

Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 23.Б09-мм

Разработка системы управления большими ЯЗЫКОВЫМИ МОДЕЛЯМИ

Киселев Сергей Александрович

Отчёт по учебной практике

в форме «Решение»

Научный руководитель:
ассистент кафедры системного программирования, Спирин Егор Сергеевич

Санкт-Петербург
2025

Оглавление

Введение	3
1. Постановка задачи	4
2. Обзор	5
2.1. Обзор существующих решений	5
2.2. Вывод	6
3. Описание решения	8
3.1. Архитектура	8
3.2. Реализация	11
3.3. Использование решения	15
3.4. Добавление поддержки популярных онлайн API LLM . .	19
4. Эксперимент	20
4.1. Условия эксперимента	21
4.2. Исследовательские вопросы	23
4.3. Метрики	23
4.4. Результаты	23
4.5. Обсуждение результатов	24
Заключение	26
Список литературы	27

Введение

Локальные большие языковые модели (далее - LLM), такие как Qwen [5], Deepseek [1], Llama [4] и другие, за последние годы приобрели значительную популярность благодаря широкому применению в различных сферах – от персонального использования до интеграции в качестве агентов и чат-ботов. Запуск этих моделей осуществляется на разнообразном оборудовании, начиная от мобильных устройств и заканчивая профессиональными видеокартами в дата-центрах.

При обеспечении доступа к моделям для большого числа пользователей возникает необходимость в горизонтальном масштабировании, которое реализуется путем запуска на нескольких компьютерах. В дата-центрах часто используются гетерогенное аппаратное обеспечение, включая видеокарты от NVIDIA, AMD и других производителей. [11] Для каждой конфигурации оборудования и модели существует оптимальной выбор сервера для запуска LLM и настроек для него, позволяющий достичь максимальной производительности. [11]

Для работы распределенного доступа к LLM необходима реализация маршрутизации запросов к разнородным узлам. Ручное управление конфигурациями и моделями не позволяет автоматически выделять оборудование на наиболее востребованные модели, предоставляя фиксированный объем вычислительных ресурсов для каждой модели. [11] Некоторые серверы LLM, такие как Ollama [8] или TabbyAPI [9], позволяют переключать модели, а некоторые вроде vLLM [12] не управляют переключением моделей.

Данная работа фокусируется на реализации системы, направляющей запросы пользователей к различным компьютерам, предоставляющим API больших языковых моделей. Также система будет автоматически подбирать эффективную конфигурацию для каждого ПК – модель, ПО для её запуска и его настройки. Она обеспечит автоматическую загрузку запрашиваемых моделей и выгрузку неиспользуемых, реализует механизм отслеживания ошибок для повторного выполнения запросов и внедрит кеширование для предотвращения избыточной работы. [7].

1. Постановка задачи

Целью работы является создание системы для управления серверами для запуска LLM. Для её выполнения были поставлены следующие задачи:

1. Провести обзор предметной области, разработать функциональные и нефункциональные требования к системе, сравнить существующие провайдеры LLM
2. Спроектировать и разработать сервис для управления LLM
3. Добавить поддержку популярных и собственных решений для доступа к LLM
4. Апробировать систему на реальном примере

2. Обзор

2.1. Обзор существующих решений

Анализ современных инструментов для управления LLM выявил три ключевых подхода с различными компромиссами между гибкостью и функциональностью.

2.1.1. Прокси-серверы с фиксированной инфраструктурой

LiteLLM [6] и AI Gateway реализуют архитектуру «клиент-сервер» с расширенными функциями маршрутизации:

- **Управление трафиком:** балансировка нагрузки по весам, автоматические повторные попытки
- **Оптимизации:** кеширование результатов, модерация контента
- **Мультипровайдерность:** поддержка >100 API (Anthropic, Cohere и др.)

Ограничения:

- Требуют статических IP и ручной конфигурации эндпоинтов
- Не поддерживают динамическое добавление/удаление серверов
- Отсутствует адаптация к изменению нагрузки (напр., автоскейлинг)

2.1.2. Децентрализованные распределенные сети

KoboldAI Horde [3] использует модель краудсорсинга вычислительных ресурсов:

- **Динамическая инфраструктура:** любой пользователь может добавить GPU в сеть
- **Стимулирование:** система кредитов за предоставленные ресурсы

- **Отказоустойчивость:** архитектура «работник-сервер» с автоматическим обнаружением узлов

Недостатки:

- Нет кеширования и повтора запросов при сбоях
- Ограниченный контроль качества обслуживания
- Публичная природа сети неприменима для корпоративных сценариев

2.1.3. Децентрализованные сети с механизмами награждения

Решения типа Targon [10] и Chutes.ai [2], построенные на сетях Bittensor, объединяют экономические стимулы с децентрализованной инфраструктурой.

