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

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по выполнению лабораторных работ
по дисциплине «Контейнеризация и DevOps» для студентов направления подготовки
09.03.04 Программная инженерия
Направленность (профиль)
«Разработка и сопровождение программного обеспечения»

Ставрополь 2026

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 3
Лабораторная работа 1. Создание Helm-чарта для web-приложения 5
Лабораторная работа 2. Helm-хуки для миграции базы данных 10
Лабораторная работа 3. Знакомство с Kubernetes Operators: анализ готового оператора 15
Лабораторная работа 4. GitOps с ArgoCD: установка и первый Application 20
Лабораторная работа 5. GitOps с ArgoCD: мультитенантность и Image Updater 25
Лабораторная работа 6. Политики безопасности: OPA/Gatekeeper 30
Лабораторная работа 7. Политики безопасности: Kyverno 35
Лабораторная работа 8. Nginx Ingress Controller и Cert-Manager 40
Лабораторная работа 9. Канареечный релиз через Ingress 46
Лабораторная работа 10. Serverless с Knative Serving 52
Лабораторная работа 11. Event-driven автомасштабирование с KEDA 57
Лабораторная работа 12. Сравнение GitOps-инструментов Flux и ArgoCD 62
Лабораторная работа 13. FinOps: ResourceQuota и LimitRange 67
Лабораторная работа 14. Сквозной проект: GitOps-платформа для микросервисов с mTLS 72

ВВЕДЕНИЕ

Методические указания предназначены для студентов направления подготовки 09.03.04 «Программная инженерия» (направленность (профиль) «Разработка и сопровождение программного обеспечения») и определяют порядок выполнения лабораторных работ по дисциплине «Управление жизненным циклом инженерных систем».

Дисциплина формирует у студентов системное представление об управлении жизненным циклом современных программных и инженерных систем — от проектирования и сборки до развёртывания, эксплуатации, сопровождения и вывода из эксплуатации. Особое внимание уделяется практикам контейнеризации, оркестрации и DevOps, которые сегодня составляют технологическую основу промышленной разработки и сопровождения программного обеспечения.

Цель лабораторного практикума — закрепить теоретические знания и сформировать устойчивые практические навыки работы с инструментами управления жизненным циклом контейнеризированных приложений в среде Kubernetes. В ходе выполнения работ студенты осваивают пакетный менеджер Helm, модель GitOps на базе ArgoCD и Flux, механизмы расширения Kubernetes (операторы и пользовательские ресурсы), политики безопасности (OPA/Gatekeeper, Kyverno), управление сетевым доступом и сертификатами (Ingress, Cert-Manager), serverless- и событийно-ориентированные подходы (Knative, KEDA), а также практики управления ресурсами и затратами (FinOps).

Лабораторный практикум построен по принципу последовательного усложнения: от создания и параметризации Helm-чартов к декларативному управлению развёртыванием, политикам безопасности, автоматическому масштабированию и, в завершение, к сквозному проекту построения GitOps-платформы для микросервисов. Такой порядок обеспечивает постепенное

формирование целостной инженерной компетенции по управлению жизненным циклом систем.

Каждая лабораторная работа имеет единую структуру и включает цель и содержание работы, теоретическое обоснование, разбор практического примера, порядок выполнения индивидуального задания, варианты заданий, требования к содержанию и форме отчёта, а также вопросы для защиты работы. Перед выполнением работы студенту необходимо изучить теоретический материал и пример, после чего приступить к выполнению индивидуального задания согласно полученному варианту.

Результатом выполнения каждой работы является оформленный отчёт, который сдаётся преподавателю и защищается в форме собеседования по контрольным вопросам. Выполнение лабораторного практикума в полном объёме обеспечивает достижение планируемых результатов обучения по дисциплине и формирование соответствующих компетенций.

Лабораторная работа 1.

СОЗДАНИЕ HELM-ЧАРТА ДЛЯ WEB-ПРИЛОЖЕНИЯ

Цель и содержание работы: научиться создавать Helm-чарт для веб-приложения, параметризовать Kubernetes-манифесты и использовать разные values-файлы для окружений dev и prod.

Теоретическое обоснование

Helm - это пакетный менеджер для Kubernetes. Он позволяет объединить набор связанных манифестов в единый пакет, который называется chart. В chart обычно входят Deployment, Service, Ingress, ConfigMap, Secret, ServiceAccount и другие ресурсы.

Главная идея Helm состоит в том, что повторяющиеся значения не нужно вручную менять во всех YAML-файлах. Они выносятся в values.yaml, а манифесты превращаются в шаблоны. При установке Helm подставляет значения в шаблоны и отправляет готовые ресурсы в Kubernetes API.

Релиз Helm - это конкретная установленная копия чарта. Один и тот же chart можно установить несколько раз с разными именами и values-файлами. Например, dev-релиз может иметь одну реплику и тестовый образ, а prod-релиз - несколько реплик, requests/limits и отдельные переменные окружения.

Пример

В примере создаётся Helm-чарт для простого веб-приложения на базе nginx. Чарт будет поддерживать изменение образа, числа реплик, порта сервиса и переменных окружения без ручного редактирования Deployment.

Рисунок 1 - Упрощённая схема выполняемого сценария

Шаг 1. Создать базовую структуру чарта

Команда helm create создаёт заготовку чарта. В ней уже есть Chart.yaml, values.yaml и каталог templates. На практике эту заготовку почти всегда упрощают: удаляют лишние шаблоны, которые не нужны в конкретной работе.

```
helm create webapp-chart
cd webapp-chart
ls -la
ls templates
```

Шаг 2. Описать параметры приложения в values.yaml

Файл values.yaml должен хранить все значения, которые могут отличаться между окружениями: образ, тег, число реплик, порт, переменные окружения и ресурсы контейнера.

```
replicaCount: 2

image:
  repository: nginx
  tag: "1.25"

service:
  type: ClusterIP
  port: 80

env:
  APP_ENV: dev

resources:
  requests:
    cpu: 100m
    memory: 128Mi
  limits:
    cpu: 300m
    memory: 256Mi
```

Шаг 3. Заменить жёсткие значения в Deployment на шаблонные

В templates/deployment.yaml нужно убрать фиксированные значения и заменить их обращениями к Values. Благодаря этому один и тот же шаблон будет работать для разных окружений.

```
spec:
  replicas: {{ .Values.replicaCount }}
  template:
    spec:
      containers:
        - name: {{ .Chart.Name }}
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
          ports:
            - containerPort: {{ .Values.service.port }}
          env:
            - name: APP_ENV
              value: {{ .Values.env.APP_ENV | quote }}
```

Шаг 4. Проверить результат шаблонизации до установки

Команда helm template показывает готовые Kubernetes-манифесты. Это безопасный способ найти ошибку в шаблоне до применения ресурсов в кластер.

```
helm template demo ./webapp-chart
helm lint ./webapp-chart
```

Шаг 5. Создать отдельный values-файл для production

Для production-окружения обычно задают больше реплик, фиксированную версию образа и более строгие ресурсы. Это делается отдельным файлом, а не изменением основного шаблона.

```
# prod-values.yaml
replicaCount: 3
image:
  repository: nginx
  tag: "1.25.5"
env:
  APP_ENV: prod
```

Шаг 6. Установить и обновить релиз

Сначала устанавливается dev-релиз. Затем можно изменить values и выполнить upgrade. Helm сохранит историю релиза, поэтому при необходимости можно сделать rollback.

```
helm install webapp-dev ./webapp-chart
kubectl get pods,svc
helm upgrade webapp-dev ./webapp-chart -f prod-values.yaml
helm history webapp-dev
```

Порядок выполнения индивидуального задания

1) создать Helm-чарт для приложения по своему варианту;

2) параметризовать Deployment, Service и при необходимости Ingress;

3) подготовить values-файлы;

4) проверить chart через helm lint и helm template;

5) установить релиз в кластер и выполнить хотя бы одно обновление через helm upgrade;

6) зафиксировать в отчёте команды и результаты проверки.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Статический сайт на nginx с разными replicaCount для dev/prod.
2	HTTP-сервис на httpd с параметризацией service.type и service.port.
3	FastAPI-приложение с переменной окружения APP_ENV.
4	Node.js Express-приложение с ConfigMap для настроек.
5	React/Vite frontend, отдаваемый через nginx.
6	Go HTTP API с readinessProbe и livenessProbe.
7	Spring Boot REST API с параметрами JVM через env.
8	Веб-приложение с Redis как зависимостью Helm chart.
9	Веб-приложение с PostgreSQL как зависимостью Helm chart.
10	Приложение с Ingress, host которого задаётся в values.yaml.
11	Приложение с HorizontalPodAutoscaler, включаемым флагом autoscaling.enabled.
12	Чарт с несколькими контейнерами в одном Pod: app + sidecar.
13	Чарт с Secret для учебного пароля, создаваемого из values.
14	Чарт с ServiceAccount и настройкой imagePullSecrets.

Содержание отчета и его форма

Отчет должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия Kubernetes, kubectl и Helm, список установленных компонентов. Результаты проверки: вывод kubectl/helm/argocd/flux, скриншоты интерфейсов при их использовании. Ответы на контрольные вопросы.

Вопросы для защиты работы

1. Что такое Helm chart и чем он отличается от обычного набора YAML-манифестов?

2. Для чего нужен values.yaml?

3. Что означает объект Release в Helm?

4. Зачем перед установкой выполнять helm template и helm lint?

5. Почему для dev и prod лучше использовать разные values-файлы, а не разные шаблоны?

6. Как подключаются зависимости Helm-чарта?

7. Какие данные не рекомендуется хранить в открытом values.yaml?

8. Что делает команда helm rollback?

Лабораторная работа 2.

HELM-ХУКИ ДЛЯ МИГРАЦИИ БАЗЫ ДАННЫХ

Цель и содержание работы: освоить запуск служебных задач при установке и обновлении Helm-релиза на примере автоматической миграции базы данных.

Теоретическое обоснование

Helm-хук - это ресурс Kubernetes, который выполняется в определённый момент жизненного цикла релиза: до установки, после установки, до обновления, после обновления и т.д. Для миграций чаще всего используют pre-install и pre-upgrade.

Миграция базы данных должна выполниться до запуска новой версии приложения. Если схема БД не обновлена, приложение может завершиться с ошибкой или начать работать некорректно. Поэтому миграция оформляется как Kubernetes Job с аннотациями Helm hook.

