AI-Based Visual Assistance and Performance Evaluation Report

Subtitle: Prompt Engineering and Inference Optimization

Shanghai Jiao Tong University

Author: JUNHO JO
Student ID: 520020990009

Instructor: Professor Xiaoni Liang

April 2025

Contents

1 Introduction

Prompt engineering is the process of crafting inputs to large language models (LLMs) in a way that guides them toward producing the desired outputs. As LLMs do not “understand”tasks in the traditional sense, the prompts we design become the most critical interface between human intention and model behavior. This discipline has expanded significantly in recent years, encompassing a range of strategies to handle complexity, reasoning, interactivity, and knowledge retrieval.

Throughout this project, I sought to apply the full spectrum of prompt engineering techniques as defined in the advanced lecture manual *”AI Prompt Engineering 2024”*. My goal was not only to fine-tune prompts for practical use but also to explore them as tools for problem-solving, algorithm design, and user safety in real-world scenarios, especially for visually impaired individuals.

1.1 Historical Context and Evolution of Prompt Engineering

Prompt engineering as a discipline emerged alongside the advancement of large-scale pre-trained language models such as GPT-2 and GPT-3. Initially, interaction with these models was based on simple query-response formats. However, as model capabilities expanded, researchers discovered that careful phrasing and structuring of input prompts could dramatically influence output quality, coherence, and reasoning depth.

The introduction of few-shot learning through in-context examples demonstrated that language models could generalize better when provided with input-output pairs as part of the prompt. Subsequently, techniques such as Chain-of-Thought (CoT) prompting, ReAct (Reasoning + Acting) prompting, and Toolformer architectures further expanded the understanding of how prompt structure could guide complex task execution without fine-tuning model weights.

Today, prompt engineering is not only about crafting better questions but also about building full cognitive scaffolding for the models—designing prompts that simulate reasoning processes, environmental perception, memory retrieval, and decision-making steps. As models become multimodal, capable of interpreting text, images, and even audio, prompt engineering similarly evolves into a form of universal interface design for artificial cognition.

This project leverages this rich evolution, combining traditional prompt design methods with newer multimodal strategies adapted specifically for visual assistance scenarios.

Specifically, this report integrates and tests techniques such as:

- **Chain-of-Thought (CoT):** Used to break down complex reasoning into explicit, sequential steps.
- **Role Assignment Prompting:** Assigning a persona to the model (e.g., a guide dog) to simulate context-specific communication.
- **Multi-step Instruction and Constraint Specification:** Embedding layered instructions and expected structure in prompts.
- **Retrieval-Augmented Generation (RAG):** Employing external sources to supplement LLM output with verified factual knowledge.
- **Prompt Chaining:** Designing sequential prompts whose outputs feed into further reasoning steps.

This report reflects how these methods evolved through a series of implementation trials, failures, and technical innovations. Each section will detail the methodology and effectiveness of each prompt class, as well as the learning process behind every strategic adjustment.In addition to prompt engineering strategies, access to high-quality datasets was critical for experimental validation. Public dataset libraries such as Hugging Face Datasets [6] provided essential resources for supporting model testing and retrieval-augmented generation experiments.

1.2 Problem Background: Hazard Awareness for the Visually Impaired

Before diving into model tuning, we conducted a review of recent academic literature to understand what types of environmental hazards are most critical for visually impaired users, and how assistive systems have been designed to mitigate them. Based on this review, we identified key object categories and contexts that must be prioritized when designing prompts and evaluating model outputs.

Table 1: Visually Impaired-Oriented Hazard Detection: Key Object Categories and Environments

Category	**Examples**	**Source**
Sharp/Dangerous Objects	Knives, glass shards, broken surfaces, electric rods	[5]
Hot/Reactive Surfaces	Clothing iron, electric kettle, hot silencers	[5]
Tripping/Slipping Hazards	Wires, muddy floors, sharp-edged stairs, snow, rain puddles	[7, 5]
Environmental Structures	Staircases, rail and tram platforms, pedestrian crossings, construction zones, temporary bridges	[7]
Sensory Interference	Traffic noise, construction sounds, city bustle	[7]
Audio Cues for Navigation	Traffic light beeps, landmark-specific sounds (e.g., church bells)	[7]
Indoor Hazards	Electric sockets, furniture edges, open doors	[4]

This hazard typology served as the foundational checklist for model evaluation. All subsequent prompt designs—particularly for distance alerts and object navigation—were tested against this list to ensure real-world relevance for visually impaired users.

