МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к практическим занятиям (20 работ) по дисциплине

«Архитектура и инструментальные средства информационных систем»

Сквозной кейс: SmartCampus (варианты 01–30)

1. Как устроен практикум и зачем нужна тетрадь

Практические занятия в этом курсе построены как последовательная сборка архитектурного пакета для одной информационной системы. Вместо разрозненных отчётов студент заполняет рабочую тетрадь от руки: рисует диаграммы, заполняет таблицы и фиксирует решения. Цель такого формата — не «оформление», а закрепление навыков: пока студент пишет и рисует, он проговаривает понятия и связывает их с кейсом.

В конце семестра у студента получается комплект артефактов: архитектурный бриф, сценарии качества, диаграммы C4/UML, модель данных, контракты API, интеграционные схемы, пакет решений ADR, материалы по безопасности и наблюдаемости, а также основы деплоя и эксплуатации.

1.1. Что сдаётся на каждом занятии

- Заполненные поля тетради: схемы/таблицы/краткие пояснения (аккуратно и читаемо).
- По требованию преподавателя — фото/скан страниц в электронный кабинет (для резервного хранения и проверки).
- Если практику не успели полностью — фиксируется «черновой вариант» и план завершения (что осталось).

1.2. Как оценивать (рекомендация)

Рекомендуемая шкала — 10 баллов за практику. Оценивание лучше делать по рубрике, чтобы уменьшить субъективность.

Критерий	Что проверяем	Баллы
Полнота	Все требуемые артефакты/части выполнены, ничего критичного не пропущено.	0–4
Корректность	Артефакт соответствует правилам (C4/UML/OpenAPI), нет грубых ошибок и противоречий.	0–3
Обоснование	Есть логика «почему так», учтены драйверы/качества/сбои, видны компромиссы.	0–2
Оформление	Читаемость, подписи, легенда, аккуратность, версия/дата.	0–1

1.3. Варианты 01–30

У разных студентов различаются входные параметры кейса: масштаб организации, нагрузка, требования к доступности, окна обслуживания, степень нестабильности интеграций, список рисков и т.п. Задания при этом одинаковые по структуре — меняются только числа и детали.

Важно: «правильных» архитектурных решений обычно несколько. Оценивается не выбор конкретного инструмента, а аргументация: связь решения с драйверами, учёт отказов, понятность артефактов и проверяемость требований.

2. Сквозной кейс SmartCampus

SmartCampus — это единый портал и набор сервисов, которые помогают студентам и сотрудникам получать учебные и административные услуги. Система агрегирует данные из внешних ИС (SSO, LMS, бухгалтерия, библиотека) и управляет собственными процессами (например, заявлениями).

2.1. Роли и пользователи

- Студент: смотрит расписание, подаёт заявления, отслеживает статусы, получает уведомления.
- Преподаватель: видит расписание и ведомости, публикует объявления, подтверждает отдельные действия (по политике вуза).
- Деканат/администрация: рассматривает заявления, принимает решения, формирует документы, ведёт аудит.
- Охрана/служба пропусков: выпускает пропуска и проверяет права доступа.
- Бухгалтерия: источник финансовой информации и статусов оплат (внешняя система).
- Администратор ИС: управляет ролями, справочниками, интеграциями и доступами.

2.2. Основные функциональные модули

- Расписание: быстрый просмотр по группе/преподавателю/аудитории; пик нагрузки при публикации.
- Заявления: подача, маршрутизация, решение, статусы, уведомления, история изменений (аудит).
- Успеваемость/ведомости: просмотр оценок и результатов (источник данных — LMS/внутренние системы).
- Пропуска: выпуск и управление, связь с правами и статусами (например, задолженность).
- Уведомления: email/SMS/push; гарантированная доставка и повторные попытки.

2.3. Ландшафт интеграций

В учебном кейсе предполагаются типовые внешние системы. Варианты могут отличаться протоколами и надёжностью интеграций.

- SSO (например, Keycloak): аутентификация, выдача токенов, группы/роли.
- LMS (например, Moodle): расписание/курсы/оценки/учебные данные.
- 1С (финансы): оплата, задолженность, счета (часто нестабильна и имеет окна обслуживания).
- Библиотека: сведения о книгах и задолженностях (может быть SOAP/REST).
- Шлюзы уведомлений: email, SMS, push.

2.4. Данные и требования к безопасности

Система работает с персональными данными и учебными результатами, поэтому безопасность рассматривается как архитектурный драйвер.

- Минимизация данных: хранить только то, что нужно для процессов портала.
- Принцип наименьших привилегий: роли и права должны быть ограничены, а изменения ролей — аудитироваться.
- Защита от массовых запросов: rate limiting, защита входа, контроль сессий и токенов.
- Аудит: важные действия (создание/изменение/решение по заявлению, выдача пропуска, смена роли) должны фиксироваться.

2.5. Нефункциональные требования (качества)

- Производительность: расписание открывается быстро даже в пике публикации.
- Доступность: портал должен быть доступен большую часть времени; допускаются окна обслуживания.
- Надёжность интеграций: сбои внешних систем не должны «ронять» портал целиком.
- Сопровождаемость: систему нужно менять в течение семестра, добавляя функции и интеграции без хаоса.
- Наблюдаемость: деградация должна обнаруживаться по метрикам и логам, а расследование — по трассам.

3. Карта артефактов: что появляется на каждой практике

Ниже показано, как практики связаны между собой. Если студент честно выполняет работы по порядку, то к концу семестра у него автоматически собирается полный архитектурный пакет.

Практика	Основной артефакт	Куда используется дальше
01	Architecture Brief (границы, стейкхолдеры, цели, ограничения)	Основа архитектурной модели
02	Сценарии качества (QAS) + тактики + компромиссы	Основа архитектурной модели
03	C4 L1 Context (границы, акторы, внешние системы)	Основа архитектурной модели
04	C4 L2 Container (контейнеры, БД, кэш, брокер, каналы)	Основа архитектурной модели
05	C4 L3 Component + модульные границы	Основа архитектурной модели
06	UML Sequence (happy path + отказ/таймаут)	Контракты/интеграции/данные
07	Модель данных: ERD + стратегия миграций + аудит	Контракты/интеграции/данные
08	OpenAPI: контракт + модель ошибок	Контракты/интеграции/данные
09	Интеграции: sync vs async, события + схемы сообщений	Контракты/интеграции/данные
10	Threat model (DFD + trust boundaries + STRIDE) + security checklist	Эксплуатация и решения
11	Observability: метрики/логи/трейсы + SLI/SLO + pipeline	Эксплуатация и решения
12	ADR pack: решения, альтернативы, последствия, проверки	Эксплуатация и решения
13	Docs-as-code: структура репозитория + CI проверки	Подготовка к деплою/финалу
14	Контейнеризация: Dockerfile + compose для	Подготовка к деплою/финалу

	локальной разработки	
15	Kubernetes: Deployment/Service/Ingress + ресурсы	Подготовка к деплою/финалу
16	Performance: план нагрузочного теста + узкие места	Подготовка к деплою/финалу
17	Reliability: SLO, error budget, процесс инцидентов	Подготовка к деплою/финалу
18	План рефакторинга/модернизации: от прототипа к поддерживаемости	Подготовка к деплою/финалу
19	Architecture review: чеклист + реестр рисков	Подготовка к деплою/финалу
20	Итоговый пакет + защита (story архитектуры)	Подготовка к деплою/финалу

3.1. Универсальные правила оформления

- На каждой схеме указывайте название, дату и версию (хотя бы v0.1, v0.2).
- Стрелки подписывайте глаголами и каналом (HTTP, event, SOAP, gRPC).
- Не перегружайте: если на листе больше 15 элементов — сделайте две диаграммы.
- Границы ответственности и доверия обозначайте рамкой (system boundary / trust boundary).
- Любое важное решение фиксируйте: кратко «почему так» и «что было бы иначе».