Важная часть настройки - политика удаления hook-ресурсов. Если не удалять старые Job, повторное обновление может завершиться конфликтом имён. Для этого используется helm.sh/hook-delete-policy.

Пример

В примере добавляется Job, который имитирует миграцию базы данных перед установкой и обновлением релиза. Даже если реальной БД нет, такой пример позволяет понять порядок запуска hook-ресурсов.

Шаг 1. Создать шаблон Job для миграции

В каталоге templates создаётся отдельный файл migration-job.yaml. Job запускает контейнер и выполняет команду миграции. В реальном проекте это может быть alembic upgrade head, flyway migrate или django manage.py migrate.

```
mkdir -p templates
touch templates/migration-job.yaml
```

Шаг 2. Добавить аннотации Helm hook

Аннотации сообщают Helm, что этот Job не является обычным постоянным ресурсом приложения. Он должен выполняться до установки и до обновления релиза.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: "{{ .Release.Name }}-db-migrate"
  annotations:
    "helm.sh/hook": pre-install,pre-upgrade
    "helm.sh/hook-weight": "-5"
    "helm.sh/hook-delete-policy":          before-hook-
creation,hook-succeeded
spec:
  template:
    spec:
      restartPolicy: Never
      containers:
        - name: migrate
          image: busybox:1.36
          command: ["sh", "-c"]
          args:
            - echo "Start migration"; sleep 5; echo
"Migration completed"
```

Шаг 3. Проверить порядок выполнения

При установке релиза Helm сначала создаст hook-Job. Только после успешного завершения Job будут применяться основные ресурсы чарта.

```
helm install webapp ./webapp-chart --wait
kubectl get jobs
kubectl logs job/webapp-db-migrate
```

Шаг 4. Проверить поведение при обновлении

Измените tag образа или replicaCount и выполните helm upgrade. Job должен появиться снова, потому что он привязан к pre-upgrade.

```
helm upgrade webapp ./webapp-chart --wait
kubectl get jobs
helm history webapp
```

Шаг 5. Смоделировать ошибку миграции

Для понимания поведения Helm полезно временно заменить команду Job на exit 1. Тогда установка или обновление должны завершиться ошибкой, а приложение не должно считаться успешно обновлённым.

```
args:
  - echo "Migration failed"; exit 1
```

Шаг 6. Вернуть успешную миграцию и очистить ресурсы

После эксперимента нужно вернуть успешную команду и удалить тестовый релиз. Очистка важна, чтобы старые Job и Pods не мешали следующим запускам.

```
helm upgrade webapp ./webapp-chart --wait
helm uninstall webapp
```

Порядок выполнения индивидуального задания

1) добавить hook-Job в Helm-чарт;
2) настроить pre-install и pre-upgrade;
3) настроить hook-weight и hook-delete-policy;
4) продемонстрировать успешный запуск миграции;
5) продемонстрировать поведение при ошибке миграции;
6) описать, почему миграции нельзя запускать после старта новой версии приложения.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Alembic migration для FastAPI-приложения.
2	Django manage.py migrate перед запуском backend.

№ варианта	Содержание индивидуального задания
3	Flyway migration для Java/Spring Boot.
4	Liquibase update для PostgreSQL.
5	Prisma migrate deploy для Node.js приложения.
6	TypeORM migration:run для NestJS.
7	Knex migrate:latest для Express API.
8	Rails db:migrate для Ruby on Rails.
9	Laravel php artisan migrate для PHP-приложения.
10	EF Core database update для .NET-приложения.
11	Имитация миграции через busybox с проверкой порядка hook-weight.
12	Job предварительной проверки доступности PostgreSQL перед деплоем.
13	Job создания начальных справочников/seed-данных после миграции.
14	Два hook-Job: backup перед миграцией и migration после backup.

Содержание отчета и его форма

Отчет должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия Kubernetes, kubectl и Helm, список установленных компонентов. Результаты проверки: вывод kubectl/helm/argocd/flux, скриншоты интерфейсов при их использовании. Ответы на контрольные вопросы.

Вопросы для защиты работы

1. Что такое Helm hook?
2. Чем pre-install отличается от pre-upgrade?
3. Почему миграции часто оформляют как Kubernetes Job?
4. Для чего нужен hook-weight?
5. Что делает hook-delete-policy?
6. Что произойдёт, если hook-Job завершится ошибкой?
7. Почему миграции БД должны быть идемпотентными?
8. Какие риски есть у автоматических миграций в production?

Лабораторная работа 3.

Знакомство с Kubernetes Operators: анализ готового оператора

Цель работы: понять назначение Kubernetes Operator, изучить CRD и проследить, как контроллер создаёт дочерние ресурсы на основе пользовательского ресурса.

Теоретическое обоснование

Kubernetes Operator расширяет Kubernetes API новыми типами ресурсов и автоматизирует эксплуатационную логику приложения. Например, оператор Kafka может создавать StatefulSet, Services, ConfigMaps и управлять обновлениями брокеров.

CRD, или Custom Resource Definition, добавляет в кластер новый тип объекта. Пользователь создаёт Custom Resource, а контроллер оператора наблюдает за ним и приводит реальные ресурсы к требуемому состоянию.

Ключевая идея контроллера - reconcile loop. Контроллер постоянно сравнивает желаемое состояние, описанное в spec пользовательского ресурса, с фактическим состоянием в кластере и выполняет необходимые действия.

Подробный пример выполнения

В примере рассматривается Strimzi Kafka Operator. Он устанавливает CRD Kafka, после чего пользователь может создать объект Kafka, а оператор сам создаст StatefulSet брокеров и служебные ресурсы.

Шаг 1. Создать namespace для оператора

Операторы лучше устанавливать в отдельный namespace. Так проще отличать системные ресурсы оператора от ресурсов приложения.

```
kubectl create namespace kafka
```

Шаг 2. Установить оператор

Для учебной работы можно использовать официальный манифест или Helm-чарт, если он разрешён преподавателем. После установки нужно дождаться готовности Pod оператора.

```
kubectl apply -f https://strimzi.io/install/latest?namespace=kafka -n kafka
kubectl get pods -n kafka --watch
```

Шаг 3. Найти созданные CRD

После установки в API Kubernetes появятся новые типы ресурсов. Важно увидеть, что оператор не просто запускает Pod, а расширяет API кластера.

```
kubectl get crd | grep strimzi
kubectl explain kafka.spec
```

Шаг 4. Создать пользовательский ресурс Kafka

Пользователь описывает желаемый Kafka-кластер. В spec задаются реплики, listeners, storage и параметры Zookeeper/KRaft в зависимости от версии оператора.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: demo-kafka
  namespace: kafka
spec:
  kafka:
    version: 3.7.0
    replicas: 1
    listeners:
      - name: plain
        port: 9092
        type: internal
```

```
        tls: false
    storage:
      type: ephemeral
  zookeeper:
    replicas: 1
    storage:
      type: ephemeral
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

Шаг 5. Проследить появление дочерних ресурсов

После применения Custom Resource оператор создаёт StatefulSet, Services, Pods и ConfigMaps. Эти ресурсы не были написаны вручную, их создал контроллер.

```
kubectl apply -f kafka.yaml
kubectl get kafka -n kafka
kubectl get pods,svc,statefulset -n kafka
```

Шаг 6. Изменить spec и проверить reconcile

Измените число реплик или другой безопасный параметр. Контроллер должен заметить изменение и привести ресурсы к новому состоянию.

```
kubectl edit kafka demo-kafka -n kafka
kubectl get pods -n kafka --watch
```

Шаг 7. Удалить пользовательский ресурс

При удалении Custom Resource оператор должен удалить связанные дочерние ресурсы или выполнить финализацию. Это показывает, что оператор управляет жизненным циклом приложения.

```
kubectl delete kafka demo-kafka -n kafka
kubectl get all -n kafka
```

Порядок выполнения индивидуального задания

1) выбрать оператор по варианту;

2) установить оператор в отдельный namespace;

3) найти CRD и изучить поля spec/status;

4) создать Custom Resource;

5) описать, какие дочерние ресурсы создал оператор;

6) изменить spec и показать реакцию контроллера;

7) удалить ресурс и проверить очистку.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Strimzi Kafka Operator: создать Kafka-кластер.
2	Prometheus Operator: создать ресурс Prometheus.
3	Cert-Manager: изучить CRD Certificate и Issuer.
4	External Secrets Operator: создать ExternalSecret с учебным backend.
5	RabbitMQ Cluster Operator: создать RabbitMQCluster.
6	CloudNativePG Operator: создать PostgreSQL-кластер.
7	CrunchyData Postgres Operator: изучить PostgresCluster.
8	Redis Operator: создать Redis-кластер или standalone Redis.
9	MongoDB Community Operator: создать MongoDB Community resource.
10	Elastic Cloud on Kubernetes: изучить Elasticsearch CRD.
11	Keycloak Operator: создать Keycloak instance.
12	Tekton Operator: изучить установленные CRD Pipeline/Task.

№ варианта	Содержание индивидуального задания
13	Crossplane: создать ProviderConfig и изучить managed resources.
14	Argo Rollouts: создать Rollout и сравнить с Deployment.

Содержание отчета и его форма

Отчет должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия Kubernetes, kubectl и Helm, список установленных компонентов. Результаты проверки: вывод kubectl/helm/argocd/flux, скриншоты интерфейсов при их использовании. Ответы на контрольные вопросы.

Вопросы для защиты работы

1. Что такое Kubernetes Operator?
2. Что такое CRD?
3. Чем Custom Resource отличается от стандартного Deployment?
4. Что такое reconcile loop?
5. Какие поля обычно есть в spec и status?
6. Почему операторы удобны для stateful-приложений?
7. Какие риски связаны с установкой CRD в кластер?
8. Как понять, какие ресурсы создал оператор?

Лабораторная работа 4.

GitOps с ArgoCD: установка и первый Application

Цель работы: развернуть ArgoCD и настроить автоматическую синхронизацию Kubernetes-приложения из Git-репозитория.

Теоретическое обоснование

GitOps - это подход, при котором желаемое состояние инфраструктуры и приложений хранится в Git. Любое изменение проходит через commit, review и историю изменений, а специальный контроллер синхронизирует кластер с репозиторием.

ArgoCD реализует pull-модель: не CI-сервер отправляет изменения в кластер, а ArgoCD внутри кластера сам считывает репозиторий и применяет манифесты. Это снижает необходимость хранить kubeconfig в CI.

Application в ArgoCD связывает источник манифестов, путь в репозитории и целевой кластер/namespace. Политики sync определяют, будет ли синхронизация ручной или автоматической, а также нужно ли удалять ресурсы, исчезнувшие из Git.

Пример

В примере разворачивается ArgoCD и создаётся Application, который синхронизирует простой nginx Deployment из Git-репозитория. После изменения replicas в Git ArgoCD автоматически обновит кластер.

Шаг 1. Установить ArgoCD

ArgoCD устанавливается в отдельный namespace. Для учебного сценария достаточно стандартных манифестов.