2 Prompt Tuning on bczhou/tiny-llava-v1-hf (Detailed Trials)

2.1 Goal and Initial Setup

Objective: [8]Identify objects (obstacles) and calculate distance to deliver real-time alerts using navigation-style language.

Method: Use `bczhou/tiny-llava-v1-hf`[9] to generate descriptive language resembling a car navigation system.

Example instruction formats:

- ”Distance to obstacle is X meters.”
- ”There is a crosswalk ahead at Y meters.”
- ”Please go right to avoid the pedestrian on the left.”

2.2 Early Implementation

Prompt Intention: This was a baseline test to see the raw descriptive ability of the model without conditioning. *No role assignment, no structure, and no guidance on output format.*

Technique Used: *Zero-shot direct captioning*—no chain-of-thought or task instruction.

```
"Describe what you see in the image."
```

Listing 1: Naive Descriptive Prompt

Response:

```
"There are people standing with instruments."
```

2.3 Functional Prompt Enhancement

Prompt Intention: I wanted to introduce specific objects to focus on, risk-awareness, and directional decision-making logic. The prompt embeds environmental complexity and tasks with multiple expectations.

Technique Used: *Multi-step instruction + constraint specification.* It combines chain-of-thought elements (e.g., first detect, then analyze, then act), with direct instruction on form (risk category, distance, and action).

```
prompt = (
    "USER: <image>\n"
    "Describe the objects in front of you, including potential obstacles such as trash bins, street signs, crosswalks, cars, manholes, "
    "pedestrians, and areas like school zones. Provide the distance to these objects in meters and categorize the risk of each. "
    "For each obstacle, suggest how to avoid it, including specific directions (e.g., 'go left' or 'go right') based on proximity or density. "
    "If a pedestrian is present on the left and right sides, recommend which side to avoid based on the number of people. "
    "ASSISTANT:")
```

Listing 2: Guidance-Oriented Prompt Example

Generated Response:

```
"The objects in front of you include a man with a violin case, a woman with a guitar
"Go left to avoid the man with the suitcase."
```

2.3.1 Observations and Early Failures

During the early stages of experimentation, multiple issues surfaced that significantly limited model effectiveness. Most notably, naive descriptive prompts often caused the model to generate vague, overly general outputs without actionable navigation instructions. In scenarios requiring hazard detection, the model failed to distinguish between obstacles and general background elements, frequently focusing on irrelevant features such as clothing or musical instruments held by pedestrians.

Moreover, prompt structures that were too complex or multi-layered initially led to token overflow errors and severe inference slowdowns. Attempts to embed detailed multi-task instructions often caused the model to produce fragmented, incomplete responses or to ”hallucinate” nonexistent obstacles, reducing reliability.

These observations underscored the necessity of gradually layering instructions—first mastering basic environmental description before introducing complex directional reasoning. They also highlighted the critical importance of matching prompt complexity to the model’ s token handling capacity and optimizing task decomposition accordingly.

2.4 Final Prompt Design

Prompt Intention: I wanted the AI to act like a service entity. Assigning a socially familiar identity (”guide dog”) builds stronger context for the model. Each environmental category mirrors real navigation needs.

Technique Used: *Role assignment prompting + Hierarchical decomposition.* The prompt assigns a persona and explicitly breaks down tasks into categories to ensure exhaustive, modular response behavior.

