Optimizing Research Architectures: A Comprehensive Redesign of the Research Coordinator Pipeline

Executive Summary

In the rapidly evolving landscape of automated information retrieval, the efficiency of the underlying architecture dictates not only the operational cost but also the user experience. The legacy implementation of the research_coordinator.py system—characterized by a monolithic, brute-force execution strategy—presents a critical bottleneck in scaling our research capabilities. Currently, the system operates on a naive "fire-and-forget" principle, blindly triggering expensive multi-modal retrieval processes (Web Search and YouTube Data API) for every user query, regardless of complexity or redundancy. This approach has resulted in an unsustainable cost per query of approximately $0.50 and unacceptable latency metrics due to sequential blocking I/O operations.

This report details a comprehensive architectural redesign aimed at transforming this legacy pipeline into an intelligent, state-aware research engine. The proposed solution introduces three fundamental engineering paradigms: **Semantic Caching**, **Intelligent Model Routing**, and **Asynchronous Parallelization**. By implementing a semantic cache layer backed by vector embeddings, the system can identify and intercept redundant queries—such as recognizing that "How to mix 808s" and "808 mixing guide" are semantically equivalent—thereby serving instant results at near-zero cost. For novel queries, a lightweight routing layer utilizing gemini-2.0-flash-lite acts as a cognitive dispatcher, categorizing user intent into "Simple Retrieval," "Complex Technique," or "Specific Fact," and activating only the necessary tools. Finally, the refactoring of sequential await chains into non-blocking asyncio.gather patterns ensures that unavoidable I/O operations are executed concurrently, minimizing total request latency to the duration of the single slowest component.

The following analysis provides a rigorous technical breakdown of each component, supported by code implementations, theoretical frameworks, and performance projections. It demonstrates that this redesign will not only reduce operational costs by an estimated 60-80% for high-traffic workloads but also fundamentally enhance the system's responsiveness and scalability.

1. Problem Definition and Architectural Analysis

1.1 The Legacy Monolith: Anatomy of Inefficiency

The existing research pipeline represents a classic "Generation 1" AI application structure. In this paradigm, the Large Language Model (LLM) is treated as a universal hammer, and every

user interaction is a nail. The logic flow is linear and indiscriminate: upon receiving a query, the system immediately instantiates high-latency network requests to external providers (Google Search, YouTube), awaits their completion one by one, and then passes the aggregated massive context window to an expensive reasoning model for synthesis.

This architecture suffers from three primary failure modes:

1. **Redundant Computation:** The system lacks memory. If ten users ask "What is the capital of France?" in ten minutes, the system performs ten identical web searches and ten identical LLM inference passes. This O(N) cost scaling relative to user volume is financially hazardous.[1]
2. **Resource Over-Provisioning:** A query requiring a simple fact check (e.g., "internal server IP address") triggers the same heavy-lifting machinery as a complex research request (e.g., "comparative analysis of sorting algorithms"). Using a "smart" model for "dumb" tasks is a misallocation of GPU resources and budget.[2]
3. **Blocking Latency:** The sequential execution pattern (await web -> await youtube) ensures that the user waits for the sum of all service latencies ($L_{total} = L_{web} + L_{yt}$). In a distributed system, latency should ideally be governed by the slowest single dependency ($\max(L_{web}, L_{yt})$), not their aggregate.[4]

1.2 The Economic Imperative for Redesign

The current unit economic model is unsustainable. At ~$0.50 per query, a modest user base of 1,000 daily active users (DAU) generating 5 queries each results in a daily burn rate of $2,500, or roughly $75,000 monthly.

Table 1: Cost Breakdown of Naive vs. Optimized Architecture (Projected)

Component	Naive Cost / Query	Optimized Cost / Query	Logic for Reduction
Orchestration	$0.00	$0.005	Introduction of gemini-2.0-flash-lite router.
Retrieval (Web)	$0.02	$0.01	Reduced volume via caching and routing logic.
Retrieval (Video)	$0.02	$0.005	Only triggered for "Complex

			Technique" queries.
Synthesis (LLM)	$0.46	$0.05	Expensive model replaced/minimized ; cached hits cost $0.
Total	**~$0.50**	**~$0.07**	**~86% Reduction**

The transition to an intelligent pipeline is not merely an optimization; it is a requirement for financial viability.

