## 📘Введение

Инструкция предназначена для специалистов, обеспечивающих запуск, сопровождение и мониторинг работы сервера **Neuro-ACT**, реализующего глубокий анализ текстов с использованием MoE-архитектуры на базе модели **DeepSeek-V3 32B** через Ollama.

Сервер получает текстовые данные с сервера транскрибации (**TR-Routing**) и формирует аналитический JSON-ответ с детальным психологическим и поведенческим анализом диалогов. Работа сервера основана на максимально прозрачном логировании всех этапов обработки текста: от токенизации до финального анализа экспертов и полной сборки результата.

Настоящая инструкция описывает:

* Структуру проекта и порядок запуска,
* API взаимодействие с сервером транскрибации (**TR-Routing**),
* Подробные алгоритмы логирования MoE-архитектуры:
  + токенизация,
  + выбор и маршрутизация экспертов,
  + вклад и логика работы каждого эксперта,
  + итоговая сборка результата (reasoning).
* Механизмы мониторинга и детальной обработки ошибок,
* Сохранение результатов анализа в формате Excel с полным набором параметров и логов (текст, ответ нейросети, токенизация, маршруты, веса, балансировка, решения экспертов и reasoning),
* Повторную отправку и резервные механизмы отказоустойчивости,
* Основы конфигурации и управления параметрами сервера.

Цель инструкции — обеспечить прозрачную и стандартизированную эксплуатацию системы, её полную отказоустойчивость и возможность дальнейшей адаптации модели через точную настройку роутинга и дообучение экспертов.

## 🧭 Содержание

| **№** | **Раздел** | **Стр.** |
| --- | --- | --- |
| 1 | 📌Введение | 1 |
| 2 | ⚙️Установказависимостей | 3 |
| 2.1 | Установка системных зависимостей | 4 |
| 2.2 | Установка драйверов NVIDIA для RTX 5090 | 4 |
| 2.3 | Разделение дискового пространства | 5 |
| 2.4 | Установка Ollama и DeepSeek-V3 (.gguf) | 5 |
| 2.5 | Установка Python-зависимостей | 5 |
| 2.6 | Проверка окружения | 6 |
| 3 | 📁Структура проекта | 7 |
| 4 | 📥Реализация API-приёмника от сервера TR-Routing (analyze\_api.py) | 9 |
| 5 | 🧠Реализация анализатора (analysis\_worker.py) | 12 |
| 6 | 📖Реализация логирования всех этапов обработки | 23 |
| 6.1 | log\_tokenization() — токенизация | 23 |
| 6.2 | log\_routing\_and\_scores() — маршрутизацияивеса | 23 |
| 6.3 | log\_expert\_outputs() — выводы и логика экспертов | 24 |
| 6.4 | log\_final\_aggregation() — сборка и reasoning финального ответа | 24 |
| 7 | 📊Формирование результатов в Excel и структуре файлов на диске | 26 |
| 8 | 🚀Централизованный запуск всех компонентов системы (main.py) | 34 |
| 9 | ✅Проверкасистемы (сервер Neuro-ACT) | 36 |
|  |  |  |
|  |  |  |
|  |  |  |

## 📌Цель

Реализовать отказоустойчивый, многопоточный сервер **Neuro-ACT**, который:

1. **Принимает** POST-запросы от сервера транскрибации на маршрут /api/analyze\_transcript.
2. **Валидирует** обязательные поля запроса:, text, metadata.timestamp, metadata. track\_id.
3. **Выполняет токенизацию** входного текста, формируя карту токенов и их текстовые соответствия.
4. **Передаёт текст** в Ollama (модель DeepSeek-V3 32B в формате .gguf) с поддержкой MoE-архитектуры.
5. **Логирует процесс роутинга** токенов по экспертам: Top-k выбор, веса, карту распределения токенов.
6. **Сохраняет финальный анализ** reasoning в формате JSON в каталоге logs/reasoning/<track\_id>.json.
7. **Формирует результат** в формате JSON (13 ключевых полей) и сохраняет его в структуре результатов на диске.
8. **Размечает результаты** в папке, включая:
   * транскрибированный текст,
   * ответ нейросети,
   * детальные логи токенизации, маршрутизации, reasoning и баланса экспертов,
   * итоговый Excel-отчёт.
9. **Логирует каждый этап** обработки по track\_id в специализированные лог-файлы (logs/ai\_receive.log, logs/send\_result.log, logs/errors.log).
10. **Организует работу в 4 асинхронных потока** для параллельной обработки поступающих задач.
11. **Сохраняет задачи в формате JSONL**
12. **При ошибках** сохраняет проблемные задачи в папку results/pending/ с последующей автоматической повторной отправкой через retry\_dispatcher.py.
13. **Автоматически очищает** временные и устаревшие файлы reasoning и логов по истечению заданного TTL (Time-to-Live).

## 2. Установи зависимости

### 🖥️ Аппаратная конфигурация сервера Neuro - ACT

| **Компонент** | **Характеристика** |
| --- | --- |
| **Материнская плата** | SuperMicro MBD-X13DAI-T-B, E-ATX, чипсет Intel C741 |
| **Процессор** | Intel Xeon Gold 5418Y, 24-ядерный, 2.0–3.8 ГГц |
| **Оперативная память** | 2×32 ГБ Samsung DDR5 ECC Registered (4800 МГц) |
| **Видеокарта** | Palit RTX 5090 GameRock 32 ГБ GDDR7 |
| **Накопители** | Samsung PM9A1 256 ГБ (M.2, системный), Samsung PM9A1 1 ТБ (M.2, для модели и логов) |
| **Охлаждение CPU** | SuperMicro SNK-P0088AP4 |
| **БП** | Thermaltake ToughPower GF3 1350W |
| **Корпус** | Jonsbo D500 Black |
| **ОС** | Ubuntu 24.04 Server x86\_64 |
| Python версия | **Python 3.12.2** (официально установлена) |

📂**Цель этого шага:**  
Создать полноценное окружение для запуска **DeepSeek-V3 32B (.gguf)** через **Ollama**, настроить аппаратные драйверы, логирование, поддержку GPU и корректно распределить ресурсы по дискам. Всё должно работать стабильно на сервере с конфигурацией, указанной выше.

