АО «АЛМАТИНСКИЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет инженерных и информационных технологий

Кафедра компьютерной инженерии

ПРОЕКТИРОВАНИЕ СИСТЕМ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

ЛЕКЦИЯ № 12

Развертывание AI-моделей и model serving

Теоретико-практический конспект

Составитель: Алтыбай А.

Кафедра компьютерной инженерии

Алматы, 2026

Аннотация и назначение лекции

Лекция 12 раскрывает тему «Развертывание AI-моделей и model serving» в контексте разработки промышленной AI-системы. Материал опирается на тематический план дисциплины и соединяет определения, принципы проектирования, математические оценки, проектные артефакты, практический код и аргументированный анализ рисков. Вопросы построены вокруг решений, которые предстоит принимать инженеру, а не только вокруг названий алгоритмов. [2, 3]

План занятия

1. Serving как отдельный продукт
2. REST API
3. gRPC и двоичные протоколы
4. Batch и online inference
5. Асинхронный inference
6. Контейнеризация
7. Конфигурация и секреты
8. Health и readiness
9. Практические сценарии, контроль качества и проектные задания.

Ожидаемые результаты обучения

По завершении студент способен:

1. объяснить и аргументированно применить принцип «Serving как отдельный продукт» к инженерному решению;
2. объяснить и аргументированно применить принцип «REST API» к инженерному решению;

3. объяснить и аргументированно применить принцип «gRPC и двоичные протоколы» к инженерному решению;
4. объяснить и аргументированно применить принцип «Batch и online inference» к инженерному решению;
5. объяснить и аргументированно применить принцип «Асинхронный inference» к инженерному решению;
6. объяснить и аргументированно применить принцип «Контейнеризация» к инженерному решению;
7. реализовать упрощённый проверяемый прототип и сформулировать критерии его приемки.
8. указать угрозы достоверности результатов и предложить проверки до эксплуатации.

Сквозной кейс. В течение лекции рассматриваем онлайн-сервис классификации текстов. Существенные условия: рост хвостовой задержки, плохое восстановление после отказа. Основная проверяемая характеристика: p95, 5xx rate и доступность. Числа в расчётных примерах учебные и не являются результатами реальных измерений.

1. Базовые теоретические положения из конспекта курса

Ниже представлены исходные положения учебно-методического конспекта, связанные с данной темой. Следующие разделы раскрывают их с инженерными деталями и проверяемыми примерами. [2, 3] Цель лекции. изучить способы публикации модели как сервиса, REST/gRPC API, batch/online inference, контейнеризацию и основные требования к deployment.

Ключевые понятия. API, REST, gRPC, FastAPI, serialization, Docker, container, batch serving, online serving, autoscaling, SLA, canary deployment.

1.1 Model serving

Model serving — процесс предоставления модели другим системам. Сервис принимает запрос, валидирует вход, выполняет preprocessing, вызывает модель, postprocessing и возвращает ответ. Важны стабильная схема API, версионирование и обработка ошибок.

Не следует смешивать heavy preprocessing, сетевые вызовы и модель без измерения: каждый этап может стать узким местом.

1.2 REST и gRPC

REST прост в интеграции и хорошо поддерживается веб-инфраструктурой. gRPC использует строгие protobuf-контракты и часто удобен для межсервисной коммуникации с низкими накладными расходами. Выбор зависит от экосистемы, требований к производительности и совместимости клиентов.

Для внешнего публичного API чаще выбирают REST; для внутренних сервисов возможен gRPC.

1.3 Batch и online serving

Batch serving обрабатывает набор объектов асинхронно и позволяет эффективнее использовать ресурсы. Online serving отвечает на запрос в реальном времени. В некоторых системах полезна гибридная схема: признаки пересчитываются batch, a final score — online.

Архитектура должна определить timeout, retries, fallback и поведение при недоступности модели.