2. Theoretical Framework: Semantic Intelligence in Search

To address the requirement for "Semantic Caching," we must first establish the theoretical underpinnings of how a machine "understands" that two different strings of text represent the same intent.

2.1 Vector Embeddings and High-Dimensional Spaces

Traditional caching mechanisms, such as Redis, rely on exact string matching (hashing). If the cache key is "how to mix 808s", a query for "mixing 808s tutorial" will result in a cache miss. This is insufficient for natural language interfaces.

Semantic caching utilizes **Vector Embeddings**. An embedding model (such as OpenAI's text-embedding-3-small or open-source equivalents like all-mpnet-base-v2 [6]) transforms text input into a fixed-size vector of floating-point numbers (e.g., 1,536 dimensions). In this high-dimensional space, semantic meaning is encoded in geometric proximity. Concepts that are semantically similar are mapped to points that are physically closer together in the vector space.[7]

2.2 Cosine Similarity as a Metric

To quantify "closeness," we employ **Cosine Similarity**. Unlike Euclidean distance, which measures the straight-line distance between points (and can be affected by the magnitude of the vectors, often representing document length), Cosine Similarity measures the cosine of the angle between two vectors.[1]

The formula for Cosine Similarity between vector A and vector B is:

$$\text{Similarity}(A, B) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

- **1.0**: The vectors are identical (angle is 0°).
- **0.0**: The vectors are orthogonal (angle is 90°, unrelated meanings).
- **-1.0**: The vectors are diametrically opposed (angle is 180°).

For our research coordinator, we will define a **similarity threshold** (e.g., 0.90). If an incoming query's vector has a cosine similarity of >0.90 with a stored query vector, we consider it a "hit" and return the stored response. This allows "How to mix 808s" and "808 mixing guide" to be treated as identical requests.[1]

3. Component 1: The Semantic Caching Layer

The first line of defense in our redesigned architecture is the Semantic Cache. This component intercepts user queries before they reach any API-consuming logic.

3.1 Storage Strategy: Local Vector Store

While production systems processing millions of vectors require dedicated databases like ChromaDB, Milvus, or Pinecone, the requirement here calls for a demonstrable implementation. We will architect a modular solution that can operate with a lightweight local JSON store (using NumPy for calculations) while maintaining API compatibility with ChromaDB for future scaling.[1]

Architectural Decision: We will use a "Look-Aside" cache pattern.

1. **Read-Through:** The coordinator asks the cache for the query.
2. **Hit:** Cache returns data; Coordinator returns immediately.
3. **Miss:** Coordinator executes research; Coordinator writes result to cache asynchronously.

3.2 Implementation: The SemanticCache Class

The following Python implementation utilizes a local JSON file for persistence and manual Cosine Similarity calculation. This satisfies the requirement for a "simple JSON + Cosine Similarity script" while structuring it as a robust class that can be integrated into the research_coordinator.py.[1]