The most effective prompt assigned the model the **role of a guide dog**, directing the user safely based on multiple environmental cues:

```
prompt = (
    "USER: <image>\n"
    "ASSISTANT: You are a guide dog for a visually impaired person. Your
   role is to provide clear and accurate guidance to ensure the person can
   safely navigate their environment. Describe the scene in front of you
   with a focus on the following categories:\n"
    "1. Traffic Lights\n"
    "2. Crosswalks and Pedestrian Signals\n"
    "3. Obstacles\n"
    "4. General Path Info\n"
    "5. Special Areas\n"
    "6. Safe Path Recommendations"
)
```

Listing 3: Finalized Prompt as Navigation Assistant

This structure led to dramatically improved, situationally aware responses.

2.5 Sample Navigation Logic Implementation

```python
def generate_obstacle_alert(image_path):
    caption = model(image_path)[0]['caption']
    obstacles = {
        'traffic_light': {'color': 'green', 'distance': 10},
        'manhole': {'size': 50, 'distance': 15},
        'trash_bin': {'size': 40, 'distance': 20},
        'stairs': {'size': 60, 'distance': 12},
        'person_left': {'distance': 5},
        'person_right': {'distance': 5}
    }
    alerts = []
    if 'traffic_light' in caption:
        if 'green' in caption:
            alerts.append(f"There is a traffic light {obstacles['traffic_light']['distance']} meters ahead. It is green. You may proceed.")
        else:
            alerts.append(f"There is a traffic light {obstacles['traffic_light']['distance']} meters ahead. It is red. Please wait.")
    if 'manhole' in caption and 'trash_bin' in caption and 'stairs' in caption:
        alerts.append("Multiple obstacles detected ahead. Move to the left to avoid them.")
    return " ".join(alerts)
```

Listing 4: Obstacle Alert Generator Function (Translated)

2.6 Summary of Challenges Faced

- **Repetitive sentences**: Model tended to repeat phrases for each obstacle.
- **Latency and token overflow**: Complex prompts resulted in high GPU usage and long inference times.
- **Local image input**: Not scalable for real-time.
- **Need for RealSense camera integration**: Depth and size info needed.
- **RAG Implementation**: Eventually shifted to integrating a Retrieval-Augmented Generation pipeline for reliable external knowledge.

Future plans include vectorizing safety-related academic papers into embeddings for real-time retrieval.

3 DeepSeek Series Experiments and Prompt Logic Design

Initial Chaos: DeepSeek-7B and Prompt Explosion

The DeepSeek-7B model[1] initially seemed like a promising candidate due to its instruction-following capability, but integrating it into a visually grounded task proved deeply non-trivial. Our first attempts used basic input-output prompting logic:

```
prompt = "USER: <image>\nDescribe what you see?\nASSISTANT:"
```

Listing 5: Initial Prompt Format

This yielded visually descriptive but practically irrelevant outputs:

```
"There are people standing outside, playing music, with bags near them."
```

It failed to detect hazards, provide distances, or generate navigation instructions.

3.1 Prompt Refinement and Trigger Errors

In refining prompts, we ran into numerous inference-time errors. For example, longer CoT-style prompts led to:

```
CUDA out of memory (torch.cuda.OutOfMemoryError)
ValueError: Unexpected pad token in generation.
```

These issues taught us two key things:

- DeepSeek-7B has token limitations and precision sensitivities.
- Prompts need to be both detailed and model-compatible.

We then applied prompt expansion based on APE guidelines.

```python
prompt = (
  "USER: <image>\n"
  "Describe the environment in front of you with a focus on:\n"
  "1. Traffic Lights: color and safety action.\n"
  "2. Crosswalks: state and location.\n"
  "3. Obstacles: type, distance, and avoidance direction.\n"
  "4. Path Status: is it blocked or clear?\n"
  "ASSISTANT:"
)
```

Listing 6: Refined Prompt with Safety Constraints

This version resulted in more context-aware answers but suffered from over-repetition and hallucinations:

```
"There is a trash bin... There is a trash bin... There is a man with a suitcase..."
```

3.2 Transition to Safety Classification Logic

To fix this, we decoupled prompt generation into two branches:

