Chapter 16: Resource-Aware Optimization

Resource-Aware Optimization enables intelligent agents to dynamically monitor and manage computational, temporal, and financial resources during operation. This differs from simple planning, which primarily focuses on action sequencing. Resource-Aware Optimization requires agents to make decisions regarding action execution to achieve goals within specified resource budgets or to optimize efficiency. This involves choosing between more accurate but expensive models and faster, lower-cost ones, or deciding whether to allocate additional compute for a more refined response versus returning a quicker, less detailed answer.

For example, consider an agent tasked with analyzing a large dataset for a financial analyst. If the analyst needs a preliminary report immediately, the agent might use a faster, more affordable model to quickly summarize key trends. However, if the analyst requires a highly accurate forecast for a critical investment decision and has a larger budget and more time, the agent would allocate more resources to utilize a powerful, slower, but more precise predictive model. A key strategy in this category is the fallback mechanism, which acts as a safeguard when a preferred model is unavailable due to being overloaded or throttled. To ensure graceful degradation, the system automatically switches to a default or more affordable model, maintaining service continuity instead of failing completely.

Practical Applications & Use Cases

Practical use cases include:

- **Cost-Optimized LLM Usage:** An agent deciding whether to use a large, expensive LLM for complex tasks or a smaller, more affordable one for simpler queries, based on a budget constraint.
- **Latency-Sensitive Operations:** In real-time systems, an agent chooses a faster but potentially less comprehensive reasoning path to ensure a timely response.
- **Energy Efficiency:** For agents deployed on edge devices or with limited power, optimizing their processing to conserve battery life.
- **Fallback for service reliability:** An agent automatically switches to a backup model when the primary choice is unavailable, ensuring service continuity and graceful degradation.

- **Data Usage Management:** An agent opting for summarized data retrieval instead of full dataset downloads to save bandwidth or storage.
- **Adaptive Task Allocation:** In multi-agent systems, agents self-assign tasks based on their current computational load or available time.

Hands-On Code Example

An intelligent system for answering user questions can assess the difficulty of each question. For simple queries, it utilizes a cost-effective language model such as Gemini Flash. For complex inquiries, a more powerful, but expensive, language model (like Gemini Pro) is considered. The decision to use the more powerful model also depends on resource availability, specifically budget and time constraints. This system dynamically selects appropriate models.

For example, consider a travel planner built with a hierarchical agent. The high-level planning, which involves understanding a user's complex request, breaking it down into a multi-step itinerary, and making logical decisions, would be managed by a sophisticated and more powerful LLM like Gemini Pro. This is the "planner" agent that requires a deep understanding of context and the ability to reason.

However, once the plan is established, the individual tasks within that plan, such as looking up flight prices, checking hotel availability, or finding restaurant reviews, are essentially simple, repetitive web queries. These "tool function calls" can be executed by a faster and more affordable model like Gemini Flash. It is easier to visualize why the affordable model can be used for these straightforward web searches, while the intricate planning phase requires the greater intelligence of the more advanced model to ensure a coherent and logical travel plan.

Google's ADK supports this approach through its multi-agent architecture, which allows for modular and scalable applications. Different agents can handle specialized tasks. Model flexibility enables the direct use of various Gemini models, including both Gemini Pro and Gemini Flash, or integration of other models through LiteLLM. The ADK's orchestration capabilities support dynamic, LLM-driven routing for adaptive behavior. Built-in evaluation features allow systematic assessment of agent performance, which can be used for system refinement (see the Chapter on Evaluation and Monitoring).

Next, two agents with identical setup but utilizing different models and costs will be defined.

```
# Conceptual Python-like structure, not runnable code

from google.adk.agents import Agent
# from google.adk.models.lite_llm import LiteLlm # If using models
not directly supported by ADK's default Agent

# Agent using the more expensive Gemini Pro 2.5
gemini_pro_agent = Agent(
   name="GeminiProAgent",
   model="gemini-2.5-pro", # Placeholder for actual model name if
different
   description="A highly capable agent for complex queries.",
   instruction="You are an expert assistant for complex
problem-solving."
)

# Agent using the less expensive Gemini Flash 2.5
gemini_flash_agent = Agent(
   name="GeminiFlashAgent",
   model="gemini-2.5-flash", # Placeholder for actual model name if
different
   description="A fast and efficient agent for simple queries.",
   instruction="You are a quick assistant for straightforward
questions."
)
```

A Router Agent can direct queries based on simple metrics like query length, where shorter queries go to less expensive models and longer queries to more capable models. However, a more sophisticated Router Agent can utilize either LLM or ML models to analyze query nuances and complexity. This LLM router can determine which downstream language model is most suitable. For example, a query requesting a factual recall is routed to a flash model, while a complex query requiring deep analysis is routed to a pro model.