### 🔹2.1: Установка системных зависимостей

sudo apt update

sudo apt install -y \

curl \

python3 \

python3-pip \

build-essential \

libnvidia-ml-dev \

libopenblas-dev \

unzip \

ffmpeg \

git \

nvtop \

htop \

pciutils

📊**Объяснение:**

* libnvidia-ml-dev — доступ к GPU через nvidia-smi
* nvtop — мониторинг RTX 5090 в реальном времени
* libopenblas-dev — ускорение математики для DeepSeek
* unzip, ffmpeg — системные утилиты для обработки файлов

🔹2.2: Установка драйверов NVIDIA для RTX 5090

sudo apt install software-properties-common

sudo add-apt-repository ppa:graphics-drivers/ppa

sudo apt update

sudo apt install nvidia-driver-545

sudo reboot

🔧Послеперезагрузки:

nvidia-smi

**В текущем разделе 2.3 ("Разделение дискового пространства")**  
мы создали каталоги для:

* /models/deepseek/ — модель,
* /ollama\_cache/ — кэш,
* /ollama\_ai/logs/ — логи Ollama,
* /ollama\_ai/results/ — результаты Ollama.

❗**НО:** мы **не указали**, где будет храниться основная структура **анализа задач**, которую генерирует наш сервер Neuro-ACT после обработки текста:

* Логи токенизации,
* Логи маршрутизации MoE,
* Логи reasoning,
* JSON-результаты нейросети,
* Папки для pending (ошибок),
* Готовые Excel-отчёты.

А это **ядро** нашей работы.

# 📋Как правильно дополнить

Мы должны **добавить ещё одну базовую папку**:

/srv/ai\_server/

**И там уже будет размещаться:**

* Код (\*.py файлы)
* Каталоги для логов (logs/)
* Хранилище результатов (storage/)
* Очереди задач (queues/)

## 🔹2.3: Разделение дискового пространства

💾**SSD 1 (256 GB)** — Система + Ollama

* /usr/bin/ollama
* /etc/ollama/config

💾**SSD 2 (1 TB)** — Модели, кэш Ollama и аналитика задач сервера Neuro-ACT

* /models/deepseek/
  + deepseek-custom.gguf — модель DeepSeek-V3 32B
* /ollama\_cache/
* /ollama\_ai/logs/
* /ollama\_ai/results/

➕**/srv/ai\_server/** — рабочая папка проекта Neuro-ACT:

| **Путь** | **Назначение** |
| --- | --- |
| /srv/ai\_server/ | Код всех компонентов сервера |
| /srv/ai\_server/logs/ | Логи: токенизация, MoE, reasoning, ошибки |
| /srv/ai\_server/storage/ | Результаты анализа (ai\_results, pending) |
| /srv/ai\_server/queues/ | Очереди задач на анализ |
|  |  |

### 🔧Команды для создания структуры:

sudo mkdir -p /models/deepseek /ollama\_cache /ollama\_ai/logs /ollama\_ai/results

sudo mkdir -p /srv/ai\_server/{logs,storage,queues,fallback\_db}

### 🔧Монтированиевторогодиска:

sudo mount /dev/nvme1n1p1 /models

# (и можно дополнительно прописать в /etc/fstab для автоподключения)

### 🔹2.4: Установка Ollama + DeepSeek

curl -fsSLhttps://ollama.com/install.sh | sh

ollama serve &

ollama pull deepseek-custom

🧰Альтернатива:

ollama create deepseek-custom -f /models/deepseek/Modelfile.ollama

### 🔹2.5: Установка Python-зависимостей

pipinstall \

transformers \

loguru \

pyyaml \

requests \

pandas \

matplotlib \

seaborn \

fastapi \

uvicorn \

aiohttp

### 🔹2.6: Проверка зависимостей

ollama list # должен быть deepseek-custom

ollama run deepseek-custom # ручнойзапуск

nvidia-smi # GPU должна определиться

nvtop # реальный мониторинг загрузки

### ✅На этом этапе ты:

* Подготовил систему под **Xeon + RTX 5090**
* Установил драйверы и CUDA
* Разделил диски по функциям: **модель**, **логи**, **результаты**
* Запустил Ollama и DeepSeek
* Убедился в работоспособности GPU и модели

**3. Структура проекта**

### 📂Цель этого шага:

Создать логически корректную, масштабируемую и отказоустойчивую структуру проекта **на основе архитектурных принципов сервера TR-Routing**, с учётом роли сервера Neuro - ACT.

## 🧱 Итоговая структура после выполнения

/srv/ai\_server/

├── main.py # Точка входа: запускает все процессы сервера

├── config.yaml # Настройки модели, логов, параметров MoE

├── analyze\_api.py # Приём запроса от транскрибатора

├── analysis\_worker.py # Токенизация, MoE-анализ, сборка ответа

├── result\_saver.py # Сохранение результатов на диск в структуре + Excel

├── retry\_handler\_ai.py # Повторная отправка задач из pending (опционально)

├── cleanup\_ai.py # Очистка устаревших файлов reasoning и логов

│

├── logs/

│ ├── api\_receive.log # Лог приёма задач от TR-Routing

│ ├── analysis.log # Лог процесса анализа через Ollama

│ ├── errors.log # Лог критических ошибок

│ ├── system.log # Общие события и мониторинг

│

├── queues/

│ ├── ai\_queue.jsonl # Очередь задач на обработку

│ └── retry\_queue.jsonl # Очередь для повторной отправки (опционально)

│

├── storage/

│ ├── tokenization\_logs/ # Логи токенизации (id токенов и тексты)

│ ├── moe\_logs/ # Логи маршрутизации токенов и веса экспертов

│ ├── reasoning\_logs/ # Финальные reasoning по каждому звонку

│ ├── ai\_results/ # JSON-ответы нейросети

│ ├── excel\_exports/ # Готовые Excel-файлы по каждому звонку

│ └── pending/ # Ошибочные задачи для последующей отправки

│

├── utils/

│ └── (вспомогательные скрипты: парсеры, обработчики)

│

├── models/

│ └── (описания структур данных для валидации через Pydantic, если потребуется)

### 💡Принципы построения проекта:

| **Принцип** | **Реализация** |
| --- | --- |
| 📦Модульность | Приём, анализ, сохранение, отправка — в отдельных модулях |
| 🧱Прозрачность | Логи по каждому этапу, привязка к track\_id и |
| 🔁Fallback | Только через pending/ очередь на диске (без SQLite) |
| 🧠Анализ | Структурированное сохранение результатов reasoning |
| 💽Хранилище | Чёткое разделение: сырьё, промежуточные данные, ошибки, результаты |
| 📊Расширяемость | Возможность дальнейшего добавления аналитики и визуализации |

# **4. Реализация API-приёмника (analyze\_api.py)**

## 📌Функциональность (обновлённая)

* Используем FastAPI.
* Принимаем **multipart/form-data** с полями:
  + track\_id — UUID задачи.
  + text — транскрибированный текст звонка.
  + timestamp — дата и время звонка (ISO 8601).
  + audio — оригинальный аудиофайл звонка (.mp3).

