МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ПРАКТИЧЕСКИМ РАБОТАМ

«Облачные технологии»

Направление подготовки 09.03.04 «Программная инженерия»
Направленность (профиль) «Разработка и сопровождение программного обеспечения»
Квалификация выпускника: бакалавр

Ставрополь
2026

ВВЕДЕНИЕ

Настоящие методические указания предназначены для выполнения лабораторных работ по дисциплине **«Облачные технологии»**.

Цель освоения дисциплины: сформировать у студентов практические навыки использования публичных облачных платформ (Yandex Cloud, AWS, Google Cloud) для развёртывания, масштабирования и управления приложениями, включая работу с виртуальными машинами, объектным хранилищем, управляемыми базами данных, инфраструктурой как код (Terraform), контейнеризацией в облаке (Managed Kubernetes), serverless-функциями, а также мониторингом и обеспечением безопасности.

Задачи дисциплины:

- освоение базовых моделей облачного обслуживания (IaaS, PaaS, SaaS), принципов работы облачных платформ (регионы, зоны доступности, CAPEX/OPEX);
- формирование навыков создания и настройки виртуальных машин, групп безопасности, управляемых групп инстансов и балансировщиков нагрузки для горизонтального масштабирования;
- приобретение практических умений работы с облачным объектным хранилищем (S3-совместимым): версионирование, политики доступа, pre-signed URL, статический хостинг;
- изучение управляемых баз данных (Managed PostgreSQL/MySQL): резервное копирование, репликация, подключение приложений с SSL;
- применение подхода «инфраструктура как код» (IaC) с использованием Terraform для описания и управления облачными ресурсами;
- знакомство с контейнеризацией в облаке: работа с Container Registry, создание Managed Kubernetes-кластера, деплой приложений через Deployment и Service типа LoadBalancer;
- настройка облачного мониторинга, логирования и алертинга для отслеживания состояния ресурсов.

Данные методические указания включают **9 лабораторных работ**, которые последовательно проводят студента от базовых операций с виртуальными машинами до полноценного serverless-стек с интеграцией объектного хранилища и Kubernetes. Каждая работа содержит чётко сформулированные цель и задачи, теоретическое обоснование, детальный пример выполнения, 15 индивидуальных вариантов заданий (включая вариативные части для углублённого изучения), контрольные вопросы, требования к отчёту и критерии оценивания.

Лабораторная работа № 1

Создание виртуальной машины в облаке

1. Цель работы

Получить базовые навыки работы с IaaS (инфраструктура как услуга): создание виртуальной машины (ВМ) в публичном облаке, подключение к ней по SSH, настройка групп безопасности (файервола) для управления сетевым доступом.

2. Задачи работы

1. Зарегистрироваться / войти в облачную платформу (Yandex Cloud / AWS / Google Cloud).
2. Создать виртуальную машину на основе ОС Ubuntu 22.04 LTS с параметрами: 2 vCPU, 4 ГБ RAM, диск 20 ГБ (SSD).
3. Сгенерировать пару SSH-ключей и добавить публичный ключ при создании ВМ.
4. Создать группу безопасности с правилами:
– inbound: порт 22 (SSH) для доступа по протоколу TCP из любой сети (0.0.0.0/0);
– inbound: порт 80 (HTTP) для доступа по TCP из любой сети (0.0.0.0/0);
– outbound: все разрешено.
5. Подключиться к ВМ по SSH.
6. Установить веб-сервер **nginx**, запустить его и проверить доступность через публичный IP-адрес ВМ.
7. Зафиксировать результаты в отчёте (скриншоты, команды, выводы).

3. Теоретическое обоснование

IaaS (Infrastructure as a Service) – модель облачного обслуживания, при которой пользователю предоставляются виртуальные вычислительные ресурсы (процессоры, память, диски, сети). Пользователь не управляет физической инфраструктурой, но управляет ОС, ПО, настройками файервола.

Виртуальная машина (VM) – программная эмуляция физического компьютера, работающая на гипервизоре облачного провайдера.

Группа безопасности (Security Group) – набор правил фильтрации трафика на уровне виртуального сетевого интерфейса (stateful firewall). Аналог сетевого ACL, но привязан к ВМ.

SSH-ключи – пара ключей (приватный и публичный) для аутентификации без пароля. Публичный ключ загружается в облако, приватный хранится у пользователя.

Порты:

- 22/TCP – SSH (Secure Shell) для удалённого управления.
- 80/TCP – HTTP для веб-трафика.

Nginx – легковесный веб-сервер и обратный прокси-сервер.

4. Пример практического выполнения (на платформе Yandex Cloud)

4.1. Создание ВМ в веб-консоли

1. Перейти в каталог → **Compute Cloud → Виртуальные машины → Создать ВМ**.
2. Задать имя: `lab1-vm-{номер_студента}`.
3. Выбрать **образ** – Ubuntu 22.04 LTS.
4. **Вычислительные ресурсы**:

– платформа Intel Ice Lake;
– 2 vCPU, 4 ГБ RAM.

5. **Диск**: SSD, 20 ГБ.
6. **Сетевые настройки**:

– создать группу безопасности или выбрать существующую (см. п. 4.2);
– публичный IP – автоматически.

7. **Доступ**: вставить публичный SSH-ключ (содержимое файла `~/.ssh/id_rsa.pub`).
8. Нажать **Создать**.

4.2. Создание группы безопасности

1. Перейти в **Virtual Private Cloud → Группы безопасности → Создать группу**.
2. Имя: sg-lab1-http-ssh.
3. Добавить правила:

Направление	Диапазон портов	Протокол	CIDR / источник	Назначение
Входящий	22	TCP	0.0.0.0/0	SSH доступ
Входящий	80	TCP	0.0.0.0/0	HTTP доступ
Исходящий	все	любой	0.0.0.0/0	любой

4. Привязать группу к ВМ при создании.

4.3. Подключение по SSH

bash

```
chmod 600 ~/.ssh/id_rsa
ssh -i ~/.ssh/id_rsa ubuntu@<публичный_IP_ВМ>
```

4.4. Установка и запуск nginx

bash

```
sudo apt update
sudo apt install nginx -y
sudo systemctl enable nginx
sudo systemctl start nginx
sudo systemctl status nginx
```

4.5. Проверка доступности

На самой ВМ:

bash

```
curl http://localhost
```

Должна отобразиться HTML-страница приветствия nginx.

В браузере локального компьютера: http://<публичный_IP_ВМ> → страница «Welcome to nginx».

4.6. Скриншоты для отчёта

- Список ВМ с созданной машиной и её публичным IP.
- Правила группы безопасности (таблица с портами 22 и 80).
- Вывод команды `curl http://localhost` в терминале.
- Страница nginx в браузере.

5. Индивидуальные задания (15 вариантов)

Каждый вариант отличается **дополнительным требованием** (вариативная часть). Базовое задание (создание ВМ, SSH, nginx) одинаково для всех.

№ варианта	Дополнительное задание (вариативная часть)
1	Изменить порт веб-сервера с 80 на 8080. Настроить группу безопасности для порта 8080. Проверить доступность.
2	Установить не nginx, а Apache2 (`sudo apt install apache2 -y`). Проверить доступность.
3	После установки nginx скопировать собственную HTML-страницу с текстом «Вариант 3, ФИО» в `/var/www/html/index.html`.
4	Добавить в группу безопасности порт 443 (HTTPS) и установить самоподписанный сертификат (openssl), настроить nginx на HTTPS.
5	Создать ВМ с диском 30 ГБ вместо 20 ГБ. Вывести в отчёт команду `df -h`.
6	Вместо Ubuntu 22.04 использовать **Debian 11**. Установить nginx.
7	После установки nginx создать файл `info.php` с `<?php phpinfo(); ?>`, установить PHP-FPM, настроить nginx для обработки PHP. Проверить `http://IP/info.php`.
8	Создать ВМ с тэгами `env=lab1, student=ФИО`. В отчёте показать фильтрацию ВМ по тэгам в консоли.

№ варианта	Дополнительное задание (вариативная часть)
9	Добавить второй диск (10 ГБ), смонтировать его в /mnt/data, создать на нём файл test.txt через SSH.
10	Ограничить доступ к порту 80 только с вашего IP-адреса (узнать через curl ifconfig.me). Проверить, что с другого IP доступ закрыт.
11	Установить Docker и запустить nginx в контейнере (docker run -d -p 80:80 nginx). Группа безопасности – порт 80.
12	Создать ВМ без публичного IP, но через NAT-шлюз. Организовать доступ через **bastion host** (дополнительная ВМ с публичным IP). В отчёте описать схему.
13	Настроить автоматическое обновление ОС (unattended-upgrades) и веб-сервера. Показать логи обновлений.
14	Создать ВМ с резервным диском (snapshot schedule – ежедневно в 02:00). В отчёте скриншот расписания.
15	Установить **fail2ban** для защиты SSH. Настроить бан после 3 неудачных попыток. Показать логи блокировок.

Указание: студент выполняет базовую часть (обязательно) и одну вариативную часть в соответствии с номером варианта, выданным преподавателем.

6. Контрольные вопросы (для защиты)

1. Что такое IaaS? Приведите два примера облачных провайдеров IaaS.
2. Чем отличается публичный IP от приватного в облачной ВМ?

3. Для чего нужны группы безопасности? Чем они отличаются от традиционного сетевого экрана (firewall) на сервере?

4. Какой протокол и порт используется для SSH? Почему не рекомендуется открывать порт 22 для всех IP (0.0.0.0/0) в production?

5. Что такое SSH-ключ? Как сгенерировать пару ключей на Linux/macOS/Windows?

6. Как проверить, что веб-сервер (nginx) запущен без открытия браузера?

7. Какая команда выводит список всех правил iptables (или nftables) на самой ВМ?

8. В чём разница между `sudo apt update` и `sudo apt upgrade`?

9. Что произойдёт, если в группе безопасности не разрешить исходящий трафик (outbound)? Почему по умолчанию исходящий трафик обычно разрешён?

10. Как изменить тип виртуальной машины (количество vCPU/RAM) после её создания в облаке? Всегда ли это возможно без перезагрузки?

7. Требования к отчёту

Отчёт должен быть оформлен в текстовом редакторе (Word, LibreOffice) в соответствии с ГОСТ (шрифт Times New Roman, 14 пт, полуторный интервал, поля: левое 30 мм, правое 15 мм, верхнее/нижнее 20 мм).

Содержание отчёта:

1. **Титульный лист** – ФИО, группа, вариант, название работы, год.
2. **Цель и задачи работы** (по п. 1 и 2 настоящего описания).
3. **Теоретическое обоснование** – кратко (0,5–1 стр.) основные понятия: IaaS, VM, Security Group, SSH.
4. **Практическая часть**:
   - пошаговое описание действий (что создавалось, какие команды выполнялись);

- для **базовой части** – скриншоты (ВМ, группы безопасности, терминал с curl, браузер);
- для **вариативной части** – дополнительные скриншоты и пояснения.

5. **Результаты выполнения** – таблица с указанием параметров ВМ (vCPU, RAM, диск, публичный IP), статус nginx (active), результат curl.
6. **Анализ результатов** – отвечают ли результаты поставленной цели? Какие возникли трудности? Как были преодолены?
7. **Ответы на контрольные вопросы** (письменно, развёрнуто).
8. **Вывод** – что освоено, какие навыки получены (например: «В ходе работы освоил создание ВМ в облаке, настройку файервола, подключение по SSH, установку и проверку веб-сервера»).

8. Критерии оценивания (по 5-балльной шкале)

Оценка	Критерии
«Отлично»	Работа выполнена полностью, **базовая + вариативная части**. ВМ создана корректно, SSH доступен, nginx работает по HTTP. Группа безопасности настроена верно. Отчёт структурирован, содержит все скриншоты и пояснения. В выводах – анализ, в ответах на вопросы – правильные развёрнутые ответы. Защита: уверенно отвечает на дополнительные вопросы.
«Хорошо»	Работа выполнена полностью, но **только базовая часть** (вариативная отсутствует). Основной функционал

Оценка	Критерии
	работает. Отчёт содержит незначительные недочёты (нет одного скриншота, мелкие ошибки в тексте). Защита: отвечает на большинство вопросов, но не на все.
«Удовлетворительно»	Работа выполнена частично. ВМ создана, но SSH или nginx работают с ошибками (например, порт 80 не открыт в группе безопасности, страница не загружается). Отчёт неполный (например, нет ответов на вопросы или скриншотов). Студент затрудняется при ответах.
«Неудовлетворительно»	Работа не представлена, либо ВМ не создана, либо ключевой функционал отсутствует (нет доступа по SSH, нет веб-сервера). Отсутствует отчёт или он не соответствует требованиям.

Лабораторная работа № 2
Управляемая группа инстансов и балансировщик нагрузки

1. Цель работы

Научиться горизонтальному масштабированию приложения в облаке с использованием управляемых групп инстансов (Instance Groups) и балансировщика нагрузки. Освоить настройку автомасштабирования по метрикам CPU и обеспечение отказоустойчивости приложения.

2. Задачи работы

1. Создать **шаблон виртуальной машины** (образ с предустановленным nginx и скриптом, выводящим hostname).
2. Создать **управляемую группу инстансов** (Instance Group / Managed Instance Group) из 2–3 ВМ.
3. Настроить **автомасштабирование** по загрузке CPU (при превышении 70% в течение заданного времени — добавить новую ВМ).
4. Создать **балансировщик нагрузки** (Load Balancer) перед группой инстансов.
5. Настроить **проверку здоровья** (health check) по HTTP-пути /health.
6. Провести **нагрузочное тестирование** (wrk, ab или yandex-tank) для активации автомасштабирования.
7. Проверить **отказоустойчивость**: отключить одну ВМ и убедиться, что сервис остаётся доступным через балансировщик.
8. Оформить отчёт со скриншотами и анализом.

3. Теоретическое обоснование

Управляемая группа инстансов (Instance Group) – сервис облачного провайдера, который автоматически создаёт и поддерживает заданное количество однотипных ВМ (инстансов) на основе шаблона. Обеспечивает:

- **Автомасштабирование** – изменение количества ВМ в зависимости от нагрузки (CPU, RAM, сетевые метрики или расписание).
- **Автовосстановление** – пересоздание ВМ при сбое (health checks).
- **Равномерное размещение** по зонам доступности.

Балансировщик нагрузки (Load Balancer) – распределяет входящий трафик между несколькими ВМ, скрывая реальные IP-адреса. Типы:

- **Внутренний** – для трафика внутри облачной сети.
- **Внешний** – для доступа из интернета.

Алгоритмы балансировки: round-robin, least connections, session affinity (sticky sessions).

Health Check – периодические запросы балансировщика к каждому инстансу (HTTP, TCP, gRPC). Если инстанс не отвечает на проверку, балансировщик исключает его из ротации, а группа инстансов пересоздаёт его (или масштабирует).

Горизонтальное масштабирование – увеличение количества экземпляров приложения (ВМ), а не мощности каждой ВМ (вертикальное масштабирование).

