

M00. Старт: окружение, модели и инструменты

Теория модуля

Содержание

1	Зачем этот модуль	2
2	Кто такой AI-инженер	2
2.1	Карта курса	2
2.2	Как учиться по курсу	3
3	Окружение: воспроизводимость прежде всего	3
3.1	Проблема «у меня работает»	3
3.2	uv	3
4	Контейнеры и Apple Silicon	4
4.1	Образ, контейнер, volume	4
4.2	Почему в Docker на Mac нет GPU	4
4.3	Unified memory: почему Mac вообще тянет LLM	5
5	Модели: где живут и как с ними говорить	5
5.1	Три способа получить модель	5
5.2	OpenAI-совместимый API	5
5.3	Токены и контекстное окно	7
5.4	Рассуждающие модели	7
6	Скорость и память	7
6.1	Две фазы генерации	7
6.2	Почему decode упирается в память	8
6.3	Сколько памяти занимает модель	8
7	Кэш ответов и кассеты	9
8	Учиться вместе с AI-агентом	9
9	Вопросы для самопроверки	10
10	Ответы	10
11	Что почитать	11

1 Зачем этот модуль

За ближайшие месяцы вы разберёте устройство больших языковых моделей, научитесь строить RAG-системы и мультиагентные приложения на LangChain и LangGraph, писать MCP-серверы и оценивать качество LLM-систем. Всё это требует рабочего места, на котором:

- любой урок запускается одной командой и через полгода запустится так же;
- модели доступны из кода по единому интерфейсу — неважно, крутятся они у вас на ноутбуке или на сервере;
- медленная генерация не превращает каждый перезапуск ноутбука в ожидание.

Этот модуль короткий, но он закладывает понятия, которые пройдут через весь курс: **OpenAI-совместимый API, токены и контекстное окно, скорость генерации, память, которую занимает модель, и кассеты** — записанные ответы моделей.

i Практика

Уроки модуля: 01-environment (окружение), 02-docker (контейнеры и сервисы), 03-models (первый разговор с моделью), 04-encode (AI-агент как напарник). Упражнения — в exercises/.

2 Кто такой AI-инженер

Ещё недавно, чтобы применить машинное обучение, нужно было собрать данные, обучить модель и вывести её в продакшн. С появлением больших языковых моделей (LLM) центр тяжести сместился: самые способные модели уже обучены, и основная работа — **построить вокруг модели надёжную систему**.

Роль	Главный вопрос	Типичные задачи
Data scientist	Что говорят данные?	анализ, эксперименты, статистика
ML-инженер	Как обучить и выкатить модель?	признаки, обучение, сервинг, мониторинг
AI-инженер	Как построить продукт на готовых моделях?	промтты, RAG, агенты, инструменты, оценка качества, стоимость

AI-инженер не обязан обучать модели с нуля, но обязан понимать, **как они устроены**: почему модель «забывает» середину длинного контекста, почему генерация медленнее чтения промпта, откуда берутся галлюцинации и почему агент заикливается. Поэтому курс начинается с устройства LLM и только потом переходит к приложениям.

2.1 Карта курса

1. **Фундамент-экспресс** (M01–M05) — математика, классическое ML, PyTorch, NLP и обзор остальных областей. Сжато: только то, что понадобится дальше.

2. **Устройство LLM** (M06–M08) — трансформер своими руками, как обучают LLM, как устроен инференс.
3. **Работа с LLM** (M09–M13) — промпты и контекст, структурированный вывод и вызов инструментов, эмбединги, RAG.
4. **Агенты** (M14–M21) — агенты с нуля, LangChain, LangGraph, MCP, экосистема фреймворков, coding-агенты.
5. **Надёжность** (M22–M23) — оценка качества, observability, безопасность.
6. **Продакшн и смежные темы** (M24–M27) и **капстоун-проекты**.

2.2 Как учиться по курсу

Каждая тема проходит спираль **интуиция** ☒ **формализм** ☒ **код с нуля** ☒ **фреймворк** ☒ **проект**. Код с нуля — не упражнение ради упражнения: только написав ReAct-агента на чистом Python, понимаешь, что именно делает за тебя LangGraph и где он может ошибиться.

Второй принцип — **всё измеряется**. Улучшение промпта или RAG-пайплайна считается улучшением, только если оно видно на тестовом наборе. Привычка измерять — главное, что отличает инженера от человека, который «потыкал в чат».