## 🧠 Алгоритм работы (обновлённый)

1. Валидировать обязательные поля запроса: track\_id, text, timestamp, audio.
2. Создатьпапку/results/<track\_id>/.
3. Сохранитьaudio.mp3впапку/results/<track\_id>/.
4. Сформировать задачу на анализ и сохранить в файл очереди queues/ai\_queue.jsonl.
5. Логировать приём задачи в logs/api\_receive.log.
6. В случае ошибки логировать в logs/errors.log.
7. Отправить статус в мониторинг: queued\_for\_analysis.

## ✅FastAPI-приёмник multipart от TR-Routing

from fastapi import FastAPI, File, Form, UploadFile, HTTPException

from fastapi.responses import JSONResponse

from fastapi.middleware.cors import CORSMiddleware

import os

import json

from loguru import logger

from send\_monitor\_event import send\_monitor\_event # Длямониторингазадач

# 1. Инициализация FastAPI

app = FastAPI()

# 2. Настройка CORS

app.add\_middleware(

CORSMiddleware,

allow\_origins=["\*"],

allow\_methods=["\*"],

allow\_headers=["\*"],

)

# 3. Пути

BASE\_DIR = "/srv/ai\_server"

QUEUE\_FILE = os.path.join(BASE\_DIR, "queues", "ai\_queue.jsonl")

RESULTS\_DIR = os.path.join(BASE\_DIR, "results")

RECEIVE\_LOG = os.path.join(BASE\_DIR, "logs", "api\_receive.log")

ERROR\_LOG = os.path.join(BASE\_DIR, "logs", "errors.log")

os.makedirs(os.path.dirname(QUEUE\_FILE), exist\_ok=True)

os.makedirs(os.path.dirname(RECEIVE\_LOG), exist\_ok=True)

logger.add(RECEIVE\_LOG, level="INFO", rotation="500 KB")

logger.add(ERROR\_LOG, level="ERROR", rotation="500 KB")

# 4. API-эндпоинт

@app.post("/api/analyze\_transcript")

async def analyze\_transcript(

track\_id: str = Form(...),

text: str = Form(...),

timestamp: str = Form(...),

audio: UploadFile = File(...)

):

try:

# 4.1 Валидация обязательных полей

if not track\_id.strip() or not text.strip() or not timestamp.strip():

raise HTTPException(status\_code=400, detail="Missing required fields")

# 4.2 Созданиепапкирезультата

result\_dir = os.path.join(RESULTS\_DIR, track\_id)

os.makedirs(result\_dir, exist\_ok=True)

# 4.3 Сохранениеаудиофайла

audio\_path = os.path.join(result\_dir, "audio.mp3")

with open(audio\_path, "wb") as f:

content = await audio.read()

f.write(content)

# 4.4 Формированиезадачи

task = {

"track\_id": track\_id,

"text": text,

"metadata": {

"timestamp": timestamp

}

}

with open(QUEUE\_FILE, "a", encoding="utf-8") as f:

f.write(json.dumps(task, ensure\_ascii=False) + "\n")

# 4.5 Отправка события в мониторинг

send\_monitor\_event(

track\_id=track\_id,

status="queued\_for\_analysis",

comment="Задача получена и поставлена в очередь на анализ"

)

# 4.6 Логуспешногоприёма

logger.info(f"[RECEIVED] track\_id: {track\_id}")

return JSONResponse(content={"status": "queued"}, status\_code=200)

except Exception as e:

logger.error(f"[ERROR] analyze\_transcript | {str(e)}")

raise HTTPException(status\_code=500, detail="Server error")

# 5. Запуск standalone

if \_\_name\_\_ == "\_\_main\_\_":

import uvicorn

uvicorn.run("analyze\_api:app", host="0.0.0.0", port=11435, reload=True)

## 📋Что реализовано (обновлённое)

| **Этап** | **Действие** | **Статус** |
| --- | --- | --- |
| 1 | Приём multipart от TR-Routing | ✅ |
| 2 | Валидацияполей: track\_id, text, timestamp, audio | ✅ |
| 3 | Сохранение аудиофайла в /results/<track\_id>/audio.mp3 | ✅ |
| 4 | Сохранение задачи в очередь .jsonl | ✅ |
| 5 | Логирование событий | ✅ |
| 6 | Отправка статуса в мониторинг | ✅ |

# **5: Реализация анализатора (analysis\_worker.py)**

## 📌Архитектура:

• Запускает 4 процесса через multiprocessing.Process.  
• Использует Ollama (DeepSeek-V3 32B) через REST API.  
• Выполняет логирование:

* reasoning финальногоответа (log\_final\_aggregation). • Сохраняет результаты анализа в /storage/:
* JSON-ответ.
* reasoning лог. • Формирует итоговую структуру /results/TASK-<track\_id>/.  
  • Логирует все этапы обработки в logs/analysis.log.  
  • При ошибках логирует в logs/errors.log.  
  • ❌ Не отправляет статусы в AI-Monitor.

## 🧠 Новый алгоритм каждого потока:

1. Читает первую задачу из ai\_queue.jsonl.
2. Передаёт текст в Ollama для генерации (run\_ollama\_generation).
3. Сохраняет:
   * результат анализа в storage/ai\_results/,
   * reasoning вstorage/reasoning\_logs/.
4. Логирует успешную обработку задачи в logs/analysis.log.
5. При ошибках записывает их в logs/errors.log.