Optimization techniques can further enhance the LLM router's effectiveness. Prompt tuning involves crafting prompts to guide the router LLM for better routing decisions. Fine-tuning the LLM router on a dataset of queries and their optimal model choices improves its accuracy and efficiency. This dynamic routing capability balances response quality with cost-effectiveness.

```
# Conceptual Python-like structure, not runnable code

from google.adk.agents import Agent, BaseAgent
from google.adk.events import Event
from google.adk.agents.invocation_context import InvocationContext
import asyncio

class QueryRouterAgent(BaseAgent):
   name: str = "QueryRouter"
   description: str = "Routes user queries to the appropriate LLM
agent based on complexity."

   async def _run_async_impl(self, context: InvocationContext) ->
AsyncGenerator[Event, None]:
       user_query = context.current_message.text # Assuming text
input
       query_length = len(user_query.split()) # Simple metric: number
of words

       if query_length < 20: # Example threshold for simplicity vs.
complexity
           print(f"Routing to Gemini Flash Agent for short query
(length: {query_length})")
           # In a real ADK setup, you would 'transfer_to_agent' or
directly invoke
           # For demonstration, we'll simulate a call and yield its
response
           response = await
gemini_flash_agent.run_async(context.current_message)
           yield Event(author=self.name, content=f"Flash Agent
processed: {response}")
       else:
           print(f"Routing to Gemini Pro Agent for long query
(length: {query_length})")
           response = await
gemini_pro_agent.run_async(context.current_message)
           yield Event(author=self.name, content=f"Pro Agent
processed: {response}")
```

The Critique Agent evaluates responses from language models, providing feedback that serves several functions. For self-correction, it identifies errors or inconsistencies, prompting the answering agent to refine its output for improved quality. It also systematically assesses responses for performance monitoring, tracking metrics like accuracy and relevance, which are used for optimization.

Additionally, its feedback can signal reinforcement learning or fine-tuning; consistent identification of inadequate Flash model responses, for instance, can refine the router agent's logic. While not directly managing the budget, the Critique Agent contributes to indirect budget management by identifying suboptimal routing choices, such as directing simple queries to a Pro model or complex queries to a Flash model, which leads to poor results. This informs adjustments that improve resource allocation and cost savings.

The Critique Agent can be configured to review either only the generated text from the answering agent or both the original query and the generated text, enabling a comprehensive evaluation of the response's alignment with the initial question.

```
CRITIC_SYSTEM_PROMPT = """
You are the **Critic Agent**, serving as the quality assurance arm of
our collaborative research assistant system. Your primary function is
to **meticulously review and challenge** information from the
Researcher Agent, guaranteeing **accuracy, completeness, and unbiased
presentation**.
Your duties encompass:
* **Assessing research findings** for factual correctness,
thoroughness, and potential leanings.
* **Identifying any missing data** or inconsistencies in reasoning.
* **Raising critical questions** that could refine or expand the
current understanding.
* **Offering constructive suggestions** for enhancement or exploring
different angles.
* **Validating that the final output is comprehensive** and balanced.
All criticism must be constructive. Your goal is to fortify the
research, not invalidate it. Structure your feedback clearly, drawing
attention to specific points for revision. Your overarching aim is to
ensure the final research product meets the highest possible quality
standards.
"""
```

The Critic Agent operates based on a predefined system prompt that outlines its role, responsibilities, and feedback approach. A well-designed prompt for this agent must clearly establish its function as an evaluator. It should specify the areas for critical focus and emphasize providing constructive feedback rather than mere dismissal. The prompt should also encourage the identification of both strengths and weaknesses, and it must guide the agent on how to structure and present its feedback.

Hands-On Code with OpenAI

This system uses a resource-aware optimization strategy to handle user queries efficiently. It first classifies each query into one of three categories to determine the most appropriate and cost-effective processing pathway. This approach avoids wasting computational resources on simple requests while ensuring complex queries get the necessary attention. The three categories are:

- simple: For straightforward questions that can be answered directly without complex reasoning or external data.
- reasoning: For queries that require logical deduction or multi-step thought processes, which are routed to more powerful models.
- internet_search: For questions needing current information, which automatically triggers a Google Search to provide an up-to-date answer.

