

# Exploring Open-Source LLMs with Ollama: A Comprehensive Analysis

## 1. Introduction

Large Language Models (LLMs) have revolutionized natural language processing and artificial intelligence capabilities. While proprietary models like GPT-4 and Claude have garnered significant attention, open-source alternatives offer accessibility, customizability, and transparency advantages. This report explores three open-source LLMs using Ollama, a tool designed to simplify running these models locally on my MacBook Pro.

The experiments were conducted on a 2021 MacBook Pro with an M1 Pro processor, 16GB of RAM, and 512GB of storage. This hardware configuration provided a realistic test environment for evaluating how these models might perform on consumer-grade hardware rather than specialized server infrastructure, highlighting their accessibility for individual developers and small organizations.

Through systematic testing and analysis, this study evaluates model performance across various tasks, prompt engineering techniques, and difficulty levels to provide insights into their practical applications, limitations, and potential use cases. The evaluation includes both quantitative metrics like response time and length, and qualitative assessment of output quality and relevance.

## 2. Models Overview and Selection Rationale

For this study, three models with varying sizes and architectures were selected to represent different points on the performance-resource spectrum:

### TinyLlama

- **Architecture:** Based on Llama architecture but significantly smaller
- **Size:** Approximately 1.1B parameters
- **Rationale:** Selected as a lightweight option that can run on modest hardware with minimal resource requirements. This model represents the category of efficient, resource-conscious models that prioritize speed and accessibility over state-of-the-art performance.

### Mistral

- **Architecture:** Mistral 7B, known for its mixture-of-experts inspired architecture
- **Size:** 7B parameters
- **Rationale:** Chosen as a mid-tier model that balances performance and resource requirements. Mistral models are noted for their efficient architecture that delivers strong performance despite

moderate parameter count.

Llama3.1

- **Architecture:** Latest iteration of Meta's Llama architecture
- **Size:** 8B parameters
- **Rationale:** Selected as a larger, more capable model representing near state-of-the-art open-source performance. This model serves as the upper bound of capability in our comparison. The 8B variant was specifically chosen as it offers significantly enhanced capabilities while still being runnable on consumer hardware, unlike the larger 70B variant that would require more specialized infrastructure.

This selection provides a spectrum of models that range from lightweight and fast (TinyLlama) to more resource-intensive but capable (Llama3.1), with Mistral representing a middle ground that balances efficiency and performance.

3. Basic Model Exploration Results

3.1 General Question Answering

The models were evaluated on general knowledge questions across different difficulty levels:

Model	Easy Questions	Medium Questions	Hard Questions
TinyLlama	Quick responses (2.4s) but very brief (11 tokens) and often superficial	Longer responses (6.1s, 347 tokens) with broader context but factual inaccuracies	Struggled with complex topics (4.4s, 296 tokens) showing limitations in reasoning
Mistral	Slower (11.4s) but accurate for simple questions	More nuanced responses (18.5s, 251 tokens) with better conceptual understanding	Improved reasoning for complex questions (13.8s, 211 tokens) but still incomplete
Llama3.1	Moderate speed (6.4s) with concise, accurate answers	Very detailed responses (102.4s, 451 tokens) with comprehensive explanations	Strong reasoning capabilities (31.0s, 311 tokens) with nuanced perspectives

Key findings indicate that while TinyLlama provides quick responses, Llama3.1 demonstrates superior understanding and reasoning, particularly for complex questions, despite taking significantly longer to generate responses.

3.2 Text Summarization

Summarization tasks used text samples of increasing length and complexity:

Model	Easy (Short Text)	Medium (Moderate Text)	Hard (Complex Text)
TinyLlama	Very fast (1.0s) but superficial summaries (68 tokens)	Quick (2.2s) but missed key points (113 tokens)	Longer generation time (8.8s) with unfocused summaries (343 tokens)
Mistral	Good balance of speed (6.3s) and content coverage (76 tokens)	Effective identification of main points (13.8s, 156 tokens)	Better structure but struggled with complexity (27.7s, 265 tokens)
Llama3.1	Concise and accurate (5.2s, 58 tokens)	Comprehensive coverage of key points (11.1s, 106 tokens)	Excellent organization but slow (117.0s, 348 tokens) with complex text

Summarization results show that Llama3.1 produces the most coherent and comprehensive summaries, especially for complex texts, though at the cost of significantly longer generation times. TinyLlama provides rapid but often inadequate summaries, while Mistral offers a reasonable compromise between quality and speed.