## ✅analysis\_worker.py

import os

import time

import json

import multiprocessing

from loguru import logger

from ollama\_api import run\_ollama\_generation

from log\_aggregation import log\_final\_aggregation

# 1. Пути

BASE\_DIR = "/srv/ai\_server"

QUEUE\_FILE = os.path.join(BASE\_DIR, "queues", "ai\_queue.jsonl")

RESULT\_DIR = os.path.join(BASE\_DIR, "storage", "ai\_results")

REASONING\_DIR = os.path.join(BASE\_DIR, "storage", "reasoning\_logs")

AI\_LOG = os.path.join(BASE\_DIR, "logs", "analysis.log")

ERROR\_LOG = os.path.join(BASE\_DIR, "logs", "errors.log")

# 2. Логирование

logger.add(AI\_LOG, level="INFO", rotation="1 MB")

logger.add(ERROR\_LOG, level="ERROR", rotation="1 MB")

# 3. Обработкаоднойзадачи

def process\_task(task\_line: str):

try:

task = json.loads(task\_line)

track\_id = task["track\_id"]

text = task["text"]

timestamp = task["metadata"]["timestamp"]

logger.info(f"[START] track\_id ={track\_id}")

# 3.1 Генерациячерез Ollama

response\_text, moe\_info = run\_ollama\_generation(

prompt=text,

track\_id = track\_id

)

# 3.2 Сохранение reasoning

os.makedirs(REASONING\_DIR, exist\_ok=True)

reasoning\_data = {

" track\_id": track\_id,

"timestamp": timestamp,

"text\_original": text,

"response": response\_text,

"moe\_info": moe\_info

}

reasoning\_path = os.path.join(REASONING\_DIR, f"{track\_id}.json")

with open(reasoning\_path, "w", encoding="utf-8") as f:

json.dump(reasoning\_data, f, indent=2, ensure\_ascii=False)

# 3.3 Сохранениерезультатаанализа

os.makedirs(RESULT\_DIR, exist\_ok=True)

result\_data = {

" track\_id": track\_id,

"timestamp": timestamp,

"analysis": {

"System Data": "",

"Contract": "", # Показаполняетсяпозже

"Debt Amount (RUB)": "",

"Identification": "",

"Dialogue Duration (min)": "",

"Payment Amount": "",

"Payment Date": "",

"Effectiveness (1-5)": "",

"Comments": "",

"Debtor Psychological Analysis": "",

"Employee Psychological Analysis": "",

"Employee Effectiveness": "",

"Final Analysis": ""

}

}

result\_path = os.path.join(RESULT\_DIR, f"{track\_id}.json")

with open(result\_path, "w", encoding="utf-8") as f:

json.dump(result\_data, f, indent=2, ensure\_ascii=False)

logger.info(f"[ANALYZED] track\_id ={ track\_id }")

except Exception as e:

logger.error(f"[ERROR] task processing failed: {str(e)}")

# 4. Циклодноговоркера

def worker\_loop(worker\_id):

logger.info(f"[WORKER {worker\_id}] started.")

while True:

try:

if not os.path.exists(QUEUE\_FILE):

time.sleep(1)

continue

with open(QUEUE\_FILE, "r", encoding="utf-8") as f:

lines = f.readlines()

if not lines:

time.sleep(1)

continue

task\_line = lines[0]

remaining\_lines = lines[1:]

# Перезаписываем очередь без обработанной задачи

with open(QUEUE\_FILE, "w", encoding="utf-8") as f:

f.writelines(remaining\_lines)

process\_task(task\_line)

except Exception as e:

logger.error(f"[WORKER {worker\_id}] unexpected error: {str(e)}")

time.sleep(3)

# 5. Запускворкеров

def start\_workers():

processes = []

for i in range(4): # 4 параллельныхпроцесса

p = multiprocessing.Process(target=worker\_loop, args=(i,))

p.start()

processes.append(p)

for p in processes:

p.join()

# 6. Точказапуска standalone

if \_\_name\_\_ == "\_\_main\_\_":

start\_workers()

# 📋Что реализовано в обновлённой версии:

| **Этап** | **Действие** | **Статус** |
| --- | --- | --- |
| 1️⃣ | Чтение задач из очереди | ✅ |
| 2️⃣ | Генерация ответа через Ollama | ✅ |
| 3️⃣ | Сохранение reasoning (JSON) | ✅ |
| 4️⃣ | Сохранение результата анализа (структурированный JSON analysis) | ✅ |
| 5️⃣ | Логирование этапов в analysis.log | ✅ |
|  |  |  |

## 📦log\_aggregation.py → log\_final\_aggregation()

📌Этот модуль логирует результат работы всей модели:  
финальный ответ + детали сборки из сегментов, сгенерированных разными экспертами (если MoE используется с агрегацией на уровне сегментов).

Онсоздаётфайлlogs/reasoning/< track\_id>.json.

**✅Код: log\_aggregation.py**

import os

import json

import yaml

from datetime import datetime

# === 1. Загрузка конфигурации ===

with open("config.yaml", "r") as f:

config = yaml.safe\_load(f)

reasoning\_log\_dir = config["log\_paths"]["reasoning"]

log\_format = config.get("log\_format", {})

indent = log\_format.get("indent", 2)

ensure\_ascii = log\_format.get("ensure\_ascii", False)

# === 2. log\_final\_aggregation ===

def log\_final\_aggregation(response\_text: str, moe\_info: dict, track\_id: str):

"""

Логирует финальный сгенерированный ответ + вклад каждого эксперта.

