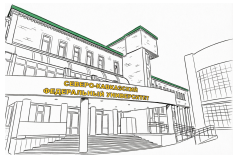

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт перспективной инженерии

Департамент цифровых, робототехнических систем и электроники

АРХИТЕКТУРА ИНФОРМАЦИОННЫХ СИСТЕМ

Методические указания
к выполнению лабораторных работ

Направление подготовки: 09.03.02 «Информационные системы и технологии»
Профиль: «Прикладное программирование и интернет-технологии»

Квалификация (степень) выпускника: бакалавр
Форма обучения: очная

Ставрополь – 2026

УДК 004.41:004.056
ББК 32.973.26-018.2

АРХИТЕКТУРА ИНФОРМАЦИОННЫХ СИСТЕМ

Методические указания к лабораторным работам

Аннотация: Методические указания содержат комплект из 9 лабораторных работ по дисциплине «Архитектура информационных систем». Каждая работа включает цель, задачи, теоретический минимум, пошаговый разбор примера, фрагменты программного кода для моделирования архитектурных решений и генерации спецификаций, индивидуальное задание для самостоятельного выполнения, а также вопросы для самоконтроля. Материал охватывает ключевые этапы проектирования современных ИС: от выбора архитектурного стиля и многовидового моделирования до микросервисной декомпозиции, проектирования API, облачной оркестрации и оценки качества по методике ATAM. Рекомендуемый объем — 36 часов лабораторных занятий. Работы выполняются с использованием инструментов визуального моделирования (PlantUML, Draw.io), средств контейнеризации (Docker, Kubernetes) и среды Python для автоматизации проектирования и анализа архитектурных метрик.
Составитель: к.т.н. Д.Л. Винокурский
Рецензент: к.ф.м.н. Е.В. Крахоткина

Содержание

Введение

Дисциплина «Архитектура информационных систем» занимает системообразующее место в подготовке специалистов, способных проектировать сложные, масштабируемые и отказоустойчивые программно-аппаратные комплексы. Понимание архитектурных принципов, умение выбирать стили проектирования и документировать решения является необходимым условием для создания современных цифровых продуктов, работающих в распределённых и облачных средах. **Актуальность дисциплины** обусловлена следующими факторами:

- Переход от монолитных систем к микросервисным и событийно-ориентированным архитектурам требует новых подходов к декомпозиции и управлению зависимостями;
- Рост требований к масштабируемости и доступности (SLA 99.95% и выше) диктует необходимость использования облачных паттернов и оркестрации;
- Стандартизация архитектурного документирования (ISO/IEC/IEEE 42010, C4) повышает прозрачность и сопровождаемость проектов;
- Внедрение практик DevOps и SRE напрямую зависит от качества архитектурных решений на ранних этапах жизненного цикла.

Цель методических указаний — предоставить обучающимся структурированный практический материал для освоения методов проектирования, моделирования и оценки архитектуры информационных систем. **Задачи лабораторного практикума:**

1. Сформировать навыки выбора архитектурного стиля в зависимости от бизнес-требований и ограничений;
2. Научить применять многовидовое моделирование и стандарты документирования архитектуры;
3. Освоить методы декомпозиции систем, проектирования компонентов и контрактов взаимодействия;
4. Развить способность реализовывать микросервисные решения, проектировать REST/gRPC API и настраивать интеграцию через брокеры сообщений;
5. Подготовить к оценке качества архитектуры, выявлению рисков и принятию обоснованных проектных решений.

Организация выполнения работ:

- Каждая лабораторная работа рассчитана на 4 академических часа;
- Перед началом работы студент обязан изучить теоретический материал и подготовить схему/диаграмму в соответствии с заданием;
- Практическая часть выполняется в среде Python с использованием библиотек для генерации спецификаций, работы с YAML/JSON и моделирования архитектурных сценариев;
- Отчёт по работе должен содержать: цель, архитектурные диаграммы, листинги кода, таблицы решений, анализ компромиссов, выводы;
- Защита работы включает демонстрацию артефактов проектирования и устный ответ по методологическим основам.

