УДК 004.75:004.4’13
DOI https://doi.org/10.32782/2663-5941/2025.2.2/03

Бегліцов С. В.
Державний університет інформаційно-комунікаційних технологій

ПЛАВНЕ ПЕРЕКЛЮЧЕННЯ В ХМАРНЕ СЕРЕДОВИЩЕ З АВТОМАТИЗАЦІЄЮ TERRAFORM І КОНТЕЙНЕРИЗАЦІЄЮ DOCKER: ІНТЕГРАЦІЯ ІНФРАСТРУКТУРИ ЯК КОД ТА BLUE-GREEN РОЗГОРТАННЯ

Статтю присвячено дослідженню міграції в хмарне середовище за допомогою Infrastructure as Code (IaC) із фокусом на ключових атрибутах хмарних обчислень. У роботі розкрито особливості традиційних архітектур, що вимагають ручного виділення ресурсів, та визначено переваги хмарних технологій, які забезпечують гнучкість завдяки самообслуговуванню на вимогу, широкому мережевому доступу, спільному пулу ресурсів, швидкій еластичності та вимірюваним послугам.

Розглянуто основні моделі хмарної інфраструктури: SaaS, PaaS та IaaS. З’ясовано, що для автоматизації найбільш актуальною є модель IaaS, яка забезпечує детальний контроль над віртуальними машинами та сховищем. PaaS спрощує розробку завдяки попередньо налаштованим середовищам, тоді як SaaS дозволяє організаціям використовувати керовані програмні рішення.

Доведено ефективність використання IaC для оркестрації хмарної інфраструктури шляхом визначення її через конфігураційні файли, що зберігаються у системах контролю версій, проходять тестування та автоматично розгортаються. Визначено роль Terraform, який, використовуючи HashiCorp Configuration Language (HCL), забезпечує автоматизоване виділення ресурсів.

Розглянуто стратегії міграції, що варіюються від простої заміни компонентів до повної реконфігурації застосунків (Cloudify). Показано, що IaC мінімізує ручне втручання та ризики помилок. Описано декларативний синтаксис Terraform та його інтеграцію з AWS, Azure та Google Cloud для гнучкого розгортання.

Окрему увагу приділено контейнеризованим застосункам, які отримують переваги від Docker. Визначено, що Docker пакує залежності в портативні контейнери, а Docker Compose керує конфігурацією багатоконтейнерних середовищ, оптимізуючи PHP-застосунки через Redis, Memcached та Opcache. Розглянуто можливість інтеграції інструментів налагодження, таких як xDebug та SOAP.

З’ясовано, що стратегія blue-green розгортання забезпечує плавне оновлення, підтримуючи два ідентичні середовища продакшену та перемикаючись між ними з мінімальним простоєм. Підтверджено, що Terraform ефективно забезпечує розгортання цих середовищ, тоді як Docker прискорює налаштування сервісів і масштабування.

Доведено, що Terraform автоматизує взаємодію з API хмарних платформ, інтегруючи AWS Lambda та API Gateway у CI/CD пайплайни. Визначено, що Terraform Cloud додатково оптимізує керування станом та спрощує командну роботу. У результаті дослідження підтверджено, що Terraform і Docker спрощують міграцію в хмару, автоматизуючи виділення інфраструктури, покращуючи ефективність розгортання та знижуючи операційні ризики.

Ключові слова: *декларативний синтаксис, автоматизована оркестрація, пайплан, контейнер, версії розгортання, дебагінг.*

Постановка проблеми. Концепція хмарних обчислень асоціюється з сучасною та масштабованою доставкою програмного забезпечення, що диктується необхідністю динамічного виділення ресурсів за запитом, спрощених операцій та вищої надійності. Перехід наявних локальних додатків у хмарну інфраструктуру вимагає значного планування задля забезпечення мінімального часу простою, максимальної ефективності та можливості повторення усієї операції у інших контекстах. Принцип інфраструктури як коду (Infrastructure as Code – IaC) з’явився в якості ефективної методології у забезпеченні зазначених потреб, особливо за сумісництва з технологіями контейнеризації, як наприклад Docker. Паралельно з цим, використання протестованих механізмів хешування у виді Redis, Memcached, та PHP Opcache сприяє продуктивності додатків після їхнього розгортання. При детальному розгляданні цих аспектів, стає зрозумілим необхідність застосування автоматизаційних технологій, в якості якої у даному дослідженні впроваджується Terraform. Загалом, опи-

сана стратегія відіграватиме основу для переходу з наземної інфраструктури, де відбуватиметься тестування контейнеризованих сервісів, у хмарну.

