

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»**

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ

по дисциплине «Управление облачными и гибридными платформами»

Направление подготовки 09.03.04 – Программная инженерия.

Профиль подготовки – Разработка, мониторинг и управление
жизненным циклом инженерных систем.

Квалификация выпускника - магистратура.

Очная форма обучения.

Ставрополь

2026

Содержание

Введение

БЛОК 1. Основы безопасности облачных платформ (ЛР №1–4)

Лабораторная работа №1. Развертывание защищенной облачной среды с использованием Terraform

Лабораторная работа №2. Управление секретами и ролями с HashiCorp Vault и IAM

Лабораторная работа №3. Настройка сетевой безопасности: VPC, Security Groups, VPN

Лабораторная работа №4. Безопасность контейнеров: анализ образов и политик Pod Security

БЛОК 2. Анализ и автоматизация безопасности (ЛР №5–9)

Лабораторная работа №5. Статический анализ конфигураций IaC (Checkov/tfsec)

Лабораторная работа №6. Написание политик безопасности на Rego/OPA

Лабораторная работа №7. Compliance как Code с Cloud Custodian

Лабораторная работа №8. Безопасность CI/CD Pipeline (GitLab CI / GitHub Actions)

Лабораторная работа №9. Аудит облачной инфраструктуры (Prowler/ScoutSuite)

БЛОК 3. Расширенные темы и управление затратами (ЛР №10–15)

Лабораторная работа №10. Управление затратами в облаке (FinOps)

Лабораторная работа №11. Service Mesh и mTLS (Istio)

Лабораторная работа №12. Внедрение Continuous Compliance

Лабораторная работа №13. Безопасное резервное копирование в гибридном облаке

Лабораторная работа №14. Логирование и мониторинг безопасности

Лабораторная работа №15. Балансировка нагрузки и атаки типа "Thundering Herd"

БЛОК 4. Итоговый проект (ЛР №16–18)

Лабораторная работа №16. Проектирование многооблачной архитектуры (AWS + YC)

Лабораторная работа №17. Реализация политик безопасности для «Bring Your Own Cloud»

Лабораторная работа №18. Комплексный проект: Аудит «Управление облачной платформой»

Введение

Дисциплина «Управление облачными и гибридными платформами» предназначена для магистрантов направления 09.04.04 «Программная инженерия» (профиль «Разработка, мониторинг и управление жизненным циклом инженерных систем») и изучается во втором семестре первого курса. Трудоёмкость составляет 3 зачётные единицы (108 академических часов), форма контроля — зачёт с оценкой.

Цель дисциплины - формирование у обучающихся системных знаний и практических навыков по безопасному проектированию, развертыванию и управлению облачными и гибридными инфраструктурами, обеспечивающими непрерывное соответствие требованиям безопасности на всех этапах жизненного цикла инженерных систем.

Практикум включает 18 лабораторных работ, сгруппированных в четыре тематических блока. Первый блок посвящён основам безопасности облачных платформ: развертыванию защищённой инфраструктуры как код, управлению секретами, сетевой изоляции и контейнерной безопасности. Второй блок охватывает анализ и автоматизацию: статический анализ IaC-конфигураций, написание политик на Rego/OPA, compliance как код, безопасность CI/CD-пайплайнов и комплексный аудит облачной среды. Третий блок расширяет компетенции в области FinOps, сервис-мешей с mTLS, непрерывного комплаенса, резервного копирования с шифрованием и централизованного логирования безопасности. Четвёртый блок представляет собой итоговый проект, объединяющий все полученные навыки в задачах проектирования многооблачной архитектуры, разработки политик для модели BYOC и выполнения полного цикла аудита безопасности облачной платформы.

Каждая лабораторная работа имеет единую структуру: цель, задачи, теоретическое обоснование, пример практического выполнения, индивидуальные задания трёх уровней сложности (базовый — 15 вариантов, средний — 15 вариантов, повышенный — 15 вариантов), контрольные вопросы, требования к отчёту и критерии оценивания. Такой подход обеспечивает дифференциацию обучения и позволяет студентам выбирать уровень погружения в зависимости от подготовки и профессиональных интересов.

Методической основой практикума выступает компетенция ПК-8 «Способен интегрировать практики безопасности на всех этапах ЖЦ, включая статический и динамический анализ кода, анализ уязвимостей зависимостей, безопасность контейнеров и облачной инфраструктуры, обеспечивая непрерывное соответствие требованиям безопасности». Лабораторные работы последовательно формируют индикаторы достижения данной компетенции: от сканирования IaC-конфигураций (ИД-5) и интеграции инструментов безопасности в CI/CD-пайплайны (ИД-6) до рантайм-мониторинга (ИД-7) и обеспечения непрерывного соответствия через автоматизированный сбор доказательств (ИД-8).

Лабораторная работа №1. Развертывание защищенной облачной среды с использованием Terraform

1. Цель

Освоить методологию «Инфраструктура как Код» (IaC) для безопасного проектирования и развертывания облачной среды. Научиться применять принцип наименьших привилегий при создании сервисных аккаунтов, описывать изолированные сетевые контуры (VPC, Security Groups) и управлять жизненным циклом облачных ресурсов с помощью Terraform.

2. Задачи

1. Создать сервисный аккаунт, наделенный минимально необходимыми правами для управления сетевыми и вычислительными ресурсами.
2. Разработать Terraform-конфигурацию, описывающую изолированную облачную сеть (VPC), подсеть и группу безопасности с запретом всего входящего трафика кроме SSH/ICMP с доверенного IP.
3. Выполнить стадию plan и apply, проанализировать вывод на предмет соответствия политикам безопасности до применения.
4. Обеспечить хранение чувствительных данных (state-файлов) с включенным шифрованием.
5. Валидировать полученную инфраструктуру на соответствие принципу наименьших привилегий (инвентаризация открытых портов, проверка ролей аккаунта).

3. Теоретическое обоснование

Метод «Инфраструктура как код» (IaC) предполагает описание желаемого состояния облачных ресурсов в декларативных файлах конфигурации, что исключает дрейф конфигураций и позволяет применять практики DevSecOps на ранних этапах. Terraform от HashiCorp является инструментом, реализующим IaC-подход через провайдер-специфичные ресурсы и модули.

Ключевые понятия безопасности в рамках данной ЛР:

- Принцип наименьших привилегий (PoLP) — сервисная учетная запись, используемая Terraform, должна иметь разрешения только на те действия (roles/compute.admin, roles/vpc.admin), которые необходимы для выполнения конкретной работы, и не должна обладать правами на чтение секретов или управление IAM.
- Сетевая микросегментация — вынесение ресурсов в отдельный VPC и разграничение доступа через Security Groups (межсетевой экран уровня VM) позволяет минимизировать поверхность атаки. Правила должны следовать модели "запрещено всё, что не разрешено явно".
- Управление состоянием (State) — Terraform хранит маппинг ресурсов в файле terraform.tfstate, который содержит конфиденциальные данные (IP-адреса, идентификаторы). Хранение state должно осуществляться в удаленном бэкенде (S3, Object Storage) с включенным server-side шифрованием, а не в локальной файловой системе или репозитории Git.

Связь с компетенцией ПК-8 (ИД-5): В данной работе закладывается основа для сканирования IaC-конфигураций (Checkov, tfsec). Любая написанная конфигурация, прежде чем попасть в пайплайн, должна пройти статический анализ на предмет наличия жестко заданных секретов и нарушений требований стандартов (CIS, PCI DSS).

4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

Для всех вариантов провайдер: Yandex Cloud или AWS. Необходимо предоставить main.tf и скриншот Security Group.

1. Развернуть VPC с одной публичной подсетью. Security Group открывает только 22 порт для всего Интернета (0.0.0.0/0). (Академическая иллюстрация, наихудшая практика для сравнения).
2. Развернуть VPC + 1 BM. В SG добавить разрешение 22 порта только для 10.100.0.0/16.
3. Использовать дефолтный VPC. Запретить весь входящий трафик, кроме 22 порта для 192.168.0.0/24.
4. Развернуть инфраструктуру, в которой порт 22 открыт только для IP-адреса, указанного через переменную `my_ip`.
5. Создать 2 подсети (публичная и приватная). Security Group только на публичной: разрешить SSH с 172.16.0.0/12.
6. Настроить правило SG, разрешающее порт 22 только для префикса `fd00::/8` (IPv6).
7. Развернуть BM без внешнего IP. SG разрешает входящий SSH только от балансировщика нагрузки (10.1.0.0/24).
8. Создать SG, разрешающую весь исходящий трафик, но запрещающую весь входящий (шлюзовой сервер).
9. При деплое в metadata добавить не только SSH-ключ, но и скрипт, который через `ufw` блокирует все порты, кроме 22.
10. Использовать динамический блок `ingress` в Terraform для итерации по списку разрешенных портов: `[22, 80]`.
11. Имя VPC задается через `random_pet` в Terraform. SG разрешает только ICMP.
12. Написать конфигурацию, где тег `SecurityTeam` обязателен для SG. Terraform должен выдавать ошибку при отсутствии тега (использовать `validation`).
13. Создать BM с локальным диском. Вынести чувствительные переменные (`folder_id`, `token`) в `*.auto.tfvars`, добавить файл в `.gitignore`.
14. Использовать `terraform plan -out=plan.cfg`. Показать опцию `-detailed-exitcode` в CI.
15. Разрешённые CIDR-ы передаются списком (`list(string)`) через `variables`.

4.2. Средний уровень (15 вариантов)

Требуется предоставление модульной структуры и скрипта валидации.

1. Создать модуль `secure-web-server`, который принимает `trusted_cidr` и создает VPC, подсеть, SG и BM. SG открывает 22 и 443.
2. Реализовать политику `deny-all` в SG: добавить нулевое правило с низким приоритетом, явно запрещающее любой трафик.
3. Настроить удаленный бэкенд на S3 (или Object Storage) с включенным `encrypt = true` и запретом публичного доступа через `bucket policy`.
4. Интегрировать `precondition` в ресурс `yandex_vpc_security_group`, проверяющий, что для порта 22 не разрешен 0.0.0.0/0.
5. Использовать data-источники для получения ID образа (Ubuntu 22.04) и динамического выбора зоны доступности.
6. Настроить `lifecycle { create_before_destroy = true }` для Security Group, чтобы минимизировать время разрыва правил при обновлении.

7. Создать две Security Groups (SSH и Web). VM использует обе. Проверить аддитивность правил.
8. Написать Terraform-конфигурацию, где ключи доступа аккаунта берутся из Vault (data source vault_generic_secret).
9. Реализовать variable_validation, запрещающее создание подсети с маской менее /24.
10. Реализовать output, возвращающий публичный IP только если у сети есть nat = true (использовать sensitive = true).
11. Развертывание двух идентичных сред (stage/prod) через for_each на уровне модуля.
12. VM должна создаваться только при условии, что переменная enable_monitoring = true (использовать count).
13. Добавить connection и provisioner remote-exec для автоматической установки fail2ban после деплоя.
14. Создать Security Group с диапазоном портов 3000-4000. Проверить наложение правил.
15. Сгенерировать SSH-ключи через tls_private_key, публичную часть передать в metadata.

4.3. Повышенный уровень (15 вариантов)

Предполагает использование Compliance as Code и сканеров.

1. Написать Sentinel/OPA-политику, запрещающую создание Security Group с правилом cidr_blocks = ["0.0.0.0/0"] для портов, отличных от 80/443.
2. Интегрировать tfsec в процесс валидации конфигурации. Доказать, что конфигурация проходит без ошибок MEDIUM и выше.
3. Реализовать Cloud-Init конфигурацию, которая настраивает iptables с дефолтной политикой DROP.
4. Автоматически ротировать сервисные ключи через time_rotating и обновлять state.
5. Использовать terraform console для вычисления количества открытых портов в генерируемой SG.
6. Написать custom provider check через terraform validate -json.
7. Настроить AWS Config / Yandex Audit Trails для логирования изменения SG. (Концептуальное задание, Terraform пишет в трейл).
8. Создать federated идентификацию для сервисного аккаунта (OIDC) вместо статических ключей. Описать процесс в README.
9. Разработать модуль, который создает VPC с flow-логами, записываемыми в бакет с включенным WORM-режимом (Object Lock).
10. Использовать check блок (Terraform 1.5+) для верификации того, что после деплоя проверка curl на закрытый порт 80 возвращает ошибку соединения.
11. Реализовать IaC, где VM имеет две сетевые карты в разных, изолированных подсетях (одна management, вторая data).
12. Настройка полного CI (GitLab CI) с этапами validate, tflint, tfsec, plan и apply only on main.

13. Написать Cloud Custodian политику, которая автоматически удаляет Security Group с 0.0.0.0/0 через 24 часа после создания, и test эту политику через pytest.
14. Выполнить terraform plan с флагом -refresh-only и реализовать обработку дрейфа конфигураций с оповещением в Slack (webhook).
15. Создать кастомный Terraform Provider, возвращающий статус проверки SG (разрешает ли он * на 0.0.0.0).

5. Контрольные вопросы

1. Почему хранение terraform.tfstate в публичном Git-репозитории является критичной уязвимостью?
2. Чем принципиально отличается безопасность сетевой модели Security Group от модели Network ACL (на примере AWS)?
3. В чем разница между аутентификацией через статические ключи и через OIDC-федерацию для Terraform?
4. Какие методы статического анализа IaC вы знаете и какие уязвимости они могут выявить на этапе plan?
5. Объясните, как работает механизм precondition и postcondition в Terraform и для чего он применяется в security-контексте?

6. Требования к отчету

Отчет оформляется в формате PDF/Markdown. Обязательные разделы:

1. Титульный лист: ФИО, группа, вариант.
2. Архитектурная схема: Нарисованная в draw.io (или аналоге) схема целевой сети с указанием подсетей, масок, групп безопасности и потоков трафика.
3. Тексты конфигураций: Полный листинг всех созданных .tf файлов с комментариями к security-решениям.
4. Доказательство безопасности:
 - Скриншот вывода terraform plan.
 - Скриншот Security Group из интерфейса облачного провайдера, подтверждающий отсутствие избыточных разрешений.
 - Вывод сканера tfsec или checkov (для среднего/высокого уровня в зависимости от варианта) с отсутствием ошибок.
5. Заключение: Анализ принципа наименьших привилегий для сервисного аккаунта и сетевых правил.

7. Критерии оценивания

7.1. Базовый уровень

- Корректно создан VPC и подсеть с валидными CIDR.
- Security Group открывает строго указанные в варианте порты.
- VM успешно создана и доступна по разрешенному протоколу.
- В отчете присутствуют скриншоты правил SG и вывода terraform apply.
- Оценка «удовлетворительно»: все 5 критериев выполнены полностью. При частичном нарушении (например, доступ открыт шире, чем в задании, но VM работает) — оценка снижается.

7.2. Средний уровень

- Выполнены все требования базового уровня.
- Конфигурация разбита на модули или использованы переменные с валидацией.
- State хранится в бэкенде с включенным шифрованием.
- Продемонстрирована работа механизма precondition или lifecycle.
- В отчете представлен вывод линтера (TFLint) или статический JSON-анализ плана.
- Оценка «хорошо»: выполнены все 4 критерия (наложение на базовый подразумевает отсутствие критических ошибок ниже).

7.3. Повышенный уровень

- Выполнены все требования среднего уровня.
- Реализована как минимум одна кастомная политика как код (OPA/Sentinel/Cloud Custodian), блокирующая небезопасную конфигурацию.
- CI/CD пайплайн (скриншот) отрабатывает стадии tflint, tfsec и plan.
- Ни один security scanner не выдает предупреждений уровней CRITICAL или HIGH.
- Проведен успешный аудит дрейфа конфигураций (terraform apply -refresh-only).
- Оценка «отлично»: выполнены все 5 критериев. При отсутствии одного из автоматизированных политик — оценка «хорошо» (с комментарием о потенциале усиления).

7.4. Снижение оценки

На любом уровне:

- Если terraform destroy не выполнен (оставлены ресурсы в облаке) — минус 1 балл.
- Наличие секретов (ключей) в незашифрованном виде в репозитории — минус 2 балла, возврат на доработку.
- Отсутствие структурного описания правил SG в отчете — оценка не выше «удовлетворительно».

Лабораторная работа №2. Управление секретами и ролями с HashiCorp Vault и IAM

1. Цель

Освоить принципы централизованного управления секретами в облачных и гибридных средах. Научиться настраивать HashiCorp Vault для динамической генерации учетных данных, создавать ролевые политики доступа (AppRole, IAM-подобные политики) и интегрировать прикладное ПО с Vault API для отказа от статических секретов в коде.

2. Задачи

1. Развернуть HashiCorp Vault в dev-режиме и освоить базовые операции через CLI и HTTP API.
2. Создать механизм аутентификации AppRole, эмулирующий ролевой доступ облачных ресурсов, и назначить политики минимально необходимых прав.
3. Настроить движок динамических секретов базы данных (Database Secrets Engine) для PostgreSQL, обеспечив создание временных учетных записей.
4. Разработать простое Python-приложение, получающее динамические креды из Vault и выполняющее запрос к БД.
5. Продемонстрировать истечение срока аренды (lease) и автоматическую ротацию пароля в PostgreSQL.

3. Теоретическое обоснование

Проблема "секретного расплзания" (Secret Sprawl): Традиционный подход хранения паролей в переменных окружения, конфигурационных файлах или жестком кодировании приводит к неконтролируемому распространению чувствительных данных по окружениям, репозиториям и логам CI/CD. HashiCorp Vault решает эту проблему, выступая единым источником истины (Source of Truth) для всех типов секретов.

Ключевые концепции безопасности:

- Методы аутентификации (Auth Methods): способ, которым пользователь или машина подтверждают свою личность перед Vault. Метод AppRole специально разработан для машин (сервисов, VM, CI/CD раннеров) и предполагает получение Role ID (идентификатор роли) и Secret ID (одноразовый пароль), что идеально вписывается в модель безопасности облачных платформ (аналог IAM ролей для Compute Instances без использования статических ключей).
- Политики (Policies): декларативные HCL-документы, определяющие, какие пути (paths) в Vault доступны и какие операции разрешены (create, read, update, delete, list). Политики реализуют Принцип наименьших привилегий (PoLP) на уровне хранилища секретов, что соответствует индикатору ИД-8пк-8 (реализация Compliance as Code).
- Динамические секреты: В отличие от статических (key-value), динамические секреты генерируются по запросу. При подключении к базе данных Vault создает уникальную временную роль в СУБД, привязанную к аренде (lease). По истечении TTL этой аренды Vault автоматически удаляет роль из базы данных. Это исключает риск утечки долгоживущих паролей и обеспечивает эфемерность доступа — основу современного DevSecOps.
- Lease (Аренда): Каждый динамический секрет имеет период действия (TTL). Приложение должно уметь отслеживать TTL и запрашивать перевыпуск секрета до истечения срока, либо корректно обрабатывать Permission denied. Максимальный TTL ограничивает время жизни даже продлеваемого секрета.

Связь с компетенцией ПК-8: Данная лабораторная работа закладывает практический фундамент для задачи ИД-5пк-8 (сканирование конфигураций) и ИД-6пк-8 (интеграция Quality Gates), так как в CI/CD пайплайнах автоматические системы деплоя должны обращаться к Vault посредством ролей, а не хардкодить DB_PASSWORD в переменных GitLab.

4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

Провайдер: Vault dev-режим (Docker).

1. Записать key-value секрет secret/myapp/config с ключами api_key и db_host. Прочитать через CLI.
2. Создать политику, которая разрешает чтение пути secret/data/ и запрещает запись. Проверить доступ.
3. Настроить AppRole роль с токеном TTL 30 минут. Войти под ней через CLI.
4. Создать статический KV-секрет. Написать Python-скрипт использующий токен для чтения этого пути.
5. Создать два namespace (если версия Enterprise Sandbox) или два изолированных KV-движка (secret/app1 и secret/app2). Показать изоляцию прав.
6. Движок database подключен к PostgreSQL. Получить динамические креды через CLI.
7. Написать политику AppRole, запрещающую доступ к sys/ и auth/.
8. Получить динамические креды БД. Подключиться к PostgreSQL вручную через psql и проверить SELECT права.
9. Создать второй KV-движок с версионированием (kv2). Записать 3 версии секрета и откатить на первую.
10. Вывести lease_id и lease_duration после получения динамических кредов.
11. Настроить токен с политикой default и app-policy. Проверить, что чтение sys/mounts разрешено через default.
12. Приложение на Python логирует время истечения аренды.
13. Использовать curl вместо CLI для получения secret-id для AppRole (REST API).
14. Создать в Vault policy, которая использует параметризованные пути (path templating) с {{identity.entity.name}}.
15. Отозвать (revoke) токен доступа через CLI и проверить, что приложение потеряло доступ к Vault.

4.2. Средний уровень (15 вариантов)

Требуется написание полноценного Python-приложения и работа с lease.

1. Python-приложение обрабатывает VaultAuthExpired исключение и перевыпускает токен.
2. Настроить базу данных так, чтобы роль app-readonly-role также имела право на выполнение хранимой процедуры.
3. Реализовать механизм renew (продления) lease в Python-приложении (крон или асинхронная задача).

4. Политика приложения запрещает доступ к любым путям в `secret/`, кроме конкретного, заданного через `bound_claims`.
5. Использовать `response wrapping` для безопасной доставки `secret-id` приложению. Продемонстрировать однократное разворачивание (`unwrap`).
6. Настроить `audit-устройство` (`file`) и предоставить лог аудита с записью о получении кредов. Лог не должен содержать самих паролей.
7. Интеграция с PostgreSQL, где `creation_statements` назначают права только на выборку из одной конкретной таблицы, а не всей схемы.
8. Создать две роли БД: `reader` и `writer`. Python-приложение получает креды для той роли, которая указана в параметре командной строки.
9. Настроить `max_connections` на уровне роли БД, чтобы Vault создавал не более 2 одновременных активных пользователей.
10. Реализовать мониторинг истечения аренд: Python-скрипт каждые 5 секунд выводит оставшееся время TTL.
11. Настроить политику Vault так, чтобы разрешить `renew` только если оставшийся TTL меньше 30 секунд (условный доступ не предусмотрен, но эмулируется кодом приложения).
12. Использовать `token_lookup_self` для получения информации о сроке действия токена и правах.
13. Docker Compose: поднять Vault + PostgreSQL + Python-приложение в одной сети. Исключить открытие порта БД наружу.
14. Написать тест (`pytest`) с использованием `hvac TestClient`, проверяющий, что после `revoke` запрос к БД падает.
15. Креды на подключение к самому Vault для `AppRole` передаются через зашифрованный файл (GPG), расшифровываемый в рантайме.

4.3. Повышенный уровень (15 вариантов)

Предполагает использование Transit, Auto-unseal, интеграции с Kubernetes и облачные практики.

1. Настроить Transit Secret Engine для шифрования данных в приложении. Python шифрует лог-файл ключом, сгенерированным в Vault.
2. Kubernetes Auth Method: Vault аутентифицирует Pod в Minikube по Service Account и выдает секреты БД.
3. Настроить Vault Agent с шаблоном для автоматического инжекта секретов в sidecar-контейнер. Приложение читает файл из общей памяти.
4. Реализовать полный zero-trust: PostgreSQL не имеет мастер-пароля после конфигурации, пароль root генерируется Vault'ом и хранится только в его памяти.
5. Secrets Engine для AWS/Azure/Yandex Cloud: Python-приложение получает временные IAM-ключи и создает бакет в Object Storage.
6. Настроить Auto-unseal Vault с использованием облачного KMS (AWS KMS/Yandex KMS). Демонстрация процесса.
7. Написать Sentinel/OPA политику в Vault (если версия Enterprise) или эмулировать её в Python: запретить выдачу секрета, если TTL > 24h.
8. Производить ротацию корневых учетных данных PostgreSQL через Vault API (`rotate-root`), не прерывая работу приложения.

9. Настроить Namespaces (Enterprise Sandbox) и показать, что приложение из NS "dev" не может прочитать секреты NS "prod".
10. Внедрить PKI Secrets Engine: Vault становится СА. Приложение получает TLS-сертификат для mTLS подключения к БД.
11. Python-приложение использует "Lease Cache" (in-memory dictionary) для минимизации обращений к Vault при сохранении актуальности.
12. Apache Airflow DAG, который получает динамический пароль из Vault, выполняет ETL-задачу и отправляет сигнал о завершении (с освобождением lease).
13. Использование Vault как OIDC-провайдера для доступа к внутреннему веб-приложению (Vault Identity Token).
14. Интеграция с HSM (эмуляция SoftHSM) для разблокировки Vault.
15. Разработать кастомный плагин аутентификации (backend) для Vault (например, проверка MAC-адреса VM).

5. Контрольные вопросы

1. В чем ключевое различие между статическим key-value секретом и динамическим секретом Database Engine с точки зрения безопасного жизненного цикла пароля?
2. Объясните механизм Response Wrapping. Для чего он предназначен в CI/CD пайплайнах?
3. Какие методы защиты Secret ID от компрометации существуют при использовании AppRole в облачных средах (например, на AWS EC2)?
4. Что произойдет с долгоиграющим приложением, если истечет TTL аренды динамического секрета? Как грамотно обрабатывать эту ситуацию?
5. Какие метаданные передаются в IAM-политиках Vault (Identity, Aliases) и как они связаны с гетерогенными бэкендами аутентификации?

6. Требования к отчету

Отчет оформляется в формате PDF. Состав:

1. Титульный лист.
2. Архитектурная диаграмма: Поток получения секретов (Python App → Vault API → PostgreSQL) с указанием протоколов, портов и методов аутентификации.
3. Конфигурации и код:
 - app-policy.hcl (политика Vault) с комментариями.
 - app.py (полный листинг приложения).
 - Вывод psql с результатом работы SELECT current_user.
4. Доказательства безопасности:
 - Скриншот вывода vault lease list, демонстрирующий наличие или отсутствие активной аренды.
 - Лог аудита Vault (упоминание запроса, но не сам пароль в теле).
 - Скриншот ошибки подключения к БД после истечения TTL (или после revoke).

5. Заключение: разбор преимуществ динамических секретов перед статическими в гибридных системах.

7. Критерии оценивания

7.1. Базовый уровень («удовлетворительно»)

- Vault запущен, KV или DB движок активен.
- Присутствует корректная политика с разрешением и запретом путей.
- Получен статический или динамический секрет через CLI/Python.
- Продемонстрировано подключение к PostgreSQL.
- Нет грубых ошибок (секрет не зашит в коде в репозитории).
- Снижение: политика дает избыточные права (на запись там, где достаточно чтения) — минус 1 балл.

7.2. Средний уровень («хорошо»)

- Выполнены все требования базового уровня.
- Python-приложение использует AppRole без статического корневого токена.
- Реализована обработка ошибок аутентификации и/или продление аренды.
- Продемонстрирована изоляция: токен не может прочитать секреты, к которым политика не дает прав.
- В отчете присутствует скриншот audit-лога.
- Снижение: отсутствует механизм переподключения к БД при истечении аренды — минус 1 балл.

7.3. Повышенный уровень («отлично»)

- Выполнены все требования среднего уровня.
- Интегрирован один из облачных IAM-методов (Kubernetes Auth, облачный IAM, mTLS через PKI).
- Реализован Advanced движок (Transit, Dynamic Cloud IAM Keys, Rotate Root).
- Написан юнит-тест, проверяющий работоспособность получения секретов и отзыва.
- Показана работа Vault Agent или асинхронное обновление секретов без остановки приложения.
- Снижение: отсутствует автоматическое тестирование или ротация — минус 1 балл, оценка «хорошо» при всех остальных компонентах.

7.4. Общие штрафы

- Ключи доступа к Vault или БД сохранены в незашифрованном виде в системе контроля версий — возврат отчета на доработку, максимальная оценка «удовлетворительно».
- Если после занятия не сброшены тестовые root-токены — минус 2 балла.

Лабораторная работа №3. Настройка сетевой безопасности: VPC, Security Groups, VPN

1. Цель

Освоить практические методы построения защищенных сетевых топологий в облачных средах. Научиться проектировать многоуровневую архитектуру с публичными и приватными подсетями, настраивать шифрованные туннели (Site-to-Site VPN) и реализовывать модель доступа, исключаящую прямое взаимодействие с чувствительными ресурсами из интернета.

2. Задачи

1. Спроектировать и развернуть изолированную облачную сеть (VPC), содержащую минимум две подсети: публичную (DMZ) и приватную (Backend).
2. Настроить группы безопасности (Security Groups), обеспечивающие строгую фильтрацию трафика между сегментами и из внешней сети.
3. Развернуть VPN-сервер (WireGuard или OpenVPN) на узле в публичной подсети и организовать Site-to-Site туннель с локальной рабочей станцией.
4. Разместить тестовый чувствительный ресурс (базу данных PostgreSQL) в приватной подсети, исключив назначение публичного IP-адреса.
5. Верифицировать доступность приватного ресурса исключительно через VPN-туннель (транзитом через публичный узел) и отсутствие прямого доступа из интернета.

3. Теоретическое обоснование

Многоуровневая сетевая архитектура (Tiered Network) — фундаментальный принцип защиты облачных инфраструктур, предполагающий разделение ресурсов по уровням чувствительности и ограничение сетевого взаимодействия между ними. В классической реализации выделяют публичную подсеть (DMZ, фронтальные сервисы с выходом в интернет) и приватную подсеть (базы данных, бэкенды, системы хранения секретов — без прямого выхода в интернет).

Ключевые компоненты безопасности:

- VPC (Virtual Private Cloud) — логически изолированный участок облака, в котором администратор полностью контролирует IP-адресацию, маршрутизацию и правила фильтрации. Ресурсы в разных VPC по умолчанию не имеют связности.
- Security Groups (SG) — виртуальный межсетевой экран уровня инстанса (stateful). Правила SG определяют разрешенный входящий и исходящий трафик на основе протоколов, портов и источников/назначений (CIDR или другие SG). Ключевая особенность: stateful-обработка — если входящий пакет разрешен, ответный трафик автоматически пропускается, что упрощает правила для ответных соединений.
- VPN (Virtual Private Network) — технология создания зашифрованного канала связи поверх публичных сетей. Для гибридных облачных сред критически важны протоколы WireGuard (высокая производительность, малый объем кода, современная криптография) и OpenVPN (зрелость, поддержка enterprise-интеграций). VPN-шлюз в публичной подсети выступает единственной точкой входа в приватный сегмент.