Python

```
import json
```

```python
import numpy as np
import os
from typing import List, Dict, Optional, Tuple
from datetime import datetime

# In a real scenario, import your embedding client (e.g., OpenAI, SentenceTransformers)
# from sentence_transformers import SentenceTransformer

class SemanticCache:
    """
    A lightweight semantic cache using local JSON storage and Cosine Similarity.
    Designed to intercept redundant research queries.
    """
    def __init__(self, cache_file: str = "semantic_cache.json", threshold: float = 0.92):
        self.cache_file = cache_file
        self.threshold = threshold
        # self.encoder = SentenceTransformer('all-mpnet-base-v2') # Uncomment for real embeddings
        self.cache_data = self._load_cache()

    def _load_cache(self) -> List:
        """Loads the cache from disk."""
        if not os.path.exists(self.cache_file):
            return
        try:
            with open(self.cache_file, 'r') as f:
                return json.load(f)
        except json.JSONDecodeError:
            return

    def _save_cache(self):
        """Persists the cache to disk."""
        with open(self.cache_file, 'w') as f:
            json.dump(self.cache_data, f, indent=2)

    def _get_embedding(self, text: str) -> List[float]:
        """
        Generates a vector embedding for the given text.
        For this report's simulation, we return a normalized random vector.
        In production, replace with: return self.encoder.encode(text).tolist()
        """
        # Mocking embedding for demonstration purposes
        # Utilizing a deterministic seed based on text hash to simulate stability
        np.random.seed(hash(text) % 2**32)
        vector = np.random.rand(768)
```

```
        return (vector / np.linalg.norm(vector)).tolist()

    def _cosine_similarity(self, vec_a: List[float], vec_b: List[float]) -> float:
        """Calculates Cosine Similarity between two vectors."""
        a = np.array(vec_a)
        b = np.array(vec_b)
        if np.linalg.norm(a) == 0 or np.linalg.norm(b) == 0:
            return 0.0
        return np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))

    def lookup(self, query: str) -> Optional[str]:
        """
        Searches the cache for a semantically similar query.
        Returns the cached response if a match > threshold is found.
        """
        query_vector = self._get_embedding(query)
        best_score = -1.0
        best_entry = None

        print(f"--- Cache Lookup for: '{query}' ---")

        for entry in self.cache_data:
            score = self._cosine_similarity(query_vector, entry['vector'])

            # Log scores for debugging visibility
            # print(f"Checking against: '{entry['query']}' | Score: {score:.4f}")

            if score > best_score:
                best_score = score
                best_entry = entry

        if best_score >= self.threshold:
            print(f"✅ CACHE HIT. Matched: '{best_entry['query']}' (Score: {best_score:.4f})")
            return best_entry['response']

        print(f"❌ CACHE MISS. Nearest neighbor score: {best_score:.4f}")
        return None

    def store(self, query: str, response: str):
        """Stores a new query-response pair in the cache."""
        vector = self._get_embedding(query)
        new_entry = {
            "query": query,
```

```
            "vector": vector,
            "response": response,
            "timestamp": datetime.utcnow().isoformat()
        }
        self.cache_data.append(new_entry)
        self._save_cache()
        print(f"💾 Stored result for '{query}' in cache.")

# Example Usage Simulation
if __name__ == "__main__":
    cache = SemanticCache()

    # 1. First Query (Cache Miss)
    q1 = "How to mix 808s"
    if not cache.lookup(q1):
        # Simulate fetching data
        fake_response = "To mix 808s, use saturation, sidechain compression, and EQ."
        cache.store(q1, fake_response)

    # 2. Second Query (Semantically Similar - Should Hit)
    q2 = "808 mixing guide"
    # Note: With random mock embeddings, this won't mathematically match,
    # but logically this is the flow. In a real system, vectors for q1 and q2 would be close.
    result = cache.lookup(q2)
```

3.3 Insight: The "Memory" Effect

Implementing this layer fundamentally changes the nature of the research tool. It evolves from a stateless functional pipeline into a system with "memory." Over time, the vector store accumulates a knowledge base specific to the user's domain. This has second-order implications for **personalization**: future iterations could weight the similarity search not just by query content, but by user preference vectors, effectively prioritizing results that align with the user's past successful interactions.[11]

However, this introduces the challenge of **Cache Invalidation**. Unlike data in a database, research results expire. A cached answer for "NVIDIA stock price" is valid for minutes; "How to mix 808s" is valid for years. The timestamp field in the store method is crucial here. The lookup logic can be enhanced to check max_cache_age_hours based on the query type (e.g., News = 1 hour, Tutorials = 1 year).[10]

4. Component 2: Intelligent Model Routing

When the cache returns a miss, the system must decide *how* to research the topic. The naive

system's flaw is treating all misses equally. The optimized system introduces a "Brain" (Router) to classify intent.

4.1 Selection of the Classifier: Gemini 2.0 Flash-Lite

For the routing layer, we require a model that is extremely fast (low Time-To-First-Token) and extremely cheap, as it will run on every non-cached query. We have selected **Gemini 2.0 Flash-Lite** for this role.

