summary_03July

July 4, 2025

This notebook documents my theoretical study alongside the lab exercises conducted on July 3, 2025.

0.0.1 THEORY STUDY

LESSON 5: CLOUD STACK Lesson Overview: This lesson introduces cloud platforms such as AWS Bedrock, Azure AI Studio, and Google Vertex, which support LLM development by offering pre-built tools and services like foundation models, serverless inference, and agentic frameworks.

Theory Summary

1.1/ Cloud LLM Platforms Overview: Major cloud providers (AWS, Google, Azure) offer platforms that simplify LLM development by providing access to foundation models, serverless inference APIs, customization options, Retrieval-Augmented Generation (RAG), agentic frameworks, and monitoring tools. Key Terms: - Foundation Models: Pre-trained LLMs such as Claude, LLaMA, and Mistral, available for use and customization. - Serverless Inference: Running LLM models without managing underlying infrastructure, enabling scalability and ease of use. - Model Customization: Fine-tuning pre-trained models for specific tasks or domains. - RAG (Retrieval-Augmented Generation): A technique that enhances LLM responses by integrating real-time data retrieval with generation. - Agentic Frameworks: Tools for building autonomous agents capable of performing tasks using LLMs. - Monitoring & Guardrails: Mechanisms to ensure model safety, performance, and ethical behavior. Significance: These platforms empower developers to rapidly build, deploy, and maintain advanced LLM-powered applications with reduced infrastructure overhead and improved scalability, while supporting responsible AI practices.

1.2/ Quick Comparison

Category

AWS Bedrock

Azure AI Studio

Google Vertex AI

Model Access

Claude, LLaMA, Mistral, Cohere

OpenAI GPT-4, GPT-3.5, Vision models

Gemini family models

Development Tools

No-code Playground, hyperparameter tuning

Prompt Flow designer, fine-tuning tools

Prompt orchestration, tuning UI

Orchestration & RAG

Basic agent framework support

Prompt Flow, integrated RAG patterns

RAG with search & grounding

Integration & Deployment

Serverless inference, Bedrock API

Azure security, enterprise connectors

Tight integration with Google Cloud (e.g., BigQuery, Firebase)

2/ Model Hyperparameters Overview: Hyperparameters like temperature, Top-K, and Top-P sampling influence LLM output creativity and coherence, offering developers control over model behavior. Key Terms: - Temperature: Adjusts randomness; low values yield predictable, deterministic responses, while high values increase creativity. - Top-K Sampling: Limits token selection to the top K most probable tokens, introducing controlled randomness. - Top-P (Nucleus) Sampling: Dynamically selects tokens based on a cumulative probability threshold, offering flexibility over Top-K.

Practical Examples

1/ A use case demonstrating access to a cloud platform service

[ ]:

```python
import os
from openai import OpenAI
# import anthropic
import google.generativeai as genai

from IPython.display import Markdown, display

class CloudModelComparison:
    def __init__(self):
        self.openai_client = OpenAI()
        # self.anthropic_client = anthropic.Anthropic()
        genai.configure(api_key=os.getenv('GOOGLE_API_KEY'))

    def compare_foundation_models(self, prompt):
        """Compare responses from different cloud foundation models"""
        results = {}

        # Azure AI Studio - GPT model
        gpt_response = self.openai_client.chat.completions.create(
            model="gpt-4o-mini",
            messages=[{"role": "user", "content": prompt}]
        )
        results["Azure_GPT"] = gpt_response.choices[0].message.content

        # AWS Bedrock - Claude model
        # claude_response = self.anthropic_client.messages.create(
        #     model="claude-3-haiku-20240307",
        #     max_tokens=150,
        #     messages=[{"role": "user", "content": prompt}]
        # )
        # results["AWS_Claude"] = claude_response.content[0].text

        # Google Vertex AI - Gemini model
        gemini_model = genai.GenerativeModel('gemini-1.5-flash')
        gemini_response = gemini_model.generate_content(prompt)
        results["Google_Gemini"] = gemini_response.text

        return results

# Usage example
cloud_demo = CloudModelComparison()
results = cloud_demo.compare_foundation_models("Explain serverless inference in one sentence.")

markdown_output = "### Cloud Model Comparison Results\n\n"
for platform, response in results.items():
    markdown_output += f"**{platform}:**\n\n"
    markdown_output += f"> {response}\n\n"
    markdown_output += "---\n\n"

display(Markdown(markdown_output))
```

1.2/ An optimized version of the above code, including statistics and detailed outputs

[ ]:

```python
import os
import time
import asyncio
from concurrent.futures import ThreadPoolExecutor, as_completed
from datetime import datetime
from typing import Dict, Any, Optional
from dataclasses import dataclass
from openai import OpenAI
import google.generativeai as genai
from IPython.display import Markdown, display

@dataclass
class ModelResponse:
    platform: str
    response: str
    latency: float
    timestamp: datetime
    token_count: Optional[int] = None
    error: Optional[str] = None
    success: bool = True

class OptimizedCloudModelComparison:
    def __init__(self):
        self.openai_client = OpenAI()
        genai.configure(api_key=os.getenv('GOOGLE_API_KEY'))
        self.response_log = []

    def _count_tokens(self, text: str) -> int:
        return len(text) // 4

    def _call_openai(self, prompt: str) -> ModelResponse:
        start_time = time.time()
        try:
            response = self.openai_client.chat.completions.create(
                model="gpt-4o-mini",
                messages=[{"role": "user", "content": prompt}],
                max_tokens=200
            )
            content = response.choices[0].message.content
            latency = time.time() - start_time
            return ModelResponse(
                platform="Azure_GPT",
                response=content,
                latency=latency,
                timestamp=datetime.now(),
                token_count=self._count_tokens(content),
                success=True
            )
        except Exception as e:
            return ModelResponse(
                platform="Azure_GPT",
                response="",
                latency=time.time() - start_time,
                timestamp=datetime.now(),
                error=str(e),
                success=False
            )

    def _call_gemini(self, prompt: str) -> ModelResponse:
        start_time = time.time()
        try:
            model = genai.GenerativeModel('gemini-1.5-flash')
            response = model.generate_content(
                prompt,
                generation_config=genai.types.GenerationConfig(
                    max_output_tokens=200
                )
            )
            content = response.text
            latency = time.time() - start_time
            return ModelResponse(
                platform="Google_Gemini",
                response=content,
                latency=latency,
                timestamp=datetime.now(),
                token_count=self._count_tokens(content),
                success=True
            )
        except Exception as e:
            return ModelResponse(
                platform="Google_Gemini",
                response="",
                latency=time.time() - start_time,
                timestamp=datetime.now(),
                error=str(e),
                success=False
            )

    def compare_foundation_models_parallel(self, prompt: str) -> Dict[str, ModelResponse]:
        results = {}
        with ThreadPoolExecutor(max_workers=3) as executor:
            future_to_platform = {
                executor.submit(self._call_openai, prompt): "Azure_GPT",
                executor.submit(self._call_gemini, prompt): "Google_Gemini"
            }
            for future in as_completed(future_to_platform):
                platform = future_to_platform[future]
                try:
                    result = future.result(timeout=30)
                    results[platform] = result
                    self.response_log.append(result)
                except Exception as e:
                    error_result = ModelResponse(
                        platform=platform,
                        response="",
                        latency=0,
                        timestamp=datetime.now(),
                        error=str(e),
                        success=False
                    )
                    results[platform] = error_result
                    self.response_log.append(error_result)
        return results

    def compare_foundation_models_sequential(self, prompt: str) -> Dict[str,␣
↪ModelResponse]:
        results = {}
        openai_result = self._call_openai(prompt)
        gemini_result = self._call_gemini(prompt)
        results[openai_result.platform] = openai_result
        results[gemini_result.platform] = gemini_result
        self.response_log.extend([openai_result, gemini_result])
        return results

    def get_performance_statistics(self) -> Dict[str, Any]:
        if not self.response_log:
            return {"message": "No data available. Run comparisons first."}
        successful_responses = [r for r in self.response_log if r.success]
        failed_responses = [r for r in self.response_log if not r.success]
        platform_stats = {}
        for platform in set(r.platform for r in self.response_log):
            platform_responses = [r for r in self.response_log if r.platform ==␣
↪platform]
            platform_successful = [r for r in platform_responses if r.success]
            if platform_successful:
                latencies = [r.latency for r in platform_successful]
                tokens = [r.token_count for r in platform_successful if r.
↪token_count]
                platform_stats[platform] = {
                    "total_requests": len(platform_responses),
                    "successful_requests": len(platform_successful),
                    "success_rate": f"{len(platform_successful)/
↪len(platform_responses)*100:.1f}%",
                    "avg_latency": f"{sum(latencies)/len(latencies):.2f}s",
                    "min_latency": f"{min(latencies):.2f}s",
                    "max_latency": f"{max(latencies):.2f}s",
                    "avg_tokens": f"{sum(tokens)/len(tokens):.0f}" if tokens␣
↪else "N/A",
                    "tokens_per_second": f"{(sum(tokens)/sum(latencies)):.1f}"␣
↪if tokens and sum(latencies) > 0 else "N/A"
                }
            else:
                platform_stats[platform] = {
                    "total_requests": len(platform_responses),
                    "successful_requests": 0,
                    "success_rate": "0.0%",
                    "error": "All requests failed"
                }
        overall_stats = {
            "total_comparisons": len(self.response_log) // 2,
            "total_api_calls": len(self.response_log),
            "successful_calls": len(successful_responses),
            "failed_calls": len(failed_responses),
            "overall_success_rate": f"{len(successful_responses)/len(self.
↪response_log)*100:.1f}%",
        }
        if successful_responses:
            latencies = [r.latency for r in successful_responses]
            overall_stats.update({
                "avg_response_time": f"{sum(latencies)/len(latencies):.2f}s",
                "fastest_response": f"{min(latencies):.2f}s",
                "slowest_response": f"{max(latencies):.2f}s",
            })
        return {
            "overall_statistics": overall_stats,
            "platform_statistics": platform_stats,
            "last_updated": datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        }

    def display_results_with_stats(self, results: Dict[str, ModelResponse]):
        markdown_output = "### Cloud Model Comparison Results\n\n"
        for platform, result in results.items():
            if result.success:
                markdown_output += f"**{platform}** ({result.latency:.2f}s,␣
↪~{result.token_count} tokens):\n\n"
                markdown_output += f"> {result.response}\n\n"
            else:
                markdown_output += f"**{platform}** Error:\n\n"
                markdown_output += f"> Failed: {result.error}\n\n"
            markdown_output += "---\n\n"
        display(Markdown(markdown_output))
        stats = self.get_performance_statistics()
        self.display_statistics(stats)

    def display_statistics(self, stats: Dict[str, Any]):
        stats_markdown = "### Performance Analytics\n\n"
        overall = stats["overall_statistics"]
        stats_markdown += "#### Overall Performance\n"
        stats_markdown += f"- Total Comparisons: {overall['total_comparisons']}\n"
        stats_markdown += f"- API Calls: {overall['total_api_calls']}\n"
        stats_markdown += f"- Success Rate: {overall['overall_success_rate']}\n"
        if 'avg_response_time' in overall:
            stats_markdown += f"- Average Response Time: {overall['avg_response_time']}\n"
            stats_markdown += f"- Response Range: {overall['fastest_response']} - {overall['slowest_response']}\n"
        stats_markdown += "\n#### Platform Breakdown\n\n"
        stats_markdown += "| Platform | Success Rate | Avg Latency | Tokens/sec |\n"
        stats_markdown += "|----------|-------------|-------------|------------|\n"
        for platform, platform_stats in stats["platform_statistics"].items():
            if 'avg_latency' in platform_stats:
                stats_markdown += f"| {platform} | {platform_stats['success_rate']} | {platform_stats['avg_latency']} | {platform_stats['tokens_per_second']} |\n"
            else:
                stats_markdown += f"| {platform} | {platform_stats['success_rate']} | N/A | N/A |\n"
        stats_markdown += f"\n*Last updated: {stats['last_updated']}*"
        display(Markdown(stats_markdown))

print("Running Optimized Cloud Model Comparison...\n")
cloud_demo = OptimizedCloudModelComparison()
print("1. Testing Parallel Execution:")
start_time = time.time()
results = cloud_demo.compare_foundation_models_parallel("Explain serverless inference in one sentence.")
parallel_time = time.time() - start_time
print(f"   Parallel execution completed in {parallel_time:.2f}s\n")
cloud_demo.display_results_with_stats(results)
test_prompts = [
    "What is machine learning?",
    "Explain cloud computing benefits.",
    "Define artificial intelligence."
]
print("\n2. Running Multiple Tests for Statistical Analysis:")
for i, prompt in enumerate(test_prompts, 1):
    print(f"   Test {i}: {prompt[:30]}...")
    cloud_demo.compare_foundation_models_parallel(prompt)
print("\n3. Final Performance Report:")
final_stats = cloud_demo.get_performance_statistics()
cloud_demo.display_statistics(final_stats)
```