Формирует структуру:

- track\_id

- response

- expert\_contributions

- timestamp

"""

expert\_contributions = moe\_info.get("expert\_contributions", {})

if not expert\_contributions:

expert\_contributions = {}

log\_data = {

" track\_id": track\_id,

"timestamp": datetime.utcnow().isoformat(),

"response": response\_text,

"expert\_contributions": expert\_contributions

}

os.makedirs(reasoning\_log\_dir, exist\_ok=True)

log\_path = os.path.join(reasoning\_log\_dir, f"{ track\_id }.json")

with open(log\_path, "w", encoding="utf-8") as f:

json.dump(log\_data, f, indent=indent, ensure\_ascii=ensure\_ascii)

**📂Пример результата:**

{

"track\_id": "ab12-456c",

"timestamp": "2025-04-10T11:45:00Z",

"response": "Анализ звонка: Клиент сообщил, что не может оплатить до 15 числа...",

"expert\_contributions": {

"2": {

"segments": [0, 1, 2],

"weight": 0.58

},

"5": {

"segments": [3, 4],

"weight": 0.42

}

}

}

**🧠 Что делает этот модуль:**

| **Этап** | **Описание** |
| --- | --- |
| Загружает конфигурацию | Пути и форматирование из config.yaml |
| Принимает финальный ответ | Текст response, полученный от Ollama |
| Извлекает вклад экспертов | Изmoe\_info["expert\_contributions"] |
| Добавляет timestamp | В ISO-формате (UTC) |
| Сохраняет лог | Вlogs/reasoning/< track\_id>.json |

Отлично, завершаем блок вспомогательных модулей.  
Сейчас реализуем:

**📦ollama\_api.py → run\_ollama\_generation()**

📌Этот модуль отвечает за:

* отправку текста (prompt) на Ollama-сервер через REST API,
* получение финального ответа модели,
* извлечениеполнойmoe\_info: routing, weights, expert\_outputs, contributions идр.

**✅Код: ollama\_api.py**

import requests

import yaml

import json

# === 1. Загрузка конфигурации ===

with open("config.yaml", "r") as f:

config = yaml.safe\_load(f)

ollama\_url = config["ollama\_url"]

model\_name = config["model\_name"]

# === 2. Запрос к Ollama через RESTAPI

def run\_ollama\_generation(prompt: str, track\_id: str):

"""

Отправляет текст на Ollama, получает:

- response: итоговый ответ модели

- moe\_info: логика маршрутизации, веса, ответы экспертов

"""

endpoint = f"{ollama\_url}/api/generate"

payload = {

"model": model\_name,

"prompt": prompt,

"options": {

"moe\_log": True,

"stream": False,

"temperature": config["moe\_control"]["sharpen\_temperature"],

"dropout": config["moe\_control"]["dropout\_rate"],

" track\_id": track\_id # Теперьпередаёмtrack\_id, чтобылогироватьинференс

}

}

try:

res = requests.post(endpoint, json=payload, timeout=60)

res.raise\_for\_status()

data = res.json()

response\_text = data.get("response", "")

moe\_info = data.get("moe\_info", {})

return response\_text, moe\_info

except Exception as e:

print(f"[ERROR] Ollama API failed: {e}")

return "", {}

**📥Чтоотправляется:**

{

"model": "deepseek-custom",

"prompt": "Клиент отказывается платить и говорит...",

"options": {

"moe\_log": true,

"stream": false,

"temperature": 0.7,

"dropout": 0.1,

" track\_id": "ab12-456c"

}

}

**📤Что возвращает:**

{

"response": "Клиент сообщил, что сможет внести платёж...",

"moe\_info": {

"routing": {...},

"experts\_summary": {...},

"expert\_outputs": {...},

"expert\_contributions": {...}

}

}

* структура данных изменилась,
* вызов функции run\_ollama\_generation теперь принимает track\_id.

# 📘ollama\_api.py

# 🧠Что делает этот модуль:

| **Этап** | **Описание** |
| --- | --- |
| Загружает конфиг | Берёт URL Ollama и имя модели (например, deepseek-custom) из config.yaml. |
| Отправляет prompt | Отправляет текст (prompt) на Ollama через POST-запрос к /api/generate. |
| Получает response | Получает финальный текстовый ответ модели (response). |
| Извлекает moe\_info | Извлекаетполныеданные MoE-обработки: routing, experts\_summary, expert\_outputs, expert\_contributions. |
| Возвращает в Python | Возвращает пару: response\_text, moe\_info для дальнейшей обработки. |

# 📦Где используется этот модуль:

* Вanalysis\_worker.py, вблоке:

response\_text, moe\_info = run\_ollama\_generation(prompt=text, track\_id = track\_id)

# 🎉Все вспомогательные модули теперь реализованы:

| **Модуль** | **Назначение** |
| --- | --- |
| log\_aggregation.py | Логирует финальный ответ модели (response) и вклад экспертов (expert\_contributions) в формате reasoning JSON по track\_id. |
| ollama\_api.py | Отправляет текст (prompt) на Ollama сервер, получает response и moe\_info. |

# ✅Что реализовано:

| **Этап** | **Описание** | **Статус** |
| --- | --- | --- |
| 1️⃣ | Чтение задач из очереди .jsonl | ✅ |
| 2️⃣ | Запуск инференса в Ollama (DeepSeek) | ✅ |
| 3️⃣ | Получение и логирование ответа модели (response) | ✅ |
| 4️⃣ | Извлечение и логирование полной информации MoE (moe\_info) | ✅ |
| 5️⃣ | Логирование reasoning структурывфайлы/storage/reasoning\_logs/<<track\_id>>.json | ✅ |

# 📢Итог:

✅Вспомогательные модули теперь упрощены,  
✅ Полностью соответствуют новой задаче,  
✅ Готовы к масштабированию и надёжной работе.

## 🔹6.1 Формирование единого reasoning JSON

📌Что логируется:

* track\_id, timestamp;
* Оригинальный текст звонка (text\_original);
* Финальный ответ нейросети (response);
* Полная информация о маршрутах токенов, активности экспертов, тексты экспертов (moe\_info);
* Reasoning Summary:
  + основной эксперт,
  + итоговый routing\_score,
  + комментарий модели.

📂Пример reasoning JSON:

{

"track\_id": "a1b2c3d4-5678-9101",

"timestamp": "2025-04-26T10:30:00",

"text\_original": "Здравствуйте, я звоню узнать про оплату...",

"response": "Клиент сообщил, что оплатит до 5 числа.",

"moe\_info": {

"routing": {...},

"experts\_summary": {...},

"expert\_outputs": {...},

"expert\_contributions": {...}

},

"reasoning\_summary": {

"main\_expert\_id": 2,

"routing\_score": 0.87,

"dominant\_expert\_comment": "Оплата ожидается до пятого числа."

}

}

🗂️ Сохраняется:

* /storage/reasoning\_logs/<track\_id>.json

## 🔹6.2 Сохранение результата анализа нейросети

📌Что сохраняется:

* track\_id, timestamp;
* Финальный текст анализа (response\_text).