Ключевые особенности:

- Система рейтинговой оценки качества ответов для распределения вознаграждений
- Так как майнеры получают деньги, присутствуют модели которые сложно запускать с небольшим бюджетом

Ограничения:

- Высокий порог входа для майнеров (требования к оборудованию)
- Нет автоматической балансировки нагрузки сети

2.2. Вывод

Для устранения ограничений предлагается система, комбинирующая лучшие качества из рассмотренных реализаций.

2.2.1. Функциональные требования:

1. Распределение запросов пользователя по компьютерам
2. Возможность подключения и отключения компьютеров во время работы программы
3. Наличие кеша промптов
4. Автоматическое переключение моделей, когда это выгодно

2.2.2. Нефункциональные требования:

1. Производительность системы (время отклика, пропускная способность)
2. Масштабируемость (возможность добавления новых узлов)
3. Надежность (устойчивость к сбоям)
4. Эффективность использования ресурсов (оптимальное распределение нагрузки)

3. Описание решения

Все части решения написаны при помощи Python и используют пакетный менеджер uv. Коммуникация между компонентами системы происходит при помощи HTTP-запросов и WebSockets.

3.1. Архитектура

В данном разделе описывается высокоуровневая концепция системы, её компоненты и логика их взаимодействия.

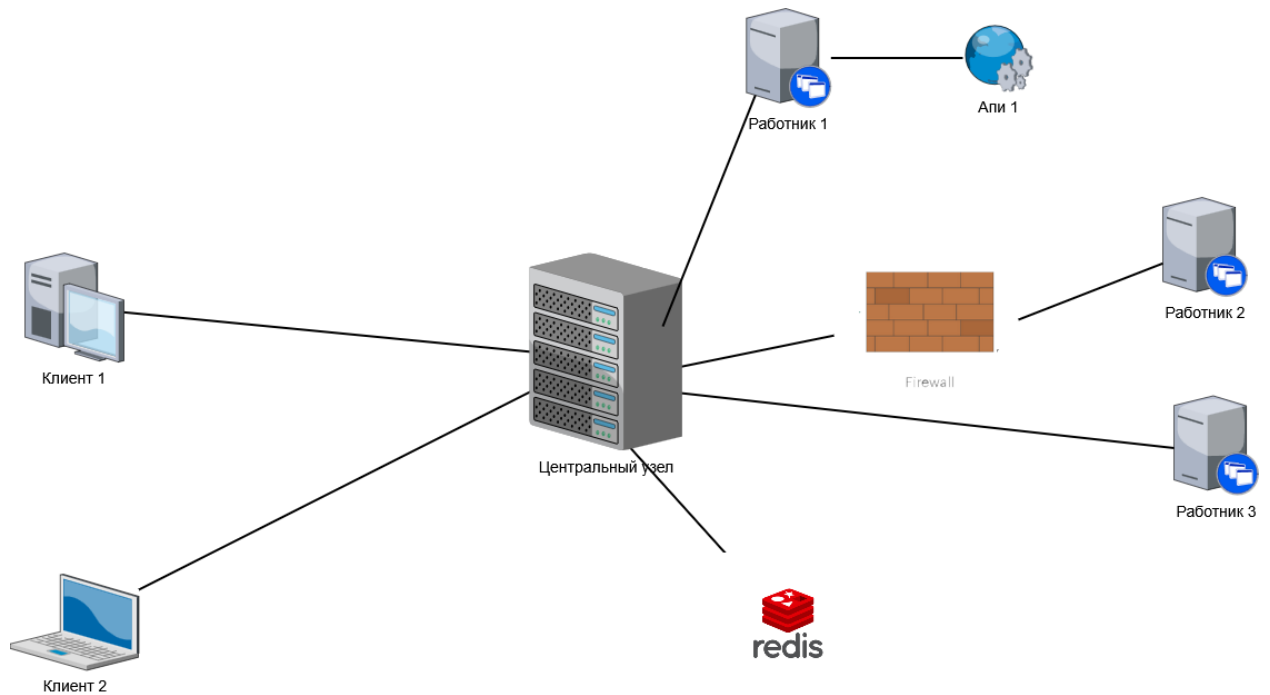
3.1.1. Обзор и цели

Для эффективной реализации кеширования, повторения запросов и возможности отложенного получения результата была выбрана архитектура "Клиенты-Сервер-Работники" (Client-Server-Worker).