- **Latency:** It is optimized for high-throughput, low-latency tasks, making it ideal for middleware classification where every millisecond of delay blocks the user.[12]
- **Cost Efficiency:** It is significantly cheaper than "Pro" or "Thinking" models, allowing us to maintain the "Orchestration" cost line item at negligible levels.[14]
- **Structured Output:** A critical requirement for programmatic routing is reliable output. We cannot parse free text. Gemini 2.0 supports response_schema natively, allowing us to enforce a Pydantic schema for the routing decision.[12]

4.2 The Taxonomy of Intent

We define three distinct research strategies based on the user's intent. This taxonomy covers the spectrum of information retrieval needs.[2]

Table 2: Research Intent Categories and Routing Logic

Category	Intent Description	Tooling Strategy	Why?
Simple Retrieval	Queries for singular facts, recent news events, weather, stock prices, or surface-level definitions.	web_search Only	YouTube is noisy and inefficient for simple facts. Reasoning models are overkill.
Complex Technique	Procedural "How-to" requests, tutorials, deep dives into skills, educational content, or multi-faceted analysis.	web_search + youtube_search	Video content is often superior for procedural learning (e.g., coding, cooking, music production).

Specific Fact	Queries referencing internal proprietary data, company acronyms, project codes, or specific codified knowledge.	internal_kb Only	Public web search will hallucinate or fail on private data. Security requires isolating this path.

4.3 Implementation: The ResearchPolicy Class

This class encapsulates the routing logic. It interacts with the Gemini API to classify the query and returns a RoutePlan object. We use Pydantic to define the strict schema, ensuring the LLM never returns malformed JSON.[16]

Python

```
import os
from enum import Enum
from typing import List, Optional
from pydantic import BaseModel, Field
# Assuming usage of google-genai SDK
# from google import genai

class ResearchCategory(str, Enum):
    SIMPLE = "Simple Retrieval"
    COMPLEX = "Complex Technique"
    SPECIFIC = "Specific Fact"

class RoutePlan(BaseModel):
    category: ResearchCategory = Field(..., description="The classified category of the user's request.")
    reasoning: str = Field(..., description="A brief explanation of why this category was chosen.")
    tools: List[str] = Field(..., description="The list of tools strictly required for this category.")
    search_queries: List[str] = Field(..., description="Optimized search queries to use for the tools.")

class ResearchPolicy:
    """
    The decision-making engine of the pipeline.
    Uses Gemini 2.0 Flash-Lite to categorize user intent.
```

```
    """
    def __init__(self, api_key: str):
        self.api_key = api_key
        # self.client = genai.Client(api_key=api_key) # Initialize Gemini Client
        self.model_id = "gemini-2.0-flash-lite"

    async def determine_route(self, user_query: str) -> RoutePlan:
        """
        Classifies the user query into a research strategy.
        """
        print(f"🤔 Routing Query via {self.model_id}...")

        system_instruction = """
        You are an expert Research Coordinator. Your task is to analyze the user's query and route it to the most efficient data sources.

        CLASSIFICATION RULES:
        1. Simple Retrieval: Quick facts, news, definitions, surface-level info. Tool: ['web_search'].
        2. Complex Technique: Tutorials, how-to guides, deep learning, skill acquisition, or multi-modal synthesis. Tools: ['web_search', 'youtube_search'].
        3. Specific Fact: Internal company data, proprietary knowledge, project codes, employee details. Tool: ['internal_kb'].

        OUTPUT:
        Return a strict JSON object matching the RoutePlan schema.
        Generate 1-2 optimized search queries based on the intent.
        """

        # Mocking the API call for this report simulation
        # In production:
        # response = await self.client.models.generate_content_async(
        #     model=self.model_id,
        #     contents=user_query,
        #     config={
        #         "response_mime_type": "application/json",
        #         "response_schema": RoutePlan,
        #         "system_instruction": system_instruction
        #     }
        # )
        # return response.parsed

        # Simulation Logic based on keywords to demonstrate flow
        lower_q = user_query.lower()
        if "mix" in lower_q or "tutorial" in lower_q or "guide" in lower_q:
```

```
        return RoutePlan(
            category=ResearchCategory.COMPLEX,
            reasoning="Query implies procedural knowledge acquisition best served by video and text.",
            tools=["web_search", "youtube_search"],
            search_queries=[user_query, f"{user_query} tutorial"]
        )
    elif "internal" in lower_q or "schedule" in lower_q or "q3" in lower_q:
        return RoutePlan(
            category=ResearchCategory.SPECIFIC,
            reasoning="Query requests proprietary/internal information.",
            tools=["internal_kb"],
            search_queries=[user_query]
        )
    else:
        return RoutePlan(
            category=ResearchCategory.SIMPLE,
            reasoning="Query seeks general factual information available on the public web.",
            tools=["web_search"],
            search_queries=[user_query]
        )
```

