Raport projekt 1

Zaawansowane rozwiązania chmurowe

01.12.2025

Dominika Kołowrotkiewicz

Spis treści

1 Repozytorum

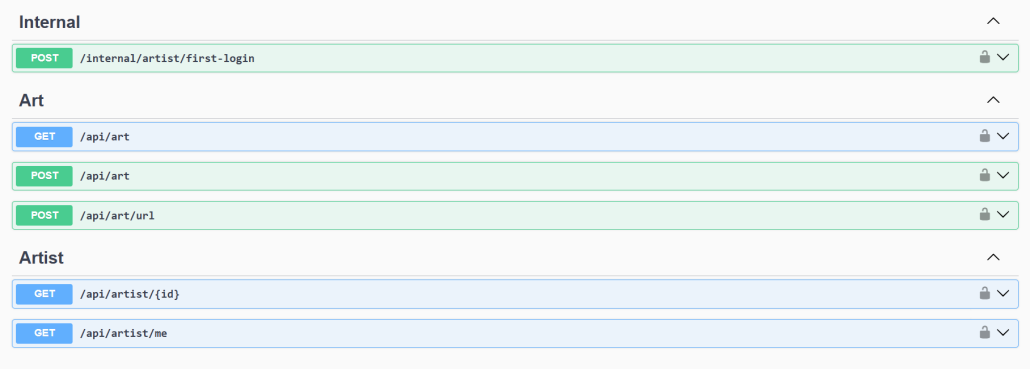
Cały kod dostępny jest w repozytorium na githubie: ArtGallery .

2 Opis aplikacji

Aplikacja ArtGallery pozwala na przeglądanie dodanych oraz dodawanie obrazów. Część funkcjonalności wymaga założenia konta i autoryzacji. Do stworzenia backendu użyto **Spring Boot 3** a dla frontendu **Vite/React**.

2.1 Endpointy

Aplikacja korzysta ze swaggera do przeglądu dostępnych endpointów. Endpointy podzielone są na 3 sekcje: *Art*, *Artist* i *Internal*. Pierwsze 2 dostępne są z frontendu (niektóre wymagają autoryzacji), a endpoint z kategorii *Ineternal* wykorzystywany jest przez Lambdę.



2.2 Konfiguracja dockera

Obrazy obu części aplikacji budowane są za pomocą plików Dockerfile lokalnie (ale proces jest zautomatyzowany w terraform).

3 Konfiguracja VPC

Tworzona jest infrastruktura sieciowa AWS składająca się z VPC, publicznych i prywatnych subnetów oraz odpowiednich tablic routingu. Powstaje VPC z włączonym DNS. Do VPC podłączany jest Internet Gateway, który umożliwia dostęp do internetu dla zasobów w publicznych subnetach.

Publiczne i prywatne subnety tworzone są w pętli na podstawie przekazanych list CIDR i AZ. Publiczne subnety mają włączone automatyczne nadawanie publicznych IP, a prywatne nie. Do publicznych subnetów przypisywana jest tablica routingu z trasą 0.0.0.0/0 kierującą ruch przez IGW, dzięki czemu mają one dostęp do internetu. Prywatne subnety mają własną tablicę routingu bez dostępu do internetu.

3.1 Kod

3.1.1 VPC

```
resource "aws_vpc" "vpc" {
  cidr_block           = var.vpc_cidr
  enable_dns_support   = true
  enable_dns_hostnames = true
  tags                 = local.common_tags
```

```
}
```

3.1.2 IGW

```
resource "aws_internet_gateway" "igw" {
  vpc_id = aws_vpc.vpc.id
  tags   = merge(local.common_tags, { Component = "igw" })
}
```