4. Теоретический минимум и шпаргалки

4.1. Архитектура как набор значимых решений

Архитектурой называют те решения, которые сложно и дорого менять: границы модулей, способ интеграции, модель данных, модель безопасности, подход к деплою и наблюдаемости. Архитектор (или команда) не угадывает «идеальную» схему, а сначала фиксирует драйверы (цели, ограничения, риски), затем выбирает решения и обязательно проверяет их гипотезами (прототипами, тестами).

- Если решение не влияет на риски и не связано с драйверами — это обычно не архитектура, а деталь реализации.
- У каждого решения должны быть альтернативы и последствия (что стало лучше, что стало хуже).
- Хорошая архитектура объяснима: её можно защитить на языке целей, качеств и измеримых требований.

4.2. Драйверы и Architecture Brief

Architecture Brief — короткий документ на 1–2 страницы, который помогает договориться о главном. Он особенно полезен, когда в проекте много стейкхолдеров и интеграций.

Граница системы	Что входит в SmartCampus, что остаётся во внешних системах.
Стейкхолдеры и concerns	Кто заинтересован и какие вопросы задаёт (скорость, безопасность, эксплуатация).
Бизнес-цели	2–3 цели + метрики успеха.
Ключевые сценарии	3–5 сценариев использования (расписание, заявление, уведомление и т.п.).
Качества	4–6 атрибутов качества, из них 2 измеримых (SLO/производительность).
Ограничения	Сроки, команда, стек, деплой, регуляторика.
Риски и проверки	Что неизвестно и как проверяем (прототип, нагрузочный тест, интервью).

4.3. Сценарии качества (QAS) и тактики

Нефункциональные требования важно формулировать измеримо. Сценарии качества переводят слова «быстро», «надёжно», «безопасно» в тестируемый формат: что произошло, в каких условиях, как система реагирует и какой показатель считается успехом.

Шаблон QAS:

- Источник → Стимул → Среда → Артефакт → Отклик → Метрика.

Пример (для расписания):

- Источник: студент; стимул: запрос /schedule; среда: пик публикации; артефакт: API расписания; отклик: ответ; метрика: p95 < 300 ms при 600 RPS.

Ниже — примеры тактик, которые часто используются в архитектуре порталов:

Качество	Типовые тактики
Performance	кэширование (Redis/CDN), индексы и оптимизация запросов, batching, асинхронность, уменьшение payload
Availability	timeouts/retries, circuit breaker, очереди и outbox, деградация функционала, репликация
Security	RBAC/ABAC, MFA для админов, аудит, rate limiting, валидация входа, секреты в vault
Modifiability	модульные границы, контракты, ADR, изоляция интеграций адаптерами, тесты архитектуры
Observability	корреляция traceId, структурные логи, RED/USE метрики, алерты по SLO, трассировка end-to-end

4.4. Диаграммы C4 и UML: как делать читабельно

В практикуме используется C4 как основной язык архитектурных представлений. Главное правило C4 — «зум»: каждый уровень отвечает на свой набор вопросов и не смешивает лишние детали.

- C4 Level 1 (Context): акторы и внешние системы, граница ответственности. Без внутренних БД и сервисов.
- C4 Level 2 (Container): приложения/сервисы/БД/кэш/брокер, технологии, синхронные и асинхронные связи.
- C4 Level 3 (Component): компоненты внутри одного контейнера; правила модульных границ и зависимости.

Чеклист для любой диаграммы:

- Есть заголовок, версия и дата.
- Есть легенда (например, разные линии для sync/async).
- Стрелки подписаны (глагол + канал).
- Есть рамка system boundary; для безопасности — trust boundary.
- Диаграмма читабельна: 8–15 элементов.

4.5. API и OpenAPI: контракт важнее реализации

Для архитектуры важно договориться о контрактах: какие операции есть, какие статусы и ошибки возможны, какие правила идемпотентности и версионирования применяются. OpenAPI удобно использовать как единый источник правды.

- Ошибки должны быть стандартизированы: один формат ошибки на все эндпоинты (код, сообщение, детали, traceId).
- Для POST/PUT продумайте идемпотентность (например, Idempotency-Key).
- Версионирование: либо /api/v1, либо через заголовки; главное — стабильность и политика изменений.

Пример модели ошибки (json):

```
{
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Invalid request",
    "details": [{"field":"groupId","issue":"required"}],
    "traceId": "00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-
01"
```

```
  }
}
```

4.6. Интеграции: синхронно и асинхронно

Синхронные интеграции (HTTP/REST/SOAP) проще, но делают систему зависимой от доступности внешней ИС. Асинхронные интеграции (события/очереди) увеличивают устойчивость, но требуют продуманной идемпотентности, схем сообщений и обработки ошибок.

- Если внешняя система нестабильна — лучше ставить очередь и работать со статусами (pending/processing).
- Для надёжной публикации событий используйте шаблон outbox (события пишутся в БД вместе с бизнес-операцией).
- События версионируются: type + version; изменения — через добавление полей, а не ломание схемы.

4.7. Security by Design: STRIDE в 10 минут

Модель угроз начинается с потоков данных (DFD) и границ доверия. Затем по каждому элементу и потоку проверяются категории STRIDE.

STRIDE	Смысл	Примеры мер
S	Spoofing (подмена личности)	MFA, защита сессий, подпись токенов, проверка audience/issuer
T	Tampering (подмена данных)	TLS, подпись сообщений, контроль целостности, запрет прямых правок БД
R	Repudiation (отказ от действий)	аудит и журналы, неизменяемые записи, time sync
I	Information disclosure (утечка)	минимизация данных, шифрование, маскирование, контроль доступа
D	Denial of service (отказ)	rate limiting, WAF, очереди, деградация, защита от тяжёлых запросов
E	Elevation of privilege (повышение прав)	RBAC/ABAC, least privilege, ревью ролей, разделение админ-доступов

4.8. Observability и SLO

Наблюдаемость — это способность понять, что происходит в системе, по её внешним сигналам: метрикам, логам и трассам. Для архитектуры важно заранее договориться, какие SLI/SLO мы считаем ключевыми и какие алерты поднимаем.

- Метрики сервиса: подход RED (Rate, Errors, Duration). Метрики ресурсов: USE (Utilization, Saturation, Errors).
- Трассировка: один traceId проходит через все сервисы; это ускоряет расследование инцидентов.
- SLO лучше задавать по успешным запросам и по p95/p99, а не по средним значениям.

Практическая работа 01. Architecture Brief: Stakeholders, Goals, Constraints

Фокус: Зафиксировать границу системы и архитектурные драйверы, чтобы дальше проектировать осмысленно

Ожидаемый результат

- Architecture Brief на 1–2 страницы (в тетради — структурировано по пунктам).
- Таблица stakeholders & concerns (не менее 8 стейкхолдеров).
- Короткий список рисков и как их проверить (прототип/тест/интервью).

Теория и пояснения

Архитектурный бриф нужен, чтобы в проекте не спорили «на вкус», а опирались на зафиксированные драйверы. Бриф отвечает на вопросы: что строим, для кого, зачем, при каких ограничениях и какие риски считаем самыми опасными. На практике бриф используется как точка опоры: если вы меняете архитектурное решение, вы должны показать, какой драйвер от этого выигрывает и какой риск уменьшается.