Такой подход решает несколько ключевых задач:

- **Централизация управления:** Сервер выступает в роли единой точки входа и координатора, что упрощает логику очередей и кеширования.
- **Масштабируемость и отказоустойчивость:** Можно подключать и отключать любое количество работников (воркеров) без остановки системы. Если воркер выходит из строя, его задачи перераспределяются.
- **Обход сетевых ограничений:** Архитектура позволяет воркерам работать за файрволами или в сетях без статического IP-адреса (например, на сервисах Google Collab или Kaggle). Воркеры сами инициируют соединение с сервером, которому единственному в системе требуется статический адрес.

Рис. 1: Архитектура системы



3.1.2. Компоненты и их роли

Система состоит из трех логических компонентов:

- **Клиент (Client):** Источник запросов. Его задача — отправить задачу в систему и получить результат. Клиент не взаимодействует с воркерами напрямую. Он может либо дождаться ответа в реальном времени, либо отключиться и запросить результат позже.
- **Сервер (Server):** Центральный узел системы. Его обязанности включают:
 - Прием и валидация запросов от клиентов.
 - Кеширование: Проверка наличия готового ответа на запрос для его мгновенного возврата.
 - Управление очередью: Постановка новых запросов в очередь на обработку.
 - Координация воркеров: Регистрация доступных воркеров, распределение задач между ними в соответствии с их возможностями и нагрузкой.

- Возврат результатов: Доставка готовых ответов от воркеров клиентам.
- **Воркер (Worker):** Исполнительный узел. Его основная задача — выполнять вычисления (генерацию текста). Воркер:
 - Регистрируется на сервере, сообщая о своей готовности к работе и своих возможностях (какие модели он может запустить).
 - Запрашивает задачи из общей очереди на сервере.
 - Выполняет задачу, управляя локальными ресурсами (например, загружая и выгружая модели в видеопамять).
 - Возвращает результат на сервер.

3.1.3. Жизненный цикл запроса

1. **Отправка:** Клиент отправляет запрос на генерацию на сервер.
2. **Проверка кеша:** Сервер проверяет, не выполнялся ли идентичный запрос ранее. Если да, результат из кеша немедленно возвращается клиенту.
3. **Постановка в очередь:** Если ответа в кеше нет, запрос помещается в центральную очередь. Клиенту сообщается идентификатор задачи.
4. **Распределение:** Свободный воркер, способный обработать данный тип запроса, забирает задачу с сервера. Сервер блокирует задачу, чтобы она не была взята другим воркером.
5. **Исполнение:** Воркер выполняет задачу (например, обращается к модели).
6. **Возврат результата:** Воркер отправляет результат обратно на сервер. Результат может возвращаться целиком или по частям.

7. **Доставка и кеширование:** Сервер сохраняет результат в кеш для будущих запросов и передает его клиенту, который ожидает ответ или запросил его по идентификатору.
8. **Обработка сбоев:** Если воркер отключается или не возвращает результат за отведенное время (таймаут), сервер снимает блокировку с задачи, и она становится доступной для другого воркера.

3.2. Реализация

3.2.1. Сервер

Сервер представляет собой приложение, написанное с использованием фреймворка FastAPI.

3.2.2. Взаимодействие с клиентами

Реализовано два типа API:

1. **Собственное API:** Клиент сначала регистрируется через HTTP-запрос, а затем устанавливает постоянное соединение через WebSockets. Этот способ позволяет асинхронно отправлять большое количество запросов без дополнительных затрат на установку HTTP-соединения для каждого, а также получать результаты по мере их готовности. Клиент может отключиться и позже забрать результат.
2. **OpenAI-Compatible API:** Для максимальной совместимости с существующими библиотеками и инструментами сервер предоставляет эндпоинт, полностью совместимый с API OpenAI. Для него также работает кеширование и автоматическое распределение по воркерам, однако клиент должен поддерживать HTTP-соединение открытым до получения ответа.