Требования к программному обеспечению:

- Язык программирования: Python 3.9+;
- Библиотеки: Pydantic, FastAPI (для генерации OpenAPI), PyYAML, requests, networkx, matplotlib; : PlantUML, Draw.io/Lucidchart, Postman/Insomnia, Docker Desktop;

- Среда разработки: Jupyter Notebook, VS Code или PyCharm.

1 Введение в архитектуру информационных систем. Анализ архитектурных стилей

Цель: Сформировать представление о базовых архитектурных стилях и критериях их выбора для типовых предметных областей. **Задачи:**

- Изучить основные архитектурные стили: монолит, клиент-сервер, слоистая, микросервисная, событийно-ориентированная архитектура;
- Освоить матрицу оценки стилей по атрибутам качества (масштабируемость, сопровождаемость, время вывода на рынок, сложность);
- Научиться обосновывать выбор стиля на основе бизнес-требований и технических ограничений;
- Реализовать скрипт для автоматизированного сравнения стилей по заданным весовым коэффициентам.

Теоретическое описание: Архитектурный стиль определяет фундаментальную организацию системы, набор элементов и правила их взаимодействия. Выбор стиля является стратегическим решением, влияющим на весь жизненный цикл ИС. Оценка проводится по атрибутам качества: модифицируемость (Q_m), производительность (Q_p), доступность (Q_a), безопасность (Q_s). Итоговый балл стиля S_i вычисляется как взвешенная сумма: $Score(S_i) = \sum_j w_j \cdot Q_{ij}$, где w_j — важность атрибута, $Q_{ij} \in [1, 5]$ — оценка соответствия. **Разобранный пример:** 1. Задача: система электронного документооборота для среднего бизнеса.
2. Атрибуты: масштабируемость ($w = 0.3$), скорость разработки ($w = 0.25$), сопровождаемость ($w = 0.25$), отказоустойчивость ($w = 0.2$).
3. Оценки стилей (1-5): Молит (3,5,4,2), Микросервисы (5,2,5,5), Событийная (4,2,5,4).
4. Расчёт: $Score_{micro} = 0.3 \cdot 5 + 0.25 \cdot 2 + 0.25 \cdot 5 + 0.2 \cdot 5 = 4.25$ — оптимальный выбор.
Программный код (оценка архитектурных стилей):

```
import numpy as np
import pandas as pd

#
weights = np.array([0.30, 0.25, 0.25, 0.20])  #
                                        ,                    ,
                        ,

#                                                (              :              ,
                      :                    )
styles = ['          ', '                        ', '
                        _                        ', '                  _
                                 ']
matrix = np.array([
    [3, 5, 4, 2],  #
    [5, 2, 5, 5],  #
    [4, 2, 5, 4],  # EDA
    [4, 3, 4, 4]   # SOA
])

scores = matrix @ weights
df = pd.DataFrame({'        ': styles, '        ': scores})
df = df.sort_values('        ', ascending=False).reset_index(drop=
   True)
```

```
19
20 print("                                                    :\n")
21 print(df.to_string(index=False))
22 print(f"\ n                                    : {df.iloc[0]['
              ']} (         : {df.iloc[0]['          ']:.2f})")
```

Индивидуальное задание: Проведите сравнительный анализ архитектурных стилей для системы онлайн-бронирования отелей. Определите веса атрибутов на основе сценариев использования, выполните расчёт и обоснуйте выбор. **Вопросы для самопроверки:**

1. В чём принципиальное отличие микросервисной архитектуры от сервис-ориентированной (SOA)?
2. Почему монолитная архитектура часто остаётся предпочтительной на старте проекта?
3. Как атрибут «сопровождаемость» влияет на долгосрочную стоимость владения системой?

2 Моделирование архитектуры с использованием 4+1 вида представлений

Цель: Освоить многовидовое описание архитектуры в соответствии с методологией Круктена (4+1 View Model) и стандартом ISO/IEC/IEEE 42010. **Задачи:**

- Изучить назначение каждого вида: логический, процессный, физического развёртывания, разработки и сценариев;
- Научиться идентифицировать стейкхолдеров и их информационные потребности; еализовать генерацию PlantUML-диаграмм для логического и процессного видов из спецификации;
- Проверить согласованность представлений на сквозных сценариях.