- Bastion Host / Jump Server — концептуально VPN-сервер выполняет роль хоста-бастиона: укрепленного узла, принимающего на себя все внешние подключения и транслирующего их во внутреннюю сеть после строгой аутентификации.

Принцип нулевого доверия (Zero Trust) в сетевом контексте: даже находясь внутри VPC, ресурсы не должны доверять друг другу по умолчанию. Каждое межсервисное соединение должно быть явно разрешено правилом SG. Например, приватная БД принимает входящие соединения на порт 5432 только от Security Group публичного VPN-узла, а не от всей подсети 10.0.0.0/8.

Связь с компетенцией ПК-8 (ИД-5, ИД-7): Настройка Security Groups как кода (в Terraform или консоли) является частью практики «Инфраструктура как Код», подлежащей статическому анализу (ИД-5). Мониторинг сетевых потоков (VPC Flow Logs, tcpdump на VPN-шлюзе) и обнаружение попыток прямого подключения к приватному ресурсу реализуют практики рантайм-мониторинга (ИД-7).

4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

Провайдер: Yandex Cloud или AWS (на выбор студента). Требуется предоставить схему, конфигурации и доказательство подключения.

1. Yandex Cloud: одна публичная подсеть, одна приватная. VPN — WireGuard. Доступ к NGINX на приватной ВМ через VPN.
2. AWS: VPC + 2 подсети. VPN — OpenVPN (образ из Marketplace). Приватный ресурс — Apache.
3. Yandex Cloud: до приватной ВМ можно достучаться по SSH только через VPN (Port Forwarding через bastion).
4. AWS: Security Group приватной ВМ ссылается на Security Group VPN-сервера (не на IP).
5. Yandex Cloud: NAT-шлюз для приватной подсети для обновлений, но без входящих правил.
6. AWS: VPN-сервер — t2.micro, приватная ВМ — t3.micro с MySQL.
7. Yandex Cloud: используется только протокол UDP для VPN (порт 51820), TCP-порты VPN не открыты.
8. AWS: демонстрация Flow Logs, в которых видны ТОЛЬКО разрешенные пакеты (REJECT отсутствует для правильного трафика).
9. Yandex Cloud: на VPN-шлюзе настроен iptables, маскирующий трафик для подсети 10.20.2.0/24.
10. AWS: Route Table приватной подсети имеет маршрут 0.0.0.0/0 только на NAT Gateway, не на IGW.
11. Yandex Cloud: tcpdump на VPN-шлюзе показывает только SSH и WireGuard трафик, отсутствие сканирований портов.
12. AWS: использование `aws ec2 describe-security-groups` для автоматической проверки, что нет 0.0.0.0/0 на портах, кроме VPN.
13. Yandex Cloud: VPN-сервер и БД находятся в разных зонах доступности (ru-central1-a и ru-central1-b).
14. AWS: CloudWatch-метрика VPNConnectionState или кастомный health-check WireGuard.

15. Yandex Cloud: статический маршрут в VPC для направления трафика 10.99.0.0/24 через VM VPN-шлюза.

4.2. Средний уровень (15 вариантов)

Требуется Terraform-конфигурация всей сетевой инфраструктуры и автоматизированное тестирование.

1. Полная автоматизация: Terraform поднимает VPC, подсети, SG, обе VM, инициализирует WireGuard ключи через cloud-init и tls_private_key.
2. Реализовать Double-hop VPN: локальная машина → VPN в облаке → еще одна VM-прокси во второй приватной подсети.
3. Настроить pritunl (OpenVPN с веб-интерфейсом) в Docker на публичной VM и интегрировать с LDAP/FreeIPA на приватной VM.
4. Написать Ansible-плейбук, который автоматически разворачивает WireGuard на обоих концах, включая обмен публичными ключами.
5. Создать автоматический тест (pytest+paramiko), который до поднятия VPN проверяет недоступность БД, а после — успешный SELECT.
6. Настроить VPN-туннель между двумя VPC в разных регионах (Transit VPC) через два WireGuard-инстанса.
7. Настроить мониторинг: Prometheus + Grafana на VPN-шлюзе собирают метрики трафика туннеля (wg show), алерт при падении handshake.
8. Настроить Split-tunneling: через VPN идут только запросы к 10.20.0.0/16, остальной трафик локальной машины идет напрямую.
9. Использовать iptables на VPN-шлюзе для ограничения: с туннеля можно обращаться только к портам 5432 и 22 приватной сети.
10. Настроить резервный VPN-шлюз (High Availability) с Keepalived и общим виртуальным IP.
11. Внедрить сертификатную аутентификацию для OpenVPN, используя Easy-RSA. Ключи генерирует Vault (PKI engine).
12. Terraform output генерирует готовый конфиг-файл WireGuard для локального клиента и сохраняет в S3-бакет с шифрованием.
13. Настроить IdP-интеграцию: доступ к VPN возможен только после OIDC-аутентификации через Vault (для OpenVPN auth plugin).
14. Развернуть приватный Kubernetes-кластер в приватной подсети, доступ к API-серверу — только через VPN.
15. Сравнительный бенчмарк: замер пропускной способности БД (pgbench) напрямую (пока есть доступ) и через VPN-туннель, анализ оверхеда.

4.3. Повышенный уровень (15 вариантов)

Предполагает использование Service Mesh, mTLS и Compliance-проверок.

1. Внедрить Istio Service Mesh на обеих VM (или мини-кластер). Настроить mTLS между VPN-шлюзом и приватным сервисом.
2. Включить VPC Flow Logs и написать Cloud Function/Lambda, автоматически блокирующую SG-правило при обнаружении сканирования портов.
3. Настроить WireGuard с постквантовой криптографией (экспериментальная), сравнить handshake latency.

4. Написать OPA-политику, проверяющую Terraform-план: запрещено создание VM в приватной подсети с публичным IP (`associate_public_ip_address = true`).
5. Развернуть Pomerium/oauth2-проxy перед VPN: доступ к туннелю возможен только после успешной SSO-аутентификации.
6. Реализовать Chaos Engineering тест: случайный обрыв VPN-туннеля и автоматическое переключение на резервный канал (VPN-over-VPN).
7. Настроить Multi-cloud VPN: локальная машина → Yandex Cloud → AWS (транзит через облако), используя разные протоколы на участках.
8. Реализовать Network Policy в Kubernetes (Calico) для подов в приватной подсети, разрешающую ingress только от pod'a с лейблом `role=vpngateway`.
9. Настроить углубленный аудит: auditbeat на VPN-шлюзе собирает все SSH-сессии и передает в ELK для анализа инцидентов.
10. Разработать кастомный Terraform провайдер/ресурс, который проверяет доступность приватного эндпоинта через VPN и отображает статус.
11. Развернуть HashiCorp Boundary для доступа к приватной БД: Boundary-контроллер в публичной подсети, worker — в приватной, протокол — поверх WireGuard.
12. Написать Sentinel-политику, которая запрещает terraform apply, если SG для порта 22 содержит 0.0.0.0/0 (CIS AWS 4.1).
13. Реализовать DPI (Deep Packet Inspection) на VPN-шлюзе с помощью Suricata для обнаружения SQL-инъекций в трафике к БД.
14. Настроить полную автоматическую ротацию ключей WireGuard (раз в 60 минут) с синхронизацией через Vault и wg syncconf без разрыва соединений.
15. Построить инфраструктуру с использованием только IPv6 в облаке. Настроить WireGuard поверх IPv6-only сети, включая туннель IPv4-трафика (Dual Stack Light).

5. Контрольные вопросы

1. Почему ссылка на Security Group (вместо CIDR-блока) в правилах входящего трафика считается более безопасной и удобной практикой при автоматическом масштабировании?
2. В чем отличие модели сетевой безопасности на основе Security Groups от модели на основе Network ACL? Почему SG называют stateful, а NACL — stateless?
3. Какие криптографические примитивы использует WireGuard и почему его кодовая база считается более безопасной (меньшая поверхность атаки) по сравнению с OpenVPN?
4. Объясните механизм Hairpin NAT применительно к VPN-шлюзу. Почему он критичен для доступа из приватной подсети обратно к VPN-клиентам?
5. Каким образом интеграция Vault PKI с OpenVPN позволяет реализовать Zero Trust архитектуру с автоматическим отзывом сертификатов?

6. Требования к отчету

Отчет в формате PDF. Обязательные разделы:

1. Титульный лист.
2. Архитектурная схема сети: детальная диаграмма (draw.io или аналог) с указанием IP-адресации, подсетей, Security Groups, VPN-туннеля и направления трафика.

3. Конфигурационные файлы:

- Terraform-манифесты (network.tf, security.tf, compute.tf) с комментариями.
- Конфигурации WireGuard/OpenVPN сервера и клиента (ключи замаскированы).

4. Доказательства изоляции и связности:

- Скриншот вывода `curl ifconfig.me / ping` с приватной ВМ (отсутствие интернета).
- Вывод `wg show` с подтверждением handshake и трафика.
- Скриншот успешного `psql` подключения к приватной БД с локальной машины.
- Скриншот ошибки подключения при выключенном VPN.

5. Анализ логов: фрагмент вывода `tcpdump` или VPC Flow Logs, демонстрирующий прохождение трафика только через разрешенные порты.

6. Заключение: обоснование необходимости multi-tier сетевой архитектуры в гибридных облачных средах.

7. Критерии оценивания

7.1. Базовый уровень («удовлетворительно»)

- VPC содержит минимум две подсети с различной маршрутизацией.
- Security Groups настроены, присутствует минимум одно правило, ограничивающее источник трафика.
- VPN-туннель установлен (handshake в `wg show` или статус OpenVPN).
- Приватная ВМ не имеет публичного IP.
- Подтверждено успешное подключение к приватному ресурсу через VPN.
- Снижение: SG разрешает прямой доступ к БД из интернета (0.0.0.0/0) — работа не принимается.

7.2. Средний уровень («хорошо»)

- Выполнены все критерии базового уровня.
- Инфраструктура развернута полностью через Terraform (не вручную в консоли).
- Настроен форвардинг и MASQUERADE, обеспечивающие доступ из VPN-туннеля в приватную сеть.
- Приватная БД принимает соединения только от SG VPN-шлюза, а не по CIDR.
- В отчете присутствует автоматизированный тест (скрипт) проверки доступности.
- Снижение: отсутствует `tcpdump` или Flow Logs — минус 1 балл.

7.3. Повышенный уровень («отлично»)

- Выполнены все критерии среднего уровня.

- Внедрен дополнительный уровень безопасности: mTLS, OPA-политика на Terraform-план, ротация ключей или SSO-аутентификация перед VPN.
- Реализован мониторинг состояния туннеля с алертами (Prometheus + Grafana или CloudWatch).
- Продемонстрирована работа Compliance as Code (политика, блокирующая неправильную SG).
- Выполнен успешный аудит логов безопасности (Suricata, auditbeat, VPC Flow Logs в SIEM).
- Снижение: политика как код не реализована или не блокирует нарушение — минус 1 балл (оценка «хорошо»).

7.4. Общие штрафы на всех уровнях

- Приватный ключ WireGuard/OpenVPN выложен в открытом виде в репозиторий — возврат на доработку, максимальная оценка «удовлетворительно».
- Инфраструктура не уничтожена (terraform destroy не выполнен) — минус 2 балла.
- Отсутствует схема сети с IP-адресами — минус 1 балл.

Лабораторная работа №4. Безопасность контейнеров: анализ образов и политик Pod Security

1. Цель

Освоить инструменты и методики обеспечения безопасности контейнеризированных приложений на этапах сборки и развертывания. Научиться проводить статический анализ Docker-образов на наличие уязвимостей (CVE) и применять встроенные политики Pod Security Admission (PSA) для предотвращения запуска привилегированных рабочих нагрузок в Kubernetes.

2. Задачи

1. Собрать эталонный (заведомо уязвимый) и исправленный (hardened) Docker-образы приложения.
2. Провести сканирование образов с помощью Trivy, проанализировать полученный отчет об уязвимостях, классифицировать CVE по уровням критичности.
3. Развернуть тестовый Kubernetes-кластер (minikube / kind) с активированным Pod Security Admission в режиме enforce.
4. Создать манифесты подов с различными нарушениями политик безопасности (привилегированный режим, запуск от root, hostPath volume) и продемонстрировать их блокировку.
5. Адаптировать манифест пода к требованиям restricted-политики, обеспечив безопасный запуск контейнера.

3. Теоретическое обоснование

Безопасность контейнеров — многоуровневая задача, охватывающая этапы сборки образа, хранения в реестре и рантайм-эксплуатации в оркестраторе. Современный DevSecOps-пайплайн требует автоматизированного анализа на каждом из этих этапов.

Сканирование образов (Static Analysis):
Контейнерный образ состоит из слоев, каждый из которых может содержать уязвимые библиотеки и пакеты. Trivy (Aqua Security) — сканер с открытым исходным кодом, анализирующий:

- Установленные OS-пакеты (dpkg, rpm, apk) на наличие известных CVE (Common Vulnerabilities and Exposures);
- Зависимости приложений (Gemfile.lock, package-lock.json, pom.xml);
- IaC-конфигурации (Dockerfile, Kubernetes-манифесты) на предмет нарушения лучших практик (misconfigurations).

Ключевые метрики: CRITICAL, HIGH, MEDIUM, LOW — определяют приоритет исправления. Наличие хотя бы одной CRITICAL-уязвимости должно блокировать продвижение образа в production-реестр (Quality Gate, ИД-6пк-8).

Pod Security Admission (PSA):
Начиная с Kubernetes 1.25, механизм PodSecurityPolicy (PSP) заменен на встроенный контроллер Pod Security Admission. PSA оперирует тремя уровнями политик:

- Privileged — unrestricted, полный доступ (не рекомендуется);
- Baseline — запрещает наиболее опасные практики: запуск от root, привилегированные контейнеры, hostPath volumes, использование hostNetwork/hostPID;
- Restricted — накладывает дополнительные ограничения, требуя явного указания securityContext с allowPrivilegeEscalation: false, runAsNonRoot: true, seccompProfile: RuntimeDefault, и ограничения возможностей (capabilities drop ALL).

Политики применяются на уровне namespace'ов с помощью лейблов:

- pod-security.kubernetes.io/enforce: уровень принудительного применения (блокировка создания пода);
- pod-security.kubernetes.io/audit: уровень аудита (логгирование нарушений);
- pod-security.kubernetes.io/warn: уровень предупреждений (warning при создании).

Связь с компетенцией ПК-8:

- ИД-5пк-8: Сканирование конфигураций (Trivy fs scanner для Dockerfile и манифестов).
- ИД-7пк-8: Мониторинг состояния безопасности контейнеров в рантайме (PSA в режиме enforce — предотвращение запуска привилегированных подов).
- ИД-8пк-8: Непрерывное соответствие — политики PSA как реализация Compliance as Code для Kubernetes-кластера.

4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

Окружение: локальный Docker, minikube/kind с PSA restricted.

1. Собрать образ nginx:1.21 (старая версия). Trivy выявить HIGH-уязвимости, предоставить отчет.
2. Написать Deployment манифест с privileged: true, продемонстрировать его блокировку restricted-политикой.
3. Создать под с runAsUser: 0 (root) в namespace restricted. Зафиксировать ошибку.

4. Активировать политику `restricted` в режиме `warn` (не `enforce`). Показать, что под запускается, но администратор получает `warning`.
5. Создать два `namespace`: `dev` (`baseline`) и `prod` (`restricted`). Один и тот же привилегированный под запускается в `dev` и блокируется в `prod`.
6. Использовать `Trivy` для сканирования официального образа `python:3.9-alpine`. Выписать все `CRITICAL`-уязвимости.
7. Запустить под с `hostNetwork: true` в `restricted-namespace`. Зафиксировать блокировку.
8. Написать манифест пода, нарушающий `restricted`-политику из-за отсутствия `allowPrivilegeEscalation: false`.
9. `Trivy scan` образа `node:18-bullseye`. Сравнить количество уязвимостей с `node:20-bookworm-slim`.
10. Создать `namespace` с `PSA enforce: baseline`. Попытаться запустить под с `volume hostPath`. Проверить, что `baseline` блокирует.
11. Сканирование `Dockerfile` через `trivy config Dockerfile`. Выявить `best-practice violations` (например, `USER root`).
12. Создать под с `seccompProfile: Unconfined` в `restricted-namespace`. Зафиксировать блокировку.
13. Использовать `trivy image --ignore-unfixed` для исключения уязвимостей без исправлений.
14. Под с `capabilities.add: ["NET_ADMIN"]` и `capabilities.drop: []` в `restricted-namespace`. Проверить блокировку.
15. `Trivy` в режиме `--light` (только фиксированные уязвимости, быстрый скан).

4.2. Средний уровень (15 вариантов)

Требуется CI/CD-интеграция и автоматизация проверок.

1. Настроить `Trivy` в `GitHub Actions`/`GitLab CI`: пушить образ только при 0 `CRITICAL`-уязвимостей (`Quality Gate`).
2. Написать скрипт (`bash`/`Python`), который парсит `JSON`-отчет `Trivy` и при наличии хотя бы одной `HIGH`-уязвимости завершается с кодом 1.
3. Создать `Helm`-чарт с `securityContext` по умолчанию, соответствующим `restricted`-политике.
4. Реализовать `Mutating Admission Webhook` (на базе `OPA Gatekeeper` или `Kyverno`), автоматически добавляющий `seccompProfile: RuntimeDefault` ко всем подам.
5. `Trivy` в режиме `client-server`: развернуть `Trivy Server`, настроить периодическое сканирование образов в локальном реестре `Harbor`.
6. Написать `Kyverno ClusterPolicy`, запрещающую использование образов с тегом `:latest`.
7. Собрать `multi-stage Docker`-образ, где сборочные зависимости не попадают в финальный образ. `Trivy` подтверждает отсутствие `build-only` уязвимостей на финальном слое.
8. Настроить `Falco` в тестовом кластере, демонстрирующий обнаружение запуска привилегированного контейнера (даже если `PSA` пропустил).

9. Сгенерировать SBOM (Software Bill of Materials) для образа через `trivy image --format cyclonedx` и проанализировать зависимости.
10. Написать Python-скрипт, сравнивающий два Trivy-отчета (до и после обновления базового образа) и выводящий разницу в CVE.
11. Интегрировать Trivy с DefectDojo для управления уязвимостями: push отчета через API.
12. Создать кастомный Dockerfile на базе `scratch` с единственным бинарником (Go-приложение). Trivy показывает 0 уязвимостей.
13. Kyverno-политика, блокирующая поды без `resource limits` и `requests` в `namespace prod`.
14. Настроить `cosign` (Sigstore) для подписи образа, проверить через Trivy подлинность подписи.
15. Развернуть Kubernetes Dashboard с `read-only` сервисным аккаунтом в `restricted-namespace`. Показать, что сама Dashboard работает, но её под запускается с соблюдением `restricted`.

4.3. Повышенный уровень (15 вариантов)

Требуется построение комплексной системы безопасности контейнеров.

1. Развернуть Open Policy Agent (OPA) Gatekeeper с `constraint`'ами из библиотеки Gatekeeper Library. Продемонстрировать блокировку пода без `runAsNonRoot`.
2. Реализовать полный DevSecOps-пайплайн: GitHub Actions → Trivy scan → Cosign sign → Kyverno verify → deploy в `restricted-namespace`.
3. Написать e2e-тест (`pytest + kubernetes client`), который проверяет, что привилегированный под действительно не создается в `prod namespace`.
4. Настроить сканирование образов в рантайме (`runtime scanning`) с помощью Tetragon или Tracex. Обнаружить попытку выполнения `chroot` внутри контейнера.
5. Развернуть Harbor с Trivy-сканером. Настроить политику реестра, блокирующую pull образов с CRITICAL-уязвимостями.
6. Внедрить Kubernetes NetworkPolicies в дополнение к PSA: под в `restricted`-пространстве не может инициировать исходящие соединения кроме как к DNS.
7. Создать кастомный PSA-уровень (Admission Controller на базе ValidatingWebhookConfiguration), который дополнительно проверяет наличие метки `owner` у пода.
8. Интеграция с Vault: под получает секреты через Vault Agent Injector. PSA-политика разрешает только этот специфичный `init`-контейнер.
9. Написать OPA-политику, проверяющую Trivy JSON-отчет: `deny[msg] { input.Results[].Vulnerabilities[].Severity == "CRITICAL" }`.
10. Построить многокластерную архитектуру: Hub-кластер с ArgoCD деплоит в Spoke-кластер. Kyverno на Spoke блокирует под, если он не из `signed image`.
11. Trivy в режиме `--vuln-type library` для анализа зависимостей Java-приложения (`pom.xml`). Обнаружить уязвимость Log4Shell (CVE-2021-44228).
12. Разработать Kubernetes Operator (на `Kopf/kopf`), который автоматически создает NetworkPolicy для каждого пода в `restricted-namespace` (`default-deny + explicit allow`).

13. Настроить непрерывный мониторинг CVE: Prometheus-экспортер парсит ленту NVD/CISA KEV и алертит при новых критических уязвимостях.

14. Реализовать проверку контейнеров на неизменяемость (immutable): readOnlyRootFilesystem: true enforced через Kyverno, контейнер работает корректно (tmpfs для временных данных).

15. Форк и доработка OPA Gatekeeper Constraint Template: проверка, что все образы в поде используют digest, а не тег (image@sha256:...).

5. Контрольные вопросы

1. В чем различие между Pod Security Admission (PSA) и PodSecurityPolicy (PSP)? Почему PSA считается более предпочтительным механизмом?

2. Как работает механизм seccompProfile: RuntimeDefault и какие системные вызовы он блокирует по умолчанию?

3. Объясните концепцию Distroleless-образов. Почему их использование снижает количество уязвимостей и уменьшает поверхность атаки?

4. Что такое SBOM и как его использование в связке с Trivy и Cosign помогает в обеспечении безопасности цепочки поставок ПО (Supply Chain Security)?

5. Опишите сценарий обхода restricted-политики при использовании init-контейнеров. Как Kyverno/OPA Gatekeeper могут предотвратить такие обходы?

6. Требования к отчету

Отчет в формате PDF. Обязательные разделы:

1. Титульный лист.
2. Схема безопасности образов и кластера: диаграмма потока (Build → Trivy Scan → Registry → PSA/Kyverno → Runtime).
3. Результаты сканирования:
 - Таблица с количеством уязвимостей по уровням (CRITICAL / HIGH / MEDIUM / LOW) для каждого просканированного образа.
 - Скриншот самого опасного CVE (описание, CVSS score, исправляющая версия).
 - JSON/HTML-отчет Trivy (полный) в приложении.
4. Демонстрация Pod Security Admission:
 - Листинг манифеста привилегированного пода и скриншот ошибки его создания.
 - Листинг исправленного манифеста и скриншот успешного запуска.
 - Вывод команды `kubectl describe ns secure-workloads` с метками PSA.
5. Интеграция в CI/CD (для среднего/повышенного уровня):
 - Фрагмент `.gitlab-ci.yml` / `.github/workflows/` с Quality Gate на основе Trivy.
6. Заключение: оценка влияния restricted-политик на процесс разработки и безопасность контейнерных сред.

7. Критерии оценивания

7.1. Базовый уровень («удовлетворительно»)

- Собран минимум один Docker-образ (уязвимый).
- Выполнено сканирование Trivy, предоставлен отчет (текстовый вывод или скриншот).
- Создан namespace с PSA restricted.
- Продемонстрирована блокировка привилегированного пода (скриншот ошибки).
- Предоставлен исправленный манифест, успешно запущенный в restricted-namespacе.
- Снижение: не указаны уровни PSA (enforce/warn/audit) — минус 1 балл.

7.2. Средний уровень («хорошо»)

- Выполнены все критерии базового уровня.
- Trivy интегрирован в CI/CD-пайплайн (скриншот пайплайна с шагом Trivy).
- Реализован Quality Gate: пайплайн падает при наличии CRITICAL-уязвимостей.
- Использованы Kyverno/OPA Gatekeeper для дополнительных политик (не только PSA).
- Предоставлен SBOM (CycloneDX) для образа.
- Снижение: Quality Gate не блокирует пайплайн при CRITICAL — минус 2 балла.

7.3. Повышенный уровень («отлично»)

- Выполнены все критерии среднего уровня.
- Развернут приватный реестр (Harbor) с политиками безопасности образов.
- Внедрена подпись образов (Cosign) и проверка подписи при деплое.
- Реализованы e2e-тесты безопасности (автоматическая проверка блокировки привилегированного пода).
- Настроена система рантайм-детектирования аномалий (Falco/Tetragon).
- Снижение: e2e-тесты не покрывают сценарий блокировки — минус 1 балл.

7.4. Общие штрафы

- Образы с CRITICAL-уязвимостями целенаправленно запущены в production-namespacе — минус 2 балла.
- PSA-политики ослаблены до privileged для «удобства» — работа не принимается.
- Отсутствует очистка ресурсов кластера после выполнения — минус 1 балл.

Лабораторная работа №5. Статический анализ конфигураций IaC (Checkov / tfsec)

1. Цель

Освоить методологию «Shift Left for IaC» — внедрение автоматизированной проверки безопасности Terraform-конфигураций на ранних этапах разработки, до применения в облаке. Научиться выявлять нарушения стандартов CIS, PCI DSS и лучших практик с помощью статических анализаторов Checkov и tfsec, а также интегрировать их в pre-commit хуки и CI/CD-пайплайны.

2. Задачи

1. Установить и настроить инструменты статического анализа Checkov и/или tfsec на локальной рабочей станции.
2. Выполнить прогон сканера по существующей Terraform-конфигурации (из лабораторной работы №1), зафиксировать все найденные нарушения.
3. Проанализировать результаты сканирования: классифицировать нарушения по уровням критичности (CRITICAL, HIGH, MEDIUM, LOW), определить их соответствие конкретным пунктам CIS-бенчмарка или PCI DSS.
4. Провести рефакторинг Terraform-кода, устранив все критические и высокие нарушения, добиться «чистого» прогона (passed).
5. Внедрить Checkov как pre-commit хук (через pre-commit framework) и продемонстрировать блокировку коммита с небезопасным кодом.

3. Теоретическое обоснование

Парадигма «Shift Left» в контексте IaC предполагает перенос проверок безопасности как можно ближе к моменту написания кода разработчиком, а не на этап post-deployment аудита. Каждая ошибка в Terraform-конфигурации, обнаруженная на стадии написания кода, в десятки раз дешевле исправления в production-окружении.

Инструменты статического анализа IaC:

- Checkov (Palo Alto / Bridgecrew) — мультиплатформенный сканер с открытым исходным кодом. Анализирует Terraform, CloudFormation, Kubernetes, Helm, Dockerfile, GitHub Actions. Проверяет конфигурации на соответствие более чем 750 встроенным политикам, покрывающим CIS, PCI DSS, SOC 2, HIPAA, NIST. Использует графовое представление ресурсов для cross-resource анализа (например, проверка, что бакет защищен политикой, а не только ACL).
- tfsec (Aqua Security) — легковесный анализатор, специализирующийся исключительно на Terraform. Работает быстрее Checkov, имеет минимальные зависимости. Поддерживает кастомные политики на Rego (язык OPA). Удобен для CI/CD из-за нативного SARIF-вывода и детерминированной CLI.

Ключевые категории проверок (CIS-бенчмарки):

- Networking: Security Groups не должны содержать правил с 0.0.0.0/0 для портов 22, 3389, 5432 и других чувствительных сервисов (CIS AWS 4.1, 4.2).
- Storage: Object Storage Buckets не должны быть публично доступны (acl != "public-read", политика не разрешает Principal: "*") (CIS AWS 2.1).
- Compute: Инстансы не должны иметь публичные IP в частных подсетях (associate_public_ip_address = false), должны использоваться IMDSv2, EBS-диски должны быть зашифрованы.
- IAM: Пароли не должны быть жестко закодированы, доступ к ресурсам должен осуществляться через роли, минимальные права.

Связь с компетенцией ПК-8:

- ИД-5пк-8 (целевой индикатор данной ЛР): *Сканирует конфигурации инфраструктуры как кода (IaC) на соответствие политикам безопасности и лучшим*

практикам до их применения. Checkov и tfsec — прямые инструменты для реализации этого индикатора.

- ИД-6пк-8: Интеграция сканера в CI/CD-пайплайн как Quality Gate, блокирующего деплой при нарушениях.

Методология устранения нарушений:

1. Идентификация: найти все Failed-проверки.
2. Классификация: отсортировать по Severity (CRITICAL → HIGH → MEDIUM → LOW).
3. Анализ: понять корневую причину (неправильный resource attribute, отсутствующий блок, неверная архитектура).
4. Исправление: модифицировать Terraform-код.
5. Верификация: повторный прогон сканера.
6. Суппрессия (с осторожностью): если нарушение ложно-положительное или осознанно принятый риск, добавить skip-комментарий с обоснованием (например, # checkov:skip=CKV_AWS_123:Open SSH only for bastion in DMZ).

4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

Все варианты: запуск Checkov на Terraform-конфигурации из ЛР №1 (или предоставленной преподавателем). Требуется предоставить скриншот Failed-проверок и скриншот исправленного прогона с Passed.

1. Исправить все нарушения, связанные с Security Group (CKV_YC_1, CKV_AWS_23 и аналоги).
2. Исправить нарушения, связанные с Object Storage Bucket (публичный доступ).
3. Исправить нарушения, связанные с незашифрованными дисками Compute Instance.
4. Исправить нарушения, связанные с жестко закодированными секретами (CKV_SECRET_*). Вынести в переменные с sensitive = true.
5. Исправить нарушения по отсутствию тегов (CKV_YC_10, CKV_AWS_126) — добавить обязательные теги Environment, Owner.
6. Устранить нарушения IMDSv2 (AWS) — использовать metadata_options { http_tokens = "required" }.
7. Исправить нарушения, связанные с публичным NAT-шлюзом в неподходящей зоне.
8. Убрать 0.0.0.0/0 из всех egress-правил Security Group (кроме явно разрешенных политикой).
9. Запустить tfsec вместо Checkov и сравнить результаты. Предоставить оба отчета.
10. Настроить Checkov с флагом --skip-check для легитимного нарушения (например, тестовый стенд). Документировать причину.
11. Проверить не только Terraform, но и Kubernetes-манифесты из ЛР №4 в одной директории (--framework terraform,kubernetes).