Порядок выполнения (шаги)

1. Опишите границу системы: 1–2 предложения «SmartCampus делает …, но не делает …». Отдельно выпишите внешние системы (SSO/LMS/1C/библиотека и т.п.).
2. Составьте список стейкхолдеров. Для каждого выпишите 2–3 concerns (вопроса/опасения): скорость, доступность, безопасность, аудит, удобство, сопровождение.
3. Сформулируйте 3 бизнес-цели и к каждой — метрику успеха (как измерим результат). Пример метрики: «доля заявлений, обработанных без ручных уточнений» или «время ответа расписания p95».
4. Выпишите ограничения и допущения (минимум 6): сроки, команда, стек, деплой, «нельзя в облако», окна обслуживания, нестабильность интеграций.
5. Соберите 3–5 ключевых сценариев (use cases), которые важнее всего для архитектуры. Не путайте «функции» и «сценарии»: сценарий имеет акторов, шаги и условия.
6. Составьте список рисков/неизвестностей (минимум 3) и рядом — как вы их проверите. Пример: «потянет ли БД пик расписания» → «нагрузочный тест на прототипе запроса».
7. Сведите всё в бриф: аккуратно, коротко, без лишнего. Укажите дату и версию (v0.1).

Пример или шаблон

```
Пример формулировки границы:
```

«SmartCampus — это портал и backend, который управляет процессом заявлений и агрегирует данные для пользователей. Аутентификация и первичные данные об обучении остаются в SSO/LMS; финансовые данные — в 1С. SmartCampus не является системой-источником для оценок и финансов, но хранит статусы и историю заявлений».

Типовые ошибки

- Слишком много деталей реализации (таблицы БД, классы, конкретные библиотеки) вместо смысла и драйверов.
- Цели без метрик («повысить качество», «улучшить скорость») — невозможно проверить.
- Нет рисков и плана проверки: в результате архитектура строится на предположениях.
- Стейкхолдеры перечислены формально, но их concerns одинаковые и не отражают реальных конфликтов.
- Внешние системы перечислены, но не указано, что у них бывает недоступность и как портал должен себя вести.

Критерии оценивания (10 баллов)

Часть	Баллы	Что ожидается
Граница и контекст	3	Понятно, что входит/не входит; перечислены внешние системы и ответственность.
Стейкхолдеры и concerns	3	Не менее 8; concerns конкретные, не повторяются; видны конфликты.
Цели, сценарии, метрики	3	3 цели + метрики; 3–5 ключевых сценариев, влияющих на архитектуру.
Ограничения и риски	1	Ограничения/допущения >= 6; риски >= 3 и есть способ проверки.

Практическая работа 02. Quality Attribute Scenarios (QAS) and Tactics

Фокус: Перевести качества (performance/availability/security/...) в измеримые сценарии и подобрать тактики

Ожидаемый результат

- Таблица QAS минимум по 4 качествам (желательно 6).
- Таблица «качество → тактики → как проверим».
- Описание минимум 2 компромиссов (trade-offs) и почему выбран именно такой баланс.

Теория и пояснения

Сценарии качества нужны, потому что «быстро» и «надёжно» означают разное для разных людей. QAS задаёт измеримость: в каких условиях система должна выдержать нагрузку, и что считается успехом. Тактики — это повторяемые архитектурные приёмы, которые улучшают нужное качество (например, кэширование ускоряет ответы, но усложняет согласованность).

Порядок выполнения (шаги)

8. Выберите 4–6 атрибутов качества, которые важны для вашего варианта (подсказка: расписание почти всегда требует performance; интеграции — availability; ПДн — security).
9. Для каждого качества сформулируйте 1–2 QAS по шаблону: Источник → Стимул → Среда → Артефакт → Отклик → Метрика.
10. Подберите к каждому QAS 2–3 тактики и рядом напишите, как проверите (нагрузочный тест, алерт по метрике, аудит логов, security review).
11. Выберите 2 конфликта качеств (например, availability vs consistency; security vs usability) и опишите компромисс: что выиграли и чем пожертвовали.
12. Проверьте согласованность: метрики не противоречат друг другу; условия тестирования реалистичны; тактики действительно влияют на метрику.

Пример или шаблон

```
Пример QAS строки (performance):

Источник: студент; Стимул: GET /schedule; Среда: пик публикации;
Артефакт: API расписания; Отклик: ответ; Метрика: p95 < 300 ms при 600
RPS.



Пример QAS строки (availability):

Источник: студент; Стимул: запрос статуса заявления; Среда: 1С
недоступна; Артефакт: модуль заявлений; Отклик: возвращается статус
```

«данные временно недоступны» + последнее известное значение; Метрика: не более 1% ошибок 5xx за 30 дней.

Типовые ошибки

- Использование «среднего времени» вместо p95/p99: среднее скрывает хвостовые задержки.
- Метрики без условий: «p95 < 300 ms» — но при какой нагрузке и где измеряем?
- Тактики не связаны с метрикой (например, «микросервисы» как тактика performance — это не тактика).
- Компромиссы не описаны: студент перечисляет плюсы, но не показывает минусы и последствия.
- Нет способа проверки: требования существуют «на бумаге», но никто не понимает, как подтвердить.

Критерии оценивания (10 баллов)

Часть	Баллы	Что ожидается
QAS (качества и метрики)	5	Минимум 4 качества, у каждого есть сценарий, метрика и условия.
Тактики и проверка	3	Тактики релевантны; есть «как проверим» для ключевых сценариев.
Компромиссы	2	Два trade-off описаны честно: что улучшили, что усложнили/ухудшили.

Практическая работа 03. C4 Level 1: System Context Diagram

Фокус: Нарисовать контекст: граница SmartCampus, акторы, внешние системы и каналы взаимодействия

Ожидаемый результат

- Контекстная диаграмма C4 L1 (от руки) с подписями связей и легендой.
- Короткий чеклист качества диаграммы (8 пунктов) с отметками.

Теория и пояснения

Контекстная диаграмма отвечает на вопрос «что за система и с кем она общается». На этом уровне нельзя показывать внутренние сервисы, базы данных и детали реализации. Хороший контекст помогает стейкхолдерам договориться о границе ответственности и об интеграциях.

Подсказки:

- Если вы не уверены в канале — честно подпишите «HTTP/REST (уточнить)» и добавьте это в список рисков.
- Не стесняйтесь рисовать две версии: v0.1 (черновик) и v0.2 (после правок).

Порядок выполнения (шаги)

13. Нарисуйте рамку системы (system boundary) и подпишите её «SmartCampus».
14. Вокруг рамки разместите акторов (людей/роли) и внешние системы. Элементов не должно быть слишком много: лучше 8–12.
15. Проведите связи и подпишите каждую связь глаголом + каналом (например, «просматривает расписание (HTTP)», «получает токен (OIDC)», «запрашивает задолженность (REST)»).
16. Добавьте легенду: различайте синхронные и асинхронные связи (например, сплошная и пунктирная линии).
17. Сделайте самопроверку по чеклисту: читаемость, подписи, границы, нет лишних деталей.

Типовые ошибки

- На контексте рисуют внутреннюю БД или микросервисы — это уровень контейнеров/компонентов.
- Стрелки без подписей: диаграмма превращается в «паутину».
- Нет рамки системы — непонятно, где заканчивается ответственность SmartCampus.
- Слишком много элементов (20+): диаграмма нечитаема.

Критерии оценивания (10 баллов)

Часть	Баллы	Что ожидается

Состав элементов	3	Есть все ключевые акторы и внешние системы, лишних деталей нет.
Подписи и легенда	3	Подписи связей (глагол+канал), различение sync/async, легенда.
Границы и смысл	3	Ясна граница ответственности; видны основные сценарии.
Оформление	1	Читаемо, аккуратно, есть дата/версия.

Практическая работа 04. C4 Level 2: Container Diagram

Фокус: Показать крупные части системы: клиенты, backend, хранилища, кэш, брокер, интеграции

Ожидаемый результат

- Контейнерная диаграмма C4 L2 (от руки) с технологиями и типами связей.
- Короткое пояснение: почему выбран такой набор контейнеров (3–5 предложений).

