Benchmarking LLMs and Vision Models on Jetson Orin AGX 64GB

Introduction

This benchmarking study was conducted to evaluate the performance of large language models (LLMs) and multimodal vision-language models on an NVIDIA Jetson AGX Orin 64GB platform. We used the ollama runtime and API to manage model execution and inference, including the new vision capabilities introduced in models like LLaVA, Gemma3, Llama3.2-Vision, and Moondream. The results are on pages 3-15.

Hardware Setup

- **Device**: NVIDIA Jetson AGX Orin
- **Carrier Board**: SeeedStudio J501
- **Memory**: 64GB RAM (unified)
- **OS**: Ubuntu with Jetpack 6.0+b106
- **CUDA**: 12.2
- **Kernel**: 5.15.136-tegra (PREEMPT)

Runtime Environment

- Models were executed via the ollama Python API or HTTP interface.
- Power mode 3 (50W) was used.
- Swap memory was disabled.
- Benchmarking scripts recorded:
    - CPU and RAM usage (via `psutil`)
    - Swap and system load
    - Power usage
    - Token throughput (tokens/sec)
    - Inference duration
    - Success/failure of outputs

Script Behavior

- Prompts are run serially and/or in parallel depending on benchmark goal.
- GPU stats (via `tegrastats`) were optionally collected.
- Model memory handling was optimized by limiting concurrent model loads and removing unused models between runs.

The Power usage was measured using the *jetson-stats/jtop* library. The power measured for the entire Jetson board is the reported value for *Avg Power Usage (W).*

Prompt Design

Text Benchmarks

Each model was prompted with five diverse, complex tasks to simulate real-world usage:

1. **Programming**: Step-by-step PyTorch CNN implementation.
2. **Economics**: Impact of inflation on low-income households.
3. **Code Comprehension**: Parse JSON and extract unique keys.
4. **Science Explanation**: Why seasons change, for a child.
5. **Creative Dialogue**: Sherlock Holmes converses with Alan Turing.

The exact prompts used can be found in Appendix B.

Vision Benchmarks

Three vision tasks were defined with aligned prompts and standardized images:

1. **Image Captioning**: General scene description.
2. **Object Detection**: Identification of visible entities.
3. **Scene Understanding**: In-depth analysis of image context.

Each task used real-world JPEGs (f1 racetrack, street intersection, and hiking images), passed to models via base64 encoding or path reference, depending on API support.

The exact prompts and images can be found in Appendix B.

Results Summary

The following metrics were gathered and analyzed for each model:

- **Total Tokens Generated**
- **Total Inference Duration (s)**
- **Total Model Load Time (s)**
- **Time until first Token (s)**
- **Average RAM Used (GB)**
- **Average Power Usage (Watt)**
- **Average Throughput (tokens/sec)**
- **Average Throughput per second per Watt (tokens/watt)**
- **Energy per token (Joules/token)**
- **Success Rate (% of successful completions)**

Separate plots were created for:

- **Throughput by model**
- **Throughput per second per watt by model**
- **Energy per token by model**
- **RAM usage by model**
- **Model load times**
- **Time until first token returned**
- **Success rates (for vision models)**

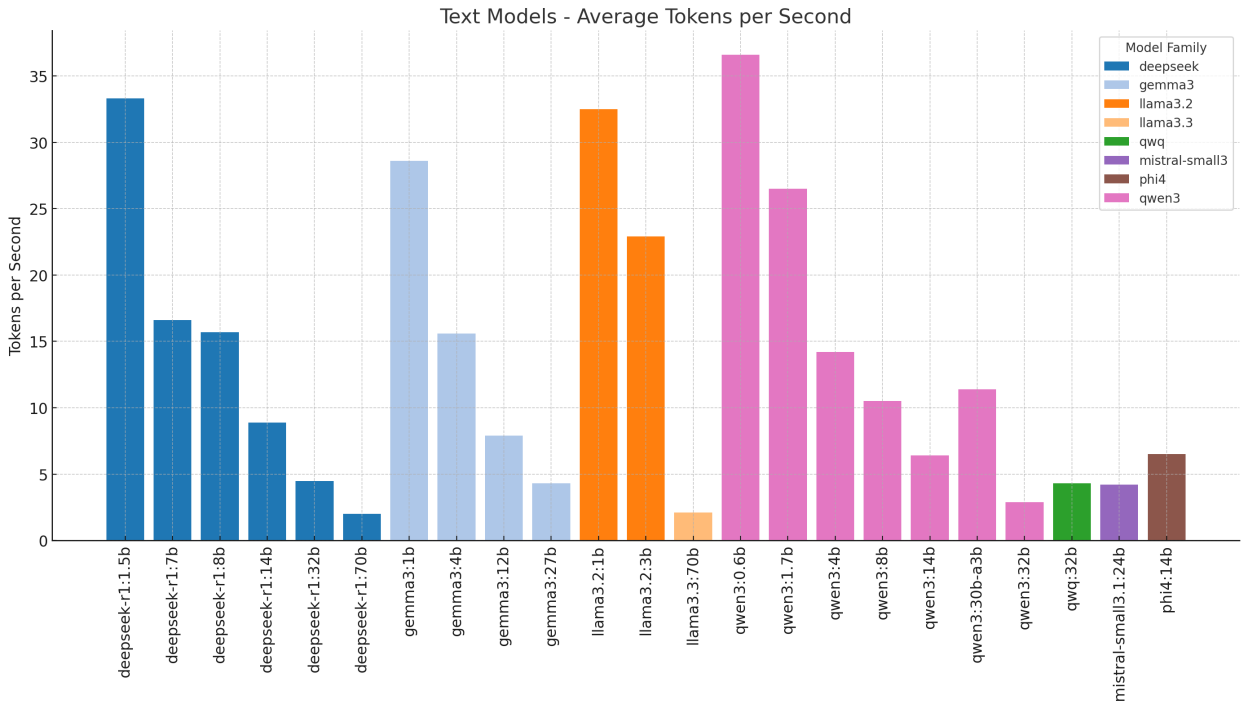
Text Model Results

model	Total Tokens	Total Duration (s)	Model Load Time (s)	Time until first token (s)	Avg RAM Used (GB)	Avg Power Usage (W)	Avg Tokens/sec	Avg TPS/Watt	Joules/ Token
deepseek-r1:1.5b	5384	163.1	3.8	0.132	2.4	20.8	**33.0**	**1.587**	**0.630**
deepseek-r1:7b	3909	253.8	10	0.232	7.0	29.2	**15.4**	**0.527**	**1.896**
deepseek-r1:8b	4094	271.9	12.4	0.266	9.1	31.6	**15.1**	**0.476**	**2.099**
deepseek-r1:14b	4701	573.2	23.5	0.366	14.7	32.9	**8.2**	**0.249**	**4.012**
deepseek-r1:32b	4794	1120.2	63.5	0.606	26.4	35.2	**4.3**	**0.122**	**8.225**
deepseek-r1:70b	5096	2685.6	140.2	1.008	49.9	38.3	**1.9**	**0.050**	**20.184**
gemma3:1b	3171	115.6	3.92	0.266	1.6	19	**27.4**	**1.444**	**0.693**
gemma3:4b	3277	221.9	7.3	0.332	6.5	27.1	**14.8**	**0.545**	**1.835**
gemma3:12b	3918	539.7	10.4	0.41	16.9	31.8	**7.3**	**0.228**	**4.380**
gemma3:27b	3972	1006.3	38	0.624	28.1	34.9	**3.9**	**0.113**	**8.842**
llama3.2:1b	1888	60.8	5.3	0.198	3.7	22.7	**31.1**	**1.368**	**0.731**
llama3.2:3b	1709	78.4	6.6	0.184	5.9	26.9	**21.8**	**0.810**	**1.234**
llama3.3:70b	2593	1327.9	141	1.01	49.9	38.3	**2.0**	**0.051**	**19.614**
qwq:32b	6913	1637.4	58.9	0.586	26.3	35.8	**4.2**	**0.118**	**8.480**
mistral-small3.1:24b	1956	496	17.3	0.44	15.9	35.5	**3.9**	**0.111**	**9.002**
phi4:14b	2123	364.5	22.1	0.256	10.6	34	**5.8**	**0.171**	**5.837**
qwen3:0.6b	3936	109.8	5.2	0.12	1.8	18.4	**35.8**	**1.948**	**0.513**
qwen3:1.7b	7826	305.7	9.2	0.19	2.5	25.6	**25.6**	**1.000**	**1.000**
qwen3:4b	7705	565.8	14.2	0.236	4.3	27.9	**13.6**	**0.488**	**2.049**
qwen3:8b	7880	771.7	20.9	0.356	6.4	31.9	**10.2**	**0.320**	**3.124**
qwen3:14b	7045	1116.5	35.3	0.388	10.4	34.3	**6.3**	**0.184**	**5.436**
qwen3:30b-a3b	8578	762.8	66.3	0.594	11.2	21.6	**11.2**	**0.521**	**1.921**
qwen3:32b	7999	2843.5	77.9	0.642	21.7	34.8	**2.8**	**0.081**	**12.371**

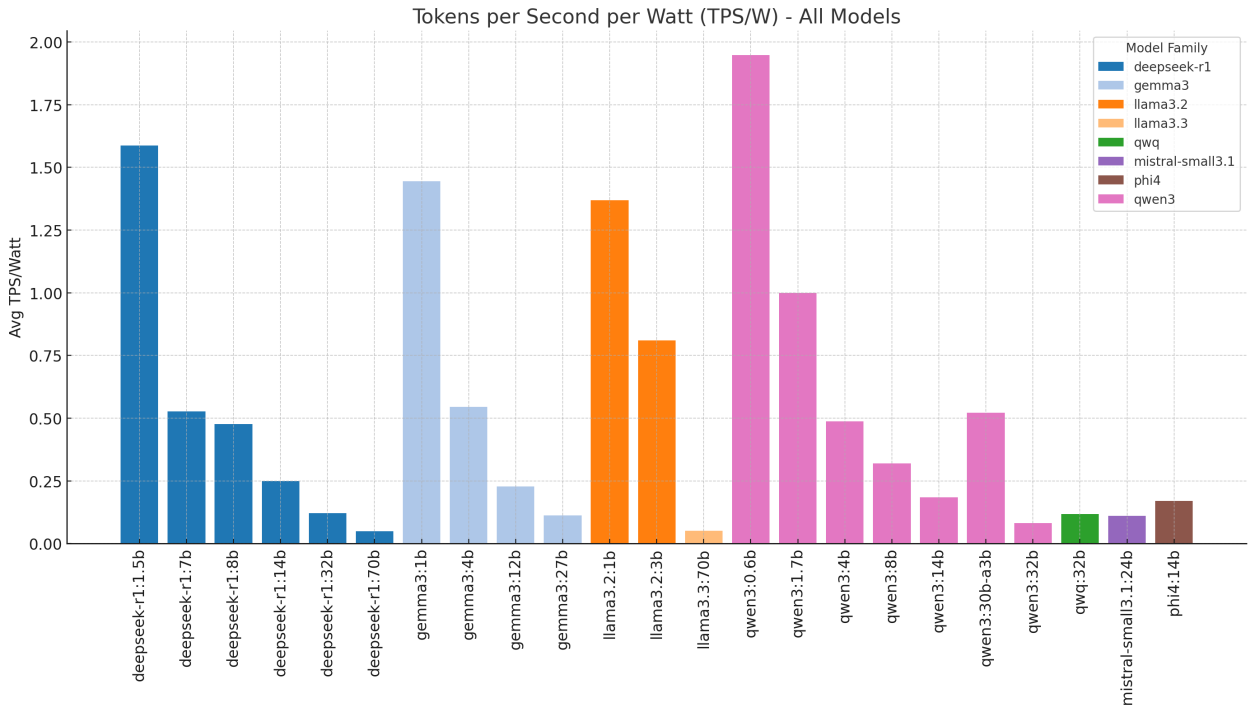
Table 1: text model results across all models

Text-only models demonstrated a clear inverse relationship between size and speed. Lightweight models such as `deepseek-r1:1.5b`, `llama3.2:1b`, `gemma3:1b`, `qwen3:0.6b`, and `qwen3:1.7b` achieved throughput above **25 tokens/sec**, with minimal memory and CPU overhead - making them ideal for low-latency applications or mobile deployment.