5. Component 3: Asynchronous Parallelization

The final piece of the redesign addresses the latency bottleneck. The legacy system's sequential await pattern effectively serialized operations that had no interdependencies.

5.1 The Physics of I/O Bound Concurrency

In Python, the Global Interpreter Lock (GIL) prevents CPU-bound tasks from running in parallel on a single thread. However, network requests (API calls) are **I/O bound**. When an await keyword is encountered for a network request, the event loop releases the control, allowing other tasks to run.

By switching from sequential await to asyncio.gather, we schedule all necessary API calls on the event loop simultaneously.

- **Sequential Time:** $T_{total} = T_{web} + T_{youtube} + T_{kb}$
- **Parallel Time:** $T_{total} = \max(T_{web}, T_{youtube}, T_{kb}) + \epsilon$ (where ϵ is overhead)

5.2 Handling Failures in Parallel Streams

A robust distributed system must handle partial failure. If the YouTube API is down or rate-limited, the Web Search should still proceed. asyncio.gather provides the

return_exceptions=True parameter. This ensures that if one task raises an Exception, it is caught and returned in the results list, rather than crashing the entire gathering process.[18]

5.3 Implementation: The Refactored ResearchCoordinator

This module integrates the Cache and Policy to execute the research.

Python

```python
import asyncio
from typing import List, Any

# Mocking external tools
async def search_web(query: str) -> str:
    print(f"   🌐 Searching Web for: {query}")
    await asyncio.sleep(1.5) # Simulate 1.5s latency
    return f"Web Results: Top hits for {query}..."

async def search_youtube(query: str) -> str:
    print(f"   📺 Searching YouTube for: {query}")
    await asyncio.sleep(2.0) # Simulate 2.0s latency
    return f"YouTube Results: Video guide for {query}..."

async def search_internal_kb(query: str) -> str:
    print(f"   🗄️ Searching Internal KB for: {query}")
    await asyncio.sleep(0.5) # Simulate 0.5s latency (faster internal network)
    return f"Internal KB: Confidential document regarding {query}..."

class ResearchCoordinator:
    def __init__(self, cache: SemanticCache, policy: ResearchPolicy):
        self.cache = cache
        self.policy = policy

    async def execute_research(self, user_query: str) -> str:
        print(f"\n🚀 STARTING RESEARCH: '{user_query}'")

        # 1. Semantic Cache Check
        cached_result = self.cache.lookup(user_query)
        if cached_result:
            return f"\n{cached_result}"
```

```
        # 2. Intelligent Routing
        plan = await self.policy.determine_route(user_query)
        print(f"📋 PLAN: {plan.category.value} | Tools: {plan.tools}")

        # 3. Parallel Execution via asyncio.gather
        tasks =

        # Dispatch tasks based on the plan
        # Note: We use the optimized search queries generated by the router
        primary_query = plan.search_queries

        if "web_search" in plan.tools:
            tasks.append(search_web(primary_query))

        if "youtube_search" in plan.tools:
            tasks.append(search_youtube(primary_query))

        if "internal_kb" in plan.tools:
            tasks.append(search_internal_kb(primary_query))

        # EXECUTE PARALLEL FETCH
        print(f"⚡ Firing {len(tasks)} tasks in parallel...")
        raw_results = await asyncio.gather(*tasks, return_exceptions=True)

        # 4. Result Synthesis & Error Handling
        synthesized_data = self._process_results(raw_results, plan)

        # 5. Update Cache
        self.cache.store(user_query, synthesized_data)

        return synthesized_data

    def _process_results(self, results: List[Any], plan: RoutePlan) -> str:
        """
        Combines results, filtering out failed tasks.
        """
        valid_content =
        errors =

        for res in results:
            if isinstance(res, Exception):
                errors.append(str(res))
```

```
            else:
                valid_content.append(res)

        if not valid_content:
            return "Research failed. All tools returned errors."

        # In a real app, you might send this content back to the LLM for final summarization.
        # Here we just format it.
        final_output = f"--- Research Report: {plan.category.value} ---\n"
        final_output += f"Reasoning: {plan.reasoning}\n\n"
        final_output += "\n\n".join(valid_content)

        if errors:
            final_output += f"\n\n: Some tools failed: {', '.join(errors)}"

        return final_output
```