The code is under the MIT license and available on Github: (https://github.com/mahtabsyed/21-Agentic-Patterns/blob/main/16_Resource_Aware_Opt_LLM_Reflection_v2.ipynb)

```
# MIT License
# Copyright (c) 2025 Mahtab Syed
# https://www.linkedin.com/in/mahtabsyed/

import os
import requests
import json
from dotenv import load_dotenv
from openai import OpenAI

# Load environment variables
load_dotenv()
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
GOOGLE_CUSTOM_SEARCH_API_KEY =
os.getenv("GOOGLE_CUSTOM_SEARCH_API_KEY")
GOOGLE_CSE_ID = os.getenv("GOOGLE_CSE_ID")

if not OPENAI_API_KEY or not GOOGLE_CUSTOM_SEARCH_API_KEY or not
GOOGLE_CSE_ID:
   raise ValueError(
       "Please set OPENAI_API_KEY, GOOGLE_CUSTOM_SEARCH_API_KEY, and 
GOOGLE_CSE_ID in your .env file."
   )

client = OpenAI(api_key=OPENAI_API_KEY)

# --- Step 1: Classify the Prompt ---
def classify_prompt(prompt: str) -> dict:
   system_message = {
       "role": "system",
       "content": (
           "You are a classifier that analyzes user prompts and 
returns one of three categories ONLY:\n\n"
           "- simple\n"
           "- reasoning\n"
           "- internet_search\n\n"
           "Rules:\n"
           "- Use 'simple' for direct factual questions that need no 
reasoning or current events.\n"
           "- Use 'reasoning' for logic, math, or multi-step 
inference questions.\n"
           "- Use 'internet_search' if the prompt refers to current 
events, recent data, or things not in your training data.\n\n"
           "Respond ONLY with JSON like:\n"
           '{ "classification": "simple" }'
       ),
   }

   user_message = {"role": "user", "content": prompt}

   response = client.chat.completions.create(
       model="gpt-4o", messages=[system_message, user_message], 
temperature=1
   )

   reply = response.choices[0].message.content
   return json.loads(reply)

# --- Step 2: Google Search ---
def google_search(query: str, num_results=1) -> list:
   url = "https://www.googleapis.com/customsearch/v1"
   params = {
       "key": GOOGLE_CUSTOM_SEARCH_API_KEY,
       "cx": GOOGLE_CSE_ID,
       "q": query,
       "num": num_results,
   }
```

```
   try:
       response = requests.get(url, params=params)
       response.raise_for_status()
       results = response.json()


       if "items" in results and results["items"]:
           return [
               {
                   "title": item.get("title"),
                   "snippet": item.get("snippet"),
                   "link": item.get("link"),
               }
               for item in results["items"]
           ]
       else:
           return []
   except requests.exceptions.RequestException as e:
       return {"error": str(e)}

# --- Step 3: Generate Response ---
def generate_response(prompt: str, classification: str,
search_results=None) -> str:
   if classification == "simple":
       model = "gpt-4o-mini"
       full_prompt = prompt
   elif classification == "reasoning":
       model = "o4-mini"
       full_prompt = prompt
   elif classification == "internet_search":
       model = "gpt-4o"
       # Convert each search result dict to a readable string
       if search_results:
           search_context = "\n".join(
               [
                   f"Title: {item.get('title')}\nSnippet:
{item.get('snippet')}\nLink: {item.get('link')}"
                   for item in search_results
               ]
           )
       else:
           search_context = "No search results found."
       full_prompt = f"""Use the following web results to answer the
user query:

{search_context}
```

```
Query: {prompt}"""

   response = client.chat.completions.create(
       model=model,
       messages=[{"role": "user", "content": full_prompt}],
       temperature=1,
   )

   return response.choices[0].message.content, model

# --- Step 4: Combined Router ---
def handle_prompt(prompt: str) -> dict:
   classification_result = classify_prompt(prompt)
   # Remove or comment out the next line to avoid duplicate printing
   # print("\n🔍 Classification Result:", classification_result)
   classification = classification_result["classification"]

   search_results = None
   if classification == "internet_search":
       search_results = google_search(prompt)
       # print("\n🔍 Search Results:", search_results)

   answer, model = generate_response(prompt, classification,
search_results)
   return {"classification": classification, "response": answer,
"model": model}
test_prompt = "What is the capital of Australia?"
# test_prompt = "Explain the impact of quantum computing on
cryptography."
# test_prompt = "When does the Australian Open 2026 start, give me
full date?"

result = handle_prompt(test_prompt)
print("🔍 Classification:", result["classification"])
print("🧠 Model Used:", result["model"])
print("🧠 Response:\n", result["response"])
```