Аналіз останніх досліджень і публікацій. Сучасні наукові дослідження присвячені автоматизації процесів перенесення сервісів наземної інфраструктури в хмарне середовище, що є актуальним завданням у сфері інформаційних технологій.

Одним із важливих аспектів цієї тематики є розробка методів встановлення необхідних розширень для PHP у контейнерних середовищах без використання PECL, яке було вимкнено у версії PHP 7.4. Так, запропонована О. Лавале методологія передбачає застосування інструментів docker-php-ext-configure та docker-php-ext-install для підтримки чистоти та модульності Docker-образів відповідно до специфічних вимог проєкту. Також розглядаються труднощі, пов'язані з ручним встановленням розширень, зокрема таких, що залежать від підмодулів і багатостадійного збирання, як-от MongoDB. У результаті дослідження продемонстровано практичні підходи до створення ефективного Docker-середовища для PHP, що забезпечує оптимізацію кешування, серіалізації та інтеграцію поширених розширень [1].

Аспектами автоматизації управління хмарними ресурсами займалися такі дослідники як Б. Кішіяма, Х. Ванг, Д. Лопес, Дж. Янг [2]. Вони проаналізували ефективність інструментів Google Cloud Deployment Manager і Terraform у розгортанні інфраструктури. Здійснений порівняльний аналіз дозволив встановити, що Terraform забезпечує стабільно швидший час деактивації ресурсів, що підтверджується тестуванням із використанням команди Linux time. Google Cloud Deployment Manager, у свою чергу, демонструє вищу нативну інтеграцію з ресурсами Google Cloud, тоді як Terraform надає більшу універсальність, що робить його доцільним для мультихмарних середовищ. У роботі також сфокусовано увагу на використанні концепції Infrastructure as Code (IaC). Зокрема, розглянуто переваги та виклики її впровадження та визначено значущість технічної експертизи для ефективного управління хмарною інфраструктурою [2].

Крім того, дослідники зосереджуються на продуктивності PHP-фреймворків, методах забезпечення безпеки та надійності кодової бази, а також на оптимізації автоматизованого розгортання. Зокрема, розглядаються питання конфігурації веб-додатків у хмарних середовищах, тестування ефективності підходів до Infrastructure as Code, а також створення механізмів забезпечення відмовостійкості. Серед праць, присвячених цим темам, слід відзначити дослідження М. Лаазірі, К. Бенмусси, С. Хоулджі, Л. Керкеба [3], С. Чінтапатли [4], А. Рахмана, А. Партхо, П. Моррісона, Л. Вільямса [5], В. Семенюка [6], С. Далла Пальми, Д. Ді Нуччі, Ф. Паломби, Д. Тамбуррі [7], Ю. Жао, І. Лу, К. Чжу, Ч. Чен, Х. Хуана [8], Е. Оздоґана, О. Черана, М. Устюндаґа [9], С. Ніфа, Л. Кляйсснера [10], А. Абуалькішика, А. Алвана, Й. Гульзара [11], Бату Дж. [12], Н. Мухопадхяя, Б. Теварі [13], Й. Омофоєви, А. Гребе, П. Леусманна [14], М. Елеракі, В. Аніса Азіза, Дж. Солімана [15] тощо.

Беручи до уваги наявні дослідження тим не менше можна стверджувати, що питання автоматизації процесів перемикання сервісів наземної інфраструктури в хмарне середовище залишається актуальним та потребує подальшого наукового опрацювання, зокрема щодо розробки нових методів інтеграції та оптимізації управління ресурсами у гетерогенних середовищах.

