Writing High Performance AI Agents in Python: Insights from building Modulo

Architecting LLM Applications

In this talk we will explore design patterns for building LLM-based applications that have

1

Faster user request processing

2

Higher token throughput and low time to first token

3

Optimal system architectures for performance

4

... all without sacrificing response accuracy.

Not Covered in this Talk

Accuracy or Correctness of AI agents

Multi-agent co-operation and orchestration

... These are all separate talks in themselves ...

Why Performance Matters

Building high-performance LLM applications requires strategic thinking. This talk shares practical strategies from developing a coding agent that fixes bugs.

User Experience

Fast responses keep users engaged and satisfied

Cost Efficiency

Optimized systems reduce operational expenses

Scalability

Performance enables growth without bottlenecks

Mindset for LLM Development

Expect Failures

Anything can go wrong. Implement retries everywhere to handle unpredictable behavior.

Verify Outputs

Assume LLM output may be wrong. Add guardrails and validation depending on your application's needs.

Tokens Are Precious

Reduce token usage. Use regular programs where possible and lower-cost models for non-critical tasks.

Evaluate Everything

Add test cases, verify compilation, and validate results. Anything verifiable will eventually be solved by LLMs.

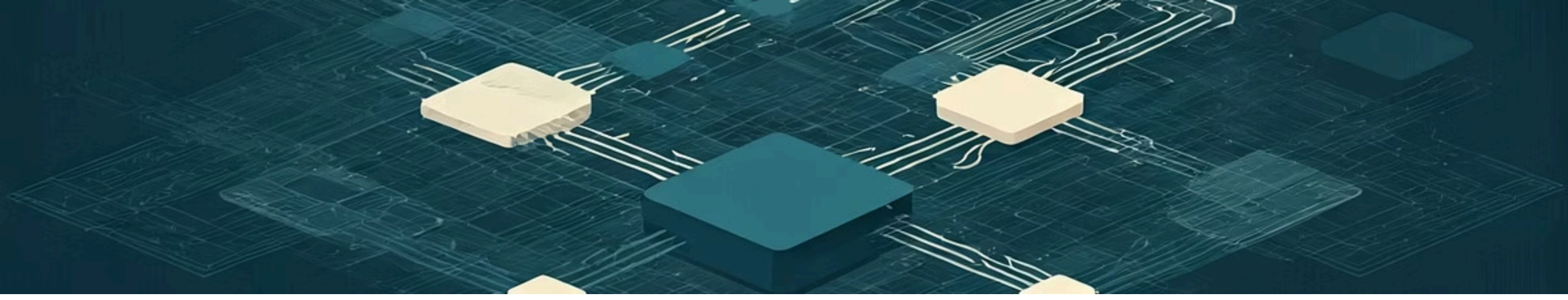
Embrace Parallelism

LLM applications benefit greatly from parallel processing strategies.

Know Your Workload

Understand dependencies, parallelization potential, and atomicity requirements for your specific use case.

Key Insight: The more you depend on LLMs for your product, the more uncertainty exists in accuracy. Balance is essential.



Design Patterns for Higher Token Throughput

When users request services—image generation, internet searches, or alerting agents—developers must consider critical factors:

Input Analysis

How many documents need processing? What's the total token count?

Output Requirements

How many output tokens must be delivered to the user?

Work Breakdown

Can the work be broken into stages for better efficiency?

Response Time

Does the user expect an instant response or can processing be asynchronous?

Classical Patterns Apply

When you spot multiple input documents, staged processing, and chunking possibilities, proven distributed system patterns become relevant:

Scatter Gather

Hadoop-style parallel processing

Pipelining

Microservices working independently

High Parallelism

Maximize concurrent operations

> But before you dive into implementation, always start with the math behind working with LLMs.

Do Your Own Math

Calculate realistic expectations before building. For a given function, how soon will an impatient user expect a response?

Standard Mode

Quick responses for simple queries

Deep Research Mode

Extended processing for complex tasks

An Example:

1

Initial Research

30 seconds upfront to gather sources and create a plan

2

Section Generation

10 seconds per section to process tokens and generate output for each section independently

3

Final Assembly

Combine all independent sections and reword into a cohesive narrative.

Each section output depends on the previous section, creating natural stages in your pipeline.

