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

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ

по дисциплине «Автоматизация тестирования и качество ПО»

Направление подготовки 09.03.04 – Программная инженерия.

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

Квалификация выпускника - магистратура.

Очная форма обучения.

Ставрополь

2026

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Лабораторный практикум по дисциплине «Автоматизация тестирования и качество ПО» предназначен для формирования у обучающихся системных знаний и практических навыков в области стратегий, методов и инструментов автоматизированного тестирования, а также их эффективной интеграции в сквозные конвейеры непрерывной интеграции, поставки и развёртывания (CI/CD).

Практикум реализует компетентностный подход: студент не просто усваивает теоретические положения, но и приобретает навыки их применения для решения реальных профессиональных задач в области обеспечения качества сложных программных и программно-аппаратных (инженерных) систем.

Структура и логика практикума выстроены по принципу «от простого к сложному» и охватывают полный цикл автоматизации тестирования:

1. Модульное тестирование (ЛР №1) - фундамент: написание unit-тестов, измерение покрытия кода, настройка порогов качества.
2. Интеграционное тестирование API (ЛР №2) - взаимодействие компонентов: pytest + requests, коллекции Postman, запуск через Newman.
3. UI-автоматизация (ЛР №3) - пользовательский интерфейс: паттерн Page Object Model, Playwright, сквозные E2E-сценарии.
4. Нагрузочное тестирование (ЛР №4) - производительность: k6, smoke/spike/stress тесты, анализ метрик и поиск точки насыщения.
5. Статический анализ безопасности (ЛР №5) - SAST: Semgrep, Bandit, выявление уязвимостей в исходном коде на ранних этапах.
6. Качественные ворота в CI/CD (ЛР №6) - оркестрация: объединение всех видов тестирования в единый пайплайн с автоматическими блокировками при недостижении порогов качества.
7. Тестирование асинхронных систем (ЛР №7) - очереди сообщений: RabbitMQ, Producer/Consumer, retry-логика, Dead Letter Queue, Testcontainers.

8. Динамический анализ безопасности (ЛР №8) - DAST: OWASP ZAP, сканирование работающего приложения, сравнительный анализ SAST и DAST.

Тематический охват лабораторных работ по видам тестирования:

Вид тестирования	**Лабораторные работы**
Функциональное (модульное, интеграционное, E2E)	№1, №2, №3
Нефункциональное (нагрузочное)	№4
Безопасность (SAST, DAST)	№5, №8
Асинхронные системы	№7
CI/CD и Quality Gates	№6

Применяемый инструментарий - современные опенсорсные средства, доступные и разрешённые к использованию на территории РФ: pytest, Playwright, k6, Semgrep, OWASP ZAP, Testcontainers, GitLab CI / GitHub Actions, Docker.

Уровни сложности. Каждая лабораторная работа содержит вариативные задания трёх уровней:

Базовый - обязательный минимум, формирующий core-компетенции.

Средний - углубление знаний, параметризация, расширенные сценарии.

Повышенный - исследовательские и проектные задачи, интеграция в CI/CD, анализ метрик, сравнение инструментов.

Формируемые компетенции. В результате выполнения лабораторного практикума студент овладевает навыками, соответствующими индикаторам компетенции ПК-1: проектирование архитектуры CI/CD-конвейеров (ИД-1), реализация автоматизированных процессов тестирования (ИД-2), интеграция

инструментов безопасности (ИД-3), оценка эффективности и оптимизация пайплайнов (ИД-7).

Межпредметные связи. Лабораторный практикум опирается на знания, полученные при изучении дисциплин «Проектирование и оптимизация конвейеров CI/CD» и «Системная интеграция и управление конфигурациями», а также на практическое владение языками программирования (Python/JavaScript/Java) и системами контроля версий (Git).

Отчётность. По каждой лабораторной работе оформляется электронный отчёт, содержащий цель и задачи, теоретическое обоснование, листинги кода и конфигураций, результаты выполнения (скриншоты, логи, метрики), анализ результатов, ответы на контрольные вопросы и вывод.

Критерии оценивания детализированы для каждого уровня сложности в каждой лабораторной работе и включают проверку корректности выполнения, достижения целевых метрик, интеграции в CI/CD (для среднего и повышенного уровней), а также качества оформления отчёта и глубины ответов на контрольные вопросы.

Последовательное выполнение всех восьми лабораторных работ формирует у студентов готовность к самостоятельной профессиональной деятельности в качестве инженера по автоматизации тестирования (QA Automation Engineer), DevOps-инженера или специалиста по качеству программного обеспечения (QA Engineer) в высокотехнологичных компаниях.

Лабораторная работа №1. Модульное тестирование и анализ покрытия кода

1. Цель работы: Сформировать практические навыки разработки модульных тестов с использованием фреймворка pytest, измерения покрытия кода инструментом coverage.py и настройки минимально допустимого порога покрытия для обеспечения качества кодовой базы.

2. Задачи

1. Разработать предметный модуль (калькулятор, валидатор или иной компонент) с заданной функциональностью.
2. Написать комплект модульных тестов, покрывающих позитивные, негативные и граничные сценарии.
3. Настроить измерение покрытия кода с помощью pytest-cov / coverage.py.
4. Сгенерировать отчёт о покрытии в текстовом и HTML-форматах, проанализировать непокрытые участки.
5. Настроить порог покрытия (например, 80%) и реализовать принудительное падение тестового запуска при недостижении порога.

3. Теоретическое обоснование

Модульное тестирование (Unit Testing) - это метод тестирования, при котором отдельные модули (функции, методы, классы) исходного кода проверяются на корректность в изоляции от остальной системы. Цели модульного тестирования:

- Раннее выявление дефектов (сдвиг влево — Shift-Left Testing).
- Документирование поведения кода через тесты.
- Обеспечение безопасности рефакторинга.
- Снижение стоимости исправления ошибок.

Принципы модульного тестирования (FIRST):

Принцип	Расшифровка

Принцип	Расшифровка
Fast	Тесты должны выполняться быстро (миллисекунды)
Isolated	Тесты не должны зависеть друг от друга и от внешних ресурсов
Repeatable	Результат теста должен быть детерминирован
Self-validating	Тест должен однозначно сообщать об успехе/провале
Timely	Тесты пишутся до или одновременно с кодом (TDD)

Структура теста (паттерн AAA):

- Arrange - подготовка тестовых данных и окружения.
- Act - выполнение тестируемого действия.
- Assert - проверка результата.

Покрытие кода (Code Coverage) - метрика, показывающая долю исходного кода, выполненную в процессе тестирования. Основные виды:

- Line Coverage - доля выполненных строк кода.
- Branch Coverage - доля выполненных ветвей условных операторов (if/else).
- Function/Method Coverage - доля вызванных функций.

Пороговое значение покрытия (обычно 80–90%) используется как один из критериев Quality Gate - автоматической проверки качества, блокирующей слияние кода при недостижении порога.

4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

Разработать модуль согласно варианту. Написать не менее 10 модульных тестов. Настроить покрытие и добиться уровня не ниже 80%.

№	Тема модуля	Описание функциональности
1	Валидатор email	Проверка формата, наличия @, допустимых символов, длины
2	Конвертер температур	Цельсий ↔ Фаренгейт ↔ Кельвин, проверка абсолютного нуля
3	Генератор паролей	Генерация с настройкой длины, состава (буквы/цифры/спецсимволы)
4	Текстовый анализатор	Подсчёт слов, символов, предложений, частоты встречаемости символов
5	Калькулятор скидок	Расчёт цены со скидкой (фикс./процент), проверка границ скидки 0–100%
6	Валидатор телефонных номеров	Проверка формата +7(XXX)XXX-XX-XX, учёт длины и допустимых символов
7	Конвертер дат	Перевод между форматами ДД.ММ.ГГГГ, MM/DD/YYYY, YYYY-MM-DD
8	Анализатор треугольника	По трём сторонам: тип (равносторонний/равнобедренный/разносторонний), валидность
9	Ипотечный калькулятор	Ежемесячный платёж, переплата, график (аннуитетный)
10	Валидатор ИНН	Проверка контрольной суммы для 10- и 12-значного ИНН

№	Тема модуля	Описание функциональности
11	Конвертер римских чисел	Арабские → римские и обратно (1–3999)
12	Калькулятор строк	Сложение строк с разделителем, реверс, поиск подстроки
13	Генератор slug	Из строки формировать URL-slug (транслитерация, замена пробелов на дефисы)
14	Валидатор паролей	Проверка сложности: длина ≥ 8, наличие заглавных, строчных, цифр, спецсимволов
15	Калькулятор НДС	Начисление и выделение НДС (20%, 10%, 0%), проверка отрицательных сумм

4.2. Средний уровень (15 вариантов)

Дополнительно к базовому уровню: использовать параметризацию тестов (pytest.mark.parametrize), фикстуры (pytest.fixture) и маркеры (pytest.mark.skip, xfail). Достичь покрытия не ниже 90%.

№	Тема модуля	Дополнительные требования
	Библиотека для работы с матрицами	Сложение, умножение, транспонирование; фикстуры для создания матриц
	Парсер CSV-файлов	Чтение, валидация колонок, агрегация; фикстура с временным файлом (tmp_path)
	Планировщик задач	Добавление задач с приоритетом, дедлайном; проверка сортировки; skip для

№	Тема модуля	Дополнительные требования
		будущих дат
	Кэш с TTL (Time-To-Live)	Добавление, получение, удаление, инвалидация по времени; фикстура с time_mock
	Валидатор JSON-схемы	Проверка структуры JSON по заданной схеме; параметризация схем
	Конвертер валют	Конвертация с кэшированием курсов; фикстура с mock-курсами; xfail для нестабильных API
	Система логгирования	Форматирование, уровни, фильтрация; параметризация уровней
	Калькулятор IP-адресов	Валидация IPv4/IPv6, расчёт подсети, broadcast; параметризация корректных/некорректных адресов
	Анализатор текста на палиндромы	Поиск всех палиндромов в тексте, игнорирование регистра/пробелов; фикстура с корпусом текстов
0	Сериализатор объектов	Объект → dict/JSON/XML с вложенными структурами; фикстуры для тестовых объектов
1	Очередь с приоритетами	Enqueue/dequeue с учётом приоритета; параметризация входных

№	Тема модуля	Дополнительные требования
		последовательностей
2	Хеш-таблица (словарь)	Реализация с разрешением коллизий; xfail для известных коллизий
3	Конвертер единиц измерения	Длина, масса, время с цепочками преобразований; параметризация маршрутов
4	Шаблонизатор строк	Замена {{variable}} на значения из контекста; фикстура с шаблонами
5	Калькулятор интервалов дат	Разница в днях/месяцах/годах, исключение выходных; параметризация праздничных дней

4.3. Повышенный уровень (15 вариантов)

Дополнительно: настроить pre-commit hook с порогом покрытия, интеграцию с GitLab CI/GitHub Actions, генерацию HTML-отчёта о покрытии, реализовать mutation testing (mutmut) и сравнить результаты. Достичь покрытия 95%+.

	Тема модуля	Расширенные требования
	Event Emitter (паттерн «Наблюдатель»)	Подписка, отписка, emit событий; тестирование порядка вызова; mutation testing
	Parser математических выражений	Парсинг и вычисление "(2+3)*4"; поддержка скобок; CI-интеграция

	Тема модуля	Расширенные требования
	Finite State Machine (конечный автомат)	Определение состояний и переходов; проверка недопустимых переходов; HTML-отчёт
	Пул соединений (Connection Pool)	Выдача, возврат, таймаут, max-size; тесты на конкурентность
	Система плагинов (Plugin System)	Динамическая загрузка модулей; изоляция тестов; pre-commit hook
	Парсер Markdown → HTML	Обработка заголовков, списков, ссылок, жирного/курсива; CI-интеграция
	Rate Limiter (ограничитель частоты)	Token bucket, sliding window; тесты с моками времени
	Система конфигурации (Config Manager)	Загрузка из YAML/JSON/ENV, валидация, значения по умолчанию
	Калькулятор булевых выражений	Построение AST, вычисление с переменными; mutation testing на операторах
0	Компрессор строк (Run-Length Encoding)	Кодирование/декодирование, обработка спецсимволов; CI + HTML-отчёт
1	In-Memory база данных	CRUD, фильтрация, индексация, транзакции (commit/rollback)

	Тема модуля	Расширенные требования
2	Шедулер периодических задач	Cron-подобный парсер, расчёт следующего запуска; моки времени
3	JSONPath-подобный движок	Запросы к JSON: $.store.book[0].title, фильтры, wildcards
4	Система версионирования (SemVer)	Парсинг, сравнение, инкремент major/minor/patch; CI-интеграция
5	Шифратор (AES/Цезарь)	Шифрование/дешифрование, валидация ключей; mutation testing + pre-commit

5. Контрольные вопросы

1. Что такое модульное тестирование и каковы его основные цели?
2. Опишите паттерн AAA (Arrange-Act-Assert). Приведите пример.
3. Какие виды покрытия кода существуют? В чём отличие line coverage от branch coverage?
4. Почему 100% покрытие кода не гарантирует отсутствие ошибок?
5. Какие категории тестовых сценариев необходимо учитывать при разработке модульных тестов (позитивные, негативные, граничные)?
6. Для чего используется декоратор @pytest.mark.parametrize? Приведите пример.
7. Что такое фикстура в pytest? Какие области видимости (scope) фикстур существуют?
8. Как настроить минимальный порог покрытия с помощью --cov-fail-under?
9. Каково назначение mutation testing и чем он дополняет метрику покрытия?

10. Как интегрировать проверку покрытия в CI/CD-пайплайн для автоматической блокировки недостаточно покрытого кода?

6. Требования к отчёту

1. Титульный лист: ФИО, группа, вариант, номер и тема работы.
2. Цель и задачи работы: формулируются индивидуально на основе задания.
3. Теоретическое обоснование: краткое описание модульного тестирования, покрытия кода, паттернов (1–2 стр.).
4. Практическая часть:
   - Листинг разработанного модуля с пояснениями.
   - Листинг модульных тестов с комментариями к ключевым тест-кейсам.
   - Содержимое файлов конфигурации (pyproject.toml, .pre-commit-config.yaml, .gitlab-ci.yml при наличии).
5. Результаты выполнения:
   - Скриншот успешного запуска тестов с отчётом о покрытии.
   - Скриншот падения тестов при недостижении порога покрытия (с непокрытым кодом).
   - Скриншот HTML-отчёта о покрытии с выделением непокрытых строк.
6. Анализ результатов: выявленные проблемы, непокрытые участки, принятые меры, достигнутые метрики.
7. Ответы на контрольные вопросы: развёрнутые ответы на 3–5 вопросов по выбору преподавателя.
8. Вывод: освоенные навыки, достигнута ли цель, практическая ценность.

7. Критерии оценивания

7.1. Базовый уровень — оценка «удовлетворительно»

Критерий	ес
Модуль реализован без синтаксических ошибок, соответствует варианту	
Написано не менее 10 модульных тестов, охватывающих позитивные и негативные сценарии	
Все тесты проходят успешно (PASSED)	
Настроен запуск coverage.py / pytest-cov	
Сгенерирован и приложен к отчёту текстовый отчёт о покрытии	
Отчёт оформлен в соответствии с требованиями	

Снижение оценки: отсутствие отчёта о покрытии, менее 10 тестов, наличие падающих тестов без обоснования, несоответствие темы варианту.

7.2. Средний уровень - оценка «хорошо»

Все критерии базового уровня плюс:

Критерий	ес	К
Использована параметризация тестов (не менее 3 параметризованных тест-кейсов)		+
Применены фикстуры pytest (не менее 2 фикстур с разным scope)		+
Использованы маркеры (skip, xfail) с обоснованием		+

Критерий	Вес
Достигнуто покрытие не ниже 90% (подтверждено отчётом)	+
HTML-отчёт о покрытии приложен к работе	+
На все контрольные вопросы даны полные ответы	+

Снижение оценки: отсутствие параметризации, покрытие ниже 90%, неполные ответы на контрольные вопросы.

7.3. Повышенный уровень - оценка «отлично»

Все критерии базового и среднего уровней плюс:

Критерий	Вес
Настроен pre-commit hook, блокирующий коммит при покрытии ниже порога	
Настроен CI/CD-пайплайн (GitLab CI / GitHub Actions) с автоматическим запуском тестов и проверкой покрытия	
Продемонстрировано падение пайплайна при недостижении порога (приложен скриншот CI-лога)	
Выполнен mutation testing (mutmut / mutpy), результаты проанализированы и приложены к отчёту	
Достигнуто покрытие не ниже 95% (подтверждено отчётом)	

Критерий	ес
Проведён анализ качества тестов: проверка на устойчивость (flaky-тесты), информативность сообщений об ошибках	

Снижение оценки: отсутствие CI/CD-интеграции или pre-commit, пропуск mutation testing, покрытие ниже 95%, формальные ответы на вопросы без понимания.

