AI Model Optimisation, Quantisation and Speed up.

Optimization	Speed Gain	Accuracy Change	Implementation Time
Quantization (q4_K_M)	2-3x	-2%	10 min
Context reduction (2048)	1.5x	0% (for short docs)	5 min
Streaming responses	Perceived 3x	0%	30 min
Caching layer	10-100x (hits)	0%	2 hours
Preprocessing	1.5x	0%	4 hours
RAG implementation	1.2x	+40%	1 day
Instance upgrade (T4→A10G)	2x	0%	1 hour
Few-shot prompting	1x	+25%	2 hours
Combined optimizations	**5-10x**	**+30-50%**	**2-3 days**

1. MODEL-LEVEL OPTIMIZATIONS

1.1 Quantization

1.1.1 WHY do this?

1. Your LLM model stores every number (weight/parameter) in high precision format
2. Think of it like using a ruler with millimeter markings when you only need centimeter accuracy
3. A 7 billion parameter model in full 16-bit precision consumes about 14GB of memory
4. Each mathematical operation during inference uses this full precision, which is slow and memory-intensive
5. The reality: for most practical tasks, you don't need this level of precision
6. Quantization converts high-precision numbers to lower precision (4-bit or 8-bit):
    1. Dramatically reduces memory usage
    2. Speeds up computations
    3. Minimal accuracy loss (typically 2-5%)
    4. Gain 2-3x speed improvement
    5. Reduce memory by 75%

1.1.2 WHAT to do:

1. Replace your current model with a quantized version
2. Ollama provides models in different quantization levels:
    1. q4_0: 4-bit, fastest
    2. q4_K_M: 4-bit medium quality, balanced (RECOMMENDED)
    3. q5_K_M: 5-bit, better quality
    4. q8_0: 8-bit, minimal accuracy loss
3. Test quantized versions against your specific workload
4. Find the right balance between speed and accuracy

1.1.3 HOW to implement:

1. Check current model:
    1. List all models currently loaded in Ollama
    2. Identify which model you're using
2. Download quantized versions:
    1. Start with q4_K_M as it offers the best balance
    2. Download q4_0 for maximum speed
    3. Download q5_K_M for better quality
    4. Download q8_0 for minimal accuracy loss
3. Test each quantization:
    1. Run the same test prompt through each version
    2. Time the results for each
    3. Compare outputs for accuracy
4. Choose optimal version:
    1. For simple extraction tasks → use q4_0
    2. For general document processing → use q4_K_M
    3. For complex reasoning → use q5_K_M
    4. For critical accuracy needs → use q8_0
5. Make it default:
    1. Set the chosen quantization as your default model
    2. Remove other versions to save disk space

1.2 Model Size Selection

1.2.1 WHY do this?

1. Bigger models aren't always better for your specific task
2. A 70B model can be 10x slower than 7B with only 15% better results
3. Performance issues with oversized models:
    1. Exponentially slower processing
    2. Higher GPU requirements
    3. Increased costs
    4. No proportional benefit
4. Newer specialized smaller models often outperform older larger models:
    1. Mistral 7B vs Llama2 13B
    2. Phi 2.7B for simple tasks
    3. Better training data and techniques

1.2.2 WHAT to do:

1. Match model size to task complexity
2. Test smaller models first before assuming you need larger ones
3. Use specialized models for specific domains
4. Measure both accuracy and speed for your specific use case
5. Calculate efficiency score: accuracy percentage ÷ processing time

1.2.3 HOW to implement:

1. Download candidate models:
    1. Phi 2.7B-q4: for very fast, simple tasks
    2. Mistral 7B-q4_K_M: for general purpose
    3. Llama2 7B-q4_K_M: alternative general purpose
    4. Llama2 13B-q4_K_M: for complex reasoning
2. Create test dataset:
    1. Select 20-50 documents from actual workload
    2. Ensure representative samples
    3. Include edge cases
    4. Have known correct answers
3. Benchmark each model:
    1. Process test documents through each model
    2. Record time taken for each
    3. Check accuracy of results
    4. Note any differences in output quality
4. Calculate efficiency scores:
    1. Formula: (Accuracy % / Seconds) = efficiency score
    2. Compare scores across models
    3. Consider your priorities (speed vs accuracy)
5. Make selection decision:
    1. Use smallest model that meets accuracy threshold
    2. Document your choice and rationale
    3. Set as default for production
6. Model selection matrix:
    1. Simple Q&A and summaries → Phi 2.7B-q4 (2GB VRAM, 5-8x faster)
    2. General tasks → Mistral 7B-q4 (4GB VRAM, 3-4x faster)
    3. Complex reasoning → Llama2 13B-q5 (8GB VRAM, 1.5-2x faster)
    4. Expert-level tasks → Llama2 70B-q4 (40GB VRAM, baseline)
7. Common findings:
    1. Mistral 7B often performs as well as Llama2 13B
    2. Phi 2.7B handles 70% of simple tasks adequately
    3. Specialized smaller models beat generic larger ones

1.3 Context Window Optimization

1.3.1 WHY do this?

1. The context window is how much text the model can "see" at once
2. Processing larger contexts is exponentially slower:
    1. Quadratic complexity of attention mechanisms
    2. Doubling context size roughly quadruples processing time
3. Example waste: 8K token context for 1K token documents
    1. You're making the model process 8x more data than necessary
    2. Each token position must attend to all other positions
4. Typical document sizes:

        1. Invoice: 500-1000 tokens
        2. Contract: 2000-3000 tokens
        3. Report: 5000-8000 tokens
    5. Using default maximum context is inefficient:
        1. Like bringing a moving truck to transport a suitcase
        2. Wastes processing time on empty space
        3. Reduces throughput unnecessarily

1.3.2 WHAT to do:

1. Analyze your actual document sizes to determine needed context
2. Set context sizes based on document type:
    1. Small documents (under 1000 words) → 2048 tokens
    2. Medium documents (1000-3000 words) → 4096 tokens
    3. Large documents (over 3000 words) → 8192 tokens
3. Set context dynamically based on input size
4. Avoid one-size-fits-all maximum context approach

1.3.3 HOW to implement:

1. Analyze your document collection:
    1. Take representative sample of 100-200 documents
    2. Count words in each document
    3. Calculate statistics:
        1. Average document size
        2. Median document size
        3. 95th percentile size
        4. Maximum size
2. Choose default context size:
    1. Use context that accommodates 95% of documents
    2. Add 25% buffer for prompts and responses
    3. Round up to nearest standard size (2048, 4096, 8192)
3. Create dynamic sizing logic:
    1. Measure actual content length before processing
    2. Apply formula: word count × 1.33 = approximate tokens
    3. Add 500 token buffer for prompts and responses
    4. Select appropriate context tier:
        1. If calculated need < 2048 → use 2048
        2. If between 2048-4096 → use 4096
        3. If over 4096 → use 8192
4. Handle outliers:
    1. For documents exceeding largest context:
        1. Truncate to most relevant sections
        2. Use separate processing pipeline
        3. Implement chunking strategy
    2. For very small documents:
        1. Still use minimum 2048 context

        2. Processing time is negligible anyway
5. Implement in processing pipeline:
    1. Create function that analyzes input
    2. Returns optimal context size
    3. Pass to Ollama API call automatically
    4. No manual intervention required
6. Track usage patterns:
    1. Log which context sizes are used
    2. Identify if most documents use smaller contexts
    3. Verify your sizing strategy is optimal
    4. Adjust thresholds if needed
7. Context sizing decision tree:
    1. Document under 500 words → 2048 tokens (1-2 pages)
    2. Document 500-1500 words → 4096 tokens (2-5 pages)
    3. Document 1500+ words → 8192 tokens (5+ pages)
8. Priority-based adjustments:
    1. If speed is critical → Always use 2048, truncate documents
    2. If accuracy is critical → Use minimum that fits full document
    3. Balanced approach → Use sizing logic above

---

2. INFRASTRUCTURE OPTIMIZATIONS

2.1 Right-Sizing EC2 Instance

2.1.1 WHY do this?

1. Instance selection directly impacts cost and performance
2. Wrong instance size causes problems:
    1. Too large: Wasting money on unused capacity
    2. Too small: Performance bottlenecks and queuing
3. Cost implications:
    1. GPU instances range from $0.50 to $5+ per hour
    2. Wrong choice costs thousands per month
    3. 24/7 operation: $360-3600/month difference
4. Different GPU generations have vastly different characteristics:
    1. A10G is 2x faster than T4 at similar cost
    2. Newer architectures more power-efficient
    3. Different memory bandwidth and compute capabilities
5. Utilization patterns reveal right-sizing opportunities:
    1. Consistently under 50% utilized → overpaying
    2. Constantly above 90% → need to upgrade
    3. Optimal: 60-80% average utilization

2.1.2 WHAT to do:

1. Monitor actual GPU utilization for extended period (24-48 hours)
2. Measure both GPU compute utilization and memory usage
3. Analyze patterns to identify right instance size

4. Decision matrix:
    1. If consistently underutilized → downsize
    2. If constantly maxed out → upsize
    3. If moderate utilization → consider newer GPU generation
5. Calculate cost-effectiveness: requests per dollar metric

2.1.3 HOW to implement:

1. Set up continuous monitoring:
    1. Use nvidia-smi to query GPU stats
    2. Log memory usage every 60 seconds
    3. Log compute utilization every 60 seconds
    4. Run for minimum 24 hours, preferably 48-72 hours
    5. Capture different times of day and usage patterns
2. Analyze the collected data:
    1. Calculate average memory usage
    2. Calculate peak memory usage
    3. Calculate average GPU compute utilization
    4. Calculate peak GPU utilization
    5. Identify patterns:
        1. Time of day variations
        2. Usage spikes
        3. Idle periods
3. Interpret utilization metrics:
    1. If memory usage stays below 50%:
        1. Consider smaller instance
        2. Or load multiple models
    2. If memory usage consistently above 90%:
        1. Need larger instance
        2. Risk of OOM errors
    3. If GPU compute below 40%:
        1. Underutilizing instance
        2. Can increase parallel processing
    4. If GPU compute above 95%:
        1. At capacity
        2. Requests likely queuing
        3. Need to scale
4. Benchmark different instance types:
    1. Create baseline with current instance:
        1. Measure requests per hour
        2. Measure average latency
        3. Note current cost per hour
    2. Calculate current efficiency:
        1. Requests per hour ÷ cost per hour
        2. This is your baseline metric
    3. Research alternative instances:
        1. g4dn.xlarge: T4 16GB, $0.526/hr

        2. g5.xlarge: A10G 24GB, $1.006/hr
        3. g5.2xlarge: A10G 24GB, $1.212/hr
        4. For 13B models: g5.2xlarge or larger
        5. For 70B models: g5.12xlarge (4x A10G)
5. Calculate cost-benefit for each option:
    1. Estimate throughput on new instance:
        1. A10G is approximately 2x faster than T4
        2. Adjust request rate accordingly
    2. Calculate efficiency metric:
        1. (Estimated requests/hour × accuracy) ÷ cost/hour
        2. Higher score = better value
    3. Example calculation:
        1. g4dn.xlarge: (100 req/hr × 0.90) ÷ $0.526 = 171 points
        2. g5.xlarge: (200 req/hr × 0.90) ÷ $1.006 = 179 points
        3. g5.xlarge wins despite higher cost
6. Run live benchmark on candidate instances:
    1. Spin up temporary instance
    2. Deploy your setup
    3. Run actual workload for 1 hour
    4. Measure real throughput and latency
    5. Compare against projections
    6. Terminate test instance
7. Make migration decision:
    1. Choose instance with best cost-effectiveness
    2. Plan migration:
        1. Schedule during low-traffic period
        2. Prepare new instance
        3. Test thoroughly
        4. Update DNS/routing
        5. Monitor closely after switch
    3. Keep old instance available for rollback
    4. Validate performance meets expectations
8. Ongoing monitoring:
    1. Continue tracking utilization
    2. Re-evaluate quarterly
    3. As usage grows, reassess instance size
    4. Take advantage of new instance types as released

2.2 GPU Memory Optimization

2.2.1 WHY do this?

1. Ollama loads entire models into GPU memory and keeps them resident
2. Problems with uncontrolled memory usage:
    1. Memory fragmentation over time
    2. Out Of Memory (OOM) errors causing crashes
    3. Swapping to system RAM (extremely slow)

    4. System instability
3. Concurrent request handling consumes additional memory:
    1. Each active request uses memory for activations
    2. Multiple requests can quickly exhaust available memory
    3. Without limits, system becomes unstable
4. Multiple model loading creates issues:
    1. Each model loaded consumes full model size in memory
    2. Switching between models causes loading delays
    3. Memory fragmentation increases
5. Consequences of poor memory management:
    1. Crashes mid-request
    2. Slow performance from memory pressure
    3. Unpredictable response times
    4. Service downtime

2.2.2 WHAT to do:

1. Configure Ollama to manage memory efficiently
2. Limit how many models stay loaded simultaneously
3. Control concurrent request processing based on available memory
4. Set queue limits to prevent memory exhaustion
5. Monitor memory usage continuously
6. Recommended settings:
    1. Keep only 1 model loaded at a time
    2. Process 2 concurrent requests for 16GB GPU
    3. Process 4 concurrent requests for 24GB+ GPU
    4. Queue limit of 10-20 requests

2.2.3 HOW to implement:

1. Locate Ollama service configuration:
    1. Use systemctl to edit Ollama service
    2. Creates override configuration file
    3. Preserves original settings
2. Configure memory management parameters:
    1. OLLAMA_MAX_LOADED_MODELS:
        1. Purpose: Limits models kept in memory
        2. Set to 1 for optimal memory usage
        3. Prevents memory fragmentation
        4. Forces model unloading when switching
    2. OLLAMA_NUM_PARALLEL:
        1. Purpose: Controls concurrent request processing
        2. For 16GB GPU: set to 2
        3. For 24GB GPU: set to 4
        4. For 32GB+ GPU: set to 6-8
        5. Too high causes OOM errors
        6. Too low underutilizes GPU

        3. OLLAMA_MAX_QUEUE:
            1. Purpose: Limits waiting requests
            2. Set to 10 for moderate traffic
            3. Set to 20 for high traffic
            4. Prevents infinite queuing
            5. Requests beyond limit are rejected
        4. CUDA_VISIBLE_DEVICES:
            1. Purpose: Explicitly specify which GPU to use
            2. Set to 0 for single GPU systems
            3. Prevents confusion in multi-GPU systems
3. Apply configuration changes:
    1. Save configuration file
    2. Reload systemd daemon to recognize changes
    3. Restart Ollama service
    4. Verify service starts successfully
    5. Check logs for any errors
4. Verify new settings:
    1. Check service status
    2. Confirm environment variables loaded
    3. Test with sample request
    4. Monitor GPU memory during test
5. Implement continuous memory monitoring:
    1. Create monitoring script that:
        1. Queries GPU memory every 5 seconds
        2. Displays current usage vs total
        3. Calculates utilization percentage
        4. Shows GPU compute utilization
    2. Set up alerts:
        1. Warning when memory exceeds 85%
        2. Critical alert when exceeds 90%
        3. Log to file for trend analysis
6. Test under load:
    1. Send multiple concurrent requests
    2. Observe memory usage patterns
    3. Verify requests process without errors
    4. Check that queuing works as expected
    5. Ensure no OOM crashes occur
7. Tune based on observations:
    1. If memory consistently below 60%:
        1. Can increase NUM_PARALLEL by 1
        2. Test new setting
        3. Monitor for issues
    2. If memory often exceeds 90%:
        1. Decrease NUM_PARALLEL by 1
        2. Prevents crashes
        3. Improves stability
    3. If many requests timing out:

        1. Increase MAX_QUEUE
        2. Or scale horizontally with more instances
    4. If seeing OOM errors:
        1. Immediately reduce NUM_PARALLEL
        2. Restart service
        3. Investigate memory leaks
8. Configuration optimization guide:
    1. Conservative (high stability):
        1. MAX_LOADED_MODELS=1
        2. NUM_PARALLEL=1
        3. MAX_QUEUE=5
        4. Use when stability is critical
    2. Balanced (recommended):
        1. MAX_LOADED_MODELS=1
        2. NUM_PARALLEL=2 (16GB) or 4 (24GB)
        3. MAX_QUEUE=10
        4. Best for most production workloads
    3. Aggressive (maximum throughput):
        1. MAX_LOADED_MODELS=1
        2. NUM_PARALLEL=4 (16GB) or 6-8 (24GB+)
        3. MAX_QUEUE=20
        4. Requires careful monitoring

2.3 CUDA and Driver Optimization

2.3.1 WHY do this?

1. NVIDIA drivers are the software layer between your application and GPU
2. Outdated drivers cause significant performance issues:
    1. 10-30% performance degradation
    2. Missing optimizations for recent GPU architectures
    3. Unpatched performance bugs
    4. Compatibility issues with newer software
3. CUDA toolkit is essential for GPU computation:
    1. Provides libraries for GPU operations
    2. Enables frameworks like PyTorch to use GPU
    3. Wrong version causes instability
    4. Missing toolkit prevents GPU usage entirely
4. Common problems from misconfiguration:
    1. GPU not recognized by Ollama
    2. Crashes during inference
    3. Falling back to CPU (extremely slow)
    4. Reduced throughput
5. Many EC2 instances ship with outdated software:
    1. Drivers from 6-12 months ago
    2. Missing recent performance improvements
    3. Not optimized for your specific GPU

2.3.2 WHAT to do:

1. Update to latest stable NVIDIA driver for your GPU generation
2. Install appropriate CUDA toolkit version
3. Verify compatibility between driver, CUDA, and Ollama
4. Set environment variables correctly
5. Test entire stack after updates
6. Reboot system to ensure clean initialization

2.3.3 HOW to implement:

1. Check current driver version:
    1. Run nvidia-smi command
    2. Note "Driver Version" in output
    3. Note "CUDA Version" it reports
    4. Record GPU model name
2. Check CUDA toolkit installation:
    1. Run nvcc --version command
    2. If command not found, toolkit not installed
    3. If found, note the version
3. Research recommended versions:
    1. Visit NVIDIA website
    2. Find latest stable driver for your GPU:
        1. For T4 GPUs: 535.x or newer
        2. For A10G GPUs: 535.x or newer
        3. Check Ubuntu/Amazon Linux compatibility
    3. Identify matching CUDA toolkit:
        1. CUDA 12.x for recent drivers
        2. CUDA 11.x for older setups
    4. Verify Ollama compatibility with these versions
4. Prepare for installation:
    1. Stop Ollama service to avoid conflicts
    2. Backup current driver information
    3. Note current performance baseline
    4. Plan for potential rollback
5. Install NVIDIA driver (Ubuntu/Debian):
    1. Update package repository
    2. Add NVIDIA package repository if needed
    3. Install nvidia-driver-535 package (or latest)
    4. Process takes 5-10 minutes
    5. Ignore warnings about existing driver
6. Install NVIDIA driver (Amazon Linux):
    1. Install build dependencies (gcc, kernel-devel)
    2. Download driver installer from NVIDIA
    3. Run installer with --silent --driver flags
    4. Installer compiles kernel module
    5. Takes 10-15 minutes

7. Install CUDA toolkit:
    1. Download appropriate installer for your OS
    2. Run installer script
    3. Choose components to install:
        1. Select CUDA toolkit
        2. Select CUDA libraries
        3. Select documentation
        4. Select samples (optional)
    4. Accept default installation paths
    5. Installation takes 5-10 minutes
8. Configure environment variables:
    1. Edit shell configuration file (.bashrc or .profile)
    2. Add PATH variable:
        1. Include /usr/local/cuda/bin
        2. Allows running CUDA utilities
    3. Add LD_LIBRARY_PATH variable:
        1. Include /usr/local/cuda/lib64
        2. Allows finding CUDA libraries at runtime
    4. Add CUDA_HOME variable:
        1. Set to /usr/local/cuda
        2. Used by some applications
    5. Save file and reload configuration
9. Reboot system:
    1. Essential for clean driver initialization
    2. Ensures all services start with new driver
    3. Prevents conflicts from old driver remnants
    4. Takes 2-3 minutes
10. Verify installation after reboot:
    1. Run nvidia-smi:
        1. Should show new driver version
        2. Should detect GPU correctly
        3. Should show memory available
        4. No errors in output
    2. Run nvcc --version:
        1. Should show CUDA toolkit version
        2. Should match what you installed
    3. Test PyTorch/CUDA integration:
        1. Run Python test script
        2. Verify torch.cuda.is_available() returns True
        3. Verify can allocate GPU memory
    4. Test Ollama:
        1. Start Ollama service
        2. Run simple inference
        3. Verify uses GPU (check nvidia-smi)
        4. Compare performance to baseline
11. Benchmark performance improvement:
    1. Run same test prompts as before update

2. Measure latency for each
3. Calculate average improvement
4. Should see 5-15% improvement minimum
5. Document improvements for justification

12. Troubleshooting common issues:
    1. If nvidia-smi shows no devices:
        1. Driver didn't install correctly
        2. Check /var/log/nvidia-installer.log
        3. May need to disable nouveau driver
        4. Reinstall driver
    2. If CUDA not found:
        1. Environment variables not set
        2. Reload shell configuration
        3. Verify paths are correct
    3. If Ollama doesn't use GPU:
        1. Restart Ollama service
        2. Check Ollama logs
        3. Verify CUDA libraries accessible
        4. May need to reinstall Ollama
13. Maintenance schedule:
    1. Check for driver updates quarterly
    2. Read release notes for performance improvements
    3. Test updates in staging before production
    4. Keep documentation of versions in use

---

3. OLLAMA-SPECIFIC OPTIMIZATIONS

3.1 Modelfile Customization

3.1.1 WHY do this?

1. Ollama's default settings are conservative for broad compatibility
2. Problems with default settings:
    1. Not optimized for your specific use case
    2. May use excessive context
    3. May generate longer responses than needed
    4. Sampling parameters may not match your requirements
3. Benefits of custom Modelfile:
    1. Bake optimizations directly into model
    2. Ensure consistency across all API calls
    3. Eliminate need to pass options every time
    4. Create specialized models for different tasks
    5. Simplify application code
4. Optimization opportunities:
    1. Reduce context for typical documents
    2. Limit output length to actual needs
    3. Tune sampling for accuracy vs creativity
    4. Include task-specific system prompts

5. Set appropriate temperature and top_k

3.1.2 WHAT to do:

1. Create custom Modelfile specifying exact model behavior
2. Set context window matching your typical document size
3. Limit output length to what you actually need
4. Configure sampling parameters for your use case:
    1. Deterministic settings for data extraction
    2. Creative settings for summaries
5. Include system prompt tailored to your task
6. Build and use custom model instead of base model

3.1.3 HOW to implement:

1. Create new Modelfile:
    1. Open text editor
    2. Create file named "Modelfile"
    3. No file extension needed
2. Specify base model:
    1. Add FROM directive
    2. Specify your chosen model (e.g., llama2:7b-q4_K_M)
    3. This inherits the model's weights
3. Configure GPU settings:
    1. Add PARAMETER num_gpu directive:
        1. Set to 1 for single GPU systems
        2. Set to 2 for dual GPU systems
        3. Enables tensor parallelism
    2. Add PARAMETER num_thread directive:
        1. Set to 8 for most systems
        2. Controls CPU threads for non-GPU operations
        3. Affects preprocessing speed
4. Configure context and generation limits:
    1. Add PARAMETER num_ctx directive:
        1. Set to 2048 for most documents
        2. Set to 4096 for longer documents
        3. Set to 8192 for very long documents
        4. Lower = faster processing
    2. Add PARAMETER num_batch directive:
        1. Set to 512 typically
        2. Controls prompt processing batch size
        3. Higher = faster but more memory
    3. Add PARAMETER num_predict directive:
        1. Set to 512 for typical responses
        2. Set to 256 for short extractions
        3. Set to 1024 for longer outputs
        4. Limits maximum output length

5. Configure sampling parameters for accuracy:
    1. Add PARAMETER temperature directive:
        1. Set to 0.1-0.2 for factual extraction
        2. Set to 0.7 for summaries
        3. Set to 0.8 for creative writing
        4. Lower = more deterministic
    2. Add PARAMETER top_p directive:
        1. Set to 0.9 for focused outputs
        2. Controls nucleus sampling
        3. Limits token selection probability mass
    3. Add PARAMETER top_k directive:
        1. Set to 10-20 for accuracy
        2. Set to 40-50 for creativity
        3. Limits number of token choices
    4. Add PARAMETER repeat_penalty directive:
        1. Set to 1.1 typically
        2. Prevents repetitive output
        3. Higher values = stronger penalty
6. Add system prompt for your use case:
    1. Add SYSTEM directive with triple quotes
    2. For invoice processing:
        1. "You are a precise document analysis assistant."
        2. "Extract information accurately and concisely."
        3. "Always respond in structured format."
        4. "Do not add explanations unless requested."
    3. For general Q&A:
        1. "You are a helpful assistant."
        2. "Provide clear, accurate responses."
        3. "Be concise and direct."
    4. Keep system prompt under 100 words
7. Save Modelfile:
    1. Save in accessible directory
    2. Note the file location
    3. Ensure plain text format
8. Build custom model:
    1. Run Ollama create command
    2. Specify name for custom model:
        1. Use descriptive names (e.g., "invoice-extractor-fast")
        2. Include use case in name
        3. Include optimization level if applicable
    3. Provide path to Modelfile
    4. Build process takes 10-30 seconds
    5. Custom model now available to use
9. Test custom model:
    1. Run ollama list to verify model exists
    2. Run test inference with custom model
    3. Compare to base model results

4. Verify settings are applied:
    1. Check response format matches expectations
    2. Verify output length is limited
    3. Confirm deterministic behavior (if configured)
10. Use custom model in production:
    1. Update application code
    2. Replace base model name with custom model name
    3. Remove options that are now baked in
    4. Simplifies API calls
    5. Ensures consistency
11. Create multiple specialized models:
    1. For data extraction:
        1. Low temperature (0.1)
        2. Small context (2048)
        3. Limited output (256)
        4. Name: "data-extractor"
    2. For summarization:
        1. Medium temperature (0.7)
        2. Medium context (4096)
        3. Moderate output (512)
        4. Name: "summarizer"
    3. For Q&A:
        1. Low temperature (0.2)
        2. Medium context (4096)
        3. Moderate output (512)
        4. Name: "qa-assistant"
12. Maintain custom models:
    1. Document what each model does
    2. Keep Modelfiles in version control
    3. Update when base models updated
    4. Test after each update
    5. Remove unused custom models

3.2 Batch Processing

3.2.1 WHY do this?

1. Processing documents one-by-one is highly inefficient
2. Problems with sequential processing:
    1. GPU sits idle between requests
    2. Network round trips add latency
    3. CPU cores not fully utilized
    4. Can't take advantage of GPU parallelism
3. Time waste example:
    1. 100 documents at 3 seconds each
    2. Sequential: 300 seconds total
    3. Batch of 4: 75 seconds total

    4. 4x improvement with batching
4. GPU design favors parallel processing:
    1. Thousands of cores designed for concurrent operations
    2. Underutilized with single requests
    3. Can process multiple requests simultaneously
    4. Memory bandwidth shared efficiently
5. Batching benefits:
    1. Higher GPU utilization
    2. Better throughput (documents per second)
    3. Lower cost per document
    4. Faster overall processing

3.2.2 WHAT to do:

1. Process multiple documents concurrently instead of sequentially
2. Use thread pools or async programming for parallel requests
3. Configure Ollama to handle multiple parallel requests
4. Balance parallelism with available resources
5. Find optimal number of concurrent workers:
    1. Too few: Wastes GPU capacity
    2. Too many: Causes memory issues and queuing
6. Typical configuration:
    1. 2 workers for 16GB GPU
    2. 4 workers for 24GB GPU
    3. 6-8 workers for 32GB+ GPU

3.2.3 HOW to implement:

1. Configure Ollama for parallel processing:
    1. Edit Ollama service configuration
    2. Set OLLAMA_NUM_PARALLEL to desired value
    3. Start with 2 for testing
    4. Restart Ollama service
2. Choose concurrency approach:
    1. Thread-based concurrency:
        1. Simpler to implement
        2. Works well for I/O-bound tasks
        3. Python ThreadPoolExecutor
    2. Async/await concurrency:
        1. More efficient for high concurrency
        2. Better for network-heavy workloads
        3. Python asyncio with aiohttp
3. Implement thread-based batch processing:
    1. Import threading libraries
    2. Create function to process single document:
        1. Takes document content as input
        2. Sends to Ollama API

        3. Returns result with metadata
        4. Handles errors gracefully
    3. Create batch processing function:
        1. Takes list of documents
        2. Takes max_workers parameter
        3. Creates ThreadPoolExecutor
        4. Submits all documents to pool
        5. Waits for completion
        6. Returns all results
4. Design processing workflow:
    1. Organize documents into batches
    2. Start with batch size matching workers
    3. Submit batch to processing pool
    4. Track completion of each document
    5. Collect results as they finish
    6. Log progress and errors
5. Implement error handling:
    1. Wrap each document processing in try-except
    2. Log errors with document ID
    3. Return error status in result
    4. Don't let one failure stop entire batch
    5. Collect failed documents for retry
6. Add progress tracking:
    1. Log when each document starts
    2. Log when each document completes
    3. Show running progress percentage
    4. Display estimated time remaining
    5. Report throughput (docs per second)
7. Test with different worker counts:
    1. Start with 1 worker (sequential baseline)
    2. Test with 2 workers
    3. Test with 4 workers
    4. Test with 6 workers
    5. Test with 8 workers
    6. For each, measure:
        1. Total processing time
        2. Throughput (docs/second)
        3. GPU memory usage
        4. GPU utilization percentage
        5. Error rate
8. Analyze results to find optimal workers:
    1. Plot throughput vs worker count
    2. Throughput should increase then plateau
    3. Watch for diminishing returns
    4. Monitor GPU memory:
        1. If exceeds 90%, reduce workers
        2. If below 60%, can increase workers

5. Choose worker count where:
    1. Throughput is maximized
    2. Memory stays under 85%
    3. Error rate stays low
9. Implement dynamic batching:
    1. If queue is long, use more workers
    2. If queue is short, use fewer workers
    3. Adapt to load automatically
    4. Prevents resource waste during quiet periods
    5. Maximizes throughput during busy periods
10. Add request queuing:
    1. Create queue to hold pending documents
    2. Workers pull from queue
    3. When queue full, reject new requests
    4. Prevents unbounded memory growth
    5. Provides backpressure to clients
11. Optimize for your workload:
    1. If documents are similar size:
        1. Use fixed worker count
        2. Simpler configuration
    2. If documents vary greatly in size:
        1. Sort by size before processing
        2. Process small documents first
        3. Prevents stragglers delaying batch
    3. If some documents are priority:
        1. Implement priority queue
        2. Process high-priority first
12. Monitor production performance:
    1. Track average batch processing time
    2. Monitor worker utilization
    3. Log failed documents for investigation
    4. Adjust worker count based on patterns
    5. Scale horizontally if single instance maxed
13. Expected improvements:
    1. 2 workers: 1.8-2x faster than sequential
    2. 4 workers: 3-3.5x faster than sequential
    3. 6 workers: 4-5x faster (with large GPU)
    4. 8 workers: 4.5-5.5x faster (with very large GPU)
    5. Beyond 8: Diminishing returns typically