12. Использовать checkov --output sarif и просмотреть отчет в VSCode SARIF Viewer.
13. Запустить Checkov в Docker-контейнере, смонтировать директорию с кодом.
14. Настроить --external-checks-dir с одним кастомным чеком (пример из документации Checkov).
15. Вывести только CRITICAL и HIGH нарушения из Checkov-отчета (jq-запрос или флаг --check).

4.2. Средний уровень (15 вариантов)

Требуется pre-commit интеграция и написание кастомной политики.

1. Настроить pre-commit хук с checkov + terraform_fmt + terraform_validate. Продемонстрировать блокировку коммита.
2. Написать кастомную политику Checkov на Python: проверка, что имя бакета не содержит слово public.
3. Написать кастомную политику Checkov на YAML/JSON (проверка наличия обязательного тега CostCenter).
4. Интегрировать Checkov в GitHub Actions. Пайплайн создает комментарий в PR с результатами сканирования.
5. Настроить Checkov Baseline: зафиксировать текущее состояние как baseline, детектировать новые нарушения.
6. Использовать checkov --bc-api-key для интеграции с Bridgecrew Cloud (бесплатный tier), показать Dashboard с историей сканирований.
7. Написать скрипт на Python, который парсит checkov --output json и формирует сводную таблицу по Severity.
8. Интеграция с Atlassian Bitbucket Pipelines: Checkov сканирует на каждом push, артефакты сохраняются.
9. Написать Rego-политику для tfsec, проверяющую наличие prevent_destroy lifecycle у production-бакетов.
10. Настроить multi-framework сканирование: Terraform + Dockerfile + Kubernetes в одной команде.
11. Создать Makefile/Justfile цель make security-check, запускающую checkov, tfsec и tflint.
12. Реализовать автоматическую генерацию .checkov.yaml конфига с исключениями из комментариев в коде.
13. Сравнительный анализ: Checkov vs tfsec vs Terrascan на одном наборе конфигураций. Таблица в отчете.
14. Внедрить Checkov в Jenkins Pipeline с публикацией HTML-отчета.
15. Написать тест (pytest), проверяющий, что Checkov-отчет не содержит Failed с severity HIGH и выше.

4.3. Повышенный уровень (15 вариантов)

Требуется разработка кастомных политик, ОРА-интеграция и Enterprise-практики.

1. Разработать библиотеку кастомных политик Checkov (3+ чеков на Python), покрывающих корпоративные требования.

2. Написать OPA (Rego) политику для Terrascan/tfsec, проверяющую, что все S3-бакеты используют KMS-ключ с ротацией.
3. Реализовать полный Compliance Pipeline: Terraform plan → OPA Conftest → Checkov → ручное approval → apply.
4. Интеграция Checkov с DefectDojo: автоматический импорт SARIF-отчетов и tracking уязвимостей.
5. Написать Sentinel-политику (для Terraform Cloud/Enterprise), дублирующую Checkov-чек CKV_AWS_23.
6. Создать кастомный GitHub Action, который запускает Checkov и автоматически создает Issue при CRITICAL-нарушениях.
7. Развернуть Checkov в режиме Serverless (AWS Lambda), запускаемой при push в CodeCommit.
8. Внедрить мониторинг дрейфа политик: сравнение текущего Checkov-скана с эталонным baseline, алерт в PagerDuty.
9. Разработать оператор Kubernetes, который запускает Checkov на ConfigMap'ы с Terraform-конфигами и валидирует их.
10. Интеграция с SOAR (Shuffle/n8n): при обнаружении CRITICAL нарушения автоматически создается инцидент.
11. Написать кастомный провайдер для Checkov, проверяющий специфичные ресурсы Yandex Cloud, не покрытые базовыми чеками.
12. Реализовать A/B тестирование политик: новый чек запускается в --soft-fail режиме месяц, затем переводится в --hard-fail.
13. Разработать веб-дашборд (Streamlit/Grafana) для визуализации Checkov-отчетов в динамике по времени.
14. Интеграция с Argo Workflows: шаг Checkov перед шагом Terraform apply в CI/CD.
15. Форк Checkov: добавить кастомный output-формат, совместимый с корпоративной системой (например, XML с определённой схемой).

5. Контрольные вопросы

1. В чем разница между --soft-fail и --hard-fail в Checkov? Как правильно использовать эти режимы в CI/CD-пайплайне?
2. Какие существуют подходы к управлению ложноположительными срабатываниями (false positives) в статических анализаторах IaC? Назовите минимум три.
3. Объясните концепцию "Resources Graph" (граф ресурсов) в Checkov. Почему для проверки безопасности Object Storage недостаточно анализа отдельного ресурса и требуется анализ связанных ресурсов?
4. Чем кастомные политики на Python (Checkov) отличаются от политик на Rego (OPA/tfsec) с точки зрения гибкости, производительности и порога входа?
5. Как соотносятся между собой этапы terraform validate, tfint, checkov/tfsec и terraform plan в контексте DevSecOps-пайплайна? В каком порядке они должны выполняться и почему?

6. Требования к отчету

Отчет в формате PDF. Обязательные разделы:

1. Титульный лист.
2. Схема процесса Shift Left для IaC: диаграмма (draw.io), отображающая этапы: Developer → IDE/pre-commit → Git Push → CI (Checkov) → Artifact Registry → Deploy.
3. Результаты первичного сканирования:
 - Полный вывод Checkov с таблицей Failed-проверок (скриншот).
 - Классификация нарушений по Severity и CIS-пунктам (таблица).
4. Исправленный код:
 - Diff (разница) между исходной и исправленной конфигурацией.
 - Листинг финального main.tf с комментариями о внесенных изменениях.
 - Скриншот повторного прогона Checkov с результатом Passed checks: N, Failed checks: 0.
5. Pre-commit интеграция (для среднего/повышенного уровня):
 - Листинг .pre-commit-config.yaml.
 - Скриншот блокировки коммита с уязвимостями.
6. CI/CD интеграция (для среднего/повышенного уровня):
 - Фрагмент пайплайна (.gitlab-ci.yml / .github/workflows/).
 - Скриншот упавшего пайплайна при нарушениях.
7. Заключение: оценка эффективности методологии Shift Left для предотвращения уязвимостей в облачной инфраструктуре.

7. Критерии оценивания

7.1. Базовый уровень («удовлетворительно»)

- Checkov установлен и успешно запущен на конфигурации из ЛР №1 (или предоставленной).
- Зафиксирован первичный отчет с Failed-проверками (скриншот).
- Устранены ВСЕ нарушения уровня CRITICAL и HIGH (скриншот повторного прогона).
- Код выложен в репозиторий (ссылка в отчете).
- В отчете присутствует классификация нарушений по Severity.
- Снижение: остались CRITICAL-нарушения — минус 2 балла, работа требует доработки.

7.2. Средний уровень («хорошо»)

- Выполнены все критерии базового уровня.
- Внедрен pre-commit хук с Checkov (доказательство блокировки коммита).
- Продемонстрирована работа кастомной политики (Python или YAML).
- Настроен .checkov.yaml с документированными skip-правилами для легитимных исключений.
- В отчете присутствует SARIF-отчет и его визуализация.

- Снижение: pre-commit хук не блокирует коммит с CRITICAL — минус 2 балла.

7.3. Повышенный уровень («отлично»)

- Выполнены все критерии среднего уровня.
- Разработана библиотека из минимум 3 кастомных политик.
- Настроен CI/CD с автоматическим Quality Gate (блокировка merge при HIGH+).
- Внедрена OPA-политика (Rego) в дополнение к Checkov.
- Продемонстрирована интеграция с внешней системой (DefectDojo, Bridgecrew Cloud, SOAR).
- Снижение: отсутствует OPA-политика — минус 1 балл (оценка «хорошо» при выполнении остальных критериев).

7.4. Общие штрафы

- Skip-комментарии используются для маскировки реальных нарушений без обоснования — минус 2 балла, возврат на доработку.
- В финальной конфигурации обнаружены незамеченные Checkov-ом проблемы (ручная проверка преподавателем) — минус 1 балл.
- Отсутствует diff между исходным и исправленным кодом — минус 1 балл.

Лабораторная работа №6. Написание политик безопасности на Rego/OPA

1. Цель

Освоить методологию «Политика как Код» (Policy as Code) с использованием Open Policy Agent (OPA) и языка Rego. Научиться разрабатывать, тестировать и интегрировать декларативные политики безопасности, проверяющие инфраструктурные конфигурации на соответствие корпоративным стандартам и регуляторным требованиям до их применения.

2. Задачи

1. Установить и настроить OPA в локальном окружении, освоить базовый синтаксис языка Rego.
2. Разработать набор политик, реализующих типовые корпоративные требования к облачной инфраструктуре (запрет публичных бакетов, обязательные теги, ограничение типов инстансов).
3. Выполнить модульное тестирование политик через opa test и отладить их в Rego Playground.
4. Интегрировать OPA с Terraform: экспортировать terraform plan в JSON, выполнить проверку плана через OPA.
5. Реализовать сценарий блокировки несоответствующей конфигурации и документировать процесс принятия решений на основе оценки политик.

3. Теоретическое обоснование

Open Policy Agent (OPA) — open-source движок для унифицированного применения политик в облачной инфраструктуре. OPA отделяет логику принятия решений от бизнес-логики приложения, позволяя централизованно управлять политиками без изменения кода сервисов. Проект является грейдуированным в CNCF и широко используется в Kubernetes (в составе Gatekeeper), Terraform, Envoy, Kafka.

Язык Rego — декларативный язык запросов, вдохновленный Datalog и предназначенный для выражения политик над структурированными данными (JSON, YAML). В отличие от императивных языков, Rego описывает *ожидаемое состояние*, а не алгоритм проверки. Ключевая концепция: политика — это набор правил, результатом которых является множество (set) нарушений. Если нарушений нет (пустое множество) — ресурс соответствует политике.

Основные конструкции Rego:

- Правила (Rules): Именованные выражения, вычисляющие значения. Правило `deny[msg]` — идиоматический способ вернуть множество нарушений. Правило `allow` возвращает `true/false` для разрешительных политик.
- Запросы (Queries): Логические выражения, связывающие переменные с данными. `data.aws.instances[instance].type == "t2.micro"` свяжет переменную `instance` со всеми инстансами типа `t2.micro`.
- Итерации: Выражение `some i; input.resources[i]` итерирует по массиву, связывая переменную `i` с индексом.
- Комprehension (генераторы множеств): `{ msg | instance := input.instances[_]; instance.type == "t2.nano" }` строит множество всех сообщений для инстансов с типом `t2.nano`.
- Импорт данных: `import data.terraform` делает данные из внешнего модуля доступными.
- Встроенные функции: `contains()`, `regex.match()`, `startswith()`, `count()`, `object.get()`.

Интеграция OPA с Terraform: Terraform не имеет встроенной поддержки OPA, но интеграция осуществляется через JSON-представление плана:

1. `terraform plan -out=tfplan` — создает бинарный файл плана.
2. `terraform show -json tfplan > tfplan.json` — экспортирует план в машиночитаемый JSON.
3. `opa eval --input tfplan.json --data policies/ "data.terraform.analysis.authz"` — оценивает политики, передавая план как `input`.

Структура Terraform Plan JSON содержит массив `resource_changes` — все изменения ресурсов, где каждое изменение имеет поля `address`, `type`, `change.actions` (`create/update/delete`), `change.after` (желаемое состояние после применения). Именно `change.after` и является объектом проверки политикой.

Связь с компетенцией ПК-8:

- ИД-5пк-8: Сканирование IaC на соответствие политикам безопасности. OPA + Rego — инструмент реализации кастомных политик, дополняющих Checkov/tfsec.
- ИД-8пк-8 (целевой индикатор): *Обеспечение непрерывного соответствия через автоматизированный сбор доказательств*. OPA-политики, интегрированные в CI/CD, реализуют Compliance as Code, генерируя доказательства проверки (`policy evaluation results`) на каждый коммит.

Преимущества OPA перед статическими анализаторами:

- Единый язык политик для всех слоев стека (инфраструктура, Kubernetes, приложения, API-шлюзы).
- Возможность кастомизации без изменения исходного кода инструмента.
- Модульное тестирование политик через `opa test`.
- Интеграция с системами принятия решений (Envoy External Authorization, Kubernetes Admission Webhooks).

4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

Все варианты: написать политики Rego, протестировать через `opa test`, проверить на одном тестовом Terraform Plan JSON.

1. Политика `deny_untagged_resource`: все ресурсы должны иметь тег `Owner`. Написать тесты.
2. Политика `deny_small_instance`: запретить инстансы с менее чем 2 vCPUs (слабое звено безопасности без мониторинга).
3. Политика `deny_port_22_all`: Security Group не должна содержать правила SSH с 0.0.0.0/0.
4. Политика `deny_latest_ami`: запретить использование AMI с тегом `latest` (требовать конкретную версию).
5. Политика `deny_unencrypted_volume`: все диски должны быть зашифрованы.
6. Политика `deny_missing_description`: все Security Group rules должны иметь поле `description`.
7. Политика `deny_default_vpc`: запретить создание ресурсов в default VPC.
8. Политика `require_https_only`: бакеты должны иметь политику, разрешающую только HTTPS-запросы.
9. Политика `deny_single_az`: запретить развертывание инстансов только в одной зоне доступности (проверка через подсети).
10. Политика `require_resource_limits`: все `yandex_compute_instance` должны иметь ограничение `core_fraction` не более 20.
11. Политика `deny_deletion_protection_missing`: критичные ресурсы (БД, бакеты) должны иметь `deletion_protection = true`.
12. Политика `require_logging_enabled`: Object Storage бакеты должны иметь включенное логирование.
13. Политика `deny_nat_in_public_subnet`: запретить NAT-шлюзы в подсетях с прямым интернет-доступом.
14. Политика `require_backup_policy`: все БД-инстансы должны иметь привязанную backup policy.
15. Политика `deny_iam_user_with_password`: запретить создание IAM пользователей с паролем (только access keys/роли).

4.2. Средний уровень (15 вариантов)

Требуется написание комплексных политик с перекрестным анализом ресурсов и расширенным тестированием.

1. Cross-resource политика: Security Group, разрешающая доступ на порт 5432, должна использоваться ТОЛЬКО инстансами с тегом `Role: database`.
2. Политика с динамическими данными: разрешенные типы инстансов загружаются из внешнего JSON-файла (`data.allowed_types`).
3. Написать правило с `regex.match`: имена бакетов должны соответствовать паттерну `^[a-z0-9-]{3,63}$`.
4. Политика "запрет изменений": ресурсы с тегом `Immutable: true` не должны присутствовать в плане с действием `update` или `delete`.
5. Иерархическая политика: если тег `Environment: prod`, то автоматически требовать теги `BackupSchedule` и `DRPlan`.

6. Политика с подсчетом ресурсов: в одном VPC не может быть более 3 подсетей с публичным доступом.
7. Политика с временными окнами: запрещено создание ресурсов в выходные дни (проверка `time.clock()` в тестах).
8. Политика Network ACL: правила не должны иметь номера ниже 100 без явного `approval`-тега.
9. Интеграция с `opa eval --partial`: показать сгенерированный SQL-подобный датасет нарушений.
10. Политика с оценкой стоимости: запретить инстансы, чья стоимость ($\text{cores} \times 10 + \text{memory} \times 5$) превышает 200.
11. Написать мета-политику: в директории `policies/` должно быть не менее 3 `.rego` файлов (рекурсивная проверка самого OPA).
12. Политика сравнения двух Terraform-воркспейсов: запрещено, чтобы ресурсы в `dev` имели больший размер, чем в `prod`.
13. Политика с `some ... in`: все подсети должны иметь уникальные CIDR в рамках VPC (перебор всех пар подсетей).
14. Написать OPA Bundle и продемонстрировать его применение через `opa run --server --bundle`.
15. Политика "allowed regions": ресурсы разрешено создавать только в регионах из списка ["ru-central1", "eu-west-1"].

4.3. Повышенный уровень (15 вариантов)

Требуется enterprise-интеграция, OPA-сервер и расширенные сценарии.

1. Развернуть OPA как сервер (REST API) с загруженными политиками. Написать Python-клиент, отправляющий Terraform Plan JSON и получающий решение.
2. Написать OPA-политику с использованием `http.send()` для запроса к Regula (проверка CIS через внешний сервис) в реальном времени.
3. Реализовать Decision Logging: OPA-сервер логирует все решения в Elasticsearch, визуализация в Kibana (Grafana).
4. Разработать систему с JWT-аутентификацией: разные политики для разных команд на основе `claims` в токене (`bundles per team`).
5. Написать OPA-политику для Kubernetes Admission Control (через Gatekeeper ConstraintTemplate), запрещающую поды без `securityContext`.
6. Интеграция OPA с Envoy External Authorization: политика, проверяющая заголовок `X-API-Key` и разрешающая доступ только к определенным `endpoints`.
7. Разработать CI-оркестратор: Python-скрипт, который на основе OPA-решения автоматически создает Jira-тикет при CRITICAL-нарушениях.
8. Написать политику с `time.add_date()`: SSL-сертификаты (ACM) должны обновляться за 30 дней до истечения. Проверка через дату окончания.
9. Реализовать A/B Testing политик: OPA-сервер возвращает разные решения на основе хэша от `request ID` (канареечное развертывание политик).
10. Разработать кастомный Terraform Provider, который вызывает OPA перед `apply` и отменяет операцию при `allow == false`.

11. Интеграция OPA с HashiCorp Sentinel: политика Sentinel вызывает OPA через HTTP API для принятия решения (двойная валидация).
12. Написать Rego-политику для анализа CloudTrail/Audit Trail логов: обнаружение создания ресурсов без тегов постфактум.
13. Реализовать самообучающуюся политику (с использованием OPA http.send): запрос к API инвентаризации для получения списка разрешенных AMI.
14. Разработать OPA Wasm-модуль (компиляция Rego в WebAssembly) и выполнить его в браузере для клиентской валидации Terraform-конфигураций.
15. Создать full-cycle Compliance Pipeline: pre-commit (opa test) → CI (opa eval) → Admission Control (OPA Server) → Runtime Audit (Post-deployment OPA check).

5. Контрольные вопросы

1. Объясните ключевые различия между императивным и декларативным подходом к написанию политик. Почему Rego является декларативным языком?
2. Как работает механизм унификации (unification) в Rego? Приведите пример, где переменная связывается с несколькими значениями.
3. Что такое partial evaluation в OPA и для каких сценариев он предназначен? Как он связан с производительностью?
4. Какие существуют стратегии версионирования OPA-политик в CI/CD? Как обеспечить обратную совместимость политик при обновлении?
5. В чем различие между подходами OPA-as-Library (Wasm) и OPA-as-Server (REST API) для принятия решений? Приведите примеры сценариев использования каждого.

6. Требования к отчету

Отчет в формате PDF. Обязательные разделы:

1. Титульный лист.
2. Архитектура Policy as Code: диаграмма (draw.io) потока проверки (Terraform Plan → JSON → OPA → Decision → CI/CD Gate).
3. Написанные политики:
 - Полный листинг всех .rego файлов с построчными комментариями.
 - Обоснование каждой политики (ссылка на CIS или корпоративный стандарт).
4. Тестирование политик:
 - Листинг тестовых файлов.
 - Вывод opa test -v с результатами (скриншот).
5. Результаты интеграции с Terraform:
 - Фрагмент Terraform-конфигурации с нарушениями.
 - Команда экспорта плана и запуска OPA.
 - Вывод OPA-оценки (JSON с violations).
 - Скриншот блокировки в CI/CD (если применимо).
6. Рефакторинг до соответствия:

- Исправленная Terraform-конфигурация.
- Повторный вывод OPA с "allow": true.

7. Заключение: анализ эффективности Policy as Code для предотвращения дрейфа конфигураций.

7. Критерии оценивания

7.1. Базовый уровень («удовлетворительно»)

- Написана минимум 1 политика Rego, проверяющая одно корпоративное требование.
- Политика покрыта unit-тестами (минимум 2 теста: compliant и non-compliant сценарии).
- Продемонстрирована интеграция с Terraform Plan JSON (ручной запуск).
- В отчете присутствует скриншот OPA-оценки с violations.
- Снижение: политика не покрывает edge-кейсы (например, отсутствующие поля) — минус 1 балл.

7.2. Средний уровень («хорошо»)

- Выполнены все критерии базового уровня.
- Написано минимум 3 политики, покрывающих разные аспекты (теги, безопасность, ресурсы).
- Реализован cross-resource анализ (минимум 1 политика с анализом связей).
- Настроен CI/CD с автоматическим прогоном OPA (Makefile + GitLab CI/GitHub Actions).
- Unit-тесты содержат parametrized test cases (табличные тесты).
- Снижение: отсутствует cross-resource анализ — минус 2 балла.

7.3. Повышенный уровень («отлично»)

- Выполнены все критерии среднего уровня.
- Развернут OPA Server с Decision Logging (Elasticsearch + визуализация).
- Реализована внешняя интеграция (http.send()) в политике.
- Политики компилированы в Wasm и выполнены в альтернативном рантайме.
- Написана политика для Kubernetes Admission Control или Envoy External Auth (вторая система).
- Выполнен аудит реального Terraform-плана и сгенерирован Compliance Report.
- Снижение: отсутствует вторая система интеграции — минус 1 балл (оценка «хорошо»).

7.4. Общие штрафы

- Политика содержит правило allow = true без проверок (пропускает всё) — работа не принимается.
- Отсутствуют unit-тесты — минус 2 балла на любом уровне.

- Результаты ОРА-проверки не сохранены в артефактах CI/CD — минус 1 балл

Лабораторная работа №7. Compliance как Code с Cloud Custodian

1. Цель

Освоить практику «Compliance as Code» для непрерывного мониторинга и автоматического исправления нарушений политик безопасности в работающей облачной инфраструктуре. Научиться проектировать, тестировать и применять политики Cloud Custodian, реализующие замкнутый цикл управления соответствием: обнаружение → оповещение → коррекция → аудит.

2. Задачи

1. Установить и настроить Cloud Custodian для работы с целевым облачным провайдером (Yandex Cloud / AWS).
2. Разработать политику (policy), выявляющую облачные ресурсы, не соответствующие корпоративным требованиям безопасности.

3. Выполнить прогон политики в режиме dry-run (только инвентаризация нарушений) и проанализировать сгенерированный отчет.
4. Применить политику в активном режиме с автоматической коррекцией (исправление или остановка несоответствующего ресурса).
5. Настроить генерацию структурированного отчета в JSON и продемонстрировать его использование для аудита и сбора доказательств соответствия.

3. Теоретическое обоснование

Проблема дрейфа конфигураций (Configuration Drift): Даже при идеально написанной IaC-конфигурации и строгих CI/CD Quality Gates, работающая облачная среда со временем деградирует: администраторы вносят ручные изменения через консоль, автоматическое масштабирование создает ресурсы с дефолтными настройками, устаревшие ресурсы не удаляются. Это приводит к расхождению между желаемым состоянием (IaC) и фактическим состоянием (Runtime).

Cloud Custodian — open-source rules engine (CNCF Sandbox), реализующий концепцию «Compliance as Code». В отличие от статических анализаторов IaC (Checkov, tfsec), Custodian работает с *running resources* через API облачного провайдера, обеспечивая:

- Инвентаризацию — обнаружение всех ресурсов, соответствующих заданным критериям;
- Фильтрацию — отбор ресурсов с нарушениями политик;
- Коррекцию — автоматические действия по исправлению (stop, terminate, tag, encrypt, notify);
- Аудит — структурированные отчеты в JSON/CSV для сбора доказательств соответствия.

Ключевые концепции Cloud Custodian:

- Policy — основной артефакт, YAML-документ, описывающий: resource (тип ресурса), filters (критерии отбора нарушителей), actions (действия по исправлению), mode (способ запуска).
- Filters — предикаты, сужающие множество проверяемых ресурсов. Поддерживают композицию (and, or, not), value matching (key: value), regex, age-based фильтры, tag-based фильтры.
- Actions — идемпотентные операции над ресурсами. Custodian гарантирует безопасность: если ресурс удален между dry-run и run, действие не выполнится.
- Modes — способы запуска:
 - pull — ручной или cron-запуск, Custodian опрашивает облачное API;
 - phd (Policy Hub Dispatch) — событийно-ориентированный запуск через EventBridge/Audit Trails;
 - periodic — автоматический запуск по расписанию (Lambda + CloudWatch Event).

Архитектура работы (Pull-режим):

text

Cloud Custodian CLI → API Calls (List/Describe) → Filter Resources → Match Policy → Execute Actions → Generate Report

Связь с компетенцией ПК-8:

- ИД-7пк-8: *Мониторит состояние безопасности контейнеров и инфраструктуры в рантайме, выявляя инциденты.* Cloud Custodian выполняет именно эту функцию, обнаруживая «неправильные» ресурсы в реальном времени.

- ИД-8пк-8 (целевой индикатор): *Обеспечивает непрерывное соответствие (continuous compliance) через автоматизированный сбор доказательств и генерацию отчетов.* Custodian-отчеты в JSON-формате являются юридически значимыми артефактами для аудиторов.

Принцип безопасной коррекции:
Перед применением активных действий (stop, terminate, delete) ОБЯЗАТЕЛЬНО выполнять dry-run. Custodian в dry-run-режиме показывает, какие ресурсы *были бы* затронуты, без выполнения действий. Это предотвращает случайное удаление продуктивных ресурсов.

4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

Провайдер: Yandex Cloud или AWS. Требуется предоставить YAML-политику, скриншот dry-run и доказательство действия.

1. Остановка всех инстансов с тегом Environment: dev в нерабочее время (ручной запуск, фильтрация по тегу).
2. Удаление неиспользуемых дисков (статус UNATTACHED, старше 7 дней).
3. Обнаружение Security Group с правилом 0.0.0.0/0 на порт 22. Только инвентаризация (dry-run), без действий.
4. Поиск Object Storage бакетов с публичным ACL. Действие: отправка email-уведомления.
5. Остановка инстансов без тега Backup (value absent).
6. Обнаружение инстансов с публичным IP. Действие: добавление тега PublicExposed: true.
7. Поиск и удаление снапшотов старше 30 дней.
8. Обнаружение неиспользуемых статических публичных IP адресов.
9. Поиск инстансов, не соответствующих approved-типам (например, разрешены только standard-v1, standard-v2).
10. Обнаружение подсетей без привязанной Route Table (Yandex Cloud).
11. Поиск сервисных аккаунтов с правами admin или editor. Действие: маркировка тегом Risk: High.
12. Обнаружение Load Balancer'ов без настроенного health check.
13. Поиск инстансов с core_fraction > 20 (экономия бюджета). Действие: отправка уведомления.
14. Обнаружение KMS-ключей без настроенной ротации.
15. Поиск бакетов без versioning. Действие: Dry-run + отчет.

4.2. Средний уровень (15 вариантов)

Требуется написание комплексных политик с несколькими фильтрами и действиями, режим periodic.

1. Политика с двумя действиями: stop + tag. Остановить и пометить инстансы без тега Environment.

2. Политика с композитными фильтрами (and/or). Остановить экземпляры, которые одновременно: без тега Compliant, в зоне ru-central1-a, старше 24 часов.
3. Политика с фильтром по возрасту (age): удалить снимки старше 14 дней, но только если у них нет тега Retain: true.
4. Политика с value_type: resource_count: если количество экземпляров в зоне превышает 10, отправить алерт.
5. Создание политики с mode: periodic (Lambda + EventBridge или аналог в Yandex Cloud Functions + Trigger).
6. Политика с действием webhook: при обнаружении нарушения отправлять POST-запрос в Slack/Teams с деталями.
7. Реализовать политику mark-for-op и marked-for-op: пометить ресурс на удаление через 7 дней, удалить по метке.
8. Политика, проверяющая шифрование дисков: обнаружение незашифрованных загрузочных дисков, действие — уведомление.
9. Cross-account политика (AWS Organizations): проверка всех аккаунтов в OU на наличие публичных S3.
10. Политика с on-error: continue: обработать частичный сбой API, задокументировать ошибки в отчете.
11. Политика с кастомным JMESPath-фильтром: обнаружение экземпляров с метаданными, содержащими SSH-ключи.
12. Настройка вывода в различные форматы: JSON + CSV + HTML-отчет (через шаблоны Jinja2).
13. Политика, проверяющая соответствие naming convention: имена экземпляров должны соответствовать regex `^[a-z]-[a-z]+\d{3}$`.
14. Интеграция с внешней CMDB: политика запрашивает список разрешенных владельцев через REST API (кастомный скрипт).
15. Политика "soft-delete": вместо удаления — остановка и создание снимка перед остановкой.

4.3. Повышенный уровень (15 вариантов)

Требуется кастомизация Cloud Custodian, интеграция с событийной архитектурой и автоматическое реагирование.

1. Разработка кастомного фильтра (Python-класс): проверка экземпляров через внешний vulnerability scanner API.
2. Реализация политики с mode: cloudtrail / mode: audit-trail: реагирование на событие создания ресурса в реальном времени.
3. Написание кастомного действия (action): интеграция с PagerDuty для создания инцидентов при CRITICAL-нарушениях.
4. Развертывание Cloud Custodian в Serverless-режиме (AWS Lambda / Yandex Cloud Functions) с автоматической коррекцией.
5. Создание сложного workflow: обнаружение несоответствующего экземпляра → остановка → снимок → уведомление → эскалация через 24 часа.
6. Интеграция с OPA: Custodian вызывает OPA для принятия решения о коррекции (кастомный action с HTTP-коллбэком).