Теоретическое описание: Модель 4+1 адресует различные точки зрения: логический вид (разработчики, функциональная декомпозиция), процессный (интеграторы, потоки управления), вид разработки (программисты, модульная структура), физический (инженеры, топология сети). Сценарии (+1) валидируют архитектуру на реальных use-cases. Каждый вид использует специфические нотации (UML Class, Sequence, Deployment, Component Diagrams). **Разобранный пример:** 1. Система: платформа дистанционного обучения.
2. Стейкхолдеры: преподаватель, студент, администратор, DevOps.
3. Логический вид: классы Course, User, Lecture, Assessment.
4. Процессный вид: потоки «загрузка материала», «прохождение теста», «уведомление».
5. Генерация PlantUML для автоматизации документирования. **Программный код (генерация PlantUML из YAML-спецификации):**

```
import yaml
import textwrap

spec_yaml = """
system: "
                  "
logical_components:
  - name: AuthService
    responsibilities: [                              ,
                                      ]
  - name: CourseService
    responsibilities: [                                  ,
                      ]
  - name: AssessmentService
    responsibilities: [          ,                 ,
                           ]
relationships:
  - from: AuthService
    to: CourseService
    type: uses
  - from: CourseService
    to: AssessmentService
    type: triggers
"""

spec = yaml.safe_load(spec_yaml)
```

```
uml_lines = [f'@startuml\n!theme plain\ntitle
        : {spec["system"]}\npackage "{spec["system"]}" {{']

for comp in spec['logical_components']:
    uml_lines.append(f'  class {comp["name"]} {{')
    for resp in comp['responsibilities']:
        uml_lines.append(f'    + {resp}')
    uml_lines.append('  }')

for rel in spec['relationships']:
    uml_lines.append(f'  {rel["from"]} --> {rel["to"]} : {rel["type
       "]}')

uml_lines.append('}\n@enduml')
plantuml_code = '\n'.join(uml_lines)

print("                                  PlantUML       :")
print(plantuml_code)
#                                                                .
   puml                                VS Code / PlantUML Server
```

Индивидуальное задание: Разработайте 4+1 вид представлений для системы онлайн-очереди в госучреждении. Сгенерируйте диаграммы логического и процессного видов, опишите 2 валидирующих сценария. **Вопросы для самопроверки:**

1. Почему логический вид не должен содержать информацию о развёртывании?
2. Как сценарии помогают выявить пробелы в архитектурном проектировании?
3. Какие инструменты позволяют поддерживать синхронизацию между 5 видами представлений?

3 Проектирование компонентной архитектуры и интерфейсов взаимодействия

Цель: Научиться декомпозировать систему на компоненты, специфицировать контракты взаимодействия и управлять связностью/зацеплением. **Задачи:**

- Изучить принципы компонентного проектирования: высокая связность, низкое зацепление, чёткие контракты;
- Освоить методы выделения границ компонентов (Domain-Driven Design, Bounded Contexts);
- Научиться описывать интерфейсы с использованием схем данных и версионирования; еализовать генератор контрактов на основе Pydantic и JSON Schema.

Теоретическое описание: Компонент — заменяемый модуль системы с чётко определёнными интерфейсами. Зацепление (coupling) измеряет степень зависимости компонентов, связность (cohesion) — степень внутренней согласованности. Контракт включает: сигнатуру операций, форматы данных, семантику ошибок, гарантии доставки, версию. Нарушение контрактов ведёт к каскадным отказам и усложнению рефакторинга. **Разобранный пример:** 1. Домен: обработка заказов интернет-магазина.
2. Компоненты: Catalog, Cart, Order, Payment, Notification.
3. Контракт Cart → Order: CreateOrderRequest items[], total, userId, Response orderId, status.
4. Версионирование: v1/v2, backward-compatible изменения. **Программный код (генерация JSON Schema контрактов):**

```
from pydantic import BaseModel, Field, EmailStr
from pydantic.json_schema import models_json_schema

class OrderItem(BaseModel):
    product_id: str = Field(..., description="
                                                ")
    quantity: int = Field(..., ge=1, description="
                             ")
    price: float = Field(..., gt=0, description="
                       ")

class CreateOrderRequest(BaseModel):
    user_id: str = Field(..., description="
                                  ")
    items: list[OrderItem]
    shipping_address: str
    payment_method: str = Field(..., pattern="^(card|cash|transfer)$"
       )

#                     JSON Schema         API
schema = models_json_schema([CreateOrderRequest], title="Order API
   Contract")
print("                  CreateOrderRequest (JSON Schema):")
import json
print(json.dumps(schema, indent=2, ensure_ascii=False))
```