Теория и пояснения

Контейнер в C4 — это исполняемая единица: приложение, сервис, база данных, очередь, кэш. На этом уровне важно показать, где хранятся данные, где кэшируются ответы, где происходит асинхронная обработка, и что является точками отказа.

Порядок выполнения (шаги)

18. Возьмите контекст (практика 03) и перенесите внутрь границы SmartCampus основные контейнеры: web, mobile, backend API.
19. Добавьте хранилища: основную БД (PostgreSQL), кэш (Redis) и при необходимости брокер сообщений (Kafka/RabbitMQ).
20. Покажите интеграции: где синхронные вызовы (HTTP/SOAP), а где события (event). Подпишите каналы.
21. Для каждого контейнера подпишите ответственность и технологию (например, «Backend API (Java/Kotlin)», «Redis cache», «Message broker»).
22. Отметьте точки устойчивости: таймауты/ретраи, деградация, кэширование для расписания, очереди для заявлений/уведомлений.

Пример или шаблон

```
Мини-шаблон подписей контейнеров:

- Web UI (браузер) — пользовательский интерфейс

- Mobile App — мобильный интерфейс

- Backend API (Java/Kotlin) — бизнес-логика и агрегирование

- PostgreSQL — данные заявлений, пользователей, аудит

- Redis — кэш расписания и справочников

- Broker — события заявлений и уведомлений
```

Типовые ошибки

- Смешение уровней: на контейнерной диаграмме рисуют компоненты/классы.

- Не указаны технологии: теряется практический смысл для эксплуатации и команды.
- Нет различения sync/async: непонятно, где возможна деградация при сбоях интеграций.
- Не показано хранение аудита/истории: затем сложно обосновать требования неизменяемости.

Критерии оценивания (10 баллов)

Часть	Баллы	Что ожидается
Набор контейнеров	4	Показаны клиенты, backend, БД, кэш/брокер при необходимости.
Связи и подписи	3	Каналы подписаны, различены sync/async, понятны интеграции.
Устойчивость и качества	2	Есть элементы, связанные с QAS: кэш, очереди, ретраи, деградация.
Оформление	1	Читаемость, версия/дата, легенда.

Практическая работа 05. C4 Level 3: Components + Modular Boundaries

Фокус: Разделить backend на компоненты/модули по доменам и зафиксировать правила зависимостей

Ожидаемый результат

- Диаграмма компонентов C4 L3 (внутри Backend API) или схема модулей.
- Короткие правила модульных границ: что кому можно вызывать (3–6 правил).
- Список 3 «красных флагов» (что будет признаком плохого разбиения).

Теория и пояснения

На старте учебного проекта часто выгоден модульный монолит: один деплой, но строгие границы внутри. Границы лучше строить по доменам (расписание, заявления, пропуска, уведомления) и изолировать интеграции через адаптеры. Идея: «внутри домена — свои модели и правила; наружу — контракт».

Порядок выполнения (шаги)

23. Выберите контейнер Backend API и нарисуйте внутри 8–12 компонентов (или модулей) как ответственности: Schedule, Applications, Passes, Notifications, Auth, Admin, Integration adapters и т.п.
24. Проведите зависимости между компонентами. Разрешайте зависимости так, чтобы домены не слипались в один «комок».
25. Отдельно выделите слой интеграций (adapters/gateways) к Moodle/1C/библиотеке и запретите прямые вызовы внешних систем из доменной логики.
26. Сформулируйте правила модульных границ. Пример: «Applications не импортирует код Schedule; общие типы — только через контракт или события».
27. Определите, какие данные принадлежат домену (owner). Это поможет позже в моделировании БД и событий.

Типовые ошибки

- Компоненты = классы (слишком мелко) или наоборот 2 компонента «всё и сразу» (слишком грубо).
- Единый «utils/common» модуль, куда складывают доменную логику — это почти всегда путь к хаосу.
- Прямые вызовы внешних API из домена без адаптеров: потом трудно тестировать и менять интеграции.
- Нет правил зависимостей: диаграмма превращается в картинку без управляемости.

Критерии оценивания (10 баллов)

Часть	Баллы	Что ожидается

Разбиение на домены/модули	4	Компоненты отражают ответственности; количество разумное.
Правила зависимостей	3	Есть явные правила; интеграции изолированы адаптерами.
Связь с драйверами	2	Показано, как разбиение помогает изменяемости/тестируемости.
Оформление	1	Читаемо, есть легенда/версия.

Практическая работа 06. Sequence Diagrams: Happy Path + Failure Modes

Фокус: Описать ключевой сценарий во времени и показать, как система ведёт себя при сбоях

Ожидаемый результат

- Sequence diagram «happy path» для сценария заявления.
- Sequence diagram или альтернативная ветка (alt) для сбоя (например, брокер недоступен).
- Таблица: таймауты/ретраи/идемпотентность (минимум 4 пункта).

Теория и пояснения

Sequence-диаграммы полезны там, где важны порядок вызовов и реакция на ошибки. В архитектуре особенно важно явно обозначить: где ставим таймаут, сколько ретраев допускаем, какой ответ отдаём пользователю и как обеспечиваем идемпотентность.

Порядок выполнения (шаги)

28. Выберите сценарий из тетради (подача заявления → решение → уведомление) и перечислите участников (UI, Backend, БД, Broker, Notification service, деканат).
29. Нарисуйте happy path: запрос от студента, валидация, запись в БД, публикация события, обновление статуса, уведомление.
30. Добавьте failure mode: например, брокер недоступен. Покажите, что бизнес-операция не теряется (outbox/ретраи), а пользователь получает понятный статус.
31. Подпишите на диаграмме места таймаутов и ретраев (например, к внешним системам), и где нужна идемпотентность (например, повторная обработка события).
32. Сделайте таблицу правил: таймаут, ретраи, backoff, дедупликация сообщений, порядок обновления статусов.

Типовые ошибки

- На диаграмме нет ошибок и альтернатив: выглядит идеально, но не готово к реальному миру.
- Ретраи без ограничений: могут усилить нагрузку и устроить лавину.
- Нет идемпотентности: повтор события создаёт дубликаты заявлений/уведомлений.
- Синхронно зовут нестабильную внешнюю систему внутри пользовательского запроса без деградации.

Критерии оценивания (10 баллов)

Часть	Баллы	Что ожидается

Happy path	4	Сценарий полон и логичен; участники выбраны правильно.
Failure mode	3	Показана устойчивость; нет потери операции; понятный статус.
Таймауты/ретраи/ идемпотентность	2	Есть конкретные правила и места применения.
Оформление	1	Читаемо, подписи сообщений и веток.

Практическая работа 07. Data Modeling: ERD + Migrations Strategy

Фокус: Сформировать модель данных и правила изменения схемы без потери истории

Ожидаемый результат

- ERD (основные сущности и связи).
- Стратегия миграций (как меняем схему: версии, инструменты, откаты).
- Решение по аудиту/истории (append-only для решений по заявлениям).

Теория и пояснения

Модель данных должна поддерживать процессы и требования к аудиту. Если история решений по заявлениям должна быть неизменяемой, это влияет на схему: нужны отдельные таблицы истории (audit log) или подход append-only, где новые записи добавляются, а не перезаписываются.

Порядок выполнения (шаги)

33. Выделите сущности из входных данных (User, Application, Decision и т.п.) и добавьте ключевые атрибуты (id, createdAt, status).
34. Нарисуйте связи: один студент → много заявлений; одно заявление → много событий истории; роли → пользователи и т.п.
35. Выберите стратегию идентификаторов (UUID или последовательности) и кратко обоснуйте.
36. Продумайте аудит: что логируем, где храним, как защищаем от изменений (append-only, запрет UPDATE на историю).
37. Опишите стратегию миграций: как применяем изменения схемы (например, Flyway/Liquibase), как версионируем, как делаем «безопасные» миграции.