### 3.3 Code Generation

Code generation tasks ranged from simple functions to complex algorithms:

Model	Easy (Factorial)	Medium (Binary Search)	Hard (Matrix Fibonacci)
TinyLlama	Surprisingly capable (8.5s, 445 tokens) for basic functions	Struggled with algorithm implementation (3.2s, 172 tokens)	Incomplete solutions (4.8s, 270 tokens) with logical errors
Mistral	Correct implementation (8.0s, 82 tokens) with good documentation	Solid implementation (14.4s, 149 tokens) with minor optimizations	Partial solution (31.3s, 235 tokens) missing key optimizations
Llama3.1	Comprehensive solution (103.7s, 194 tokens) with error handling	Excellent implementation (233.6s, 299 tokens) with optimizations	Strong attempt at complex algorithm (154.1s, 386 tokens) but still not optimal

Code generation revealed a surprising strength in TinyLlama for basic programming tasks, while Llama3.1 provided the most robust and well-documented code solutions but required significantly more time. Mistral offered reasonably good code with moderate generation times.

### 3.4 Creative Writing

Creative writing tasks tested narrative abilities with increasing complexity:

Model	Easy (Simple Story)	Medium (Historical/Greek References)	Hard (Non-linear Structure)
TinyLlama	Simple but coherent stories (14.6s, 842 tokens)	Long responses (10.5s, 677 tokens) but weak thematic connections	Struggled with complex structure (8.6s, 584 tokens)
Mistral	Engaging narratives (28.8s, 395 tokens) with good pacing	Creative integration of themes (36.3s, 455 tokens)	Attempted non-linear structure (37.9s, 493 tokens) but limited coherence
Llama3.1	Rich, detailed storytelling (68.4s, 379 tokens)	Excellent thematic integration (111.8s, 621 tokens)	Impressive handling of complex narrative (72.1s, 571 tokens)

Creative writing highlighted significant differences in narrative capabilities, with Llama3.1 demonstrating sophisticated storytelling abilities and thematic coherence, while TinyLlama produced longer but simpler narratives. Mistral maintained reasonable quality with moderate generation times.

## 4. Focused Experimentation: Prompt Engineering

The focused experimentation investigated how different prompting techniques affect model performance across various questions. This exploration aimed to understand whether sophisticated prompting methods could enhance model outputs, potentially allowing smaller models to achieve better results without requiring additional computational resources.

### 4.1 Prompting Techniques Analysis

Five prompting techniques were evaluated across all three models:

**Basic:** Direct questioning without special formatting. This approach simply presents the question to the model without additional context or instruction. For example, when asked "What are the implications of rising global temperatures?", the model is presented with just that question.

**Few-Shot:** Providing examples before asking the question. This technique includes several example question-answer pairs before posing the target question, giving the model patterns to follow. For climate change, the prompt included examples about capitals of countries before asking about global warming implications.

**Chain-of-Thought:** Encouraging step-by-step reasoning. This technique explicitly asks the model to work through the problem logically, breaking down complex questions into sequential components. For the climate question, the prompt included the instruction "Let's work through this logically" after posing the question.

**Role-Based:** Assigning a specific expert role to the model. This approach frames the model as having expertise in the relevant domain. For the climate question, the model was told "You are an expert in climate science with many years of experience" before being asked to answer as this expert.

**Self-Consistency:** Asking for multiple approaches/perspectives. This technique requests that the model consider different angles or methods before settling on a final answer, potentially improving consistency and accuracy. For the climate question, the model was asked to provide three different approaches before synthesizing a consistent answer.