Индивидуальное задание: Спроектируйте компонентную архитектуру для системы бронирования авиабилетов. Опишите 3 ключевых интерфейса, сгенерируйте их JSON

Schema, предложите стратегию обратной совместимости при изменении формата даты.

Вопросы для самопроверки:

1. Как принцип единой ответственности (SRP) проявляется на уровне компонентов?
2. Почему контракты должны версионироваться, даже если изменения кажутся незначительными?
3. В чём разница между контрактным тестированием и интеграционным тестированием?

4 Реализация микросервисной архитектуры: паттерны декомпозиции и коммуникации

Цель: Освоить практические приёмы перехода от монолита к микросервисам, выбрать стратегии декомпозиции и типы коммуникации. **Задачи:**

- Изучить паттерны декомпозиции: по бизнес-домену (DDD), по поддомену, Strangler Fig, по данным;
- Сравнить синхронную (HTTP/RPC) и асинхронную (очереди, события) коммуникацию; еализовать базовый микросервис с межсервисным вызовом и обработкой ошибок;
- Оценить влияние распределённости на транзакционную целостность (Saga, 2PC).

Теоретическое описание: Микросервисная архитектура предполагает независимое развёртывание небольших сервисов, владеющих своими данными. Синхронная коммуникация проще в отладке, но создаёт жёсткие зависимости и риск каскадных отказов. Асинхронная повышает отказоустойчивость и масштабируемость, но усложняет отслеживание транзакций. Паттерн Strangler Fig позволяет постепенно выносить функциональность из монолита без «большого взрыва». **Разобранный пример:** 1. Задача: выделение сервиса уведомлений из монолита.
2. Паттерн: Strangler Fig + API Gateway.
3. Коммуникация: OrderService → Message Broker → NotificationService.
4. Реализация: FastAPI endpoint, вызов через ‘httpx‘, обработка retry и circuit breaker.
Программный код (микросервис с межсервисным вызовом):

```python
from fastapi import FastAPI, HTTPException
import httpx
import asyncio

app = FastAPI(title="Notification Service")

NOTIFICATION_API_URL = "http://notification-worker:8081/send"

@app.post("/api/v1/notify")
async def send_notification(user_id: str, message: str):
    payload = {"user_id": user_id, "content": message}
    async with httpx.AsyncClient(timeout=3.0) as client:
        try:
            response = await client.post(NOTIFICATION_API_URL, json=
                payload)
            response.raise_for_status()
            return {"status": "sent", "message_id": response.json().
                get("id")}
        except httpx.RequestError as e:
            #               retry / dead-letter queue

            raise HTTPException(status_code=503, detail=f"
                Notification service unavailable: {e}")

#              : uvicorn app:app --reload --port 8080
```

Индивидуальное задание: Разработайте стратегию декомпозиции монолитной CRM-системы на 4 микросервиса. Определите тип коммуникации между ними, опишите об-

работку распределённых транзакций при создании клиента и счёта. **Вопросы для самопроверки:**

1. Когда предпочтительнее использовать синхронный REST, а когда асинхронные сообщения?
2. Как паттерн Saga решает проблему распределённых транзакций без 2PC?
3. Почему shared database является антипаттерном в микросервисной архитектуре?

5 Проектирование и документирование REST/gRPC API

Цель: Научиться проектировать, версионировать и документировать API с использованием стандартов OpenAPI и Protocol Buffers. **Задачи:**

- Изучить принципы REST: ресурсы, методы, идемпотентность, статус-коды, HATEOAS;
- Освоить gRPC: protobuf-схемы, стриминг, HTTP/2, преимущества для высоконагруженных систем;
- Научиться генерировать интерактивную документацию (Swagger/Redoc) и клиентские заглушки; еализовать автоматическую валидацию и генерацию спецификаций.

