

Весы, GPU и первый токен: как принять LLM в эксплуатацию

Коллеги принесли модель.
Теперь нам её эксплуатировать.

Егор Андреев

Впрод

«Вот моделька. Запустите в прод»

На тестовом запросе отвечает

Доступ нужен пользователям. Контейнер, сеть, мониторинг и дежурство нам знакомы. Теперь надо понять особенности самой нагрузки.

Что унесём с собой

На что смотреть в первые дни. Для части проблем разберём конкретные решения; для остальных обозначим, куда копать. Всё подкрепим опытом команды.

Сначала проверим запуск, затем полезную мощность, восстановление и обновления.

О спикере



Егор Андреев

CEO AdminDivision и Впрод

20 лет в IT и эксплуатации

AdminDivision

DevOps, high-load, Kubernetes.

Запускаем и сопровождаем бизнес-критичные системы.

Впрод

Создаём платформу LLM-инференса для провайдеров и корпоративных команд.

Собственный опыт и опыт других команд

Сегодня разбираем задачи эксплуатации, с которыми столкнулись сами.

Что проверим в первые дни

01 GPU

хватит ли памяти и совместим ли стек.

02 Большие веса

как доставить версию и сохранить откат.

03 Долгий старт

когда вернуть трафик и где искать отказ.

04 Работа под нагрузкой

как устроена генерация и сколько полезных запросов выдержит сервис.

05 Отказоустойчивость

какой резерв переживёт выбранный отказ.

06 Тюнинг и обновления

как менять настройки и модели с проверкой результата.

После каждой основной проблемы будет ответ. Следующий уровень соберём в два слайда «Что дальше».

Сначала вместимость, потом скорость

Веса модели

Состояние
активных запросов

Рабочая память
движка

Проверяем конкретную карту

Модель GPU, доступную память и связи между картами. После весов нужно оставить место для длинного контекста и параллельных запросов.

Если одной карты мало

При **tensor parallelism** GPU совместно считают слои и часто обмениваются данными. Скорость соединений влияет на ответ.

Без NVLink проверяем PCIe и схему распределения

В некоторых конфигурациях полезнее pipeline parallelism: модель делится по слоям. Универсального коэффициента замедления нет.

«Поместилась» и «успевает отвечать» проверяем отдельно. Скорость определим в блоке метрик.

Оценим память до первого запуска

Решение | Конкретное решение

Учебная модель с обычным full attention

27 млрд параметров, BF16. 32 слоя, 8 KV-голов, размер головы 128; KV-cache тоже BF16. Одна GPU, независимые контексты, без общего префикса.

Веса: $27 \text{ млрд} \times 2 \text{ байта} / 2^{30} \approx 50,3 \text{ ГиБ}$
KV на токен: $2 \times 32 \times 8 \times 128 \times 2 = 128 \text{ КиБ}$
8 192 токена в одном запросе: 1 ГиБ KV-cache
16 таких запросов: 16 ГиБ KV-cache

Слагаемое	Учебный бюджет
Веса	50,3 ГиБ
KV-cache	16 ГиБ
Рабочие буферы, graphs и накладные расходы	условно 4 ГиБ
Итого до дополнительного запаса	70,3 ГиБ

Для условных 80 ГиБ доступной VRAM остаётся 9,7 ГиБ. Бюджет буферов уточняем при запуске; затем нагружаем максимальным допустимым контекстом и параллельностью.

Hugging Face: KV-cache · vLLM: conserving memory

Из конфигурации модели считаем нижнюю оценку; реальный запуск и нагрузка проверяют допущения.

Карта есть. Поддержит ли её наш стек?

7.0

Compute capability
NVIDIA V100

В текущих требованиях vLLM для NVIDIA: **7.5 и выше**. Также важно поколение: оно определяет форматы чисел и вычислительные ядра.

Принимаем проверенный комплект: **GPU + драйвер + образ движка + формат весов + параметры запуска**.



vLLM: требования к GPU · NVIDIA: compute capability

GPU Operator помогает сопровождать эти компоненты. Финальная проверка: реальное вычисление в контейнере.