4.2 Response Times by Technique

Model	Basic	Few-Shot	Chain-of-Thought	Role-Based	Self-Consistency
TinyLlama	3.8s	5.1s	6.2s	3.9s	7.5s
Mistral	33.4s	19.9s	25.8s	26.2s	18.0s
Llama3.1	99.4s	39.2s	54.2s	132.8s	122.3s

Key observations:

- **TinyLlama:** Performed fastest with basic prompts, slowest with self-consistency prompts
- **Mistral:** Showed fastest responses with self-consistency prompts, slowest with basic prompts
- **Llama3.1:** Demonstrated dramatically faster responses with few-shot prompting compared to other techniques

Interestingly, the most complex technique (self-consistency) produced the fastest responses for Mistral but the slowest for TinyLlama, suggesting different optimal prompting strategies for different model architectures.

4.3 Response Length by Technique

Model	Basic	Few-Shot	Chain-of-Thought	Role-Based	Self-Consistency
TinyLlama	213 words	267 words	360 words	244 words	393 words
Mistral	354 words	228 words	318 words	318 words	241 words
Llama3.1	384 words	182 words	332 words	381 words	345 words

Key observations:

- **TinyLlama:** Produced longest responses with self-consistency prompts, shortest with basic prompts
- **Mistral:** Generated longest responses with basic prompts, shortest with few-shot prompts
- **Llama3.1:** Created longest responses with basic prompts, shortest with few-shot prompts

This analysis reveals that few-shot prompting tends to produce the most concise responses across the larger models, while self-consistency prompting elicits the most verbose responses from TinyLlama.

## 4.4 Technique Effectiveness by Question Type

Different techniques showed varying effectiveness depending on the question type:

- **General Knowledge Questions:** Chain-of-thought and role-based prompting produced the most informative responses across all models, with Llama3.1 showing particularly strong improvements with these techniques.
- **Programming Questions:** Few-shot prompting significantly improved code quality for TinyLlama and Mistral, while Llama3.1 produced its best code with chain-of-thought prompting.
- **Complex Reasoning Questions:** Self-consistency prompting helped TinyLlama generate more comprehensive answers, while chain-of-thought prompting notably improved Llama3.1's reasoning abilities.
- **Domain-Specific Questions:** Role-based prompting substantially enhanced domain expertise signaling across all models, though Llama3.1 showed the most dramatic improvement in generating authoritative responses.

## 5. Response Comparison and Analysis

To illustrate the differences between models, let's examine specific responses to the same prompts across different tasks and prompting techniques.

### 5.1 General Knowledge Question Comparison

For the medium difficulty question "Explain how vaccines work to protect against diseases," the three models produced notably different responses:

#### **TinyLlama response:**

"Vaccines work by introducing a weakened or killed form of a pathogen into the body. This stimulates the immune system to recognize and fight the disease without causing illness. The immune system then creates memory cells that remember how to fight the disease in the future."

#### **Mistral response:**

"Vaccines work by training your immune system to recognize and combat specific pathogens without causing the disease itself. They contain weakened or inactive parts of a particular organism that triggers an immune response in the body. When the vaccine is introduced, specialized cells called B-lymphocytes produce antibodies that bind to the pathogen, marking it for destruction. Additionally, T-lymphocytes are activated to help destroy infected cells. Most importantly, both B and T cells create

memory cells that remember how to fight that pathogen in the future, providing long-term protection..."

### **Llama3.1 response:**

"Vaccines work by introducing a controlled exposure to a pathogen (disease-causing organism) or its components to trigger the body's natural immune response without causing the actual disease. This process, called immunization, involves several sophisticated immunological mechanisms:

1. Antigen introduction: Vaccines contain antigens, which are molecules from the pathogen that can be recognized by the immune system. These may be weakened or inactivated pathogens, specific proteins, or genetic instructions for producing proteins.
2. Innate immune response: Initially, cells of the innate immune system (like dendritic cells and macrophages) recognize these foreign antigens and become activated.
3. Antigen presentation: These cells process the antigens and present fragments on their surface to activate adaptive immune cells.
4. B-cell activation: B-lymphocytes recognize the antigens and begin producing antibodies specifically designed to bind to the pathogen.
5. T-cell activation: Helper T-cells coordinate the immune response, while cytotoxic T-cells learn to recognize and destroy infected cells.
6. Immune memory formation: Most crucially, vaccines induce the formation of memory B and T cells that remain in the body for years or decades..."