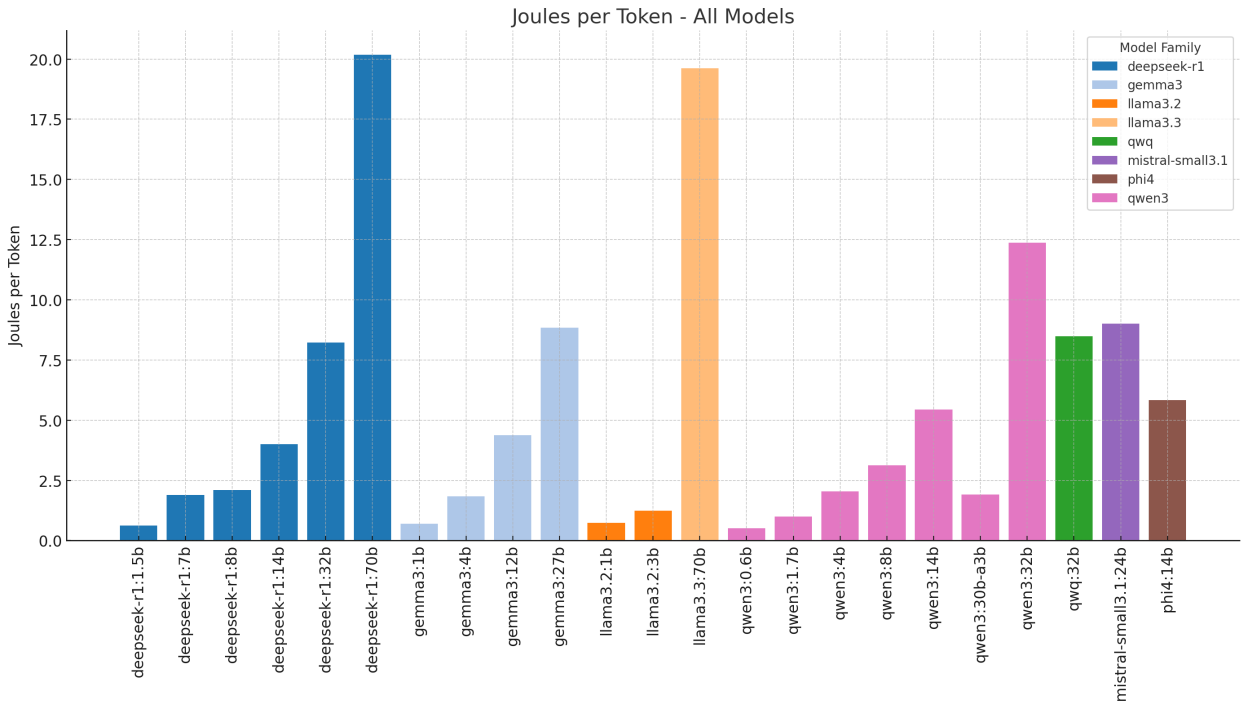
Larger models like `deepseek-r1:70b` and `llama3.3:70b` offered deeper reasoning capabilities but showed **substantially lower throughput** (~2 tokens/sec) and significantly higher RAM demands (up to 50 GB). Notably, even the 70B models completed all prompts successfully without system failure.



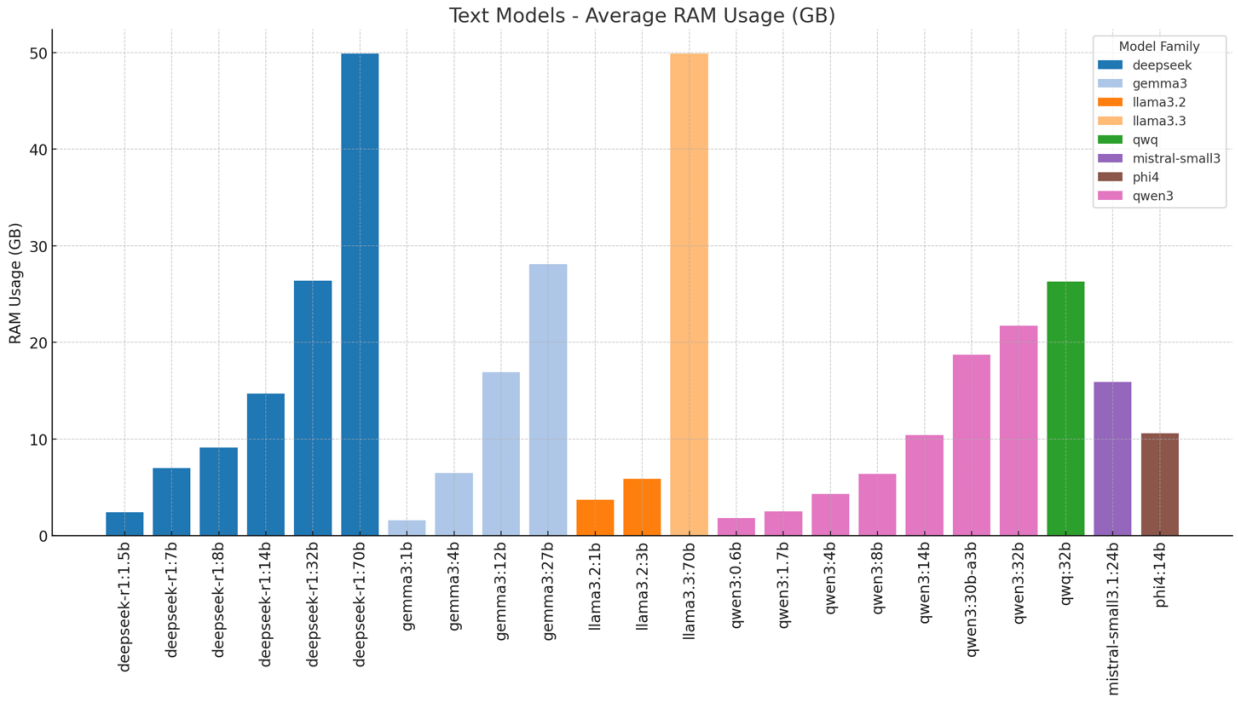
Plot 1: text model tokens/second results



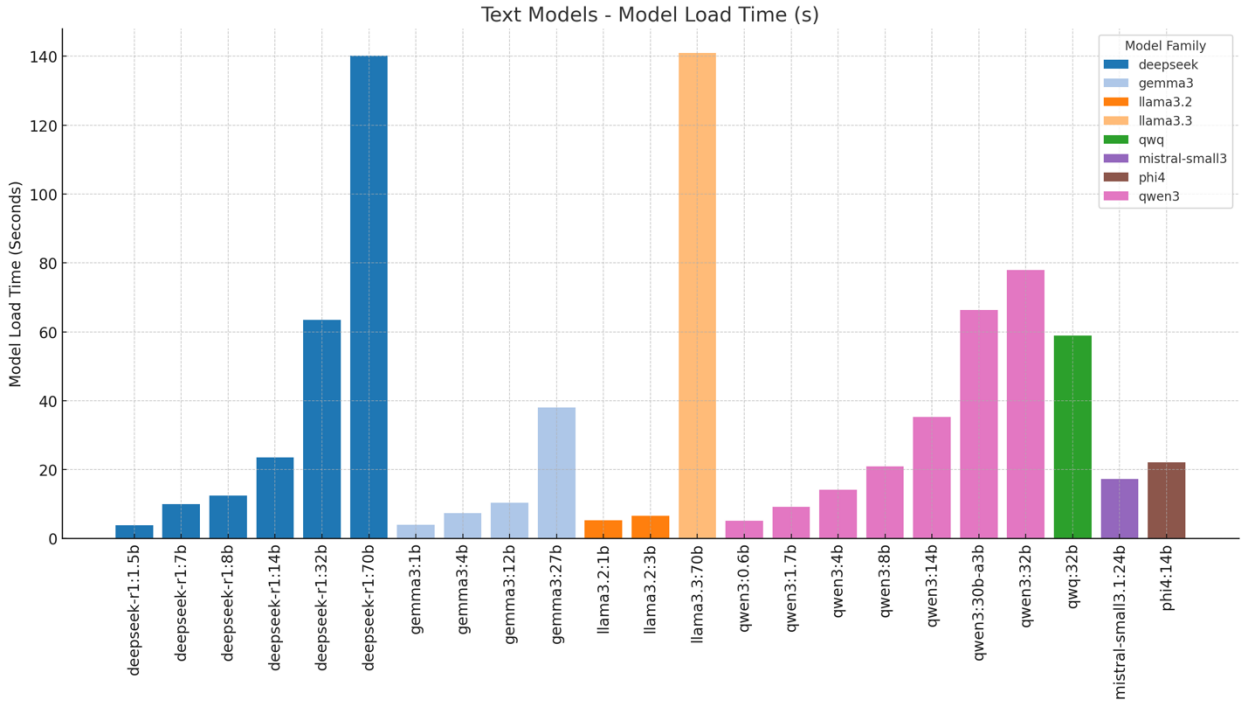
Plot 2: text model average tokens per second per watt



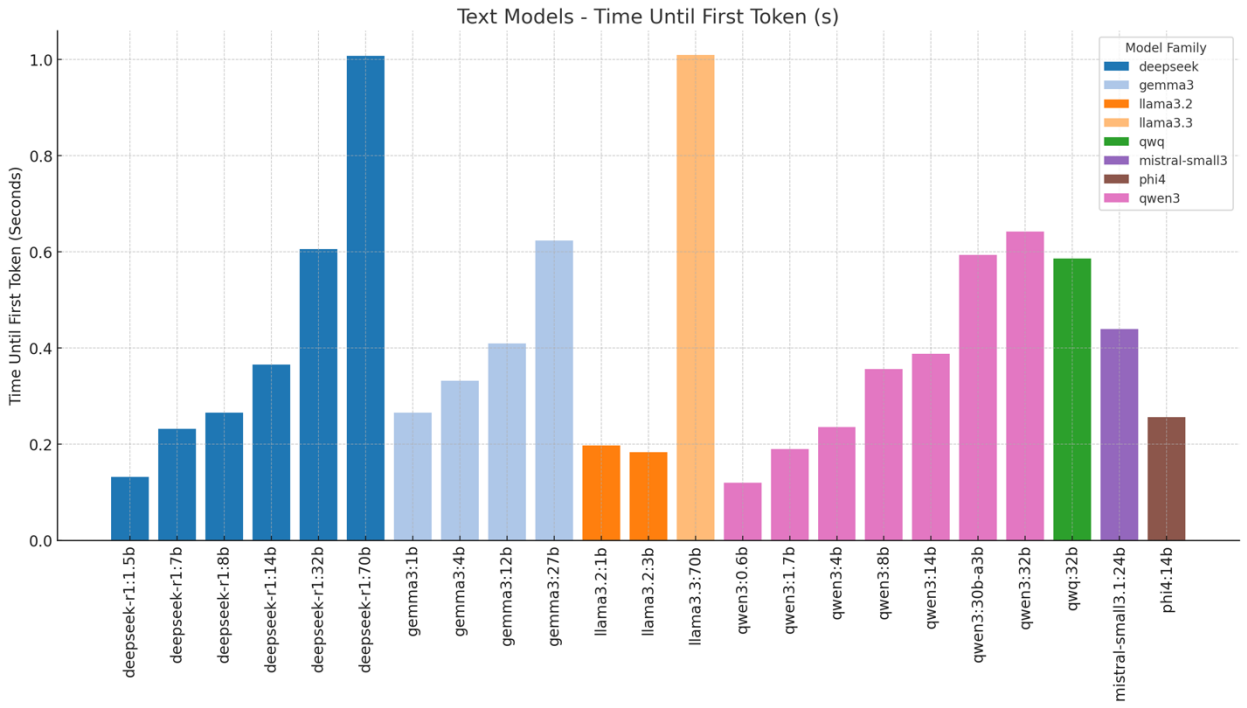
Plot 3: text model average joules per token



