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 «Продвинутая разработка программного обеспечения» для студентов направления подготовки 09.04.04 Программная инженерия
Направленность (профиль)
«Разработка, мониторинг и управление жизненным циклом инженерных систем»

Ставрополь 2026

Оглавление

ВВЕДЕНИЕ

Настоящие методические указания содержат лабораторные работы по дисциплине **«Продвинутая разработка программного обеспечения»** для студентов магистратуры, обучающихся по направлению подготовки **09.04.04 «Программная инженерия»** (профиль «Разработка, мониторинг и управление жизненным циклом инженерных систем»).

В современной индустрии разработки программного обеспечения способность быстро, надёжно и безопасно доставлять изменения в продукт стала критическим конкурентным преимуществом. Практики непрерывной интеграции (Continuous Integration, CI) и непрерывной поставки/развертывания (Continuous Delivery/Deployment, CD) являются стандартом де-факто для организаций, стремящихся к высокой скорости вывода продукта на рынок при сохранении качества и устойчивости систем.

Особую значимость эти практики приобретают в контексте инженерных систем - от распределённых веб-сервисов до встроенных автономных устройств (БПЛА, робототехника), где надёжность и безопасность обновлений имеют критическое значение.

Целью выполнения лабораторных работ является формирование у студентов практических навыков проектирования, реализации, оптимизации и сопровождения сквозных конвейеров CI/CD для программных систем и встроенного ПО с интеграцией практик автоматизированного тестирования, проверки безопасности, управления артефактами и конфигурациями.

В процессе выполнения лабораторных работ студент должен решить следующие задачи:

1. Научиться настраивать инструменты статического анализа кода и управления качеством (SonarQube, ESLint/Flake8) с интеграцией в CI/CD пайплайн.
2. Освоить автоматизацию unit-тестирования и измерения покрытия кода (pytest, pytest-cov, JUnit) с установкой пороговых значений качества.

3. Приобрести навыки статического анализа безопасности (SAST) с использованием инструментов Bandit, SpotBugs и интеграции их в конвейер.

4. Изучить принципы рефакторинга и SOLID, научиться контролировать их соблюдение автоматизированными средствами.

5. Освоить применение паттернов проектирования GoF и принципов чистой архитектуры для создания тестируемых и сопровождаемых компонентов пайплайнов.

6. Научиться использовать функциональное программирование для построения предсказуемых этапов обработки данных в пайплайне.

7. Приобрести навыки асинхронного и многопоточного программирования для ускорения выполнения этапов CI/CD.

8. Освоить практики управления конфигурациями как кодом (Configuration as Code) и безопасного хранения секретов (переменные CI/CD, HashiCorp Vault).

9. Научиться собирать, версионировать (SemVer) и публиковать артефакты (Python wheel, Docker-образы) в бинарные репозитории.

10. Освоить деплой приложений в Kubernetes через CI/CD с использованием стратегий Blue-Green и Canary.

11. Изучить GitOps подход и инструмент синхронизации ArgoCD.

12. Научиться настраивать наблюдаемость пайплайна (Prometheus, Grafana) и проверять SLI/SLO после деплоя.

13. Разработать интегрированный сквозной CI/CD пайплайн для выбранного приложения, объединяющий все изученные практики.

Лабораторный практикум состоит из **16 работ**, объединённых в четыре тематических блока:

Блок	№ ЛР	Тема
Блок 1. Качество, тестирование и безопасность	1–3	Статический анализ, unit-тесты, SAST, рефакторинг, SOLID

Блок	№ ЛР	Тема
Блок 2. Архитектура и паттерны	4–6	Паттерны GoF, чистая архитектура, функциональное программирование
Блок 3. Производительность и конфигурации	7–9	Асинхронность/параллелизм, конфигурации и секреты, Docker-образы
Блок 4. Продвинутый CD и наблюдаемость	10–16	Полный пайплайн, Kubernetes, GitOps, стратегии развертывания, мониторинг, финальный проект

Каждая лабораторная работа содержит следующие разделы:

- Тема - краткое наименование работы.
- Цель - формулировка ожидаемого результата выполнения.
- Задачи - перечень конкретных шагов, которые необходимо выполнить.
- Краткое теоретическое обоснование - основные понятия и принципы.
- 15 индивидуальных вариантов - разноуровневые задания для самостоятельного выполнения.
- Контрольные вопросы - для проверки понимания материала.
- Требования к отчёту - структура и содержание отчёта.
- Критерии оценивания - шкала и показатели для выставления оценки.
- Программное обеспечение -перечень инструментов, доступных в Российской Федерации.

Формируемая компетенция

Выполнение лабораторных работ направлено на формирование **профессиональной компетенции ПК-1**:

ПК-1. Способен проектировать, реализовывать, оптимизировать и поддерживать сквозные конвейеры непрерывной интеграции, поставки и развертывания (CI/CD) для программного обеспечения и встроенных систем, интегрируя практики автоматизированного тестирования, проверки безопасности, управления артефактами и конфигурациями.

Перечень программного обеспечения

В лабораторных работах используется программное обеспечение, доступное на территории Российской Федерации:

- Системы CI/CD: SourceCraft (Яндекс), GitLab CE, GitFlic.
- Язык программирования: Python 3.9+ (с возможностью замены на Java).
- Тестирование: pytest, pytest-cov, unittest.
- Статический анализ: SonarQube Community Edition, Bandit, Flake8.
- Контейнеризация: Docker, Docker Compose.
- Оркестрация: Minikube (Kubernetes), kubectl.
- GitOps: ArgoCD.
- Мониторинг: Prometheus, Grafana.
- Управление артефактами: Sonatype Nexus OSS, GitLab Registry, SourceCraft Container Registry.
- Управление секретами: HashiCorp Vault, переменные CI/CD.
- Нагрузочное тестирование: k6.

Организация выполнения лабораторных работ

1. Работы выполняются последовательно в порядке возрастания номеров, так как каждая последующая работа опирается на результаты предыдущих.

2. Студент получает индивидуальный вариант от преподавателя (номер варианта соответствует номеру студента в списке группы по модулю 15).

3. Выполнение работ предполагает самостоятельное написание кода, конфигурационных файлов и скриптов автоматизации.

4. Результаты каждой работы оформляются в виде отчёта, содержащего:

- титульный лист;
- цель и задачи;
- ход выполнения с краткими пояснениями;
- листинги кода и конфигураций (при необходимости — со ссылкой на репозиторий);
- скриншоты результатов выполнения пайплайна;
- анализ полученных результатов;
- ответы на контрольные вопросы;
- выводы.

5. Готовый отчёт предоставляется преподавателю для проверки, после чего работа защищается устно.

Рекомендации по успешному выполнению

Планирование времени. Каждая лабораторная работа рассчитана на 2–4 академических часа аудиторной работы и столько же на самостоятельную подготовку. Рекомендуется распределять выполнение равномерно в течение семестра.

Создание репозитория. Для всех работ рекомендуется создать единый Git-репозиторий, в котором будут сохраняться все версии кода, конфигураций и отчётов.

Настройка CI/CD. Начиная с первой работы, целесообразно постепенно наращивать пайплайн, добавляя в него новые этапы. К финальной работе (№16) пайплан должен стать полноценным сквозным конвейером.

Документирование. Фиксируйте принятые архитектурные решения, сложности, с которыми столкнулись, и способы их преодоления — это поможет при защите и при выполнении финального проекта.

Консультации. При возникновении вопросов или затруднений необходимо обращаться к преподавателю — как в аудиторное время, так и дистанционно (через электронную почту или мессенджеры).

Требования к техническому обеспечению

Для выполнения лабораторных работ потребуется:

- Персональный компьютер с процессором не менее 4 ядер, ОЗУ от 8 ГБ (рекомендуется 16 ГБ), свободным дисковым пространством от 50 ГБ.
- Операционная система: Linux (рекомендуется Ubuntu 20.04+), macOS или Windows 10/11 с WSL2.
- Установленные Docker и Docker Compose.
- Minikube (для работ с Kubernetes).
- Git и учётная запись на платформе SourceCraft / GitLab / GitHub.
- Python 3.9+ и среда разработки (VS Code, PyCharm или аналоги).

Лабораторная работа №1

Статический анализ кода, unit-тесты и покрытие в CI/CD

Цель: Настроить в CI/CD автоматический запуск статического анализа, unit-тестов и измерения покрытия кода с блокировкой сборки при нарушении Quality Gate.

Задачи:

1. Развернуть SonarQube (локально или в Docker).
2. Создать тестовый проект на Python с преднамеренными ошибками и «запахами кода».
3. Написать unit-тесты с использованием pytest и измерить покрытие (pytest-cov).
4. Настроить CI/CD пайплайн (SourceCraft/GitLab CI) с этапами: тесты → статический анализ → проверка покрытия.
5. Настроить Quality Gate с порогами (покрытие ≥ 80%, отсутствие критических ошибок).
6. Добиться автоматической блокировки сборки при нарушении Quality Gate.

