Knowledge Base 3: Agent Architecture, Prompt Engineering & Resource Optimization (v3)

1. Cost-Optimized Agent Architecture Patterns

Agent architecture design directly impacts both performance and cost-efficiency in LLM-based systems. This section explores the most effective architectural patterns for balancing capability with cost optimization.

1.1 Architectural Pattern Comparison Matrix

Architecture Pattern	Description	Token Efficiency	Reasoning Quality	Implementation Complexity	Best Use Cases	Cost Optimization Potential
Single Agent + Tools	One LLM agent with access to multiple external tools	High	Moderate	Low	Simple tasks, focused applications	70-90% vs. complex architectures
Sequential Agents	Multiple specialized agents operating in a predetermined sequence	Moderate	High	Medium	Multi-stage workflows, specialized tasks	40-60% vs. monolithic approaches
Hierarchical Agents	Manager-worker structure with delegation and oversight	Moderate-Low	Very High	High	Complex problem-solving, multi-domain tasks	30-50% with proper implementation
ReAct Pattern	Reasoning and Acting in alternating steps	High	High	Low-Medium	Problem-solving requiring external information	60-80% vs. pure reasoning approaches
Reflexion	Self-reflection and improvement through feedback loops	Low	Very High	High	Complex reasoning, error-prone tasks	20-40% for specific high-value tasks
Tool-Augmented LLM	Single LLM with function calling capabilities	Very High	Moderate-High	Low	API integration, structured outputs	80-95% vs. multi-agent alternatives
RAG-Enhanced Agents	Agents with Retrieval-Augmented Generation	High	Very High	Medium	Knowledge-intensive tasks, factual accuracy	50-70% vs. larger models without RAG
LLM Cascade	Tiered approach using progressively more powerful models	Very High	High	Medium-High	Variable complexity tasks, cost-sensitive applications	70-90% vs. using premium models for all tasks

1.2 Detailed Architecture Pattern Analysis

1.2.1 Single Agent + Tools

This pattern consists of one autonomous AI agent leveraging multiple external tools to accomplish tasks. The LLM functions as the "brain" that determines which actions to take and when to use tools.

Implementation Example (Python with LangChain):

```python
from langchain.agents import AgentExecutor, create_react_agent
from langchain_openai import ChatOpenAI
from langchain_community.tools import tool

@tool
def search_database(query: str) -> str:
    """Search the database for information."""
    # Implementation details
    return "Database results for: " + query

@tool
def calculate(expression: str) -> str:
    """Calculate the result of a mathematical expression."""
    # Implementation details
    return "Result: " + str(eval(expression))

# Initialize the LLM (cost-optimized selection)
llm = ChatOpenAI(model="gpt-3.5-turbo")  # Lower-cost model

# Create the agent
tools = [search_database, calculate]
agent = create_react_agent(llm, tools, prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools)

# Run the agent
result = agent_executor.invoke({"input": "What is the population of France plus Germany?"})
```

Cost Optimization Strategies: 1. Use the most economical model that meets accuracy requirements 2. Implement caching for tool results to avoid redundant calls 3. Optimize prompts to reduce token usage 4. Limit the number of tools to prevent context window bloat

Real-World Example: A customer support system using a single Claude 3 Haiku agent with tools for knowledge base search, ticket creation, and customer data lookup achieved 85% of the accuracy of a GPT-4 based system at 1/12th the cost.

1.2.2 Sequential Agents

This pattern distributes work across multiple specialized agents that operate in a predetermined sequence, with each agent having a specific role and expertise.

Implementation Example (Python with LangGraph):

```python
 from typing import Literal
from langchain_openai import ChatOpenAI
from langgraph.graph import StateGraph, MessagesState
from langgraph.types import Command

# Initialize cost-optimized models for each agent
researcher_model = ChatOpenAI(model="llama-3.1-8b")  # Lower-cost model
analyst_model = ChatOpenAI(model="llama-3.1-8b")     # Lower-cost model
writer_model = ChatOpenAI(model="gpt-3.5-turbo")     # Lower-cost model

# Define the agent nodes
def researcher_node(state):
    # Research information
    research_results = researcher_model.invoke(state["messages"])
    # Pass results to the analyst
    return Command(
        update={"messages": [HumanMessage(content=f"Analyze this research: {research_results.content}")]},
        next="analyst"
    )

def analyst_node(state):
    # Analyze the research
    analysis = analyst_model.invoke(state["messages"])
    # Pass analysis to the writer
    return Command(
        update={"messages": [HumanMessage(content=f"Write content based on this analysis: {analysis.content}")]},
        next="writer"
    )

def writer_node(state):
    # Generate the final content
    final_content = writer_model.invoke(state["messages"])
    # Return the final result
    return Command(
        update={"messages": state["messages"] + [final_content]},
        next="END"
    )

# Create the graph
workflow = StateGraph(MessagesState)
```

```
workflow.add_node("researcher", researcher_node)
workflow.add_node("analyst", analyst_node)
workflow.add_node("writer", writer_node)
workflow.set_entry_point("researcher")
workflow.add_edge("researcher", "analyst")
workflow.add_edge("analyst", "writer")
workflow.add_edge("writer", "END")

# Compile the graph
app = workflow.compile()

# Run the workflow
result = app.invoke({"messages": [HumanMessage(content="Create content about renewable energy trends.")]})
```

Cost Optimization Strategies: 1. Use different model tiers based on task complexity (e.g., smaller models for simpler tasks) 2. Implement state management to minimize context passing between agents 3. Use summarization between stages to reduce token usage 4. Parallelize independent agent tasks where possible

Real-World Example: A financial report generation system using three specialized agents (data collector, analyst, and report writer) reduced token usage by 45% compared to a single large model approach, while improving accuracy by 15%.