Теоретическое описание: REST опирается на HTTP-методы и ресурсно-ориентированную модель. Идемпотентность гарантирует, что повторный вызов не изменит состояние сервера (GET, PUT, DELETE). gRPC использует бинарный формат protobuf, обеспечивает строгую типизацию, встроенную генерацию кода и эффективную сериализацию. Выбор зависит от требований к пропускной способности, latency, экосистеме клиентов. **Разобранный пример:** 1. Домен: управление пользователями.
2. REST: GET /users/id, POST /users, PATCH /users/id/profile.
3. gRPC: rpc GetUser(GetUserRequest) returns (User); rpc StreamUsers(Empty) returns (stream User).
4. Генерация OpenAPI 3.0 спецификации из Pydantic-моделей. **Программный код (генерация OpenAPI спецификации):**

```python
from fastapi import FastAPI
from pydantic import BaseModel
from fastapi.openapi.utils import get_openapi
import json

app = FastAPI(title="User Management API", version="1.2.0")

class UserCreate(BaseModel):
    username: str
    email: str
    role: str = "user"

class UserResponse(BaseModel):
    id: str
    username: str
    email: str
    created_at: str

@app.post("/users", response_model=UserResponse)
def create_user(user: UserCreate):
    return {"id": "usr_123", **user.dict(), "created_at": "2026-06-01
       T10:00:00Z"}

#                 OpenAPI JSON
openapi_spec = get_openapi(
    title=app.title,
    version=app.version,
    routes=app.routes,
)
```

```
print("OpenAPI 3.0 Spec (                    ):")
print(json.dumps(openapi_spec["paths"], indent=2, ensure_ascii=False)
    )
```

Индивидуальное задание: Спроектируйте REST и gRPC версии API для системы складского учёта. Сравните спецификации по объёму, читаемости и поддержке стриминга. Сгенерируйте клиентский код для Python. **Вопросы для самопроверки:**

1. Почему PATCH предпочтительнее PUT для частичного обновления ресурсов?
2. Как gRPC решает проблему backward compatibility при изменении protobuf-схем?
3. В чём разница между контракт-фёрст и код-фёрст подходами к проектированию API?

6 Архитектура данных: стратегии хранения, репликации и кэширования

Цель: Научиться выбирать стратегии хранения данных, проектировать репликацию и кэширование с учётом требований к согласованности и производительности. **Задачи:**

- Изучить CAP-теорему и BASE-принципы для распределённых хранилищ;
- Сравнить реляционные, документо-ориентированные, key-value и графовые СУБД;
- Освоить паттерны кэширования: Cache-Aside, Write-Through, Read-Through; еализовать симулятор нагрузки с кэшем для оценки hit-rate и latency.

Теоретическое описание: Теорема CAP утверждает, что распределённая система может одновременно гарантировать не более двух свойств: Consistency, Availability, Partition tolerance. Кэширование снижает нагрузку на БД, но требует стратегии инвалидации. Репликация (master-slave, multi-master, quorum) повышает доступность и read-масштабируемость, но вносит задержку согласования (replication lag). **Разобранный пример:** 1. Система: лента новостей социальной сети (1M read/sec, 10K write/sec).
2. Выбор: PostgreSQL (пользователи/метаданные), Redis (кэш ленты), Cassandra (история постов).
3. Паттерн: Cache-Aside с TTL 15 мин, Write-Behind для асинхронной записи.
4. Симуляция: сравнение latency с/без кэша при Zipf-распределении запросов. **Программный код (симуляция кэша и расчёт hit-rate):**

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(42)
N_REQUESTS = 10000
CACHE_SIZE = 500  #                                                    items


#                                                                        (
                    "               "              )
items = np.arange(1, 2000)
prob = 1 / (items**1.2)
prob /= prob.sum()
requests = np.random.choice(items, size=N_REQUESTS, p=prob)

#                LRU-
cache = set()
hits = 0
for req in requests:
    if req in cache:
        hits += 1
        cache.add(req)  #                                          (
                                )
    else:
        if len(cache) >= CACHE_SIZE:
            cache.pop()  #
        cache.add(req)

hit_rate = hits / N_REQUESTS
```

```python
print(f"                        : {CACHE_SIZE}")
print(f"Hit Rate: {hit_rate:.2%}")
print(f"
        : {hit_rate:.2%}")

