

Lab 6.4 Repeat a Familiar Experiment at Larger Scale

Gavin · AI-assisted course design

Contents

1	Goal	3
2	Background	3
3	Prerequisites	3
4	Workflow overview	4
5	Tasks	4
5.1	Task 1 — Make a prediction you could get wrong	4
5.2	Task 2 — Reuse the verified deployment	4
5.3	Task 3 — Freeze the inputs, controls, and budget	5
5.4	Task 4 — Run the bounded comparison	6
5.5	Task 5 — Read the measurements at their actual boundary	10
5.6	Task 6 — Save resource evidence and finish cleanup	10
6	What did we just do?	11
7	More Learning	12

1 Goal

Repeat the measurement habits of Lab 4.1 — Measure Inference and Sampling using the `openai/gpt-oss-20b` deployment you established in Lab 6.3. Change one generation setting, retain every attempt, and write one modest conclusion about this particular run.

Time: about 45–60 minutes for predictions, analysis, and explanation. The new request sequence has a 15-minute wall-clock limit. Complete preparation before renting compute and finish analysis after shutting it down.

Deliverable: one frozen plan, complete request and response records, a small timing summary, resource evidence, cleanup evidence, and a 250–400-word report. A documented blocked or partial run is useful evidence, but is not a completed measurement.

! Execution status and cost boundary

The interface and arithmetic below have been reviewed against pinned source. This Lab does not supply measured GPU results: its AWS launch, model loading, and generation sequence have not been executed as part of authoring. Syntax or source checks cannot establish hardware compatibility or successful inference.

Continue directly from Lab 6.3 only if you deliberately included this Lab within that session’s time and spending limit. Do not keep a GPU instance running while writing the report. If the earlier session has ended, follow Lab 6.3’s complete launch and preflight procedure again; this page does not authorize a larger machine, a longer session, or automatic retries.

2 Background

In Lab 4.1, you separated token selection, repeated computation, and the boundaries of a timing measurement. Here you reuse those ideas on a larger specimen, without repeating the whole earlier Lab or transplanting its GPT-NeoX instrumentation.

The question is deliberately small: **with the same prompt and runtime, what changes when the maximum completion budget rises from 64 to 128 tokens?** Observe response duration, actual output length, stopping reason, and repeatability. A larger allowance need not produce a longer answer: ordinary end-of-sequence stopping remains enabled. It also need not produce a visible final answer, because `gpt-oss` may spend the allowance generating reasoning text.

This is a real-model **inference** experiment. It is not a real-model routing experiment. In Lab 2.6, the complete routing trace belonged to a small arithmetic toy; reading the `gpt-oss` configuration did not turn that trace into trained-model evidence. The server used here returns text, tokens, and runtime information, not the layer-by-layer expert assignments required to repeat that routing investigation.

3 Prerequisites

- Your Lab 4.1 records and its timing definitions from Lesson 4.1.
- The experimental controls discussed in Lesson 6.1.

- A verified Lab 6.3 session with enough time and spending allowance remaining. Reuse your Lab 1.4 workbench record and Lab 6.3's evidence-transfer and cleanup procedure.

4 Workflow overview

1. Read your earlier measurements and write predictions before inspecting new outputs.
2. Confirm the existing model, runtime, resource allowance, and shutdown deadline.
3. Freeze two prompts, the generation controls, and the complete request schedule.
4. Run four warm-ups and twelve measured requests on the same loaded server.
5. Save the evidence, shut down paid resources, then compare the records and explain their limits.

5 Tasks

5.1 Task 1 — Make a prediction you could get wrong

Save `predictions.md` before sending any request. Answer briefly:

- Will the 128-token condition take longer for both prompts? What result would change your mind?
- If both conditions stop naturally after the same number of tokens, what difference do you expect?
- Will three repeated greedy requests produce identical raw token IDs? Why is a fixed seed insufficient to promise this across hardware or builds?
- Could the larger allowance produce more reasoning text without a final answer?
- What could unchanged GPU memory readings tell you, and what could they not tell you?

Copy the timing definitions from your earlier Lab 4.1 report into a comparison note. Mark any missing earlier measurement as missing. You will compare methods and limitations; you are not trying to reproduce the same numerical timings on a different model and machine.