Постановка завдання. Метою статті є розробка методів та засобів автоматизації процесів переключення сервісів наземної інфраструктури в хмарне середовище із використанням інструменту інфраструктури як код Terraform у середовищі AWS та Docker-контейнеризації у парадигмі «blue-green» розгортання.

Виклад основного матеріалу. Задля кращого розуміння необхідності проведення хмарної міграції за посередництва IaC, необхідно усвідомити значимість ознак концепцій хмарних обчислень, які відіграють головну роль у підході поточного дослідження. Традиційна архітектура часто передбачає ручне виділення ресурсів, що своєю суттю сприяє виникненню помилок. Натомість, архітектури хмарних обчислень вирізняються п'ятьма ознаками:

Самообслуговування на вимогу – означає, що користувачі мають змогу автоматично запитувати ресурсі у манері самообслуговування;

– широкий мережевий доступ – вказує на те, що ресурси є доступними через Інтернет, типово через стандартизовані механізми;

– спільний пул ресурсів – гарантує, що базові ресурси керуються в якості спільного пулу, часто віртуалізованого, та можуть використовуватися одночасно кількома клієнтами;

– швидка еластичність – вказує на можливість забезпечення більшої кількості ресурсів на вимогу та звільняти їх за необхідності;

– вимірювана послуга – означає моніторування використання ресурсів та вимірювання оптимального використання.

Загалом, поняття хмарної інфраструктури підтримується трьома головними моделями: Software as a Service (SaaS – програмне забезпечення в якості послуги), Platform as a Service (PaaS – платформа в якості послуги), and Infrastructure as a Service (IaaS – інфраструктура в якості послуги). При необхідності мати справу з більш комплексними автоматизаційними парадигмами, IaaS зазвичай є найбільш релевантною, оскільки вона надає адміністратора можливість контролювати на доволі детальному рівні інфраструктурні ресурси, включаючи віртуальні машини, міркування з приводу зберігання даних та інші послуги низького рівня. У той же час, PaaS уможливлює спрощення розробки, оскільки вона пропонує вже конфігуроване середовище, у якому відбувається розгортання коду. SaaS, альтернативним чином, абстрагує цілий процес розгортання на ще вищому рівні, дозволяючи організаціям використовувати повноцінні програмні рішення через веб.

Незалежно від того, яка сервісна модель є обраною, IaC може виявитися особливо корисною у процесі оркестрації виділення та обслуговування хмарних середовищ. Концептуальна особливість використання цієї моделі полягає у тому, що усе середовище, включаючи сервери, мережі, БД з'єднання, технології хешування тощо можуть виражатися у вигляді конфігураційних файлів, або скриптів. Ці вираження зберігаються у системах контролю версій, тестуються та потім автоматично розгортаються за потреби. Terraform є одним з основних інструментів, використовуваних у даному контексті, оскільки застосовує декларативно-конфігураційну мову HashiCorp Configuration Language (HCL), таким чином дозволяючи користувачам визначати бажаний стан інфраструктури, після чого оркеструючи створення, або видалення ресурсів задля узгодження поточного стану системи з бажаним.

Оскільки хмарні міграції існують у різних формах, важливим є вибір належного типу в залежності від ступеню необхідного перетворення. Найбільш прямолінійний підхід хмарної міграції полягає у заміщенні конкретних локальних компонентів еквівалентними хмарними сервісами. Інший метод може включати часткову міграцію деяких функцій. Підхід «Cloudify» вимагає повного переналаштування додатку таким чином, ніби він з самого початку був розроблений в межах хмарної інфраструктури. У будь-якому із зазначених підходів, принцип IaC прискорює процес міграції, оскільки за необхідності впровадження будь-якої зміни, ручна перебудова стає непотрібною, та необхідні зміни можуть бути інтегрованими автоматичним чином послідовно та неодноразово, що, у свою чергу, майже анулює вірогідність виникнення будь-яких помилок, пов'язаних з людським фактором та прискорює процес міграції.