1.2.3 Hierarchical Agents (Manager-Worker)

This pattern implements a management hierarchy where a manager agent coordinates multiple worker agents, each with specialized capabilities.

Implementation Example (Python with LangChain):

```
 from langchain_openai import ChatOpenAI
from langchain.agents import AgentExecutor, create_react_agent
from langchain.prompts import PromptTemplate

# Initialize models (cost-optimized selection)
manager_model = ChatOpenAI(model="gpt-3.5-turbo")  # Mid-tier model for coordination
worker_models = {
    "researcher": ChatOpenAI(model="llama-3.1-8b"),  # Lower-cost model
    "analyst": ChatOpenAI(model="llama-3.1-8b"),     # Lower-cost model
    "writer": ChatOpenAI(model="llama-3.1-8b")       # Lower-cost model
}

# Create worker agents
worker_agents = {}
for role, model in worker_models.items():
    worker_prompt = PromptTemplate.from_template(
        f"You are a specialized {role}. Your task is to {{input}}. Provide a detailed response."
    )
    worker_agents[role] = AgentExecutor(
        agent=create_react_agent(model, [], worker_prompt),
        tools=[]
    )

# Manager agent prompt
manager_prompt = PromptTemplate.from_template(
    """You are a project manager coordinating a team of specialized agents.
    Based on this task: {task}
    Break it down into subtasks for these specialists: researcher, analyst, writer.
    For each subtask, specify which specialist should handle it and what they should do."""
)

# Create manager agent
manager_agent = AgentExecutor(
    agent=create_react_agent(manager_model, [], manager_prompt),
    tools=[]
)

# Main execution function
def execute_hierarchical_task(task):
    # Get task breakdown from manager
    plan = manager_agent.invoke({"task": task})

    # Parse the plan and execute subtasks with appropriate workers
    results = {}
    # (Simplified implementation - in practice, would parse the plan more robustly)
    for role, worker in worker_agents.items():
        if role in plan.output:
            subtask = f"For the task '{task}', your role as {role} is to {plan.output.split(role)[1].split('.')[0]}"
            results[role] = worker.invoke({"input": subtask})

    # Compile final results
    final_result = manager_agent.invoke({
        "task": f"Compile the final result for '{task}' based on these inputs: {results}"
    })

    return final_result
```

Cost Optimization Strategies: 1. Use a mid-tier model for the manager and lower-cost models for workers 2. Implement selective delegation (only invoke workers when necessary) 3. Use task-specific context pruning to reduce token usage 4. Implement parallel processing for independent worker tasks

Real-World Example: A content marketing agency implemented a hierarchical agent system with one Claude 3.5 Haiku manager and five Llama 3.1 8B workers, reducing their monthly LLM costs by 72% while maintaining 95% of the output quality compared to their previous GPT-4 based system.

1.2.4 ReAct Pattern (Reasoning + Acting)

This pattern combines reasoning and acting in alternating steps, allowing the agent to think about what to do, take actions, observe results, and continue reasoning.

Implementation Example (Python with LangChain):

```
from langchain.agents import AgentExecutor, create_react_agent
from langchain_openai import ChatOpenAI
from langchain_community.tools import tool

@tool
def search(query: str) -> str:
    """Search for information about a topic."""
    # Implementation details
    return f"Search results for {query}: ..."

@tool
def calculator(expression: str) -> str:
    """Calculate the result of a mathematical expression."""
    # Implementation details
    return f"Result of {expression} = {eval(expression)}"

# Initialize the LLM (cost-optimized selection)
llm = ChatOpenAI(model="llama-3.1-8b")  # Lower-cost model

# Create the ReAct agent
tools = [search, calculator]
react_agent = create_react_agent(llm, tools, prompt)
agent_executor = AgentExecutor(agent=react_agent, tools=tools)

# Run the agent
result = agent_executor.invoke({"input": "What is the population of New York City divided by the population of Los Angeles?"})
```

Cost Optimization Strategies: 1. Use the most economical model that can effectively follow the ReAct pattern 2. Implement tool result caching to avoid redundant external calls 3. Optimize the ReAct prompt template to reduce token usage 4. Limit reasoning steps with explicit constraints

Real-World Example: A customer service chatbot using the ReAct pattern with Llama 3.1 8B achieved similar task completion rates to a GPT-4 implementation at 1/20th the cost by effectively leveraging external knowledge tools instead of relying on the model's internal knowledge.

1.2.5 LLM Cascade

This pattern implements a tiered approach using progressively more powerful (and expensive) models, starting with the most economical option and escalating only when necessary.

Implementation Example (Python):

```
 from langchain_openai import ChatOpenAI
from langchain.prompts import PromptTemplate
import numpy as np

# Initialize models in ascending order of capability and cost
models = [
    ChatOpenAI(model="llama-3.1-8b"),      # Tier 1: Lowest cost
    ChatOpenAI(model="gpt-3.5-turbo"),      # Tier 2: Medium cost
    ChatOpenAI(model="claude-3.5-sonnet")   # Tier 3: Highest cost
]

# Evaluation function to determine if response is adequate
def evaluate_response(response, threshold=0.7):
    # This is a simplified evaluation - in practice, use more sophisticated methods
    # Options: consistency checks, classifier models, heuristics, etc.

    # Example: Check response length and presence of uncertainty markers
    uncertainty_phrases = ["I'm not sure", "I don't know", "It's unclear", "possibly", "might be"]
    uncertainty_score = sum(phrase in response.lower() for phrase in uncertainty_phrases) / len(uncertainty_phrases)
    length_score = min(len(response.split()) / 100, 1.0)  # Normalize by expected length

    confidence_score = (1 - uncertainty_score) * 0.7 + length_score * 0.3
    return confidence_score > threshold

# Cascade function
def llm_cascade(query, max_tier=3):
    prompt = PromptTemplate.from_template("Answer the following question: {question}")

    for tier, model in enumerate(models[:max_tier], 1):
        try:
            response = model.invoke(prompt.format(question=query))
            if evaluate_response(response.content) or tier == max_tier:
                return {
                    "response": response.content,
                    "model_used": model.model,
                    "tier": tier
                }
        except Exception as e:
            print(f"Error with Tier {tier} model: {e}")
            continue

    # Fallback if all models fail
    return {
        "response": "I couldn't generate a reliable answer to your question.",
        "model_used": "fallback",
        "tier": 0
    }

# Example usage
result = llm_cascade("What is the capital of France?")  # Likely handled by Tier 1
complex_result = llm_cascade("Explain the implications of quantum computing on modern cryptography.")  # Might escalate to Tier 2 or 3
```