```
kubectl create namespace argocd
```

```
    kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
    kubectl get pods -n argocd --watch
```

Шаг 2. Получить доступ к Web UI

В локальном кластере проще использовать port-forward. Начальный пароль хранится в Kubernetes Secret.

```
    kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 -d
    kubectl port-forward svc/argocd-server -n argocd 8080:443
```

Шаг 3. Подготовить репозиторий с манифестами

В репозитории создаётся каталог apps/demo. В нём размещаются deployment.yaml и service.yaml. Это и будет источник истины для ArgoCD.

```
    apps/demo/deployment.yaml
    apps/demo/service.yaml

    git add apps/demo
    git commit -m "add demo app"
    git push
```

Шаг 4. Создать Application

Application можно создать через UI или применить YAML. Важно указать repoURL, path и destination namespace.

```
    apiVersion: argoproj.io/v1alpha1
    kind: Application
    metadata:
      name: demo-app
      namespace: argocd
```

```
    spec:
      project: default
      source:
        repoURL:    https://github.com/example/k8s-gitops-
demo.git
        targetRevision: main
        path: apps/demo
      destination:
        server: https://kubernetes.default.svc
        namespace: demo
      syncPolicy:
        automated:
          prune: true
          selfHeal: true
        syncOptions:
          - CreateNamespace=true
```

Шаг 5. Проверить автоматическую синхронизацию

После применения Application ArgoCD должен создать namespace demo и ресурсы приложения. Статус должен стать Synced/Healthy.

```
    kubectl apply -f application.yaml
    kubectl get app -n argocd
    kubectl get all -n demo
```

Шаг 6. Проверить изменение через Git

Измените replicas в deployment.yaml с 1 на 3 и отправьте commit. Через некоторое время ArgoCD применит изменение.

```
    git commit -am "scale demo to 3 replicas"
    git push
    kubectl get deploy -n demo -w
```

Шаг 7. Проверить self-heal

Вручную измените Deployment через kubectl. ArgoCD должен вернуть значение из Git, потому что включён selfHeal.

```
kubectl scale deploy demo --replicas=1 -n demo
kubectl get deploy demo -n demo -w
```

Порядок выполнения индивидуального задания

1) установить ArgoCD;

2) создать Git-репозиторий с приложением;

3) создать Application;

4) включить automated sync, prune и self-heal;

5) проверить изменение через commit;

6) проверить восстановление после ручного изменения ресурса.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Application для plain YAML-манифестов.
2	Application для Kustomize-приложения.
3	Application для Helm-чарта из репозитория.
4	Application с автоматическим созданием namespace.
5	Application с ручной синхронизацией и последующим переходом на auto-sync.
6	Application с prune=true и удалением ресурса из Git.
7	Application с selfHeal=true и проверкой drift.
8	Application для двух окружений dev и stage.
9	Application, синхронизирующий ConfigMap и Deployment.

№ варианта	Содержание индивидуального задания
10	Application с Ingress и Service.
11	Application с private repository через repository credentials.
12	Application, использующий targetRevision не main, а отдельную ветку.
13	Application с sync waves для двух манифестов.
14	Application с Helm values-файлом для prod.

Содержание отчета и его форма

Отчет должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия Kubernetes, kubectl и Helm, список установленных компонентов. Результаты проверки: вывод kubectl/helm/argocd/flux, скриншоты интерфейсов при их использовании. Ответы на контрольные вопросы.

Вопросы для защиты работы

1. Что такое GitOps?
2. Чем pull-модель отличается от push-модели CI/CD?
3. Какие компоненты входят в ArgoCD?
4. Что описывает Application?
5. Для чего нужны prune и self-heal?
6. Что означает статус OutOfSync?
7. Почему ручные изменения в кластере нежелательны при GitOps?

Лабораторная работа 5.

GitOps с ArgoCD: мультитенантность и Image Updater

Цель работы: настроить разделение приложений по проектам ArgoCD и автоматическое обновление версии контейнерного образа через ArgoCD Image Updater.

Теоретическое обоснование

Мультитенантность позволяет нескольким командам использовать один ArgoCD, но с разными границами доступа. AppProject ограничивает допустимые репозитории, namespace, кластеры и типы ресурсов.

Image Updater решает задачу обновления версий образов. Он отслеживает registry и изменяет тег образа в Git или параметрах приложения. В GitOps-процессе предпочтительно, чтобы изменение попадало в Git как commit или pull request.

Автоматическое обновление образов повышает скорость поставки, но требует аккуратных правил: нельзя бесконтрольно разворачивать latest в production, нужно использовать semver-стратегии, ограничения registry и ревью изменений.

Пример

В примере создаётся AppProject team-a, который разрешает приложения только из одного репозитория и только в namespace team-a. Затем приложение настраивается на автоматическое обновление тега образа.

Шаг 1. Создать namespace команды

Namespace помогает отделить ресурсы одной команды от другой. AppProject дополнительно запрещает ArgoCD деплоить приложение за пределы разрешённой области.

```
kubectl create namespace team-a
```

Шаг 2. Описать AppProject

В проекте указываются разрешённые репозитории и destinations. Если Application попытается использовать другой namespace или repoURL, ArgoCD отклонит синхронизацию.

```
apiVersion: argoproj.io/v1alpha1
kind: AppProject
metadata:
  name: team-a
  namespace: argocd
spec:
  sourceRepos:
    - https://github.com/example/team-a-apps.git
  destinations:
    - namespace: team-a
      server: https://kubernetes.default.svc
```

Шаг 3. Перевести Application в проект

В spec.project нужно указать имя созданного проекта. После этого приложение наследует ограничения AppProject.

```
spec:
  project: team-a
  destination:
    server: https://kubernetes.default.svc
    namespace: team-a
```

Шаг 4. Установить Image Updater

В учебном сценарии Image Updater можно установить через Helm. Он должен иметь доступ к ArgoCD и к репозиторию, если используется write-back в Git.

```
    helm repo add argo https://argoproj.github.io/argo-helm
    helm upgrade --install argocd-image-updater argo/argocd-image-updater -n argocd
```

Шаг 5. Добавить аннотации отслеживания образа

Аннотации указывают, какой образ отслеживать и какую стратегию обновления применять. Для учебного примера можно использовать semver и теги вида 1.0.0, 1.0.1.

```
    metadata:
      annotations:
        argocd-image-updater.argoproj.io/image-list: app=ghcr.io/example/team-a-app
        argocd-image-updater.argoproj.io/app.update-strategy: semver
        argocd-image-updater.argoproj.io/write-back-method: git
```

Шаг 6. Опубликовать новый тег и проверить обновление

После появления нового тега Image Updater должен изменить значение в Git или параметрах ArgoCD. После синхронизации в кластере появится Pod с новым image.

```
    kubectl -n argocd logs deploy/argocd-image-updater
    kubectl get deploy -n team-a -o jsonpath="{.items[0].spec.template.spec.containers[0].image}"
```

Порядок выполнения индивидуального задания

1) создать AppProject по варианту;

2) ограничить репозитории и namespace;

3) создать или изменить Application, привязав его к проекту;

4) установить и настроить Image Updater;

5) проверить обновление тега образа;

6) описать, какие ограничения защищают кластер от неправильного деплоя.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Проект team-a с одним namespace и одним репозиторием.
2	Проект team-b с двумя namespace: team-b-dev и team-b-prod.
3	Проект, разрешающий только Helm-приложения из одного repoURL.
4	Проект, запрещающий cluster-scoped ресурсы.
5	Проект с разрешением только ConfigMap, Deployment и Service.
6	Image Updater со стратегией semver.
7	Image Updater со стратегией newest-build для dev.
8	Image Updater с write-back в Git.
9	Image Updater с обновлением Helm parameter image.tag.
10	Автообновление образа из Docker Hub.
11	Автообновление образа из GitHub Container Registry.
12	Два Application в одном AppProject с разными namespace.
13	Проверка ошибки при попытке деплоя в запрещённый namespace.
14	Сценарий dev auto-update и prod только после ручного merge.

Содержание отчета и его форма

Отчет должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия Kubernetes, kubectl и Helm, список установленных компонентов. Результаты проверки: вывод kubectl/helm/argocd/flux, скриншоты интерфейсов при их использовании. Ответы на контрольные вопросы.

Вопросы для защиты работы

1. Что такое AppProject?
2. Какие ограничения можно задать в AppProject?
3. Зачем нужна мультитенантность в GitOps-платформе?
4. Как Image Updater узнаёт о новых версиях образа?
5. Чем semver-стратегия лучше latest для production?
6. Что означает write-back в Git?
7. Почему автоматическое обновление образов может быть опасным?
8. Как проверить, что приложение действительно обновило image?

Лабораторная работа 6.

Политики безопасности: OPA/Gatekeeper

Цель работы: реализовать политики admission-контроля с помощью OPA/Gatekeeper и запретить небезопасные Pod-конфигурации.

Теоретическое обоснование

OPA использует язык Rego для описания правил принятия решений. Gatekeeper подключает OPA к Kubernetes admission pipeline и проверяет ресурсы до их сохранения в API Server.

ConstraintTemplate содержит Rego-код и схему пользовательского Constraint. Constraint определяет область применения политики и параметры.

Политики безопасности как код позволяют формализовать требования: запрет latest, запрет privileged, обязательные requests/limits, разрешённые registry и другие правила.

Подробный пример выполнения

В примере создаётся политика, запрещающая контейнерные образы с тегом latest. Затем добавляется проверка privileged-контейнеров.

Шаг 1. Установить Gatekeeper

Gatekeeper устанавливает CRD и admission webhook. После установки нужно дождаться готовности controller-manager.