5.2 Task 2 — Reuse the verified deployment

Keep Lab 6.3's runtime, exact specimen, and build:

- llama.cpp source revision `d81235049384534c167caea52b85a694f6103d14`.
- `ggml-org/gpt-oss-20b-GGUF`, revision `ef9b12f2ff56c69cf32153a02784e7a3c88bf524`, file `gpt-oss-20b-MXFP4.gguf`.
- Weight-file SHA-256 `27cd6c432c7672cb812a92f611cf3ba7bbc35928262bb1e1253ff4ee6ae35901`.
- Local server `http://127.0.0.1:8080`, model alias `gpt-oss-20b`, and the same AWS instance. Run the client **on that instance**, so the measured request does not cross your SSH connection.

Reuse the earlier manifest and verified hash; do not download another copy or change the representation. Retain the build command, server command and log, GPU model/VRAM, CPU, driver, operating system, context allocation, offload report, KV-cache types, thread settings, and effective attention backend. The model name or the request to offload all layers is not evidence that offloading succeeded.

Keep Lab 6.3's 4,096-token context, one parallel slot, batch and microbatch settings, low reasoning effort, and disabled cross-request prompt reuse. No training, adapters, tools, agents, internet retrieval, speculative model, or hidden-state collection is needed. Keep the endpoint bound to loopback; do not open its port to the internet.

Disabling cross-request prompt reuse does **not** disable the normal KV cache used while generating one response. This Lab makes no cache-on/cache-off correctness or speed claim.

Before continuing, check the health endpoint, the saved model identity, and the remaining session allowance. The existing 30-minute server supervisor deadline must accommodate this run; do not restart its clock to extend the session. Do not add a model-generation test: Lab 6.3 already supplied that evidence. If verification fails, record `not_run`, export the available evidence, and clean up.

5.3 Task 3 — Freeze the inputs, controls, and budget

Reuse the two exact source strings from Lab 4.1:

- P1: The small red boat crossed the lake.
- P2: A notebook lay beside the window.

For this chat model, place each string in a separate user message and use the fixed system message Continue the user's text with one short sentence. Apply the model's existing chat template through the server. Do not send the earlier plain base-model prompt as if it were the same serialized input.

Save the message JSON, rendered prompt, tokenizer output, and actual prompt-token count. Different tokenizers can divide identical text differently, and the chat wrapper adds tokens. The GGUF representation, trained checkpoint, architecture, post-training, runtime, hardware, precision, and prompt format also differ from Pythia-14M. Any side-by-side examples are illustrative; they cannot isolate model size, establish scaling laws, or rank general accuracy or capability.

Use these controls throughout:

- One independent conversation and one request at a time; no conversation history between trials.
- Greedy selection (`temperature=0`), fixed seed 17, neutral penalties, no extra sampling filters, and unchanged low reasoning effort.
- Normal EOS stopping, with only `max_tokens` changed between 64 and 128. Do not suppress EOS or force a minimum length.
- At most 1,024 serialized input tokens per request, checked before generation. This leaves ample room within the 4,096-token context.
- Four warm-ups: one for each prompt/budget combination. Keep their records, but exclude them from measured summaries.
- Three measured repetitions per combination. Reverse the complete condition order for repetition 2; retain the exact schedule.

There are sixteen generation requests. The sum of their requested output allowances is

$$2 \text{ prompts} \times (64 + 128) \times (1 \text{ warm-up} + 3 \text{ measurements}) = 1,536.$$

This includes reasoning and control tokens, not just visible answer text. The pinned runtime documents a possible small token-limit overrun when completing a partial UTF-8 character. The script records actual counts and stops if a returned count exceeds its request; 1,536 is the **requested allowance**, not a proof of a hard backend limit. Do not silently discard an overrun or increase the allowance.

Allow at most 60 seconds for any HTTP operation and 15 minutes for the entire script. These are safety bounds, not performance promises. Keep the independent instance shutdown deadline from Lab 6.3 in place. A client timeout does not establish that the server stopped computing, and neither a stopped client nor a stopped model process ends EC2 billing.