Лабораторная работа №2. Интеграционное тестирование REST API

1. Цель работы: Сформировать практические навыки автоматизированного интеграционного тестирования REST API с использованием связки pytest + requests, а также инструментов Postman и Newman для создания, выполнения и автоматизации коллекций запросов в CI/CD-окружении.

2. Задачи:

1. Разработать и развернуть простой REST API-сервис на основе Flask или FastAPI с набором эндпоинтов (CRUD и бизнес-логика).
2. Написать комплект интеграционных тестов на pytest + requests, покрывающих все эндпоинты, включая позитивные, негативные и граничные сценарии.
3. Создать коллекцию запросов в Postman с переменными окружения, тестовыми скриптами (pre-request, post-response) и параметризацией.
4. Экспортировать коллекцию Postman и выполнить её через CLI-утилиту Newman с формированием отчёта.
5. Сравнить два подхода (pytest и Postman/Newman) по expressiveness, поддерживаемости и интеграции в CI/CD.

3. Теоретическое обоснование

Интеграционное тестирование (Integration Testing) - это уровень тестирования, на котором проверяется корректное взаимодействие между несколькими модулями, компонентами или внешними системами. В отличие от модульного тестирования, интеграционные тесты работают с реальными (или близкими к реальным) зависимостями: базой данных, сетью, файловой системой, HTTP-сервером.

REST API (Representational State Transfer) - архитектурный стиль взаимодействия распределённых систем, основанный на следующих принципах:

Принцип	Описание
Клиент-серверная архитектура	Разделение ответственности между клиентом и сервером
Отсутствие состояния (Stateless)	Каждый запрос содержит всю необходимую информацию

Принцип	Описание
Кэшируемость	Ответы могут быть помечены как кэшируемые
Единообразие интерфейса	Использование стандартных HTTP-методов и кодов состояния
Многоуровневая система	Возможность проксирования и балансировки

HTTP-методы в тестировании REST API:

Метод	Назначение	Идемпотентность	Пример проверок
GET	Получение ресурса	Да	Код 200, тело ответа, отсутствие изменений
POST	Создание ресурса	Нет	Код 201, Location header, сохранение в БД
PUT	Полное обновление	Да	Код 200/204, корректность обновления
PATCH	Частичное обновление	Нет	Код 200, изменены только указанные поля
DELETE	Удаление ресурса	Да	Код 204, повторный запрос → 404

Сравнение подходов к автоматизации тестирования API:

Критерий	**pytest + requests**	**Postman + Newman**
Порог входа	Требует знания Python	Низкий, визуальный интерфейс
Параметризация	Полная (pytest.mark.parametrize)	Через CSV/JSON-файлы данных
CI/CD интеграция	Нативная (pytest - CLI)	Через Newman CLI
Отладка	Через pdb/IDE	Встроенная консоль Postman
Поддерживаемость	Код, Git-friendly	JSON-коллекции, сложнее code review
Программная логика	Полная (циклы, условия)	Ограничена JS-скриптами

Newman - CLI-утилита для запуска коллекций Postman без графического интерфейса. Ключевые возможности:

- Запуск коллекций из экспортированного JSON.
- Переопределение переменных окружения через --env-var или файлы окружения.
- Генерация отчётов в форматах CLI, JSON, JUnit XML, HTML (с доп. репортерами).

- Интеграция в CI/CD-пайплайны (GitLab CI, GitHub Actions, Jenkins).
- Поддержка итераций данных (-d data.json).

4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

Разработать REST API-сервис согласно варианту (2–4 эндпоинта), написать не менее 10 интеграционных тестов на pytest + requests, создать коллекцию Postman (не менее 5 запросов с тестовыми скриптами).

№	Тема API	Эндпоинты	Особенности
	Библиотека книг	GET /books, GET /books/{id}, POT /books, DELETE /books/{id}	Поиск по автору и жанру
	Список контактов	GET /contacts, POST /contacts, PUT /contacts/{id}, DELETE /contacts/{id}	Валидация email и телефона
	Счётчик посещений	POST /pageviews, GET /stats (по дням/url)	Агрегация по датам
	Каталог товаров	GET /products, POST /products, GET /products/{id}	Фильтрация по цене и категории
	Заметки (Notes)	CRUD заметок + PATCH для archive/pin	Поиск по тегам
	Регистрация событий	POST /events, GET /events?type=&from=&to=	Фильтрация по типу и диапазону дат
	Голосовалка	POST /polls, POST	Защита от

№	Тема API	Эндпоинты	Особенности
		/polls/{id}/vote, GET /polls/{id}/results	повторного голоса
	Корзина покупок	POST /cart/items, GET /cart, DELETE /cart/items/{id}	Расчёт итоговой суммы
	Сокращатель ссылок	POST /links, GET /{shortcode}, GET /links/{id}/stats	Редирект и подсчёт переходов
0	Комментарии	CRUD комментариев к постам	Модерация (скрытие по флагу)
1	Список желаний	CRUD wishlist items + PATCH приоритета	Сортировка по приоритету
2	Планировщик рецептов	CRUD рецептов + GET /recipes?ingredients=	Фильтрация по ингредиентам
3	Система тегов	CRUD тегов + POST /items/{id}/tags (привязка)	Множественная привязка
4	Отзывы (Reviews)	CRUD отзывов с рейтингом 1–5	Расчёт среднего рейтинга
5	Бронирование ресурсов	POST /bookings, GET /bookings?resource=&date=	Проверка конфликтов времени

4.2. Средний уровень (15 вариантов)

Дополнительно: параметризация тестов, data-driven тестирование (CSV/JSON), папки и переменные коллекции в Postman, запуск Newman с экспортом HTML-отчёта (htmlextra), тестирование аутентификации (JWT/Basic Auth).

	Тема API	Дополнительные требования
	Пользователи и роли	JWT-аутентификация, эндпоинты: регистрация, логин, профиль; параметризация ролей
	Блог с категориями	Вложенные ресурсы: /posts/{id}/comments; коллекция Postman с папками; CSV-датасет
	Трекер задач	CRUD задач + статусы (todo/in_progress/done); параметризация по статусам
	Система подписок	Подписка/отписка, список подписчиков; Pre-request Script для авторизации
	Файловое хранилище	Загрузка файла (multipart), метаданные, скачивание; Newman с HTML-отчётом
	Чат (сообщения)	Отправка, получение, пометка прочитанным; параметризация отправителей
	API-key авторизация	Валидация ключа, лимит запросов, headers: X-API-Key; тесты на 401/403
	Гео-сервис	Координаты, поиск ближайших точек; параметризация координат
	Интернет-магазин	Поиск, фильтры, пагинация (limit/offset); data-driven из JSON

	Тема API	Дополнительные требования
0	Расписание (cron-like)	CRUD расписаний, расчёт следующего срабатывания; параметризация cron-выражений
1	OAuth2 Client Credentials	Получение токена, обновление, запрос с токеном; Pre-request Script для автоматического получения
2	Кинотека	CRUD фильмов + рейтинги; агрегация средней оценки; папки коллекции
3	Инвентаризация	Приход/расход/остатки; проверка бизнес-логики (не уйти в минус)
4	Нотификации	Создание, список, пометка прочитанными; параметризация каналов (email/sms/push)
5	АБ-тесты (feature flags)	CRUD экспериментов, распределение вариантов; тесты на процентное распределение

4.3. Повышенный уровень (15 вариантов)

Дополнительно: контрактное тестирование (схемы JSON Schema / OpenAPI), нагрузочный аспект (замер времени ответа, пороговые значения), интеграция Newman в GitLab CI / GitHub Actions с Quality Gate (пайплайн падает при падении тестов или превышении времени ответа), написание кастомного репортера для Newman.

№	**Тема API**	**Расширенные требования**
	Микросервис заказов	Контрактное тестирование через OpenAPI-схему; валидация ответов; CI с Quality Gate

№	Тема API	Расширенные требования
	Платёжный шлюз (mock)	Идемпотентность запросов, проверка сигнатур; кастомный Newman-репортер
	GraphQL-обёртка	GraphQL вместо REST; тесты через POST /graphql; сравнение с REST-подходом
	API Gateway (rate limiting)	Проверка заголовков X-RateLimit-*, тесты на превышение лимита
	Users API с пагинацией	Link-заголовки (RFC 5988), обход всех страниц; тесты производительности (время ответа < 200ms)
	Webhooks-система	Регистрация, отправка, проверка доставки; мок сервера для приёма webhook
	Мультиарендное API	Tenant-ID в заголовке, изоляция данных; параметризация арендаторов
	Content API с версионированием	Заголовок Accept-Version; тесты обратной совместимости
	API с кэшированием (ETag)	Проверка 304 Not Modified; тесты на инвалидацию кэша
0	Long-running операции	POST /jobs → 202 Accepted, GET /jobs/{id}/status; поллинг статуса
1	Потоковая передача (SSE)	Server-Sent Events; тестирование стримов через специальный клиент

№	Тема API	Расширенные требования
2	API с HATEOAS	Проверка _links в ответах; навигация по API через ссылки
3	Поисковый API (Elastic-like)	Полнотекстовый поиск, фильтры, агрегации; параметризация запросов
4	API с идемпотентностью	Idempotency-Key; проверка повторных запросов с одинаковым ключом
5	CI/CD-Orchestrator	Запуск pipeline, проверка статуса; интеграция Newman в собственный пайплайн с Quality Gate

5. Контрольные вопросы

1. Чем интеграционное тестирование отличается от модульного? Приведите примеры проверок, характерных для каждого уровня.
2. Что такое REST API и каковы его основные принципы (Stateless, Cacheable, Uniform Interface)?
3. Какие коды состояния HTTP наиболее важны при тестировании API? Опишите группы 2xx, 4xx, 5xx.
4. В чём разница между методами PUT и PATCH? Когда какой следует использовать?
5. Какие типы проверок можно автоматизировать в Postman с помощью скриптов (Pre-request и Post-response)?
6. Как организовать Data-Driven тестирование в Postman/Newman? Какие форматы данных поддерживаются?
7. Какие преимущества и недостатки у подхода тестирования через Postman/Newman по сравнению с pytest + requests?
8. Как интегрировать Newman в CI/CD-пайплайн и какие отчёты он может генерировать для систем CI?

9. Что такое контрактное тестирование API? Какую роль играет OpenAPI/Swagger-схема?

10. Как проверить идемпотентность API-методов? Для каких HTTP-методов она гарантируется стандартом?

6. Требования к отчёту

1. Титульный лист: ФИО, группа, вариант, номер и тема лабораторной работы.

2. Цель и задачи работы.

3. Теоретическое обоснование: краткое описание REST API, интеграционного тестирования, инструментов (1–2 стр.).

4. Практическая часть:
   - Листинг разработанного REST API-сервиса с комментариями.
   - Листинг интеграционных тестов на pytest + requests (полный код).
   - Скриншот коллекции Postman со структурой папок и запросов.
   - Примеры скриптов Pre-request и Post-response для ключевых запросов.
   - Содержимое файла данных (CSV/JSON) для Data-Driven тестирования (если применимо).
   - Листинг CI/CD-файла (.gitlab-ci.yml / .github/workflows/test.yml) для повышенного уровня.

5. Результаты выполнения:
   - Скриншот запуска тестов pytest с пройденными тестами.
   - Скриншот запуска коллекции в Postman (Runner) с результатами.
   - Скриншот запуска Newman в CLI с итоговой таблицей (executed/failed).
   - Скриншот HTML-отчёта Newman (для среднего и повышенного уровня).
   - Скриншот падения пайплайна в CI/CD при ошибке тестов (для повышенного уровня).

6. Сравнительный анализ: таблица сравнения подходов pytest и Postman/Newman с выводами.

7. Ответы на контрольные вопросы: 3–5 вопросов по выбору преподавателя.

8. Вывод: освоенные навыки, достигнута ли цель, практическая ценность для профессиональной деятельности.

7. Критерии оценивания

7.1. Базовый уровень - оценка «удовлетворительно»

Критерий
REST API-сервис реализован и запускается без ошибок (не менее 2 эндпоинтов)
Написано не менее 10 интеграционных тестов на pytest + requests (позитивные + негативные)
Создана коллекция Postman (не менее 5 запросов)
В коллекции Postman присутствуют тестовые скрипты (проверка статус-кода и тела ответа)
Коллекция экспортирована в JSON, запущена через Newman с CLI-отчётом
Все тесты проходят успешно
Отчёт оформлен в соответствии с требованиями

Снижение оценки: отсутствие Postman-коллекции или тестовых скриптов в ней, менее 10 тестов pytest, отсутствие запуска Newman, отсутствие негативных сценариев.

7.2. Средний уровень - оценка «хорошо»

Все критерии базового уровня плюс:

Критерий
Использована параметризация в pytest (не менее 5 параметризованных тест-кейсов)
Реализовано Data-Driven тестирование (CSV/JSON файл) для Postman/Newman
Коллекция Postman организована в папки, использованы переменные коллекции и окружения
Реализована аутентификация (JWT/Basic) с Pre-request Script для получения токена
Newman запущен с HTML-репортером (htmlextra), отчёт приложен
Сгенерирован отчёт о покрытии эндпоинтов (перечень эндпоинтов и статус покрытия тестами)
На все контрольные вопросы даны развёрнутые ответы

Снижение оценки: отсутствие параметризации или Data-Driven, нет аутентификации, нет HTML-отчёта Newman, неполные ответы на вопросы.

7.3. Повышенный уровень - оценка «отлично»

Все критерии базового и среднего уровней плюс:

Критерий
Реализована валидация ответов по JSON Schema (OpenAPI)
Настроен CI/CD-пайплайн (GitLab CI / GitHub Actions) с запуском pytest и Newman
Настроен Quality Gate: пайплайн падает при ошибке тестов

Критерий
ИЛИ превышении порога времени ответа
Реализован и приложен кастомный Newman-репортер (или расширенный скрипт отчёта)
Проведён сравнительный анализ времени выполнения тестов в pytest vs Newman
Протестированы дополнительные аспекты: rate limiting, идемпотентность, пагинация (в зависимости от варианта)
Достигнуто покрытие эндпоинтов 100% (все эндпоинты, все методы, все коды ответов)

Снижение оценки: отсутствие CI/CD-интеграции, отсутствие Quality Gate, пропуск контрактного тестирования или кастомного репортера, формальный сравнительный анализ.

Лабораторная работа №3. Автоматизация тестирования UI с Page Object Model

1. Цель работы: Сформировать практические навыки автоматизации тестирования пользовательского интерфейса веб-приложений с использованием паттерна Page Object Model (POM) и инструмента Playwright,

обеспечивающего структурный, поддерживаемый и масштабируемый код UI-автотестов.

2. Задачи:

1. Выбрать тестовое веб-приложение (интернет-магазин, блог или иной ресурс с формами и навигацией) и проанализировать его структуру страниц.
2. Спроектировать иерархию Page Object-классов для ключевых страниц приложения (не менее 3 страниц).
3. Реализовать Page Object Model на Python с использованием Playwright, инкапсулируя локаторы и действия страниц.
4. Написать сквозной E2E-сценарий, охватывающий многошаговый пользовательский путь (логин → навигация → действие → проверка результата).
5. Реализовать негативный сценарий (например, попытка входа с неверными данными) и проверку соответствующих сообщений об ошибках.
6. Настроить формирование скриншотов и видеозаписи для упавших тестов.

3. Теоретическое обоснование

Автоматизация UI-тестирования - это процесс программного управления браузером для имитации действий пользователя и проверки корректности отображения и функционирования пользовательского интерфейса. Основные цели:

- Сокращение времени регрессионного тестирования.
- Повышение повторяемости и точности проверок.
- Обеспечение качества критических пользовательских путей (Critical Path).

