Многофункциональный сервис на базе LLM

Контекст курса

Пройденные темы:

- Лекции 1-3: Основы LLM, PEFT, оценка качества
- Лекция 4: Ограничения LLM и способы их преодоления

Сегодня: От теории к практике — архитектура комплексных сервисов

Дальше: Агентные системы, дизайн решений, бизнес-метрики

Цели лекции

Архитектура

Понять архитектуру production LLM-сервиса

Интеграция

Научиться интегрировать Tool Use, RAG, Memory

Мониторинг

Освоить практики мониторинга и observability

Паттерны

Изучить реальные паттерны и антипаттерны

Мультимодальность

Понять принципы мультимодальной интеграции

От концепции к реализации

Лекция 4: Что можно добавить к LLM

- Tools, RAG, Memory, Multimodal

Лекция 5: Как это собрать воедино

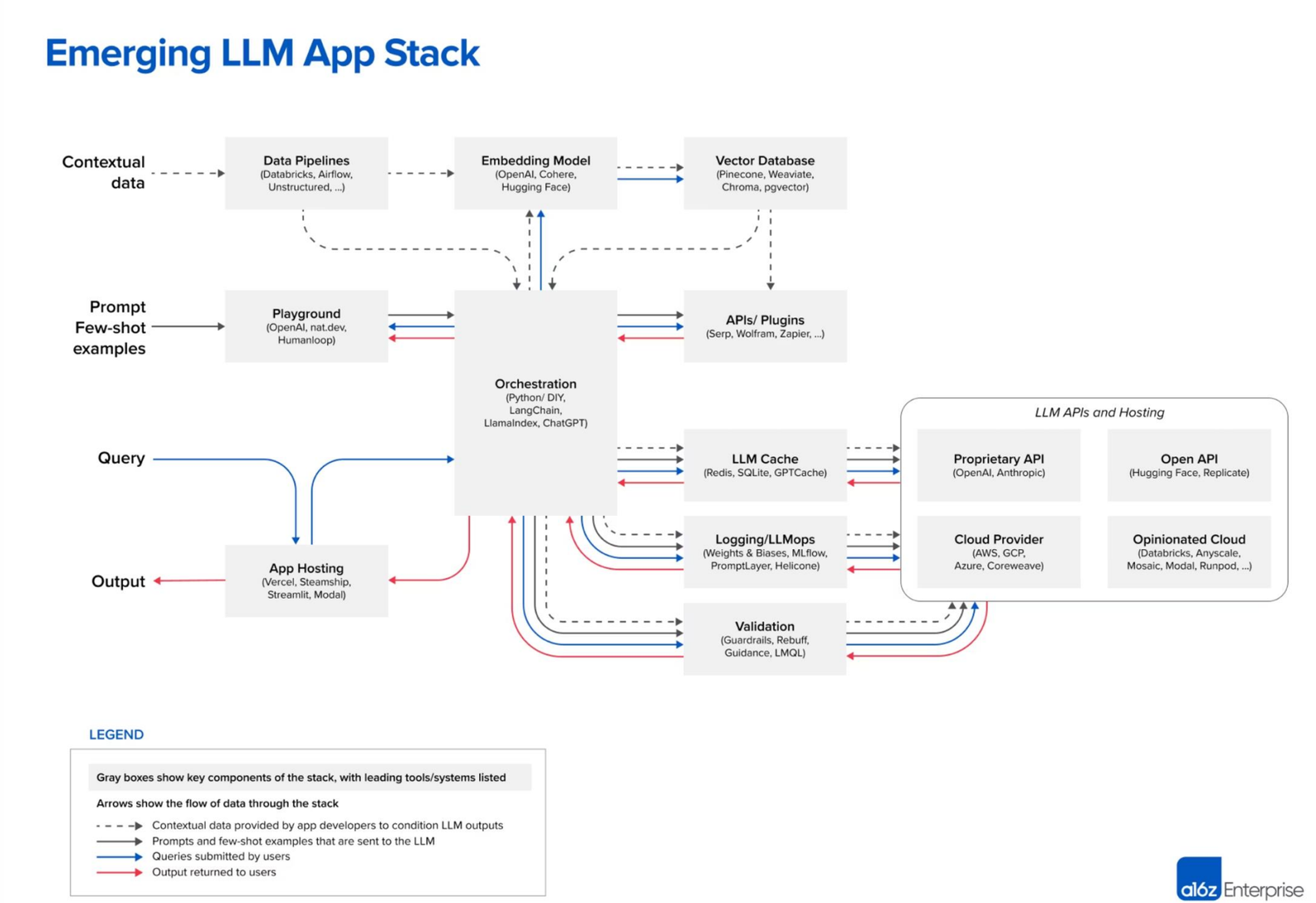
- Архитектура, интеграция, production-готовность

Не путать с:

- Дизайн бизнес-решений (лекция 8/9)
- Агентные системы (лекция 6/7)

Архитектура многофункционального сервиса

Базовая архитектура



Статья: Emerging Architectures for LLM Applications, 2023 | https://a16z.com/emerging-architectures-for-llm-applications/

Компоненты системы

Core:

- LLM Provider (OpenAI: https://openai.com/, Anthropic: https://anthropic.com/, self-hosted)
- Prompt Management
- Response Processing

Extensions:

- RAG System (векторная БД, embeddings)
- Tool Registry (доступные функции)
- Memory Store (контекст пользователя)

Infrastructure:

- Caching Layer
- Queue System
- Monitoring & Logging

Layered Architecture

Presentation Layer:

API endpoints, WebSocket, UI

Business Logic Layer:

- Оркестрация запросов
- Обработка ошибок
- Валидация

Data Access Layer:

- Векторные базы данных
- Реляционные БД для метаданных
- Кеш (Redis: https://redis.io/)

Integration Layer:

- LLM API clients
- External tools/APIs

Паттерны архитектуры

1. Pipeline Pattern

- Последовательная обработка
- Каждый шаг — отдельный компонент

2. Orchestrator Pattern

- Центральный координатор
- Динамический выбор компонентов

3. Event-Driven Pattern

- Асинхронная обработка
- Масштабируемость

Выбор архитектурного паттерна

Pipeline подходит для:

- Фиксированной последовательности операций
- Предсказуемых сценариев

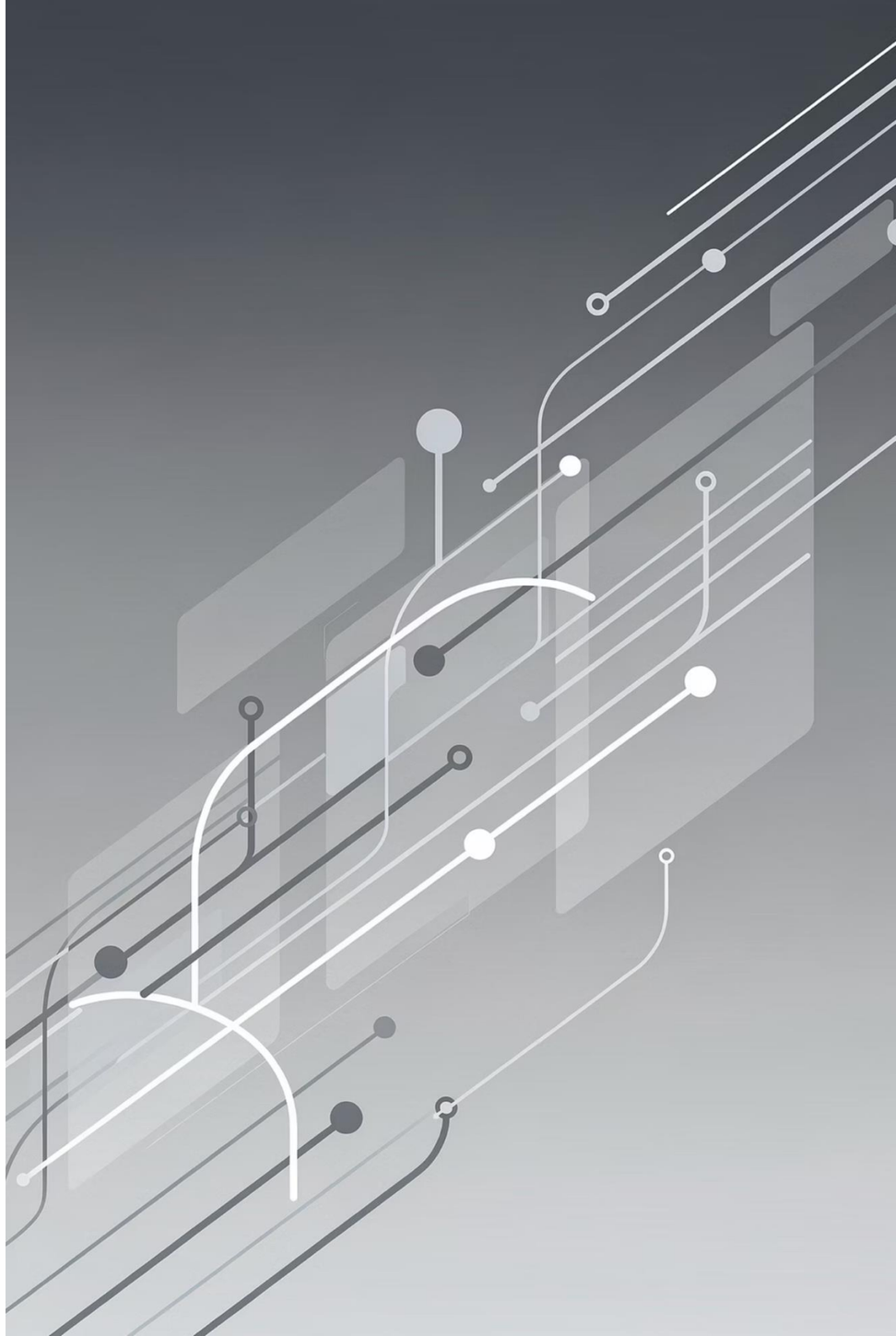
Orchestrator подходит для:

- Динамических сценариев
- Условной логики

Event-Driven подходит для:

- Высокой нагрузки
- Асинхронных операций

Практическая реализация RAG



RAG: от теории к практике

На лекции 4 мы узнали концепцию

Сегодня разберем:

01

Выбор векторной БД

02

Chunking стратегии

03

Embedding модели

04

Поиск и ранжирование

05

Интеграция в LLM pipeline

Выбор векторной базы данных

Open-source:

- **Chroma** — простота, Python-native
- **Weaviate** — production-ready, богатый функционал
- **Qdrant** — высокая производительность

Managed:

- **Pinecone** — полностью управляемый сервис
- **Zilliz** (Milvus Cloud) — масштабируемость

Гибридные:

- **pgvector** — расширение PostgreSQL

Критерии выбора: скорость, масштабируемость, стоимость, зрелость

Chunking стратегии

Fixed-size chunking:

- Фиксированное количество токенов (например, 512)
- Простота, но может разрывать контекст

Semantic chunking:

- Разбиение по смысловым границам (параграфы, секции)
- Лучше сохраняет контекст

Sliding window:

- Перекрывающиеся окна
- Предотвращает потерю информации на границах

Рекомендация: начните с semantic chunking + overlap 10-20%

Embedding модели

Важно:

- Используйте одну модель для индексации и поиска
- Учитывайте размерность (влияет на хранение)
- Для многоязычных задач выбирайте multilingual модели

Популярные модели:

- **OpenAI text-embedding-3-large** — 3072 dimensions https://platform.openai.com/docs/guides/embeddings
- **Cohere embed-v3** — многоязычная https://cohere.com/embeddings
- **sentence-transformers** — open-source (e.g., all-MiniLM-L6-v2) https://www.sbert.net/

Summary | Performance per Model Size | Performance per Task Type | Performance per task | Performance per language | Task information

Filter...

Rank (Bo…	Model	Zero-shot	Memory Us…	Number of P…	Embedding D…	Max Tokens	Mean (T…	Mean (TaskT…	Bitext …	Classification	Clustering
1	KaLM-Embedding-Gemma3-12B-2511	73%	44884	11.8	3840	32768	**72.32**	**62.51**	**83.76**	**77.88**	55.77
2	llama-embed-nemotron-8b	99%	28629	7.5	4096	32768	69.46	61.09	81.72	73.21	54.35
3	Qwen3-Embedding-8B	99%	14433	7.6	4096	32768	70.58	61.69	80.89	74.00	**57.65**
4	gemini-embedding-001	99%			3072	2048	68.37	59.59	79.28	71.82	54.59
5	Qwen3-Embedding-4B	99%	7671	4.0	2560	32768	69.45	60.86	79.36	72.33	57.15
6	Octen-Embedding-8B	99%	14433	7.6	4096	32768	67.84	60.28	80.35	66.68	55.68
7	Seed1.6-embedding-1215	89%			2048	32768	70.26	61.34	78.68	76.75	56.78