Метрика CPU > 70% – типовой порог для автомасштабирования: если средняя загрузка CPU в группе превышает 70% в течение N минут, добавляется новая ВМ.

4. Пример практического выполнения (Yandex Cloud)

4.1. Создание шаблона ВМ с nginx и скриптом hostname

Создадим **образ** ВМ через **Launch Template** или напрямую при создании группы.

User-data скрипт (cloud-init) для установки nginx и кастомной страницы:

```
bash
#!/bin/bash
apt update -y
apt install -y nginx
systemctl enable nginx
systemctl start nginx

# Создаём страницу, выводящую hostname и IP
HOSTNAME=$(hostname)
IP_ADDR=$(hostname -I | awk '{print $1}')
cat <<EOF > /var/www/html/index.html
<!DOCTYPE html>
<html>
<head><title>VM: $HOSTNAME</title></head>
<body>
<h1>Запрос обработан виртуальной машиной</h1>
<p>Hostname: <strong>$HOSTNAME</strong></p>
<p>IP-адрес: $IP_ADDR</p>
</body>
</html>
EOF

# Создаём эндпоинт /health для проверки здоровья
echo "OK" > /var/www/html/health
```

4.2. Создание управляемой группы инстансов

1. В консоли Yandex Cloud: **Compute Cloud → Группы инстансов → Создать группу**.
2. **Параметры группы**:
   - Имя: ig-lab2-http

- Шаблон: задать параметры ВМ (2 vCPU, 4 GB RAM, Ubuntu 22.04, диск 20 ГБ, публичный IP – не нужен, балансировщик будет внешним).
- Ключи: добавить публичный SSH-ключ.
- **Cloud-init** (вставить скрипт из п.4.1).

3. **Масштабирование**:
   - Размер группы: **фиксированный** или **автомасштабируемый**.
   - Выбрать **автомасштабирование**:
     - Минимальное количество ВМ: 2
     - Максимальное: 5
     - Метрика: CPU (порог > 70% в течение 2 минут)
     - Период измерения: 60 сек.
     - Количество ВМ для добавления: 1
4. **Зоны доступности**: выбрать 2 зоны (например, ru-central1-a, ru-central1-b).
5. **Интеграция с балансировщиком**: пока пропустить (создадим отдельно).

4.3. Создание балансировщика нагрузки

1. **Load Balancer → Создать балансировщик**.
   - Тип: **Внешний**.
   - Имя: lb-lab2.
2. **Обработчик (Listener)**:
   - Протокол: HTTP, порт 80.
   - Целевая группа: создать новую целевую группу.
   - **Проверка здоровья**: HTTP, путь /health, порт 80, интервал 5 сек, таймаут 2 сек, порог успеха 2, порог отказа 2.
3. **Подключить целевую группу** к группе инстансов:
   - При создании группы инстансов указать **целевую группу** балансировщика, либо добавить существующую группу инстансов в целевую группу после создания.

4.4. Нагрузочное тестирование (активация автомасштабирования)

Установка wrk на отдельной ВМ или локальной машине:

bash

```
sudo apt install wrk -y
```

Запуск нагрузки на публичный IP балансировщика:

bash

```
wrk -t4 -c100 -d120s http://<LB_PUBLIC_IP>/
```

- -t4 – 4 потока
- -c100 – 100 одновременных соединений
- -d120s – длительность 2 минуты

Ожидаемый результат: Через 2–3 минуты после превышения CPU > 70% группа инстансов создаст дополнительную ВМ (размер группы увеличится с 2 до 3). График в мониторинге покажет рост CPU и увеличение количества инстансов.

4.5. Проверка отказоустойчивости

1. Найти в консоли **ВМ** в составе группы.
2. Остановить одну ВМ вручную (или удалить).
3. Наблюдать: балансировщик перестаёт отправлять трафик на эту ВМ (через health check), группа инстансов автоматически создаёт новую ВМ взамен.
4. В течение всего процесса сервис остаётся доступен по IP балансировщика.

4.6. Скриншоты для отчёта

- Шаблон ВМ (параметры и cloud-init).
- Группа инстансов с правилами автомасштабирования.
- Балансировщик нагрузки с настройками health check.
- График загрузки CPU и количества инстансов до/во время/после теста.
- Вывод wrk или ab.
- Демонстрация отказоустойчивости: curl http://<LB_IP>/ несколько раз – hostname меняется (если sticky sessions не включён).

5. Индивидуальные задания (15 вариантов)

Каждый вариант – базовая часть (создание группы, балансировщика, автошкалирование по CPU) + **дополнительное требование**.

№ вар.	Дополнительное задание (вариативная часть)
1	Настроить **autoscaling по расписанию**: с 9:00 до 18:00 – минимум 3 ВМ, в остальное время – 1 ВМ. Показать расписание.
2	Вместо CPU использовать метрику **сетевого трафика** (входящие байты > 10 МБ/с в течение 1 мин). Настроить и проверить.
3	Добавить **второй балансировщик** – внутренний (для gRPC), настроить проверку здоровья по gRPC. Показать оба.
4	Реализовать **session affinity** (sticky sessions) на балансировщике на основе cookie. Проверить, что запросы от одного клиента попадают на одну ВМ.
5	Настроить **автомасштабирование на основе очереди сообщений** (например, длина очереди RabbitMQ или Yandex Message Queue). Создать тестовый продюсер.
6	Вместо nginx использовать **Node.js приложение** (простой HTTP-сервер, выводящий hostname). Написать Dockerfile, собрать образ, запустить на ВМ через Docker.
7	Создать **группу с ВМ на Windows Server** (IIS вместо nginx). Настроить автомасштабирование и балансировку HTTP.
8	Добавить **TLS-терминацию на балансировщике**: порт 443, самоподписанный сертификат. Перенаправлять HTTP→HTTPS.
9	Настроить **health check с пользовательским HTTP-заголовком** (X-Health-Request: true) и проверкой кода ответа 200.
10	Использовать **Terraform** для описания всей инфраструктуры (шаблон, группа, балансировщик). Приложить файлы .tf.

№ вар.	Дополнительное задание (вариативная часть)
11	Интегрировать с **Cloud Monitoring и алертом**: создать алерт, который срабатывает при увеличении размера группы >3. В отчёте скриншот алерта.
12	Настроить **межзональное размещение** (принудительно: по 1 ВМ в каждой из 3 зон доступности) и показать распределение.
13	Реализовать **канареечное развертывание (canary)**: сначала обновить 20% ВМ до новой версии приложения, остальные оставить старыми. Балансировщик направляет трафик на обе версии.
14	Создать **глобальный балансировщик** (Network Load Balancer) для распределения трафика между группами в разных регионах (если поддерживается провайдером).
15	Написать **скрипт на Python (boto3)**, который создаёт группу инстансов и балансировщик в AWS (Auto Scaling Group + ALB). Приложить код.

6. Контрольные вопросы (для защиты)

1. В чём разница между горизонтальным и вертикальным масштабированием? Когда какой подход предпочтительнее?
2. Что такое **шаблон ВМ**? Почему он важен для управляемых групп?
3. Какие метрики (помимо CPU) можно использовать для автомасштабирования в облаке?
4. Как балансировщик нагрузки узнаёт, что инстанс «здоров»? Опишите механизм health check.
5. Что произойдёт, если все инстансы в группе станут нездоровыми (failed health check)?
6. Какие типы балансировщиков существуют (L4/L7, внешний/внутренний)? В каких случаях применяется L7?
7. Какой алгоритм распределения трафика используется в вашей лабораторной работе? Можно ли его изменить?

8. Что такое **cooldown period** в автомасштабировании? Зачем он нужен?

9. Как проверить отказоустойчивость приложения без остановки всех ВМ?

10. Почему при использовании балансировщика ВМ обычно не назначают публичный IP? Каковы преимущества?

7. Требования к отчёту

Отчёт оформляется согласно ГОСТ (шрифт 14 пт, полуторный интервал). Содержит:

1. **Титульный лист**.
2. **Цель и задачи** (из п.1–2).
3. **Теоретическое обоснование** (определения: Instance Group, Load Balancer, health check, горизонтальное масштабирование).
4. **Практическая часть**:
   - Пошаговые действия с командами и настройками (облачная консоль или CLI/Terraform).
   - Скриншоты: шаблон ВМ, создание группы с правилами масштабирования, балансировщик с health check, график автомасштабирования (метрики CPU и количество инстансов), вывод нагрузочного теста.
   - Для вариативной части – дополнительные скриншоты и пояснения.
5. **Результаты выполнения** (таблица: время теста, количество ВМ, средняя загрузка CPU, RPS, latency).
6. **Анализ результатов** (сработало ли автомасштабирование? почему да/нет? как проверили отказоустойчивость?).
7. **Ответы на контрольные вопросы** (письменно).
8. **Вывод** (освоенные навыки: управление группами, балансировка, автомасштабирование, тестирование нагрузки).

8. Критерии оценивания (по 5-балльной шкале)

Оценка	Критерии
«Отлично»	Выполнена **базовая + вариативная части**. Группа инстансов создана корректно, автомасштабирование по CPU срабатывает при нагрузочном тесте. Балансировщик работает, health check настроен. Отказоустойчивость подтверждена. Отчёт полный, все скриншоты на месте, выводы аналитические. Защита: уверенные ответы на все вопросы.
«Хорошо»	Выполнена **только базовая часть** (автомасштабирование по CPU без вариативного задания). Основной функционал работает, но, возможно, health check настроен неоптимально или тест проведён не полностью. Отчёт без грубых ошибок. Защита: ответы на большинство вопросов.
«Удовлетворительно»	Работа выполнена частично: группа инстансов создана, но автомасштабирование не срабатывает (или не настроено), балансировщик есть, но health check неверный. Отчёт неполный (например, нет графика нагрузки). Студент плохо отвечает на вопросы.

Оценка	Критерии
«Неудовлетворительно»	Группа инстансов не создана / балансировщик отсутствует / сервис не доступен. Нет доказательств автомасштабирования. Отчёт не предоставлен или не соответствует требованиям.

Лабораторная работа № 3

Объектное хранилище: загрузка и раздача файлов

1. Цель работы

Освоить S3-совместимое облачное объектное хранилище: создание бакетов, управление объектами, настройка версионирования, генерация временных ссылок (pre-signed URL), организация статического хостинга сайта, применение политик доступа.

2. Задачи работы

1. Создать бакет (bucket) в объектном хранилище.
2. Изучить основные операции с объектами:
   - загрузка (upload) файлов через веб-консоль и CLI (AWS CLI / yc CLI);
   - скачивание (download);
   - удаление.
3. Включить **версионирование** (versioning) бакета, проверить восстановление предыдущей версии.
4. Сгенерировать **pre-signed URL** для предоставления временного доступа к приватному объекту (срок действия 1 час).
5. Настроить **статический хостинг** сайта из бакета:
   - загрузить index.html и error.html;
   - задать индексную страницу и страницу ошибки в настройках бакета;
   - сделать бакет публичным (или настроить политику доступа).
6. Настроить **политику бакета** (bucket policy), ограничивающую доступ по IP-адресу (разрешить доступ только с определённого IP).
7. Зафиксировать результаты в отчёте (скриншоты, команды CLI, ссылка на опубликованный статический сайт).

3. Теоретическое обоснование

Объектное хранилище (Object Storage) – масштабируемое хранилище неструктурированных данных (файлы, бэкапы, образы, логи, статика). Данные хранятся в виде **объектов** (файл + метаданные), каждый объект идентифицируется уникальным **ключом (key)** внутри **бакета (bucket)**. Типичный пример – Amazon S3 и совместимые реализации (Yandex Object Storage, VK Cloud Storage, SberCloud Object Storage).

Ключевые понятия:

- **Бакет** – контейнер для объектов (имя глобально уникально в рамках облака).
- **Объект** – файл с метаданными (Content-Type, размер, ETag и пр.).
- **Классы хранения** – стандартный, холодный (Infrequent Access), архивный (Glacier) – различаются стоимостью и временем доступа.
- **Версионирование** – сохранение нескольких версий одного объекта. Позволяет восстановить удалённый или изменённый объект.
- **Pre-signed URL** – временная ссылка на приватный объект, сгенерированная с помощью ключа доступа. Позволяет предоставить ограниченный по времени доступ без раскрытия учётных данных.
- **Статический хостинг** – возможность хостить статический сайт (HTML, CSS, JS, изображения) прямо из бакета. Бакет становится публично доступным по HTTP/HTTPS с поддержкой индексной страницы и страницы ошибок.
- **Bucket Policy** – JSON-документ, определяющий права доступа к бакету и объектам (на основе IAM, IP, Referer и т.д.).

4. Пример практического выполнения (Yandex Cloud + AWS CLI)

Для единообразия используем **Yandex Object Storage** (S3-совместимый). Команды аналогичны для AWS S3 (отличаются эндпоинтом).

4.1. Создание бакета через веб-консоль

1. В консоли Yandex Cloud: **Object Storage → Бакеты → Создать бакет**.
2. Имя бакета: lab3-student-{номер} (глобально уникальное).
3. Класс хранилища: **Стандартное**.
4. Доступ: **Ограниченный** (пока закрытый).
5. Создать.

4.2. Установка и настройка AWS CLI (для S3-совместимого API)

bash

```
# Установка (Ubuntu)
sudo apt install awscli -y

# Настройка профиля для Yandex Cloud
aws configure --profile yc
# AWS Access Key ID: <ваш статический ключ из Yandex Cloud>
# AWS Secret Access Key: <секретный ключ>
# Default region: ru-central1
# Default output format: json

# Проверка: список бакетов
aws --endpoint-url https://storage.yandexcloud.net s3 ls --profile yc
```

4.3. Загрузка файлов через CLI и консоль

Через CLI (загрузка произвольного файла):

bash

```
echo "Hello from Object Storage" > test.txt
aws --endpoint-url https://storage.yandexcloud.net s3 cp test.txt s3://lab3-student-{номер}/test.txt --profile yc
```

Через веб-консоль:

Зайти в бакет → **Загрузить** → выбрать файлы (например, photo.jpg).

4.4. Включение версионирования

bash

```
# Включение версионирования
aws --endpoint-url https://storage.yandexcloud.net s3api put-bucket-versioning \
  --bucket lab3-student-{номер} \
  --versioning-configuration Status=Enabled \
  --profile yc

# Проверка
aws --endpoint-url https://storage.yandexcloud.net s3api get-bucket-versioning \
  --bucket lab3-student-{номер} --profile yc
```

Демонстрация версионирования:

1. Загрузить файл info.txt версия 1.
2. Изменить содержимое и загрузить снова.
3. В консоли Object Storage → бакет → **Показать версии**.
4. Скачать предыдущую версию.

4.5. Pre-signed URL (временная ссылка)

Сгенерировать ссылку, действительную в течение 1 часа, на приватный объект test.txt:

bash

```
aws --endpoint-url https://storage.yandexcloud.net s3 presign \
  s3://lab3-student-{номер}/test.txt \
  --expires-in 3600 \
  --profile yc
```

Результат – URL вида https://storage.yandexcloud.net/...?X-Amz-.... Проверить: открыть ссылку в браузере (через 1 час доступ закроется).