Отже декларативний синтаксис Terraform є центральною ланкою запропонованого підходу, який звертається до провайдерів (таких як AWS, Azure або Google Cloud Platform) для налаштування та управління необхідними ресурсами. До того ж, характерне використання даним інструментом змінних надалі посилює його адаптивність. Змінні можуть бути запроваджені у декількох формах: змінні середовища, аргументи командного рядку, або виділені конфігураційні файли. Таким чином, це робить створення інфраструктури гнучкішим та схильним до багаторазового використання. Розширені робочі процеси Terraform також можуть покладатися на Terraform Cloud, що є хостинговим сервісом, який забезпечує централізоване зберігання стану, функції колаборації, інтеграцію контролю версіями та забезпечення дотримання політик. При використанні Terraform Cloud, команди можуть позначати конкретні робочі простори та точно визначати спосіб версійного контролю та аудиту своїх розгортань.

У контексті контейнеризованих додатків, Docker постає в якості критично важливого інструменту, оскільки він здійснює пакування додатку з усіма його залежностями: фреймворками, бібліотеками, середовищами виконання у портативні контейнери. Це пакування гарантує ідентичну функціональність додатку у різних середовищах. Типовий робочий процес може починатися з встановлення образу Docker, який включає середовище виконання (PHP, Java, або Python) та додаткові бібліотеки, від яких залежить певний додаток. Втім, шляхом застосування Docker Compose, існує можливість визначення мультиконтейнерних конфігурацій, конкретизації того, які контейнери запускатимуть БД, кеш-шари, або брокери повідомлень та як вони взаємодіятимуть, таким чином об'єднуючи локальні середовища розробки серед різних членів команди, знижуючи ризик конфліктів версій, або проблем сумісності.

При застосуванні зазначених Docker концепцій до додатків на базі PHP, існує можливість застосування оптимізаційних та хешових технологій, як наприклад Redis та Memcached, які сприяють зниженні навантаження БД шляхом збереження часто керованих даних у оперативній пам'яті. Redis, наприклад також може оперувати в якості системи

обміну повідомленнями з підтримкою публікації та підписки (pub / sub), у той час, як Memcached відіграє роль рішення більш спеціалізованого та горизонтально масштабованого кешування. Інша ключова підланка полягає у активації Opcache, що значно покращує продуктивність PHP кешуванням байткоду компільованого скрипта таким чином, щоб він не мав необхідності бути повторно парсованим та компільованим за кожним запитом. Встановлення xDebug для здійснення дебагінгу, або SOAP для забезпечення веб-сервісних функцій є опціональним за умови покладання на аргументи збірки у Dockerfile.

Пропонований підхід передбачає, що усі описані технології безшовно узгоджуватимуться за посередництва стратегії «blue-green» розгортання. У рамках даної моделі, адміністраторами керуються два ідентичних продакшн-середовищ: блакитного та зеленого, один з яких перебуватиме у стані активного використання. При готовності нової версії додатку, або його базової інфраструктури, резервне середовище підлягає оновленню, тестуванню та валідації. За умови успішного тестування, конфігурація маршрутизації, або балансу навантаження переходить з старого середовища до нового, забезпечуючи вихід нової версії з майже жодним часом простою. При виникненні проблем, потік запитів може бути негайно переспрямованим назад до старого середовища. Простота виділення інфраструктури в Terraform забезпечує прямолінійний спосіб розгортання цих паралельних середовищ, оскільки він гарантує, що кожне середовище виділяється ідентично та у різних стадіях розгортання. Контейнеризація за допомогою Docker прискорює процес конфігурації та масштабування сервісів у двох середовищах шляхом зниження їх об’єму та портування, що робить можливим їх встановлення за декілька секунд.

