Detailed Study Plan – Optimizing Inference for a LoRA-fine-tuned 0.6 B Qwen-2.5 Model

This document extends the original study plan by providing **detailed weekly tasks** along with concrete instructions for the code you need to write. It is designed for engineers who want to **systematically learn and implement** cutting-edge inference optimizations for a small language model (SLM) fine-tuned with **Low-Rank Adaptation (LoRA)**. Each week builds on the previous one, helping you develop a robust benchmarking framework before integrating advanced techniques such as **quantization, continuous batching, key-value caching, speculative decoding** and **hardware scaling**.

Your 0.6 B Qwen-2.5 model (hosted on Hugging Face at codefactory4791/intent-classification-qwen) was fine-tuned to perform intent classification. The evaluation dataset contains 750 diverse user queries across 23 labels, providing a realistic benchmark for latency and accuracy. The histograms below show the distribution of query lengths:

Distribution of query word counts

Distribution of query character counts

Throughout the plan you will write **Python scripts** to load your model, process the evaluation dataset and measure metrics. When working through this plan, store all scripts in a version-controlled repository with folders like week1/, week2/, etc. Each script should accept command-line arguments (e.g., batch size, quantization type) so that you can re-run experiments easily.

Week 1 – Baseline profiling and understanding bottlenecks

Objectives

- Establish a **baseline** for latency, throughput and memory usage.
- Learn to profile the **pre-fill (compute-bound) vs decode (memory-bound)** phases of transformer inference.
- Build an initial **benchmarking script** that will be reused throughout the course.

Reading tasks

1. Read Clarifai's guide on LLM inference optimization up to the discussion on pre-fill vs decode bottlenecks. Note that the pre-fill phase involves large matrix multiplications and is compute-bound, while the decode phase is memory-bound due to KV cache access 【755841117274582†L990-L1000】.
2. Review the Deepsense.ai section on **continuous batching** to understand why static (fixed) batching causes head-of-line blocking 【685976405934923†L345-L362】.
3. Read the RunPod primer's section on computational bottlenecks and hardware utilization 【810860962161577†L210-L244】.

Implementation tasks

1. **Set up your environment.**
   - Create a new Python virtual environment. Install torch, transformers, peft, datasets, bitsandbytes, accelerate, vllm, scipy and pandas. Document versions in a requirements.txt file.
   - Download eval_df.csv (already provided) and load it using pandas. Inspect the distribution of query lengths (word count and character count) to understand typical input sizes.
2. **Write week1_profile.py.** This script should:
   - Load your LoRA-fine-tuned Qwen model using the transformers + peft API. Use AutoModelForSequenceClassification for classification tasks, load the base model and apply the LoRA adapter.
   - Accept command-line arguments for batch size (default 1), maximum sequence length and device (cuda vs cpu). Use argparse for argument parsing.
   - Iterate through the evaluation dataset and compute **per-query latency** (time taken for a single forward pass) and **throughput** (queries per second). Log results to a CSV file with columns: batch_size, num_samples, avg_latency_ms, throughput_qps, accuracy, timestamp.
   - Use the **PyTorch profiler** (torch.profiler.profile) to separate the **pre-fill** and **decode** phases of inference. When using torch.profiler.profile(activities=[ProfilerActivity.CPU, ProfilerActivity.CUDA]), annotate the forward pass so you can identify compute-bound vs memory-bound operations. Save the profiler trace as a .json or .pt file for later analysis.
   - Measure **GPU memory consumption** using torch.cuda.max_memory_allocated() and include it in your CSV.
   - Compute classification **accuracy** (macro F1 or accuracy) using sklearn.metrics. You will reuse this function later to ensure that optimizations do not degrade accuracy.
3. **Write week1_plot_benchmarks.py.** This script will read the CSV generated by week1_profile.py and produce simple plots (latency vs batch size, throughput vs batch size). Use matplotlib and save figures as PNGs. These visuals will help you identify baseline bottlenecks.

What you will measure

- **Latency per query (ms)** and **throughput (queries per second)** as baseline metrics.
- **Pre-fill vs decode time** using the PyTorch profiler; this will reveal whether the model is compute-bound or memory-bound.
- **GPU memory footprint** and **classification accuracy** to ensure no performance regressions.

Record your observations in a week1_report.md file summarizing the baseline results, bottlenecks and next steps.