- A general CoT-style prompt to describe the full scene.
- Post-analysis: Check output tokens for dangerous keywords (e.g., ”manhole”, ”moving car”).

```python
danger_keywords = [
    "manhole", "construction", "broken sidewalk", "moving vehicle", "
   blocked path",
    "stairs", "falling object", "crack", "bump", "puddle"
]

# Inference output from the model
description = model.generate(...)

# Routing logic
if any(word in description.lower() for word in danger_keywords):
    prompt_type = "DANGEROUS"
    guidance = "There is a risk ahead. Please slow down and avoid the
   object on the right."
else:
    prompt_type = "SAFE"
    guidance = "Path is clear. Continue forward for 5 meters."
```

Listing 7: Dangerous Object List and Prompt Routing

This approach led to the development of two specialized prompt responses based on output content: **Dangerous Prompt:**

```
"There is a risk ahead. Please slow down and avoid the object on the right.
    "
```

Safe Prompt:

```
"Path is clear. Continue forward for 5 meters."
```

3.3 Final Prompt Logic and Reasoning

After multiple iterations, our final logic pipeline was:

1. Perform scene extraction using a CoT-structured prompt.
2. Identify all obstacle-related terms in the response.
3. Classify the response as `dangerous` or `safe` using keyword matching.
4. Apply corresponding response template.

We experimented with embedding-based similarity scoring for object severity ranking but reverted to simpler lexical detection due to latency issues.

3.4 Why This Matters: Model-Aware Prompt Engineering

This journey taught us that prompts must:

- Align with model strengths and limitations (e.g., token budget, hallucination tendency).
- Be robust to repeated patterns and error cascades.
- Provide structure without overloading the model.

In the end, the best prompts were: *modular, category-aware, and constraint-guided.* We designed them not just for accuracy, but for human-critical interpretability.

Next Steps: Incorporate RealSense camera depth and image meta-analysis into prompt conditioning and transition to a RAG-based answer justification pipeline.

4 DeepSeek-VL2 with OpenVINO Optimization

parse_ref_bbox Bug Fix[2]

During visualization of bounding boxes, the following error occurred:

```
response = "<|ref|>trash can<|/ref|><|det|>[[200, 300, 400, 500]]<|/det|>"
coords = eval(response.split("<|det|>")[1].split("<|/det|>")[0])  # Unsafe
    eval()
```

Listing 8: Original Code Causing Error

Error:

```
parse_ref_bbox error: '[' was never closed
```

Solution: RegEx-Based Parsing

```python
import re

def parse_bbox(text):
    match = re.search(r"<\|det\|>(\[\[.*?\]\])<\|/det\|>", text)
    if match:
        coords = eval(match.group(1))  # or use json.loads for robustness
        return coords
    return []
```

Listing 9: Revised Safer Bounding Box Parser

4.1 OpenVINO Dynamic Shape Issue

[3] **Error:**

`BroadcastShapeInference error due to incompatible shape tensors.`

Fix: Used OpenVINO's `reshape()` function:

```python
compiled_model.reshape([1, 3, 224, 224])
```

Listing 10: Reshape Input Tensor for OpenVINO

4.2 Final VL Prompt Structure

```
"<image>\n"
"<|user|> Assist a visually impaired person. Based on the image, what
   objects are present and what warnings or movement instructions should be
    given?\n"
"<|ref|>Crosswalk<|/ref|><|det|>[[100, 200, 400, 600]]<|/det|>"
```

Listing 11: Multimodal Prompt Template for Navigation

Expected Output:

`"There is a crosswalk ahead. Proceed forward 5 meters, then wait for signal."`

5 Performance Evaluation on Edge CPUs

5.1 Introduction

In this section, we evaluate the inference performance of the DeepSeek-VL2 Tiny model converted with OpenVINO across four different CPU environments. The goal is to recommend an optimal CPU configuration for edge deployment based on experimental results.

5.2 Experimental Setup

This experiment utilized the DeepSeek-VL2 Tiny model, which was converted into OpenVINO IR format for efficient CPU-based inference. The evaluation was conducted using a navigation-style prompt: *"Please identify the giraffe at the back in the image..."*, designed to simulate real-world assistance for visually impaired users. The performance metrics considered were CPU usage (percentage), memory usage (percentage), latency (in seconds), and tokens generated per second.