Поддерживаемые функции: генерация продолжения чата, обработка изображений и видео (зрение), вызов функций (function calling).

Взаимодействие с воркерами Воркеры регистрируются и отключаются через специальные HTTP-эндпоинты. После регистрации воркер устанавливает WebSocket-соединение для получения уведомлений о новых задачах и для их получения.

Кеширование и очередь Для кеширования результатов используется Redis, обеспечивая быстрое чтение и запись. Очередь запросов реализована как in-memory очередь внутри приложения FastAPI для максимальной производительности.

3.2.3. Воркер

Воркер также является приложением на FastAPI, спроектированным для автономной работы и управления моделями.

Конфигурация и запуск

При запуске воркер считывает файл `config.json`, который содержит всю необходимую информацию:

- Адрес центрального сервера и имя воркера.
- Параметры API-ключей и URL для различных серверов больших языковых моделей.
- Переменные окружения.
- Список конфигураций моделей, доступных для запуска.

Пример конфигурации одной модели:

```
{
  "alias": "qwen25v1",
  "backend": "vLLM",
  "quant": "AWQ",
  "wrapper": "none",
  "context_length": 65536,
  "api_name": "/mnt/e/models/Qwen_Qwen2.5-VL-7B-Instruct-AWQ",
  "load_options": {
```

```

"num_gpu_layers": 999,
"gpu_split": [1],
"run_path": "/home/ubuntu/miniconda3/bin/conda",
"run_arguments": "run --no-capture-output -n vllm vllm serve ...",
"environment": {
  "CUDA_VISIBLE_DEVICES": "0"
}
},
"performance_metrics": {
  "parallel_requests": 3,
  "ram_requirement": 1000,
  "vram_requirement": [20000, 0],
  "benchmark_results": {
    "tokens_per_second": 42.5
  }
}
}
}

```

3.2.4. Автоматическая оптимизация и управление ресурсами

- **Бенчмарки:** При первом запуске с новой конфигурацией воркер автоматически проводит бенчмарк. С помощью бинарного поиска он определяет оптимальное количество параллельных запросов (`parallel requests`), при котором достигается максимальная пропускная способность (`tokens/second`). Это значение используется для запроса задач у сервера.
- **Измерение ресурсов:** Во время бенчмарка измеряется пиковое потребление видеопамяти (VRAM) с помощью PyTorch и оперативной памяти (RAM).
- **”Горячие” и ”холодные” модели:** Воркер сообщает серверу, какие модели у него уже загружены в память (”горячие”), а какие требуют загрузки, возможно, с выгрузкой других моделей (”хо-

лодные”). Сервер при распределении задач отдает приоритет ”горячим” моделям для минимизации задержек.

- **Автоматическая загрузка/выгрузка:** Воркер самостоятельно управляет загрузкой и выгрузкой моделей на основе требований к VRAM/RAM для входящих задач.

Рис. 2: Поиск оптимального числа параллельных запросов для конфигурации

Benchmark Results Summary					
Parallel Requests	Status	Throughput (req/s)	Time per req (s)	RAM (MB)	VRAM (MB)
1	✓	1.45	0.69	625	2238, 152
2	✓	12.82	0.08	626	2240, 152
4	✓	16.40	0.06	626	2240, 152
8	✓	25.24	0.04	626	2240, 152
16	✓	26.24	0.04	626	2240, 152
32	✓	28.46	0.04	627	2240, 152
40	✓	24.07	0.04	628	2242, 152
52	✓	27.00	0.04	628	2242, 152
58	✓	25.96	0.04	628	2242, 152
61	✓	25.82	0.04	628	2242, 152
62	✓	25.65	0.04	628	2242, 152
63	✓	24.00	0.04	628	2242, 152
64	✓	25.86	0.04	628	2242, 152

🔧 Optimal Configuration Parallel Requests: 32 Throughput: 28.46 requests/second Time per request: 0.04 seconds RAM Usage: 627 MB VRAM Usage: 2240 MB, 152 MB					
--	--	--	--	--	--

Модульная структура: Wrappers и Backends

Для гибкости и расширяемости логика работы с моделями разделена на два слоя:

- **Wrappers (Обертки):** Модули, применяющие специфичную логику к запросу до его отправки в бэкенд. Например, для моделей Mistral используется wrapper, который корректно форматирует историю чата и запросы на вызов функций с помощью библиотеки `mistral-common`. Это гарантирует одинаковое поведение модели независимо от того, поддерживает ли инференс-сервер этот формат нативно.