Проверяем совместимость по цепочке

Решение | Куда копать

Шаг	Где проверяем	Что должно получиться
1. GPU и формат	Таблица compute capability NVIDIA; GPU requirements и quantization matrix vLLM	Карта поддерживает нужный движок и формат
2. Драйвер и образ	NVIDIA CUDA Compatibility; CUDA в закреплённом образе	Реальное CUDA-вычисление проходит в этом контейнере
3. Kubernetes	Component matrix GPU Operator; ресурс nvidia.com/gpu на узле	Тестовый pod получает GPU и выполняет вычисление
4. Модель и API	Supported Models vLLM; карточка модели, tokenizer, chat template	Запускается нужная revision; проходит запрос приложения

Сохраняем GPU, драйвер, digest образа, revision весов и параметры. Сверяем документацию с установленной версией.

Успешный `nvidia-smi` показывает видимость карты; вычисление и запрос модели проверяем отдельно.

NVIDIA: CUDA Compatibility · GPU Operator: component matrix · vLLM: supported models

Первый шаг: выписать точный комплект. Последняя проверка: запрос приложения на этом комплекте.

Десятки гигабайт превращаются в терабайт

Вес — численные параметры нейросети, полученные при обучении. Движок загружает их и использует для вычисления следующего токена.

Модель и формат	Официальный репозиторий
Qwen3.8-27B · BF16	55,6 ГБ
GLM-5.3 · FP8	756 ГБ
GLM-5.3-BF16	1,51 ТБ

Параметры × байт на параметр
27 млрд × 2 ≈ 54 ГБ

Это оценка BF16 до учёта состава поставки. В MoE для токена работает часть экспертов, но скачиваем веса всех экспертов.

Квантование уменьшает размер весов. Совместимость и качество выбранного варианта проверяем отдельно.

Размеры репозитория Hugging Face: Qwen3.8-27B · GLM-5.3 · GLM-5.3-BF16

Размер файлов ещё не определяет полный расход VRAM.

Эти веса ещё надо доставить

Решение | Конкретное решение

Размер	Идеальный канал 1 Гбит/с
55,6 ГБ	7,4 минуты
756 ГБ	101 минута
1,51 ТБ	201 минута

Нижняя оценка: размер × 8 / скорость канала. Ещё без конкуренции, записи на диск, проверки файлов и загрузки в GPU.

Готовим поставку заранее

- Место под текущую и новую версии; старую сохраняем для отката.
- Фиксируем revision, контрольные суммы, tokenizer и chat template.
- В закрытый контур везём образ движка и зависимости.
- Одновременный старт реплик может упереться в одно хранилище.



Если восстановиться нужно за 15 минут, скачивание 756 ГБ за ≥101 минуту уже не подходит. Готовим локальную копию заранее.

VM вернулась. Можно вернуть трафик?

Ожидание

Хост и VM снова доступны. Кажется, основная часть восстановления уже закончилась.

Что увидела команда

В одном нашем внедрении до ответа модели прошло ещё около **25 минут**: требовались загрузка модели и публикация маршрута.

VM доступна → модель загружается → вычисления готовы →
маршрут открыт → ответ обычным ключом

Время отдельных стадий этого эпизода не измерено. На другом комплекте старт займёт другое время.

Готовность хоста и готовность LLM-сервиса проверяем отдельно.

Даём старту время и проверяем готовность

Решение | Конкретное решение

Учебные условия: в отдельном тесте холодный старт занимает 30 минут. Это пример настройки; он не относится к истории про 25 минут.

Проверка	Как настраиваем	Как подтверждаем
Startup	Опрос раз в 10 с, до 210 неудач: окно около 35 мин	В выбранном окне движок проходит инициализацию
Readiness	Готовность движка после прогрева; трафик открываем после проверки маршрута	Обычный ключ получает законченный ответ
Liveness	Проверяем зависание процесса; действует после успешной startup probe	Длинная загрузка и очередь не вызывают цикл рестартов

С локальными весами и с холодным хранилищем измеряем запуск отдельно. Если срок восстановления короче измеренного старта, нужна уже прогретая копия или пересмотр срока.