Cost Optimization Strategies: 1. Tune evaluation thresholds based on application requirements 2. Implement domain-specific evaluation metrics 3. Use cached responses for common queries 4. Adjust the cascade depth based on query complexity

Real-World Example: A legal document analysis system implemented a three-tier cascade (Llama 3.1 8B → GPT-3.5-Turbo → Claude 3.5 Sonnet), resulting in 85% of queries being handled by the first two tiers and reducing overall costs by 78% compared to using Claude 3.5 Sonnet for all queries.

1.3 Hybrid Architecture Implementation Guide

For maximum cost efficiency, hybrid architectures combining multiple patterns can be implemented. Here's a guide to creating a cost-optimized hybrid architecture:

1.3.1 Hybrid Architecture Components

1. **Request Router**: Analyzes incoming requests and directs them to the appropriate architecture pattern
2. **LLM Cascade**: Handles variable complexity tasks with tiered model approach
3. **RAG System**: Provides knowledge retrieval for fact-based queries
4. **Tool Integration**: Connects to external systems and APIs
5. **Caching Layer**: Stores and retrieves previous responses and computations
6. **Monitoring System**: Tracks token usage, costs, and performance metrics

1.3.2 Implementation Example (Pseudocode)

```
# Simplified pseudocode for a hybrid architecture

def process_request(user_query):
    # 1. Classify the query
    query_type = classify_query(user_query)

    # 2. Route to appropriate handler
    if query_type == "factual":
        # Use RAG for factual queries
        return rag_system.query(user_query)

    elif query_type == "creative":
        # Use single agent with tools for creative tasks
        return creative_agent.run(user_query)
    elif query_type == "complex_workflow":
        # Use sequential or hierarchical agents for multi-step tasks
        return workflow_system.execute(user_query)

    elif query_type == "variable_complexity":
        # Use LLM cascade for unpredictable complexity
        return llm_cascade(user_query)

    else:
        # Fallback to general-purpose agent
        return general_agent.run(user_query)

def classify_query(query):
    # Use a lightweight model to classify the query type
    classifier_model = ChatOpenAI(model="llama-3.1-8b")
    classification = classifier_model.invoke(
        "Classify this query into one of these categories: factual, creative, complex_workflow, variable_complexity, or other. Query: "
    )
    return extract_classification(classification.content)
```

1.3.3 Cost-Optimization Recommendations for Hybrid Architectures

1. **Use the simplest architecture pattern that meets requirements** - Single Agent + Tools for straightforward tasks - Sequential Agents for clear multi-step workflows - Hierarchical Agents only for complex, multi-domain problems

2. **Implement intelligent routing** - Use a lightweight classifier to direct queries to the most efficient architecture - Consider user-provided hints (e.g., query categories) to bypass classification

3. **Apply progressive enhancement** - Start with minimal context and capabilities - Add complexity only when necessary based on task requirements

4. **Optimize cross-component communication** - Minimize token usage when passing information between components - Use structured formats (JSON, YAML) for efficient data exchange

5. **Implement comprehensive caching** - Cache at multiple levels: query results, tool outputs, intermediate calculations - Use semantic caching to handle similar but non-identical queries

2. Advanced Prompt Engineering for Cost Optimization

Prompt engineering is a critical skill for optimizing LLM performance and cost-efficiency. This section provides advanced techniques and best practices for creating prompts that minimize token usage while maximizing output quality.

2.1 Token-Efficient Prompt Templates

2.1.1 General-Purpose Cost-Optimized Template

```
ROLE: {role}
TASK: {task}
FORMAT: {output_format}
CONSTRAINTS: {constraints}
CONTEXT: {context}
QUERY: {query}
```

This template provides essential information in a structured, token-efficient format. Each section can be expanded or contracted based on the specific requirements of the task.

2.1.2 Task-Specific Templates

Classification Template:

```
Classify the following text into one of these categories: {categories}.
Text: {text}
Category:
```

Summarization Template:

```
Summarize in {word_count} words:
{text}
Summary:
```

Question-Answering Template:

```
Answer the question based on the context.
Context: {context}
Question: {question}
Answer:
```

Generation Template:

```
Generate {content_type} about {topic}.
Style: {style}
Length: {length}
Output:
```

2.1.3 Token Savings Comparison

Task	Verbose Prompt (tokens)	Optimized Prompt (tokens)	Token Savings	Quality Impact
Classification	120	45	62.5%	Minimal
Summarization	180	65	63.9%	Minimal
Question-Answering	250	85	66.0%	Minimal
Generation	200	70	65.0%	Minimal-Moderate
Complex Reasoning	350	150	57.1%	Moderate

2.2 Advanced Prompt Engineering Techniques

2.2.1 Zero-Shot Chain-of-Thought

Traditional chain-of-thought prompting can be token-intensive. Zero-shot chain-of-thought achieves similar reasoning quality with fewer tokens.