GPT 4.1 Case Study

What if we were to process 100,000 tokens in ONE LLM call v/s process them in 16 parallel LLM calls?

Time to First Token

Based on benchmarks it was found that each input token adds 0.24ms to the time to first token.

One LLM call would mean - 0.24 × 100 × 1000 = 24000 ms = **24s.**

16 parallel calls - 0.24 * 100 × 1000 / 16 = **1.5s**

Why this happens

When a prompt is submitted, the LLM processes the entire input through its transformer layers before outputting the first token. This phase, known as prefilling, requires attention mechanisms to compute interactions between each input token, which has quadratic complexity ($O(n^2)$ where n = number of tokens) in standard transformer architectures. However, in practice, advanced implementation optimizations make the observed increase much closer to linear for most prompt sizes within the allowed context window.

GPT 4.1 Case Study

What if we were to process 100,000 tokens in ONE LLM call v/s process them in 16 parallel LLM calls?

Lets say our number of expected output tokens is 10,000.

Decoding

GPT 4.1 generates approximately 132.7 output tokens per second.

This means generating each output token takes **7.5 ms**.

Generating 10,000 output tokens would mean - 10 × 1000 / 132.7 = **75.4 seconds.**

16 parallel calls - 75.4 / 16 = **5 seconds**.

GPT 4.1 Case Study

Total time to process

Prefill time + Decoding time

One LLM call (100K i/p, 10k o/p tokens) : 24 sec + 75.4 sec = 100 sec

16 parallel calls : 1.5 sec + 5 sec = 6.5 sec

Impact

Performance optimizations can determine whether your LLM application is usable or unusable.

Strategies to Improve Token Throughput

Process in Fixed Chunks

Break tasks into smaller LLM calls and run them in parallel

Batching

Use batch=True to process multiple requests efficiently

Context Compression

Summarize and compress context to reduce token usage

Memory Optimization

Progressive summarization for coding agents

Real-World Example

While building Modulo, the single biggest bottleneck was the LLM API. For 50 files, a naive approach sends one gigantic prompt and waits for tens of thousands of output tokens.

At 50 output tokens/sec, 30,000 tokens takes **600+ seconds (10 minutes)**.

What we did:

- Each file is decomposed into one or more chunks.
- LLM call for each chunk is done in a separate thread.

On a standard VM with 16 vCPUs, this means the LLM API generates:

16 × 50 = 800 output tokens per second which means the time is reduced to 40 seconds (from 10 minutes)!!

Demo of scatter-gather and its performance impact

> **Trade-off:** More smaller LLM calls means more prompts to stitch together. Outputs may conflict across chunks, requiring a final consistency pass.