4.6. Статический хостинг

Создание HTML-файлов:

index.html:

html

```
<!DOCTYPE html>
<html>
<head><title>ЛР3 - Объектное хранилище</title></head>
<body>
<h1>Статический сайт из бакета Object Storage</h1>
<p>Выполнил: <ваше ФИО>, группа X</p>
</body>
</html>
```

error.html:

```
html
<html><body><h1>Ошибка 404 – страница не найдена</h1></body></html>
```

Загрузка в бакет:

```
bash
aws --endpoint-url https://storage.yandexcloud.net s3 cp index.html s3://lab3-student-{номер}/index.html --profile yc
aws --endpoint-url https://storage.yandexcloud.net s3 cp error.html s3://lab3-student-{номер}/error.html --profile yc
```

Настройка статического хостинга через CLI:

```
bash
# Применяем конфигурацию
aws --endpoint-url https://storage.yandexcloud.net s3api put-bucket-website \
  --bucket lab3-student-{номер} \
  --website-configuration '{
    "IndexDocument": {"Suffix": "index.html"},
    "ErrorDocument": {"Key": "error.html"}
  }' \
  --profile yc
```

Делаем бакет публичным (для статического сайта необходимо публичное чтение объектов):

Либо через политику бакета:

bash

```
cat > policy.json <<EOF
{
 "Version": "2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Principal": "*",
   "Action": "s3:GetObject",
   "Resource": "arn:aws:s3:::lab3-student-{номер}/*"
  }
 ]
}
EOF

aws --endpoint-url https://storage.yandexcloud.net s3api put-bucket-policy \
  --bucket lab3-student-{номер} \
  --policy file://policy.json \
  --profile yc
```

Доступ к сайту:

- Эндпоинт статического хостинга: https://lab3-student-{номер}.website.yandexcloud.net
- Или http://lab3-student-{номер}.website.yandexcloud.net

4.7. Политика бакета: доступ только с определённого IP

Создать политику, разрешающую чтение объектов только с IP-адреса 123.123.123.123 (заменить на свой IP, который можно узнать через curl ifconfig.me).

json

```
{
```

```
"Version": "2012-10-17",
"Statement": [
 {
  "Effect": "Allow",
  "Principal": "*",
  "Action": "s3:GetObject",
  "Resource": "arn:aws:s3:::lab3-student-{номер}/*",
  "Condition": {
   "IpAddress": {
    "aws:SourceIp": "123.123.123.123/32"
   }
  }
 }
]
}
```

Применить политику (предварительно отключив публичный доступ, если был):

bash

```
aws --endpoint-url https://storage.yandexcloud.net s3api put-bucket-policy \
  --bucket lab3-student-{номер} \
  --policy file://ip-policy.json \
  --profile yc
```

Проверить: доступ к объекту через браузер возможен только с разрешённого IP, с другого IP – ошибка 403.

4.8. Скриншоты для отчёта

- Список бакетов с созданным бакетом.
- Версионирование: список версий одного объекта.
- Команда `aws s3 presign` и полученный URL в браузере.
- Настройки статического хостинга в консоли (вкладка «Веб-сайт»).
- Открытый статический сайт в браузере.

- Применённая bucket policy (JSON) и результат доступа с разных IP.

5. Индивидуальные задания (15 вариантов)

Базовое задание (создание бакета, загрузка файлов, версионирование, pre-signed URL, статический хостинг) + вариативная часть.

№ вар.	Дополнительное задание (вариативная часть)
1	Настроить **жизненный цикл объектов** (lifecycle rule): все файлы с расширением .log через 7 дней перемещать в холодное хранилище (Infrequent Access), через 30 дней удалять.
2	Создать **корзину с SSE-S3 шифрованием** (серверное шифрование) и загрузить файл, проверить, что он зашифрован (метаданные x-amz-server-side-encryption).
3	Настроить **CORS** для бакета, чтобы статический сайт мог делать AJAX-запросы к своему же бакету. Показать preflight OPTIONS запрос.
4	Использовать **AWS CLI** для синхронизации локальной папки с бакетом (aws s3 sync). Показать инкрементальную загрузку.
5	Создать **политику бакета**, запрещающую удаление объектов пользователям, кроме одного сервисного аккаунта (использовать Deny). Проверить.
6	Настроить **event-уведомления**: при загрузке нового объекта в бакет отправлять сообщение в очередь (Yandex Message Queue / AWS SQS). В отчёте скриншот сообщения.

№ вар.	Дополнительное задание (вариативная часть)
7	Реализовать **копирование объектов между бакетами** (внутри одного облака). Показать копирование с изменением класса хранения.
8	Добавить **теги** (tags) к объектам и выполнить фильтрацию по тегам через CLI (например, список всех объектов с тегом project=lab3).
9	Сгенерировать **pre-signed URL для PUT-запроса** (чтобы разрешить пользователю загрузить файл без авторизации, но с ограничением по времени). Проверить через curl -X PUT.
10	В статическом сайте использовать **кастомную страницу ошибки 403** (Forbidden) – создать 403.html и настроить в политике.
11	Создать **S3-совместимое хранилище у альтернативного провайдера** (VK Cloud / SberCloud) и повторить базовые операции. Сравнить эндпоинты.
12	Настроить **доступ к бакету через Cloud CDN** (Content Delivery Network): подключить CDN к бакету, выдать доменное имя. Показать заголовки ответа.
13	Реализовать **автоматическую распаковку архивов** через серверную функцию (Object Storage + Cloud Function): при загрузке ZIP-файла автоматически распаковывать его в тот же бакет.

№ вар.	Дополнительное задание (вариативная часть)
14	Настроить **журналирование доступа** (access logging) для бакета: логи запросов сохранять в другой бакет. Проанализировать один лог-файл.
15	Использовать **Terraform** для описания бакета, версионирования, политики и статического хостинга. Приложить файлы .tf и вывод terraform apply.

6. Контрольные вопросы (для защиты)

1. В чём принципиальное отличие объектного хранилища от файловой системы (иерархия vs плоское пространство)?
2. Что такое **S3-совместимость**? Назовите двух облачных провайдеров в РФ, поддерживающих S3 API.
3. Как включить версионирование бакета? Можно ли его отключить после включения?
4. Что такое pre-signed URL? Какие параметры в него входят (как генерируется подпись)?
5. Какие есть классы хранения в Object Storage? Когда применяется «холодное» хранилище?
6. Для чего нужен статический хостинг? Какие ограничения у такого хостинга по сравнению с полноценным веб-сервером?
7. Как настроить статический хостинг, чтобы индексной страницей был не index.html, а home.html?
8. В чём разница между **bucket policy** и **IAM-политиками**? Когда что использовать?
9. Как ограничить доступ к объектам по IP-адресу? Какие риски при использовании aws:SourceIp?

10. Что такое **CORS** и когда его нужно настраивать для Object Storage?

7. Требования к отчёту

Отчёт оформляется по ГОСТ. Содержит:

1. **Титульный лист**.
2. **Цель и задачи**.
3. **Теоретическое обоснование** (понятия: бакет, объект, версионирование, pre-signed URL, статический хостинг, bucket policy).
4. **Практическая часть**:
   - Создание бакета (скриншот).
   - Команды CLI для загрузки файлов, включения версионирования, генерации pre-signed URL (с результатами выполнения).
   - Настройка статического хостинга: файлы HTML, конфигурация, публичная ссылка на сайт (скриншот).
   - Применение bucket policy с ограничением по IP (скриншот политики и проверка доступности с разрешённого/запрещённого IP).
   - **Вариативная часть** – дополнительные скриншоты/код.
5. **Результаты выполнения** (таблица: операции, команды, полученные URL, статус-коды).
6. **Анализ результатов** (сравнение public-доступа и pre-signed URL, надёжность версионирования, эффективность статического хостинга).
7. **Ответы на контрольные вопросы**.
8. **Вывод** (освоены операции с объектным хранилищем, навыки работы с S3 API, организация статического сайта, управление доступом).

8. Критерии оценивания

Оценка	Критерии
«Отлично»	Выполнены **базовая + вариативная части**. Бакет создан, версионирование включено, pre-signed URL работает, статический сайт доступен по публичной ссылке, bucket policy ограничивает доступ по IP. Отчёт содержит все скриншоты, команды CLI, ссылки. Защита: полные ответы на все вопросы.
«Хорошо»	Выполнена **только базовая часть**. Все основные операции есть, но вариативное задание не выполнено. Возможны незначительные недочёты (например, pre-signed URL сгенерирован, но не проверен). Отчёт без грубых ошибок.
«Удовлетворительно»	Работа выполнена частично: бакет создан, но, например, нет версионирования или статический хостинг не настроен. Pre-signed URL не демонстрируется. Отчёт неполный.
«Неудовлетворительно»	Бакет не создан, операции с CLI не выполнены, нет доказательств работы статического хостинга. Отчёт отсутствует или не соответствует заданию.

Лабораторная работа № 4

Введение в Terraform: создание виртуальной машины

1. Цель работы

Освоить базовые принципы инфраструктуры как код (IaC — Infrastructure as Code) с использованием инструмента Terraform, научиться описывать облачные ресурсы декларативно, управлять их жизненным циклом (создание, изменение, удаление).

2. Задачи работы

1. Установить Terraform и настроить провайдер для облачной платформы (Yandex Cloud / AWS).
2. Написать конфигурационный файл Terraform для создания виртуальной машины с заданными параметрами (имя, зона доступности, образ ОС, ресурсы CPU/RAM, диск).
3. Объявить переменные (vm_name, zone, image_id, public_ssh_key и др.) для параметризации конфигурации.
4. Использовать **output** (выходные переменные) для вывода публичного IP-адреса созданной ВМ.
5. Выполнить команды жизненного цикла: terraform init (инициализация), terraform plan (план изменений), terraform apply (применение).
6. Подключиться к созданной ВМ по SSH, убедиться в её работоспособности.
7. Удалить инфраструктуру через terraform destroy.
8. Оформить отчёт с полным кодом .tf-файлов, листингами выполнения команд и скриншотами.

3. Теоретическое обоснование

Infrastructure as Code (IaC) – подход к управлению инфраструктурой (серверы, сети, диски, базы данных) с помощью описательных конфигурационных файлов, а не ручного создания через веб-консоль или скриптов. Преимущества: воспроизводимость, версионирование, автоматизация, повторное использование.

Terraform – инструмент с открытым исходным кодом от HashiCorp, реализующий декларативный подход: вы описываете **желаемое состояние** инфраструктуры, а Terraform вычисляет разницу и приводит реальное состояние к желаемому.

Ключевые понятия Terraform:

- **Провайдер (Provider)** – плагин, обеспечивающий взаимодействие с API конкретного облака (Yandex Cloud, AWS, Google Cloud, Azure и др.).
- **Ресурс (Resource)** – компонент инфраструктуры (ВМ, диск, сеть, группа безопасности). Синтаксис: `resource "тип" "локальное_имя" { параметры }`.
- **Переменная (Variable)** – параметр, значение которого можно задать при запуске (`-var` или файл `.tfvars`). Уменьшает дублирование кода.
- **Выходная переменная (Output)** – значение, которое Terraform показывает после `apply` (например, IP-адрес ВМ).
- **Состояние (State)** – файл `terraform.tfstate`, где Terraform хранит соответствие между описанными ресурсами и реальными объектами в облаке.

Жизненный цикл Terraform:

- `terraform init` – загрузка провайдеров, инициализация бэкенда.
- `terraform plan` – показывает, какие ресурсы будут созданы, изменены или удалены.
- `terraform apply` – применяет изменения, создавая/обновляя инфраструктуру.
- `terraform destroy` – удаляет все ресурсы, описанные в конфигурации.

Преимущества Terraform перед ручным созданием или скриптами (CLI/API): декларативность, идемпотентность (повторный apply не ломает инфраструктуру), управление зависимостями между ресурсами.

4. Пример практического выполнения (Yandex Cloud + Terraform)

4.1. Установка Terraform

Linux/macOS:

bash

```
wget -O terraform.zip https://releases.hashicorp.com/terraform/1.8.0/terraform_1.8.0_linux_amd64.zip
unzip terraform.zip
sudo mv terraform /usr/local/bin/
terraform --version
```

Windows: скачать с официального сайта и добавить в PATH.

4.2. Настройка провайдера Yandex Cloud

Требуется предварительно:

- Зарегистрироваться в Yandex Cloud, создать каталог.
- Получить **OAuth-токен** (для аутентификации) или статический ключ доступа (сервисный аккаунт). Рекомендуется: создать сервисный аккаунт с ролью editor и создать статический ключ доступа (access_key, secret_key).

Структура файлов:

text

```
lab4-terraform/
├── provider.tf      # провайдер и настройки
├── variables.tf     # объявления переменных
├── main.tf          # основные ресурсы (ВМ, сеть)
├── outputs.tf       # выходные значения
└── terraform.tfvars # значения переменных (не в репозиторий)
```

4.3. Код конфигурации

provider.tf:

hcl

```
terraform {
  required_providers {
    yandex = {
      source  = "yandex-cloud/yandex"
      version = "~> 0.110"
    }
  }
}

provider "yandex" {
  token     = var.yc_token      # OAuth-токен или статический ключ
  cloud_id  = var.yc_cloud_id
  folder_id = var.yc_folder_id
  zone      = var.zone
}
```

variables.tf:

hcl

```
variable "yc_token" {
  description = "Yandex Cloud OAuth token or static key"
  sensitive   = true
}

variable "yc_cloud_id" {
  description = "Yandex Cloud ID"
}

variable "yc_folder_id" {
  description = "Yandex Cloud Folder ID"
```

```
}

variable "zone" {
  description = "Availability zone"
  default     = "ru-central1-a"
}

variable "vm_name" {
  description = "Name of the VM"
  default     = "terraform-vm"
}

variable "public_ssh_key" {
  description = "SSH public key for VM access"
  sensitive   = true
}

variable "image_id" {
  description = "Ubuntu 22.04 image ID"
  default     = "fd8m8g1to9nqe2r7gdfm"  # актуальный ID для Yandex Clou
d

}
```

main.tf (создание ВМ, необходимые ресурсы: сеть, подсеть):

```
hcl
# Создание облачной сети
resource "yandex_vpc_network" "lab4_net" {
  name = "lab4-network"
}

# Создание подсети в заданной зоне
```

```
resource "yandex_vpc_subnet" "lab4_subnet" {
  name           = "lab4-subnet"
  zone           = var.zone
  network_id     = yandex_vpc_network.lab4_net.id
  v4_cidr_blocks = ["10.10.0.0/24"]
}

# Группа безопасности (порты 22 и 80)
resource "yandex_vpc_security_group" "ssh-http" {
  name       = "sg-terraform-lab4"
  network_id  = yandex_vpc_network.lab4_net.id

  ingress {
    protocol       = "TCP"
    description    = "SSH"
    v4_cidr_blocks = ["0.0.0.0/0"]
    port           = 22
  }
  ingress {
    protocol       = "TCP"
    description    = "HTTP"
    v4_cidr_blocks = ["0.0.0.0/0"]
    port           = 80
  }
  egress {
    protocol       = "ANY"
    description    = "any"
    v4_cidr_blocks = ["0.0.0.0/0"]
    from_port      = 0
    to_port        = 65535
```

```
  }
}

# Создание виртуальной машины
resource "yandex_compute_instance" "vm_lab4" {
  name        = var.vm_name
  platform_id = "standard-v3"
  zone        = var.zone

  resources {
    cores  = 2
    memory = 4
  }

  boot_disk {
    initialize_params {
      image_id = var.image_id
      size     = 20
    }
  }

  network_interface {
    subnet_id          = yandex_vpc_subnet.lab4_subnet.id
    security_group_ids = [yandex_vpc_security_group.ssh-http.id]
    nat                = true   # публичный IP
  }

  metadata = {
    ssh-keys = "ubuntu:${var.public_ssh_key}"
  }
```

```
}
```

outputs.tf:

hcl

```
output "vm_external_ip" {
  description = "Public IP address of the VM"
  value       = yandex_compute_instance.vm_lab4.network_interface[0].nat_ip_address
}

output "vm_internal_ip" {
  description = "Internal IP address of the VM"
  value       = yandex_compute_instance.vm_lab4.network_interface[0].ip_address
}
```

terraform.tfvars (создать вручную, не коммитить):

hcl

```
yc_token     = "ваш_токен"
yc_cloud_id  = "b1gabc..."
yc_folder_id = "b1gdef..."
public_ssh_key = "ssh-rsa AAAAB3N... user@host"
vm_name      = "lab4-tf-vm"
zone         = "ru-central1-a"
```