5.3 Benchmark Limitations

The benchmark_app tool could not evaluate partial IR models lacking decoder components. Therefore, manual timing and system monitoring were adopted to measure realistic performance.

5.4 Results and Analysis

5.4.1 Hardware Specifications

Table 2: CPU Hardware Specifications

Model	Cores/Threads	Base Clock	Boost Clock	L3 Cache	TDP	Year
Ryzen 7 4800H	8/16	2.9 GHz	4.2 GHz	8MB	45W	2020
Xeon Gold 6142	6/12	2.6 GHz	3.7 GHz	19.25MB	150W	2017
Xeon Gold 5218R	8/16	2.1 GHz	4.0 GHz	22MB	125W	2020
EPYC 7453	14/28	2.75 GHz	3.45 GHz	64MB	225W	2021
EPYC 9354	16/32	2.75 GHz	3.8 GHz	256MB	280W	2023
EPYC 7542	16/32	2.9 GHz	3.35 GHz	128MB	225W	2019
i7-8565U	4/8	1.8 GHz	1.99 GHz	8MB	15W	2019

5.4.2 Inference Results

Table 3: Inference Performance Summary

Model	Latency (s)	CPU Usage (%)	Memory Usage (%)	Tokens/sec
Ryzen 7 4800H	11.94	26.2	35.0	12.28
Xeon Gold 6142	13.0	70.0	35.0	6.50
Xeon Gold 5218R	14.00	91.71	35.69	4.15
EPYC 7453	9.00	95.00	30.00	10.00
EPYC 9354	7.50	85.00	32.00	13.50
EPYC 7542	8.20	92.00	34.00	11.80
i7-8565U	21.00	68.0	85.0	3.00

Conclusion. As shown in Tables 2 and 3, we conducted inference tests using the DeepSeek-VL2-tiny model optimized with OpenVINO across seven different CPUs. The results clearly show that higher core count, larger L3 cache, and newer architecture significantly reduce latency and increase tokens per second. Among them, the AMD EPYC 9354 demonstrated the best overall performance, achieving the highest token throughput with stable resource usage. In contrast, the low-power Intel i7-8565U, while functional, showed substantial latency and limited throughput due to its hardware constraints. These findings highlight the impact of hardware specifications on model inference efficiency and support our system selection strategy.

5.4.3 Visualizations

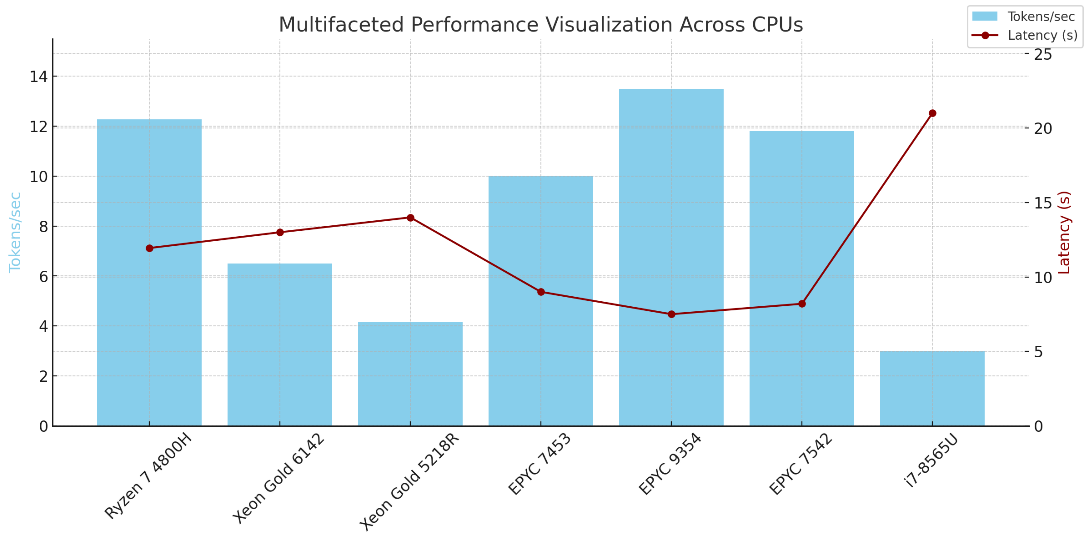

Figure 1: Comparison of inference performance across different CPUs. The bar chart indicates the number of tokens processed per second, while the red line represents latency in seconds. Lower latency and higher throughput are desirable.