3 Окружение: воспроизводимость прежде всего

3.1 Проблема «у меня работает»

Экосистема Python для ML обновляется очень быстро. Урок, написанный под langchain одной версии, через полгода может не запуститься на другой. Решение — **зафиксировать всё**:

- версию интерпретатора (в курсе — Python 3.12);
- точные версии всех пакетов, включая транзитивные зависимости, — в **lock-файле** `uv.lock`;
- системное окружение — в **Docker-образе**.

3.2 uv

uv — менеджер пакетов и окружений для Python. Он заменяет `pip`, `venv`, `pip-tools`, `pyenv` и `pipx`, а работает в десятки раз быстрее за счёт реализации на Rust и глобального кэша пакетов.

Команда	Что делает
<code>uv sync</code>	создаёт <code>.venv</code> и ставит ровно то, что записано в <code>uv.lock</code>
<code>uv add httpx</code>	добавляет зависимость в <code>pyproject.toml</code> и обновляет <code>uv.lock</code>
<code>uv run python script.py</code>	запускает команду внутри окружения проекта

```
uv python install 3.12
```

скачивает нужную версию Python

💡 Интуиция

`pyproject.toml` — это *намерение* («мне нужен `httpx` версии не ниже 0.28»), а `uv.lock` — *факт* («стоит `httpx` 0.28.1 с такими-то хэшами»). В `git` коммитятся оба файла.

У каждого модуля курса своё окружение (`modules/Mxx/pyproject.toml`). Так зависимости модуля про `LangGraph` не конфликтуют с зависимостями модуля про `CrewAI`, а общая библиотека курса `mlcourse` подключается к каждому модулю как локальный пакет.

4 Контейнеры и Apple Silicon

4.1 Образ, контейнер, volume

Контейнер — изолированный процесс со своей файловой системой, сетью и набором программ. В отличие от виртуальной машины, контейнер использует ядро хостовой ОС, поэтому стартует за секунды.

- **Образ** (image) — неизменяемый шаблон: ОС, Python, `uv`, системные библиотеки. Описывается `Dockerfile`.
- **Контейнер** — запущенный экземпляр образа.
- **Volume** — папка, которая живёт дольше контейнера. В курсе через `volume` в контейнер монтируется весь репозиторий: правите файлы в редакторе на Mac — контейнер сразу видит изменения.
- **Docker Compose** — описание нескольких связанных контейнеров в одном YAML-файле: `JupyterLab`, векторная база `Qdrant`, `Postgres` и т. д.

4.2 Почему в Docker на Mac нет GPU

`Docker Desktop` на macOS запускает контейнеры внутри лёгкой Linux-виртуалки: ядро Linux нужно контейнерам, а у macOS своё ядро. Графический ускоритель Apple (Metal) внутрь этой виртуалки не пробрасывается. Отсюда правило курса:

- **в Docker** — всё, что не требует GPU: LLM-приложения, RAG, агенты, базы данных, сервисы;
- **нативно на Mac** — обучение нейросетей (PyTorch с бэкендом MPS, MLX) и сама **Ollama**, которая использует Metal для ускорения.

Контейнер обращается к Ollama на хосте по специальному имени `host.docker.internal`. Библиотека курса делает это автоматически: если код запущен в контейнере, `localhost` в настройках заменяется на `host.docker.internal`.

4.3 Unified memory: почему Mac вообще тянет LLM

В обычном ПК у видеокарты своя память (VRAM), и модель должна в неё поместиться. У Apple Silicon **общая (unified) память**: CPU и GPU работают с одним и тем же пулом RAM. Поэтому Mac с 16 ГБ может запустить модель, которой нужно ~6–8 ГБ, — большинству видеокарт с 8 ГБ VRAM это не под силу.

Обратная сторона — **пропускная способность памяти**. У M1 она около 68 ГБ/с, у серверных GPU — 2–3 ТБ/с. Ниже мы увидим, что именно она ограничивает скорость генерации.