Week 2 – Model-level optimization: quantization, pruning and distillation

Objectives

- Reduce compute and memory requirements via **quantization** (int8, int4, FP8) and **pruning**.
- Investigate whether a **smaller student model** trained via distillation can meet accuracy targets with lower latency.
- Understand the trade-offs between model size, precision and performance.

Reading tasks

1. Study the Clarifai guide's section on **quantization** 【755841117274582†L684-L700】 and note the typical 2–4× memory reduction and moderate speed-ups when using int8/int4 quantization 【755841117274582†L734-L739】.
2. Read Deepsense.ai's quantization summary: int8/int4 quantization provides ~2× speed-up and 2–4× memory reduction 【685976405934923†L439-L444】.
3. Review Qwen's speed benchmark for FP8, int8 and GPTQ quantizations 【429538402927001†L147-L158】.

Implementation tasks

1. **Write week2_quantize.py.** This script should:
   - Load the model and optionally apply quantization using a library such as bitsandbytes, AutoAWQ or GPTQ. Provide CLI flags like --quantization-type (choices: none, int8, int4, fp8, gptq).
   - For int8/int4 quantization, use the PeftModel.from_pretrained with device_map and load_in_8bit=True (for bitsandbytes) or use AutoAWQ for AWQ quantization. For FP8, use torch.ao.quantization if your GPU supports FP8.
   - After quantization, run the same evaluation loop as in week1_profile.py to measure latency, throughput, GPU memory usage and accuracy. Save metrics to a CSV file (e.g., week2_quant_results.csv).
   - For each quantization level, include a warm-up pass to avoid measuring initialization overhead.
2. **Write week2_prune.py.** This script will:
   - Load the model and apply **magnitude-based pruning** using torch.nn.utils.prune. Provide CLI arguments for the pruning fraction (e.g., --prune-fraction 0.2 to remove 20% of smallest-magnitude weights).
   - Fine-tune the pruned model for a few epochs (optional but recommended) using your training data. This step can be skipped if training resources are limited.
   - Evaluate the pruned model on the evaluation dataset, measuring latency, throughput, memory usage and accuracy. Record results in week2_prune_results.csv.

3. **Optional: Write week2_distill.py.** If you wish to explore knowledge distillation, implement a script that trains a smaller student model to mimic the predictions of your LoRA-fine-tuned model. Use a distillation loss (e.g., KL divergence) to match the teacher's soft labels. Measure the student's latency and accuracy.

What you will measure

- **Latency and throughput** under different quantization schemes and pruning fractions.
- **Memory footprint** reduction relative to the baseline.
- **Accuracy** to ensure that quantization and pruning do not degrade model quality beyond an acceptable threshold (e.g., <1% drop in F1 score).
- In the optional distillation experiment, evaluate whether a smaller student model can achieve similar accuracy with significantly lower latency.

Summarize your findings in week2_report.md, including tables or charts comparing the various quantization/pruning configurations.

Week 3 – Dynamic and continuous batching with vLLM

Objectives

- Implement **dynamic batching** to overcome head-of-line blocking and improve GPU utilization.
- Learn how **vLLM** implements continuous batching with **PagedAttention** and test it on your model.
- Build client-server scripts to simulate real-world request patterns.

Reading tasks

1. Read Deepsense.ai's explanation of **continuous batching** and its trade-offs 【685976405934923†L345-L362】.
2. Study the vLLM documentation and Medium article describing **PagedAttention** and the scheduler architecture【834475903237014†L118-L162】.

Implementation tasks

1. **Write week3_vllm_server.py.** This script should:
   - Start a vLLM OpenAI-compatible API server using your base model with LoRA adapters. Use the command-line interface provided by vLLM: python -m vllm.entrypoints.openai.api_server --model <path-to-model> --dtype auto --enable_lora. Set --max-model-len based on the maximum sequence length in your dataset. If you have multiple LoRA adapters, set --lora-modules accordingly.
   - Expose the server on a configurable port (default 8000). Save logs for later analysis.
2. **Write week3_vllm_client.py.** This script will simulate concurrent requests:

- Generate payloads by converting each evaluation query into the appropriate JSON format for the vLLM server (e.g., {'model': 'qwen', 'prompt': <text>} for classification). Use asynchronous HTTP clients such as aiohttp or httpx.
- Send requests with varying concurrency levels (e.g., 1, 4, 8, 16 parallel requests) and record **per-request latency** and **overall throughput**. Use a Poisson process to simulate random arrival times if you wish to test continuous batching under realistic loads.
- Compare results with and without dynamic batching by toggling --disable-streaming and adjusting the vLLM server's batching parameters (--scheduler-policy, --max-num-seqs etc.).
- Save metrics (latency, throughput, GPU utilization if accessible) into a CSV file week3_vllm_results.csv.