З боку практичного аспекту, робота Terraform передбачає взаємодію з різними хмарними API (application programming interface – прикладний програмний інтерфейс). Наприклад, у випадку AWS, типова конфігурація Terraform може визначати функцію AWS Lambda та під’єднати її до ресурсу API Gateway. У своїй простішій формі, конфігураційний файл посилається на функцію Lambda, встановлюючи function_name, runtime, handler, and filename. Далі він описує API Gateway

```
resource "aws_lambda_function" "my_lambda_function" {
  function_name = "my-lambda-function"
  runtime       = "nodejs14.x"
  handler       = "index.handler"
  filename      = "${path.module}/lambda_function_payload.zip"
}

resource "aws_api_gateway_rest_api" "my_api_gateway" {
  name        = "my-api-gateway"
  description = "My API Gateway"
}

resource "aws_api_gateway_resource" "my_api_gateway_resource" {
  rest_api_id = aws_api_gateway_rest_api.my_api_gateway.id
  parent_id   = aws_api_gateway_rest_api.my_api_gateway.root_resource_id
}

resource "aws_api_gateway_method" "my_api_gateway_method" {
  rest_api_id = aws_api_gateway_rest_api.my_api_gateway.id
  resource_id = aws_api_gateway_resource.my_api_gateway_resource.id
  http_method = "GET"
  authorization = "NONE"
}

resource "aws_api_gateway_integration" "my_api_gateway_integration" {
  rest_api_id             = aws_api_gateway_rest_api.my_api_gateway.id
  resource_id             = aws_api_gateway_resource.my_api_gateway_resource.id
  http_method             = aws_api_gateway_method.my_api_gateway_method.http_method
  integration_http_method = "POST"
  type                    = "AWS_PROXY"
  uri                     = aws_lambda_function.my_lambda_function.invoke_arn
}

resource "aws_api_gateway_deployment" "my_api_gateway_deployment" {
  depends_on = [aws_api_gateway_integration.my_api_gateway_integration]
  rest_api_id = aws_api_gateway_rest_api.my_api_gateway.id
  stage_name  = "dev"
}
```

Рис. 1. Визначення функції Lambda та API Gateway

```
cat <<EOF > main.tf
provider "aws" {
}

terraform {
  backend "s3" {
    bucket = "dev-area"
    key    = "Terraform/lambda/\${state_name}.tfstate"
  }

  required_providers {
    aws = {
      version = ">= 4.67.0"
      source  = "hashicorp/aws"
    }
  }
}
EOF

cat <<EOL > lambda.tf
resource "aws_lambda_function" "\${state_name}_function" {
  function_name = var.lambda_name
  filename      = "lambda.zip"
  handler       = "exports.handler"
  runtime       = "nodejs18.x"
  role          = "arn:aws:iam::\${account_id}:role/LambdaRole"
}
EOL
```

Рис. 2. Генерація Terraform-конфігураційних файлів

за посередництва інтеграції яка маршрутизує трафік HTTP до конкретної Lambda. Такі визначення мають змогу посилатися одне на одного, що гарантує Terraform-оркестрацію у правильному порядку.

На рис. 1 наведено короткий приклад конфігурації Terraform, яка визначає функцію Lambda та API Gateway, за своєю суттю демонструючи принцип поєднання даних ресурсів один з одним за допомогою декларативного синтаксису.

У повністю автоматизованому робочому процесі, дані ресурси зберігаються у системі контролю версій, а пайплайн може запускати команди Terraform (init, plan, apply). Файл стану Terraform, який відстежує ці ресурси, може зберігатися у середовищі Amazon S3, що гарантуватиме використання однакових станів посилання будь-яким членом команди, який запускає Terraform. Типовий скрипт, як показано на рис. 2, може генерувати певні Terraform-конфігураційні файли в залежності від змінних середовища, що дозволяє застосовувати гнучкі правила іменування та розділення середовищ.