3.1.3 Subnety

```
resource "aws_subnet" "public" {
  for_each = { for idx, cidr in var.public_subnet_cidrs : idx => { cidr = cidr, az = var.azs[idx] } }

  vpc_id                  = aws_vpc.vpc.id
  cidr_block              = each.value.cidr
  availability_zone       = each.value.az
  map_public_ip_on_launch = true

  tags = merge(local.common_tags, {
    Component = "public-subnet"
    AZ        = each.value.az
  })
}

resource "aws_subnet" "private" {
  for_each = { for idx, cidr in var.private_subnet_cidrs : idx => { cidr = cidr, az = var.azs[idx] } }

  vpc_id            = aws_vpc.vpc.id
  cidr_block        = each.value.cidr
  availability_zone = each.value.az

  tags = merge(local.common_tags, {
    Component = "private-subnet"
    AZ        = each.value.az
  })
}
```

3.1.4 Tablica publiczne

```
resource "aws_route_table" "public" {
  vpc_id = aws_vpc.vpc.id
  tags   = merge(local.common_tags, { Component = "public-rt" })
}

resource "aws_route" "public_0_default" {
  route_table_id         = aws_route_table.public.id
  destination_cidr_block = "0.0.0.0/0"
  gateway_id             = aws_internet_gateway.igw.id
}

resource "aws_route_table_association" "public_assoc" {
```

```
  for_each = aws_subnet.public
  subnet_id      = each.value.id
  route_table_id = aws_route_table.public.id
}
```

3.1.5 Tablica prywatna

```
resource "aws_route_table" "private" {
  vpc_id = aws_vpc.vpc.id
  tags   = merge(local.common_tags, { Component = "private-rt" })
}

resource "aws_route_table_association" "private_assoc" {
  for_each = aws_subnet.private
  subnet_id      = each.value.id
  route_table_id = aws_route_table.private.id
}
```

4 Konfiguracja Load Balancer i Security Groups

W projekcie zastosowano Application Load Balancer (ALB), umieszczony w publicznych podsieciach VPC, dzięki czemu może przyjmować ruch z Internetu i kierować go do usług ECS działających w prywatnych podsieciach.

ALB posiada własną grupę bezpieczeństwa, która pozwala na ruch HTTP na porcie 80 z dowolnego źródła. Komunikacja z kontenerami odbywa się wyłącznie wewnątrz VPC (kontenery mają przypisane prywatne adresy IP i nie otrzymują publicznych IP).

Dla warstwy aplikacyjnej zdefiniowano dwa target groupy (dla frontendu i dla backendu). Każdy target group wykorzystuje health checki HTTP z krótkimi interwałami, aby szybko wykrywać niedostępne zadania Fargate i usuwać je z ruchu.

Listener na porcie 80 domyślnie przekazuje ruch do frontendu, a dodatkowa reguła kieruje żądania pasujące do określonego wzorca ścieżki (np. /api/*) do backendu.

Grupy bezpieczeństwa kontrolują komunikację z ECS. Frontend i backend mają oddzielne security groups, które zezwalają wyłącznie na ruch przychodzący z security group ALB (odpowiednio na port 80 lub 8080). Wychodzące połączenia z kontenerów są dozwolone dla całego Internetu, co umożliwia im pobieranie zależności czy komunikację z usługami AWS.

4.1 Kod

4.1.1 Load balancer

```
resource "aws_lb" "app" {
  name               = var.lb_name
  load_balancer_type = "application"
  security_groups = [aws_security_group.alb.id]
  subnets            = var.public_subnet_ids
}
```

4.1.2 Security Group Load balancera

```
resource "aws_security_group" "alb" {
  name   = "${var.lb_name}-sg"
```

```
  vpc_id = var.vpc_id

  ingress {
    from_port = 80
    to_port   = 80
    protocol  = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  egress {
    from_port = 0
    to_port   = 0
    protocol  = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

4.1.3 Target groups

```
resource "aws_lb_target_group" "fe" {
  name        = var.fe_tg_name
  port        = var.fe_port
  protocol    = "HTTP"
  vpc_id      = var.vpc_id
  target_type = "ip"
  health_check {
    path    = var.fe_health_path
    matcher = "200-399"
    interval            = 10
    timeout             = 5
    healthy_threshold   = 2
    unhealthy_threshold = 2
  }
}

resource "aws_lb_target_group" "be" {
  name        = var.be_tg_name
  port        = var.be_port
  protocol    = "HTTP"
  vpc_id      = var.vpc_id
  target_type = "ip"
  health_check {
    path    = var.be_health_path
    matcher = "200-399"
    interval            = 10
    timeout             = 5
    healthy_threshold   = 2
    unhealthy_threshold = 2
  }
}
```

4.1.4 Listener i zasada przekierowywania