7. Реализация "Compliance Score" — метрика процента соответствующих ресурсов, экспорт в Prometheus/CloudWatch.
8. Разработка кастомного provisioning-скрипта: Custodian не только удаляет, но и пересоздает ресурс с правильной конфигурацией (auto-remediation).
9. Интеграция с Jira Service Management: автоматическое создание тикетов при обнаружении нарушений, закрытие при устранении.
10. Построение Compliance Dashboard (Grafana): визуализация состояния соответствия на основе Custodian-отчетов.
11. Разработка ChatOps-интерфейса: запуск политик через Slack-команды с выводом результатов в тред.
12. Реализация Multi-Cloud политики: единая политика, проверяющая и AWS, и Yandex Cloud ресурсы (унифицированный отчет).
13. Написание системы автоматического approval: политика в dry-run, результат отправляется на approval в GitOps (Pull Request с планируемыми действиями).
14. Интеграция с системой Threat Intelligence: политика сравнивает публичные IP инстансов с фидами угроз, при совпадении — изоляция.
15. Реализация полного цикла Continuous Compliance: IaC (Terraform) → Pre-deploy Check (Checkov) → Deploy → Post-deploy Audit (Custodian) → Auto-remediation → Feedback Loop в IaC.

5. Контрольные вопросы

1. В чем принципиальное различие между статическими анализаторами IaC (Checkov, tfsec) и Cloud Custodian с точки зрения этапа жизненного цикла, на котором они применяются?
2. Объясните механизм работы mark-for-op и marked-for-op в Cloud Custodian. Почему такая двухэтапная стратегия безопаснее прямого удаления?
3. Какие режимы запуска (mode) поддерживает Cloud Custodian? В чем разница между pull, periodic и cloudtrail? Приведите примеры использования каждого.
4. Как Cloud Custodian обеспечивает идемпотентность действий? Что произойдет, если ресурс был удален вручную между dry-run и apply?
5. Какие метаданные собирает Cloud Custodian в процессе выполнения политики и как их можно использовать для автоматизированного сбора доказательств соответствия (compliance evidence)?

6. Требования к отчету

Отчет в формате PDF. Обязательные разделы:

1. Титульный лист.
2. Архитектура Compliance as Code: диаграмма (draw.io) цикла непрерывного соответствия (Resource Creation → Custodian Detection → Notification/Action → Compliance Report → Auditor Review).
3. Разработанные политики:
 - Полный листинг всех YAML-политик с построчными комментариями.
 - Обоснование каждой политики (ссылка на CIS/PCI DSS/корпоративный стандарт).

4. Результаты dry-run:
 - Скриншот команды `custodian run --dry-run` с выводом.
 - Содержимое `resources.json` (фрагмент), демонстрирующее обнаруженные нарушения.
 - Таблица обнаруженных ресурсов с указанием причины нарушения.
5. Результаты активного применения:
 - Скриншот команды `custodian run` с выводом выполненных действий.
 - Скриншот состояния ресурса в облачной консоли ДО и ПОСЛЕ применения политики.
6. Отчет для аудита:
 - Сгенерированный `compliance-report.json` (в приложении).
 - Сводная таблица: количество проверенных ресурсов, количество нарушений, выполненные действия.
7. Заключение: оценка эффективности автоматической коррекции нарушений и места Cloud Custodian в DevSecOps-конвейере.

7. Критерии оценивания

7.1. Базовый уровень («удовлетворительно»)

- Cloud Custodian установлен, успешно выполнен `custodian version`.
- Написана минимум 1 политика с фильтром и действием.
- Выполнен dry-run, предоставлен скриншот вывода.
- Политика НЕ содержит действия `terminate/delete` в первом прогоне (безопасность).
- В отчете присутствует `resources.json` с результатами проверки.
- Снижение: политика написана некорректно (синтаксическая ошибка YAML) — минус 2 балла, доработка.

7.2. Средний уровень («хорошо»)

- Выполнены все критерии базового уровня.
- Написано минимум 3 политики, покрывающих разные типы ресурсов (`compute`, `storage`, `network`).
- Реализована политика с `mark-for-op` и проверка меток `marked-for-op`.
- Настроен режим `periodic` или интеграция с CI/CD (Makefile/пайплайн).
- Сгенерирован сводный отчет (JSON + человекочитаемый формат).
- Снижение: отсутствует композитный фильтр (`and/or`) — минус 2 балла.

7.3. Повышенный уровень («отлично»)

- Выполнены все критерии среднего уровня.
- Разработан кастомный фильтр или действие на Python.
- Развернут Cloud Custodian в serverless-режиме (Lambda/Cloud Functions) с автоматическим запуском.

- Интегрирована система оповещения (Slack/PagerDuty/Jira).
- Построен Compliance Dashboard (Grafana) с историческими данными.
- Выполнен полный цикл: dry-run → approval → apply → verify → report.
- Снижение: отсутствует кастомный компонент — минус 1 балл (оценка «хорошо» при остальных критериях).

7.4. Общие штрафы

- Политика содержит action: terminate без предварительного dry-run и approval — минус 2 балла, инструктаж по безопасности.
- Удалены продуктивные ресурсы (даже тестовые, но без возможности восстановления) — минус 3 балла.
- Отсутствуют артефакты аудита (resources.json, метаданные) — минус 1 балл.

Лабораторная работа №8. Безопасность CI/CD Pipeline (GitLab CI / GitHub Actions)

1. Цель

Освоить практику построения безопасных CI/CD-пайплайнов для развертывания облачной инфраструктуры с использованием временных учетных данных. Научиться настраивать OIDC-федерацию между CI/CD-системой и облачным провайдером, исключая хранение долгоживущих статических ключей доступа в переменных окружения.

2. Задачи

1. Разработать структуру тестового проекта: исходный код приложения, Dockerfile, Terraform-конфигурация инфраструктуры.
2. Настроить OIDC-федерацию между GitLab CI (или GitHub Actions) и облачным провайдером (Yandex Cloud / AWS) для получения временных IAM-токенов.
3. Реализовать пайплайн с этапами: сборка образа → сканирование Trivy (Quality Gate) → публикация в Container Registry → деплой инфраструктуры через Terraform с OIDC-аутентификацией.
4. Продемонстрировать, что в настройках CI/CD отсутствуют статические секреты (ключи доступа), а аутентификация выполняется через механизм AssumeRoleWithWebIdentity (или аналог).
5. Верифицировать работу Quality Gate: пайплайн блокируется при обнаружении CRITICAL-уязвимостей в образе.

3. Теоретическое обоснование

Проблема статических ключей в CI/CD: Традиционный подход к аутентификации CI/CD-пайплайнов в облачных провайдерах предполагает создание сервисного аккаунта, генерацию статического ключа доступа (access key + secret key) и сохранение его в переменных окружения CI/CD (GitLab Variables, GitHub Secrets). Такой подход порождает критические риски:

- Ключи имеют неограниченный срок действия (или длительный, до года);
- Ключи часто хранятся в нескольких местах (CI/CD переменные, .env-файлы разработчиков, резервные копии);
- Компрометация репозитория (утечка .gitlab-ci.yml с echo \$SECRET) раскрывает ключи;
- Ротация ключей — ручной процесс, который редко выполняется вовремя;
- Ключи предоставляют постоянный доступ, а не временный, ограниченный временем выполнения пайплайна.

OIDC-федерация (OpenID Connect) решает эту проблему, заменяя статические ключи динамическим обменом токенов. Механизм работы:

1. CI/CD-система (GitLab, GitHub) выступает в роли OIDC-провайдера (Identity Provider, IdP).
2. При запуске пайплайна GitLab генерирует JWT-токен, содержащий claims (утверждения) о контексте запуска: sub (project_path:ref_type:ref), iss (URL GitLab-инстанса), aud (аудитория), project_id, environment.
3. Этот JWT передается в команду `aws sts assume-role-with-web-identity` или `aws iam create-token`, где облачный провайдер (Service Provider, SP) проверяет подпись JWT и его claims.

4. В случае успешной проверки облачный провайдер выпускает временные учетные данные (STS-токен), ограниченные по времени (обычно 1 час) и правам (IAM-роль, которую настроил администратор).

5. Эти временные учетные данные используются Terraform/CLI для выполнения операций. После завершения пайплайна токен автоматически истекает.

Схема OIDC-аутентификации:

text

GitLab CI Runner → OIDC Discovery → GitLab JWKS Endpoint

↓ JWT

Cloud IAM (OIDC IdP Trust) → Validate JWT signature + claims

↓ Temporary Token

Terraform/CLI → Cloud API (с временными правами)

Связь с компетенцией ПК-8:

- ИД-5пк-8: Сканирование IaC-конфигураций и образов на этапе CI/CD.
- ИД-6пк-8 (целевой индикатор): *Интегрирует инструменты безопасности в CI/CD-пайплайны, настраивая автоматические шлюзы качества (Quality Gates), блокирующие поставку при обнаружении критических уязвимостей.* Данная ЛР реализует полный DevSecOps-пайплайн с Quality Gate на основе Trivy.
- ИД-7пк-8: Мониторинг состояния безопасности (Custodian как post-deployment шаг).

Компоненты пайплайна:

1. Build Stage: Сборка Docker-образа, присвоение тега.
2. Security Scan Stage (Quality Gate): Сканирование образа Trivy. При обнаружении CRITICAL/HIGH — блокировка пайплайна.
3. Publish Stage: Публикация образа в Container Registry (Yandex Container Registry / AWS ECR).
4. Deploy Stage: Инициализация Terraform с OIDC-аутентификацией, terraform plan, terraform apply.
5. Post-Deploy Audit (опционально): Запуск Cloud Custodian для проверки развернутой инфраструктуры.

4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

Провайдер: Yandex Cloud или AWS. Требуется предоставить .gitlab-ci.yml / .github/workflows/, скриншоты этапов.

1. Настроить OIDC-федерацию GitLab CI → Yandex Cloud. Продемонстрировать успешный terraform init с временным токеном.
2. Настроить OIDC-федерацию GitHub Actions → AWS. Пайплайн выводит aws sts get-caller-identity без статических ключей.
3. Пайплайн с этапом Trivy-сканирования. Quality Gate блокирует при CRITICAL.

4. Пайплайн с публикацией образа в Container Registry через временную аутентификацию.
5. Terraform-деплой простой ВМ с Nginx через OIDC-аутентификацию.
6. Разделение пайплайна на stage: plan (автоматически) и apply (manual approval).
7. Интеграция tfint в пайплайн перед plan.
8. git-secrets шаг: проверка отсутствия секретов в коммите.
9. Автоматическое тегирование образа: git short sha + ветка.
10. Пайплайн с окружениями (staging/production), разные OIDC-роли для каждого.
11. Добавление шага terraform fmt -check для валидации стиля кода.
12. Использование docker build --secret для передачи чувствительных файлов без сохранения в слоях.
13. Пайплайн с блокировкой деплоя в пятницу вечером (условие в rules).
14. Интеграция dockle для проверки best-practices Dockerfile.
15. Артефакты пайплайна: сохранение tfplan как артефакта, скачивание для аудита.

4.2. Средний уровень (15 вариантов)

Требуется комплексный пайплайн с канареечным разворачиванием или multi-cloud.

1. Реализовать multi-cloud пайплайн: одно приложение деплоится и в Yandex Cloud, и в AWS через разные OIDC-роли.
2. Canary-деплой: создать две ВМ (blue/green), переключить трафик через Load Balancer.
3. Terraform destroy для feature-веток: автоматическое удаление инфраструктуры при мерже.
4. Интеграция Checkov в пайплайн (шаг до Terraform plan). Quality Gate блокирует при нарушениях.
5. Реализовать Dynamic Review Environments: для каждого MR создается временная инфраструктура с уникальным URL.
6. Интеграция с Vault: пайплайн получает секреты БД через Vault AppRole, а не хардкодит в переменных.
7. Terraform State Locking: использовать S3/Object Storage как бэкенд с DynamoDB/Redis для блокировки.
8. Автоматическая ротация секретов: пайплайн перевыпускает сертификаты Let's Encrypt после деплоя.
9. Пайплайн с матричным тестированием: параллельный деплой в 3 региона, проверка доступности.
10. Slack-уведомления: при успешном деплое и при failure (разные шаблоны).
11. Интеграция infracost в пайплайн: оценка стоимости инфраструктуры до apply, блокировка при превышении бюджета.

12. Подпись образов Cosign: пайплайн подписывает образ, деплой проверяет подпись.
13. Пайплайн с GitOps (Pull-based): вместо terraform apply создается PR в GitOps-репозиторий.
14. Сканирование зависимостей: добавление pip-audit или npm audit шага.
15. Chaos-тестирование пайплайна: намеренная инъекция уязвимости, проверка блокировки Quality Gate.

4.3. Повышенный уровень (15 вариантов)

Требуется построение enterprise-grade пайплайна с полным DevSecOps-циклом.

1. Разработать кастомный GitLab CI шаблон (template) для переиспользования в других проектах.
2. Реализовать Policy-Based Deployment: перед terraform apply OPA проверяет план на соответствие корпоративным политикам.
3. Интеграция с SIEM: все шаги пайплайна логируются в ELK, включая OIDC-токены (анонимизированно).
4. Автоматический rollback: при failure health-check после деплоя автоматический terraform destroy и откат на предыдущую версию.
5. Multi-stage approval chain: deploy в staging → автотесты → approval security-team → deploy в production.
6. Интеграция с ServiceNow/Jira: пайплайн создает Change Request, ждет approval, затем деплоит.
7. Реализация Software Supply Chain Security: SBOM-генерация, подпись, provenance attestation (SLSA Level 3).
8. Zero-Trust пайплайн: каждый шаг выполняется в изолированном контейнере с отдельной OIDC-ролью (минимальные права).
9. GitLab Dedicated + OIDC с корпоративным IdP (Azure AD, Keycloak): сквозная аутентификация.
10. Мониторинг пайплайна: Prometheus-экспортер собирает метрики длительности этапов, частоты failure.
11. Автоматическая генерация Compliance Report после деплоя: Terraform state + Custodian + OPA = единый PDF-отчет.
12. Self-healing пайплайн: при transient error (сеть) автоматический retry с экспоненциальной задержкой.
13. Интеграция с системой управления уязвимостями (DefectDojo): Trivy-отчеты автоматически импортируются.
14. Кросс-проектная оркестрация: деплой микросервисов в правильном порядке с проверкой зависимостей.
15. Air-gapped deployment: пайплайн, работающий в изолированной среде без доступа к интернету, с локальными зеркалами образов.

5. Контрольные вопросы

1. Объясните механизм работы OIDC-федерации: какие claims содержит JWT-токен GitLab CI и как облачный провайдер проверяет его подлинность?

2. В чем преимущества OIDC-федерации перед статическими ключами доступа? Назовите минимум 4 аспекта (безопасность, эксплуатация, аудит, ротация).
3. Как работает Quality Gate в контексте DevSecOps-пайплайна? Почему блокировка пайплайна при CRITICAL-уязвимостях важнее скорости доставки?
4. Какие меры защиты артефактов пайплайна (Docker-образов, tfplan, отчетов) необходимо предпринять для предотвращения атак на цепочку поставок (Supply Chain)?
5. Опишите сценарий компрометации CI/CD-системы через утечку статического ключа и объясните, как OIDC минимизирует ущерб от такого инцидента.
6. Требования к отчету

Отчет в формате PDF. Обязательные разделы:

1. Титульный лист.
2. Архитектура пайплайна: диаграмма (draw.io) этапов CI/CD с указанием инструментов, точек Quality Gate и механизма OIDC-аутентификации.
3. Настройка OIDC-федерации:
 - Скриншоты конфигурации IAM-федерации в облачной консоли.
 - Скриншот доверительной политики (subject mapping).
 - Скриншот переменных CI/CD (отсутствие ключей доступа!).
4. Конфигурация пайплайна:
 - Полный листинг `.gitlab-ci.yml` / `.github/workflows/deploy.yml` с комментариями.
 - Листинг Terraform-конфигурации (`provider.tf`, `main.tf`).
5. Результаты выполнения:
 - Скриншот лога пайплайна, демонстрирующий получение временного токена.
 - Скриншот успешного `terraform apply` с OIDC.
 - Скриншот блокировки пайплайна при CRITICAL-уязвимости (Quality Gate).
6. Подтверждение безопасности:
 - Вывод `terraform state list` развернутых ресурсов.
 - Trivy-отчет (SARIF/HTML) в приложении.
7. Заключение: обоснование необходимости перехода на OIDC-федерацию для всех production CI/CD-пайплайнов.

7. Критерии оценивания

7.1. Базовый уровень («удовлетворительно»)

- Реализован пайплайн с этапами: build, scan (Trivy), deploy.
- OIDC-федерация настроена, продемонстрирована работа без статических ключей.
- Trivy-скан выполняется, отчет сохранен как артефакт.
- Terraform успешно развернул инфраструктуру.

- В отчете есть скриншоты логов пайплайна и переменных CI/CD.
- Снижение: обнаружены статические ключи в переменных CI/CD — минус 2 балла.

7.2. Средний уровень («хорошо»)

- Выполнены все критерии базового уровня.
- Quality Gate реально блокирует пайплайн (продемонстрирован failure с CRITICAL).
- Реализован ручной этап approval перед apply.
- Настроены окружения (staging/production) с разными OIDC-ролями.
- Артефакты сохраняются с разумным сроком хранения.
- Снижение: Quality Gate настроен как allow_failure: true — минус 2 балла.

7.3. Повышенный уровень («отлично»)

- Выполнены все критерии среднего уровня.
- Реализован GitOps-подход или Policy-Based Deployment с OPA.
- Внедрена подпись образов (Cosign) и проверка подписи при деплое.
- Реализован автоматический rollback при failure.
- Пайплайн включает Canary-деплой или Blue/Green стратегию.
- Интегрирован SBOM-генератор и DefectDojo.
- Снижение: отсутствует автоматический rollback — минус 1 балл (оценка «хорошо»).

7.4. Общие штрафы

- Использование статических ключей в production-окружении — работа не принимается, передача.
- Отсутствие этапа terraform destroy после выполнения (оставлены ресурсы) — минус 2 балла.
- Trivy-отчет показывает CRITICAL, но пайплайн помечен как успешный — минус 2 балла.

Лабораторная работа №9. Аудит облачной инфраструктуры (Prowler / ScoutSuite)

1. Цель

Освоить методику автоматизированного аудита безопасности облачной инфраструктуры с использованием специализированных сканеров. Научиться проводить оценку соответствия стандартам CIS, выявлять высокорисковые конфигурации, анализировать многокомпонентные отчёты и формировать план устранения нарушений.

2. Задачи

1. Установить и настроить инструмент аудита (Prowler для AWS / ScoutSuite для мультиоблачной среды) для работы с тестовым облачным окружением.
2. Выполнить полный аудит облачной среды по всем доступным проверкам (CIS, PCI DSS, GDPR, SOC 2).
3. Проанализировать сгенерированный отчёт (HTML/JSON), классифицировать нарушения по уровням критичности (Critical / High / Medium / Low).
4. Выявить минимум 5 значимых нарушений и подготовить детальный план их устранения.
5. Реализовать исправление минимум 2 выявленных нарушений и выполнить повторный аудит для подтверждения устранения.

3. Теоретическое обоснование

Необходимость аудита облачной инфраструктуры. В отличие от статического анализа IaC (ЛР №5) и рантайм-мониторинга отдельных ресурсов (ЛР №7), комплексный аудит облака решает задачу целостной оценки безопасности всей среды. Даже при идеально настроенных CI/CD-пайплайнах с течением времени накапливаются проблемы: «зомби-ресурсы» (забытые инстансы), дрейф конфигураций, избыточные IAM-права, неиспользуемые ключи доступа. Многие нарушения невозможно обнаружить анализом одного ресурса — они проявляются только при кросс-сервисном анализе (например, Security Group, открывающая доступ к базе данных, которая использует дефолтный пароль).

Инструменты аудита:

- Prowler (Open Source, Toni de la Fuente) — специализированный сканер для AWS, Azure и GCP. Проводит более 300 проверок, покрывающих CIS Benchmarks, PCI DSS, GDPR, HIPAA, NIST 800-53, ENS (испанский стандарт), Australian ISM. Особенность: быстрый (использует AWS API напрямую), легко интегрируется в CI/CD, поддерживает вывод в HTML/CSV/JSON/ASFF (AWS Security Hub).
- ScoutSuite (NCC Group) — мультиоблачный сканер (AWS, Azure, GCP, Oracle Cloud). Отличается глубоким кросс-сервисным анализом: строит граф ресурсов и выявляет сложные цепочки нарушений (например, EC2 с публичным IP → Security Group с 0.0.0.0/0 → IAM роль с AdministratorAccess). Генерирует интерактивные HTML-отчёты с навигацией по сервисам.

Ключевые категории проверок (CIS AWS Foundations):

- IAM (Identity and Access Management): Отсутствие root API ключей, включенный MFA для всех пользователей, отсутствие неиспользуемых ключей старше 90 дней, политики паролей, минимальные права.
- Logging (Мониторинг): Включенный CloudTrail во всех регионах, логирование S3-бакетов, логирование VPC Flow Logs, защита логов от удаления.
- Monitoring (Алертинг): Настроенные алерты на вход root-пользователя, изменение сетевых ACL, изменение политик S3, неудачные попытки входа в консоль.
- Networking: Security Groups без правила 0.0.0.0/0 для портов 22, 3389, 5432; VPC Flow Logs включены; дефолтные Security Groups не используются.

Методология аудита:

1. Сбор данных (Discovery): Сканер опрашивает облачное API, собирает конфигурации всех ресурсов.
2. Оценка (Assessment): Каждый чек (check) оценивает ресурс на соответствие определённому критерию (PASS / FAIL / MANUAL).
3. Классификация (Severity Rating): Нарушения маркируются как Critical (прямая угроза компрометации), High (значительно повышает риск), Medium (не соответствует лучшим практикам), Low (информационное).
4. Формирование отчёта (Reporting): Генерация структурированного отчёта с детализацией каждого нарушения и рекомендациями по исправлению.
5. План коррекции (Remediation Plan): Приоритизированный список действий для устранения нарушений.

Связь с компетенцией ПК-8:

- ИД-7пк-8: *Мониторит состояние безопасности инфраструктуры в рантайме, выявляя инциденты и аномальное поведение.* Аудит — периодическая форма такого мониторинга.

- ИД-8пк-8 (целевой индикатор): *Обеспечивает непрерывное соответствие через автоматизированный сбор доказательств и генерацию отчетов.* HTML/JSON-отчёты Prowler/ScoutSuite являются юридически значимыми доказательствами для внешних аудиторов.

4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

Провайдер: AWS (свободный tier) или Yandex Cloud. Инструмент: Prowler или ScoutSuite.

1. Запустить Prowler по CIS AWS Foundations Level 1. Предоставить скриншот summary с количеством PASS/FAIL.
2. Провести аудит только сервиса S3. Выявить публичные бакеты.
3. Провести аудит IAM: пользователи без MFA, неиспользуемые ключи доступа.
4. Аудит Security Groups: найти правила с 0.0.0.0/0 для портов 22, 3389, 3306.
5. Запустить ScoutSuite в режиме --services s3,iam. Сравнить результаты с Prowler.
6. Проанализировать HTML-отчёт ScoutSuite. Сделать скриншот Dashboard и страницы IAM.
7. Выполнить аудит EBS-дисков: найти незашифрованные тома.
8. Аудит CloudTrail: все ли регионы охвачены? Настроено ли логирование S3?
9. Выявить Security Groups, не используемые ни одним инстансом.
10. Найти IAM-пользователей с паролем старше 90 дней (без ротации).
11. Аудит RDS: публично доступные базы данных, отсутствие шифрования.
12. Prowler с флагом --severity critical: отчёт только по критическим нарушениям.
13. Генерация отчёта в CSV и загрузка в Google Sheets для анализа.
14. Сравнительный прогон: до и после исправления одного нарушения (скриншоты).
15. Запуск Prowler в Docker-контейнере с монтированием AWS-credentials.

4.2. Средний уровень (15 вариантов)

Требуется мультирегиональный аудит, автоматизированная коррекция, кастомные проверки.

1. Мультирегиональный аудит: Prowler по всем регионам (eu-west-1, eu-central-1, us-east-1).
2. Интеграция Prowler с AWS Security Hub: отправка findings в формате ASFF.
3. Написать скрипт (Python/Bash), который парсит JSON-отчёт Prowler и формирует Jira-тикеты на каждое CRITICAL-нарушение.
4. Настроить еженедельный запуск аудита через AWS Lambda + EventBridge.
5. Сравнительный аудит: Prowler vs ScoutSuite vs Steampipe. Сводная таблица обнаруженных нарушений.
6. Кастомная проверка Prowler: написать чек, проверяющий наличие тега DataClassification на S3-бакетах.

7. Аудит по стандарту PCI DSS: включить соответствующие проверки.
8. Интеграция с Slack: бот публикует сводку CRITICAL-нарушений после каждого аудита.
9. Мультиоблачный аудит ScoutSuite: AWS + Azure в одном прогоне.
10. Настроить Prowler с Assume Role: аудит целевого аккаунта из аудиторского.
11. Реализовать автоматическую коррекцию: скрипт читает JSON-отчёт и применяет исправления (например, включает Block Public Access на S3).
12. Генерация отчёта для руководства: HTML-шаблон с графиками (matplotlib) по результатам аудита.
13. Аудит конкретного VPC: проверить все ресурсы внутри одной сети.
14. Настроить мониторинг аудита: Prometheus-экспортер метрик количества FAIL по severity.
15. Использование Prowler с --output-asff и визуализация в AWS Security Hub.

4.3. Повышенный уровень (15 вариантов)

Требуется построение системы непрерывного аудита с авторемедиацией и интеграцией в GRC.

1. Разработать систему Continuous Auditing: Lambda-функция запускает Prowler каждый час, отправляет метрики в CloudWatch, алерты при новых FAIL.
2. Интеграция с корпоративной GRC-системой (ServiceNow/Archer): автоматическая синхронизация findings.
3. Написать OPA-политику, которая валидирует результаты Prowler-аудита: `deny[msg] { input.Findings[].Severity == "CRITICAL" }`.
4. Развернуть Prowler как централизованный аудиторский сервис с веб-интерфейсом (Flask + Celery).
5. Полный цикл: CI/CD → Deploy → Prowler Audit → Auto-Remediation → Re-audit → Compliance Report (единый пайплайн).
6. Customный дашборд аудита в Grafana с историческими данными (Elasticsearch/Kibana).
7. Разработка customного чек-провайдера для Prowler (Yandex Cloud API).
8. Машинное обучение на результатах аудита: предсказание вероятных будущих нарушений (аномалии в паттернах конфигураций).
9. Интеграция с SOAR (Shuffle/Tines): при CRITICAL-нарушении запускается playbook автоматической изоляции ресурса.
10. Мультиаккаунтный аудит через AWS Organizations: единый отчёт по всем аккаунтам.
11. Реализация Compliance Score: процент соответствия по CIS, динамика по неделям, алерт при снижении.
12. Сравнение результатов аудита с IaC (Terraform state): выявление дрейфа конфигураций.
13. Разработка ChatOps-интерфейса: аудит запускается командой в Slack, результат возвращается в тред.

14. Интеграция Prowler с Threat Intelligence: проверка публичных IP/доменов в фидах угроз.
15. Форк Prowler с добавлением поддержки Yandex Cloud (разработка минимум 10 CIS-чеков).

5. Контрольные вопросы

1. Чем отличается подход Prowler (на основе чек-листов) от подхода ScoutSuite (на основе графа ресурсов)? В каких сценариях какой инструмент предпочтительнее?
2. Какие права необходимы сервисному аккаунту аудитора для выполнения полного сканирования облачной среды? Обоснуйте принцип наименьших привилегий для аудиторской роли.
3. Объясните концепцию Compliance Score. Как его расчёт и визуализация помогают в управлении безопасностью облака?
4. Почему мультирегиональный аудит важен даже если компания использует только один регион? Какие глобальные ресурсы (IAM, S3) могут остаться непроверенными?
5. Как интегрировать инструменты аудита (Prowler/ScoutSuite) в процесс Continuous Compliance? Опишите архитектуру решения с автоматической коррекцией нарушений.

6. Требования к отчету

Отчет в формате PDF. Обязательные разделы:

1. Титульный лист.
2. Архитектура аудита: диаграмма (draw.io) компонентов (Scanner → Cloud API → Report Engine → CI/CD).
3. Конфигурация и запуск:
 - Команда запуска аудита с параметрами.
 - Скриншот вывода консоли с суммарной статистикой (PASS/FAIL).
4. Анализ результатов:
 - Таблица топ-10 нарушений с приоритетом (CRITICAL → MEDIUM).
 - Скриншоты HTML-отчёта (Dashboard, IAM, S3, EC2).
 - Описание 3 наиболее критичных нарушений: в чём угроза, как эксплуатировать (моделирование атаки).
5. План коррекции:
 - Таблица с приоритетами, ответственными, сроками.
 - Команды/конфигурации для исправления минимум 2 нарушений.
6. Повторный аудит:
 - Команда повторного запуска.
 - Сводная таблица ДО/ПОСЛЕ: количество FAIL по severity.
 - Скриншоты, подтверждающие переход проверок в PASS.
7. Заключение: оценка зрелости безопасности тестовой среды, рекомендации по внедрению регулярного аудита.

7. Критерии оценивания

7.1. Базовый уровень («удовлетворительно»)

- Prowler или ScoutSuite установлен и успешно запущен.
- Выполнен аудит минимум 3 сервисов (IAM, S3, EC2).
- Сгенерирован HTML-отчёт, предоставлены скриншоты Dashboard.
- Составлена таблица нарушений с severity (минимум 5 записей).
- Исправлено минимум 1 нарушение, подтверждено повторным аудитом.
- Снижение: отчёт не проанализирован (просто скриншот без описания) — минус 1 балл.

7.2. Средний уровень («хорошо»)

- Выполнены все критерии базового уровня.
- Аудит проведён по CIS-бенчмарку (полный, не выборочный).
- Сформирован формализованный план коррекции с приоритетами и сроками.
- Исправлено минимум 3 нарушения разного типа (IAM, Storage, Network).
- Настроена интеграция в CI/CD (GitLab CI/GitHub Actions) или Lambda.
- Снижение: план коррекции отсутствует или неформален — минус 2 балла.

7.3. Повышенный уровень («отлично»)

- Выполнены все критерии среднего уровня.
- Реализована автоматическая коррекция минимум 1 типа нарушений (скрипт).
- Настроен еженедельный запуск и/или алертинг по результатам аудита.
- Интегрирован OPA для валидации результатов аудита.
- Построен Compliance Dashboard (Grafana/AWS QuickSight) с историей аудитов.
- Выполнен мультиаккаунтный или мультиоблачный аудит.
- Снижение: отсутствует автоматическая коррекция — минус 1 балл (оценка «хорошо»).