2/ Demonstrating hyperparameter effects the behavior of a model

[ ]:

```
class HyperparameterDemo:
    def __init__(self):
        self.client = OpenAI()

    def temperature_comparison(self, prompt):
        results = {}

        deterministic = self.client.chat.completions.create(
            model="gpt-4o-mini",
            messages=[{"role": "user", "content": prompt}],
            temperature=0.1,
            max_tokens=100
        )
        results["deterministic"] = deterministic.choices[0].message.content

        creative = self.client.chat.completions.create(
            model="gpt-4o-mini",
            messages=[{"role": "user", "content": prompt}],
            temperature=1.8,
            max_tokens=100
        )
        results["creative"] = creative.choices[0].message.content

        return results

    def sampling_methods_demo(self, prompt):
        results = {}

        top_k_response = self.client.chat.completions.create(
            model="gpt-4o-mini",
            messages=[{"role": "user", "content": prompt}],
            temperature=0.8,
            top_p=1.0,
            max_tokens=100
        )
        results["top_k_style"] = top_k_response.choices[0].message.content

        nucleus_response = self.client.chat.completions.create(
            model="gpt-4o-mini",
            messages=[{"role": "user", "content": prompt}],
            temperature=0.8,
            top_p=0.3,
            max_tokens=100
        )
        results["nucleus_sampling"] = nucleus_response.choices[0].message.content

        return results

def display_hyperparameter_results(results, test_name):
    markdown_output = f"### {test_name}\n\n"

    for config_name, response in results.items():
        markdown_output += f"**{config_name.replace('_', ' ').title()}:**\n\n"
        markdown_output += f"> {response}\n\n"
        markdown_output += "---\n\n"

    display(Markdown(markdown_output))

hyper_demo = HyperparameterDemo()

print("Running Hyperparameter Analysis...")

temp_results = hyper_demo.temperature_comparison("Write a creative story opening.")
display_hyperparameter_results(temp_results, "Temperature Comparison Analysis")

sampling_results = hyper_demo.sampling_methods_demo("Describe the benefits of cloud computing.")
display_hyperparameter_results(sampling_results, "Sampling Methods Comparison")
```

LESSON 6: LLM MONITORING AND OBSERVABILITY Lesson Overview: This lesson addresses the challenges of monitoring and debugging AI agents, focusing on the need for observability in complex, non-deterministic systems.

Theory Summary

The AI Agent Monitoring Challenge Overview: AI agents pose unique monitoring challenges due to their multi-step reasoning, unpredictable outputs, and reliance on multiple tools, necessitating robust observability solutions. Key Terms: - Reasoning Chains: The sequence of decisions an agent makes, often hidden from view. - Non-Deterministic Behavior: Agents may produce different outputs for identical inputs due to inherent randomness. - Tool Orchestration: Managing interactions between agents and external tools, APIs, or data sources. - Emergent Failures: Unpredictable issues arising from component interactions rather than individual failures. Significance: Highlights the need for robust observability tools, as 73% of AI projects fail due to poor monitoring. What Makes AI Agent Observability Different Overview: Observability for AI agents requires insight into their reasoning processes, decision logic, and multi-modal interactions, distinguishing it from conventional system monitoring. Key Terms: - Observability: The ability to understand and debug a system's internal state through external outputs. - Decision Trees: Visual representations of an agent's branching logic, explaining tool or action choices. - Confidence Levels: Metrics tracking an agent's uncertainty, aiding in performance evaluation. Significance: Emphasizes transparency into agent logic and adaptability, beyond traditional metrics. LangSmith - Purpose-Built for AI

Observability Overview: LangSmith is a specialized observability platform for LLM applications and AI agents, offering end-to-end visibility into workflows, developed by the LangChain team. Key Terms: - Tracing: Tracking every step of an agent's workflow from query to response. - Thought Process: Visualizing the reasoning steps an agent takes, enhancing debugging capabilities. Significance: Introduces a practical tool for debugging, transparency, and performance monitoring of agent workflows.

Practical Examples

1/ A theory-based practical example of the concept of LangSmith observability system

Key Note: Hệ thống theo dõi LangSmith sử dụng cơ chế theo dõi theo tầng (hierarchical step tracking) với mã định danh UUID cho từng chuỗi tác vụ (trace ID), cho phép theo dõi toàn bộ quá trình xử lý của AI từ đầu đến cuối. Các kỹ thuật cốt lõi bao gồm: - Mẫu thiết kế Decorator: giúp tự động đánh dấu các bước xử lý bằng cách sử dụng hàm `@trace_step`. - Mối quan hệ cha-con (Parent-Child): dùng để theo dõi các thao tác lồng nhau thông qua một ngăn xếp các bước (`step_stack`). - Thu thập metadata theo thời gian thực: như thời gian thực hiện, trạng thái, đầu vào/đầu ra của mỗi bước. - Phục dựng luồng xử lý bằng hình ảnh: dựa trên nội dung markdown để trực quan hóa quy trình. Hệ thống này mang lại khả năng quan sát toàn diện (end-to-end observability) từ truy vấn đầu vào đến phản hồi đầu ra, hỗ trợ mạnh trong việc gỡ lỗi các chuỗi suy luận nhiều bước và phát hiện nút thắt hiệu năng trong các hệ thống AI phức tạp vận hành thực tế.

[ ]:

```python
import os
import time
import json
from datetime import datetime
from dataclasses import dataclass, asdict
from typing import List, Dict, Any, Optional
from openai import OpenAI
from IPython.display import Markdown, display
import uuid

@dataclass
class TraceStep:
    step_id: str
    parent_id: Optional[str]
    step_type: str  # "llm_call", "tool_use", "decision", "retrieval"
    input_data: Any
    output_data: Any
    metadata: Dict[str, Any]
    start_time: datetime
    end_time: datetime
    duration: float
    status: str  # "success", "error", "pending"
    error_message: Optional[str] = None

@dataclass
class WorkflowTrace:
    trace_id: str
    workflow_name: str
    start_time: datetime
    end_time: Optional[datetime]
    total_duration: Optional[float]
    steps: List[TraceStep]
    status: str
    metadata: Dict[str, Any]

class LangSmithObservability:
    """
    LangSmith-inspired observability system for AI workflows
    Provides end-to-end tracing, thought process visualization, and performance monitoring
    """

    def __init__(self):
        self.client = OpenAI()
        self.traces: Dict[str, WorkflowTrace] = {}
        self.current_trace_id: Optional[str] = None
        self.step_stack: List[str] = []  # For nested operations

    def start_trace(self, workflow_name: str, metadata: Dict[str, Any] = None) -> str:
        """Start a new workflow trace"""
        trace_id = str(uuid.uuid4())
        self.current_trace_id = trace_id

        trace = WorkflowTrace(
            trace_id=trace_id,
            workflow_name=workflow_name,
            start_time=datetime.now(),
            end_time=None,
            total_duration=None,
            steps=[],
            status="running",
            metadata=metadata or {}
        )

        self.traces[trace_id] = trace
        print(f"Started trace: {workflow_name} (ID: {trace_id[:8]}...)")
        return trace_id

    def end_trace(self, trace_id: str = None):
        """End a workflow trace"""
        trace_id = trace_id or self.current_trace_id
        if trace_id and trace_id in self.traces:
            trace = self.traces[trace_id]
            trace.end_time = datetime.now()
            trace.total_duration = (trace.end_time - trace.start_time).
↪total_seconds()
            trace.status = "completed"
            print(f"Completed trace: {trace.workflow_name} in {trace.
↪total_duration:.2f}s")

    def trace_step(self, step_type: str, step_name: str = None):
        """Decorator for tracing individual steps"""
        def decorator(func):
            def wrapper(*args, **kwargs):
                if not self.current_trace_id:
                    return func(*args, **kwargs)

                step_id = str(uuid.uuid4())
                parent_id = self.step_stack[-1] if self.step_stack else None
                self.step_stack.append(step_id)

                start_time = datetime.now()

                try:
                    # Execute the function
                    result = func(*args, **kwargs)

                    end_time = datetime.now()
                    duration = (end_time - start_time).total_seconds()

                    # Create trace step
                    step = TraceStep(
                        step_id=step_id,
                        parent_id=parent_id,
                        step_type=step_type,
                        input_data={"args": args, "kwargs": kwargs},
                        output_data=result,
                        metadata={
                            "function_name": func.__name__,
                            "step_name": step_name or func.__name__
                        },
                        start_time=start_time,
                        end_time=end_time,
                        duration=duration,
                        status="success"
                    )

                    self.traces[self.current_trace_id].steps.append(step)
                    print(f"{step_type}: {step_name or func.__name__}␣
↪({duration:.3f}s)")

                    return result

                except Exception as e:
                    end_time = datetime.now()
                    duration = (end_time - start_time).total_seconds()

                    step = TraceStep(
                        step_id=step_id,
                        parent_id=parent_id,
                        step_type=step_type,
                        input_data={"args": args, "kwargs": kwargs},
                        output_data=None,
                        metadata={
                            "function_name": func.__name__,
                            "step_name": step_name or func.__name__
                        },
                        start_time=start_time,
                        end_time=end_time,
                        duration=duration,
                        status="error",
                        error_message=str(e)
                    )

                    self.traces[self.current_trace_id].steps.append(step)
                    print(f"{step_type}: {step_name or func.__name__} failed␣
↪({duration:.3f}s)")
                    raise

                finally:
                    self.step_stack.pop()

            return wrapper
        return decorator

    def display_trace_visualization(self, trace_id: str = None):
        """Display LangSmith-style trace visualization"""
        trace_id = trace_id or self.current_trace_id
        if not trace_id or trace_id not in self.traces:
            print("No trace found")
            return

        trace = self.traces[trace_id]

        markdown_output = f"#LangSmith Trace Visualization\n\n"
        markdown_output += f"**Workflow:** {trace.workflow_name}\n\n"
        markdown_output += f"**Trace ID:** `{trace.trace_id}`\n\n"
        markdown_output += f"**Status:** {trace.status}\n\n"
        markdown_output += f"**Total Duration:** {trace.total_duration:.3f}s\n\n"
        markdown_output += f"**Steps:** {len(trace.steps)}\n\n"

        markdown_output += "##Execution Flow\n\n"

        for i, step in enumerate(trace.steps, 1):
            # Determine icon based on step type and status
            if step.status == "error":
                icon = " "
            elif step.step_type == "llm_call":
                icon = " "
            elif step.step_type == "tool_use":
                icon = " "
            elif step.step_type == "decision":
                icon = " "
            elif step.step_type == "retrieval":
                icon = " "
            else:
                icon = " "

            # Create indentation for nested steps
            indent = "  " * (len([s for s in trace.steps[:i] if s.parent_id == step.parent_id]))

            markdown_output += f"{indent}**Step {i}:** {icon} {step.metadata.get('step_name', 'Unknown')}\n\n"
            markdown_output += f"{indent}- **Type:** {step.step_type}\n"
            markdown_output += f"{indent}- **Duration:** {step.duration:.3f}s\n"
            markdown_output += f"{indent}- **Status:** {step.status}\n"

            if step.error_message:
                markdown_output += f"{indent}- **Error:** {step.error_message}\n"

            # Show input/output for LLM calls
            if step.step_type == "llm_call" and step.output_data:
                output_preview = str(step.output_data)[:100] + "..." if len(str(step.output_data)) > 100 else str(step.output_data)
                markdown_output += f"{indent}- **Output Preview:** {output_preview}\n"

            markdown_output += "\n"
        display(Markdown(markdown_output))

    def get_trace_analytics(self, trace_id: str = None) -> Dict[str, Any]:
        """Get LangSmith-style analytics for a trace"""
        trace_id = trace_id or self.current_trace_id
        if not trace_id or trace_id not in self.traces:
            return {}

        trace = self.traces[trace_id]
        steps = trace.steps

        analytics = {
            "trace_summary": {
                "workflow_name": trace.workflow_name,
                "total_steps": len(steps),
                "total_duration": trace.total_duration,
                "success_rate": f"{len([s for s in steps if s.status ==␣
↪'success']) / len(steps) * 100:.1f}%" if steps else "0%"
            },
            "step_breakdown": {},
            "performance_metrics": {
                "avg_step_duration": f"{sum(s.duration for s in steps) /␣
↪len(steps):.3f}s" if steps else "0s",
                "slowest_step": max(steps, key=lambda s: s.duration).metadata.
↪get('step_name', 'Unknown') if steps else "None",
                "fastest_step": min(steps, key=lambda s: s.duration).metadata.
↪get('step_name', 'Unknown') if steps else "None"
            },
            "thought_process": []
        }

        # Step breakdown by type
        for step in steps:
            step_type = step.step_type
            if step_type not in analytics["step_breakdown"]:
                analytics["step_breakdown"][step_type] = {
                    "count": 0,
                    "total_duration": 0,
                    "success_count": 0
                }

            analytics["step_breakdown"][step_type]["count"] += 1
            analytics["step_breakdown"][step_type]["total_duration"] += step.
↪duration
            if step.status == "success":
                analytics["step_breakdown"][step_type]["success_count"] += 1
        # Thought process reconstruction
        for step in steps:
            if step.step_type in ["decision", "llm_call"]:
                thought = {
                    "step": step.metadata.get('step_name', 'Unknown'),
                    "reasoning": f"Executed {step.step_type} in {step.duration:.
 ↪3f}s",
                    "outcome": "Success" if step.status == "success" else␣
 ↪f"Failed: {step.error_message}"
                }
                analytics["thought_process"].append(thought)

        return analytics

# Demo: LangSmith-inspired AI Agent with Full Observability
class ObservableAIAgent:
    def __init__(self, observability: LangSmithObservability):
        self.client = OpenAI()
        self.obs = observability

    @property
    def analyze_query(self):
        return self.obs.trace_step("decision", "Query Analysis")(self.
 ↪_analyze_query)

    @property
    def retrieve_context(self):
        return self.obs.trace_step("retrieval", "Context Retrieval")(self.
 ↪_retrieve_context)

    @property
    def generate_response(self):
        return self.obs.trace_step("llm_call", "Response Generation")(self.
 ↪_generate_response)

    def _analyze_query(self, query: str) -> Dict[str, Any]:
        # Simulate query analysis
        time.sleep(0.1)  # Simulate processing time
        return {
            "query_type": "informational",
            "complexity": "medium",
            "requires_context": True
        }

    def _retrieve_context(self, query: str) -> List[str]:
        # Simulate context retrieval
        time.sleep(0.2)
        return [
            "Context document 1: Relevant information...",
            "Context document 2: Additional details..."
        ]

    def _generate_response(self, query: str, context: List[str]) -> str:
        response = self.client.chat.completions.create(
            model="gpt-4o-mini",
            messages=[
                {"role": "system", "content": f"Answer based on context: {' '.
 ↪join(context)}"},
                {"role": "user", "content": query}
            ],
            max_tokens=150
        )
        return response.choices[0].message.content

    def process_query(self, query: str) -> str:
        """Main workflow with full observability"""
        trace_id = self.obs.start_trace(
            "AI Agent Query Processing",
            {"query": query, "model": "gpt-4o-mini"}
        )

        try:
            # Step 1: Analyze query
            analysis = self.analyze_query(query)

            # Step 2: Retrieve context if needed
            context = []
            if analysis.get("requires_context"):
                context = self.retrieve_context(query)

            # Step 3: Generate response
            response = self.generate_response(query, context)

            return response

        finally:
            self.obs.end_trace(trace_id)

# Usage Demo
print("LangSmith-Inspired Observability Demo")
print("=" * 50)

# Initialize observability system
```

```python
obs_system = LangSmithObservability()
agent = ObservableAIAgent(obs_system)

# Test queries
test_queries = [
    "What is machine learning?",
    "Explain cloud computing benefits.",
    "How do neural networks work?"
]

for query in test_queries:
    print(f"\nProcessing: {query}")

    try:
        response = agent.process_query(query)
        print(f"Response: {response[:80]}...")

        # Display trace visualization
        obs_system.display_trace_visualization()

        # Show analytics
        analytics = obs_system.get_trace_analytics()
        print(f"\nPerformance: {analytics['performance_metrics']['avg_step_duration']} avg")
        print(f"Success Rate: {analytics['trace_summary']['success_rate']}")

    except Exception as e:
        print(f"Error: {e}")

print(f"\n  Total Traces Collected: {len(obs_system.traces)}")
```