This Python code implements a prompt routing system to answer user questions. It begins by loading necessary API keys from a .env file for OpenAI and Google Custom Search. The core functionality lies in classifying the user's prompt into three categories: simple, reasoning, or internet search. A dedicated function utilizes an OpenAI model for this classification step. If the prompt requires current information, a Google search is performed using the Google Custom Search API. Another function then generates the final response, selecting an appropriate OpenAI model based on the classification. For internet search queries, the search results are provided as context to the model. The main handle_prompt function orchestrates this workflow, calling the classification and search (if needed) functions before generating the response. It returns the classification, the model used, and the generated answer. This system efficiently directs different types of queries to optimized methods for a better response.

Hands-On Code Example (OpenRouter)

OpenRouter offers a unified interface to hundreds of AI models via a single API endpoint. It provides automated failover and cost-optimization, with easy integration through your preferred SDK or framework.

```
import requests
import json
response = requests.post(
 url="https://openrouter.ai/api/v1/chat/completions",
 headers={
   "Authorization": "Bearer <OPENROUTER_API_KEY>",
   "HTTP-Referer": "<YOUR_SITE_URL>", # Optional. Site URL for
rankings on openrouter.ai.
   "X-Title": "<YOUR_SITE_NAME>", # Optional. Site title for rankings
on openrouter.ai.
 },
 data=json.dumps({
   "model": "openai/gpt-4o", # Optional
   "messages": [
     {
       "role": "user",
       "content": "What is the meaning of life?"
     }
   ]
 })
)
```

This code snippet uses the requests library to interact with the OpenRouter API. It sends a POST request to the chat completion endpoint with a user message. The request includes authorization headers with an API key and optional site information. The goal is to get a response from a specified language model, in this case, "openai/gpt-4o".

Openrouter offers two distinct methodologies for routing and determining the computational model used to process a given request.

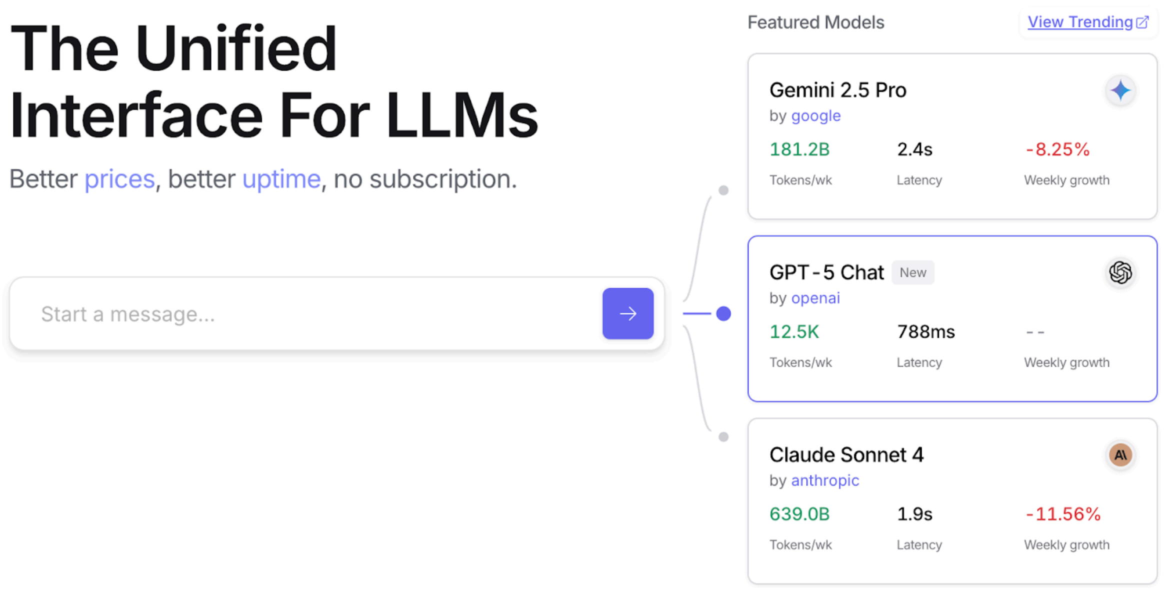
- **Automated Model Selection:** This function routes a request to an optimized model chosen from a curated set of available models. The selection is predicated on the specific content of the user's prompt. The identifier of the model that ultimately processes the request is returned in the response's metadata.

```
{
 "model": "openrouter/auto",
 ... // Other params
}
```