Краткое теоретическое обоснование

Статический анализ кода позволяет выявлять ошибки, «запахи кода» и уязвимости без выполнения программы. SonarQube - платформа для непрерывной проверки качества. Unit-тесты проверяют отдельные компоненты, покрытие кода показывает долю протестированного кода. В парадигме CI/CD эти практики встраиваются в пайплайн для автоматического контроля качества.

Индивидуальные задания

Каждый вариант - тип «запахов кода» и целевая метрика покрытия

№	Тип задач в коде	Целевое покрытие	Особое правило Quality Gate
1	Длинные методы (>30 строк)	≥85%	Сложность метода ≤10

№	Тип задач в коде	Целевое покрытие	Особое правило Quality Gate
2	Дублирование кода	≥80%	Дублирование <5%
3	Магические числа	≥75%	Комментарии в ключевых местах
4	Неиспользуемые переменные	≥90%	Нет неиспользуемых параметров
5	Глубокое ветвление (if>4)	≥80%	Глубина ветвления ≤4
6	Слишком много параметров (>5)	≥85%	Аргументов ≤5
7	Игнорирование исключений	≥80%	Нет пустых catch/except
8	Мутируемые значения по умолчанию	≥75%	Нет mutable defaults
9	Глобальные переменные	≥90%	Максимум 2 глобальные переменные
10	Переопределение встроенных имен	≥85%	Нет переопределений
11	Цикломатическая сложность >15	≥80%	Сложность ≤15

№	Тип задач в коде	Целевое покрытие	Особое правило Quality Gate
12	Небезопасные функции (eval, exec)	≥90%	Запрещены eval/exec
13	Жёстко закодированные пути	≥75%	Пути через конфиги
14	Отсутствие type hints	≥85%	Type hints во всех функциях
15	Неоптимальные операции с большими данными	≥80%	O(n) вместо $O(n^2)$

Контрольные вопросы:

1. Что такое Quality Gate и зачем он нужен в CI/CD?
2. Какие метрики качества кода вы знаете?
3. В чём разница между statement coverage, branch coverage и function coverage?
4. Как SonarQube интегрируется с GitLab CI / SourceCraft?
5. Почему покрытие 100% не гарантирует отсутствие ошибок?
6. Какие типичные «запахи кода» выявляет статический анализ?
7. Как настроить автоматический запуск тестов при push в репозиторий?

Требования к отчету:

- Титульный лист
- Код проекта (ссылка на репозиторий)
- Скриншоты пайплайна с успешным/неуспешным прохождением
- Скриншот отчёта SonarQube с метриками

- Вывод о достижении Quality Gate

Критерии оценивания (макс. 10 баллов):

Критерий	Баллы
Проект содержит вариантные «запахи кода»	2
Unit-тесты написаны и проходят	2
Покрытие соответствует варианту	2
SonarQube настроен и выдает отчёт	2
CI/CD пайплайн работает, Quality Gate настроен	2

Программное обеспечение (РФ):

- SonarQube Community Edition
- Docker / Docker Compose
- SourceCraft (Яндекс) или GitLab CE
- Python 3.9+, pytest, pytest-cov
- Git

Лабораторная работа №2

SAST (статическая проверка безопасности) в CI/CD

Цель: Интегрировать инструмент статического анализа безопасности (Bandit) в CI/CD пайплайн для обнаружения уязвимостей на ранних этапах.

Задачи:

1. Написать код с типовыми уязвимостями (инъекции, небезопасное хранение, уязвимые зависимости).
2. Настроить Bandit с профилями безопасности.
3. Интегрировать Bandit в CI/CD пайплайн.

4. Настроить автоматическую остановку пайплайна при обнаружении уязвимостей высокого уровня.

5. Сформировать отчёт с классификацией уязвимостей (CWE, CVSS).

Краткое теоретическое обоснование

SAST (Static Application Security Testing) анализирует исходный код на наличие уязвимостей без его выполнения. Bandit - инструмент для Python. В DevSecOps SAST встраивается в CI/CD для обеспечения безопасности на этапе разработки. Уязвимости классифицируются по CWE (Common Weakness Enumeration) и критичности.

Индивидуальные задания

Каждый вариант - тип уязвимости

№	Основная уязвимость (CWE)	Дополнительная уязвимость
1	SQL инъекция (CWE-89)	Жёстко закодированный пароль
2	Command injection (CWE-78)	Небезопасная загрузка файлов
3	Path traversal (CWE-22)	Отсутствие проверки доступа
4	XSS (CWE-79)	Вывод непроверенных данных
5	Небезопасное хранение паролей (CWE-256)	Использование MD5
6	Уязвимости deserialization (CWE-502)	Pickle с внешними данными

№	Основная уязвимость (CWE)	Дополнительная уязвимость
7	Утечка информации в логах (CWE-532)	Логирование паролей
8	Небезопасное использование eval (CWE-95)	exec с пользовательским вводом
9	Отсутствие защиты CSRF (CWE-352)	Нет токенов
10	Небезопасная генерация случайных чисел (CWE-330)	random вместо secrets
11	Открытые перенаправления (CWE-601)	Нет валидации URL
12	Гонка состояний (CWE-362)	Неатомарные операции с файлами
13	Небезопасная обработка исключений (CWE-209)	Подробные сообщения клиенту
14	Уязвимые зависимости (CWE-1104)	Устаревшая библиотека
15	Отсутствие таймаутов (CWE-400)	Бесконечное ожидание

Контрольные вопросы:

1. Чем SAST отличается от DAST?

2. Какие типы уязвимостей может найти статический анализ?
3. Что такое CWE и CVSS?
4. Как Bandit определяет уровень критичности уязвимости?
5. Почему SAST не может найти все уязвимости?
6. Как интегрировать проверку зависимостей (safety, pip-audit)?
7. Что такое DevSecOps?

Требования к отчету:

- Код с уязвимостями по варианту
- Конфигурация Bandit (bandit.yaml)
- Скриншот пайплайна с обнаружением уязвимостей
- Отчёт Bandit (JSON/HTML)
- Обоснование блокировки/пропуска

Критерии оценивания (макс. 10 баллов):

Критерий	**Баллы**
Код содержит требуемые уязвимости	2
Bandit настроен корректно	2
CI/CD пайплайн запускает Bandit	2
Блокировка при высокой критичности работает	2
Отчёт сформирован и проанализирован	2

ПО:

- Python 3.9+, Bandit
- SourceCraft/GitLab CI
- Docker (опционально)
- safety/pip-audit (для зависимостей)

Лабораторная работа №3

Рефакторинг и SOLID в CI/CD

Цель: Применить рефакторинг и SOLID принципы для улучшения качества кода с автоматическим контролем в CI/CD пайплайне.

Задачи:

1. Получить проект с нарушением SOLID принципов и «запахами кода».
2. Выполнить рефакторинг для соответствия SOLID.
3. Настроить CI/CD с автоматическим запуском статического анализа до и после.
4. Сравнить метрики качества (SonarQube) до и после.
5. Доказать улучшение тестируемости через unit-тесты.

Краткое теоретическое обоснование

SOLID = пять принципов объектно-ориентированного проектирования.

Рефакторинг - процесс улучшения структуры кода без изменения внешнего поведения. В CI/CD автоматический статический анализ позволяет отслеживать динамику качества. Улучшение тестируемости - следствие соблюдения SOLID (особенно DIP).

Индивидуальные задания

каждый вариант - исходный «анти-SOLID» код

№	Нарушенный принцип	Характер проблемы
1	SRP	Класс содержит бизнес-логику, логирование, сохранение в БД
2	OCP	Функция с цепочкой if/elif для каждого нового типа
3	LSP	Класс-наследник сужает предусловия (требует

№	Нарушенный принцип	Характер проблемы
		больше)
4	ISP	Интерфейс с 10 методами, из которых нужны 2
5	DIP	Класс сам создаёт зависимости (new/**init**)
6	SRP + OCP	Комбинированное нарушение
7	LSP + ISP	Комбинированное нарушение
8	Все пять принципов	Полный анти-шаблон «божественный класс»
9	DIP + SRP	Высокая связанность + много ответственности
10	OCP + LSP	Проблемы с наследованием
11	SRP + ISP	Класс с широким интерфейсом и несколькими зонами ответственности
12	LSP + DIP	Наследование с нарушением контрактов
13	Только OCP	Сложная условная логика добавления функционала
14	Только DIP	Жёсткое связывание с конкретной БД/API
1	SRP + OCP +	Комплексный рефакторинг

№	Нарушенный принцип	Характер проблемы
5	DIP	

Контрольные вопросы:

1. В чём суть каждого из пяти принципов SOLID?
2. Как нарушение LSP может привести к ошибкам?
3. Почему DIP упрощает тестирование?
4. Какие метрики SonarQure показывают улучшение после рефакторинга?
5. Что такое «божественный класс» и почему он вреден?
6. Как автоматизировать проверку SOLID в CI/CD?