```
    kubectl apply -f https://raw.githubusercontent.com/open-policy-agent/gatekeeper/release-3.14/deploy/gatekeeper.yaml
    kubectl get pods -n gatekeeper-system --watch
```

Шаг 2. Создать ConstraintTemplate

Template описывает новый тип Constraint и Rego-правило. Rego получает объект Kubernetes в input.review.object.

```
    apiVersion: templates.gatekeeper.sh/v1
    kind: ConstraintTemplate
    metadata:
      name: k8sdisallowlatesttag
    spec:
      crd:
        spec:
          names:
            kind: K8sDisallowLatestTag
      targets:
        - target: admission.k8s.gatekeeper.sh
          rego: |
            package k8sdisallowlatesttag
            violation[{"msg": msg}] {
              container                                    :=
input.review.object.spec.containers[_]
              endswith(container.image, ":latest")
              msg  :=  sprintf("forbidden  image  tag:  %v",
[container.image])
            }
```

Шаг 3. Создать Constraint

Constraint применяет шаблон к Pod-ресурсам в указанной области. Для начала удобно ограничиться тестовым namespace.

```
    apiVersion: constraints.gatekeeper.sh/v1beta1
    kind: K8sDisallowLatestTag
```

```
  metadata:
    name: disallow-latest-tag
  spec:
    match:
      kinds:
        - apiGroups: [""]
          kinds: ["Pod"]
```

Шаг 4. Проверить блокировку нарушения

Pod с nginx:latest должен быть отклонён admission webhook. В отчёте нужно сохранить текст ошибки.

```
  kubectl run bad-pod --image=nginx:latest
```

Шаг 5. Проверить разрешённый сценарий

Pod с фиксированным тегом должен создаться. Это важно: политика не должна блокировать корректные ресурсы.

```
  kubectl run good-pod --image=nginx:1.25
  kubectl get pods
```

Шаг 6. Добавить вторую политику

По аналогии создайте Template, проверяющий securityContext.privileged == true. После применения privileged Pod должен отклоняться.

```
  kubectl run privileged-pod --image=nginx:1.25 --
privileged
```

Порядок выполнения индивидуального задания

1) установить Gatekeeper;

2) создать ConstraintTemplate по варианту;

3) создать Constraint;

4) проверить запрещённый и разрешённый ресурсы;

5) описать структуру input.review.object;

6) сделать вывод о преимуществах и сложности Rego.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Запрет image:latest.
2	Запрет privileged: true.
3	Запрет hostNetwork: true.
4	Запрет hostPID и hostIPC.
5	Запрет hostPath volume.
6	Требование resources.requests.
7	Требование resources.limits.
8	Разрешение образов только из docker.io/library или ghcr.io заданной организации.
9	Запрет запуска контейнера от root.
10	Требование readOnlyRootFilesystem: true.
11	Запрет добавления Linux capability SYS_ADMIN.
12	Требование label owner у namespace.
13	Требование annotation security-review у Deployment.
14	Запрет Service type LoadBalancer в учебном namespace.

Содержание отчета и его форма

Отчет должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия Kubernetes, kubectl и Helm, список установленных компонентов. Результаты проверки: вывод

kubectl/helm/argocd/flux, скриншоты интерфейсов при их использовании. Ответы на контрольные вопросы.

Вопросы для защиты работы

1. Что такое admission control?
2. Как Gatekeeper подключается к Kubernetes?
3. Чем ConstraintTemplate отличается от Constraint?
4. Что такое Rego?
5. Где в input находится проверяемый Pod?
6. Почему нужно тестировать и запрещённые, и разрешённые сценарии?
7. Какие политики лучше реализовать на уровне admission webhook?
8. Какие ошибки в политиках могут нарушить работу кластера?

Лабораторная работа 7.

Политики безопасности: Kyverno

Цель работы: реализовать validate- и mutate-политики в Kubernetes-native формате YAML с помощью Kyverno.

Теоретическое обоснование

Kyverno позволяет описывать политики в YAML, близком к обычным Kubernetes-манифестам. Это удобно для команд, которые не хотят изучать отдельный язык политик.

Validate-правила проверяют ресурс и при нарушении отклоняют его. Mutate-правила изменяют ресурс до сохранения, например добавляют limits или labels.

Kyverno хорошо подходит для типовых Kubernetes-политик: запрет hostPath, запрет latest, добавление labels, генерация ConfigMap/Secret, проверка подписей образов.

Пример

В примере создаются две политики: первая запрещает hostPath volume, вторая автоматически добавляет limits контейнеру, если они не указаны.

Шаг 1. Установить Kyverno

Kyverno устанавливается в отдельный namespace и запускает admission-controller.

```
    helm repo add kyverno https://kyverno.github.io/kyverno/
    helm repo update
    helm upgrade --install kyverno kyverno/kyverno -n kyverno --create-namespace
    kubectl get pods -n kyverno --watch
```

Шаг 2. Создать validate-политику запрета hostPath

hostPath даёт Pod доступ к файловой системе узла, поэтому часто запрещается в обычных приложениях.

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: disallow-hostpath
spec:
  validationFailureAction: Enforce
  rules:
    - name: hostpath-not-allowed
      match:
        any:
          - resources:
              kinds: ["Pod"]
      validate:
        message: "hostPath volumes are not allowed"
        pattern:
          spec:
            =(volumes):
              - X(hostPath): "null"
```

Шаг 3. Проверить блокировку Pod

Создайте Pod с hostPath. Kyverno должен отклонить его и показать сообщение из политики.

```
kubectl apply -f bad-hostpath-pod.yaml
```

Шаг 4. Создать mutate-политику для limits

Mutate добавляет значения по умолчанию. Это не заменяет полноценный LimitRange, но демонстрирует механизм автоматической модификации ресурсов.

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: add-default-limits
spec:
  rules:
    - name: add-limits
      match:
        any:
          - resources:
              kinds: ["Pod"]
      mutate:
        patchStrategicMerge:
          spec:
            containers:
              - (name): "*"
                resources:
                  limits:
                    +(cpu): 200m
                    +(memory): 256Mi
```

Шаг 5. Проверить результат mutate

Создайте Pod без limits и посмотрите сохранённый spec. В нём должны появиться добавленные значения.

```
kubectl run mutate-test --image=nginx:1.25
kubectl get pod mutate-test -o yaml | grep -A6 resources
```

Порядок выполнения индивидуального задания

1) установить Kyverno;

2) создать validate-политику по варианту;

3) создать mutate- или generate-политику по варианту;

4) проверить нарушение и успешный сценарий;

5) показать итоговый YAML ресурса после mutate;

6) сравнить удобство Kyverno с Gatekeeper.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Запрет hostPath.
2	Запрет image:latest.
3	Добавление default limits.
4	Добавление label app.kubernetes.io/managed-by.
5	Добавление annotation owner.
6	Запрет privileged containers.
7	Запрет hostNetwork.
8	Требование runAsNonRoot.
9	Требование imagePullPolicy: IfNotPresent для dev.
10	Генерация ConfigMap при создании namespace.
11	Запрет Service type NodePort.
12	Требование requests и limits одновременно.
13	Запрет пустого Ingress host.
14	Проверка, что Deployment имеет минимум 2 replicas.

Содержание отчета и его форма

Отчет должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия Kubernetes, kubectl и Helm, список установленных компонентов. Результаты проверки: вывод kubectl/helm/argocd/flux, скриншоты интерфейсов при их использовании. Ответы на контрольные вопросы.

Вопросы для защиты работы

1. Что такое Kyverno?
2. Чем validate отличается от mutate?
3. Что делает validationFailureAction: Enforce?
4. Когда mutate-политика полезнее validate?
5. Почему hostPath опасен?
6. Как проверить, что mutate действительно изменил ресурс?
7. Чем Kyverno проще Gatekeeper?
8. Какие политики не стоит автоматически исправлять через mutate?

Лабораторная работа 8.

Nginx Ingress Controller и Cert-Manager

Цель работы: настроить входящий HTTP/HTTPS-доступ к приложению через Nginx Ingress Controller и автоматическое создание TLS-сертификата через Cert-Manager.

Теоретическое обоснование

Ingress описывает правила маршрутизации HTTP/HTTPS-трафика к Service внутри кластера. Сам по себе Ingress - только объект API; для его работы нужен Ingress Controller.

Nginx Ingress Controller принимает внешний трафик и маршрутизирует его по host/path. Поведение часто настраивается аннотациями: rewrite, rate-limit, cors, ssl-redirect и другими.

Cert-Manager автоматизирует выпуск и обновление сертификатов. Он создаёт CertificateRequest, проходит ACME challenge и сохраняет сертификат в Kubernetes Secret.

Пример

В примере приложение публикуется через Ingress по host app.127.0.0.1.nip.io. Cert-Manager выпускает учебный сертификат через staging issuer или создаётся self-signed issuer, если ACME в локальной среде недоступен.

Шаг 1. Установить Nginx Ingress Controller

Контроллер создаёт Service, через который внешний трафик попадает в кластер. В minikube может потребоваться minikube tunnel или addon ingress.

```
helm upgrade --install ingress-nginx ingress-nginx \
  --repo https://kubernetes.github.io/ingress-nginx \
  --namespace ingress-nginx --create-namespace
```

```
kubectl get pods,svc -n ingress-nginx
```

Шаг 2. Развернуть тестовое приложение

Для проверки достаточно nginx Deployment и Service. Важно, чтобы Service выбирал Pod по labels.