Паттерн Page Object Model (POM) - архитектурный паттерн проектирования автотестов, при котором каждая страница (или значимый компонент) приложения представляется отдельным классом. Ключевые принципы:

Принцип	Описание
Инкапсуляция	Локаторы элементов и методы взаимодействия скрыты внутри класса страницы
Абстракция	Тестовый сценарий оперирует высокоуровневыми действиями, а не прямыми кликами по локаторам
Повторное использование	Методы Page Object могут использоваться в разных тестах
Устойчивость к изменениям	При изменении вёрстки правки вносятся только в один класс страницы

Playwright vs Selenium: сравнительный анализ:

Критерий	Playwright	Selenium
Поддержка браузеров	Chromium, Firefox, WebKit	Chrome, Firefox, Safari, Edge, IE
Скорость	Выше (автоожидание, оптимизированный протокол)	Ниже (WebDriver-протокол)
Автоожидания	Встроенные (auto-wait)	Требуют явных/неявных ожиданий

Критерий	Playwright	Selenium
Мобильное тестирование	Эмуляция устройств	Appium (отдельный инструмент)
Сетевые перехваты	Встроенный API для mock-запросов	Через прокси (BrowserMob)
Параллельный запуск	Встроенный sharding	Через Selenium Grid / Docker
Установка	pip install playwright && playwright install	pip install selenium + WebDriver

Типичная иерархия Page Object в E2E-тестировании интернет-магазина:

BasePage
├── HeaderComponent (общий для всех страниц)
├── LoginPage
├── CatalogPage (список товаров)
├── ProductPage (детали товара)
├── CartPage
├── CheckoutPage
└── OrderConfirmationPage

Компонентный подход (Component Object) — развитие POM, при котором повторяющиеся UI-компоненты (хедер, футер, модальные окна, строка поиска) выделяются в отдельные классы и переиспользуются на разных страницах:

4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

Разработать Page Object Model для указанного приложения (не менее 3 страниц), написать не менее 8 тестов, включая позитивные, негативные и один E2E-сценарий.

	Тестовое веб-приложение	**Страницы**	**Ключевой E2E-сценарий**
	SauceDemo	Login, Inventory, Cart	Логин → Сортировка → Добавление → Корзина
	The Internet	Login, Add/Remove Elements, Checkboxes	Логин → Работа с элементами → Выход
	Форма регистрации (локальная)	Register, Welcome, Profile	Регистрация → Подтверждение → Просмотр профиля
	DemoQA	Text Box, Check Box, Radio Button	Заполнение формы → Отправка → Проверка результата
	Блог (локальный)	Posts List, Post Detail, Login	Логин → Создание поста → Просмотр в списке
	Калькулятор (веб)	Calculator, History	Вычисление → Проверка результата → История
	TODO-приложение (веб)	Task List, Task Form	Создание → Редактирование → Удаление задачи

	Тестовое веб-приложение	Страницы	Ключевой E2E-сценарий
	Опросник (веб-форма)	Survey, Results	Заполнение → Отправка → Просмотр результатов
	ParaBank	Login, Accounts Overview, Transfer	Логин → Просмотр счетов → Перевод
0	Интернет-магазин (локальный)	Catalog, Product, Cart	Поиск → Детали → Добавление в корзину
1	Админ-панель (локальная)	Login, Dashboard, Users List	Логин → Навигация → Проверка данных
2	Форма обратной связи	Contact Form, Success Page	Заполнение → Валидация → Отправка
3	Библиотека (веб-каталог)	Search, Book Detail, Favorites	Поиск → Просмотр → Добавление в избранное
4	Погодный виджет (веб)	Search, Forecast	Поиск города → Просмотр прогноза
5	Automation Practice	Login, Catalog, Cart	Логин → Фильтр → Добавление → Корзина

4.2. Средний уровень (15 вариантов)

Дополнительно: выделение повторяющихся компонентов в Component Object (Header, Footer, SearchBar), параметризация тестов, использование Fluent-интерфейса, проверка визуальных состояний (disabled-кнопки, анимации загрузки), скриншоты при падении.

	Приложение	Дополнительные требования
	SauceDemo (расширенный)	Component Object: Header, Footer; Fluent-методы; проверка всех сортировок
	Интернет-магазин (локальный)	Поиск с фильтрами, пагинация; параметризация поисковых запросов
	Корпоративный портал	Компонент навигации, breadcrumbs; тесты ролей (admin/manager/user)
	Дашборд с графиками	Компонент фильтра дат; проверка обновления графиков
	Kanban-доска	Компонент карточки, колонки; Drag-and-Drop (перемещение карточек)
	Форма с многошаговым wizard	Компонент шагов; Fluent-переходы; валидация каждого шага
	Каталог с бесконечным скроллом	Виртуализация; проверка подгрузки; скриншоты
	Чат (веб-интерфейс)	Компоненты: список диалогов, окно сообщений; проверка отправки
	Админка с таблицами	Компонент таблицы (сортировка, фильтрация, пагинация)
0	Event Management	Календарь, форма события; параметризация дат
	E-commerce с AJAX-	Компонент мини-корзины; проверка без

	Приложение	Дополнительные требования
1	корзиной	перезагрузки
2	Личный кабинет банка	Компонент списка карт; проверка маскирования номера
3	Система бронирования	Календарь, временные слоты; проверка занятых слотов
4	Маркетплейс	Сравнение товаров; компонент сравнения
5	Новостной портал	Компонент ленты; infinite scroll; проверка lazy-load изображений

4.3. Повышенный уровень (15 вариантов)

Дополнительно: визуальные регрессионные тесты (сравнение скриншотов с эталоном через pixelmatch или expect(locator).to_have_screenshot()), мокирование API-ответов на уровне браузера (page.route()), настройка CI/CD (GitHub Actions / GitLab CI) с параллельным запуском и шардированием, видеоотчёт о прогоне тестов.

	Тема	Расширенные требования
	E-commerce с визуальными тестами	Скриншотное сравнение страниц каталога, товара, корзины
	SPA (React/Vue) приложение	Мокирование API через page.route(); проверка состояний загрузки
	Мобильная версия магазина	Эмуляция устройства (iPhone 12); touch-действия; адаптивная вёрстка

	Тема	Расширенные требования
	Социальная сеть (mock)	Мок API; сложный E2E: регистрация → пост → лайк → комментарий
	Панель мониторинга (WebSocket)	Проверка real-time обновлений; мок WebSocket-соединений
	Финансовый дашборд	Визуальное сравнение графиков; CI с параллельным запуском в 3 браузерах
	Dark Mode приложение	Переключение темы; скриншотное сравнение обеих тем
	Мультиязычный интерфейс	Смена локали; проверка переводов; параметризация языков
	PWA (Progressive Web App)	Offline-режим; Service Worker; проверка установки
0	Приложение с капчей	Обход капчи через тестовый ключ или мок; проверка блокировки при превышении попыток
1	Интернет-банк (полный E2E)	Логин → Счёт → Перевод → История (5+ шагов); CI/CD с Quality Gate
2	A/B тестирование UI	Проверка разных вариантов страниц; скриншотное сравнение
3	Приложение с WebGL/Canvas	Проверка интерактивных элементов canvas

	Тема	Расширенные требования
4	Кросс-браузерное тестирование	Параллельный запуск в Chromium + Firefox + WebKit; сравнение поведения
5	Design System компонентов	Визуальные тесты каждого компонента (кнопки/формы/модалки)

5. Контрольные вопросы

1. Что такое паттерн Page Object Model и какие проблемы он решает в автоматизации UI-тестирования?
2. Какие принципы инкапсуляции и абстракции применяются при проектировании Page Object-классов?
3. Чем отличается подход Component Object от Page Object? В каких случаях следует выделять компоненты?
4. Какие преимущества предоставляет Playwright по сравнению с Selenium? Назовите не менее трёх.
5. Что такое Fluent Page Object и какие преимущества он даёт при написании тестовых сценариев?
6. Какие стратегии ожидания элементов используются в UI-автоматизации? Чем автоожидания Playwright отличаются от явных/неявных ожиданий Selenium?
7. Как организовать Data-Driven тестирование в UI-автотестах? Приведите пример параметризации.
8. Какие существуют подходы к обработке динамических элементов (изменяющиеся ID, асинхронная подгрузка)?
9. Что такое визуальное регрессионное тестирование и как его реализовать в Playwright?
10. Как интегрировать UI-автотесты в CI/CD-пайплайн? Какие метрики качества можно отслеживать?

6. Требования к отчёту

1. Титульный лист: ФИО, группа, вариант, номер и тема работы.
2. Цель и задачи работы.
3. Теоретическое обоснование: описание паттерна POM, сравнение Playwright и Selenium, структура UI-автотестов (1–2 стр.).
4. Практическая часть:
   - Схема иерархии Page Object-классов (UML-диаграмма или блок-схема).
   - Листинг базового класса BasePage.
   - Листинг всех Page Object-классов с локаторами и методами (с комментариями).
   - Листинг E2E-тестов с выделением Arrange/Act/Assert.
   - Листинг файла конфигурации conftest.py, pytest.ini.
   - Для среднего уровня: листинг Component Object (Header, Footer, etc.).
   - Для повышенного уровня: листинг CI/CD-файла, конфигурация шардирования.
5. Результаты выполнения:
   - Скриншот успешного прогона всех тестов в терминале (с зелёными PASSED).
   - Скриншот HTML-отчёта о тестировании (pytest-html).
   - Скриншоты страниц приложения с выполненными действиями (до/после).
   - Для среднего уровня: скриншоты упавших тестов с автоматическими скриншотами.
   - Для повышенного уровня: скриншот визуального сравнения, скриншот CI/CD-пайплайна с параллельными джобами.
6. Анализ результатов: описание проблем (flaky-тесты, таймауты, нестабильные локаторы), принятые меры, анализ стабильности тестов.
7. Ответы на контрольные вопросы: 3–5 вопросов.
8. Вывод.

7. Критерии оценивания

7.1. Базовый уровень — оценка «удовлетворительно»

Критерий
Реализовано не менее 3 Page Object-классов (BasePage + 2 страницы)
Локаторы вынесены в атрибуты классов (инкапсуляция)
Написано не менее 8 тестов (позитивные + негативные + 1 E2E)
E2E-сценарий включает не менее 3 последовательных шагов
Все тесты стабильно проходят (не менее 2 прогонов)
Приложены скриншоты выполнения
Отчёт оформлен по требованиям

Снижение оценки: менее 3 Page Object-классов, локаторы в теле тестов, менее 8 тестов, падающие тесты без обоснования.

7.2. Средний уровень — оценка «хорошо»

Все критерии базового уровня плюс:

Критерий
Выделены Component Object для повторяющихся элементов (Header, Footer, etc.)
Применён Fluent-интерфейс (методы возвращают self для цепочек)
Использована параметризация тестов (минимум 3 параметризованных тест-кейса)
Реализованы проверки визуальных состояний (disabled, loading, error

Критерий
states)
Настроено автоматическое сохранение скриншотов при падении тестов
Сгенерирован HTML-отчёт (pytest-html)
На все контрольные вопросы даны развёрнутые ответы

Снижение оценки: отсутствие Component Object, нет Fluent-методов, нет параметризации, нет скриншотов упавших тестов.

7.3. Повышенный уровень — оценка «отлично»

Все критерии базового и среднего уровней плюс:

Критерий
Реализованы визуальные регрессионные тесты (expect(locator).to_have_screenshot() или pixelmatch)
Настроено мокирование API-запросов через page.route() (не менее 2 замоканных эндпоинтов)
Настроен CI/CD-пайплайн (GitLab CI / GitHub Actions) с параллельным запуском (sharding)
Реализована видеозапись тестового прогона (video retain-on-failure)
Проведён прогон минимум в 2 браузерах (Chromium + Firefox или WebKit)
Проведён анализ стабильности тестов: выявлены и устранены flaky-тесты (повторный прогон 5+ раз)
Отчёт содержит список локаторов с обоснованием выбора и резервными

Критерий

локаторами

Снижение оценки: отсутствие визуальных тестов, отсутствие CI/CD, нет мокирования API, прогон только в одном браузере, наличие flaky-тестов без анализа.

Лабораторная работа №4. Нагрузочное тестирование с k6

1. Цель

Сформировать практические навыки планирования, разработки и выполнения нагрузочного тестирования веб-приложений с использованием инструмента k6, а также анализа полученных метрик производительности для определения узких мест и точки насыщения системы.

2. Задачи

1. Развернуть тестируемое веб-приложение (REST API или веб-сервис) в локальном или контейнеризированном окружении.
2. Изучить архитектуру и возможности инструмента k6, освоить написание тестовых сценариев на JavaScript.
3. Разработать сценарии для трёх типов нагрузочного тестирования: smoke (дымовое), spike (пиковое) и stress (стрессовое).
4. Настроить сбор и экспорт метрик производительности (RPS, latency percentiles, error rate, HTTP-статусы).
5. Провести серию тестов с постепенным увеличением нагрузки для выявления точки насыщения системы.

6. Проанализировать результаты, построить графики зависимости времени ответа и пропускной способности от нагрузки, определить пределы производительности.

3. Теоретическое обоснование

Нагрузочное тестирование (Load Testing) — это вид нефункционального тестирования, направленный на оценку производительности системы при ожидаемой и экстремальной нагрузке. Основные цели:

- Определение максимальной пропускной способности системы.
- Выявление времени отклика при различных уровнях нагрузки.
- Обнаружение узких мест (bottlenecks) в архитектуре.
- Проверка стабильности системы под длительной нагрузкой.
- Валидация требований к производительности (SLA/SLO).

Типы нагрузочного тестирования:

Тип	Описание	Цель	Характер нагрузки
Smoke (дымовое)	Минимальная нагрузка для проверки работоспособности скриптов и окружения	Быстрая валидация готовности к тестированию	1–5 VUs, короткая продолжительность
Load (нагрузочное)	Ожидаемая рабочая нагрузка	Оценка производительности в штатном режиме	Постепенное наращивание до целевого RPS
Stress	Нагрузка	Определени	Линейный

Тип	Описание	Цель	Характер нагрузки
(стрессовое)	выше ожидаемой, вплоть до отказа	е пределов системы и характера деградации	рост до точки отказа
Spike (пиковое)	Резкое кратковременное увеличение нагрузки	Проверка реакции на внезапные всплески	Мгновенный скачок VUs, короткая продолжительность
Soak (выдержка)	Длительная нагрузка на ожидаемом уровне	Выявление утечек памяти, деградации со временем	Стабильная нагрузка часами
Breakpoint	Поиск точки насыщения	Определение предельной пропускной способности	Ступенчатое увеличение до падения метрик

Ключевые метрики производительности:

Метрика	Обозначение	Описание	Целевое значение
RPS / Throughput	Запросов/сек	Пропускная способность системы	Максимально возможное

Метрика	Обозначение	Описание	Целевое значение
Latency (P50)	p(50)	Медианное время ответа (50% запросов быстрее)	< 200ms для веб
Latency (P95)	p(95)	95-й перцентиль (только 5% запросов медленнее)	< 500ms для веб
Latency (P99)	p(99)	99-й перцентиль (длинный хвост)	< 1000ms
Error Rate	%	Доля ошибочных запросов	< 1% (или 0%)
Peak RPS	max req/s	Максимальная достигнутая пропускная способность	Характеристика предела
Concurrent VUs	VUs	Количество одновременных виртуальных пользователей	Индикатор нагрузки

k6 — современный опенсорсный инструмент нагрузочного тестирования, разработанный Grafana Labs. Ключевые особенности:

Особенность	Описание
Язык	JavaScript (ES6+), порог входа низкий

Особенность	Описание
сценариев	
Архитектура	Написан на Go, высокая производительность
Модель исполнения	VUs (Virtual Users) + итерации / открытая модель
Метрики	Встроенные (http_req_duration, http_req_failed) + пользовательские (Trend, Counter, Gauge)
Пороги (Thresholds)	Встроенный механизм Pass/Fail по метрикам
CI/CD интеграция	Нативная поддержка вывода JUnit XML, JSON, CSV
Расширяемость	Модули (xk6) для Kafka, Redis, SQL, WebSocket, browser
Облачный сервис	Grafana Cloud k6 для масштабных тестов

Модели исполнения в k6:

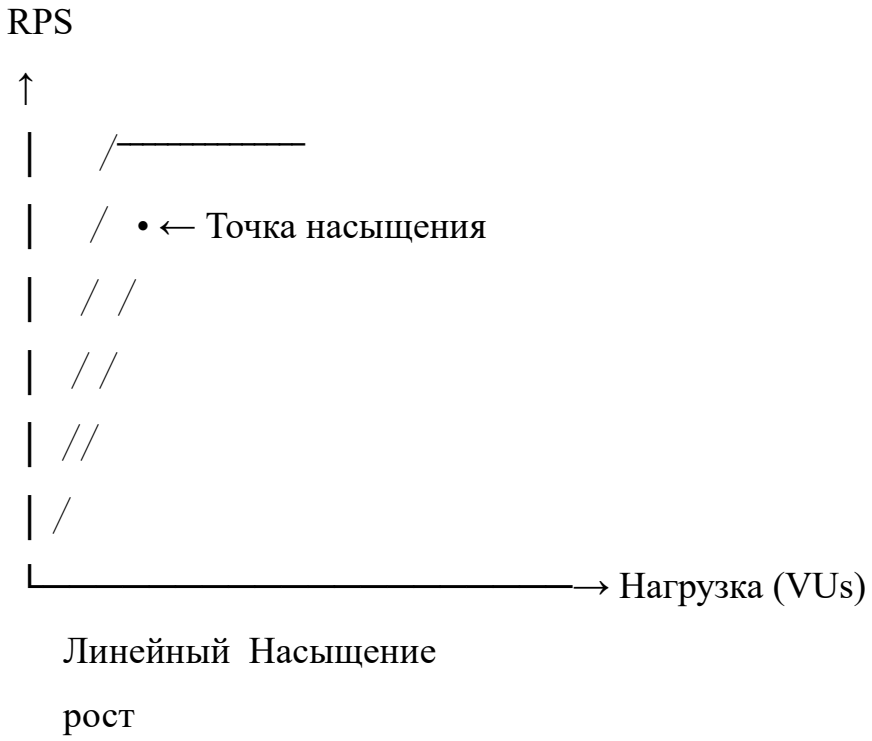
1. Закрытая модель (Closed Model) — фиксированное количество VUs:

- Постоянное число пользователей
- Характерно для синхронных систем

2. Открытая модель (Open Model) — фиксированная скорость прибытия запросов:

- constant-arrival-rate
- ramping-arrival-rate
- Характерно для асинхронных API

Точка насыщения (Saturation Point) — уровень нагрузки, при котором дальнейшее увеличение VUs или RPS не приводит к росту пропускной способности, а время ответа начинает экспоненциально расти. Определяется эмпирически через построение графика зависимости RPS от нагрузки.