4.4. Выполнение Terraform

bash

```
terraform init
terraform plan
terraform apply
```

После apply увидим вывод:

text

```
Apply complete! Resources: 4 added, 0 changed, 0 destroyed.
```

Outputs:

vm_external_ip = "84.201.xxx.xxx"

vm_internal_ip = "10.10.0.5"

4.5. Подключение к ВМ по SSH и проверка

bash

ssh -i ~/.ssh/id_rsa ubuntu@<vm_external_ip>

sudo apt update && sudo apt install nginx -y

curl http://localhost

4.6. Удаление инфраструктуры

bash

terraform destroy

4.7. Скриншоты для отчёта

- Вывод terraform plan (план создания 4 ресурсов).
- Вывод terraform apply с полученными IP.
- Подключение по SSH к ВМ (терминал).
- Листинг .tf-файлов (код).
- Вывод terraform destroy после завершения.

5. Индивидуальные задания (15 вариантов)

Базовое задание (создание ВМ через Terraform с переменными, output, SSH) + вариативная часть.

№ вар.	Дополнительное задание (вариативная часть)
1	Добавить **второй диск** (volume) размером 10 ГБ и автоматически смонтировать его при создании ВМ (через user_data или provisioner remote-exec).

№ вар.	Дополнительное задание (вариативная часть)
2	Использовать **data source** для получения последнего образа Ubuntu 22.04 (вместо жестко закодированного image_id). Показать конфигурацию.
3	Создать не одну, а **несколько ВМ** с помощью count (например, 3 ВМ). Вывести их IP-адреса списком.
4	Реализовать **зависимость между ресурсами** явно через depends_on или неявно (ссылки). Прокомментировать.
5	Вынести переменные в **отдельный файл** variables.auto.tfvars и показать, как переопределить значения через -var флаг.
6	Добавить **метаданные** (metadata), где передать user-data (cloud-init) для автоматической установки nginx. Проверить доступность по HTTP.
7	Настроить **terraform remote state** (бэкенд) в Object Storage (S3). Показать конфигурацию бэкенда и выполнить миграцию состояния.
8	Использовать **модули** Terraform: создать свой модуль vm_with_nginx и вызвать его в main.tf.
9	Добавить **provisioner** (remote-exec или local-exec) для вывода сообщения после создания ВМ.
10	Создать **группу безопасности через** dynamic **блок** (передать список портов переменной).
11	Использовать terraform taint для пометки ресурса на пересоздание, показать процесс.
12	Добавить **условное создание ресурса** с помощью count = var.create_vm ? 1 : 0. Проверить оба варианта.

№ вар.	Дополнительное задание (вариативная часть)
13	Настроить **переменные типа** map **и** list для параметров ВМ (например, разные зоны и имена).
14	Создать **output** в формате JSON (команда terraform output -json). Показать вывод.
15	Развернуть ту же инфраструктуру в **AWS** (провайдер aws), адаптировать ресурсы aws_instance, aws_security_group. Приложить оба варианта.

6. Контрольные вопросы (для защиты)

1. Что такое Infrastructure as Code (IaC)? Назовите два популярных инструмента.
2. Чем декларативный подход (Terraform) отличается от императивного (Ansible, скрипты)?
3. Назначение файла terraform.tfstate. Почему его нельзя хранить в локальной папке в командной работе?
4. Какие команды Terraform нужно выполнить, чтобы создать инфраструктуру с нуля?
5. В чём разница между terraform plan и terraform apply?
6. Как передать переменную в Terraform без указания значения в файле .tfvars?
7. Что произойдёт, если изменить image_id у существующей ВМ и выполнить terraform apply?
8. Как получить идентификатор созданного ресурса (например, ВМ) после apply?
9. Что такое output и для чего он нужен?
10. Как удалить только один ресурс из конфигурации (не все), не используя destroy?

7. Требования к отчёту

Отчёт оформляется по ГОСТ. Содержит:

1. **Титульный лист**.
2. **Цель и задачи работы**.
3. **Теоретическое обоснование** (понятия: IaC, Terraform, провайдер, ресурс, переменная, output, state).
4. **Практическая часть**:
   - Установка и проверка Terraform.
   - Полный листинг .tf-файлов (4–5 файлов) с комментариями.
   - Выполнение terraform init, plan, apply (копия вывода команд).
   - Скриншот подключения по SSH к ВМ (или проверка через curl).
   - Вывод terraform destroy (для подтверждения удаления).
   - **Вариативная часть** – дополнительные файлы/команды.
5. **Результаты выполнения** (таблица: ресурсы, их параметры, полученные IP).
6. **Анализ результатов** (сравнение ручного создания ВМ и через Terraform, преимущества IaC).
7. **Ответы на контрольные вопросы**.
8. **Вывод** (освоены основы Terraform, умение описывать ВМ, сеть, переменные, управлять состоянием).

8. Критерии оценивания

Оценка	Критерии
«Отлично»	Выполнены **базовая + вариативная части**. Конфигурация Terraform корректна, terraform apply создаёт ВМ, SSH-доступ работает. Использованы переменные и output. Код структурирован,

Оценка	Критерии
	отформатирован (terraform fmt). Отчёт содержит все листинги и скриншоты. Защита: глубокие ответы.
«Хорошо»	Выполнена **только базовая часть**. ВМ создаётся, но, возможно, не использован terraform.tfvars или output выводит не все нужные параметры. Вариативное задание не выполнено. Незначительные недочёты в коде.
«Удовлетворительно»	Работа выполнена частично: конфигурация есть, но ВМ не создаётся (ошибки провайдера, неверные переменные), либо SSH недоступен. Отчёт неполный, нет скриншотов apply.
«Неудовлетворительно»	Terraform не установлен, конфигурация отсутствует или не работает. Отчёт не предоставлен.

Лабораторная работа № 5
Управляемая база данных PostgreSQL в облаке

1. Цель работы

Научиться создавать управляемый кластер реляционной базы данных (Managed PostgreSQL) в облаке, настраивать сете доступ, подключаться к БД из приложения с использованием SSL, выполнять базовые операции (создание таблиц, вставка, выборка) и управлять резервным копированием.

2. Задачи работы

1. Создать кластер **Managed PostgreSQL** (один хост-мастер) в облачной платформе.
2. Создать базу данных, пользователя и назначить пароль.
3. Настроить группу безопасности для доступа к кластеру по порту 5432 (только с разрешённых IP-адресов).
4. Разработать простое приложение на **Python** (или Node.js), которое:
   - подключается к БД по SSL;
   - создаёт таблицу users;
   - вставляет несколько записей (не менее 3);
   - выполняет SELECT и выводит результат в консоль.
5. Создать **ручную резервную копию** кластера через консоль облака.
6. Оформить отчёт со скриншотами конфигурации кластера, кодом приложения и выводом запросов.

3. Теоретическое обоснование

Управляемая база данных (Managed Database) — сервис облачного провайдера, который берёт на себя администрирование СУБД: установка, патчинг, резервное копирование, репликация, мониторинг, автоматический

failover. Пользователь управляет только схемой данных, пользователями и запросами.

Преимущества Managed DB перед self-hosted:

- автоматические ежедневные бэкапы с возможностью PITR (Point-in-Time Recovery);
- автоматическое обновление версий СУБД;
- масштабирование (увеличение CPU/RAM/диска без простоя);
- отказоустойчивость (автоматическое переключение на реплику);
- встроенный мониторинг метрик.

Ключевые параметры создания кластера:

- **Класс хоста** (CPU / RAM) — влияет на производительность.
- **Тип диска** (SSD, HDD) и размер.
- **Зона доступности** — размещение в одном дата-центре.
- **Версия PostgreSQL** (например, 15, 16).
- **SSL** — обязательное шифрование соединения в production.

Подключение приложения:

- host = публичный или приватный IP кластера (для внешних подключений требуется включить публичный доступ и настроить группы безопасности).
- порт = 5432 (стандартный для PostgreSQL).
- аутентификация: пользователь / пароль.
- SSL-сертификат — необходимо скачать сертификат CA (например, для Yandex Cloud это https://storage.yandexcloud.net/cloud-certs/CA.pem) и указать его в параметрах соединения.

Группы безопасности (Security Groups):

- Разрешают входящий трафик на порт 5432 только с определённых IP (например, IP рабочей станции студента или подсети VPC).

Резервное копирование:

- Автоматическое (ежедневное, в заданное окно) с хранением до N дней.

- Ручное (снимок) — создаётся по требованию, хранится отдельно.

4. Пример практического выполнения (Yandex Cloud + Python)

4.1. Создание кластера Managed PostgreSQL

1. В консоли Yandex Cloud: **Managed Databases → PostgreSQL → Создать кластер**.
2. Параметры:
   - Имя: lab5-postgres-cluster
   - Версия: 15
   - Класс хоста: s2.micro (2 vCPU, 8 GB RAM) — подойдёт для лабораторной.
   - Тип диска: network-ssd, размер 20 ГБ.
   - Количество хостов: 1 (мастер).
   - Зона доступности: ru-central1-a.
   - Публичный доступ: **Включён** (чтобы подключаться с рабочей станции).
3. Создать **базу данных**:
Имя БД: lab5db
4. Создать **пользователя**:
Имя: appuser, пароль: сгенерировать надёжный.
5. Настроить группу безопасности (создать новую или добавить правило в существующую):
   - Входящий трафик: порт 5432, протокол TCP, источник — ваш IP (можно узнать через curl ifconfig.me).

4.2. Получение SSL-сертификата

bash

```
mkdir ~/.postgresql
wget "https://storage.yandexcloud.net/cloud-certs/CA.pem" -O ~/.postgresql/root.crt
chmod 0600 ~/.postgresql/root.crt
```

4.3. Пример приложения на Python (psycopg2)

Установка зависимостей:

bash

```
pip install psycopg2-binary
```

Код приложения app.py**:**

python

```
import psycopg2
import ssl
import os

# Параметры подключения
DB_HOST = "ваш_хост_мастера"  # например, rc1a-xxxxx.mdb.yandexcloud.net
DB_PORT = 5432
DB_NAME = "lab5db"
DB_USER = "appuser"
DB_PASSWORD = "ваш_пароль"

# Путь к SSL-сертификату
SSL_CERT = os.path.expanduser("~/.postgresql/root.crt")

def main():
    conn = None
    try:
        # Подключение с SSL
        conn = psycopg2.connect(
            host=DB_HOST,
            port=DB_PORT,
            dbname=DB_NAME,
            user=DB_USER,
```

```
        password=DB_PASSWORD,
        sslmode="require",
        sslrootcert=SSL_CERT
    )
    conn.autocommit = False
    cur = conn.cursor()

    # Создание таблицы
    cur.execute("""
        CREATE TABLE IF NOT EXISTS users (
            id SERIAL PRIMARY KEY,
            name TEXT NOT NULL,
            email TEXT UNIQUE NOT NULL,
            created_at TIMESTAMP DEFAULT NOW()
        )
    """)
    print("Таблица users создана (или уже существует).")

    # Вставка данных
    users_data = [
        ("Алексей Иванов", "alex@example.com"),
        ("Мария Петрова", "maria@example.com"),
        ("Иван Сидоров", "ivan@example.com")
    ]
    for name, email in users_data:
        cur.execute(
            "INSERT INTO users (name, email) VALUES (%s, %s) ON CON
FLICT (email) DO NOTHING",
            (name, email)
        )
```

```
            conn.commit()
            print(f"Добавлено записей: {len(users_data)}")

            # Выборка
            cur.execute("SELECT id, name, email, created_at FROM users ORDER BY id")
            rows = cur.fetchall()
            print("\nСодержимое таблицы users:")
            for row in rows:
                print(f"ID: {row[0]}, Name: {row[1]}, Email: {row[2]}, Created: {row[3]}")

            cur.close()
        except Exception as e:
            print(f"Ошибка: {e}")
            if conn:
                conn.rollback()
        finally:
            if conn:
                conn.close()

    if __name__ == "__main__":
        main()
```

Запуск:

bash

python app.py

Ожидаемый вывод:

text

Таблица users создана (или уже существует).

Добавлено записей: 3

Содержимое таблицы users:

ID: 1, Name: Алексей Иванов, Email: alex@example.com, Created: 2025-..

.

ID: 2, Name: Мария Петрова, Email: maria@example.com, Created: 2025-.

..

ID: 3, Name: Иван Сидоров, Email: ivan@example.com, Created: 2025-...

4.4. Ручное резервное копирование

1. В консоли → кластер PostgreSQL → вкладка **Резервные копии → Создать резервную копию**.
2. Задать имя (например, lab5-backup-2025).
3. Дождаться завершения.
4. Скриншот созданной копии (размер, дата) приложить в отчёт.

4.5. Скриншоты для отчёта

- Параметры кластера (тип хоста, диск, версия, публичный доступ).
- Правила группы безопасности (порт 5432 с разрешённым IP).
- Вывод работы Python-скрипта (терминал).
- Список резервных копий (ручная + автоматическая).

5. Индивидуальные задания (15 вариантов)

Базовое задание (создание кластера, подключение приложения, таблица users) + вариативная часть.

№ вар.	Дополнительное задание (вариативная часть)
1	Создать кластер с **двумя хостами** (мастер + реплика) в разных зонах. Проверить репликацию: вставить запись на мастере и прочитать на реплике.