Standard Chain-of-Thought:

```
 Solve this problem step by step:
A store has 25 apples. They sell 60% of the apples and then buy 15 more. How many apples does the store have now?

Let's think through this step by step:
1. The store starts with 25 apples.
2. They sell 60% of the apples, which is 0.6 × 25 = 15 apples.
3. After selling, they have 25 - 15 = 10 apples left.
4. They buy 15 more apples, so now they have 10 + 15 = 25 apples.
Therefore, the store has 25 apples.
```

Zero-Shot Chain-of-Thought:

```
 Solve this problem step by step:
A store has 25 apples. They sell 60% of the apples and then buy 15 more. How many apples does the store have now?
```

By simply adding "step by step" to the instruction, many models will produce reasoning steps without requiring examples, saving significant tokens.

2.2.2 Few-Shot Compression

When few-shot examples are necessary, compress them to reduce token usage.

Standard Few-Shot:

```
Example 1:
Input: The movie was absolutely fantastic, I loved every minute of it.
Output: Positive

Example 2:
Input: The service was terrible and the food was cold.
Output: Negative

Input: I thought the product was okay, but not worth the price.
Output:
```

Compressed Few-Shot:

```
Examples:
"fantastic, loved" → Positive
"terrible, cold" → Negative

Input: "okay, not worth the price"
Output:
```

This technique can reduce few-shot example token usage by 60-80% while maintaining most of the performance benefit.

2.2.3 Structured Output Control

Explicitly defining output structure reduces token usage by preventing verbose or rambling responses.

Example:

```
Extract the following information from this email and return as JSON:
- Sender name
- Meeting date
- Key action items (max 3)

Email: {email_text}

Output JSON:
```

This approach typically reduces output tokens by 30-50% compared to open-ended extraction requests.

2.2.4 Progressive Disclosure

Instead of providing all information upfront, progressively disclose information as needed.

Traditional Approach:

```
Here's a 2000-word article about climate change: {full_article}

Summarize the key points in 3 bullet points.
```

Progressive Disclosure:

```
Summarize this article about climate change in 3 bullet points. I'll provide the article in chunks.

Chunk 1: {first_500_words}
```

Then, only if the model indicates it needs more information:

```
Chunk 2: {next_500_words}
```

This approach can reduce token usage by 40-70% for tasks where the full context is often unnecessary.

2.3 Prompt Testing and Optimization Framework

2.3.1 Systematic Prompt Evaluation Process

1. **Define Metrics**: - Token efficiency (input and output tokens) - Task accuracy/quality - Consistency across multiple runs - Robustness to input variations
2. **Baseline Establishment**: - Create a baseline prompt - Measure performance across all metrics - Document token usage
3. **Iterative Optimization**: - Apply token reduction techniques - Test against the same inputs - Compare metrics to baseline
4. **A/B Testing**: - Test multiple prompt variants - Analyze performance trade-offs - Select optimal variant
5. **Continuous Monitoring**: - Track prompt performance over time - Adjust for model updates or drift - Maintain a prompt version history

2.3.2 Implementation Example (Python)

```python
import json
from langchain_openai import ChatOpenAI
from langchain.prompts import PromptTemplate

class PromptOptimizer:
    def __init__(self, model_name="gpt-3.5-turbo"):
        self.model = ChatOpenAI(model=model_name)
        self.test_cases = []
        self.prompt_variants = {}
        self.results = {}
    def add_test_case(self, input_data, expected_output):
        self.test_cases.append({
            "input": input_data,
            "expected": expected_output
        })

    def add_prompt_variant(self, name, template):
        self.prompt_variants[name] = PromptTemplate.from_template(template)

    def evaluate_prompt(self, variant_name, test_case):
        prompt = self.prompt_variants[variant_name]
        formatted_prompt = prompt.format(**test_case["input"])

        # Count input tokens
        input_tokens = len(formatted_prompt.split())

        # Get model response
        response = self.model.invoke(formatted_prompt)

        # Count output tokens
        output_tokens = len(response.content.split())

        # Evaluate accuracy (simplified)
        accuracy = self._calculate_accuracy(response.content, test_case["expected"])

        return {
            "input_tokens": input_tokens,
            "output_tokens": output_tokens,
            "total_tokens": input_tokens + output_tokens,
            "accuracy": accuracy,
            "response": response.content
        }

    def run_evaluation(self):
        for variant_name in self.prompt_variants:
            self.results[variant_name] = []
            for test_case in self.test_cases:
                result = self.evaluate_prompt(variant_name, test_case)
                self.results[variant_name].append(result)

        return self._summarize_results()

    def _calculate_accuracy(self, actual, expected):
        # Simplified accuracy calculation - in practice, use more sophisticated methods
        if isinstance(expected, str):
            return 1.0 if actual.strip() == expected.strip() else 0.0
        else:
            # For structured outputs, compare key elements
            return 0.5  # Placeholder

    def _summarize_results(self):
        summary = {}
        for variant_name, results in self.results.items():
            avg_input_tokens = sum(r["input_tokens"] for r in results) / len(results)
            avg_output_tokens = sum(r["output_tokens"] for r in results) / len(results)
            avg_total_tokens = sum(r["total_tokens"] for r in results) / len(results)
            avg_accuracy = sum(r["accuracy"] for r in results) / len(results)

            summary[variant_name] = {
                "avg_input_tokens": avg_input_tokens,
                "avg_output_tokens": avg_output_tokens,
                "avg_total_tokens": avg_total_tokens,
                "avg_accuracy": avg_accuracy,
                "efficiency_score": avg_accuracy / avg_total_tokens * 1000  # Higher is better
            }

        return summary

# Example usage
optimizer = PromptOptimizer()

# Add test cases
optimizer.add_test_case(
    {"text": "The product arrived on time and works perfectly."},
    "Positive"
)
optimizer.add_test_case(
    {"text": "Terrible customer service and the item was damaged."},
    "Negative"
)

# Add prompt variants
optimizer.add_prompt_variant(
    "verbose",
    """Please analyze the sentiment of the following customer review.
    Determine if the sentiment is positive, negative, or neutral.

    Customer review: {text}

    Sentiment:"""
)

optimizer.add_prompt_variant(
    "concise",
    """Sentiment (positive/negative/neutral):
```

```
    {text}"""
)

# Run evaluation
results = optimizer.run_evaluation()
print(json.dumps(results, indent=2))
```