Пример или шаблон

```
Пример идеи append-only для решений:

- Таблица Application (текущее состояние)

- Таблица ApplicationHistory (каждое изменение статуса отдельной записью)

- Таблица AuditLog (кто/когда/что сделал, traceId)

Правило: историю не обновляем, только добавляем.
```

Типовые ошибки

- Смешивают справочники и транзакционные данные; нет нормализации.
- Хранят историю «в одном поле» (json) без возможности поиска и аудита.
- Не продумана миграция: схема «как в голове», но неясно, как менять в проде.
- Нет индексов по критичным запросам (расписание/поиск заявлений).

Критерии оценивания (10 баллов)

Часть	Баллы	Что ожидается
ERD и связи	5	Сущности и связи отражают кейс; нет грубых противоречий.
Аудит/история	3	История неизменяема; понятно, что и зачем логируется.
Миграции	2	Есть стратегия версионирования и безопасного изменения схемы.

Практическая работа 08. API Design: OpenAPI + Error Model

Фокус: Сформировать контракт API: эндпоинты, схемы запросов/ответов, ошибки и правила

Ожидаемый результат

- Список эндпоинтов с назначением (минимум 4).
- Единая модель ошибки + 5 примеров ошибок (validation/auth/not found/conflict/rate limit).
- Короткие правила: идемпотентность, пагинация, версионирование (по выбору).

Теория и пояснения

API — это контракт между командами и компонентами. Даже если реализации нет, контракт уже позволяет обсуждать интеграцию, безопасность и тестирование. Модель ошибок особенно важна: без неё интеграторы делают хаос из «500 Internal Server Error».

Порядок выполнения (шаги)

38. Возьмите эндпоинты из тетради и для каждого напишите: назначение, кто может вызывать (роль), какие основные поля в запросе/ответе.
39. Определите общую модель ошибки: код, сообщение, детали, traceId. Запишите как JSON-шаблон.
40. Для каждого типа ошибки придумайте пример ситуации и ожидаемый HTTP статус (400/401/403/404/409/429).
41. Продумайте идемпотентность хотя бы для одного POST (например, Idempotency-Key) и опишите правило.
42. Если есть списковые методы — добавьте правила пагинации и сортировки (limit/offset или cursor).

Пример или шаблон

```
openapi: 3.0.3

paths:

  /applications:

    post:

      summary: Submit application

      responses:

        "201": { description: Created }

        "400": { description: Validation error }

components:
```

```
schemas:
  Error:
    type: object
    properties:
      error:
        type: object
        properties:
          code: { type: string }
          message: { type: string }
          traceId: { type: string }
```

Типовые ошибки

- Смешивают бизнес-ошибки и технические: всё превращается в 500.
- Нет единого формата ошибок: каждый эндпоинт возвращает свой JSON.
- Не продуманы права доступа: любой пользователь может вызвать админ-операцию.
- Нет идемпотентности: повтор запроса создаёт дубликаты.

Критерии оценивания (10 баллов)

Часть	Баллы	Что ожидается
Контракт эндпоинтов	4	Назначение и роли понятны; схемы разумные.
Модель ошибок	4	Единый формат + примеры; статусы выбраны адекватно.
Правила (идемпотентность/пагинация/версия)	2	Есть минимум 2 правила и они применимы.

Практическая работа 09. Integration Patterns: Sync vs Async, Event Schemas

Фокус: Спроектировать устойчивую интеграцию с внешними системами и события внутри системы

Ожидаемый результат

- Сравнение sync vs async для одной интеграции (таблица плюсы/минусы).
- Схема одного события (envelope + payload) и правила версионирования.
- Механизм защиты от потери/дублирования (outbox/идемпотентный consumer).

Теория и пояснения

Если портал делает синхронный запрос в нестабильную систему во время пользовательского запроса, то сбой внешней системы превращается в сбой портала. Архитектурно это лечится очередями и статусами: пользователь получает быстрый ответ «принято», а обработка продолжается асинхронно.

Порядок выполнения (шаги)

43. Выберите интеграцию (например, 1С) и сформулируйте проблему: почему нельзя зависеть от доступности в момент запроса.
44. Сделайте таблицу: sync vs async. Укажите, что меняется в UX (статусы), что в эксплуатации (очередь), что в консистентности (eventual).
45. Опишите событие для выбранного процесса (например, ApplicationSubmitted). Укажите поля envelope: eventId, type, version, occurredAt, traceId.
46. Опишите, как защищаетесь от потери события: outbox (событие хранится в БД и публикуется отдельным процессом).
47. Опишите, как защищаетесь от дублей: идемпотентный consumer (дедупликация по eventId).

Пример или шаблон

```
{
  "eventId": "uuid",
  "type": "ApplicationSubmitted",
  "version": 1,
  "occurredAt": "2026-02-12T10:00:00Z",
  "traceId": "00-....",
  "payload": {
    "applicationId": "uuid",
```

```
    "studentId": "uuid",
    "kind": "ACADEMIC",
    "status": "SUBMITTED"
  }
}
```

Типовые ошибки

- Событие без версии: потом невозможно развивать контракт.
- Нет outbox: при падении брокера события теряются.
- Нет идемпотентности: повторная доставка создаёт дубликаты.
- Слишком «тяжёлые» события: тащат личные данные, которые не нужны подписчикам.

Критерии оценивания (10 баллов)

Часть	Баллы	Что ожидается
Обоснование sync/async	4	Таблица плюсы/минусы, учтены UX и отказоустойчивость.
Схема события	3	Есть envelope+payload, версия, traceId.
Надёжность доставки	3	Outbox/ретраи/ дедупликация описаны корректно.

Практическая работа 10. Security by Design: Threat Modeling + STRIDE Checklist

Фокус: Построить модель угроз и связать её с мерами защиты

Ожидаемый результат

- DFD (data flow diagram) с trust boundaries.
- Таблица STRIDE минимум 8 угроз: угроза → объект → мера защиты.
- Security checklist минимум 12 пунктов для API/UI/админки.

Теория и пояснения

Безопасность начинается не с «поставим WAF», а с понимания потоков данных. DFD показывает, где идут данные, где они хранятся и где пересекают границы доверия (браузер → backend, backend → внешние ИС). STRIDE помогает системно пройтись по типам угроз и не забыть важные вещи: подмена, утечки, отказ в обслуживании и т.п.

Порядок выполнения (шаги)

48. Нарисуйте DFD: процессы (UI, backend, интеграции), хранилища (БД, кэш), внешние сущности (SSO, 1С, шлюзы).
49. Отметьте trust boundaries (например, между интернетом и кластером, между кластером и 1С).
50. Заполните таблицу угроз по STRIDE: выберите 8+ угроз, привязав к конкретному элементу или потоку.
51. Для каждой угрозы укажите минимум одну меру (RBAC, MFA, аудит, rate limiting, шифрование, валидация, CSP).
52. Соберите security checklist: что проверять при разработке и ревью (например, «все input валидируются», «секреты не в git», «аудит ролей»).

Типовые ошибки

- Угрозы общие («могут украсть данные») без привязки к потоку или компоненту.
- Меры не соответствуют угрозе (например, шифрование как ответ на DoS).
- Нет границ доверия: непонятно, где именно нужно усиливать контроль.
- Секреты и доступы не описаны: кто имеет доступ к чему и как ротировать.

Критерии оценивания (10 баллов)

Часть	Баллы	Что ожидается
DFD и границы доверия	3	Потоки и хранилища понятны, trust boundaries обозначены.
STRIDE таблица	4	8+ угроз, каждая привязана и имеет меры.