6. End-to-End System Integration Example

To demonstrate the efficacy of the redesign, we simulate the pipeline processing the user's specific query examples.

6.1 Scenario 1: "How to mix 808s" (First Run)

1. **Cache:** Miss.
2. **Router:** Analysis detects "mix" and procedural intent. Classifies as **Complex Technique**.
3. **Plan:** Tools = [web_search, youtube_search].
4. **Execution:** search_web (1.5s) and search_youtube (2.0s) launch simultaneously.
5. **Latency:** Total wait is ~2.0s (bounded by YouTube).
6. **Result:** User gets a mix of articles and video guides. Result stored in Cache.

6.2 Scenario 2: "How to mix 808s" (Second Run)

1. **Cache:** Hit (Exact match or high similarity).
2. **Execution:** Return stored JSON immediately.
3. **Latency:** ~0.05s.
4. **Cost:** $0.00.

6.3 Scenario 3: "Internal Q3 Deployment Schedule"

1. **Cache:** Miss.
2. **Router:** Analysis detects "Internal" and "Schedule". Classifies as **Specific Fact**.
3. **Plan:** Tools = [internal_kb].
4. **Execution:** Only search_internal_kb is called. Public web APIs are **not** touched.

5. **Security:** Internal data is not leaked to public search prompts; public noise does not pollute the answer.

7. Performance Benchmarking and Future Outlook

7.1 Quantitative Analysis

Applying Amdahl's Law to our latency reduction:

If the sequential part of the program (Orchestration/Routing) takes 5% of the time, and the parallelizable part (Retrieval) takes 95%, parallelization offers massive speedups.

- **Original:** Routing (0s) + Web (1.5s) + YouTube (2.0s) = **3.5s**
- **New:** Routing (0.3s) + Max(Web 1.5s, YouTube 2.0s) = **2.3s**
  - *Improvement:* ~34% faster for complex queries.
  - *Improvement:* ~99% faster for cached queries.

7.2 Scalability and Migration to ChromaDB

As the dataset grows beyond 10,000 queries, the linear scan ($O(N)$) in our JSON SemanticCache will become a bottleneck. The architecture allows for a seamless swap to **ChromaDB**. Chroma uses **HNSW (Hierarchical Navigable Small World)** graphs to perform Approximate Nearest Neighbor (ANN) search in logarithmic time ($O(\log N)$).[6]

To migrate, the SemanticCache class lookup method would simply change from iterating a list to calling collection.query(query_embeddings=[vec], n_results=1). This future-proofing ensures the system remains viable as it scales to enterprise levels.

8. Conclusion

The transition from the legacy "naive" pipeline to this engineered "Router-Solver" architecture represents a maturation of the AI research tool. By respecting the heterogeneity of user requests—acknowledging that a request for a stock price differs fundamentally from a request for a tutorial—we optimize resource allocation. The introduction of **Semantic Caching** creates a system that learns and becomes more efficient with use, rather than one that perpetually repeats its work. The adoption of **Asyncio** ensures that the system's speed is limited only by external network physics, not internal mismanagement.