4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

Разработать сценарий smoke-теста и нагрузочного теста для указанного API/приложения. Провести тесты, собрать метрики (RPS, P95, P99, error rate), построить минимум 2 графика, сформулировать выводы.

	Тестируемое приложение	Эндпоинты под нагрузкой	Особенности
	Todo API (из ЛР №2)	GET /todos, POST /todos, GET /todos/{id}	Смешанная нагрузка (70% read, 30% write)
	JSONPlaceholder (мок)	GET /posts, GET /posts/{id}, POST /posts	Только внешние запросы, без записи в БД
	Библиотека книг (своё API)	GET /books, GET /books?genre=, POST /books	Поиск с параметрами
	Сокращатель ссылок	POST /links, GET /{shortcode}	Создание + редирект
	Каталог товаров	GET /products, GET /products?category=	Фильтрация
	Система комментариев	GET /comments?post_id=, POST /comments	Вложенные запросы
	Пользовательский сервис	POST /register, POST /login	Аутентификация
	Корзина покупок	GET /cart, POST /cart/items, DELETE /cart/items/{id}	Состояние между запросами

	Тестируемое приложение	Эндпоинты под нагрузкой	Особенности
	Система голосования	POST /polls/{id}/vote, GET /polls/{id}/results	Подсчёт голосов
0	Сервис уведомлений	POST /notifications, GET /notifications	Запись и чтение
1	Гео-сервис	GET /locations?lat=&lon=&radius=	Вычисления расстояний
2	Конвертер валют	GET /convert?from=&to=&amount=	Вычисления на лету
3	Генератор отчётов	POST /reports (долгий запрос), GET /reports/{id}	Асинхронная обработка
4	Поисковый API	GET /search?q=	Полнотекстовый поиск
5	Файловое хранилище	POST /upload (multipart), GET /files/{id}	Загрузка файлов

4.2. Средний уровень (15 вариантов)

Дополнительно: реализовать spike-тест и stress-тест, использовать пользовательские метрики (Trend, Counter, Rate), настроить пороги (thresholds) для автоматического Pass/Fail, реализовать параметризацию данных, написать скрипт для анализа и визуализации результатов.

	Приложение	Дополнительные требования
	E-commerce API	Spike на поиск в Чёрную пятницу;

	Приложение	Дополнительные требования
		thresholds для P95 поиска
	Todo API с БД (PostgreSQL)	Stress-тест с 4 стадиями роста; кастомный Trend для времени транзакций
	Платёжный шлюз (mock)	Spike 50→500 VUs; Counter ошибок; проверка идемпотентности
	Чат-сервер (WebSocket)	Spike подключений; метрики latency сообщений
	Auth-сервер (OAuth2)	Stress на /token endpoint; Rate для ошибок аутентификации
	API Gateway	Spike на роутинг; пороги для времени проксирования
	CDN (статический контент)	Stress на загрузку файлов разного размера (1KB-10MB)
	Recommendation API	Spike при наплыве пользователей; P99 для ML-инференса
	Микросервис заказов	Stress-тест цепочки: создать → оплатить → подтвердить
0	GraphQL API	Нагрузка с разной глубиной запросов; защита от N+1
1	Image Processing API	Stress на ресайз + фильтры; метрики времени обработки

	Приложение	Дополнительные требования
2	Rate-Limited API	Spike для проверки срабатывания лимитов (429 Too Many Requests)
3	Event Sourcing API	Стресс на запись событий; метрики retention
4	ML Model Serving	Нагрузка на инференс; P95/P99 latency модели
5	Webhook Dispatcher	Spike на массовую отправку; метрики очереди

4.3. Повышенный уровень (15 вариантов)

Дополнительно: настроить Grafana + InfluxDB/Prometheus для real-time мониторинга в процессе теста, реализовать автоматический поиск точки насыщения, интегрировать k6 в CI/CD с Quality Gate (пайплайн падает при нарушении порогов), провести сравнительный анализ производительности при разных конфигурациях (1 worker vs 2 workers, с кэшем/без), настроить распределённый запуск k6 (k6 operator или xk6-disruptor).

	Задание	Расширенные требования
	Todo API с мониторингом	Docker Compose: API + InfluxDB + Grafana; дашборд с RPS, P95, ошибками
	Сравнение конфигураций	1 worker vs 4 workers; графики сравнения; CI/CD с Quality Gate
	E-commerce полный цикл	k6 + Grafana Cloud; поиск точки насыщения; CI с блокировкой при P95 > SLA

	Задание	Расширенные требования
	Service Mesh (Istio)	Нагрузка с disruption (отказ подов); xk6-disruptor
	Serverless (Lambda)	Cold start vs warm start; сравнение latency
	Микросервисы с каскадом	Stress-тест каскадных вызовов; определение bottleneck-сервиса
	БД под нагрузкой	Сравнение PostgreSQL vs MongoDB; метрики на уровне БД
	Redis кэширование	Stress-тест с кэшем/без; анализ hit/miss ratio
	Kafka-based система	Нагрузка на продюсер/консьюмер; xk6-kafka
0	Мобильное API (GraphQL)	Сравнение REST vs GraphQL; метрики payload size
1	CI/CD: Canary-анализ	k6 в CI: сравнение метрик текущей и новой версии
2	Auto-scaling проверка	Stress-тест с мониторингом HPA; проверка времени scale-up
3	gRPC сервис	xk6-grpc; сравнение с REST по RPS и latency
	Circuit Breaker	Stress до срабатывания CB; метрики

	Задание	Расширенные требования
4	тест	восстановления
5	Full Observability	Traces (Jaeger) + Metrics (Prometheus) + Logs; корреляция во время теста

5. Контрольные вопросы

1. Какие виды нагрузочного тестирования существуют и чем они различаются по целям и характеру нагрузки?
2. Что такое точка насыщения (saturation point) системы и как её определить эмпирически?
3. Какие ключевые метрики производительности собираются при нагрузочном тестировании? Что означают P50, P95, P99?
4. В чём разница между закрытой (closed) и открытой (open) моделями нагрузки в k6? В каких случаях какую применять?
5. Как работают thresholds (пороги) в k6 и как их использовать для автоматического Pass/Fail теста?
6. Какие типы пользовательских метрик доступны в k6 (Counter, Gauge, Rate, Trend) и для чего они применяются?
7. Как организовать Data-Driven нагрузочное тестирование в k6 (параметризация запросов)?
8. Каковы основные отличия k6 от JMeter? В каких сценариях предпочтительнее каждый из инструментов?
9. Как интегрировать k6 в CI/CD-пайплайн и какие метрики использовать для Quality Gate?
10. Как мониторить состояние системы во время нагрузочного теста? Какие метрики ОС и приложения критически важны?

6. Требования к отчёту

1. Титульный лист: ФИО, группа, вариант, номер и тема работы.
2. Цель и задачи работы.

3. Теоретическое обоснование: виды нагрузочного тестирования, метрики, архитектура k6 (1–2 стр.).

4. Практическая часть:

- Описание тестируемого приложения и его конфигурации (Docker, БД, количество workers).
- Листинг всех сценариев k6 (smoke, load, spike, stress) с комментариями.
- Параметры запуска и сбор метрик.
- Скрипт анализа результатов (Python/JS) с поиском точки насыщения.
- Листинг CI/CD-файла (для повышенного уровня).

5. Результаты выполнения:

- Скриншот вывода k6 в терминале для каждого типа теста.
- Таблица с агрегированными метриками (RPS, P50, P95, P99, Error Rate, Peak VUs).
- Графики: RPS от времени, P95 Latency от времени, VUs от времени.
- График поиска точки насыщения (RPS vs VUs с выделенной точкой).
- Для повышенного уровня: скриншоты дашборда Grafana.

6. Анализ результатов:

- Определение точки насыщения (при каком количестве VUs).
- Характер деградации системы (рост ошибок, рост latency).
- Узкие места (CPU, память, БД, сеть).
- Рекомендации по улучшению производительности.

7. Сравнительная таблица результатов smoke / load / spike / stress тестов.

8. Ответы на контрольные вопросы.

9. Вывод.

7. Критерии оценивания

7.1. Базовый уровень — оценка «удовлетворительно»

Критерий

Тестируемое приложение развёрнуто и доступно

Реализован сценарий smoke-теста (все запросы проходят, проверки корректны)

Реализован сценарий нагрузочного теста с ramping-vus (не менее 4 стадий)

Собраны и зафиксированы метрики: RPS, P95, P99, error rate (для load-теста)

Построены графики RPS и P95 Latency от времени

Сформулированы выводы о производительности системы при ожидаемой нагрузке

Отчёт оформлен по требованиям

Снижение оценки: отсутствие smoke-теста, нет графиков, метрики не собраны, выводы отсутствуют или формальны, тесты падают.

7.2. Средний уровень — оценка «хорошо»

Все критерии базового уровня плюс:

Критерий

Реализован сценарий spike-теста (резкий скачок VUs)

Реализован сценарий stress-теста с поиском точки насыщения

Использованы пользовательские метрики (Trend, Counter, Rate)

Критерий

Настроены пороги (thresholds) — тест падает при нарушении SLA

Реализована параметризация данных (чтение из JSON/генерация)

Написан скрипт анализа с автоматическим определением точки насыщения

Построен график зависимости RPS от VUs с выделением точки насыщения

На все контрольные вопросы даны развёрнутые ответы

Снижение оценки: отсутствие spike или stress теста, нет пользовательских метрик, нет thresholds, точка насыщения не определена.

7.3. Повышенный уровень — оценка «отлично»

Все критерии базового и среднего уровней плюс:

Критерий

Настроен мониторинг в реальном времени (Grafana + InfluxDB/Prometheus) с дашбордом

Дашборд зафиксирован скриншотами во время теста

---

k6 интегрирован в CI/CD-пайплайн с Quality Gate (блокировка при нарушении порогов)

Проведено сравнение производительности при разных конфигурациях (workers, БД, кэш)

Настроен распределённый запуск (или тест с xk6-disruptor)

Критерий

Определены узкие места системы и предложены конкретные меры по оптимизации

Проведён анализ стабильности результатов (повторные прогоны с оценкой разброса метрик)

Отчёт содержит рекомендации по масштабированию системы

Снижение оценки: отсутствие мониторинга, отсутствие CI/CD, нет сравнения конфигураций, формальные рекомендации без конкретики.

Лабораторная работа №5. Статический анализ безопасности кода (SAST)

1. Цель

Сформировать практические навыки применения инструментов статического анализа безопасности кода (SAST) для раннего выявления уязвимостей, классификации их по степени критичности и последующего исправления в соответствии с лучшими практиками безопасной разработки (Secure SDLC).

2. Задачи

1. Разработать тестовое приложение на Python, содержащее преднамеренные уязвимости различных категорий (инъекции, хранение секретов, небезопасная десериализация, утечка данных, недостатки криптографии).
2. Изучить принципы работы и правила (rules) SAST-инструментов Semgrep и Bandit.
3. Выполнить статический анализ разработанного приложения с помощью Semgrep (основной) и Bandit (дополнительный).

4. Проанализировать полученные отчёты, классифицировать найденные уязвимости по категориям и степени критичности (CVSS / OWASP Top 10).

5. Устранить все найденные уязвимости в коде приложения, применяя безопасные альтернативы и паттерны.

6. Провести повторный SAST-анализ исправленного приложения и подтвердить отсутствие ранее найденных уязвимостей.

7. Настроить pre-commit hook или CI/CD-этап для автоматического SAST-сканирования.

3. Теоретическое обоснование

SAST (Static Application Security Testing) — метод анализа безопасности, при котором исходный код, байт-код или бинарный код приложения проверяется на наличие уязвимостей без его фактического выполнения. SAST относится к методологии «белого ящика» (White Box Testing) и реализует принцип Shift-Left Security — обнаружение уязвимостей на максимально ранних этапах жизненного цикла разработки.

Место SAST в Secure SDLC:

text

Планирование → Разработка → Сборка → Тестирование → Развёртывание → Эксплуатация

↑

SAST ——┤ (Shift Left)

DAST ————————————————→ (на запущенной системе)

Преимущества и ограничения SAST:

Преимущества	Ограничения
Выявление уязвимостей до выполнения кода	Ложноположительные срабатывания (False Positives)

Преимущества	Ограничения
Интеграция в IDE и CI/CD (быстрая обратная связь)	Не выявляет уязвимости конфигурации и среды выполнения
Покрытие всех ветвей кода (даже редко исполняемых)	Ограничен языковой поддержкой правил
Низкая стоимость исправления (ранний этап)	Не обнаруживает бизнес-логические уязвимости
Автоматизация и масштабируемость	Требует настройки правил под конкретный проект

Сравнение SAST и DAST:

Критерий	SAST	DAST
Когда применяется	На этапе разработки/сборки	На этапе тестирования/эксплуатации
Доступ к коду	Требуется	Не требуется
Тип анализа	Статический (без выполнения)	Динамический (на работающей системе)
Выявляемые уязвимости	Инъекции, утечки, ошибки конфигурации в коде	XSS, CSRF, ошибки аутентификации во время выполнения
Ложные срабатывания	Выше	Ниже

Критерий	SAST	DAST
Покрытие	Все пути выполнения	Только реально выполняемые пути

Семейство уязвимостей OWASP Top 10 (2021) — фокус SAST:

Категория	Примеры	Обнаружение SAST
A01: Broken Access Control	Отсутствие проверки прав	Частично (паттерны)
A02: Cryptographic Failures	Слабые алгоритмы, хардкод ключей	✓ Хорошо
A03: Injection	SQLi, OS Command, LDAP	✓ Отлично
A04: Insecure Design	Отсутствие валидации, rate limiting	Частично
A05: Security Misconfiguration	Debug=True, открытые порты	✓ Хорошо
A06: Vulnerable Components	Устаревшие зависимости	Через SCA
A07: Identification Failures	Слабые пароли, отсутствие MFA	Частично
A08: Software Integrity Failures	Небезопасная десериализация	✓ Хорошо

Категория	Примеры	Обнаружение SAST
A09: Logging & Monitoring Failures	Утечка sensitive data в логи	✓ Хорошо
A10: SSRF	Подделка серверных запросов	Частично

Semgrep — опенсорсный SAST-инструмент, использующий паттерн-матчинг на основе абстрактного синтаксического дерева (AST). Ключевые особенности:

Особенность	Описание
Язык правил	Декларативный DSL, похожий на целевой язык
Поддерживаемые языки	Python, JavaScript, Java, Go, C/C++, Ruby, PHP, Rust, Terraform, Dockerfile (30+)
Реестр правил	Semgrep Registry: 2000+ готовых правил от сообщества
Режимы	Поиск (search), автоматическое исправление (autofix)
Интеграция	CLI, GitHub Actions, GitLab CI, pre-commit, VS Code
Производительность	Многопоточное сканирование, инкрементальное (только изменённые файлы)

Bandit — SAST-инструмент, специализированный для Python и разработанный OpenStack Security Group. Особенности:

- Анализирует AST-дерево Python-модулей.
- Фокус на типичных проблемах безопасности в Python-коде.
- Встроенные плагины для проверок: использование eval, pickle, os.system, subprocess shell=True.
- Профили: низкая/средняя/высокая строгость.