```
resource "aws_lb_listener" "http" {
  load_balancer_arn = aws_lb.app.arn
  port              = 80
  protocol          = "HTTP"

  default_action {
    type             = "forward"
    target_group_arn = aws_lb_target_group.fe.arn
  }
}

resource "aws_lb_listener_rule" "api" {
  listener_arn = aws_lb_listener.http.arn
  priority     = 10

  action {
    type             = "forward"
    target_group_arn = aws_lb_target_group.be.arn
  }

  condition {
    path_pattern {
      values = var.api_path_pattern
    }
  }
}
```

4.1.5 Security Group Load balancera

```
resource "aws_security_group" "frontend" {
  vpc_id = module.vpc.vpc_id
  ingress {
    from_port = 80
    to_port   = 80
    protocol  = "tcp"
    security_groups = [module.alb.alb_sg_id]
  }
  egress {
    from_port = 0
    to_port   = 0
    protocol  = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_security_group" "backend" {
  vpc_id = module.vpc.vpc_id
  ingress {
    from_port = 8080
    to_port   = 8080
```

```
    protocol  = "tcp"
    security_groups = [module.alb.alb_sg_id]
  }
  egress {
    from_port = 0
    to_port   = 0
    protocol  = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

5 Konfiguracja AWS Cognito

User Pool

Najważniejsze elementy konfiguracji User Pool to:

- automatyczna weryfikacja adresu e-mail,
- wysyłanie kodu potwierdzającego przy rejestracji (CONFIRM_WITH_CODE),
- wymaganie atrybutu `email` prze tworzeniu konta,
- polityka złożoności hasła definiowana zmiennymi,
- konfiguracja odzyskiwania konta przez zweryfikowany e-mail (ostatecznie nie zaimplementowana w aplikacji ale gotowa do użycia).

User Pool Client

Najistotniejsze cechy konfiguracji klienta to:

- ustawienie czasów ważności tokenów (w godzinach),
- ukrywanie informacji o istnieniu użytkownika przy wystąpieniu błędów,
- dozwolone flowy logowania:
  - `ALLOW_USER_SRP_AUTH` — bezpieczne logowanie SRP,
  - `ALLOW_USER_PASSWORD_AUTH` — logowanie za pomocą loginu i hasła,
  - `ALLOW_REFRESH_TOKEN_AUTH` — odświeżanie tokenów bez ponownego logowania.

5.1 Kod

5.1.1 UserPool

```
resource "aws_cognito_user_pool" "pool" {
  name = var.name

  email_configuration {
    email_sending_account = "COGNITO_DEFAULT"
  }

  auto_verified_attributes = var.auto_verified_attributes
```

```
  dynamic "verification_message_template" {
    for_each = var.enable_email_link_confirm ? [1] : []
    content {
      default_email_option = "CONFIRM_WITH_CODE"
    }
  }

  schema {
    attribute_data_type      = "String"
    name                     = "email"
    developer_only_attribute = false
    mutable                  = true
    required                 = true
  }

  password_policy {
    minimum_length    = var.password_policy.minimum_length
    require_uppercase = var.password_policy.require_uppercase
    require_lowercase = var.password_policy.require_lowercase
    require_numbers   = var.password_policy.require_numbers
    require_symbols   = var.password_policy.require_symbols
  }

  account_recovery_setting {
    recovery_mechanism {
      name     = "verified_email"
      priority = 1
    }
  }

  lambda_config {
    post_confirmation = var.post_confirmation_lambda_arn
  }
}
```