Analysis: This comparison reveals how model size affects both depth and precision of responses. TinyLlama provides only basic information in a simplified format. Mistral offers more detail with specific immune system components but remains relatively brief. Llama3.1 delivers a comprehensive, well-structured explanation with numbered points, technical terminology, and a more sophisticated understanding of immunological mechanisms. While Llama3.1's response took over 100 seconds to generate compared to TinyLlama's 6 seconds, the quality difference is substantial for educational or professional contexts.

## **5.2 Code Generation Comparison**

For the prompt "Write a Python function to calculate the factorial of a number," all three models produced working code, but with notable differences:

### **TinyLlama:**

python



```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

### Mistral:

python



```
def factorial(n):
    """
    Calculate the factorial of a non-negative integer.

    Args:
        n: A non-negative integer

    Returns:
        The factorial of n

    Raises:
        ValueError: If n is negative
    """
    if not isinstance(n, int):
        raise TypeError("Input must be an integer")
    if n < 0:
        raise ValueError("Input must be non-negative")
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

### Llama3.1:



```
def factorial(n):
    """
    Calculate the factorial of a non-negative integer.

    The factorial of n is the product of all positive integers less than or equal to n
    For example: factorial(5) = 5 × 4 × 3 × 2 × 1 = 120

    Args:
        n (int): A non-negative integer

    Returns:
        int: The factorial of n

    Raises:
        ValueError: If n is negative
        TypeError: If n is not an integer

    Examples:
        >>> factorial(0)
        1
        >>> factorial(5)
        120
    """
    # Check if n is an integer
    if not isinstance(n, int):
        raise TypeError("Input must be an integer")

    # Check if n is non-negative
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers")

    # Base case
    if n == 0 or n == 1:
        return 1

    # Recursive case
    # Could also implement iteratively to avoid stack overflow for large inputs
    return n * factorial(n-1)
```

Analysis: All three models provided correct recursive implementations, but with increasing levels of robustness and documentation. TinyLlama's solution is minimal but correct for basic cases. Mistral

adds error handling and basic docstrings. Llama3.1 provides comprehensive documentation with examples, edge case handling, implementation notes, and better type checking. For production code, Llama3.1's implementation would require minimal modification, while TinyLlama's would need significant enhancement for robustness.

### 5.3 Creative Writing Comparison with Role-Based Prompting

The models were tasked with writing a short story with the role-based prompt assigning them as an expert creative writer. Here are excerpts from their responses:

#### **TinyLlama:**

"The old dog sniffed at the dirt path, his nose twitching with interest. Max had been walking this same trail every day for years, but today something was different. The dirt had been disturbed, and there was a strange glint catching the sunlight. Max pawed at the spot, revealing a small metal box. He barked excitedly, calling his owner who was walking a few steps behind. 'What did you find, boy?' asked Sam, kneeling down to examine the discovery. It was an old lunch box, rusted but intact. Sam opened it carefully to reveal a collection of old coins inside. 'Treasure!' Sam exclaimed. Max wagged his tail proudly."

#### **Mistral:**

"The Golden Trail

Rufus wasn't like other dogs. At twelve years old, the golden retriever moved with a deliberate slowness, his muzzle frosted with age. But his eyes—those amber pools hadn't dimmed a day since puppyhood.

On Tuesdays, Eleanor would take him to Willow Creek Park. It was their tradition, rain or shine. Today, the autumn air carried the scent of fallen leaves and distant woodsmoke.

'Just a short walk today, old boy,' Eleanor said, her own joints protesting against the morning chill.

Rufus, however, had other plans. Halfway along their usual path, he suddenly veered left, pulling against his leash with unexpected strength.