```
kubectl create deployment web --image=nginx:1.25
kubectl expose deployment web --port=80 --target-port=80
kubectl get svc web
```

Шаг 3. Создать Ingress

Ingress связывает host и path с Service. Аннотация ssl-redirect управляет перенаправлением HTTP на HTTPS.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: web-ingress
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
spec:
  ingressClassName: nginx
  rules:
    - host: app.127.0.0.1.nip.io
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: web
```

```
                    port:
                      number: 80
```

Шаг 4. Установить Cert-Manager

Cert-Manager добавляет CRD Issuer, ClusterIssuer и Certificate. После установки нужно дождаться готовности webhook.

```
    kubectl apply -f https://github.com/cert-manager/cert-manager/releases/download/v1.13.0/cert-manager.yaml
    kubectl get pods -n cert-manager --watch
```

Шаг 5. Создать ClusterIssuer

Для локального выполнения можно использовать selfSigned issuer. Для окружения с доступным доменом используется ACME staging issuer.

```
    apiVersion: cert-manager.io/v1
    kind: ClusterIssuer
    metadata:
      name: selfsigned-issuer
    spec:
      selfSigned: {}
```

Шаг 6. Добавить TLS в Ingress

Аннотация cert-manager.io/cluster-issuer сообщает Cert-Manager, какой issuer использовать. Сертификат будет сохранён в Secret.

```
    metadata:
      annotations:
        cert-manager.io/cluster-issuer: selfsigned-issuer
    spec:
      tls:
        - hosts:
            - app.127.0.0.1.nip.io
          secretName: web-tls
```

Шаг 7. Проверить доступ

Проверяются Ingress, Certificate и Secret. Для self-signed сертификата браузер покажет предупреждение, но TLS-соединение будет создано.

```
kubectl get ingress
kubectl get certificate
kubectl get secret web-tls
curl -k https://app.127.0.0.1.nip.io
```

Порядок выполнения индивидуального задания

1) установить ingress-nginx;

2) развернуть приложение;

3) создать Ingress по варианту;

4) установить Cert-Manager;

5) создать Issuer или ClusterIssuer;

6) настроить TLS и проверить Secret с сертификатом;

7) описать, как запрос проходит от клиента до Pod.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Ingress для одного host и одного Service.
2	Ingress с двумя path: /api и /web.
3	Ingress с rewrite-target.
4	Ingress с rate-limit аннотацией.
5	Ingress с CORS-аннотациями.
6	Ingress с принудительным ssl-redirect.

№ варианта	Содержание индивидуального задания
7	Ingress с selfSigned ClusterIssuer.
8	Ingress с ACME staging ClusterIssuer.
9	Ingress для двух host на один Service.
10	Ingress для двух host на разные Service.
11	Ingress с basic-auth Secret.
12	Ingress с custom timeout аннотациями.
13	Ingress с pathType Exact и Prefix для сравнения.
14	Ingress с отдельным TLS Secret, созданным вручную.

Содержание отчета и его форма

Отчет должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия Kubernetes, kubectl и Helm, список установленных компонентов. Результаты проверки: вывод kubectl/helm/argocd/flux, скриншоты интерфейсов при их использовании. Ответы на контрольные вопросы.

Вопросы для защиты работы

1. Что такое Ingress?
2. Почему одного Ingress-ресурса недостаточно без контроллера?
3. Что такое IngressClass?
4. Как Ingress связан с Service?
5. Для чего Cert-Manager создаёт Secret?

6. Чем Issuer отличается от ClusterIssuer?

7. Чем self-signed сертификат отличается от Let’s Encrypt сертификата?

8. Что делает аннотация ssl-redirect?

Лабораторная работа 9.

Канареечный релиз через Ingress

Цель работы: реализовать canary deployment и распределение трафика между стабильной и новой версией приложения через Nginx Ingress.

Теоретическое обоснование

Канареечный релиз - это постепенный выпуск новой версии приложения для небольшой доли пользователей. Если ошибок нет, доля трафика увеличивается до 100%.

Nginx Ingress поддерживает canary-маршрутизацию через аннотации. Основной Ingress направляет трафик на стабильный Service, а canary Ingress - на Service новой версии.

Распределение можно задавать по весу, заголовку, cookie или другим признакам. В лабораторной работе основной сценарий - weighted routing.

Пример

В примере создаются две версии приложения: v1 и v2. Основной Ingress отправляет трафик на v1, а canary Ingress с весом 10 отправляет примерно 10% запросов на v2.

Рисунок 1 - Упрощённая схема выполняемого сценария

Шаг 1. Создать два Deployment

Для наглядности версии должны возвращать различимый текст. Можно использовать nginx с разными index.html через ConfigMap.

```
kubectl create deployment app-v1 --image=nginx:1.25
kubectl create deployment app-v2 --image=nginx:1.25
kubectl label deploy app-v1 version=v1
kubectl label deploy app-v2 version=v2
```

Шаг 2. Создать два Service

Service app-main выбирает Pod версии v1, Service app-canary выбирает Pod версии v2.

```
apiVersion: v1
kind: Service
metadata:
  name: app-main
spec:
  selector:
    app: app-v1
  ports:
    - port: 80
---
apiVersion: v1
kind: Service
metadata:
  name: app-canary
spec:
  selector:
    app: app-v2
```

```
  ports:
    - port: 80
```

Шаг 3. Создать основной Ingress

Основной Ingress не содержит canary-аннотаций. Он обслуживает стабильную версию.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app-main
spec:
  ingressClassName: nginx
  rules:
    - host: canary.127.0.0.1.nip.io
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: app-main
                port:
                  number: 80
```

Шаг 4. Создать canary Ingress

Canary Ingress использует тот же host и path, но отправляет часть трафика на Service новой версии.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
```

```
    name: app-canary
    annotations:
      nginx.ingress.kubernetes.io/canary: "true"
      nginx.ingress.kubernetes.io/canary-weight: "10"
  spec:
    ingressClassName: nginx
    rules:
      - host: canary.127.0.0.1.nip.io
        http:
          paths:
            - path: /
              pathType: Prefix
              backend:
                service:
                  name: app-canary
                  port:
                    number: 80
```

Шаг 5. Проверить распределение

Выполните серию запросов и подсчитайте ответы v1/v2. Распределение не будет идеально 90/10 на малом числе запросов, но тенденция должна быть заметна.

```
  for i in {1..100}; do curl -s http://canary.127.0.0.1.nip.io; done | sort | uniq -c
```

Шаг 6. Постепенно увеличить долю canary

Измените вес на 50, затем на 100. Это имитирует безопасный rollout новой версии.

```
    kubectl annotate ingress app-canary
nginx.ingress.kubernetes.io/canary-weight="50" --
overwrite
    kubectl annotate ingress app-canary
nginx.ingress.kubernetes.io/canary-weight="100" --
overwrite
```

Порядок выполнения индивидуального задания

1) создать две версии приложения;
2) создать два Service;
3) настроить основной и canary Ingress;
4) измерить распределение трафика при весах 10, 50 и 100;
5) описать, как выполнить откат на v1;
6) сделать вывод о применимости canary-релизов.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Весовой canary 90/10.
2	Весовой canary 80/20.
3	Постепенный rollout 10 -> 30 -> 60 -> 100.
4	Canary по HTTP-заголовку.
5	Canary по cookie.
6	Canary только для path /api.
7	Canary для frontend-приложения.
8	Canary для backend API.
9	Canary с отдельным ConfigMap версии v2.
10	Canary с измерением 200 запросов.

№ варианта	Содержание индивидуального задания
11	Откат canary при условной ошибке v2.
12	Сравнение canary через Ingress и через два Service без Ingress.
13	Canary с TLS Ingress.
14	Canary с rate-limit для новой версии.

Содержание отчета и его форма

Отчет должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия Kubernetes, kubectl и Helm, список установленных компонентов. Результаты проверки: вывод kubectl/helm/argocd/flux, скриншоты интерфейсов при их использовании. Ответы на контрольные вопросы.

Вопросы для защиты работы

1. Что такое canary deployment?
2. Зачем выпускать новую версию на малую долю трафика?
3. Как Nginx Ingress понимает, что Ingress является canary?
4. Почему распределение 90/10 не всегда точное на 10 запросах?
5. Как откатить canary-релиз?
6. Чем canary отличается от blue-green?
7. Какие метрики нужно наблюдать при использовании canary?
8. Почему важно иметь разные Service для v1 и v2?

Лабораторная работа 10.

Serverless с Knative Serving

Цель работы: развернуть Knative Service и изучить масштабирование приложения до нуля и обратно при поступлении HTTP-запросов.

Теоретическое обоснование

Knative Serving добавляет поверх Kubernetes serverless-слой для HTTP-сервисов. Разработчик описывает Knative Service, а платформа создаёт Configuration, Revision, Route и необходимые Deployment.

Revision фиксирует конкретную версию конфигурации сервиса. При изменении образа или параметров создаётся новая Revision. Трафик можно распределять между Revision, что позволяет делать canary-релизы.

Ключевая возможность Knative - масштабирование до нуля. Если запросов нет, Pod удаляются; при новом запросе Activator помогает поднять Pod обратно.

Пример

В примере создаётся Knative Service helloworld-go. Сервис вызывается через Kourier, затем без трафика масштабируется до нуля.

Шаг 1. Установить Knative Serving

В учебном окружении установку может выполнить преподаватель. Если установка выполняется самостоятельно, сначала ставятся CRD и core-компоненты Serving.

```
kubectl apply -f serving-crds.yaml
kubectl apply -f serving-core.yaml
kubectl get pods -n knative-serving
```

Шаг 2. Настроить сетевой слой

Knative нужен сетевой слой для маршрутизации HTTP. Один из простых вариантов - Kourier.

```
    kubectl apply -f kourier.yaml
    kubectl patch configmap/config-network \
      -n knative-serving \
      --type merge \
      -p                                  '{"data":{"ingress-
class":"kourier.ingress.networking.knative.dev"}}'
```

Шаг 3. Создать Knative Service

Knative Service похож на описание приложения, но не требует вручную создавать Deployment, Service и Ingress.

```
    apiVersion: serving.knative.dev/v1
    kind: Service
    metadata:
      name: hello
    spec:
      template:
        spec:
          containers:
            - image: gcr.io/knative-samples/helloworld-go
              env:
                - name: TARGET
                  value: "Knative"
```

Шаг 4. Проверить созданные ресурсы

После применения появятся ksvc, revision и route. Они показывают, как Knative разбивает сервис на внутренние сущности.

```
    kubectl apply -f service.yaml
    kubectl get ksvc
```

```
    kubectl get revisions
    kubectl get routes
```

Шаг 5. Вызвать сервис

В локальном кластере часто используют port-forward к Kourier service и передают Host header из URL Knative Service.

```
    kubectl port-forward -n kourier-system svc/kourier 8080:80
    curl -H "Host: hello.default.example.com" http://localhost:8080
```

Шаг 6. Наблюдать scale-to-zero

После периода без запросов Pod должен исчезнуть. Затем новый запрос снова создаст Pod.

```
    kubectl get pods -w
    # подождать без запросов
    curl -H "Host: hello.default.example.com" http://localhost:8080
```

Шаг 7. Настроить minScale и maxScale

minScale=1 отключает масштабирование до нуля для конкретного сервиса, а maxScale ограничивает верхний предел реплик.

```
metadata:
  annotations:
    autoscaling.knative.dev/minScale: "1"
    autoscaling.knative.dev/maxScale: "5"
```

Порядок выполнения индивидуального задания

1) установить или использовать готовый Knative Serving;

2) создать Knative Service;

3) проверить ksvc, revision и route;

4) вызвать сервис;

5) зафиксировать масштабирование до нуля;

6) настроить minScale/maxScale;

7) сравнить Knative Service с обычным Deployment.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	HelloWorld Go с scale-to-zero.
2	HelloWorld Python с переменной окружения TARGET.
3	Knative Service с minScale=1.
4	Knative Service с maxScale=3.
5	Смена образа и появление новой Revision.
6	Распределение трафика 80/20 между двумя Revision.
7	Настройка containerConcurrency.
8	Настройка timeoutSeconds.
9	Сервис с ConfigMap в env.
10	Сервис с Secret в env.
11	Сервис, возвращающий имя Pod.
12	Сравнение холодного старта при scale-to-zero и minScale=1.
13	Knative Service для простого FastAPI образа.
14	Knative Service с ограничениями ресурсов.