Kubernetes: startup, readiness, liveness

Измерили старт → задали окно → прогрели → проверили весь путь → открыли трафик.

Модель в Deploying. Перезагрузим сервер?

Что видели

В интерфейсе висело Deploying. Похоже на очередную проблему GPU или драйвера. GPU-сервер перезагрузили.

Что оказалось

`requests > limits`: Kubernetes отвергал Deployment до создания pod. Диагностика смотрела только на pod и не показывала причину.

После исправления ресурсов модель начала загружаться. Команда добавила валидацию и события всей цепочки объектов.

Если pod нет, начинаем с того, кто должен был его создать.

Идём от созданного объекта к следующему

Решение | Конкретное решение

Где остановились	Первый шаг
Deployment отвергнут, pod нет	Ошибка API в логах создателя; ресурсы сгенерированного Deployment; события управляющего объекта
Deployment есть, ReplicaSet или pod нет	describe Deployment/ReplicaSet, события и admission-ошибки
Pod Pending	GPU-ресурсы, занятость, requests, ограничения размещения, PVC
Pod есть, движок не готов	Логи загрузки, доступ к весам, поддержка модели, VRAM и NCCL
Движок готов, клиент ждёт	Service, шлюз, ключ, разрешение на модель, таймауты

До выкладки проверяем **итоговый манифест**:

```
kubectl apply --dry-run=server -f deployment.yaml
```

Dry-run проверяет ресурсы и admission. Планирование и загрузку проверяем запуском.

Kubernetes: dry-run

Нужны статус, длительность и ошибка каждого этапа. Проверка ресурсов выполняется до выдачи «Deploying».

Как из запроса получается ответ

До генерации

Сообщения, история и описание инструментов → **chat template** собирает вход → **токенизатор** переводит его в ID токенов → очередь движка.

Prefill

Обрабатываем входные токены и строим KV-cache. Выбираем первый выходной токен.

Decode

Последовательно генерируем продолжение, используя обработанный контекст. Детокенизатор превращает токены в текст; клиент получает фрагменты потока.

Токен — элемент словаря конкретного токенизатора. Им может быть часть слова, знак или другое представление текста. Границы задаёт токенизатор; слова и токены не совпадают один к одному.

[Hugging Face: tokenizer algorithms](#) · [Hugging Face: KV-cache](#)

Длинный вход добавляет работу до первого токена. Длинный выход продлевает генерацию.

KV-cache экономит вычисления и занимает память

Внутри одного запроса

На prefill вычислили **Key и Value** для входных токенов. При генерации следующего токена повторно используем эти результаты; историю целиком заново не обрабатываем.

Prefill: [состояние входных токенов]

Decode: [сохранённое состояние][+ токен][+ токен]...

Для обычного full attention с ростом сохранённого контекста и числа одновременных запросов растёт память кеша. При её нехватке появляются вытеснения, повторные вычисления и ожидание.

Между разными запросами

Prefix caching может переиспользовать состояние совпадающего начала запросов. Совпадение смысла вопросов этого не обеспечивает.

Hugging Face: KV-cache · vLLM: prefix caching

KV-cache одного запроса и общий кеш префиксов решают разные задачи.

Проверяем кеш на своей истории запросов

Решение | Куда копать

Если растут очередь и вытеснения

- Посмотреть длины входа и выхода, активные запросы, заполнение KV-cache и preemptions.
- Ограничить допустимый контекст и нагрузку по сценарию; повторить тот же прогон.
- Проверить память, TTFT, паузы потока и goodput вместе.

Если часто повторяется начало

Сначала подтвердить совпадающий префикс после chat template и токенизации. Проверить prefix caching на закреплённой версии движка.

Сравнить холодный и прогретый кеш на одной нагрузке. Хороший результат с прогретым префиксом не описывает первый запрос.

vLLM: optimization · vLLM: prefix caching

Начинаем с длины запросов и давления на кеш. Эффект подтверждаем повторяемым прогоном.