7.4. Общие штрафы

- Аудит проводился с правами AdministratorAccess (избыточные права аудитора) — минус 1 балл, инструктаж по PoLP.
- Критическое нарушение (например, публичный бакет с реальными данными) не исправлено после аудита — минус 2 балла.
- Отсутствуют артефакты отчёта (HTML/JSON файлы) — минус 1 балл.

Лабораторная работа №10. Управление затратами в облаке (FinOps)

1. Цель

Освоить практические методы анализа и оптимизации расходов на облачную инфраструктуру в рамках концепции FinOps. Научиться настраивать бюджетные ограничения с автоматическим оповещением, анализировать структуру затрат и внедрять политики автоматического отключения ресурсов в нерабочее время для снижения операционных расходов.

2. Задачи

1. Провести анализ текущих и исторических затрат облачной инфраструктуры с помощью встроенных инструментов провайдера (Billing Dashboard / Cost Explorer).
2. Настроить детализированные бюджеты с пороговыми значениями (50%, 80%, 100%) и мультиканальными оповещениями (email, Slack, Webhook).
3. Разработать и применить политику автоматического останова/запуска ресурсов по расписанию (нерабочее время, выходные).
4. Оценить экономический эффект от внедрения автоматической остановки ресурсов (расчёт сэкономленных средств).
5. Сформировать рекомендации по дальнейшей оптимизации затрат на основе анализа utilisation report.

3. Теоретическое обоснование

FinOps (Cloud Financial Operations) — культурная и операционная практика, объединяющая финансы, технологии и бизнес для управления облачными затратами с целью максимизации ценности облака. FinOps основан на трёх принципах:

- Прозрачность (Visibility): Все команды имеют доступ к данным о расходах своих ресурсов.
- Ответственность (Accountability): Каждая команда несёт финансовую ответственность за потребляемые ресурсы (Showback / Chargeback).
- Оптимизация (Optimization): Постоянный цикл улучшения: анализ → планирование → внедрение → измерение.

Жизненный цикл FinOps:

1. Inform — сбор и визуализация данных о затратах.
2. Optimize — поиск и реализация возможностей для сокращения расходов.
3. Operate — внедрение процессов непрерывного контроля и управления затратами.

Ключевые метрики FinOps:

- Total Cloud Spend — общие затраты на облако за период.
- Cost per Unit — стоимость на единицу бизнес-метрики (на пользователя, на транзакцию).
- Utilization Rate — процентное использование оплаченных ресурсов (CPU, память, диск).
- Waste — затраты на неиспользуемые или простаивающие ресурсы.
- Reserved vs On-Demand — соотношение зарезервированных мощностей к потребляемым по факту.

Инструменты анализа затрат:

- Yandex Cloud Billing — консольный инструмент с детализацией по сервисам, проектам, тегам; экспорт в CSV.
- AWS Cost Explorer — продвинутая аналитика с прогнозированием, правами доступа (RI/SP), фильтрацией по тегам.
- Infracost — оценка стоимости Terraform-плана до применения (Shift Left for FinOps).

Стратегии оптимизации:

- Автоматический останов по расписанию — выключение dev/staging окружений в нерабочее время экономит до 65% затрат на compute.
- Right-sizing — подбор оптимального размера инстанса под фактическую нагрузку.
- Reserved Instances / Savings Plans — долгосрочные обязательства в обмен на скидку (до 72%).
- Удаление неиспользуемых ресурсов — «зомби-диски», неиспользуемые IP, старые снапшоты.
- Tiered Storage — перемещение холодных данных в дешёвые классы хранения (S3 Glacier, Object Storage Ice).

Связь с компетенцией ПК-8: Задача №7 дисциплины: *Изучить экономические аспекты безопасности и оптимизации затрат (FinOps) в гибридных облаках.* Безопасность и финансы тесно связаны: неиспользуемые ресурсы не только тратят бюджет, но и расширяют поверхность атаки. Автоматический останов dev-окружений снижает и затраты, и риск компрометации.

Финансовые риски облачной безопасности:

- Неограниченный бюджет → возможность скрытого майнинга криптовалют (стоимость резко возрастает).
- Отсутствие алертов → компрометация аккаунта и запуск дорогих ресурсов (GPU-инстансы) остаются незамеченными.
- Неиспользуемые публичные IP → платные и уязвимые для атак.

4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

1. Проанализировать затраты за последний месяц в консоли Yandex Cloud Billing. Предоставить CSV-отчёт с детализацией по сервисам.
2. Настроить бюджет с порогом 80% и email-уведомлением. Продемонстрировать скриншот конфигурации.
3. Создать CloudWatch / Yandex Monitoring алерт при превышении дневного лимита.
4. Разработать скрипт (bash), который выводит список всех инстансов с их стоимостью в час.
5. Настроить AWS Cost Anomaly Detection с алертом при аномальном росте расходов.
6. Экспортировать данные о расходах в Google Sheets и построить график по дням.

7. Найти 3 самых дорогих ресурса в облаке. Предложить оптимизацию (таблица).
8. Написать Python-скрипт, который парсит CSV из Billing и группирует расходы по тегам.
9. Настроить Budget в AWS на 500 USD/мес с уведомлением через SNS.
10. Проанализировать затраты на Object Storage: выявить дорогие бакеты, рассчитать стоимость per GB.
11. Сравнить стоимость инстансов разных поколений (standard-v1 vs standard-v3).
12. Настроить алерт в Yandex Cloud при превышении бюджета на 50%.
13. Рассчитать стоимость незашифрованных EBS-дисков и сравнить с зашифрованными.
14. Создать отчёт по тегам: распределение затрат по Environment (dev/staging/prod).
15. Использование Infracost для оценки стоимости Terraform-плана до деплоя.

4.2. Средний уровень (15 вариантов)

1. Реализовать автоматическую остановку инстансов с тегом Scheduled=true в нерабочее время (Cloud Function + Trigger).
2. Настроить Slack-уведомления при превышении порогов бюджета (50%, 80%, 100%).
3. Автоматическое удаление неиспользуемых EBS-дисков / снапшотов старше 30 дней.
4. Разработать дашборд в Grafana (CloudWatch Data Source) с метриками затрат по сервисам.
5. Реализовать Right-sizing рекомендации на основе CloudWatch метрик (CPU < 10% → предложить меньший тип).
6. Настроить автоматический переход объектов в холодное хранилище (Object Storage Ice) через Lifecycle Policy.
7. Интеграция Infracost в CI/CD: блокировка деплоя, если стоимость превышает лимит.
8. Создать политику Custodian для автоматической остановки инстансов без тега CostCenter.
9. Настроить кросс-аккаунтный бюджет в AWS Organizations.
10. Разработать Python-скрипт, генерирующий PDF-отчёт о затратах (matplotlib + fpdf).
11. Реализовать Chargeback/Showback: отчёт по командам на основе тегов.
12. Сравнение On-Demand vs Reserved Instances (Compute Savings Plan). Расчёт точки окупаемости.
13. Автоматическое удаление старых AMI/образов и связанных снапшотов.
14. Настройка AWS Budget Actions: автоматическое применение IAM-политики Deny при 100% бюджета.

15. Интеграция FinOps с Jira: автоматическое создание тикета при превышении командного бюджета.

4.3. Повышенный уровень (15 вариантов)

1. Разработать комплексную систему FinOps: сбор метрик → Infracost → Grafana → Slack-алерты → авто-стоп.

2. Машинное обучение для прогнозирования облачных затрат (Facebook Prophet / ARIMA).

3. Интеграция с ERP-системой (SAP/Oracle): автоматическая выгрузка данных о затратах.

4. Мультиоблачный FinOps дашборд (AWS + Yandex Cloud + Azure) в реальном времени.

5. Разработка кастомного Terraform Provider для финансовых операций (бюджеты, алерты).

6. Реализация самообучающейся системы Right-sizing с автоматическим ресайзом инстансов.

7. Интеграция FinOps с ServiceNow: создание Change Request при превышении бюджета.

8. Внедрение Anomaly Detection на базе ML для выявления аномальных затрат (возможный взлом).

9. Разработка Kubernetes-оператора FinOps, который управляет ресурсами подов на основе стоимости.

10. Создание портала самообслуживания для разработчиков с видимостью стоимости их ресурсов.

11. Интеграция FinOps с системой Service Desk: автоматическое согласование бюджета при создании ресурсов.

12. Разработка модели Total Cost of Ownership (TCO): сравнение облака vs on-premise.

13. Реализация FinOps для serverless-архитектур: анализ стоимости Lambda/Cloud Functions.

14. Интеграция с системой CMDB: автоматическое обновление стоимости активов.

15. Создание открытого FinOps-фреймворка для Yandex Cloud (набор скриптов/политик).

5. Контрольные вопросы

1. Объясните три фазы жизненного цикла FinOps (Inform, Optimize, Operate) и приведите примеры инструментов для каждой фазы.

2. Каковы ключевые различия между моделями Showback и Chargeback? В каких случаях каждая из них уместна?

3. Как автоматическая остановка инстансов в нерабочее время влияет на безопасность облачной инфраструктуры?

4. Какие финансовые индикаторы могут сигнализировать о компрометации облачного аккаунта?

5. Как интегрировать практики FinOps в CI/CD-пайплайн? Какие шлюзы качества (Quality Gates) по стоимости можно настроить?

6. Требования к отчету

Отчет в формате PDF. Обязательные разделы:

1. Титульный лист.
2. Анализ текущих затрат:
 - Детализация затрат по сервисам (таблица + круговая диаграмма).
 - Динамика затрат по дням за месяц (линейный график).
 - Топ-5 самых дорогих ресурсов.
3. Конфигурация бюджета:
 - Скриншоты настроек бюджета с порогами и алертами.
 - Листинг Terraform-конфигурации бюджета (если применимо).
 - Демонстрация сработавшего алерта (email/Slack).
4. Автоматизация управления ресурсами:
 - Листинг скрипта/функции для остановки по расписанию.
 - Конфигурация триггера (Cron/EventBridge).
 - Скриншот логов выполнения (остановка/запуск).
5. Расчёт экономического эффекта:
 - Таблица сэкономленных средств по ресурсам.
 - Прогноз годовой экономии.
6. Рекомендации по дальнейшей оптимизации:
 - 5 конкретных предложений с расчётной экономией.
 - Приоритет внедрения.
7. Заключение: оценка зрелости FinOps-практик в тестовой среде.

7. Критерии оценивания

7.1. Базовый уровень («удовлетворительно»)

- Проведён анализ затрат за последний месяц (скриншоты + CSV).
- Настроен минимум один бюджет с порогом и email-уведомлением.
- Написан скрипт (bash/Python) для вывода списка инстансов с почасовой стоимостью.
- Предложено минимум 3 рекомендации по оптимизации (таблица).
- Снижение: анализ затрат проведён поверхностно (без цифр) — минус 1 балл.

7.2. Средний уровень («хорошо»)

- Выполнены все критерии базового уровня.
- Реализована автоматическая остановка инстансов по расписанию (Cloud Function + Trigger).

- Настроены мультиканальные уведомления (email + Slack/webhook).
- Выполнена интеграция Infracost в CI/CD.
- Рассчитан экономический эффект (таблица + прогноз).
- Снижение: автоматическая остановка не покрывает выходные — минус 1 балл.

7.3. Повышенный уровень («отлично»)

- Выполнены все критерии среднего уровня.
- Построен FinOps-дашборд (Grafana) с историческими данными.
- Внедрена система Chargeback/Showback по тегам.
- Реализованы Budget Actions (автоматические действия при 100% бюджета).
- Интегрирована система прогнозирования затрат (ML/статистика).
- Снижение: отсутствует дашборд — минус 1 балл (оценка «хорошо»).

7.4. Общие штрафы

- Не выполнена очистка ресурсов после демонстрации (оставлены тестовые инстансы) — минус 2 балла.
- Бюджет настроен без ограничений (любое превышение допустимо) — минус 1 балл.
- Отсутствует расчёт экономического эффекта — минус 2 балла.

Лабораторная работа №11. Service Mesh и mTLS (Istio)

1. Цель

Освоить практику построения безопасного сервис-меша с взаимной TLS-аутентификацией (mTLS) между микросервисами в Kubernetes. Научиться настраивать строгую политику mTLS, управлять доступом на основе ролей через AuthorizationPolicy и контролировать восточно-западный трафик (east-west) в соответствии с концепцией Zero Trust.

2. Задачи

1. Развернуть тестовый Kubernetes-кластер (kind/minikube) и установить Istio Service Mesh.
2. Разработать и задеплоить два тестовых микросервиса, взаимодействующих по HTTP.
3. Последовательно включить mTLS: от permissive-режима к strict-режиму, верифицировать шифрование трафика через анализ сертификатов.

4. Настроить AuthorizationPolicy для реализации микросегментации: разрешить доступ только от сервиса А к сервису Б, блокируя все остальные соединения.

5. Продемонстрировать работу политик: успешный запрос между разрешёнными сервисами и возврат 403 Forbidden для запрещённых.

3. Теоретическое обоснование

Service Mesh — выделенный инфраструктурный слой для управления коммуникациями между сервисами, реализованный через sidecar-прокси (Envoy в случае Istio), которые перехватывают весь входящий и исходящий трафик подов. Это позволяет централизованно применять политики безопасности, маршрутизации и наблюдаемости без изменения кода приложений.

Ключевые концепции безопасности Istio:

- Взаимный TLS (mTLS) — протокол, при котором обе стороны соединения аутентифицируют друг друга с помощью X.509 сертификатов. В отличие от стандартного TLS (где только сервер предъявляет сертификат), mTLS гарантирует, что:
 - Идентификация: Сервис А уверен, что общается именно с сервисом Б (а не с подставным узлом).
 - Шифрование: Весь трафик между сервисами зашифрован в прозрачном режиме.
 - Целостность: Данные не могут быть изменены при передаче.
- PeerAuthentication — ресурс Istio, определяющий политику аутентификации для рабочих нагрузок:
 - PERMISSIVE — sidecar принимает как mTLS, так и plain-text трафик (режим миграции).
 - STRICT — sidecar принимает ТОЛЬКО mTLS-соединения.
- AuthorizationPolicy — ресурс Istio, реализующий управление доступом на основе ролей (RBAC) для сервис-меша. Политики поддерживают фильтрацию по:
 - source — пространство имён, принципал (service account), IP-блок.
 - operation — HTTP-метод, путь, заголовки (для HTTP), порты.
 - conditions — кастомные атрибуты из контекста запроса.
- Istio CA (Certificate Authority) — встроенный центр сертификации Istiod, который автоматически выпускает и ротирует X.509 сертификаты для каждого sidecar-прокси. Срок действия сертификата по умолчанию 24 часа, что исключает проблему долгоживущих ключей.

Принцип Zero Trust в контексте сервис-меша: Даже находясь внутри одного кластера, сервисы не должны доверять друг другу по умолчанию. Istio реализует Zero Trust через:

1. Аутентификацию (mTLS) — каждый запрос аутентифицируется, анонимный трафик блокируется.
2. Авторизацию (AuthorizationPolicy) — аутентифицированный запрос проверяется на соответствие политикам доступа.
3. Шифрование — весь трафик зашифрован, предотвращая перехват внутри кластера.

Связь с компетенцией ПК-8:

- ИД-7пк-8 (целевой индикатор): *Мониторит состояние безопасности контейнеров и инфраструктуры в рантайме, выявляя инциденты и аномальное поведение. Знает архитектуру Service Mesh (Istio, Linkerd), принципы mTLS.* Данная ЛР полностью реализует этот индикатор.

- ИД-8пк-8: Настройка политик авторизации как реализация Compliance as Code для микросервисных взаимодействий.

4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

1. Установить Istio, задеплоить два сервиса. Проверить связь, включить STRICT mTLS. Показать, что запросы работают.
2. Создать PeerAuthentication в PERMISSIVE-режиме. Продемонстрировать, что и mTLS, и plain-text работают.
3. Написать AuthorizationPolicy, разрешающую GET-запросы только от namespace default.
4. Настроить политику DENY для метода DELETE на backend.
5. Показать содержимое X.509 сертификата, используемого для mTLS (openssl s_client).
6. Создать три сервиса (frontend, middleware, backend). Проверить цепочку вызовов с mTLS.
7. Включить STRICT mTLS только для одного namespace, оставив другой в PERMISSIVE.
8. Написать AuthorizationPolicy с фильтрацией по HTTP-заголовку X-API-Key.
9. Проверить логи sidecar (istio-proxy) на наличие записей о блокировке RBAC.
10. Задеплоить сервис без sidecar (инжекция отключена), показать ошибку подключения в STRICT-режиме.
11. Настроить AuthorizationPolicy, разрешающую доступ только с определённого IP-блока.
12. Создать политику с условием request.auth.claims[iss] для JWT.
13. Проверить метрики mTLS через istioctl proxy-status.
14. Показать, что при отключении mTLS (DISABLE) сертификаты не используются.
15. Использовать istioctl authz check для проверки прав конкретного пода.

4.2. Средний уровень (15 вариантов)

1. Настроить JWT-аутентификацию: RequestAuthentication + AuthorizationPolicy по claims.
2. Реализовать многоуровневую политику: ALLOW frontend→backend, DENY frontend→backend/admin/*.
3. Настроить egress-трафик через Istio: разрешить внешние вызовы только к одному хосту.
4. Внедрить взаимную аутентификацию между сервисами из разных неймспейсов.

5. Настроить канареечное развертывание с mTLS: Istio VirtualService + DestinationRule.
6. Интеграция с внешним IdP: OAuth2-Прoxy перед сервисами, Istio проверяет JWT.
7. Настроить мониторинг mTLS-сертификатов: алерт при истечении срока.
8. Реализовать Rate Limiting через Istio EnvoyFilter.
9. Создать гибридную политику: часть сервисов в STRICT, часть в PERMISSIVE, мост между ними.
10. Настроить взаимную аутентификацию с кастомным CA (не Istio CA).
11. Импорт внешних сертификатов в Istio через Secret.
12. Анализ логов access log Envoy: показать разницу между ALLOW и DENY записями.
13. Интеграция с HashiCorp Vault: Istio получает сертификаты из Vault PKI.
14. Построение графа зависимостей сервисов через Kiali с отображением mTLS-статуса.
15. Тестирование производительности: latency overhead от mTLS (ab/siege с и без).

4.3. Повышенный уровень (15 вариантов)

1. Развернуть мультикластерный Service Mesh (Istio multicluster) с общим mTLS.
2. Реализовать Zero Trust архитектуру: ALLOW-политика по умолчанию DENY, только явные разрешения.
3. Интеграция с SPIRE для федеративного управления идентификацией в сервис-меше.
4. Настройка External Authorization: Istio делегирует решение OPA через ext_authz gRPC.
5. Кастомный EnvoyFilter для инъекции security-заголовков (HSTS, CSP).
6. Реализовать Secure Naming: Istio проверяет, что сертификат сервера соответствует имени сервиса.
7. Настроить автоматическую ротацию сертификатов с нулевым даунтаймом (graceful rotation).
8. Интеграция с системой обнаружения аномалий (Falco/Tetragon) для мониторинга подозрительного трафика.
9. Разработка оператора Kubernetes, автоматически создающего AuthorizationPolicy на основе аннотаций.
10. Нагрузочное тестирование mTLS с 1000+ сервисов (оценка масштабируемости Istio CA).
11. Интеграция с Cilium: сравнение Istio mTLS vs Cilium Network Policy + WireGuard.
12. Реализация политики на основе географического расположения (Geo-IP фильтрация через Envoy WASM).

13. Построение Compliance-отчёта: все межсервисные соединения зашифрованы (пруфы через Kiali + логи).

14. Интеграция с системой управления API (Kong/APISIX) с пробросом mTLS до бэкенда.

15. Разработка кастомного WASM-фильтра для Envoy, проверяющего целостность данных.

5. Контрольные вопросы

1. Чем STRICT-режим mTLS отличается от PERMISSIVE? В каком порядке следует внедрять mTLS в production-кластере?

2. Как Istio автоматически ротирует сертификаты sidecar-прокси? Что произойдёт, если истечёт срок действия сертификата во время активного соединения?

3. Объясните разницу между PeerAuthentication и AuthorizationPolicy. Почему mTLS без AuthorizationPolicy недостаточен для Zero Trust?

4. Как работает SPIFFE (Secure Production Identity Framework for Everyone) в контексте Istio? Что содержит поле URI SAN в сертификате sidecar?

5. Какие метрики и логи доступны для аудита mTLS-соединений? Как обнаружить попытку подключения без mTLS в STRICT-режиме?

6. Требования к отчету

Отчет в формате PDF. Обязательные разделы:

1. Титульный лист.
2. Архитектура сервис-меша: диаграмма (draw.io) с подписанными компонентами (Frontend, Backend, Sidecar, Istiod, CA).
3. Установка и конфигурация Istio:
 - Команды установки.
 - Вывод `kubectl get pods -n istio-system`.
 - Конфигурация PeerAuthentication (STRICT).
4. Деплой микросервисов:
 - YAML-манифесты Deployments и Services.
 - Код тестовых приложений (основные фрагменты).
 - Скриншот подов с 2/2 контейнерами.
5. Верификация mTLS:
 - Вывод `openssl s_client`, показывающий сертификат.
 - Скриншот с `tcpdump` (зашифрованный трафик).
 - Скриншот Kiali Graph с замками безопасности.
6. AuthorizationPolicy:
 - YAML-манифест политики.
 - Скриншот успешного запроса (200 OK).
 - Скриншот запрещённого запроса (403 Forbidden).
 - Логи sidecar с записью о блокировке.

7. Заключение: обоснование необходимости сервис-меша и mTLS в микросервисной архитектуре.

7. Критерии оценивания

7.1. Базовый уровень («удовлетворительно»)

- Istio установлен, кластер работает.
- Два сервиса задеплоены с инжектированным sidecar.
- PeerAuthentication настроена в STRICT-режиме.
- Сервисы успешно взаимодействуют.
- Создана минимум 1 AuthorizationPolicy (ALLOW или DENY).
- Снижение: Istio установлен, но sidecar не инжектирован (1/1 вместо 2/2) — минус 2 балла.

7.2. Средний уровень («хорошо»)

- Выполнены все критерии базового уровня.
- Продемонстрирована работа mTLS (сертификаты, зашифрованный трафик).
- Создана гранулярная политика (по HTTP-методам, путям, принципалам).
- Установлен Kiali, показан граф с замками безопасности.
- Выполнена проверка через istioctl authz check.
- Снижение: политики авторизации не покрывают все методы API — минус 1 балл.

7.3. Повышенный уровень («отлично»)

- Выполнены все критерии среднего уровня.
- Настроена JWT-аутентификация (RequestAuthentication + AuthorizationPolicy).
- Интегрирован внешний IdP или Vault PKI.
- Реализована Zero Trust политика (ALLOW с default DENY).
- Внедрён мониторинг mTLS с алертами.
- Выполнено нагрузочное тестирование с оценкой оверхеда.
- Снижение: отсутствует внешняя интеграция — минус 1 балл (оценка «хорошо»).

7.4. Общие штрафы

- mTLS отключён после выполнения (режим DISABLE вместо STRICT) — минус 2 балла.
- AuthorizationPolicy разрешает всем (*) — не принимается как выполненная работа.
- Отсутствуют логи sidecar с подтверждением блокировки — минус 1 балл.
- Кластер не очищен после выполнения — минус 1 балл.

Лабораторная работа №12. Внедрение Continuous Compliance

1. Цель

Освоить практику «Compliance as Code» для обеспечения непрерывного соответствия инфраструктуры корпоративным политикам безопасности. Научиться разворачивать OPA Gatekeeper в Kubernetes, разрабатывать кастомные политики на Rego, внедрять их как ConstraintTemplate и Constraint, и проверять блокировку несоответствующих ресурсов на этапе создания.

2. Задачи

1. Развернуть OPA Gatekeeper в тестовом Kubernetes-кластере (kind/minikube) как Admission Controller.
2. Разработать ConstraintTemplate на языке Rego, реализующий корпоративное требование безопасности (например, обязательное наличие метки policy, запрет привилегированных подов, запрет образов с тегом latest).
3. Создать Constraint — экземпляр политики с конкретными параметрами (например, список разрешённых реестров образов).
4. Провести модульное тестирование политик Rego через gator test или opa test.
5. Продемонстрировать блокировку создания пода/ресурса, нарушающего политику, и успешное создание соответствующего ресурса.
6. Внедрить аудит существующих ресурсов на соответствие политикам.

3. Теоретическое обоснование

Проблема дрейфа конфигураций в Kubernetes. В отличие от статического анализа манифестов до деплоя (Checkov, tfsec — ЛР №5) и пост-деплой аудита (Prowler/ScoutSuite — ЛР №9), Continuous Compliance решает задачу *непрерывного контроля*: политики проверяются в реальном времени при каждом изменении ресурса (создание, обновление, удаление) через механизм Admission Webhooks. Это исключает возможность временного существования несоответствующих ресурсов.

OPA Gatekeeper — проект CNCF, реализующий Kubernetes Admission Controller на базе Open Policy Agent (OPA). Gatekeeper расширяет стандартный OPA, вводя концепцию ConstraintTemplate и Constraint как Kubernetes Custom Resources, что позволяет управлять политиками через привычный kubectl и GitOps-инструменты.

Ключевые компоненты Gatekeeper:

- ConstraintTemplate — шаблон политики на языке Rego. Определяет:

- Логике проверки (violation rules);
- Параметры, которые можно настраивать в Constraint (через parameters);
- Схему входных данных (какой ресурс проверяется).
- Constraint — экземпляр ConstraintTemplate с конкретными значениями параметров. Например, ConstraintTemplate K8sRequiredLabels с параметром labels: ["owner", "cost-center"] становится Constraint'ом RequireOwnerLabel.
- Audit — Gatekeeper периодически сканирует все существующие ресурсы и выявляет нарушения, записывая их в status.violations Constraint'a. Это позволяет обнаружить ресурсы, созданные до внедрения политики.
- Enforcement Action — действие при нарушении:
 - deny — блокировать создание/обновление ресурса (по умолчанию);
 - dryrun — логировать нарушение, но не блокировать (режим наблюдения);
 - warn — возвращать warning, но разрешать.

Связь с изученными технологиями:

- ЛР №6 (OPA/Rego): Gatekeeper использует тот же язык Rego, что и OPA для Terraform, но адаптирует его к Kubernetes API.
- ЛР №4 (Pod Security): Pod Security Admission (PSA) — встроенный механизм Kubernetes; Gatekeeper — гибкая альтернатива с кастомными политиками.
- ЛР №7 (Cloud Custodian): Custodian работает на уровне облачных ресурсов (Compute, Storage), Gatekeeper — на уровне Kubernetes-объектов.

Связь с компетенцией ПК-8:

- ИД-8пк-8 (целевой индикатор): *Обеспечивает непрерывное соответствие (continuous compliance) систем требованиям стандартов безопасности и регуляторным нормам через автоматизированный сбор доказательств и генерацию отчетов.* Gatekeeper реализует этот индикатор для Kubernetes-уровня.
- ИД-5пк-8: ConstraintTemplate — это IaC-политика, которая сама подлежит статическому анализу и тестированию.
- ИД-7пк-8: Audit-режим Gatekeeper — рантайм-мониторинг состояния контейнеров и инфраструктуры.

4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

1. Развернуть Gatekeeper. Создать ConstraintTemplate и Constraint, требующий наличия метки policy: compliant на всех подах.
2. Политика: запретить поды с hostNetwork: true.
3. Политика: запретить поды с privileged: true в securityContext.
4. Политика: требовать указания runAsNonRoot: true во всех контейнерах.
5. Политика: запретить поды без resource.limits и resource.requests.
6. Политика: разрешить образы только из реестра docker.io/library/* (запрет внешних).

7. Политика: запретить использование emptyDir volume.
8. Политика: требовать наличия метки environment со значением из списка dev/staging/prod.
9. Политика: запретить поды в namespace default (все рабочие нагрузки должны быть в именованных NS).
10. Политика: запретить Service типа LoadBalancer.
11. Политика: требовать указания seccompProfile: RuntimeDefault.
12. Политика: запретить контейнеры с allowPrivilegeEscalation: true.
13. Тестирование политик через gator test (минимум 2 compliant, 2 non-compliant сценария).
14. Политика: требовать наличия метки cost-center с форматом CC-\d{3} (regex).
15. Аудит существующих ресурсов: найти все поды, нарушающие политику обязательных меток.

4.2. Средний уровень (15 вариантов)

1. Политика: запретить Ingress без указания TLS.
2. Политика: ограничить количество реплик в Deployment (max 10).
3. Политика: запретить удаление Namespace с меткой protected: true.
4. Политика с external data: проверка образа через внешний API (Trivy scan).
5. Создать мульти-ConstraintTemplate (один шаблон → несколько Constraint с разными параметрами).
6. Политика: запретить ConfigMap с ключами, содержащими password, secret, token.
7. Интеграция Gatekeeper с OPA через http.send() в Rego (запрос к внешнему сервису при валидации).
8. Внедрение Gatekeeper в CI/CD (пайплайн проверяет ConstraintTemplate перед деплоем).
9. Политика: все поды должны иметь readiness и liveness probes.
10. Политика: запретить использование дефолтного ServiceAccount (должен быть явно указан).
11. Dry-run политики перед enforcement (сбор статистики, затем включение deny).
12. Политика: разрешить только определённые storageClasses в PersistentVolumeClaim.
13. Политика с исключениями: запрет latest, кроме namespace dev и experiments.
14. Интеграция с Falco: при нарушении Gatekeeper отправляет событие в Falco Sidekick.
15. Кастомный мониторинг: Grafana-дашборд по метрикам Gatekeeper.

4.3. Повышенный уровень (15 вариантов)

1. Разработать библиотеку из 5+ ConstraintTemplate, покрывающих CIS Kubernetes Benchmark.