5.1.2 PoolClient

```
resource "aws_cognito_user_pool_client" "client" {
  name         = var.app_client_name
  user_pool_id = aws_cognito_user_pool.pool.id

  generate_secret               = false
  prevent_user_existence_errors = "ENABLED"

  explicit_auth_flows = [
    "ALLOW_USER_SRP_AUTH",
    "ALLOW_USER_PASSWORD_AUTH",
    "ALLOW_REFRESH_TOKEN_AUTH"
  ]
```

```
  access_token_validity        = var.app_client_times.access_token_validity_hours
  id_token_validity            = var.app_client_times.id_token_validity_hours
  refresh_token_validity       = var.app_client_times.refresh_token_validity_hours

  token_validity_units {
    access_token  = "hours"
    id_token      = "hours"
    refresh_token = "hours"
  }
}
```

6 Konfiguracja S3

W konfiguracji Terraform tworzony jest bucket S3, w którym włączona jest wersjonizacja obiektów oraz szyfrowanie po stronie serwera przy użyciu algorytmu AES–256. Dodana jest również **konfiguracja CORS**, która umożliwia wykonywanie zapytań z mojej aplikacji.

Ważnym aspektem zabezpieczenia S3 są **ustawiena dostępu**. W mojej konfiguracji dostęp **GET jest publiczny** co zezwala na szybsze ładowanie obrazów na frontendzie. Natomiast **PUT i POST są dozwolone tylko z mojego backendu**, ponieważ wymagają uprawnień IAM mimo dopuszczenia ich w CORS. Sam upload obrazu wykonuje się po stronie frontendu przez wygenerowane przez backend **presigned URL**, co pozwala na zmniejszenie czasu wykonania operacji.

6.1 Kod

6.1.1 Bucket

```
resource "aws_s3_bucket" "art_bucket" {
  bucket = var.bucket_name
  force_destroy = true
}
```

6.1.2 Wersjonowanie i enkrypcja

```
resource "aws_s3_bucket_versioning" "versioning" {
  bucket = aws_s3_bucket.art_bucket.id

  versioning_configuration {
    status = "Enabled"
  }
}

resource "aws_s3_bucket_server_side_encryption_configuration" "encryption" {
  bucket = aws_s3_bucket.art_bucket.bucket

  rule {
    apply_server_side_encryption_by_default {
      sse_algorithm = "AES256"
    }
  }
}
```

6.1.3 Dostępność

```
resource "aws_s3_bucket_public_access_block" "public_access" {
  bucket = aws_s3_bucket.art_bucket.id

  block_public_acls       = false
  ignore_public_acls      = false
  block_public_policy     = false
  restrict_public_buckets = false
}

resource "aws_s3_bucket_cors_configuration" "cors" {
  bucket = aws_s3_bucket.art_bucket.id

  cors_rule {
    allowed_headers = ["*"]
    allowed_methods = ["PUT", "POST", "GET"]
    allowed_origins = [ "http://${var.alb_dns}"]
    expose_headers  = ["ETag"]
    max_age_seconds = 3000
  }
}


resource "aws_s3_bucket_policy" "public_read" {
  bucket = aws_s3_bucket.art_bucket.id

  policy = jsonencode({
    Version = "2012-10-17",
    Statement = [{
      Sid       = "PublicReadGetObject",
      Effect    = "Allow",
      Principal = "*",
      Action    = "s3:GetObject",
      Resource  = "${aws_s3_bucket.art_bucket.arn}/*"
    }]
  })

  depends_on = [aws_s3_bucket_public_access_block.public_access]
}
```

7 Konfiguracja AWS RDS

W konfiguracji tworzona jest instancja bazy **PostgreSQL** wraz z pełnym zestawem zasobów potrzebnych do bezpiecznego działania w prywatnej sieci VPC.

Hasło i dane logowania

Hasło do bazy jest generowane losowo przy użyciu random_password i ma 24 znaki, w tym znaki specjalne. Login i hasło są przechowywane w AWS Secrets Manager w zaszyfrowanej postaci. Instancja RDS pobiera hasło bezpośrednio z Secret Managera.