5.4 Task 4 — Run the bounded comparison

The following client is specific to the pinned llama.cpp build. Prepare lab25_reprise.py locally before starting your paid session, then place an unchanged copy at \$HOME/llmcourse-gpt-oss/lab25_reprise.py on the instance using the terminal/file workflow you already use for course scripts. Compare file hashes before running. It uses only Python's standard library and contacts only the existing loopback server. The verbose and return_tokens options expose the pinned build's diagnostic record, including raw generated IDs; they are not portable OpenAI API fields. Their serialization overhead is included equally in every measured request. Pinned request schema, pinned response implementation

```
import json
import signal
import sys
import time
import urllib.error
import urllib.request
from pathlib import Path

BASE = "http://127.0.0.1:8080"
class NoRedirect(urllib.request.HTTPRedirectHandler):
    def redirect_request(self, req, fp, code, msg, headers, newurl):
        return None

HTTP = urllib.request.build_opener(
    urllib.request.ProxyHandler({}), NoRedirect())
OUT = Path(sys.argv[1])
OUT.mkdir(parents=True, exist_ok=False)
PROMPTS = {
    "P1": "The small red boat crossed the lake.",
    "P2": "A notebook lay beside the window.",
}
COMMON = {
    "model": "gpt-oss-20b", "stream": False,
    "temperature": 0, "seed": 17, "samplers": ["temperature"],
    "top_k": 0, "top_p": 1, "min_p": 0,
    "repeat_penalty": 1, "presence_penalty": 0,
    "frequency_penalty": 0, "cache_prompt": False,
    "reasoning_effort": "low", "ignore_eos": False,
    "verbose": True, "return_tokens": True,
}
conditions = [(p, cap) for p in PROMPTS for cap in (64, 128)]
schedule = [("warmup", 0, p, cap) for p, cap in conditions]
```

```

for rep in (1, 2, 3):
    order = conditions[::-1] if rep == 2 else conditions
    schedule += [("measured", rep, p, cap) for p, cap in order]

def save(name, value):
    (OUT / name).write_text(
        json.dumps(value, indent=2, ensure_ascii=False) + "\n",
        encoding="utf-8")

def expired(signum, frame):
    raise TimeoutError("HTTP wall-clock limit reached")

signal.signal(signal.SIGALRM, expired) # Linux client on the instance
deadline = time.monotonic() + 900
http_index = 0

def post(path, body):
    global http_index
    remaining = deadline - time.monotonic()
    if remaining <= 0:
        raise TimeoutError("15-minute sequence limit reached")
    label = f"http-{http_index:02d}"
    http_index += 1
    save(label + "-request.json", {"path": path, "body": body})
    request = urllib.request.Request(
        BASE + path, data=json.dumps(body).encode("utf-8"),
        headers={"Content-Type": "application/json"})
    raw, code, error_text = b"", None, None
    limit = 1024 * 1024 # Bound retained response bytes to 1 MiB.
    signal.setitimer(signal.ITIMER_REAL, min(60, remaining))
    t0 = time.perf_counter()
    try:
        try:
            with HTTP.open(request, timeout=60) as response:
                code = response.status
                raw = response.read(limit + 1)
        except urllib.error.HTTPError as error:
            code = error.code
            with error:
                raw = error.read(limit + 1)
    except BaseException as error:
        error_text = f"{type(error).__name__}: {error}"
        raise
    finally:
        seconds = time.perf_counter() - t0
        signal.setitimer(signal.ITIMER_REAL, 0)
        (OUT / (label + "-response.bin")).write_bytes(raw[:limit])
        save(label + "-http.json", {

```

```

        "status_code": code, "client_seconds": seconds,
        "response_truncated": len(raw) > limit,
        "transport_error": error_text})
    if len(raw) > limit:
        raise ValueError("Response exceeds 1 MiB; saved bounded prefix")
    if code is None or not 200 <= code < 300:
        raise RuntimeError(f"HTTP failure {code}; raw response retained")
    return json.loads(raw), seconds

def request_for(prompt_id, cap):
    return {**COMMON, "max_tokens": cap, "messages": [
        {"role": "system", "content":
            "Continue the user's text with one short sentence."},
        {"role": "user", "content": PROMPTS[prompt_id]},
    ]}

save("plan.json", {"common": COMMON, "prompts": PROMPTS,
    "schedule": schedule, "requested_output_allowance": 1536,
    "input_limit": 1024, "script": Path(__file__).read_text()})
save("status.json", {"status": "preflight"})
inputs = {}
try:
    for prompt_id in PROMPTS:
        body = request_for(prompt_id, 128)
        rendered, _ = post("/apply-template", body)
        counted, _ = post("/v1/chat/completions/input_tokens", body)
        tokens, _ = post("/tokenize", {
            "content": rendered["prompt"], "add_special": True,
            "parse_special": True, "with_pieces": True})
        inputs[prompt_id] = {"request": body, "rendered": rendered,
            "counted": counted, "tokens": tokens}
        save("inputs.json", inputs)
        n = counted["input_tokens"]
        if not (0 < n <= 1024) or n != len(tokens["tokens"]):
            raise ValueError("Input length/tokenization check failed")

    for index, (phase, rep, prompt_id, cap) in enumerate(schedule):
        body = request_for(prompt_id, cap)
        row = {"index": index, "phase": phase, "rep": rep,
            "prompt_id": prompt_id, "cap": cap, "request": body,
            "status": "started"}
        name = f"trial-{index:02d}.json"
        save(name, row) # A killed client leaves a started record.
        result, seconds = post("/v1/chat/completions", body)
        row.update(response=result, client_seconds=seconds,
            status="received")
        save(name, row) # Preserve evidence before validation.
        usage = result["usage"]