№ вар.	Дополнительное задание (вариативная часть)
2	Настроить **автоматическое резервное копирование** с интервалом 24 часа и PITR (Point-in-Time Recovery). Выполнить восстановление БД на 1 час назад.
3	Реализовать **пул соединений** через psycopg2.pool.SimpleConnectionPool (не менее 5 соединений). Показать многопоточный доступ.
4	Вместо Python использовать **Node.js** (модуль pg с SSL). Код приложения в отчёте.
5	Добавить в таблицу users поле balance (numeric) и написать хранимую процедуру (на PL/pgSQL) для перевода средств между пользователями. Вызвать её из приложения.
6	Настроить **чтение SSL-сертификата** не из файловой системы, а из переменной окружения или секрета (Lockbox). Показать изменение кода.
7	Создать **дополнительную БД** (например, testdb) и пользователя с ограниченными правами (только SELECT). Подключиться к ней и продемонстрировать запрет на вставку.
8	Включить **расширение** pg_stat_statements для сбора статистики запросов. Выполнить несколько запросов и показать вывод из представления.
9	Настроить **мониторинг метрик** БД в Cloud Monitoring: создать дашборд с графиками (CPU, IOPS, connections). Скриншот дашборда.
10	Использовать **Terraform** для создания кластера PostgreSQL и группы безопасности. Приложить .tf-файлы.
11	Создать **алерт** в Cloud Monitoring: при превышении числа соединений > 80% от максимального отправлять уведомление на email.

№ вар.	Дополнительное задание (вариативная часть)
12	Реализовать **приложение на FastAPI**, которое выводит список пользователей через HTTP-эндпоинт /users. Подключение к БД через пул.
13	Создать кластер с **более производительным классом хоста** (s2.large) и провести нагрузочный тест (например, 1000 вставок). Сравнить время с минимальным классом.
14	Настроить **белый список IP** только для адреса учебной лаборатории (или VPN). Показать, что с другого IP подключение невозможно.
15	Экспортировать данные из таблицы в **файл CSV** прямо из приложения (через COPY). Сохранить файл в Object Storage.

6. Контрольные вопросы (для защиты)

1. Какие преимущества даёт использование Managed DB по сравнению с самостоятельной установкой PostgreSQL на ВМ?
2. Что такое PITR и как он работает в облачных БД?
3. Зачем нужен SSL при подключении к БД? Как проверить, что соединение зашифровано?
4. Как изменить класс хоста (количество vCPU/RAM) у существующего кластера? Будет ли простой?
5. Какие типы дисков (SSD/HDD) поддерживаются? Когда имеет смысл выбирать HDD?
6. Что такое **репликация**? В чём разница между синхронной и асинхронной?
7. Как создать резервную копию вручную? Можно ли из неё восстановить одну таблицу?
8. Как ограничить доступ к кластеру только с определённых IP-адресов?

9. Какие метрики PostgreSQL наиболее важны для мониторинга (назовите 5 штук)?

10. Можно ли подключиться к Managed PostgreSQL из функции (serverless) без публичного IP? Как?

7. Требования к отчёту

Отчёт оформляется по ГОСТ. Содержит:

1. **Титульный лист**.
2. **Цель и задачи работы** (из п.1–2).
3. **Теоретическое обоснование** (определения: Managed DB, репликация, PITR, SSL, группы безопасности).
4. **Практическая часть**:
   - Пошаговое создание кластера PostgreSQL (скриншоты конфигурации).
   - Правила группы безопасности (скриншот).
   - Код приложения (листинг с комментариями).
   - Вывод приложения в терминале (скриншот).
   - Создание ручной резервной копии (скриншот в списке бэкапов).
   - **Вариативная часть** – дополнительные скриншоты/код/команды.
5. **Результаты выполнения** (таблица: параметры кластера, результат SELECT, список резервных копий).
6. **Анализ результатов** (быстрота создания, простота подключения, надёжность бэкапов).
7. **Ответы на контрольные вопросы** (письменно).
8. **Вывод** (освоены навыки создания управляемой БД, подключения с SSL, написания приложения-клиента, управления бэкапами).

8. Критерии оценивания

Оценка	Критерии
«Отлично»	Выполнены **базовая + вариативная части**. Кластер создан корректно, группа безопасности разрешает доступ только с нужного IP. Приложение на Python/Node.js подключается по SSL, создаёт таблицу, вставляет и читает данные. Резервная копия создана. Отчёт содержит все скриншоты, листинги, анализ. Защита: глубокие ответы на вопросы.
«Хорошо»	Выполнена **только базовая часть**. Кластер создан, приложение работает, резервная копия есть. Но вариативное задание отсутствует. Незначительные недочёты в коде или отчёте.
«Удовлетворительно»	Работа выполнена частично: кластер создан, но подключение по SSL не настроено (например, приложение использует нешифрованное соединение) или приложение не создаёт таблицу. Резервная копия не создана. Отчёт неполный.
«Неудовлетворительно»	Кластер не создан, либо приложение не подключается, либо отсутствует

Оценка	Критерии
	демонстрация работы с БД. Отчёт не предоставлен.

Лабораторная работа № 6

Terraform: полный стек (ВМ + БД + сеть)

1. Цель работы

Научиться описывать **взаимозависимую облачную инфраструктуру** с использованием Terraform: сеть (VPC, подсеть), группу безопасности, виртуальную машину и управляемый кластер PostgreSQL, а также организовывать безопасное хранение состояния (state) в удалённом бэкенде.

2. Задачи работы

1. Разработать Terraform-конфигурацию, которая создаёт:
   - **VPC** (облачную сеть) и **одну подсеть**.
   - **Группу безопасности** с правилами: SSH (порт 22), HTTP (порт 80), доступ к PostgreSQL (порт 5432) из подсети ВМ.
   - **Виртуальную машину** (Ubuntu 22.04) с публичным IP и установкой тестового приложения (например, psql клиента).
   - **Кластер Managed PostgreSQL** (один хост-мастер) в той же подсети (или с доступом из подсети ВМ).
2. Передать **выходные переменные (output)**: публичный IP-адрес ВМ, hostname (FQDN) кластера БД, имя БД.
3. Использовать **переменные окружения** для хранения секретов (пароль БД, API-ключ/токен провайдера), не фиксировать их в .tf-файлах.
4. Выполнить `terraform apply` и убедиться, что с ВМ возможно подключиться к БД (установить PostgreSQL-клиент и выполнить SELECT).
5. Настроить **удалённый бэкенд** (Object Storage S3-совместимый) для хранения файла terraform.tfstate.
6. Оформить отчёт: привести код конфигурации, схему инфраструктуры (ручная диаграмма), подтверждение работоспособности подключения ВМ → БД.

3. Теоретическое обоснование

Взаимозависимость ресурсов в Terraform – один ресурс может ссылаться на другой (например, ВМ ссылается на идентификатор подсети, а подсеть – на сеть). Terraform автоматически строит граф зависимостей и создаёт ресурсы в правильном порядке.

Удалённый бэкенд (remote backend) – хранилище файла состояния (terraform.tfstate) вне локальной файловой системы. Необходим для командной работы, предотвращения конфликтов и сохранения состояния при удалении локальных файлов. Для S3-совместимых хранилищ используется бэкенд s3 (с указанием эндпоинта, ключей, имени бакета). Преимущества: блокировка записи (locking) для предотвращения одновременных изменений.

Переменные окружения – способ передачи чувствительных данных (токены, пароли) без их жёсткого кодирования в конфигурации. Terraform автоматически считывает переменные вида TF_VAR_<имя_переменной>. Пример: export TF_VAR_db_password=secret.

Структура типового проекта Terraform:

- provider.tf – провайдеры (yandex, aws) и настройки бэкенда.
- variables.tf – объявления переменных.
- main.tf – ресурсы (сеть, БД, ВМ).
- outputs.tf – выходные значения.
- terraform.tfvars – значения переменных (не секретных) – **в git не добавлять**.

Связь ВМ и БД:

- БД должна быть доступна по сети: либо ВМ и БД в одной подсети/VPC, либо настроены правила группы безопасности, разрешающие трафик от ВМ к БД.
- Для подключения приложения на ВМ используется hostname (FQDN) кластера и порт 5432.

4. Пример практического выполнения (Yandex Cloud + Terraform + удалённый бэкенд)

4.1. Предварительные шаги

- Создать **сервисный аккаунт** в Yandex Cloud с ролями:
  - editor (для создания ресурсов);
  - storage.uploader (для записи state в Object Storage).
- Создать статический ключ доступа (access_key, secret_key) для сервисного аккаунта.
- Создать **бакет Object Storage** для хранения state (например, tfstate-lab6-{студент}).

4.2. Структура файлов

text

```
lab6-terraform-full/
├── provider.tf
├── backend.tf
├── variables.tf
├── main.tf
├── outputs.tf
└── terraform.tfvars (не включать в отчёт, но показать пример)
```

4.3. Код конфигурации

provider.tf (провайдер yandex, версия):

hcl

```
terraform {
  required_providers {
    yandex = {
      source  = "yandex-cloud/yandex"
      version = "~> 0.110"
    }
  }
}
```

```
provider "yandex" {
  token     = var.yc_token
  cloud_id  = var.yc_cloud_id
  folder_id = var.yc_folder_id
  zone      = var.zone
}
```

backend.tf (удалённое состояние в Object Storage):

hcl

```
terraform {
  backend "s3" {
    endpoint   = "storage.yandexcloud.net"
    bucket     = "tfstate-lab6-student"
    region     = "ru-central1"
    key        = "lab6/terraform.tfstate"
    access_key = var.static_access_key
    secret_key = var.static_secret_key

    # Блокировки (для Yandex Cloud – опционально, но можно использов
ать DynamoDB-совместимый сервис)
    skip_region_validation      = true
    skip_credentials_validation = true
    skip_requesting_account_id  = true
  }
}
```

variables.tf:

hcl

```
variable "yc_token" {
  description = "Yandex Cloud OAuth token"
  sensitive   = true
```

```
}
variable "yc_cloud_id" {}
variable "yc_folder_id" {}
variable "zone" {
  default = "ru-central1-a"
}
variable "static_access_key" {
  sensitive = true
}
variable "static_secret_key" {
  sensitive = true
}
variable "vm_name" {
  default = "lab6-vm"
}
variable "db_password" {
  description = "Password for PostgreSQL user"
  sensitive   = true
}
variable "db_user" {
  default = "appuser"
}
variable "db_name" {
  default = "lab6db"
}
variable "public_ssh_key" {
  sensitive = true
}
```

main.tf (сеть, подсеть, группа безопасности, ВМ, кластер БД):

hcl

```
# Сеть VPC
resource "yandex_vpc_network" "lab6_net" {
  name = "lab6-network"
}

# Подсеть
resource "yandex_vpc_subnet" "lab6_subnet" {
  name           = "lab6-subnet"
  zone           = var.zone
  network_id     = yandex_vpc_network.lab6_net.id
  v4_cidr_blocks = ["10.20.0.0/24"]
}

# Группа безопасности
resource "yandex_vpc_security_group" "lab6_sg" {
  name       = "lab6-sg"
  network_id = yandex_vpc_network.lab6_net.id

  ingress {
    protocol       = "TCP"
    description    = "SSH from anywhere"
    v4_cidr_blocks = ["0.0.0.0/0"]
    port           = 22
  }
  ingress {
    protocol       = "TCP"
    description    = "HTTP from anywhere"
    v4_cidr_blocks = ["0.0.0.0/0"]
    port           = 80
  }
```

```
  # Доступ к БД из подсети ВМ (можно открыть для всей подсети)
  ingress {
    protocol       = "TCP"
    description    = "PostgreSQL from internal subnet"
    v4_cidr_blocks = ["10.20.0.0/24"]
    port           = 5432
  }
  egress {
    protocol       = "ANY"
    description    = "any outbound"
    v4_cidr_blocks = ["0.0.0.0/0"]
    from_port      = 0
    to_port        = 65535
  }
}

# Виртуальная машина
resource "yandex_compute_instance" "vm" {
  name        = var.vm_name
  platform_id = "standard-v3"
  zone        = var.zone

  resources {
    cores  = 2
    memory = 4
  }

  boot_disk {
    initialize_params {
      image_id = "fd8m8g1to9nqe2r7gdfm"  # Ubuntu 22.04
```

```
    size     = 20
  }
 }

 network_interface {
  subnet_id          = yandex_vpc_subnet.lab6_subnet.id
  security_group_ids = [yandex_vpc_security_group.lab6_sg.id]
  nat                = true
 }

 metadata = {
  ssh-keys = "ubuntu:${var.public_ssh_key}"
  user-data = <<-EOF
   #!/bin/bash
   apt update
   apt install -y postgresql-client
   EOF
 }
}

# Кластер Managed PostgreSQL
resource "yandex_mdb_postgresql_cluster" "lab6_pg" {
 name        = "lab6-postgres"
 environment = "PRESTABLE"
 network_id  = yandex_vpc_network.lab6_net.id

 config {
  version = 15
  resources {
   resource_preset_id = "s2.micro"
```

```
    disk_type_id       = "network-ssd"
    disk_size          = 20
  }
 }

 host {
  zone      = var.zone
  subnet_id = yandex_vpc_subnet.lab6_subnet.id
  assign_public_ip = false   # БД не имеет публичного IP, доступ только
из VPC
 }
}

# База данных и пользователь
resource "yandex_mdb_postgresql_database" "db" {
 cluster_id = yandex_mdb_postgresql_cluster.lab6_pg.id
 name       = var.db_name
 owner      = var.db_user
}

resource "yandex_mdb_postgresql_user" "user" {
 cluster_id = yandex_mdb_postgresql_cluster.lab6_pg.id
 name       = var.db_user
 password   = var.db_password
 permission {
  database_name = yandex_mdb_postgresql_database.db.name
 }
}
```

outputs.tf:

hcl

```
output "vm_public_ip" {
  value = yandex_compute_instance.vm.network_interface[0].nat_ip_address
}

output "db_host" {
  value = yandex_mdb_postgresql_cluster.lab6_pg.host[0].fqdn
}

output "db_name" {
  value = yandex_mdb_postgresql_database.db.name
}

output "db_user" {
  value = yandex_mdb_postgresql_user.user.name
}
```

terraform.tfvars.example (не включать в реальный код, только пример):

hcl

```
yc_token       = "t1.9euelZ..."
yc_cloud_id    = "b1g..."
yc_folder_id   = "b1g..."
static_access_key = "YCAJE..."
static_secret_key = "YCM..."
public_ssh_key   = "ssh-rsa AAAAB3N..."
db_password      = "StrongP@ssw0rd"
```

4.4. Выполнение

bash

```
# Установка переменных окружения (секреты не в файлах)
export TF_VAR_yc_token="t1.9euelZ..."
export TF_VAR_static_access_key="YCAJE..."
export TF_VAR_static_secret_key="YCM..."
```

export TF_VAR_db_password="StrongP@ssw0rd"

terraform init

terraform plan

terraform apply

4.5. Проверка подключения ВМ → БД

После apply получить публичный IP ВМ и FQDN БД из output.