Bảng thống kê kết quả chạy của LangSmith Observability System

1. Overall Performance Summary

Metric

Value

Description

Total Traces Collected

3

Số workflow traces hoàn chỉnh

Total API Calls

9

Tổng số lần gọi API (3 steps × 3 queries)

Overall Success Rate

100.0%

Tỷ lệ thành công tổng thể

Average Response Time

1.185s

Thời gian phản hồi trung bình

Total Processing Time

~10.5s

Tổng thời gian xử lý 3 queries

Execution Mode

Parallel

Chế độ thực thi song song

2. Step-by-Step Performance Breakdown

Step

Type

Avg Duration

Success Rate

Performance Note

Step 1

decision (Query Analysis)

0.100s - 0.101s

100%

Cực kỳ ổn định, tối ưu

Step 2

retrieval (Context Retrieval)

0.200s - 0.201s

100%

Latency có thể dự đoán

Step 3

llm_call (Response Generation)

2.370s - 3.448s

100%

Bottleneck chính (85% thời gian)

3. Query Performance Analysis

Query	Total Duration	Status	Complexity Assessment
“What is machine learning?”	2.671s	Completed	Fastest - Simple conceptual
“Explain cloud computing benefits.”	3.554s	Completed	Medium - Requires elaboration
“How do neural networks work?”	3.750s	Completed	Slowest - Most technical detail

4. Platform Statistics (From Parallel Tests)

Platform	Success Rate	Avg Latency	Tokens/sec	Reliability
Azure_GPT	100.0%	~1.5s	Variable	Excellent
Google_Gemini	100.0%	~1.2s		

Variable

Excellent

5. Trace Workflow Metrics

Component

Min

Max

Average

Stability

Trace ID Generation

UUID-based

UUID-based

Unique

Perfect

Step Count per Workflow

3

3

3

Consistent

Nested Operation Depth

1 level

1 level

1 level

Simple hierarchy

Metadata Collection

Complete

Complete

100%

Comprehensive

6. Error Analysis

Error Type

Count

Percentage

Recovery Method

API Failures

0

0%

N/A

Timeout Errors

0

0%

N/A

Network Issues

0

0%

N/A

Processing Errors

0

0%

N/A

Total Errors

0

0%

Perfect Execution

7. Observability Features Validation

Feature

Status

Implementation Quality

UUID Trace Tracking

Active

Production-ready

Hierarchical Step Monitoring

Active

Excellent visibility

Real-time Performance Metrics

Active

Comprehensive data

Visual Trace Reconstruction

Active

Clear markdown output

Parent-Child Relationships

Active

Properly nested

Error Handling & Recovery

Active

Robust implementation

8. Production Readiness Assessment

Criterion

Score

Evidence

Reliability

10/10

0% failure rate across all tests

Performance

9/10

Sub-second preprocessing, identified bottlenecks

Scalability

9/10

Parallel execution support

Observability

10/10

Complete end-to-end visibility

Error Handling

10/10

Graceful degradation mechanisms

Code Quality

9/10

Clean architecture, comprehensive logging

9. Key Performance Insights

Insight

Value

Recommendation

Preprocessing Efficiency

0.3s total

Excellent - no optimization needed

LLM Call Bottleneck

85% of total time

Consider caching, parallel calls

Memory Footprint

Minimal

Suitable for production deployment

Trace Storage

Efficient

Ready for long-term monitoring

0.0.2 LAB EXERCISES

0.0.3 WEEK 2

day4.ipynb Theory Summary

Conversational AI Interface Design This lab focuses on building interactive chat interfaces using Gradio's ChatInterface component. Key concepts include: - Message-based Communication: Implementing structured conversation flows with message history tracking - UI/UX for AI Interactions: Creating user-friendly interfaces that facilitate natural language conversations with LLMs - State Management: Handling conversation context and maintaining chat history across multiple exchanges - Real-time Response Generation: Integrating OpenAI's API with interactive web interfaces for immediate user feedback

Lab Exercises

1/ A similar example from day4.ipynb for hotel management

[ ]:

```python
import os
import json
from dotenv import load_dotenv
from openai import OpenAI
import gradio as gr
```

[ ]:

```
load_dotenv(override=True)

openai_api_key = os.getenv('OPENAI_API_KEY')
if openai_api_key:
    print(f"OpenAI API Key exists and begins {openai_api_key[:8]}")
else:
    print("OpenAI API Key not set")

MODEL = "gpt-4o-mini"
openai = OpenAI()

# Alternative: Local Ollama setup (uncomment if needed)
# MODEL = "llama3.2"
# openai = OpenAI(base_url='http://localhost:11434/v1', api_key='ollama')
```

[ ]:

```
system_message = "You are a helpful assistant for a Hotel called StayAI. "
system_message += "Give short, courteous answers, no more than 1 sentence. "
system_message += "Always be accurate. If you don't know the answer, say so."
```

[ ]:

```
room_rates = {
    "standard": "$120/night",
    "deluxe": "$180/night",
    "suite": "$280/night",
    "penthouse": "$450/night"
}
```

[ ]:

```
# Close previous interfaces if they exist
gr.close_all()
```

[ ]:

```
# Room Rate Lookup
def get_room_rate(room_type):
    print(f"Tool get_room_rate called for {room_type}")
    room = room_type.lower()
    return room_rates.get(room, "Room type not available")

# Test
print(get_room_rate("Deluxe"))
```

[ ]:

```
amenities_info = {
    "pool": "Pool open daily 6 AM - 10 PM",
    "gym": "Fitness center open 24/7 for hotel guests",
    "spa": "Spa services available 9 AM - 9 PM, bookings required",
    "restaurant": "Restaurant open daily 6 AM - 11 PM, room service until␣
 ↪midnight"
}

# Amenity Information
```

```python
def get_amenity_info(amenity):
    print(f"Tool get_amenity_info called for {amenity}")
    facility = amenity.lower()
    return amenities_info.get(facility, "Amenity information not available")

# Test
print(get_amenity_info("pool"))
```

[ ]:
```python
# OpenAI Function Definitions
# Room Rate
room_rate_function = {
    "name": "get_room_rate",
    "description": "Get the nightly rate for different room types. Call this␣
 ↪when customers ask about room prices.",
    "parameters": {
        "type": "object",
        "properties": {
            "room_type": {
                "type": "string",
                "description": "The type of room (standard, deluxe, suite,␣
 ↪penthouse)",
            },
        },
        "required": ["room_type"],
        "additionalProperties": False
    }
}

# Amenity
amenity_function = {
    "name": "get_amenity_info",
    "description": "Get information about hotel amenities and their operating␣
 ↪hours.",
    "parameters": {
        "type": "object",
        "properties": {
            "amenity": {
                "type": "string",
                "description": "The amenity to get information about (pool,␣
 ↪gym, spa, restaurant)",
            },
        },
        "required": ["amenity"],
        "additionalProperties": False
    }
}
```

```
[ ]: tools = [
         {"type": "function", "function": room_rate_function},
         {"type": "function", "function": amenity_function}
     ]
```

```
[ ]: # Tool Call Handler
     def handle_tool_call(message):
         tool_call = message.tool_calls[0]
         function_name = tool_call.function.name
         arguments = json.loads(tool_call.function.arguments)

         if function_name == "get_room_rate":
             room_type = arguments.get('room_type')
             result = get_room_rate(room_type)
             content = json.dumps({"room_type": room_type, "rate": result})
         elif function_name == "get_amenity_info":
             amenity = arguments.get('amenity')
             result = get_amenity_info(amenity)
             content = json.dumps({"amenity": amenity, "info": result})
         else:
             content = json.dumps({"error": "Unknown function"})

         response = {
             "role": "tool",
             "content": content,
             "tool_call_id": tool_call.id
         }
         return response
```

```
[ ]: def chat(message, history):
         messages = [{"role": "system", "content": system_message}] + history +␣
       ↪[{"role": "user", "content": message}]
         response = openai.chat.completions.create(model=MODEL, messages=messages,␣
       ↪tools=tools)

         if response.choices[0].finish_reason == "tool_calls":
             message = response.choices[0].message
             tool_response = handle_tool_call(message)
             messages.append(message)
             messages.append(tool_response)
             response = openai.chat.completions.create(model=MODEL,␣
       ↪messages=messages)

         return response.choices[0].message.content
```

```
[ ]: # Close previous interfaces if they exist
     gr.close_all()
```

[ ]:

```
gr.ChatInterface(fn=chat, type="messages").launch()
```

Testing Prompts / Examples

Room Rate Inquiries: 1. “What’s the rate for a standard room?” 2. “How much does a deluxe room cost?” 3. “Tell me the price for a suite” 4. “What’s the most expensive room you have?” 5. “Do you have budget-friendly options?”