---

4. NETWORK & DATA TRANSFER OPTIMIZATIONS

4.1 File Upload Compression

4.1.1 WHY do this?

1. Network transfer is often the slowest part of the pipeline
2. Typical file upload times:

        1. 1MB file on slow connection: 8-10 seconds
        2. 10MB file on slow connection: 60-90 seconds
        3. 10MB file on fast connection: 5-10 seconds
    3. LLM processing is fast by comparison:
        1. Actual inference: 3-5 seconds
        2. Upload taking longer than processing is wasteful
    4. Network bandwidth costs money:
        1. AWS charges for data transfer
        2. Incoming data costs
        3. Cross-region transfer especially expensive
    5. Large uploads cause operational problems:
        1. Timeouts on slow connections
        2. Poor user experience
        3. Increased infrastructure costs
    6. Compression benefits:
        1. Text files: 70-80% size reduction
        2. PDFs: 60-75% size reduction
        3. Images: 50-60% size reduction (if not already compressed)
        4. Dramatically faster uploads

4.1.2 WHAT to do:

1. Compress files on client side before upload
2. Use gzip compression (fast, widely supported, excellent ratio)
3. Server receives compressed file and decompresses
4. Decompression is extremely fast (milliseconds)
5. Net benefit: Massive reduction in upload time
6. Example improvement:
    1. 10MB PDF → 2MB compressed
    2. Upload time: 60s → 12s
    3. 5x improvement before any LLM work

4.1.3 HOW to implement:

1. Choose compression method:
    1. gzip: Standard, widely supported, good compression
    2. Compression level 6: Balance of speed and ratio
    3. Level 9: Maximum compression, slower
    4. Level 1: Fast compression, lower ratio
    5. Recommend level 6 for general use
2. Implement client-side compression:
    1. Before upload, read file into memory
    2. Compress entire file content using gzip
    3. Typical compression process:
        1. Load file as bytes
        2. Pass to gzip.compress()
        3. Specify compression level (6)

        4. Get compressed bytes back
    4. Takes 100-500ms for typical documents
3. Prepare compressed upload:
    1. Create filename with .gz extension
    2. Set Content-Encoding header to "gzip"
    3. Upload compressed data
    4. Server knows file is compressed
4. Implement server-side decompression:
    1. Check if filename ends with .gz
    2. Or check Content-Encoding header
    3. If compressed:
        1. Read uploaded bytes
        2. Decompress using gzip.decompress()
        3. Takes 10-50ms typically
        4. Get original file content
    4. If not compressed:
        1. Use file content directly
        2. Maintain backward compatibility
5. Save decompressed file:
    1. Remove .gz extension from filename
    2. Write decompressed content to temporary file
    3. Process as normal
    4. Clean up temporary file after processing
6. Add compression metadata:
    1. Log original file size
    2. Log compressed size
    3. Calculate compression ratio
    4. Track bandwidth saved
    5. Use for reporting and optimization
7. Handle compression errors:
    1. Catch decompression failures
    2. May indicate corrupted upload
    3. Return clear error to client
    4. Log for investigation
8. Optimize compression level by file type:
    1. Text files: Use level 9 (compress very well)
    2. PDFs: Use level 6 (good balance)
    3. Already-compressed images: Skip compression
    4. Binary data: Use level 6
9. Implement intelligent compression:
    1. Check file size before compressing
    2. If under 100KB, skip compression:
        1. Small files transfer quickly anyway
        2. Compression overhead not worth it
    3. If over 100KB, compress:
        1. Significant time savings
        2. Worth the CPU cost

10. Add client-side progress indication:
    1. Show "Compressing..." message
    2. Show "Uploading..." with progress bar
    3. Show "Processing..." after upload complete
    4. Better user experience
11. Measure improvement:
    1. Track upload times before compression
    2. Track upload times after compression
    3. Calculate average improvement
    4. Document bandwidth savings
    5. Calculate cost savings
12. Special considerations:
    1. For mobile clients:
        1. Compression saves cellular data
        2. Important for users with limited plans
        3. Battery impact minimal
    2. For enterprise clients:
        1. Reduces internal network load
        2. Faster document processing
        3. Lower cloud costs

4.2 Streaming Responses

4.2.1 WHY do this?

1. Default behavior waits for complete response before returning
2. Problems with buffered responses:
    1. User sees nothing during generation
    2. Appears frozen or unresponsive
    3. No way to tell if system is working
    4. Poor user experience
3. Typical generation times:
    1. 500-token response at 20 tokens/second
    2. Total time: 25 seconds
    3. User sees nothing for 25 seconds
    4. Many users think system crashed
4. Streaming solves perception problem:
    1. Tokens appear as generated
    2. User sees immediate feedback
    3. Feels 3-5x faster (same actual time)
    4. Can verify output quality early
5. Additional benefits:
    1. Users can stop generation early if wrong
    2. Saves compute time on bad outputs
    3. Better engagement
    4. More natural interaction

4.2.2 WHAT to do:

1. Enable streaming mode in Ollama API calls
2. Modify server to forward streaming tokens to clients
3. Update client to display tokens as they arrive
4. Show typing indicator during generation
5. Mark completion when done
6. Implement cancellation capability

4.2.3 HOW to implement:

1. Enable streaming in Ollama API:
    1. Set "stream" parameter to true (not false)
    2. Changes response format completely
    3. Returns stream of JSON objects instead of single object
    4. Each object contains one or more tokens
2. Understand streaming response format:
    1. Each line is separate JSON object
    2. Contains "response" field with token(s)
    3. Contains "done" field (false until complete)
    4. Final message has "done": true
    5. Final message includes statistics
3. Set up server-side streaming:
    1. Choose streaming technology:
        1. Server-Sent Events (SSE): Standard, simple
        2. WebSocket: Bidirectional, more complex
        3. Chunked Transfer Encoding: HTTP/1.1 standard
    2. SSE recommended for one-way streaming
    3. WebSocket if need client→server during stream
4. Implement SSE endpoint on server:
    1. Create endpoint that returns StreamingResponse
    2. Set Content-Type to "text/event-stream"
    3. Set Cache-Control to "no-cache"
    4. Keep connection open during generation
5. Create generator function:
    1. Makes Ollama API call with stream=true
    2. Iterates over response lines
    3. For each line:
        1. Parse JSON object
        2. Extract token from "response" field
        3. Check "done" flag
        4. Yield token to client
    4. Close when done=true received
6. Handle streaming data flow:
    1. Ollama generates token → Server receives → Server forwards → Client displays
    2. Minimize buffering at each step

    3. Flush immediately after each token
    4. Maintain low latency (under 50ms per token)
7. Implement client-side streaming:
    1. Establish connection to SSE endpoint
    2. Listen for incoming messages
    3. For each message:
        1. Parse event data
        2. Extract token
        3. Append to display immediately
        4. Update UI
    4. Close connection when complete
8. Add UI indicators:
    1. Show typing indicator when starting:
        1. Animated dots
        2. Pulsing cursor
        3. "Generating..." text
    2. Display tokens as they arrive:
        1. Append to text area
        2. Smooth scrolling
        3. Maintain formatting
    3. Show completion indicator:
        1. Remove typing indicator
        2. Show final statistics
        3. Enable user actions
9. Implement cancellation:
    1. Add "Stop" button to UI
    2. When clicked:
        1. Close SSE connection
        2. Send cancel request to server
        3. Server stops Ollama processing
    3. Clean up partial response
    4. Allow starting new request
10. Handle errors during streaming:
    1. Network disconnection:
        1. Show error message
        2. Offer to retry
        3. Keep partial response visible
    2. Server error:
        1. Parse error from stream
        2. Display to user
        3. Log for debugging
    3. Timeout:
        1. Set reasonable timeout (60-120s)
        2. Cancel if exceeded
        3. Inform user
11. Optimize streaming performance:
    1. Batch multiple tokens if generated quickly:

            1. Buffer for 50ms
            2. Send batch together
            3. Reduces network overhead
        2. Use compression for stream:
            1. gzip encoding if supported
            2. Reduces bandwidth
        3. Minimize processing between tokens:
            1. Direct forwarding
            2. No unnecessary transformations
12. Test streaming implementation:
        1. Test with slow generations (large models)
        2. Verify tokens appear promptly
        3. Check cancellation works
        4. Test error handling
        5. Measure latency per token
        6. Verify no buffering delays
13. Fallback for non-streaming clients:
        1. Support both streaming and non-streaming
        2. Detect client capabilities
        3. Use streaming if supported
        4. Fall back to buffered if not
        5. Maintain compatibility
14. Performance metrics:
        1. Measure time to first token:
            1. Should be under 1 second
            2. Critical for perceived speed
        2. Measure tokens per second:
            1. Should match model capability
            2. No significant overhead
        3. User perception improvement:
            1. Feels 3-5x faster
            2. Better satisfaction scores

4.3 Caching Layer

4.3.1 WHY do this?

1. Many queries and documents are repeated
2. Real-world patterns:
    1. Users ask same question about document multiple times
    2. Common documents processed repeatedly
    3. Similar queries with same intent
    4. FAQs hit repeatedly
3. LLM inference is expensive:
    1. Time cost: 3-5 seconds per query
    2. Compute cost: GPU cycles
    3. Money cost: Instance hours

4. Cache benefits:
    1. Cache hit returns in 1-5 milliseconds
    2. That's 1000x faster than generation
    3. 100% accuracy (same response)
    4. Zero GPU load
5. Cost savings example:
    1. 1000 queries per day
    2. 50% cache hit rate
    3. 500 queries saved × 3 seconds = 25 minutes GPU time saved daily
    4. Scales with volume
6. Common caching scenarios:
    1. Same document queried multiple times
    2. Common templates processed repeatedly
    3. FAQ-style questions
    4. Standard data extraction patterns

4.3.2 WHAT to do:

1. Implement Redis caching layer
2. Store previous LLM responses keyed by request hash
3. Before every LLM call, check cache
4. If cache hit, return immediately
5. If cache miss, process with LLM and store result
6. Set appropriate Time-To-Live (TTL) values
7. Monitor cache hit rate
8. Target 30-80% hit rate for good ROI

4.3.3 HOW to implement:

1. Install Redis on EC2 instance:
    1. For Ubuntu/Debian:
        1. Update package repository
        2. Install redis-server package
        3. Takes 2-3 minutes
    2. For Amazon Linux:
        1. Install from amazon-linux-extras
        2. Install redis6 package
        3. Takes 2-3 minutes
    3. Start Redis service
    4. Enable auto-start on boot
    5. Verify running with ping command
2. Configure Redis for caching:
    1. Edit Redis configuration file
    2. Set maximum memory:
        1. Allocate 1-2GB for caching
        2. More memory = more cache entries
        3. Balance with application needs

    3. Set eviction policy:
        1. allkeys-lru recommended
        2. Evicts least recently used items
        3. Ensures cache stays within memory limit
    4. Disable disk persistence (optional):
        1. Caching doesn't need durability
        2. Improves performance
        3. Simpler operation
    5. Restart Redis to apply changes
3. Design cache key structure:
    1. Cache key must uniquely identify request
    2. Include in key:
        1. Model name
        2. Exact prompt text
        3. All options (temperature, context size, etc.)
    3. Create deterministic string from these
    4. Hash with MD5 or SHA256
    5. Use hash as cache key
4. Implement cache checking logic:
    1. Create function to generate cache key:
        1. Take model, prompt, options as input
        2. Combine into single string
        3. Sort options to ensure consistency
        4. Hash the combined string
        5. Return hash as cache key
    2. Create function to check cache:
        1. Generate cache key
        2. Query Redis with key
        3. If key exists, return value
        4. If not, return None
5. Wrap LLM calls with caching:
    1. Before calling Ollama:
        1. Generate cache key
        2. Check if key exists in Redis
        3. If exists:
            1. Retrieve cached response
            2. Parse JSON
            3. Add 'cached': true flag
            4. Return immediately
        4. If not exists:
            1. Proceed with LLM call
    2. After Ollama returns:
        1. Take response
        2. Convert to JSON string
        3. Store in Redis with cache key
        4. Set TTL (expiration time)
        5. Return response to client

6. Set appropriate TTL values:
    1. Consider data freshness requirements:
        1. Static content: 7 days (604800 seconds)
        2. Semi-static (product data): 24 hours (86400 seconds)
        3. Dynamic content: 1 hour (3600 seconds)
        4. Real-time data: 5 minutes (300 seconds)
    2. Different TTLs for different content types
    3. Longer TTL = higher hit rate but staler data
    4. Shorter TTL = fresher data but lower hit rate
7. Implement cache statistics tracking:
    1. Create stats object with counters:
        1. Total hits
        2. Total misses
        3. Total errors
    2. Increment on each operation:
        1. Hit: when cache returns value
        2. Miss: when cache has no value
        3. Error: when cache operation fails
    3. Calculate metrics:
        1. Hit rate: hits / (hits + misses) × 100%
        2. Total queries: hits + misses
        3. Time saved: hits × average_generation_time
8. Create cache statistics endpoint:
    1. Expose endpoint (e.g., /cache/stats)
    2. Return statistics:
        1. Total hits and misses
        2. Hit rate percentage
        3. Current cache size
        4. Memory usage
    3. Useful for monitoring and optimization
9. Implement cache warming:
    1. For known common queries:
        1. Generate responses in advance
        2. Store in cache before requests arrive
        3. Ensures fast response for common cases
    2. Run during off-peak hours
    3. Update periodically
    4. Prioritize most common queries
10. Handle cache invalidation:
    1. When underlying data changes:
        1. Clear related cache entries
        2. Or wait for TTL expiration
    2. Manual invalidation capability:
        1. Clear all cache
        2. Clear specific pattern
        3. Useful for updates
    3. Version-based invalidation:

        1. Include version in cache key
        2. Increment version on data change
        3. Old cache automatically ignored
11. Optimize memory usage:
    1. Monitor Redis memory:
        1. Check with INFO memory command
        2. Track memory growth
        3. Ensure stays within limits
    2. Adjust max memory if needed:
        1. Increase if evicting too frequently
        2. Decrease if wasting memory
    3. Analyze cache entry sizes:
        1. Some responses very large
        2. Consider size limits
        3. Don't cache extremely large responses
12. Implement cache warmth monitoring:
    1. Track cache age:
        1. How long since entry created
        2. How long since last accessed
    2. Identify cold entries:
        1. Never accessed
        2. Wasting memory
        3. Could be evicted
    3. Optimize TTL based on access patterns
13. Handle Redis failures gracefully:
    1. Wrap all Redis operations in try-except
    2. If Redis unavailable:
        1. Log error
        2. Skip caching
        3. Process request normally
        4. Don't fail entire request
    3. Monitor Redis health:
        1. Set up alerts
        2. Auto-restart if needed
    4. Consider Redis redundancy for production
14. Test caching effectiveness:
    1. Run with caching disabled:
        1. Measure baseline performance
        2. Record all request times
    2. Run with caching enabled:
        1. Let cache warm up (100-200 requests)
        2. Measure performance
        3. Record cache hits and misses
    3. Calculate improvement:
        1. Average response time reduction
        2. Cache hit rate achieved
        3. Cost savings