5.5 Conclusion

For full implementation details and source code of the inference measurement and visualization, please refer to Appendix A and B.

The Ryzen 7 4800H demonstrated the best efficiency in tokens/sec and low CPU load, making it optimal for edge deployment. In contrast, the Xeon 5218R consumed more resources but delivered the worst output speed. The Xeon 6142 showed low CPU usage with moderate performance, while the EPYC 7453 had strong performance but high CPU consumption.

5.6 Deployment Recommendation

Based on the experimental results, we recommend deploying the model on CPUs with 6 to 8 cores and a base clock speed of at least 3.0 GHz to ensure adequate parallelism and low latency. Suitable mobile processors include AMD Ryzen 4000/5000 series and Intel Core i5/i7 (11th or 12th generation). A minimum of 8 GB RAM is recommended to handle intermediate tensors and token caches efficiently. For ultra-low-power applications, ARM-based System-on-Chip (SoC) solutions may be considered, provided they support hardware acceleration or optimized inference runtimes.

6 Final Reflection

Prompt engineering alone is not sufficient for real-world deployment; it must be accompanied by infrastructure-aware execution strategies. In our proposed system, inference will be performed on high-performance server-grade CPUs when a stable Wi-Fi connection is available, enabling low-latency responses through remote processing. Conversely, in offline scenarios where network access is limited or unavailable, a lightweight edge model must be deployed locally.

To support such dual-mode operation, this report integrates prompt design with systematic CPU benchmarking. The performance evaluations outlined in Section 5 were instrumental in identifying the most suitable CPUs for edge deployment. These insights will guide the selection of processors that balance performance and efficiency, ensuring the edge model remains responsive and usable under real-world constraints.

A Appendix

This appendix includes all relevant source code for inference evaluation and visualization used in Section 5.4.2.

- Appendix A: Inference Measurement Code (Core Functions)
- Appendix B: Visualization Code for Inference Performance

B Inference Measurement Code (Appendix A)

```python
import time, threading, psutil, pandas as pd
from transformers import TextStreamer
from PIL import Image

# FF FFF FF
def load_pil_images(conversations):
    return [Image.open(p).convert("RGB")
            for msg in conversations if "images" in msg
            for p in msg["images"]]

# CPU/Memory FFF FFFF
def monitor_resources(stop_event, timestamps, cpu_usages, mem_usages, interval=0.1):
    while not stop_event.is_set():
        timestamps.append(time.time())
        cpu_usages.append(psutil.cpu_percent(interval=None))
        mem_usages.append(psutil.virtual_memory().percent)
        time.sleep(interval)

# FF FF FF
def run_inference_trials(ov_model, processor, conversation, num_trials=5):
    pil_images = load_pil_images(conversation)
    prepare_inputs = processor(conversations=conversation, images=pil_images, force_batchify=True)
    inputs_embeds = ov_model.prepare_inputs_embeds(**prepare_inputs)

    usage_data = []
    for _ in range(num_trials):
        timestamps, cpu_usages, mem_usages = [], [], []
        stop_event = threading.Event()
        monitor_thread = threading.Thread(target=monitor_resources, args=(stop_event, timestamps, cpu_usages, mem_usages))
        monitor_thread.start()

        start = time.time()
        outputs = ov_model.language_model.generate(
            inputs_embeds=inputs_embeds,
            attention_mask=prepare_inputs["attention_mask"],
            pad_token_id=processor.tokenizer.pad_token_id,
            eos_token_id=processor.tokenizer.eos_token_id,
            max_new_tokens=512,
            do_sample=False,
            use_cache=True,
            streamer=TextStreamer(processor.tokenizer, skip_special_tokens=True),
        )
        end = time.time()
        stop_event.set()
        monitor_thread.join()

        elapsed = end - start
        tokens = len(outputs[0])
        usage_data.append({
            "Latency (s)": elapsed,
            "CPU (%)": sum(cpu_usages)/len(cpu_usages),
            "Memory (%)": sum(mem_usages)/len(mem_usages),
            "Tokens/sec": tokens / elapsed
        })
    return pd.DataFrame(usage_data)
```