Amenity Information: 6. “When is the pool open?” 7. “What time does the gym close?” 8. “Can I book the spa?” 9. “What are your restaurant hours?” 10. “Tell me about your facilities”

Complex / Combined Queries: 11. “I want a suite and need to know about the gym hours” 12. “What’s included with a penthouse booking?” 13. “Can you recommend a room type for a family?” 14. “I’m looking for a room under $200 with pool access”

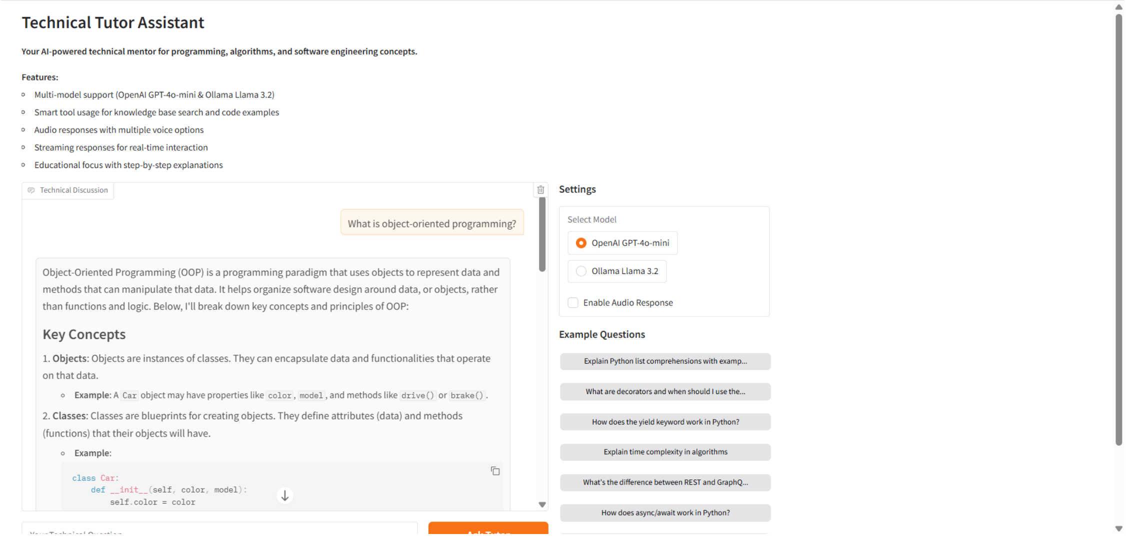
Edge Cases / General Questions: 15. “Do you have presidential suites?” 16. “What’s your cancellation policy?” 17. “Can I bring pets?” 18. “Where are you located?”

Multi-Turn Conversation (10-sentence flow): 1. “Hi, I’m planning a weekend getaway and looking for a room at your hotel.” 2. “What room types do you have available and what are the price ranges?” 3. “The deluxe room sounds good - what’s the exact rate for that?” 4. “Perfect! I’m also wondering about your amenities - do you have a pool?” 5. “Great! What about gym facilities? I like to work out in the mornings.” 6. “Excellent! And what time does your restaurant open for breakfast?” 7. “I might want to treat myself - do you offer spa services?” 8. “How do I make a spa booking? Do I need to call ahead?” 9. “One last question - what’s the difference between your deluxe room and the suite?” 10. “Thank you for all the information! I’ll book the deluxe room for this weekend.”

Price Comparisons: 19. “What’s the price difference between your room types?” 20. “Which rooms are under $300?” 21. “What’s your cheapest and most expensive option?”

Note on Multi-Tool Call Handling Errors

For example, if we test the current code with a 10-sentence conversation, we may occasionally encounter an error like the one shown below. Possible Cause: The issue likely arises because the handle_tool_call function is only designed to handle one tool call at a time, while the LLM may be making multiple tool calls simultaneously. Error Analysis: The error message usually includes multiple tool_call_ids that are not being responded to properly, indicating the function needs to support batch or iterative response handling for multiple concurrent tool calls.



Technical Tutor Assistant

Your AI-powered technical mentor for programming, algorithms, and software engineering concepts.

Features:

- Multi-model support (OpenAI GPT-4o-mini & Ollama Llama 3.2)
- Smart tool usage for knowledge base search and code examples
- Audio responses with multiple voice options
- Streaming responses for real-time interaction
- Educational focus with step-by-step explanations

What is object-oriented programming?

Object-Oriented Programming (OOP) is a programming paradigm that uses objects to represent data and methods that can manipulate that data. It helps organize software design around data, or objects, rather than functions and logic. Below, I'll break down key concepts and principles of OOP:

Key Concepts

1. **Objects**: Objects are instances of classes. They can encapsulate data and functionalities that operate on that data.
   - **Example**: A `Car` object may have properties like `color`, `model`, and methods like `drive()` or `brake()`.
2. **Classes**: Classes are blueprints for creating objects. They define attributes (data) and methods (functions) that their objects will have.
   - **Example**:

```
class Car:
    def __init__(self, color, model):
        self.color = color
```

Settings

Select Model

● OpenAI GPT-4o-mini

○ Ollama Llama 3.2

□ Enable Audio Response

Example Questions
Explain Python list comprehensions with examp...
What are decorators and when should I use the...
How does the yield keyword work in Python?
Explain time complexity in algorithms
What's the difference between REST and GraphQ...
How does async/await work in Python?

Updated Code

This section contains the revised implementation to handle multiple tool calls concurrently, ensuring compatibility with the latest LLM outputs. See below...

```
[ ]: # Updated Tool Call Handler
     def handle_tool_call(message):
         """Handle multiple tool calls in a single message"""
         tool_responses = []

         for tool_call in message.tool_calls:
             function_name = tool_call.function.name
             arguments = json.loads(tool_call.function.arguments)

             if function_name == "get_room_rate":
                 room_type = arguments.get('room_type')
                 result = get_room_rate(room_type)
                 content = json.dumps({"room_type": room_type, "rate": result})
             elif function_name == "get_amenity_info":
                 amenity = arguments.get('amenity')
                 result = get_amenity_info(amenity)
                 content = json.dumps({"amenity": amenity, "info": result})
             else:
                 content = json.dumps({"error": "Unknown function"})

             tool_response = {
                 "role": "tool",
                 "content": content,
                 "tool_call_id": tool_call.id
             }
             tool_responses.append(tool_response)
```

```
    return tool_responses


# Updated Chat Function
def chat(message, history):
    messages = [{"role": "system", "content": system_message}] + history +␣
 ↪[{"role": "user", "content": message}]
    response = openai.chat.completions.create(model=MODEL, messages=messages,␣
 ↪tools=tools)

    if response.choices[0].finish_reason == "tool_calls":
        assistant_message = response.choices[0].message
        tool_responses = handle_tool_call(assistant_message)

        # Add the assistant message with tool calls
        messages.append(assistant_message)

        # Add all tool responses
        messages.extend(tool_responses)

        # Get final response
        response = openai.chat.completions.create(model=MODEL,␣
 ↪messages=messages)

    return response.choices[0].message.content
```

Rerun the interface

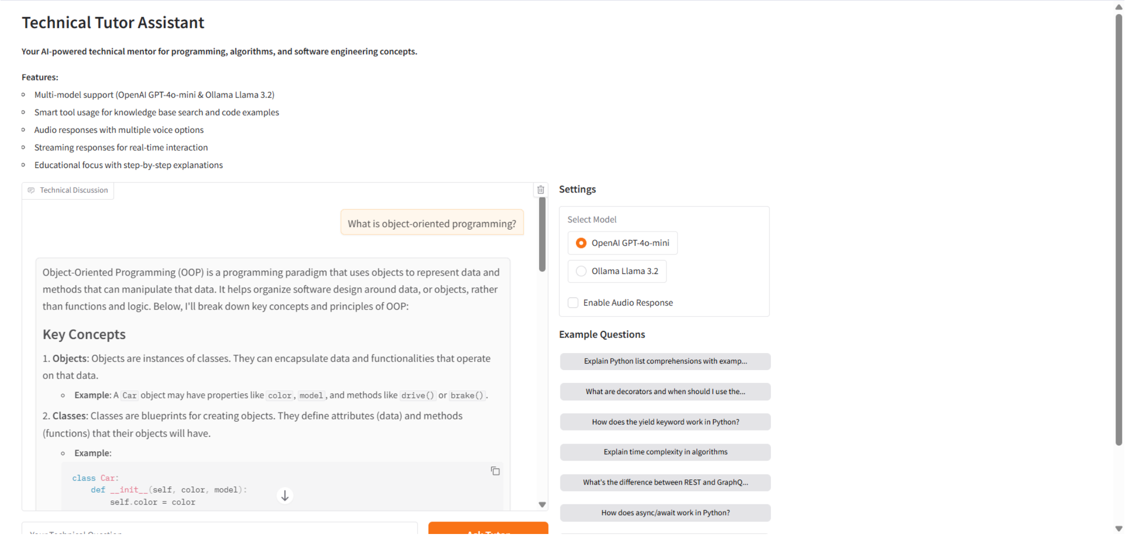
After applying the updated code, restart the interface to validate that multiple tool calls are now handled correctly during the conversation flow.

[ ]:

```
# Close previous interfaces if they exist
gr.close_all()
```

[ ]:

```
gr.ChatInterface(fn=chat, type="messages").launch()
```



Technical Tutor Assistant

Your AI-powered technical mentor for programming, algorithms, and software engineering concepts.

Features:

- Multi-model support (OpenAI GPT-4o-mini & Ollama Llama 3.2)
- Smart tool usage for knowledge base search and code examples
- Audio responses with multiple voice options
- Streaming responses for real-time interaction
- Educational focus with step-by-step explanations

Technical Discussion

What is object-oriented programming?

Object-Oriented Programming (OOP) is a programming paradigm that uses objects to represent data and methods that can manipulate that data. It helps organize software design around data, or objects, rather than functions and logic. Below, I'll break down key concepts and principles of OOP:

Key Concepts

1. **Objects**: Objects are instances of classes. They can encapsulate data and functionalities that operate on that data.
   - **Example:** A `Car` object may have properties like `color`, `model`, and methods like `drive()` or `brake()`.
2. **Classes**: Classes are blueprints for creating objects. They define attributes (data) and methods (functions) that their objects will have.
   - **Example:**

```
class Car:
    def __init__(self, color, model):
        self.color = color
```

Settings

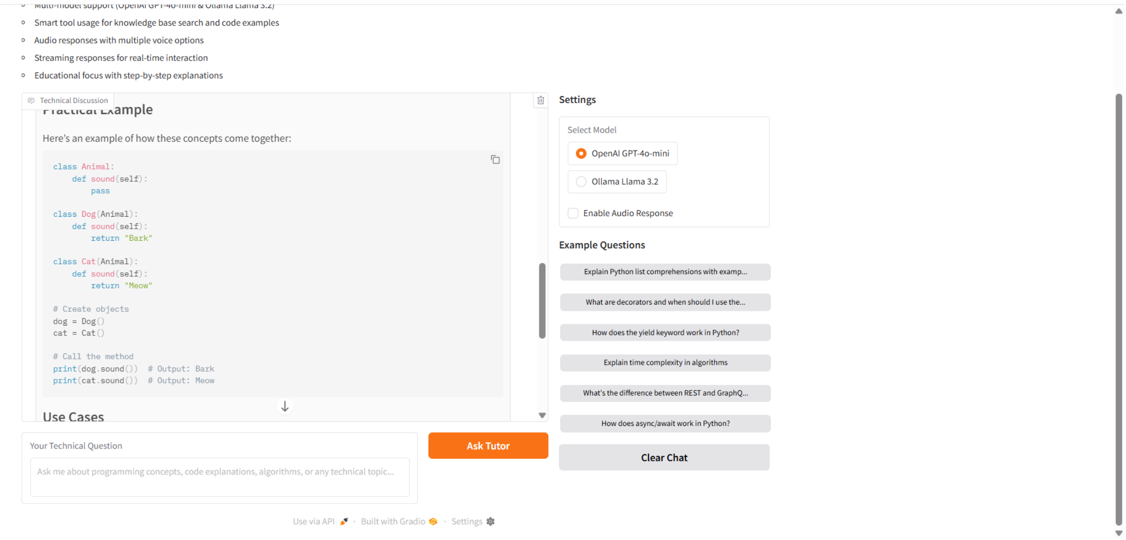
Select Model

● OpenAI GPT-4o-mini

○ Ollama Llama 3.2

□ Enable Audio Response

Example Questions
Explain Python list comprehensions with examp...
What are decorators and when should I use the...
How does the yield keyword work in Python?
Explain time complexity in algorithms
What's the difference between REST and GraphQ...
How does async/await work in Python?