- **Sequential Model Fallback:** This mechanism provides operational redundancy by allowing users to specify a hierarchical list of models. The system will first attempt to process the request with the primary model designated in the sequence. Should this primary model fail to respond due to any number of error conditions—such as service unavailability, rate-limiting, or content filtering—the system will automatically re-route the request to the next specified model in the sequence. This process continues until a model in the list successfully executes the request or the list is exhausted. The final cost of the operation and the model identifier returned in the response will correspond to the model that successfully completed the computation.

```
{
 "models": ["anthropic/claude-3.5-sonnet", "gryphe/mythomax-l2-13b"],
 ... // Other params
}
```

OpenRouter offers a detailed leaderboard (https://openrouter.ai/rankings) which ranks available AI models based on their cumulative token production. It also offers latest models from different providers (ChatGPT, Gemini, Claude) (see Fig. 1)


The Unified Interface For LLMs

Better prices, better uptime, no subscription.

Start a message...

Featured Models

View Trending

Gemini 2.5 Pro by google	181.2B Tokens/wk	2.4s Latency	-8.25% Weekly growth
GPT-5 Chat New by openai	12.5K Tokens/wk	788ms Latency	-- Weekly growth
Claude Sonnet 4 by anthropic	639.0B Tokens/wk	1.9s Latency	-11.56% Weekly growth

Fig. 1: OpenRouter Web site (https://openrouter.ai/)

Beyond Dynamic Model Switching: A Spectrum of Agent Resource Optimizations

Resource-aware optimization is paramount in developing intelligent agent systems that operate efficiently and effectively within real-world constraints. Let's see a number of additional techniques:

Dynamic Model Switching is a critical technique involving the strategic selection of large language models based on the intricacies of the task at hand and the available computational resources. When faced with simple queries, a lightweight, cost-effective LLM can be deployed, whereas complex, multifaceted problems necessitate the utilization of more sophisticated and resource-intensive models.

Adaptive Tool Use & Selection ensures agents can intelligently choose from a suite of tools, selecting the most appropriate and efficient one for each specific sub-task, with careful consideration given to factors like API usage costs, latency, and execution time. This dynamic tool selection enhances overall system efficiency by optimizing the use of external APIs and services.

Contextual Pruning & Summarization plays a vital role in managing the amount of information processed by agents, strategically minimizing the prompt token count and reducing inference costs by intelligently summarizing and selectively retaining only the most relevant information from the interaction history, preventing unnecessary computational overhead.

Proactive Resource Prediction involves anticipating resource demands by forecasting future workloads and system requirements, which allows for proactive allocation and management of resources, ensuring system responsiveness and preventing bottlenecks.

Cost-Sensitive Exploration in multi-agent systems extends optimization considerations to encompass communication costs alongside traditional computational costs, influencing the strategies employed by agents to collaborate and share information, aiming to minimize the overall resource expenditure.

Energy-Efficient Deployment is specifically tailored for environments with stringent resource constraints, aiming to minimize the energy footprint of intelligent agent systems, extending operational time and reducing overall running costs.

Parallelization & Distributed Computing Awareness leverages distributed resources to enhance the processing power and throughput of agents, distributing computational workloads across multiple machines or processors to achieve greater efficiency and faster task completion.

Learned Resource Allocation Policies introduce a learning mechanism, enabling agents to adapt and optimize their resource allocation strategies over time based on feedback and performance metrics, improving efficiency through continuous refinement.

Graceful Degradation and Fallback Mechanisms ensure that intelligent agent systems can continue to function, albeit perhaps at a reduced capacity, even when resource constraints are severe, gracefully degrading performance and falling back to alternative strategies to maintain operation and provide essential functionality.

At a Glance

What: Resource-Aware Optimization addresses the challenge of managing the consumption of computational, temporal, and financial resources in intelligent systems. LLM-based applications can be expensive and slow, and selecting the best model or tool for every task is often inefficient. This creates a fundamental trade-off between the quality of a system's output and the resources required to produce it.