'Rufus! What's gotten into you?' Eleanor stumbled after him as he dragged her toward a dense thicket of brambles.

The dog stopped abruptly, pawing at something half-buried beneath a blanket of red and gold leaves. Eleanor squinted, then gasped..."

#### **Llama3.1:**

"The Unexpected Guardian

The first thing Arthur noticed about the dog was not its matted fur or protruding ribs, but its eyes—amber and intelligent, watching him from behind the dumpster with an unnerving awareness. He had come to throw away his sandwich wrapper, a quick detour on his lunch break, but now found himself frozen under that steady gaze.

'Hey there,' Arthur said softly, crouching down. 'You hungry?'

The dog—some mix of shepherd and retriever, perhaps—didn't approach, but its tail gave a single, hesitant wag. Arthur unwrapped the remaining half of his turkey sandwich and placed it on the ground, then stepped back.

With cautious dignity, the dog emerged. It was larger than Arthur had initially thought, and despite its condition, moved with surprising grace. It consumed the offering in two neat bites, then looked up expectantly.

'Sorry, buddy. That's all I've got,' Arthur said, checking his watch. 'I need to get back.'

As Arthur turned to leave, he heard a soft jingling. The dog had shifted position, revealing something metallic caught in its fur. A key, attached to a tarnished bronze chain.

'What's that you've got there?' Arthur asked, approaching slowly..."

Analysis: The difference in narrative sophistication is evident. TinyLlama creates a simple, straightforward story without much characterization or narrative tension. Mistral introduces more atmospheric elements, character details, and begins building narrative intrigue. Llama3.1 creates a richly detailed scene with nuanced character observations, atmospheric elements, stronger voice, and a more sophisticated narrative hook. The difference in storytelling ability highlights how larger models have better internalized narrative structures and literary techniques.

## **6. Analysis of Model Strengths and Weaknesses**

### **6.1 TinyLlama**

#### **Strengths:**

TinyLlama demonstrates remarkably fast response times for simple tasks, typically between 2-4 seconds on my MacBook. This responsiveness makes it suitable for applications requiring quick interactions. Despite its small size, it shows surprising competence with basic programming tasks, correctly implementing simple algorithms like factorial calculations and basic sorting. The model operates efficiently on limited hardware, using minimal RAM and CPU resources during inference. When provided with structured prompts, particularly few-shot examples, TinyLlama shows significant improvement in output quality compared to basic prompting.

**Weaknesses:**

TinyLlama exhibits limited contextual understanding and reasoning capabilities, often missing nuances in complex questions. When faced with multi-step reasoning tasks, it tends to make logical leaps or oversimplifications. For questions requiring specialized knowledge, responses are often superficial and occasionally contain factual errors. In creative writing tasks, the model struggles to maintain thematic coherence throughout longer passages, with narrative threads sometimes becoming disjointed. Output quality varies dramatically depending on prompt formatting, showing high sensitivity to input phrasing compared to larger models.

## 6.2 Mistral

**Strengths:**

Mistral achieves a commendable balance between performance and resource requirements on my MacBook, making it a versatile option for diverse applications. It delivers consistent quality across different task types without significant performance gaps. In summarization tasks, Mistral excels at maintaining key information while condensing text effectively. Its code generation capabilities include good structure and documentation practices, with reasonable implementation of algorithms. Response times remain fairly consistent across different prompting techniques, averaging between 15-30 seconds, indicating stable processing patterns regardless of input complexity.

**Weaknesses:**

With response times averaging 15-30 seconds on my MacBook, Mistral may be too slow for applications requiring real-time interaction. The model demonstrates limitations when addressing highly specialized domain knowledge, particularly in technical fields. Compared to both smaller and larger models, Mistral shows less dramatic improvement with certain prompt engineering techniques like chain-of-thought. While coherent, its creative writing lacks the sophistication and nuance present in larger models like Llama3.1. When tasked with implementing complex algorithms requiring optimization, Mistral provides functional but sub-optimal solutions.