1.4 Контейнеризация

Docker упаковывает приложение, зависимости и runtime в образ. Это уменьшает различия между средами разработчика, тестирования и

production. Контейнер не заменяет воспроизводимость модели: веса, tokenizer и конфигурация также должны версионироваться.

Образ следует делать минимальным, не включать секреты и запускать процесс с минимальными привилегиями.

1.5 Пример FastAPI endpoint

В production к примеру ниже добавляются schema validation, logging, auth, rate limiting, мониторинг и обработка исключений.

from fastapi import FastAPI from pydantic import BaseModel

app = FastAPI()

class Request(BaseModel): features: list[float]

@app.post(”/predict”) def predict(req: Request): y = model.predict([req.features])[0] return {”prediction”: float(y), ”model_version”: ”1.0”}

1.6 Выводы

▷ Model serving превращает модель в управляемый программный контракт.

▷ Batch и online deployment решают разные требования к свежести и latency.

▷ Контейнеризация улучшает переносимость, но не заменяет версионирование ML-артефактов.

▷ Production API должен иметь валидацию, наблюдаемость и стратегию отказа.

1.7 Контрольные вопросы

▷ Какие этапы выполняет типичный prediction service?

▷ Когда gRPC может быть предпочтительнее REST?

- В чем преимущества batch inference?
- Какие артефакты кроме Docker image необходимо версионировать?
- Что такое fallback в AI-сервисе?

2. Систематическое изложение темы и инженерные решения

2.1 Serving как отдельный продукт

Model serving преобразует файл обученной модели в надежный сервис с контрактом и ограничениями по времени. Нужны версии схем, обработка ошибок и health probes. [2]

> **Проверяемый пример.** Для классификатора установите /predict, /health и валидируемый тип запроса.

Вопрос для обсуждения. Как проверить выполнение такого требования: Для классификатора установите /predict, /health и валидируемый тип запроса.

2.2 REST API

HTTP/JSON удобен для публичного интерфейса, но сериализация и формат данных могут влиять на задержку. Входная валидация проводится до inference. [3]

> **Проверяемый пример.** Создайте Pydantic-модель с ограничениями на длину текста и поля категории.

Вопрос для обсуждения. Как проверить выполнение такого требования: Создайте Pydantic-модель с ограничениями на длину текста и поля категории.

2.3 gRPC и двоичные протоколы

gRPC использует контрактные схемы и часто эффективен для внутренних RPC-вызовов. Выбор оценивают по инфраструктуре и клиентским библиотекам. [4]

Проверяемый пример. Сопоставьте мобильный публичный REST API и внутренний RPC между сервисами.

Вопрос для обсуждения. Как проверить выполнение такого требования: Сопоставьте мобильный публичный REST API и внутренний RPC между сервисами.

2.4 Batch и online inference

Batch обеспечивает throughput за счет накопления работ; online ориентирован на индивидуальную задержку. Гибридная система может обслуживать оба режима. [5]

Проверяемый пример. Для миллиона ночных записей используйте пакетный job, а для одного клиента онлайн-эндпоинт.

Вопрос для обсуждения. Как проверить выполнение такого требования: Для миллиона ночных записей используйте пакетный job, а для одного клиента онлайн-эндпоинт.

2.5 Асинхронный inference

Длительные задачи отправляют в очередь, возвращают job_id и поддерживают проверку статуса. Отмена и истечение срока задачи определяются заранее. [6]

Проверяемый пример. Разработайте API загрузки видео и последующего получения анализа.

Вопрос для обсуждения. Как проверить выполнение такого требования: Разработайте API загрузки видео и последующего получения анализа.

2.6 Контейнеризация

Docker закрепляет системные зависимости, входную команду и окружение; он не заменяет тестирование безопасности образа. Не следует помещать секреты внутрь image. [1]

> **Проверяемый пример.** Опишите Dockerfile с непривилегированным пользователем и фиксированными зависимостями.