Классификация критичности уязвимостей (CVSS v3.1):

Уровень	Баллы CVSS	Пример
Critical	9.0–10.0	Удалённое выполнение кода (RCE)
High	7.0–8.9	SQL-инъекция, утечка секретов
Medium	4.0–6.9	XSS, слабые криптоалгоритмы
Low	0.1–3.9	Раскрытие отладочной информации

Стандарт CWE (Common Weakness Enumeration) — каталог типов уязвимостей, используемый для классификации:

- CWE-89: SQL Injection
- CWE-78: OS Command Injection
- CWE-798: Hardcoded Credentials
- CWE-502: Deserialization of Untrusted Data
- CWE-327: Use of Broken Cryptographic Algorithm
- CWE-259: Hardcoded Password
- CWE-532: Sensitive Information in Logs

4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

Разработать приложение (Python или JS) с 5+ преднамеренными уязвимостями из списка, провести SAST-анализ Semgrep/Bandit, составить отчёт, исправить все уязвимости, подтвердить исправление повторным анализом.

	Тип приложения	Обязательные категории уязвимостей
	REST API (Flask/FastAPI)	SQLi, hardcoded secret, eval, pickle, debug-mode
	Веб-сервер (Express.js)	SQLi, eval, xss-reflected, hardcoded JWT, небезопасные заголовки
	CLI-утилита	OS command injection, hardcoded secrets, unsafe yaml.load, tempfile race condition
	Скрипт обработки данных	Eval, pickle десериализация, path traversal, logging secrets
	Бот (Telegram/Discord)	Command injection, hardcoded API token, eval, insecure requests
	Парсер файлов	Path traversal, XML external entity (XXE), unsafe deserialization
	Микросервис аутентификации	SQLi, weak password hashing, hardcoded JWT secret, отсутствие rate limiting
	Сервис загрузки файлов	Path traversal, RCE через загрузку, небезопасные имена файлов

	Тип приложения	Обязательные категории уязвимостей
	Генератор отчётов	SSTI (Server-Side Template Injection), XSS, eval
0	Платёжный модуль	Hardcoded keys, weak encryption, logging карт, отсутствие валидации
1	Админ-панель	SQLi, XSS stored, CSRF отсутствует, debug=True, default credentials
2	Планировщик задач	OS command injection, unsafe cron парсинг, pickle
3	Поисковый сервис	SQLi, NoSQL-инъекция (если MongoDB), XSS, path traversal в индексе
4	API кэширования	Redis injection, pickle, hardcoded password
5	Конфигурационный менеджер	YAML deserialization, раскрытие секретов, слабый мастер-ключ

4.2. Средний уровень (15 вариантов)

Дополнительно: написать собственные правила для Semgrep (3+), интегрировать pre-commit hook, классифицировать уязвимости по CWE и CVSS, настроить подавление ложных срабатываний (nosemgrep), сравнить результаты Semgrep и Bandit.

	Приложение	Дополнительные требования

	Приложение	Дополнительные требования
	REST API с JWT	Собственное правило: проверка срока жизни токена; сравнение Semgrep vs Bandit
	Express.js сервер	Правило: обнаружение res.send с пользовательским вводом (XSS); интеграция pre-commit
	Асинхронный обработчик	Правило: aiohttp без таймаутов; классификация по CVSS
	ORM-сервис (SQLAlchemy)	Правило: обнаружение text() без параметров; nosemgrep для false positives
	WebSocket-сервер	Правило: отсутствие валидации origin; сравнение инструментов
	GraphQL API	Правило: отсутствие depth limiting; pre-commit hook
	gRPC-сервис	Правило: insecure credentials; CWE classification
	Lambda-функция	Правило: environment variables exposed; сравнение Semgrep/Bandit
	Микросервис на Go	Правило: использование template без escaping; nosemgrep
0	Django-приложение	Правило: DEBUG=True в Settings; классификация по OWASP

	Приложение	Дополнительные требования
1	Terraform-конфигурация	Правило: открытые security groups; сравнение с Checkov
2	Dockerfile	Правило: latest tag, root user; pre-commit интеграция
3	Ansible-плейбук	Правило: хардкод паролей; CWE mapping
4	Kubernetes-манифесты	Правило: privilege escalation; nosemgrep подавление
5	CI/CD-пайплайн (GitLab)	Правило: unprotected variables; CVSS scoring

4.3. Повышенный уровень (15 вариантов)

Дополнительно: автоматическая фиксация уязвимостей через Semgrep autofix, интеграция в CI/CD с Quality Gate, настройка пайплайна с SARIF и Security Dashboard, тюнинг правил для снижения false positive rate (<5%), исследование эзотерических уязвимостей (прототипное загрязнение, timing attacks, XXE, SSTI, десериализационные гаджеты).

	Тема	Расширенные требования
	Полный Secure SDLC	autofix + CI/CD с Security Gate (пайплайн блокируется)
	Security Dashboard	SARIF загрузка в GitHub Security / GitLab Vulnerability Report
	Тюнинг правил	Анализ false positives: снижение до

	Тема	Расширенные требования
		<5%, кастомные severity
	Prototype Pollution (JS)	Обнаружение через кастомное правило Semgrep
	Timing Attack	Специализированное правило для небезопасного сравнения
	XXE (XML External Entity)	Правило для xml.etree/lxml без защиты; autofix
	SSTI в Jinja2/Twig	Правило для render_template_string с пользовательским вводом
	Десериализационные гаджеты	Анализ цепочек гаджетов (pickle/yaml); autofix
	Insecure Deserialization (Java)	Правило для ObjectInputStream без фильтра
0	SSRF через HTTP-клиент	Правило для requests.get с пользовательским URL
1	LDAP Injection	Правило для небезопасных фильтров LDAP
2	NoSQL Injection (MongoDB)	Правило для $where, небезопасных операторов
3	Open Redirect	Правило для redirect() с пользовательским вводом

	Тема	Расширенные требования
4	Regular Expression DoS (ReDoS)	Правило для злых регулярных выражений
5	Multi-language SAST	Правила для Python + JS + Dockerfile в одном проекте; CI/CD с SARIF

5. Контрольные вопросы

1. Что такое SAST и в чём его отличие от DAST? На каких этапах SDLC применяется каждый из методов?
2. Каковы основные преимущества и ограничения статического анализа безопасности?
3. Какие категории уязвимостей OWASP Top 10 наиболее эффективно обнаруживаются SAST-инструментами?
4. Как работает Semgrep на уровне AST? Что такое паттерн-матчинг и чем он отличается от сигнатурного анализа?
5. Какие типы уязвимостей специфичны для Python и обнаруживаются Bandit/Semgrep (eval, pickle, shell injection)?
6. Что такое CWE (Common Weakness Enumeration) и CVSS (Common Vulnerability Scoring System)? Как они используются при анализе отчётов SAST?
7. Как организовать подавление ложноположительных срабатываний (false positives) в Semgrep? Что такое nosemgrep?
8. Какие безопасные альтернативы существуют для pickle.loads(), eval(), os.system(), hashlib.md5()?
9. Как интегрировать SAST-инструменты в CI/CD-пайплайн и настроить Quality Gate на основе severity найденных уязвимостей?
10. Что такое Semgrep autofix и в каких случаях его применение безопасно, а в каких — требует ручной проверки?

6. Требования к отчёту

1. Титульный лист: ФИО, группа, вариант, номер и тема работы.
2. Цель и задачи работы.
3. Теоретическое обоснование: принципы SAST, классификация уязвимостей, обзор инструментов (1–2 стр.).
4. Практическая часть:
   - Листинг уязвимого приложения с выделением и нумерацией каждой уязвимости и указанием CWE.
   - Конфигурация правил Semgrep (.semgrep.yml) и/или профиля Bandit с комментариями.
   - Листинг (фрагменты) отчёта SAST с пояснениями.
   - Листинг исправленного приложения с комментариями о применённых исправлениях.
   - Скрипт сравнения результатов SAST до и после исправления.
   - Листинг pre-commit/CI/CD конфигурации.
5. Результаты выполнения:
   - Скриншот вывода Semgrep/Bandit с найденными уязвимостями (терминал).
   - Таблица уязвимостей: ID, CWE, CVSS (расчётный), severity, файл:строка, статус исправления.
   - Скриншот вывода повторного анализа, показывающий 0 findings (или только допустимые).
   - График или диаграмма: распределение уязвимостей по категориям до и после.
   - Скриншот срабатывания pre-commit хука (блокировка коммита).
6. Сравнительный анализ Semgrep vs Bandit (для среднего уровня).
7. Анализ ложноположительных срабатываний (для повышенного: количество, причины, способ подавления).
8. Ответы на контрольные вопросы.
9. Вывод: освоенные навыки, достигнута ли цель, практическая ценность SAST в процессе разработки.

7. Критерии оценивания

7.1. Базовый уровень — оценка «удовлетворительно»

Критерий

Разработано приложение с не менее чем 5 преднамеренными уязвимостями из разных категорий

Настроен и запущен Semgrep или Bandit

Найдены и зафиксированы в отчёте все 5+ уязвимостей

Все уязвимости классифицированы (CWE и/или OWASP)

Разработана исправленная версия приложения с устранением всех найденных уязвимостей

Проведён повторный SAST-анализ, подтверждающий отсутствие ранее найденных уязвимостей

Отчёт оформлен по требованиям

Снижение оценки: менее 5 уязвимостей, отсутствие классификации, исправлены не все уязвимости, повторный анализ не проведён, отсутствует обоснование исправлений.

7.2. Средний уровень — оценка «хорошо»

Все критерии базового уровня плюс:

Критерий

Написано не менее 3 собственных правил для Semgrep

Настроен pre-commit hook, блокирующий коммит при наличии уязвимостей severity ERROR

Критерий

Проведена классификация уязвимостей по CWE с указанием CVSS-балла

Использован механизм nosemgrep для подавления как минимум 1 ложного срабатывания с обоснованием

Проведено сравнение результатов Semgrep и Bandit, выявлены расхождения

Таблица уязвимостей содержит столбцы: CWE, CVSS, severity, рекомендация по исправлению

На все контрольные вопросы даны развёрнутые ответы

Снижение оценки: отсутствие собственных правил, отсутствие pre-commit, нет CVSS-оценок, нет сравнения инструментов.

7.3. Повышенный уровень — оценка «отлично»

Все критерии базового и среднего уровней плюс:

Критерий

Применён Semgrep autofix для автоматического исправления не менее 3 уязвимостей

SAST интегрирован в CI/CD-пайплайн (GitLab CI / GitHub Actions) с Quality Gate

---

Отчёт SAST экспортирован в SARIF-формате и визуализирован (GitHub Security / GitLab Dashboard)

Проведён тюнинг правил: false positive rate снижен до <5% (подтверждено расчётом)

Критерий

Исследована и зафиксирована минимум 1 эзотерическая уязвимость (SSTI, XXE, prototype pollution, etc.)

Проведён анализ времени выполнения SAST-сканирования и предложены оптимизации

Отчёт содержит раздел «Метрики безопасности» (Security Debt, % устранённых уязвимостей)

Снижение оценки: отсутствие autofix, CI/CD или SARIF, нет тюнинга правил, нет эзотерической уязвимости.

Лабораторная работа №6. Настройка качественных ворот в CI/CD

1. Цель

Сформировать практические навыки проектирования и реализации комплексного CI/CD-пайплайна, интегрирующего все изученные виды автоматизированного тестирования, с настройкой Quality Gates — автоматических проверок, блокирующих продвижение артефакта при недостижении заданных пороговых значений ключевых метрик качества.

2. Задачи

1. Спроектировать архитектуру CI/CD-пайплайна, определяющую последовательность этапов тестирования и критерии перехода между ними.
2. Настроить файл конфигурации CI/CD (.gitlab-ci.yml и/или .github/workflows/) с этапами сборки, модульного тестирования, SAST, интеграционного тестирования и нагрузочного тестирования.
3. Реализовать Quality Gates для модульного тестирования (покрытие кода ≥ 80%), SAST (отсутствие HIGH/CRITICAL уязвимостей) и нагрузочного тестирования (p95 latency ≤ SLA).

4. Настроить публикацию отчётов о тестировании (JUnit XML, coverage report, SAST SARIF, нагрузочные метрики) в качестве артефактов пайплайна.

5. Реализовать условное выполнение этапов (нагрузочное тестирование только в main-ветке или по расписанию).

6. Провести демонстрацию срабатывания Quality Gates: успешный пайплайн и пайплайн, заблокированный из-за недостижения порогов.

3. Теоретическое обоснование

CI/CD-пайплайн (Continuous Integration / Continuous Delivery Pipeline) — автоматизированная последовательность этапов, через которые проходит код от момента коммита до развёртывания в целевое окружение. В контексте качества ПО пайплайн выполняет роль оркестратора всех видов тестирования.

Quality Gate (Качественные ворота) — автоматическая проверка, которая блокирует дальнейшее продвижение артефакта (слияние MR/PR, переход на следующий этап, развёртывание) при недостижении заданных метрик качества. Quality Gates реализуют принцип "Fail Fast" — быстрое обнаружение проблем и предотвращение их распространения.

Принципы проектирования Quality Gates:

Принцип	Описание
Измеримость	Каждые ворота основаны на количественной метрике с чётким порогом
Автоматизация	Проверка выполняется без участия человека, результат бинарный (Pass/Fail)
Контекстность	Пороги зависят от типа приложения и этапа разработки

Принцип	Описание
Постепенность	Ворота ужесточаются по мере приближения к production
Воспроизводимость	Результат проверки одинаков при повторных запусках

Сравнение GitLab CI и GitHub Actions:

Критерий	GitLab CI	GitHub Actions
Файл конфигурации	.gitlab-ci.yml	.github/workflows/*.yml
Модель выполнения	Стадии (stages) + джобы	Workflows → Jobs → Steps
Артефакты	artifacts:, reports:	actions/upload-artifact
Кэширование	cache:	actions/cache
Переменные окружения	UI + .gitlab-ci.yml	UI + env:
Quality Gates	allow_failure: false	if: failure() + actions
Интеграция с ревью кода	Встроенные MR-виджеты	Через Status Checks
Бесплатные минуты	400 мин/мес (Free)	2000 мин/мес (Free)

Виды Quality Gates в пайплайне тестирования:

Этап	Метрика	Типовой порог	Действие при провале
Unit Tests	Coverage %	≥ 80%	Блокировка MR
Unit Tests	Pass rate	100%	Блокировка MR
SAST	HIGH/CRITICAL vulns	0	Блокировка MR
SAST	MEDIUM vulns	< 5	Предупреждение
Integration	Test pass rate	100%	Блокировка MR
Load	P95 latency	< SLA (500ms)	Блокировка деплоя
Load	Error rate	< 1%	Блокировка деплоя

Артефакты тестирования в CI/CD:

Формат	Назначение	Инструменты
JUnit XML	Результаты тестов	pytest, JUnit, Jest, Newman
Cobertura	Покрытие кода	pytest-cov,

Формат	Назначение	Инструменты
XML		JaCoCo, coverage.py
SARIF	Результаты SAST	Semgrep, CodeQL, Bandit
JSON	Метрики производительности	k6, JMeter, Locust
HTML	Человекочитаемые отчёты	Все вышеперечисленные

Паттерн "Пирамида тестирования" в CI/CD:

text

```
        ╱  E2E (медленные)  ╲        ← по расписанию / main
       ╱   Интеграционные    ╲       ← каждый MR
      ╱    SAST / Линтеры     ╲      ← каждый коммит
     ╱   Модульные тесты       ╲     ← каждый коммит
    ╱_____________________________╲
```

Время выполнения: быстрее → медленнее

Частота запуска: чаще → реже

4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

Настроить CI/CD-пайплайн для указанного проекта с этапами build → unit tests → SAST → integration tests. Реализовать Quality Gates для модульного тестирования (покрытие ≥ 80%) и SAST (нет HIGH уязвимостей). Настроить публикацию отчётов.

	Проект	CI/CD-система	Особенности
	Калькулятор (Python)	GitLab CI	Покрытие через pytest-cov, SAST через Bandit
	Todo API (FastAPI)	GitHub Actions	Покрытие + SAST Semgrep
	Express.js сервер	GitLab CI	Jest coverage, ESLint security plugin
	Django-приложение	GitHub Actions	Coverage + Bandit
	Библиотека для матриц	GitLab CI	Покрытие 85% (повышенный порог)
	Spring Boot API	GitLab CI	JaCoCo coverage, SpotBugs
	Flask-микросервис	GitHub Actions	SAST Semgrep + Bandit
	Go-утилита (CLI)	GitLab CI	Go test -cover, gosec
	React-приложение	GitHub Actions	Jest coverage, ESLint, npm audit
0	GraphQL API (Python)	GitLab CI	Покрытие 80%, Semgrep

	Проект	CI/CD-система	Особенности
1	Валидатор данных	GitHub Actions	Множественные SAST-инструменты
2	Парсер конфигураций	GitLab CI	Покрытие 90%, Bandit
3	Node.js библиотека	GitHub Actions	Jest + ESLint security
4	REST-клиент (Python)	GitLab CI	Интеграционные тесты с мок-сервером
5	Монорепозиторий (2 сервиса)	GitHub Actions	Разные пороги для разных сервисов

4.2. Средний уровень (15 вариантов)

Дополнительно: добавить этап нагрузочного тестирования с Quality Gate по P95 latency, реализовать условное выполнение (нагрузочные тесты только в main), настроить автоматический комментарий в MR/PR с результатами Quality Gates, настроить динамическое окружение для review-приложений.