Benchmark: MTEB (Massive Text Embedding Benchmark) https://huggingface.co/spaces/mteb/leaderboard

Поиск: от простого к сложному

Базовый поиск:

- Cosine similarity по embeddings
- Топ-K документов

Гибридный поиск:

- Векторный поиск + BM25 (keyword search)
- Комбинация результатов (Reciprocal Rank Fusion)

Reranking:

- Дополнительная модель для переранжирования
- Cohere Rerank: https://cohere.com/rerank
- Cross-encoders: https://www.sbert.net/examples/applications/cross-encoder/README.html

RAG Pipeline: шаг за шагом

```
# Псевдокод
def rag_pipeline(query, top_k=3):
 # 1. Генерация embedding для запроса
 query_embedding = embedding_model.encode(query)

 # 2. Поиск в векторной БД
 results = vector_db.search(
 query_embedding,
 top_k=top_k
 )

 # 3. Reranking (опционально)
 reranked = rerank_model.rank(query, results)

 # 4. Формирование контекста
 context = "\n\n".join([doc.content for doc in reranked])

 # 5. Формирование промпта
 prompt = f"Context: {context}\n\nQuestion: {query}"

 # 6. Запрос к LLM
 response = llm.complete(prompt)

 return response, reranked # возвращаем источники
```

Метаданные и фильтрация

Добавляйте метаданные:

```
{
  "content": "...",
  "metadata": {
    "source": "document_name.pdf",
    "page": 42,
    "section": "Introduction",
    "timestamp": "2024-01-15",
    "author": "John Doe"
  }
}
```

Фильтрация перед поиском:

- По дате: только свежие документы
- По автору: только определенные источники
- По типу: только PDF или только внутренние docs

Проблемы RAG и решения

1

Проблема 1: Не находит релевантную информацию

Решение: улучшить chunking, попробовать гибридный поиск

2

Проблема 2: Слишком много нерелевантного контекста

Решение: reranking, фильтрация по threshold similarity

3

Проблема 3: Информация распределена по документам

Решение: увеличить top_k, multi-hop retrieval

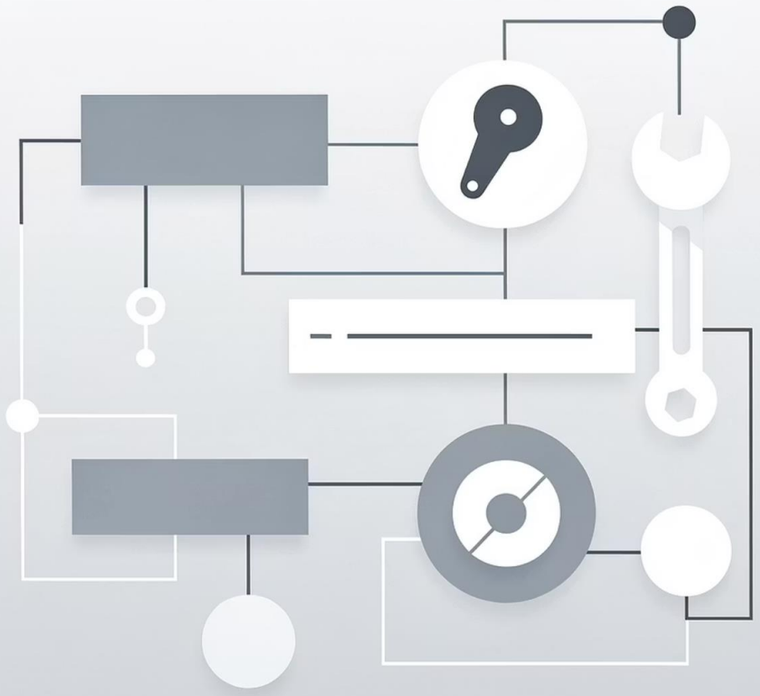
4

Проблема 4: Устаревшие данные

Решение: периодическая переиндексация, timestamp фильтрация

RAG в продакшене: checklist

- Стратегия переиндексации (incremental vs full)
- Версионирование embedding модели
- Мониторинг quality (relevance metrics)
- Fallback при недоступности векторной БД
- Кеширование частых запросов
- Логирование источников для citation



Интеграция Tool Use в production

От примера к production

На лекции 4:

простой калькулятор в ноутбуке

В продакшене нужно:

- Безопасность
- Надежность
- Управление ошибками
- Мониторинг
- Версионирование

Tool Registry Pattern

```python
class ToolRegistry:
    def __init__(self):
        self.tools = {}

    def register(self, name, func, schema):
        """Регистрация инструмента"""
        self.tools[name] = {
            "function": func,
            "schema": schema,
            "version": "1.0"
        }

    def get_schemas(self):
        """Получить схемы для LLM"""
        return [tool["schema"] for tool in self.tools.values()]

    def execute(self, name, args):
        """Безопасное выполнение"""
        if name not in self.tools:
            raise ToolNotFoundError(name)

        tool = self.tools[name]
        return tool["function"](**args)
```

Безопасность Tool Use

Принципы:

01

Whitelist подход — только разрешенные функции

02

Sandbox выполнение — изоляция для опасных операций

03

Rate limiting — ограничение вызовов

04

Валидация параметров — проверка входных данных

05

Логирование — запись всех вызовов

Никогда не позволяйте:

- Прямое выполнение кода (eval, exec)
- Доступ к файловой системе без ограничений
- SQL-инъекции через параметры

Error Handling для Tools

```python
def safe_tool_execution(tool_name, args):
    try:
        result = tool_registry.execute(tool_name, args)
        return {"status": "success", "data": result}

    except ToolNotFoundError:
        return {"status": "error", "message": "Tool not found"}

    except ValidationError as e:
        return {"status": "error", "message": f"Invalid args: {e}"}

    except TimeoutError:
        return {"status": "error", "message": "Tool execution timeout"}

    except Exception as e:
        logger.error(f"Tool {tool_name} failed: {e}")
        return {"status": "error", "message": "Internal error"}
```

Асинхронные инструменты

Для долгих операций:

```python
# Паттерн с callback
async def long_running_tool(params, callback_url):
    task_id = generate_id()

    # Запускаем задачу в фоне
    queue.enqueue(execute_task, task_id, params)

    # Немедленно возвращаем task_id
    return {
        "status": "pending",
        "task_id": task_id,
        "callback_url": callback_url
    }

# LLM получит результат позже через callback
```

Мониторинг Tool Use

Метрики:

- Частота вызовов каждого инструмента
- Success rate по инструментам
- Latency выполнения
- Частота ошибок

Алерты:

- Инструмент падает >5% запросов
- Latency превышает SLA
- Новый инструмент не используется (возможно, плохое описание)

Мультимодальность в сервисе

Интеграция Vision моделей

Использование:

- Анализ документов (OCR + понимание)
- Понимание диаграмм и графиков
- Обработка скриншотов

Модели:

- GPT-4 Vision: https://platform.openai.com/docs/guides/vision
- Claude 3.5 Sonnet: https://www.anthropic.com/claude
- Gemini Pro Vision: https://ai.google.dev/gemini-api

Паттерн:

Image → Base64 encoding → LLM API → Text response

Обработка изображений: best practices

1

Preprocessing:

- Resize больших изображений (экономия токенов)
- Конвертация в supported форматы

2

Промпты для vision:

- Четкие инструкции: "Опиши текст на изображении"
- Structured output: "Верни JSON с полями..."

3

Кеширование:

- Image hash → response cache
- Экономия на повторных запросах

Audio интеграция

Speech-to-Text:

- Whisper (OpenAI): https://openai.com/research/whisper
- Speech-to-Text API (Google, Azure)

Pipeline:

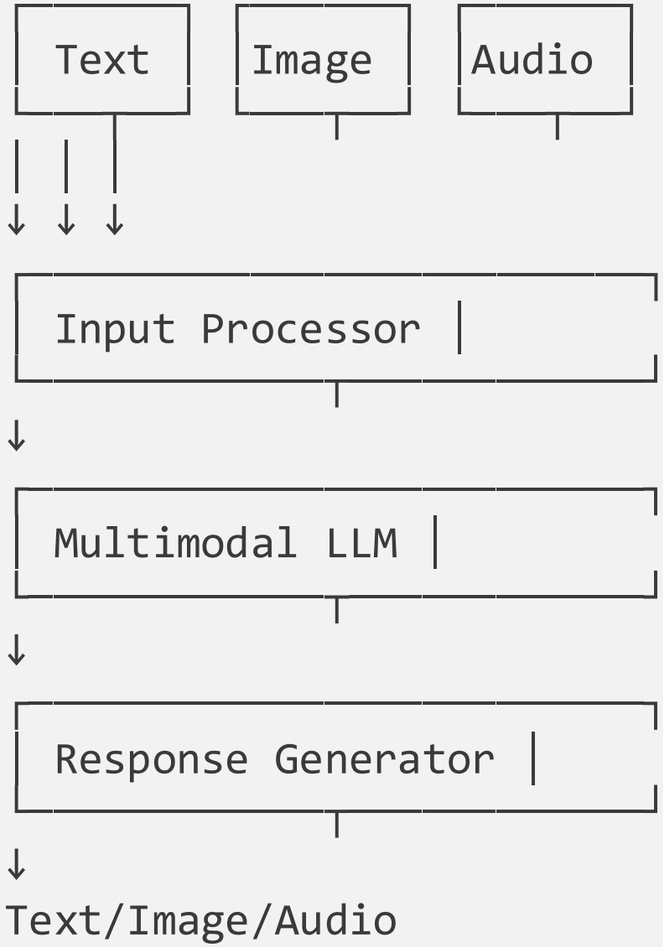
Audio → STT → LLM → TTS → Audio response

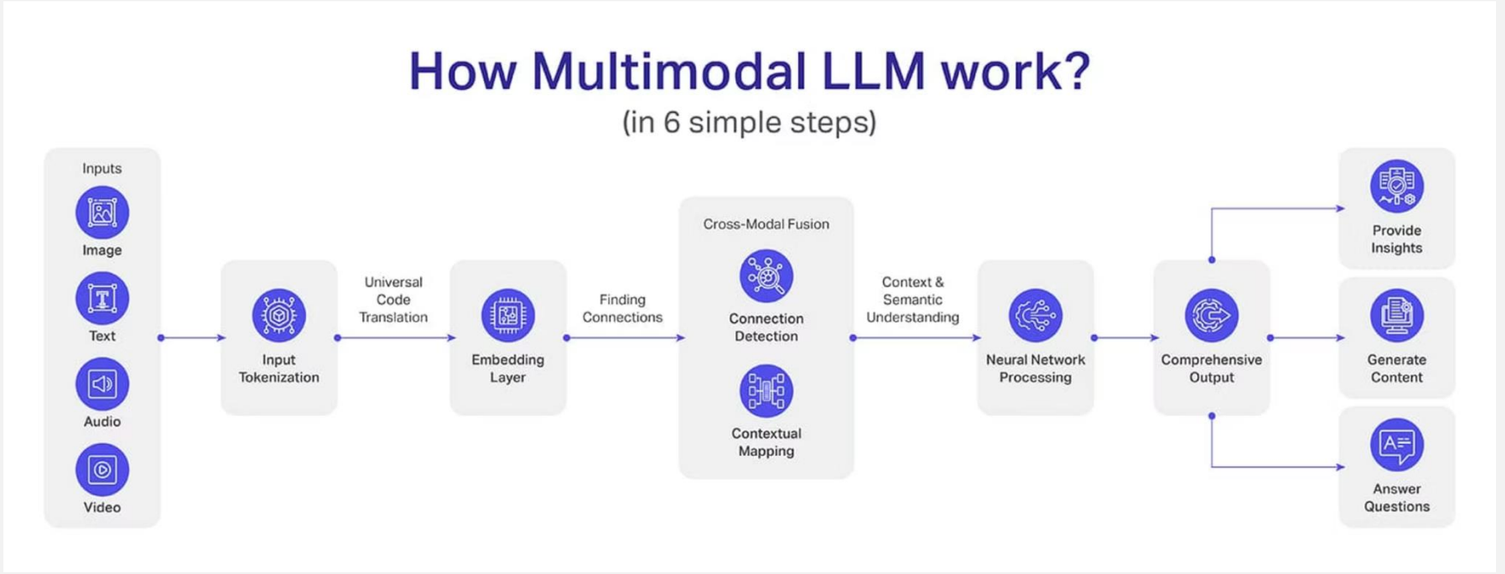
Use case: голосовые ассистенты, транскрипция встреч