Вопрос для обсуждения. Как проверить выполнение такого требования: Опишите Dockerfile с непривилегированным пользователем и фиксированными зависимостями.

2.7 Конфигурация и секреты

Окружение разделяет код и deployment settings; ключи должны поступать из секретного хранилища. Различайте параметры модели и секреты API. [2]

> **Проверяемый пример.** Проверьте, что журнал не содержит token или API key.

Вопрос для обсуждения. Как проверить выполнение такого требования: Проверьте, что журнал не содержит token или API key.

2.8 Health и readiness

Liveness показывает, что процесс жив; readiness — что сервис готов принимать предсказания. Загрузка большой модели может занимать время после запуска контейнера. [3]

Проверяемый пример. Не принимайте запросы до окончания десериализации модели и прогрева.

Вопрос для обсуждения. Как проверить выполнение такого требования: Не принимайте запросы до окончания десериализации модели и прогрева.

2.9 Масштабирование

Горизонтальное масштабирование увеличивает число реплик, но предъявляет требования к памяти, concurrency и statefulness. Bottleneck может находиться не в модели. [4]

Проверяемый пример. Определите, какую часть latency создают preprocessing, сеть и вызов GPU.

Вопрос для обсуждения. Как проверить выполнение такого требования: Определите, какую часть latency создают preprocessing, сеть и вызов GPU.

2.10 Нагрузочное тестирование

p50/p95 latency, throughput, доля 5xx и использование памяти измеряются при заданном профиле запросов. Средние значения при нулевой нагрузке недостаточны. [5]

Проверяемый пример. Сравните производительность при 1, 20 и 100 конкурентных клиентах.

Вопрос для обсуждения. Как проверить выполнение такого требования: Сравните производительность при 1, 20 и 100 конкурентных клиентах.

2.11 Canary и rollback

Новую версию направляют ограниченной доле запросов, контролируют guardrails и откатывают при нарушении SLO. Маршрутизация должна со-

хранять совместимость схем. [6]

Проверяемый пример. Спланируйте 5% canary с автоматическим возвратом к v1 при росте 5xx.

Вопрос для обсуждения. Как проверить выполнение такого требования: Спланируйте 5% canary с автоматическим возвратом к v1 при росте 5xx.

2.12 Cloud/edge trade-offs

Облачное размещение упрощает масштабирование, но влияет на стоимость и юрисдикцию данных; edge снижает сетевую зависимость, но ограничен памятью. [1]

Проверяемый пример. Сравните классификацию видеопотока локально и в облаке при нестабильной сети.

Вопрос для обсуждения. Как проверить выполнение такого требования: Сравните классификацию видеопотока локально и в облаке при нестабильной сети.

3. Математические инструменты и количественные критерии

Для перехода от неформального решения к проверяемой постановке введём следующие величины.

$$Q = \frac{N_{\text{completed}}}{\Delta t} \tag{1}$$

$$\text{Utilization} \approx \frac{\lambda}{k\mu} \tag{2}$$

Если средняя интенсивность поступления λ приближается к суммарной скорости обработки $k\mu$, очередь может резко расти.

При выборе числового критерия нужно отличать обучающую функцию потерь, продуктовую ценность и эксплуатационные ограничения. Первую можно оптимизировать дифференцированием, вторая определяется процессом принятия решения, а третьи проверяются нагрузочными испытаниями, политиками безопасности и мониторингом.