Security checklist	3	12+ пунктов, покрывает API/UI/админку/секреты.

Практическая работа 11. Observability: Logs, Metrics, Traces, SLO

Фокус: Спроектировать наблюдаемость так, чтобы деградации находились быстро и расследовались по данным

Ожидаемый результат

- Список метрик (минимум 6) и лог-событий (минимум 6), привязанных к сценариям.
- 2 SLI/SLO (например, для расписания и заявлений).
- Схема observability pipeline (от сервиса до дашборда/алерта).

Теория и пояснения

Наблюдаемость — часть архитектуры. Если вы заранее не договорились о метриках и трассировке, то в реальном инциденте команда будет «гадать» по симптомам. Хороший минимум: RED-метрики по ключевым endpoint'ам, структурные логи по бизнес-событиям и end-to-end tracing.

Порядок выполнения (шаги)

53. Выберите 2–3 ключевых пользовательских сценария (расписание, подача заявления, получение статуса).
54. Для каждого сценария выпишите метрики: rate, errors, duration (p95/p99). Добавьте метрики внешних интеграций (timeouts, circuit breaker open).
55. Составьте список лог-событий: что логируем при создании заявления, при изменении статуса, при ошибке интеграции (обязательно traceId).
56. Сформулируйте 2 SLO: например, успешность /schedule и задержка p95; и для заявлений — доля успешных операций.
57. Нарисуйте pipeline: как метрики/логи/трейсы собираются (agent/collector), куда попадают (storage), где смотрим (дашборд), как алертим.

Типовые ошибки

- Метрики не привязаны к сценариям: есть «CPU%», но нет метрик по пользовательским операциям.
- Логи не структурированы и без корреляции: невозможно собрать цепочку событий.
- SLO формулируют без окна и без источника данных (где измеряем).
- Алерты настроены на «шум»: слишком чувствительные и не отражают пользовательский опыт.

Критерии оценивания (10 баллов)

Часть	Баллы	Что ожидается
Метрики и логи	5	>=6 метрик и >=6 лог-событий, привязка к сценариям, traceId.

SLI/SLO	3	2 SLO измеримы, есть окно и критерий, понятно как проверять.
Pipeline схема	2	Схема понятна, отражает сбор и потребление сигналов.

Практическая работа 12. ADR Pack: Decisions, Alternatives, Consequences

Фокус: Зафиксировать ключевые архитектурные решения так, чтобы они были объяснимы и проверяемы

Ожидаемый результат

- 3 ADR по шаблону (контекст → решение → альтернативы → последствия → как проверим).
- Список отложенных решений (deferred) минимум 5.
- Связь ADR с драйверами (в каждом ADR 1–2 драйвера).

Теория и пояснения

ADR (Architecture Decision Record) — короткая запись о решении. Её ценность в будущем: через месяц никто не помнит, почему выбрали именно так, а ADR сохраняет контекст, альтернативы и последствия.

Порядок выполнения (шаги)

58. Выберите 3 архитектурно значимых решения из вашего кейса (например: стиль системы; интеграции sync vs async; кэширование расписания).
59. Для каждого ADR опишите контекст: что болит, какие драйверы, какие ограничения.
60. Запишите решение и почему оно подходит. Укажите минимум 2 альтернативы и почему они хуже в текущих условиях.
61. Опишите последствия: что усложнится (эксплуатация, консистентность, разработка).
62. Добавьте «как проверим» (proof): прототип, нагрузочный тест, spike, security review.
63. Соберите список deferred decisions: решения, которые пока не принимаем, но помним (например, выбор брокера, формат схем сообщений).

Пример или шаблон

```
Шаблон ADR (минимум):

Title: ...

Status: proposed/accepted

Context: ...

Decision: ...

Alternatives: A/B/...

Consequences: ...
```

```
How to validate: ...
```

Типовые ошибки

- ADR превращают в эссе на 5 страниц. В ADR важна краткость и структурность.
- Нет альтернатив: тогда это не решение, а «так получилось».
- Нет последствий: команда не понимает цену выбора.
- Нет проверки: решение не подтверждено и может быть ошибкой.

Критерии оценивания (10 баллов)

Часть	Баллы	Что ожидается
3 ADR по структуре	6	Каждый содержит контекст, решение, альтернативы, последствия, проверку.
Связь с драйверами	2	В каждом ADR явно указаны драйверы/качества.
Deferred decisions	2	>=5 отложенных решений с кратким пояснением.

Практическая работа 13. Docs-as-Code: Repository Structure + CI Checks

Фокус: Сделать документацию частью разработки: структура репозитория и автоматические проверки

Ожидаемый результат

- Структура репозитория (дерево папок) для docs/architecture, docs/adr, docs/api, docs/diagrams.
- Список CI-проверок (минимум 6): ссылки, форматирование, OpenAPI lint, сборка диаграмм и т.п.
- Сценарий публикации документации (например, GitHub Pages/GitLab Pages).

Теория и пояснения

Документация стареет, если её пишут «в отдельном мире». Docs-as-code означает: дока хранится рядом с кодом, проходит ревью и проверяется автоматически. Так уменьшается риск «документация не соответствует системе».

Порядок выполнения (шаги)

64. Нарисуйте структуру репозитория: где лежит архитектура, где ADR, где OpenAPI, где диаграммы.
65. Определите формат: Markdown + генератор (MkDocs/Docusaurus) или просто Markdown с публикацией.
66. Составьте список CI проверок: markdown lint, link check, OpenAPI lint (spectral), сборка диаграмм (PlantUML), unit tests (если есть).
67. Опишите правило ревью: кто должен approve изменения архитектуры/ADR.
68. Опишите публикацию: по merge в main автоматически собирается сайт и публикуется.

Пример или шаблон

```
Пример дерева папок:

repo/

  docs/

    architecture/

    adr/

    api/

    diagrams/

  src/

  .github/workflows/
```

Типовые ошибки

- Документация живёт отдельно от репозитория → быстро устаревает.
- Нет автоматических проверок → в доку попадают битые ссылки и неверные схемы.
- ADR не версионируются и не проходят ревью: решения «плавают».

Критерии оценивания (10 баллов)

Часть	Баллы	Что ожидается
Структура репо	3	Понятно где что лежит; удобно искать архитектуру и контракты.
CI проверки	5	>=6 проверок, они релевантны (OpenAPI, ссылки, диаграммы).
Публикация и процесс	2	Есть понятный сценарий публикации и ревью.

Практическая работа 14. Containerization: Dockerfile + Compose for Local Dev

Фокус: Подготовить контейнеры для локальной разработки и воспроизводимого окружения

Ожидаемый результат

- Dockerfile (концепт) для backend.
- docker-compose.yml (концепт) с postgres, redis, broker, mock-сервисом.
- Список 6 best practices для Docker (безопасность, размеры образов, переменные окружения).

Теория и пояснения

Контейнеризация помогает сделать окружение воспроизводимым: одинаково на машинах студентов и в CI. Важно придерживаться best practices: минимальные базовые образы, многоступенчатая сборка, не хранить секреты в образе, задавать healthcheck.

Порядок выполнения (шаги)

69. Определите сервисы для локального окружения (из тетради): backend, postgres, redis, broker, mock внешней системы.
70. Набросайте Dockerfile: базовый образ, сборка, запуск приложения; при возможности — multi-stage.
71. Набросайте docker-compose.yml: сети, порты, volumes, переменные окружения, depends_on.
72. Опишите, где и как хранятся конфиги: .env (локально), secrets (в проде).
73. Запишите best practices: не запускать root, закреплять версии, минимизировать слои, сканировать уязвимости.