	Проект	Дополнительные требования
	E-commerce API	k6 нагрузочные тесты, P95 < 500ms Gate, PR-комментарий
	Микросервис	Spike-тест, условный запуск в main

	Проект	Дополнительные требования
	аутентификации	
	Сервис загрузки файлов	Нагрузочный тест с разными размерами, динамический review-app
	Поисковый движок (Elastic)	k6 тест поиска, P95 < 300ms, PR-виджет
	API Gateway	Rate limiting тест, QA-отчёт в MR
	Платёжный сервис (mock)	Stress-тест + Quality Gate, автоматический бейдж
	Аналитический пайплайн	k6 + Grafana, PR-комментарий с графиками
	Чат-сервер (WebSocket)	WS-нагрузочный тест, review-app
	Image Processing API	Нагрузка с файлами, Gate по error rate
0	Recommendation Engine	P99 < 1000ms Gate, условный запуск
1	URL Shortener	Redis-нагрузка, динамическое окружение
2	Inventory Service	Нагрузка с БД, PR-комментарий

	Проект	Дополнительные требования
3	Notification Service	Spike-тест очередей, Gate по latency
4	Config Management API	Нагрузка + Gate, review-app
5	Монорепо 3 сервиса	Координация Quality Gates между сервисами

4.3. Повышенный уровень (15 вариантов)

Дополнительно: каскадные Quality Gates (разные пороги для разных веток), интеграция с Security Dashboard (GitLab Vulnerability Report / GitHub Security), кастомные метрики и визуализация трендов, настройка политик слияния (merge only if all gates passed), реализация canary-анализа через сравнение метрик, полная матрица тестирования.

	Задание	Расширенные требования
	Полный CI/CD с каскадом	Feature: coverage 70%, Main: 90%; роли в MR
	Security-First Pipeline	SARIF загрузка, Security Dashboard, политика слияния
	Канареечный анализ	Полный CI/CD + canary-деплой + сравнение метрик
	Метрики качества во времени	InfluxDB/Grafana дашборд с трендами Coverage/Findings/P95

	Задание	Расширенные требования
	Матричное тестирование	Python 3.9/3.10/3.11 × PostgreSQL/MySQL
	Multi-arch сборка	amd64 + arm64, разные пороги
	Service-Level Objectives	SLO-based Gates: 99.9% availability, p95 < 200ms
	Cross-service Quality Gates	Сервис A тесты + Сервис B тесты → общий Gate
	Policy-as-Code (Open Policy Agent)	OPA-правила для Quality Gates
0	GitLab Ultimate Security	DAST + SAST + Dependency Scanning + Container Scanning
1	Progressive Delivery Gates	10% → 50% → 100% трафика с проверкой метрик
2	Compliance Pipeline	SOC2/GDPR-ворота, аудит-лог всех Quality Gates
3	Self-healing Pipeline	Автоматический откат при провале Gate + уведомление
4	Chaos Engineering Gate	Chaos Mesh/k6-disruptor + анализ метрик
5	AI-assisted Gate decisions	ML-модель для динамических порогов на основе истории

5. Контрольные вопросы

1. Что такое Quality Gates в CI/CD и какова их роль в обеспечении качества ПО?
2. Какие метрики качества наиболее часто используются в качестве Quality Gates? Обоснуйте выбор пороговых значений.
3. В чём разница между allow_failure: true и allow_failure: false в GitLab CI? Как это влияет на дальнейшие этапы пайплайна?
4. Какие форматы отчётов о тестировании используются в CI/CD (JUnit, Cobertura, SARIF) и для чего нужна их стандартизация?
5. Как организовать условное выполнение этапов пайплайна (например, нагрузочное тестирование только для main-ветки)?
6. Каковы преимущества и недостатки размещения Quality Gates на разных этапах пайплайна (ранние vs поздние ворота)?
7. Как обеспечить атомарность и воспроизводимость Quality Gates в распределённых CI/CD-системах?
8. Какие механизмы визуализации результатов Quality Gates доступны в GitLab и GitHub (Merge Request виджеты, Security Dashboards)?
9. Как организовать каскадные Quality Gates с разными порогами для разных веток (feature → develop → main)?
10. Какие проблемы могут возникнуть при внедрении строгих Quality Gates в legacy-проектах и какие стратегии миграции можно применить?

6. Требования к отчёту

1. Титульный лист: ФИО, группа, вариант, номер и тема работы.
2. Цель и задачи работы.
3. Теоретическое обоснование: принципы CI/CD пайплайнов, Quality Gates, артефакты (1–2 стр.).
4. Практическая часть:
   - Архитектурная схема спроектированного пайплайна (блок-схема с этапами и воротами).

- Полный листинг .gitlab-ci.yml и/или .github/workflows/ci.yml с комментариями к каждому этапу.
- Описание Quality Gates: метрика, пороговое значение, этап, действие при провале.
- Скрипты проверки Quality Gates (встроенные в CI/CD).

5. Результаты выполнения:

- Скриншот успешного пайплайна (все этапы зелёные, все Gates пройдены).
- Скриншот пайплайна, заблокированного на этапе unit-tests (покрытие < порога).
- Скриншот пайплайна, заблокированного на этапе SAST (найдены HIGH-уязвимости).
- Скриншот MR/PR с виджетом Quality Gates (статус проверок).
- Скриншот опубликованных артефактов (coverage report, SAST report).
- Для повышенного: скриншот Security Dashboard с трендами.

6. Таблица Quality Gates: этап, метрика, порог, фактическое значение, статус.

7. Анализ результатов: анализ причин провала Gates (для сценариев с блокировкой), способы исправления.

8. Сравнение GitLab CI и GitHub Actions (для среднего уровня).

9. Ответы на контрольные вопросы.

10. Вывод.

7. Критерии оценивания

7.1. Базовый уровень — оценка «удовлетворительно»

Критерий
Настроен файл CI/CD (.gitlab-ci.yml или .github/workflows/ci.yml) не менее чем с 4 стадиями

Критерий

Этап unit-tests запускает тесты и проверяет покрытие (порог задан)

Этап SAST запускает Semgrep/Bandit и проверяет отсутствие HIGH/CRITICAL уязвимостей

Настроена публикация отчётов как артефактов (JUnit, coverage, SAST)

Продемонстрирован успешный прогон пайплайна (все Gates пройдены)

Продемонстрирован прогон с блокировкой (покрытие < порога ИЛИ найдены HIGH-уязвимости)

Отчёт оформлен по требованиям

Снижение оценки: менее 4 стадий, отсутствие Quality Gate, артефакты не публикуются, отсутствует демонстрация блокировки.

7.2. Средний уровень — оценка «хорошо»

Все критерии базового уровня плюс:

Критерий

Добавлен этап нагрузочного тестирования (k6) с Quality Gate по P95 latency и/или error rate

---

Реализовано условное выполнение этапов (нагрузочное тестирование — main/schedule)

Настроен автоматический комментарий в MR/PR с результатами Quality Gates

---

Настроено динамическое окружение для review-приложений (или заглушка с пояснением)

Критерий

Проведён сравнительный анализ GitLab CI vs GitHub Actions

Оформлена архитектурная схема пайплайна

На все контрольные вопросы даны развёрнутые ответы

Снижение оценки: отсутствие нагрузочного этапа, нет условного выполнения, нет PR-комментария, нет схемы.

7.3. Повышенный уровень — оценка «отлично»

Все критерии базового и среднего уровней плюс:

Критерий

Реализованы каскадные Quality Gates с разными порогами для разных веток (feature/main)

Настроена интеграция с Security Dashboard (SARIF загрузка)

Реализован сбор метрик качества во времени и их визуализация (Grafana / GitLab Pages)

Настроены политики слияния (merge only if all Gates passed)

Реализован один из сценариев: canary-анализ, матричное тестирование, или Open Policy Agent Gates

---

Проведён анализ стабильности пайплайна: ложные срабатывания Gates, время выполнения, узкие места

Отчёт содержит раздел по оптимизации пайплайна (время, ресурсы, параллелизация)

Снижение оценки: отсутствие каскадных ворот, нет Security Dashboard, нет визуализации, нет canary/матрицы/OPA.

Лабораторная работа №7. Тестирование асинхронных систем и очередей сообщений

1. Цель

Сформировать практические навыки интеграционного тестирования асинхронных распределённых систем, использующих очереди сообщений (RabbitMQ), с применением контейнеризированного тестового окружения через библиотеку Testcontainers для обеспечения изолированных, воспроизводимых и автоматизированных проверок.

2. Задачи

1. Разработать простую систему асинхронного обмена сообщениями, состоящую из Producer (отправитель) и Consumer (обработчик) на основе RabbitMQ.
2. Развернуть RabbitMQ в тестовом окружении с использованием Docker Testcontainers (Python testcontainers или Java testcontainers).
3. Написать интеграционные тесты, проверяющие полный цикл: отправка сообщения → получение → обработка → подтверждение.
4. Протестировать сценарии обработки ошибок: некорректное сообщение, повторная обработка (retry), перенаправление в Dead Letter Queue (DLQ).
5. Реализовать тесты на идемпотентность Consumer'a (повторная обработка одного сообщения).

6. Настроить автоматический запуск тестов в CI/CD с использованием Docker-in-Docker или Testcontainers Cloud.

3. Теоретическое обоснование

Асинхронные системы и очереди сообщений — архитектурный паттерн, при котором компоненты системы взаимодействуют через промежуточный брокер сообщений, не требуя одновременной доступности отправителя и получателя. Это обеспечивает:

- Разделение во времени (Temporal Decoupling): Producer и Consumer могут работать независимо.
- Устойчивость к нагрузке: очередь сглаживает пики нагрузки (load leveling).
- Надёжность: сообщения сохраняются на диске и не теряются при сбоях.
- Масштабирование: можно увеличивать количество Consumer'ов для параллельной обработки.

Ключевые понятия AMQP (RabbitMQ):

Компонент	Описание
Producer	Отправляет сообщения в Exchange
Exchange	Принимает сообщения и маршрутизирует их в очереди по правилам
Binding	Связь между Exchange и Queue с ключом маршрутизации
Queue	Буфер сообщений, ожидающих обработки
Consumer	Получает и обрабатывает сообщения из очереди

Компонент	Описание
Acknowledgement (ACK)	Подтверждение успешной обработки сообщения
NACK / Reject	Отказ от сообщения (с requeue или без)
Dead Letter Queue (DLQ)	Очередь для сообщений, которые не удалось обработать
TTL (Time-To-Live)	Время жизни сообщения в очереди
Durable	Сохраняется ли очередь/сообщение после перезапуска брокера

Паттерны обработки ошибок в асинхронных системах:

text

Producer → Exchange → Queue → Consumer

↓

Обработка ОК? —— Да —→ ACK (удаление из очереди)

↓

Нет

↓

Повторная попытка? — Да —→ Requeue (вернуть в очередь)

↓

Нет

↓

Dead Letter Queue (DLQ)

↓

Анализ / алертинг / ручная обработка

Стратегии повторной обработки (Retry Policies):

Стратегия	Описание	Применение
Immediate Retry	Немедленная повторная обработка	Временные сбои (сеть)
Delayed Retry	Повтор через увеличивающиеся интервалы (exponential backoff)	Перегрузка зависимых сервисов
Max Retries	Ограничение количества попыток	Предотвращение бесконечного цикла
DLQ	Перенаправление после исчерпания попыток	Ручной разбор проблем

Testcontainers — библиотека с открытым исходным кодом для создания ephemeral (временных) контейнеров Docker непосредственно из тестового кода. Принцип работы:

1. Тест запускается
2. Testcontainers проверяет наличие Docker-образа
3. Запускается контейнер с нужным сервисом (RabbitMQ, PostgreSQL, Kafka...)
4. Тест ожидает готовности контейнера (healthcheck)
5. Выполняется тест с реальным сервисом
6. Контейнер автоматически останавливается и удаляется

Преимущества Testcontainers для тестирования очередей:

Преимущество	Описание

Преимущество	Описание
Изоляция	Каждый тест получает свой экземпляр RabbitMQ
Воспроизводимость	Одинаковое окружение на CI и локально
Отсутствие моков	Тест работает с реальным брокером
Автоматизация	Не требуется ручная настройка окружения
Очистка	После тестов не остаётся «мусора»

Сравнение подходов к тестированию очередей:

Подход	Плюсы	Минусы
Mock брокера	Очень быстро, не нужен Docker	Не проверяет реальное поведение (сериализация, таймауты, DLQ)
Встроенный брокер	Быстро, без внешних зависимостей	Может не поддерживать все фичи (например, DLQ в embedded RabbitMQ)
Docker Compose	Реальное окружение	Ручной запуск, проблемы с параллельными тестами
Testcontainers	Автоматически, изолированно, реально	Требует Docker, медленнее моков

Идемпотентность Consumer'a — способность обработчика корректно реагировать на повторную доставку одного и того же сообщения (at-least-once delivery). Достигается через:

- Уникальный идентификатор сообщения (Message ID).

- Хранилище обработанных идентификаторов (Redis, БД).
- Идемпотентные операции (UPSERT вместо INSERT).

Метрики для тестирования очередей:

Метрика	Значение
Время от отправки до ACK	< 1000ms
Количество сообщений в DLQ	0 (для корректных сообщений)
Rate отправки	Пропускная способность Producer'a
Rate обработки	Пропускная способность Consumer'a
Retry rate	% сообщений, потребовавших повтор

4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

Разработать систему Producer-Consumer для указанного сценария. Написать интеграционные тесты с Testcontainers, проверяющие: успешную отправку и обработку сообщения, обработку некорректного сообщения (DLQ).

	Сценарий	Тип сообщений	Особенности
	Email-рассылка	Отправка писем подписчикам	Валидация email, персонализация
	Генерация PDF-отчётов	Создание отчётов по шаблону	Разные форматы, статус готовности

	Сценарий	Тип сообщений	Особенности
	Обработка изображений	Ресайз / наложение водяного знака	Разные размеры, форматы
	Система лайков/оценок	Учёт голосов пользователей	Дедупликация голосов
	СМС-уведомления	Отправка SMS через шлюз	Лимиты, валидация номера
	Индексация документов	Добавление в поисковый индекс	Полнотекстовый поиск
	Синхронизация каталога	Обновление цен/остатков	Конфликтующие обновления
	Система достижений	Начисление баллов/badges	Проверка условий
	Webhook-диспетчер	Отправка webhook на URL	Retry при недоступности
0	Генератор превью	Создание миниатюр видео	Длительность, формат
1	Архивация данных	Сжатие и перемещение в cold storage	Размер файлов
2	Модерация контента	Проверка текста на спам/мат	Классификация

	Сценарий	Тип сообщений	Особенности
3	Платёжный процессинг	Списание средств	Идемпотентность критична
4	Экспорт данных	Выгрузка в CSV/Excel	Большие объёмы
5	Пуш-уведомления	Отправка в Firebase/APNs	Токены устройств

4.2. Средний уровень (15 вариантов)

Дополнительно: реализовать retry с exponential backoff, Dead Letter Queue с причинами, тесты на конкурентную обработку (несколько Consumer'ов), тесты на порядок обработки (при необходимости FIFO).

	Сценарий	Дополнительные требования
	Оркестрация заказа	Шаги: резервирование → оплата → доставка; компенсирующие транзакции
	Потоковая обработка логов	Парсинг → агрегация → алертинг; окна времени
	Микросервисная сага (Saga)	Распределённая транзакция с компенсацией
	Приоритетная очередь	Сообщения с приоритетом 1–10; порядок обработки
	Пакетная обработка (Batching)	Накопление до N сообщений или T секунд
	Dead Letter с	Ручной replay из DLQ; exponential

	Сценарий	Дополнительные требования
	повтором	backoff
	Конкурентные Consumer'ы	3+ Consumer'a; проверка балансировки (round-robin)
	Отложенные задачи (Delayed)	TTL + Delayed Exchange; планирование в будущем
	Circuit Breaker в Consumer	Остановка обработки при лавине ошибок
0	RPC через очереди	Request-Reply паттерн; correlation ID
1	Подтверждение доставки	Publisher Confirms; обработка NACK от брокера
2	Шардирование очередей	Consistent hashing для ключа маршрутизации
3	Истечение (TTL) сообщений	Сообщение удаляется, если не обработано за N секунд
4	Обратное давление (Backpressure)	Prefetch limit, замедление Producer'a
5	Вложенные задачи	Одно сообщение порождает дочерние задачи

4.3. Повышенный уровень (15 вариантов)

Дополнительно: настроить CI/CD с Docker-in-Docker для Testcontainers, реализовать метрики и мониторинг очереди в тестах, тестирование с network

disruption (потеря соединения), сравнение RabbitMQ vs Kafka через Testcontainers, тестирование схемы сообщений (JSON Schema / Avro).