Після успішного створення цих файлів, пайплайни CI / CD у таких платформах, як GitHub Actions здатні посилатися на них. Типовим чином, пайплан здійснить конфігурацію облікових даних AWS, викличе скрипт та запустить terraform init-reconfigure, потім terraform plan та terraform apply. Terraform Cloud також може замінити S3-бекенд, якщо команда надає перевагу повністю хостованому рішенню з додатковими колаборайними функціями та функціями політик. На рис. 3 наведена одна з можливих конфігурацій Terraform Cloud.

```
terraform {
  required_version = ">= 0.13.0"
  backend "remote" {
    organization = "organization_name"
    workspaces {
      name = "example-workspace"
    }
```

Рис. 3. Конфігурація Terraform Cloud

Контейнеризація самого програмного забезпечення також являє собою наступну ланку хмарної міграції. Для додатків PHP, образи Docker можуть бути побудовані на основі базового образу (php: 7.1-fpm) та потім розширені за допомогою додаткових бібліотек, як наприклад BCMath, Redis, Memcached, або xDebug для дебагінгу, або SOAP за необхідності. Консолідація усіх необхідних модулів у єдиний образ, який підтримує кешування та дебагінг, може значно впорядкувати про-

```
FROM php:7.1-fpm

RUN apt-get update && apt-get install -y \
    libmemcached-dev \
    zlib1g-dev \
    libpq-dev \
    libxml2-dev \
    curl \
    libcurl4-openssl-dev

RUN docker-php-ext-install pdo pdo_mysql pdo_pgsql gd mcrypt bcmath \
    && pecl install redis memcached \
    && docker-php-ext-enable redis memcached

ARG INSTALL_SOAP=false
ARG INSTALL_XDEBUG=false

RUN if [ "$INSTALL_SOAP" = "true" ]; then \
      docker-php-ext-install soap; \
    fi \
    && if [ "$INSTALL_XDEBUG" = "true" ]; then \
      pecl install xdebug && docker-php-ext-enable xdebug; \
    fi

COPY configs/php.ini /usr/local/etc/php/php.ini
COPY configs/php-fpm.pool.conf /usr/local/etc/php-fpm.d/www.conf
COPY configs/xdebug.ini /usr/local/etc/php/conf.d/xdebug.ini
COPY configs/opcache.ini /usr/local/etc/php/conf.d/opcache.ini

EXPOSE 9000
WORKDIR /var/www
CMD ["php-fpm"]
```

Рис. 4. Конфігурація Dockerfile

цес локальної розробки. Docker Compose, у свою чергу пов'язує ці контейнери між собою, щоб MySQL, Nginx, RabbitMQ мали змогу оперувати паралельно з PHP додатком без необхідності проведення ручного процесу встановлення.

Зразок Dockerfile для PHP середовища може включати компоненти, які встановлюють різні модулі (pdo, pdo_mysql, pdo_pgsql, gd, mcrypt, bcmath), а потім застосовують pecl для зовнішніх модулей: redis або memcached. Аргументи збірки (ARG) можуть визначати, чи будуть xDebug, SOAP або інші компоненти скомпільовані в образ. На рис. 4 окреслено цей принцип.

Ця конфігурація може включати опціональні функції за умови, що аргументи збірки встановлені до true. При необхідності проведення дебагінгу, можуть бути впроваджені такі аргументи як – build-arg INSTALL_XDEBUG=true – build-arg INSTALL_SOAP=true. Для продакшн-образів, значення цих аргументів можуть залишатися стандартними.

Не менш критично важливим аспектом у оптимізації PHP середовищ є активація Opcache, оскільки як вже було зазначено, шляхом збереження попередньо скомпільованого байткоду, цей інструмент усуває накладні витрати на парсування та компіляцію скриптів при кожному запиті. Після створення образу, він може бути просто запущеним за посередництва прив'язання до порту 9000 та змонтувавши локальну директорію, яка містить код додатку:

docker build -t php – 7.1-custom.

docker run -d -p 9000:9000 – v $(pwd): / var / www php – 7.1-custom

Треба також враховувати, що Docker Compose в змозі визначити усі компоненти (PHP-FPM, кеш, БД, черги повідомлень) у вигляді єдиного YAML-файлу, критично спрощуючи налаштування середовища для кожного члена команди. Будь-який розробник, який вперше долучається до роботи над проектом, має змогу клонувати репозиторій, запустити одну команду (docker-compose up -d) та мати стандартизоване середовище майже одразу.