This redesign not only solves the immediate financial bleeding ($0.50/query -> ~$0.07/query) but establishes a robust, extensible foundation for future capabilities, such as personalized re-ranking and multi-agent collaboration. The system is no longer just a script; it is a scalable platform.

Works cited

1. Semantic Caching and Memory Patterns for Vector Databases ..., accessed February 10, 2026, https://www.dataquest.io/blog/semantic-caching-and-memory-patterns-for-vector-databases/
2. Bringing intelligent, efficient routing to open source AI with vLLM Semantic Router - Red Hat, accessed February 10, 2026, https://www.redhat.com/en/blog/bringing-intelligent-efficient-routing-open-source-ai-vllm-semantic-router
3. Beyond Basic RAG: Improving Your Knowledge Agents with Intent-Driven Architectures, accessed February 10, 2026, https://promptql.io/blog/beyond-basic-rag-promptqls-intent-driven-solution-to-query-inefficiencies
4. python 3.x - Asyncio gather difference - Stack Overflow, accessed February 10, 2026, https://stackoverflow.com/questions/73988859/asyncio-gather-difference
5. Mastering Python asyncio.gather and asyncio.as_completed for LLM ..., accessed February 10, 2026, https://python.useinstructor.com/blog/2023/11/13/learn-async/
6. Implementing semantic cache to improve a RAG system with FAISS ..., accessed February 10, 2026, https://huggingface.co/learn/cookbook/semantic_cache_chroma_vector_database
7. 6. Semantic Search with ChromaDB - Go Beyond Keywords Using Embeddings - YouTube, accessed February 10, 2026, https://m.youtube.com/watch?v=NVJ6rfQuwAY
8. Semantic Cache: How to Speed Up LLM and RAG Applications - Medium, accessed February 10, 2026, https://medium.com/@svosh2/semantic-cache-how-to-speed-up-llm-and-rag-applications-79e74ce34d1d
9. Implementing Cosine Similarity in Python - Tiger Data, accessed February 10, 2026, https://www.tigerdata.com/learn/implementing-cosine-similarity-in-python
10. How to Build Semantic Caching - OneUptime, accessed February 10, 2026, https://oneuptime.com/blog/post/2026-01-30-llmops-semantic-caching/view
11. Implementing Semantic Caching: A Step-by-Step Guide to Faster, Cost-Effective GenAI Workflows | by Arun Shankar | Google Cloud - Medium, accessed February 10, 2026, https://medium.com/google-cloud/implementing-semantic-caching-a-step-by-step-guide-to-faster-cost-effective-genai-workflows-ef85d8e72883
12. Gemini 2.0 Flash-Lite | Generative AI on Vertex AI - Google Cloud Documentation, accessed February 10, 2026, https://docs.cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-0-flash-lite
13. Gemini 2.0 Flash Lite - API, Providers, Stats - OpenRouter, accessed February 10, 2026, https://openrouter.ai/google/gemini-2.0-flash-lite-001
14. Gemini 2.0 model updates: 2.0 Flash, Flash-Lite, Pro Experimental - Google Blog,

accessed February 10, 2026, https://blog.google/innovation-and-ai/models-and-research/google-deepmind/gemini-model-updates-february-2025/
15. Structured outputs | Gemini API - Google AI for Developers, accessed February 10, 2026, https://ai.google.dev/gemini-api/docs/structured-output
16. gemini-samples/examples/gemini-structured-outputs.ipynb at main - GitHub, accessed February 10, 2026, https://github.com/philschmid/gemini-samples/blob/main/examples/gemini-structured-outputs.ipynb
17. Structured output - Gemini by Example, accessed February 10, 2026, https://geminibyexample.com/020-structured-output/
18. Exception handling in asyncio - Piccolo Blog, accessed February 10, 2026, https://piccolo-orm.com/blog/exception-handling-in-asyncio/
19. Asyncio gather() Handle Exceptions - Super Fast Python, accessed February 10, 2026, https://superfastpython.com/asyncio-gather-exception/