Plot 4: text model average RAM usage results



Plot 5: text models model load time

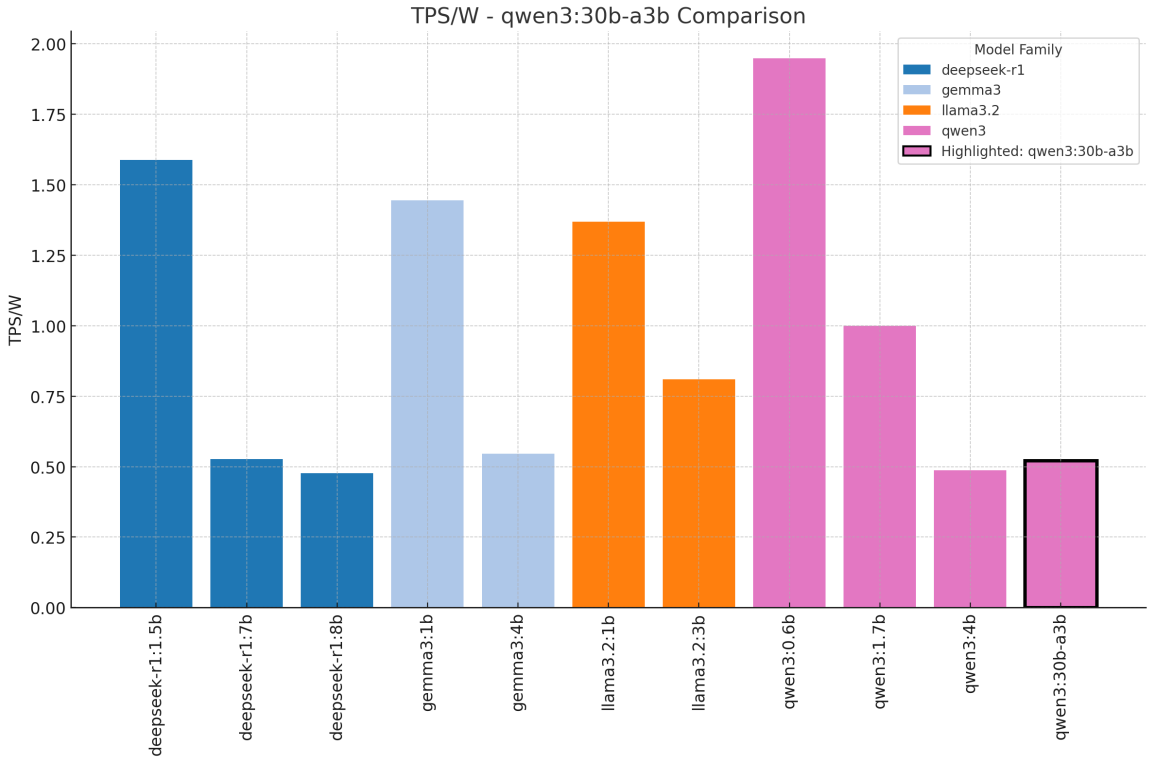


Plot 6: text model average time until first token

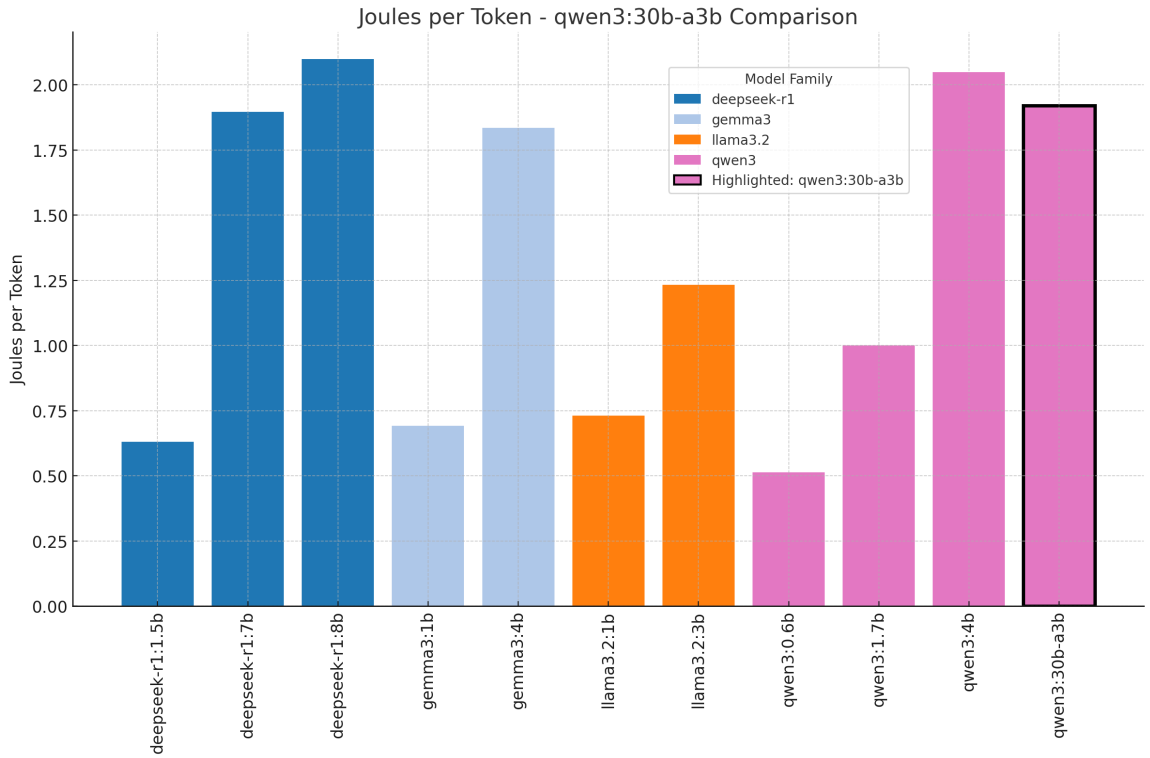
The Model Load time is defined as the amount of time it takes to load the model into memory using the ollama API, and to receive the first token response from the model using a test input prompt. The code snippet used to measure this value is shown in Appendix A.

Most of the model families displayed a downward trend of tokens per second per watt (TPS/W) when increasing the parameter count. The smallest models in each model family (`deepseek-r1:1.5b`, `gemma3:1b`, `llama3.2:1b`, `qwen3:0.6b`) all displayed high throughput per watt above 1.25 TPS/W and low energy usages below 0.75 Joules/token, suggesting they could be good choices for energy constrained deployments. The `qwen3:30b-a3b` model displays an impressive energy throughput of almost 0.5 TPS/W – outperforming* many of the models that have less parameters than the total parameters (30b) and outperforming all the models that have more parameters than the total amount of parameters. Comparing `qwen3:30b-a3b` to models with similar numbers of active parameters, it has similar performance to the 4b parameter models from the qwen3 and gemma3 model families (plots 7 and 8 shown below).

* Outperforming other models strictly on the benchmark of tokens per second per watt

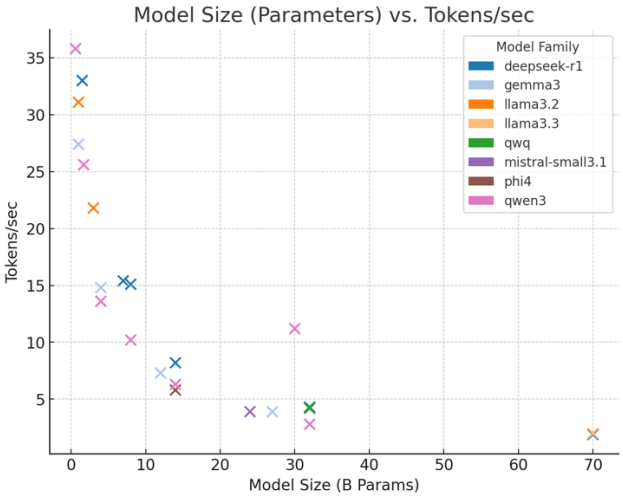


Plot 7: text model comparison subset for tokens per second per watt

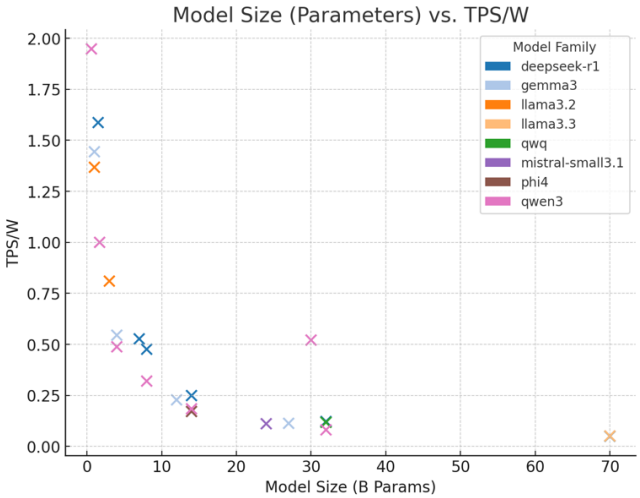


Plot 8: text model comparison subset for joules per token

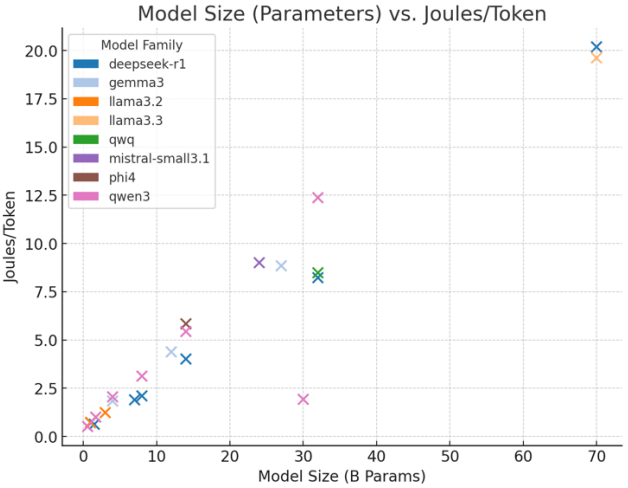
Below are plots of the various metrics (*Tokens/second, TPS/W, and Joules/Token*) against the parameter count of the models they pertain to. These visualizations provide a valuable reference for model selection when considering deployments that prioritize efficiency, power usage, or throughput.



Plot 9: model size vs tokens/s



Plot 10: model size vs TPS/W



Plot 11: model size vs joules/token

The *Tokens/second* and *TPS/W* plots both demonstrate a trend like a *reciprocal function*. As model parameter size increases, the throughput (Tokens/second) and efficiency (TPS/W) tend to diminish significantly.

Given the clear diminishing returns as model size scales up, applying an *elbow rule* might be effective in model selection. In this approach, the point at which the curve flattens represents an optimal balance between model size and efficiency. For example, models in the 3-4B parameter range appear to reside near this optimal region, offering high throughput and efficiency for their size.

The *Joules/Token* plot also displays a growing trend with model size, indicating that energy cost per token increases as the model becomes more complex. This suggests that for deployments where power consumption is a critical factor, smaller models like `qwen3:0.6b`, `gemma3:1b`, and `deepseek-r1:1.5b` provide the lowest energy consumption.

Overall, these plots suggest that:

- For high efficiency and low power consumption, smaller models (<4B parameters) are ideal.
- For balanced performance and efficiency, the *elbow point* in the curves is an effective cutoff.
- For power-sensitive deployments, careful consideration of the *Joules/Token* metric is crucial.

More plots showing the tradeoffs between similar metrics for each of the models is included in Appendix C.

Vision Model Results

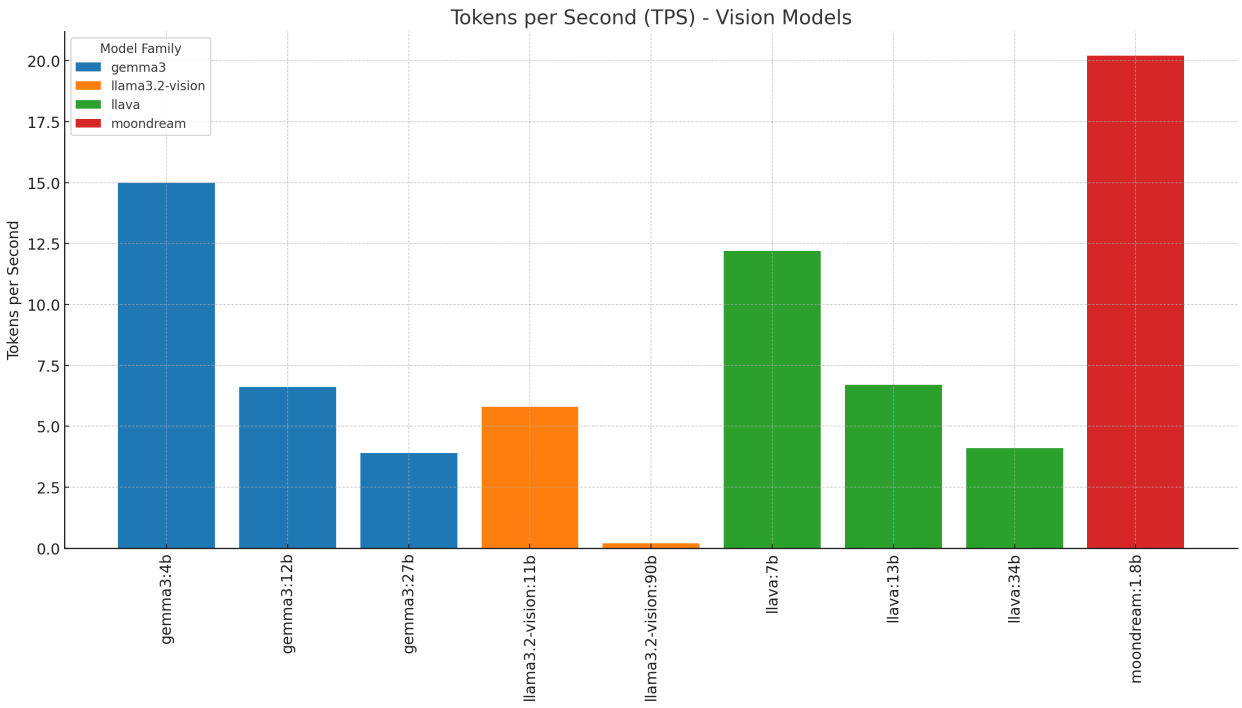
model	Total Tokens	Total Duration (s)	Model Load Time (s)	Time until first token (s)	Avg RAM Used (GB)	Avg Power Usage (W)	Success Rate	Tokens /s	Avg TPS/ Watt	Joules/ Token
gemma3:4b	666	44.4	7.3	2.3	7.7	25.8	100	**15.0**	**0.582**	**1.718**
gemma3:12b	597	89.9	10.4	2.8	18.1	30.4	100	**6.6**	**0.218**	**4.579**
gemma3:27b	682	173	38	3.6	29.4	33.8	100	**3.9**	**0.117**	**8.567**
llama3.2-vision:11b	381	65.4	66.8	7.7	11.3	28.1	100	**5.8**	**0.207**	**4.831**
llama3.2-vision:90b	27	172.7	N/A	N/A	56	N/A	0	**0.2**	**N/A**	**N/A**
llava:7b	276	22.6	17.8	1.1	9.2	28.6	100	**12.2**	**0.427**	**2.340**
llava:13b	282	41.9	24.7	1.7	28.4	29.9	100	**6.7**	**0.225**	**4.450**
llava:34b	386	93.2	75.1	3.7	27.3	34.0	100	**4.1**	**0.122**	**8.209**
moondream:1.8b	206	10.2	7	0.8	7.9	18.2	100	**20.2**	**1.109**	**0.902**