3. **Write week3_static_vs_dynamic.py.** This script will take the metrics from baseline (Week 1) and vLLM experiments to produce comparative plots of latency and throughput vs number of concurrent requests. Use these plots to demonstrate how continuous batching improves throughput at the cost of slightly higher per-request latency.

What you will measure

- **Throughput and latency** under dynamic batching, dynamic arrival patterns and different concurrency levels.
- The effect of **PagedAttention** vs naive attention on GPU memory usage and latency. If your environment supports it, capture GPU metrics via nvidia-smi in a separate monitoring thread.

Document your observations in week3_report.md. Discuss how dynamic batching solves head-of-line blocking and when it may negatively impact latency for individual requests.

Week 4 – Key-value caching, attention kernel optimizations and compilation

Objectives

- Implement **KV caching** in a custom inference loop to reuse attention computations across tokens.
- Explore **PagedAttention**, **FlashAttention** and **FlashInfer** kernels for faster attention.
- Use torch.compile to generate optimized kernels and measure improvements.

Reading tasks

1. Read Deepsense.ai's section on **KV caching** to understand how caching past key and value vectors reduces redundant computation during decoding 【685976405934923†L389-L407】.
2. Study Clarifai's description of **PagedAttention** 【755841117274582†L991-L999】 and note how vLLM partitions KV caches into blocks to reduce memory fragmentation.

3. Read about **FlashAttention** and **FlashInfer** kernels; understand that they reduce memory bandwidth requirements and improve attention throughput 【344230638020752†L90-L104】.

Implementation tasks

1. **Write week4_kvcache.py.** This script will demonstrate manual KV caching:
   - Load your model and evaluation dataset. For each query, perform inference token by token, storing the past_key_values returned by the model. Reuse these cached keys/values when generating subsequent tokens.
   - Compare latency of decoding with and without caching. Use high-length synthetic queries to simulate long contexts. Record average per-token decode time.
   - Output metrics to week4_kvcache_results.csv.
2. **Write week4_attention.py.** This script will benchmark attention kernels:
   - Enable **FlashAttention** if available: install flash-attn (pip install flash-attn in your environment if possible) and set torch.backends.cuda.enable_flash_sdp(True). For transformers, set use_flash_attention_2=True when calling the model. Compare decode time to the default attention implementation.
   - Test **FlashInfer** via vLLM by launching the vLLM server with --attention-type flashinfer. Measure latency improvements on long sequences.
   - Use torch.compile on your model with mode='max-autotune' and set fullgraph=True. Compare compiled vs uncompiled inference. Note any compile overhead when switching LoRA adapters.
   - Save all metrics in week4_attention_results.csv.
3. **Write week4_analysis.py.** Aggregate results from week4_kvcache_results.csv and week4_attention_results.csv into plots summarizing per-token latency across different techniques. Interpret whether KV caching or kernel optimizations provide the biggest gains for your workload.

What you will measure

- **Per-token decode latency** with and without KV caching.
- Speed-ups provided by **FlashAttention**, **FlashInfer** and **torch.compile** relative to baseline attention.
- Impact on **memory usage** and any changes in accuracy due to new kernels (should be negligible). If accuracy drops, verify your implementation.

Document your findings in week4_report.md, including recommendations on which kernel and caching strategies to adopt.

Week 5 – Multi-adapter serving with LoRAX and dynamic LoRA

Objectives

- Learn how to serve multiple LoRA adapters efficiently on one GPU using **LoRAX** or a custom dynamic loading solution.
- Measure the overhead of switching adapters and evaluate memory footprint.
- Test **Turbo LoRA** for multi-token speculative decoding.

Reading tasks

1. Read Predibase's blog on **LoRA eXchange (LoRAX)** to understand how dynamic adapter loading and tiered weight caching enable serving hundreds of LoRA adapters 【529320029173163†L106-L116】.
2. Review how **Turbo LoRA** combines LoRA with speculative decoding to deliver 2–3× higher throughput 【529320029173163†L118-L139】.
3. Study Predibase's comparison of FP8 vs FP16 quantization for LoRA models 【529320029173163†L155-L166】.