Подключиться по SSH к ВМ:

bash

ssh ubuntu@<vm_public_ip>

На ВМ (уже установлен postgresql-client благодаря cloud-init):

bash

psql "host=<db_host> port=5432 dbname=lab6db user=appuser password=<db_password> sslmode=require"

Выполнить SQL:

sql

CREATE TABLE test (id int);

INSERT INTO test VALUES (1);

SELECT * FROM test;

\q

Убедиться, что подключение успешно.

4.6. Схема инфраструктуры (пример для отчёта)

(Рисунок от руки или в draw.io)

- VPC (10.20.0.0/24)
  - Подсеть 10.20.0.0/24
    - ВМ (10.20.0.10, публичный IP 84.201.x.x)
    - Кластер PostgreSQL (10.20.0.20, без публичного IP)
- Группа безопасности: разрешены порты 22, 80, 5432.

4.7. Скриншоты для отчёта

- Вывод terraform plan (создание 8+ ресурсов).

- Вывод terraform apply с output.
- Файл backend.tf (конфигурация удалённого state).
- Подключение по SSH к ВМ и выполнение psql (скриншот терминала).
- Список ресурсов в консоли облака (ВМ, кластер БД, сеть, группа безопасности).
- Файл состояния в Object Storage (бакет с terraform.tfstate).

5. Индивидуальные задания (15 вариантов)

Базовое задание (сеть + ВМ + БД + удалённый бэкенд) + вариативная часть.

№ вар.	Дополнительное задание (вариативная часть)
1	Добавить **вторую подсеть** в другой зоне и разместить в ней реплику БД (два хоста PostgreSQL: мастер и реплика).
2	Использовать **модули Terraform** (официальный модуль для VPC или собственный модуль для ВМ). Показать вызов модуля.
3	Включить **блокировку state** с помощью Yandex Lockbox или DynamoDB (через dynamodb_table в бэкенде S3).
4	Создать **Application Load Balancer** перед ВМ (в рамках той же конфигурации) и целевую группу из одной ВМ.
5	В user-data ВМ автоматически склонировать простое Python-приложение, которое подключается к БД и выводит список таблиц.
6	Использовать terraform import для импорта существующего ресурса (например, вручную созданной сети) в state.
7	Добавить **provisioner** remote-exec, который на ВМ создаёт таблицу в БД и вставляет тестовые данные.

№ вар.	Дополнительное задание (вариативная часть)
8	Настроить **параметры БД** через postgresql_config (например, изменить max_connections).
9	В output вывести **команду подключения** к БД с уже подставленными параметрами (psql строка).
10	Создать **Service Account** и привязать его к ВМ (метаданные) вместо использования статического ключа в провайдере.
11	Добавить **бакет Object Storage** в конфигурацию и выдать ВМ права на запись в этот бакет через сервисный аккаунт.
12	Использовать for_each для создания нескольких ВМ (например, 2 ВМ) с разными именами, но одной БД.
13	Организовать **межпроектные ссылки** (data source) для получения ID образа или ID сети из другого каталога.
14	Добавить **графическое описание зависимостей** через terraform graph и сгенерировать PNG. Включить в отчёт.
15	Написать **Makefile** с целями init, plan, apply, destroy для автоматизации работы с Terraform.

6. Контрольные вопросы (для защиты)

1. Почему в данной работе используется удалённый бэкенд? Какие проблемы решает хранение state в Object Storage?
2. Как Terraform определяет порядок создания ресурсов (ВМ зависит от подсети, подсеть от сети)?
3. Что произойдёт, если два разработчика одновременно запустят terraform apply с общим удалённым бэкендом без блокировки?

4. Каким образом передать пароль БД в конфигурацию, не сохраняя его в файлах .tf или .tfvars?

5. Зачем нужна группа безопасности, разрешающая доступ к БД только из подсети ВМ, а не из всего интернета?

6. Можно ли создать кластер БД без публичного IP и подключиться к нему с ВМ, у которой есть публичный IP? Как?

7. Что такое data source в Terraform? Приведите пример использования.

8. Как изменить существующий ресурс (например, увеличить диск БД) с помощью Terraform? Будет ли downtime?

9. Как удалить всю инфраструктуру, созданную Terraform? Какие ресурсы будут удалены?

10. Что такое «дрейф конфигурации» (configuration drift) и как Terraform с ним борется?

7. Требования к отчёту

Отчёт оформляется по ГОСТ. Содержит:

1. **Титульный лист**.
2. **Цель и задачи работы**.
3. **Теоретическое обоснование** (понятия: взаимозависимости, удалённый бэкенд, переменные окружения, S3-бэкенд).
4. **Практическая часть**:
   - Полные листинги .tf-файлов (provider, backend, variables, main, outputs).
   - Пример файла terraform.tfvars (без реальных секретов).
   - Команды инициализации и применения.
   - **Схема инфраструктуры** (рисунок).
   - Скриншоты: terraform plan, terraform apply, output.
   - Подключение к ВМ и выполнение psql (скриншот).
   - Список ресурсов в консоли облака (подтверждение).

- Файл state в Object Storage (скриншот бакета).
- **Вариативная часть** – дополнительные скриншоты/код.

5. **Результаты выполнения** (таблица с адресами, hostname БД, статус подключения).
6. **Анализ результатов** (сравнение ручного создания и IaC, преимущества удалённого state).
7. **Ответы на контрольные вопросы**.
8. **Вывод** (освоено: описание полного стека ресурсов, управление зависимостями, удалённое состояние, безопасность секретов).

8. Критерии оценивания

Оценка	Критерии
«Отлично»	Выполнены **базовая + вариативная части**. Конфигурация Terraform создаёт сеть, ВМ, БД, все зависимости корректны. Удалённый бэкенд настроен и работает. С ВМ успешно устанавливается соединение с БД через psql. Отчёт содержит все листинги, схему, скриншоты, аналитику. Защита: уверенные ответы.
«Хорошо»	Выполнена **только базовая часть** (без вариативной). Инфраструктура создаётся, но, возможно, удалённый бэкенд настроен не полностью (state локальный). Подключение ВМ→БД работает. Отчёт имеет незначительные недочёты.

Оценка	Критерии
«Удовлетворительно»	Работа выполнена частично: Terraform-конфигурация есть, но при apply возникают ошибки (например, сеть создана, БД нет). Удалённый бэкенд не настроен. Доказательства подключения к БД неполные. Отчёт беден на скриншоты.
«Неудовлетворительно»	Конфигурация отсутствует или не работает (синтаксические ошибки, невозможность применить). Нет подтверждения связи ВМ–БД. Отчёт не предоставлен.

Лабораторная работа № 7
Container Registry и простой деплой

1. Цель работы

Освоить контейнеризацию в облаке: создание Docker-образа простого веб-приложения, его загрузка (push) в облачный Container Registry, последующий запуск контейнера на виртуальной машине с предустановленным Docker, обеспечение внешнего доступа к приложению.

2. Задачи работы

1. Разработать простое веб-приложение (Flask / FastAPI / Go / Node.js), которое отвечает на HTTP-запросы сообщением, например, «Hello from container».
2. Написать **Dockerfile** для сборки образа приложения.
3. Создать **Container Registry** (реестр образов) в облачной платформе.
4. Выполнить аутентификацию в реестре (Docker login).
5. Собрать образ локально, присвоить ему тег с адресом реестра, выполнить **push** в реестр.
6. Создать виртуальную машину (через консоль или Terraform) с установленным Docker.
7. Настроить группу безопасности: открыть порт 80 для входящего трафика.
8. На ВМ выполнить **pull** образа из реестра и запустить контейнер с пробросом порта (например, `-p 80:8080`).
9. Проверить доступность приложения через публичный IP ВМ в браузере или с помощью `curl`.
10. Оформить отчёт: Dockerfile, команды push/pull, скриншот работающего приложения.

3. Теоретическое обоснование

Контейнеризация – метод изоляции приложений на уровне операционной системы, позволяющий упаковать приложение со всеми зависимостями в единый образ, который можно запустить в любом окружении, поддерживающем контейнеры (Docker).

Container Registry – облачное хранилище Docker-образов (аналог Docker Hub, но внутри облака). Позволяет хранить приватные и публичные образы, интегрируется с IAM для управления доступом, поддерживает сканирование уязвимостей. Примеры: Yandex Container Registry, AWS ECR (Elastic Container Registry), Google Container Registry (GCR), VK Cloud Container Registry.

Dockerfile – текстовый файл с инструкциями для сборки образа. Основные команды: `FROM` (базовый образ), `WORKDIR`, `COPY`, `RUN`, `EXPOSE`, `CMD` (или `ENTRYPOINT`).

Push / Pull – операции загрузки образа из локального окружения в реестр (push) и скачивания образа из реестра на сервер (pull). Перед push необходимо выполнить `docker login` с учётными данными облака (IAM-токен, статический ключ или OAuth).

Запуск контейнера на ВМ: после установки Docker на ВМ, команда `docker run -d -p <хост_порт>:<контейнер_порт> --name <имя> <образ>` запускает контейнер в фоновом режиме, пробрасывая порт хоста (например, 80) на порт внутри контейнера (например, 8080). Группа безопасности ВМ должна разрешать входящий трафик на порт хоста.

4. Пример практического выполнения (Yandex Cloud + Flask + Docker)

4.1. Простое Flask-приложение

Создадим файл `app.py`:

python

```
from flask import Flask
```

```
import os

app = Flask(__name__)

@app.route('/')
def hello():
    return f"Hello from container! Host: {os.uname().nodename}"

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8080)
```

4.2. Dockerfile

Создадим файл Dockerfile:

dockerfile

```
FROM python:3.11-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY app.py .

EXPOSE 8080

CMD ["python", "app.py"]
```

Файл requirements.txt:

text

```
flask==3.0.0
```

4.3. Сборка образа и тегирование

Локально (на рабочей станции) соберём образ:

bash

```
docker build -t flask-hello:latest .
```

4.4. Создание Container Registry в облаке

1. В консоли Yandex Cloud: **Container Registry → Создать реестр**.
2. Имя: lab7-registry.
3. (Опционально) Настроить права доступа (по умолчанию приватный).

4.5. Аутентификация в реестре

Способ 1 – с помощью IAM-токена:

bash

```
yc iam create-token | docker login --username iam --password-stdin cr.yandex
```

Способ 2 – с помощью статического ключа (сервисный аккаунт):

bash

```
docker login --username json_key --password "$(cat key.json)" cr.yandex
```

4.6. Push образа

Присвоить тег с адресом реестра:

bash

```
docker tag flask-hello:latest cr.yandex/<registry_id>/flask-hello:v1
docker push cr.yandex/<registry_id>/flask-hello:v1
```

4.7. Создание ВМ с Docker (через консоль)

- Создать ВМ Ubuntu 22.04 (2 vCPU, 4 GB RAM, диск 20 GB).
- Группа безопасности: добавить правило для порта 80 (TCP, 0.0.0.0/0).
- В поле **user-data** (cloud-init) указать скрипт установки Docker:

bash

```
#!/bin/bash
apt update
apt install -y docker.io
systemctl enable docker
```

systemctl start docker

4.8. На ВМ: pull и запуск контейнера

Подключиться по SSH к ВМ:

bash

ssh ubuntu@<публичный_IP>

Выполнить аутентификацию в Container Registry (те же команды, что в п.4.5) и запустить:

bash

docker pull cr.yandex/<registry_id>/flask-hello:v1

docker run -d --name flask-app -p 80:8080 cr.yandex/<registry_id>/flask-hello:v1

Проверить, что контейнер работает:

bash

docker ps

curl http://localhost

4.9. Проверка из браузера

В браузере перейти по адресу http://<публичный_IP_ВМ>. Должно отобразиться сообщение: Hello from container! Host: ...

4.10. Скриншоты для отчёта

- Файлы app.py, Dockerfile, requirements.txt.
- Выполненная команда docker push (вывод).
- Список образов в Container Registry (через консоль).
- Команды docker pull и docker run на ВМ.
- Результат curl http://localhost на ВМ.
- Страница приложения в браузере (публичный IP).

5. Индивидуальные задания (15 вариантов)

Базовое задание (простое приложение, Dockerfile, push в реестр, запуск на ВМ) + вариативная часть.

№ вар.	Дополнительное задание (вариативная часть)
1	В Dockerfile использовать **многостадийную сборку** (например, для Go: сначала сборка, затем минимальный образ scratch).
2	Добавить в приложение **эндпоинт** /health, возвращающий {"status":"ok"}. Настроить проверку здоровья при запуске (healthcheck в Dockerfile).
3	Использовать **Terraform** для создания Container Registry, ВМ и настройки групп безопасности. Приложить .tf-файлы.
4	Запустить на ВМ **два контейнера** (то же приложение на разных портах, например, 80 и 8080) через docker-compose.
5	Вместо Flask использовать **Node.js (Express)**. Предоставить Dockerfile и код.
6	Настроить **автоматическую сборку образа** при push в Git-репозиторий (через Yandex Cloud Triggers + Cloud Functions). Описать конфигурацию.
7	Сохранить **секреты** (например, переменную окружения MESSAGE) в Yandex Lockbox и передать в контейнер при запуске (-e).
8	Добавить **тегирование образа** несколькими тегами (v1, latest) и проверить, что оба тега указывают на один образ.
9	Создать **сервисный аккаунт** для ВМ, который имеет права только на pull из Container Registry (без логина по docker login на ВМ). Использовать YC_ переменные окружения.

№ вар.	Дополнительное задание (вариативная часть)
10	Запустить контейнер с **ограничением ресурсов** (`--memory=256m --cpus=0.5`). Показать, что ограничение работает.
11	В приложении выводить **IP-адрес контейнера** (внутренний) и имя хоста. Доказать, что контейнер изолирован.
12	Использовать **иной облачный реестр** (например, AWS ECR или VK Cloud Container Registry). Описать отличия в аутентификации.
13	Вместо ручного создания ВМ использовать **Managed Kubernetes** (одноузловой кластер) и запустить там этот же образ через `kubectl run`.
14	Настроить **сканирование уязвимостей** образа в Container Registry, показать отчёт (наличие или отсутствие критических CVE).
15	Добавить **логгирование** в приложение (вывод в stdout) и настроить сбор логов контейнера в Cloud Logging (через драйвер `fluentd`).

6. Контрольные вопросы (для защиты)

1. Что такое Container Registry и чем он отличается от Docker Hub?
2. Какие команды Docker используются для отправки образа в реестр?
3. Для чего нужна аутентификация в реестре (`docker login`)? Какие способы аутентификации поддерживаются в Yandex Cloud / AWS?

4. Что такое тег образа? Почему рекомендуется использовать теги, отличные от latest, в production?

5. Какая инструкция в Dockerfile определяет порт, который будет слушать приложение внутри контейнера?

6. Чем отличается CMD от ENTRYPOINT в Dockerfile?

7. Как проверить, что контейнер запущен на ВМ и слушает нужный порт?

8. Почему при запуске контейнера используется флаг -d? Что делает флаг --rm?

9. Можно ли запустить контейнер без предварительного pull образа? Что произойдёт?