Table 2: vision model results across all models

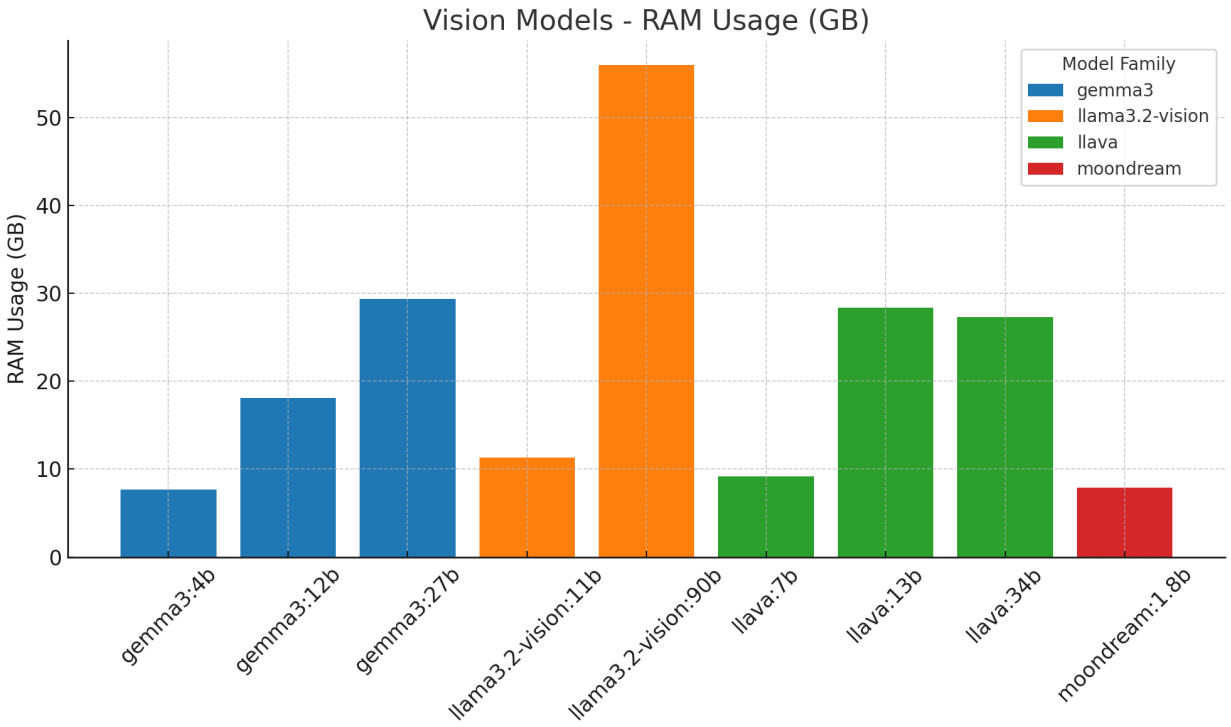
In the multimodal benchmarks, smaller models like `moondream:1.8b` and `llava:7b` achieved the **highest token generation speeds** (12–20 tokens/sec) with consistent 100% success rates and minimal system resource impact.

Notably, the `llava:13b` model used slightly more RAM than the `llava:34b` model, which is unexpected since the parameter count is nearly three times as much. This benchmark was re-run multiple times, and the RAM usage maintained the tabulated values. Currently, no reason can be provided for this behavior.

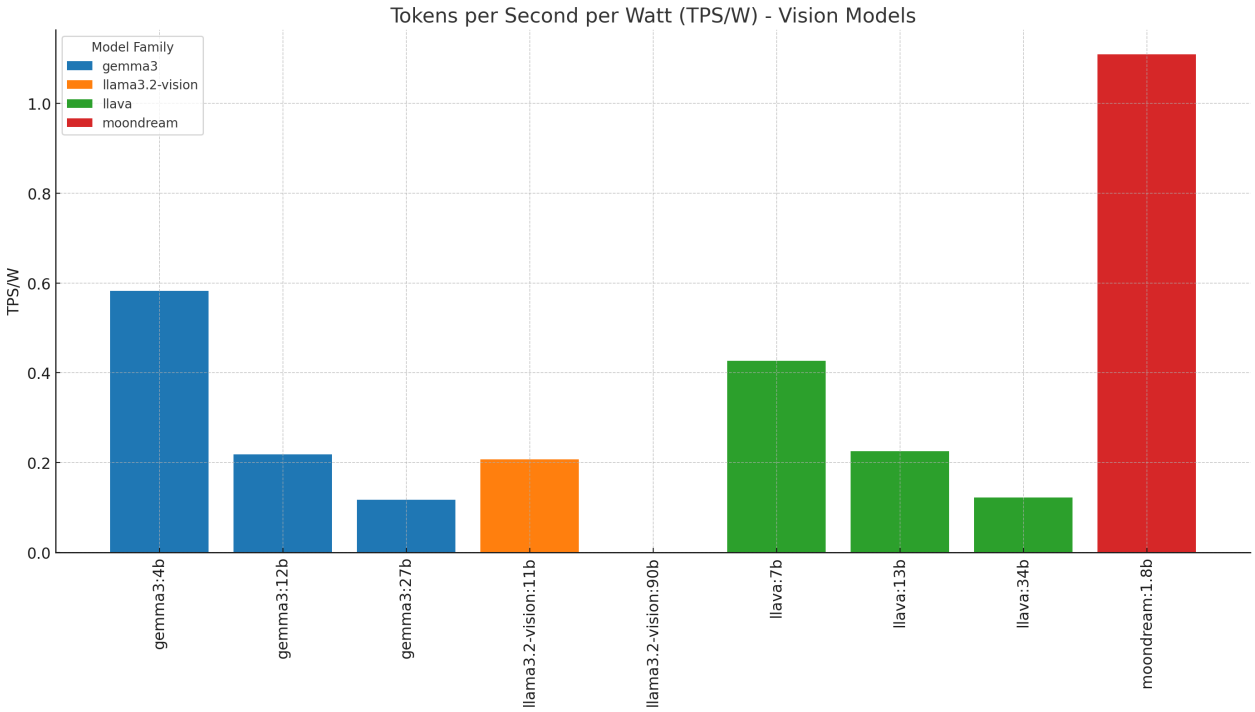
However, as model size increased, inference times lengthened, and performance declined - particularly for `llama3.2-vision:90b`, which failed all prompts due to resource exhaustion or timeout. Interestingly, `llama3.2-vision:11b` was able to complete all tasks with reasonable performance, suggesting it may be a sweet spot for Jetson-scale vision workloads.



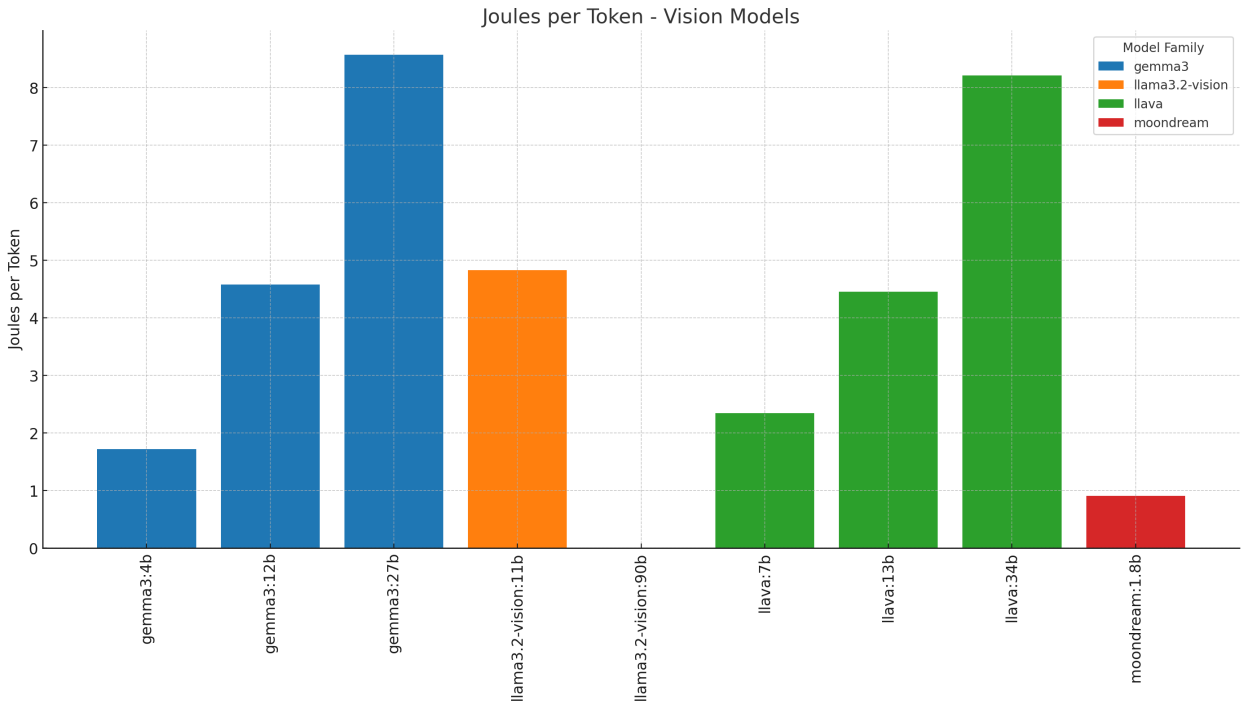
Plot 12: vision model tokens/second results



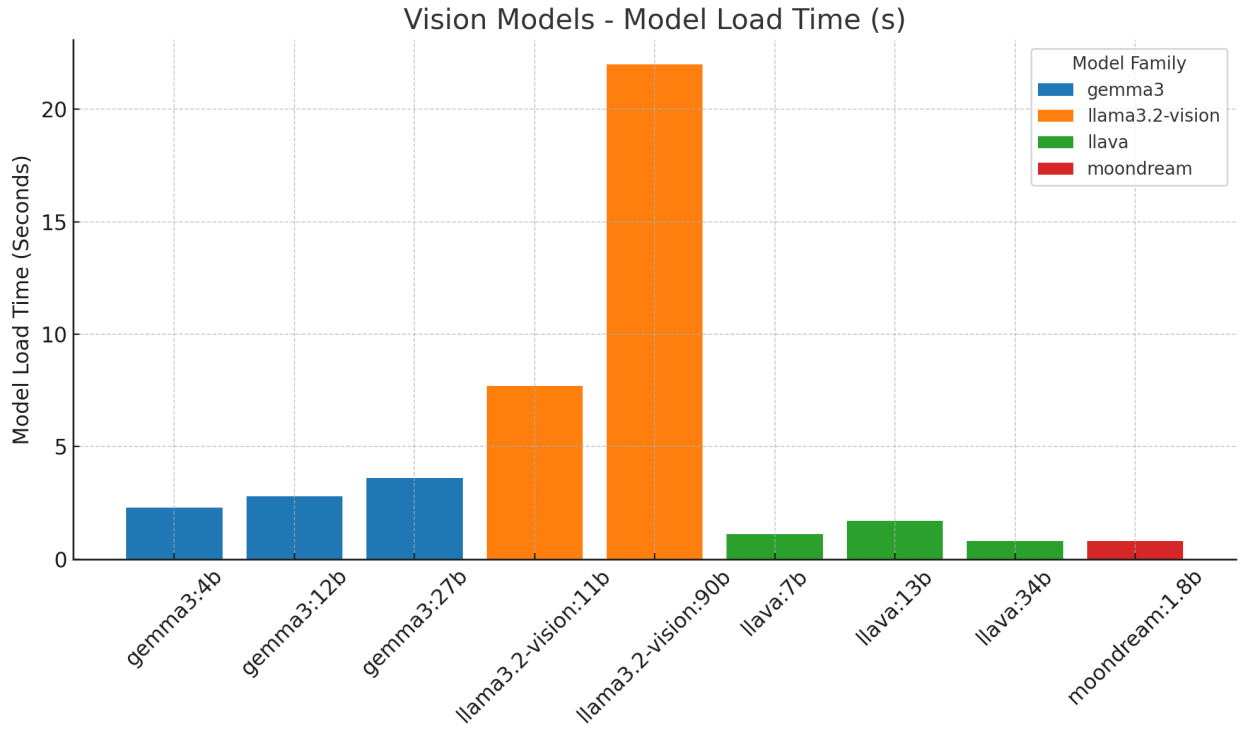
Plot 13: vision model average RAM usage results



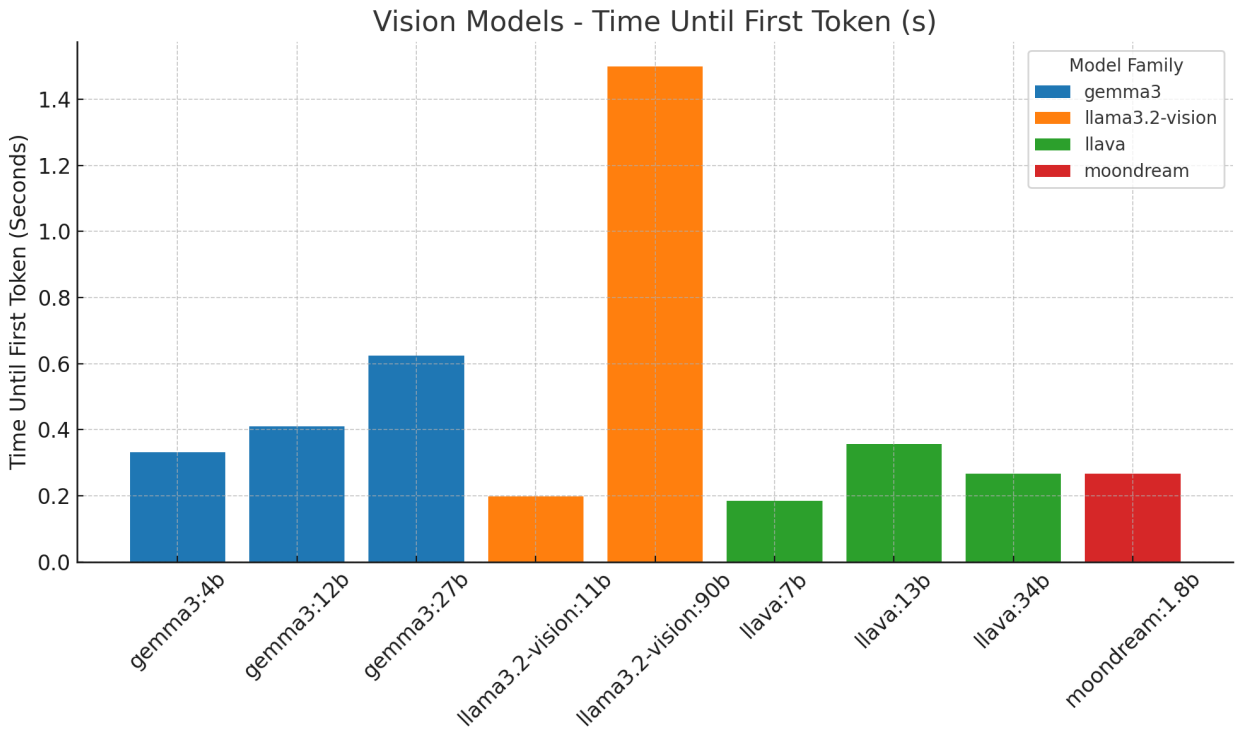
Plot 14: vision model tokens per second per watt