2. Интеграция Gatekeeper с Kyverno: сравнение двух движков политик, миграция политик между ними.
3. Реализовать Mutation Policy (Assign/Mutate): автоматическое добавление меток при создании ресурсов.
4. Разработка оператора для автоматического создания Constraint на основе аннотаций Namespace.
5. Интеграция с HashiCorp Vault: Gatekeeper проверяет подпись образа через Vault Transit Engine.
6. Multi-cluster Gatekeeper: синхронизация политик между кластерами через GitOps.
7. Реализация политики с учётом времени: запрет деплоя в пятницу после 18:00 (time-based policy).
8. Интеграция с системой ServiceNow: автоматическое создание Change Request при попытке нарушения.
9. Реализация самообслуживания: разработчики создают Constraint через Self-Service Portal (Backstage plugin).
10. A/B тестирование политик: канареечное применение новых версий ConstraintTemplate.
11. Реализация Gatekeeper External Data Provider на Python для интеграции с корпоративным CMDB.
12. Интеграция с Sigstore/Cosign: проверка подписи образов через Gatekeeper.
13. Полный Compliance Pipeline: Rego Policy → gator test → Git PR Review → Gatekeeper Apply → Audit → Dashboard.
14. Интеграция Gatekeeper с OPA для принятия решений за пределами Kubernetes (Terraform, Envoy).
15. Разработка веб-интерфейса для управления политиками (визуальный редактор Constraint).

5. Контрольные вопросы

1. В чём различие между ValidatingAdmissionPolicy (Kubernetes 1.26+, CEL) и Gatekeeper (Rego)? Какие преимущества даёт Rego перед CEL для сложных политик?
2. Объясните концепцию «Separation of Concerns» в Gatekeeper: почему политика разделена на ConstraintTemplate и Constraint? Какую роль играет каждый из этих ресурсов?
3. Как работает механизм аудита в Gatekeeper? Каким образом он обнаруживает ресурсы, созданные до внедрения политики?
4. Какие метрики Gatekeeper критичны для мониторинга? Как они помогают в управлении инцидентами безопасности?
5. Сравните подходы Gatekeeper (Admission Control) и Cloud Custodian (Post-deployment Scanning). В каких сценариях каждый из них предпочтительнее?

6. Требования к отчету

Отчет в формате PDF. Обязательные разделы:

1. Титульный лист.

2. Архитектура Continuous Compliance: диаграмма (draw.io) компонентов (API Server → Gatekeeper Webhook → OPA → Audit).
3. Установка Gatekeeper:
 - Команды установки (Helm).
 - Вывод `kubectl get pods -n gatekeeper-system`.
 - Скриншот `ValidatingWebhookConfiguration`.
4. Разработанные политики:
 - Полный листинг `ConstraintTemplate` (с комментариями к Rego-правилам).
 - Листинг `Constraint` с параметрами.
5. Тестирование политик:
 - Листинг `suite.yaml` для gator.
 - Вывод `gator test` с PASS.
 - Скриншот создания compliant-пода.
 - Скриншот ошибки создания non-compliant-пода (403 Forbidden).
6. Аудит существующих ресурсов:
 - Вывод `kubectl describe constraint` с violations.
 - Таблица нарушений (ресурс, причина, дата обнаружения).
7. Заключение: сравнительный анализ Gatekeeper и Pod Security Admission.

7. Критерии оценивания

7.1. Базовый уровень («удовлетворительно»)

- Gatekeeper установлен и работает.
- Создан минимум 1 `ConstraintTemplate` с политикой Rego.
- Создан минимум 1 `Constraint`.
- Продемонстрирована блокировка non-compliant-пода (скриншот ошибки).
- Продемонстрирован успешный запуск compliant-пода.
- Политика покрывает хотя бы один тип нарушения (отсутствие метки, latest tag, privileged).
- Снижение: политика не блокирует под (ошибка в Rego) — минус 2 балла, доработка.

7.2. Средний уровень («хорошо»)

- Выполнены все критерии базового уровня.
- Создано минимум 2 `ConstraintTemplate` (разные типы политик: метки + безопасность).
- Настроено тестирование через `gator test` (скриншот PASS).
- Выполнен аудит существующих ресурсов (выявлены violations, если есть).
- Политики покрывают edge-кейсы (`initContainers`, несколько контейнеров).

- Снижение: отсутствуют gator-тесты — минус 2 балла.

7.3. Повышенный уровень («отлично»)

- Выполнены все критерии среднего уровня.
- Реализован Mutation Policy (автоматическое добавление меток/securityContext).
- Внедрена интеграция с внешним сервисом (External Data Provider или http.send()).
- Настроен мониторинг Gatekeeper (Prometheus + Grafana-дашборд).
- Политики покрывают минимум 3 разных аспекта (labels, security, images, resources).
- Внедрена полная CI/CD-интеграция с GitOps (ArgoCD/Flux).
- Снижение: отсутствует Mutation Policy — минус 1 балл (оценка «хорошо»).

7.4. Общие штрафы

- Gatekeeper удалён до проверки преподавателем (нечего проверять) — минус 2 балла.
- Политика содержит violation только с пустым msg (неинформативно) — минус 1 балл.
- Отсутствует очистка кластера (ресурсы оставлены) — минус 1 балл.
- Non-compliant под успешно создаётся (политика не работает) — работа не принимается.

Лабораторная работа №13. Безопасное резервное копирование в гибридном облаке

1. Цель

Освоить практические методы организации защищенного резервного копирования данных в облачное объектное хранилище. Научиться реализовывать клиентское шифрование (client-side encryption) перед передачей данных в облако, настраивать политики жизненного цикла объектов и механизмы защиты от случайного или злонамеренного удаления резервных копий.

2. Задачи

1. Создать и настроить S3-совместимый бакет в Yandex Cloud Object Storage с включенным серверным шифрованием и политикой жизненного цикла.

2. Разработать Python-скрипт резервного копирования, реализующий шифрование данных на стороне клиента (AES-256-GCM) перед отправкой в объектное хранилище.
3. Реализовать механизм безопасного хранения ключа шифрования с интеграцией HashiCorp Vault или локального KMS.
4. Настроить политику хранения (lifecycle rule) для автоматического перехода старых резервных копий в холодное хранилище и их удаления по истечении срока.
5. Включить механизм MFA Delete (или его аналог) для защиты от случайного удаления объектов.
6. Разработать и документировать процедуру восстановления данных из резервной копии.

3. Теоретическое обоснование

Модель разделенной ответственности в облаке подразумевает, что провайдер отвечает за безопасность облака (физическая инфраструктура, гипервизор), а клиент — за безопасность в облаке (данные, доступы, шифрование). Резервное копирование находится на стыке: провайдер гарантирует доступность и сохранность объекта после его загрузки, но клиент отвечает за конфиденциальность данных до их попадания в облако и за управление ключами шифрования.

Почему только серверного шифрования недостаточно?

- Server-Side Encryption (SSE): Данные шифруются на стороне облачного провайдера после получения. Ключ шифрования контролируется провайдером (SSE-S3), клиентом через облачный KMS (SSE-KMS) или клиентом через собственный ключ (SSE-C). При этом данные передаются по защищённому TLS-каналу, но на короткий момент существуют в открытом виде на стороне сервера до шифрования.
- Client-Side Encryption (CSE): Данные шифруются ДО отправки в облако. Ключ никогда не покидает контролируемую среду клиента. Даже при компрометации облачного провайдера (утечка данных на стороне хранилища) злоумышленник получит только зашифрованные блобы. Это критически важно для резервных копий, содержащих персональные данные, коммерческую тайну, финансовую информацию.

Ключевые криптографические концепции:

- AES-256-GCM — симметричный алгоритм шифрования, обеспечивающий одновременно конфиденциальность и аутентификацию (Authenticated Encryption with Associated Data, AEAD). GCM-режим добавляет Authentication Tag, позволяющий обнаружить подделку или повреждение зашифрованных данных.
- Ключ шифрования данных (DEK, Data Encryption Key) — генерируется случайно для каждой операции (file-level encryption) или сессии (session-level encryption).
- Мастер-ключ (KEK, Key Encryption Key) — используется для шифрования DEK перед сохранением. Хранится в Vault/KMS.
- Envelope Encryption — паттерн: DEK шифрует данные, KEK шифрует DEK. Зашифрованный DEK сохраняется вместе с данными (в метаданных объекта), что позволяет безопасно хранить ключи и управлять доступом.

Политики жизненного цикла (Lifecycle Rules):

- Transition — автоматический переход объектов между классами хранения:
 - STANDARD (горячее) → COLD (холодное) через N дней;
 - COLD → ICE (ледяное) через M дней.

- Expiration — автоматическое удаление объектов старше N дней.
- Noncurrent Version Expiration — удаление старых версий объектов (при включенном versioning).
- Abort Incomplete Multipart Upload — удаление незавершённых составных загрузок.

Защита от удаления:

- Versioning — при включении, удаление объекта создаёт delete marker, но не удаляет предыдущие версии.
- Object Lock (WORM, Write Once Read Many) — запрещает удаление и перезапись объекта в течение заданного периода:
 - Governance mode — требуется специальное разрешение для удаления;
 - Compliance mode — удаление невозможно никому, включая root-аккаунт.
- MFA Delete — для удаления объекта или отключения versioning требуется подтверждение через MFA-токен.

Связь с компетенцией ПК-8:

- ИД-5пк-8: Конфигурация бакета и политик как код (Terraform) подлежит статическому анализу.
- ИД-7пк-8: Мониторинг целостности резервных копий (checksum verification).
- ИД-8пк-8: Автоматизация резервного копирования как часть непрерывного соответствия требованиям (регуляторы требуют наличия резервных копий и процедур восстановления).

4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

1. Создать S3-бакет через Terraform с включенным versioning и серверным шифрованием.
2. Написать Python-скрипт, загружающий файл в бакет с использованием ServerSideEncryption: aws:kms.
3. Реализовать lifecycle rule: переход в COLD через 30 дней, удаление через 180.
4. Настроить статический веб-сайт на бакете НЕЛЬЗЯ (запретить публичный доступ).
5. Реализовать загрузку с client-side шифрованием (AES-256-CBC, библиотека cryptography).
6. Создать скрипт восстановления: скачать, расшифровать, проверить SHA-256.
7. Настроить IAM-политику, разрешающую доступ к бакету только с определённого IP.
8. Реализовать хранение ключа шифрования в переменной окружения (не хардкод).
9. Добавить метаданные к объекту: дата бэкапа, хеш, тип шифрования.

10. Написать скрипт, листинга содержимого бакета с фильтром по префиксу.
11. Настроить lifecycle rule для удаления незавершённых multipart uploads.
12. Реализовать проверку целостности бэкапа через head_object.
13. Создать файл RESTORE_PROCEDURE.md с пошаговой инструкцией.
14. Настроить бакет с Object Lock на 30 дней.
15. Сравнить размер файла до и после шифрования (таблица).

4.2. Средний уровень (15 вариантов)

1. Реализовать Envelope Encryption: DEK шифрует данные, KEK шифрует DEK.
2. Интегрировать Vault: получение ключа шифрования через AppRole.
3. Загрузка с многопоточностью (multipart upload) для файлов > 1 GB.
4. Версионирование объектов: реализовать point-in-time recovery (восстановление на определённую дату).
5. Интеграция с Prometheus: метрика backup_success{filename="..."} 1/0.
6. Реализовать инкрементальный бэкап (rsync-подобный, только изменённые блоки).
7. Сжатие данных (zstd/lzma) перед шифрованием.
8. Автоматическая ротация ключей: раз в месяц новый KEK, перешифрование DEK.
9. Мультирегиональное копирование: репликация бакета в другой регион.
10. Интеграция с CloudTrail/Audit Trail: логирование всех операций с бэкапами.
11. Тестирование восстановления: еженедельный автоматический restore test.
12. Шифрование имён файлов (в дополнение к содержимому).
13. Реализовать квоты на бакет: запрет загрузки при превышении лимита.
14. Нотификации при завершении бэкапа (Slack webhook).
15. Сравнение производительности: AES-256-GCM vs ChaCha20-Poly1305.

4.3. Повышенный уровень (15 вариантов)

1. Реализовать полноценную систему резервного копирования БД PostgreSQL с WAL-архивированием в S3.
2. Разработать Kubernetes Operator для автоматического бэкапа PVC в S3.
3. Интеграция с HashiCorp Vault Advanced: Transit Engine для шифрования, PKI для сертификатов.
4. Реализация пороговой криптографии (Shamir's Secret Sharing) для ключа шифрования.
5. Disaster Recovery Automation: полный пайплайн восстановления из бэкапа в новом регионе.
6. Интеграция с системой управления инцидентами (PagerDuty) при failure бэкапа.
7. Разработка веб-интерфейса самообслуживания для восстановления файлов.

8. Реализация WORM-хранилища поверх S3 с неизменяемостью (immutable backups).
9. Федеративное управление ключами: интеграция с несколькими KMS (облачный + on-prem).
10. Content-Addressable Storage: дедупликация на стороне клиента (как restic/borg).
11. Реализация ransomware protection: бэкапы в Immutable S3, Air-Gap копии.
12. Compliance-дашборд: SLA бэкапов, Recovery Point Objective (RPO), Recovery Time Objective (RTO).
13. Мультиоблачный бэкап: данные реплицируются в AWS S3 и Yandex Object Storage.
14. Постквантовое шифрование: использование CRYSTALS-Kyber для key encapsulation.
15. Разработка Disaster Recovery Plan с автоматическим восстановлением через Chaos Engineering (Gremlin/Chaos Mesh).

5. Контрольные вопросы

1. В чем разница между server-side encryption (SSE-KMS) и client-side encryption (CSE)? В каком сценарии CSE критически необходим?
2. Объясните принцип Envelope Encryption. Почему не рекомендуется шифровать все данные напрямую мастер-ключом?
3. Как работает режим GCM в AES? Какие три свойства безопасности он обеспечивает?
4. Какие механизмы защиты от случайного или злонамеренного удаления резервных копий существуют в S3-совместимых хранилищах?
5. Какие метрики (SLI/SLO) необходимо отслеживать для обеспечения надёжности системы резервного копирования?

6. Требования к отчету

Отчет в формате PDF. Обязательные разделы:

1. Титульный лист.
2. Архитектура системы бэкапов: диаграмма (draw.io): Source Data → Client-Side Encryption → S3 Bucket → KMS → Lifecycle → Recovery.
3. Конфигурация инфраструктуры:
 - Terraform-листинг бакета с политиками (versioning, encryption, lifecycle, object lock).
 - Скриншот настроек бакета из консоли.
4. Скрипт резервного копирования:
 - Полный листинг backup.py с комментариями.
 - Объяснение криптографических операций.
5. Демонстрация работы:
 - Скриншот успешной загрузки (лог backup.py).
 - Скриншот содержимого бакета (через list).

- Скриншот успешного восстановления с проверкой SHA-256.
- 6. Процедура восстановления:
 - RESTORE_PROCEDURE.md (полностью).
 - Расчёт RTO / RPO для сценария полной потери данных.
- 7. Заключение: оценка надёжности и безопасности реализованной системы.

7. Критерии оценивания

7.1. Базовый уровень («удовлетворительно»)

- Создан S3-бакет с включенным versioning и серверным шифрованием.
- Написан скрипт, загружающий файл с server-side encryption.
- Реализован скрипт восстановления.
- Настроена минимум 1 lifecycle rule.
- В отчёте присутствует процедура восстановления.
- Снижение: ключ шифрования захардкожен в скрипте — минус 2 балла.

7.2. Средний уровень («хорошо»)

- Выполнены все критерии базового уровня.
- Реализовано client-side шифрование (AES-256-GCM).
- Реализован Envelope Encryption (DEK + KEK).
- Интегрирован Vault для хранения мастер-ключа.
- Настроен Object Lock на бакете.
- Снижение: отсутствует Vault-интеграция — минус 2 балла.

7.3. Повышенный уровень («отлично»)

- Выполнены все критерии среднего уровня.
- Реализован инкрементальный бэкап или бэкап БД с WAL.
- Настроен автоматический restore test.
- Интегрирован мониторинг (Prometheus метрики, алерты).
- Реализована мультирегиональная репликация или мультиоблачное хранение.
- Проведён Chaos Engineering тест (фактическое восстановление из бэкапа).
- Снижение: отсутствует автоматический restore test — минус 1 балл (оценка «хорошо»).

7.4. Общие штрафы

- Ключи шифрования сохранены в репозитории — работа не принимается, пересдача.
- Бакет публично доступен (acl = "public-read") — работа не принимается.
- Отсутствует проверка целостности (SHA-256 mismatch не обнаружен) — минус 2 балла.

Лабораторная работа №14. Логирование и мониторинг безопасности

1. Цель

Освоить практические методы построения централизованной системы сбора, анализа и мониторинга логов безопасности в гибридной облачной среде. Научиться настраивать аудит-трейлы облачных провайдеров, интегрировать логи с SIEM-системой, создавать дашборды безопасности и настраивать алерты на подозрительную активность.

2. Задачи

1. Включить и настроить облачный аудит-трейл (CloudTrail / Audit Trails) для регистрации всех API-вызовов и событий безопасности.
2. Настроить сбор логов на уровне приложений и контейнеров (stdout/stderr логирование, аудит Kubernetes API).
3. Развернуть ELK-стек (Elasticsearch + Logstash + Kibana) или использовать Managed SIEM для централизованного хранения логов.
4. Создать дашборд безопасности в Kibana/Grafana с визуализацией ключевых метрик.
5. Настроить алерты на критические события безопасности (несанкционированный вход в консоль, изменение Security Groups, удаление трейлов).

3. Теоретическое обоснование

Модель трехуровневого логирования безопасности в облачных средах:

- Уровень провайдера (Cloud Provider Audit): CloudTrail (AWS), Audit Trails (Yandex Cloud), Activity Log (Azure) — регистрируют каждый API-вызов к облачному провайдеру. Фиксируют: кто, когда, откуда, какое действие выполнил, с каким результатом. Это фундамент для расследования инцидентов и compliance-аудита.
- Уровень платформы (Platform Audit): Kubernetes Audit Logs — запись всех обращений к API-серверу кластера. Фиксирует создание/удаление подов, изменение ConfigMaps/Secrets, попытки повышения привилегий.
- Уровень приложений (Application Logs): Логи самих сервисов — аутентификация, авторизация, ошибки доступа, подозрительная активность. Включает логи веб-серверов (access.log), баз данных (pgAudit), контейнеров (stdout/stderr).

Ключевые события безопасности, требующие мониторинга (по CIS/PCI DSS):

Категория	Событие	Severity	Реакция
IAM	Console Login без MFA	High	Проверка, уведомление
IAM	Root-пользователь вошёл в консоль	Critical	Немедленная проверка
IAM	Создание/удаление IAM пользователя	High	Подтверждение авторизации
Security Group	Изменение правил (authorize/revoke)	High	Проверка изменений
Security Group	Добавление 0.0.0.0/0	Critical	Автоматический откат
Storage	Изменение Bucket Policy (публичный доступ)	Critical	Блокировка, алерт
Audit Trail	Остановка/удаление трейла	Critical	Немедленное расследование
Network	Создание/изменение VPC Peering	Medium	Логирование
KMS	Доступ к ключу шифрования	High	Аудит
Compute	Запуск/остановка инстанса	Low	Информационное

Стандарты форматирования логов:

- ECS (Elastic Common Schema): Унифицированная схема полей для Elasticsearch.

Поля: @timestamp, event.action, event.outcome, user.name, source.ip, cloud.provider.

- CEF (Common Event Format): Стандарт ArcSight для SIEM. Формат: CEF:0|Vendor|Product|Version|Signature ID|Name|Severity|Extension.

- LECEF (Logstash CEF Codec): Плагин для преобразования произвольных логов в CEF.

Связь с компетенцией ПК-8:

- ИД-7пк-8: Мониторит состояние безопасности контейнеров и инфраструктуры в рантайме, выявляя инциденты и аномальное

поведение. Централизованный сбор и анализ логов — ключевой компонент рантайм-мониторинга.

- ИД-8пк-8: Логи аудита как доказательства соответствия регуляторным требованиям (compliance evidence).

Нормативные требования к логированию:

- PCI DSS 10: Логирование и мониторинг доступа к данным держателей карт.
- CIS AWS 3.x / Yandex Cloud 3.x: Включение CloudTrail/Audit Trails во всех регионах, логирование S3, защита логов от удаления.
- GDPR Art. 30: Записи о процессах обработки персональных данных.
- ФСТЭК (Россия): Регистрация событий безопасности (РД СБТ).

4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

1. Включить Audit Trails в Yandex Cloud (или CloudTrail в AWS). Подтвердить запись логов в бакет.
2. Развернуть локальный ELK (Docker Compose) и настроить приём логов.
3. Написать Logstash-конфигурацию для парсинга JSON-логов Audit Trails.
4. Создать Kibana Index Pattern для индекса audit-trails-.*.
5. Настроить Filebeat для отправки логов доступа NGINX в Elasticsearch.
6. Реализовать поиск по логам: найти все события ConsoleLogin за последние 24 часа.
7. Настроить сбор логов Kubernetes API Server (Audit Policy) в кластер.
8. Написать Python-скрипт, который подключается к ES и выводит топ-10 IP-адресов.
9. Создать простой дашборд в Kibana с метрикой "API Calls per User".
10. Включить логирование на уровне пода (stdout → Fluentd → ES).
11. Настроить алерт в Kibana Watcher на создание IAM-пользователя.
12. Проанализировать логи и найти попытки доступа с необычных геолокаций.
13. Настроить Index Lifecycle Management (ILM) для логов: hot (7d) → warm (30d) → delete (90d).
14. Реализовать визуализацию "Failed vs Successful Logins" (Kibana Lens).
15. Создать алерт при обнаружении события DeleteTrail (остановка аудита).

4.2. Средний уровень (15 вариантов)

1. Настроить полноценный SIEM-дашборд безопасности (минимум 6 визуализаций).
2. Реализовать обогащение логов: GeoIP, threat intelligence через MaxMind/AbuseIPDB.
3. Интеграция с Slack: алерты о CRITICAL-событиях в реальном времени.
4. Написать кастомный Logstash-фильтр для классификации событий по MITRE ATT&CK.

5. Настроить алерт на brute-force атаку (5+ failed logins за 1 минуту).
6. Реализовать сбор логов из нескольких источников (CloudTrail + Kubernetes + Application).
7. Мониторинг целостности логов: проверка, что трейл не был остановлен.
8. Интеграция с Yandex Managed ELK (облачный Elasticsearch).
9. Настройка аудита изменений Security Groups через AWS Config / Yandex Audit Trails.
10. Реализовать anomaly detection в Kibana: выбросы по количеству API-вызовов.
11. Создание алерта на создание публичного S3-бакета.
12. Настройка RBAC в Kibana (роли: viewer, analyst, admin).
13. Экспорт дашборда как код (Terraform / Kibana Saved Objects API).
14. Интеграция с системой тикетов (Jira) при обнаружении инцидента.
15. Мониторинг здоровья самого ELK (Elasticsearch cluster health, Logstash pipeline).

4.3. Повышенный уровень (15 вариантов)

1. Построение полноценного SOC-дашборда с интеграцией threat intelligence (MISP, OTX).
2. Реализация Machine Learning для обнаружения аномалий (Elastic ML Jobs).
3. Интеграция с SOAR (Shuffle/n8n): автоматический playbook при инциденте.
4. Multi-cloud логирование (AWS CloudTrail + Yandex Audit Trails) в единый SIEM.
5. Разработка кастомного Elasticsearch Ingest Pipeline для нормализации логов в ECS.
6. Реализация Retention Policy в соответствии с требованиями PCI DSS / GDPR.
7. Интеграция с HashiCorp Vault для безопасного хранения учётных данных ES.
8. Построение системы корреляции событий (Correlation Engine) на базе ElastAlert 2.
9. Разработка кастомного Kibana плагина для визуализации MITRE ATT&CK Matrix.
10. Автоматическая изоляция скомпрометированного ресурса при обнаружении (AWS Lambda / Cloud Function).
11. Интеграция с системой DLP для обнаружения утечек данных в логах.
12. Real-time потоковая обработка логов через Kafka перед Elasticsearch.
13. Построение Threat Hunting дашборда с предобученными детектами.
14. Интеграция с системой управления уязвимостями (DefectDojo) для enrichment логов.
15. Разработка собственного SIEM-агента на Rust/Go для сбора логов с Windows/Linux.

5. Контрольные вопросы

1. Какие события безопасности в облачной среде требуют немедленного алерта (severity CRITICAL)? Приведите 5 примеров с обоснованием.
2. Объясните архитектуру ELK-стека в контексте SIEM. Как обеспечить отказоустойчивость и сохранность логов?
3. Что такое ECS (Elastic Common Schema) и почему стандартизация полей логов критически важна для SIEM?
4. Какие существуют способы защиты логов аудита от подделки и удаления? Как доказать целостность логов при расследовании инцидента?
5. Опишите процесс Threat Hunting на основе логов: как аналитик может обнаружить скрытую активность злоумышленника, не детектируемую стандартными алертами?

6. Требования к отчету

Отчет в формате PDF. Обязательные разделы:

1. Титульный лист.
2. Архитектура системы логирования: диаграмма (draw.io) потоков данных от источников до Kibana/Grafana.
3. Настройка аудит-трейлов:
 - Скриншоты конфигурации CloudTrail / Audit Trails.
 - Команды Terraform/CLI для создания трейла.
4. Развертывание ELK:
 - Docker Compose файл (фрагменты).
 - Logstash pipeline (основные правила).
5. Дашборд безопасности:
 - Скриншот общего дашборда (общий вид).
 - Скриншоты отдельных панелей (Failed Logins, Top Users, Geo Map).
 - JSON-экспорт дашборда (в приложении).
6. Настройка алертов:
 - Листинг Watcher-алерта на ConsoleLogin.
 - Скриншот сработавшего алерта (email/Slack).
7. Анализ логов:
 - Вывод скрипта analyze_security_logs.py.
 - Примеры найденных аномалий.
8. Заключение: maturity-оценка текущего уровня мониторинга безопасности.

7. Критерии оценивания

7.1. Базовый уровень («удовлетворительно»)

- Включен аудит-трейл облачного провайдера (скриншот).
- Развернут ELK-стек (минимум Elasticsearch + Kibana).

- Настроен приём логов (Index Pattern создан).
- Создан хотя бы один простой дашборд (2-3 панели).
- Настроен хотя бы 1 алерт (ConsoleLogin).
- Снижение: дашборд не отражает security-метрики — минус 1 балл.

7.2. Средний уровень («хорошо»)

- Выполнены все критерии базового уровня.
- Настроено обогащение логов (GeoIP, severity classification).
- Дашборд содержит минимум 6 визуализаций.
- Настроены минимум 3 алерта на разные события.
- Реализована интеграция с мессенджером/почтой для алертов.
- Снижение: отсутствует обогащение логов — минус 2 балла.

7.3. Повышенный уровень («отлично»)

- Выполнены все критерии среднего уровня.
- Внедрён Machine Learning для anomaly detection.
- Реализована multi-cloud интеграция логов.
- Настроена корреляция событий (сложные правила).
- Построен SOC-дашборд с MITRE ATT&CK mapping.
- Разработан кастомный пайплайн/скрипт для автоматизации.
- Снижение: отсутствует ML/anomaly detection — минус 1 балл (оценка «хорошо»).

7.4. Общие штрафы

- Логи хранятся без шифрования (ES без xpack.security) — минус 2 балла.
- Алерты не срабатывают (некорректный watcher) — минус 1 балл.
- Индекс логирования не настроен (логи пишутся в один бесконечный индекс) — минус 1 балл.

Лабораторная работа №15. Балансировка нагрузки и защита от атак типа «Thundering Herd»

1. Цель

Изучить принципы безопасной настройки облачных балансировщиков нагрузки (ALB/ELB/NLB) и механизмы защиты бэкенд-систем от лавинообразных отказов (Thundering Herd). Освоить методики нагрузочного тестирования, анализа поведения системы при инвалидации кэша и внедрения защитных паттернов (Circuit Breaker, Request Coalescing, Jitter, Rate Limiting).

2. Задачи

1. Спроектировать и развернуть трёхуровневую архитектуру: балансировщик нагрузки → пул приложений → база данных с кэширующим слоем (Redis).

2. Настроить Application Load Balancer (ALB/ELB) с health checks, sticky sessions и защитой от DDoS (AWS Shield / WAF rate limiting).
3. Разработать тестовое приложение, эмулирующее дорогой запрос к БД с кэшированием результата.
4. Провести нагрузочное тестирование с помощью k6 в трёх сценариях: нормальный режим, инвалидация кэша без защиты, инвалидация кэша с защитными механизмами.
5. Внедрить защитные паттерны (Circuit Breaker, Request Coalescing, Exponential Backoff with Jitter) и измерить их эффективность.
6. Проанализировать результаты, построить графики зависимости latency/throughput/error rate от нагрузки.

3. Теоретическое обоснование

Проблема «Thundering Herd» (эффект лавины) — ситуация, при которой множество клиентов одновременно пытаются получить доступ к ресурсу, который временно недоступен или аннулирован. Классический сценарий: истекает срок действия популярного элемента кэша, и сотни параллельных запросов одновременно направляются к базе данных, создавая пиковую нагрузку, превышающую её пропускную способность. Результат — каскадный отказ: БД перегружается → запросы зависают → занимают все соединения пула → балансировщик помечает бэкенд как unhealthy → полный отказ сервиса.

Ключевые механизмы защиты:

1. Request Coalescing (Слияние запросов)
Вместо отправки N параллельных запросов к БД, только один запрос выполняется, а остальные ожидают его результата. Реализуется через:

- Блокировки (Mutex/Lock): Первый поток захватывает блокировку, выполняет запрос, сохраняет в кэш, остальные ждут.
- Promise/Future Sharing: Все ожидающие запросы получают ссылку на один и тот же промис.

2. Circuit Breaker (Предохранитель)
Паттерн, предотвращающий каскадные отказы. Отслеживает количество ошибок и при превышении порога «размыкает цепь» — немедленно возвращает ошибку, не нагружая отказавший сервис. Состояния:

- Closed: Нормальная работа, запросы проходят.
- Open: Запросы блокируются, возвращается fallback-ответ.
- Half-Open: Пробный запрос для проверки восстановления сервиса.