Содержание отчета и его форма

Отчет должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия Kubernetes, kubectl и Helm, список установленных компонентов. Результаты проверки: вывод

kubectl/helm/argocd/flux, скриншоты интерфейсов при их использовании. Ответы на контрольные вопросы.

Вопросы для защиты работы

1. Что такое Knative Serving?
2. Что такое Knative Service?
3. Чем Revision отличается от Deployment?
4. Что означает scale-to-zero?
5. Зачем нужен Activator?
6. Как minScale влияет на холодный старт?
7. Как распределить трафик между Revision?
8. Когда Knative лучше обычного Deployment?

Лабораторная работа 11.

Event-driven автомасштабирование с KEDA

Цель работы: настроить автомасштабирование Deployment на основе внешнего события или метрики с помощью KEDA ScaledObject.

Теоретическое обоснование

KEDA расширяет стандартный HPA и позволяет масштабировать приложения на основе внешних источников: Kafka lag, RabbitMQ queue length, Prometheus query, Redis list length, cron-расписание и других триггеров.

Основной объект KEDA - ScaledObject. Он указывает, какой Deployment масштабировать, до какого min/max числа реплик и какой trigger использовать.

В отличие от обычного HPA, KEDA может масштабировать Deployment до нуля, если источник событий пуст. Это полезно для consumers, batch-worker и фоновых обработчиков.

Пример

В примере используется Kafka-триггер: если в топике orders появляется lag, KEDA увеличивает число реплик consumer Deployment.

Шаг 1. Установить KEDA

KEDA устанавливает operator и metrics API, через который HPA получает внешние метрики.

```
    helm repo add kedacore https://kedacore.github.io/charts
    helm repo update
    helm upgrade --install keda kedacore/keda -n keda --create-namespace
    kubectl get pods -n keda
```

Шаг 2. Создать Deployment consumer

Deployment должен быть масштабируемым объектом. Для учебного примера можно использовать контейнер, который имитирует обработку сообщений.

```
    apiVersion: apps/v1
    kind: Deployment
    metadata:
      name: orders-consumer
    spec:
      replicas: 0
      selector:
        matchLabels:
          app: orders-consumer
      template:
        metadata:
          labels:
            app: orders-consumer
        spec:
          containers:
            - name: consumer
              image: busybox:1.36
              command: ["sh", "-c", "while true; do echo
consume; sleep 5; done"]
```

Шаг 3. Создать ScaledObject

ScaledObject связывает Deployment с Kafka. В metadata триггера задаются bootstrapServers, topic, consumerGroup и lagThreshold.

```
    apiVersion: keda.sh/v1alpha1
    kind: ScaledObject
    metadata:
```

```
  name: orders-consumer-scale
spec:
  scaleTargetRef:
    name: orders-consumer
  minReplicaCount: 0
  maxReplicaCount: 5
  triggers:
    - type: kafka
      metadata:
        bootstrapServers: kafka.kafka.svc:9092
        topic: orders
        consumerGroup: orders-group
        lagThreshold: "10"
```

Шаг 4. Проверить созданный HPA

KEDA создаёт или управляет HPA. Это показывает связь KEDA с нативным механизмом Kubernetes autoscaling.

```
kubectl apply -f scaledobject.yaml
kubectl get scaledobject
kubectl get hpa
```

Шаг 5. Сгенерировать нагрузку

Отправьте сообщения в топик или используйте выбранный источник событий. При росте lag KEDA должна увеличить replicas.

```
kubectl get deploy orders-consumer -w
kubectl get pods -w
```

Шаг 6. Проверить масштабирование обратно

После обработки очереди и уменьшения lag Deployment должен вернуться к minReplicaCount, например к 0.

```
kubectl describe scaledobject orders-consumer-scale
kubectl get deploy orders-consumer
```

Порядок выполнения индивидуального задания

1) установить KEDA;

2) создать Deployment worker по варианту;

3) создать ScaledObject;

4) создать или имитировать источник событий;

5) показать масштабирование вверх и вниз;

6) объяснить отличие KEDA от HPA.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Kafka trigger по lag топика orders.
2	RabbitMQ trigger по длине очереди.
3	Redis trigger по длине списка.
4	Prometheus trigger по HTTP RPS.
5	Cron trigger для включения worker по расписанию.
6	PostgreSQL trigger по результату SQL-запроса.
7	NATS JetStream trigger.
8	AWS SQS-совместимый trigger в учебном mock.
9	Azure Queue-совместимый trigger в учебном mock.
10	ScaledJob вместо ScaledObject.
11	minReplicaCount=1 и maxReplicaCount=3.

№ варианта	Содержание индивидуального задания
12	Сравнение lagThreshold 5 и 20.
13	Worker с ограничениями resources.
14	Масштабирование двух разных Deployment от двух очередей.

Содержание отчета и его форма

Отчет должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия Kubernetes, kubectl и Helm, список установленных компонентов. Результаты проверки: вывод kubectl/helm/argocd/flux, скриншоты интерфейсов при их использовании. Ответы на контрольные вопросы.

Вопросы для защиты работы

1. Что такое KEDA?
2. Что такое ScaledObject?
3. Как KEDA связана с HPA?
4. Чем event-driven autoscaling отличается от CPU autoscaling?
5. Что означает minReplicaCount=0?
6. Что такое Kafka lag?
7. Почему нужно ограничивать maxReplicaCount?
8. Для каких задач KEDA подходит лучше всего?

Лабораторная работа 12.

Сравнение GitOps-инструментов Flux и ArgoCD

Цель работы: сравнить ArgoCD и Flux на практическом примере синхронизации одного приложения из Git-репозитория.

Теоретическое обоснование

ArgoCD и Flux реализуют GitOps, но делают акцент на разных сценариях. ArgoCD известен удобным Web UI и объектом Application, а Flux ориентирован на Kubernetes-native CRD и тесную работу с Kustomize/Helm.

Flux использует набор контроллеров: source-controller, kustomize-controller, helm-controller, notification-controller и image-automation-controller. Каждый контроллер отвечает за отдельную часть GitOps-процесса.

Сравнение инструментов должно учитывать не только факт синхронизации, но и удобство bootstrap, модель доступа, UI, поддержку Helm/Kustomize, image automation, уведомления и поведение при drift.

Пример

В примере одно и то же приложение синхронизируется двумя способами: через ArgoCD Application и через Flux GitRepository + Kustomization.

Шаг 1. Подготовить общий каталог приложения

В Git создаётся каталог apps/compare-demo с Deployment и Service. Он будет использоваться обоими инструментами.

```
    apps/compare-demo/deployment.yaml
    apps/compare-demo/service.yaml
    git add apps/compare-demo && git commit -m "add
compare demo" && git push
```

Шаг 2. Создать ArgoCD Application

Application указывает тот же repoURL и path. После sync проверяется статус Synced/Healthy.

```
    apiVersion: argoproj.io/v1alpha1
    kind: Application
    metadata:
      name: compare-demo
      namespace: argocd
    spec:
      project: default
      source:
        repoURL:      https://github.com/example/gitops-
compare.git
        targetRevision: main
        path: apps/compare-demo
      destination:
        server: https://kubernetes.default.svc
        namespace: argo-demo
      syncPolicy:
        automated:
          prune: true
          selfHeal: true
        syncOptions:
          - CreateNamespace=true
```

Шаг 3. Установить или bootstrap Flux

Flux обычно устанавливается через flux bootstrap, но в учебном сценарии можно установить CLI и применить CRD/контроллеры по инструкции преподавателя.

```
    flux check
```

```
flux install
```

Шаг 4. Создать GitRepository и Kustomization

GitRepository описывает источник, а Kustomization - путь и интервал синхронизации.

```
apiVersion: source.toolkit.fluxcd.io/v1
kind: GitRepository
metadata:
  name: compare-repo
  namespace: flux-system
spec:
  interval: 1m
  url: https://github.com/example/gitops-compare.git
  ref:
    branch: main
---
apiVersion: kustomize.toolkit.fluxcd.io/v1
kind: Kustomization
metadata:
  name: compare-demo
  namespace: flux-system
spec:
  interval: 1m
  path: ./apps/compare-demo
  prune: true
  sourceRef:
    kind: GitRepository
    name: compare-repo
  targetNamespace: flux-demo
```

Шаг 5. Проверить drift и обновление

Измените replicas в Git и сравните, как быстро оба инструмента применяют изменение. Затем вручную измените Deployment и посмотрите восстановление.

```
kubectl get deploy -n argo-demo
kubectl get deploy -n flux-demo
flux get kustomizations
kubectl get app -n argocd
```

Порядок выполнения индивидуального задания

1) подготовить одно приложение в Git;

2) синхронизировать его через ArgoCD;

3) синхронизировать его через Flux;

4) выполнить изменение в Git и ручной drift;

5) заполнить таблицу сравнения;

6) сформулировать, в каких сценариях удобнее ArgoCD, а в каких Flux.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Plain YAML приложение.
2	Kustomize приложение.
3	Helm chart приложение.
4	Два окружения dev/prod.
5	Проверка drift self-heal.
6	Проверка prune при удалении Service.
7	Сравнение диагностики ошибок YAML.
8	Сравнение работы с private repo.
9	Сравнение image automation.