📂Пример ai\_result JSON:

{

"track\_id": "a1b2c3d4-5678-9101",

"timestamp": "2025-04-26T10:30:00",

"analysis": "Клиент сообщил, что оплатит до 5 числа."

}

🗂️ Сохраняется:

* /storage/ai\_results/<track\_id>.json

## 🔹6.3 Логика хранения

| **Тип логирования** | **Путь сохранения** |
| --- | --- |
| Reasoning (полный) | /storage/reasoning\_logs/<track\_id>.json |
| Финальный результат анализа | /storage/ai\_results/<track\_id>.json |

## 📦Порядок действий в analysis\_worker.py:

1. Запроск Ollama черезrun\_ollama\_generation().
2. Сборка reasoning JSON черезlog\_final\_aggregation(response\_text, moe\_info, track\_id).
3. Сохранение результата анализа.
4. Отправка reasoning в мониторинг.
5. Логирование этапов в logs/analysis.log.

## ✅Что больше ****не используется****:

| **Старые модули** | **Статус** | **Причина** |
| --- | --- | --- |
| log\_tokenization() | ❌ | Токенизация встроена в moe\_info |
| log\_routing\_and\_scores() | ❌ | Routing встроен в moe\_info |
| log\_expert\_outputs() | ❌ | Выводы экспертов встроены в moe\_info |

# 📢ИТОГ

✅Теперь всё логирование происходит **через единый reasoning JSON**, полностью отражая всю работу нейросети над каждым звонком,  
что позволяет:

* строить Excel-отчёты,
* проводить аналитику,
* контролировать качество MoE,
* интегрировать в мониторинг и 1С.

# **📋7 📊 Формирование результатов и структуры файлов на диске**

## 📒Цель:

Формирование итоговой структуры хранения данных для каждого звонка, обработанного сервером **Neuro-ACT**, в единой папке с последующей генерацией сводного отчёта в формате **Excel**.  
Все материалы звонка должны быть упорядочены по track\_id, обеспечивая полную трассировку результатов.

## 📂Структура итоговой папки /results/<track\_id>/

| **Файл** | **Описание** |
| --- | --- |
| audio.mp3 | Оригинальный аудиофайл звонка (копируется из TR-Routing) |
| transcript.txt | Расшифрованный текст звонка |
| ai\_result.json | JSON-ответ нейросети (финальный вывод по звонку) |
| reasoning.json | Reasoning JSON: маршруты токенов, эксперты, reasoning |
| summary.xlsx | Сводный Excel-отчёт по звонку |

## 📋Этапы формирования структуры:

### 7.1 Сохранение транскрибированного текста и JSON-ответа

* Файл transcript.txt создаётся на основе текста транскрипции, полученной из TR-Routing.
* Файл ai\_result.json создаётся на основе результата нейросети (storage/ai\_results/<track\_id>.json).

### 7.2 Сохранение логов MoE (токенизация, маршруты, работа экспертов)

* Файлreasoning.jsonпереноситсяизstorage/reasoning\_logs/<track\_id>.json.
* Он включает:
  + Полную карту маршрутизации токенов,
  + Выводы экспертов,
  + Финальный ответ,
  + Reasoning summary (итоговая логика обработки звонка).

### 7.3 Сборка сводного Excel-отчёта

* Генерируется файл summary.xlsx.
* В Excel сохраняются:
  + track\_id, timestamp;
  + Исходный текст (text\_original);
  + Ответ нейросети (response);
  + Основной эксперт (main\_expert\_id);
  + Routing score (routing\_score);
  + Итоговый комментарий reasoning.

Структура Excel будет стандартизирована для удобной последующей загрузки в CRM-систему или мониторинг.

## 📋Итоговый вид итоговой папки:

/results/DC-10458\_CALL-a1b2c3d4/

├── audio.mp3

├── transcript.txt

├── ai\_result.json

├── reasoning.json

├── summary.xlsx

# 📢Основные требования к результату:

* Все файлы для одного звонка хранятся в **одной папке**/results/< track\_id>/.
* Reasoning JSON формируется по утверждённой структуре без разрозненных логов.
* Все пути прописываются в конфигурационном файле config.yaml для централизованного управления.
* Структура данных должна быть готова к последующей автоматической обработке.

# **📋7.4 Проектирование структуры summary.xlsx**

## 📒Цель:

Создать понятный, стандартизированный **Excel-отчёт** (summary.xlsx) для каждого звонка, который:

* Чётко отражает все этапы обработки звонка нейросетью;
* Включает ключевую информацию для анализа и отчётности;
* Является готовым для последующей интеграции в CRM, мониторинг или аналитику.

## 📂Структура таблицы summary.xlsx:

| **Столбец** | **Описание данных** |
| --- | --- |
| № | Порядковый номер строки (для удобства) |
| Track ID | Уникальный идентификатор звонка (track\_id) |
| Timestamp | Время обработки звонка (timestamp) |
| Original Text | Исходный текст звонка (text\_original) |
| AI Response | Ответ нейросети (response) |
| Main Expert ID | ID основного эксперта, который доминировал при обработке (main\_expert\_id) |
| Routing Score | Итого confidence маршрутизациитокенов (routing\_score) |
| Reasoning Summary | Итоговый комментарий reasoning (пояснение результата) |
|  |  |
|  |  |

## 📄Пример строки в summary.xlsx:

| **№** | **Contract ID** | **Track ID** | **Task ID** | **Timestamp** | **Original Text** | **AI Response** | **Main Expert ID** | **Routing Score** | **Reasoning Summary** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | DC-10458 | DC-10458\_CALL-a1b2c3d4 | a1b2c3d4-5678 | 2025-04-26T10:30:00 | "Здравствуйте, хотел уточнить дату оплаты..." | "Клиент подтвердил оплату до 5 числа." | 2 | 0.87 | "Основное мнение: клиент оплатит до пятого числа." |

# 📢Источник данных для каждого поля:

| **Поле** | **Где берётся** |
| --- | --- |
| Track ID | Из reasoning JSON (track\_id) |
| Timestamp | Из reasoning JSON (timestamp) |
| Original Text | Из reasoning JSON (text\_original) |
| AI Response | Из reasoning JSON (response) |
| Main Expert ID | Из reasoning JSON (reasoning\_summary.main\_expert\_id) |
| Routing Score | Из reasoning JSON (reasoning\_summary.routing\_score) |
| Reasoning Summary | Из reasoning JSON (reasoning\_summary.dominant\_expert\_comment) |
|  |  |

# 📦Планируемые шаги для генерации summary.xlsx:

1. Прочитатьфайлreasoning\_logs/<track\_id>.json.
2. Извлечь необходимые поля.
3. Создать новую строку в таблице Excel.
4. Сохранить Excel-файл summary.xlsx в папку /results/<track\_id>/.