3. Exponential Backoff with Jitter
При обнаружении перегрузки клиенты делают повторные попытки с экспоненциально растущей задержкой и случайным джиттером для размазывания нагрузки:

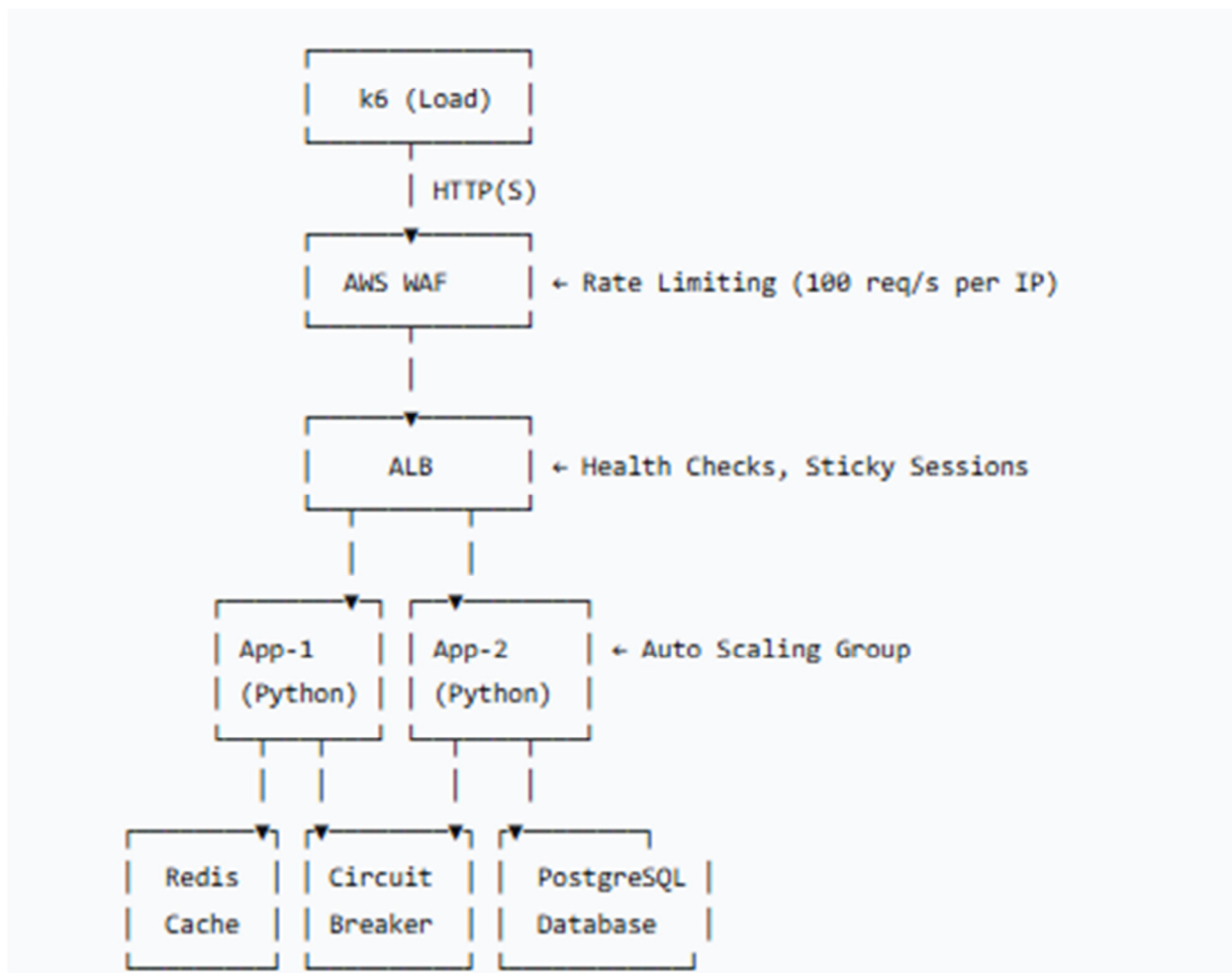
$$\text{wait_time} = \min(\text{base} * 2^{\text{attempt}}, \text{max_wait}) + \text{random}(0, \text{jitter})$$

4. Rate Limiting (Ограничение скорости)
Ограничение количества запросов в единицу времени от одного клиента/IP. Реализуется на уровне:

- WAF (AWS WAF Rate-based Rules): Блокировка IP при превышении лимита.
- API Gateway: Throttling по API-ключу.
- Application-level: Token Bucket / Leaky Bucket алгоритмы.

5. Stale Cache Serving (Обслуживание устаревшего кэша)
Вместо удаления истёкшего элемента кэша, он продолжает отдаваться клиентам, пока в фоновом режиме выполняется асинхронное обновление.

Архитектура тестового стенда:



Метрики для анализа:

- Latency (p50, p95, p99): Время ответа в миллисекундах.
- Throughput (RPS): Количество успешных запросов в секунду.
- Error Rate (%): Доля запросов, завершившихся ошибкой (5xx, timeout).
- DB Connections: Количество активных соединений с БД.
- Cache Hit Ratio (%): Доля запросов, обслуженных из кэша.

Связь с компетенцией ПК-8:

- ИД-7пк-8: Мониторинг состояния инфраструктуры в рантайме, выявление аномального поведения (резкий скачок нагрузки при падении кэша).
- ИД-6пк-8: Настройка Quality Gate в CI/CD — нагрузочное тестирование как обязательный этап перед деплоем.

4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

1. Развернуть ALB + 2 EC2 инстанса с простым приложением. Проверить health checks.

2. Реализовать кэширование в Redis с TTL 30 секунд.
3. Написать k6-скрипт для нормального режима нагрузки (50 VUs).
4. Провести тест с инвалидацией кэша БЕЗ защиты. Зафиксировать latency и ошибки.
5. Настроить WAF rate limiting (100 req/s per IP).
6. Реализовать Request Coalescing через threading.Lock.
7. Настроить Circuit Breaker (5 ошибок → размыкание на 30 секунд).
8. Построить график Latency Distribution (p50/p95/p99).
9. Сравнить два сценария: с кэшем и без кэша.
10. Добавить Stale Cache (отдача устаревших данных при отказе БД).
11. Настроить Docker Compose с PostgreSQL + Redis + приложение.
12. Реализовать Exponential Backoff в клиенте (k6).
13. Настроить алерты CloudWatch по метрикам ALB (5xx errors, latency).
14. Провести анализ логов ALB (access logs) после нагрузки.
15. Написать отчёт с выводами (минимум 10 страниц).

4.2. Средний уровень (15 вариантов)

1. Реализовать BCE защитные механизмы (Coalescing + Circuit Breaker + Stale Cache).
2. Провести сравнительное тестирование трёх сценариев (таблица + графики).
3. Настроить автоскейлинг (Auto Scaling Group) на основе метрик нагрузки.
4. Интеграция с Grafana: дашборд с метриками в реальном времени.
5. Реализовать Rate Limiting на уровне приложения (Token Bucket).
6. Настроить Redis Cluster для отказоустойчивости кэша.
7. Сравнить Circuit Breaker vs Bulkhead паттерны.
8. Интеграция с Prometheus: экспорт кастомных метрик.
9. Нагрузочное тестирование с 1000+ VUs (распределённый k6).
10. Реализовать graceful degradation (деградация функциональности при отказе БД).
11. Анализ стоимости: On-Demand vs Reserved Instances для пиковой нагрузки.
12. Сравнение ALB vs NLB для данного сценария.
13. Хаос-инжиниринг: принудительное отключение Redis во время нагрузки.
14. Написать Terraform-модуль для всей инфраструктуры.
15. Видеодемонстрация нагрузки и поведения системы.

4.3. Повышенный уровень (15 вариантов)

1. Реализовать мультирегиональную балансировку (Route53 + ALB в 2 регионах).
2. Внедрение Service Mesh (Istio) с Circuit Breaker и Retry на уровне sidecar.

3. Машинное обучение для предсказания пиков нагрузки и проактивного кэширования.
4. Реализовать Consensus-алгоритм для синхронизации кэша между регионами.
5. Разработать кастомный Kubernetes Operator для управления Circuit Breaker.
6. Интеграция с eBPF (Cilium) для анализа сетевых задержек.
7. Полный Chaos Engineering эксперимент (Gremlin/Chaos Mesh).
8. Анализ Total Cost of Ownership (TCO) разных стратегий защиты.
9. Реализация распределённого кэша с CRDT (Conflict-free Replicated Data Types).
10. Интеграция с CDN (CloudFront) для edge-кэширования.
11. A/B тестирование разных стратегий инвалидации кэша.
12. Разработка DSL для описания политик защиты нагрузки.
13. Исследование: сравнение Thundering Herd в REST vs GraphQL vs gRPC.
14. Публикация статьи/white paper по результатам исследования.
15. Разработка Open Source библиотеки для защиты от Thundering Herd.

5. Контрольные вопросы

1. Что такое эффект «Thundering Herd»? Опишите классический сценарий возникновения и механизм каскадного отказа.
2. Как работает паттерн Request Coalescing? Почему простой блокировки недостаточно для высоконагруженных систем?
3. Чем Circuit Breaker отличается от Retry? В каком порядке они должны применяться?
4. Почему Exponential Backoff необходимо дополнять Jitter? Какие проблемы решает случайная задержка?
5. Как настроить WAF rate limiting, чтобы отличать легитимную нагрузку от атаки?

6. Требования к отчету

Отчет в формате PDF. Обязательные разделы:

1. Титульный лист.
2. Архитектура стенда: диаграмма (draw.io) компонентов и потоков данных.
3. Конфигурация инфраструктуры:
 - Terraform-листинг ALB, WAF, Auto Scaling.
 - Docker Compose / Kubernetes манифесты.
4. Реализация защитных механизмов:
 - Код Request Coalescing, Circuit Breaker, Backoff.
 - Объяснение выбора параметров (threshold, timeout).
5. Нагрузочное тестирование:

- k6-скрипты для всех трёх сценариев.
 - Таблица результатов.
 - Графики latency/throughput/error rate.
6. Сравнительный анализ:
- Таблица ДО/ПОСЛЕ внедрения защиты.
 - Расчёт процента улучшения по каждой метрике.
7. Выводы и рекомендации:
- Какие механизмы дали наибольший эффект.
 - Рекомендации по внедрению в production.

7. Критерии оценивания

7.1. Базовый уровень («удовлетворительно»)

- Развёрнут ALB с health checks.
- Реализован кэширующий слой (Redis).
- Проведён нагрузочный тест без защиты (скриншоты k6).
- Зафиксированы метрики (latency, errors).
- Снижение: тестирование проведено без инвалидации кэша — минус 2 балла.

7.2. Средний уровень («хорошо»)

- Выполнены все критерии базового уровня.
- Реализованы минимум 2 защитных механизма (Coalescing, Circuit Breaker).
- Проведены 3 сценария тестирования.
- Построены сравнительные графики.
- Достигнуто снижение P99 latency > 50%.
- Снижение: Circuit Breaker не срабатывает — минус 2 балла.

7.3. Повышенный уровень («отлично»)

- Выполнены все критерии среднего уровня.
- Реализованы ВСЕ 4 защитных механизма.
- Настроено автоскейлирование на основе метрик.
- Интегрирован мониторинг (Grafana + Prometheus).
- Проведён Chaos Engineering эксперимент.
- P99 latency снижена > 80%.
- Снижение: отсутствует Chaos Engineering — минус 1 балл (оценка «хорошо»).

7.4. Общие штрафы

- Нагрузочный тест вызвал реальный отказ продуктивной системы — работа не принимается.

- Отсутствуют скриншоты кб-результатов — минус 3 балла.
- Защитные механизмы закомментированы/не работают — минус 5 баллов.

Лабораторная работа №16. Проектирование многооблачной архитектуры (AWS + Yandex Cloud)

1. Цель

Применить комплексные знания, полученные в рамках дисциплины, для проектирования отказоустойчивой и безопасной многооблачной системы. Освоить практику распределения компонентов приложения между облачными провайдерами, настройки защищённой межоблачной связности, анализа рисков на стыке облаков и расчёта совокупной стоимости владения (ТСО).

2. Задачи

1. Спроектировать архитектуру распределённого приложения с размещением Frontend-уровня в AWS и Backend + БД в Yandex Cloud.
2. Разработать Terraform-конфигурации для обоих провайдеров, реализующие единую инфраструктуру как код.
3. Настроить защищённый межоблачный канал связи (VPN WireGuard/IPSec или выделенный Interconnect) между AWS VPC и Yandex Cloud VPC.
4. Реализовать сквозную аутентификацию и авторизацию между компонентами в разных облаках (mTLS, OIDC).
5. Провести комплексный анализ рисков безопасности на стыке облаков и разработать матрицу угроз.
6. Рассчитать детализированный бюджет эксплуатации за месяц с учётом всех компонентов и каналов связи.

3. Теоретическое обоснование

Многооблачная стратегия (Multi-Cloud) — подход к построению ИТ-инфраструктуры, при котором организация использует двух или более облачных провайдеров одновременно. Ключевые мотивации:

- Отказоустойчивость (Resilience): Выход из строя одного провайдера не приводит к полной недоступности сервиса.
- Географическое распределение: Размещение компонентов ближе к конечным пользователям для снижения задержек.
- Отсутствие vendor lock-in: Возможность миграции между провайдерами.
- Best-of-breed: Использование сильных сторон каждого провайдера (CDN у AWS, управляемый PostgreSQL у Yandex Cloud).
- Compliance: Хранение данных в юрисдикции конкретной страны (152-ФЗ для РФ).

Ключевые аспекты безопасности на стыке облаков:

- Data in Transit: Весь трафик между облаками должен быть зашифрован (IPSec/WireGuard + mTLS на уровне приложений). Недопустима передача данных по публичному интернету без шифрования.
- Identity Federation: Единая система аутентификации через OIDC-федерацию (AWS IAM ↔ Yandex Cloud Federation ↔ корпоративный IdP).

- Network Segmentation: Минимально возможные CIDR-блоки для межоблачного пиринга, строгие Security Group с обеих сторон, разрешающие только необходимые порты.

- Secrets Management: Ключи VPN, сертификаты mTLS, пароли БД хранятся в HashiCorp Vault, доступном из обоих облаков.

- Audit Trail Correlation: Логи аудита из AWS CloudTrail и Yandex Cloud Audit Trails коррелируются в единой SIEM-системе.

Модель угроз для межоблачного взаимодействия (STRIDE):

Категория	Угроза	Компонент	Severity	Контрмера
spoofing	Подделка VPN-туннеля	VPN Gateway	CRITICAL	Pre-shared ключ + PKI
tampering	MITM между облаками	Inter-cloud link	CRITICAL	IPSec/WireGuard + mTLS
repudiation	Невозможность доказать действия	Cross-cloud API	HIGH	Correlation Audit Logs
information disclosure	Утечка данных в транзите	Data in Transit	CRITICAL	End-to-end encryption
denial of service	Разрыв VPN-канала	VPN Gateway	HIGH	Redundant tunnels
elevation of privilege	Компрометация одного облака → доступ ко второму	IAM Federation	HIGH	Separate credentials, no cross-cloud IAM trust

Связь с компетенцией ПК-8:

- ИД-5пк-8: IaC-конфигурации для обоих провайдеров подлежат статическому анализу Checkov/tfsec.

- ИД-7пк-8: Межоблачный мониторинг VPN-туннелей, SIEM.

- ИД-8пк-8: Единый Compliance Dashboard для многооблачной среды.

4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

1. Спроектировать архитектуру: Frontend (AWS) + Backend (YC). Нарисовать схему.
2. Написать Terraform для AWS VPC + EC2 (простой frontend).
3. Написать Terraform для Yandex Cloud VPC + Compute Instance (простой backend).

4. Настроить VPN WireGuard между облаками (manual setup).
5. Настроить Security Groups с обеих сторон, разрешающие только нужный трафик.
6. Рассчитать стоимость эксплуатации (таблица, как в примере).
7. Провести анализ рисков (минимум 5 угроз, STRIDE).
8. Задokumentировать архитектуру в README.md.
9. Настроить DNS (Route53) для маршрутизации на AWS Frontend.
10. Применить terraform plan к обеим конфигурациям (скриншот).
11. Проверить связность: ping между инстансами.
12. Настроить базовый мониторинг (CloudWatch + Yandex Monitoring).
13. Создать диаграмму в draw.io.
14. Написать Ansible-плейбук для установки VPN.
15. Провести самооценку архитектуры по CIS Benchmarks.

4.2. Средний уровень (15 вариантов)

1. Реализовать High Availability: Multi-AZ в каждом облаке.
2. Замена WireGuard на IPSec (StrongSwan) с IKEv2.
3. Интеграция с Vault для управления ключами VPN.
4. mTLS между Frontend (AWS) и Backend (YC) через Istio.
5. Единая система мониторинга: Prometheus + Grafana для обоих облаков.
6. Реализация Disaster Recovery: резервный регион в каждом облаке.
7. Интеграция CloudTrail + Audit Trails в единый SIEM.
8. Многооблачное CI/CD: GitLab CI деплоит в оба облака параллельно.
9. Автоматический failover: Route53 health check переключает трафик.
10. Сравнение стоимости: однооблачная vs многооблачная архитектура.
11. Compliance Check: проверить архитектуру на PCI DSS.
12. Реализация Network Policies (Calico/Cilium) для сегментации.
13. Chaos Engineering: тест на разрыв VPN-туннеля.
14. CDN (CloudFront) для статики + Object Storage для бэкапов.
15. Подготовка презентации архитектуры для СТО.

4.3. Повышенный уровень (15 вариантов)

1. Полная автоматизация: Terraform + Ansible + CI/CD для всей архитектуры.
2. Service Mesh между облаками (Istio multi-cluster).
3. Multi-Cloud Kubernetes Federation (KubeFed).
4. Единая система аутентификации через OIDC-федерацию.
5. Автоматическое переключение трафика между облаками (GSLB).
6. Cost Optimization с автоматическим скейлингом между облаками.

7. Разработка Threat Model (STRIDE + MITRE ATT&CK).
8. Построение Compliance Dashboard для многооблачной среды.
9. Интеграция с SOAR для автоматического реагирования на инциденты.
10. Сравнение трёх провайдеров (AWS + YC + Azure) для одной задачи.
11. Реализация Data Sovereignty: автоматическое размещение данных в нужной юрисдикции.
12. Мультиоблачная Service Catalog для разработчиков.
13. Расчет TCO на 3 года с учётом роста нагрузки.
14. Разработка оператора Kubernetes для оркестрации между облаками.
15. Подготовка статьи/white paper по многооблачной архитектуре.

5. Контрольные вопросы

1. Какие ключевые риски безопасности возникают на стыке двух облачных провайдеров? Как их митигировать?
2. Сравните WireGuard и IPSec для межоблачного VPN. Какой протокол предпочтительнее в production-среде и почему?
3. Как обеспечить сквозную аутентификацию (end-to-end) между компонентами, находящимися в разных облаках?
4. Какие финансовые и операционные риски необходимо учесть при выборе многооблачной стратегии?
5. Как организовать единую систему аудита и compliance для многооблачной инфраструктуры?

6. Требования к отчету

Отчет в формате PDF. Обязательные разделы:

1. Титульный лист.
2. Архитектурная схема: (draw.io/Visio) всех компонентов, подсетей, потоков данных.
3. Бизнес-обоснование: почему выбрана многооблачная архитектура.
4. Terraform-код: полные листинги для AWS и Yandex Cloud (в приложении).
5. Конфигурация VPN: настройки, скриншоты wg show, логи handshake.
6. Матрица рисков безопасности: STRIDE-анализ с контрмерами.
7. Расчёт стоимости: детализированная таблица + графики.
8. Результаты верификации: Checkov, Prowler, тесты связности.
9. Заключение: рекомендации по внедрению.

7. Критерии оценивания

7.1. Базовый уровень («удовлетворительно»)

- Спроектирована архитектура: схема + описание.
- Terraform-код для обоих провайдеров синтаксически корректен.
- Настроен VPN между облаками (скриншот связности).

- Рассчитан бюджет (таблица).
- Проведён базовый анализ рисков (5 угроз).
- Снижение: terraform plan содержит ошибки — минус 2 балла.

7.2. Средний уровень («хорошо»)

- Выполнены все критерии базового уровня.
- Реализован HA (Multi-AZ) в обоих облаках.
- Настроен mTLS между сервисами.
- Выполнена интеграция IAM (OIDC между облаками).
- Проведён анализ 10+ угроз с контрмерами.
- Детализированный расчёт стоимости (включая скрытые затраты).
- Снижение: отсутствует mTLS — минус 2 балла.

7.3. Повышенный уровень («отлично»)

- Выполнены все критерии среднего уровня.
- Полная автоматизация деплоя (скрипт + CI/CD).
- Реализован мультиоблачный Service Mesh.
- Единый Compliance Dashboard.
- Автоматический failover при разрыве VPN.
- Сравнение TCO с однооблачным решением.
- Снижение: отсутствует автоматический failover — минус 1 балл (оценка «хорошо»).

7.4. Общие штрафы

- Ключи VPN/БД в открытом виде в репозитории — работа не принимается.
- Отсутствует шифрование трафика между облаками — работа не принимается.
- Ресурсы не удалены после выполнения — минус 3 балла.
- Бюджет не учитывает скрытые затраты (трафик, NAT) — минус 1 балл.

Лабораторная работа №17. Реализация политик безопасности для «Bring Your Own Cloud»

1. Цель

Разработать комплексную систему политик безопасности (Policy as Code) для гибридной среды «Bring Your Own Cloud», обеспечивающую автоматическую валидацию облачных ресурсов на соответствие корпоративным требованиям безопасности, тегирования, региональных ограничений и комплаенса до их развертывания.

2. Задачи

1. Спроектировать модель разграничения ответственности (RACI-матрицу) между командами в модели BYOC.
2. Разработать библиотеку из 10+ политик на языке Rego, покрывающих ключевые аспекты безопасности: тегирование, сетевую безопасность, шифрование, IAM, соответствие регионам, управление затратами.
3. Реализовать модульные тесты для каждой политики через opa test с покрытием compliant и non-compliant сценариев.
4. Интегрировать политики в CI/CD-пайплайн как Quality Gate, блокирующий деплой Terraform-модулей, не прошедших валидацию.
5. Создать механизм exception-менеджмента (документированные исключения из политик) и панель визуализации compliance-статуса.

3. Теоретическое обоснование

Модель «Bring Your Own Cloud» (BYOC) — организационный подход, при котором бизнес-подразделения или команды разработки самостоятельно выбирают и управляют облачными ресурсами в рамках корпоративных политик безопасности. В отличие от централизованной модели (единая облачная команда), BYOC требует автоматизации контроля соответствия, так как ручной аудит не масштабируется.

Ключевые вызовы BYOC с точки зрения безопасности:

- Shadow IT: Команды создают ресурсы в обход централизованных процессов.
- Дрейф конфигураций: Отсутствие единых стандартов приводит к разным настройкам в разных командах.
- Неуправляемые затраты: Каждая команда использует свои типы инстансов без оптимизации.

- Нарушение комплаенса: Ресурсы создаются в запрещённых регионах или без обязательных тегов.
- Утечка данных: Отсутствие единых правил шифрования и контроля доступа.

Модель разграничения ответственности (RACI для BYOC):

Активность	Cloud Foundation Team	Security Team	DevOps Team	FinOps Team
Создание VPC/Network	R	A	C	I
Настройка Security Groups	A	R	C	I
Управление IAM-ролями	C	R	A	I
Шифрование хранилищ	A	R	C	I
Тегирование ресурсов	I	C	R	A
Выбор типа инстанса	C	I	R	A
Мониторинг затрат	I	C	A	R
Аудит безопасности	I	R	C	I
<i>R=Responsible, A=Accountable, C=Consulted, I=Informed</i>				

Категории политик (10+):

Категория	Политика	Severity	ID
Tagging	Все ресурсы должны иметь тег Owner	High	BYOC-TAG-001
Tagging	Тег Environment обязателен	High	BYOC-TAG-002
Tagging	Тег CostCenter с форматом CC-{4}	Medium	BYOC

	Категория	Политика	Severity	ID
	Tagging		MEDIUM	-TAG-003
	Tagging	Tag Compliance должен быть true	CRITICAL	BYOC -TAG-004
	Network	Запрет Security Group с 0.0.0.0/0 на порт 22	CRITICAL	BYOC -NET-001
	Network	Все Security Group должны иметь description	MEDIUM	BYOC -NET-002
	Compute	Запрет инстансов > 8 vCPU	MEDIUM	BYOC -CMP-001
	Compute	Все диски должны быть зашифрованы	CRITICAL	BYOC -CMP-002
	Compute	Запрет публичных IP в частных подсетях	HIGH	BYOC -CMP-003
0	IAM	Запрет IAM-пользователей с паролем (только роли)	HIGH	BYOC -IAM-001
1	Storage	Запрет публичных бакетов	CRITICAL	BYOC -STG-001
2	Region	Разрешённые регионы: ru-central1, eu-west-1	HIGH	BYOC -REG-001
3	Naming	Имена ресурсов должны соответствовать regex	LOW	BYOC -NAM-001
4	Backup	Критичные ресурсы должны иметь backup policy	HIGH	BYOC -BKP-001
5	FinOps	Запрет инстансов с core_fraction: 100	MEDIUM	BYOC -FIN-001

Связь с компетенцией ПК-8:

- ИД-5пк-8: Политики Rego валидируют Terraform-конфигурации до применения.

- ИД-8пк-8: Система политик реализует Continuous Compliance для BYOC-модели.

4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

Студенты разрабатывают 5 политик из обязательного списка и тесты к ним.

1. Написать политики BYOC-TAG-001, BYOC-TAG-002, BYOC-NET-001, BYOC-STG-001, BYOC-REG-001.
2. Набор: тегирование (3 политики) + сетевая безопасность (2 политики).
3. Набор: вычислительные ресурсы (3 политики) + IAM (2 политики).
4. Набор: хранилища (3 политики) + регионы (2 политики).
5. Политики для конкретного провайдера (только AWS).
6. Политики для конкретного провайдера (только Yandex Cloud).
7. Дополнить тесты для всех 10+ политик (полное покрытие).
8. Настроить CI/CD интеграцию (GitLab CI скриншоты).
9. Создать exception-файл с 3 документированными исключениями.
10. Написать политику naming convention (BYOC-NAM-001).
11. Написать политику backup requirements (BYOC-BKP-001).
12. Написать политику FinOps (BYOC-FIN-001: запрет core_fraction 100).
13. Сравнить результаты OPA с Checkov/tfsec.
14. Создать RACI-матрицу для BYOC.
15. Написать документацию по политикам (Policy Catalog).

4.2. Средний уровень (15 вариантов)

1. Реализовать ВСЕ 15 политик из каталога.
2. Добавить политики для Kubernetes (BYOC-K8S-001: запрет privileged).
3. Cross-resource анализ: SG может ссылаться только на ресурсы с тегом Role: database.
4. Реализовать dynamic exceptions через Vault (http.send в Rego).
5. Создать Grafana-дашборд compliance-статуса.
6. Интеграция с Slack: нотификации при CRITICAL violations.
7. Реализовать automatic waiver expiry (исключения с истекающим сроком).
8. Сравнение OPA vs Sentinel vs Checkov custom policies.
9. Политика multi-cloud naming (единый формат для AWS + YC).
10. Реализовать scoring: Compliance Score = PASS / (PASS + FAIL).
11. Интеграция с ServiceNow: автоматические Change Request при violations.
12. Канареечное внедрение политик (dry-run 30 дней, затем enforce).
13. Нагрузочное тестирование OPA на 1000+ ресурсов.

14. Разработать Self-Service Policy Portal для команд.
15. Интеграция с GitOps (ArgoCD pre-sync hook через OPA).

4.3. Повышенный уровень (15 вариантов)

1. Разработка кастомного OPA-бандла для BYOC (Bundle + Wasm compilation).
2. Реализация Policy as Code Pipeline: GitHub → OPA Test → Bundle Build → S3 → OPA Server.
3. Интеграция с корпоративным IdP (Keycloak) для RBAC на политики.
4. Разработка BYOC Policy Operator для Kubernetes.
5. Машинное обучение: предсказание non-compliant ресурсов на основе исторических данных.
6. Полная автоматизация: Terraform → OPA → Approval → Apply → Audit → Dashboard.
7. Разработка Terraform Provider для BYOC Policies.
8. Интеграция с системой Threat Intelligence (обогащение политик данными об угрозах).
9. Compliance Scoreboard для C-level (дашборд в реальном времени).
10. Разработка Decentralized Policy Governance через блокчейн (proof-of-compliance).
11. Интеграция с eBPF (Falco) для рантайм-валидации политик.
12. Разработка OPA Wasm plugin для браузерной валидации Terraform.
13. Auto-remediation: при violation автоматически создаётся PR с исправлением.
14. Сравнительный анализ: OPA vs Kyverno vs jsPolicy.
15. Построение системы Continuous Authorization (CA) на базе OPA.

5. Контрольные вопросы

1. В чём ключевые отличия модели BYOC от централизованной облачной модели? Какие риски безопасности уникальны для BYOC?
2. Объясните концепцию «Separation of Duties» в контексте BYOC. Как RACI-матрица помогает разграничить ответственность?
3. Почему exception-менеджмент критически важен для Compliance as Code? Какие механизмы предотвращают бесконтрольное накопление исключений?
4. Как масштабировать систему политик при росте числа команд и облачных аккаунтов? Какие архитектурные решения (Bundle, Wasm, Server) применимы?
5. Какие метрики эффективности системы BYOC-политик следует отслеживать? Как оценить зрелость Compliance as Code?

6. Требования к отчету

Отчет в формате PDF. Обязательные разделы:

1. Титульный лист.
2. Модель BYOC: RACI-матрица, архитектурная схема контроля, зоны ответственности.

3. Каталог политик: таблица всех 10+ политик с ID, severity, описанием.
4. Код политик Rego: полные листинги всех файлов с комментариями.
5. Модульные тесты: листинг тестов, скриншот opa test с PASS.
6. Exception Management: листинг approved-waivers.yaml, процесс согласования исключений.
7. CI/CD интеграция: .gitlab-ci.yml, скриншот блокировки пайплайна.
8. Compliance Report: HTML/PDF-отчёт, сгенерированный для тестового плана.
9. Заключение: оценка эффективности, рекомендации по внедрению BYOC-политик.