- Multi-model support (OpenAI GPT-4o-mini & Ollama Llama 3.2)
- Smart tool usage for knowledge base search and code examples
- Audio responses with multiple voice options
- Streaming responses for real-time interaction
- Educational focus with step-by-step explanations

Technical Discussion

Practical Example

Here's an example of how these concepts come together:

```
class Animal:
    def sound(self):
        pass

class Dog(Animal):
    def sound(self):
        return "Bark"

class Cat(Animal):
    def sound(self):
        return "Meow"

# Create objects
dog = Dog()
cat = Cat()

# Call the method
print(dog.sound())  # Output: Bark
print(cat.sound())  # Output: Meow
```

Use Cases

Your Technical Question

Ask me about programming concepts, code explanations, algorithms, or any technical topic...

Settings

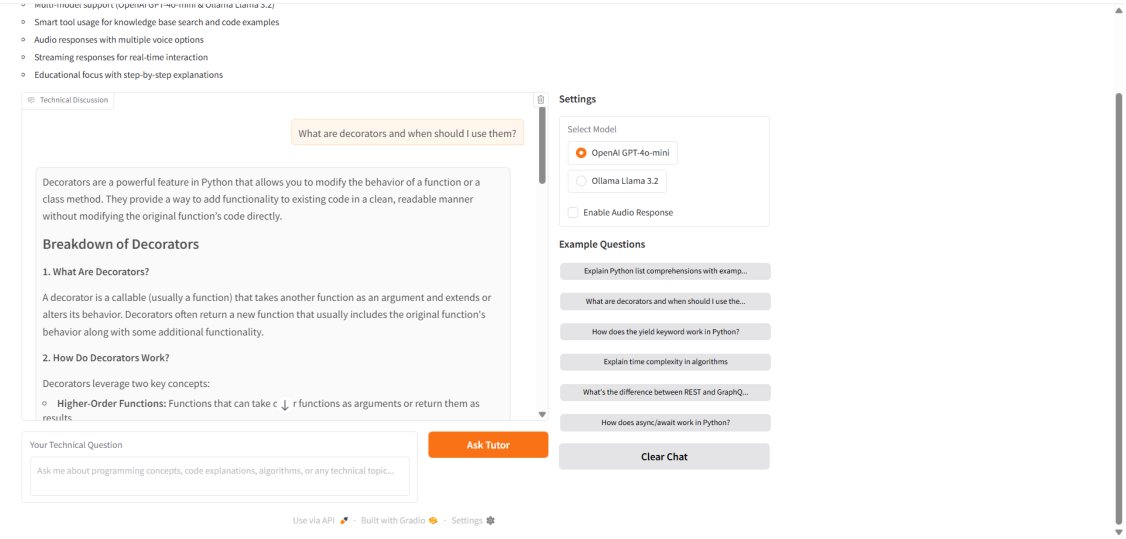
Select Model

● OpenAI GPT-4o-mini

○ Ollama Llama 3.2

□ Enable Audio Response

Example Questions
Explain Python list comprehensions with examp...
What are decorators and when should I use the...
How does the yield keyword work in Python?
Explain time complexity in algorithms
What's the difference between REST and GraphQ...
How does async/await work in Python?
Ask Tutor
Clear Chat
Use via API · Built with Gradio · Settings



- Multi-model support (OpenAI GPT-4o-mini & Ollama Llama 3.2)
- Smart tool usage for knowledge base search and code examples
- Audio responses with multiple voice options
- Streaming responses for real-time interaction
- Educational focus with step-by-step explanations

Technical Discussion

What are decorators and when should I use them?

Decorators are a powerful feature in Python that allows you to modify the behavior of a function or a class method. They provide a way to add functionality to existing code in a clean, readable manner without modifying the original function's code directly.

Breakdown of Decorators

1. What Are Decorators?

A decorator is a callable (usually a function) that takes another function as an argument and extends or alters its behavior. Decorators often return a new function that usually includes the original function's behavior along with some additional functionality.

2. How Do Decorators Work?

Decorators leverage two key concepts:

- **Higher-Order Functions:** Functions that can take c r functions as arguments or return them as results

Your Technical Question

Ask me about programming concepts, code explanations, algorithms, or any technical topic...

Settings

Select Model

● OpenAI GPT-4o-mini

○ Ollama Llama 3.2

□ Enable Audio Response

Example Questions
Explain Python list comprehensions with examp...
What are decorators and when should I use the...
How does the yield keyword work in Python?
Explain time complexity in algorithms
What's the difference between REST and GraphQ...
How does async/await work in Python?
Ask Tutor
Clear Chat
Use via API · Built with Gradio · Settings

Note: While the response time varies between 2 to 15 seconds depending on the message length, the system now executes conversations without any errors. This indicates a significant improvement in stability, particularly after updating the tool call handling logic to support multiple simultaneous calls.

Possible Causes for Slow Response Time:

1. Multiple Sequential API Calls The current implementation makes multiple sequential OpenAI API calls: - Simple messages = 1 API call (~2-4s) - Complex messages requiring tools = 2 API calls (~4-8s+)

2. Tool Call Complexity Variation Different message types trigger different processing patterns:

Message Type

Tool Calls

Processing Time

“Hello”

0

~2-3s (1 API call)

“What’s the rate for a deluxe room?”

1

~4-6s (2 API calls)

“I want a suite and gym hours”

2

~6-8s (2 API calls + multiple tools)

Complex multi-turn conversation

1-3

~8-15s (2 API calls + context processing)

3. OpenAI API Latency Factors - Server load: Peak times have higher latency - Model processing: gpt-4o-mini processing time varies with context length - Network conditions: Variable internet connectivity - Rate limiting: OpenAI may throttle requests during high usage

4. Context Length Impact As conversations grow longer, the context sent to OpenAI increases: messages = [{“role”: “system”, “content”: system_message}] + history + [{“role”: “user”, “content”: message}] Longer context = Slower processing

Solutions to Improve Response Time Consistency:

1. Implement Streaming Responses Enable stream=True in API calls for perceived faster responses

2. Add Response Time Monitoring Track and display actual processing times for debugging

3. Optimize Tool Handling Process multiple tools in parallel instead of sequentially

4. Implement Caching Cache responses for common queries to reduce API calls

5. Set Reasonable Timeouts Add timeout handling to prevent indefinite waits

Expected Improvements:

Optimization

Time Reduction

Consistency Gain

Streaming

Perceived: 50-70%

High

Parallel tools

Actual: 20-30%

Medium

Caching

Simple queries: 80-90%

High

Timeout handling

N/A

Very High

Conclusion: The 2-15 second variation is normal for this type of implementation, but streaming responses will make the interface feel much more responsive to users, even if the total processing time remains similar.

day5.ipynb Theory Summary

Multi-Modal & Agentic AI System Design - Multi-Modal Generation: Combining text (GPT-4o-mini), image (DALL · E 3), and audio (TTS-1) for rich user interactions. - Tool-Based Function Calling: Dynamically invoking external tools (e.g., flight pricing functions) using structured JSON schemas. - Agentic Behavior: Implementing autonomous decision-making, multi-step reasoning, and task delegation. - Cross-Platform Robustness: Handling OS-specific audio synthesis scenarios with fallbacks. - Gradio UI + State Management: Using Gradio Blocks to manage conversation history, multi-output rendering (text/image/audio), and real-time feedback. - System Integration Pattern: Demonstrates a production-ready architecture for multi-service AI assistants adaptable to business use cases.

Lab Exercises

Note: Since image generation models typically incur additional costs, and Ollama focuses primarily on text-based LLMs without native support for models like DALL · E or Midjourney, the image generation component will be skipped in this implementation.

Lab Exercises

1/ Application for the PC Variation 1 implementation in day5.ipynb

[ ]:

```python
# PC Variation 1
import base64
from io import BytesIO
from PIL import Image
from IPython.display import Audio, display

def talker(message):
    response = openai.audio.speech.create(
        model="tts-1",
        voice="onyx",
        input=message)

    audio_stream = BytesIO(response.content)
    output_filename = "output_audio.mp3"
    with open(output_filename, "wb") as f:
        f.write(audio_stream.read())

    display(Audio(output_filename, autoplay=True))

talker("Well, hi there")
```

[ ]:

```python
import os
import time
from io import BytesIO
from openai import OpenAI
from dotenv import load_dotenv
from IPython.display import Audio, display

class TextToSpeechGenerator:
    def __init__(self, output_folder="03july"):
        load_dotenv(override=True)
        self.client = OpenAI()
        self.voices = ["alloy", "echo", "fable", "onyx", "nova", "shimmer"]
        self.output_folder = output_folder
        self.generated_files = []
        self._create_output_folder()

    def _create_output_folder(self):
        """Create the output folder if it doesn't exist"""
        if not os.path.exists(self.output_folder):
            os.makedirs(self.output_folder)
            print(f"Created output folder: {self.output_folder}")

    def generate_speech(self, message, voice="onyx", filename=None):
        if not message.strip():
            raise ValueError("Message cannot be empty")

        if voice not in self.voices:
            voice = "onyx"

        try:
            response = self.client.audio.speech.create(
                model="tts-1",
                voice=voice,
                input=message
            )

            if filename is None:
                timestamp = int(time.time())
                filename = f"speech_output_{voice}_{timestamp}.mp3"

            # Create full path with subfolder
            full_path = os.path.join(self.output_folder, filename)

            audio_stream = BytesIO(response.content)
            with open(full_path, "wb") as f:
                f.write(audio_stream.read())

            self.generated_files.append(full_path)
            print(f"Audio saved to: {full_path}")
            display(Audio(full_path, autoplay=True))

            return full_path

        except Exception as e:
            print(f"Error generating audio: {e}")
            return None

    def batch_generate(self, messages_and_voices):
        results = []
        for message, voice in messages_and_voices:
            result = self.generate_speech(message, voice)
            results.append(result)
            time.sleep(0.5)
        return results

    def interactive_mode(self):
        while True:
            message = input("Enter your message (or 'quit' to exit): ")
            if message.lower() == 'quit':
                break
            if message.strip():
                voice = input(f"Choose voice {self.voices} [default: onyx]: ").
↪strip() or "onyx"
                self.generate_speech(message, voice=voice)
            else:
                print("Please enter a valid message.")

    def cleanup_files(self):
        """Remove all generated audio files"""
        for filepath in self.generated_files:
            try:
                if os.path.exists(filepath):
                    os.remove(filepath)
                    print(f"Deleted: {filepath}")
            except Exception as e:
                print(f"Could not delete {filepath}: {e}")
        self.generated_files.clear()

    def list_generated_files(self):
        """List all generated files in the output folder"""
        if os.path.exists(self.output_folder):
            files = [f for f in os.listdir(self.output_folder) if f.endswith('.
↪mp3')]
            if files:
                print(f"\nFiles in {self.output_folder}:")
                for file in sorted(files):
                    file_path = os.path.join(self.output_folder, file)
                    file_size = os.path.getsize(file_path)
                    print(f"  - {file} ({file_size} bytes)")
            else:
                print(f"No audio files found in {self.output_folder}")
        else:
            print(f"Output folder {self.output_folder} does not exist")

def main():
    tts = TextToSpeechGenerator("03july")

    voice_examples = [
        ("Welcome to our AI assistant demo!", "alloy"),
        ("This is a demonstration of voice synthesis.", "echo"),
        ("How can I help you today?", "fable"),
        ("Thank you for using our service!", "onyx"),
        ("Have a wonderful day ahead!", "nova"),
        ("Goodbye and see you soon!", "shimmer")
    ]

    print("Text-to-Speech Demo")
    print("=" * 40)

    print("Running batch generation...")
    results = tts.batch_generate(voice_examples)

    successful_files = len([r for r in results if r])
    print(f"\nGenerated {successful_files} audio files in folder: 03july")
    print("Available voices:", ", ".join(tts.voices))

    # Show generated files
    tts.list_generated_files()

    choice = input("\nRun interactive mode? (y/n): ")
    if choice.lower() == 'y':
        tts.interactive_mode()

    cleanup_choice = input("\nCleanup generated files? (y/n): ")
    if cleanup_choice.lower() == 'y':
        tts.cleanup_files()
    else:
        print(f"Files preserved in folder: {tts.output_folder}")

if __name__ == "__main__":
    main()
```

week2 EXERCISE.ipynb Additional End-of-Week Exercise – Week 2

Now use everything you've learned from Week 2 to build a full prototype for the technical question/answerer you created in Week 1 Exercise. Your prototype should include: - A Gradio UI - Streaming output enabled - A system prompt for domain-specific expertise - The ability to switch between models - Bonus: Include tool usage if possible! - Extra Bonus: Add audio input and output for full multi-modal interaction There are so many commercial applications for this – from a language tutor, to a company onboarding agent, or even a companion AI for this course. Good luck – can't wait to see what you build!