```

```

    detail = result["__verbose"]
    expected = inputs[prompt_id]["counted"]["input_tokens"]
    if usage["prompt_tokens"] != expected:
        raise ValueError("Observed input count changed")
    if detail["prompt"] != inputs[prompt_id]["rendered"]["prompt"]:
        raise ValueError("Observed serialized prompt changed")
    if result["timings"]["cache_n"] != 0:
        raise ValueError("Unexpected cross-request prompt reuse")
    if not 0 <= usage["completion_tokens"] <= cap:
        raise ValueError("Observed completion exceeds allowance")
    if not isinstance(detail["tokens"], list):
        raise ValueError("Raw generated token IDs unavailable")
    row["status"] = "checked"
    save(name, row)
    save("status.json", {"status": "sequence_finished",
                        "analysis": "pending", "cleanup": "pending"})
except BaseException as error:
    save("status.json", {"status": "stopped",
                        "error": f"{type(error).__name__}: {error}",
                        "cleanup": "required"})

raise

```

The formatting and token-count endpoints do not generate model completions. Their results are saved outside the measured sample. Each HTTP operation preserves its request, status, and response bytes before JSON parsing, including completed HTTP error responses. An oversized response retains a labeled prefix and stops; an interrupted transfer may have no retained body and remains incomplete. The generation checks compare the server's observed prompt with the preflight record; do not remove a failing check simply to finish the schedule. Pinned endpoint implementation

Take the before resource snapshot described in Task 6, then run the prepared client from the instance, with the server already healthy and the independent shutdown deadline still active. The outer GNU timeout also bounds the client process; it does not terminate the separate model server:

```

ROOT="$HOME/llmcourse-gpt-oss"
timeout --signal=TERM --kill-after=5s 15m \
    python3 "$ROOT/lab25_reprise.py" "$ROOT/evidence/lab25"

```

Use a new output directory. Place the already-written predictions beside its records afterward without changing their original content, and retain their original timestamp or hash. An existing directory, missing response field, failed check, HTTP error, timeout, or interrupted run is a stop, not an invitation to launch replacement requests. Save the command's exit status, traceback, and server log. A record left as started has an unknown outcome; do not count it as zero tokens or zero cost.

If the client fails or times out, stop the server using Lab 6.3's recorded-process procedure, export the available evidence, and complete instance cleanup. Investigate from saved files. Do not restart the sequence inside the same allowance or automatically upgrade the instance.

5.5 Task 5 — Read the measurements at their actual boundary

For each prompt/budget condition, list all three measured durations and their median, minimum, and maximum. Put the actual completion count and stopping reason beside each duration. Retain the four warm-up records separately. Three repetitions describe variation in this session; they are not a robust estimate of population performance.

`client_seconds` begins immediately before the local HTTP operation and ends after the full response body arrives. It includes local transport, server scheduling, generation, diagnostic serialization, and response transfer; it excludes model loading, request construction, and JSON parsing by the client. It is neither GPU kernel time nor time to first token. Do not add GPU synchronization in the separate client process and claim it synchronizes the server’s device work.

Keep the response’s timings fields under their original names. The server’s prompt and prediction timings have different boundaries from the client’s full-request duration. If you calculate `completion_tokens / client_seconds`, label it **generated tokens per full local request second**. It includes prompt processing and response overhead; its reciprocal is not inter-token latency. Do not turn missing values into zero or silently substitute a different timer.