№ варианта	Содержание индивидуального задания
10	Сравнение Helm values.
11	Сравнение bootstrap-процесса.
12	Сравнение мультитенантности.
13	Сравнение UI/CLI подхода.
14	Сравнение времени синхронизации при interval 1m и 5m.

Содержание отчета и его форма

Отчет должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия Kubernetes, kubectl и Helm, список установленных компонентов. Результаты проверки: вывод kubectl/helm/argocd/flux, скриншоты интерфейсов при их использовании. Ответы на контрольные вопросы.

Вопросы для защиты работы

1. Что общего у Flux и ArgoCD?
2. В чём различие подхода Application и GitRepository/Kustomization?
3. Почему ArgoCD часто удобен для демонстрации через UI?
4. Какие контроллеры есть во Flux?
5. Как оба инструмента обрабатывают drift?
6. Что такое prune?
7. В каких случаях Flux может быть предпочтительнее?
8. В каких случаях ArgoCD может быть предпочтительнее?

Лабораторная работа 13.

FinOps: ResourceQuota и LimitRange

Цель работы: настроить ограничения ресурсов namespace с помощью ResourceQuota и LimitRange и проверить предотвращение перерасхода CPU/Memory.

Теоретическое обоснование

FinOps в Kubernetes начинается с управляемого потребления ресурсов. Если приложения не задают requests и limits, планировщик не может корректно распределять нагрузку, а один сервис может вытеснить другие.

ResourceQuota ограничивает суммарное потребление namespace: количество Pod, Service, PVC, requests и limits. LimitRange задаёт значения по умолчанию и допустимые диапазоны для отдельных Pod/Container.

Эти механизмы помогают бороться с проблемой шумного соседа, когда одно приложение потребляет непропорционально много ресурсов и ухудшает работу остальных.

Подробный пример выполнения

В примере создаётся namespace team-a с квотой requests.cpu=1, requests.memory=1Gi, limits.cpu=2, limits.memory=2Gi и максимумом 5 Pod. Затем проверяется Pod без ресурсов и Pod, превышающий квоту.

Шаг 1. Создать namespace

Квоты применяются на уровне namespace, поэтому для эксперимента нужен отдельный namespace.

```
kubectl create namespace team-a
```

Шаг 2. Создать ResourceQuota

ResourceQuota задаёт суммарный предел ресурсов в namespace. При превышении Kubernetes отклонит создание нового объекта.

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: team-a-quota
  namespace: team-a
spec:
  hard:
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
    pods: "5"
    services: "5"
```

Шаг 3. Создать LimitRange

LimitRange добавляет значения по умолчанию, если контейнер создан без requests/limits.

```
apiVersion: v1
kind: LimitRange
metadata:
  name: default-limits
  namespace: team-a
spec:
  limits:
    - type: Container
      default:
        cpu: 200m
```

```
            memory: 256Mi
          defaultRequest:
            cpu: 100m
            memory: 128Mi
```

Шаг 4. Проверить Pod без ресурсов

Создайте Pod без resources и посмотрите его итоговый YAML. Kubernetes должен добавить значения из LimitRange.

```
    kubectl run no-resources --image=nginx:1.25 -n team-a
    kubectl get pod no-resources -n team-a -o yaml | grep -A8 resources
```

Шаг 5. Проверить Pod в пределах квоты

Pod с requests.cpu=800m должен создаться, если суммарная квота ещё не превышена.

```
    kubectl apply -f pod-ok.yaml
    kubectl describe resourcequota team-a-quota -n team-a
```

Шаг 6. Проверить превышение квоты

Pod с requests.cpu=1500m должен быть отклонён, потому что превышает доступный requests.cpu.

```
    kubectl apply -f pod-too-big.yaml # ожид. ошибка exceeded quota
```

Шаг 7. Проанализировать использование

В отчёте нужно показать used/hard и объяснить, почему конкретный Pod был разрешён или запрещён.

```
    kubectl describe resourcequota -n team-a
```

```
kubectl describe limitrange -n team-a
```

Порядок выполнения индивидуального задания

1) создать namespace по варианту;

2) создать ResourceQuota;

3) создать LimitRange;

4) проверить Pod без resources;

5) проверить Pod в пределах квоты;

6) проверить Pod с превышением квоты;

7) сделать вывод о связи requests/limits и FinOps.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Namespace frontend: небольшие CPU/memory лимиты.
2	Namespace backend: больше CPU, меньше Pod.
3	Namespace analytics: больше memory, ограничение Pod.
4	Namespace dev: мягкие дефолты LimitRange.
5	Namespace stage: строгие limits и малое число Service.
6	Квота только на Pod и Service.
7	Квота на requests/limits CPU.
8	Квота на requests/limits memory.
9	LimitRange с min/max CPU.

№ варианта	Содержание индивидуального задания
10	LimitRange с min/max memory.
11	Проверка превышения количества Pod.
12	Проверка превышения количества Service.
13	Сравнение Pod с resources и без resources.
14	Два namespace с разными профилями ресурсов.

Содержание отчета и его форма

Отчет должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия Kubernetes, kubectl и Helm, список установленных компонентов. Результаты проверки: вывод kubectl/helm/argocd/flux, скриншоты интерфейсов при их использовании. Ответы на контрольные вопросы.

Вопросы для защиты работы

1. Что такое ResourceQuota?
2. Что такое LimitRange?
3. Чем request отличается от limit?
4. Почему Pod без requests/limits нежелателен?
5. Что означает exceeded quota?
6. Как ResourceQuota помогает FinOps?
7. Что такое проблема шумного соседа?
8. Почему слишком маленькие лимиты тоже опасны?

Лабораторная работа 14.

Сквозной проект: GitOps-платформа для микросервисов с mTLS

Цель работы: интегрировать Helm, ArgoCD, политики безопасности, TLS/mTLS и автомасштабирование в единую учебную GitOps-платформу.

Теоретическое обоснование

Итоговая лабораторная объединяет технологии курса. Студент должен показать не отдельный манифест, а связанный сценарий: приложение упаковано в Helm, доставляется через GitOps, защищается политиками, публикуется через TLS и масштабируется по событиям.

mTLS означает mutual TLS: не только клиент проверяет сервер, но и сервер проверяет клиентский сертификат. Это повышает доверие между сервисами или между клиентом и Ingress.

Сквозной проект должен быть воспроизводимым: вся конфигурация хранится в Git, секреты не публикуются в открытом виде, а результат можно развернуть заново на чистом namespace.

Подробный пример выполнения

В примере строится минимальная платформа из двух микросервисов: product-service и order-service. Оба упакованы в Helm-чарты, синхронизируются ArgoCD, защищены Kyverno-политиками, публикуются через Ingress с TLS, а order-worker масштабируется через KEDA.

Шаг 1. Спроектировать структуру репозитория

Репозиторий должен отделять приложения, чарты и окружения. Это упрощает ревью и повторное развёртывание.

```
repo/
  charts/
```

```
        product-service/
        order-service/
      environments/
        dev/
          product-values.yaml
          order-values.yaml
          application.yaml
      policies/
        kyverno-disallow-latest.yaml
        kyverno-add-limits.yaml
```

Шаг 2. Упаковать сервисы в Helm

Каждый сервис получает свой chart. В values задаются image, tag, resources, service и ingress.

```
    helm create charts/product-service
    helm create charts/order-service
    helm lint charts/product-service
    helm template product charts/product-service -f
environments/dev/product-values.yaml
```

Шаг 3. Добавить ArgoCD Application

ArgoCD должен синхронизировать окружение из Git. Для нескольких приложений можно использовать app-of-apps или отдельные Application.

```
    apiVersion: argoproj.io/v1alpha1
    kind: Application
    metadata:
      name: microservices-dev
      namespace: argocd
    spec:
      source:
```

```
        repoURL:
https://github.com/example/microservices-platform.git
        targetRevision: main
        path: environments/dev
      destination:
        server: https://kubernetes.default.svc
        namespace: micro-dev
      syncPolicy:
        automated:
          prune: true
          selfHeal: true
        syncOptions:
          - CreateNamespace=true
```

Шаг 4. Добавить политики безопасности

Минимальный набор: запрет latest и автоматическое добавление limits. Политики применяются до деплоя приложений.

```
    kubectl    apply    -f    policies/kyverno-disallow-
latest.yaml
    kubectl apply -f policies/kyverno-add-limits.yaml
```

Шаг 5. Настроить TLS на входе

Ingress получает сертификат через Cert-Manager. В учебной среде допустим self-signed issuer, в публичной - ACME staging.

```
    metadata:
      annotations:
        cert-manager.io/cluster-issuer:      selfsigned-
issuer
    spec:
      tls:
```

```
      - hosts: ["shop.127.0.0.1.nip.io"]
        secretName: shop-tls
```

Шаг 6. Настроить mTLS

Для учебного варианта можно включить проверку клиентского сертификата на Nginx Ingress. CA сохраняется в Secret, а Ingress требует клиентский сертификат.

```
  kubectl create secret generic client-ca --from-file=ca.crt=ca.crt

  metadata:
    annotations:
      nginx.ingress.kubernetes.io/auth-tls-secret: default/client-ca
      nginx.ingress.kubernetes.io/auth-tls-verify-client: "on"
```

Шаг 7. Добавить KEDA для order-worker

order-worker масштабируется от очереди или имитации события. Важно показать scale up и scale down.

```
  apiVersion: keda.sh/v1alpha1
  kind: ScaledObject
  metadata:
    name: order-worker-scale
  spec:
    scaleTargetRef:
      name: order-worker
    minReplicaCount: 0
    maxReplicaCount: 5
    triggers:
```

```
      - type: cron
        metadata:
          timezone: Europe/Moscow
          start: "0 * * * *"
          end: "10 * * * *"
          desiredReplicas: "2"
```

Шаг 8. Провести демонстрацию

Демонстрация должна показывать связанный сценарий: commit в Git, синхронизация ArgoCD, работа политик, HTTPS/mTLS и масштабирование worker.