Пример или шаблон

```
Пример (упрощённый) Dockerfile:

FROM eclipse-temurin:21-jre

WORKDIR /app

COPY build/libs/app.jar app.jar

USER 10001

EXPOSE 8080

ENTRYPOINT ["java","-jar","app.jar"]
```

Типовые ошибки

- Секреты (пароли/токены) прописаны прямо в Dockerfile или compose.

- Образ огромный и собирается долго; нет кэширования слоёв.
- Запуск под root без необходимости.
- В compose нет volumes для БД → данные теряются между перезапусками.

Критерии оценивания (10 баллов)

Часть	Баллы	Что ожидается
Dockerfile	4	Структура разумная, учтены безопасность и запускаемость.
Compose	4	Есть нужные сервисы, переменные, volumes; логичное окружение.
Best practices	2	>=6 правил, они релевантны.

Практическая работа 15. Kubernetes Basics: Deployment, Service, Ingress, Resources

Фокус: Собрать базовый deployment-манифест и продумать ресурсы и доступность в кластере

Ожидаемый результат

- Набор манифестов (концепт): Deployment, Service, Ingress.
- Requests/limits для backend (обоснование в 2–3 строках).
- Пробы: readiness/liveness (что проверяют).

Теория и пояснения

Kubernetes не делает систему «надёжной автоматически». Он даёт примитивы: деплой, масштабирование, рестарты, балансировку. Архитектор должен продумать ресурсы, точки отказа, конфигурацию и наблюдаемость.

Порядок выполнения (шаги)

74. Определите, какие компоненты деплоятся в кластер (backend, возможно broker, redis) и какие остаются внешними (например, 1С).
75. Набросайте Deployment для backend: образ, порты, переменные окружения, ConfigMap/Secret (концептуально).
76. Добавьте readinessProbe (готовность принимать трафик) и livenessProbe (жив ли процесс).
77. Набросайте Service и Ingress: как пользователи попадают в систему, где TLS.
78. Укажите requests/limits и количество реплик. Свяжите с SLO и ресурсами кластера из варианта.

Пример или шаблон

```
Мини-фрагмент Deployment:

spec:

  replicas: 2

  template:

    spec:

      containers:

      - name: backend

        image: smartcampus/backend:1.0

        resources:

          requests: { cpu: "500m", memory: "512Mi" }
```

```
          limits:   { cpu: "1",    memory: "1Gi" }
```

Типовые ошибки

- Не указаны requests/limits: в реальном кластере это приводит к непредсказуемости.
- Путают readiness и liveness: сервис либо не получает трафик, либо рестартится бесконечно.
- Нет TLS/Ingress: непонятно, как пользователи безопасно попадают в систему.
- Реплики не связаны с доступностью: одна реплика = один SPOF.

Критерии оценивания (10 баллов)

Часть	Баллы	Что ожидается
Манифесты	5	Deployment/Service/ Ingress присутствуют и логичны.
Ресурсы и реплики	3	requests/limits и replicas обоснованы драйверами/ресурсами.
Пробы	2	readiness/liveness описаны корректно.

Практическая работа 16. Performance: Load Test Plan + Bottlenecks

Фокус: Составить план нагрузочного тестирования и определить, где искать узкие места

Ожидаемый результат

- План нагрузочного теста (цель, сценарий, профиль нагрузки, метрики, критерии успеха).
- Список гипотез узких мест (минимум 5) и как проверить каждую.
- Черновик скрипта k6/JMeter (концепт) для /schedule.

Теория и пояснения

Производительность — это не только «мощнее сервер». Это запросы в БД, кэш, размер ответов, пулы соединений, конкуренция, блокировки. Нагрузочный тест нужен, чтобы отделить факты от ощущений.

Порядок выполнения (шаги)

79. Зафиксируйте цель: например, выдержать пиковую нагрузку расписания с p95 ниже заданного порога.
80. Опишите сценарий: какие параметры запроса меняются (группа, дата), сколько уникальных пользователей, какова длительность теста.
81. Определите метрики: p95/p99 latency, error rate, RPS, нагрузка на БД, hit rate кэша.
82. Составьте список гипотез узких мест (например: медленный SQL, отсутствие индекса, холодный кэш, большой payload, ограничение пула).
83. Набросайте тестовый скрипт (k6): этапы ramp-up, steady, ramp-down; проверка статусов и времени ответа.
84. Опишите, как анализируете результаты: что смотрим в Grafana/логах/профайлере, как принимаем решение о кэше/индексе.

Пример или шаблон

```
Пример (упрощённый) k6 сценария:

import http from 'k6/http';

import { check } from 'k6';

export const options = { stages: [ {duration:'2m', target:200},
{duration:'5m', target:600} ] };

export default function() {

  const res = http.get('http://host/schedule?date=today&group=G-101');

  check(res, { 'status 200': (r) => r.status === 200 });
```

```
}
```

Типовые ошибки

- Нет критериев успеха: тест прогнали, но непонятно, хорошо или плохо.
- Тестят «всё подряд» вместо критичного сценария.
- Не фиксируют условия окружения (ресурсы, версии) — результаты нельзя сравнивать.
- Смотрят только latency, но не error rate и не нагрузку на БД/кэш.

Критерии оценивания (10 баллов)

Часть	Баллы	Что ожидается
План теста	5	Цель, профиль нагрузки, метрики и критерии успеха определены.
Гипотезы и проверки	3	>=5 узких мест и способ проверки каждого.
Скрипт/черновик	2	Есть скрипт или структура сценария для инструмента.

Практическая работа 17. Reliability: SLO, Error Budget, Incident Workflow

Фокус: Связать надёжность сервиса с процессом изменений и реакцией на инциденты

Ожидаемый результат

- 2 SLO + расчёт error budget (в процентах или минутах).
- Схема процесса инцидента: обнаружение → triage → устранение → постмортем.
- Список 6 практик надёжности (timeouts, retries, circuit breaker, bulkhead, бэкапы и т.п.).

Теория и пояснения

SLO задаёт целевое качество сервиса, а error budget показывает, сколько «ошибок» можно позволить в периоде. Это связывает эксплуатацию и разработку: если бюджет сгорел — надо стабилизировать систему, а не выпускать новые фичи.

Порядок выполнения (шаги)

85. Возьмите 2 ключевых SLO (например, расписание и заявления). Запишите окно измерения (например, 30 дней).
86. Посчитайте error budget: при 99.5% за 30 дней допустимо 0.5% ошибок/недоступности. Переведите это в минуты (30 дней ≈ 43 200 минут).
87. Опишите, какие события считаются «ошибкой» для SLO (5xx, таймауты, недоступность).
88. Нарисуйте процесс инцидента: кто получает алерт, как классифицируем (sev1/sev2), где фиксируем таймлайн, как закрываем.
89. Составьте список практик надёжности и привяжите к вашим интеграциям/контейнерам: таймауты, ретраи с backoff, circuit breaker, очереди, деградация.
90. Укажите окна обслуживания из варианта и как вы минимизируете влияние (banner, read-only режим, отключение функций).

Типовые ошибки

- SLO без определения SLI (что измеряем) и без источника данных.
- Error budget посчитан неверно или не привязан к процессу изменений.
- Инциденты описаны как «что-нибудь сделаем», нет ролей и шагов.
- Не учтены окна обслуживания внешних систем.

Критерии оценивания (10 баллов)

Часть	Баллы	Что ожидается
SLO и расчёт бюджета	5	2 SLO + корректный error

		budget и определение ошибок.
Процесс инцидента	3	Понятные шаги, роли и артефакты (таймлайн/постмортем).
Практики надёжности	2	>=6 практик и связь с архитектурой.