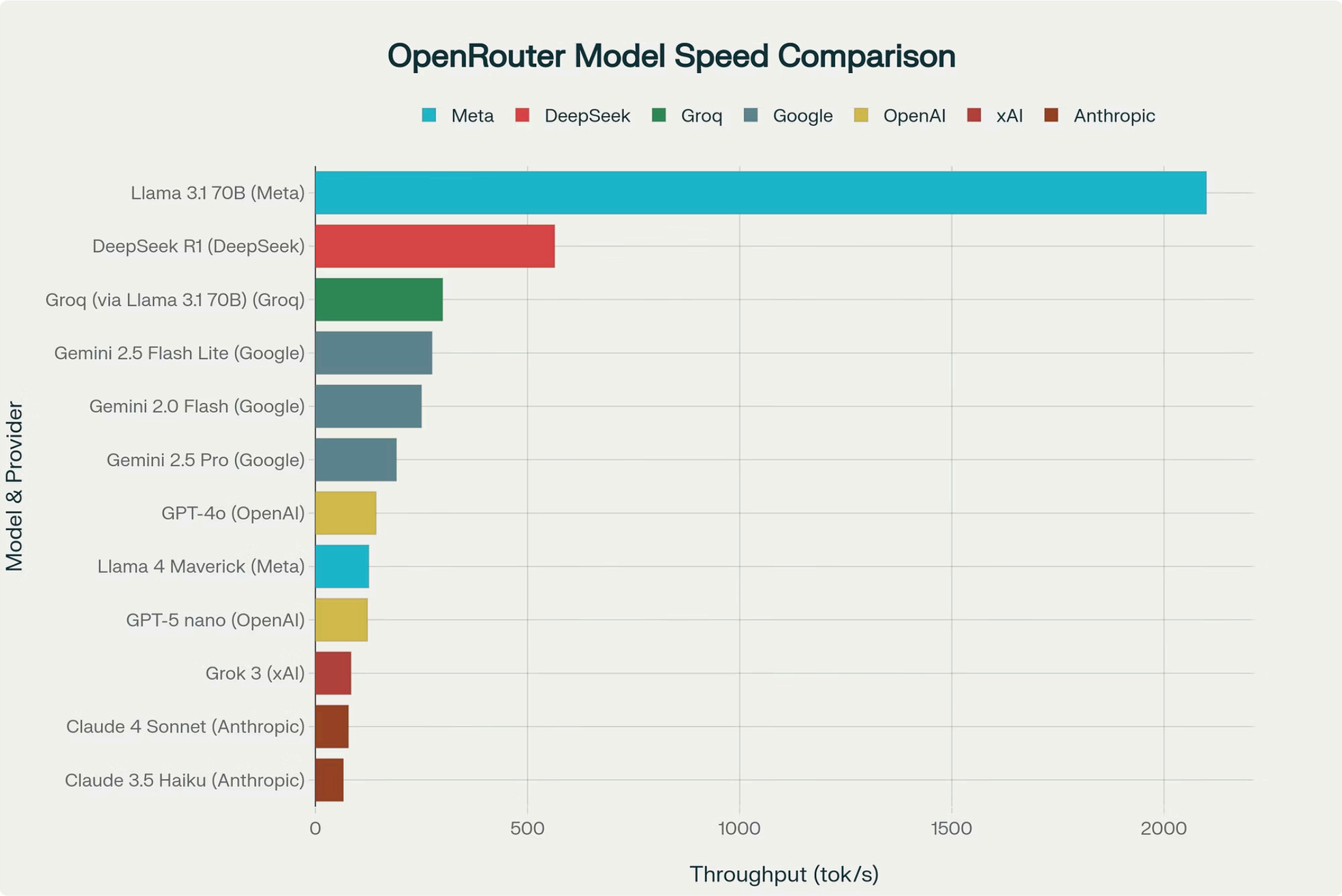
Additional motivation for chunking approaches

1. **Token-per-minute (TPM) limits**: API rate limits often restrict how much text you can send per minute. Chunking can help you plan how many tokens you plan to send in a time interval.
    a. For example, OpenAI's 30,000 tokens-per-minute limit for tier 1 customers means you'll never utilize more than ~25% of a 128K token context window, even if you only make one request per minute.