#                                                    Hit Rate

cache_sizes = np.linspace(50, 1000, 20, dtype=int)
rates = []
for size in cache_sizes:
    cache_s = set()
    h = 0
    for req in requests:
        if req in cache_s:
            h += 1
        else:
            if len(cache_s) >= size:
                cache_s.pop()
            cache_s.add(req)
    rates.append(h / N_REQUESTS)

plt.plot(cache_sizes, rates, marker='o')
plt.xlabel('                        (                  )'); plt.ylabel(
   'Hit Rate')
plt.grid(True, alpha=0.3); plt.title('
                                                        ')
plt.show()
```

Индивидуальное задание: Спроектируйте архитектуру данных для системы онлайн-аукционов. Обоснуйте выбор СУБД, стратегию репликации и кэширования. Смоделируйте hit-rate при изменении распределения запросов. **Вопросы для самопроверки:**

1. Почему в высоконагруженных read-системах часто выбирают eventual consistency?
2. Как cache stampede возникает и какие паттерны его предотвращают?
3. В чём разница между шардированием и партиционированием данных?

7 Интеграционные паттерны: событийно-ориентированная архитектура и брокеры сообщений

Цель: Освоить механизмы асинхронной интеграции через брокеры сообщений и паттерны событийно-ориентированной архитектуры (EDA). **Задачи:**

- Изучить принципы EDA: события, продюсеры, консьюмеры, топологии pub/sub, point-to-point;
- Сравнить брокеры: RabbitMQ (очереди), Apache Kafka (логи событий), NATS (высокая пропускная способность);
- Реализовать паттерны: Event Sourcing, CQRS, Outbox, Dead Letter Queue;
- Смоделировать поток событий и анализ задержек доставки.

Теоретическое описание: EDA заменяет жёсткие вызовы на реакцию на события. Событие — неизменяемый факт произошедшего. Kafka хранит события в append-only логе, позволяя повторное чтение (replay). RabbitMQ использует роутинг через exchanges. Гарантии доставки: at-most-once, at-least-once, exactly-once. Паттерн Outbox решает проблему двойной записи (БД + брокер) в одной транзакции. **Разобранный пример:** 1. Процесс: оформление заказа → оплата → отгрузка → уведомление.
2. Архитектура: OrderService → Kafka Topic "orders.created"→ PaymentService, InventoryService.
3. Обработка: idempotency key, retry policy, DLQ для отравленных сообщений.
4. Симуляция: расчёт времени end-to-end при разной пропускной способности консьюмеров. **Программный код (моделирование EDA-потока):**

```
import time
import queue
import threading
import numpy as np

class EventBroker:
    def __init__(self):
        self.topics = {}
        self.lock = threading.Lock()

    def subscribe(self, topic, callback):
        with self.lock:
            self.topics.setdefault(topic, []).append(callback)

    def publish(self, topic, event):
        with self.lock:
            for cb in self.topics.get(topic, []):
                #

                cb(event)

#
processing_times = []
def payment_handler(event):
    time.sleep(np.random.exponential(0.05))  #

    processing_times.append(time.time() - event['ts'])

```

```
broker = EventBroker()
broker.subscribe("orders.created", payment_handler)

#
start = time.time()
for i in range(1000):
    broker.publish("orders.created", {"id": f"ord_{i}", "ts": time.
       time()})
    time.sleep(0.001)  # 1000 events/sec