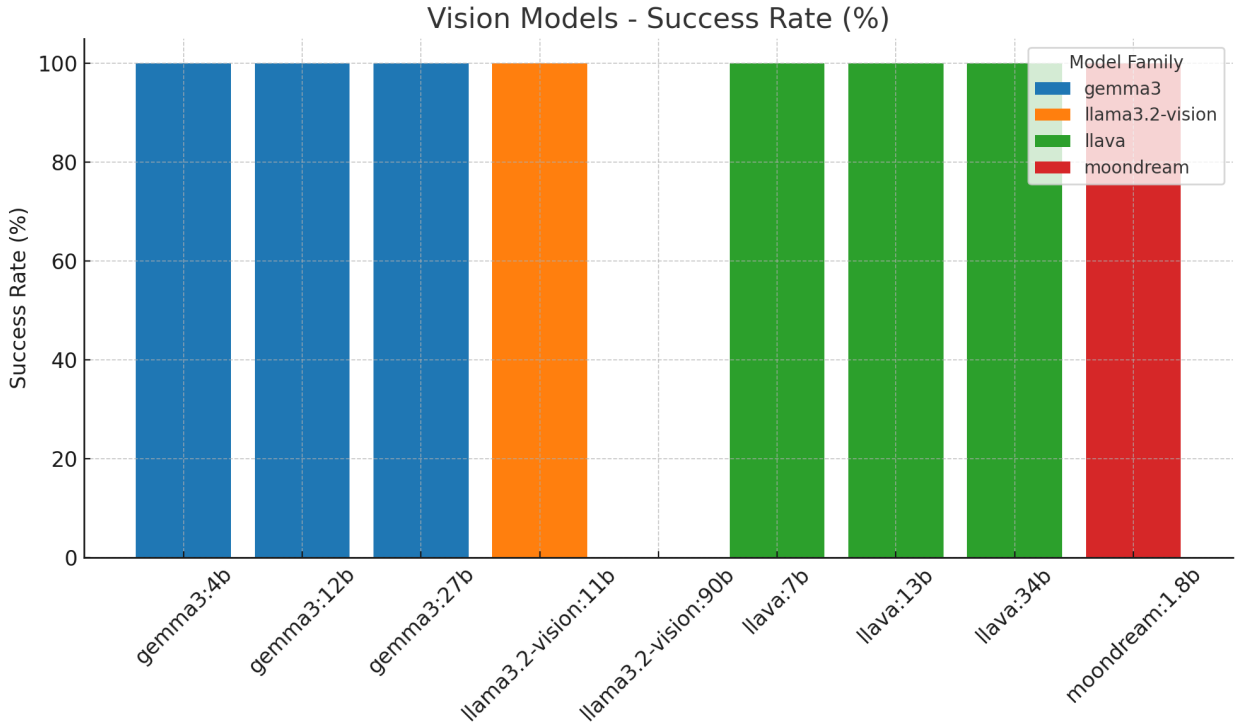
Plot 15: vision model joules per token



Plot 16: vision model load time

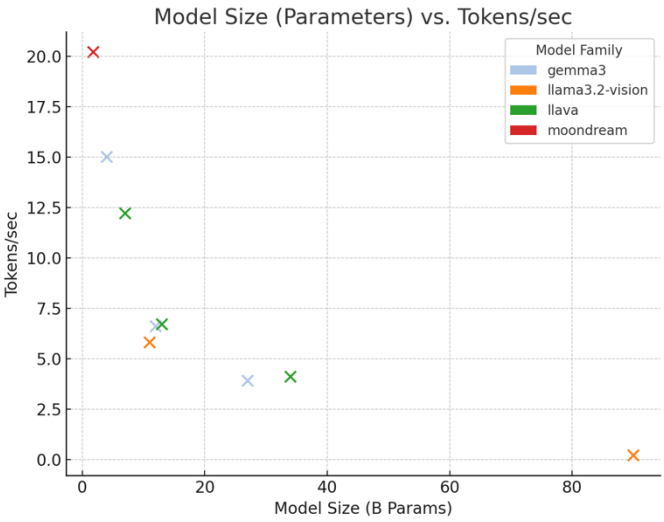


Plot 17: vision model time until first token

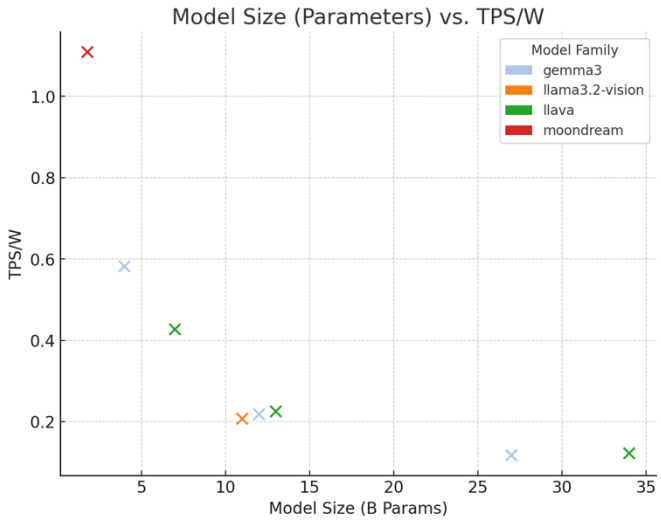


Plot 18: vision model success rate results

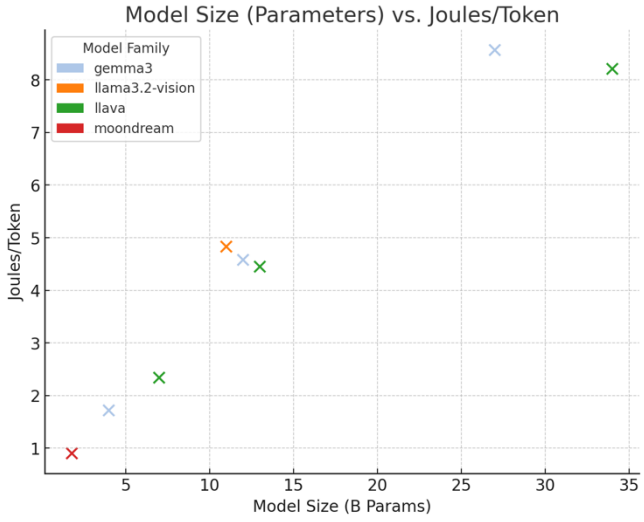
The above visualizations provide key insights into the efficiency and performance of various vision models across different metrics such as *Tokens/sec*, *TPS/W*, *Joules/Token*, *RAM Usage*, and *Power Consumption*.



Plot 19: model size vs tokens/s



Plot 20: model size vs TPS/W



Plot 21: model size vs Joules/token

The visualizations of *Model Size vs. Tokens/sec*, *Model Size vs. TPS/W*, and *Model Size vs. Joules/Token* reveal important trends in model efficiency and performance. As expected, smaller models generally exhibit higher throughput (*Tokens/sec*) and greater power efficiency (*TPS/W*), while also consuming less energy per token (*Joules/Token)*. Notably, `moondream:1.8b` stands out, achieving exceptional throughput and energy efficiency, making it highly suitable for edge deployments or power-sensitive applications.

In contrast, larger models such as `llava:34b` and `gemma3:27b` demonstrate significantly reduced efficiency in terms of both throughput and energy consumption.

The *TPS/W* plots further illustrate a nearly reciprocal trend, highlighting the steep drop in power efficiency as models increase in parameter count. The *Joules/Token* plot reinforces that larger models consume more energy for each token processed, marking them as less sustainable for long-duration tasks in energy constrained deployments.

Applying an elbow rule approach—selecting the point where the curve flattens—could be beneficial for model selection, ensuring the highest efficiency per watt consumed while avoiding steep increases in energy costs. Models like `gemma3:4b` and `llava:7b` present balanced trade-offs between size, efficiency, and power usage.

Conclusion

This benchmarking provides a comparative view of model efficiency and feasibility on edge hardware. Key observations include:

- Smaller models (1B-7B) offered higher TPS and efficiency but were limited in reasoning depth.
- Larger models (14B-90B) showed slower throughput and increased memory strain, with occasional timeouts or failures.
- Smaller models (0.6B-4B) showed higher throughput per second per watt and a lower energy cost per token (Joules/token).
- Vision models varied significantly in reliability—Moondream and LLaVA performed well, while some larger LLaMA variants struggled under RAM constraints.

This document may serve as a foundation for continued benchmarking, deployment optimization, and on-device AI research.

Appendices

Appendix A

The code used to benchmark the image and vision models is provided below, separated by model modality.