- **Backends (Бэкенды):** Модули-адаптеры для взаимодействия с различными серверами инференса. Каждый бэкенд реализует общий интерфейс для загрузки, выгрузки и выполнения запросов, преобразуя внутренний формат запроса в формат, понятный конкретному API.
 - Реализованные бэкенды: `vllm.py`, `ollama.py`, `openai.py`, `tabby.py`.
 - Метод `get_type()` в каждом бэкенде указывает, является ли он управляемым локально (как vLLM) или это внешний облачный API.
 - Все бэкенды поддерживают полный набор функций: продолжение чата и текста, обработку изображений/видео, стриминг ответов и передачу дополнительных параметров генерации.

3.3. Использование решения

3.3.1. При помощи библиотеки

Клиент для системы реализован при помощи асинхронной библиотеки для Python. Библиотека доступна в Python Package Index, в следствии чего её можно легко и быстро установить.

Пример использования библиотеки:

```
# example_usage.py
import asyncio
import time
from circuit import CircuitForBatchProcessing

async def task(circuit, id):
    # Первый раунд чата:
    conv = [
        {
```

```

        "role": "user",
        "content": f"К какой категории относится этот текст."
    }
]
result = await circuit.chat(["llama8b"], conv)
print(f"Task {id}/1 done:" + str(result))

# Начало замера времени
start_time = time.time()

# Запуск 100 задач для задания "my_processing_job".
CircuitForBatchProcessing.dispatch(
    job_name="my_chat_job",
    task_func=task,
    num_tasks=500,
    api_url="my-system-ip:6325" # Required parameter.
)

# Конец замера времени
end_time = time.time()

# Вывод времени выполнения
print(f"Total time taken: {end_time - start_time:.2f} seconds")

```

В примере показывается запрос к серверу на выполнение 500 задач для большой языковой модели.

Класс `CircuitForBatchProcessing` позволяет создавать работу, задания для которой будут генерироваться пользовательской функцией `task func`. Необходимо задать число задач и индекс задачи будет передаваться в функцию как параметр.

В пользовательской функции генерации работ можно использовать функцию `circuit.chat(models, conv)`. Данная асинхронная функция создаёт задачу на генерацию продолжения чата в параметре `conversation`

одной из моделей в списке `models`.

В `conversation` могут быть текстовые сообщения с прикрепленными изображениями, видео и вызовами функций.

Функция `circuit.chat` содержит в себе скрытую логику батчинга запросов и загрузки кеша. Так, например если нужно выполнить 1 000 00 задач, функция `circuit.chat` заблокирует выполнение всего миллиона экземпляров этой функции и будет постепенно продолжать каждую из них пачками. Сначала будет выполнено 500 функций, потом 1000 и т.д.

В случае если клиент отключиться и подключиться позже, `circuit.chat` сразу загрузит ответы на выполненные промпты из кеша и не нужно будет ждать заново.

В одной функции генерации задачи можно использовать несколько `circuit.chat` и обращаться к одной или нескольким моделям таким образом проводя разные операции с данными или продолжая часть с LLM.

Предполагается, что функция `CircuitForBatchProcessing.dispatch` будет использоваться для обработки больших массивов данных так как она предоставляет индекс задачи по которому можно, например, брать ячейку из датасета, а также позволяет получить надежное выполнение задач без необходимости писать промежуточные сохранения или кеш на стороне клиента.

3.3.2. При помощи стандартных библиотек OpenAI

Для доступа к OpenAI-Compatible Endpoint можно использовать стандартную библиотеку OpenAI. Возможна интеграция в решения которые использовали OpenAI-Compatible API без переписывания кода.

Пример для Python:

```
from openai import OpenAI

# Initialize the client with a custom endpoint
client = OpenAI(
    base_url="https://your-custom-endpoint.com/v1"
```

```

)

# Make a simple chat completion request
response = client.chat.completions.create(
    model="gemma-3-27b", # or your specific model name
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Hello!"}
    ],
    temperature=0.7,
    max_tokens=100
)

# Print the response
print(response.choices[0].message.content)

```

3.3.3. При помощи HTTP запросов

Также для доступа к эндпоинту можно использовать обычные HTTP запросы или библиотеки для них в любом языке программирования.