	Тема	Расширенные требования
	RabbitMQ в CI/CD	GitLab CI / GitHub Actions + Docker-in-Docker + Testcontainers
	Kafka vs RabbitMQ	Оба брокера через Testcontainers; сравнение latency и throughput
	Chaos Engineering	Network disruption в Docker (toxiproxy); тесты на разрыв соединения
	Avro Schema Testing	Валидация сообщений по Avro-схеме; Schema Registry
	Мониторинг очередей	Prometheus + Grafana метрики в тестах; проверка алертов
	Exactly-Once семантика	Идемпотентный Producer + дедупликация + транзакции
	Multi-broker federation	RabbitMQ Federation; тест отказоустойчивости
	Dynamic Rebalance Testing	Добавление/удаление Consumer'ов в рантайме
	Large Message Handling	Сообщения > 10MB; chunking стратегия
0	Kafka Streams тестирование	TopologyTestDriver + реальный Kafka через Testcontainers

	Тема	Расширенные требования
1	Message Tracing	OpenTelemetry tracing через очередь; корреляция трейсов
2	Protocol Buffers	Protobuf-сериализация + валидация; сравнение с JSON
3	Аудит сообщений	Логирование всех операций; проверка audit log в тестах
4	Контроль доступа	Virtual Hosts, пользователи, права; тесты изоляции
5	Сравнение AMQP брокеров	RabbitMQ vs ActiveMQ vs Qpid через Testcontainers

5. Контрольные вопросы

1. Что такое очередь сообщений и какие проблемы архитектуры она решает (decoupling, load leveling, resilience)?
2. Что такое Dead Letter Queue (DLQ) и в каких сценариях сообщения попадают в DLQ?
3. Какие стратегии повторной обработки (retry) существуют и чем отличается immediate retry от exponential backoff?
4. Что такое Testcontainers и какие преимущества он даёт при тестировании интеграций с внешними сервисами?
5. Как обеспечить идемпотентность Consumer'а? Почему это важно при at-least-once доставке?
6. В чём разница между basic.ack, basic.nack и basic.reject в AMQP? Когда применять каждый?
7. Какие гарантии доставки сообщений существуют (at-most-once, at-least-once, exactly-once) и как они реализуются в RabbitMQ?

8. Как организовать параллельную обработку сообщений несколькими Consumer'ами? Что такое prefetch count?

9. Какие метрики мониторинга очередей важны для оценки здоровья системы (queue depth, consumer lag, redelivered rate)?

10. Как интегрировать Testcontainers в CI/CD-пайплайн, если на раннере нет прямого доступа к Docker?

6. Требования к отчёту

1. Титульный лист: ФИО, группа, вариант, номер и тема работы.
2. Цель и задачи работы.
3. Теоретическое обоснование: асинхронные системы, AMQP, паттерны обработки ошибок, Testcontainers (1–2 стр.).
4. Практическая часть:
   - Листинг Producer'a (с комментариями).
   - Листинг Consumer'a с логикой retry/DLQ (с комментариями).
   - Листинг обработчиков сообщений.
   - Листинг интеграционных тестов (все тестовые классы).
   - Диаграмма потоков: Producer → Exchange → Queue → Consumer → DLQ.
   - Конфигурация CI/CD (если применимо).
5. Результаты выполнения:
   - Скриншот успешного запуска всех тестов с выводом pytest.
   - Скриншот веб-интерфейса RabbitMQ (management UI) с очередями и сообщениями (в процессе теста).
   - Скриншот логов Consumer'a с сообщениями об обработке (ACK/NACK/DLQ).
   - Тест-кейсы в таблице: сценарий, ожидаемый результат, фактический результат.
6. Анализ результатов:
   - Оценка производительности (время обработки сообщений).

- Анализ обнаруженных проблем (гонки, таймауты, нестабильность).
- Объём покрытия сценариев тестами.

7. Ответы на контрольные вопросы.

8. Вывод.

7. Критерии оценивания

7.1. Базовый уровень — оценка «удовлетворительно»

Критерий

Разработаны Producer и Consumer с корректной логикой

Настроена топология RabbitMQ: Exchange, Queue, DLQ

Интеграционные тесты используют Testcontainers (RabbitMQ запускается в контейнере)

Написан тест на успешную отправку и обработку одного сообщения

Написан тест на обработку некорректного сообщения (попадание в DLQ)

Все тесты проходят стабильно (не менее 2 прогонов)

Отчёт оформлен по требованиям

Снижение оценки: Producer/Consumer не работают, нет Testcontainers (используется локальный RabbitMQ), нет теста DLQ, тесты нестабильны, отсутствует диаграмма потоков.

7.2. Средний уровень — оценка «хорошо»

Все критерии базового уровня плюс:

Критерий

Критерий

Реализована логика retry (не менее 3 попыток = MAX_RETRIES)

Реализован exponential backoff или delayed retry

Написан тест на успешный retry (сообщение обработано после N попыток)

Написан тест на исчерпание попыток и попадание в DLQ

Написан тест на идемпотентность (дубликат не обрабатывается повторно)

Протестирована конкурентная обработка (несколько Consumer'ов) или порядок сообщений

На все контрольные вопросы даны развёрнутые ответы

Снижение оценки: отсутствие retry-логики, отсутствие backoff, нет теста идемпотентности, нет конкурентного теста.

7.3. Повышенный уровень — оценка «отлично»

Все критерии базового и среднего уровней плюс:

Критерий

Testcontainers интегрированы в CI/CD-пайплайн (GitLab CI / GitHub Actions) с Docker-in-Docker

---

Реализован мониторинг очередей в тестах (проверка метрик: глубина, rate)

Написан тест на разрыв соединения / network disruption (через toxiproxy или аналоги)

Критерий

Проведено сравнение RabbitMQ и Kafka (через Testcontainers) или тестирование схем (Avro/Protobuf)

Реализованы Publisher Confirms (подтверждение доставки)

Проведён анализ производительности: throughput (сообщений/сек), latency (P50/P95)

Отчёт содержит секцию «Мониторинг» с метриками очереди

Снижение оценки: отсутствие CI/CD интеграции, нет network disruption теста, нет сравнения/схем, нет анализа производительности.

Лабораторная работа №8. Динамический анализ безопасности (DAST) и сравнение с SAST

1. Цель

Сформировать практические навыки применения динамического анализа безопасности (DAST) веб-приложений с использованием OWASP ZAP, а также провести сравнительный анализ эффективности подходов SAST и DAST для формирования целостной стратегии безопасной разработки (DevSecOps).

2. Задачи

1. Развернуть тестовое веб-приложение с преднамеренными уязвимостями, часть из которых обнаруживается статически, часть — динамически, а часть — обоими методами.
2. Изучить архитектуру и режимы работы OWASP ZAP (пассивное сканирование, активное сканирование, AJAX Spider).
3. Настроить и выполнить автоматизированное DAST-сканирование целевого приложения в режиме Automated Scan и через API (ZAP Python Client).

4. Проанализировать полученный отчёт DAST: классифицировать найденные уязвимости по категориям OWASP Top 10, severity и CWE.

5. Сопоставить результаты DAST с результатами SAST, полученными в Лабораторной работе №5 (для одного и того же приложения).

6. Выявить категории уязвимостей, обнаруженные только SAST, только DAST и обоими методами.

7. Сформулировать выводы о дополняющей роли SAST и DAST в стратегии DevSecOps и рекомендации по их совместному применению.

3. Теоретическое обоснование

DAST (Dynamic Application Security Testing) — метод анализа безопасности, при котором тестируется работающее приложение путём отправки специально сформированных запросов и анализа ответов. DAST относится к методологии «чёрного ящика» (Black Box Testing) и имитирует действия злоумышленника, атакующего приложение извне.

Сравнение SAST и DAST (развёрнутое):

Критерий	SAST (Статический)	DAST (Динамический)
Методология	Белый ящик (анализ исходного кода)	Чёрный ящик (анализ работающего приложения)
Когда применяется	На этапе разработки (IDE, commit, CI)	На этапе тестирования (staging, production)
Требуется доступ к коду	Да	Нет

Критерий	SAST (Статический)	DAST (Динамический)
Требуется запущенное приложение	Нет	Да
Обнаруживает	SQLi в коде, hardcoded secrets, eval(), unsafe crypto, уязвимости в библиотеках	XSS, CSRF, небезопасные заголовки, утечки данных, ошибки конфигурации сервера
Ложноположительные (FP)	Высокий (синтаксический анализ без контекста выполнения)	Ниже (проверяет реальное поведение)
Ложноотрицательные (FN)	Ниже для покрытых правил	Выше (зависит от глубины краулинга)
Покрытие кода	100% (все ветви)	Только достижимые URL (зависит от Spider)
Скорость	Быстро (секунды/минуты)	Медленно (минуты/часы)
Интеграция в	Легко (быстрый этап)	Сложнее

Критерий	SAST (Статический)	DAST (Динамический)
CI/CD		(требует развёртывания)

Типы уязвимостей, обнаруживаемых DAST (OWASP ZAP):

Категория OWASP	Примеры алертов ZAP
A01: Broken Access Control	Directory Browsing, Forced Browsing
A02: Cryptographic Failures	Weak SSL/TLS, Insecure Cookies
A03: Injection	SQL Injection, OS Command Injection, XPath Injection
A05: Security Misconfiguration	Server Leaks Information, Debug Enabled, Default Files
A06: Vulnerable Components	Outdated JS Libraries, Known CVEs
A07: Identification Failures	Weak Session Tokens, Missing Authentication
A09: Logging & Monitoring	Information Disclosure in Errors

Режимы работы ZAP для CI/CD:

1. Baseline Scan (пассивное, быстрое): только Spider + Passive Scanner. Безопасно, занимает минуты.
2. Full Scan (активное, полное): Spider + Passive + Active Scanner. Может повредить данные, занимает часы.
3. API Scan (для REST/GraphQL): сканирование на основе OpenAPI/Swagger-схемы.

4. Задания для индивидуального выполнения

4.1. Базовый уровень (15 вариантов)

Развернуть веб-приложение с уязвимостями, запустить DAST-сканирование OWASP ZAP (Baseline Scan), составить отчёт с алертами, выполнить базовое сравнение с результатами SAST из ЛР №5.

	Приложение	Фокус DAST-сканирования
	Flask-приложение из ЛР №5 (расширенное)	XSS, SQLi, Command Injection, Path Traversal
	Django-приложение с DEBUG=True	Information Disclosure, Admin panel, CSRF
	Node.js Express с EJS	SSTI, XSS, insecure headers
	Java Spring Boot REST API	Missing security headers, CORS, error handling
	PHP-приложение (простое)	SQLi, XSS, file inclusion
	Go-веб-сервер	Path traversal, verbose errors, no security headers
	Ruby on Rails (учебное)	Mass assignment, CSRF, insecure cookies

	Приложение	Фокус DAST-сканирования
	Статический сайт с формами	XSS, open redirect, HTML injection
	GraphQL API (Python)	Introspection, depth limiting, injection
0	Микросервис с JWT	Token leakage, weak signature, missing headers
1	WebSocket-приложение	No origin check, injection, CSWSH
2	Single Page Application (React)	Client-side XSS, insecure localStorage
3	CMS (учебная, упрощённая)	Default credentials, file upload, XSS
4	API Gateway (mock)	Rate limiting bypass, header injection
5	IoT Dashboard (mock)	Default passwords, verbose errors

4.2. Средний уровень (15 вариантов)

Дополнительно: полное активное сканирование (Full Scan), AJAX Spider для SPA, сравнение SAST и DAST с категоризацией по OWASP Top 10, автоматизация через ZAP Python API, отчёт в нескольких форматах.

	Приложение	Дополнительные требования

	Приложение	Дополнительные требования
	E-commerce (учебный)	Full Scan + AJAX Spider; сравнение с SAST по OWASP Top 10
	Банковское приложение (mock)	Активное сканирование всех форм; Python API автоматизация
	Блог-платформа	Full Scan; сравнение false positives SAST vs DAST
	Социальная сеть (mock)	AJAX Spider для динамических страниц; HTML + XML отчёты
	Файловое хранилище (веб)	Path traversal, file upload; полная автоматизация
	Админ-панель	Full Scan с аутентификацией (ZAP Context); Python API
	Платёжный шлюз (mock)	Активное сканирование API; сравнение CWE coverage
	Healthcare-портал	HIPAA-релевантные уязвимости; детальный отчёт
	E-learning платформа	XSS в комментариях; Spider + Active Scan
0	Форумы / доска объявлений	Все формы; сравнение SAST-vs-DAST матрица
	Микросервисная	Сканирование 2+

	Приложение	Дополнительные требования
1	архитектура	микросервисов; агрегированный отчёт
2	Progressive Web App	Service Worker, offline; AJAX Spider
3	Booking-система	Календарь, формы бронирования; активное сканирование
4	Real-time Dashboard (WebSocket)	WebSocket-уязвимости; ZAP WebSocket add-on
5	Multi-tenant приложение	Изоляция tenant'ов; сравнение SAST/DAST находок

4.3. Повышенный уровень (15 вариантов)

Дополнительно: интеграция DAST в CI/CD (GitLab CI/GitHub Actions) с Quality Gate на основе алертов, настройка ZAP Context с аутентификацией для аутентифицированного сканирования, кастомные scan policies, сравнение ZAP с дополнительным инструментом (Nikto/Wapiti/Arachni), анализ эволюции уязвимостей во времени.

	Тема	Расширенные требования
	Complete DevSecOps Pipeline	CI/CD: SAST → DAST → Quality Gate → Deploy
	Authenticated DAST Scanning	ZAP Context с login-формой; сканирование залогиненной сессии

	Тема	Расширенные требования
	ZAP vs Nikto vs Wapiti	Сравнительное исследование 3 DAST-инструментов
	Custom Scan Policies	Policy: только инъекции + XSS; сравнение с полным сканом
	DAST Quality Gate в CI/CD	GitLab CI + падение пайплайна при High алертах
	OpenAPI-driven DAST	Сканирование на основе Swagger/OpenAPI-схемы
	Time-based Vulnerability Tracking	History: сканирование weekly, тренды, regression
	DAST + WAF Testing	Проверка, блокирует ли WAF обнаруженные атаки
	Container Security + DAST	Trivy (контейнер) + ZAP (приложение) — интеграция
0	ZAP Automation Framework	YAML-based план: spider → passive → active → report
1	DAST для мобильного backend	Сканирование REST API через прокси; JWT-сессии
2	Compliance Reporting (PCI-DSS)	DAST-отчёт в контексте PCI-DSS требований
	False Positive / False	Ручная верификация всех

	Тема	Расширенные требования
3	Negative анализ	алертов; расчёт точности
4	ZAP + Selenium/Playwright	Комбинированное сканирование: E2E тесты как seed для Spider
5	Complete DevSecOps Assessment	SAST + DAST + SCA + IaC scanning → Unified Security Dashboard

5. Контрольные вопросы

1. В чём принципиальная разница между SAST и DAST с точки зрения методологии (белый ящик vs чёрный ящик)?
2. Какие категории уязвимостей OWASP Top 10 лучше обнаруживаются SAST, какие — DAST, а какие — только при совместном использовании?
3. Как работает OWASP ZAP: режимы пассивного и активного сканирования, Spider, AJAX Spider?
4. Почему DAST не может обнаружить hardcoded credentials в исходном коде, но может обнаружить их использование через заголовки HTTP?
5. Что такое ложноположительные (False Positive) и ложноотрицательные (False Negative) срабатывания? Как соотносятся FP/FN rates у SAST и DAST?
6. Как организовать аутентифицированное DAST-сканирование (ZAP Context, login sequence)?
7. Какие форматы отчётов поддерживает OWASP ZAP и какой формат предпочтителен для интеграции в CI/CD?

8. Как интегрировать DAST в CI/CD-пайплайн с Quality Gate на основе severity алертов?

9. В чём преимущество использования ZAP Automation Framework (YAML plans) перед Python API?

10. Какова оптимальная стратегия внедрения SAST и DAST в DevSecOps с учётом скорости, полноты покрытия и стоимости исправления уязвимостей?