3.1 Пошаговый расчёт на учебных данных

Рассмотрим две конфигурации одной системы. Вариант A имеет условную оценку качества $q_A = 0.86$ и задержку $t_A = 90$ мс; вариант B — $q_B = 0.89$ и $t_B = 280$ мс. Предположим, что требование заказчика задаёт $q \geq 0.85$ и $p95 \leq 150$ мс. Тогда A удовлетворяет обоим условиям, тогда как B нарушает второе. Следовательно, выбор B только потому, что его offline-метрика выше, не является обоснованным. Числа иллюстративны; реальный вывод требует измерений при одинаковой нагрузке и одинаковом наборе данных.

$$\text{accept}(A) = \mathbf{1}\{0.86 \geq 0.85 \wedge 90 \leq 150\} = 1,$$
$$\text{accept}(B) = \mathbf{1}\{0.89 \geq 0.85 \wedge 280 \leq 150\} = 0.$$

3.2 Как интерпретировать число, а не только вычислить его

Сначала следует проверить единицы измерения, состав выборки и смысл агрегирования. Например, p95 на спокойном стенде нельзя переносить на рабочую нагрузку с иным числом одновременных запросов. Затем исследуют чувствительность к изменениям входных данных и неопределённость оценок. Наконец, фиксируют последствия порога: какую ошибку он предотвращает, а какую может увеличить. Этот подход делает количественный критерий частью инженерной аргументации, а не украшением отчёта. [2, 3]

4. Практическое программирование на Python

Пример предназначен для демонстрации ключевого инженерного принципа. Он не претендует на обработку реального production-трафика. Запускать код следует в изолированном окружении и фиксировать версии библиотек, если они используются. Вывод зависит от указанных входных данных; для стохастических алгоритмов он может меняться.

4.1 Основной воспроизводимый пример

```python
from fastapi import FastAPI
from pydantic import BaseModel, Field

app = FastAPI()
class Input(BaseModel):
    value: float = Field(ge=0.0, le=100.0)

@app.get("/health")
def health():
    return {"status": "ok"}

@app.post("/predict")
def predict(body: Input):
    score = min(1.0, body.value / 100.0)  # baseline only
    return {"score": score, "model_version": "baseline-v1"}
# Run: uvicorn app:app --host 0.0.0.0 --port 8000
```

Проверка результатов. Сначала убедитесь, что код завершился без исключений; проверьте формы массивов, схему результата и отсутствие недопустимых значений. Если используются внешние данные, включите в отчёт способ их получения и идентификатор версии. Измените один входной параметр и объясните, почему наблюдаемый результат изменился или остался прежним.

4.2 Компактная проверка контракта или метрики

```
# Example HTTP test after starting the service:
# curl -X POST http://127.0.0.1:8000/predict \
#      -H "Content-Type: application/json" \
#      -d '{"value": 52}'
print("Verify HTTP status, JSON schema and latency")
```

При самостоятельном повторении предложите минимум один негативный тест: некорректное значение, пустой вход, несовместимую версию схемы либо нарушение SLO. Переход от успешного демонстрационного запуска к надежной системе требует именно таких проверок.

5. Разбор трёх инженерных ситуаций

5.1 Ситуация 1: Serving как отдельный продукт

Заказчик использует онлайн-сервис классификации текстов. Новое требование заставляет рассмотреть принцип «Serving как отдельный продукт». Основная неопределённость состоит в том, как одновременно обеспечить p95, 5xx rate и доступность, не допустить плохое восстановление после отказа и сохранить возможность изменения модели.

Постановка. Model serving преобразует файл обученной модели в надежный сервис с контрактом и ограничениями по времени. Нужны версии схем, обработка ошибок и health probes.

Действие инженера. Для классификатора установите /predict, /health и валидируемый тип запроса.

Альтернативы. Сравните три подхода: (A) простое правило без ML, (B) модульное ML-решение с отдельной проверкой входных данных и (C) интегрированный сервис со сложной моделью. Для каждого укажите потребность в данных, ожидаемые инженерные риски, возможность тестирования и минимальную инфраструктуру. Таблица ниже задаёт форму отчёта, а не готовый рейтинг.