Пример с консольной утилитой Curl:

```

curl -X POST "https://your-custom-endpoint.com/v1/chat/completions" \
-H "Content-Type: application/json" \
-H "Authorization: Bearer api-key" \
-d '{
    "model": "gemma-3-27b",
    "messages": [
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Hello!"}
    ],
    "temperature": 0.7,
    "max_tokens": 100
}'

```

}'

3.4. Добавление поддержки популярных онлайн API LLM

Поддержка стандартизированного OpenAI API была добавлена во время реализации взаимодействия с локальными серверами LLM.

Отдельно была добавлена поддержка российских API GigaChat от Сбера и YandexGPT. Они используют свой формат апи, поэтому была необходима трансляция запросов в обе стороны для каждого из них.

Также как и с остальными серверами LLM, поддержка реализуется в папке backends: gigachat.py yandex.py

Для всех API реализована полная поддержка:

1. Продолжение чатов
2. Поддержка изображений и видео
3. Поддержка получения отправки запросов по частям
4. Поддержка установки дополнительных параметров

4. Эксперимент

Было проверено как работает реализованная система и LiteLLM в реальных условиях.

Максимальная эффективность системы на которой было запущено API большой языковой модели (с vLLM) достигалась при 50 одновременных запросах. LiteLLM использует обычные OpenAI-Compatible запросы для коммуникации и при 50 одновременных запросах наблюдались потери, которые заставляли LiteLLM делать повторные попытки соединения, что приводило к заметной задержке. Поэтому, также были проведены тесты с 20 одновременными запросами и заметно более высокой стабильностью. Система реализованная в данной работе работает на WebSockets с надежной доставкой и повторная отправка пакетов не контролируется.

Каждый запрос - обращение к модели с просьбой написать длинную историю на несколько страниц. Однако, на всех запросах стоит ограничение длины генерации в несколько предложений (50 токенов). Таким образом, каждый запрос выполняется примерно одинаковое число времени. Также, в запрос добавляется случайно сгенерированное число, которое не позволяет использовать кеш.

Рис. 3: Конфигурация 1

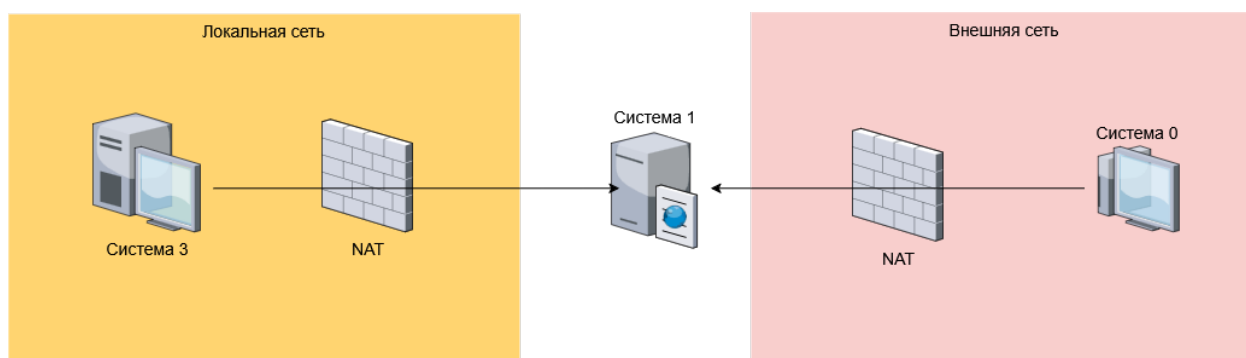
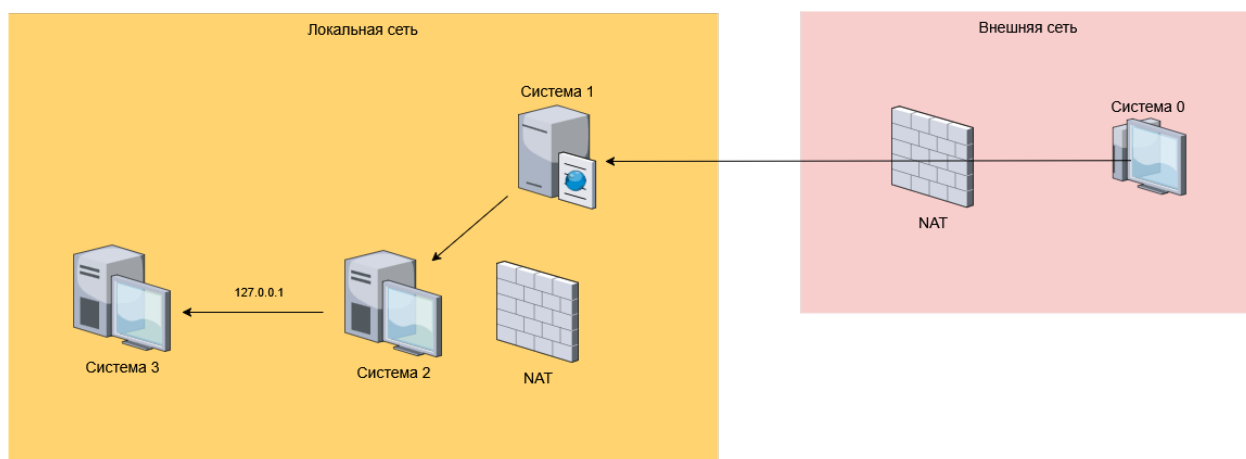