Text Model Prompts

```
import time
import json
import os
import gc
import threading
import matplotlib.pyplot as plt
import psutil
import subprocess
from datetime import datetime
import ollama

# -------------------- CLI Argument --------------------
if len(sys.argv) < 2:
    print("Usage: python3 benchmark_multi_models.py <model1> <model2> ...")
    sys.exit(1)

MODELS = sys.argv[1:]

# -------------------- Configuration --------------------
PROMPTS = [
    "Explain step-by-step how to implement a convolutional neural network in
PyTorch.",
    "What are the economic consequences of inflation on low-income
households?",
    "Write a Python program that parses a JSON file and extracts all unique
keys.",
    "What causes seasons to change on Earth? Explain in detail for a 10-
year-old.",
    "Simulate a conversation between Sherlock Holmes and Alan Turing in
1950."
]
RAM_THRESHOLD_GB = 1

# -------------------- GPU Tracker --------------------
class JetsonTracker:
    def __init__(self):
        self.stop_event = threading.Event()
        self.peak_gpu_mem_mb = 0
        self.power_readings = []
        self.thread = threading.Thread(target=self._track_metrics)

    def _track_metrics(self):
        from jtop import jtop
        with jtop() as jetson:
            while not self.stop_event.is_set():
                if jetson.ok():
                    # Track GPU memory
```

```
                    mem = jetson.stats.get("GPU_MEMORY", {}).get("val", 0)
                    self.peak_gpu_mem_mb = max(self.peak_gpu_mem_mb, mem)

                    # Track power usage (in watts)
                    power_dict = {}
                    for rail, data in jetson.power.items():
                        if isinstance(data, dict) and "power" in data:
                            power_dict[rail] = data["power"]

                    # Store the entire power reading
                    self.power_readings.append(power_dict)
                time.sleep(0.5)  # Sample twice per second for better
granularity

    def start(self):
        self.thread.start()

    def stop(self):
        self.stop_event.set()
        self.thread.join()

        # Process power readings
        avg_power = {}
        total_power = 0

        if self.power_readings:
            # Calculate average power for each rail
            rails = set()
            for reading in self.power_readings:
                rails.update(reading.keys())

            for rail in rails:
                values = [r.get(rail, 0) for r in self.power_readings if
rail in r]
                if values:
                    avg_power[rail] = sum(values) / len(values)

            # Calculate total average power - be careful with units!
            # jtop reports power in mW, but we want to display in W
            if avg_power:
                total_power = sum(avg_power.values()) / 1000.0  # Convert mW
to W

        return {
            "peak_gpu_memory_gb": round(self.peak_gpu_mem_mb / 1024, 2),  #
MB to GB
            "avg_power_by_rail": {rail: power/1000.0 for rail, power in
avg_power.items()},  # Convert to W
            "avg_total_power": round(total_power, 2)  # Total power in watts
        }

# -------------------- Helpers --------------------
def get_system_metrics():
    mem = psutil.virtual_memory()
    swap = psutil.swap_memory()
    cpu = psutil.cpu_percent(interval=1)
    return {
```

```
        "cpu_percent": cpu,
        "ram_used_gb": round(mem.used / (1024**3), 2),
        "ram_percent": mem.percent,
        "swap_used_gb": round(swap.used / (1024**3), 2),
        "swap_percent": swap.percent,
        "ram_available_gb": round(mem.available / (1024**3), 2)
    }

def calculate_efficiency_metrics(tokens, duration, power_watts):
    """Calculate efficiency metrics like tokens/s/watt"""
    if duration <= 0 or power_watts <= 0:
        return {
            "tokens_per_sec": 0,
            "tokens_per_watt": 0,
            "tokens_per_sec_per_watt": 0,
            "energy_joules": 0,
            "energy_per_token": 0
        }

    tokens_per_sec = tokens / duration
    energy_joules = power_watts * duration  # Energy in joules (watt-
seconds)

    return {
        "tokens_per_sec": round(tokens_per_sec, 2),
        "tokens_per_watt": round(tokens / energy_joules * duration, 2),
        "tokens_per_sec_per_watt": round(tokens_per_sec / power_watts, 4),
        "energy_joules": round(energy_joules, 2),
        "energy_per_token": round(energy_joules / tokens, 4) if tokens > 0
else 0
    }

def pull_model(model):
    try:
        result = ollama.pull(model)
        if "status" in result and result["status"] == "success":
            print(f"📦 Pulled model '{model}' successfully.")
        else:
            print(f"✔ Model '{model}' already exists locally.")
        return True
    except Exception as e:
        print(f"❌ Failed to pull model '{model}': {e}")
        return False

def plot_metric(model, results, key, ylabel, title, filename):
    plt.figure(figsize=(10, 6))
    x = list(range(1, len(results) + 1))

    # Handle nested keys with dot notation (e.g.,
"power_metrics.avg_total_power_watts")
    if "." in key:
        parts = key.split(".")
        y = []
        for r in results:
            value = r
            for part in parts:
                if isinstance(value, dict) and part in value:
```

```
                    value = value[part]
                else:
                    value = 0
                    break
            y.append(value)
    else:
        y = [r.get(key, 0) for r in results]

    plt.plot(x, y, marker='o', label=model)
    plt.title(title)
    plt.xlabel("Prompt #")
    plt.ylabel(ylabel)
    plt.grid(True)
    plt.legend()
    plt.savefig(filename)
    plt.close()

# -------------------- Benchmark Loop --------------------
for MODEL in MODELS:
    LOG_DIR = f"ollama_benchmark_logs/{MODEL.replace('/', '_')}"
    os.makedirs(LOG_DIR, exist_ok=True)

    available_ram = get_system_metrics()["ram_available_gb"]
    if available_ram < RAM_THRESHOLD_GB:
        print(f"❌ Skipping model '{MODEL}' due to low available RAM:
{available_ram} GB")
        continue

    if not pull_model(MODEL):
        continue

    # Measure model load time
    print(f"⏳ Timing model load for '{MODEL}'...")
    load_start = time.time()
    power_tracker = JetsonTracker()
    power_tracker.start()

    try:
        # Create a client to measure initial model load time
        client = ollama.Client()
        # Send a quick prompt to load the model into memory
        response = client.generate(model=MODEL, prompt="Hello",
stream=False)
        load_end = time.time()

        # Get power metrics during model load
        load_metrics = power_tracker.stop()
        model_load_time_sec = round(load_end - load_start, 2)
        model_load_power = load_metrics["avg_total_power"]

        print(f"✅ Model loaded in {model_load_time_sec} seconds (avg power:
{model_load_power}W)")
    except Exception as e:
        power_tracker.stop()
        print(f"❌ Model failed to load/respond: {e}")
        continue
```

```
    results = []

    for i, prompt in enumerate(PROMPTS):
        print(f"\n🧠 Running prompt {i+1}/{len(PROMPTS)} for model
'{MODEL}'")

        sys_before = get_system_metrics()
        tracker = JetsonTracker()
        tracker.start()

        # Time model load for this specific prompt
        model_load_start = time.time()
        buffer = ""
        first_token_time = None
        last_token_time = None

        try:
            # Stream the response to measure time to first token
            for chunk in client.generate(model=MODEL, prompt=prompt,
stream=True):
                token = chunk.get("response", "")
                if token:
                    if not first_token_time:
                        first_token_time = time.time()
                    last_token_time = time.time()
                    buffer += token

            end_time = last_token_time if last_token_time else time.time()
            metrics = tracker.stop()
            sys_after = get_system_metrics()

            tokens = len(buffer.split())
            duration = end_time - model_load_start
            avg_power = metrics["avg_total_power"]

            # Calculate efficiency metrics
            efficiency = calculate_efficiency_metrics(tokens, duration,
avg_power)

            tps = efficiency["tokens_per_sec"]
            time_to_first_token = round(first_token_time - model_load_start,
2) if first_token_time else None
            model_load_time = round(model_load_start - model_load_start, 2)
# This will be 0.0

            record = {
                "model": MODEL,
                "prompt_index": i,
                "prompt": prompt,
                "tokens": tokens,
                "duration_sec": round(duration, 2),
                "tps": tps,
                "model_load_time_sec": model_load_time_sec,  # Use the
initial model load time
                "time_to_first_token_sec": time_to_first_token,
                "sys_before": sys_before,
```

```
                "sys_after": sys_after,
                "peak_gpu_memory_gb": metrics["peak_gpu_memory_gb"],
                "power_metrics": {
                    "avg_total_power_watts": metrics["avg_total_power"],
                    "avg_power_by_rail": metrics["avg_power_by_rail"]
                },
                "efficiency_metrics": efficiency,
                "timestamp": datetime.now().isoformat()
            }

            results.append(record)

            with open(os.path.join(LOG_DIR, f"prompt_{i+1}_output.txt"),
"w") as out_file:
                out_file.write(buffer)

            print(f"✅ Done: {tokens} tokens in {duration:.2f}s (tps={tps})
| TTF={time_to_first_token}s | Load={model_load_time_sec}s |
Power={metrics['avg_total_power']}W |
Efficiency={efficiency['tokens_per_sec_per_watt']} tokens/s/W")

        except Exception as e:
            print(f"❌ Benchmark failed for prompt {i+1}: {e}")
            tracker.stop()

    with open(os.path.join(LOG_DIR, "benchmark_results.json"), "w") as f:
        json.dump(results, f, indent=2)

    # Generate plots
    if results:
        # Basic metrics
        plot_metric(MODEL, results, "tps", "Tokens/sec", "Throughput per
Prompt", os.path.join(LOG_DIR, "throughput_plot.png"))
        plot_metric(MODEL, results, "duration_sec", "Latency (sec)",
"Latency per Prompt", os.path.join(LOG_DIR, "latency_plot.png"))
        plot_metric(MODEL, results, "time_to_first_token_sec", "Time (sec)",
"Time to First Token per Prompt", os.path.join(LOG_DIR, "ttft_plot.png"))

        # Power and efficiency metrics
        power_values = [r["power_metrics"]["avg_total_power_watts"] for r in
results]
        efficiency_values =
[r["efficiency_metrics"]["tokens_per_sec_per_watt"] for r in results]
        energy_per_token = [r["efficiency_metrics"]["energy_per_token"] for
r in results]

        plot_metric(MODEL, results, "power_metrics.avg_total_power_watts",
"Power (W)", "Power Usage per Prompt", os.path.join(LOG_DIR,
"power_plot.png"))
        plot_metric(MODEL, results,
"efficiency_metrics.tokens_per_sec_per_watt", "Tokens/sec/W", "Efficiency
per Prompt", os.path.join(LOG_DIR, "efficiency_plot.png"))
        plot_metric(MODEL, results, "efficiency_metrics.energy_per_token",
"Joules/token", "Energy per Token", os.path.join(LOG_DIR,
"energy_per_token_plot.png"))

        # Create a new plot for model load time
```

```
        plt.figure(figsize=(10, 6))
        plt.bar(["Model Load Time"], [model_load_time_sec])
        plt.title(f"Model Load Time for {MODEL}")
        plt.ylabel("Time (sec)")
        plt.grid(True, axis='y')
        plt.savefig(os.path.join(LOG_DIR, "model_load_time_plot.png"))
        plt.close()

        # Create summary plot with multiple metrics for model comparison
        plt.figure(figsize=(12, 10))

        plt.subplot(2, 2, 1)
        plt.bar(["Avg TPS"], [sum(r["tps"] for r in results) /
len(results)])
        plt.title("Average Throughput (tokens/sec)")
        plt.grid(True, axis='y')

        plt.subplot(2, 2, 2)
        plt.bar(["Avg Power"], [sum(power_values) / len(power_values)])
        plt.title("Average Power Consumption (W)")
        plt.grid(True, axis='y')

        plt.subplot(2, 2, 3)
        plt.bar(["Avg Efficiency"], [sum(efficiency_values) /
len(efficiency_values)])
        plt.title("Average Efficiency (tokens/sec/W)")
        plt.grid(True, axis='y')

        plt.subplot(2, 2, 4)
        plt.bar(["Avg Energy/Token"], [sum(energy_per_token) /
len(energy_per_token)])
        plt.title("Average Energy per Token (J/token)")
        plt.grid(True, axis='y')

        plt.tight_layout()
        plt.savefig(os.path.join(LOG_DIR, "summary_metrics.png"))
        plt.close()

    # Memory flush with fallback options
    cache_cleared = False

    # Try using sync which usually doesn't require sudo
    try:
        subprocess.run(["sync"], check=True)
        print("🔄 Synced file system buffers.")
    except Exception as e:
        print(f"⚠️ Could not sync: {e}")

    # Option 1: Try sudo with -n (non-interactive) flag which will fail
rather than prompt for password
    try:
        subprocess.run(["sudo", "-n", "bash", "-c", "echo 3 >
/proc/sys/vm/drop_caches"],
                       check=False, stderr=subprocess.PIPE,
stdout=subprocess.PIPE)
        print("🧠 Dropped Linux memory caches using sudo.")
        cache_cleared = True
```

```
    except Exception:
        # Option 2: Try direct write if running as root (unlikely but
possible)
        try:
            with open("/proc/sys/vm/drop_caches", "w") as f:
                f.write("3")
            print("🧠 Dropped Linux memory caches directly.")
            cache_cleared = True
        except Exception:
            # Option 3: Inform and continue
            print("📝 Skipping cache drop (requires sudo). Consider adding
this line to sudoers:")
            print("    yourusername ALL=(ALL) NOPASSWD: /bin/bash -c echo 3
> /proc/sys/vm/drop_caches")

    # Always run garbage collection regardless of cache drop success
    gc.collect()
    print("♻️ Python garbage collection completed.")

    print(f"\n✅ Benchmark complete for '{MODEL}'. Results saved in:
{LOG_DIR}")
```