Вариант	Данные и зависимости	Проверка	Основной риск
Правила	Фиксированные условия	Набор тест-кейсов	Ограниченная гибкость
ML-модуль	Версионированные признаки	Offline и API tests	Drift и skew
Сложный сервис	Несколько компонентов	End-to-end и нагрузка	Каскадный отказ

Проверка решения. Подготовьте две позитивные и две негативные пробы; задайте метрику успеха, базовый сценарий и критерий остановки внедрения. Обсудите, какие допущения останутся непроверенными даже после удачного теста.

5.2 Ситуация 2: Контейнеризация

Заказчик использует онлайн-сервис классификации текстов. Новое требование заставляет рассмотреть принцип «Контейнеризация». Основная неопределённость состоит в том, как одновременно обеспечить p95, 5xx rate и доступность, не допустить плохое восстановление после отказа и сохранить возможность изменения модели.

Постановка. Docker закрепляет системные зависимости, входную команду и окружение; он не заменяет тестирование безопасности образа. Не следует помещать секреты внутрь image.

Действие инженера. Опишите Dockerfile с непривилегированным пользователем и фиксированными зависимостями.

Альтернативы. Сравните три подхода: (A) простое правило без ML, (B) модульное ML-решение с отдельной проверкой входных данных и (C) интегрированный сервис со сложной моделью. Для каждого укажите потребность в данных, ожидаемые инженерные риски, возможность тестирования и минимальную инфраструктуру. Таблица ниже задаёт форму отчёта, а не готовый рейтинг.

Вариант	Данные и зависимости	Проверка	Основной риск
Правила	Фиксированные условия	Набор тест-кейсов	Ограниченная гибкость
ML-модуль	Версионированные признаки	Offline и API tests	Drift и skew
Сложный сервис	Несколько компонентов	End-to-end и нагрузка	Каскадный отказ

Проверка решения. Подготовьте две позитивные и две негативные пробы; задайте метрику успеха, базовый сценарий и критерий остановки внедрения. Обсудите, какие допущения останутся непроверенными даже после удачного теста.

5.3 Ситуация 3: Canary и rollback

Заказчик использует онлайн-сервис классификации текстов. Новое требование заставляет рассмотреть принцип «Canary и rollback». Основная неопределённость состоит в том, как одновременно обеспечить p95, 5xx rate и доступность, не допустить плохое восстановление после отказа и сохранить возможность изменения модели.

Постановка. Новую версию направляют ограниченной доле запросов, контролируют guardrails и откатывают при нарушении SLO. Маршрутизация должна сохранять совместимость схем.

Действие инженера. Спланируйте 5% canary с автоматическим возвратом к v1 при росте 5xx.

Альтернативы. Сравните три подхода: (A) простое правило без ML, (B) модульное ML-решение с отдельной проверкой входных данных и (C) интегрированный сервис со сложной моделью. Для каждого укажите потребность в данных, ожидаемые инженерные риски, возможность тестирования и минимальную инфраструктуру. Таблица ниже задаёт форму отчёта, а не готовый рейтинг.

Вариант	Данные и зависимости	Проверка	Основной риск
Правила	Фиксированные условия	Набор тест-кейсов	Ограниченная гибкость
ML-модуль	Версионированные признаки	Offline и API tests	Drift и skew
Сложный сервис	Несколько компонентов	End-to-end и нагрузка	Каскадный отказ

Проверка решения. Подготовьте две позитивные и две негативные пробы; задайте метрику успеха, базовый сценарий и критерий остановки внедрения. Обсудите, какие допущения останутся непроверенными даже после удачного теста.

6. Самостоятельная работа: индивидуальные проектные задания

Для каждого задания студент создаёт отдельный небольшой артефакт: схему, набор тестов, код либо таблицу сравнения, прикладывает наблюдаемый результат и объясняет ограничения.

1. **Serving как отдельный продукт.** Для классификатора установите /predict, /health и валидируемый тип запроса. Сформулируйте проверяемое требование, продемонстрируйте альтернативу и приложите протокол хотя бы одного эксперимента.