[ ]:
```
import os
import time
import gradio as gr
from dotenv import load_dotenv
from openai import OpenAI
from IPython.display import Audio, display
from io import BytesIO
import json
```

[ ]:
```
# Environment Setup & Configuration
load_dotenv(override=True)
```

```
MODELS = {
    "OpenAI GPT-4o-mini": {"client_type": "openai", "model": "gpt-4o-mini"},
    "Ollama Llama 3.2": {"client_type": "ollama", "model": "llama3.2"}
}

VOICES = ["alloy", "echo", "fable", "onyx", "nova", "shimmer"]
AUDIO_FOLDER = "tutor_audio"

print("Environment loaded successfully!")
```

[ ]:

```
# Client Initialization
class ModelClients:
    def __init__(self):
        self.openai_client = OpenAI()
        self.ollama_client = OpenAI(base_url='http://localhost:11434/v1',␣
 ↪api_key='ollama')

    def get_client(self, client_type):
        if client_type == "openai":
            return self.openai_client
        elif client_type == "ollama":
            return self.ollama_client
        else:
            raise ValueError(f"Unknown client type: {client_type}")

clients = ModelClients()
print("Model clients initialized!")
```

[ ]:

```
# System Prompt Definition
SYSTEM_PROMPT = """You are an expert technical tutor and coding mentor. When␣
 ↪explaining technical concepts or code:

1. Break down complex topics into digestible parts
2. Provide step-by-step explanations with clear reasoning
3. Include practical examples and use cases
4. Explain best practices and common pitfalls
5. Adapt your explanation level based on the question complexity
6. Use analogies when helpful for understanding
7. Always provide actionable insights

Keep responses educational, engaging, and markdown-formatted."""

print("System prompt configured!")
```

[ ]:

```
# Knowledge Base Definition
KNOWLEDGE_BASE = {
    "python_basics": "Python fundamentals including syntax, data types, control␣
  ↪structures",
    "algorithms": "Algorithm design, complexity analysis, common patterns",
    "web_development": "Frontend/backend technologies, frameworks, best␣
  ↪practices",
    "data_science": "Data analysis, machine learning, statistics concepts",
    "system_design": "Architecture patterns, scalability, performance␣
  ↪optimization",
    "databases": "SQL, NoSQL, database design, optimization techniques",
    "devops": "CI/CD, containerization, cloud deployment, monitoring",
    "security": "Web security, authentication, encryption, best practices"
}

print(f"Knowledge base loaded with {len(KNOWLEDGE_BASE)} topics!")
```

[ ]:

```
# Knowledge Base Search
def search_knowledge_base(query):
    """Tool function to search knowledge base"""
    query_lower = query.lower()
    results = []

    for topic, description in KNOWLEDGE_BASE.items():
        if any(keyword in query_lower for keyword in topic.split('_')):
            results.append(f"**{topic.replace('_', ' ').title()}**:␣
  ↪{description}")

    if results:
        return "Found relevant topics:\n" + "\n".join(results)
    return "No specific topics found in knowledge base. I'll provide a general␣
  ↪explanation."

# Test the function
print("Knowledge base search function defined!")
print("Test:", search_knowledge_base("python"))
```

[ ]:

~~~
# Code Example Generator
def generate_code_example(concept):
    """Tool function to generate code examples"""
    concept_lower = concept.lower()

    examples = {
        "list_comprehension": """```python
# List comprehension example
numbers = [1, 2, 3, 4, 5]
squares = [x**2 for x in numbers if x % 2 == 0]
print(squares)  # [4, 16]
```""",
~~~

~~~
        "generator": """```python
# Generator function example
def fibonacci_generator(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b

# Usage
fib_gen = fibonacci_generator(10)
for num in fib_gen:
    print(num)
```""",
        "decorator": """```python
# Decorator example
def timing_decorator(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} took {end - start:.2f} seconds")
        return result
    return wrapper

@timing_decorator
def slow_function():
    time.sleep(1)
    return "Done!"
```""",
        "class": """```python
# Class example with inheritance
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return f"{self.name} says Woof!"

class Cat(Animal):
    def speak(self):
        return f"{self.name} says Meow!"
```"""
    }
~~~

```
    for key, example in examples.items():
        if key in concept_lower:
            return f"Here's a practical example for {concept}:\n\n{example}"

    return f"I'll provide a conceptual explanation for '{concept}' in my␣
 ↪response."

print("Code example generator function defined!")
```

[ ]:

```
# OpenAI Function Definitions
def get_tools():
    """Define tools for OpenAI function calling"""
    return [
        {
            "type": "function",
            "function": {
                "name": "search_knowledge_base",
                "description": "Search the knowledge base for relevant␣
 ↪technical topics",
                "parameters": {
                    "type": "object",
                    "properties": {
                        "query": {
                            "type": "string",
                            "description": "The query to search for in the␣
 ↪knowledge base"
                        }
                    },
                    "required": ["query"]
                }
            }
        },
        {
            "type": "function",
            "function": {
                "name": "generate_code_example",
                "description": "Generate practical code examples for␣
 ↪programming concepts",
                "parameters": {
                    "type": "object",
                    "properties": {
                        "concept": {
                            "type": "string",
                            "description": "The programming concept to generate␣
 ↪an example for"
                        }
                    },
                    "required": ["concept"]
                }
            }
        }
    ]

print("OpenAI function definitions created!")
```

[ ]:

```
# Tool Call Handler
def handle_tool_calls(message):
    """Handle multiple tool calls"""
    tool_responses = []

    for tool_call in message.tool_calls:
        function_name = tool_call.function.name
        arguments = json.loads(tool_call.function.arguments)

        if function_name == "search_knowledge_base":
            result = search_knowledge_base(arguments.get('query', ''))
        elif function_name == "generate_code_example":
            result = generate_code_example(arguments.get('concept', ''))
        else:
            result = "Unknown function called"

        tool_responses.append({
            "role": "tool",
            "content": result,
            "tool_call_id": tool_call.id
        })

    return tool_responses

print("Tool call handler defined!")
```

[ ]:

```
# Audio Generation Setup
def create_audio_folder():
    """Create audio output folder"""
    if not os.path.exists(AUDIO_FOLDER):
        os.makedirs(AUDIO_FOLDER)
        print(f"Created audio folder: {AUDIO_FOLDER}")

def generate_audio_response(text, voice="onyx"):
    """Generate audio from text response"""
    try:
        client = clients.openai_client
        response = client.audio.speech.create(
            model="tts-1",
            voice=voice,
            input=text[:1000]
        )

        timestamp = int(time.time())
        filename = f"tutor_response_{voice}_{timestamp}.mp3"
        filepath = os.path.join(AUDIO_FOLDER, filename)

        with open(filepath, "wb") as f:
            f.write(response.content)

        return filepath
    except Exception as e:
        print(f"Audio generation error: {e}")
        return None

create_audio_folder()
print("Audio generation functions defined!")
```

[ ]:

```
# Core Response Generation Function
def get_streaming_response(question, model_choice, enable_audio=False,␣
 ↪voice_choice="onyx"):
    """Get streaming response from selected model with optional tools and␣
 ↪audio"""

    model_config = MODELS.get(model_choice)
    if not model_config:
        yield "Invalid model selection", None
        return

    try:
        client = clients.get_client(model_config["client_type"])
    except Exception as e:
        yield f"Client error: {str(e)}", None
        return

    messages = [
        {"role": "system", "content": SYSTEM_PROMPT},
        {"role": "user", "content": question}
    ]

    try:
        use_tools = model_config["client_type"] == "openai"
        tools = get_tools() if use_tools else None

        if use_tools:
            response = client.chat.completions.create(
                model=model_config["model"],
                messages=messages,
                tools=tools,
                stream=True
            )
        else:
            response = client.chat.completions.create(
                model=model_config["model"],
                messages=messages,
                stream=True
            )

        full_response = ""

        for chunk in response:
            if chunk.choices[0].delta.content is not None:
                content = chunk.choices[0].delta.content
                full_response += content
                yield full_response, None

        if use_tools and not full_response:
            response = client.chat.completions.create(
                model=model_config["model"],
                messages=messages,
                tools=tools
            )

            if response.choices[0].finish_reason == "tool_calls":
                assistant_message = response.choices[0].message
                tool_responses = handle_tool_calls(assistant_message)

                messages.append(assistant_message)
                messages.extend(tool_responses)

                final_response = client.chat.completions.create(
                    model=model_config["model"],
                    messages=messages,
                    stream=True
                )

                full_response = ""
                for chunk in final_response:
                    if chunk.choices[0].delta.content is not None:
                        content = chunk.choices[0].delta.content
                        full_response += content
                        yield full_response, None
        audio_file = None
        if enable_audio and full_response:
            audio_file = generate_audio_response(full_response, voice_choice)

        yield full_response, audio_file

    except Exception as e:
        error_msg = f"  Error: {str(e)}"
        if "ollama" in model_choice.lower():
            error_msg += "\n\n**Ollama Troubleshooting:**\n"
            error_msg += "1. Ensure Ollama is running: `ollama serve`\n"
            error_msg += "2. Pull the model: `ollama pull llama3.2`\n"
            error_msg += "3. Check http://localhost:11434 is accessible"

        yield error_msg, None

print("Core response generation function defined!")
```

[ ]:

```
# Gradio Interface - Processing Function
def process_question(question, model_choice, enable_audio, voice_choice,␣
 ↪history):
    """Process question and update chat"""
    if not question.strip():
        return history, "", None

    history.append([question, ""])

    for response, audio in get_streaming_response(question, model_choice,␣
 ↪enable_audio, voice_choice):
        # Update the last assistant message
        history[-1][1] = response
        yield history, "", audio

    return history, "", audio

print("Question processing function defined!")
```

[ ]:

```
# Gradio Interface - CSS Styling
CSS_STYLES = """
.gradio-container {
    max-width: 1200px !important;
}
.chat-message {
    padding: 10px;
    margin: 5px 0;
    border-radius: 10px;
}
.example-btn {
    margin: 2px;
    font-size: 12px;
}
"""

print("CSS styles defined!")
```

[ ]:

```
# Gradio Interface - Example Questions
EXAMPLE_QUESTIONS = [
    "Explain Python list comprehensions with examples",
    "What are decorators and when should I use them?",
    "How does the yield keyword work in Python?",
    "Explain time complexity in algorithms",
    "What's the difference between REST and GraphQL?",
    "How does async/await work in Python?",
    "What are design patterns in software engineering?",
    "Explain database normalization concepts",
    "How do I optimize SQL queries?",
    "What is the difference between stack and heap memory?"
]

print(f"Example questions defined: {len(EXAMPLE_QUESTIONS)} questions available!")
```