Sieć

Tworzony jest db_subnet_group, umieszczający bazę w prywatnych subnetach. Dostęp do bazy jest tylko pzez port 5432 dla wskazanych security groups.

Instancja bazy

Tworzona jest instancja PostgreSQL. Włączono szyfrowanie dysku (storage_encrypted = true). Backupy zachowywane są przez 1 dzień. Dodatkowo pominięto tworzenie snapshotu przy usunięciu bazy.

7.1 Kod

7.1.1 Hasło i dane logowania

```
resource "aws_secretsmanager_secret" "db_credentials" {
  name = "${var.db_name}-db-credentials"
  recovery_window_in_days  = 0
  tags = var.tags
}

resource "random_password" "db" {
  length           = 24
  special          = true
  override_special = "!#$%^&*()-_=+[]{}<>?:.,;"
  min_lower        = 1
  min_upper        = 1
  min_numeric      = 1
  min_special      = 1
}

resource "aws_secretsmanager_secret_version" "db_credentials" {
  secret_id = aws_secretsmanager_secret.db_credentials.id

  secret_string = jsonencode({
    username = var.username
    password = random_password.db.result
  })
}
```

7.1.2 Sieć

```
resource "aws_db_subnet_group" "subnet_group" {
  name       = "${var.db_name}-subnets"
  subnet_ids = var.private_subnet_ids
  tags       = var.tags
}

resource "aws_security_group" "rds" {
  name        = "${var.db_name}-rds-sg"
  description = "RDS PostgreSQL SG"
  vpc_id      = var.vpc_id
  tags        = var.tags
```

```
  ingress {
    from_port       = 5432
    to_port         = 5432
    protocol        = "tcp"
    security_groups = var.ingress_security_group_ids
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

7.1.3 Instancja bazy

```
resource "aws_db_instance" "postgres" {
  identifier              = "${var.db_name}-pg"
  engine                  = "postgres"
  engine_version          = var.engine_version
  instance_class          = var.instance_class
  db_name                 = var.db_name

  username                = var.username
  password = jsondecode(aws_secretsmanager_secret_version.db_credentials.secret_string)["password"]


  allocated_storage       = var.allocated_storage
  storage_encrypted       = true
  skip_final_snapshot     = true
  publicly_accessible     = false
  vpc_security_group_ids  = [aws_security_group.rds.id]
  db_subnet_group_name    = aws_db_subnet_group.subnet_group.name
  multi_az                = var.multi_az
  deletion_protection     = var.deletion_protection

  backup_retention_period = 1
  apply_immediately       = true

  tags = var.tags
}
```

8 Konfiguracja lambdy

Lambdy użyto do automatycznego tworzenia profili użytkownika w bazie postgres przy wykryciu potwierdzenia weryfikacji konta. Profil przechowuje dodatkowe dane użytkownika jak wyświetlana nazwa czy id przez przypisane są należące do użytkownika obrazy.

Folder **lambda.zip** zawiera wymagane biblioteki node oraz plik index.js, który wysyła zapytanie na backend do utworzenia profilu. Wykorzystano nagłówek **X-Internal-Secret** z wartością znaną tylko lambdzie i

backendowi z powodu braku tokenu do autoryzacji.

Aby lambda miała dostęp do zdarzeń cognito, utworzono zasób **aws_lambda_permission** z source_arn mojego user poola. Dodatkowo dodano lambda_config w user poolu podłączono arn lambdy.

8.1 Kod

8.1.1 Lambda

```
resource "aws_lambda_function" "lambda_function" {
  function_name = var.function_name
  handler       = var.handler
  runtime       = var.runtime
  role          = var.existing_role_arn
  filename      = var.filename

  environment {
    variables = var.environment
  }
}
```

8.1.2 Pozwolenie na dostęp do zdarzeń cognito