10. Как обновить работающий контейнер до новой версии образа без остановки сервиса? Опишите шаги.

7. Требования к отчёту

Отчёт оформляется по ГОСТ. Содержит:

1. **Титульный лист**.
2. **Цель и задачи работы**.
3. **Теоретическое обоснование** (контейнеризация, Container Registry, Dockerfile, push/pull, запуск контейнера).
4. **Практическая часть**:
   - Код приложения (например, app.py) и requirements.txt.
   - Содержимое Dockerfile.
   - Команды сборки, тегирования, аутентификации, push (с выводом терминала).
   - Скриншот созданного реестра (консоль облака).
   - Создание ВМ (скриншот параметров, user-data).
   - Подключение по SSH к ВМ, команды pull и run.
   - Скриншот curl http://localhost и страницы в браузере.
   - **Вариативная часть** – дополнительные скриншоты/код.

5. **Результаты выполнения** (таблица: образ, тег, адрес в реестре, публичный IP ВМ, результат запроса).

6. **Анализ результатов** (сравнение запуска контейнера из реестра vs локальный образ, преимущества облачного реестра).

7. **Ответы на контрольные вопросы**.

8. **Вывод** (освоены: работа с Container Registry, push/pull, запуск контейнеров на ВМ).

8. Критерии оценивания

Оценка	Критерии
«Отлично»	Выполнены **базовая + вариативная части**. Приложение написано, Dockerfile корректен. Образ успешно запушен в Container Registry. На ВМ установлен Docker, контейнер запущен, приложение доступно по публичному IP. Отчёт содержит все листинги, скриншоты, анализ. Защита: уверенные ответы.
«Хорошо»	Выполнена **только базовая часть**. Все основные шаги выполнены, но вариативное задание отсутствует. Возможны небольшие недочёты (например, не показан `docker pull`). Отчёт без грубых ошибок.
«Удовлетворительно»	Работа выполнена частично: образ собран, но push не выполнен или контейнер на ВМ

Оценка	Критерии
	не запущен. Приложение не доступно извне. Отчёт неполный.
«Неудовлетворительно»	Отсутствует Dockerfile, образ не собран, реестр не создан, ВМ не развёрнута. Отчёт не предоставлен.

Лабораторная работа № 8
Managed Kubernetes Cluster — первый деплой

1. Цель работы

Познакомиться с Kubernetes как облачным сервисом (Managed Kubernetes): создание кластера, настройка доступа через kubectl, деплой контейнеризованного приложения из Container Registry, публикация сервиса через Load Balancer, выполнение обновления приложения с использованием rolling update.

2. Задачи работы

1. Создать **Managed Kubernetes Cluster** (управляемый кластер) в облачной платформе с одной группой узлов (node group) из 1–2 узлов.
2. Настроить kubectl для доступа к кластеру (получить и применить kubeconfig).
3. Используя образ из Container Registry (созданный в ЛР 7), написать YAML-манифесты:
   - **Deployment** – для управления репликами приложения;
   - **Service** типа **LoadBalancer** – для доступа из интернета.
4. Развернуть приложение в кластере (kubectl apply -f).
5. Получить публичный IP-адрес балансировщика (LoadBalancer) и проверить доступность приложения в браузере.
6. Выполнить **обновление приложения**: изменить версию образа (например, v1 → v2) в Deployment, применить изменения, наблюдать rolling update.
7. Зафиксировать результаты в отчёте: YAML-манифесты, скриншоты работающего приложения, вывод команд kubectl get pods, services, deployments.

3. Теоретическое обоснование

Kubernetes (K8s) – открытая платформа для оркестрации контейнеров, автоматизирующая развёртывание, масштабирование и управление контейнеризованными приложениями.

Managed Kubernetes – сервис облачного провайдера, который предоставляет готовый кластер K8s, берёт на себя управление control plane (мастер-узлы), обновления, мониторинг. Пользователь управляет только worker-узлами (node groups) и приложениями.

Ключевые понятия:

- **Cluster** – совокупность master-узлов (control plane) и worker-узлов.
- **Node group** – группа однотипных worker-узлов (BM), может масштабироваться автоматически.
- **Pod** – минимальная единица развёртывания, один или несколько контейнеров.
- **Deployment** – декларативное описание желаемого состояния подов (реплики, стратегия обновления, образы).
- **Service** – абстракция, обеспечивающая стабильный доступ к подам (ClusterIP, NodePort, LoadBalancer).
- **LoadBalancer** – тип сервиса, который создаёт внешний облачный балансировщик с публичным IP.

Rolling update – стратегия обновления Deployment, при которой старые поды заменяются новыми постепенно (по умолчанию: maxSurge=25%, maxUnavailable=25%), что обеспечивает zero-downtime.

Интеграция с Container Registry – кластер должен иметь доступ к реестру (обычно через сервисный аккаунт или IAM). При создании кластера можно указать сервисный аккаунт с правами на чтение образов.

4. Пример практического выполнения (Yandex Cloud + Managed K8s)

4.1. Создание Managed Kubernetes Cluster

1. В консоли Yandex Cloud: **Managed Service for Kubernetes → Создать кластер**.
2. Параметры:
   - Имя: lab8-cluster
   - Версия Kubernetes: последняя стабильная (например, 1.28).
   - **Сервисный аккаунт** – должен иметь роли: k8s.cluster-api.cluster-admin, k8s.tunnels.cluster-agent.
   - **Сеть и подсеть**: выбрать существующую VPC (или создать новую).
3. **Группа узлов (node group)**:
   - Имя: lab8-nodes
   - Тип ВМ: 2 vCPU, 4 GB RAM (s2.micro).
   - Количество узлов: 1 (можно 2 для демонстрации отказоустойчивости).
   - Диск: 20 ГБ SSD.
4. Создать кластер (5–10 минут).

4.2. Получение kubeconfig и настройка kubectl

bash

```
# Установить kubectl (если не установлен)
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
chmod +x kubectl
sudo mv kubectl /usr/local/bin/

# Получить конфигурацию для кластера
yc managed-kubernetes cluster get-credentials lab8-cluster --external
# или через консоль: скачать kubeconfig вручную
```

Проверить доступ:

bash

```
kubectl cluster-info
```

kubectl get nodes

4.3. Подготовка YAML-манифестов

Используем образ из ЛР 7: cr.yandex/<registry_id>/flask-hello:v1. Создадим файл deployment.yaml:

yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flask-hello
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: flask-hello
  template:
    metadata:
      labels:
        app: flask-hello
    spec:
      containers:
      - name: app
        image: cr.yandex/<registry_id>/flask-hello:v1
        ports:
        - containerPort: 8080
        readinessProbe:
          httpGet:
            path: /health
            port: 8080
          initialDelaySeconds: 5
```

```
        livenessProbe:
          httpGet:
            path: /
            port: 8080
          initialDelaySeconds: 15
```

Создадим файл service.yaml:

yaml

```
apiVersion: v1
kind: Service
metadata:
  name: flask-hello-svc
spec:
  type: LoadBalancer
  selector:
    app: flask-hello
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```

4.4. Деплой приложения

bash

```
kubectl apply -f deployment.yaml
kubectl apply -f service.yaml
```

4.5. Получение публичного IP LoadBalancer'а

bash

```
kubectl get svc
```

Дождаться, когда EXTERNAL-IP из <pending> превратится в реальный IP (например, 84.201.xxx.xxx).

Открыть в браузере: http://<EXTERNAL-IP> → увидеть сообщение Flask-приложения.

4.6. Обновление приложения (rolling update)

Изменить образ на новую версию (например, v2). Предварительно собрать и запушен второй образ flask-hello:v2 (изменив сообщение в коде, например, "Hello from container v2").

Отредактировать deployment.yaml (заменить v1 на v2) или выполнить:

bash

kubectl set image deployment/flask-hello app=cr.yandex/<registry_id>/flask-hello:v2

kubectl rollout status deployment/flask-hello

Наблюдать, как старые поды завершаются, новые создаются.

Проверить, что приложение продолжает отвечать во время обновления (можно выполнить watch curl http://<EXTERNAL-IP>).

4.7. Дополнительные команды для отчёта

bash

kubectl get pods -o wide

kubectl describe deployment flask-hello

kubectl logs <pod-name>

4.8. Скриншоты для отчёта

- Параметры созданного кластера K8s в консоли (версия, группа узлов).
- Вывод kubectl get nodes.
- YAML-манифесты (Deployment, Service).
- Вывод kubectl get svc с IP LoadBalancer’a.
- Страница приложения в браузере.
- Процесс rolling update (например, kubectl rollout status и смена версий).
- Логи подов (выборочно).

5. Индивидуальные задания (15 вариантов)

Базовое задание (создание кластера, деплой Deployment + LoadBalancer, rolling update) + вариативная часть.

№ вар.	Дополнительное задание (вариативная часть)
1	Добавить **Ingress Controller** (например, nginx-ingress) и настроить маршрутизацию по хосту (virtual hosting).
2	Включить **автомасштабирование подов (HPA)** на основе CPU (target 50%). Нагрузить приложение, показать увеличение реплик.
3	Использовать **ConfigMap** для передачи переменных окружения (например, MESSAGE=HelloConfig). Обновить ConfigMap без перезапуска пода (через переменную).
4	Добавить **PersistentVolumeClaim** для PostgreSQL (StatefulSet) и связать с приложением. Показать сохранение данных после перезапуска.
5	Развернуть **Monitoring стек** (Prometheus + Grafana) в кластере через Helm и показать дашборд с метриками пода.
6	Настроить **Cluster Autoscaler** для группы узлов (добавление новых worker-узлов при нехватке ресурсов). Провести нагрузочный тест.
7	В Deployment использовать **initContainer** для предварительной настройки (например, ожидание доступности БД).

№ вар.	Дополнительное задание (вариативная часть)
8	Создать **Service типа NodePort** вместо LoadBalancer и получить доступ через порт узла (с публичным IP узла).
9	Применить **Canary deployment** (через два Deployment с разными версиями и Service, разделяющий трафик 90/10 через weighted или subset).
10	Описать всю инфраструктуру (кластер K8s, node group, deployment, service) с помощью **Terraform (провайдер yandex_kubernetes_cluster)**. Приложить .tf-файлы.
11	Развернуть **несколько микросервисов** (например, frontend + backend) и связать их через внутренний Service (ClusterIP).
12	Добавить **проверки liveness и readiness** в Deployment (уже есть в примере, но изменить пути ответа на /health и /). Показать, как probe влияет на рестарты.
13	Использовать **private registry** с авторизацией: создать secret типа docker-registry для доступа к приватному реестру и привязать к ServiceAccount.
14	Настроить **Network Policy**, ограничивающую доступ к подам приложения только с определённого CIDR (например, из office).
15	Выполнить **helm chart** для приложения (создать свой chart с deployment, service, hpa). Установить через helm install.

6. Контрольные вопросы (для защиты)

1. Чем отличается Managed Kubernetes от самостоятельной установки K8s на ВМ?
2. Как получить доступ к кластеру через kubectl? Что такое kubeconfig?
3. Для чего в Deployment указан selector?
4. В чем разница между типами Service: ClusterIP, NodePort, LoadBalancer?
5. Как в Managed Kubernetes реализован внешний LoadBalancer? Откуда берётся публичный IP?
6. Что такое rolling update и как его инициировать? Как откатить (rollback) обновление?
7. Как проверить, что приложение действительно запущено в кластере, а не на отдельной ВМ?
8. Что происходит, если под в Deployment умирает (crash loop)?
9. Как ограничить ресурсы (CPU/RAM) для пода? Зачем это нужно?
10. Как интегрировать Managed Kubernetes с Container Registry без явного docker login на узлах?

7. Требования к отчёту

Отчёт оформляется по ГОСТ. Содержит:

1. **Титульный лист**.
2. **Цель и задачи работы**.
3. **Теоретическое обоснование** (Managed K8s, Deployment, Service, LoadBalancer, rolling update).
4. **Практическая часть**:
   - Скриншоты создания кластера и группы узлов.
   - Команда получения kubeconfig и kubectl cluster-info.
   - Листинг YAML-манифестов (Deployment, Service).
   - Вывод команд kubectl apply, kubectl get pods, kubectl get svc.

- Скриншот приложения в браузере.
- Процесс rolling update: команда set image, rollout status, проверка доступности во время обновления.
- **Вариативная часть** – дополнительные скриншоты/листинги.

5. **Результаты выполнения** (таблица: имя Deployment, реплики, образы до и после, внешний IP, статус).
6. **Анализ результатов** (удобство управляемого K8s, скорость деплоя, zero-downtime обновление).
7. **Ответы на контрольные вопросы**.
8. **Вывод** (освоены: создание кластера, работа с kubectl, деплой через манифесты, масштабирование и обновление приложений).

8. Критерии оценивания

Оценка	Критерии
«Отлично»	Выполнены **базовая + вариативная части**. Кластер создан, kubectl настроен. Deployment и Service работают, приложение доступно через LoadBalancer. Rolling update выполнен без прерывания сервиса. Отчёт полный, содержит все команды и скриншоты. Защита: уверенные ответы.
«Хорошо»	Выполнена **только базовая часть**. Все основные шаги выполнены, приложение доступно, но вариативное задание отсутствует. Незначительные недочёты в отчёте.

Оценка	Критерии
«Удовлетворительно»	Работа выполнена частично: кластер создан, но деплой не работает (ошибки в манифестах, LoadBalancer не получает IP). Rolling update не показан.
«Неудовлетворительно»	Кластер не создан, kubectl не настроен, приложение не развёрнуто. Отчёт отсутствует.

Лабораторная работа № 9
Развёртывание serverless-функции с триггером на объектное хранилище

1. Цель работы

Освоить создание и развёртывание serverless-функции (FaaS — Function as a Service), интегрированной с объектным хранилищем через триггер, на примере обработки файлов: автоматическое преобразование загруженного объекта и сохранение результата в другой бакет.

2. Задачи работы

1. Подготовить инфраструктуру (вручную через консоль или с помощью Terraform):
   - Создать **входной бакет** (input-bucket-{id}) и **выходной бакет** (output-bucket-{id}).
2. Написать serverless-функцию на **Python**:
   - Функция принимает событие о загруженном файле (event от Object Storage).
   - Скачивает файл из входного бакета.
   - Выполняет простое преобразование (например, подсчёт строк в текстовом файле, создание превью изображения, конвертацию формата).
   - Сохраняет результат обработки в выходной бакет.
   - Пишет логи (с использованием print или logging).
3. Настроить **триггер**:
   - Создать триггер на создание объекта (ObjectCreate) во входном бакете.
   - Привязать триггер к созданной функции.
   - Назначить сервисному аккаунту необходимые роли: storage.uploader, functions.invoker.
4. Выполнить тестирование:

- Загрузить тестовый файл (например, .txt или изображение) во входной бакет.
- Проверить логи функции в Cloud Logging.
- Убедиться, что результат обработки появился в выходном бакете.