Практическая работа 18. Refactoring Plan: From Prototype to Maintainable System

Фокус: Составить план улучшения архитектуры, если прототип начал «пахнуть» техдолгом

Ожидаемый результат

- Список проблем (минимум 6) и их влияние (скорость изменений/качество/риски).
- План рефакторинга на 3 этапа (что делаем сначала, что потом).
- Правила модульных зависимостей (обновлённые) и меры контроля (ревью/тесты).

Теория и пояснения

Техдолг чаще всего появляется из-за отсутствия границ: общий utils, циклические зависимости, общая таблица «users» для всего и сразу, прямые вызовы интеграций. План рефакторинга должен быть постепенным: маленькие шаги, которые уменьшают риск.

Порядок выполнения (шаги)

91. По симптомам из варианта выпишите конкретные проблемы (например, циклические зависимости, общий utils, домены «слились»).
92. Для каждой проблемы укажите последствия: что становится сложно (тестирование, изменения, безопасность, перформанс).
93. Сформируйте целевую структуру модулей (можно ссылаться на практику 05) и правила зависимостей.
94. Составьте план на 3 этапа: (1) быстрые безопасные улучшения, (2) структурные изменения, (3) оптимизации и «полировка».
95. Для каждого этапа укажите «как проверим, что стало лучше» (метрики, количество циклов, время сборки, покрытие тестами).

Типовые ошибки

- План слишком общий («перепишем всё») — это не план, а желание.
- Нет приоритизации: начинают с красоты кода вместо снижения рисков.
- Нет проверки эффекта: непонятно, стало лучше или просто по-другому.

Критерии оценивания (10 баллов)

Часть	Баллы	Что ожидается
Диагностика проблем	4	>=6 проблем, каждая описана конкретно и с последствиями.
План этапов	4	3 этапа, есть приоритеты и риски, шаги

		реалистичны.
Правила и контроль	2	Есть правила зависимостей и механизм их соблюдения.

Практическая работа 19. Architecture Review: Checklist + Risk Register

Фокус: Провести ревизию архитектуры перед защитой и подготовить реестр рисков

Ожидаемый результат

- Чеклист ревью (минимум 25 пунктов) по направлениям: данные, интеграции, безопасность, эксплуатация, качество артефактов.
- Risk register: минимум 8 рисков с оценкой severity/likelihood и планом mitigation.
- Список «top-3 риска» и что делаете в первую очередь.

Теория и пояснения

Архитектурное ревью — это не поиск «идеала», а поиск рисков и противоречий. Цель: выявить места, где система может сломаться или стать очень дорогой в сопровождении.

Порядок выполнения (шаги)

96. Соберите все артефакты (практики 01–18) и пройдитесь по ним как по единому пакету: нет ли противоречий между диаграммами, QAS и ADR.
97. Составьте чеклист по 5 направлениям: (1) функциональность и границы, (2) данные и аудит, (3) интеграции и устойчивость, (4) безопасность, (5) эксплуатация и наблюдаемость.
98. Заполните риск-реестр: для каждого риска укажите вероятность, влияние, признаки (signals) и меры снижения.
99. Выберите top-3 риска и опишите конкретные действия (прототип, тест, изменение архитектуры, дополнительный контроль).
100. Сформируйте итог: какие вопросы остаются открыты и как вы их закроете.

Типовые ошибки

- Чеклист формальный: пункты не привязаны к вашему кейсу.
- Риски без мер: «есть риск» — и всё.
- Нет приоритизации: все риски одинаковые.
- Не ищут противоречия между артефактами (например, ADR говорит про события, а диаграммы — только sync).

Критерии оценивания (10 баллов)

Часть	Баллы	Что ожидается
Чеклист	4	>=25 пунктов, покрывает 5 направлений, применим к кейсу.
Risk register	4	>=8 рисков, оценка и mitigation конкретные.
Top-3 и план	2	Есть приоритеты и

		конкретные действия.

Практическая работа 20. Final Architecture Package + Defense

Фокус: Собрать пакет артефактов в единую историю и подготовить защиту

Ожидаемый результат

- Список артефактов итогового пакета (по чеклисту) с пометкой «готово/нужно улучшить».
- Короткий «архитектурный рассказ» на 5–7 минут: драйверы → решения → как обеспечили качества.
- Слайд/лист «главные риски и следующий шаг».

Теория и пояснения

На защите оценивается не красота диаграмм, а способность объяснить архитектуру: какие драйверы были главными, какие решения приняты, как обеспечили performance/availability/security, как система переживает сбои интеграций и как её эксплуатировать.

Порядок выполнения (шаги)

101. Соберите артефакты в порядок: brief → QAS → C4 (L1–L3) → sequence → data → API → integrations → security → observability → ADR → deployment.
102. Пройдитесь по чеклисту итогового пакета (в конце методички) и отметьте пробелы.
103. Подготовьте рассказ: 1) контекст и границы, 2) драйверы и качества, 3) ключевые решения (ADR), 4) как справляемся со сбоями, 5) как мониторим и разворачиваем.
104. Сформулируйте 3 главных риска и что вы бы сделали, если бы проект продолжался ещё месяц (план улучшений).
105. Отрепетируйте: объяснение должно быть логичным и укладываться во время.

Типовые ошибки

- Пересказывают все детали вместо истории: аудитории становится скучно и непонятно.
- Решения не связаны с драйверами: звучит как набор технологий.
- Нет обсуждения рисков и отказов: выглядит «идеально», но нереалистично.

Критерии оценивания (10 баллов)

Часть	Баллы	Что ожидается
Полнота пакета	4	Артефакты присутствуют и согласованы.
Аргументация	4	Решения связаны с драйверами и качествами, учтены сбои.

Защита	2	Понятная структура рассказа, время соблюдено.

Приложение А. Чеклист итогового архитектурного пакета

- ☐ Architecture Brief: границы, стейкхолдеры, цели, ограничения, риски.
- ☐ QAS: сценарии качества с метриками и способами проверки.
- ☐ C4 L1 Context: акторы и внешние системы, каналы, легенда.
- ☐ C4 L2 Container: контейнеры, БД/кэш/брокер, технологии, sync/async связи.
- ☐ C4 L3 Component: модули/компоненты и правила зависимостей.
- ☐ Sequence: happy path + failure mode (таймауты/ретраи/идемпотентность).
- ☐ ERD + миграции + аудит/история.
- ☐ OpenAPI: эндпоинты + модель ошибок + правила идемпотентности.
- ☐ Интеграции: sync vs async, событие с версией, outbox/дедупликация.
- ☐ Security: DFD + STRIDE + checklist.
- ☐ Observability: метрики/логи/трейсы + SLO + pipeline.
- ☐ ADR pack: 3 решения + альтернативы + последствия + проверки.
- ☐ Docs-as-code: структура и CI проверки.
- ☐ Docker/Compose для локальной разработки.
- ☐ Kubernetes: манифесты, ресурсы, probes.
- ☐ Performance: план теста и узкие места.
- ☐ Reliability: error budget и процесс инцидентов.
- ☐ Refactoring plan: 3 этапа улучшений.
- ☐ Architecture review: чеклист и risk register.

Приложение B. Короткие шаблоны (для переписывания в тетрадь)

B.1. Шаблон таблицы stakeholders & concerns

Стейкхолдер	Concerns (вопросы/опасения)	Метрика/критерий
...	...	...
...	...	...
...	...	...
...	...	...
...	...	...

B.2. Шаблон QAS

Источник	Стимул	Среда	Артефакт	Отклик	Метрика
...	...	...	...	...	...
...	...	...	...	...	...
...	...	...	...	...	...
...	...	...	...	...	...

B.3. Шаблон ADR

Title: ...

Status: proposed / accepted / deprecated

Context: ...

Decision: ...

Alternatives: ...

Consequences: ...

How to validate: ...