```
resource "aws_lambda_permission" "allow_post_confirmation" {
  statement_id  = "AllowExecutionFromCognito"
  action        = "lambda:InvokeFunction"
  function_name = var.post_confirmation_lambda_arn
  principal     = "cognito-idp.amazonaws.com"
  source_arn    = aws_cognito_user_pool.pool.arn
}
```

9 Konfiguracja ECR

Repozytoria

Konfiguracja tworzy **dwa repozytoria ECR** (dla frontendu i dla backendu). Oba z włączonym szyfrowaniem AES-256, skanowaniem obrazów przy przesyłaniu oraz możliwością modyfikacji tagów. Ustawiono również **politykę usuwania starszych obrazów**, która w każdym repozytorium zachowuje jedynie ostatnie 10 wersji. Repozytoria mają także włączoną opcję force_delete, co pozwala na ich usunięcie nawet wtedy, gdy zawierają obrazy.

Tworzenie obrazów

Stowrzono moduł do tworzenia i publikowania obrazów na wcześniej utworzonych repozytoriach. Podczas budowania generowana jest unikalna wersja obrazu oparta na znaczniku czasu, dzięki czemu każda wersja aplikacji trafia do ECR z nowym tagiem. Obraz budowany jest z podanego kontekstu i Dockerfile, po czym otrzymuje dodatkowy tag latest. Następnie oba tagi są wypychane do ECR przy użyciu zasobów docker_registry_image. Po wykonaniu push Terraform odczytuje digest obrazu z ECR, co pozwala wykorzystać go później np. w konfiguracji ECS.

9.1 Kod

9.1.1 Repozytoria

```
resource "aws_ecr_repository" "frontend" {
  name = "art-frontend"
  image_tag_mutability = "MUTABLE"
  encryption_configuration { encryption_type = "AES256" }
  image_scanning_configuration { scan_on_push = true }
  force_delete = true
  tags = { Project = "art-gallery", Component = "frontend" }
}

resource "aws_ecr_repository" "backend" {
  name = "art-backend"
  image_tag_mutability = "MUTABLE"
  encryption_configuration { encryption_type = "AES256" }
  image_scanning_configuration { scan_on_push = true }
  force_delete = true
  tags = { Project = "art-gallery", Component = "backend" }
}

resource "aws_ecr_lifecycle_policy" "frontend" {
  repository = aws_ecr_repository.frontend.name
  policy     = jsonencode({ rules = [{ rulePriority=1, description="Keep last 10 images",
    selection={ tagStatus="any", countType="imageCountMoreThan", countNumber=10 },
    action={ type="expire" } }]})
}

resource "aws_ecr_lifecycle_policy" "backend" {
  repository = aws_ecr_repository.backend.name
  policy     = aws_ecr_lifecycle_policy.frontend.policy
}
```

9.1.2 Tworzenie obrazów

```
locals {
  image_version = formatdate("YYYYMMDD-HHmmss", timestamp())
}

resource "docker_image" "image" {
  name = "${var.repo_url}:${local.image_version}"
  build {
    context    = var.path
    dockerfile = "Dockerfile"
    platform   = var.platform
  }
}

resource "docker_tag" "latest" {
  source_image = docker_image.image.name
  target_image = "${var.repo_url}:latest"
```

```
}

resource "docker_registry_image" "versioned" {
  name = docker_image.image.name
}

resource "docker_registry_image" "latest" {
  name       = docker_tag.latest.target_image
  depends_on = [docker_tag.latest]
}