15. Expected results:
    1. First request: Normal speed (cache miss)
    2. Subsequent identical requests: 1000x faster
    3. If 50% hit rate:
        1. Overall 2x speedup average
        2. 50% reduction in GPU load
        3. Significant cost savings
    4. If 70% hit rate:
        1. 3x+ speedup average
        2. 70% reduction in GPU load
        3. Major cost savings
16. Cache optimization strategies:
    1. Normalize prompts:
        1. Remove whitespace variations
        2. Lowercase if case-insensitive
        3. Increases hit rate
    2. Semantic deduplication:
        1. Detect similar queries
        2. Return same cached response
        3. Requires embedding comparison
    3. Partial caching:
        1. Cache document embeddings separately
        2. Reuse for similar queries
        3. More complex but powerful

---

5. APPLICATION-LEVEL OPTIMIZATIONS

5.1 Prompt Engineering for Speed

5.1.1 WHY do this?

1. Prompt length directly affects processing time
2. Every word in prompt is processed through attention mechanism:
    1. Longer prompts take more time to process
    2. Processing happens before generation even starts
    3. Adds latency to every request
3. Typical problems with prompts:
    1. Verbose instructions (200+ words)
    2. Polite conversational language
    3. Redundant explanations
    4. Unnecessary context
4. Processing overhead by prompt length:
    1. 50-word prompt: 0.5-1 second processing
    2. 100-word prompt: 1-1.5 seconds processing
    3. 200-word prompt: 2-3 seconds processing
    4. Before generation even starts
5. LLMs don't need human-style communication:
    1. No benefit from politeness

        2. Don't need detailed explanations
        3. Work better with concise instructions
        4. Clearer instructions = better results
    6. Common inefficiencies:
        1. "Please carefully analyze..."
        2. "Make sure to be thorough..."
        3. "I would like you to..."
        4. Multiple paragraphs of context

5.1.2 WHAT to do:

1. Rewrite prompts to be concise and direct
2. Remove unnecessary words:
    1. Polite language (please, carefully, thoroughly)
    2. Meta-instructions (make sure to, be detailed)
    3. Redundant context
    4. Verbose formatting
3. Use structured formats for clarity
4. Replace paragraphs with concise templates
5. Target 20-50 words for most prompts
6. Expected improvement: 30-50% faster processing

5.1.3 HOW to implement:

1. Audit existing prompts:
    1. List all prompts used in application
    2. Count words in each
    3. Identify prompts over 100 words
    4. Mark for optimization
2. Identify unnecessary elements:
    1. Polite phrases:
        1. "Please" → remove
        2. "Carefully" → remove
        3. "Thoroughly" → remove
        4. "I would like you to" → remove
    2. Meta-instructions:
        1. "Make sure to" → remove
        2. "Be detailed" → replace with specific requirements
        3. "Take your time" → remove
    3. Conversational elements:
        1. "Hello" → remove
        2. "Thank you" → remove
        3. Casual language → make direct
3. Apply optimization techniques:
    1. Use imperative voice:
        1. Before: "Please analyze this document and provide a summary"
        2. After: "Summarize this document"

    2. Remove redundancy:
        1. Before: "Extract the date and also get the time"
        2. After: "Extract date and time"
    3. Be specific without explanation:
        1. Before: "I need you to find the invoice number, which is usually at the top"
        2. After: "Extract invoice number"
    4. Use structured formats:
        1. Before: "Give me the name, address, and phone number"
        2. After: "Extract:\n1. Name\n2. Address\n3. Phone"
4. Create optimized prompt templates:
    1. For data extraction:
        1. Before: "Please carefully analyze the document and extract all relevant information including dates, amounts, and names"
        2. After: "Extract from document:\n- Date: [YYYY-MM-DD]\n- Amount: [number]\n- Name: [text]"
    2. For summarization:
        1. Before: "Please read through the entire document and provide a comprehensive summary covering all the main points"
        2. After: "Summarize in 3 points:\n- Topic\n- Findings\n- Conclusion"
    3. For classification:
        1. Before: "Based on the content, please determine what category this document belongs to"
        2. After: "Classify as: [invoice/contract/report/other]"
5. Implement output format specifications:
    1. Add clear output format to prompt:
        1. "Output as JSON only"
        2. "Respond in bullet points"
        3. "Single word answer"
    2. Prevents verbose outputs
    3. Reduces generation time
6. Test optimized prompts:
    1. Select representative test cases
    2. Run original prompt, time it
    3. Run optimized prompt, time it
    4. Compare results:
        1. Processing time reduction
        2. Output quality maintained
        3. Format consistency improved
    5. Iterate if quality degrades
7. Create prompt library:
    1. Document all optimized prompts
    2. Organize by use case:
        1. Data extraction prompts
        2. Summarization prompts
        3. Classification prompts
        4. Q&A prompts

    3. Include usage notes
    4. Store in version control
8. Establish prompt guidelines:
    1. Target word count: 20-50 words
    2. Use imperative voice
    3. No polite language
    4. No meta-instructions
    5. Specific output format
    6. No redundancy
9. Implement prompt validation:
    1. Check prompt length before using
    2. Warn if over 100 words
    3. Flag for optimization
    4. Track average prompt length
10. Examples of optimized prompts:
    1. Invoice extraction:
        1. Bad (112 words): "I need you to carefully review this invoice document and extract all the important information. Please make sure to get the invoice number, the date when it was issued, the total amount that needs to be paid, and the name of the vendor. Also look for any line items and their prices. Be thorough and make sure you don't miss anything. Format the response in a way that's easy to read and understand."
        2. Good (24 words): "Extract from invoice: invoice_number, date (YYYY-MM-DD), total_amount, vendor_name, line_items with prices. Output as JSON."
    2. Document summarization:
        1. Bad (87 words): "Please read through this entire document carefully and provide me with a comprehensive summary that covers all the main points, key findings, and important conclusions. Make sure your summary captures the essence of the document and doesn't leave out any critical information. The summary should be well-organized and easy to understand for someone who hasn't read the original document."
        2. Good (18 words): "Summarize document in 3 points: main topic, key findings, conclusions. Maximum 50 words per point."
    3. Classification:
        1. Bad (45 words): "Based on the content and context of this document, please determine what type or category it belongs to. Consider all the information present and make your best judgment about what this document is."
        2. Good (12 words): "Classify document type: [invoice, contract, report, letter, email, other]"
11. Measure overall improvement:
    1. Track prompt processing time before optimization
    2. Track prompt processing time after optimization
    3. Calculate percentage reduction

    4. Expected: 30-50% faster
    5. Document savings
12. Maintain prompt quality:
    1. Periodically review prompts
    2. Test for accuracy degradation
    3. Update as needed
    4. Share best practices across team

5.2 Pre-processing Pipeline

5.2.1 WHY do this?

1. Raw files contain much unnecessary content
2. Problems with unprocessed files:
    1. PDFs include embedded fonts, images, layout info
    2. None of this helps with text extraction
    3. LLM must process all this noise
    4. Wastes processing time and reduces accuracy
3. Common unnecessary content:
    1. Formatting metadata
    2. Page headers and footers
    3. Repeated boilerplate text
    4. Embedded images (unless using VLM)
    5. Navigation elements
    6. Advertisements
4. Document issues that slow processing:
    1. 50-page PDF: 95% boilerplate, 5% relevant content
    2. High-resolution images with simple text
    3. Excessive whitespace and formatting
    4. Mixed languages and encodings
5. Benefits of preprocessing:
    1. Extract only relevant text
    2. Clean up formatting issues
    3. Normalize content
    4. Reduce input size by 60-80%
    5. Faster LLM processing
    6. Better accuracy (less noise)

5.2.2 WHAT to do:

1. Create preprocessing pipeline for different file types
2. Extract text and discard formatting for PDFs
3. Use lightweight OCR for images
4. Clean extracted text:
    1. Remove excessive whitespace
    2. Normalize formatting
    3. Fix encoding issues

4. Remove boilerplate
5. Truncate to maximum relevant length
6. Identify and extract only relevant sections
7. Expected benefit: 1.5x faster processing, better accuracy

5.2.3 HOW to implement:

1. Design preprocessing architecture:
    1. Input: Raw file (PDF, image, text, etc.)
    2. Processing steps:
        1. Type detection
        2. Text extraction
        3. Cleaning
        4. Truncation
    3. Output: Clean, normalized text ready for LLM
2. Implement file type detection:
    1. Check file extension
    2. Verify with magic numbers (file headers)
    3. Support types:
        1. PDF documents
        2. Images (PNG, JPG, JPEG)
        3. Text files (TXT, MD, CSV)
        4. Word documents (DOCX)
        5. Others as needed
3. Implement PDF text extraction:
    1. Use PDF library (PyPDF2, pdfplumber, or similar)
    2. Extract text from each page
    3. Optimization strategies:
        1. Limit to first N pages (e.g., 10 pages)
        2. Skip pages with no text
        3. Ignore images unless necessary
        4. Discard formatting information
    4. Combine page text with separators
    5. Typical processing: 100-500ms per document
4. Implement image text extraction:
    1. Use OCR library (Tesseract, pytesseract)
    2. Before OCR, optimize image:
        1. Resize if very large (over 4K resolution)
        2. Downscale to 1080p or lower
        3. 4x reduction in processing time
        4. No accuracy loss for text
    3. Convert to grayscale
    4. Apply OCR to extract text
    5. Typical processing: 500ms-2s per image
5. Implement text file handling:
    1. Simple read for .txt, .md files
    2. Handle encoding issues:

        1. Try UTF-8 first
        2. Fall back to latin-1
        3. Detect encoding if needed
    3. Very fast: under 100ms
6. Implement text cleaning:
    1. Remove excessive whitespace:
        1. Replace multiple spaces with single space
        2. Replace multiple newlines with single newline
        3. Trim leading/trailing whitespace
    2. Normalize formatting:
        1. Standardize quotes
        2. Fix common encoding issues
        3. Remove special characters if not needed
    3. Remove common boilerplate:
        1. "Page X of Y"
        2. Repeated headers/footers
        3. Confidentiality notices
        4. Standard legal disclaimers
7. Implement smart truncation:
    1. Set maximum length (e.g., 3000 words)
    2. If document exceeds maximum:
        1. Strategy 1: Take first N words only
        2. Strategy 2: Take first and last sections
        3. Strategy 3: Extract most relevant paragraphs
    3. Add marker indicating truncation
    4. Log original vs truncated length
8. Implement section extraction:
    1. For long documents, extract relevant sections:
        1. Use keywords to identify relevant parts
        2. Extract paragraphs containing keywords
        3. Combine into focused subset
    2. Example for invoice processing:
        1. Look for sections with numbers
        2. Look for dates
        3. Look for currency symbols
        4. Extract those sections only
9. Create preprocessing class structure:
    1. Initialize with configuration:
        1. Maximum length
        2. Supported file types
        3. Cleaning rules
    2. Main process() method:
        1. Accepts file path
        2. Detects type
        3. Extracts text
        4. Cleans text
        5. Returns clean text

    3. Type-specific methods:
        1. _extract_pdf()
        2. _extract_image()
        3. _extract_text()
    4. Utility methods:
        1. clean_text()
        2. truncate()
        3. extract_sections()
10. Implement error handling:
    1. Handle missing files
    2. Handle corrupted PDFs
    3. Handle unsupported formats
    4. Handle OCR failures
    5. Return meaningful error messages
    6. Log errors for investigation
11. Add processing metadata:
    1. Track original file size
    2. Track extracted text length
    3. Track processing time
    4. Log warnings for issues
    5. Return metadata with clean text
12. Optimize processing speed:
    1. Cache extracted text by file hash
    2. Skip re-processing identical files
    3. Use fastest libraries available
    4. Parallelize when processing multiple files
    5. Lazy load heavy dependencies
13. Test preprocessing pipeline:
    1. Test with various PDF types:
        1. Text-based PDFs
        2. Scanned PDFs
        3. Mixed content PDFs
    2. Test with various images:
        1. High resolution
        2. Low quality
        3. Different formats
    3. Test with edge cases:
        1. Empty files
        2. Corrupted files
        3. Non-text PDFs
        4. Encrypted PDFs
14. Measure preprocessing benefit:
    1. Benchmark without preprocessing:
        1. Send raw file to LLM
        2. Measure time and accuracy
    2. Benchmark with preprocessing:
        1. Preprocess file

        2. Send clean text to LLM
        3. Measure time and accuracy
    3. Calculate improvement:
        1. Speed: Typically 1.5x faster
        2. Accuracy: Often 10-20% better
        3. Cost: Lower token usage
15. Integration with LLM pipeline:
    1. File upload → Preprocessing → LLM → Response
    2. Preprocessing happens automatically
    3. Clean text passed to LLM
    4. No changes needed in LLM code
    5. Modular and maintainable
16. Handle special cases:
    1. For tables:
        1. Try to preserve structure
        2. Use CSV-like formatting
        3. Helps LLM understand relationships
    2. For lists:
        1. Maintain numbering
        2. Keep hierarchy
        3. Improves comprehension
    3. For forms:
        1. Extract field labels and values
        2. Format as key-value pairs
        3. Clear structure for LLM

5.3 Dynamic Context Sizing

5.3.1 WHY do this?

1. Fixed large context wastes resources
2. Processing cost grows quadratically with context size:
    1. 2048 tokens: Baseline processing time
    2. 4096 tokens: 4x slower than 2048
    3. 8192 tokens: 16x slower than 2048
3. Common mismatch scenarios:
    1. 3-page invoice with 8K context set
    2. Only needs 2K context
    3. Processing 16x slower than necessary
4. Most documents don't need maximum context:
    1. Typical invoice: 500-1000 tokens
    2. Typical email: 200-500 tokens
    3. Typical contract: 2000-3000 tokens
    4. Reports: 5000-8000 tokens
5. One-size-fits-all approach is inefficient:
    1. Small documents penalized with large context
    2. Large documents might not fit

        3. Inconsistent performance
    6. Dynamic sizing benefits:
        1. Match context to actual need
        2. Fast processing for small documents
        3. Adequate capacity for large documents
        4. Consistent optimization

5.3.2 WHAT to do:

1. Measure each document's actual size before processing
2. Calculate required context based on content length
3. Set context dynamically per request
4. Use tiered approach:
    1. Small documents: 2048 context
    2. Medium documents: 4096 context
    3. Large documents: 8192 context
5. Add buffer for prompts and responses
6. Expected benefit: 2-4x faster for small documents

5.3.3 HOW to implement:

1. Create context sizing function:
    1. Input: Document text content
    2. Processing:
        1. Count words in document
        2. Estimate tokens (words × 1.33)
        3. Add buffer (500 tokens)
        4. Round up to standard size
    3. Output: Optimal context size
2. Implement word counting:
    1. Split text on whitespace
    2. Count resulting elements
    3. Fast operation: under 1ms
    4. Accurate enough for estimation
3. Convert words to tokens:
    1. Rule of thumb: 1 word ≈ 0.75 tokens
    2. Or: 1 token ≈ 1.33 words
    3. Formula: tokens = words × 1.33
    4. Slightly overestimate for safety
4. Add buffer for overhead:
    1. System prompt: 50-100 tokens
    2. User instructions: 100-200 tokens
    3. Output space: 200-500 tokens
    4. Total buffer: 500 tokens recommended
    5. Ensures enough space for full interaction
5. Implement context tiers:
    1. Calculate needed tokens