Compare repeated raw generated token-ID sequences within each condition, keeping the reasoning and final-content fields distinct. Record identical and different sequences alike. A changed seed is not part of this experiment, and greedy selection does not use random sampling in the usual way; the fixed seed documents a control rather than guaranteeing identical floating-point computation. Do not change kernels or determinism settings after seeing a mismatch. Record it as an observation to investigate separately.

For the two budgets, distinguish:

- both runs stopped naturally at comparable lengths;
- one or both exhausted the allowance (`finish_reason` is `length`);
- an empty final-content field with reasoning or other generated tokens present;
- an error or unavailable result.

Do not claim that increasing a cap caused more work if the actual completion lengths did not change. Do not interpret truncated reasoning as a completed answer, or answer fluency as a correctness score. This experiment has no accuracy rubric or task benchmark.

5.6 Task 6 — Save resource evidence and finish cleanup

Use Lab 6.3’s existing resource log and process identity. Take one timestamped GPU-memory snapshot before the request sequence and one after it, outside the timed requests. On its NVIDIA instance, a read-only snapshot can be taken with:

```
nvidia-smi --query-gpu=timestamp,uuid,name,memory.used,memory.total \  
--format=csv  
nvidia-smi --query-compute-apps=pid,process_name,used_memory \  
--format=csv
```

Save both outputs and the command exit status, labeled before or after; match the server PID rather than assigning another process's memory to your model. Preserve N/A or unsupported fields as unavailable. NVIDIA query documentation

These are snapshots of reported device and process usage, not per-request peaks or an isolated KV-cache increment. The server may reserve memory for the configured context before generation, so a flat reading does not show that longer outputs need no storage. The checkpoint's disk bytes, process RAM, GPU allocation, reserved KV buffers, and active expert computations are different quantities. Reuse the earlier loading evidence instead of reloading merely to obtain a prettier baseline.

Export the small Lab 6.3 and Lab 6.4 evidence directories using the access and transfer route you already established. Verify that the local copies open and that their hashes match before removing the instance or its disposable storage. Do not export model weights or include credentials in the report.

Then perform Lab 6.3's full shutdown and cleanup procedure: end the model process, terminate the disposable GPU instance when finished, and check for remaining chargeable volumes, snapshots, addresses, or other resources. Record the observed instance state and retained resources, including why any resource is being kept. Closing the terminal or stopping inference is insufficient. Finish the report locally; reconcile the session's eventual billing record when it becomes available rather than presenting a price estimate as a final bill.

Your report should contain:

1. The question, original predictions, and actual completion status.
2. The frozen controls, specimen/build identity, schedule, and every outcome.
3. The timing summary, raw-token repeatability, truncation evidence, and resource snapshots.
4. One supported conclusion, two limitations, and cleanup evidence.

An appropriately narrow conclusion might take this form: "For these two serialized prompts, on this recorded build and instance, changing the allowance from 64 to 128 produced [the measured change or lack of change]. The counts and stopping reasons show [your evidence]. This does not establish [a named unsupported generalization]." Fill the brackets from your records, not an expected answer.

6 What did we just do?

You returned to a familiar inference experiment with a larger model and a different deployment. The transferable achievement is a bounded comparison whose inputs, settings, measurements, and costs can be checked. It does not establish that the larger model is generally better, that your CPU timings scale predictably to GPUs, or that you observed which experts the model used.

A real MoE-routing extension would need a separate, version-specific instrument that demonstrably exposes router scores or selected expert indices at known layers and token positions. This llama-server workflow supplies no such trace. Do not copy Pythia hooks into it, ask the model to name its experts, or infer routing from answer text. Keep that as a separately scoped question if it genuinely interests you.

Now choose one small question of your own. Reuse the least expensive setup that can answer it, write down what would count as evidence, and begin with a small check before expanding. This Lab gives you another method to draw on; a useful next investigation can still be local, modest, and inconclusive.

7 More Learning

- Lab 4.1 — Measure Inference and Sampling: revisit the difference between sampling, matched-prefix cache checks, and runtime profiling. Only a bounded part of that Lab was repeated here.
- Pinned llama.cpp server reference: check this build's request controls, token counting, timings, and reasoning fields before adapting the client.
- Pinned GGUF artifact repository: preserve the actual representation used rather than treating every file with the model's name as interchangeable.
- gpt-oss Model Card, version 1: distinguish the published architecture and evaluation claims from the much smaller claim supported by your own experiment.
- Lab 2.6 — Trace and Perturb Expert Routing: review the evidence required to turn a configuration description into an actual trained-model routing trace.