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

whois

Software Engineer and Team Lead with over 13 years of experience building performant and scalable distributed systems at Google, VMware, Arista Networks and SAP

M.S. in CS from Dartmouth College

BIT Mesra alumni

Founder and CEO of [Modulo]

What is Modulo?

The design and development of Modulo was inspired by a number of problems that are as yet unsolved!

"Analyzing Bug Reports is time consuming"

- Parse through logs and artifacts
- Read through 30+ comments
- Only to find the bug is not actionable due to log rollover, or is not reproducible.

"LLMs can solve all the problems"

- But its hard to validate LLM generated bug fixes.
- An average bug report with over 1 billion tokens is too much for frontier models.

"Monitoring tools produce too many spurious alerts"

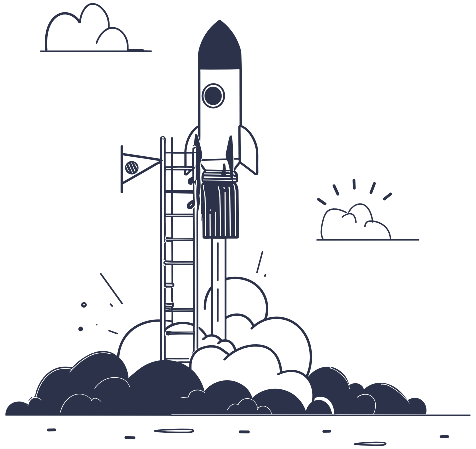
- Should I be worried about this alert?
- Did a recent change cause this issue?

Challenges with fixing bugs using frontier AI

- Over reliance on LLMs is a problem
  - LLM calls are slow! And every tool that uses them becomes slow.
  - A lot of existing solutions such as mutation testing are ignored and under valued.

- Existing tools focus on converging on ONE correct fix for a problem (high precision).

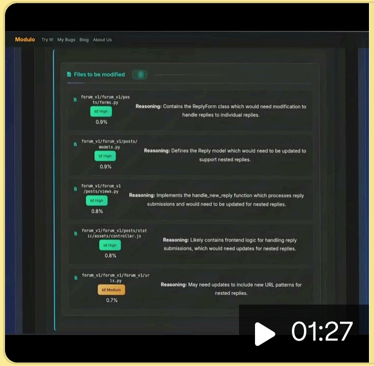
- If that solution doesn't work developers have to effectively guide the tool to try a specific approach until something works.
  - After a few iterations, context summarization etc, do developers even know what context the LLM is working with?



How is Modulo different?

Modulo MVP Demo

Modulo Web App [Link]