Требования к отчету:

- Исходный и рефакторированный код
- Скриншоты SonarQube до и после
- Unit-тесты для проверки поведения (до и после)
- Вывод об улучшении метрик

Критерии оценивания (макс. 10 баллов):

Критерий	Баллы
Рефакторинг выполнен без изменения поведения	2
Улучшены метрики SonarQube	2
SOLID принципы применены корректно	2
CI/CD пайплайн показывает разницу	2
Unit-тесты демонстрируют улучшение тестируемости	2

ПО:

- Python 3.9+

- SonarQube
- SourceCraft/GitLab CI
- pytest

Лабораторная работа №4

Паттерны GoF для CI/CD компонентов

Цель: Реализовать паттерны проектирования GoF и интегрировать их в CI/CD пайплайн для демонстрации практической пользы.

Задачи:

1. Реализовать три паттерна: Singleton (конфигурация), Observer (уведомления), Strategy (стратегия деплоя).
2. Написать unit-тесты для каждого паттерна.
3. Интегрировать реализованные компоненты в CI/CD пайплайн.
4. Показать возможность смены стратегии деплоя через конфигурацию.

Краткое теоретическое обоснование

Паттерны GoF - типовые решения повторяющихся задач проектирования. Singleton гарантирует единственный экземпляр, Observer реализует механизм подписки-уведомлений, Strategy позволяет алгоритмам варьироваться независимо от клиента.

Индивидуальные задания

набор паттернов и контекст

№	Паттерны для реализации	Дополнительное требование
1	Singleton + Observer	Логирование всех событий
2	Singleton + Strategy	Поддержка 3 стратегий
3	Observer + Strategy	Обновление стратегии по событию

№	Паттерны для реализации	Дополнительное требование
4	Factory Method + Singleton	Создание конфигураций
5	Abstract Factory + Observer	Фабрика с уведомлениями
6	Builder + Strategy	Построение сложного пайплайна
7	Prototype + Observer	Клонирование конфигураций
8	Adapter + Strategy	Адаптация стороннего API
9	Decorator + Observer	Декорирование уведомлений
10	Facade + Singleton	Фасад для CI/CD системы
11	Command + Observer	Команды для пайплайна
12	Iterator + Strategy	Итерация по стратегиям
13	State + Observer	Состояния пайплайна
14	Template Method + Factory	Шаблон пайплайна
15	Visitor + Singleton	Посетитель для конфигурации

Контрольные вопросы:

1. В чём отличие паттерна Strategy от State?
2. Зачем нужен Singleton, если он нарушает DIP?
3. Как Observer помогает в мониторинге пайплайна?

4. Какие ещё паттерны полезны для CI/CD?
5. Как протестировать паттерны изолированно?

Требования к отчету:

- Код с реализованными паттернами
- Диаграмма классов
- Unit-тесты для каждого паттерна
- Демонстрация работы в CI/CD

Критерии оценивания (макс. 10 баллов):

Критерий	**Баллы**
Паттерны реализованы корректно	3
Unit-тесты покрывают поведение	2
CI/CD демонстрирует смену стратегии	2
Код соответствует стандартам	1
Отчёт содержит диаграммы и объяснения	2

ПО:

- Python 3.9+
- SourceCraft/GitLab CI
- pytest

Docker (для стратегий деплоя)

Лабораторная работа №5

Чистая архитектура и управление артефактами

Цель: Реализовать приложение по чистой архитектуре (Hexagonal) и настроить управление артефактами сборки в CI/CD пайплайне.

Задачи:

1. Создать приложение с доменным слоем, портами и адаптерами.
2. Реализовать минимум два адаптера (in-memory, файловый/БД).
3. Настроить CI/CD для сборки Python wheel (или JAR).
4. Настроить публикацию артефакта в бинарный репозиторий (Nexus / GitLab Registry).
5. Реализовать семантическое версионирование на основе git-тегов.

Краткое теоретическое обоснование

Чистая архитектура (Hexagonal/Onion) отделяет бизнес-логику от внешних систем через порты и адаптеры. Это позволяет легко подменять внешние зависимости. Управление артефактами - ключевая практика CI/CD для версионирования и хранения результатов сборки.

Индивидуальные задания

предметная область + типы адаптеров

№	Предметная область	Доменный слой	Адаптеры
1	Управление задачами	Задачи, статусы	In-memory + файл JSON
2	Калькулятор метрик	Метрики, алерты	In-memory + PostgreSQL
3	Журнал событий	События, фильтры	In-memory + MongoDB
4	Система пользователей	Пользователи, роли	In-memory + Redis
5	Хранилище конфигов	Конфиги, версии	In-memory + etcd

№	Предметная область	Доменный слой	Адаптеры
6	Очередь сообщений	Сообщения, топики	In-memory + RabbitMQ
7	Система нотификации	Уведомления	In-memory + SMTP
8	Логгер	Логи, уровни	In-memory + файл
9	Кэш	Ключ-значение, TTL	In-memory + Redis
10	Валидатор данных	Схемы, правила	In-memory + JSON schema
11	Генератор отчётов	Отчёт, формат	In-memory + PDF/HTML
12	Мониторинг статусов	Проверки, статусы	In-memory + HTTP
13	Система рейтингов	Оценки, рейтинг	In-memory + PostgreSQL
14	Фильтрация данных	Фильтры, условия	In-memory + Pandas
15	Планировщик задач	Задачи, расписание	In-memory + cron

Контрольные вопросы:

1. В чём разница между портом и адаптером в Hexagonal архитектуре?

2. Почему доменный слой не должен зависеть от фреймворков?
3. Что такое семантическое версионирование (SemVer)?
4. Зачем нужен бинарный репозиторий?
5. Как автоматически определить версию сборки по git?

Требования к отчету:

- Код с разделением на слои
- Конфигурация CI/CD для сборки wheel
- Доказательство публикации артефакта
- Скриншот версионирования

Критерии оценивания (макс. 10 баллов):

Критерий	**Баллы**
Архитектура соблюдена (слои, порты)	3
Два адаптера работают	2
CI/CD собирает и публикует артефакт	3
SemVer настроен	2

ПО:

- Python 3.9+ (setup.py/pyproject.toml)
- Sonatype Nexus OSS / GitLab Registry
- SourceCraft/GitLab CI
- Docker

Лабораторная работа №6

Функциональное программирование для пайплайнов

Цель: Использовать принципы функционального программирования (чистые функции, иммутабельность) для построения надежных этапов CI/CD.

Задачи:

1. Реализовать модуль обработки данных (логи/метрики) с использованием чистых функций.
2. Использовать неизменяемые структуры данных.
3. Построить цепочку композиции функций для трансформации данных.
4. Протестировать модуль без использования моков.
5. Интегрировать модуль в CI/CD пайплайн.

Краткое теоретическое обоснование

Функциональное программирование оперирует чистыми функциями (без побочных эффектов) и неизменяемыми данными. Это даёт предсказуемость, лёгкость тестирования и потокобезопасность. Композиция функций позволяет строить сложные преобразования.

Индивидуальные задания

тип данных и операция

№	Тип входных данных	Цепочка преобразований
1	Логи HTTP-запросов	Фильтр ошибок → Группировка по коду → Подсчёт
2	Метрики сервера	Фильтр аномалий → Нормализация → Агрегация
3	JSON-события	Валидация → Трансформация → Сортировка
4	CSV с транзакциями	Фильтр по дате → Суммирование → Форматирование
5	Массив чисел	Map (x^2) → Filter (>10) → Reduce

№	Тип входных данных	Цепочка преобразований
		(сумма)
6	Список пользователей	Filter (активные) → Map (email) → Unique
7	Дерево каталогов	Flatten → Filter (по расширению) → Group by dir
8	Временной ряд	Скользящее среднее → Выбросы → Интерполяция
9	Конфигурации	Merge (дефолты + оверрайд) → Validate → Freeze
10	Текстовые строки	Tokenize → Stop words → Stemming → Frequency
11	Данные датчиков	Калибровка → Медианный фильтр → Децимация
12	Git-логи	Парсинг → Группировка по автору → Статистика
13	Артефакты сборки	Фильтр по версии → Сортировка → Последние 5
14	Метрики тестов	Время выполнения → Топ медленных → Отчёт
15	События деплоя	Фильтр успешных → Группировка

№	Тип входных данных	Цепочка преобразований
		по сервису

Контрольные вопросы:

1. Что такое чистая функция?
2. Почему иммутабельность важна для параллелизма?
3. В чём преимущество композиции функций перед императивной цепочкой?
4. Как тестировать чистые функции без моков?
5. Какие функции высшего порядка поддерживаются в Python?

Требования к отчету:

- Код с чистыми функциями и композицией
- Unit-тесты для каждой функции (без моков)
- Демонстрация работы цепочки
- Интеграция в CI/CD

Критерии оценивания (макс. 10 баллов):

Критерий	Баллы
Все функции чистые (нет side effects)	3
Использована композиция	2
Тесты проходят без моков	2
Иммутабельность соблюдена	1
Интеграция в CI/CD продемонстрирована	2

ПО:

- Python 3.9+

- SourceCraft/GitLab CI
- pytest

Лабораторная работа №7

Асинхронность и параллелизм в CI/CD

Цель: Освоить асинхронное и многопоточное выполнение задач для ускорения пайплайнов и организации потокобезопасного доступа к ресурсам.