6. Требования к отчёту

1. Титульный лист: ФИО, группа, вариант, номер и тема работы.
2. Цель и задачи работы.
3. Теоретическое обоснование: DAST, сравнение с SAST, архитектура OWASP ZAP, принцип комплементарности (1–2 стр.).
4. Практическая часть:
   - Листинг тестового веб-приложения с уязвимостями (с комментариями: какие уязвимости для какого метода).
   - Скрипт автоматизированного DAST-сканирования (Python API или YAML Automation Framework).
   - Скрипт сравнительного анализа SAST vs DAST.
   - Конфигурация CI/CD (если применимо).
5. Результаты выполнения:
   - Скриншот вывода ZAP в процессе сканирования (Spider %, Active Scan %).
   - Фрагмент HTML/JSON-отчёта ZAP с примерами алертов.
   - Сравнительная таблица SAST vs DAST (все находки с пометками).
   - Диаграмма Венна: SAST-only / Both / DAST-only.
   - График покрытия категорий OWASP Top 10 методами SAST и DAST.
6. Сравнительный анализ:
   - Категоризация находок: SAST-only, DAST-only, Both.

- Объяснение причин, почему определённые уязвимости обнаружены только одним методом.
- Оценка False Positive/False Negative для каждого метода.

7. Рекомендации по совместному применению SAST и DAST в DevSecOps.

8. Ответы на контрольные вопросы.

9. Вывод: подтверждение гипотезы о комплементарности SAST и DAST.

7. Критерии оценивания

7.1. Базовый уровень — оценка «удовлетворительно»

Критерий
Разработано/развёрнуто веб-приложение с уязвимостями (не менее 5 категорий)
Запущено DAST-сканирование OWASP ZAP (Baseline Scan — пассивное)
Получен отчёт ZAP (HTML или JSON) с алертами
Алерты классифицированы по severity и типам
Выполнено базовое сравнение с результатами SAST из ЛР №5 (таблица)
Сформулированы выводы о различиях в находках SAST и DAST
Отчёт оформлен по требованиям

Снижение оценки: менее 5 категорий уязвимостей, только пассивное сканирование без алертов, отсутствие сравнения с SAST, выводы формальны.

7.2. Средний уровень — оценка «хорошо»

Все критерии базового уровня плюс:

Критерий

Выполнено активное сканирование (Full Scan) с атакующими payload'ами

Использован AJAX Spider (если приложение SPA) или обосновано его отсутствие

Сканирование автоматизировано через ZAP Python API (скрипт приложен)

Проведена категоризация находок по OWASP Top 10

Построена диаграмма Венна (SAST / DAST / Both)

Сгенерированы отчёты минимум в 2 форматах (HTML + JSON)

Объяснены причины, почему определённые уязвимости найдены только одним методом

На все контрольные вопросы даны развёрнутые ответы

Снижение оценки: нет активного сканирования, нет автоматизации через API, нет категоризации OWASP, нет диаграммы Венна.

7.3. Повышенный уровень — оценка «отлично»

Все критерии базового и среднего уровней плюс:

Критерий

DAST интегрирован в CI/CD-пайплайн с Quality Gate (блокировка при High алертах)

Настроен ZAP Context с аутентификацией (сканирование залогиненной сессии)

Критерий

Проведено сравнение ZAP с дополнительным DAST-инструментом (Nikto, Wapiti, Arachni)

Проведён анализ False Positive/False Negative (ручная верификация 5+ алертов)

Построен график эволюции уязвимостей (если история сканирований) или метрики полноты

Использован ZAP Automation Framework (YAML) или OpenAPI-driven сканирование

Отчёт содержит раздел «Метрики и KPI безопасности»

Сформулированы конкретные рекомендации по внедрению в DevSecOps

Снижение оценки: отсутствие CI/CD интеграции, нет аутентификации, нет сравнения инструментов, нет анализа FP/FN, рекомендации размыты.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

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

Дисциплина «Автоматизация тестирования и качество ПО» является **дисциплиной по выбору** вариативной части образовательной программы (Б1.В.ДВ.01.02) и направлена на формирование у магистрантов системных знаний и практических навыков в области стратегий, методов и инструментов автоматизированного тестирования для обеспечения качества сложных программных и программно-аппаратных (инженерных) систем.

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 «Автоматизация тестирования и качество ПО»;
- 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 проектирования, реализации и интеграции различных видов автоматизированного тестирования в CI/CD-пайплайн в рамках выполнения индивидуального сквозного проекта.

Задачи самостоятельной работы:

1. Изучить теоретический материал по темам дисциплины в соответствии с учебным планом.
2. Подготовиться к выполнению лабораторных работ (№1-8) и осмыслению их результатов.
3. Разработать и реализовать сквозной проект по созданию комплексного пайплайна автоматизированного тестирования (Лабораторная работа №8 дисциплины).
4. Сформировать комплексный отчёт, включающий все виды тестирования, анализ метрик качества и настройку Quality Gates.
5. Подготовиться к промежуточной аттестации (зачёту с оценкой) по дисциплине.

6. Развить навыки самостоятельного поиска и анализа научно-технической информации в области автоматизации тестирования ПО.

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 инструментов и методов автоматизации тестирования (модульного, API, UI, нагрузочного, безопасности) и их интеграции в CI/CD-пайплайн с качественными воротами (Quality Gates) в рамках выполнения индивидуального сквозного проекта.

1.2. Формируемые компетенции и индикаторы

В результате выполнения самостоятельной работы и защиты сквозного проекта студент должен обладать следующими компетенциями и индикаторами (в соответствии с ОПОП):

Компетенция	Индикаторы
ПК-1. Способен проектировать, реализовывать, оптимизировать и поддерживать сквозные конвейеры непрерывной интеграции, поставки и развертывания (CI/CD) для программного	**ИД-1** $_{\text{ПК-1}}$ Проектирует архитектуру CI/CD-пайплайнов, определяя этапы сборки, тестирования и развертывания. **ИД-2** $_{\text{ПК-1}}$ Реализует автоматизированные процессы непрерывной интеграции, включая компиляцию кода, запуск модульных и интеграционных тестов, статический анализ.

Компетенция	Индикаторы
обеспечения и встроенных систем, интегрируя практики автоматизированного тестирования, проверки безопасности, управления артефактами и конфигурациями.	**ИД-3** $_{\text{ПК-1}}$ Интегрирует инструменты безопасности (SAST, DAST, SCA) в конвейеры CI/CD для автоматического сканирования уязвимостей на ранних этапах разработки. **ИД-7 (ПК-1).** Оценивает эффективность работы CI/CD-конвейеров и оптимизирует их для сокращения времени цикла разработки и повышения надежности развертываний.

1.3. Трудоёмкость самостоятельной работы

Общая трудоёмкость самостоятельной работы по дисциплине составляет **44 часа**.

Вид работы	**Часы**
Изучение теоретического материала	10
Подготовка к лабораторным работам (№1-7)	14
Выполнение сквозного проекта (Лабораторная работа №8)	12
Подготовка к зачёту с оценкой	8
Итого	**44**

1.4. Общая характеристика сквозного проекта

Сквозной проект выполняется студентом **индивидуально** в течение всего семестра параллельно с изучением дисциплины и выполнением лабораторных работ (№1-7). Защита проекта проводится на 8-м лабораторном занятии.

Тема проекта: «Разработка и внедрение комплексного пайплайна автоматизированного тестирования для обеспечения качества программной системы».

Результат проекта: Комплексный отчёт и работающий CI/CD-пайплайн, включающий:

- Набор модульных, интеграционных, UI и нагрузочных тестов.
- Инструменты статического (SAST) и динамического (DAST) анализа безопасности.
- Настройку Quality Gates (покрытие кода, уязвимости, производительность).
- Отчёты о тестировании и аналитику.
- Оценку эффективности и рекомендации по улучшению качества.

2. МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ИЗУЧЕНИЮ ТЕОРЕТИЧЕСКОГО МАТЕРИАЛА

2.1. Темы для самостоятельного изучения

	Тема	Литература	Часы
	Основы автоматизации тестирования. Пирамида тестирования.	[1, гл. 1-2]	1
	Модульное тестирование и анализ покрытия	[2,	1

	Тема	Литература	Часы
	кода (pytest, coverage.py).	гл. 4-5]	
	Интеграционное тестирование REST API (pytest, Postman, Newman).	[1, гл. 6]	1
	Автоматизация тестирования UI. Паттерн Page Object Model.	[3, гл. 3-4]	1
	Нагрузочное тестирование (k6). Метрики производительности.	[4]	1
	Статический анализ безопасности (SAST): Semgrep, Bandit.	[5]	1
	Динамический анализ безопасности (DAST): OWASP ZAP.	[6]	1
	Качественные ворота (Quality Gates) в CI/CD. Артефакты тестирования.	[7]	1
	Тестирование асинхронных систем. Testcontainers.	[8]	1
0	Сравнение SAST и DAST. Интеграция в DevSecOps.	[5, 6]	1

2.2. Рекомендуемая литература

Основная литература:

1. Криспин, Л. Гибкое тестирование: практическое руководство для тестировщиков ПО и гибких команд / Л. Криспин, Д. Грегори. — М.: Вильямс, 2021. — 592 с.

2. Окран, М. Pytest: руководство по эффективному тестированию на Python / М. Окран. — М.: ДМК-Пресс, 2022. — 368 с.

Дополнительная литература:

3. Сундарам, А. Playwright для автоматизации тестирования / А. Сундарам. — М.: ДМК-Пресс, 2023. — 284 с.

4. k6 Documentation (русскоязычная версия). — URL: https://k6.io/docs/ (дата обращения: 01.05.2026).

5. Semgrep Documentation. — URL: https://semgrep.dev/docs (дата обращения: 01.05.2026).

6. OWASP ZAP Documentation. — URL: https://www.zaproxy.org/docs/ (дата обращения: 01.05.2026).

7. GitLab CI/CD Documentation. — URL: https://docs.gitlab.com/ee/ci/ (дата обращения: 01.05.2026).

8. Testcontainers Documentation. — URL: https://testcontainers.com/ (дата обращения: 01.05.2026).

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 (локально или в Docker).
- Использовать официальную документацию как основной источник для выполнения лабораторных работ.

3. МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ

3.1. Технологический стек

Для выполнения лабораторных работ и сквозного проекта рекомендуется использовать следующий стек инструментов (все инструменты являются открытыми и доступны на территории РФ):

- **CI/CD:** GitLab CI / GitHub Actions.
- **Тестирование:** pytest, pytest-cov, Postman, Newman, Playwright, k6.
- **Безопасность:** Semgrep, Bandit, OWASP ZAP.
- **Контейнеризация и тестовое окружение:** Docker, Testcontainers.
- **Визуализация:** Allure Framework, pytest-html.
- **Язык программирования:** Python 3.9+.

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 (тесты, скрипты, конфигурации).
4. Выполнить проверку в тестовой среде (локально или в CI).
5. Оформить отчёт по установленной форме.
6. Защитить работу перед преподавателем.
7. Интегрировать результаты лабораторной работы в сквозной проект.

3.3. Таблица соответствия лабораторных работ и этапов сквозного проекта

Сквозной проект выполняется на протяжении всего семестра. Каждая лабораторная работа добавляет новый компонент в проект.

№ ЛР	Тема лабораторной работы	Этап сквозного проекта	Результат этапа
	Модульное тестирование и анализ покрытия кода	Фундамент качества	Набор unit-тестов, отчёт о покрытии (≥80%), настройка порога
	Интеграционное тестирование REST API	API-тестирование	Тесты API через pytest + requests, коллекция Postman, запуск Newman
	Автоматизация тестирования UI с Page Object Model	UI-автоматизация	Page Object Model, E2E-сценарии, проверка состояний
	Нагрузочное тестирование с k6	Производительность	Сценарии k6 (smoke, stress, spike), отчёты, точка насыщения
	Статический анализ безопасности кода (SAST)	Безопасность (ранняя)	Отчёты Semgrep/Bandit, классификация уязвимостей, исправления
	Настройка качественных ворот	Оркестрация	Единый CI/CD-пайплайн со всеми

№ ЛР	Тема лабораторной работы	Этап сквозного проекта	Результат этапа
	в CI/CD		видами тестов и Quality Gates
	Тестирование асинхронных систем и очередей	Асинхронность	Тесты Producer/Consumer + DLQ через Testcontainers
	Динамический анализ безопасности (DAST) и сравнение с SAST	Безопасность (поздняя)	DAST-отчёт OWASP ZAP, сравнительный анализ SAST/DAST

4. ВОПРОСЫ ДЛЯ СОБЕСЕДОВАНИЯ ПО САМОСТОЯТЕЛЬНОМУ ИЗУЧЕНИЮ ЛИТЕРАТУРЫ

1. Что такое модульное тестирование? В чём его отличие от интеграционного?
2. Какие виды покрытия кода существуют? Почему 100% покрытие не гарантирует отсутствие ошибок?
3. Каковы основные принципы REST API? Какие коды состояния HTTP важны при тестировании?
4. Что такое паттерн Page Object Model и какие проблемы он решает?
5. Какие виды нагрузочного тестирования существуют? Что такое точка насыщения?

6. Чем отличается SAST от DAST? Какие уязвимости лучше обнаруживает каждый метод?

7. Что такое Quality Gates в CI/CD? Какие метрики можно использовать в качестве ворот?

8. Какие форматы отчётов о тестировании поддерживаются в CI/CD (JUnit, Cobertura, SARIF)?

9. Что такое Dead Letter Queue (DLQ) и для чего она используется при тестировании очередей?

10. Какие стратегии повторной обработки (retry) существуют при асинхронном взаимодействии?

11. Как работает Testcontainers и в чём его преимущество перед моками?

12. Как настроить автоматическое падение пайплайна при недостижении порога покрытия?

13. Какие инструменты статического анализа безопасности подходят для Python?

14. Как выполнить аутентифицированное DAST-сканирование в OWASP ZAP?

15. Каковы оптимальные пороговые значения для покрытия кода, времени ответа и количества уязвимостей?

5. КРИТЕРИИ ОЦЕНИВАНИЯ КОМПЕТЕНЦИЙ

5.1. Критерии оценивания лабораторных работ

Оценка	Критерии
5 (отлично)	Работа выполнена в полном объёме, тесты/скрипты работают без ошибок, студент демонстрирует глубокое понимание всех этапов,

Оценка	Критерии
	отвечает на дополнительные вопросы, отчёт оформлен по требованиям (для повышенного уровня — также CI/CD-интеграция).
4 (хорошо)	Работа выполнена полностью, но есть небольшие замечания (неоптимальная конфигурация, неполный отчёт, не все дополнительные требования). 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 автоматизации тестирования, не может объяснить свои действия.

5.2. Критерии оценивания сквозного проекта (ЛР №8)

Критерий	**Вес, %**	**5 отлично**	**4 хорошо**	**3 удовлетворительно**	**2 неудовлетворительно**
Полнота реализац	20	Реализовано 6+ видов	Реализовано	Реализовано 3 вида	Реализовано менее 3 видов

Критерий	Вес, %	5 отлично	4 хорошо	3 удовлетворительно	2 неудовлетворительно
ии видов тестирования		тестирования	4-5 видов		
Качество конфигурации CI/CD (Quality Gates)	20	Quality Gates настроены для всех этапов, пайплайн оптимален	Quality Gates для 3+ этапов	Quality Gates для 1-2 этапов	Quality Gates отсутствуют
Покрытие кода и успешность тестов	15	Покрытие ≥ 80%, все тесты проходят	Покрытие 70-80%, тесты в основном проходят	Покрытие 60-70%, есть падающие тесты	Покрытие < 60%, тесты отсутствуют
SAST + DAST	15	Оба метода применены, уязвимости исправлены,	Оба метода применены, не	Применён только один метод	Безопасность не тестировалась

Критерий	**Вес, %**	**5 отлично**	**4 хорошо**	**3 удовлетворительно**	**2 неудовлетворительно**
		Quality Gate по безопасности настроен	все уязвимости исправлены		
Нагрузочное тестирование	10	Проведены 3+ типа тестов, определена точка насыщения	Проведены 2 типа тестов	Проведён 1 тип тестов	Тесты не проводились
Автоматизация тестирования UI	10	Реализован POM, 3+ E2E-сценария	Реализован POM, 1-2 E2E-сценария	UI-тесты без POM	UI-тесты отсутствуют
Качество отчёта и презентации	10	Отчёт полный, структурированный,	Отчёт с небольшими недочё	Отчёт минимальный, презентация	Отчёт отсутствует

Критерий	**Вес, %**	**5 отлично**	**4 хорошо**	**3 удовлетворительно**	**2 неудовлетворительно**
		скриншоты есть, презентация готова, уверенные ответы	тами	отсутствует	

Шкала перевода итогового балла в оценку:

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