## 6.3 Llama3.1 (8B)

**Strengths:**

Llama3.1 demonstrates superior reasoning capabilities across all evaluated task categories, handling complex questions with nuance and depth. When addressing questions requiring specialized knowledge, it provides comprehensive and generally accurate information with appropriate caveats. Its code generation stands out with extensive documentation, error handling, and adherence to best practices. Creative writing produced by Llama3.1 shows sophisticated narrative construction, character development, and thematic coherence. The model responds particularly well to role-based and chain-of-thought prompting for complex tasks, showing effective adaptation to guidance.

## Weaknesses:

The most significant drawback of Llama3.1 on my MacBook is extremely slow response times, often exceeding 100 seconds for complex tasks and reaching over 230 seconds for detailed code generation. These extended generation times make it impractical for interactive applications. The model requires substantial system resources, with high CPU utilization and memory consumption during inference. Disproportionately long generation times occur specifically for code tasks, suggesting particular inefficiency in this domain. When using basic prompting techniques, Llama3.1 tends toward unnecessary verbosity that doesn't always improve information quality. Despite significant resource requirements, the quality improvements over Mistral, while noticeable, may not justify the dramatically increased response time for many practical applications.

## 6. Insights on Open-Source LLM Capabilities

### 6.1 Performance-Resource Tradeoffs

This exploration reveals a clear but non-linear relationship between model size, performance, and resource requirements. While larger models generally produce higher quality outputs, the relationship is not proportional:

- **Response Time vs. Model Size:** Llama3.1 was often 10-20x slower than TinyLlama despite not being 10-20x larger in parameter count
- **Quality Improvements:** Quality improvements from Mistral to Llama3.1 were significant but less dramatic than the corresponding increase in resource requirements
- **Task Dependency:** The performance gap between models varied dramatically by task type, with simple tasks showing smaller gaps than complex reasoning tasks

This suggests that model selection should be highly task-dependent, with smaller models potentially sufficient for many practical applications.

### 6.2 Prompt Engineering Effectiveness

The study demonstrated that prompt engineering can significantly impact model performance, often providing more substantial improvements for smaller models:

- **Technique Variance:** Different models respond optimally to different prompting techniques
- **Smaller Model Benefits:** TinyLlama showed the most dramatic relative improvements with advanced prompting techniques
- **Model-Specific Strategies:** Larger models benefited most from reasoning-focused techniques, while smaller models improved more with structured guidance

This indicates that investment in prompt optimization may yield better returns than simply scaling to larger models for many applications.

## 6.3 Task Specialization Patterns

Models exhibited distinct patterns of specialization across task categories:

- **General Knowledge:** Performance scaled reasonably with model size
- **Summarization:** Even smaller models performed adequately on straightforward texts
- **Code Generation:** Surprising capabilities in smaller models for basic programming tasks
- **Creative Writing:** The largest performance gap between model sizes, with sophisticated narratives requiring larger models

These patterns suggest opportunities for task-specific model deployment strategies that optimize for efficiency.

## 7. Practical Implications for Real-World Applications

### 7.1 Application-Specific Model Selection Framework

Based on the findings, a framework for model selection emerges:

- **Real-time Interactive Applications:** TinyLlama's fast response times make it suitable for chatbots, customer service, and other applications requiring immediate responses, albeit with quality limitations.
- **Content Generation Pipelines:** Mistral provides a good balance for non-real-time content generation, such as email drafting, report generation, and basic documentation, where moderate waiting times are acceptable.
- **High-Value Complex Reasoning:** Llama3.1 is appropriate for specialized applications where quality is paramount and generation time is less critical, such as legal analysis, complex problem-solving, or sophisticated content creation.