2. **REST API.** Создайте Pydantic-модель с ограничениями на длину текста и поля категории. Сформулируйте проверяемое требование, продемонстрируйте альтернативу и приложите протокол хотя бы одного эксперимента.

3. **gRPC и двоичные протоколы.** Сопоставьте мобильный публичный REST API и внутренний RPC между сервисами. Сформулируйте проверяемое требование, продемонстрируйте альтернативу и приложите протокол хотя бы одного эксперимента.

4. **Batch и online inference.** Для миллиона ночных записей используйте пакетный job, а для одного клиента онлайн-эндпоинт. Сформулируйте проверяемое требование, продемонстрируйте альтернативу и приложите протокол хотя бы одного эксперимента.

5. **Асинхронный inference.** Разработайте API загрузки видео и последующего получения анализа. Сформулируйте проверяемое требование, продемонстрируйте альтернативу и приложите протокол хотя бы одного эксперимента.

6. **Контейнеризация.** Опишите Dockerfile с непривилегированным пользователем и фиксированными зависимостями. Сформулируйте проверяемое требование, продемонстрируйте альтернативу и приложите протокол хотя бы одного эксперимента.

7. **Конфигурация и секреты.** Проверьте, что журнал не содержит token или API key. Сформулируйте проверяемое требование, продемонстри-

руйте альтернативу и приложите протокол хотя бы одного эксперимента.

8. **Health и readiness.** Не принимайте запросы до окончания десериализации модели и прогрева. Сформулируйте проверяемое требование, продемонстрируйте альтернативу и приложите протокол хотя бы одного эксперимента.

9. **Масштабирование.** Определите, какую часть latency создают preprocessing, сеть и вызов GPU. Сформулируйте проверяемое требование, продемонстрируйте альтернативу и приложите протокол хотя бы одного эксперимента.

10. **Нагрузочное тестирование.** Сравните производительность при 1, 20 и 100 конкурентных клиентах. Сформулируйте проверяемое требование, продемонстрируйте альтернативу и приложите протокол хотя бы одного эксперимента.

11. **Canary и rollback.** Спланируйте 5% canary с автоматическим возвратом к v1 при росте 5xx. Сформулируйте проверяемое требование, продемонстрируйте альтернативу и приложите протокол хотя бы одного эксперимента.

12. **Cloud/edge trade-offs.** Сравните классификацию видеопотока локально и в облаке при нестабильной сети. Сформулируйте проверяемое требование, продемонстрируйте альтернативу и приложите протокол хотя бы одного эксперимента.

7. Вопросы для текущего и итогового контроля

1. В чём назначение и ограничения принципа «Serving как отдельный продукт»?

2. Как проверить на практике утверждение: «Для классификатора установите /predict, /health и валидируемый тип запроса.»?

3. В чём назначение и ограничения принципа «REST API»?

4. Как проверить на практике утверждение: «Создайте Pydantic-модель с ограничениями на длину текста и поля категории.»?
5. В чём назначение и ограничения принципа «gRPC и двоичные протоколы»?
6. Как проверить на практике утверждение: «Сопоставьте мобильный публичный REST API и внутренний RPC между сервисами.»?
7. В чём назначение и ограничения принципа «Batch и online inference»?
8. Как проверить на практике утверждение: «Для миллиона ночных записей используйте пакетный job, а для одного клиента онлайн-эндпоинт.»?
9. В чём назначение и ограничения принципа «Асинхронный inference»?
10. Как проверить на практике утверждение: «Разработайте API загрузки видео и последующего получения анализа.»?
11. В чём назначение и ограничения принципа «Контейнеризация»?
12. Как проверить на практике утверждение: «Опишите Dockerfile с непривилегированным пользователем и фиксированными зависимостями.»?
13. В чём назначение и ограничения принципа «Конфигурация и секреты»?
14. Как проверить на практике утверждение: «Проверьте, что журнал не содержит token или API key.»?
15. В чём назначение и ограничения принципа «Health и readiness»?
16. Как проверить на практике утверждение: «Не принимайте запросы до окончания десериализации модели и прогрева.»?
17. В чём назначение и ограничения принципа «Масштабирование»?
18. Как проверить на практике утверждение: «Определите, какую часть latency создают preprocessing, сеть и вызов GPU.»?
19. В чём назначение и ограничения принципа «Нагрузочное тестирование»?