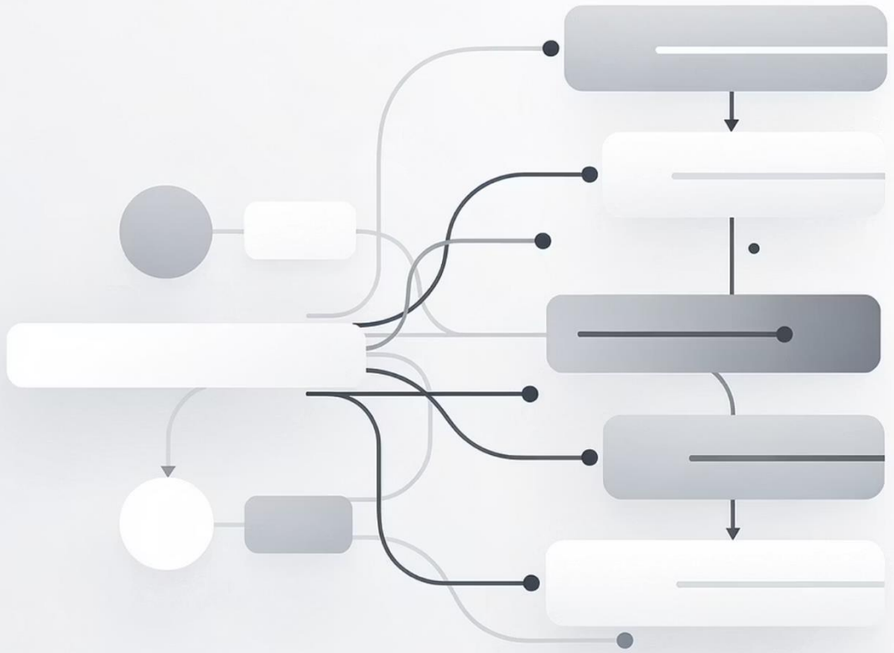
Text-to-Speech:

- OpenAI TTS: https://platform.openai.com/docs/guides/text-to-speech
- ElevenLabs: https://elevenlabs.io/ (качественная генерация)

Мультимодальный workflow







Оркестрация и управление состоянием

Оркестратор: центральная логика

Задачи оркестратора:

1. Маршрутизация запроса
2. Вызов нужных компонентов (RAG, Tools)
3. Управление контекстом
4. Сборка финального ответа

```python
class Orchestrator:
    def __init__(self, llm, rag, tools, memory):
        self.llm = llm
        self.rag = rag
        self.tools = tools
        self.memory = memory

    async def process(self, user_id, query):
        # 1. Загрузка контекста
        context = self.memory.get(user_id)

        # 2. Определение, нужен ли RAG
        if self.needs_rag(query):
            docs = await self.rag.search(query)
            context.add_documents(docs)

        # 3. Запрос к LLM с доступными tools
        response = await self.llm.complete(
            query,
            context=context,
            tools=self.tools.get_schemas()
        )

        # 4. Выполнение tools при необходимости
        if response.tool_calls:
            results = await self.execute_tools(response.tool_calls)
            # Повторный запрос с результатами
            response = await self.llm.complete(
                query,
                context=context,
                tool_results=results
            )

        # 5. Сохранение в память
        self.memory.add(user_id, query, response)

        return response
```

Управление контекстом

Проблема: context window ограничен

```python
class ContextManager:
    def __init__(self, max_tokens=8000):
        self.max_tokens = max_tokens

    def build_context(self, history, system_prompt, current_query):
        context = [system_prompt]
        tokens_used = count_tokens(system_prompt)

        # Добавляем историю с конца
        for message in reversed(history):
            msg_tokens = count_tokens(message)
            if tokens_used + msg_tokens > self.max_tokens:
                # Не влезает — суммируем оставшуюся историю
                summary = self.summarize(history[:-len(context)])
                context.insert(1, summary)
                break
            context.insert(1, message)
            tokens_used += msg_tokens

        context.append(current_query)
        return context
```

Стратегии:

1. **Sliding window** — сохраняем последние N сообщений
2. **Summarization** — суммируем старую историю
3. **Selective retention** — сохраняем только важное
4. **Hybrid approach**

State Management

Stateless сервис:

- Каждый запрос независим
- Контекст в внешнем хранилище (Redis: https://redis.io/, DB)

Stateful сервис:

- WebSocket соединения
- Сессия живет в памяти сервера
- Сложнее масштабировать

Рекомендация: Stateless + Redis для сессий

Caching стратегии

Уровни кеширования:

1

Prompt-level cache:

- Идентичные промпты → кешированный ответ
- Работает для FAQ

2

Embedding cache:

- Кеширование векторов документов
- Не пересчитываем при каждом запросе

3

Tool result cache:

- Детерминированные функции
- Кеш по параметрам

4

LLM response cache:

- Некоторые провайдеры поддерживают prompt caching
- Anthropic Prompt Caching: https://docs.anthropic.com/claude/docs/prompt-caching

Monitoring и Observability

Что мониторить

Performance:

- Latency (p50, p95, p99)
- Throughput (requests/sec)
- Error rate

LLM specific:

- Token usage (input/output)
- Cost per request
- Response quality (user feedback)

Components:

- RAG retrieval quality
- Tool execution success rate
- Cache hit rate

Структурированное логирование

```
import structlog

logger = structlog.get_logger()

# При каждом запросе
logger.info(
 "llm_request",
 user_id=user_id,
 query=query[:100], # первые 100 символов
 model="gpt-4",
 tokens_in=input_tokens,
 tokens_out=output_tokens,
 latency_ms=latency,
 cost_usd=cost,
 rag_docs_count=len(retrieved_docs),
 tools_used=[tool.name for tool in tools_called],
 success=True
)
```

Tracing с OpenTelemetry

Распределенный tracing:

```
Request [span: request_id]
├─ RAG Search [span: rag_123]
│  ├─ Embedding [span: embed_456]
│  └─ Vector Search [span: search_789]
├─ LLM Call [span: llm_abc]
└─ Tool Execution [span: tool_def]
   └─ External API [span: api_ghi]
```

Позволяет:

- Найти bottleneck
- Понять, где упал запрос
- Оптимизировать медленные компоненты

Мониторинг качества

Автоматические метрики:

- Response length (слишком короткие?)
- Sentiment (негативные ответы?)
- Citation rate (RAG использован?)