Implementation tasks

1. **If you have multiple LoRA adapters**, write week5_lorax_server.py and week5_lorax_client.py:
   - Use the open-source LoRAX or Predibase SaaS to serve your model. Configure it to load LoRA adapters on demand. For open-source LoRAX, follow installation instructions and run the server with your base model and a list of adapters. For the client, send classification requests specifying which adapter to load.
   - Measure **adapter switching latency**, **per-request latency** and **memory usage**. Compare LoRAX to a naive approach where you load a separate model per adapter. Save metrics to week5_lorax_results.csv.
2. **Write week5_dynamic_lora.py** to implement a simplified dynamic LoRA server without LoRAX:
   - Load the base model once and write functions to **merge** and **unmerge** LoRA adapters on the fly using peft.tuners.lora.merge_adapter() and unmerge_adapter(). Use torch.compile with hotswap=True to avoid recompilation 【344230638020752†L151-L162】.
   - Build an asynchronous API (e.g., using fastapi or flask) that accepts requests with an adapter_id parameter. When a new adapter is requested, unmerge the current LoRA weights and merge the requested adapter. Cache frequently used adapters in memory.
   - Measure latency and throughput for adapter switching and classification. Save results in week5_dynamic_lora_results.csv.
3. **Write week5_turbo_lora_test.py** (optional). If you have access to Turbo LoRA adapters or can train one yourself (see Week 6), benchmark Turbo LoRA using vLLM or Predibase. Record throughput and latency improvements relative to standard LoRA. Save metrics to week5_turbo_lora_results.csv.

What you will measure

- **Adapter loading and switching latency** when serving multiple LoRA adapters concurrently.
- **Memory footprint** of serving many adapters using LoRAX vs naive loading.
- **Throughput improvement** offered by Turbo LoRA relative to standard LoRA (if tested).

Write week5_report.md summarizing your experiments and recommending a serving architecture for multi-adapter scenarios.

Week 6 – Speculative decoding and Turbo LoRA

Objectives

- Understand **speculative decoding** as a method to accelerate token generation.
- Train or evaluate a **Turbo LoRA** speculator model to generate multiple tokens per step.
- Compare speculative decoding to standard decoding in terms of latency and accuracy.

Reading tasks

1. Review Clarifai's explanation of speculative inference and its trade-offs 【755841117274582†L768-L804】. Note that acceptance rates depend on the similarity between the draft and main models; low acceptance may reduce benefits.
2. Read Predibase's description of Turbo LoRA and how it combines LoRA with speculative decoding【529320029173163†L118-L139】.

Implementation tasks

1. **Write week6_train_turbo_lora.py** (optional but recommended if you have training resources):
   - Fine-tune your Qwen model with the Turbo LoRA algorithm. This typically involves training a speculator network to predict multiple next tokens, with the base model verifying them. Follow Predibase's training guidelines: configure the speculator to output 2–3 tokens per step and use FP8 quantization to maximize speed.
   - Save the resulting Turbo LoRA adapter and evaluate on your training and evaluation datasets.
2. **Write week6_eval_turbo_lora.py.** This script will:
   - Load the Turbo LoRA adapter and run inference on the evaluation dataset using vLLM or your dynamic server.
   - Compare per-query latency and throughput against your best standard LoRA configuration from previous weeks. Record acceptance rates (how many speculative tokens are actually accepted) if available.
   - Save metrics in week6_turbo_results.csv.

What you will measure

- **Throughput gain** (tokens per second) when using Turbo LoRA vs standard LoRA.

- **Latency per query** and acceptance rates for speculative decoding.
- **Accuracy** on the evaluation dataset to ensure that speculative decoding does not degrade predictions.

Write week6_report.md summarizing your training or evaluation results and drawing conclusions about the practical value of speculative decoding for SLM classification tasks.

Week 7 – Hardware benchmarking and scaling

Objectives

- Understand how **different GPUs** (A100, H100, L40S) affect inference speed and cost.
- Measure the benefits of **FP8 support** and improved memory bandwidth on newer hardware.
- Plan a scaling strategy based on your performance targets and budget.

Reading tasks

1. Review FriendliAI's benchmark showing 4× higher throughput and up to 1.8× lower latency when serving Llama-2-70B on H100 vs A100 GPUs【496322137731782†L31-L79】.
2. Read Qwen's speed benchmark for FP8 and int8 quantizations across different input lengths【429538402927001†L147-L158】.
3. Study the RunPod guide's discussion of GPU TensorCore utilization and memory layout optimization【810860962161577†L210-L244】.