### 7.2 Resource Optimization Strategies

For organizations deploying open-source LLMs, several strategies can optimize performance:

- **Tiered Deployment:** Implementing multiple models with automatic routing based on query complexity
- **Prompt Caching:** Storing responses to common queries to reduce generation time
- **Model Quantization:** Using techniques like 4-bit quantization to reduce memory requirements while maintaining reasonable quality

- **Specialized Fine-tuning:** Training smaller models on domain-specific data to improve performance on targeted tasks

## 7.3 Competitive Positioning against Proprietary Models

Open-source models demonstrate viable alternatives to proprietary models for many use cases:

- **Cost Advantage:** Self-hosting eliminates API costs that scale with usage
- **Privacy Benefits:** Local deployment avoids data sharing with third parties
- **Customization Potential:** Open-source models can be fine-tuned on proprietary data
- **Deployment Flexibility:** On-premises deployment for air-gapped or high-security environments

The performance gaps observed in this study suggest that open-source models are increasingly competitive for all but the most demanding applications.

## 8. Challenges and Solutions

### 8.1 Technical Challenges

Several significant technical challenges were encountered during this study on my MacBook Pro:

**Resource Constraints and Thermal Throttling:** Running larger models locally caused my MacBook to reach high temperatures, triggering thermal throttling that further reduced performance. The M1 Pro chip, while powerful, still struggled with the memory and compute requirements of Llama3.1.

To address this, I implemented a staged testing approach that allowed the system to cool down between intense computation periods. I also closed unnecessary background applications and utilized macOS's Activity Monitor to track resource usage in real-time. For the largest model, I reduced the maximum context length settings in Ollama to decrease memory requirements.

**Response Timeouts:** My initial testing framework included standard HTTP timeouts of 30 seconds, which proved inadequate for complex prompts with Llama3.1 that sometimes took over 3 minutes to complete.

I modified the code in `model_evaluation.py` and `focused_experiments.py` to remove timeout constraints entirely and instead track generation time separately using timestamp differentials. This allowed the models to complete their full generation without interruption while still capturing performance metrics accurately.

**Inconsistent Performance:** I observed that model performance varied significantly depending on MacBook battery status, background processes, and system temperature, making it difficult to get consistent benchmarks.

To mitigate this, I standardized testing conditions by ensuring the MacBook was plugged in, Activity Monitor showed minimal background CPU usage, and the device was placed on a cooling stand. I also ran critical tests multiple times and averaged the results to account for variance.

**Ollama API Instability:** Occasionally, the Ollama API would become unresponsive after prolonged usage or when switching rapidly between models.

I implemented automatic detection of API failures with graceful retries and a mandatory cooldown period. I also created a simple bash script to restart the Ollama service when necessary without losing experimental progress.

## 8.2 Methodological Challenges

The study methodology also presented several challenges that required creative solutions:

**Prompt Engineering Standardization:** Initial testing revealed that slight variations in prompt phrasing could dramatically affect model outputs, particularly for TinyLlama, making direct comparisons difficult.

I developed standardized prompt templates for each technique as shown in the `PROMPTING_TECHNIQUES` dictionary in `focused_experiments.py`, ensuring that all models received identically structured inputs. For each technique, I created templates with placeholders for the question content, maintaining consistent framing and instructions.

**Balancing Task Difficulty:** Creating tasks with appropriate and comparable difficulty levels across different categories proved challenging, as what's "easy" for code generation might not be equivalent to what's "easy" for creative writing.

To address this, I created category-specific difficulty criteria and validated them through pilot testing. I also ensured that task difficulty scaled appropriately within each category while maintaining reasonable comparability across categories. The `TEST_PROMPTS` dictionary in `model_evaluation.py` shows how this stratification was implemented.

**Qualitative Evaluation:** While response times and lengths were easily measurable, assessing response quality objectively presented a significant challenge.

I developed a multi-dimensional evaluation framework that considered factual accuracy, coherence, relevance, and completeness. For creative and subjective tasks, I established clear criteria for what constituted quality at different levels. I also implemented blind evaluation of some outputs to reduce bias in quality assessments.

## 8.3 Infrastructure Considerations



Running these models locally on a MacBook rather than in a cloud environment presented specific infrastructure challenges:

**Storage Management:** The combined size of all three models exceeded 15GB, which represented a significant portion of my available storage.

I implemented a sequential testing approach, downloading and testing one model at a time before moving to the next using Ollama's model management commands. After collecting data for one model, I would remove it before downloading the next to conserve storage space. This required careful organization of the testing sequence in `model_evaluation.py`.