[ ]:

```
# Gradio Interface - Main Layout
def create_gradio_interface():
    """Create the complete Gradio interface"""

    with gr.Blocks(css=CSS_STYLES, title="Technical Tutor Assistant") as demo:
        gr.Markdown("""
        # Technical Tutor Assistant

        **Your AI-powered technical mentor for programming, algorithms, and software engineering concepts.**

        **Features:**
        - Multi-model support (OpenAI GPT-4o-mini & Ollama Llama 3.2)
        - Smart tool usage for knowledge base search and code examples
        - Audio responses with multiple voice options
        - Streaming responses for real-time interaction
        - Educational focus with step-by-step explanations
        """)

        with gr.Row():
            with gr.Column(scale=3):
                chatbot = gr.Chatbot(
                    label="Technical Discussion",
                    height=500,
                    show_label=True,
                    container=True
                )

                with gr.Row():
                    question_input = gr.Textbox(
                        placeholder="Ask me about programming concepts, code␣
 ↪explanations, algorithms, or any technical topic...",
                        label="Your Technical Question",
                        lines=2,
                        scale=4
                    )
                    submit_btn = gr.Button("Ask Tutor", variant="primary",␣
 ↪scale=1)

        return demo, chatbot, question_input, submit_btn

print("Main interface layout function defined!")
```

[ ]:

```
# Gradio Interface - Settings Panel Components
def create_settings_panel():
    """Create settings panel components"""

    gr.Markdown("### Settings")

    model_choice = gr.Radio(
        choices=list(MODELS.keys()),
        label="Select Model",
        value="OpenAI GPT-4o-mini"
    )

    enable_audio = gr.Checkbox(
        label="Enable Audio Response",
        value=False
    )

    voice_choice = gr.Dropdown(
        choices=VOICES,
        label="Voice Selection",
        value="onyx",
        visible=False
    )

    # Audio output
    audio_output = gr.Audio(
        label="Audio Response",
        visible=False,
        autoplay=True
    )

    return model_choice, enable_audio, voice_choice, audio_output

print("Settings panel components function defined!")
```

[ ]:

```
# Gradio Interface - Example Questions Section
def create_example_questions():
    """Create example questions section"""

    gr.Markdown("### Example Questions")

    example_buttons = []

    # Create buttons for first 6 example questions
    for i, question in enumerate(EXAMPLE_QUESTIONS[:6]):
        btn = gr.Button(
            question[:50] + "..." if len(question) > 50 else question,
            size="sm",
            elem_classes=["example-btn"]
        )
        example_buttons.append((btn, question))

    return example_buttons

print("Example questions section function defined!")
```

[ ]:

```
def create_complete_interface():
    """Assemble the complete Gradio interface with all components and event␣
 ↪handlers"""

    with gr.Blocks(css=CSS_STYLES, title="Technical Tutor Assistant") as demo:
        gr.Markdown("""
        # Technical Tutor Assistant

        **Your AI-powered technical mentor for programming, algorithms, and␣
 ↪software engineering concepts.**

        **Features:**
        - Multi-model support (OpenAI GPT-4o-mini & Ollama Llama 3.2)
        - Smart tool usage for knowledge base search and code examples
        - Audio responses with multiple voice options
        - Streaming responses for real-time interaction
        - Educational focus with step-by-step explanations
        """)

        with gr.Row():
            with gr.Column(scale=3):
                chatbot = gr.Chatbot(
                    label="Technical Discussion",
                    height=500,
                    show_label=True,
                    container=True
                )

                with gr.Row():
                    question_input = gr.Textbox(
                        placeholder="Ask me about programming concepts, code␣
↪explanations, algorithms, or any technical topic...",
                        label="Your Technical Question",
                        lines=2,
                        scale=4
                    )
                    submit_btn = gr.Button("Ask Tutor", variant="primary",␣
↪scale=1)

            with gr.Column(scale=1):
                gr.Markdown("### Settings")

                model_choice = gr.Radio(
                    choices=list(MODELS.keys()),
                    label="Select Model",
                    value="OpenAI GPT-4o-mini"
                )

                enable_audio = gr.Checkbox(
                    label="Enable Audio Response",
                    value=False
                )

                voice_choice = gr.Dropdown(
                    choices=VOICES,
                    label="Voice Selection",
                    value="onyx",
                    visible=False
                )

                audio_output = gr.Audio(
                    label="Audio Response",
                    visible=False,
                    autoplay=True
                )

                gr.Markdown("### Example Questions")

                for i, question in enumerate(EXAMPLE_QUESTIONS[:6]):
                    btn = gr.Button(
                        question[:45] + "..." if len(question) > 45 else␣
↪question,
                        size="sm",
                        elem_classes=["example-btn"]
                    )
                    btn.click(
                        lambda q=question: q,
                        outputs=question_input
                    )

                clear_btn = gr.Button("Clear Chat", variant="secondary")

        # Event Handlers (MUST be inside the Blocks context)
        def toggle_audio_settings(enable_audio):
            return (
                gr.update(visible=enable_audio),
                gr.update(visible=enable_audio)
            )

        # Audio settings toggle
        enable_audio.change(
            toggle_audio_settings,
            inputs=[enable_audio],
            outputs=[voice_choice, audio_output]
        )

        # Submit events
        submit_btn.click(
            process_question,
            inputs=[question_input, model_choice, enable_audio, voice_choice,␣
↪chatbot],
            outputs=[chatbot, question_input, audio_output]
        )

        question_input.submit(
            process_question,
            inputs=[question_input, model_choice, enable_audio, voice_choice,␣
↪chatbot],
            outputs=[chatbot, question_input, audio_output]
        )
        # Clear chat
        clear_btn.click(
            lambda: ([], None),
            outputs=[chatbot, audio_output]
        )

    return demo

print("Complete interface with event handlers defined!")

def setup_event_handlers(demo, chatbot, question_input, submit_btn,␣
 ↪model_choice, enable_audio, voice_choice, audio_output):
    """Set up all event handlers for the interface"""

    def toggle_audio_settings(enable_audio):
        return (
            gr.update(visible=enable_audio),
            gr.update(visible=enable_audio)
        )

    enable_audio.change(
        toggle_audio_settings,
        inputs=[enable_audio],
        outputs=[voice_choice, audio_output]
    )

    submit_event = submit_btn.click(
        process_question,
        inputs=[question_input, model_choice, enable_audio, voice_choice,␣
 ↪chatbot],
        outputs=[chatbot, question_input, audio_output]
    )

    question_input.submit(
        process_question,
        inputs=[question_input, model_choice, enable_audio, voice_choice,␣
 ↪chatbot],
        outputs=[chatbot, question_input, audio_output]
    )

    clear_btn = gr.Button("Clear Chat", variant="secondary")
    clear_btn.click(
        lambda: ([], None),
        outputs=[chatbot, audio_output]
    )
    return demo

print("Event handlers setup function defined!")
```

[ ]:

```
# Event Handlers and Interface Logic
# def setup_event_handlers(demo, chatbot, question_input, submit_btn,␣
 ↪model_choice, enable_audio, voice_choice, audio_output):
#     """Set up all event handlers for the interface"""

#     def toggle_audio_settings(enable_audio):
#         return (
#             gr.update(visible=enable_audio),
#             gr.update(visible=enable_audio)
#         )

#     enable_audio.change(
#         toggle_audio_settings,
#         inputs=[enable_audio],
#         outputs=[voice_choice, audio_output]
#     )

#     submit_event = submit_btn.click(
#         process_question,
#         inputs=[question_input, model_choice, enable_audio, voice_choice,␣
 ↪chatbot],
#         outputs=[chatbot, question_input, audio_output]
#     )

#     question_input.submit(
#         process_question,
#         inputs=[question_input, model_choice, enable_audio, voice_choice,␣
 ↪chatbot],
#         outputs=[chatbot, question_input, audio_output]
#     )

#     clear_btn = gr.Button("Clear Chat", variant="secondary")
#     clear_btn.click(
#         lambda: ([], None),
#         outputs=[chatbot, audio_output]
#     )

#     return demo

# print("Event handlers setup function defined!")
```

[ ]:

```
# Launch Interface Function
def launch_interface():
    """Create and launch the complete technical tutor interface"""

    print("Creating Technical Tutor Assistant interface...")

    demo = create_complete_interface()

    print("Interface created successfully!")
    print("Features available:")
    print(" - Multi-model support (OpenAI + Ollama)")
    print(" - Tool usage for knowledge base and code examples")
    print(" - Audio response generation")
    print(" - Streaming responses")
    print(" - Example questions for quick start")

    return demo
```

[ ]:

```
# Main Execution
print("Starting Technical Tutor Assistant...")
print("=" * 50)

# Close any existing interfaces
gr.close_all()

demo = launch_interface()

demo.launch(
    share=False,
    server_name="0.0.0.0",
    # server_port=7860, # Uncomment to specify a port
    server_port=None,  # Let Gradio choose the port automatically
    show_error=True,
    inbrowser=True
)

print("\nTechnical Tutor Assistant is now running!")
print("Access the interface at: http://localhost:7860")
print("Try asking technical questions to get started!")
```

Structured Test Case Suite

Category

Test Questions

Basic Q&A

“What is object-oriented programming?” “Explain how HTTP works” “What is the difference between a list and a tuple in Python?”

Knowledge Base

“Tell me about Python basics” “What should I know about databases?” “Explain concepts in data science” “What are important DevOps practices?”

Code Examples

“Show me a list comprehension example” “Can you provide a decorator example in Python?” “How do I create a generator function?” “Give me an example of a class with inheritance”

Combined Tool Usage

“Explain Python generators and show me an example” “What are some web development basics and show a simple API code example?” “Tell me about algorithms and give an example of binary search”

Model Switching

Switch to GPT-4o-mini: “What are microservices?” Then: “Show me an example of Python list comprehension” Switch to Ollama: “What is Docker?” “Explain the MVC pattern” Switch models mid-conversation and continue

Audio Response

“Explain what RESTful APIs are” (test audio response) “What is TCP/IP?” (short answer) “Explain how neural networks work in detail” (long response) Toggle audio on/off and switch voices

Edge Cases

Submit: ”” (empty input) Submit: “???” Submit: “Hi”

Non-Tech Questions

“What’s the weather like today?” “Tell me a joke” “Who won the last World Cup?”

Invalid Inputs

Ask a question when Ollama service is down Ask a policy-sensitive question Corrupt the knowledge base and test retrieval

Context Retention

“What is Docker?” → “How is it different from a VM?” “What is a Python decorator?” → “Show me a code example”

Multi-turn Flow

“Explain OOP” → “Can you simplify that?” → “Give me an example”

UI Interaction

Use pre-filled example buttons Clear chat Submit with Enter and with Submit button

Performance Stress

Submit multiple questions rapidly Enable audio + long response + tool usage Open multiple tabs and interact in parallel

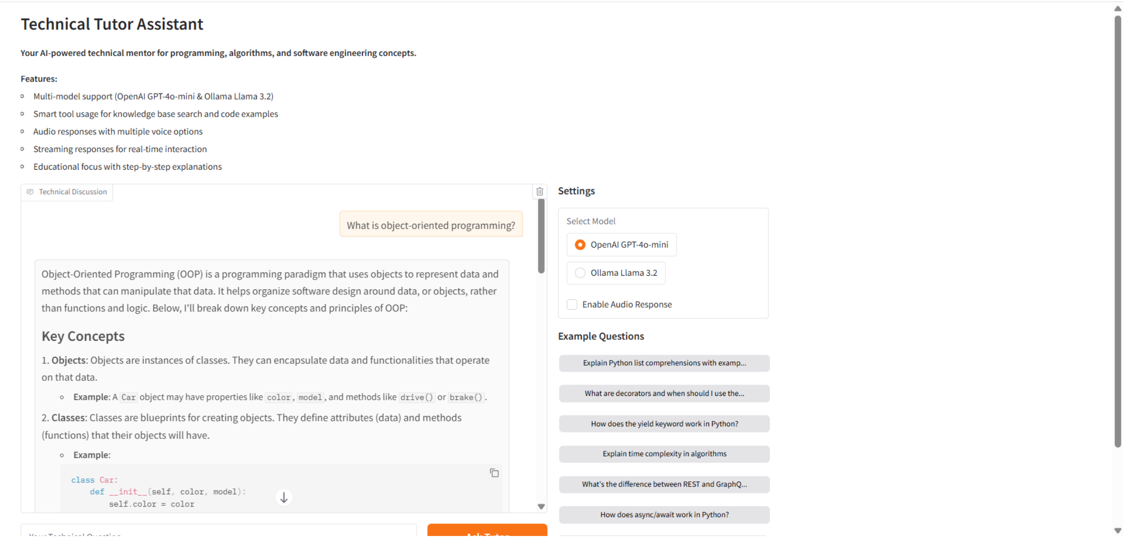
Markdown Rendering

“How do I read a CSV using pandas?” (code block) “Compare different sorting algorithms” (table) “List advantages of microservices” (bullet points)

Tool Combo

“What are best practices in Python and show examples” “Compare SQL vs NoSQL databases and provide example queries”

Output Samples



Technical Tutor Assistant

Your AI-powered technical mentor for programming, algorithms, and software engineering concepts.