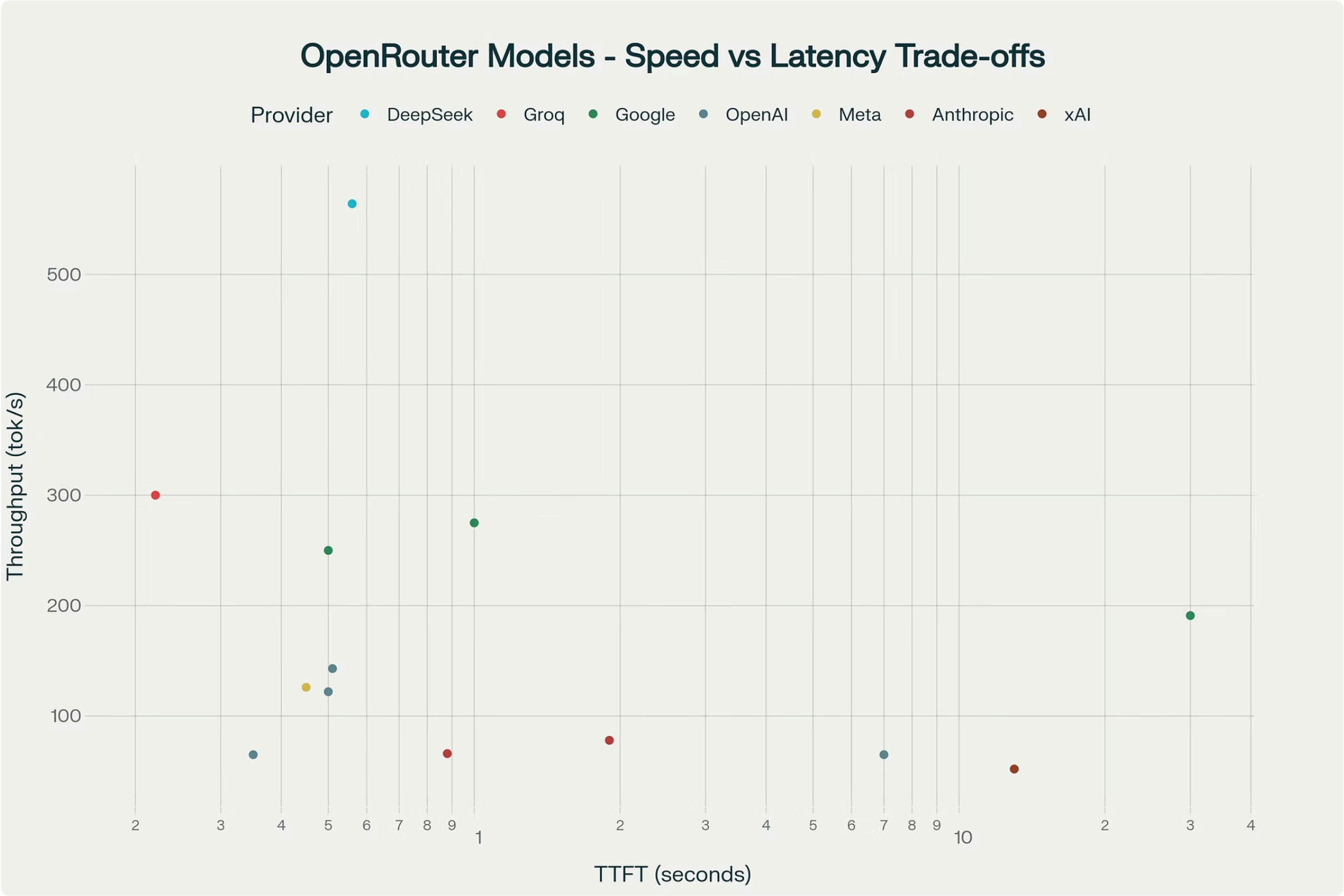
2. **"Lost in the Middle" Effect:** This widely cited paper from researchers at UC Berkeley and Google found that LLMs often suffer from a significant "position bias" with long inputs. They are best at retrieving information located at the beginning or the end of a long input context, but their performance drops substantially for information placed in the middle. This "lost in the middle" problem indicates an inability to effectively utilize the entire context.
        i. One shot prompting with lots of input and output tokens lead to poor results.

3. **"The illusion of thinking"**: Paper from researchers at Apple refers to a rapid decline in a large reasoning model's accuracy when the required number of sequential reasoning steps exceeds its capability. This collapse occurs suddenly; for tasks like the BFish puzzle, accuracy might drop from nearly 100% at four steps to near 0% at five steps, suggesting models rely on pattern matching rather than generalizable reasoning.

Know your LLM model's output tokens/sec



Throughput and Latency of various models



Jump Through Stages with Microservices

Moving from monolith to microservices architecture unlocks massive performance gains through parallelism and asynchronous processing.