«Быстро отвечает» состоит из нескольких времён

t0: запрос → t1: первый токен → ... → tn: последний токен → tend: завершение

Показатель	Единица	Граница измерения
TTFT	с или мс	t1 – t0: до первого выбранного выходного токена
ITL	мс	Между последовательными выходными токенами; у клиента отдельно интервалы событий потока
TPOT	мс/токен	(tn – t1) / (Nout – 1), если Nout > 1
E2E	с	tend – t0: до подтверждённого окончания ответа

Пример: первый токен в 1,2 с; последний в 5,2 с; завершение в 5,3 с; 101 выходной токен. TTFT = **1,2 с**, TPOT = **40 мс/токен**, E2E = **5,3 с**.

HTTP-байт, токен и событие потока могут приходить в разное время. Для пользователя измеряем первый видимый текст.

vLLM: latency metrics

Измеряем у клиента и смотрим хвосты p95/p99. Серверные времена объясняют, где потерялось время.

Заполняем профиль нагрузки и требования

Решение | Конкретное решение

Учебный сервис: чат поддержки

Поле	Что проверяем в примере
Вход	p50: 2 048 токенов; p95: 8 192; допустимый максимум: 8 192
Выход	p50: 128 токенов; p95 и установленный предел: 512
Приход запросов	Ступени 1 → 2 → 5 → 10 запросов/с; всплеск 20 запросов/с на 15 с
Сложные случаи	Длинная история; холодный кеш; обычный и повторяющийся префикс
Требование	Первый видимый текст ≤2 с, завершение ≤30 с, без ошибки/обрыва, нужный формат

Цель сервиса: за 7 дней **≥99% допустимых запросов** одновременно выполняют эти требования. Учебные пороги согласуем с владельцем приложения.

Для агента проверяем ещё корректность инструментов и время всей цепочки; для пакета — годные задания к сроку. Это отдельные профили и критерии.

Первые данные берём из приложения или представительных примеров. Считаем токены его tokenizer.

Goodput: сколько полезных запросов выдержим?

Goodput — число запросов в секунду, которые выполнили заданные требования к задержке и результату.

Пример SLO для чата

SLO — цель качества сервиса: за 7 дней не менее **99% допустимых запросов** получают первый видимый текст за **≤ 2 с** и завершаются за **≤ 30 с**, без ошибки, обрыва и нарушения формата. Пороги выбираем под свой сценарий.

В знаменателе доли: валидные авторизованные запросы в пределах квоты. Нехватка мощности остаётся плохим событием; политику отмен фиксируем заранее.

Условная минута: 600 завершились, 540 выполнили требования

10

Всех завершённых запросов/с

$600 / 60 = 10$ запросов/с

9

Goodput · полезных запросов/с

$540 / 60 = 9$ запросов/с

90%

Доля выполнения требований

$540 / 600 = 90\%$

Минутный пример показывает расчёт. Семидневный SLO оцениваем за полные семь дней.

В примере выполнено 90% требований при цели 99%. Эту нагрузку пока не принимаем: проверяем меньший поток или устраняем узкое место.

Измеряем весь путь на повторяемой нагрузке

Решение | Конкретное решение



Клиент	Обычный ключ: начало, поток, завершение, нужный режим API.
Шлюз	Ошибки, отказы, таймауты; модели и команды.
Движок	TTFT, очередь, фазы, KV-cache, вытеснения, длины запросов.
Узел / GPU	Память, загрузка, ошибки, рестарты и обмен между картами.

Нагрузочный прогон

- Рабочие длины входа/выхода и смесь сценариев.
- Постепенно повышаем частоту; отдельно всплеск и холодный кеш.
- Задаём независимую частоту запросов: ожидание ответа в генераторе иначе само снижает нагрузку.
- Сохраняем модель, движок, GPU, параметры, профиль; в отчёте есть ошибки и таймауты.

Прогоняем заполненный профиль, ступени нагрузки и всплеск. Сохраняем сырые результаты, ошибки, версии и конфигурацию.