Задачи:

1. Реализовать асинхронную загрузку данных из нескольких источников (asyncio/aiohttp).
2. Настроить параллельные джобы в CI/CD (parallel matrix).
3. Реализовать потокобезопасный кэш с блокировками (race condition → устранение).
4. Сравнить время выполнения синхронного, асинхронного и параллельного подходов.

Краткое теоретическое обоснование

Асинхронность (asyncio) позволяет эффективно работать с I/O-операциями без блокировки потока. Многопоточность используется для CPU-bound задач. В CI/CD параллелизм джобов увеличивает пропускную способность. Race condition - проблема одновременного доступа к общему ресурсу.

Индивидуальные задания

тип нагрузки + количество источников

№	Тип нагрузки	Источники / задачи	Кэшируемый ресурс
1	I/O (HTTP)	10 API	Токены авторизации
2	I/O (файлы)	50 файлов	Список загруженных
3	I/O (БД)	5 БД	Результаты запросов

№	Тип нагрузки	Источники / задачи	Кэшируемый ресурс
4	CPU (расчёты)	8 ядер	Промежуточные данные
5	Смешанная	5 + 5	Конфигурации
6	I/O (gRPC)	4 сервиса	Метрики сервисов
7	I/O + CPU	3 + 3	Кэш результатов
8	Только CPU	Многопоточный расчёт	Счётчик выполненных
9	I/O (websocket)	20 подключений	Данные потоков
10	I/O (S3-like)	100 объектов	ETag объектов
11	I/O (Kafka)	5 топиков	Offsets
12	CPU + I/O	Матрица + сеть	Промежуточные матрицы
13	Только asyncio	200 задач	Пул соединений
14	Только threading	16 потоков	Очередь результатов
15	Parallel CI/CD	4 матрицы	Кэш зависимостей

Контрольные вопросы:

1. В чём разница между asyncio и threading?

2. Что такое GIL и как он влияет на многопоточность в Python?

3. Как возникает race condition?

4. Как обеспечить потокобезопасность кэша?

5. Какие есть стратегии параллелизации в CI/CD (matrix, parallel, stages)?

Требования к отчету:

- Код асинхронного/многопоточного решения
- Результаты замеров времени (синхр./асинхр./паралл.)
- Скриншот parallel джобов в CI/CD
- Демонстрация работы потокобезопасного кэша

Критерии оценивания (макс. 10 баллов):

Критерий	Баллы
Асинхронное решение работает	2
Многопоточное решение работает	2
Parallel CI/CD джобы настроены	2
Race condition воспроизведён и устранён	2
Измерения производительности представлены	2

ПО:

- Python 3.9+ (asyncio, aiohttp, threading)
- SourceCraft/GitLab CI (parallel matrix)
- Docker

Лабораторная работа №8

Управление конфигурациями и секретами

Цель: Настроить безопасное управление конфигурациями (Config as Code) и секретами в CI/CD пайплайне.

Задачи:

1. Организовать конфигурации окружений (dev/staging/prod) в репозитории.
2. Настроить переменные окружения и секреты в CI/CD системе.
3. Интегрировать HashiCorp Vault для динамических секретов.
4. Обеспечить маскирование секретов в логах.
5. Реализовать подстановку конфигураций на этапе деплоя.

Краткое теоретическое обоснование

Configuration as Code - практика хранения конфигураций в репозитории вместе с кодом. Секреты (пароли, ключи) не должны попадать в код. CI/CD системы предоставляют механизмы защищённых переменных. HashiCorp Vault - инструмент для централизованного управления секретами с динамической генерацией.

Индивидуальные задания

состав конфигураций и способ хранения секретов

	Окружения	Количество секретов	Интеграция Vault
	dev, prod	2 (DB pass, API key)	Да
	dev, staging, prod	3	Да
	prod only	5	Нет (только CI vars)
	dev, prod	4	Да (database и redis)
	staging, prod	6	Да (с ротацией)

	Окружения	Количество секретов	Интеграция Vault
	dev, staging	2	Нет
	prod	8	Да (динамические)
	dev, prod	3	Да (токены)
	dev, staging, prod	5	Да (TLS сертификаты)
0	prod	10	Да (HashiCorp + K8s)
1	dev, prod	4 + файлы	Нет
2	staging only	2	Да (lease duration)
3	dev, staging, prod	7	Да (подход AppRole)
4	prod	4	Да (KV v2 engine)
5	dev, prod	5 + SSH ключи	Да (SSH secrets)

Контрольные вопросы:

1. Почему нельзя хранить секреты в репозитории?

2. Чем отличаются переменные CI/CD системы от Vault?
3. Что такое динамические секреты и зачем они нужны?
4. Как защитить секреты от вывода в лог?
5. Что такое «политика минимальных привилегий» для секретов?

Требования к отчету:

- Файлы конфигураций окружений
- Скриншот настроенных CI/CD переменных
- Демонстрация интеграции с Vault
- Лог выполнения без раскрытия секретов

Критерии оценивания (макс. 10 баллов):

Критерий	Баллы
Config as Code организован	2
Секреты не попали в код	2
CI/CD переменные настроены	2
Vault интеграция работает	2
Логи не содержат секретов	2

ПО:

- SourceCraft/GitLab CI (переменные)
- HashiCorp Vault (Open Source)
- Docker
- Python (hvac библиотека)

Лабораторная работа №9

Управление артефактами: Docker-образы и registry

Цель: Настроить сборку, версионирование и публикацию Docker-образов в container registry как части CI/CD пайплайна.

Задачи:

1. Написать multi-stage Dockerfile для минимизации размера образа.
2. Настроить автоматическую сборку образа в CI.
3. Публиковать образ в container registry с тегами (latest, git-tag, commit-hash).
4. Реализовать кэширование слоёв Docker в CI/CD.
5. Доказать уменьшение размера образа за счёт multi-stage.

Краткое теоретическое обоснование

Docker-образы - стандартный формат поставки приложений. Multi-stage build позволяет отделить среду сборки от runtime, уменьшая размер. Container registry (Docker Hub / SourceCraft Registry / GitLab Registry) — хранилище образов. Тегирование по git-метаданным обеспечивает прослеживаемость.

Индивидуальные задания

приложение и требования к образу

№	Тип приложения	Требования к образу	Особенность
1	FastAPI	<200 MB, Alpine	Poetry управление
2	Django	<500 MB, slim	Миграции на старте
3	Flask + ML	<1 GB, CUDA?	Тяжелые зависимости
4	CLI утилита	<50 MB, scratch	Статическая сборка
5	Celery worker	<300 MB	Без исходников
6	Jupyter	<1 GB	С ноутбуками
7	gRPC сервер	<250 MB	Много протобафов
8	Простой HTTP	<30 MB	distroless

№	Тип приложения	Требования к образу	Особенность
	сервер		
9	Prometheus exporter	<100 MB	На базе Go -> Python
10	Telegram бот	<150 MB	Секреты через ENV
11	WebSocket сервер	<200 MB	Поддержка масштабирования
12	Очередь задач	<250 MB	RQ + Redis
13	Статический сайт	<20 MB	Nginx + static files
14	API gateway	<150 MB	На FastAPI + gunicorn
15	Медиа-процессор	<600 MB	FFmpeg внутри

Контрольные вопросы:

1. Что такое multi-stage build и зачем он нужен?
2. Какие стратегии тегирования Docker-образов вы знаете?
3. Чем отличается image registry от image repository?
4. Как кэшировать слои Docker в CI/CD?
5. Какие практики безопасности образов существуют?

Требования к отчету:

- Dockerfile (multi-stage)
- CI/CD конфигурация сборки/публикации
- Скриншот registry с тегами
- Сравнение размера до/после multi-stage

Критерии оценивания (макс. 10 баллов):

Критерий	Баллы
Dockerfile корректен	2
Multi-stage реализован	2
Образ публикуется в registry	2
Тегирование автоматизировано	2
Размер уменьшен (доказано)	2

ПО:

- Docker / Docker Buildx
- SourceCraft Container Registry / GitLab Container Registry
- SourceCraft/GitLab CI

Лабораторная работа №10

Полный пайплайн CI/CD для микросервиса

Цель: Разработать сквозной пайплайн CI/CD «от коммита до деплоя» со всеми этапами.

Задачи:

1. Разработать простой микросервис (FastAPI).
2. Настроить пайплайн: lint → test → bandit → build → package → deploy (Docker Compose).
3. Реализовать smoke-тесты после деплоя.

4. Настроить автоматическую остановку при падении любого этапа.
5. Документировать пайплайн.