Implementation tasks

1. **Write week7_hardware_benchmark.py.** This script should:
   - Run your best-performing model configuration (e.g., quantized, dynamic batching, KV caching, compiled) on different hardware. If you only have one GPU locally, use cloud providers (e.g., Hugging Face Inference Endpoints, RunPod, or Predibase) to access other GPUs. Provide CLI parameters to specify the target device (e.g., --device a100, --device h100).
   - For each hardware type, measure throughput (tokens per second), average latency per query, memory usage and cost per million tokens (if pricing is known). Use nvidia-smi or vendor APIs to record GPU utilization and memory bandwidth.
   - Save results to week7_hardware_results.csv.
2. **Write week7_scaling_planner.py.** This script will:
   - Read week7_hardware_results.csv and compute cost–performance curves. Create plots of throughput vs cost for different hardware types. Highlight FP8 vs FP16 performance where applicable.
   - Suggest a scaling strategy: for example, if your target latency is <50 ms and throughput of >500 queries/s, identify which GPU(s) and how many instances you need. Include an estimation of monthly cost based on usage.

What you will measure

- **Throughput and latency** on different GPUs.
- **Impact of FP8 support** compared to FP16 or int8 quantization.
- **Cost efficiency** (throughput per dollar). Use approximate GPU pricing from your cloud provider.

Write week7_report.md summarizing your hardware experiments and recommending which GPU(s) to use for your deployment scenario.

Week 8 – Integration, monitoring and autoscaling

Objectives

- Integrate the best optimization techniques into a unified inference service.
- Implement monitoring and **autoscaling** to handle variable workloads.
- Prepare a final report summarizing the performance improvements achieved.

Reading tasks

1. Read additional sections from RunPod on **dynamic resource allocation** and monitoring 【810860962161577†L210-L244】.
2. Study Predibase's notes on autoscaling policies for LoRA inference 【529320029173163†L173-L208】.

Implementation tasks

1. **Write week8_server.py.** This unified server should incorporate:
   - Your preferred serving framework (vLLM, custom FastAPI server, or LoRAX) configured with the optimal model variant (quantized, compiled, with continuous batching, dynamic LoRA loading, etc.).
   - Middleware for **request batching**, **KV caching** and **speculative decoding** if Turbo LoRA is adopted.
   - Logging and metrics collection (e.g., using prometheus_client for Python). Expose metrics such as **request latency**, **throughput**, **GPU utilization** and **error rates** at an endpoint (e.g., /metrics).
2. **Write week8_monitoring_dashboard.py.** This script or notebook will:
   - Deploy a simple dashboard using Grafana or Plotly Dash to visualize metrics collected by your server. Show trends of latency, throughput and GPU utilization over time.
   - Configure alert rules (e.g., send an alert if latency exceeds a threshold or if GPU utilization is below a desired range).
3. **Write week8_autoscaling.py.** If you deploy on a cloud platform that supports autoscaling (e.g., Kubernetes with Horizontal Pod Autoscaler or Predibase/RunPod), implement policies such as scaling up when GPU utilization >70% for 1 minute or when

average latency >100 ms. Document the autoscaling configuration (YAML files or code snippets).

4. **Write week8_final_report.md.** This final report should:

   – Summarize the key findings from each week, including baseline metrics and improvements achieved through each optimization.
   – Present comparative plots (latency vs throughput, memory usage vs accuracy) across all techniques.
   – Provide actionable recommendations for deploying your LoRA-fine-tuned Qwen-2.5 model in production, taking into account cost, latency requirements and scalability.

What you will measure

- **End-to-end latency** of your integrated inference service under realistic workloads.
- **System stability** during load spikes and after autoscaling events.
- **Effectiveness of monitoring and alerting**, verifying that metrics reflect the underlying system state.

The final deliverable for Week 8 is a comprehensive report and a working inference service ready for production experiments.

Concluding remarks

By following this eight-week plan, you will develop not only theoretical knowledge but also a **comprehensive codebase** for serving LoRA-fine-tuned SLMs efficiently. Each week's scripts are designed to be modular so that they can be reused and extended. Make sure to document your experiments thoroughly and version-control your code. The references cited throughout the plan come from up-to-date research and production best practices 【834475903237014†L118-L162】 【529320029173163†L106-L116】.

When generating each script, remember to include clear **documentation** and **command-line arguments** so that you (or other engineers) can reproduce the experiments later. Use the CSV outputs and plots to compare different optimization strategies systematically. Most importantly, keep an eye on **accuracy** while optimizing for speed—there is little value in a fast model that yields poor predictions.