    2. Round up to nearest tier:
        1. If ≤ 2048 → use 2048
        2. If 2049-4096 → use 4096
        3. If 4097-8192 → use 8192
        4. If > 8192 → use 8192 (truncate document)
    3. Return selected tier
6. Create dynamic sizing logic:
    1. Function: calculate_optimal_context(text)
    2. Steps:
        1. word_count = count_words(text)
        2. token_estimate = word_count × 1.33
        3. needed_tokens = token_estimate + 500
        4. if needed_tokens ≤ 2048: return 2048
        5. elif needed_tokens ≤ 4096: return 4096
        6. else: return 8192
    3. Simple, fast, effective
7. Integrate with processing pipeline:
    1. After preprocessing document:
        1. Get clean text
        2. Calculate optimal context
        3. Pass to LLM with context parameter
    2. No user intervention needed
    3. Automatic optimization
    4. Per-request adaptation
8. Handle edge cases:
    1. Empty documents:
        1. Use minimum context (2048)
        2. Fast processing
        3. No errors
    2. Extremely large documents:
        1. Use maximum context (8192)
        2. Truncate document if necessary
        3. Log warning
        4. Consider chunking strategy
    3. Multi-language documents:
        1. Different languages have different token densities
        2. Add 20% extra buffer
        3. Prevents context overflow
9. Implement context overflow handling:
    1. If document exceeds maximum context:
        1. Truncate to fit
        2. Strategies:
            1. Keep first N tokens
            2. Keep first and last sections
            3. Extract most relevant parts
        3. Log truncation event
        4. Inform user if appropriate

10. Track context usage statistics:
    1. Log selected context for each request
    2. Aggregate statistics:
        1. Percentage using 2048: X%
        2. Percentage using 4096: Y%
        3. Percentage using 8192: Z%
    3. Validate sizing strategy
    4. Adjust tiers if needed
11. Optimize for common patterns:
    1. If 80% of documents use 2048:
        1. Strategy is working well
        2. Significant time savings
    2. If most use 8192:
        1. Documents larger than expected
        2. May need to adjust approach
        3. Consider preprocessing improvement
12. Create size-specific optimizations:
    1. For 2048 context:
        1. Can use more aggressive quantization
        2. Can increase parallel processing
        3. Fast turnaround
    2. For 8192 context:
        1. Use higher quality quantization
        2. Reduce parallel processing
        3. Prioritize accuracy
13. Implement smart pre-checking:
    1. Before full processing:
        1. Quick size estimate
        2. Reject if too large
        3. Save processing time
        4. Better error messages
    2. Suggest alternatives:
        1. "Document too large, please summarize"
        2. "Upload smaller sections"
        3. Helpful user experience
14. Test dynamic sizing:
    1. Test with small documents (500 words):
        1. Should use 2048 context
        2. Fast processing
        3. Good accuracy
    2. Test with medium documents (2000 words):
        1. Should use 4096 context
        2. Balanced speed/capacity
        3. Good accuracy
    3. Test with large documents (6000 words):
        1. Should use 8192 context
        2. Slower but handles full content

3. Best accuracy
4. Compare to fixed 8192 context:
    1. Small docs: 16x faster with dynamic
    2. Medium docs: 4x faster with dynamic
    3. Large docs: Same speed

15. Document sizing decisions:
    1. Create internal documentation
    2. Explain tier selection logic
    3. Document buffer calculations
    4. Share with team
    5. Maintain as requirements change

---

6. ACCURACY IMPROVEMENTS

6.1 RAG (Retrieval-Augmented Generation)

6.1.1 WHY do this?

1. LLMs have fundamental limitations with long documents
2. Problems with direct long document processing:
    1. Context window limits (max 8K-32K tokens)
    2. Attention degrades over long contexts
    3. Model may miss relevant details
    4. Must scan through entire document
    5. Noise from irrelevant sections
3. Example problem scenario:
    1. 50-page technical manual
    2. User asks specific question
    3. Answer is in 2 paragraphs on page 37
    4. LLM must process all 50 pages
    5. May miss the relevant paragraphs
    6. Wastes processing on irrelevant content
4. RAG solves this by:
    1. Finding relevant sections first (semantic search)
    2. Only sending relevant content to LLM
    3. Concentrated, focused context
    4. Much higher accuracy
5. Accuracy improvements:
    1. 40-60% better for specific queries
    2. Especially effective for long documents
    3. Better for multi-document search
    4. Essential for knowledge bases
6. Trade-offs:
    1. Adds 200-500ms overhead (retrieval time)
    2. More complex to implement
    3. Requires embedding model
    4. But accuracy gains usually worth it

6.1.2 WHAT to do:

1. Implement two-stage retrieval-augmented process
2. Stage 1: Semantic search to find relevant content
3. Stage 2: Send only relevant content to LLM
4. Requirements:
    1. Break documents into chunks
    2. Create embeddings for each chunk
    3. Store in vector database (FAISS)
    4. Implement retrieval logic
    5. Integrate with LLM pipeline

6.1.3 HOW to implement:

1. Install required libraries:
    1. sentence-transformers: For embeddings
    2. faiss-cpu or faiss-gpu: For vector search
    3. Both are lightweight and fast
    4. Installation takes 5-10 minutes
2. Choose embedding model:
    1. Recommended: 'all-MiniLM-L6-v2'
    2. Characteristics:
        1. 384-dimensional vectors
        2. Fast inference (10-20ms per chunk)
        3. Good accuracy for general text
        4. Lightweight (80MB model)
    3. Alternatives for specific domains:
        1. 'all-mpnet-base-v2': Better accuracy, slower
        2. 'paraphrase-MiniLM-L6-v2': Good for semantic similarity
3. Implement document chunking:
    1. Purpose: Break large documents into searchable pieces
    2. Chunk size guidelines:
        1. 300-500 words per chunk (recommended)
        2. 200 words minimum (too small loses context)
        3. 1000 words maximum (too large reduces precision)
    3. Chunking strategy:
        1. Split on word boundaries
        2. Add overlap between chunks (50-100 words)
        3. Overlap ensures context continuity
        4. Prevents losing information at boundaries
    4. Implementation steps:
        1. Split text into words
        2. Group into chunks of target size
        3. Create overlapping segments
        4. Store chunk text and metadata
4. Create embeddings for chunks:
    1. Load embedding model

    2. For each chunk:
        1. Pass chunk text to model
        2. Get 384-dimensional vector
        3. Takes 10-20ms per chunk
    3. Batch processing for efficiency:
        1. Process 32-64 chunks at once
        2. 10-20x faster than one-by-one
        3. Typical: 100 chunks in 1-2 seconds
    4. Store embeddings with chunk references
5. Build FAISS index:
    1. Create index structure:
        1. Use IndexFlatL2 for simplicity
        2. Measures L2 (Euclidean) distance
        3. Exact search, no approximation
    2. Add embeddings to index:
        1. Convert embeddings to numpy array
        2. Ensure float32 dtype
        3. Add to FAISS index
        4. Fast operation: 1000 vectors in <100ms
    3. Index is now ready for search
6. Implement query embedding:
    1. When user submits query:
        1. Convert query text to embedding
        2. Use same embedding model
        3. Takes 10-20ms
        4. Returns same dimensional vector as chunks
7. Implement similarity search:
    1. Use FAISS to find similar chunks:
        1. Pass query embedding to index.search()
        2. Specify k (number of results, typically 3-5)
        3. Returns distances and indices
        4. Very fast: under 1ms for 1000 chunks
    2. Retrieve chunk text:
        1. Use returned indices
        2. Look up original chunk text
        3. These are most relevant to query
8. Combine retrieved chunks:
    1. Take top k chunks (typically 3)
    2. Combine into focused context:
        1. Separate with newlines
        2. Add markers if helpful
        3. Keep under context limit
    3. This becomes LLM input
9. Construct RAG prompt:
    1. Structure:
        1. "Context from document:"
        2. [Retrieved chunks]

        3. "Question: [user query]"
        4. "Answer based on context above:"
    2. Clear separation of context and query
    3. Instructs model to use context
10. Send to LLM:
    1. Use focused context (not full document)
    2. Much smaller input
    3. All content is relevant
    4. Higher accuracy
    5. Faster processing
11. Create RAG class structure:
    1. Initialization:
        1. Load embedding model
        2. Initialize empty index
        3. Store chunk list
    2. Methods:
        1. chunk_document(text): Break into chunks
        2. build_index(document): Create FAISS index
        3. retrieve(query, k): Find relevant chunks
        4. query_with_rag(query): Full RAG pipeline
    3. Clean, reusable structure
12. Handle multiple documents:
    1. Index chunks from all documents
    2. Add document metadata to chunks:
        1. Document ID
        2. Chunk number
        3. Page number if available
    3. Return document source with results
    4. Enables multi-document search
13. Optimize retrieval parameters:
    1. Number of chunks (k):
        1. k=3: Focused, fast (recommended starting point)
        2. k=5: More context, slightly slower
        3. k=10: Comprehensive, may include noise
    2. Chunk size:
        1. Smaller: More precise retrieval
        2. Larger: More context per chunk
        3. Test to find optimal for your data
    3. Overlap amount:
        1. More overlap: Better context continuity
        2. Less overlap: More unique content
        3. 50-100 words typical
14. Implement re-ranking (optional):
    1. After initial retrieval:
        1. Re-score chunks with different metric
        2. Consider query-chunk relevance
        3. Reorder by score

    2. Improves precision
    3. Adds minimal overhead
15. Test RAG effectiveness:
    1. Create test set:
        1. Long documents
        2. Specific questions
        3. Known correct answers
    2. Test without RAG:
        1. Send full document to LLM
        2. Measure accuracy
        3. Record as baseline
    3. Test with RAG:
        1. Use retrieval pipeline
        2. Measure accuracy
        3. Compare to baseline
    4. Expected improvement: 40-60% better accuracy
16. Monitor RAG performance:
    1. Track retrieval time
    2. Track retrieval quality:
        1. Are relevant chunks retrieved?
        2. Manual spot-checking
        3. User feedback
    3. Track end-to-end latency
    4. Optimize if needed
17. RAG vs full document trade-offs:
    1. RAG advantages:
        1. Much higher accuracy for specific queries
        2. Works with documents beyond context limit
        3. Faster processing (smaller input)
        4. Better for multi-document scenarios
    2. RAG disadvantages:
        1. Adds retrieval overhead (200-500ms)
        2. More complex to implement
        3. Requires embedding model
        4. May miss relevant info if retrieval fails
    3. When to use RAG:
        1. Documents over 3000 words
        2. Specific question-answering
        3. Multi-document search
        4. Knowledge base queries
    4. When to skip RAG:
        1. Short documents (under 1000 words)
        2. General summaries
        3. Simple classification
        4. Speed is critical

6.2 Few-Shot Prompting

6.2.1 WHY do this?

1. LLMs perform significantly better with examples
2. Zero-shot limitations:
    1. Model must guess at exact format
    2. Uncertain about level of detail
    3. May miss edge case handling
    4. Output format may vary
3. Few-shot benefits by providing examples:
    1. Shows exact output structure
    2. Demonstrates desired level of detail
    3. Illustrates edge case handling
    4. Ensures format consistency
4. Accuracy improvements:
    1. 25-35% better extraction accuracy
    2. More consistent formatting
    3. Better handling of variations
    4. Especially valuable for structured data
5. When few-shot is most effective:
    1. Structured data extraction (JSON, CSV)
    2. Specific formatting requirements
    3. Domain-specific tasks
    4. Consistent output format needed
6. Examples teach patterns:
    1. Model learns from input-output pairs
    2. Applies pattern to new inputs
    3. More reliable than instructions alone

6.2.2 WHAT to do:

1. Modify prompts to include 2-4 high-quality examples
2. Each example shows:
    1. Input similar to actual use case
    2. Exact desired output format
    3. Proper handling of edge cases
3. Structure: Task description → Examples → Actual input
4. Keep examples concise but representative
5. Quality over quantity (3 good examples beats 10 mediocre)

6.2.3 HOW to implement:

1. Identify tasks needing few-shot prompting:
    1. Data extraction tasks
    2. Format conversion tasks
    3. Classification with specific categories
    4. Any task with structured output

2. Create example selection criteria:
    1. Representative of real data:
        1. Use actual examples from your dataset
        2. Cover common patterns
        3. Include edge cases
    2. Diversity:
        1. Show different variations
        2. Different formats
        3. Different complexity levels
    3. Quality:
        1. Examples must be perfect
        2. Correct format
        3. Complete information
        4. No errors
3. Design prompt structure:
    1. Section 1: Task description (brief)
    2. Section 2: Examples (2-4 examples)
    3. Section 3: Actual input
    4. Section 4: Output marker
    5. Clear separation between sections
4. Format each example:
    1. Label clearly:
        1. "Example 1:"
        2. "Input: [example input]"
        3. "Output: [example output]"
    2. Keep consistent structure across examples
    3. Use same formatting in all examples
5. Create few-shot template for invoice extraction:
    1. Task description:
        1. "Extract structured data from invoice text"
    2. Example 1:
        1. Input: "Invoice #12345, Date: 2024-01-15, Total: $1,500"
        2. Output: {"invoice": "12345", "date": "2024-01-15", "total": 1500}
    3. Example 2:
        1. Input: "INV-98765 dated 2024-02-20, Amount: $2,750"
        2. Output: {"invoice": "98765", "date": "2024-02-20", "total": 2750}
    4. Example 3 (edge case):
        1. Input: "Invoice number: A-5555, Date: March 10, 2024, Total: 890 dollars"
        2. Output: {"invoice": "A-5555", "date": "2024-03-10", "total": 890}
    5. Actual input:
        1. "Now complete this:"
        2. "Input: [user's actual invoice text]"
        3. "Output:"
6. Implement programmatically:
    1. Create function: create_few_shot_prompt()
    2. Parameters:

        1. task_description: Brief explanation
        2. examples: List of (input, output) tuples
        3. new_input: User's actual input
    3. Function builds prompt:
        1. Start with task description
        2. Add each example with formatting
        3. Add actual input section
        4. Add output marker
        5. Return complete prompt
7. Store example sets:
    1. For each task type, maintain example set:
        1. Invoice extraction: 3 examples
        2. Date formatting: 3 examples
        3. Entity extraction: 3 examples
    2. Store in configuration or database
    3. Easy to update and maintain
8. Choose optimal number of examples:
    1. Test with different counts:
        1. 1 example (one-shot)
        2. 2 examples
        3. 3 examples
        4. 5 examples
    2. Measure accuracy for each
    3. Measure processing time for each
    4. Find sweet spot (usually 2-3)
    5. More examples = slower but not always more accurate
9. Handle example length:
    1. Keep examples concise:
        1. Show pattern, not exhaustive detail
        2. 20-50 words per example typical
    2. If inputs are long:
        1. Use representative excerpts
        2. Show full output structure
    3. Balance: Enough detail to learn, not so much it slows down
10. Create examples for common patterns:
    1. Date formatting:
        1. Example 1: "January 15, 2024" → "2024-01-15"
        2. Example 2: "03/20/2024" → "2024-03-20"
        3. Example 3: "Dec 5, 23" → "2023-12-05"
    2. Currency extraction:
        1. Example 1: "$1,500.00" → 1500.00
        2. Example 2: "1234.56 USD" → 1234.56
        3. Example 3: "€ 890" → 890.00
    3. Entity extraction:
        1. Example 1: "John Smith from Acme Corp" → {"name": "John Smith", "company": "Acme Corp"}