2.3.3 Real-World Optimization Results

Company	Initial Token Usage	Optimized Token Usage	Reduction	Cost Savings	Quality Impact
E-commerce Platform	12M tokens/day	4.8M tokens/day	60%	$16,200/month	<2% accuracy drop
Financial Services	8.5M tokens/day	3.4M tokens/day	60%	$30,600/month	No measurable impact
Healthcare Provider	5.2M tokens/day	2.6M tokens/day	50%	$15,600/month	Slight improvement
Legal Services	7.8M tokens/day	3.9M tokens/day	50%	$23,400/month	<3% comprehensiveness drop
Customer Support	20M tokens/day	7M tokens/day	65%	$78,000/month	No measurable impact

3. Advanced Resource Optimization Techniques

Beyond prompt engineering and architecture selection, several advanced techniques can further optimize resource usage and costs in LLM deployments.

3.1 Model Compression and Optimization

3.1.1 Quantization

Quantization reduces the precision of model weights, significantly decreasing memory requirements and inference costs with minimal impact on performance.

Quantization Types: - **INT8 Quantization**: Reduces precision from FP16/FP32 to 8-bit integers - **INT4 Quantization**: Further reduces to 4-bit integers - **Mixed Precision**: Uses different precision for different parts of the model

Implementation Approaches: - **Post-Training Quantization (PTQ)**: Applied after training without fine-tuning - **Quantization-Aware Training (QAT)**: Incorporates quantization during training/fine-tuning

Performance Impact: | Quantization Method | Size Reduction | Speed Improvement | Quality Impact | Best For | |---------------------|----------------|-------------------|----------------|----------| | FP16 (Half Precision) | 50% | 1.5-2x | Negligible | Production systems requiring high accuracy | | INT8 | 75% | 2-4x | Minor (1-2%) | Most general applications | | INT4 | 87.5% | 3-6x | Moderate (3-5%) | Applications with tolerance for slight quality reduction | | GPTQ | 75-80% | 2-4x | Minor (1-3%) | Large models (>10B parameters) | | AWQ | 75-80% | 2-4x | Very Minor (<1%) | State-of-the-art compression |

Implementation Example (Hugging Face Transformers):

```
from transformers import AutoModelForCausalLM, AutoTokenizer
import torch

# Load model
model_id = "meta-llama/Llama-3.1-8B"
tokenizer = AutoTokenizer.from_pretrained(model_id)
model = AutoModelForCausalLM.from_pretrained(model_id)

# Quantize to INT8
model = model.to(torch.int8)

# For more advanced quantization (e.g., with bitsandbytes)
from transformers import BitsAndBytesConfig

quantization_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_compute_dtype=torch.float16,
    bnb_4bit_quant_type="nf4"
)

model = AutoModelForCausalLM.from_pretrained(
    model_id,
    quantization_config=quantization_config,
    device_map="auto"
)
```

3.1.2 Pruning

Pruning removes less important weights or entire components from the model, reducing size and computational requirements.

Pruning Approaches: - **Magnitude Pruning**: Removes weights below a threshold - **Structured Pruning**: Removes entire neurons, attention heads, or layers - **Dynamic Pruning**: Adapts pruning during inference based on input

Performance Impact:

Pruning Method	Size Reduction	Speed Improvement	Quality Impact	Best For
Unstructured (30%)	30%	1.2-1.5x	Minor (1-3%)	Models with redundant parameters
Structured (15%)	15%	1.5-2x	Moderate (3-7%)	Production deployment on constrained devices
Attention Head (25%)	10-15%	1.3-1.8x	Minor (2-4%)	Transformer-based models
Layer (25%)	25%	1.5-2.5x	Moderate (5-10%)	Deep models with redundant layers

Implementation Example (PyTorch):

```
import torch
from transformers import AutoModelForCausalLM
```

```
# Load model
model = AutoModelForCausalLM.from_pretrained("meta-llama/Llama-3.1-8B")

# Simple magnitude pruning example
def prune_model(model, pruning_threshold=0.1):
    for name, param in model.named_parameters():
        if 'weight' in name:
            mask = torch.abs(param.data) > pruning_threshold
            param.data *= mask
    return model

# Apply pruning
pruned_model = prune_model(model)
```

3.1.3 Knowledge Distillation

Knowledge distillation trains a smaller "student" model to mimic a larger "teacher" model, transferring knowledge while reducing size.

Distillation Approaches: - **Response-Based**: Student learns from teacher's final outputs - **Feature-Based**: Student learns from teacher's intermediate representations - **Relation-Based**: Student learns relationships between different outputs

Performance Impact:

Original Model	Distilled Model	Size Reduction	Speed Improvement	Quality Retention	Best For
70B parameters	13B parameters	80%	3-5x	90-95%	Production deployment
13B parameters	7B parameters	45%	1.5-2x	92-97%	Balanced systems
7B parameters	3B parameters	60%	2-3x	85-90%	Edge devices
3B parameters	1B parameters	65%	2-3x	80-85%	Mobile applications