print(f"                    : {len(processing_times)}              "
   )
print(f"                              : {np.mean(processing_times)
   *1000:.1f}     ")
print(f"P99                 : {np.percentile(processing_times, 99)
   *1000:.1f}     ")
```

Индивидуальное задание: Спроектируйте EDA-архитектуру для системы логистики доставки. Опишите события, топологии брокера, механизмы обработки ошибок и гарантированной доставки. **Вопросы для самопроверки:**

1. В чём разница между событием (event) и командой (command) в CQRS?
2. Почему Kafka предпочтительнее RabbitMQ для Event Sourcing?
3. Как Outbox Pattern предотвращает потерю событий при падении БД?

8 Архитектурные решения для облачных сред: контейнеризация и оркестрация

Цель: Научиться адаптировать архитектуру под облачные платформы, использовать контейнеризацию и оркестрацию для обеспечения масштабируемости и отказоустойчивости. **Задачи:**

- Изучить принципы Cloud-Native: двенадцатифакторные приложения, immutable infrastructure, declarative config;
- Освоить Docker: multi-stage builds, оптимизация образов, security best practices;
- Изучить Kubernetes: Pods, Deployments, Services, ConfigMaps, HPA, Health Checks;
- Реализовать генератор базовых манифестов K8s из описания сервисов.

Теоретическое описание: Облачная архитектура требует stateless-сервисов, внешнего конфигурирования, graceful shutdown и readiness/liveness проб. Kubernetes управляет жизненным циклом контейнеров, обеспечивает service discovery, load balancing и auto-scaling. Helm упрощает пакетирование. Service Mesh (Istio/Linkerd) добавляет observability, mTLS и traffic splitting на уровне инфраструктуры. **Разобранный пример:** 1. Приложение: веб-фронтенд + API-бэкенд + PostgreSQL.
2. Контейнеризация: multi-stage Dockerfile для frontend (Nginx), slim-образ для backend.
3. K8s: Deployment с resource limits, HPA по CPU 70%, Service type ClusterIP, ConfigMap для env.
4. Генерация YAML-манифестов программно. **Программный код (генератор K8s Deployment YAML):**

```
import yaml

def generate_k8s_deployment(name, image, replicas=2, cpu_req="100m",
    mem_req="128Mi", port=8080):
     spec = {
         "apiVersion": "apps/v1",
         "kind": "Deployment",
         "metadata": {"name": name},
         "spec": {
             "replicas": replicas,
             "selector": {"matchLabels": {"app": name}},
             "template": {
                 "metadata": {"labels": {"app": name}},
                 "spec": {
                     "containers": [{
                         "name": name,
                         "image": image,
                         "ports": [{"containerPort": port}],
                         "resources": {
                             "requests": {"cpu": cpu_req, "memory":
                                mem_req},
                             "limits": {"cpu": "500m", "memory": "512
                                Mi"}
                         },
                         "livenessProbe": {
                             "httpGet": {"path": "/health", "port":
                                port},
                             "initialDelaySeconds": 5, "periodSeconds"
```

```
                                    : 10
                            }
                        }]
                    }
                }
            }
        }
        return yaml.dump(spec, default_flow_style=False, sort_keys=False)

print("Kubernetes Deployment Manifest:\n")
print(generate_k8s_deployment("payment-service", "registry.company.io
    /payment:v1.2", replicas=3))
```

Индивидуальное задание: Спроектируйте облачную архитектуру для SaaS-плат аналитики. Опишите стратегию развёртывания, управление секретами, настройку HPA и план отката (rollback). **Вопросы для самопроверки:**

1. Почему liveness и readiness пробы должны проверять разные состояния приложения?
2. Как multi-stage builds снижают attack surface и размер образов?
3. В чём разница между StatefulSet и Deployment в Kubernetes?

9 Оценка качества архитектуры: методика ATAM и анализ рисков

Цель: Сформировать компетенции по системной оценке архитектурных решений, выявлению чувствительных точек и управлению рисками с использованием ATAM. **Задачи:**

- Изучить фазы ATAM: презентация, дерево полезности (Utility Tree), анализ подходов, выявление sensitivity/tradeoff points, риски;
- Научиться строить дерево полезности с оценкой приоритетов (High/Medium/Low);
- Выявлять архитектурные риски по матрице вероятность × воздействие;
- Реализовать расчёт риск-экспозиции и ранжирование mitigation-планов.

Теоретическое описание: ATAM (Architecture Tradeoff Analysis Method) — структурированный метод оценки архитектуры. Дерево полезности связывает бизнес-цели с архитектурными атрибутами (Performance, Modifiability, Security, Availability). Sensitivity point — параметр, сильно влияющий на атрибут. Tradeoff point — компромисс между атрибутами. Риск оценивается как $Risk = Probability \times Impact$. Стратегии: avoid, transfer, mitigate, accept. **Разобранный пример:** 1. Система: платформа онлайн-банкинга.
2. Сценарий качества: «Обработка 5000 транзакций/сек при latency < 200 мс» (Priority: High).
3. Tradeoff: кэширование повышает производительность, но снижает consistency.
4. Риск: «Единая точка отказа в платежном шлюзе» ($P = 0.3, I = 0.9 \rightarrow Risk = 0.27$).
5. Mitigation: активное резервирование + circuit breaker. **Программный код (расчёт риск-экспозиции ATAM):**

