.priv-sub.id
    security_groups             = [aws_security_group.ec2-securitygroups.id]
  }

}

# Autoscaling group, 1-3 instances, created in private subnet
resource "aws_autoscaling_group" "pcls-autoscaling" {
  desired_capacity = 2
  max_size         = 4
  min_size         = 1

  target_group_arns = [aws_lb_target_group.pcls-targetgroup.arn]

  vpc_zone_identifier = [aws_subnet.priv-sub.id]

  launch_template {
    id      = aws_launch_template.ec2-template.id
    version = "$Latest"
  }
}
```

Im ersten Abschnitt des Codes wird ein Template für EC2-Instanzen definiert. Unter anderem beinhaltet dies die Auswahl des AMI (Amazon Machine Image) und des Instanztyps. Zusätzlich wird ein Bash-Skript namens NewInstance.sh bereitgestellt, das bei der Erstellung jeder Instanz ausgeführt wird.

Im zweiten Abschnitt wird die Auto Scaling-Gruppe definiert. Das Hauptziel besteht darin, sicherzustellen, dass zu jedem Zeitpunkt mindestens eine und maximal drei Instanzen im privaten Subnetz verfügbar sind. Die zuvor erstellte Target Group, die mit dem Load Balancer verbunden ist, wird nun mit dieser Auto Scaling-Gruppe verknüpft. Die Auto Scaling-Gruppe verwendet das im ersten Abschnitt definierte Template, um EC2-Instanzen zu erstellen.

4.7 Virtual Private Cloud

```
# create aws vpc
resource "aws_vpc" "myvpc" {
  cidr_block = "10.0.0.0/16"
  enable_dns_hostnames = true
  enable_dns_support = true
}
```

Dieser Code beschreibt ein VPC mit einem IP-Adressbereich von 10.0.0.0 bis 10.0.255.255, aktiviert die Unterstützung für DNS-Hostnamen und stellt sicher, dass DNS-Support innerhalb des VPC aktiviert ist.

```
# creating 1st public subnet
resource "aws_subnet" "pub-sub-1" {
  vpc_id                  = aws_vpc.myvpc.id
  cidr_block              = "10.0.1.0/24"
  map_public_ip_on_launch = true
  availability_zone       = "eu-north-1a"
}

# creating 2nd public subnet
resource "aws_subnet" "pub-sub-2" {
  vpc_id                  = aws_vpc.myvpc.id
  cidr_block              = "10.0.2.0/24"
  map_public_ip_on_launch = true
  availability_zone       = "eu-north-1b"
}

# creating private subnet
resource "aws_subnet" "priv-sub" {
  vpc_id                  = aws_vpc.myvpc.id
  cidr_block              = "10.0.3.0/24"
  map_public_ip_on_launch = false
  availability_zone       = "eu-north-1b"
}
```

Dieser Code erstellt zwei öffentliche Subnetze (eines in jeder der beiden Availability Zonen) und ein privates Subnetz innerhalb der definierten VPC. Die öffentlichen Subnetze werden für Ressourcen wie Load Balancer und das private Subnetz für Backend-Server verwendet.

```
# route table for public subnet
resource "aws_route_table" "pub-route" {
  vpc_id = aws_vpc.myvpc.id

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.gw.id
  }
}

# route table for private subnet
resource "aws_route_table" "priv-route" {
  vpc_id = aws_vpc.myvpc.id

  route {
    cidr_block     = "0.0.0.0/0"
    nat_gateway_id = aws_nat_gateway.nat-priv-subnet.id
  }
}
```

Dieser Code erstellt zwei Routentabellen, eine für das öffentliche und eine für das private Subnetz innerhalb der VPC. Die öffentliche Routentabelle leitet den Verkehr über das Internet Gateway, während die private Routentabelle den Verkehr über ein NAT Gateway lenkt. Der CIDR-Block stellt die Default Route dar, die jede IP-Adresse verwendet.

5 Erkenntnisse und Fazit

Die Durchführung dieser Case Study gewährte uns einen faszinierenden Einblick in die Welt von AWS sowie die Anwendung von Terraform als mächtigem Werkzeug zur Ressourcenverwaltung. Terraform erwies sich als äusserst benutzerfreundlich und ermöglichte uns, AWS-Ressourcen auf eine klare und unkomplizierte Weise zu erstellen.

Unsere Hauptherausforderung bestand darin, die Funktionalität von Nextcloud nahtlos in unsere AWS-Umgebung zu integrieren. Trotz der vergleichsweisen unkomplizierten Handhabung von Terraform war es notwendig, beträchtliche Zeit in die Feinabstimmung der Nextcloud-Instanz zu investieren. Insbesondere die Erstellung neuer Instanzen mit einer vorhandenen Datenbank und bereits vorhandenen Daten gestaltete sich als anspruchsvolle Aufgabe.

Die Automatisierung des Setups gestaltete sich aufgrund der spezifischen Anforderungen von Nextcloud als herausfordernd. Das Fehlen einer umfassenden Methode zur Automatisierung bestimmter Schritte erforderte eine manuelle Intervention, um sicherzustellen, dass die neuen Instanzen korrekt konfiguriert und mit den erforderlichen Datenbank- und S3-Konfigurationen versehen wurden.

Trotz dieser Herausforderungen haben wir wertvolle Einsichten in die Prinzipien der Cloud-Architektur und deren Umsetzung gewonnen. Die Kombination aus AWS und Terraform bietet eine solide Grundlage für die Implementierung skalierbarer und kosteneffizienter Cloud-Infrastrukturen. Wir schätzen die Möglichkeit, praktische Erfahrungen zu sammeln und sind zuversichtlich, dass die gewonnenen Erkenntnisse einen Mehrwert für zukünftige Projekte darstellen werden.

Insgesamt war diese Case Study nicht nur lehrreich, sondern auch ein bedeutender Schritt in unserer Reise, Cloud-Technologien erfolgreich zu nutzen und in zukünftigen Projekten effektiv einzusetzen

Ursprünglich war geplant, den Load Balancer mithilfe von AWS Route 53 hinter eine eigene Domain zu integrieren. Jedoch traten Probleme im zugewiesenen Sandbox-Account auf, da uns verschiedene Berechtigungen für S3 und andere Dienste fehlten. Aufgrund dieser Einschränkungen haben wir das gesamte Projekt mit unseren persönlichen Free Tier-Accounts umgesetzt. Die Nutzung von Route 53 für eine eigene Domain erfordert mindestens 11 Franken, was jedoch nicht in unserem Budget vorgesehen war.

5.1 Mögliche zukünftige Anpassungen

- Implementierung eines gemeinsam genutzten Redis-Cache für alle Instanzen, um Zugriffssperren zu vermeiden.
- Automatisierung der Konfiguration der Instanzen.
- Integration von AWS CloudWatch für erweitertes Monitoring