Without a dynamic management strategy, systems cannot adapt to varying task complexities or operate within budgetary and performance constraints.

Why: The standardized solution is to build an agentic system that intelligently monitors and allocates resources based on the task at hand. This pattern typically employs a "Router Agent" to first classify the complexity of an incoming request. The request is then forwarded to the most suitable LLM or tool—a fast, inexpensive model for simple queries, and a more powerful one for complex reasoning. A "Critique Agent" can further refine the process by evaluating the quality of the response, providing feedback to improve the routing logic over time. This dynamic, multi-agent approach ensures the system operates efficiently, balancing response quality with cost-effectiveness.

Rule of thumb: Use this pattern when operating under strict financial budgets for API calls or computational power, building latency-sensitive applications where quick response times are critical, deploying agents on resource-constrained hardware such as edge devices with limited battery life, programmatically balancing the trade-off between response quality and operational cost, and managing complex, multi-step workflows where different tasks have varying resource requirements.

Visual Summary

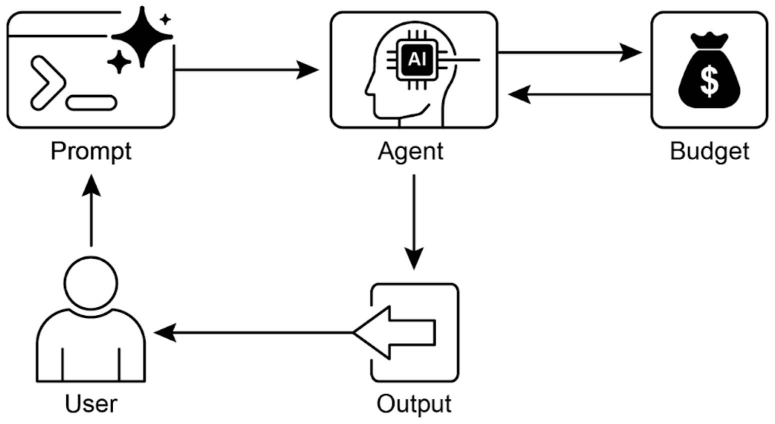


Fig. 2: Resource-Aware Optimization Design Pattern

Key Takeaways

- Resource-Aware Optimization is Essential: Intelligent agents can manage computational, temporal, and financial resources dynamically. Decisions regarding model usage and execution paths are made based on real-time constraints and objectives.
- Multi-Agent Architecture for Scalability: Google's ADK provides a multi-agent framework, enabling modular design. Different agents (answering, routing, critique) handle specific tasks.
- Dynamic, LLM-Driven Routing: A Router Agent directs queries to language models (Gemini Flash for simple, Gemini Pro for complex) based on query complexity and budget. This optimizes cost and performance.
- Critique Agent Functionality: A dedicated Critique Agent provides feedback for self-correction, performance monitoring, and refining routing logic, enhancing system effectiveness.

- Optimization Through Feedback and Flexibility: Evaluation capabilities for critique and model integration flexibility contribute to adaptive and self-improving system behavior.
- Additional Resource-Aware Optimizations: Other methods include Adaptive Tool Use & Selection, Contextual Pruning & Summarization, Proactive Resource Prediction, Cost-Sensitive Exploration in Multi-Agent Systems, Energy-Efficient Deployment, Parallelization & Distributed Computing Awareness, Learned Resource Allocation Policies, Graceful Degradation and Fallback Mechanisms, and Prioritization of Critical Tasks.

Conclusions

Resource-aware optimization is essential for the development of intelligent agents, enabling efficient operation within real-world constraints. By managing computational, temporal, and financial resources, agents can achieve optimal performance and cost-effectiveness. Techniques such as dynamic model switching, adaptive tool use, and contextual pruning are crucial for attaining these efficiencies. Advanced strategies, including learned resource allocation policies and graceful degradation, enhance an agent's adaptability and resilience under varying conditions. Integrating these optimization principles into agent design is fundamental for building scalable, robust, and sustainable AI systems.

References

1. Google's Agent Development Kit (ADK): https://google.github.io/adk-docs/
2. Gemini Flash 2.5 & Gemini 2.5 Pro: https://aistudio.google.com/
3. OpenRouter: https://openrouter.ai/docs/quickstart