# 📋ИТОГ: мини-структура итоговой папки после сборки будет:

/results/DC-10458\_CALL-a1b2c3d4/

├── audio.mp3

├── transcript.txt

├── ai\_result.json

├── reasoning.json

├── summary.xlsx <-- автоматическисгенерированныйотчёт

Отлично!  
Идём дальше чётко и системно.

# **📋7.5 Проектирование скрипта result\_builder.py**

## 📒Цель:

Создать модуль **result\_builder.py**, который:

* Автоматически формирует итоговую структуру папки /results/<track\_id>/;
* Перемещает или копирует все нужные файлы: аудио, текст транскрипции, ai\_result.json, reasoning.json;
* Генерирует **сводный Excel-файл summary.xlsx** на основе данных из reasoning.

# 📦Логикаработы result\_builder.py:

## Этап 1: Получение путей и данных

* Получаемtrack\_id.
* Ищем:
  + Аудиофайл audio.mp3 (откуда-то из TR-Routing или указанной папки),
  + Транскрипцию transcript.txt,
  + Результатанализа/storage/ai\_results/<track\_id>.json,
  + Reasoning /storage/reasoning\_logs/<track\_id>.json.

## Этап 2: Создание итоговой папки

* Создаём папку /results/<track\_id>/, если её ещё нет.

## Этап 3: Перенос/копирование файлов

* Копируем:
  + audio.mp3 → в итоговую папку,
  + transcript.txt → в итоговую папку,
  + ai\_result.json → в итоговую папку,
  + reasoning.json → в итоговую папку.

## Этап 4: Генерация Excel-файла summary.xlsx

* Извлекаем из reasoning JSON все нужные поля:
  + track\_id,
  + timestamp,
  + text\_original,
  + response,
  + main\_expert\_id,
  + routing\_score,
  + dominant\_expert\_comment.
* Создаём таблицу с этими данными.
* Сохраняем её как summary.xlsx в ту же итоговую папку.

# 📋result\_builder.py

import os

import shutil

import json

import pandas as pd

# 1. Пути

BASE\_DIR = "/srv/ai\_server"

STORAGE\_DIR = os.path.join(BASE\_DIR, "storage")

RESULTS\_DIR = os.path.join(BASE\_DIR, "results")

def build\_result\_folder(track\_id: str):

try:

# 2. Пути к исходным файлам

reasoning\_path = os.path.join(STORAGE\_DIR, "reasoning\_logs", f"{track\_id}.json")

ai\_result\_path = os.path.join(STORAGE\_DIR, "ai\_results", f"{track\_id}.json")

transcript\_path = os.path.join(RESULTS\_DIR, track\_id, "transcript.txt")

audio\_path = os.path.join(RESULTS\_DIR, track\_id, "audio.mp3")

# 3. Проверка наличия файлов

if not os.path.exists(reasoning\_path):

raise FileNotFoundError(f"Reasoning файл не найден: {reasoning\_path}")

if not os.path.exists(ai\_result\_path):

raise FileNotFoundError(f"AI Result файл не найден: {ai\_result\_path}")

if not os.path.exists(transcript\_path):

raise FileNotFoundError(f"Transcript файл не найден: {transcript\_path}")

if not os.path.exists(audio\_path):

raise FileNotFoundError(f"Audio файл не найден: {audio\_path}")

# 4. Создание итоговой папки

result\_dir = os.path.join(RESULTS\_DIR, track\_id)

os.makedirs(result\_dir, exist\_ok=True)

# 5. Копирование файлов

shutil.copy(reasoning\_path, os.path.join(result\_dir, "reasoning.json"))

shutil.copy(ai\_result\_path, os.path.join(result\_dir, "ai\_result.json"))

# Файлы transcript.txt и audio.mp3 уже должны быть в result\_dir изначально.

# 6. Генерация Excel-отчёта

create\_summary\_excel(reasoning\_path, result\_dir)

print(f"[SUCCESS] Папка результата сформирована для track\_id={track\_id}")

except Exception as e:

print(f"[ERROR] Ошибка при сборке результата: {e}")

def create\_summary\_excel(reasoning\_path: str, result\_dir: str):

try:

with open(reasoning\_path, "r", encoding="utf-8") as f:

reasoning = json.load(f)

data = {

"Track ID": reasoning.get("track\_id", ""),

"Timestamp": reasoning.get("timestamp", ""),

"Original Text": reasoning.get("text\_original", ""),

"AI Response": reasoning.get("response", ""),

"Main Expert ID": reasoning.get("reasoning\_summary", {}).get("main\_expert\_id", ""),

"Routing Score": reasoning.get("reasoning\_summary", {}).get("routing\_score", ""),

"Reasoning Summary": reasoning.get("reasoning\_summary", {}).get("dominant\_expert\_comment", "")

}

df = pd.DataFrame([data])

df.index += 1 # Нумерация строк начинается с 1

summary\_path = os.path.join(result\_dir, "summary.xlsx")

df.to\_excel(summary\_path, index\_label="№")

except Exception as e:

print(f"[ERROR] Ошибка при генерации Excel: {e}")

# 7. Пример запуска standalone

if \_\_name\_\_ == "\_\_main\_\_":

test\_track\_id = "example\_track\_id" # сюда вставить реальный track\_id для теста

build\_result\_folder(test\_track\_id)

# **🔷8. Централизованный запуск всех компонентов системы (main.py)**

📌Назначение: Организовать единый и автоматический запуск всех основных компонентов сервера ИИ **(Neuro-ACT)**, чтобы система:

• Принимала запросы от транскрибатора через /api/analyze\_transcript; • Обрабатывала тексты с логикой MoE и токенизацией; • Сохраняла reasoning-логи в структуре /storage/reasoning\_logs; • Логировала статусы обработки по track\_id в AI-Monitor; • Обеспечивала стабильную параллельную обработку задач; • Работала непрерывно как единый сервис.