Значущість контейнеризації посилюється у межах «blue-green» розгортання. Коли випускаються нові версії сервісу, операційний пайплайн може запустити контейнери в зеленому середовищі, що є ідентичним блакитному середовищу, що знаходиться в процесі роботи. Ці контейнери підлягають процесу ретельного тестування, щоб запевнитися, що кешування функціонує правильно, що нова версія коду додатку є стабільною та що уся відповідна інфраструктура, розгорнута Terraform, знаходиться в готовому стані. Тільки після успішної валідації відбувається переключення середовищ. Оскільки образи Docker охо-

плюють усі залежності, нові контейнери мають передбачувані властивості, незалежно від змін на базових серверах.

Висновки. Загалом, інтеграція IaC із декларативним виділенням ресурсів та методами контейнеризації для оптимізації стратегій хмарної міграції та використання Terraform і Docker демонструє, як автоматизована оркестрація підвищує масштабованість інфраструктури, мінімізує людський фактор і забезпечує повторюваність процесів розгортання. На відміну від традиційних підходів, запропонована методологія систематично поєднує контрольовані версіями конфігурації, автоматизацію на основі політик і адаптацію стану інфраструктури, створюючи надійну основу для динамічних хмарних середовищ.

Подальші дослідження мають зосередитися на розширенні оптимізації хмарних обчислень на основі IaC, зокрема шляхом інтеграції машинного навчання для передбачуваного масштабування, покращення міжхмарної сумісності та вдосконалення політик безпеки для автоматизованих розгортань. Додатково перспективним напрямком є дослідження інтеграції архітектур service mesh та serverless computing у IaC-фреймворки, що сприятиме подальшій оптимізації розподілу ресурсів та підвищенню ефективності роботи.

Список літератури:

1. Laviale O. Installing PHP extensions from source in your Dockerfile. *Olivier Laviale*: website. 2019. URL: https://olvlvl.com/2019-06-install-php-ext-source.html (last accessed: 28.03.2025).

2. Kishiyama B., Wang H., Lopez D., Yang J. An Overview of Infrastructure as Code (IaC) with Performance and Availability Assessment on Google Cloud Platform. *Proceedings of the Second International Conference on Advances in Computing Research (ACR'24) Lecture Notes in Networks and Systems*. 2024. P. 497–514. DOI: https://doi.org/10.1007/978-3-031-56950-0_41

3. Laaziri M., Benmoussa K., Khoulji S., Kerkeb L. A comparative study of PHP frameworks performance. *Procedia Manufacturing*. 2019. Vol. 32. P. 864–871. DOI: 10.1016/j.promfg.2019.02.295

4. Chinthapatla S. Unleashing the Power of AWS: Revolutionizing Cloud Management through Infrastructure as Code (IaC). *ResearchGate*: website. 2024. URL: https://www.researchgate.net/publication/379225463_SaiKrishna_Chinthapatla's_view_on_IaC_Unleashing_the_Power_of_AWS_Revolutionizing_Cloud_Management_through_Infrastructure_as_Code_IaC (last accessed: 28.03.2025).

5. Rahman A., Partho A., Morrison P., Williams L. What questions do programmers ask about configuration as code? *Proceedings of the 4th International Workshop on Rapid Continuous Software Engineering*. 2018. P 16–22, DOI: 10.1145/3194760.3194769

6. Semenyuk V. Automated build for docker-php image. *Docker Hub*: website. 2025. URL: https://hub.docker.com/r/vadymsemeniuk/docker-php (last accessed: 28.03.2025).

7. Dalla S. Palma, Di D. Nucci, Palomba F., Tamburri D. A. Within- project defect prediction of Infrastructure-as-Code using product and process metrics. *IEEE Transactions on Software Engineering*. 2020. № 14(8). P. 1–8.

8. Zhao J., Lu Y., Zhu K., Chen Z., Huang H. Cefuzz: A directed fuzzing framework for PHP RCE vulnerability. *Electronics*. 2022. Vol. 11. № 5. P. 758. DOI: 10.3390/electronics11050758

9. Özdoğan E., Ceran O., Üstündağ M. Systematic Analysis of Infrastructure as Code Technologies. *Gazi University Journal of Science A. Part, Engineering and Innovation*. 2023. № 10. DOI: 10.54287/gujsa.1373305

10. Neef S., Kleissner L. PHUZZ: A grey-box fuzzer for PHP web applications. *GitHub*: website. 2024. URL: https://github.com/gehaxelt/phuzz (last accessed: 28.03.2025).