Рис. 4: Конфигурация 2 с LiteLLM



При использовании LiteLLM конфигурация получается сложнее из-за необходимости наличия возможности соединения по направлению от клиента к работнику - это требует возможности прохода через Firewall и правила WSL.

Доступ извне в локальную сеть с системами 1,2,3 есть только у системы 1. На ней установлено первое прокси LiteLLM, направленное на систему 2, где установлено второе прокси LiteLLM, направленное в WSL. Системы 1 и 2 выступают как промежуточные узлы, которые просто перенаправляют HTTP запросы.

В случае с системой реализованной в данной работе, система 1 является сервером и Системы 3 (работник) и 0 (клиент) подключаются к ней напрямую без необходимости обходить NAT.

4.1. Условия эксперимента

4.1.1. Система 0 ("Клиент")

- Windows 11
- Python 3.11
- Интернет: 100 Мбит/с
- Находится за фаерволом
- Расстояние до системы 1 — ≈ 4000 км

4.1.2. Система 1 ("Сервер")

- Ubuntu 24.04.2 LTS
- Python 3.12
- Интернет: 500 Мбит/с
- Redis 8.0
- LiteLLM 1.72.0
- Находится в локальной сети с системой 2
- Единственная система с доступом извне

4.1.3. Система 2 ("Хост рабочего")

- Windows 10
- Python 3.10
- Интернет: 500 Мбит/с
- WSL 2 (с системой 3 внутри)
- LiteLLM 1.72.0
- Находится за фаерволом, доступна только из системы 1
- Временный доступ для LiteLLM

4.1.4. Система 3 ("Рабочий")

- Ubuntu 22.04.4 (в WSL 2)
- Python 3.12
- Интернет: 500 Мбит/с
- vLLM 0.9.0.1

- NVIDIA RTX 3060 Ti 8GB
- Изолирована фаерволом WSL
- Доступ только из системы 1 через 127.0.0.1

4.2. Исследовательские вопросы

RQ1 : правда ли предложенная в работе система, реализующая дополнительную функциональность (надежность, кеш, упрощенная топология сети) не даёт существенного замедления по сравнению с альтернативами?

RQ2 : насколько надежно работает система в реальных условиях?

4.3. Метрики

В качестве метрики было взято время, необходимое для того чтобы обработать N запросов.

4.4. Результаты

Измерение времени выполнения каждого из N числа запросов выполнялось 5 раз для каждой системы.

Таблица 1: Время выполнения запросов (в секундах, $\mu \pm \sigma$, меньше — лучше)

Запросы	Эта работа	LiteLLM 50	LiteLLM 20
10	6.34 ± 0.05	13.72 ± 11.80	8.05 ± 12.36
100	8.22 ± 0.04	37.07 ± 12.66	4.14 ± 0.25
500	18.11 ± 2.51	22.47 ± 0.10	17.43 ± 0.39
1000	25.10 ± 5.70	22.83 ± 0.56	35.05 ± 0.22
2500	65.51 ± 1.86	55.55 ± 1.05	87.26 ± 0.56

Таблица 2: Количество неудачных HTTP запросов ($\mu \pm \sigma$)

Кол-во запросов	Эта работа	LiteLLM (50)	LiteLLM (20)
10	-	0.8 ± 0.8	0.7 ± 1.2
100	-	3.0	0.0
500	-	2.7 ± 1.5	0.0
1000	-	0.0	0.0
2500	-	1.7 ± 2.9	0.0

4.4.1. RQ1

Пояснения

Как видно из таблицы1, при выполнении разного числа запросов, время затраченной на них системой реализованной в данной работе и LiteLLM в разных конфигурациях варьировалось.