Python Asyncio

Handle LLM API calls asynchronously to avoid blocking

Parallel Execution

File operations and network calls run concurrently, eliminating I/O wait time

Event Queue Architecture

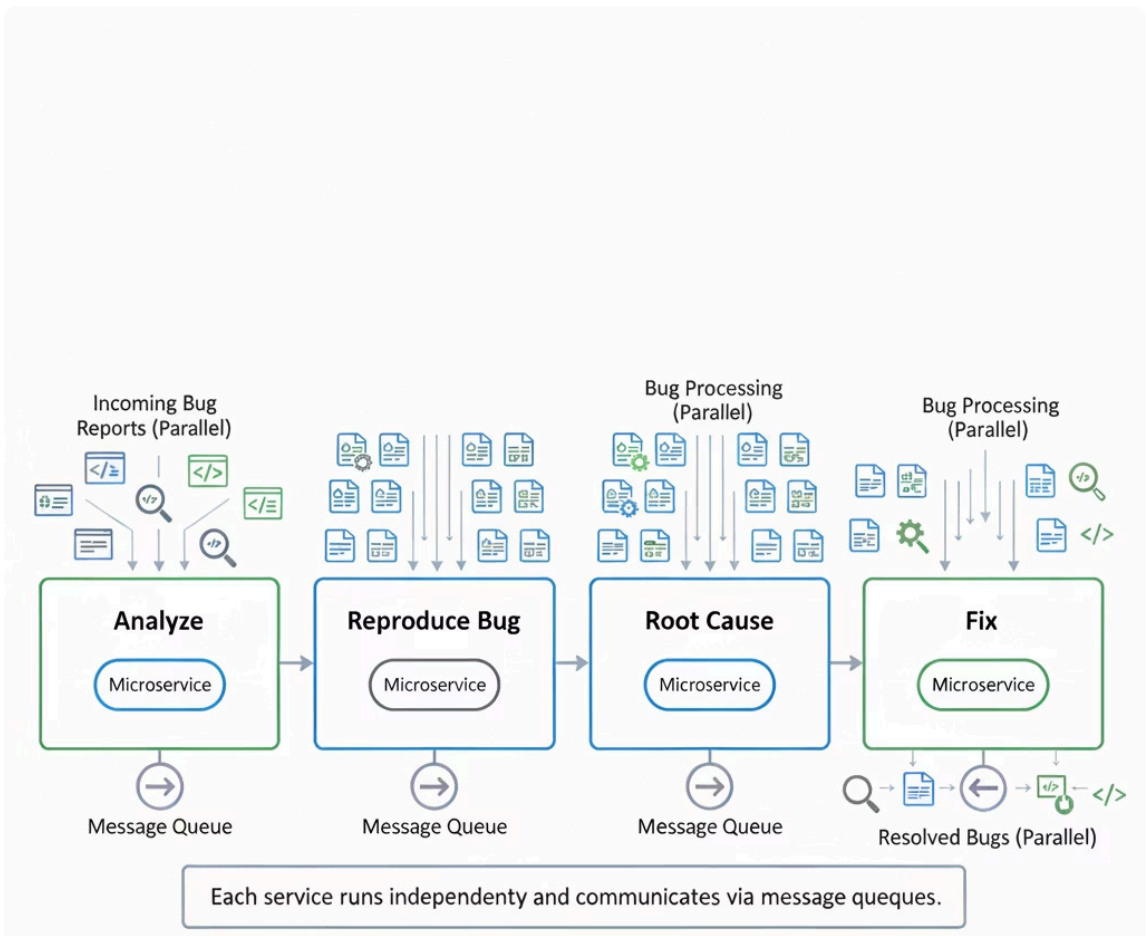
Workers communicate via RabbitMQ, processing jobs independently through pipeline stages

Benefits

- 6× faster bug fixes
- Many jobs in-flight simultaneously
- Decoupled producers and consumers

Drawbacks

- Increased complexity overhead
- More monitoring required
- Django's async ORM limitations



Modulo's bug processing pipeline can accept bug reports from various sources, and process them efficiently in a pipelined manner.

Analysis	Parse and compress bug report, comments, log files, screenshots and other artifacts
Reproduce Bug	After analysis and given the description, follow the steps to reproduce and write a test case.
Root Cause	From analysis, root cause the issue.
Fix	Generate a fix, after having root caused the issue.
Validate	If we have a bug reproduction, validate the fix to ensure the given changes fix the issue.

Demo of impact of pipelining

Cache Everything

Strategic caching dramatically improves performance and reduces costs. Choose the right approach for your use case.