✅Архитектура запуска:

| **Компонент** | **Процессов** | **Назначение** |
| --- | --- | --- |
| analyze\_api.py | 1 Thread | FastAPI: приёмзапросов/api/analyze\_transcriptот TR-Routing |
| analysis\_worker.py | 4 Process | Обработка текста, токенизация, MoE-анализ, логирование reasoning |
| (планируемо) monitor\_queues\_ai.py | 1 Process | Мониторинг состояния очередей (в перспективе) |
| (планируемо) cleanup\_ai.py | 1 Process | Очистка reasoning-логов и старых файлов (в перспективе) |

✅Особенности:

• Используется multiprocessing.Process для компонентов, работающих параллельно;  
• FastAPI запускается в threading.Thread (не блокирует обработку задач);  
• Reasoning-логи сохраняются сразу в /storage/reasoning\_logs/;  
• Статусы обработки (queued, in\_analysis, ai\_result\_generated, error\_ai\_internal) отправляются в AI-Monitor через send\_monitor\_event;  
• Проект построен с возможностью дальнейшего расширения (мониторинг, очистка).

✅main.py — реализация

import multiprocessing

import threading

import uvicorn

# Импорткомпонентов

from analyze\_api import app

from analysis\_worker import start\_workers

# === 1. FastAPI-сервер

def run\_api():

uvicorn.run(app, host="0.0.0.0", port=11435)

# === 2. Обёртка для запуска анализа

def run\_analysis():

start\_workers()

# === 3. Запуск

if \_\_name\_\_ == "\_\_main\_\_":

processes = []

# FastAPI

threading.Thread(target=run\_api, daemon=True).start()

# Анализаторы

for \_ in range(4):

p = multiprocessing.Process(target=run\_analysis)

p.start()

processes.append(p)

# Поддержкапроцессов

for p in processes:

p.join()

✅Что работает после запуска:

| **Компонент** | **Функция** | **Статус** |
| --- | --- | --- |
| analyze\_api.py | Принимает JSON от TR-Routing | ✅ |
| analysis\_worker.py | Обрабатывает текст, генерирует reasoning JSON | ✅ |
| send\_monitor\_event.py | Отправляет статусы задач в AI-Monitor | ✅ |
| send\_reasoning\_event.py | Отправляет reasoning в AI-Monitor (если нужно) | ✅ |
| main.py | Запускает систему единым блоком | ✅ |

# 📢Примечание:

* Компоненты **result\_dispatcher.py** и **retry\_handler\_ai.py** в данной версии отсутствуют (возможно подключение позже при необходимости).
* Компоненты **monitor\_queues\_ai.py** и **cleanup\_ai.py** проектируются для следующей версии и пока не запускаются.

# **📋9 ✅Проверкасистемы (сервер Neuro-ACT)**

📌Назначение: Проверить работоспособность всех компонентов системы **Neuro-ACT** после централизованного запуска через main.py, чтобы убедиться в корректности:

• Приёма текстов через /api/analyze\_transcript;  
• Обработки текстов нейросетью с токенизацией и MoE-логикой;  
• Формирования reasoning-логов и сохранения результатов;  
• Отправки статусов обработки задач в AI-Monitor;  
• Отсутствия критических ошибок в логах.

✅Этапы проверки:

| **№** | **Этап** | **Описание** | **Ожидаемый результат** |
| --- | --- | --- | --- |
| 1 | Запуск main.py | Запустить систему через python3 main.py | Все процессы стартуют без ошибок |
| 2 | Отправка тестового запроса | Отправить JSON через/api/analyze\_transcript | Статус ответа 200 OK, задача поставлена в очередь |
| 3 | Обработка задачи | Проверить логи анализа | Задача обработана, создан reasoning JSON |
| 4 | Сохранение результатов | Проверитьпапку/storage/ai\_resultsи/storage/reasoning\_logs | Созданы новые файлы: <track\_id>.json |
| 5 | Отправка статусов в мониторинг | Проверить отправку событий в AI-Monitor | Статусыqueued, in\_analysis, ai\_result\_generatedпередаются |
| 6 | Проверка логов ошибок | Проверить logs/errors.log | Нет критических ошибок |
| 7 | Сборка результатов | (Если подключен result\_builder.py) Проверить создание итоговой папки /results/<track\_id>/ | Папка создана, summary.xlsx присутствует |

✅Проверка работоспособности компонентов:

| **Компонент** | **Проверяемая функция** | **Статус** |
| --- | --- | --- |
| analyze\_api.py | Принимает JSON от TR | ✅Проверяетсяпоответу 200 OK |
| analysis\_worker.py | Обработка текста и генерация reasoning | ✅Проверяетсяпосозданиюфайлов |
| send\_monitor\_event.py | Отправка статусов задач в AI-Monitor | ✅Проверяетсяпособытиямвмониторинге |
| storage/ | Сохранение результатов обработки | ✅Проверяетсяпоналичию JSON-файлов |
| logs/analysis.log | Логирование процесса анализа | ✅Проверяетсяналичиезаписей |
| logs/errors.log | Отслеживание ошибок | ✅Проверяетсяотсутствиекритическихошибок |

✅Формат тестового запроса (пример для отправки через curl или Postman):

POST /api/analyze\_transcript HTTP/1.1

Host: 0.0.0.0:11435

Content-Type: application/json

{

"text": "Здравствуйте, хотел бы уточнить дату оплаты...",

"metadata": {

"timestamp": "2025-04-26T10:30:00",

"track\_id": "DC-10458\_CALL-a1b2c3d4"

}

}

✅Ожидаемые пути сохранения файлов:

| **Тип файла** | **Путь** |
| --- | --- |
| JSON-результат анализа | /storage/ai\_results/<track\_id>.json |
| Reasoning JSON | /storage/reasoning\_logs/<track\_id>.json |
| (Если используется) Итоговая папка | /results/<track\_id>/ |

# 📢ИТОГ:

✅После запуска main.py и выполнения теста система должна:

• Принимать запросы от TR-Routing;  
• Обрабатывать задачи в многопоточном режиме;  
• Генерировать reasoning и сохранять результаты;  
• Работать непрерывно без ошибок.