В некоторых ситуациях было получено преимущество (производительность при 500 запросах над LiteLLM 50), а в некоторых есть потенциал для оптимизации (2500 запросов над LiteLLM 50).

4.4.2. RQ2

Пояснения Система реализованная в данной работе справилась с задачами без потерь. Для объяснения результатов, полученных в таблице1, приводится таблица2 с неудачными HTTP-запросами LiteLLM в разных конфигурациях. Случайная потеря HTTP запросов при больших нагрузках заставляла LiteLLM отправлять повторные запросы, которые заметно замедляли работу на небольшом количестве запросов.

4.5. Обсуждение результатов

Стоит заметить, что система реализованная в данной работе реализует дополнительную функциональность - даже при отключении интернета или включения/отключения любого узла в сети, она сможет

продолжить работу после возврата в нормальное состояние без потери данных, где механизм повтора неудачных HTTP запросов LiteLLM не рассчитан на долгие отключения. Также, в реализованной системе на каждом запросе проверялось наличие кеша ответа в Redis.

Заключение

В итоге были достигнуты следующие результаты.

- Были рассмотрены существующие системы для управления серверами LLM
- Была реализована первая версия продукта
- Были добавлены популярные провайдеры LLM: OpenAI, YandexGPT и Gigachat
- Система была апробирована в реальных условиях

Код реализации и бенчмарки доступны на GitHub:
<https://github.com/orgs/esoteric-ai/repositories>

Список литературы

- [1] A Liu B Feng B Xue B Wang B Wu C Lu.... [Deepseek-v3 technical report](#). — URL: <https://arxiv.org/abs/2412.19437>.
- [2] Chutes. Chutes. — URL: <https://chutes.ai/> (дата обращения: 7 мая 2025 г.).
- [3] Horde KoboldAI. KoboldAI Horde. — URL: <https://github.com/Haidra-Org/AI-Horde/tree/main> (дата обращения: 7 мая 2025 г.).
- [4] How Far Can Camels Go? Exploring the State of Instruction Tuning on Open Resources / Yizhong Wang, Hamish Ivison, Pradeep Dasigi et al. // Advances in Neural Information Processing Systems / Ed. by A. Oh, T. Naumann, A. Globerson et al. — Vol. 36. — Curran Associates, Inc., 2023. — P. 74764–74786. — URL: https://proceedings.neurips.cc/paper_files/paper/2023/file/ec6413875e4ab08d7bc4d8e225263398-Paper-Datasets_and_Benchmarks.pdf.
- [5] J Bai S Bai Y Chu Z Cui K Dang X Deng... [Qwen technical report](#). — URL: <https://arxiv.org/abs/2309.16609>.
- [6] LiteLLM. LiteLLM. — URL: <https://github.com/BerriAI/litellm> (дата обращения: 7 мая 2025 г.).
- [7] Nur Arifin Akbar Ardian Webi Krida Akbar Setiawan. Reverse Engineering GitHub CoPilot: Creating an OpenAI-Compatible Endpoint for Enhanced Developer Integration // Intechno Journal : Information Technology Journal. — URL: <https://jurnal.amikom.ac.id/index.php/intechno/article/view/1895/678>.
- [8] Ollama. Ollama. — URL: <https://github.com/ollama/ollama> (дата обращения: 7 мая 2025 г.).
- [9] TabbyAPI. TabbyAPI. — URL: <https://github.com/theroyallab/tabbyAPI> (дата обращения: 7 мая 2025 г.).

- [10] Targon. Targon. — URL: <https://targon.com> (дата обращения: 7 мая 2025 г.).
- [11] Y Xiong J Huang W Huang X Yu E Li Z Ning J Zhou L Zeng X Chen. High-Throughput LLM inference on Heterogeneous Clusters. — URL: <https://arxiv.org/pdf/2504.15303>.
- [12] vLLM. vLLM. — URL: <https://github.com/vllm-project/vllm> (дата обращения: 7 мая 2025 г.).