Краткое теоретическое обоснование

Сквозной CI/CD пайплайн автоматизирует путь кода от коммита до работающего приложения. Lint проверяет стиль, test - корректность, bandit - безопасность, build собирает артефакт, package упаковывает, deploy разворачивает. Smoke-тесты верифицируют, что приложение запустилось.

Индивидуальные задания

тип микросервиса + окружение деплоя

№	Микросервис	Эндпоинты	Окружение деплоя
1	Калькулятор	/add, /sub, /mul, /div	Docker Compose
2	Конвертер валют	/convert	Docker Compose
3	Генератор паролей	/generate	Local
4	Валидатор email	/validate	Docker Compose
5	Shortener URL	/shorten, /resolve	Docker Compose
6	Погодный прокси	/weather	Docker Compose
7	JWT провайдер	/token, /verify	Docker Compose
8	Логгер запросов	/log (POST)	Local
9	Проверка паролей	/check	Docker Compose
10	Таймстемп сервер	/now, /tz	Docker Compose

№	Микросервис	Эндпоинты	Окружение деплоя
1	Health checker	/health	Docker Compose
2	Echo сервер	/echo	Local
3	Форматировщик JSON	/pretty	Docker Compose
4	MD5 хэшер	/hash	Docker Compose
5	UUID генератор	/uuid	Docker Compose

Контрольные вопросы:

1. Какие этапы обязательно должны быть в CI/CD пайплайне?
2. Чем отличается smoke test от интеграционного?
3. Почему deploy нужно отделять от build?
4. Что такое «артефакт пайплайна»?
5. Как добиться идемпотентности деплоя?

Требования к отчету:

- Код микросервиса
- Конфигурация пайплайна (.gitlab-ci.yml или sourcecraft.yml)
- Скриншоты каждого этапа
- Логи успешного/неуспешного выполнения

Критерии оценивания (макс. 10 баллов):

Критерий	**Баллы**

Критерий	Баллы
Все этапы реализованы	3
Пайплайн работает автоматически	2
Smoke-тесты проходят	2
Документация пайплайна	1
Код соответствует стандартам (lint)	2

ПО:

- Python, FastAPI
- Docker Compose
- SourceCraft/GitLab CI
- pytest, bandit, flake8

Лабораторная работа №11

Оркестрация и деплой в Kubernetes через CI/CD

Цель: Настроить автоматический деплой в Kubernetes (Minikube) через CI/CD пайплайн.

Задачи:

1. Установить Minikube (локальный K8s).
2. Написать манифесты Deployment, Service.
3. Настроить CI/CD джоб на обновление образа в кластере.
4. Реализовать rolling update стратегию.
5. Проверить работоспособность после деплоя.

Краткое теоретическое обоснование

Kubernetes - платформа оркестрации контейнеров. Deployment управляет репликами и обновлениями. Service обеспечивает сетевой доступ. Rolling update - стратегия постепенного обновления подов без даунтайма. CI/CD автоматизирует применение новых манифестов.

Индивидуальные задания

количество реплик, ресурсы, порты

№	Приложение	Реплик	CPU/ Memory	Port	Rolling update параметры
1	FastAPI	2	100m/128 Mi	8000	maxSurge=1
2	Flask	3	200m/256 Mi	5000	maxUnavailable=0
3	Django	2	500m/512 Mi	8000	maxSurge=2
4	Express-like	4	100m/128 Mi	3000	maxUnavailable=1
5	gRPC	2	250m/512 Mi	50051	maxSurge=1
6	WebSocket	3	200m/256 Mi	8080	maxUnavailable=1
7	Static file	2	50m/64Mi	80	rollingUpdate
8	Celery	3	300m/512	-	minReadySec

№	Приложение	Реплик	CPU/ Memory	Port	Rolling update параметры
			Mi		onds=5
9	Redis	1	200m/256 Mi	6379	recreate strategy
10	Postgres	1	500m/1Gi	5432	recreate
11	Nginx	2	100m/128 Mi	80	maxSurge=1
12	Prometheus	2	500m/1Gi	9090	maxUnavailable=0
13	Grafana	1	200m/256 Mi	3000	recreate
14	Spring Boot	3	500m/1Gi	8080	rollingUpdate
15	.NET Core	2	400m/512 Mi	8080	maxSurge=1

Контрольные вопросы:

1. Чем Deployment отличается от ReplicaSet?
2. Как работает rolling update?
3. Что такое readinessProbe и livenessProbe?
4. Как CI/CD может обновить образ в K8s?
5. Что такое Minikube и для чего он нужен?

Требования к отчету:

- Манифесты K8s
- CI/CD джоб на деплой
- Скриншоты kubectl get pods, services
- Демонстрация rolling update

Критерии оценивания (макс. 10 баллов):

Критерий	Баллы
Minikube установлен и работает	1
Манифесты корректны	2
CI/CD деплоит в K8s	3
Rolling update успешен	2
Приложение доступно	2

ПО:

- Minikube, kubectl
- SourceCraft/GitLab CI
- Docker registry (образы)
- Python (любое приложение)

Лабораторная работа №12

GitOps: ArgoCD + CI/CD

Цель: Внедрить GitOps подход с синхронизацией кластера через ArgoCD и CI/CD для обновления манифестов.

Задачи:

1. Создать репозиторий с K8s-манифестами.
2. Установить ArgoCD в Minikube.

3. Настроить CI/CD пайплайн на обновление тега образа в манифестах.

4. Настроить автоматическую синхронизацию ArgoCD.

5. Доказать, что изменения применяются автоматически.

Краткое теоретическое обоснование

GitOps - декларативный подход, где Git является единственным источником истины для инфраструктуры. ArgoCD - инструмент непрерывной доставки для Kubernetes, синхронизирующий кластер с Git-репозиторием. CI/CD обновляет манифесты, ArgoCD применяет изменения.

Индивидуальные задания

способ обновления тега + количество сервисов

	Сервисов	Обновление тега	Стратегия синхронизации
	1	Вручную (PR)	Auto-sync
	1	Через CI (sed)	Auto-sync + Prune
	2	Через CI (yq)	Manual sync
	2	CI + Pull Request	Auto-sync
	3	CI + kustomize	Auto-sync
	1	CI + helm upgrade	Auto-sync
	2	CI + yq (оба сервиса)	Manual
	3	CI + kustomize (редакция)	Auto-sync
	1	CI + envsubst	Auto-sync
	2	CI + sed (несколько)	Auto + Prune

	Сервисов	Обновление тега	Стратегия синхронизации
0			
1	1	PR + approval	Manual
2	2	CI + helm (values)	Auto-sync
3	3	CI + скрипт	Auto-sync
4	1	CI + утилита (yq)	Auto + self-heal
5	3	CI + kustomize + helm	Auto-sync

Контрольные вопросы:

1. Чем GitOps отличается от традиционного CI/CD?
2. Как ArgoCD узнаёт об изменениях в Git?
3. Что такое auto-sync и self-heal?
4. Как обновить тег образа без ручного редактирования манифеста?
5. Какие преимущества даёт GitOps в вопросах безопасности?

Требования к отчету:

- Репозиторий манифестов
- CI/CD пайплайн обновления тега
- Скриншот ArgoCD UI с синхронизацией
- Демонстрация автоматического применения

Критерии оценивания (макс. 10 баллов):

Критерий	Баллы
ArgoCD установлен и подключен к репозиторию	2
Манифесты в репозитории	2
CI/CD обновляет тег образа	3
ArgoCD синхронизирует изменения	3

ПО:

- Minikube
- ArgoCD
- SourceCraft/GitLab CI
- Git, yq/sed/kustomize/helm
- Docker registry

Лабораторная работа №13

Стратегии развертывания (Blue-Green, Canary)

Цель: Реализовать продвинутые стратегии развертывания (Blue-Green, Canary) в CI/CD пайплайне с автоматическим откатом.

Задачи:

1. Реализовать Blue-Green деплой с переключением сервиса.
2. Реализовать Canary деплой с постепенным переводом трафика.
3. Настроить health-проверки для автоматического отката.
4. Интегрировать стратегии в CI/CD пайплайн.

Краткое теоретическое обоснование

Blue-Green - две среды (blue и green), переключение маршрутизации. Canary -— постепенное наращивание трафика на новую версию. Health-проверки позволяют автоматически откатить изменения при ошибках.

Индивидуальные задания

стратегия + tool для управления трафиком

№	Основная стратегия	Управление трафиком	Условие отката
1	Blue-Green	Kubernetes Service	health проверка
2	Blue-Green	Ingress (nginx)	500 ошибки
3	Canary (10%→50%→100%)	Istio	latency >1s
4	Canary (5%→20%→100%)	Flagger	error rate >1%
5	Blue-Green + Canary mix	nginx	таймаут 5s
6	Canary (автоматический)	Istio	статус падения
7	Blue-Green	Traefik	готовность
8	Canary (ручной)	nginx	ручной rollback
9	A/B тестирование (условное)	Istio	header-based
10	Blue-Green	Kubernetes Service + Ingress	readinessProbe
11	Canary (weighted)	nginx	код ответа 5xx
12	Blue-Green (с даунтаймом)	kubectl patch	manual
13	Canary + Shadow трафик	Istio	mirroring

№	Основная стратегия	Управление трафиком	Условие отката
14	Blue-Green с проверкой	API-проверка	автоматический
15	Canary (10 шагов)	Flagger	SLI <=99%

Контрольные вопросы:

1. В чём отличие Blue-Green от Canary?
2. Какой инструмент позволяет управлять весом трафика в K8s?
3. Как автоматически обнаружить неудачный деплой?
4. Что такое «поляризация трафика» в Canary?
5. Какую стратегию выбрать для критического платёжного сервиса?