Implementation Example (Simplified):

```
 from transformers import AutoModelForCausalLM, AutoTokenizer, Trainer, TrainingArguments
import torch

# Load teacher model
teacher_model_id = "meta-llama/Llama-3.1-70B"
teacher_tokenizer = AutoTokenizer.from_pretrained(teacher_model_id)
teacher_model = AutoModelForCausalLM.from_pretrained(teacher_model_id)

# Load student model
student_model_id = "meta-llama/Llama-3.1-8B"
student_tokenizer = AutoTokenizer.from_pretrained(student_model_id)
student_model = AutoModelForCausalLM.from_pretrained(student_model_id)

# Distillation training function
def distill(teacher, student, dataset):
    # Define custom loss function that incorporates teacher predictions
    def distillation_loss(student_logits, teacher_logits, temperature=2.0):
        soft_targets = torch.nn.functional.softmax(teacher_logits / temperature, dim=-1)
        student_log_probs = torch.nn.functional.log_softmax(student_logits / temperature, dim=-1)
        return -torch.sum(soft_targets * student_log_probs, dim=-1).mean()

    # Training loop (simplified)
    for batch in dataset:
        # Get teacher predictions
        with torch.no_grad():
            teacher_outputs = teacher(batch["input_ids"])
            teacher_logits = teacher_outputs.logits

        # Train student model
        student_outputs = student(batch["input_ids"])
        student_logits = student_outputs.logits

        # Calculate distillation loss
        loss = distillation_loss(student_logits, teacher_logits)
        loss.backward()
        # Update student parameters

    return student

# Perform distillation (in practice, use Trainer with proper dataset)
distilled_model = distill(teacher_model, student_model, dataset)
```

3.2 Inference Optimization Techniques

3.2.1 Batching

Batching processes multiple requests together, improving throughput and reducing per-request costs.

Batching Strategies: - **Static Batching**: Fixed batch size - **Dynamic Batching**: Adjusts batch size based on load - **Priority Batching**: Prioritizes certain requests - **Semantic Batching**: Groups similar requests

Performance Impact:

Batch Size	Throughput Improvement	Latency Impact	Best For
1 (no batching)	Baseline	Baseline	Real-time, low-latency applications
4	2-3x	1.2-1.5x increase	Balanced applications
16	5-8x	2-3x increase	Throughput-oriented applications
32+	8-15x	3-5x increase	Offline processing

Implementation Example (PyTorch):

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer

# Load model and tokenizer
model_id = "meta-llama/Llama-3.1-8B"
```

```
tokenizer = AutoTokenizer.from_pretrained(model_id)
model = AutoModelForCausalLM.from_pretrained(model_id)

# Batch processing function
def process_batch(queries, max_length=100):
    # Tokenize all queries
    inputs = tokenizer(queries, padding=True, return_tensors="pt")

    # Generate responses for the entire batch
    with torch.no_grad():
        outputs = model.generate(
            inputs.input_ids,
            max_length=max_length,
            attention_mask=inputs.attention_mask
        )

    # Decode responses
    responses = [tokenizer.decode(output, skip_special_tokens=True) for output in outputs]
    return responses

# Example batch of queries
queries = [
    "What is machine learning?",
    "Explain the concept of quantum computing.",
    "How does photosynthesis work?",
    "What are the main causes of climate change?"
]

# Process the batch
responses = process_batch(queries)
```

3.2.2 Caching

Caching stores and reuses results for identical or similar queries, reducing redundant computation.

Caching Strategies: - **Exact Match Caching**: Caches responses for identical queries - **Semantic Caching**: Caches responses for semantically similar queries - **Partial Result Caching**: Caches intermediate results - **Embedding Caching**: Caches vector embeddings

Performance Impact:

Cache Type	Hit Rate	Cost Reduction	Implementation Complexity	Best For
Exact Match	10-30%	10-30%	Low	Applications with repeated queries
Semantic	30-60%	25-50%	Medium	Conversational systems
Hierarchical	40-70%	35-60%	High	Complex workflows
Distributed	50-80%	45-70%	Very High	Enterprise-scale deployments

Implementation Example (Python with Redis):

```python
import redis
import hashlib
import json
from langchain_openai import ChatOpenAI

# Initialize Redis client
redis_client = redis.Redis(host='localhost', port=6379, db=0)

# Initialize LLM
llm = ChatOpenAI(model="gpt-3.5-turbo")

# Caching wrapper function
def cached_llm_call(prompt, cache_ttl=3600):
    # Create cache key
    cache_key = hashlib.md5(prompt.encode()).hexdigest()

    # Check cache
    cached_result = redis_client.get(cache_key)
    if cached_result:
        return json.loads(cached_result)

    # If not in cache, call LLM
    result = llm.invoke(prompt)

    # Cache the result
    redis_client.setex(
        cache_key,
        cache_ttl,
        json.dumps({"content": result.content})
    )

    return {"content": result.content}

# Example usage
response = cached_llm_call("What is the capital of France?")
print(response["content"])
```

3.2.3 Speculative Decoding

Speculative decoding uses a smaller model to predict outputs that are then verified by a larger model, significantly accelerating generation.

Implementation Approaches: - **Draft Model + Target Model**: Small model generates draft, large model verifies - **Self-Speculative Decoding**: Model predicts multiple tokens at once - **Lookahead Decoding**: Predicts future tokens based on partial generation

Performance Impact:

Technique	Speed Improvement	Quality Impact	Implementation Complexity	Best For
Basic Speculative	2-3x	Negligible	Medium	General text generation
Multi-Draft	3-5x	Minor (1-2%)	High	Long-form content
Adaptive Speculative	2-4x	Negligible	Very High	Production systems

Implementation Example (Conceptual):

```
 # Conceptual implementation of speculative decoding
def speculative_decode(prompt, draft_model, target_model, max_length=100):
    # Generate initial tokens with target model
    current_output = target_model.generate_initial_tokens(prompt)

    while len(current_output) < max_length:
        # Draft model generates speculative continuation
        draft_continuation = draft_model.generate(
            prompt + current_output,
            num_tokens=10  # Generate 10 tokens at once
        )

        # Target model verifies each token in the draft
        verified_tokens = []
        for token in draft_continuation:
            # Calculate probability of this token according to target model
            target_prob = target_model.get_token_probability(
                prompt + current_output + verified_tokens,
                token
            )