20. Как проверить на практике утверждение: «Сравните производительность при 1, 20 и 100 конкурентных клиентах.»?
21. В чём назначение и ограничения принципа «Canary и rollback»?
22. Как проверить на практике утверждение: «Спланируйте 5% canary с автоматическим возвратом к v1 при росте 5xx.»?
23. В чём назначение и ограничения принципа «Cloud/edge trade-offs»?
24. Как проверить на практике утверждение: «Сравните классификацию видеопотока локально и в облаке при нестабильной сети.»?
25. Почему высокое offline-качество не гарантирует надежность промышленной AI-системы?
26. Какие артефакты нужны для воспроизводимости и анализа инцидента?

8. Глоссарий ключевых понятий

Serving как отдельный продукт
Model serving преобразует файл обученной модели в надежный сервис с контрактом и ограничениями по времени.

REST API
HTTP/JSON удобен для публичного интерфейса, но сериализация и формат данных могут влиять на задержку.

gRPC и двоичные протоколы
gRPC использует контрактные схемы и часто эффективен для внутренних RPC-вызовов.

Batch и online inference
Batch обеспечивает throughput за счет накопления работ; online ориентирован на индивидуальную задержку.

Асинхронный inference
Длительные задачи отправляют в очередь, возвращают job_id и поддерживают проверку статуса.

Контейнеризация
Docker закрепляет системные зависимости, входную команду и окружение; он не заменяет тестирование безопасности образа.

Конфигурация и секреты
Окружение разделяет код и deployment settings; ключи должны поступать из секретного хранилища.

Health и readiness
Liveness показывает, что процесс жив; readiness — что сервис готов принимать предсказания.

Масштабирование
Горизонтальное масштабирование увеличивает число реплик, но предъявляет требования к памяти, concurrency и statefulness.

Нагрузочное тестирование
p50/p95 latency, throughput, доля 5xx и использование памяти измеряются при заданном профиле запросов.

Canary и rollback

Новую версию направляют ограниченной доле запросов, контролируют guardrails и откатывают при нарушении SLO.

Cloud/edge trade-offs

Облачное размещение упрощает масштабирование, но влияет на стоимость и юрисдикцию данных; edge снижает сетевую зависимость, но ограничен памятью.

9. Итоги и связь со следующей темой

Представленные методы имеют смысл как части целостного управляемого жизненного цикла. Ключевой результат инженерного проектирования состоит не в максимальном числе компонентов или сложности модели, а в убедительной связи требований, проектных решений, измерений и механизмов сопровождения. Дальнейшее изучение курса развивает эти связи на следующем этапе жизненного цикла.

Список литературы

[1] Huyen C. Designing Machine Learning Systems. O'Reilly Media, 2022. https://www.oreilly.com/library/view/designing-machine-learning/9781098107956/.

[2] FastAPI. Official documentation. https://fastapi.tiangolo.com/.

[3] Docker Inc. Docker documentation. https://docs.docker.com/.

[4] Kleppmann M. Designing Data-Intensive Applications. O'Reilly Media, 2017. https://dataintensive.net/.

[5] Breck E. et al. The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction. 2017. https://research.google/pubs/the-ml-test-score-a-rubric-for-ml-production-readiness-and-technical-debt-reduction/.

[6] Sculley D. et al. Hidden Technical Debt in Machine Learning Systems. NeurIPS, 2015. https://research.google/pubs/hidden-technical-debt-in-machine-learning-systems/.