        2. Example 2: "Sarah Johnson, TechStart Inc." → {"name": "Sarah Johnson", "company": "TechStart Inc."}
11. Include edge cases in examples:
12. Missing information:
    1. Show how to handle with null or empty
13. Ambiguous formats:
    1. Show preferred interpretation
14. Multiple valid answers:
    1. Show which to prefer
15. Error cases:
    1. Show how to indicate can't extract
16. Test few-shot effectiveness:
17. Create test set (20-50 items)
18. Test with zero-shot (no examples):
    1. Run on test set
    2. Measure accuracy
    3. Record as baseline
19. Test with few-shot (3 examples):
    1. Run on same test set
    2. Measure accuracy
    3. Compare to baseline
20. Expected improvement: 25-35% better
21. Also check format consistency
22. Optimize examples over time:
23. Monitor errors in production
24. Identify common failure patterns
25. Add example showing correct handling
26. Update example set
27. Test improvement
28. Iterative refinement
29. Trade-offs to consider:
30. Longer prompts:
    1. Examples add 100-200 tokens
    2. Slightly slower processing
    3. Usually 0.5-1 second overhead
31. Worth it for:
    1. Structured data extraction
    2. Consistent formatting requirements
    3. Complex patterns
32. Skip for:
    1. Simple tasks (classification into 2 categories)
    2. When speed is critical
    3. Very well-defined simple tasks
33. Combine with other techniques:
    1. Few-shot + low temperature:
        1. Examples + deterministic = very consistent
        2. Best for data extraction

2. Few-shot + RAG:
    1. Examples + relevant context = high accuracy
    2. Best for complex queries
3. Few-shot + verification:
    1. Examples + double-checking = maximum accuracy
    2. Best for critical data

6.3 Temperature and Sampling Tuning

6.3.1 WHY do this?

1. Temperature controls randomness in model outputs
2. Default temperature (0.7-0.8) is optimized for creative tasks
3. Problems with wrong temperature:
    1. Too high: Inconsistent outputs, errors, hallucinations
    2. Too low: Sometimes too rigid, repetitive
    3. One-size-fits-all doesn't work
4. Different tasks need different settings:
    1. Data extraction: Need deterministic, consistent
    2. Creative writing: Need variety, interesting
    3. Summaries: Need balance
5. Temperature mechanics:
    1. Controls probability distribution over next tokens
    2. Low (0.1): Always picks most likely token
    3. High (1.0): Considers many possibilities
    4. Affects output predictability
6. Related parameters also matter:
    1. top_k: Limits number of token choices
    2. top_p: Limits cumulative probability
    3. repeat_penalty: Prevents repetition

6.3.2 WHAT to do:

1. Set temperature based on task type
2. For factual/extraction tasks:
    1. Temperature: 0.1-0.2 (very deterministic)
    2. top_k: 10-20 (focused choices)
    3. top_p: 0.9 (limit diversity)
3. For creative tasks:
    1. Temperature: 0.7-0.8 (more varied)
    2. top_k: 40-50 (more choices)
    3. top_p: 0.95 (allow diversity)

4. For summaries and balanced tasks:
    1. Temperature: 0.4-0.6 (moderate)

    2. top_k: 30-40 (balanced choices)
    3. top_p: 0.92-0.95 (moderate diversity)
5. Create configuration presets for different task types
6. Test different settings on your specific data
7. Expected improvement: 15-25% better accuracy for factual tasks

6.3.3 HOW to implement:

1. Understand temperature parameter:
    1. Range: 0.0 to 2.0
    2. 0.0: Completely deterministic (always same output)
    3. 0.1-0.2: Highly deterministic (minimal variation)
    4. 0.5-0.7: Balanced (some creativity)
    5. 0.8-1.0: Creative (significant variation)
    6. Above 1.0: Very random (often nonsensical)
    7. Default: Usually 0.7-0.8
2. Understand top_k parameter:
    1. Limits vocabulary to k most likely tokens
    2. Lower = more focused output
    3. Higher = more diverse output
    4. Range typically: 10-100
    5. 10: Very focused
    6. 40-50: Balanced
    7. 100: Very diverse
3. Understand top_p parameter:
    1. Also called "nucleus sampling"
    2. Considers tokens until cumulative probability reaches p
    3. Range: 0.0 to 1.0
    4. 0.9: Focused (90% probability mass)
    5. 0.95: Balanced (95% probability mass)
    6. 1.0: All tokens considered
4. Understand repeat_penalty parameter:
    1. Penalizes tokens that have appeared before
    2. Prevents repetitive output
    3. Range typically: 1.0-1.5
    4. 1.0: No penalty
    5. 1.1: Slight penalty (recommended)
    6. 1.3+: Strong penalty (may affect quality)
5. Create configuration presets:
    1. Accuracy preset (for data extraction):
        1. Temperature: 0.1
        2. top_k: 10
        3. top_p: 0.9
        4. repeat_penalty: 1.2
        5. Rationale: Maximum determinism and consistency
    2. Balanced preset (for general tasks):
        1. Temperature: 0.5

        2. top_k: 40
        3. top_p: 0.95
        4. repeat_penalty: 1.1
        5. Rationale: Good balance of consistency and naturalness
    3. Creative preset (for summaries, rewrites):
        1. Temperature: 0.8
        2. top_k: 50
        3. top_p: 0.95
        4. repeat_penalty: 1.1
        5. Rationale: Allow natural variation and creativity
6. Implement in API calls:
    1. Add options parameter to Ollama request
    2. Include sampling parameters:
        1. "temperature": value
        2. "top_k": value
        3. "top_p": value
        4. "repeat_penalty": value
    3. Different calls can use different presets
    4. Select preset based on task type
7. Test temperature effects:
    1. Create test set with 20-30 examples
    2. Test same input with different temperatures:
        1. Run with temperature 0.1
        2. Run with temperature 0.5
        3. Run with temperature 0.8
    3. Compare outputs:
        1. At 0.1: Highly consistent, same answer repeatedly
        2. At 0.5: Some variation, mostly consistent
        3. At 0.8: Significant variation, more creative
    4. Choose based on your needs
8. Test on factual extraction task:
    1. Task: Extract invoice number from text
    2. Test with temperature 0.1:
        1. Run same input 10 times
        2. Should get identical output each time
        3. Correct extraction consistently
    3. Test with temperature 0.8:
        1. Run same input 10 times
        2. May get variations in output
        3. Sometimes adds unnecessary text
        4. Less reliable for extraction
    4. Conclusion: Use 0.1-0.2 for extraction
9. Test on summary task:
    1. Task: Summarize document
    2. Test with temperature 0.1:
        1. Run same document 5 times
        2. Get identical summary each time

        3. May feel mechanical
    3. Test with temperature 0.7:
        1. Run same document 5 times
        2. Get varied but similar summaries
        3. More natural language
        4. Still accurate
    4. Conclusion: Use 0.6-0.8 for summaries
10. Implement task-based selection:
    1. Create mapping: task_type → configuration
    2. Examples:
        1. "extraction" → accuracy_preset
        2. "summary" → creative_preset
        3. "classification" → accuracy_preset
        4. "rewrite" → creative_preset
    3. Automatically apply correct preset
    4. No manual configuration per request
11. Fine-tune for your specific data:
    1. Start with recommended presets
    2. Test on your actual documents
    3. Adjust if needed:
        1. If extraction inconsistent → lower temperature
        2. If summaries too dry → increase temperature
        3. If too repetitive → increase repeat_penalty
    4. Document optimal settings
    5. Use consistently
12. Monitor output quality:
    1. Track consistency metrics:
        1. For same input, how often same output?
        2. For extraction, accuracy rate?
    2. Track quality metrics:
        1. User feedback
        2. Error rates
        3. Manual review samples
    3. Adjust parameters if quality degrades
13. Combine with other techniques:
    1. Low temperature + few-shot examples:
        1. Maximum consistency and accuracy
        2. Best for structured extraction
    2. Moderate temperature + RAG:
        1. Accurate but natural responses
        2. Best for Q&A systems
    3. High temperature + creative prompts:
        1. Varied, interesting outputs
        2. Best for content generation
14. Common mistakes to avoid:
    1. Using high temperature for data extraction:
        1. Causes inconsistent outputs

        2. Extraction errors
        3. Format violations
    2. Using very low temperature for creative tasks:
        1. Output feels robotic
        2. Lacks natural variation
        3. May be repetitive
    3. Not tuning for your specific use case:
        1. Defaults may not be optimal
        2. Test and optimize
        3. Document findings
15. Handle edge cases:
    1. If getting repetitive output:
        1. Increase repeat_penalty to 1.2-1.3
        2. Or increase temperature slightly
        3. Or increase top_k
    2. If getting inconsistent extraction:
        1. Decrease temperature to 0.05-0.1
        2. Decrease top_k to 5-10
        3. Add few-shot examples
    3. If output too verbose:
        1. Add output length limit to prompt
        2. Lower temperature
        3. Adjust repeat_penalty
16. Benchmark improvement:
    1. Baseline (default settings):
        1. Test on sample set
        2. Measure accuracy
        3. Record results
    2. Optimized (tuned settings):
        1. Test on same sample set
        2. Measure accuracy
        3. Compare to baseline
    3. Expected improvement:
        1. Extraction tasks: 15-25% better accuracy
        2. Classification: 10-20% better consistency
        3. Summaries: Subjective but often preferred

6.4 Multi-Step Verification

6.4.1 WHY do this?

1. LLMs make mistakes despite high accuracy
2. Common errors:
    1. Hallucinating data not in document
    2. Misreading numbers (especially similar ones)
    3. Extracting from wrong section
    4. Confusing similar field names

        5. Format errors in output
    3. Single-pass extraction error rates:
        1. Simple tasks: 5-10% error rate
        2. Complex tasks: 15-25% error rate
        3. Critical data: Unacceptable risk
    4. Verification improves accuracy because:
        1. Checking is easier than initial extraction
        2. Model catches its own errors
        3. Second look finds missed details
        4. Different perspective reveals mistakes
    5. Two-pass accuracy improvement:
        1. 15-25% reduction in errors
        2. Especially effective for numerical data
        3. Critical for high-stakes applications
    6. Trade-off consideration:
        1. Doubles processing time
        2. But dramatically improves reliability
        3. Worth it for important data

6.4.2 WHAT to do:

1. Implement two-pass verification process
2. Pass 1: Initial extraction
    1. Extract data from document
    2. Format as structured output
    3. Record results
3. Pass 2: Verification
    1. Show model original document
    2. Show extracted data
    3. Ask to verify accuracy
    4. Request corrections if needed
4. Use verification results:
    1. If verified: Use original extraction
    2. If corrections provided: Use corrected version
5. Apply selectively based on importance

6.4.3 HOW to implement:

1. Design extraction prompt (Pass 1):
    1. Clear instructions for extraction
    2. Specify output format (JSON, CSV, etc.)
    3. Request specific fields
    4. Example:
        1. "Extract from this document:"
        2. "- Invoice number"
        3. "- Date (YYYY-MM-DD format)"
        4. "- Total amount"

        5. "- Vendor name"
        6. "Output as JSON"
2. Execute first pass:
    1. Send document with extraction prompt
    2. Get initial extraction result
    3. Parse the response
    4. Store extracted data
    5. Record processing time
3. Design verification prompt (Pass 2):
    1. Include original document
    2. Include extracted data
    3. Clear verification instructions
    4. Example structure:
        1. "Original document:"
        2. "[full document text]"
        3. ""
        4. "Extracted data:"
        5. "[JSON from first pass]"
        6. ""
        7. "Verify this extraction is accurate."
        8. "Check each field against the document."
        9. "If all correct, respond: VERIFIED"
        10. "If errors found, provide corrected JSON."
4. Execute second pass:
    1. Send verification prompt
    2. Get verification result
    3. Parse response
    4. Check for "VERIFIED" keyword
    5. Record verification outcome
5. Implement result handling logic:
    1. If response contains "VERIFIED":
        1. Use original extraction
        2. Mark as verified
        3. Log successful verification
    2. If response contains corrections:
        1. Parse corrected JSON
        2. Use corrected version instead
        3. Log that correction was needed
    3. If response unclear:
        1. Default to original extraction
        2. Log ambiguous verification
        3. Flag for manual review
6. Create verification function:
    1. Function: extract_with_verification(document_text)
    2. Steps:
        1. Build extraction prompt
        2. Call LLM (first pass)

        3. Parse extraction result
        4. Build verification prompt
        5. Call LLM (second pass)
        6. Parse verification result
        7. Return final extraction with metadata
    3. Metadata includes:
        1. Was verification needed
        2. What was corrected
        3. Confidence level
7. Optimize verification for speed:
    1. Use lower temperature for verification:
        1. First pass: temperature 0.2
        2. Second pass: temperature 0.1
        3. Verification more deterministic
    2. Use smaller context if possible:
        1. Focus verification on extracted fields
        2. Don't need full document for simple checks
    3. Batch verification when possible:
        1. Verify multiple extractions together
        2. Reduces API calls
8. Implement selective verification:
    1. Not all extractions need verification
    2. Decision criteria:
        1. High-value data: Always verify
        2. Financial amounts: Always verify
        3. Legal/compliance data: Always verify
        4. Simple classification: Skip verification
        5. Low-stakes data: Skip verification
    3. Configuration:
        1. task_requires_verification = {
        2. "invoice_extraction": True,
        3. "document_classification": False,
        4. "entity_extraction": True,
        5. "simple_summary": False
        6. }
9. Implement confidence-based verification:
    1. First pass includes confidence score
    2. Verification rules:
        1. If confidence > 90%: Skip verification
        2. If confidence 70-90%: Verify
        3. If confidence < 70%: Verify + flag for manual review
    3. Saves verification on high-confidence extractions
    4. Balances speed and accuracy
10. Track verification statistics:
    1. Count total verifications
    2. Count successful verifications (VERIFIED)
    3. Count corrections needed

    4. Calculate correction rate:
        1. correction_rate = corrections / total_verifications
        2. If > 30%: First pass needs improvement
        3. If < 10%: Verification working well
    5. Track types of corrections:
        1. Number errors
        2. Date format errors
        3. Field confusion errors
        4. Missing data errors
11. Use verification insights:
    1. High correction rate indicates problems:
        1. Prompt may be unclear
        2. Need better examples
        3. Temperature may be too high
        4. Need better preprocessing
    2. Common error patterns:
        1. If often correcting same field
        2. Add specific instruction for that field
        3. Add example showing correct handling
        4. Improve first-pass prompt
12. Implement three-pass for critical data:
    1. Pass 1: Initial extraction
    2. Pass 2: Verification
    3. Pass 3: Final check if discrepancies
    4. Use for extremely critical data:
        1. Medical information
        2. Financial transactions
        3. Legal documents
        4. Compliance data
13. Handle verification disagreements:
    1. If verification contradicts original:
        1. Log the discrepancy
        2. Flag for manual review
        3. Use corrected version but with uncertainty flag
    2. If verification unclear:
        1. Attempt re-verification with clearer prompt
        2. If still unclear, flag for manual review
    3. Set confidence thresholds:
        1. High confidence: Use automatically
        2. Medium confidence: Review sample
        3. Low confidence: Review all
14. Optimize verification prompts:
    1. Be specific about what to check:
        1. "Verify invoice number matches document"
        2. "Verify date format is YYYY-MM-DD"
        3. "Verify amount is numeric"
    2. Provide checklist:

        1. "Check: ✓ Invoice number present"
        2. "Check: ✓ Date in correct format"
        3. "Check: ✓ Amount is reasonable"
    3. Request specific corrections:
        1. "If errors, provide corrected field only"
        2. "Don't repeat correct fields"
15. Test verification effectiveness:
    1. Create test set with known errors:
        1. Intentionally introduce mistakes
        2. See if verification catches them
    2. Test categories:
        1. Wrong numbers: Should catch
        2. Wrong dates: Should catch
        3. Missing fields: Should catch
        4. Format errors: Should catch
    3. Measure catch rate:
        1. errors_caught / total_errors
        2. Target: > 80% catch rate
        3. Adjust prompts if lower
16. Cost-benefit analysis:
    1. Costs of verification:
        1. 2x processing time
        2. 2x API calls
        3. Added complexity
    2. Benefits of verification:
        1. 15-25% better accuracy
        2. Fewer downstream errors
        3. Higher confidence
        4. Reduced manual review
    3. When worth it:
        1. Error cost > processing cost
        2. Critical accuracy requirements
        3. Regulatory compliance needs
    4. When not worth it:
        1. Non-critical data
        2. Speed is priority
        3. High initial accuracy (> 95%)
17. Implement async verification:
    1. For non-blocking scenarios:
        1. Return initial extraction immediately
        2. Run verification in background
        3. Update if corrections needed
        4. Notify user of changes
    2. Best for:
        1. User-facing applications
        2. When speed matters
        3. When corrections are rare

7. MONITORING & PERFORMANCE TRACKING

7.1 Comprehensive Monitoring

7.1.1 WHY do this?

1. Cannot optimize what you don't measure
2. Without monitoring you have no visibility into:
    1. Actual performance (fast or slow?)
    2. Bottlenecks (GPU, CPU, network?)
    3. Trends (improving or degrading?)
    4. Capacity limits (approaching maximum?)
3. Monitoring enables:
    1. Data-driven optimization decisions
    2. Proactive problem detection
    3. Capacity planning
    4. ROI demonstration
    5. Performance regression detection
4. Key questions monitoring answers:
    1. Are requests getting faster or slower over time?
    2. Is GPU being fully utilized?
    3. Where are bottlenecks in pipeline?
    4. When will we run out of capacity?
    5. Which optimizations are working?
5. Business benefits:
    1. Justify infrastructure investments
    2. Predict scaling needs
    3. Prevent downtime
    4. Optimize costs
6. Without monitoring:
    1. Flying blind
    2. Reactive problem solving
    3. Guessing at optimizations
    4. Can't prove improvements

7.1.2 WHAT to do:

1. Implement comprehensive logging of every request
2. Track key metrics:
    1. Latency: Time from request to response
    2. Throughput: Requests per second
    3. GPU utilization: Percentage compute used
    4. GPU memory: Amount used vs available
    5. Error rate: Failed requests percentage
    6. Token counts: Input and output tokens
3. Calculate aggregate statistics:
    1. Average latency
    2. Median latency (P50)

3. 95th percentile latency (P95)
4. 99th percentile latency (P99)
5. Standard deviation
4. Store data for trending and analysis
5. Set up alerts for anomalies
6. Create dashboards or reports

7.1.3 HOW to implement:

1. Design metrics data structure:
    1. Per-request metrics:
        1. Timestamp (when request started)
        2. Request ID (unique identifier)
        3. Model name (which model used)
        4. Prompt tokens (input size)
        5. Response tokens (output size)
        6. Latency in milliseconds
        7. GPU memory used
        8. GPU utilization percentage
        9. Success or failure flag
        10. Error message if failed
    2. Use dataclass or dictionary for storage
2. Implement metrics collection wrapper:
    1. Create function that wraps all LLM calls
    2. Before LLM call:
        1. Record start time
        2. Capture GPU stats
        3. Count prompt tokens
    3. After LLM call:
        1. Record end time
        2. Calculate latency
        3. Capture GPU stats again
        4. Count response tokens
        5. Check for errors
    4. Store all metrics
    5. Return result to caller
3. Implement GPU stats collection:
    1. Use nvidia-smi command
    2. Query specific metrics:
        1. memory.used: GPU memory in use
        2. utilization.gpu: GPU compute percentage
    3. Parse output:
        1. Extract numbers from CSV format
        2. Convert to appropriate types
    4. Cache for short period:
        1. Don't query every millisecond
        2. Query once per second

        3. Reuse value for requests in same second
4. Store metrics efficiently:
    1. Keep recent metrics in memory:
        1. List or deque of metric objects
        2. Limit to last 1000-10000 requests
        3. Prevents memory bloat
    2. Periodically write to disk:
        1. Every 100 requests
        2. Or every 5 minutes
        3. Append to log file or database
    3. Format options:
        1. JSON lines (one object per line)
        2. CSV for easy analysis
        3. Database for querying
5. Calculate aggregate statistics:
    1. Create function: get_summary_stats()
    2. Calculate from recent metrics:
        1. Total requests processed
        2. Average latency: sum / count
        3. Median latency: sort and take middle
        4. P95 latency: sort and take 95th percentile
        5. P99 latency: sort and take 99th percentile
        6. Average throughput: count / time_period
        7. Total tokens generated
        8. Error rate: errors / total
    3. Return as dictionary
    4. Cache for 1 minute to avoid recalculation
6. Implement percentile calculation:
    1. Collect all latency values
    2. Sort in ascending order
    3. For P95:
        1. Index = 0.95 × length
        2. Round to integer
        3. Return value at that index
    4. For P99:
        1. Index = 0.99 × length
        2. Round to integer
        3. Return value at that index
    5. Percentiles show worst-case performance
7. Create monitoring endpoint:
    1. Add API endpoint: GET /metrics
    2. Returns current statistics:
        1. Call get_summary_stats()
        2. Format as JSON
        3. Return to caller
    3. Accessible for dashboards
    4. No authentication needed (internal)

8. Implement time-series tracking:
    1. Track metrics over time windows:
        1. Last 1 minute
        2. Last 5 minutes
        3. Last 1 hour
        4. Last 24 hours
    2. Calculate stats for each window
    3. Identify trends:
        1. Is latency increasing?
        2. Is throughput decreasing?
        3. Is error rate rising?
9. Set up alerting thresholds:
    1. Define acceptable ranges:
        1. P95 latency < 10 seconds
        2. Error rate < 5%
        3. GPU memory < 90%
        4. Throughput > 10 req/min
    2. Check against thresholds:
        1. Every minute
        2. Or after every N requests
    3. Trigger alerts when exceeded:
        1. Log critical message
        2. Send notification (email, Slack, PagerDuty)
        3. Suggest remediation
10. Implement alert actions:
    1. High latency alert:
        1. Check GPU utilization
        2. Check if queue is backed up
        3. Suggest: Reduce parallel requests or upgrade instance
    2. High error rate alert:
        1. Check error types
        2. Check if Ollama is responsive
        3. Suggest: Restart Ollama or check logs
    3. High memory alert:
        1. Check loaded models
        2. Check parallel requests
        3. Suggest: Reduce NUM_PARALLEL setting
    4. Low throughput alert:
        1. Check if GPU underutilized
        2. Suggest: Increase parallel requests
11. Create simple dashboard:
    1. Text-based summary:
        1. Print stats to console
        2. Update every 10 seconds
        3. Show key metrics
    2. Web dashboard:
        1. Simple HTML page

        2. Auto-refresh via JavaScript
        3. Fetch from /metrics endpoint
        4. Display in tables and charts
    3. Minimal but functional
12. Track cost metrics:
    1. Calculate cost per request:
        1. Instance cost per hour
        2. Requests per hour
        3. Cost = hourly_cost / requests_per_hour
    2. Track total cost:
        1. Accumulate over time
        2. Project monthly cost
        3. Compare to budget
    3. Cost optimization opportunities:
        1. If cost per request high: Optimize speed
        2. If GPU underutilized: Increase throughput
13. Implement performance baselines:
    1. Record baseline metrics:
        1. After initial setup
        2. After major optimizations
        3. Record date and configuration
    2. Compare current to baseline:
        1. Calculate percentage change
        2. Identify improvements or regressions
    3. Maintain baseline history:
        1. Document each baseline
        2. Note what changed
        3. Track improvement trajectory
14. Track optimization impact:
    1. Before optimization:
        1. Record metrics for 24 hours
        2. Calculate baseline stats
    2. After optimization:
        1. Record metrics for 24 hours
        2. Calculate new stats
    3. Compare:
        1. Latency change: X% faster
        2. Throughput change: Y% higher
        3. Cost change: Z% lower
    4. Document and share results
15. Implement log rotation:
    1. Prevent log files from growing forever
    2. Rotation strategy:
        1. Keep last 7 days of detailed logs
        2. Keep last 30 days of summary stats
        3. Archive older data or delete
    3. Automatic rotation:

        1. Check file size daily
        2. Compress old logs
        3. Delete ancient logs
16. Handle monitoring failures gracefully:
    1. Monitoring should never break main application
    2. Wrap all monitoring in try-except
    3. If monitoring fails:
        1. Log the monitoring error
        2. Continue processing request
        3. Return result to user
    4. Monitoring is important but not critical path
17. Performance monitoring checklist:
    1. Request-level metrics: ✓
    2. GPU utilization tracking: ✓
    3. Aggregate statistics: ✓
    4. Time-series trending: ✓
    5. Alerting thresholds: ✓
    6. Dashboard or endpoint: ✓
    7. Cost tracking: ✓
    8. Baseline comparisons: ✓
    9. Log rotation: ✓
    10. Error handling: ✓

7.2 Automated Performance Testing

7.2.1 WHY do this?

1. Manual testing is inconsistent and unreliable
2. Problems with manual testing:
    1. Timing is imprecise
    2. Sample size too small
    3. Cannot detect subtle differences
    4. Human bias affects interpretation
    5. Time-consuming and tedious
3. Automated benchmarking provides:
    1. Precise timing measurements
    2. Statistical significance through repetition
    3. Objective comparisons
    4. Reproducible results
    5. Confidence in decisions
4. Critical questions automation answers:
    1. Is q4 quantization actually faster than q5?
    2. By how much? (with statistical confidence)
    3. What's the accuracy trade-off?
    4. Which configuration is optimal?
5. Without automation:

        1. Trial and error optimization
        2. Uncertain about improvements
        3. Cannot compare configurations objectively
        4. Risk of making wrong decisions
    6. Business value:
        1. Faster optimization cycles
        2. Confident infrastructure decisions
        3. Predictable performance
        4. Reduced experimentation costs

7.2.2 WHAT to do:

1. Create automated benchmark scripts
2. Test different configurations systematically
3. Run each configuration multiple times (10-20 iterations)
4. Collect timing and accuracy data
5. Calculate statistical measures:
    1. Mean (average)
    2. Median (middle value)
    3. P95 (95th percentile)
    4. Standard deviation (consistency)
6. Compare results across configurations
7. Identify optimal configuration for your workload

7.2.3 HOW to implement:

1. Design benchmark structure:
    1. Configuration to test
    2. Test prompt (representative of real workload)
    3. Number of iterations
    4. Metrics to collect
    5. Results storage
2. Create configuration matrix:
    1. Variables to test:
        1. Model: llama2:7b-q4_0, llama2:7b-q5_K_M, mistral:7b-q4_0
        2. Context size: 2048, 4096, 8192
        3. Temperature: 0.1, 0.5, 0.8
        4. Quantization level
    2. Create combinations to test:
        1. All models with 2048 context
        2. All context sizes with one model
        3. Different temperatures with one model
    3. Prioritize important combinations
3. Create test prompt:
    1. Use real example from your workload
    2. Not too short (not representative)
    3. Not too long (testing takes forever)

        4. Typical length: 200-500 words
        5. Example: "Analyze this invoice and extract..."
4. Implement benchmark function:
    1. Function: benchmark_configuration()
    2. Parameters:
        1. model: Model name
        2. num_ctx: Context size
        3. temperature: Sampling temperature
        4. test_prompt: Test input
        5. iterations: Number of runs (default 10)
    3. Process:
        1. Initialize results list
        2. For each iteration:
            1. Record start time
            2. Make API call with configuration
            3. Record end time
            4. Calculate latency
            5. Count output tokens
            6. Store results
        3. Calculate statistics
        4. Return summary
5. Execute single benchmark run:
    1. Prepare API request with configuration
    2. Use high-precision timer:
        1. time.time() in Python
        2. Microsecond precision
    3. Send request to Ollama
    4. Wait for complete response
    5. Calculate elapsed time
    6. Extract response data:
        1. Response text
        2. Token count if available
        3. Any error messages
6. Handle benchmark errors:
    1. Timeouts:
        1. Set reasonable timeout (120 seconds)
        2. Mark run as failed
        3. Continue with remaining iterations
    2. API errors:
        1. Catch exceptions
        2. Log error details
        3. Mark run as failed
    3. Report error rate:
        1. failed_runs / total_runs
        2. High error rate indicates problem
7. Calculate timing statistics:
    1. Collect all latencies from iterations

    2. Calculate mean:
        1. Sum all values
        2. Divide by count
        3. Most common measure
    3. Calculate median:
        1. Sort values
        2. Take middle value
        3. Robust to outliers
    4. Calculate P95:
        1. Sort values
        2. Take value at 95% position
        3. Represents worst-case typical performance
    5. Calculate standard deviation:
        1. Measure of consistency
        2. Low = very consistent
        3. High = highly variable
8. Calculate throughput metrics:
    1. Tokens per second:
        1. Output tokens / latency
        2. Higher is better
        3. Shows generation speed
    2. Requests per minute:
        1. 60 / average_latency
        2. Capacity metric
        3. Useful for planning
9. Store benchmark results:
    1. Create results dictionary:
        1. Configuration details
        2. Timing statistics
        3. Throughput metrics
        4. Error rate
        5. Timestamp
    2. Save to file:
        1. JSON format
        2. CSV for spreadsheets
        3. One file per benchmark run
    3. Enable comparison over time
10. Run comprehensive benchmark suite:
    1. Define all configurations to test
    2. For each configuration:
        1. Print configuration being tested
        2. Run benchmark_configuration()
        3. Display results summary
        4. Store results
        5. Brief pause between runs
    3. After all complete:
        1. Compile comparative report

2. Identify best configuration
11. Create comparison report:
    1. Tabular format:
        1. Columns: Model, Context, Temp, Avg Latency, P95, Throughput
        2. Rows: Each configuration tested
        3. Sort by key metric (e.g., latency)
    2. Highlight best options:
        1. Fastest configuration
        2. Most consistent
        3. Best throughput
        4. Best cost-effectiveness
    3. Add analysis:
        1. Trade-offs identified
        2. Recommendations
        3. Caveats or notes
12. Implement before-after testing:
    1. Baseline (before optimization):
        1. Run benchmark with current setup
        2. Record all metrics
        3. Save as baseline
    2. After optimization:
        1. Run same benchmark
        2. Record all metrics
        3. Save as optimized
    3. Calculate improvement:
        1. Latency: (baseline - optimized) / baseline × 100%
        2. Throughput: (optimized - baseline) / baseline × 100%
        3. Example: 30% faster, 50% higher throughput
13. Test accuracy alongside speed:
    1. For each configuration:
        1. Not just timing
        2. Also check output quality
    2. Compare outputs:
        1. Does it extract correct data?
        2. Is format consistent?
        3. Any hallucinations?
    3. Create accuracy score:
        1. Manual review of sample outputs
        2. Compare to known correct answers
        3. Percentage correct
    4. Optimize for accuracy AND speed
14. Implement automated scheduling:
    1. Run benchmarks regularly:
        1. Weekly full benchmark
        2. After any infrastructure change
        3. After Ollama updates
    2. Automated execution:

        1. Cron job or scheduler
        2. Runs during off-peak hours
        3. Emails results
    3. Detect performance regression:
        1. Compare to previous benchmark
        2. Alert if degradation > 10%
15. Create benchmark best practices:
    1. Warm-up runs:
        1. First few runs may be slower
        2. Model loading, cache warming
        3. Discard first 2-3 runs
        4. Use remaining for statistics
    2. Consistent environment:
        1. Run benchmarks when system idle
        2. No other processes running
        3. Same time of day if possible
    3. Sufficient iterations:
        1. Minimum 10 runs
        2. 20 runs for important decisions
        3. More runs = more confidence
16. Use results for optimization:
    1. Identify fastest configuration:
        1. Lowest average latency
        2. Acceptable accuracy
        3. Within budget
    2. Identify best value:
        1. Speed per dollar
        2. Balance of speed and quality
    3. Make data-driven decision:
        1. Not based on gut feeling
        2. Based on actual measurements
        3. Document reasoning
17. Document benchmark methodology:
    1. Test conditions:
        1. Hardware specifications
        2. Software versions
        3. Date and time
    2. Test parameters:
        1. Iterations count
        2. Test prompts used
        3. Configurations tested
    3. Results interpretation:
        1. What metrics mean
        2. How to compare
        3. Limitations and caveats

---

8. PRODUCTION-READY ARCHITECTURE

8.1 Complete Optimized System

8.1.1 WHY do this?

1. Individual optimizations help but aren't enough
2. Production systems need:
    1. Reliability: Handle errors gracefully
    2. Observability: Monitor what's happening
    3. Maintainability: Easy to update and debug
    4. Scalability: Handle growing load
    5. Consistency: Predictable performance
3. Benefits of integrated architecture:
    1. All optimizations work together
    2. Each component has clear responsibility
    3. Easy to troubleshoot problems
    4. Simple to add new features
    5. Multiplies benefits of individual optimizations
4. Problems with ad-hoc implementation:
    1. Code becomes tangled
    2. Hard to identify bottlenecks
    3. Difficult to maintain
    4. Optimizations interfere with each other
    5. Cannot scale cleanly
5. Well-designed architecture:
    1. Modular components
    2. Clear data flow
    3. Consistent error handling
    4. Comprehensive monitoring
    5. Easy to reason about
6. Investment pays off:
    1. Faster development of new features
    2. Easier optimization
    3. Better performance
    4. Higher reliability

8.1.2 WHAT to do:

1. Build integrated system combining all optimizations
2. Use API framework for request handling
3. Implement processing pipeline:
    1. Request validation
    2. Preprocessing
    3. Cache checking
    4. LLM processing
    5. Response formatting
4. Add comprehensive monitoring throughout
5. Handle errors gracefully at each stage

6. Expose metrics and health endpoints
7. Design for modularity and maintainability

8.1.3 HOW to implement:

1. Choose API framework:
    1. FastAPI (Python): Modern, async, good performance
    2. Flask (Python): Simple, widely used
    3. Express (Node.js): If using JavaScript
    4. Framework handles:
        1. HTTP request parsing
        2. Routing
        3. Response formatting
        4. Error handling basics
2. Design component architecture:
    1. API Layer:
        1. Receives requests
        2. Validates input
        3. Routes to appropriate handler
        4. Returns responses
    2. Preprocessing Module:
        1. Handles file uploads
        2. Extracts text
        3. Cleans content
        4. Optimizes for LLM
    3. Caching Layer:
        1. Checks Redis cache
        2. Returns cached results
        3. Stores new results
        4. Manages TTL
    4. LLM Service Module:
        1. Calls Ollama API
        2. Handles retries
        3. Manages timeouts
        4. Error handling
    5. Monitoring Component:
        1. Tracks all requests
        2. Calculates statistics
        3. Triggers alerts
        4. Exposes metrics
3. Implement request flow:
    1. Client sends request → API endpoint
    2. API validates request:
        1. Check required parameters
        2. Validate file format
        3. Check file size limits
        4. Return 400 error if invalid

3. Preprocessing:
    1. Save uploaded file
    2. Extract text
    3. Clean and normalize
    4. Calculate optimal context
4. Cache check:
    1. Generate cache key
    2. Query Redis
    3. If hit: Return cached result
    4. If miss: Continue to LLM
5. LLM processing:
    1. Build optimized prompt
    2. Call Ollama with configuration
    3. Handle streaming if enabled
    4. Collect response
6. Post-processing:
    1. Format response
    2. Store in cache
    3. Record metrics
    4. Return to client

4. Implement preprocessing module:
    1. Create DocumentPreprocessor class
    2. Methods:
        1. process(file_path): Main entry point
        2. extract_text(file_path): Type-specific extraction
        3. clean_text(text): Remove noise
        4. truncate(text): Limit length
    3. Configuration:
        1. Maximum length
        2. Supported file types
        3. Cleaning rules
    4. Error handling:
        1. Unsupported format
        2. Corrupted files
        3. Empty content
5. Implement caching module:
    1. Create CacheManager class
    2. Methods:
        1. get(key): Retrieve from cache
        2. set(key, value, ttl): Store in cache
        3. delete(key): Invalidate
        4. clear(): Clear all
    3. Connection management:
        1. Initialize Redis connection
        2. Handle connection failures
        3. Retry logic
    4. Metrics:

        1. Track hit rate
        2. Track miss rate
        3. Monitor performance
6. Implement LLM service module:
    1. Create LLMService class
    2. Configuration:
        1. Model name
        2. Default parameters
        3. Timeout settings
        4. Retry policy
    3. Methods:
        1. generate(prompt, options): Main inference
        2. _build_request(prompt, options): Prepare API call
        3. _handle_response(response): Parse result
        4. _handle_error(error): Error management
    4. Features:
        1. Automatic retries (3 attempts)
        2. Exponential backoff
        3. Timeout handling
        4. Streaming support
7. Implement monitoring module:
    1. Create PerformanceMonitor class
    2. Track per-request:
        1. Timestamp
        2. Latency
        3. Model used
        4. Token counts
        5. GPU stats
        6. Success/failure
    3. Calculate aggregates:
        1. Average metrics
        2. Percentiles
        3. Throughput
        4. Error rate
    4. Export functionality:
        1. Get current stats
        2. Export to file
        3. Reset counters
8. Wire components together:
    1. Initialize all components:
        1. Preprocessor
        2. Cache manager
        3. LLM service
        4. Performance monitor
    2. Create integrated processing function:
        1. Takes file and query
        2. Calls each component in sequence

        3. Handles errors at each stage
        4. Returns final result
    3. Ensure proper error propagation
9. Implement main API endpoint:
    1. POST /analyze-document
    2. Parameters:
        1. file: Uploaded document
        2. query: What to extract/analyze
        3. options: Optional configuration
    3. Process flow:
        1. Validate inputs
        2. Preprocess document
        3. Check cache
        4. Call LLM if needed
        5. Format response
        6. Record metrics
        7. Return result
    4. Response format:
        1. Success: 200 with result
        2. Error: 400/500 with message
10. Implement health check endpoint:
    1. GET /health
    2. Checks:
        1. Ollama is responsive
        2. Redis is accessible
        3. GPU is available
        4. Disk space available
    3. Returns:
        1. Status: healthy/unhealthy
        2. Components: Status of each
        3. Version info
    4. Used by load balancers
11. Implement metrics endpoint:
    1. GET /metrics
    2. Returns:
        1. Request statistics
        2. Performance metrics
        3. Cache hit rate
        4. GPU utilization
        5. Error rates
    3. Format:
        1. JSON for programmatic access
        2. Optional Prometheus format
    4. No authentication (internal)
12. Implement error handling strategy:
    1. Layer-specific handling:
        1. API layer: Validation errors → 400

    2. Preprocessing: File errors → 422
    3. Cache: Failures → Log, continue
    4. LLM: Errors → 503, retry logic
  2. Error response format:
    1. Consistent structure
    2. Error code
    3. Human-readable message
    4. Request ID for debugging
  3. Logging:
    1. All errors logged
    2. Include stack traces
    3. Include request context
13. Implement configuration management:
  1. Environment variables:
    1. OLLAMA_URL
    2. REDIS_URL
    3. MODEL_NAME
    4. CACHE_TTL
  2. Configuration file:
    1. Default values
    2. Model parameters
    3. Processing options
  3. Override hierarchy:
    1. Environment variables
    2. Config file
    3. Hardcoded defaults
14. Add request validation:
  1. File validation:
    1. Type checking
    2. Size limits (e.g., max 10MB)
    3. Format verification
  2. Parameter validation:
    1. Required fields present
    2. Valid data types
    3. Value ranges
  3. Return clear errors:
    1. "File too large (10MB limit)"
    2. "Query parameter required"
    3. "Unsupported file type"
15. Implement logging strategy:
  1. Structured logging:
    1. JSON format
    2. Consistent fields
    3. Correlation IDs
  2. Log levels:
    1. DEBUG: Detailed flow
    2. INFO: Key events

        3. WARNING: Potential issues
        4. ERROR: Failures
    3. Log rotation:
        1. Size-based (100MB files)
        2. Time-based (daily)
        3. Retention (7-30 days)
16. Add graceful shutdown:
    1. Signal handling:
        1. Catch SIGTERM/SIGINT
        2. Stop accepting new requests
        3. Finish processing current requests
        4. Close connections cleanly
        5. Exit gracefully
    2. Cleanup:
        1. Close Redis connection
        2. Flush logs
        3. Save metrics
17. Implement rate limiting:
    1. Prevent abuse:
        1. Limit requests per IP
        2. Limit requests per user
        3. Configurable thresholds
    2. Response:
        1. Return 429 when exceeded
        2. Include retry-after header
        3. Log rate limit hits
    3. Algorithm:
        1. Token bucket
        2. Or sliding window
        3. Store in Redis
18. Add request queuing:
    1. When system busy:
        1. Queue requests
        2. Process in order
        3. Prevent overwhelming GPU
    2. Queue management:
        1. Maximum queue size
        2. Reject when full
        3. Provide wait time estimates
    3. Monitoring:
        1. Track queue length
        2. Track wait times
        3. Alert if backing up
19. Testing the complete system:
    1. Unit tests:
        1. Each component independently
        2. Mock dependencies

        3. Test error cases
    2. Integration tests:
        1. Full request flow
        2. Real components
        3. End-to-end validation
    3. Load tests:
        1. Simulate many concurrent users
        2. Identify bottlenecks
        3. Verify stability
20. Deployment checklist:
    1. Configuration:
        1. All environment variables set
        2. Ollama running and configured
        3. Redis running
    2. Monitoring:
        1. Health check endpoint working
        2. Metrics endpoint working
        3. Logging configured
    3. Testing:
        1. Smoke tests pass
        2. Load tests satisfactory
    4. Documentation:
        1. API documentation
        2. Deployment guide
        3. Troubleshooting guide

---

9. QUICK WINS CHECKLIST

9.1 Immediate Actions (1-2 hours)

9.1.1 WHY prioritize these:

1. Maximum impact for minimum effort
2. Configuration changes, not code changes
3. Low risk, high reward
4. Can implement immediately
5. Combined effect: 40-60% speed improvement
6. No development required
7. Reversible if problems occur

9.1.2 WHAT to implement:

1. Switch to optimized quantized model
2. Configure Ollama memory settings
3. Reduce default context window
4. Enable streaming responses

9.1.3 HOW to execute:

1. Switch to optimized quantization:
    1. Download llama2:7b-q4_K_M (or equivalent for your model)
    2. Test with sample query
    3. If quality acceptable, make it default
    4. Update application code to use new model name
    5. Remove old unquantized model to save space
    6. Expected time: 15 minutes
    7. Expected improvement: 2-3x faster
2. Configure Ollama memory settings:
    1. Run: sudo systemctl edit ollama
    2. Add environment variables:
        1. OLLAMA_NUM_PARALLEL=2
        2. OLLAMA_MAX_LOADED_MODELS=1
    3. Save and exit
    4. Run: sudo systemctl daemon-reload
    5. Run: sudo systemctl restart ollama
    6. Verify with test query
    7. Expected time: 10 minutes
    8. Expected improvement: More stable, better utilization
3. Reduce context window:
    1. Find where you set num_ctx in code
    2. Change from 4096 or 8192 to 2048
    3. Test with typical documents
    4. Verify they still fit in context
    5. If some don't fit, use dynamic sizing
    6. Expected time: 5 minutes
    7. Expected improvement: 1.5x faster for most documents
4. Enable streaming:
    1. Find Ollama API calls in code
    2. Change "stream": false to "stream": true
    3. Update response handling to process stream
    4. Test end-to-end
    5. Expected time: 30 minutes (depending on implementation)
    6. Expected improvement: Perceived 3x faster (actual time same)
5. Verify improvements:
    1. Test with real workload
    2. Measure before and after timing
    3. Confirm 40-60% improvement
    4. Document changes made
    5. Expected time: 20 minutes
6. Total time investment: 1-2 hours
7. Total expected improvement: 40-60% faster responses

9.2 Medium-term Actions (1 day)

9.2.1 WHY these next:

1. Require development work but high value
2. One-time implementations with ongoing benefits
3. Each optimization pays dividends on every request
4. Combined effect: 2-3x faster, 30% more accurate
5. Worth the investment for production systems

9.2.2 WHAT to implement:

1. Redis caching layer
2. Document preprocessing pipeline
3. Few-shot prompt templates
4. Task-specific temperature configurations

9.2.3 HOW to execute over one day:

1. Morning: Implement Redis caching (3-4 hours):
    1. Hour 1: Install and configure Redis
        1. Install Redis server
        2. Configure memory limits
        3. Set eviction policy
        4. Test connection
    2. Hour 2: Implement cache wrapper
        1. Create cache key generation
        2. Add cache check before LLM calls
        3. Store results after LLM calls
        4. Set appropriate TTL
    3. Hour 3: Integration and testing
        1. Wire into existing code
        2. Test cache hits and misses
        3. Verify correctness
        4. Measure performance
    4. Expected improvement: 2x faster for cached queries
2. Late Morning: Build preprocessing pipeline (2-3 hours):
    1. Hour 1: Implement file handlers
        1. PDF text extraction
        2. Image OCR
        3. Text file reading
    2. Hour 2: Implement cleaning functions
        1. Whitespace normalization
        2. Text truncation
        3. Format standardization
    3. Hour 3: Integration
        1. Add to processing pipeline
        2. Test with various file types

        3. Verify output quality
    4. Expected improvement: 1.5x faster, better accuracy
3. Early Afternoon: Create few-shot templates (2 hours):
    1. Hour 1: Identify main use cases
        1. List all tasks (extraction, summary, etc.)
        2. For each, select 3 representative examples
        3. Format examples clearly
    2. Hour 2: Implement and test
        1. Create prompt templates with examples
        2. Test against baseline
        3. Measure accuracy improvement
        4. Update production prompts
    3. Expected improvement: 25-35% more accurate
4. Late Afternoon: Configure temperature settings (1-2 hours):
    1. Hour 1: Test different temperatures
        1. Run benchmarks with temp 0.1, 0.5, 0.8
        2. Measure consistency and quality
        3. Document optimal values per task
    2. Hour 2: Implement task-based config
        1. Create configuration presets
        2. Map tasks to presets
        3. Update code to use appropriate preset
        4. Test end-to-end
    3. Expected improvement: 15-20% better accuracy
5. End of day: Measure total improvement (1 hour):
    1. Run comprehensive benchmark
    2. Compare to morning baseline
    3. Expected overall: 2-3x faster, 30% more accurate
    4. Document all changes made
6. Total time investment: 8 hours (1 work day)
7. Total expected improvement: 2-3x faster, 30% more accurate