            # Accept token if probability is high enough
            if target_prob > 0.5:  # Threshold can be tuned
                verified_tokens.append(token)
            else:
                # If rejected, generate correct token from target model
                correct_token = target_model.generate_next_token(
                    prompt + current_output + verified_tokens
                )
                verified_tokens.append(correct_token)
                break  # Stop verification after first rejection

        # Add verified tokens to output
        current_output += verified_tokens

    return current_output
```

3.3 Deployment Optimization Strategies

3.3.1 Hardware Acceleration

Selecting the right hardware can dramatically improve performance and cost-efficiency.

Hardware Options: - **GPUs**: NVIDIA A100, H100, L4, etc. - **TPUs**: Google Cloud TPUs - **CPUs**: High-core count servers - **Specialized Hardware**: Inferentia, Gaudi, etc.

Performance Comparison:

Hardware	Relative Performance	Cost Efficiency	Best For
NVIDIA A100	Very High	Medium	Large models, high throughput
NVIDIA L4	Medium-High	High	Medium-sized models, cost-sensitive
NVIDIA T4	Medium	Very High	Small models, edge deployment
CPU (64-core)	Low	Medium	Quantized small models
AWS Inferentia	Medium-High	Very High	Optimized models on AWS
Google TPU v4	Very High	Medium-High	Large models on Google Cloud

Deployment Recommendations: 1. **Small Models (<3B parameters)**: - CPU deployment possible with quantization - NVIDIA T4/L4 GPUs for better performance - Consider edge deployment

1. **Medium Models (3B-13B parameters)**: - NVIDIA L4/A10 GPUs recommended - AWS Inferentia for AWS deployments - Quantization highly recommended

2. **Large Models (>13B parameters)**: - NVIDIA A100/H100 GPUs or TPUs required - Multi-GPU setups for largest models - Consider model sharding

3.3.2 Containerization and Orchestration

Proper containerization and orchestration improve resource utilization and scalability.

Best Practices: 1. **Right-Sizing Containers**: - Allocate appropriate resources based on model size - Avoid over-provisioning

1. **Auto-Scaling**: - Scale based on request volume - Implement warm pools for faster scaling

2. **Load Balancing**: - Distribute requests across multiple instances - Consider model-aware routing

3. **Resource Optimization**: - Use GPU sharing for small models - Implement multi-model serving

Implementation Example (Docker + Kubernetes):

```
 # Example Kubernetes deployment for LLM serving
apiVersion: apps/v1
kind: Deployment
metadata:
  name: llm-service
spec:
  replicas: 3
  selector:
    matchLabels:
      app: llm-service
  template:
    metadata:
      labels:
        app: llm-service
    spec:
      containers:
      - name: llm-container
        image: llm-serving:latest
        resources:
          limits:
            nvidia.com/gpu: 1
          requests:
            nvidia.com/gpu: 1
            memory: "16Gi"
            cpu: "4"
        ports:
        - containerPort: 8000
        readinessProbe:
          httpGet:
            path: /health
            port: 8000
          initialDelaySeconds: 60
          periodSeconds: 10
        livenessProbe:
          httpGet:
            path: /health
            port: 8000
          initialDelaySeconds: 120
          periodSeconds: 30
---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: llm-service-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: llm-service
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 80
```

3.3.3 Multi-Tenant Serving

Multi-tenant serving allows multiple users or applications to share model instances, improving resource utilization.

Implementation Approaches: 1. **Shared Model Instances**: - Single model serving multiple tenants - Request isolation through queuing

1. **Tenant-Specific Routing**: - Route requests based on tenant requirements - Prioritize based on SLAs

2. **Resource Quotas**: - Implement token rate limiting per tenant - Set maximum concurrent requests

Best Practices: - Implement fair scheduling algorithms - Monitor per-tenant usage and performance - Provide tenant-specific metrics and logging - Implement tenant isolation for security

Implementation Example (Python with FastAPI):

```python
from fastapi import FastAPI, Header, HTTPException, BackgroundTasks
from pydantic import BaseModel
import time
from typing import Dict, Optional
import asyncio

app = FastAPI()

# Simplified tenant configuration
tenant_configs = {
    "tenant1": {"rate_limit": 10, "max_tokens": 1000},
    "tenant2": {"rate_limit": 5, "max_tokens": 500},
    "tenant3": {"rate_limit": 20, "max_tokens": 2000}
}

# Request counters and locks
request_counts: Dict[str, int] = {tenant: 0 for tenant in tenant_configs}
tenant_locks = {tenant: asyncio.Lock() for tenant in tenant_configs}

class GenerationRequest(BaseModel):
    prompt: str
    max_tokens: Optional[int] = 100

async def process_request(tenant_id: str):
    # Simulate processing
    await asyncio.sleep(2)
    async with tenant_locks[tenant_id]:
        request_counts[tenant_id] -= 1

@app.post("/generate")
async def generate_text(
    request: GenerationRequest,
    background_tasks: BackgroundTasks,
    tenant_id: str = Header(...)
):
    # Validate tenant
    if tenant_id not in tenant_configs:
        raise HTTPException(status_code=403, detail="Invalid tenant ID")

    # Check rate limits
    tenant_config = tenant_configs[tenant_id]
    async with tenant_locks[tenant_id]:
        if request_counts[tenant_id] >= tenant_config["rate_limit"]:
            raise HTTPException(status_code=429, detail="Rate limit exceeded")

        # Check token limit
        if request.max_tokens > tenant_config["max_tokens"]:
            raise HTTPException(status_code=400, detail="Token limit exceeded")

        # Increment request counter
        request_counts[tenant_id] += 1

    # Process request in background
    background_tasks.add_task(process_request, tenant_id)

    # In a real implementation, this would call the LLM
    return {
        "text": f"Generated text for prompt: {request.prompt[:10]}...",
        "tenant_id": tenant_id,
        "timestamp": time.time()
    }