Code Block 1: text model benchmarking code

Vision Model Prompts

```
import sys
import time
import json
import os
import gc
import threading
import matplotlib.pyplot as plt
import psutil
import base64
import subprocess
import ollama
from datetime import datetime

# -------------------- CLI Argument --------------------
if len(sys.argv) < 2:
    print("Usage: python3 benchmark_vision_models.py <model1> <model2> ...")
    sys.exit(1)

MODELS = sys.argv[1:]

# -------------------- Configuration --------------------
VISION_TASKS = {
    "image_captioning": {
        "prompt": "Describe the content of this image.",
        "image_path": "example_images/f1.jpg"
    },
    "object_detection": {
        "prompt": "What objects are visible in this image?",
        "image_path": "example_images/intersection.jpg"
```

```
    },
    "scene_understanding": {
        "prompt": "Describe the scene in this image in detail.",
        "image_path": "example_images/hiking.jpg"
    }
}
RAM_THRESHOLD_GB = 1

# -------------------- Jetson Tracker --------------------
class JetsonTracker:
    def __init__(self):
        self.stop_event = threading.Event()
        self.peak_gpu_mem_mb = 0
        self.power_readings = []
        self.thread = threading.Thread(target=self._track_metrics)

    def _track_metrics(self):
        from jtop import jtop
        with jtop() as jetson:
            while not self.stop_event.is_set():
                if jetson.ok():
                    # Track GPU memory
                    mem = jetson.stats.get("GPU_MEMORY", {}).get("val", 0)
                    self.peak_gpu_mem_mb = max(self.peak_gpu_mem_mb, mem)

                    # Track power usage (in watts)
                    power_dict = {}
                    for rail, data in jetson.power.items():
                        if isinstance(data, dict) and "power" in data:
                            power_dict[rail] = data["power"]

                    # Store the entire power reading
                    self.power_readings.append(power_dict)
                time.sleep(0.5)  # Sample twice per second for better
granularity

    def start(self):
        self.thread.start()

    def stop(self):
        self.stop_event.set()
        self.thread.join()

        # Process power readings
        avg_power = {}
        total_power = 0

        if self.power_readings:
            # Calculate average power for each rail
            rails = set()
            for reading in self.power_readings:
                rails.update(reading.keys())

            for rail in rails:
                values = [r.get(rail, 0) for r in self.power_readings if
rail in r]
                if values:
```

```
                    avg_power[rail] = sum(values) / len(values)

            # Calculate total average power - be careful with units!
            # jtop reports power in mW, but we want to display in W
            if avg_power:
                total_power = sum(avg_power.values()) / 1000.0  # Convert mW
to W

        return {
            "peak_gpu_memory_gb": round(self.peak_gpu_mem_mb / 1024, 2),  #
MB to GB
            "avg_power_by_rail": {rail: power/1000.0 for rail, power in
avg_power.items()},  # Convert to W
            "avg_total_power": round(total_power, 2)  # Total power in watts
        }

# -------------------- Helpers --------------------
def get_system_metrics():
    mem = psutil.virtual_memory()
    swap = psutil.swap_memory()
    cpu = psutil.cpu_percent(interval=1)
    return {
        "cpu_percent": cpu,
        "ram_used_gb": round(mem.used / (1024**3), 2),
        "ram_percent": mem.percent,
        "swap_used_gb": round(swap.used / (1024**3), 2),
        "swap_percent": swap.percent,
        "ram_available_gb": round(mem.available / (1024**3), 2)
    }

def calculate_efficiency_metrics(tokens, duration, power_watts):
    """Calculate efficiency metrics like tokens/s/watt"""
    if duration <= 0 or power_watts <= 0:
        return {
            "tokens_per_sec": 0,
            "tokens_per_watt": 0,
            "tokens_per_sec_per_watt": 0,
            "energy_joules": 0,
            "energy_per_token": 0
        }

    tokens_per_sec = tokens / duration
    energy_joules = power_watts * duration  # Energy in joules (watt-
seconds)

    return {
        "tokens_per_sec": round(tokens_per_sec, 2),
        "tokens_per_watt": round(tokens / energy_joules * duration, 2),
        "tokens_per_sec_per_watt": round(tokens_per_sec / power_watts, 4),
        "energy_joules": round(energy_joules, 2),
        "energy_per_token": round(energy_joules / tokens, 4) if tokens > 0
else 0
    }

def pull_model(model):
    try:
        result = ollama.pull(model)
```

```
        if "status" in result and result["status"] == "success":
            print(f"📦 Pulled model '{model}' successfully.")
        else:
            print(f"✔ Model '{model}' already exists locally.")
        return True
    except Exception as e:
        print(f"❌ Failed to pull model '{model}': {e}")
        return False

def prepare_image(image_path):
    """Read and encode image for API use"""
    try:
        with open(image_path, "rb") as img_file:
            image_data = img_file.read()
        encoded_image = base64.b64encode(image_data).decode("utf-8")
        return encoded_image
    except Exception as e:
        print(f"⚠ Could not read image '{image_path}': {e}")
        return None

def plot_metric(model, results, key, ylabel, title, filename):
    plt.figure(figsize=(10, 6))
    x = list(range(1, len(results) + 1))

    # Handle nested keys with dot notation (e.g.,
"power_metrics.avg_total_power_watts")
    if "." in key:
        parts = key.split(".")
        y = []
        for r in results:
            value = r
            for part in parts:
                if isinstance(value, dict) and part in value:
                    value = value[part]
                else:
                    value = 0
                    break
            y.append(value)
    else:
        y = [r.get(key, 0) for r in results]

    plt.plot(x, y, marker='o', label=model)
    plt.title(title)
    plt.xlabel("Task #")
    plt.ylabel(ylabel)
    plt.grid(True)
    plt.legend()
    plt.savefig(filename)
    plt.close()

# -------------------- Benchmark Loop --------------------
for MODEL in MODELS:
    LOG_DIR = f"ollama_benchmark_logs/{MODEL.replace('/', '_')}_vision"
    os.makedirs(LOG_DIR, exist_ok=True)

    available_ram = get_system_metrics()["ram_available_gb"]
```

```
    if available_ram < RAM_THRESHOLD_GB:
        print(f"❌ Skipping model '{MODEL}' due to low available RAM:
{available_ram} GB")
        continue

    if not pull_model(MODEL):
        continue

    # Measure model load time
    print(f"⏳ Timing model load for '{MODEL}'...")
    load_start = time.time()
    power_tracker = JetsonTracker()
    power_tracker.start()

    try:
        # Create a client to measure initial model load time
        client = ollama.Client()

        # Check if the model is a vision model and prepare a simple image
test
        test_image_path = list(VISION_TASKS.values())[0]["image_path"]
        encoded_test_image = prepare_image(test_image_path)

        # Send a quick prompt to load the model into memory
        if any(key in MODEL for key in ["llama3.2", "gemma3", "moondream"]):
            # Use chat endpoint
            response = client.chat(
                model=MODEL,
                messages=[
                    {
                        "role": "user",
                        "content": "Hello",
                        "images": [encoded_test_image]
                    }
                ],
                stream=False
            )
        else:
            # Use generate endpoint
            response = client.generate(
                model=MODEL,
                prompt="Hello",
                images=[encoded_test_image],
                stream=False
            )

        load_end = time.time()

        # Get power metrics during model load
        load_metrics = power_tracker.stop()
        model_load_time_sec = round(load_end - load_start, 2)
        model_load_power = load_metrics["avg_total_power"]

        print(f"✅ Model loaded in {model_load_time_sec} seconds (avg power:
{model_load_power}W)")
    except Exception as e:
        power_tracker.stop()
```

```
        print(f"❌ Model failed to load/respond: {e}")
        continue

    results = []

    for i, (task_name, task) in enumerate(VISION_TASKS.items()):
        print(f"\n🧠 Running task '{task_name}' ({i+1}/{len(VISION_TASKS)})
for model '{MODEL}'")

        # Prepare image
        image_path = task["image_path"]
        encoded_image = prepare_image(image_path)
        if not encoded_image:
            continue

        sys_before = get_system_metrics()
        tracker = JetsonTracker()
        tracker.start()

        # Start timing
        model_load_start = time.time()
        buffer = ""
        first_token_time = None
        last_token_time = None

        try:
            # Process based on model type with streaming for timing metrics
            if any(key in MODEL for key in ["llama3.2", "gemma3",
"moondream"]):
                # Chat API with streaming
                first_token_received = False
                for chunk in client.chat(
                    model=MODEL,
                    messages=[
                        {
                            "role": "user",
                            "content": task["prompt"],
                            "images": [encoded_image]
                        }
                    ],
                    stream=True
                ):
                    token = chunk.get("message", {}).get("content", "")
                    if token and not first_token_received:
                        first_token_time = time.time()
                        first_token_received = True
                    if token:
                        last_token_time = time.time()
                        buffer += token
            else:
                # Generate API with streaming
                for chunk in client.generate(
                    model=MODEL,
                    prompt=task["prompt"],
                    images=[encoded_image],
                    stream=True
                ):
```

```
                    token = chunk.get("response", "")
                    if token and not first_token_time:
                        first_token_time = time.time()
                    if token:
                        last_token_time = time.time()
                        buffer += token

            # Calculate metrics
            end_time = last_token_time if last_token_time else time.time()
            metrics = tracker.stop()
            sys_after = get_system_metrics()

            tokens = len(buffer.split())
            duration = end_time - model_load_start
            avg_power = metrics["avg_total_power"]

            # Calculate efficiency metrics
            efficiency = calculate_efficiency_metrics(tokens, duration,
avg_power)

            tps = efficiency["tokens_per_sec"]
            time_to_first_token = round(first_token_time - model_load_start,
2) if first_token_time else None

            record = {
                "model": MODEL,
                "task": task_name,
                "prompt": task["prompt"],
                "tokens": tokens,
                "duration_sec": round(duration, 2),
                "tps": tps,
                "model_load_time_sec": model_load_time_sec,
                "time_to_first_token_sec": time_to_first_token,
                "sys_before": sys_before,
                "sys_after": sys_after,
                "peak_gpu_memory_gb": metrics["peak_gpu_memory_gb"],
                "power_metrics": {
                    "avg_total_power_watts": metrics["avg_total_power"],
                    "avg_power_by_rail": metrics["avg_power_by_rail"]
                },
                "efficiency_metrics": efficiency,
                "timestamp": datetime.now().isoformat(),
                "output": buffer
            }

            results.append(record)

            # Save output
            with open(os.path.join(LOG_DIR, f"{task_name}_output.txt"), "w")
as out_file:
                out_file.write(buffer)

            with open(os.path.join(LOG_DIR, f"{task_name}_metrics.json"),
"w") as f:
                json.dump(record, f, indent=2)
```

```
            print(f"✅ Task '{task_name}' complete: {tokens} tokens in
{duration:.2f}s (tps={tps}) | TTF={time_to_first_token}s |
Power={metrics['avg_total_power']}W |
Efficiency={efficiency['tokens_per_sec_per_watt']} tokens/s/W")

        except Exception as e:
            print(f"❌ Task failed: {e}")
            tracker.stop()

    # Save all results
    with open(os.path.join(LOG_DIR, "vision_benchmark_results.json"), "w")
as f:
        json.dump(results, f, indent=2)

    # Generate plots if we have results
    if results:
        # Basic metrics
        plot_metric(MODEL, results, "tps", "Tokens/sec", "Throughput per
Task", os.path.join(LOG_DIR, "throughput_plot.png"))
        plot_metric(MODEL, results, "duration_sec", "Latency (sec)",
"Latency per Task", os.path.join(LOG_DIR, "latency_plot.png"))
        plot_metric(MODEL, results, "time_to_first_token_sec", "Time (sec)",
"Time to First Token per Task", os.path.join(LOG_DIR, "ttft_plot.png"))

        # Power and efficiency metrics
        power_values = [r["power_metrics"]["avg_total_power_watts"] for r in
results]
        efficiency_values =
[r["efficiency_metrics"]["tokens_per_sec_per_watt"] for r in results]
        energy_per_token = [r["efficiency_metrics"]["energy_per_token"] for
r in results]

        plot_metric(MODEL, results, "power_metrics.avg_total_power_watts",
"Power (W)", "Power Usage per Task", os.path.join(LOG_DIR,
"power_plot.png"))
        plot_metric(MODEL, results,
"efficiency_metrics.tokens_per_sec_per_watt", "Tokens/sec/W", "Efficiency
per Task", os.path.join(LOG_DIR, "efficiency_plot.png"))
        plot_metric(MODEL, results, "efficiency_metrics.energy_per_token",
"Joules/token", "Energy per Token", os.path.join(LOG_DIR,
"energy_per_token_plot.png"))

        # Create a new plot for model load time
        plt.figure(figsize=(10, 6))
        plt.bar(["Model Load Time"], [model_load_time_sec])
        plt.title(f"Model Load Time for {MODEL}")
        plt.ylabel("Time (sec)")
        plt.grid(True, axis='y')
        plt.savefig(os.path.join(LOG_DIR, "model_load_time_plot.png"))
        plt.close()

        # Create summary plot with multiple metrics for model comparison
        plt.figure(figsize=(12, 10))

        plt.subplot(2, 2, 1)
        plt.bar(["Avg TPS"], [sum(r["tps"] for r in results) /
len(results)])
```

```
        plt.title("Average Throughput (tokens/sec)")
        plt.grid(True, axis='y')

        plt.subplot(2, 2, 2)
        plt.bar(["Avg Power"], [sum(power_values) / len(power_values)])
        plt.title("Average Power Consumption (W)")
        plt.grid(True, axis='y')

        plt.subplot(2, 2, 3)
        plt.bar(["Avg Efficiency"], [sum(efficiency_values) /
len(efficiency_values)])
        plt.title("Average Efficiency (tokens/sec/W)")
        plt.grid(True, axis='y')

        plt.subplot(2, 2, 4)
        plt.bar(["Avg Energy/Token"], [sum(energy_per_token) /
len(energy_per_token)])
        plt.title("Average Energy per Token (J/token)")
        plt.grid(True, axis='y')

        plt.tight_layout()
        plt.savefig(os.path.join(LOG_DIR, "summary_metrics.png"))
        plt.close()

    # Memory flush with fallback options
    cache_cleared = False

    # Try using sync which usually doesn't require sudo
    try:
        subprocess.run(["sync"], check=True)
        print("🔄 Synced file system buffers.")
    except Exception as e:
        print(f"⚠️ Could not sync: {e}")

    # Option 1: Try sudo with -n (non-interactive) flag which will fail
rather than prompt for password
    try:
        subprocess.run(["sudo", "-n", "bash", "-c", "echo 3 >
/proc/sys/vm/drop_caches"],
                       check=False, stderr=subprocess.PIPE,
stdout=subprocess.PIPE)
        print("🧠 Dropped Linux memory caches using sudo.")
        cache_cleared = True
    except Exception:
        # Option 2: Try direct write if running as root (unlikely but
possible)
        try:
            with open("/proc/sys/vm/drop_caches", "w") as f:
                f.write("3")
            print("🧠 Dropped Linux memory caches directly.")
            cache_cleared = True
        except Exception:
            # Option 3: Inform and continue
            print("📝 Skipping cache drop (requires sudo). Consider adding
this line to sudoers:")
```

```
            print("    yourusername ALL=(ALL) NOPASSWD: /bin/bash -c echo 3
> /proc/sys/vm/drop_caches")

    # Always run garbage collection regardless of cache drop success
    gc.collect()
    print("♻️ Python garbage collection completed.")

    print(f"\n✅ Vision benchmark complete for '{MODEL}'. Results saved in:
{LOG_DIR}")
```

Code Block 2: vision model benchmarking code

Appendix B

The exact prompts and parameters used for the text and vision model benchmarking are provided below, separated by model modality.