```
kubectl get app -n argocd
kubectl get ingress,certificate -n micro-dev
kubectl get scaledobject,hpa -n micro-dev
curl -k https://shop.127.0.0.1.nip.io
```

Порядок выполнения индивидуального задания

1) создать Git-репозиторий проекта;

2) упаковать минимум два сервиса в Helm;

3) настроить ArgoCD Application или app-of-apps;

4) добавить политики безопасности;

5) настроить TLS и mTLS;

6) настроить KEDA или согласованный event-driven сценарий;

7) провести демонстрацию и подготовить итоговый отчёт.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Product + Order services с KEDA по cron.
2	Frontend + Backend с TLS и Kyverno.
3	API Gateway + два backend-сервиса с mTLS на Ingress.
4	Order Service + worker с Kafka/KEDA.
5	Notification Service + worker с RabbitMQ/KEDA.
6	Два сервиса с ArgoCD app-of-apps.
7	Два сервиса с отдельными AppProject.
8	Платформа с Gatekeeper вместо Kyverno.
9	Платформа с canary Ingress для одного сервиса.
10	Платформа с Image Updater для dev.
11	Платформа с ResourceQuota/LimitRange для namespace.
12	Платформа с self-signed mTLS.
13	Платформа с ACME staging TLS.
14	Платформа с сравнением ручного kubectl-деплоя и GitOps-деплоя.

Содержание отчета и его форма

Отчет должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия Kubernetes, kubectl и Helm, список установленных компонентов. Результаты проверки: вывод kubectl/helm/argocd/flux, скриншоты интерфейсов при их использовании. Ответы на контрольные вопросы.

Вопросы для защиты работы

1. Какие технологии интегрированы в итоговом проекте?

2. Почему конфигурация должна храниться в Git?

3. Как ArgoCD обеспечивает self-heal?

4. Зачем нужны политики безопасности в платформе?

5. Чем TLS отличается от mTLS?

6. Где хранятся сертификаты в Kubernetes?

7. Как KEDA дополняет обычный Deployment?

8. Какие риски есть при публикации секретов в Git?

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 «Контейнеризация и DevOps» для студентов направления подготовки 09.03.04 Программная инженерия
Направленность (профиль)
«Разработка и сопровождение программного обеспечения»

Ставрополь 2026

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 3
ОБЩАЯ ХАРАКТЕРИСТИКА САМОСТОЯТЕЛЬНОЙ РАБОТЫ 3
МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ИЗУЧЕНИЮ ТЕОРЕТИЧЕСКОГО МАТЕРИАЛА ..4
МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ 5
ВОПРОСЫ ДЛЯ СОБЕСЕДОВАНИЯ ПО САМОСТОЯТЕЛЬНОМУ ИЗУЧЕНИЮ .....
ЛИТЕРАТУРЫ 6
КРИТЕРИИ ОЦЕНИВАНИЯ КОМПЕТЕНЦИЙ 7

ВВЕДЕНИЕ

Настоящие методические указания определяют порядок организации и проведения самостоятельной работы студентов по дисциплине «Управление жизненным циклом инженерных систем» и являются дополнением к методическим указаниям по выполнению лабораторных работ. Самостоятельная работа рассматривается как обязательная составляющая образовательного процесса, обеспечивающая закрепление и углубление знаний, полученных на аудиторных занятиях.

Цель самостоятельной работы — сформировать у студентов устойчивые навыки самостоятельного освоения современных инструментов управления жизненным циклом контейнеризированных приложений (Helm, Kubernetes, GitOps, политики безопасности, автомасштабирование, FinOps) и подготовить их к осознанному выполнению и защите лабораторных работ.

Указания содержат общую характеристику видов самостоятельной работы, рекомендации по изучению теоретического материала, порядок подготовки к выполнению практических (лабораторных) работ, перечень вопросов для собеседования и критерии оценивания сформированности компетенций.

ОБЩАЯ ХАРАКТЕРИСТИКА САМОСТОЯТЕЛЬНОЙ РАБОТЫ

Самостоятельная работа студента по дисциплине включает проработку теоретического материала, подготовку к лабораторным работам, самостоятельное выполнение индивидуальных заданий и подготовку к их защите. Основные виды самостоятельной работы и формы контроля приведены в таблице 1.

Таблица 1 — Виды самостоятельной работы и формы контроля

Вид самостоятельной работы	Содержание деятельности	Форма контроля
Проработка теоретического материала	Изучение лекций, учебной и официальной технической документации по контейнеризации и оркестрации	Собеседование, тестирование
Подготовка к лабораторной работе	Изучение теоретического обоснования и примера, подготовка рабочего окружения, проработка порядка выполнения	Допуск к выполнению работы
Выполнение индивидуального задания	Самостоятельное выполнение варианта задания, оформление отчёта	Проверка отчёта
Подготовка к защите работы	Подготовка ответов на контрольные вопросы, обоснование принятых технических решений	Защита отчёта

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ИЗУЧЕНИЮ ТЕОРЕТИЧЕСКОГО МАТЕРИАЛА

Изучение теоретического материала рекомендуется проводить последовательно, в соответствии с порядком следования лабораторных работ. По каждой теме необходимо ознакомиться с основными понятиями, разобрать приведённый в работе пример и проверить понимание материала по вопросам для самоконтроля. Перечень тем для самостоятельного изучения приведён в таблице 2.

Таблица 2 — Темы для самостоятельного изучения теоретического материала

№	Тема для самостоятельного изучения	Вопросы для проработки
1	Создание Helm-чарта для web-приложения	назначение Helm, структура чарта, шаблоны и values
2	Helm-хуки для миграции базы данных	жизненный цикл релиза, хуки Helm и порядок их выполнения

№	Тема для самостоятельного изучения	Вопросы для проработки
3	Знакомство с Kubernetes Operators: анализ готового оператора	модель операторов, CRD и контроллеры, паттерн reconcile
4	GitOps с ArgoCD: установка и первый Application	принципы GitOps, модель работы ArgoCD, объект Application
5	GitOps с ArgoCD: мультитенантность и Image Updater	мультитенантность, проекты ArgoCD, автоматическое обновление образов
6	Политики безопасности: OPA/Gatekeeper	контроль допуска, policy-as-code, OPA и Gatekeeper
7	Политики безопасности: Kyverno	движок политик Kyverno, типы политик и их применение
8	Nginx Ingress Controller и Cert-Manager	маршрутизация трафика, Ingress, выпуск TLS-сертификатов
9	Канареечный релиз через Ingress	стратегии развёртывания, канареечные релизы
10	Serverless с Knative Serving	модель serverless, масштабирование до нуля в Knative
11	Event-driven автомасштабирование с KEDA	событийно-ориентированное масштабирование, источники событий KEDA
12	Сравнение GitOps-инструментов Flux и ArgoCD	сравнение подходов Flux и ArgoCD, критерии выбора инструмента
13	FinOps: ResourceQuota и LimitRange	управление ресурсами и затратами, квоты и лимиты (FinOps)
14	Сквозной проект: GitOps-платформа для микросервисов с mTLS	интеграция инструментов, построение GitOps-платформы, mTLS

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ

Выполнение каждой лабораторной работы осуществляется самостоятельно в соответствии с полученным вариантом индивидуального

задания. Рекомендуется придерживаться последовательности этапов, приведённой в таблице 3.

Таблица 3 — Этапы выполнения лабораторной работы

Этап	Содержание деятельности
Подготовительный	Изучение теоретического обоснования и примера, подготовка рабочего окружения (кластер Kubernetes, необходимые утилиты), уточнение варианта задания
Основной	Самостоятельное выполнение индивидуального задания, фиксация выполняемых команд, конфигураций и полученных результатов
Оформление отчёта	Подготовка отчёта в соответствии с требованиями работы: цель, ход выполнения, листинги, скриншоты, выводы
Защита	Ответы на контрольные вопросы, обоснование принятых решений и полученных результатов

Отчёт оформляется в текстовом редакторе, листинги конфигураций и команд приводятся моноширинным шрифтом. По итогам работы формулируются выводы, отражающие достигнутые результаты и приобретённые навыки.

ВОПРОСЫ ДЛЯ СОБЕСЕДОВАНИЯ ПО САМОСТОЯТЕЛЬНОМУ ИЗУЧЕНИЮ ЛИТЕРАТУРЫ

Для подготовки к собеседованию по результатам самостоятельного изучения литературы рекомендуется проработать следующие вопросы:

1. Что такое Helm и какие задачи он решает в Kubernetes? Из каких элементов состоит чарт?

2. Как устроен жизненный цикл релиза Helm? Для чего применяются хуки и в каком порядке они выполняются?

3. Что такое оператор Kubernetes? Какова роль CRD и контроллера, в чём суть паттерна reconcile?

4. Сформулируйте принципы GitOps. Как работает ArgoCD и что описывает объект Application?

5. Как обеспечивается мультитенантность в ArgoCD? Каким образом выполняется автоматическое обновление образов?

6. В чём состоит подход policy-as-code? Как работают OPA и Gatekeeper?

7. Каковы возможности Kyverno? Какие типы политик он поддерживает?

8. Каково назначение Ingress-контроллера? Как организуется выпуск TLS-сертификатов с помощью Cert-Manager?

9. Что такое канареечный релиз и как он реализуется средствами Ingress?

10. В чём особенность модели serverless? Как реализуется масштабирование до нуля в Knative?

11. Что такое событийно-ориентированное масштабирование? Какие источники событий поддерживает KEDA?

12. Сравните инструменты Flux и ArgoCD. По каким критериям выбирается инструмент GitOps?

13. Какими средствами осуществляется управление ресурсами и затратами в Kubernetes (ResourceQuota, LimitRange)?

14. Как организуется построение GitOps-платформы для микросервисов? В чём назначение mTLS?

КРИТЕРИИ ОЦЕНИВАНИЯ КОМПЕТЕНЦИЙ

Оценивание результатов самостоятельной работы осуществляется по итогам выполнения и защиты лабораторных работ, а также собеседования. Критерии оценивания приведены в таблице 4.

Таблица 4 — Критерии оценивания

Оценка	Критерии оценивания
«Отлично»	Задание выполнено в полном объёме и без ошибок; отчёт оформлен в соответствии с требованиями; студент свободно

Оценка	Критерии оценивания
	владеет теоретическим материалом, обоснованно отвечает на все вопросы и объясняет принятые решения
«Хорошо»	Задание выполнено в полном объёме с незначительными недочётами; отчёт оформлен с небольшими замечаниями; студент владеет материалом, но допускает неточности в ответах
«Удовлетворительно»	Задание выполнено частично; отчёт содержит ошибки и недочёты; студент испытывает затруднения при ответах на вопросы
«Неудовлетворительно»	Задание не выполнено или выполнено с грубыми ошибками; отчёт отсутствует или не соответствует требованиям; студент не владеет теоретическим материалом