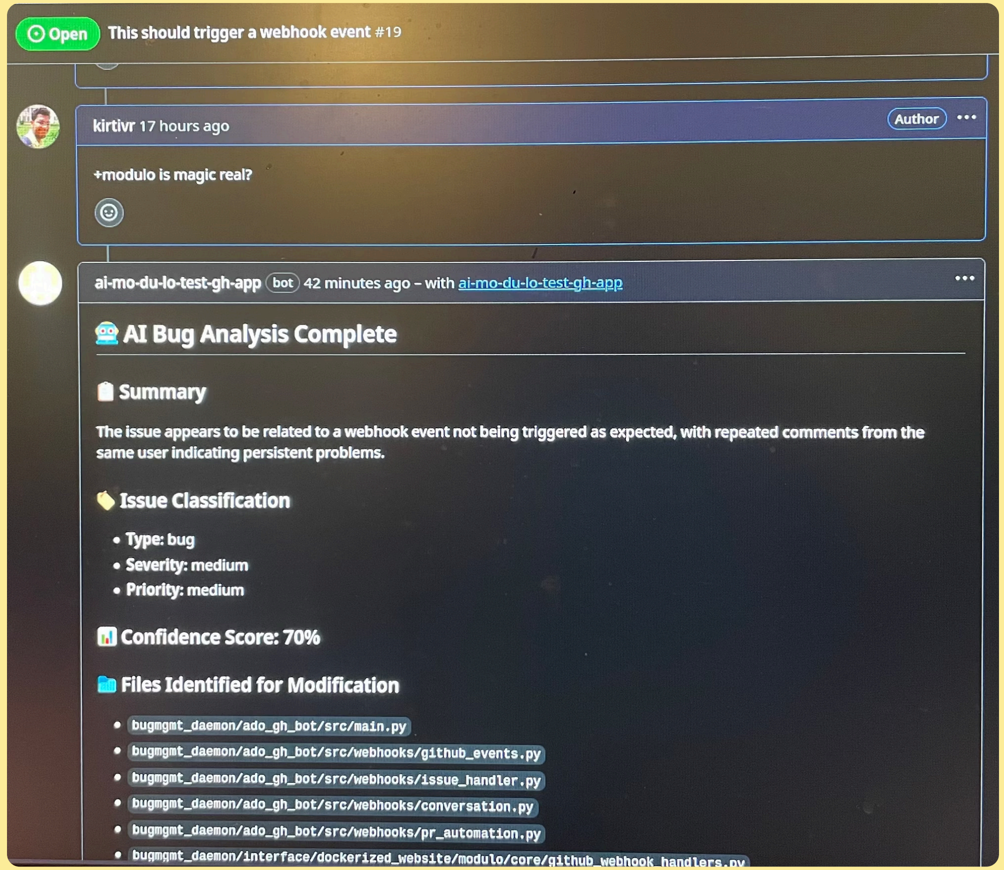


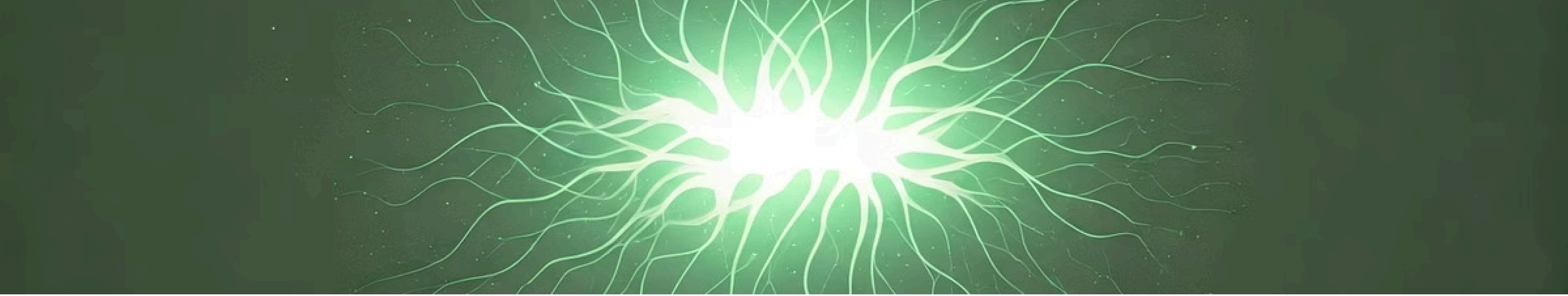
YouTube

Fix your Github Issues with Ado

Ado is a multi agent bug fixer and issue analysis engine for Github repositorie...

Modulo Github App [Preview]





Building LLM applications for speed

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

1. Higher token throughput and low time to first token
2. Optimal system architectures for high QPS
3. ... all without sacrificing response accuracy.

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

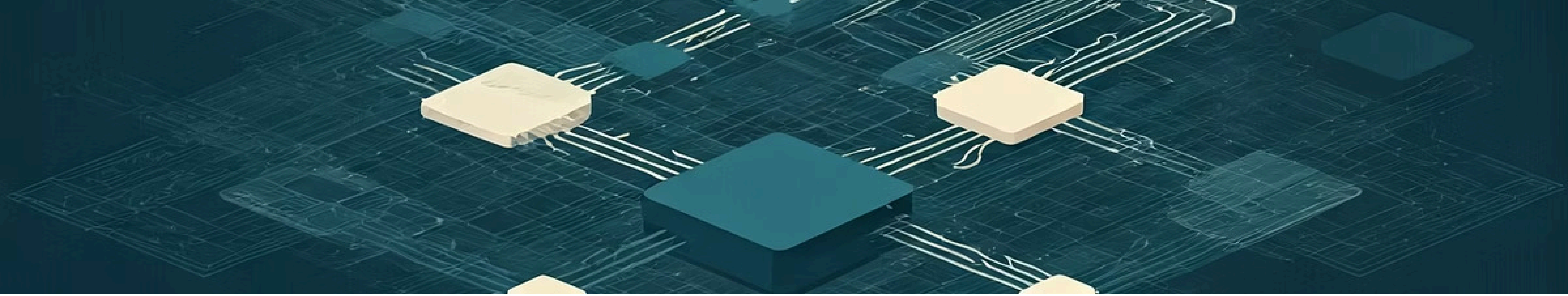
Performance enables growth without bottlenecks

Not Covered in this Talk

Strategies for improving accuracy and correctness

Multi-agent co-operation and orchestration

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



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.

Reasoning about user performance expectations

Start with setting realistic expectations before building any feature. For a given function, how soon will an impatient user expect a response?

Chat Mode

Quick responses for simple queries (**Time to First Token is key**)

Deep Research Mode

Extended processing for complex tasks (**Token Throughput is key**)

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.

Real-World Example - Find relevant files for a bug

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:

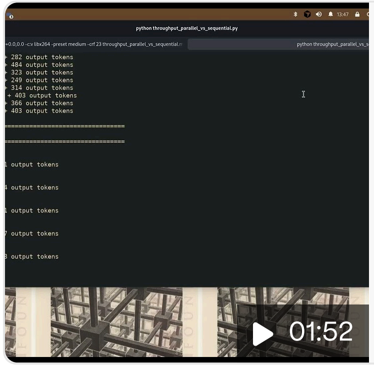
- Each file is decomposed into one or more chunks.
- LLM call for each chunk is done in a separate thread.

Using 16 parallel LLM calls on a standard VM with 16 vCPUs gives us:

$16 \times 50 = 800$ output tokens/sec which means the time is reduced to **40 seconds (from 10 minutes)!!**

This pattern works particularly well for "find a needle in a haystack" type of problems - such as root causing and generating fixes for bugs.