Метрики подсказывают следующую проверку

Решение | Куда копать

Наблюдаем	С чего начинаем
Поздний первый токен	Очередь, длина входа, prefill, попадания в prefix cache
Начало быстрое, дальше паузы	ITL, совместный пакет, память, буферизация потока
Движок быстрый, клиент ждёт	Шлюз, сеть, прокси, фильтры, точка измерения
QPS прежний, всё медленнее	Токены входа/выхода, параллельность, новый тип запросов
OOM или вытеснения	Веса и кеш, рабочие буферы, параметры, логи

Наш факап: алерт сработал. Об аварии узнали от пользователей.

GPU-нода потеряла связь с управлением: маленькие пакеты проходили, большие ломали обмен. Временно помогла настройка MTU. При разборе выяснилось, что уведомление об аварии тоже не доставилось.

Таблица задаёт направление поиска. Причину подтверждаем данными конкретных запросов.

Проверяем всю цепочку: **нарушение** → **оповещение** → **человек** → **действие**. Теперь знаем, что восстанавливать.

Резерв есть. Кому он принадлежит?

Ожидание

Ресурс выделен нам в резерв. При отказе можно будет запустить на нём нашу модель.

Наблюдение из нашего опыта

Те же ресурсы одновременно считались запасом платформы виртуализации. Затем их продали; рассчитывать на них стало нельзя.

Что называют резервом	Что ещё нужно при отказе
Свободная GPU	Доставить и загрузить модель, прогреть, открыть маршрут
GPU с локальными весами	Загрузить модель, прогреть, переключить трафик
Уже отвечающая копия	Переключить трафик и выдержать остаточную нагрузку

У резерва должны быть владелец, приоритет и правило выделения при аварии.

Резерв должен выдержать потерю одной копии

Решение | Конкретное решение

Учебный профиль: одна копия занимает 2 GPU и выполняет требования при 12 запросах/с. Пик того же профиля — 10 запросов/с.

Схема	Ресурс	После отказа одной копии
Одна копия	2 GPU	Ждём восстановления
Две копии на независимых узлах	4 GPU	Оставшейся копии хватает: $12 \geq 10$ запросов/с

Проверка результата

Под нагрузкой останавливаем одну копию. Измеряем переключение, ошибки, оборванные потоки и задержки; подтверждаем работу оставшейся копии. Проверяем общие шлюз, сеть, хранилище и питание.

Резерв закреплён за сервисом. Вычислительный ресурс в этой схеме удвоился; дополнительные расходы и допустимый простой согласуем с владельцем.

Закреплённая копия + достаточная остаточная мощность + проверенный путь переключения.

Параметры нужны для выбранного профиля

Параметр	Смысл и единица	Учебный пример
max-model-len	Максимум контекста: вход + выход, токены	12 288 токенов
max_tokens в запросе	Предел выходных токенов	512 токенов
gpu-memory-utilization	Доля VRAM под экземпляр движка	0,90 = 90%
max-num-seqs	Максимум последовательностей за итерацию	32 последовательности
max-num-batched-tokens	Бюджет токенов за шаг планировщика	8 192 токена

Значения иллюстративные. Возможность сочетаний проверяем в версии движка. Chunked prefill позволяет делить длинный вход на порции и совмещать их с decode.

vLLM: serve arguments · vLLM: chunked prefill

Предел контекста, предел выхода и бюджет одного шага — разные числа.

Меняем один предел и повторяем тот же тест

Решение | Конкретное решение

Гипотеза для нашего чата

Приложению достаточно входа до **8 192** и выхода до **512** токенов. Большой контекст нам сейчас не нужен.

	Эталон А	Кандидат В
max-model-len	32 768	12 288
Предел выхода	512	512
GPU, веса, образ и остальная конфигурация	Одинаковые	Одинаковые
Нагрузка	Заполненный профиль	Тот же профиль

Прогреваем В. Проверяем длинный запрос, очередь, кеш, OOM, TTFT/E2E, goodput и качество. Сохраняем конфигурации и результаты.