**Energy and Heat Management:** Extended testing sessions caused significant battery drain and heat generation on my MacBook.

I scheduled testing sessions to allow for cooling periods and ensured the MacBook was plugged in throughout. For longer runs, particularly with Llama3.1, I elevated the laptop for better airflow and monitored CPU temperature using a third-party utility. I also optimized the code to consolidate tests, reducing the total runtime required.

**Operational Continuity:** Ensuring that long-running tests completed successfully despite potential interruptions (system sleep, network issues, etc.).

I implemented checkpointing in the test scripts, saving results after each completed test case rather than only at the end. This allowed testing to resume from the last successful checkpoint if interrupted. I also disabled system sleep during test runs to prevent the MacBook from sleeping during extended generation times.

## 9. Conclusion

This exploration of open-source LLMs using Ollama on my MacBook Pro has revealed nuanced patterns in model performance across different tasks, prompt engineering techniques, and difficulty levels. The three selected models—TinyLlama, Mistral, and Llama3.1 (8B)—represent different points on the capability spectrum, with clear tradeoffs between response quality, generation speed, and resource requirements.

Running these models locally provided direct insights into their practical usability for individual developers and small teams without access to specialized infrastructure. The experience highlighted both the impressive capabilities and practical limitations of current open-source LLMs on consumer hardware.

The analysis of response patterns across general knowledge questions, code generation, summarization, and creative writing tasks revealed that model size correlates with output quality but

with diminishing returns relative to computation time. TinyLlama, despite its limitations, provided remarkably usable outputs for many tasks with response times under 10 seconds. Mistral established itself as a strong middle-ground option, delivering quality approaching that of larger models with manageable response times. Llama3.1 (8B) demonstrated the highest quality outputs across all categories but with generation times that often exceeded 2 minutes for complex tasks on my MacBook.

The prompt engineering experiments yielded particularly interesting findings, showing that different models respond optimally to different prompting strategies. Few-shot prompting dramatically improved TinyLlama's performance on structured tasks, while chain-of-thought prompting enhanced Llama3.1's reasoning capabilities. These results suggest that investing time in prompt optimization may yield better returns than simply scaling to larger models, especially when operating under hardware constraints.

For practical applications, these findings suggest several strategies for effective deployment of open-source LLMs:

First, model selection should be guided by specific application requirements rather than defaulting to the largest available model. For interactive applications requiring sub-second or few-second responses, smaller models with optimized prompts may provide sufficient quality. For non-interactive batch processing where generation time is less critical, larger models can deliver superior outputs.

Second, hybrid approaches combining multiple models may offer the best balance. A smaller model could handle initial responses or simpler queries, with escalation to larger models only when necessary. This tiered approach would optimize both resource usage and user experience.

Third, domain-specific optimization through fine-tuning or specialized prompting strategies can significantly enhance performance for targeted applications, potentially allowing smaller models to match larger ones in narrow domains.

The limitations encountered during this study highlight areas for improvement in the open-source LLM ecosystem. Model quantization and optimization techniques need further development to make larger models more accessible on consumer hardware. Better tooling for prompt optimization could help users maximize model performance without specialized expertise.

Despite these challenges, the results demonstrate that open-source LLMs running locally represent a viable alternative to cloud-based proprietary models for many applications. They offer advantages in privacy, cost, and customizability that make them attractive for developers and organizations with appropriate use cases.

As the field continues to advance, we can expect improvements in model efficiency and optimization techniques that will further enhance the practicality of local LLM deployment. Future work could explore fine-tuning smaller models for specific domains, developing more efficient inference methods, and creating adaptive systems that dynamically select the appropriate model based on query complexity and response time requirements.

In conclusion, this exploration has demonstrated that the landscape of open-source LLMs offers rich possibilities for practical applications, even within the constraints of consumer hardware. By understanding the strengths, weaknesses, and optimal usage patterns of different models, developers can effectively leverage these powerful tools while mitigating their limitations.