11. Abualkishik A., Alwan A., Gulzar Disaster Y. Recovery in Cloud Computing Systems: An Overview. *InternationalJournal of Advanced Computer Science and Applications.* 2020. № 11. DOI: 10.14569/IJACSA.2020.0110984

12. Batu J. Implementing Infrastructure-as-Code with Cloud Disaster Recovery Strategies. *International Journal of Computer Trends and Technology*. 2024. № 72. P. 41–45. DOI: 10.14445/22312803/IJCTT-V72I2P108

13. Mukhopadhyay N., Tewari B. P. Efficient IaC-based resource allocation for virtualized cloud platforms. In: *Advanced network technologies and intelligent computing*. Communications in Computer and Information Science. Springer International Publishing, 2022. P. 200–214. DOI: https://doi.org/10.1007/978-3-030-96040-7_16

14. Omofoyewa Y., Grebe A., Leusmann P. IaC reusability for Hybrid Cloud Environment. *ResearchGate*: website. 2021. URL: https://www.researchgate.net/publication/357281177_IaC_reusability_for_Hybrid_Cloud_Environment (last accessed: 28.03.2025).

15. Eleraky M., Anis Aziz W., Soliman J. Using Cloud Infrastructure as a Code (IaC) to Set Up Web Applications. *International Journal of Simulation: Systems, Science & Technology*. 2023. Vol. 24.

Behlitsov S. V. SEAMLESS CLOUD MIGRATION WITH TERRAFORM AUTOMATION AND DOCKER CONTAINERIZATION: INTEGRATING INFRASTRUCTURE AS CODE AND BLUE-GREEN DEPLOYMENT

The article is devoted to the study of migration to the cloud environment using Infrastructure as Code (IaC) with a focus on the key attributes of cloud computing. The paper reveals the features of traditional architectures that require manual resource allocation and identifies the advantages of cloud technologies that provide flexibility through on-demand self-service, broad network access, shared resource pool, fast elasticity, and measurable services.

The main models of cloud infrastructure are considered: SaaS, PaaS and IaaS. It has been found that the IaaS model is the most relevant for automation, as it provides detailed control over virtual machines and storage. PaaS simplifies development through preconfigured environments, while SaaS allows organizations to use managed software solutions.

The effectiveness of using IaC to orchestrate cloud infrastructure is proved by defining it through configuration files stored in version control systems, tested and automatically deployed. The role of Terraform is defined, which, using the HashiCorp Configuration Language (HCL), provides automated resource allocation.

Migration strategies are considered, ranging from simple component replacement to complete application reconfiguration (Cloudify). It is shown that IaC minimizes manual intervention and error risks. Terraform's declarative syntax and its integration with AWS, Azure, and Google Cloud for flexible deployment are described.

Special attention is paid to containerized applications that benefit from Docker. It is determined that Docker packs dependencies into portable containers, and Docker Compose manages the configuration of multi-container environments, optimizing PHP applications through Redis, Memcached, and Opcache. The possibility of integrating debugging tools such as xDebug and SOAP is considered.

It was found that the blue-green deployment strategy provides a smooth upgrade, supporting two identical production environments and switching between them with minimal downtime. It is confirmed that Terraform efficiently provides the deployment of these environments, while Docker accelerates the configuration of services and scaling.

It is proved that Terraform automates interaction with cloud platform APIs by integrating AWS Lambda and API Gateway into CI/CD pipelines. It is determined that Terraform Cloud further optimizes state management and simplifies teamwork. The study confirms that Terraform and Docker simplify cloud migration by automating infrastructure provisioning, improving deployment efficiency, and reducing operational risks.

Key words: *declarative syntax, automated orchestration, pipeline, container, versioned deployments, debugging.*