Критерии выполнены — оставляем В. Есть регрессия — возвращаем А.

vLLM: conserving memory

Одно изменение → перезапуск → тот же тест → решение по результатам.

Меньше бит — меньше памяти весов

Память весов ≈ число параметров × биты / 8

Условная модель: 27 млрд параметров	Арифметика	Память весов
16 бит	27 млрд × 16 / 8	54 ГБ
8 бит	27 млрд × 8 / 8	27 ГБ
4 бита	27 млрд × 4 / 8	13,5 ГБ

Что проверяем заново

Поддержку формата на GPU, качество ответа, корректность инструментов и скорость на том же профиле. Масштабы, метаданные и части в другой точности добавляют расход.

Квантование KV-cache настраивается отдельно. Его состояние не входит в эти размеры весов.

vLLM: quantization matrix

Это теоретические десятичные ГБ. Меньший размер не гарантирует ускорения и сохранения качества.

Выбираем квант по памяти и результату задачи

Решение | Куда копать

- По matrix движка проверить **метод + поколение GPU**. В карточке поставки — формат, revision и требования к образу.
- Запустить отдельный кандидат с теми же ограничениями и профилем. Измерить реальную память, TTFT/E2E, goodput и ошибки.
- На контрольных задачах проверить код, JSON/tool calls, инструкции и длинный контекст вместе с владельцем приложения.
- Если требования выполнены — сохранить комплект и отчёт; при регрессии вернуть предыдущий вариант.

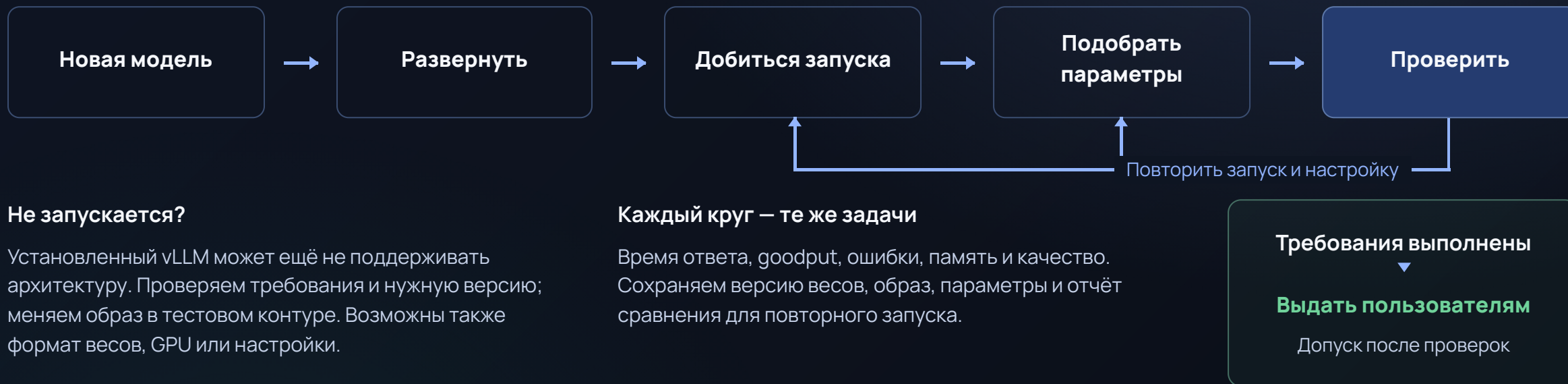
Освободившуюся память можно проверить как запас для контекста или параллельных запросов. Такой выигрыш тоже требует нового нагрузочного теста.

vLLM: quantization

Первый шаг — таблица совместимости формата. Решение о выкладке — по памяти, скорости и качеству вместе.

Вышла новая модель. Начинаем круг заново

Новые модели и их кванты испытываем постоянно. Для каждой собираем рабочую конфигурацию и сравниваем результат с текущей моделью.



vLLM: поддержка архитектур и моделей

Поток новинок утомляет ручными деплоями. Повторяемый цикл запуска, настройки и проверок становится причиной автоматизации.