5 Модели: где живут и как с ними говорить

5.1 Три способа получить модель

Способ	Примеры	Плюсы	Минусы
Локально	Ollama, llama.cpp, MLX, LM Studio	бесплатно, приватно, офлайн	ограничено памятью ноутбука, медленнее
Свой сервер	vLLM, SGLang, llama.cpp server	большие модели, контроль	нужно железо и администрирование
Облачный API	OpenAI, Anthropic, OpenRouter и др.	самые сильные модели, без железа	платно, данные уходят провайдеру

5.2 OpenAI-совместимый API

Почти все серверы и провайдеры сегодня говорят на одном языке — на HTTP-API в формате OpenAI. Это де-факто стандарт, и курс опирается именно на него: код урока не знает, где живёт модель, — меняется только адрес в `.env`.

Основные эндпоинты:

Эндпоинт	Назначение
GET <code>/v1/models</code>	список доступных моделей
POST <code>/v1/chat/completions</code>	ответ модели на диалог
POST <code>/v1/embeddings</code>	векторные представления текстов

5.2.1 Анатомия запроса

```
{
  "model": "qwen3:8b",
  "messages": [
    {"role": "system", "content": "Ты — лаконичный ассистент."},
    {"role": "user", "content": "Что такое токен?"}
```

```
],
"temperature": 0.7,
"max_tokens": 200,
"stream": false
}
```

- `messages` — история диалога. Роли: `system` (инструкции), `user` (пользователь), `assistant` (прошлые ответы модели), `tool` (результаты инструментов, M10).
- `temperature` — «смелость» выбора следующего токена: 0 — почти детерминированно, 1 — разнообразно (подробно в M08).
- `max_tokens` — предел длины ответа.
- Дополнительно: `tools` (вызов инструментов), `response_format` (JSON по схеме), `stream` (потокковая выдача).

! Модель ничего не помнит

API не хранит историю. Каждый запрос содержит **весь** диалог целиком, и модель читает его заново. «Память» чат-бота — это просто список `messages`, который приложение отправляет каждый раз. Отсюда растущая стоимость длинных диалогов и целая дисциплина управления контекстом (M09, M16).

5.2.2 Анатомия ответа

```
{
  "model": "qwen3:8b",
  "choices": [{
    "index": 0,
    "message": {"role": "assistant", "content": "Токен — это ..."},
    "finish_reason": "stop"
  }],
  "usage": {"prompt_tokens": 27, "completion_tokens": 64, "total_tokens": 91}
}
```

- `finish_reason` — почему модель остановилась: `stop` (закончила сама), `length` (упёрлась в `max_tokens`), `tool_calls` (хочет вызвать инструмент).
- `usage` — сколько токенов прочитано и сгенерировано. Облачные провайдеры берут деньги именно за токены.

5.2.3 Потокковая выдача (streaming)

При `"stream": true` сервер не ждёт конца генерации, а отправляет кусочки по мере готовности в формате **Server-Sent Events (SSE)**:

```
data: {"choices":[{"delta":{"content":"Токен"}}]}
```

```
data: {"choices":[{"delta":{"content":" — это"}}]}
```

```
data: [DONE]
```

Каждое событие — строка `data: <JSON>` и пустая строка после неё; конец потока — `data: [DONE]`. Пользователь видит первые слова через доли секунды вместо ожидания всего ответа. В упражнении ex03 вы напишете разбор SSE-потока сами.

5.3 Токены и контекстное окно

Модель работает не с символами и не со словами, а с **токенами** — кусочками текста из её словаря (обычно 100–250 тысяч элементов). Частые слова — один токен, редкие разбиваются на несколько. Для русского текста токенов, как правило, больше, чем для английского того же смысла; как устроена токенизация, разберём в M07.

Контекстное окно — максимальное число токенов, которое модель обрабатывает за раз: промпт (весь диалог) **плюс** ответ. Если сумма больше окна, сервер вернёт ошибку или обрежет начало диалога.

Осторожно

Размер окна задаётся не только моделью, но и **сервером**. Qwen3 обучена на длинных контекстах, но Ollama по умолчанию выделяет лишь 4096 токенов — для агентов этого мало. В курсе Ollama запускается с `OLLAMA_CONTEXT_LENGTH=16384`; команда `make llm-check` проверяет это.

5.4 Рассуждающие модели

Современные модели (Qwen3, DeepSeek-R1 и др.) умеют перед ответом «думать вслух»: генерировать скрытую цепочку рассуждений (thinking), а потом — ответ. Это повышает качество на сложных задачах, но стоит времени: простой вопрос «2+2?» может превратиться в 500 токенов размышлений.