Пользовательские метрики:

- Thumbs up/down
- Follow-up questions (индикатор неполного ответа)
- Conversation length

A/B тесты:

- Сравнение промптов
- Сравнение моделей
- Сравнение RAG стратегий

Observability stack

Популярные инструменты:

Logging:

- ELK Stack: https://www.elastic.co/elastic-stack
- Loki (от Grafana): https://grafana.com/oss/loki

Tracing:

- Jaeger: https://www.jaegertracing.io
- Zipkin: https://zipkin.io/
- LangSmith: https://www.langchain.com/langsmith (LLM-специфичный)

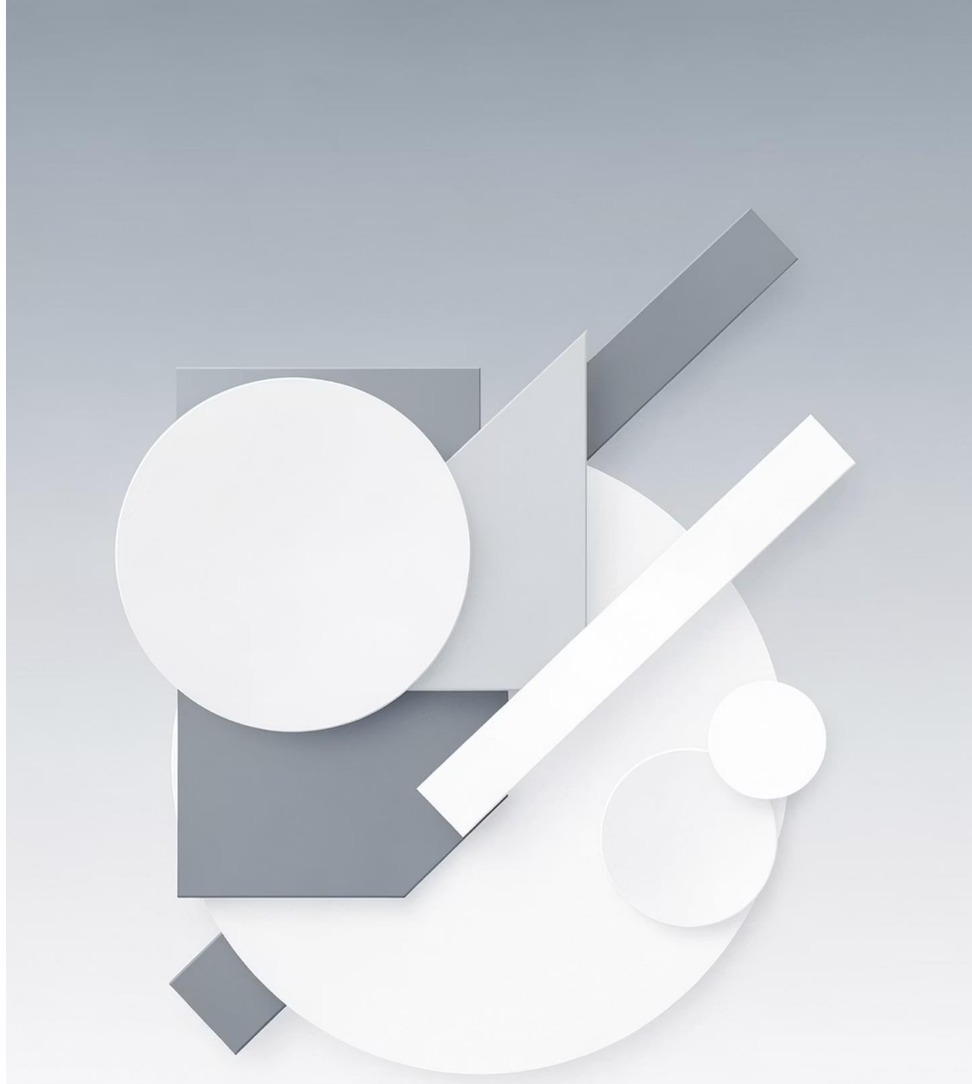
Metrics:

- Prometheus: https://prometheus.io/
- Grafana: https://grafana.com/
- Datadog: https://www.datadoghq.com

LLM-specific:

- Weights & Biases: https://wandb.ai/
- Phoenix (Arize AI): https://phoenix.arize.com/
- Helicone: https://www.helicone.ai/

Best Practices и антипаттерны



Best Practices

1 Fail gracefully

- Всегда имейте fallback
- LLM недоступна → вернуть кешированный/дефолтный ответ

2 Validate inputs

- Проверяйте длину промпта
- Санитизируйте пользовательский ввод

3 Optimize costs

- Используйте кеш
- Выбирайте модель под задачу (не всегда нужна GPT-4)

4 Monitor everything

- Логируйте все запросы
- Настройте алерты

5 Version control

- Версионируйте промпты
- Версионируйте схемы инструментов

Антипаттерны

Отсутствие rate limiting

Результат: взрыв costs или DDoS

Игнорирование ошибок LLM

Результат: пустые ответы пользователям

Хранение промптов в коде

Результат: невозможно A/B тестить

Отсутствие таймаутов

Результат: зависшие запросы

Один промпт для всех сценариев

Результат: низкое качество

Паттерн: Prompt Templates

```python
from jinja2 import Template

class PromptManager:
    def __init__(self):
        self.templates = {
            "summarization": Template("""
                Summarize the following text in {{ max_words }} words:

                {{ text }}
            """),
            "qa_with_context": Template("""
                Context: {{ context }}

                Question: {{ question }}

                Answer based on the context above.
            """)
        }

    def render(self, template_name, **kwargs):
        return self.templates[template_name].render(**kwargs)

# Использование
prompt = prompt_manager.render(
    "qa_with_context",
    context=rag_docs,
    question=user_query
)
```