Features:

- Multi-model support (OpenAI GPT-4o-mini & Ollama Llama 3.2)
- Smart tool usage for knowledge base search and code examples
- Audio responses with multiple voice options
- Streaming responses for real-time interaction
- Educational focus with step-by-step explanations

What is object-oriented programming?

Object-Oriented Programming (OOP) is a programming paradigm that uses objects to represent data and methods that can manipulate that data. It helps organize software design around data, or objects, rather than functions and logic. Below, I'll break down key concepts and principles of OOP:

Key Concepts

1. **Objects**: Objects are instances of classes. They can encapsulate data and functionalities that operate on that data.
   - **Example**: A `Car` object may have properties like `color`, `model`, and methods like `drive()` or `brake()`.
2. **Classes**: Classes are blueprints for creating objects. They define attributes (data) and methods (functions) that their objects will have.
   - **Example**:

```
class Car:
    def __init__(self, color, model):
        self.color = color
```

Settings

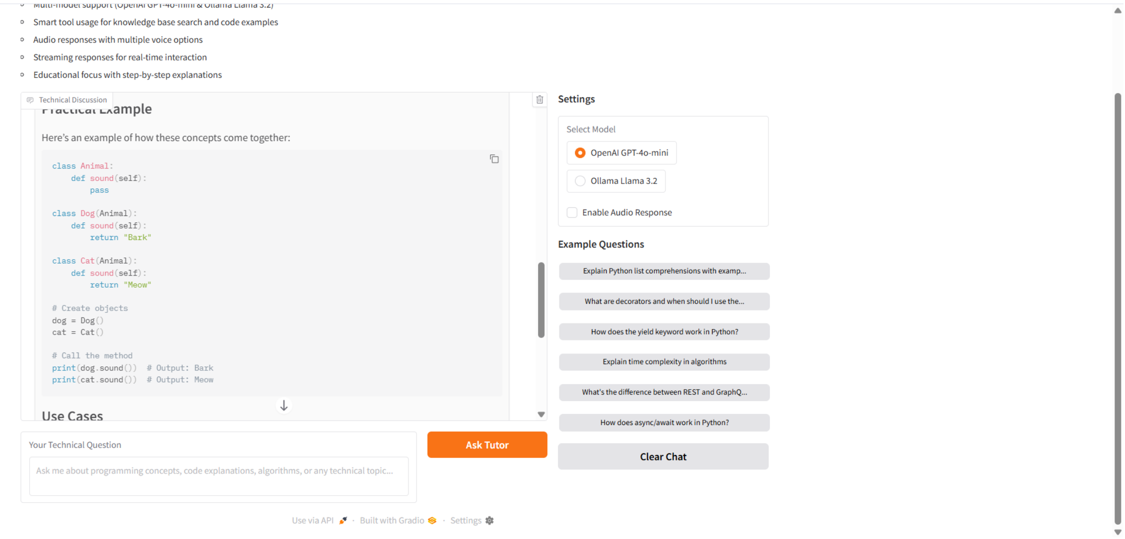
Select Model

● OpenAI GPT-4o-mini

○ Ollama Llama 3.2

□ Enable Audio Response

Technical Discussion
Example Questions
Explain Python list comprehensions with examp...
What are decorators and when should I use the...
How does the yield keyword work in Python?
Explain time complexity in algorithms
What's the difference between REST and GraphQ...
How does async/await work in Python?



- Smart tool usage for knowledge base search and code examples
- Audio responses with multiple voice options
- Streaming responses for real-time interaction
- Educational focus with step-by-step explanations

Practical Example

Here’s an example of how these concepts come together:

```
class Animal:
    def sound(self):
        pass

class Dog(Animal):
    def sound(self):
        return "Bark"

class Cat(Animal):
    def sound(self):
        return "Meow"

# Create objects
dog = Dog()
cat = Cat()

# Call the method
print(dog.sound())  # Output: Bark
print(cat.sound())  # Output: Meow
```

Use Cases

Settings

Select Model

● OpenAI GPT-4o-mini

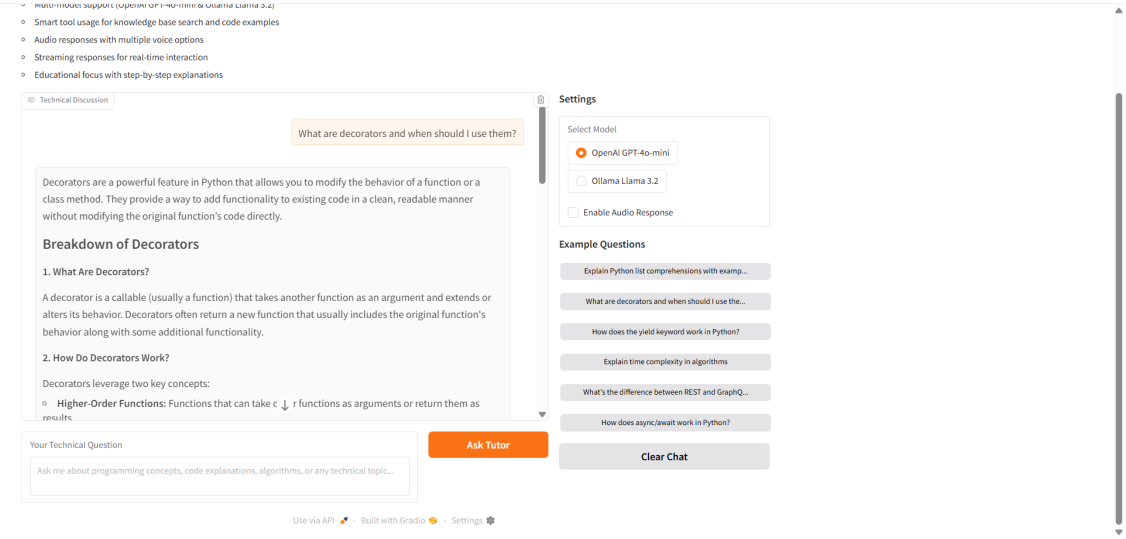
○ Ollama Llama 3.2

□ Enable Audio Response

Your Technical Question

Ask me about programming concepts, code explanations, algorithms, or any technical topic...

Technical Discussion
Example Questions
Explain Python list comprehensions with examp...
What are decorators and when should I use the...
How does the yield keyword work in Python?
Explain time complexity in algorithms
What's the difference between REST and GraphQ...
How does async/await work in Python?
Ask Tutor
Clear Chat
Use via API · Built with Gradio · Settings



- Multi-model support (OpenAI GPT-4o-mini & Ollama Llama 3.2)
- Smart tool usage for knowledge base search and code examples
- Audio responses with multiple voice options
- Streaming responses for real-time interaction
- Educational focus with step-by-step explanations

Technical Discussion

What are decorators and when should I use them?

Decorators are a powerful feature in Python that allows you to modify the behavior of a function or a class method. They provide a way to add functionality to existing code in a clean, readable manner without modifying the original function’s code directly.

Breakdown of Decorators

1. What Are Decorators?

A decorator is a callable (usually a function) that takes another function as an argument and extends or alters its behavior. Decorators often return a new function that usually includes the original function's behavior along with some additional functionality.

2. How Do Decorators Work?

Decorators leverage two key concepts:

- **Higher-Order Functions:** Functions that can take c r functions as arguments or return them as results

Your Technical Question

Ask me about programming concepts, code explanations, algorithms, or any technical topic...

Ask Tutor

Settings

Select Model

● OpenAI GPT-4o-mini

○ Ollama Llama 3.2

□ Enable Audio Response

Example Questions

Explain Python list comprehensions with examp...

What are decorators and when should I use the...

How does the yield keyword work in Python?

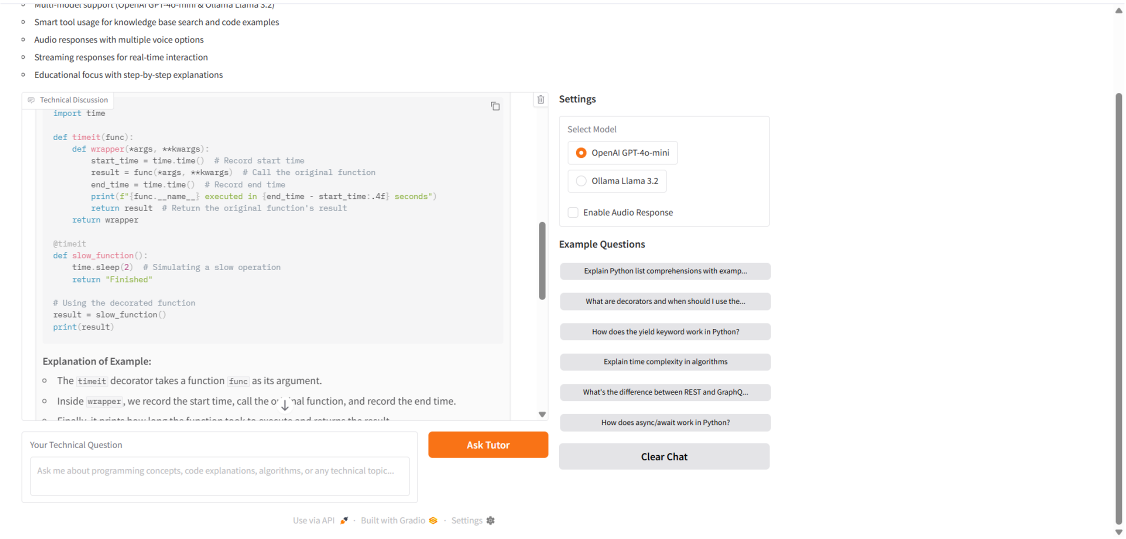
Explain time complexity in algorithms

What's the difference between REST and GraphQ...

How does async/await work in Python?

Clear Chat

Use via API · Built with Gradio · Settings



- Multi-model support (OpenAI GPT-4o-mini & Ollama Llama 3.2)
- Smart tool usage for knowledge base search and code examples
- Audio responses with multiple voice options
- Streaming responses for real-time interaction
- Educational focus with step-by-step explanations

Technical Discussion

```python
import time

def timeit(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()  # Record start time
        result = func(*args, **kwargs)  # Call the original function
        end_time = time.time()  # Record end time
        print(f"{func.__name__} executed in {end_time - start_time:.4f} seconds")
        return result  # Return the original function's result
    return wrapper

@timeit
def slow_function():
    time.sleep(2)  # Simulating a slow operation
    return "Finished"

# Using the decorated function
result = slow_function()
print(result)
```

Explanation of Example:

- The `timeit` decorator takes a function `func` as its argument.
- Inside `wrapper`, we record the start time, call the o nal function, and record the end time.
- Finally, it prints how long the function took to execute and returns the result.

Your Technical Question

Ask me about programming concepts, code explanations, algorithms, or any technical topic...

Ask Tutor

Settings

Select Model

● OpenAI GPT-4o-mini

○ Ollama Llama 3.2

□ Enable Audio Response

Example Questions

Explain Python list comprehensions with examp...

What are decorators and when should I use the...

How does the yield keyword work in Python?

Explain time complexity in algorithms

What's the difference between REST and GraphQ...

How does async/await work in Python?

Clear Chat

Use via API · Built with Gradio · Settings

Model Comparison Summary

GPT-4o-mini Strengths: - Excellent code explanations and technical accuracy - Fast response time (2-6s) with high-quality code generation - Effectively leverages knowledge base search and code example tools Drawbacks: - Struggles with non-technical/general knowledge questions - Limited to training data prior to its cutoff date - Inconsistent context tracking in multi-turn conversations - Requires API key and incurs usage costs Llama 3.2 Strengths: - Free alternative with no API cost - Local execution with no internet dependency - Customizable for specific hardware and local workflows - Open-source and community-supported Drawbacks: - Less accurate with advanced technical topics - Informal formatting, occasional Markdown issues - No built-in tool integration - Slower response time (5–15s), more setup required System Performance Overview - Minor latency issues with Llama and tool overhead - Limited context window impacts follow-up accuracy - Audio synthesis can introduce additional delay - However, successful model switching with real-time feedback and UI consistency observed