1

Semantic Caching

Cache similar queries to avoid redundant LLM calls

2

MCP Server API Caching

Use Redis for distributed caching across services

3

Cache Strategy

Know when to cache and when not to—balance freshness with performance

4

Storage Choice

Choose between Redis and in-memory solutions based on scale

5

Invalidation

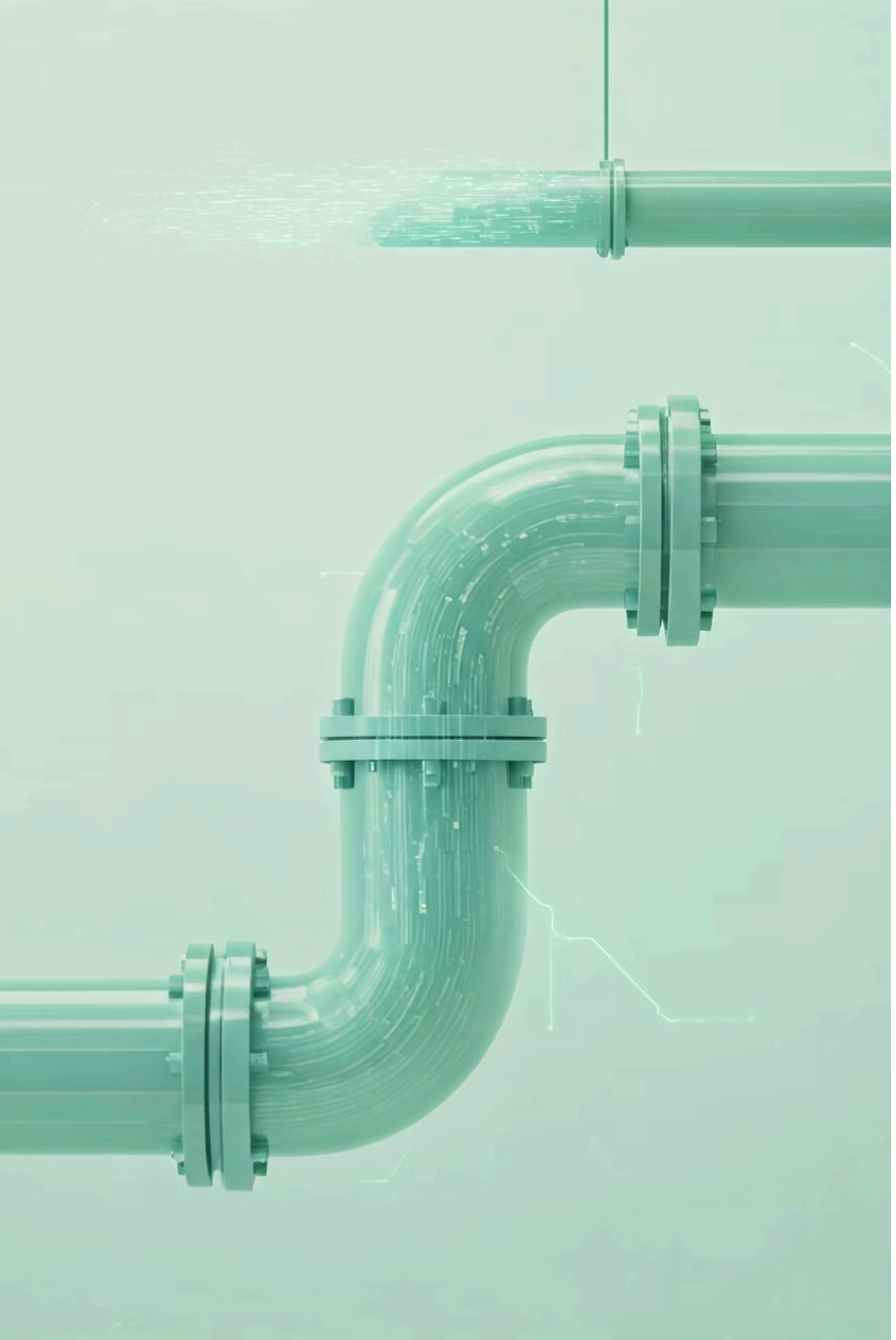
Implement effective cache invalidation to avoid stale data

Warning: Beware the pitfalls of premature optimization. Cache strategically, not everywhere.

Demo of caching based performance optimizations

Profiling a Django application

cProfiler Demo



Conclusion

With pipelining, and optimizing our LLM usage, we transformed Modulo from “nice-to-have” into a near-real-time developer assistant. The **mean time to fix** dropped from 7 minutes to about **1 minute**, greatly improving user experience.

None of these optimizations is magical on its own, but together they multiply. If you’re building a similar system, start by profiling to find your own bottleneck, and consider concurrency (threads or async) early. Break up large tasks (especially LLM calls) into parallel jobs. And remember that a flexible architecture (like microservices with a message broker) can make it easier to reason about and scale each part of your pipeline.

Q&A