Серверы возвращают рассуждения отдельно от ответа: llama.cpp — в поле `reasoning_content`, Ollama — в поле `reasoning`. Выключается режим по-разному:

Сервер	Как выключить thinking
Ollama	<code>"reasoning_effort": "none"</code>
llama.cpp	<code>"chat_template_kwargs": {"enable_thinking": false}</code>

Функция `mlcourse.no_thinking()` возвращает оба параметра сразу — каждый сервер игнорирует чужой. В агентных циклах курса thinking по умолчанию выключен ради скорости; когда он помогает, а когда нет, измерим в M07 и M09.

6 Скорость и память

6.1 Две фазы генерации

Ответ модели рождается в две фазы:

1. **Prefill** — модель читает весь промпт. Токены промпта обрабатываются **параллельно**, одной большой операцией над матрицами, поэтому эта фаза быстрая: сотни и тысячи токенов в секунду.
2. **Decode** — модель генерирует ответ **по одному токenu**: каждый следующий зависит от предыдущего.

Отсюда две метрики:

- **TTFT** (time to first token) — время до первого токена ответа, определяется в основном prefill;
- **скорость генерации** — токенов в секунду в фазе decode.

6.2 Почему decode упирается в память

Чтобы сгенерировать **один** токен, нужно прочитать из памяти **все** веса модели. Вычислений при этом немного, а данных — гигабайты. Поэтому верхняя граница скорости генерации — это пропускная способность памяти, делённая на размер модели:

$$\text{ток/с} \lesssim \frac{\text{пропускная способность памяти, ГБ/с}}{\text{размер весов, ГБ}}.$$

Для qwen3: 8b в 4-битной квантизации (файл $\approx 5,2$ ГБ) на M1 (≈ 68 ГБ/с):

$$\frac{68}{5,2} \approx 13 \text{ ток/с}.$$

На практике получается около 8 ток/с — реальные системы достигают 50–70 % теоретического предела. Эта простая формула объясняет многое: почему квантизация ускоряет генерацию (весов меньше — читать меньше), почему серверные GPU с памятью HBM быстрее в десятки раз и почему prefill и decode оптимизируют разными способами (M08).

6.3 Сколько памяти занимает модель

Веса. Модель из N параметров при b битах на параметр занимает

$$M_{\text{веса}} = N \cdot \frac{b}{8} \text{ байт}.$$

8 млрд параметров в формате bf16 (16 бит) — 16 ГБ, в 4-битной квантизации — около 5 ГБ (часть слоёв хранится точнее, поэтому в среднем выходит ≈ 5 бит на вес). Квантизация подробно разбирается в M08.

KV-кэш. Чтобы не пересчитывать весь диалог на каждом шаге decode, модель хранит для каждого прочитанного токена ключи и значения (keys, values) механизма внимания во всех слоях:

$$M_{\text{KV}} = 2 \cdot L \cdot H_{\text{kv}} \cdot d_{\text{head}} \cdot s \cdot T,$$

где L — число слоёв, H_{kv} — число голов ключей/значений, d_{head} — размерность головы, s — байт на число (2 для fp16), T — число токенов в контексте, а множитель 2 — это K и V.

i Пример: Qwen3-8B

$L = 36, H_{kv} = 8, d_{head} = 128, s = 2: 2 \cdot 36 \cdot 8 \cdot 128 \cdot 2 = 147\,456$ байт ≈ 144 КиБ **на каждый токен**. Контекст 16 384 токена — около 2,4 ГБ. Вместе с весами $\approx 7,6$ ГБ: на Mac с 16 ГБ помещается, но с запасом не разгуляешься. Что такое ключи, значения и головы внимания, разберём в M06.

7 Кэш ответов и кассеты

Даже быстрая модель отвечает секунды, медленная — минуты. Перезапускать ноутбук с десятком запросов к модели ради одной правки неудобно, а в публичном CI моделей нет вовсе. Поэтому все клиенты курса (`mlcourse.get_client()`, `get_chat_model()` и др.) пропускают запросы через **кассеты** — записанные пары «запрос  ответ».

- Кассеты урока лежат в `lessons/<урок>/cassettes/` и коммитятся в git.
- Личный кэш — в `~/.cache/mlcourse/llm/`, не коммитится.
- Ключ записи — хэш тела запроса. Изменили промпт, модель или температуру — это новый запрос и новый вызов модели.