data "aws_ecr_image" "digest" {
  repository_name = var.repo_name
  image_tag       = local.image_version
  depends_on      = [docker_registry_image.versioned]
}
```

10 Konfiguracja ECS

Kluster

Utworzono kluster ecs.

Definicja zadania

Dla frontendu i backendu utworzone zostały osobne definicje zadań ECS oparte na platformie Fargate. Każde zadanie uruchamia pojedynczy kontener, zbudowany wcześniej i zapisany w ECR. W definicji określono zasoby (CPU i pamięć), tryb sieciowy awsvpc, rolę wykonawczą do pobierania obrazów oraz rolę zadania nadającą kontenerom uprawnienia do usług AWS.

Frontend korzysta z obrazu aplikacji webowej, nasłuchuje na porcie 80 i otrzymuje zmienne środowiskowe wymagane do komunikacji z API i Cognito.

Backend uruchamia aplikację Spring Boot na porcie 8080, a jego konfiguracja obejmuje m.in. adres bazy danych, profil środowiskowy i punkt weryfikacji Cognito.

Dane wrażliwe, takie jak nazwa użytkownika i hasło do bazy, przekazywane są jako sekrety pobierane z AWS Secrets Manager.

Serwisy

Frontend i backend uruchamiane są jako dwa niezależne serwisy ECS oparte na Fargate, każdy korzystający z własnej definicji zadania. Serwisy działają w prywatnych podsieciach VPC, z przypisanymi odpowiednimi grupami bezpieczeństwa oraz bez nadawania publicznych adresów IP. Każdy serwis utrzymuje zadeklarowaną liczbę replik (desired count = 2), co zapewnia wysoką dostępność aplikacji.

Oba serwisy są zintegrowane z Application Load Balancerem: frontend z grupą docelową obsługującą port 80, a backend z grupą dla portu 8080. Dzięki temu ruch jest równomiernie rozkładany między działające zadania. Włączony jest również mechanizm „deployment circuit breaker", który automatycznie wycofuje nieudane wdrożenia. Konfiguracja wdrożeniowa dopuszcza stopniowe aktualizacje z odpowiednimi limitami zdrowych instancji, co minimalizuje ryzyko przerw w działaniu aplikacji.

10.1 Kod

10.1.1 Kluster

```
resource "aws_ecs_cluster" "cluster" {
  name = var.name
}
```

10.1.2 Defninicje zadań

```
resource "aws_ecs_task_definition" "task_definition" {
  family                   = var.family
  requires_compatibilities = ["FARGATE"]
  network_mode             = "awsvpc"
  cpu                      = var.cpu
  memory                   = var.memory
  execution_role_arn = var.execution_role_arn
  task_role_arn = var.task_role_arn

  runtime_platform {
    operating_system_family = "LINUX"
    cpu_architecture        = "X86_64"
  }

  container_definitions = jsonencode([
    merge(
      {
        name      = var.container_name
        image     = var.container_image_ref
        essential = true

        portMappings = [
          {
            containerPort = var.container_port
            protocol      = "tcp"
          }
        ]

        logConfiguration = {
          logDriver = "awslogs"
          options = {
            awslogs-group         = var.log_group_name
            awslogs-region        = var.aws_region
            awslogs-stream-prefix = var.container_name
          }
        }
      },

        length(var.environment) > 0 ? {
        environment = var.environment
      } : {},
```

```
        length(var.secrets) > 0 ? {
        secrets = [
          for s in var.secrets :
          { name = s.name, valueFrom = s.value_from }
        ]
      } : {}
    )
  ])
}
```

10.1.3 Serwisy

```
resource "aws_ecs_service" "ecs_service" {
  name            = var.name
  cluster         = var.cluster_id
  task_definition = var.task_definition
  desired_count   = var.desired_count
  launch_type     = "FARGATE"
  health_check_grace_period_seconds  = 90
  deployment_minimum_healthy_percent = 50
  deployment_maximum_percent         = 200

  deployment_circuit_breaker {
    enable   = true
    rollback = true
  }

  network_configuration {
    subnets          = var.subnets
    security_groups  = var.security_groups
    assign_public_ip = var.assign_public_ip
  }

  dynamic "load_balancer" {
    for_each = var.load_balancers
    content {
      target_group_arn = load_balancer.value.target_group_arn
      container_name   = load_balancer.value.container_name
      container_port   = load_balancer.value.container_port
    }
  }

  lifecycle { ignore_changes = [desired_count] }
}
```

11 Konfiguracja CloudWatch

Dla frontendu i backendu utworzono aws_cloudwatch_log_group, które przekazane są i podpięte w definicjach zadania ECS.

11.1 Kod

11.1.1 Logi

```
resource "aws_cloudwatch_log_group" "logs" {
  name              = var.name
  retention_in_days = var.retention_in_days
  tags              = var.tags
}
```