Text Model Prompts

```
PROMPTS = [
    "Explain step-by-step how to implement a convolutional neural network in
PyTorch.",
    "What are the economic consequences of inflation on low-income
households?",
    "Write a Python program that parses a JSON file and extracts all unique
keys.",
    "What causes seasons to change on Earth? Explain in detail for a 10-year-
old.",
    "Simulate a conversation between Sherlock Holmes and Alan Turing in
1950."
]
```

Code Block 3: text model prompts

Vision Model Prompts

```
VISION_TASKS = {
    "image_captioning": {
        "prompt": "Describe the content of this image.",
        "image_path": "example_images/f1.jpg"
    },
    "object_detection": {
        "prompt": "What objects are visible in this image?",
        "image_path": "example_images/intersection.jpg"
    },
    "scene_understanding": {
        "prompt": "Describe the scene in this image in detail.",
        "image_path": "example_images/hiking.jpg"
    }
}
```

Code Block 4: vision model prompts

Image 1: "f1.jpg"used for description vision task

Image 2: "intersection.jpg" used for object detection vision task

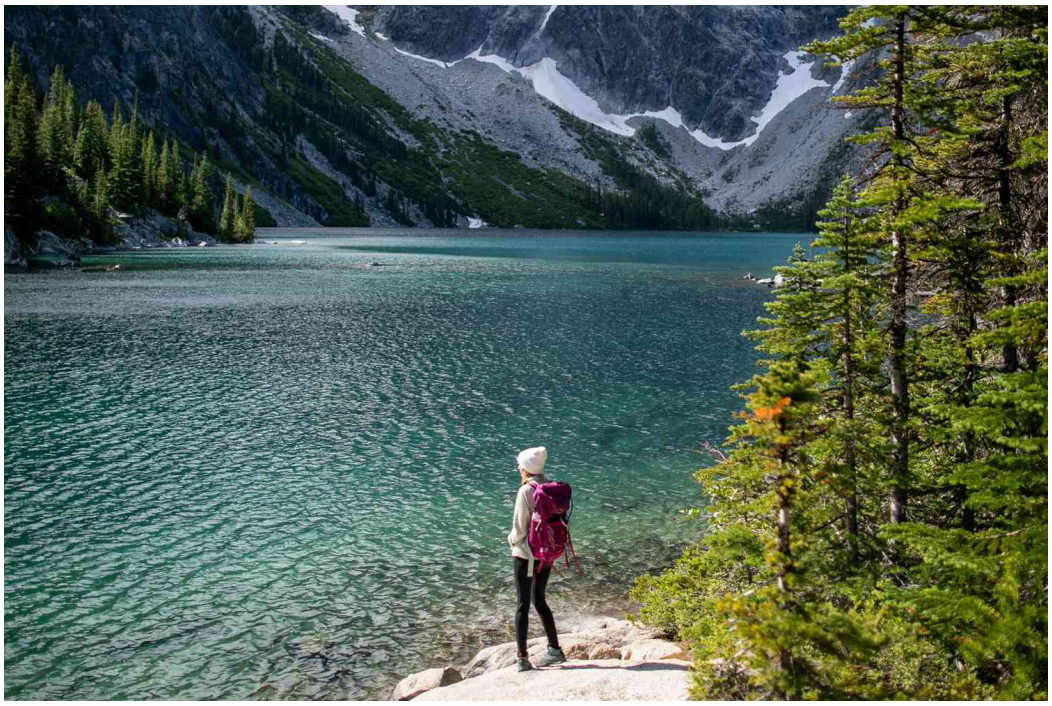
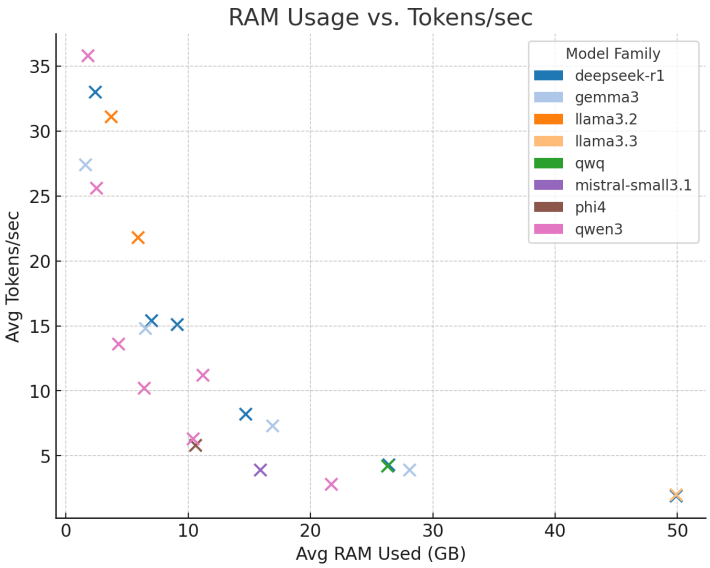


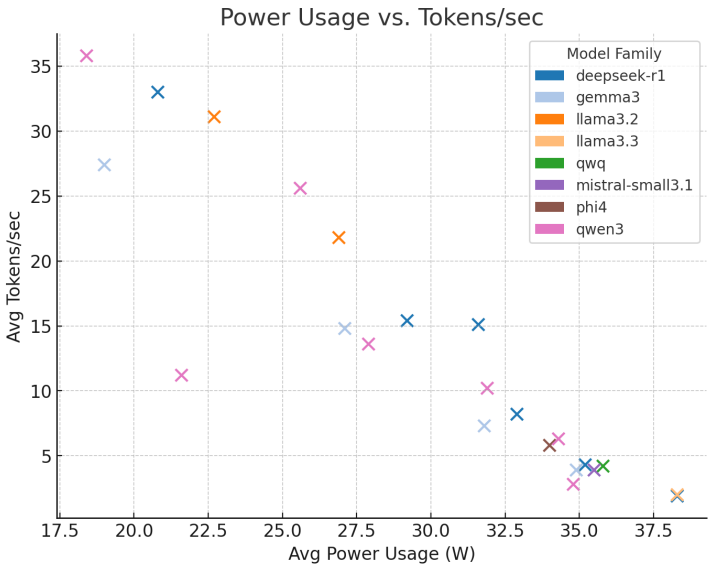
Image 3: "hiking.jpg" used for scene understanding vision task

Appendix C

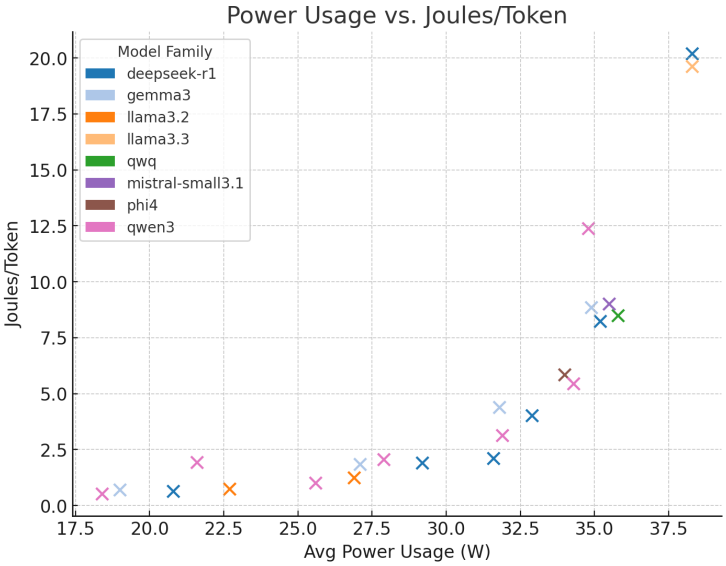
Text Model Plots



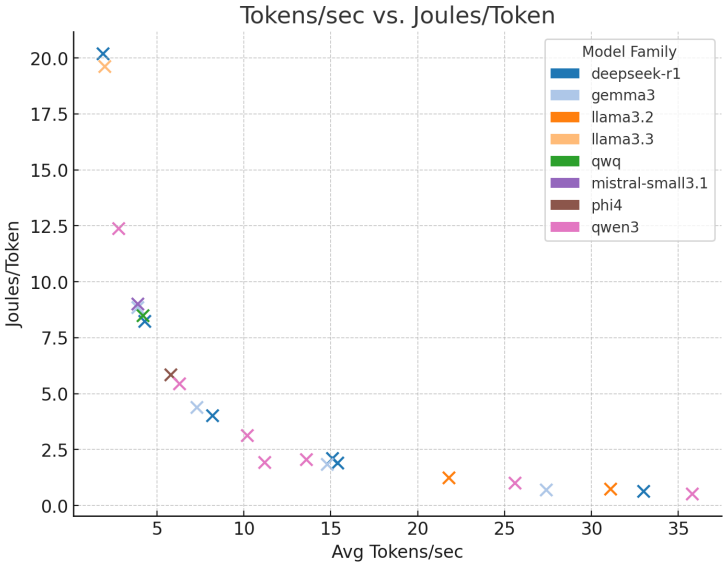
Plot 22: RAM usage vs tokens/second



Plot 23: power usage vs tokens/second

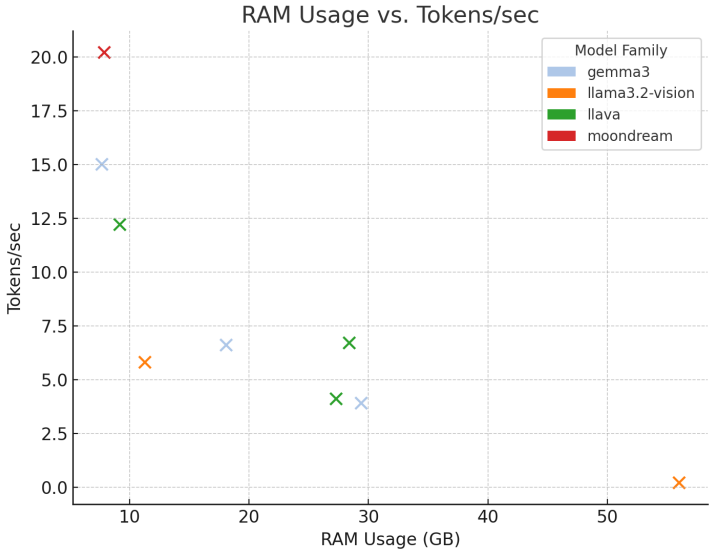


Plot 24: power usage vs joules/token

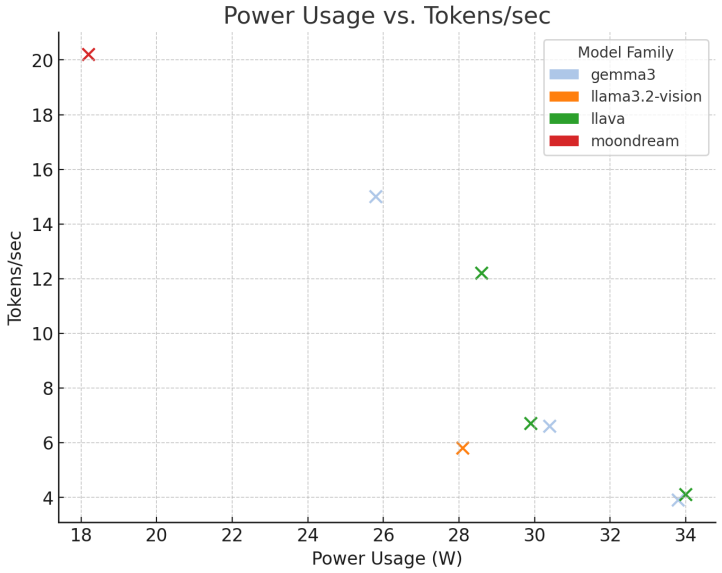


Plot 25: tokens/second vs joules/token

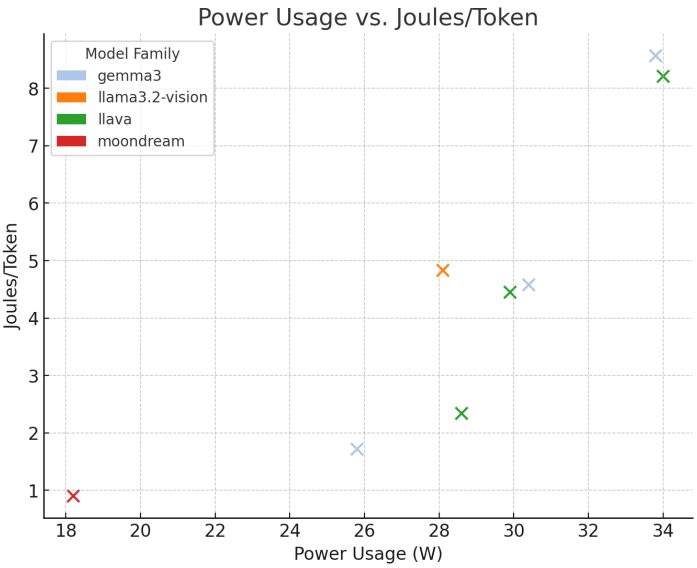
Vision Model Plots



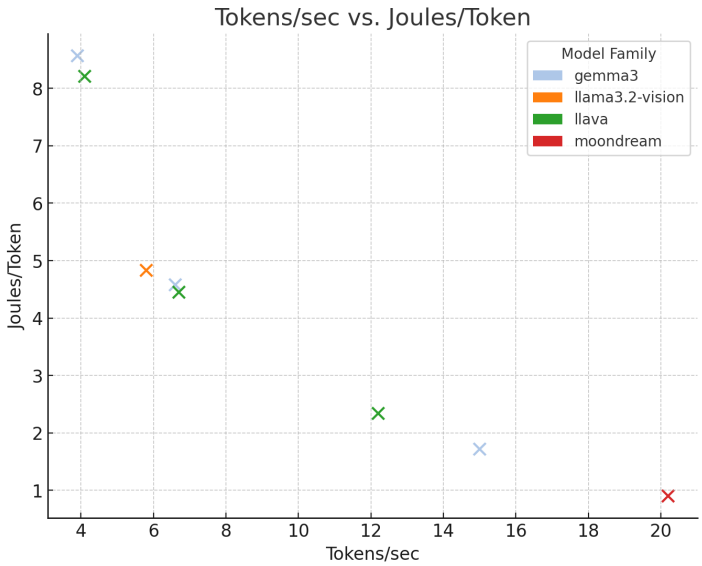
Plot 26: RAM usage vs tokens/second



Plot 27: power usage vs tokens/second



Plot 28: power usage vs joules/token



Plot 29: tokens/second vs joules/token

Appendix D

The code, raw logs, images, and plots are all accessible in this repository: https://github.com/waggle-sensor/llm-notes/tree/main/peterlebiedzinski/llm-benchmarking-01/