Требования к отчету:

- Конфигурация пайплайна со стратегией
- Демонстрация переключения трафика
- Доказательство автоматического отката (если вариант с авто)
- Логи переключения

Критерии оценивания (макс. 10 баллов):

Критерий	Баллы
Стратегия развёрнута	3
Переключение/трафик работает	3
Health-проверки настроены	2
Откат (авто/ручной) работает	2

ПО:

- Minikube, kubectl

- Nginx Ingress / Istio / Flagger
- SourceCraft/GitLab CI
- Docker registry

Лабораторная работа №14

Наблюдаемость пайплайна и мониторинг деплоя

Цель: Интегрировать Prometheus + Grafana в процесс CI/CD для отслеживания состояния после деплоя и проверки SLI/SLO.

Задачи:

1. Добавить экспорт метрик в приложение (/metrics).
2. Развернуть Prometheus и Grafana.
3. Настроить сбор метрик.
4. Добавить в CI/CD этап проверки SLI (например, успешность запросов >99%).
5. Создать дашборд для визуализации состояния после деплоя.

Краткое теоретическое обоснование

Наблюдаемость (observability) включает метрики, логи и трейсы. Prometheus собирает метрики, Grafana визуализирует. SLI (Service Level Indicators) - измеряемые показатели, SLO (Service Level Objectives) - целевые значения. CI/CD может проверять SLI после деплоя.

Индивидуальные задания

SLI метрика + порог

№	SLI метрика	Пороговое значение	Дополнительно
1	HTTP успешность	>=99.9%	5xx ошибки
2	Время ответа (p95)	<=200ms	Альфа
3	Время ответа (p99)	<=500ms	API

№	SLI метрика	Пороговое значение	Дополнительно
4	Коэффициент ошибок	<=1%	all endpoints
5	Доступность (up)	100%	readiness
6	Частота запросов	>100 rpm	load check
7	Потребление CPU	<70%	per pod
8	Потребление памяти	<512Mi	limit check
9	Количество активных соединений	<1000	web
10	Успешность БД запросов	>=99.95%	БД
11	Задержка БД	<=50ms	БД
12	Ошибки Kafka	0%	consumer
13	Время старта	<=5s	startupProbe
14	Потоков	<=200	threads
15	GC паузы	<=100ms	для Java/Go

Контрольные вопросы:

1. Чем SLI отличается от SLO?
2. Какие типы метрик собирает Prometheus (Counter, Gauge, Histogram)?
3. Как приложение должно публиковать метрики?
4. Как CI/CD может проверить SLI после деплоя?
5. Что такое «error budget»?

Требования к отчету:

- Код с /metrics
- Конфигурация Prometheus
- Скриншот дашборда Grafana
- CI/CD этап проверки SLI
- Доказательство, что пайплайн валидирует метрики

Критерии оценивания (макс. 10 баллов):

Критерий	Баллы
Метрики экспортируются	2
Prometheus собирает	2
Графана отображает	2
CI/CD проверяет SLI	2
Дашборд создан	2

ПО:

- Prometheus, Grafana
- Python (prometheus_client)
- SourceCraft/GitLab CI

Kubernetes (опционально)

Лабораторная работа №15

Автоматизация тестирования в CI/CD (расширенная)

Цель: Углубить навыки автоматизированного тестирования в пайплайне: интеграционные тесты, параметризованные тесты, генерация отчётов.

Задачи:

1. Написать интеграционные тесты с реальной БД (testcontainers / docker-compose).
2. Использовать параметризованные тесты (pytest.mark.parametrize).
3. Генерировать отчёты (Allure / pytest-html).
4. Публиковать отчёты как артефакты пайплайна.
5. Настроить автоматический запуск интеграционных тестов только при определённых условиях.

Краткое теоретическое обоснование

Интеграционные тесты проверяют взаимодействие компонентов. Testcontainers позволяет поднимать реальные зависимости в контейнерах. Параметризованные тесты уменьшают дублирование. Отчёты делают результаты доступными.

Индивидуальные задания

тип интеграции + формат отчёта

	Интеграция с	Параметризация	Отчёт
	PostgreSQL	SQL-запросы	Allure
	Redis	операции	pytest-html
	MongoDB	запросы	Allure

	Интеграция с	Параметризация	Отчёт
	RabbitMQ	publish/consume	junit.xml
	Minio (S3)	upload/download	Allure
	Elasticsearch	индексация	pytest-html
	MySQL	CRUD	Allure
	Kafka	produce/consume	Allure
	ClickHouse	аналитика	pytest-html
0	Cassandra	запросы	Allure
1	Memcached	кэширование	junit.xml
2	NATS	сообщения	Allure
3	Neo4j	графовые запросы	pytest-html

	Интеграция с	Параметризация	Отчёт
4	InfluxDB	временные ряды	Allure
5	Consul	kv + service discovery	Allure

Контрольные вопросы:

1. Чем интеграционные тесты отличаются от unit и e2e?
2. Что такое testcontainers?
3. Зачем нужна параметризация тестов?
4. Как опубликовать отчёт как артефакт пайплайна?
5. Когда стоит запускать интеграционные тесты, а когда — unit?

Требования к отчету:

- Код интеграционных тестов
- Конфигурация testcontainers / docker-compose
- Сгенерированный отчёт (HTML)
- Артефакты в CI/CD

Критерии оценивания (макс. 10 баллов):

Критерий	Баллы
Интеграционные тесты работают	3
Параметризация использована	2
Отчёт генерируется	2
Отчёт публикуется как артефакт	2
Условия запуска настроены	1

ПО:

- Python 3.9+, pytest, testcontainers-python
- Allure / pytest-html
- SourceCraft/GitLab CI
- Docker

Лабораторная работа №16

Финальный интегрированный проект

Цель: Интегрировать все изученные практики в единый сквозной CI/CD пайплайн для реального (учебного) проекта.

Задачи:

1. Разработать микросервис по выбранному варианту.
2. Реализовать полный пайплайн: тесты → SAST → сборка → образ → деплой в K8s → мониторинг.
3. Внедрить GitOps (ArgoCD) для синхронизации.
4. Реализовать стратегию развертывания (Blue-Green/Canary).
5. Настроить наблюдаемость (Prometheus + Grafana) и автоматический откат при падении SLI.
6. Документировать весь пайплайн.

Краткое теоретическое обоснование

Финальный проект объединяет DevSecOps, CI/CD, GitOps, оркестрацию, наблюдаемость - полный цикл поставки современного ПО. Цель - сформировать целостное понимание и практические навыки.

Индивидуальные задания

предметная область + особенности

№	Микросервис	Доп. внешняя система	Стратегия деплоя
1	Координатор задач	PostgreSQL	Blue-Green

№	Микросервис	Доп. внешняя система	Стратегия деплоя
2	API для метрик	Redis + Prometheus	Canary
3	Система нотификаций	RabbitMQ	Blue-Green
4	Прокси-сервер	Nginx	Canary
5	Хранилище файлов	Minio	Blue-Green
6	Обработчик событий	Kafka	Canary
7	Агрегатор логов	Elasticsearch	Blue-Green
8	Gateway	Redis + валюты	Canary
9	Система сокращения ссылок	PostgreSQL + кэш	Blue-Green
10	Анализатор трафика	ClickHouse	Canary
11	Мониторинг статусов	MongoDB	Blue-Green
12	Система рейтингов	PostgreSQL + Kafka	Canary
13	Планировщик	Redis + Celery	Blue-Green
14	API погодного прокси	внешний API + кэш	Canary

№	Микросервис	Доп. внешняя система	Стратегия деплоя
15	Конфигурационный сервер	etcd	Blue-Green

Контрольные вопросы (защита проекта):

1. Какой стек технологий использован и почему?
2. Какие этапы в пайплайне критичны для безопасности?
3. Как обеспечивается отказоустойчивость деплоя?
4. Как проверяется SLI/SLO после деплоя?
5. Какие проблемы возникли при интеграции и как решались?