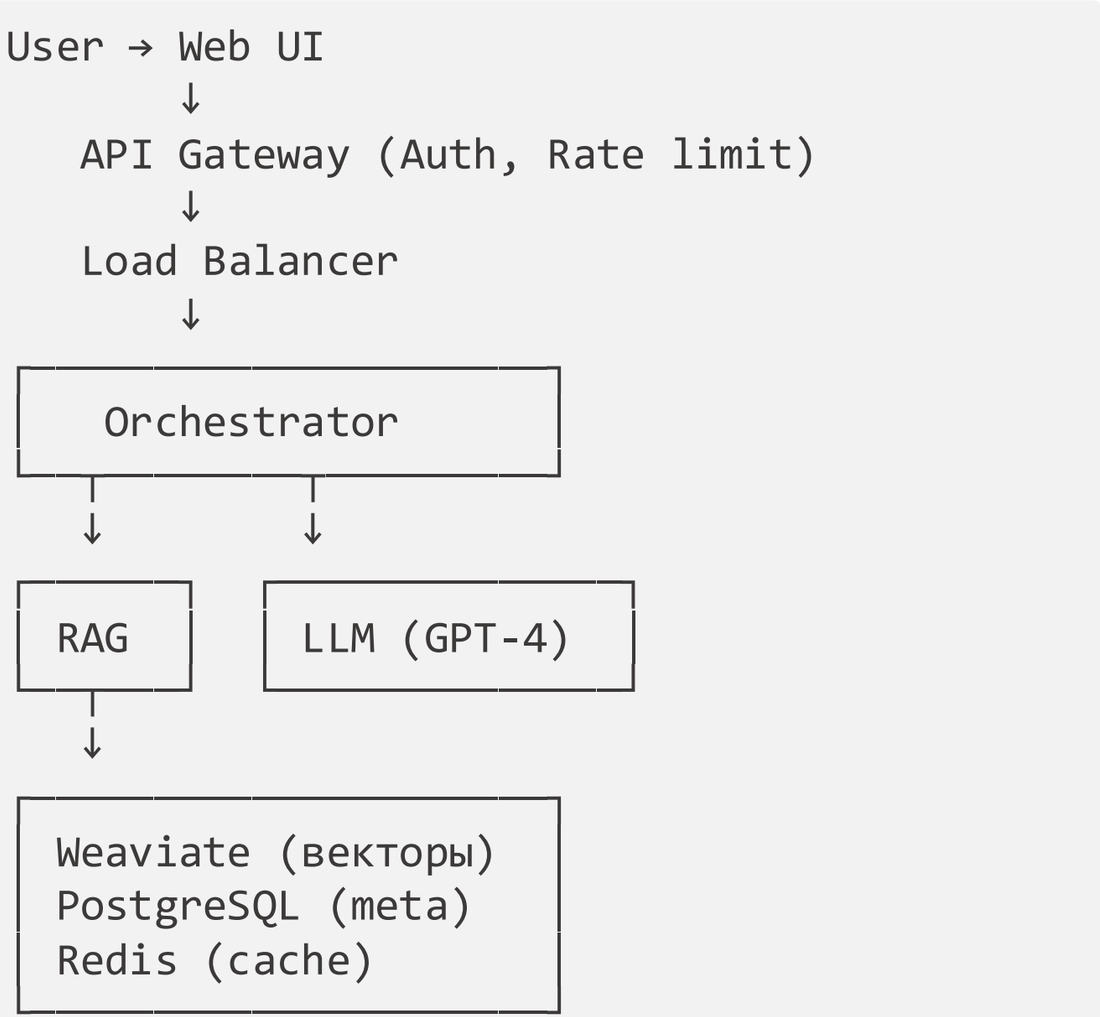
Паттерн: Circuit Breaker

Защита от каскадных сбоев:

```python
class CircuitBreaker:
    def __init__(self, failure_threshold=5, timeout=60):
        self.failure_count = 0
        self.failure_threshold = failure_threshold
        self.timeout = timeout
        self.last_failure = None
        self.state = "CLOSED"  # CLOSED, OPEN, HALF_OPEN

    def call(self, func, *args, **kwargs):
        if self.state == "OPEN":
            if time.time() - self.last_failure > self.timeout:
                self.state = "HALF_OPEN"
            else:
                raise ServiceUnavailableError("Circuit breaker is OPEN")

        try:
            result = func(*args, **kwargs)
            self.on_success()
            return result
        except Exception as e:
            self.on_failure()
            raise e

    def on_failure(self):
        self.failure_count += 1
        self.last_failure = time.time()
        if self.failure_count >= self.failure_threshold:
            self.state = "OPEN"

    def on_success(self):
        self.failure_count = 0
        self.state = "CLOSED"
```

Примеры реальных архитектур

Архитектура 1: Корпоративный Q&A бот



Особенности:

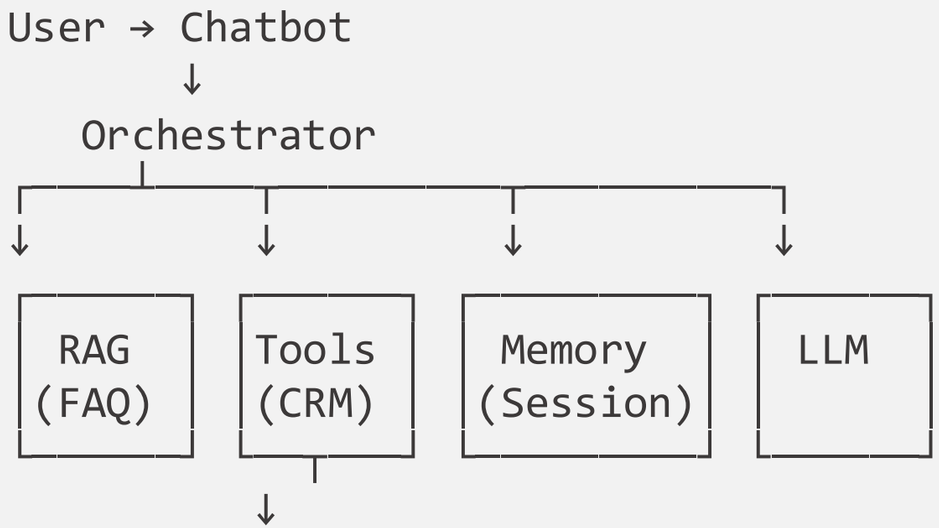
- Multi-tenant (разные компании)
- RAG по внутренним документам
- Citation источников обязательна

Используемые технологии:

- Weaviate: https://weaviate.io/
- PostgreSQL: https://www.postgresql.org/
- Redis: https://redis.io/

Пример: GigaChat B2B продукт "Сводка информации по крупному корпусу документов" https://b2b.giga.chat/cases#shortSvodkald