LLM_CACHE	Поведение
on (по умолчанию)	берёт готовый ответ из кассет или личного кэша; новые ответы — в личный кэш
off	всегда настоящий запрос
record	как on, но всё использованное сохраняется в кассеты урока
refresh	всегда настоящий запрос с перезаписью кассет
replay	только кассеты, без сети (так ноутбуки проверяются в CI)

! В продакшне

Кэширование ответов LLM — не только учебный приём. В продакшне кэш экономит деньги и время на повторяющихся запросах, а записанные ответы используются в регрессионных тестах. «Семантический кэш», который узнаёт не только одинаковые, но и похожие запросы, разберём в M24.

В кассету попадают только путь запроса, его тело и тело ответа. Адрес сервера, заголовки и ключи API не сохраняются никогда — кассеты безопасно публиковать.

8 Учиться вместе с AI-агентом

Coding-агенты вроде OpenCode читают код, запускают команды и правят файлы. Для обучения это обоюдоострый инструмент: агент может мгновенно решить упражнение — и вы ничему не научитесь.

В курсе настроен агент-**репетитор** (tutor), которому запрещено писать решения. Он объясняет, задаёт наводящие вопросы, подсказывает, где искать ошибку, и запускает тесты. Правила работы с ним:

1. Сначала попытка самостоятельно, потом вопрос репетитору.
2. Просите объяснить, а не сделать: «почему падает тест?» вместо «исправь».
3. Проверяйте ответы агента: модели ошибаются уверенно.

Устройство coding-агентов, настройку OpenCode и написание собственного агента подробно разберём в M21.

Осторожно

Coding-агенту нужен большой контекст: системный промпт и описания инструментов OpenCode занимают 10–12 тысяч токенов ещё до вашего вопроса. С окном 16K агент работает на пределе и может падать с ошибкой `exceeds the available context size`. Для комфортной работы нужно 32K и больше.

9 Вопросы для самопроверки

1. Чем `pyproject.toml` отличается от `uv.lock`? Какой из них определяет, что именно будет установлено?
2. Почему обучение нейросетей в курсе запускается нативно, а не в Docker?
3. Вы отправили модели вопрос, потом второй вопрос отдельным запросом. Почему модель не помнит первый вопрос?
4. Ответ пришёл с `finish_reason: "length"`. Что произошло и что делать?
5. Что такое TTFT и от какой фазы генерации он зависит в первую очередь?
6. Оцените верхнюю границу скорости генерации модели размером 20 ГБ на GPU с пропускной способностью памяти 1000 ГБ/с.
7. Во сколько раз вырастет KV-кэш, если увеличить контекст с 4K до 32K токенов? А размер весов?
8. Почему кассеты не содержат адрес сервера и ключ API и почему это важно для публичного репозитория?

10 Ответы

1. `pyproject.toml` описывает допустимые диапазоны версий, `uv.lock` — точные версии всех пакетов с хэшами. Устанавливается то, что записано в `uv.lock` (`uv sync`).
2. Docker Desktop на macOS запускает контейнеры в Linux-виртуалке без доступа к GPU Apple (Metal); нативно PyTorch использует MPS, а MLX и Ollama — Metal.
3. API не хранит состояние: каждый запрос самостоятелен. Чтобы модель «помнила», нужно отправлять всю историю в `messages`.
4. Ответ обрезан: модель упёрлась в `max_tokens`. Увеличить лимит, попросить ответ короче или проверить, не тратит ли модель токены на рассуждения.
5. Время до первого токена ответа; определяется в основном фазой `prefill` — чтением промпта.
6. $1000/20 = 50$ ток/с — теоретический предел; реально — около 25–35 ток/с.
7. KV-кэш растёт линейно по числу токенов — в 8 раз. Размер весов не меняется.

8. В кассету пишутся только путь, тело запроса и тело ответа. Публичный репозиторий видят все, и утёкший ключ позволил бы пользоваться моделью за чужой счёт, а адрес — атаковать сервер.

11 Что почитать

- [Документация uv](#) — разделы Projects и Locking.
- [Docker: Get started](#) — образы, контейнеры, volumes, compose.
- [Ollama: документация](#) — API, OpenAI-совместимость, переменные окружения.
- [OpenAI API Reference: Chat Completions](#) — эталон формата, на который ориентируются остальные серверы.
- [Документация OpenCode](#) — агенты, команды, провайдеры.