7. Критерии оценивания

7.1. Базовый уровень («удовлетворительно»)

- Разработано минимум 5 политик Rego.
- Для каждой политики есть минимум 2 теста (compliant + non-compliant).
- opa test проходит успешно.
- Политики интегрированы в CI/CD (скриншот).
- Создан exception-файл с минимум 1 документированным исключением.
- Снижение: тесты не покрывают edge-кейсы — минус 1 балл.

7.2. Средний уровень («хорошо»)

- Выполнены все критерии базового уровня.
- Разработано 10+ политик с полным тестовым покрытием.
- Реализован exception-менеджмент с автоматической проверкой сроков.
- Сгенерирован HTML-отчёт с визуализацией.
- Настроена блокировка CI/CD при CRITICAL violations.
- Снижение: exception-менеджмент не автоматизирован — минус 1 балл.

7.3. Повышенный уровень («отлично»)

- Выполнены все критерии среднего уровня.
- Реализованы cross-resource политики.
- Интегрирован dynamic exception lookup (Vault/HTTP API).
- Построен Compliance Dashboard (Grafana).
- Разработан кастомный OPA Bundle.
- Внедрён Self-Service Policy Portal или авто-ремедиация.
- Снижение: отсутствует дашборд — минус 1 балл (оценка «хорошо»).

7.4. Общие штрафы

- Политики содержат синтаксические ошибки Rego — минус 2 балла.
- opa test падает — работа не принимается.

- Исключения не имеют срока действия — минус 1 балл.
- Отсутствует документация политик — минус 1 балл.

Лабораторная работа №18. Комплексный проект: Аудит «Управление облачной платформой»

1. Цель

Выполнить полный цикл аудита безопасности облачной инфраструктуры, выявить уязвимости и несоответствия, разработать и внедрить комплекс мер по усилению безопасности, а также измерить эффективность предложенных решений через сравнение состояния «Как было» и «Как стало». Защитить проект как итоговую аттестационную работу по дисциплине.

2. Задачи

1. Провести всесторонний аудит существующей облачной инфраструктуры с использованием Prowler/ScoutSuite, Checkov/tfsec, Cloud Custodian.

2. Выявить и классифицировать уязвимости по уровням критичности (CRITICAL / HIGH / MEDIUM / LOW) и категориям (сеть, доступ, конфигурация, секреты, комплаенс).

3. Разработать набор OPA/Rego-политик для предотвращения выявленных нарушений в будущем.

4. Внедрить HashiCorp Vault для централизованного управления секретами, заменив статические ключи и хардкод-пароли.

5. Настроить непрерывный мониторинг безопасности (SIEM-дашборд, алерты) для отслеживания состояния платформы.

6. Подготовить профессиональный документ «Отчет об аудите и предложения по усилению безопасности облачной платформы» со сравнительным анализом ДО/ПОСЛЕ.

3. Теоретическое обоснование

Методология комплексного аудита безопасности базируется на стандартах и фреймворках:

- CIS Benchmarks — отраслевые стандарты безопасной конфигурации облачных сервисов.
- NIST Cybersecurity Framework — Identify → Protect → Detect → Respond → Recover.
- PCI DSS — требования к защите данных платёжных карт (применимы как best practice).
- ISO 27001 — международный стандарт системы управления информационной безопасностью.

Этапы комплексного аудита:

1. Инвентаризация (Discovery): Полный сбор информации о ресурсах, конфигурациях, доступах, сетевых правилах.
2. Сканирование (Assessment): Автоматизированная проверка через Prowler (CIS), Checkov (IaC), Trivy (образы), Cloud Custodian (runtime compliance).
3. Анализ (Analysis): Классификация находок, корреляция данных из разных источников, выявление корневых причин.
4. Приоритизация (Prioritization): Ранжирование уязвимостей по CVSS/severity, влиянию на бизнес, сложности эксплуатации.
5. Разработка мер (Remediation Design): Проектирование конкретных технических решений (OPA-политики, Vault, Security Groups, шифрование).
6. Внедрение (Implementation): Поэтапное применение мер с валидацией на каждом шаге.
7. Верификация (Verification): Повторный аудит для подтверждения устранения уязвимостей.
8. Документирование (Reporting): Формирование профессионального отчёта с выводами и рекомендациями.

Метрики эффективности (KPI Security Improvement):

Метрика	Формула	Целевое значение
Vulnerability Remediation Rate	$(\text{Исправлено} / \text{Найдено}) \times 100\%$	$\geq 90\%$
Mean Time to Remediate (MTTR)	$\Sigma(\text{Время исправления}) / N$	CRITICAL < 24h
Compliance Score	$\text{PASS} / (\text{PASS} + \text{FAIL}) \times 100\%$	$\geq$

Метрика	Формула	Целевое значение
		95%
Secret Hygiene Score	(Ресурсы без хардкод-секретов / Всего ресурсов) × 100%	100%
Security Posture Grade	A-F на основе CIS Benchmark coverage	A (≥ 90%)

Инструментарий комплексного проекта:

Этап	Инструмент	Лабораторная работа
Аудит инфраструктуры	Prowler, ScoutSuite	ЛР №9
Анализ IaC	Checkov, tfsec	ЛР №5
Runtime Compliance	Cloud Custodian	ЛР №7
Политики как код	OPA Gatekeeper	ЛР №6, №12
Управление секретами	HashiCorp Vault	ЛР №2
CI/CD безопасность	GitLab CI + OIDC	ЛР №8
Мониторинг и SIEM	ELK, Grafana	ЛР №14
Контейнерная безопасность	Trivy, PSA	ЛР №4
Сетевая безопасность	VPC, SG, VPN	ЛР №3
FinOps и бюджетирование	Infracost, Budgets	ЛР №10

Связь с компетенцией ПК-8: Комплексный проект демонстрирует освоение всех индикаторов компетенции ПК-8 (ИД-5, ИД-6, ИД-7, ИД-8) в рамках единой практической задачи.

4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

1. Аудит AWS: Prowler + Checkov. Выявить топ-10 нарушений.

2. Аудит Yandex Cloud: ScoutSuite (или эквивалент). Классификация.
3. Разработка 5 OPA-политик для предотвращения типовых нарушений.
4. Внедрение Vault: миграция 5 секретов из переменных окружения.
5. Создание дашборда безопасности в Grafana (минимум 4 панели).
6. Настройка алерта на ConsoleLogin (Kibana Watcher).
7. Исправление всех CRITICAL-нарушений. Повторный аудит.
8. Анализ IaC: Checkov на Terraform-конфигурации.
9. Сканирование Docker-образов Trivy. Устранение CRITICAL CVE.
10. Настройка бюджетов и FinOps-алертов.
11. Аудит IAM: пользователи без MFA, неиспользуемые ключи.
12. Аудит Storage: публичные бакеты, отсутствие шифрования.
13. Аудит Network: SG с 0.0.0.0/0, NACL, Flow Logs.
14. Сравнительный анализ инструментов (Prowler vs Checkov).
15. Формирование PDF-отчёта (LaTeX/Markdown).

4.2. Средний уровень (15 вариантов)

1. Полный цикл: аудит → анализ → меры → верификация (2 облака).
2. Автоматизация повторного аудита в CI/CD (GitLab CI schedule).
3. Разработка 10+ OPA-политик с unit-тестами.
4. Внедрение Vault с динамическими секретами БД.
5. Построение Compliance Dashboard с историей аудитов.
6. Интеграция с Jira: автосоздание тикетов по находкам.
7. Сравнение ДО/ПОСЛЕ с графиками и метриками.
8. Настройка корреляции логов (CloudTrail + Audit Trails).
9. Реализация auto-remediation для 3 типов нарушений.
10. Анализ затрат на безопасность (FinOps + Security).
11. Threat Modeling (STRIDE) для всей платформы.
12. Интеграция с SOAR (Shuffle/n8n) для автоматического реагирования.
13. Разработка Self-Service Security Portal.
14. Презентация результатов для C-Level (Executive Summary).
15. Полный документ «Security Audit Report» (>30 страниц).

4.3. Повышенный уровень (15 вариантов)

1. Multi-cloud аудит (AWS + YC + Azure) с единым отчётом.
2. Внедрение Zero Trust Architecture на основе результатов аудита.
3. Построение SOC-дашборда с ML-аномалиями.
4. Разработка кастомного фреймворка аудита (на базе OPA + Prowler).

5. Полная автоматизация: аудит → тикеты → исправление → верификация → отчёт.
6. Интеграция с корпоративной GRC-системой (ServiceNow/Archer).
7. Red Team упражнение: эксплуатация найденных уязвимостей → исправление.
8. Разработка метрик эффективности (KPI Security Program).
9. Сравнение стоимости безопасности ДО и ПОСЛЕ (ROI Security).
10. Построение Threat Intelligence платформы.
11. Разработка оператора Kubernetes для автоматического аудита.
12. Интеграция с системой менеджмента инцидентов (PagerDuty).
13. Разработка учебного кейса для других команд.
14. Публикация white paper / статьи по результатам.
15. Защита проекта перед комиссией (презентация + демо).

5. Контрольные вопросы

1. Опишите методологию комплексного аудита безопасности облачной платформы. Какие этапы являются критическими и почему?
2. Как измерить эффективность внедрённых мер безопасности? Какие KPI вы использовали бы для оценки успеха проекта?
3. Какие инструменты автоматизации наиболее эффективны для предотвращения повторного появления уязвимостей?
4. Как обеспечить баланс между безопасностью и скоростью разработки (DevSecOps) при внедрении политик?
5. Какие компетенции, полученные в рамках дисциплины, вы считаете наиболее ценными для реальной работы?

6. Требования к отчету

Отчет в формате PDF. Обязательные разделы:

1. Титульный лист (название, ФИО, группа, дата).
2. Executive Summary (1 страница для руководства).
3. Методология аудита (фреймворки, инструменты, периметр).
4. Состояние «Как было»:
 - Сводная таблица находок по severity.
 - Детальное описание топ-10 нарушений.
 - Скриншоты отчётов Prowler/Checkov/Custodian.
 - Compliance Score (диаграмма).
5. План мероприятий:
 - Приоритизация (P0-P3).
 - Матрица ответственности (RACI).
 - Сроки и статус.

6. Внедрённые меры:
 - ОРА-политики (листинги, тесты).
 - Vault (архитектура, политики доступа).
 - Исправленные конфигурации (diff).
 - SIEM-дашборд (скриншоты, алерты).
7. Состояние «Как стало»:
 - Сводная таблица находок.
 - Сравнительный анализ ДО/ПОСЛЕ (графики, таблицы).
 - Новый Compliance Score.
8. Оставшиеся риски (реестр, план обработки).
9. Рекомендации (краткосрочные, среднесрочные, долгосрочные).
10. Приложения:
 - А. Полный отчёт Prowler (HTML/CSV — в архиве).
 - Б. Полный отчёт Checkov (JSON — в архиве).
 - В. Листинги ОРА-политик.
 - Г. Конфигурация Vault (без секретов).
 - Д. Дашборды (JSON-экспорт).

7. Критерии оценивания

7.1. Базовый уровень («удовлетворительно»)

- Проведён аудит минимум одним инструментом (Prowler или Checkov).
- Выявлено и задокументировано минимум 10 нарушений.
- Разработано минимум 3 ОРА-политики.
- Внедрён Vault для хранения минимум 3 секретов.
- Исправлено минимум 5 нарушений (подтверждено повторным аудитом).
- Сформирован PDF-отчёт (минимум 15 страниц).
- Снижение: отсутствует сравнительный анализ ДО/ПОСЛЕ — минус 5 баллов.

7.2. Средний уровень («хорошо»)

- Выполнены все критерии базового уровня.
- Использовано минимум 3 инструмента аудита.
- Разработано минимум 10 ОРА-политик с тестами.
- Внедрены динамические секреты Vault.
- Исправлены BCE CRITICAL и >80% HIGH-нарушений.
- Построен SIEM-дашборд с алертами.
- Сформирован профессиональный отчёт (минимум 30 страниц).

- Снижение: остались CRITICAL-нарушения — оценка не выше «удовлетворительно».

7.3. Повышенный уровень («отлично»)

- Выполнены все критерии среднего уровня.
- Проведён multi-cloud аудит (минимум 2 провайдера).
- Внедрена auto-remediation для минимум 3 типов нарушений.
- Настроена корреляция логов между облаками.
- Compliance Score $\geq 95\%$.
- Выполнена Red Team проверка (эксплуатация + исправление).
- Разработана система KPI для постоянного мониторинга.
- Сформирован отчёт уровня внешнего аудитора (50+ страниц).
- Снижение: Compliance Score $< 90\%$ — минус 10 баллов.

7.4. Общие штрафы

- Отчёт неструктурирован, отсутствует Executive Summary — минус 5 баллов.
- Нарушения исправлены, но не подтверждены повторным аудитом — минус 10 баллов.
- Секреты скомпрометированы (засвечены в отчёте/репозитории) — работа не принимается.
- Плагиат или фабрикация результатов — работа аннулируется.

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
по организации и проведению самостоятельной работы по дисциплине
«Управление облачными и гибридными платформами»
для студентов направления подготовки
09.04.04 Программная инженерия Направленность (профиль)
«Разработка, мониторинг и управление жизненным циклом инженерных
систем»

Ставрополь 2026

ВВЕДЕНИЕ

Методические указания для самостоятельной работы (СРС) по дисциплине **«Управление облачными и гибридными платформами»** предназначены для студентов магистратуры, обучающихся по направлению 09.04.04 «Программная инженерия» (профиль «Разработка, мониторинг и управление жизненным циклом инженерных систем»).

Дисциплина относится к части, **формируемой участниками образовательных отношений (вариативная часть)**, и изучается в 3-м семестре 2-го курса. Трудоёмкость - 3 зачётные единицы (108 академических часов), форма контроля - **зачёт с оценкой**.

Цель самостоятельной работы - закрепление, углубление и систематизация теоретических знаний, полученных на лекциях, а также формирование и развитие практических навыков безопасного управления облачными и гибридными инфраструктурами в рамках выполнения лабораторных работ и индивидуального комплексного проекта (Лабораторная работа №18).

Настоящие методические указания разработаны в соответствии с:

- Федеральным государственным образовательным стандартом высшего образования по направлению 09.04.04 «Программная инженерия»;

- Учебным планом направления подготовки;
- Рабочей программой дисциплины «Управление облачными и гибридными платформами»;
- Положением об организации образовательного процесса для лиц с ОВЗ и инвалидов в ФГАОУ ВО «Северо-Кавказский федеральный университет»;
- Программой воспитания СКФУ.

1. ОБЩАЯ ХАРАКТЕРИСТИКА САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1.1. Цель и задачи самостоятельной работы

Цель самостоятельной работы - формирование у обучающихся компетенций в области безопасного проектирования, развертывания и управления облачными и гибридными инфраструктурами через:

- самостоятельное изучение теоретического материала;
- подготовку к лабораторным занятиям;
- выполнение индивидуальных заданий (в том числе повышенного уровня сложности);
- выполнение и защиту сквозного комплексного проекта (ЛР №18).

Задачи самостоятельной работы:

1. Освоить теоретические основы безопасности облачных платформ (CIS Benchmarks, OWASP Cloud Top 10, модель разделения ответственности).
2. Сформировать навыки работы с инструментами IaC (Terraform, Pulumi), статического анализа IaC (Checkov, tfsec), политик как код (OPA/Rego).
3. Закрепить практические умения настройки безопасных CI/CD-пайплайнов с OIDC-федерацией.
4. Развить навыки комплексного аудита безопасности облачной инфраструктуры (Prowler, ScoutSuite, Cloud Custodian).

5. Подготовить и защитить комплексный проект «Аудит управления облачной платформой».

1.2. Формируемые компетенции и индикаторы

В результате выполнения самостоятельной работы студент должен обладать следующими компетенциями и индикаторами:

Код компетенции	Формулировка	Индикаторы
ПК-8	Способен интегрировать практики безопасности на всех этапах жизненного цикла, включая статический и динамический анализ кода, анализ уязвимостей зависимостей, безопасность контейнеров и облачной инфраструктуры, обеспечивая непрерывное соответствие требованиям безопасности	ИД-5_{ПК-8} сканирует конфигурации IaC на соответствие политикам безопасности ИД-6_{ПК-8} - интегрирует инструменты безопасности в CI/CD-пайплайны с QualityGates ИД-7_{ПК-8} - мониторит состояние безопасности контейнеров и инфраструктуры в рантайме ИД-8_{ПК-8} - обеспечивает непрерывное

Код компетенции	Формулировка	Индикаторы
		соответствие через автоматизированный сбор доказательств

1.3. Объём самостоятельной работы

Вид работы	Часы
Изучение теоретического материала (лекции, литература, документация)	10
Подготовка к лабораторным работам (выполнение индивидуальных заданий уровней 1-3)	18
Выполнение и защита сквозного комплексного проекта (Лабораторная работа №18)	20
Подготовка к зачёту с оценкой	6
Итого	54

2. МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ИЗУЧЕНИЮ ТЕОРЕТИЧЕСКОГО МАТЕРИАЛА

2.1. Темы для самостоятельного изучения

№	Тема	Содержание	Литература	Часы
1	Модель разделения ответственности	Shared Responsibility Model для	[1, гл. 1-2]	1

№	Тема	Содержание	Литература	Часы
	в облаках	IaaS/PaaS/SaaS, сравнение AWS, Yandex Cloud, Azure		
2	CIS Benchmarks для облачных платформ	CIS AWS Foundations, CIS Yandex Cloud, основные проверки	[2, гл. 4]	2
3	HashiCorp Vault: динамические секреты	AppRole, Database Secrets Engine, Lease, Renew, Response Wrapping	[3, гл. 5-7]	2
4	Open Policy Agent (OPA) и язык Rego	Синтаксис Rego, правила, тестирование, интеграция с Terraform	[4], документация OPA	2
5	OIDC-федерация для CI/CD	Принципы OIDC, настройка доверия между GitLab CI / GitHub Actions и облаком	[5], документация GitLab	2
6	Cloud Custodian: Compliance as	Архитектура, фильтры, действия,	[6], документация	1

№	Тема	Содержание	Литература	Часы
	Code	режимы pull/periodic/event-driven	Custodian	
7	Service Mesh и mTLS (Istio)	PeerAuthentication, AuthorizationPolicy, Zero Trust, сертификаты SPIFFE	[7], документация Istio	2
8	SIEM и централизованное логирование безопасности	ELK Stack, CloudTrail/Audit Trails, корреляция событий, алертинг	[8], документация ELK	2
9	FinOps и экономическая безопасность	Бюджетирование, Infracost, Cost Anomaly Detection, Chargeback/Showback	[9]	2
10	BYOC (Bring Your Own Cloud)	RACI-матрица, модели управления, политики самообслуживания	[10]	2

Рекомендации по изучению:

- Используйте официальную документацию инструментов (Terraform, Vault, OPA, Istio) - она наиболее актуальна.

- Для закрепления материала выполняйте короткие практические примеры (minilabs) в локальной среде (Docker Desktop + Minikube).
- Ведите конспект ключевых терминов: OIDC, mTLS, Envelope Encryption, Thundering Herd, Request Coalescing, Policy as Code, Compliance Score.

2.2. Рекомендуемые источники (доступные в РФ)

1. **Брикман Д.** Terraform: инфраструктура на уровне кода / Д. Брикман. — СПб.: Питер, 2021.
2. **Колодяжный Д. Ю.** Безопасность облачных вычислений / Д. Ю. Колодяжный. — М.: ИНФРА-М, 2022.
3. **Маккой Д.** HashiCorp Vault. Управление секретами и защита данных / Д. Маккой. — М.: ДМК-Пресс, 2020.
4. Документация Open Policy Agent (OPA): <https://www.openpolicyagent.org/docs/>
5. Документация GitLab CI/CD
OIDC: https://docs.gitlab.com/ee/ci/cloud_services/
6. Документация Cloud Custodian: <https://cloudcustodian.io/docs/>
7. Документация Istio: <https://istio.io/latest/docs/>
8. Документация Elastic Stack (SIEM): <https://www.elastic.co/guide/index.html>
9. Документация Infracost: <https://www.infracost.io/docs/>
10. CIS Benchmarks (бесплатно): <https://www.cisecurity.org/benchmark/>

3. МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ ЛАБОРАТОРНЫХ РАБОТ

3.1. Технологический стек (на выбор студента)

Компонент	Рекомендуемый вариант	Альтернатива
Облачный провайдер	Yandex Cloud (грант) /	VK Cloud, Selectel

Компонент	Рекомендуемый вариант	Альтернатива
	AWS Free Tier	
IaC	Terraform (OpenTofu)	Pulumi
Статический анализ IaC	Checkov + tfsec	Terrascan
Политики как код	OPA / Regal	Sentinel (TFC)
Управление секретами	HashiCorp Vault (dev/raft)	—
CI/CD	GitLab CI / GitHub Actions	Jenkins
Service Mesh	Istio (minikube/kind)	Linkerd
Контейнерная безопасность	Trivy	Grype, Clair
Аудит безопасности	Prowler (AWS) / ScoutSuite	—
Compliance as Code	Cloud Custodian	—
Мониторинг и SIEM	Grafana + Loki (ELK)	—

3.2. Соответствие лабораторных работ и этапов самостоятельной работы

№ ЛР	Тема лабораторной работы	Этап СРС	Результат
1	Развертывание защищенной облачной среды с	Изучение основ IaC, принципа наименьших	Terraform- конфигурация VPC + SG + BM

№ ЛР	Тема лабораторной работы	Этап СРС	Результат
	использованием Terraform	привилегий	
2	Управление секретами и ролями с HashiCorp Vault и IAM	Изучение динамических секретов, AppRole	Интеграция Vault с приложением на Python
3	Настройка сетевой безопасности: VPC, Security Groups, VPN	Изучение WireGuard/IPsec	VPN-туннель между локальной машиной и облаком
4	Безопасность контейнеров: анализ образов и политик Pod Security	Изучение Trivy, PSA	Отчёт Trivy, блокировка привилегированного пода
5	Статический анализ конфигураций IaC (Checkov/tfsec)	Изучение сдвига влево (Shift Left)	Pre-commit хук + Quality Gate в CI
6	Написание политик безопасности на Rego/OPA	Изучение Policy as Code	5+ политик Rego с unit-тестами
7	Compliance как Code с Cloud Custodian	Изучение рантайм-комплаенса	Автоматическое исправление несоответствий

№ ЛР	Тема лабораторной работы	Этап СРС	Результат
8	Безопасность CI/CD Pipeline (OIDC)	Изучение OIDC-федерации	Пайплайн без статических ключей
9	Аудит облачной инфраструктуры (Prowler/ScoutSuite)	Изучение CIS Benchmarks	HTML-отчёт аудита, план коррекции
10	Управление затратами в облаке (FinOps)	Изучение бюджетов, Infracost	Бюджеты + алерты + авто-остановка
11	Service Mesh и mTLS (Istio)	Изучение Zero Trust	mTLS + AuthorizationPolicy
12	Внедрение Continuous Compliance	Изучение Gatekeeper	ConstraintTemplate + аудит
13	Безопасное резервное копирование в гибридном облаке	Изучение Envelope Encryption	Client-side encryption + Vault
14	Логирование и мониторинг безопасности	Изучение SIEM	Дашборд безопасности + алерты
15	Балансировка нагрузки и атаки «Thundering Herd»	Изучение защитных паттернов	Request Coalescing + Circuit Breaker

№ ЛР	Тема лабораторной работы	Этап СРС	Результат
16	Проектирование многооблачной архитектуры (AWS + YC)	Изучение мультиоблачных рисков	Архитектура Terraform + VPN +
17	Реализация политик безопасности для BYOC	Изучение RACI, Policy Catalog	Библиотека 10+ политик
18	Комплексный проект: Аудит «Управление облачной платформой»	Защита проекта	Отчёт «Как было → Как стало», Compliance Score

3.3. Порядок выполнения самостоятельной работы

Этап 1. Организационный (1-2 недели семестра)

- Выбор технологического стека (облачный провайдер, инструменты).
- Регистрация в Yandex Cloud / AWS Free Tier (получение гранта).
- Установка Docker, Terraform, Git, VS Code, kubectl, Helm.

Этап 2. Освоение базовых технологий (3-6 недели)

- Выполнение лабораторных работ №1-4 (Terraform, Vault, сеть, контейнеры).
- Фиксация результатов в отчётах (скриншоты, листинги).
- Консультации с преподавателем по возникающим вопросам.

Этап 3. Автоматизация безопасности (7-10 недели)

- Выполнение лабораторных работ №5-9 (Checkov, OPA, Custodian, OIDC, Prowler).

- Интеграция политик в CI/CD-пайплайн (GitLab CI / GitHub Actions).

- Начало работы над комплексным проектом (выбор целевой инфраструктуры).

Этап 4. Расширенные темы и управление (11-14 недели)

- Выполнение лабораторных работ №10-15 (FinOps, Istio, Gatekeeper, бэкапы, SIEM, Thundering Herd).

- Сбор метрик и дашбордов.

- Продолжение комплексного аудита.

Этап 5. Итоговый проект и защита (15-18 недели)

- Выполнение лабораторных работ №16-17 (многооблачная архитектура, BYOC-политики).

- Завершение и оформление комплексного проекта (ЛР №18).

- Подготовка презентации.

- Защита проекта перед преподавателем.

3.4. Рекомендации по работе в российских облачных средах

- **Yandex Cloud:** предоставляет гранты для студентов (бесплатно до 1000–2000 ₽). Регистрация через почту вуза.

- **VK Cloud:** также имеет программу для образования (тестовый период).

- **AWS Free Tier:** доступен из РФ, но требует подтверждения платёжной карты (рекомендуется использовать Yandex Cloud как основной провайдер).

- При работе с Vault, Istio, OPA используйте локальный Minikube/Kind для снижения затрат.

4. ВОПРОСЫ ДЛЯ СОБЕСЕДОВАНИЯ ПО САМОСТОЯТЕЛЬНОМУ ИЗУЧЕНИЮ ЛИТЕРАТУРЫ

1. В чем заключается модель разделения ответственности (Shared Responsibility) для IaaS, PaaS и SaaS?

2. Какие основные проверки входят в CIS Benchmark для AWS / Yandex Cloud?
3. Что такое Envelope Encryption и почему он безопаснее прямого шифрования мастер-ключом?
4. Как работает AppRole в HashiCorp Vault? В чем отличие Role ID от Secret ID?
5. Объясните синтаксис базового правила deny на языке Rego.
6. Как OIDC-федерация позволяет отказаться от статических ключей доступа в CI/CD?
7. Какие режимы запуска (mode) поддерживает Cloud Custodian? В чем различие pull и periodic?
8. Чем STRICT-режим mTLS отличается от PERMISSIVE? Как выполняется миграция?
9. Какие метрики эффективности (KPI) используются для оценки зрелости Compliance as Code?
10. Что такое Thundering Herd и как Request Coalescing решает эту проблему?
11. Какие риски безопасности характерны для мультиоблачной архитектуры?
12. В чем отличие модели BYOC (Bring Your Own Cloud) от централизованной облачной модели?
13. Как настроить алерт в Kibana/Watcher на событие ConsoleLogin?
14. Какие инструменты используются для статического анализа IaC и для аудита работающей инфраструктуры?
15. Как измерить ROI (возврат инвестиций) от внедрения политик безопасности?

5. КРИТЕРИИ ОЦЕНИВАНИЯ КОМПЕТЕНЦИЙ

5.1. Критерии оценивания лабораторных работ

Оценка лабораторной работы складывается из:

- выполнения задания в полном объёме — 40%;
- качества оформления отчёта — 20%;
- правильности ответов на контрольные вопросы — 20%;
- демонстрации работы (скриншоты, логи) — 20%.

Шкала оценки лабораторной работы

Оценка	Критерии
5 (отлично)	Задание выполнено полностью, решение работает без ошибок, отчёт оформлен по требованиям, студент уверенно отвечает на вопросы
4 (хорошо)	Задание выполнено полностью, но есть небольшие замечания (неоптимальная конфигурация, неполный отчёт)
3 (удовлетворительно)	Задание выполнено с помощью преподавателя, есть ошибки, исправляемые с наводящими вопросами
2 (неудовлетворительно)	Задание не выполнено или выполнено неверно, студент не понимает принципов работы

5.2. Критерии оценивания сквозного комплексного проекта (ЛР №18)

Критерий	Вес (%)	Отлично	Хорошо	Удовлетворительно	Неудовлетворительно
Полнота аудита	20	3+ инструмента	2 инструмента	1 инструмент	аудит не проведён
Разработк	15	10+	5-9	1-4 политики	нет

Критерий	Вес (%)	Отлично	Хорошо	Удовлетворительно	Неудовлетворительно
а политик OPA/Rego		политик + тесты	политик		политик
Исправление нарушений (Remediation)	20	>90% CRITICAL +HIGH	70-90%	50-69%	<50%
Внедрение Vault	10	динамические секреты + интеграция	статические секреты	только базовое использование	не внедрён
Мониторинг и SIEM	10	дашборд + алерты + корреляция	дашборд	базовые логи	нет
Compliance Score	10	≥95%	85-94%	70-84%	<70%
Качество отчёта и презентац	15	полный отчёт (50+ стр.),	хороший отчёт (30+	базовый отчёт (15+ стр.)	отчёт отсутствует

Критерий	Вес (%)	Отлично	Хорошо	Удовлетворительно	Неудовлетворительно
ии		Executive Summary, графики	стр.)		

Шкала перевода баллов в оценку для проекта:

Процент баллов	Оценка
90-100%	5 (отлично)
75-89%	4 (хорошо)
60-74%	3 (удовлетворительно)
<60%	2 (неудовлетворительно)

5.3. Общие требования к отчётности

Каждая лабораторная работа и комплексный проект оформляются в виде электронного отчёта (PDF / Markdown), содержащего:

1. Титульный лист (ФИО, группа, вариант, номер работы).
2. Цель и задачи работы.
3. Краткое теоретическое обоснование.
4. Практическая часть (листинги кода, конфигурации, скриншоты).
5. Анализ результатов (выводы, рекомендации, сравнительные таблицы).
6. Ответы на контрольные вопросы (3-5 вопросов по выбору преподавателя).
7. Список использованных источников.

Отчёт должен быть загружен в электронную информационно-образовательную среду СКФУ или предоставлен преподавателю на проверку в установленный срок.