Проверяем нужный приложению режим API

Решение | Конкретное решение

Наш опыт

Обычный вызов инструмента работал. **Принудительный вызов дал пять ошибок из пяти:** токены потратили, агент задачу не завершил. Причина и окончательный фикс этого эпизода неизвестны.

Проверка перед переключением

Проверяем	Как выглядит успех
Поток, JSON и нужный режим tool calls	Приложение завершает контрольную задачу
Старый ключ, включая явно выданные права	Работает после смены версии; запрет всё ещё отклоняется
Кандидат под тем же профилем	Время, ошибки, goodput, память и качество прошли критерии
Возврат старой версии	Сохранены веса, образ, настройки, маршрут и нужный ресурс

Сначала тестовая группа или малая доля трафика. Без ресурса для параллельной копии согласуем окно обновления и восстановления.

Успех одного тестового запроса не закрывает все режимы API и всех потребителей.

Что дальше: пользователи и правила запросов

Несколько команд → общий шлюз

Пример: команде А разрешена модель X, команде В — X и Y; отдельные ключи, лимиты и учёт. **LiteLLM** умеет virtual keys, model access, rate limits и spend tracking. Запуск GPU-модели и её обновление остаются задачей другого контура.

Guardrails → измеряем обязательку

Проверки входа/выхода добавляют задержку и могут ошибаться. Начать с одного правила и набора допустимых/запрещённых примеров; измерить ложные блокировки, пропуски и поведение при отказе. Права на документы проверяются отдельно.

Внешняя LLM → разрешённый маршрут

До подключения проверить качество задачи, допустимость передачи данных, API, бюджет и таймауты. Автоматический выход наружу допустим только для заранее разрешённых запросов.

LiteLLM: virtual keys · NeMo: guardrail types

Добавляем обязательку под конкретную задачу и проверяем её влияние на весь сервис.

Что дальше: автоматизация и большие кластеры

Повторяющиеся операции → автоматизация

Модели часто меняются, команд становится больше, ручные настройки расходятся. Начать с реестра версий, конфигураций, прав и отчётов; затем автоматизировать запуск, проверку, допуск и откат. У этого контура должен быть владелец.

Количество GPU само по себе не определяет число пользователей. Мощность проверяем нагрузкой; потребность в платформе — частотой и стоимостью ручной работы.

Длинный prefill мешает decode → исследуем разделение

Отдельные пулы могут уменьшить взаимное влияние фаз и дать отдельное масштабирование. Между ними нужно передавать KV-cache: появляется цена сети, баланса пулов и усложнения эксплуатации.

Сначала подтвердить узкое место по метрикам; затем сравнить с настройкой обычного движка. NVLink внутри узла не заменяет межузловую сеть.

vLLM: disaggregated prefilling

Следующий уровень начинается с доказанного ограничения и проверки выигрыша.

Что забрать перед первым запуском



GPU

есть бюджет памяти, проверенный комплект и нагрузочный тест границ.



Большие веса

версия доставлена заранее; контрольные суммы проверены; старая версия доступна для отката.



Долгий старт

окно измерено; трафик открыт после готовности; виден этап и причина отказа.



Работа под нагрузкой

заполнен профиль; выбранные требования измерены у клиента; полезная мощность подтверждена.



Отказоустойчивость

резерв закреплён; остаточной мощности хватает; переключение испытано.



Тюнинг и обновления

есть эталон, один повторяемый эксперимент, проверки приложения и путь отката.

Опыт эксплуатации уже есть. Теперь в наших проверках учтены веса, GPU, контекст и генерация.

Обсудим ваши модельки

Первый запуск?

Готовность модели и путь до ответа.

Первая нагрузка?

Профиль, метрики и ожидания пользователей.

Уже своя платформа?

Модели, команды и повторяющиеся операции.

Егор Андреев · AdminDivision / Впрод

Написать: [@egorandreev](#)

Читать: [@GrooveOps](#) · с юмором о музыке и айтишке