5. Настроить мониторинг и алертинг:

- Просмотреть метрики функции (количество вызовов, длительность, ошибки) в Cloud Monitoring.
- Создать алерт (alert) при возникновении ошибок функции (email-уведомление).

6. Выполнить очистку ресурсов (удалить триггер, функцию, бакеты) во избежание списания средств.

3. Теоретическое обоснование

Serverless (FaaS) – модель облачных вычислений, при которой разработчик загружает код функции, а провайдер полностью управляет запуском, масштабированием и инфраструктурой. Функция выполняется в ответ на событие (триггер) и работает только во время вызова (оплата за фактическое время выполнения и количество вызовов).

Триггер – механизм, связывающий событие в одном сервисе облака с вызовом функции. В данной работе используется триггер на **Object Storage**: при создании нового объекта (загрузке файла) во входном бакете автоматически вызывается функция.

Основные понятия:

- **Событие (event)** – JSON-объект, который передаётся функции и содержит метаинформацию о файле (бакет, ключ, время, размер и пр.).
- **Сервисный аккаунт** – учётная запись для автоматических операций. Функция будет выполняться под этим аккаунтом, поэтому ему нужны права на чтение из входного бакета и запись в выходной.
- **Логи** – вывод print или logging в функции автоматически попадает в Cloud Logging, где их можно просматривать и фильтровать.

- **Метрики** – облачный мониторинг собирает для каждой функции: количество вызовов, время выполнения, количество ошибок, использование памяти.

Ограничения FaaS (на примере Yandex Cloud Functions):

- Таймаут выполнения: до 10 минут.
- Максимальный размер тела запроса/ответа: 3.5 МБ.
- Доступная память: 128 МБ – 4 ГБ.

Сценарии использования: обработка изображений (создание миниатюр), ETL (извлечение, преобразование, загрузка), валидация файлов, генерация отчётов, webhook-обработчики.

4. Пример практического выполнения (Yandex Cloud Functions + Object Storage)

4.1. Создание бакетов

Через консоль Object Storage:

- Создать бакет с именем input-bucket-<студент> (например, input-bucket-ivanov).
- Создать бакет с именем output-bucket-<студент>.

Оба бакета – **приватные** (доступ по умолчанию закрыт). Функция будет использовать сервисный аккаунт для доступа.

4.2. Создание сервисного аккаунта

1. **Сервисный аккаунт → Создать**:
   - Имя: sa-func-lab9.
   - Роли: storage.uploader (чтение/запись в бакеты), functions.invoker (для вызова функций, если потребуется).
2. Создать **статический ключ доступа** (access_key / secret_key) для сервисного аккаунта – он может понадобиться для доступа из функции к S3 API. Альтернативно можно использовать IAM-токен, но проще через boto3 со статическим ключом.

4.3. Написание serverless-функции

Пример функции на Python – подсчёт строк в текстовом файле, загруженном во входной бакет. Результат сохраняется в выходной бакет как <имя_файла>.lines.

Код функции (handler.py):

python

```
import boto3
import os
from urllib.parse import unquote

# Инициализация S3-клиента с использованием переменных окружения (задаются в функции)
session = boto3.session.Session()
s3 = session.client(
    's3',
    endpoint_url='https://storage.yandexcloud.net',
    aws_access_key_id=os.environ['ACCESS_KEY'],
    aws_secret_access_key=os.environ['SECRET_KEY']
)

INPUT_BUCKET = os.environ['INPUT_BUCKET']
OUTPUT_BUCKET = os.environ['OUTPUT_BUCKET']

def handler(event, context):
    print(f"Received event: {event}")
    # Извлекаем информацию о файле из сообщения триггера
    messages = event.get('messages', [])
    if not messages:
        print("No messages in event")
        return
```

```
        for msg in messages:
            details = msg.get('details', {})
            bucket_id = details.get('bucket_id')
            object_key = unquote(details.get('object_id'))  # URL-декодирование
            print(f"Processing file: {object_key} from bucket {bucket_id}")

            # Скачиваем файл
            try:
                response = s3.get_object(Bucket=INPUT_BUCKET, Key=object_ke
y)
                content = response['Body'].read().decode('utf-8')
                lines = content.count('\n')
                print(f"File {object_key} has {lines} lines")
            except Exception as e:
                print(f"Error reading file: {e}")
                raise

            # Сохраняем результат в выходной бакет
            result_key = f"{object_key}.lines"
            result_content = f"File: {object_key}\nLines count: {lines}\n"
            s3.put_object(Bucket=OUTPUT_BUCKET, Key=result_key, Body=res
ult_content.encode('utf-8'))
            print(f"Saved result to {OUTPUT_BUCKET}/{result_key}")
```

Альтернативная функция для изображений (создание превью): Можно использовать библиотеку Pillow (предварительно добавив её в зависимости). При загрузке изображения функция создаёт уменьшенную копию и сохраняет её в выходной бакет.

4.4. Создание функции в облаке

1. В консоли **Cloud Functions → Создать функцию**:
   - Имя: lab9-file-processor.

2. Создать **версию** функции:

- Среда выполнения: python311.
- Способ загрузки: **Редактор кода** или **ZIP-архив**.
- Код: вставить handler.py.
- Точка входа: handler.handler (имя файла и функции).
- **Переменные окружения**:
  - ACCESS_KEY – статический ключ сервисного аккаунта.
  - SECRET_KEY – секрет.
  - INPUT_BUCKET – имя входного бакета.
  - OUTPUT_BUCKET – имя выходного бакета.
- **Сервисный аккаунт**: выбрать sa-func-lab9.
- Память: 512 МБ, таймаут: 30 сек.

3. Сохранить версию.

4.5. Создание триггера

1. **Cloud Functions → Триггеры → Создать триггер**.

- Тип: **Object Storage**.
- Имя: trigger-lab9-input.
- Бакет: input-bucket-<студент>.
- События: Создание объекта.
- Функция: lab9-file-processor (конкретная версия).
- Сервисный аккаунт для вызова функции: sa-func-lab9.
- Повтор попыток: включить (при ошибках).

4.6. Тестирование

1. Загрузить во входной бакет текстовый файл test.txt с несколькими строками (через консоль или aws s3 cp).

2. Перейти в **Cloud Logging →** фильтр по функции lab9-file-processor. Увидеть логи:

text

Processing file: test.txt from bucket input-bucket-...

File test.txt has 5 lines

Saved result to output-bucket.../test.txt.lines

3. Проверить выходной бакет: появился файл `test.txt.lines` с содержимым:

text

File: test.txt

Lines count: 5

4.7. Мониторинг и алертинг

- **Метрики**: в консоли Cloud Functions → выбрать функцию → вкладка **Мониторинг**. Посмотреть графики:
  - Количество вызовов (invocations)
  - Длительность (duration)
  - Ошибки (errors)
- **Создание алерта**:

1. **Cloud Monitoring → Алерты → Создать алерт**.
2. Метрика: `function.error_count` для функции `lab9-file-processor`.
3. Условие: `> 0` за 5 минут.
4. Канал уведомлений: `Email` (указать адрес).
5. Сохранить.

4.8. Очистка ресурсов

Удалить в порядке обратном созданию (через консоль или CLI):

- Триггер
- Функцию
- Бакеты (предварительно очистив содержимое)
- Сервисный аккаунт (опционально)

4.9. Скриншоты для отчёта

- Бакеты (input и output) в консоли.
- Код функции (скриншот редактора или листинг).
- Переменные окружения функции.
- Созданный триггер (список триггеров).
- Логи выполнения (Cloud Logging) после загрузки файла.

- Выходной файл в output-бакете (содержимое).
- Метрики функции (график вызовов).
- Настроенный алерт в Cloud Monitoring.

5. Индивидуальные задания (15 вариантов)

Базовое задание (обработка текстового файла – подсчёт строк) + вариативная часть.

№ вар.	Дополнительное задание (вариативная часть)
1	Функция считает **количество слов** в текстовом файле (не строк). Сохраняет результат в JSON.
2	Функция создаёт **превью изображения** (миниатюру 200×200 пикселей) с использованием библиотеки Pillow. Зависимости указать в requirements.txt.
3	Вместо подсчёта строк функция выполняет **конвертацию CSV в JSON** и сохраняет JSON в выходной бакет.
4	Функция проверяет **MIME-тип** файла, и если это не текстовый файл – отправляет уведомление в Telegram (через webhook).
5	Триггер настроен только на **определённый префикс** (например, uploads/). Загрузить файл в папку uploads – обработать, вне папки – игнорировать.
6	Функция логирует метаданные (размер, тип) обработанного файла в **Yandex Monitoring** как custom metric. Посмотреть метрику.
7	Использовать **Terraform** для создания бакетов, сервисного аккаунта, функции и триггера. Приложить .tf-файлы.
8	Функция сжимает файл (GZIP) и сохраняет сжатый вариант в выходной бакет. Проверить размер.

№ вар.	Дополнительное задание (вариативная часть)
9	Реализовать **очередь (DLQ)** для сообщений, которые функция не смогла обработать после трёх попыток (Dead Letter Queue). Настроить Yandex Message Queue.
10	Функция после обработки **удаляет исходный файл** из входного бакета (чтобы не обрабатывать повторно).
11	Функция извлекает из загруженного ZIP-архива все текстовые файлы и обрабатывает каждый, сохраняя результаты в выходной бакет.
12	Добавить в функцию **асинхронную обработку** с помощью `concurrent.futures` для нескольких файлов (если триггер сработал на batch).
13	Создать **две функции**: первая – создаёт миниатюру, вторая – распознаёт текст на изображении (OCR). Связать их через последовательный вызов.
14	Вместо Python использовать **Node.js** для аналогичной обработки (подсчёт строк, запись в выходной бакет). Предоставить код.
15	Настроить **алерт** не на ошибки, а на длительность выполнения > 10 секунд. Показать уведомление.

6. Контрольные вопросы (для защиты)

1. Что такое FaaS (Function as a Service)? В чём преимущества перед запуском приложения на ВМ?
2. Как триггер Object Storage вызывает функцию? Какую информацию содержит событие (event)?
3. Какие роли нужны сервисному аккаунту для работы функции с бакетами?

4. Как в функции получить содержимое файла из бакета? Какие библиотеки можно использовать?

5. Какие ограничения (таймаут, память, размер диска) есть у cloud-функций в выбранном облаке?

6. Как просмотреть логи выполнения функции? Что делать, если функция падает с ошибкой?

7. Что такое «холодный старт» (cold start) в serverless? Как он влияет на время ответа?

8. Как настроить повторные попытки вызова функции при ошибке?

9. Как можно ограничить одновременное количество вызовов функции (concurrency)?

10. Почему после тестирования важно удалить триггер, функцию и бакеты? Какие риски при их оставлении?

7. Требования к отчёту

Отчёт оформляется по ГОСТ. Содержит:

1. **Титульный лист**.
2. **Цель и задачи работы**.
3. **Теоретическое обоснование** (FaaS, триггер, событие, сервисный аккаунт, логи, метрики).
4. **Практическая часть**:
   - Создание бакетов (скриншот).
   - Создание сервисного аккаунта и выдача ролей.
   - Листинг кода функции (с комментариями).
   - Переменные окружения (скриншот).
   - Создание триггера (скриншот конфигурации).
   - Тестирование: загрузка файла, логи функции, появление результата в выходном бакете (скриншоты).
   - Мониторинг: графики метрик, настройка алерта (скриншот).

- Очистка ресурсов (опционально – команды или скриншот удаления).
- **Вариативная часть** – дополнительные скриншоты/код.

5. **Результаты выполнения** (таблица: имя файла, размер, результат обработки, время выполнения функции).
6. **Анализ результатов** (преимущества serverless перед ВМ для обработки событий, удобство отладки через логи).
7. **Ответы на контрольные вопросы**.
8. **Вывод** (освоены: создание serverless-функции, интеграция с Object Storage через триггер, настройка мониторинга и алертов).

8. Критерии оценивания

Оценка	Критерии
«Отлично»	Выполнены **базовая + вариативная части**. Функция корректно обрабатывает загруженный файл, результат сохраняется в выходной бакет. Логи доступны, метрики и алерт настроены. Отчёт полный, все скриншоты приложены. Защита: глубокие ответы.
«Хорошо»	Выполнена **только базовая часть** (подсчёт строк/слов). Триггер работает, функция вызывается, результат есть, но вариативное задание отсутствует. Возможны мелкие недочёты (например, нет алерта).

Оценка	Критерии
«Удовлетворительно»	Работа выполнена частично: функция создана, но триггер не срабатывает, либо результат не сохраняется в выходной бакет. Мониторинг не настроен. Отчёт неполный.
«Неудовлетворительно»	Функция не развёрнута, триггер отсутствует, тестирование не проводилось. Отчёт не предоставлен.

СПИСОК ЛИТЕРАТУРЫ

1. **Эрл, Т.** Облачные вычисления. Сервис-ориентированный подход / Т. Эрл, Р. Путтини, Х. Као ; пер. с англ. – М. : ДМК Пресс, 2021. – 450 с. – ISBN 978-5-97060-874-2.

2. **Варфоломеева, А. А.** Облачные технологии и платформы : учебное пособие / А. А. Варфоломеева, А. В. Гаврилов. – СПб. : Университет ИТМО, 2022. – 180 с.

3. **Brikman, Y.** Terraform: Up and Running / Y. Brikman. – 3rd ed. – O'Reilly Media, 2022. – 486 p. – ISBN 978-1098116743.

4. **Kubernetes: Up and Running** / B. Burns, J. Beda, K. Hightower. – 3rd ed. – O'Reilly Media, 2022. – 320 p. – ISBN 978-1098110208.

5. **Документация Yandex Cloud** [Электронный ресурс]. – Режим доступа: https://yandex.cloud/ru/docs (дата обращения: 19.05.2026).

6. **AWS Documentation** [Электронный ресурс]. – Режим доступа: https://docs.aws.amazon.com (дата обращения: 19.05.2026).

7. **Маркин, А. В.** Облачные технологии: архитектура, сервисы, практика внедрения / А. В. Маркин. – М. : Наука и технологии, 2020. – 310 с. – ISBN 978-5-00078-345-2.

8. **Николаев, Д. С.** Инфраструктура как код: подходы и инструменты / Д. С. Николаев. – М. : ДМК Пресс, 2021. – 260 с. – ISBN 978-5-97060-925-1.

9. **Романенко, Е. В.** Контейнеризация и оркестрация приложений: Docker, Kubernetes / Е. В. Романенко. – М. : Инфра-Инженерия, 2022. – 220 с. – ISBN 978-5-9729-0890-5.

10. **Poulton, N.** Docker Deep Dive / N. Poulton. – 2023. – 350 p. – ISBN 979-8448397325.

11. **Мониторинг и логирование в облачных средах** / под ред. А. А. Крылова. – М. : Техносфера, 2021. – 290 с. – ISBN 978-5-94836-561-7.

Шинкаренко, И. А. Облачные базы данных: управляемые сервисы и практика администрирования / И. А. Шинкаренко. – СПб. : Питер, 2023. – 240 с. – ISBN 978-5-4461-2003-5.