```
import pandas as pd
import matplotlib.pyplot as plt

risks = [
    {"id": "R1", "desc": "SPOF                                    ",
        "prob": 0.3, "impact": 0.9, "mitigation": "
                              "},
    {"id": "R2", "desc": "
       > 5  ", "prob": 0.5, "impact": 0.7, "mitigation": "

                    "},
    {"id": "R3", "desc": "
                                 OSS", "prob": 0.6, "impact": 0.8, "
       mitigation": "SCA-                                CI/CD"},
    {"id": "R4", "desc": "                                          rate-
       limit", "prob": 0.2, "impact": 0.6, "mitigation": "
                                  throttling"},
]

df = pd.DataFrame(risks)
df['risk_score'] = df['prob'] * df['impact']
df['level'] = df['risk_score'].apply(lambda x: '
                          ' if x>0.4 else ('              ' if x>0.2
   else '              '))
df = df.sort_values('risk_score', ascending=False)

print("                                                  (ATAM):\n
```

```
    ")
print(df[['id','desc','risk_score','level','mitigation']].to_string(
    index=False))

#
plt.scatter(df['prob']*10, df['impact']*10, s=df['risk_score']*500,
    alpha=0.7, c=df['risk_score'], cmap='Reds')
for i, row in df.iterrows():
    plt.annotate(row['id'], (row['prob']*10, row['impact']*10))
plt.xlabel('                          (%)'); plt.ylabel('
                        (%)')
plt.title('                                                    ');
    plt.grid(True, alpha=0.3)
plt.colorbar(label='Risk Score')
plt.tight_layout(); plt.show()
```

Индивидуальное задание: Проведите ATAM-анализ для архитектуры электронной коммерции. Постройте дерево полезности (минимум 6 сценариев), выявите 4 риска, рассчитуйте экспозицию и предложите mitigation-план. **Вопросы для самопроверки:**

1. Чем sensitivity point отличается от tradeoff point в контексте ATAM?
2. Как дерево полезности связывает бизнес-требования с техническими метриками?
3. Почему ATAM проводится до реализации, а не после?

Список литературы

[1] Bass L., Clements P., Kazman R. Software Architecture in Practice. 4th ed. Addison-Wesley Professional, 2021. 672 p. — ISBN 978-0-13-703066-8.

[2] Richardson C. Microservices Patterns: With examples in Java. Manning Publications, 2018. 520 p. — ISBN 978-1-61729-454-9.

[3] Фаулер М. Архитектура корпоративных программных приложений / пер. с англ. — 2-е изд. — Санкт-Петербург : Издательство «Питер», 2022. — 544 с. — ISBN 978-5-4461-2134-5.

[4] Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. O'Reilly Media, 2021. 616 p. — ISBN 978-1-492-03402-4.

[5] Burns B., Beda J. Kubernetes: Up and Running. 3rd ed. O'Reilly Media, 2022. 480 p. — ISBN 978-1-098-11029-9.

[6] ISO/IEC/IEEE 42010:2022. Systems and software engineering — Architecture description. Geneva : International Organization for Standardization, 2022. — 46 p.

[7] Kleppmann M. Designing Data-Intensive Applications. O'Reilly Media, 2017. 616 p. — ISBN 978-1-449-37332-0.

[8] Hohpe G., Woolf B. Enterprise Integration Patterns. Addison-Wesley, 2003. 736 p. — ISBN 978-0-321-20068-6.

[9] Virtanen P. et al. SciPy 1.0: fundamental algorithms for scientific computing in Python // Nature Methods. — 2020. — Vol. 17. — P. 261–272.

[10] FastAPI Documentation. URL: `https://fastapi.tiangolo.com/` (дата обращения: 01.06.2026). — Текст : электронный.