Требования к отчету:

- Ссылка на репозиторий (код + манифесты + конфиги пайплайна)
- Архитектурная схема
- Демонстрация работы пайплайна (скриншоты/видео)
- Результаты тестов и статического анализа
- Дашборд Grafana (скриншот)
- Документация пайплайна

Критерии оценивания (макс. 30 баллов):

Критерий	Баллы
Микросервис реализован	5
Полный CI/CD пайплайн (test, SAST, build)	5
Docker образ + K8s деплой	5
GitOps (ArgoCD) работает	3

Критерий	Баллы
Стратегия деплоя реализована	4
Наблюдаемость (Prometheus+Grafana)	4
Автоматический откат при падении SLI	2
Документация и защита	2

ПО:

- Python / FastAPI
- SourceCraft/GitLab CI
- Docker, Kubernetes (Minikube)
- ArgoCD
- Prometheus, Grafana
- SonarQube, Bandit
- Nexus / container registry
- Postman / curl (для тестов)

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Методические указания для самостоятельной работы по дисциплине **«Продвинутая разработка программного обеспечения»** предназначены для студентов магистратуры, обучающихся по направлению подготовки 09.04.04 «Программная инженерия» (профиль «Разработка, мониторинг и управление жизненным циклом инженерных систем»).

Дисциплина «Продвинутая разработка программного обеспечения» является одной из ключевых дисциплин вариативной части образовательной программы и направлена на формирование у магистрантов системных знаний и практических навыков в области проектирования, реализации, оптимизации и поддержки сквозных конвейеров непрерывной интеграции, поставки и развертывания (CI/CD) для программных систем и встроенного ПО.

Настоящие методические указания разработаны в соответствии с:

- Федеральным государственным образовательным стандартом высшего образования по направлению подготовки 09.04.04 «Программная инженерия»;
- Учебным планом направления подготовки 09.04.04 «Программная инженерия» (профиль «Разработка, мониторинг и управление жизненным циклом инженерных систем»);
- Рабочей программой дисциплины «Продвинутая разработка программного обеспечения»;
- Положением об организации образовательного процесса на основе рейтинговой системы оценки знаний студентов в ФГАОУ ВО «СКФУ».

Целью самостоятельной работы является закрепление и углубление теоретических знаний, полученных на лекционных занятиях, а также формирование практических навыков применения методов и инструментов CI/CD, автоматизированного тестирования, статического анализа

безопасности, управления артефактами и конфигурациями в рамках выполнения индивидуального сквозного проекта.

Задачи самостоятельной работы:

В процессе выполнения самостоятельной работы студент должен решить следующие задачи:

1. Изучить теоретический материал по темам дисциплины в соответствии с учебным планом.
2. Подготовиться к выполнению лабораторных работ (№1-16) и осмыслению их результатов.
3. Разработать и реализовать сквозной проект по созданию CI/CD пайплайна для выбранной системы (Лабораторная работа №16).
4. Сформировать комплексный отчёт, включающий конфигурацию пайплайна, результаты тестирования, статический анализ безопасности, управление артефактами и конфигурациями.
5. Подготовиться к промежуточной аттестации (экзамену) по дисциплине.
6. Развить навыки самостоятельного поиска и анализа научно-технической информации в области CI/CD и DevOps.

Настоящие методические указания призваны помочь студентам в организации самостоятельной работы по дисциплине «Продвинутая разработка программного обеспечения». Систематическое выполнение всех видов самостоятельной работы, своевременная подготовка к лабораторным занятиям и выполнение сквозного проекта позволят успешно освоить дисциплину и сформировать необходимые профессиональные компетенции.

1. ОБЩАЯ ХАРАКТЕРИСТИКА САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1.1. Цель самостоятельной работы

Целью самостоятельной работы является закрепление и углубление теоретических знаний, полученных на лекциях, а также формирование

практических навыков применения методов и инструментов CI/CD, автоматизированного тестирования, статического анализа безопасности, управления артефактами и конфигурациями в рамках выполнения индивидуального сквозного проекта.

1.2. Общая характеристика сквозного проекта

Сквозной проект выполняется студентом индивидуально в течение всего семестра параллельно с изучением дисциплины и выполнением лабораторных работ (№1-15). Защита проекта проводится на 16-м лабораторном занятии.

Тема проекта: «Разработка и внедрение CI/CD пайплайна полного цикла для программной системы»

Результат проекта: Комплексный отчёт, включающий:

- Конфигурацию CI/CD пайплайна (линтинг → тесты → SAST → сборка → образ → деплой);
- Настройку статического анализа и Quality Gate;
- Управление артефактами (версионирование, бинарный репозиторий);
- Управление конфигурациями и секретами;
- Стратегию развертывания (Blue-Green/Canary);
- GitOps синхронизацию (ArgoCD);
- Мониторинг пайплайна и наблюдаемость.

1.3. Формируемые компетенции

Компетенция	Индикаторы
ПК-1 Способен проектировать, реализовывать, оптимизировать и поддерживать сквозные конвейеры непрерывной	**ИД-1 ПК-1.** Проектирует архитектуру CI/CD пайплайна с учётом требований к тестированию, безопасности и развертыванию для различных типов приложений (монолитные, микросервисные, встраиваемые

Компетенция	Индикаторы
интеграции, поставки и развертывания (CI/CD) для программного обеспечения и встроенных систем, интегрируя практики автоматизированного тестирования, проверки безопасности, управления артефактами и конфигурациями	системы).
	ИД-2 ПК-1. Реализует пайплайн непрерывной интеграции, включающий этапы линтинга, автоматизированного тестирования (unit, интеграционные), статического анализа безопасности (SAST) и измерения покрытия кода.
	ИД-3 ПК-1. Настраивает автоматическую сборку, версионирование (SemVer) и публикацию артефактов (библиотеки, Docker-образы) в бинарный репозиторий.
	ИД-4 ПК-1. Интегрирует управление конфигурациями и секретами в пайплайн, обеспечивая безопасную подстановку переменных окружения для различных сред (dev/staging/prod).
	ИД-5 ПК-1. Оптимизирует производительность пайплайна за счёт параллелизации этапов, кэширования зависимостей и использования асинхронных/многопоточных подходов.
	ИД-6 ПК-1. Реализует стратегии развертывания (Blue-Green, Canary, Rolling Update) и автоматический откат при сбоях на

Компетенция	Индикаторы
	основе health-проверок.
	ИД-7 ПК-1. Внедряет GitOps подход с использованием ArgoCD для синхронизации состояния кластера с репозиторием манифестов.
	ИД-8 ПК-1. Настраивает наблюдаемость пайплайна (метрики, логи, алерты) и проверку SLI/SLO после деплоя.

1.4. Трудоёмкость самостоятельной работы

Вид работы	Часы
Изучение теоретического материала	20
Подготовка к лабораторным работам	30
Выполнение сквозного проекта	30
Подготовка к экзамену	10
Итого	**90**

2. МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ИЗУЧЕНИЮ ТЕОРЕТИЧЕСКОГО МАТЕРИАЛА

2.1. Темы для самостоятельного изучения

	Тема	Литература	Часы
	Основы CI/CD: непрерывная интеграция, поставка, развертывание	[1, гл. 1-3]	2
	Статический анализ кода и Quality Gate (SonarQube)	[2]	2
	Автоматизированное тестирование: unit, интеграционные, покрытие	[3, гл. 4-6]	2
	SAST (статический анализ безопасности) и DevSecOps	[4]	2
	Управление артефактами: бинарные репозитории, SemVer	[5]	2
	Управление конфигурациями и секретами (Vault, CI/CD variables)	[6]	2
	Асинхронность и параллелизм в Python (asyncio, threading)	[7]	2
	Docker и multi-stage сборка образов	[8]	2
	Оркестрация контейнеров: Kubernetes (Deployment, Service)	[9]	2
	Стратегии развертывания: Blue-Green,	[10]	2

	Тема	Литература	часы
0	Canary, Rolling Update		
1	GitOps и ArgoCD	[11]	2
2	Наблюдаемость: Prometheus, Grafana, SLI/SLO	[12]	2
3	Архитектурные паттерны для CI/CD (GoF, чистая архитектура)	[13]	2
4	Функциональное программирование для пайплайнов	[14]	2
5	Рефакторинг и SOLID в контексте CI/CD	[15, гл. 9]	2

2.2. Рекомендуемая литература

1. SonarQube Documentation – https://docs.sonarqube.org/
2. Nexus Repository Documentation – https://help.sonatype.com/
3. HashiCorp Vault Documentation – https://www.vaultproject.io/docs
4. Python asyncio Documentation – https://docs.python.org/3/library/asyncio.html
5. Docker Documentation – https://docs.docker.com/
6. Kubernetes Documentation – https://kubernetes.io/ru/docs/
7. Argo Rollouts Documentation – https://argoproj.github.io/rollouts/
8. Argo CD Documentation – https://argo-cd.readthedocs.io/
9. Prometheus Documentation – https://prometheus.io/docs/

2.3. Рекомендации по работе с литературой

- При изучении теоретического материала рекомендуется вести конспект в виде структурированных заметок.
- Для каждой темы выделить ключевые термины и составить глоссарий.
- Выполнять практические примеры из литературы в тестовой среде.
- Использовать официальную документацию как основной источник для выполнения лабораторных работ.

3. МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ ЛАБОРАТОРНЫХ РАБОТ

3.1. Структура лабораторного практикума

Лабораторный практикум состоит из 16 работ, объединённых в 4 тематических блока:

Блок	№ ЛР	Тема
Блок 1. Качество, тестирование и безопасность	1-3	Статический анализ, unit-тесты, SAST, рефакторинг, SOLID
Блок 2. Архитектура и паттерны	4-6	Паттерны GoF, чистая архитектура, функциональное программирование
Блок 3. Производительность и конфигурации	7-9	Асинхронность/параллелизм, конфигурации и секреты, Docker-образы
Блок 4. Продвинутый CD и наблюдаемость	10-16	Полный пайплайн, K8s деплой, GitOps, стратегии

Блок	№ ЛР	Тема
		развертывания, мониторинг, тестирование, финальный проект

3.2. Порядок выполнения лабораторных работ

1. Ознакомиться с теоретическим обоснованием работы.
2. Выбрать вариант задания в соответствии с номером, выданным преподавателем.
3. Разработать решение (скрипты, конфигурации, отчёты).
4. Выполнить проверку в тестовой среде.
5. Оформить отчёт по установленной форме.
6. Защитить работу перед преподавателем.

3.3. Рекомендации по подготовке к лабораторным работам

- Изучить соответствующий раздел теоретического материала перед выполнением работы.
- Подготовить тестовую среду (локальный Kubernetes, Python окружение, установленные инструменты: Docker, SonarQube, SourceCraft/GitLab CI).
- Ознакомиться с примерами выполнения из методических указаний к каждой лабораторной работе.
- При возникновении вопросов — консультироваться с преподавателем.

3.4. Рекомендации по выполнению сквозного проекта (ЛР №16)

Сквозной проект выполняется на протяжении всего семестра параллельно с лабораторными работами. Каждая лабораторная работа добавляет новый компонент в проект.

Требования к проекту:

- Выбрать систему для анализа (микросервис на FastAPI/Flask или встроенная система).

- Настроить CI/CD пайплайн со всеми этапами (линтинг → тесты → SAST → сборка → образ → деплой).
- Определить Quality Gate с порогами покрытия и качества кода.
- Настроить управление артефактами (версионирование, бинарный репозиторий).
- Реализовать управление конфигурациями и секретами.
- Выполнить деплой в Kubernetes с одной из стратегий (Blue-Green/Canary).
- Внедрить GitOps (ArgoCD) для синхронизации.
- Настроить наблюдаемость (Prometheus + Grafana).
- Представить итоговый отчёт.

4. ВОПРОСЫ ДЛЯ СОБЕСЕДОВАНИЯ ПО САМОСТОЯТЕЛЬНОМУ ИЗУЧЕНИЮ ЛИТЕРАТУРЫ

1. Что такое CI/CD и чем отличается непрерывная интеграция от непрерывной поставки?
2. Какие метрики качества кода отслеживает SonarQube?
3. Что такое Quality Gate и зачем он нужен в пайплайне?
4. Как измеряется покрытие кода тестами? Какие инструменты используются?
5. Чем SAST отличается от DAST?
6. Какие типы уязвимостей выявляет Bandit?
7. Перечислите пять принципов SOLID. В чём суть каждого?
8. Что такое «чистая архитектура» (Hexagonal/Onion)?
9. Какие паттерны GoF наиболее полезны для построения CI/CD пайплайнов?
10. В чём разница между чистыми и нечистыми функциями в функциональном программировании?
11. Что такое asyncio в Python и для каких задач оно применяется?
12. Чем асинхронность отличается от многопоточности?

13. Какие стратегии параллелизации задач поддерживаются в CI/CD системах (matrix, parallel)?

14. Как организовать безопасное хранение секретов в CI/CD?

15. Что такое Config as Code и зачем он нужен?

16. Что такое multi-stage Docker build и как он уменьшает размер образа?

17. Какие существуют стратегии развертывания в Kubernetes?

18. Чем Blue-Green отличается от Canary?

19. Что такое GitOps и какую роль в нём играет ArgoCD?

20. Какие три столпа observability вы знаете?

21. Как настроить проверку SLI/SLO в CI/CD пайплайне?

22. Что такое Error Budget и как он связан со стратегиями развертывания?

23. Какие архитектурные паттерны повышают отказоустойчивость (Circuit Breaker, Retry, Bulkhead)?

24. Как автоматизировать откат при сбое деплоя?

25. Какие инструменты CI/CD доступны в России (SourceCraft, GitFlic)?

5. КРИТЕРИИ ОЦЕНИВАНИЯ КОМПЕТЕНЦИЙ

5.1. Критерии оценивания лабораторных работ

Оценка	**Критерии**
5 (отлично)	Работа выполнена в полном объеме, пайплайн работает без ошибок, студент демонстрирует понимание всех этапов, отвечает на дополнительные вопросы, отчёт оформлен по требованиям
4 (хорошо)	Работа выполнена полностью, но есть

Оценка	Критерии
	небольшие замечания (неоптимальная конфигурация, неполный отчёт). Студент отвечает на большинство вопросов
3 (удовлетворительно)	Работа выполнена с помощью преподавателя, есть ошибки, которые студент исправляет с наводящими вопросами. Отчёт оформлен минимально
2 (неудовлетворительно)	Работа не выполнена или выполнена неверно, студент не понимает принципов, не может объяснить свои действия

5.2. Критерии оценивания сквозного проекта (ЛР №16)

Критерий	Вес, %	5 (отлично)	4 (хорошо)	3 (удовлетворительно)	2 (неудовлетворительно)
Полнота реализации этапов пайплайна	0	Реализовано 14+ этапов из 16	Реализовано 11-13 этапов	Реализовано 8-10 этапов	Реализовано менее 8 этапов
Качество конфигурации CI/CD	5	Конфигурация оптимальна, соблюдены best practices, код читаемый	Конфигурация работает, есть небольшие замечания	Конфигурация работает с ошибками, требует доработки	Конфигурация отсутствует или неработоспособна

Критерий	Вес, %	5 (отлично)	4 (хорошо)	3 (удовлетворительно)	2 (неудовлетворительно)
Тестирование и покрытие кода	0	Покрытие >80%, все тесты проходят, есть отчёт	Покрытие 70-80%, тесты в основном проходят	Покрытие 60-70%, есть падающие тесты	Покрытие <60%, тесты отсутствуют
Статический анализ и Quality Gate	0	SonarQube настроен, Quality Gate пройден, нет критических ошибок	Quality Gate пройден с замечаниями	Quality Gate не пройден, но анализ выполнен	Анализ не выполнен
Управление артефактами	0	Артефакты версионируются (SemVer), публикуются в репозиторий	Публикация есть, версионирование неполное	Артефакты публикуются без версионирования	Артефакты не публикуются
Управление конфигурациями и секретами	0	Config as Code, секреты безопасно хранятся, интеграция с Vault	Конфигурации есть, секреты в переменных CI/CD	Секреты частично раскрыты или нет интеграции	Секреты в коде

Критерий	Вес, %	5 (отлично)	4 (хорошо)	3 (удовлетворительно)	2 (неудовлетворительно)
Деплой в Kubernetes и стратегия развертывания	0	Деплой в K8s настроен, стратегия (Blue-Green/Canary) реализована	Деплой есть, стратегия частично реализована	Деплой есть, стратегия не реализована	Деплой отсутствует
GitOps (ArgoCD)		ArgoCD настроен, синхронизация работает	ArgoCD настроен, но синхронизация с задержками	ArgoCD установлен, но не синхронизирует	ArgoCD не используется
Наблюдаемость (Prometheus+ Grafana)		Дашборд создан, SLI/SLO проверяются	Дашборд есть, проверки частичные	Метрики собираются, дашборда нет	Мониторинг отсутствует
Качество оформления отчёта и презентации		Отчёт полный, структурированный, скриншоты есть, презентация готова	Отчёт с небольшими недочётами	Отчёт минимальный, презентация отсутствует	Отчёт отсутствует

ШКАЛА ПЕРЕВОДА БАЛЛОВ В ОЦЕНКУ

Итоговая оценка вычисляется как средневзвешенная по всем критериям с учётом весов.

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

ПРИМЕР РАСЧЁТА ИТОГОВОЙ ОЦЕНКИ

Критерий	Вес	Оценка студента	Взвешенный балл
Полнота этапов	20%	5	1.0
Качество конфигурации	15%	4	0.6
Тестирование и покрытие	10%	5	0.5
Статический анализ	10%	4	0.4
Управление артефактами	10%	5	0.5
Управление конфигурациями	10%	4	0.4
Деплой в K8s	10%	5	0.5

Критерий	Вес	Оценка студента	Взвешенный балл
GitOps (ArgoCD)	5%	4	0.2
Наблюдаемость	5%	4	0.2
Качество отчёта	5%	5	0.25
Итого	**100%**	**4.55**	**4.55**

Итоговая оценка: 4.55 → **5 (отлично)**