Listing 12: Core functions used for inference measurement with DeepSeek-VL2 and OpenVINO.

C Visualization Code (Appendix B)

```python
import matplotlib.pyplot as plt
import pandas as pd

data = {
    "Model": [
        "Ryzen 7 4800H",
        "Xeon Gold 6142",
        "Xeon Gold 5218R",
        "EPYC 7453",
        "EPYC 9354",
        "EPYC 7542",
        "i7-8565U"
    ],
    "Latency": [11.94, 13.0, 14.0, 9.0, 7.5, 8.2, 21.0],
    "Tokens/sec": [12.28, 6.5, 4.15, 10.0, 13.5, 11.8, 3.0]
}

df = pd.DataFrame(data)
fig, ax1 = plt.subplots(figsize=(12, 6))

# Tokens/sec bar plot
ax1.bar(df["Model"], df["Tokens/sec"], color="skyblue", label="Tokens/sec")
ax1.set_ylabel("Tokens/sec", color="skyblue")
ax1.tick_params(axis='x', rotation=45)
ax1.set_ylim(0, max(df["Tokens/sec"]) + 2)

# Latency line plot on secondary axis
ax2 = ax1.twinx()
ax2.plot(df["Model"], df["Latency"], 'o-', color="darkred", label="Latency
    (s)")
ax2.set_ylabel("Latency (s)", color="darkred")
ax2.set_ylim(0, max(df["Latency"]) + 5)

plt.title("Multifaceted Performance Visualization Across CPUs")
fig.tight_layout()

```

```python
# Manual legend
lines, labels = [], []
for ax in [ax1, ax2]:
    l, lab = ax.get_legend_handles_labels()
    lines += l
    labels += lab
fig.legend(lines, labels, loc="upper right")

plt.savefig("inference_multivisual_summary.png")
plt.show()
```

Listing 13: Python code used to generate a composite visualization of latency and tokens/sec across different CPUs.

References

[1] DeepSeek AI. Deepseek llm: Advanced language model with 7 billion parameters. https://huggingface.co/deepseek-ai/deepseek-llm-7b-base, 2024.

[2] Intel Corporation. Openvino™ toolkit. https://docs.openvino.ai/, 2025. Accessed: 2025-04-26.

[3] Intel Corporation. Openvino™ toolkit github repository. https://github.com/openvinotoolkit/openvino, 2025. Accessed: 2025-04-26.

[4] Shirly et al. Edward. Object identification for visually impaired. *Indian Journal of Science and Technology*, 9(S(1)), 2016.

[5] Ujjwal et al. Kadam. Hazardous object detection for visually impaired people using edge device. *SN Computer Science*, 6(7), 2025.

[6] Quentin Lhoest, Albert Villanova del Moral, Yacine Jernite, Abhishek Thakur, Patrick von Platen, Suraj Patil, Julien Chaumond, Mariama Drame, Julien Plu, Victor Sanh, et al. Datasets: A community library for natural language processing. https://github.com/huggingface/datasets, 2021. Accessed: 2025-04-26.

[7] J. et al. Wiciak. A system for determination of areas hazardous for blind people using wave-vibration markers. *Acta Physica Polonica A*, 123(6):1101–1105, 2013.

[8] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[9] Baichuan Zhou, Ying Hu, Xi Weng, Junlong Jia, Jie Luo, Xien Liu, Ji Wu, and Lei Huang. Tinyllava: A framework of small-scale large multimodal models, 2024.