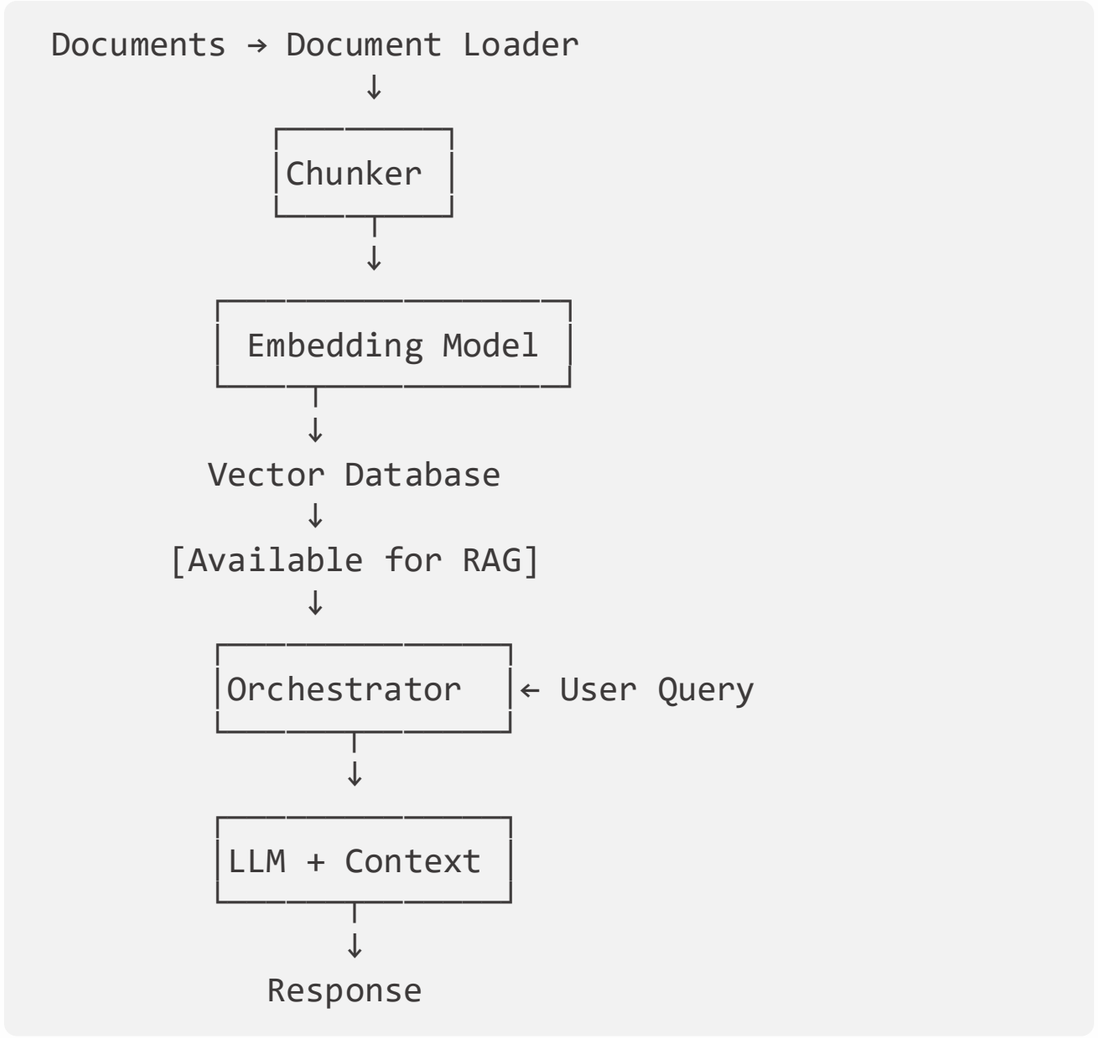
Архитектура 2: Customer Support Автоматизация



Особенности:

- Tool Use для действий (создание тикетов)
- Эскалация к человеку при неуверенности
- Мультиязычность

Архитектура 3: Document Processing Pipeline



Особенности:

- Batch processing документов
- Incremental updates (только новые docs)
- Мультимодальность (PDF с изображениями)

Пример: GigaChat B2B продукт "Личный секретарь по разбору почты" https://b2b.giga.chat/cases#PersonalHelpId

Паттерн: Hybrid Retrieval

```
def hybrid_search(query, top_k=5):
    # Векторный поиск
    vector_results = vector_db.search(
        embedding_model.encode(query),
        top_k=top_k * 2  # берем больше для фильтрации
    )

    # Keyword поиск (BM25)
    # Elasticsearch: https://www.elastic.co/
    keyword_results = elasticsearch.search(
        query=query,
        top_k=top_k * 2
    )

    # Reciprocal Rank Fusion
    combined = reciprocal_rank_fusion(
        vector_results,
        keyword_results
    )

    # Reranking
    reranked = cross_encoder.rank(query, combined)

    return reranked[:top_k]
```

Статья: Hybrid Search Explained Weaviate, 2023 | https://weaviate.io/blog/hybrid-search-explained

Заключение

Путь от идеи до продакшена

Сегодня мы прошли этот путь

Ключевые выводы

1. Архитектура важна — продумайте компоненты заранее
2. RAG требует практики — chunking, embeddings, reranking
3. Tool Use в продакшене ≠ Tool Use в примере
4. Мониторинг — не опция, а необходимость
5. Начинайте просто, усложняйте по необходимости
6. Всегда имейте fallback
7. Версионируйте промпты и компоненты

Связь с другими лекциями

1

Лекции 1-4:

Теоретическая база → Теперь знаем, как применить на практике

2

Следующая лекция:

Агентные системы → Автономные многошаговые решения

3

Дальше:

Дизайн решений, бизнес-метрики → Как выбрать, что строить и как измерять успех

Ресурсы для углубления

Статьи и блоги:

- Building LLM Applications for Production https://huyenchip.com/2023/04/11/llm-engineering.html
- Patterns for Building LLM-based Systems https://eugeneyan.com/writing/llm-patterns/
- RAG Best Practices (LlamaIndex Blog) https://blog.llamaindex.ai/

Фреймворки:

- LangChain: https://www.langchain.com/
- LangGraph: https://langchain-ai.github.io/langgraph/
- LlamaIndex: https://www.llamaindex.ai/
- Haystack: https://haystack.deepset.ai/

Векторные базы данных:

- Chroma: https://www.trychroma.com/
- Weaviate: https://weaviate.io/
- Qdrant: https://qdrant.tech/
- Pinecone: https://www.pinecone.io/
- pgvector: https://github.com/pgvector/pgvector

Embedding модели:

- OpenAI Embeddings: https://platform.openai.com/docs/guides/embeddings
- Sentence Transformers: https://www.sbert.net/
- MTEB Leaderboard: https://huggingface.co/spaces/mteb/leaderboard

Observability и мониторинг:

- LangSmith: https://www.langchain.com/langsmith
- Phoenix (Arize AI): https://phoenix.arize.com/
- Helicone: https://www.helicone.ai/
- Weights & Biases: https://wandb.ai/

Книги и курсы:

- "Building Production-Ready LLM Applications" (Databricks)
- "LLM Engineering" (Andrew Ng, DeepLearning.AI)

Следующая лекция будет про агентов

Вопросы: Чат курса