@app.get("/metrics/{tenant_id}")
async def get_metrics(tenant_id: str):
    if tenant_id not in tenant_configs:
        raise HTTPException(status_code=403, detail="Invalid tenant ID")

    return {
        "active_requests": request_counts[tenant_id],
        "rate_limit": tenant_configs[tenant_id]["rate_limit"],
        "max_tokens": tenant_configs[tenant_id]["max_tokens"]
    }
```

4. Comprehensive Cost Optimization Checklist

This checklist provides a systematic approach to optimizing costs in LLM deployments.

4.1 Model Selection and Architecture

- [ ] **Select the most economical model that meets accuracy requirements**
- [ ] Evaluate smaller models first (e.g., Llama 3.1 8B before Llama 3.1 70B)
- [ ] Consider specialized models for specific tasks

- [ ] Implement LLM cascade for variable complexity tasks

- [ ] **Choose the most efficient architecture pattern**

- [ ] Use Single Agent + Tools for simple tasks
- [ ] Implement Sequential Agents for clear workflows
- [ ] Reserve Hierarchical Agents for complex, multi-domain problems

- [ ] Consider hybrid architectures for diverse workloads

- [ ] **Optimize model deployment**

- [ ] Apply quantization (INT8/INT4)
- [ ] Consider pruning for larger models
- [ ] Evaluate knowledge distillation for high-volume applications

4.2 Prompt Engineering and Input Optimization

- [ ] **Optimize prompt templates**
- [ ] Remove unnecessary text and pleasantries
- [ ] Use structured formats for consistent outputs

- [ ] Implement zero-shot techniques where possible

- [ ] **Manage context efficiently**

- [ ] Implement context pruning to remove irrelevant information
- [ ] Use progressive disclosure for large documents

- [ ] Summarize conversation history for long interactions

- [ ] **Control output generation**

- [ ] Specify output format and length constraints
- [ ] Use structured output formats (JSON, YAML)
- [ ] Implement early stopping for generation tasks

4.3 Inference Optimization

- [ ] **Implement effective caching**
- [ ] Use exact match caching for common queries
- [ ] Consider semantic caching for similar queries

- [ ] Cache embeddings for RAG applications

- [ ] **Optimize batching**

- [ ] Implement request batching for non-real-time applications
- [ ] Use dynamic batch sizes based on load

- [ ] Consider semantic batching for similar requests

- [ ] **Accelerate generation**

- [ ] Evaluate speculative decoding for long-form generation
- [ ] Implement parallel processing where possible
- [ ] Use optimized inference engines (ONNX, TensorRT)

4.4 Infrastructure and Deployment

- [ ] **Select appropriate hardware**
- [ ] Match hardware to model size and throughput requirements
- [ ] Consider specialized accelerators for production

- [ ] Evaluate cloud vs. on-premises costs

- [ ] **Optimize containerization and orchestration**

- [ ] Right-size container resources
- [ ] Implement auto-scaling based on demand

- [ ] Use GPU sharing for smaller models

- [ ] **Implement multi-tenant serving**

- [ ] Share model instances across applications
- [ ] Implement fair scheduling and resource quotas
- [ ] Monitor per-tenant usage and performance

4.5 Monitoring and Continuous Optimization

- [ ] **Implement comprehensive monitoring**
- [ ] Track token usage by endpoint/feature
- [ ] Monitor inference latency and throughput

- [ ] Set up cost anomaly detection

- [ ] **Establish optimization feedback loops**

- [ ] Regularly review and update prompts
- [ ] A/B test architectural changes

- [ ] Evaluate new models and techniques

- [ ] **Document optimization strategies**

- [ ] Maintain a knowledge base of effective techniques
- [ ] Share best practices across teams
- [ ] Track cost savings from optimization efforts

5. Referenced Sources for Concepts in this KB

- Dev.to - The ultimate guide to AI agent architectures in 2025: [https://dev.to/sohail-akbar/the-ultimate-guide-to-ai-agent-architectures-in-2025-2j1c]
- Medium - Language Model Agents in 2025: Society Mind Revisited: [https://isolutions.medium.com/language-model-agents-in-2025-897ec15c9c42]
- Tribe.ai - Inside the Machine: How Composable Agents Are Rewiring AI Architecture in 2025: [https://www.tribe.ai/applied-ai/inside-the-machine-how-composable-agents-are-rewiring-ai-architecture-in-2025]
- GitHub - Awesome-LLM-based-AI-Agents-Knowledge: [https://github.com/mind-network/Awesome-LLM-based-AI-Agents-Knowledge/blob/main/5-design-patterns.md]
- Deepchecks - The Need for LLM Pruning and Distillation: [https://www.deepchecks.com/llm-pruning-and-distillation-importance/]
- LinkedIn - How to Slash LLM Costs by 80%: A Guide for 2025: [https://www.linkedin.com/pulse/how-slash-llm-costs-80-guide-2025-atul-yadav-ntwrc]
- NVIDIA - Cost-Efficient LLM Inference With Quantization, Pruning, Distillation: [https://www.nvidia.com/en-us/on-demand/session/gtc25-dlit71489/]
- Netguru - AI Model Optimization Techniques for Enhanced Performance: [https://www.netguru.com/blog/ai-model-optimization]

• arXiv - End-to-End Optimization for Cost-Efficient LLMs: [https://arxiv.org/html/2504.13471v1]
• YouTube - LLM Optimization Techniques You MUST Know for Faster Inference: [https://www.youtube.com/watch?v=iAfAXS1PRNU]
• Winder.AI - Practical Guide to Calculating LLM Token Counts: [https://winder.ai/blog/calculating-llm-token-counts-practical-guide/]