Demo – maximizing throughput with chunking



YouTube

Pydelhi Talk – Maximize through...

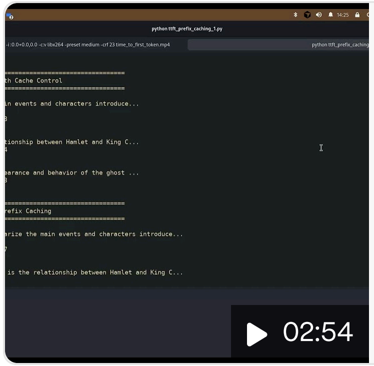
Execution time Parallel calls – 11.324 sec Sequential calls – 90.624s Token...

- Execution time
  - Parallel calls - 11.324 sec
  - Sequential calls - 90.624s

- Token throughput
  - Parallel calls - 6035 tokens/s
  - Sequential calls - 751 tokens/s

Code for this demo is [here]

Demo – improving time to first token



YouTube

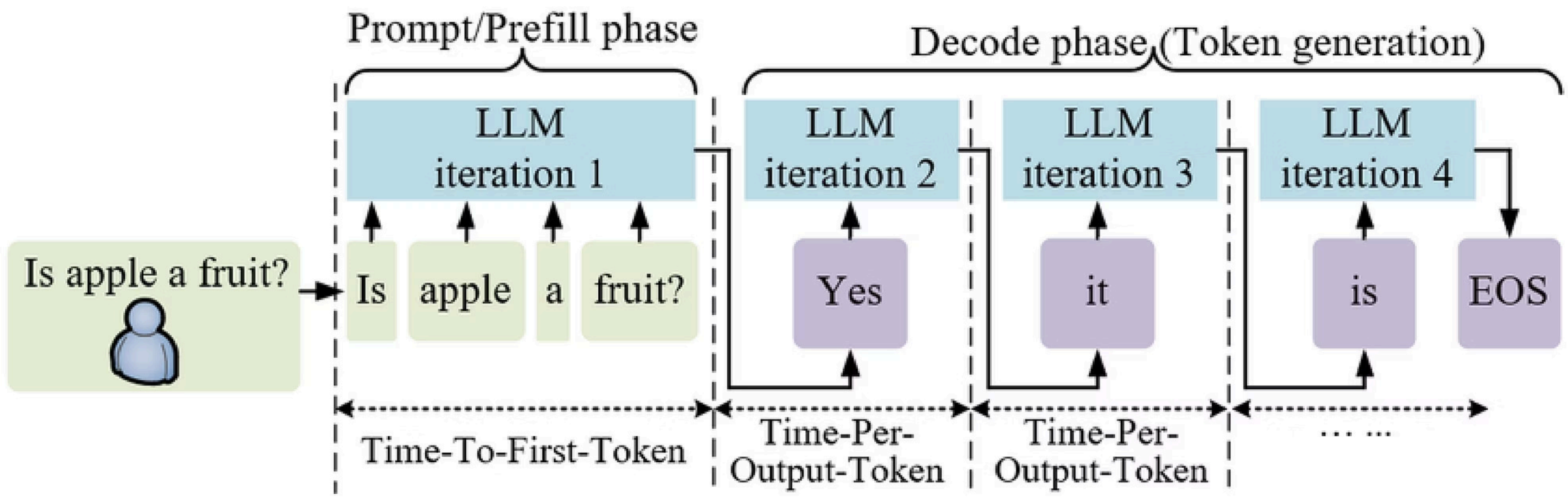
Pydelhi Talk Demo – Effect of LL...

Key to improving TTFT: Request streamed responses – tokens are...

- Key to improving TTFT:
  - Request streamed responses - tokens are returned by the LLM as soon as they are generated, in multiple responses which need to be assembled together.
  - Request prefix caching to be enabled.
- TTFT = 2.4s with streaming v/s 13 seconds without streaming.
- Note that the throughput may get worse, as it happened in our case.
  - No streaming - throughput 486 tok/s
  - streaming - throughput 244 tok/s

Code for this demo is [here]

Wait but Why?- A Case Study [TTFT]



Time to first token is directly proportional to the number of input tokens.

First output token is generated immediately after the prefill phase. The algorithm to generate subsequent output tokens is separate and is referred to as "decode phase".

Based on benchmarks it was found that for **Llama-3.1-70b each input token adds 0.05ms to the time to first token.**

For **20,000 input tokens**, one LLM call would mean - 0.05 × 100 × 200 = 1000 ms = **1s.**

Credit for image: https://dzone.com/articles/ninety-cost-reduction-prefix-caching-llms[1]

Wait but Why? [Throughput]

To measure the impact of number of output tokens, we have to see the metric **"inter-token latency" or "time per output token"**.

This metric only counts the latency between each token in the decode phase, **which means it does not depend on the number of input tokens.**

For **Llama-3.1-70b**, with 2000 output tokens, ITL was measured by NVIDIA at **30ms**.

This means generating each output token takes **30ms**.

Generating 2000 output tokens would mean - 2000 × 30 ms = **60 seconds, or 1 minute.**

Output tokens > Input tokens

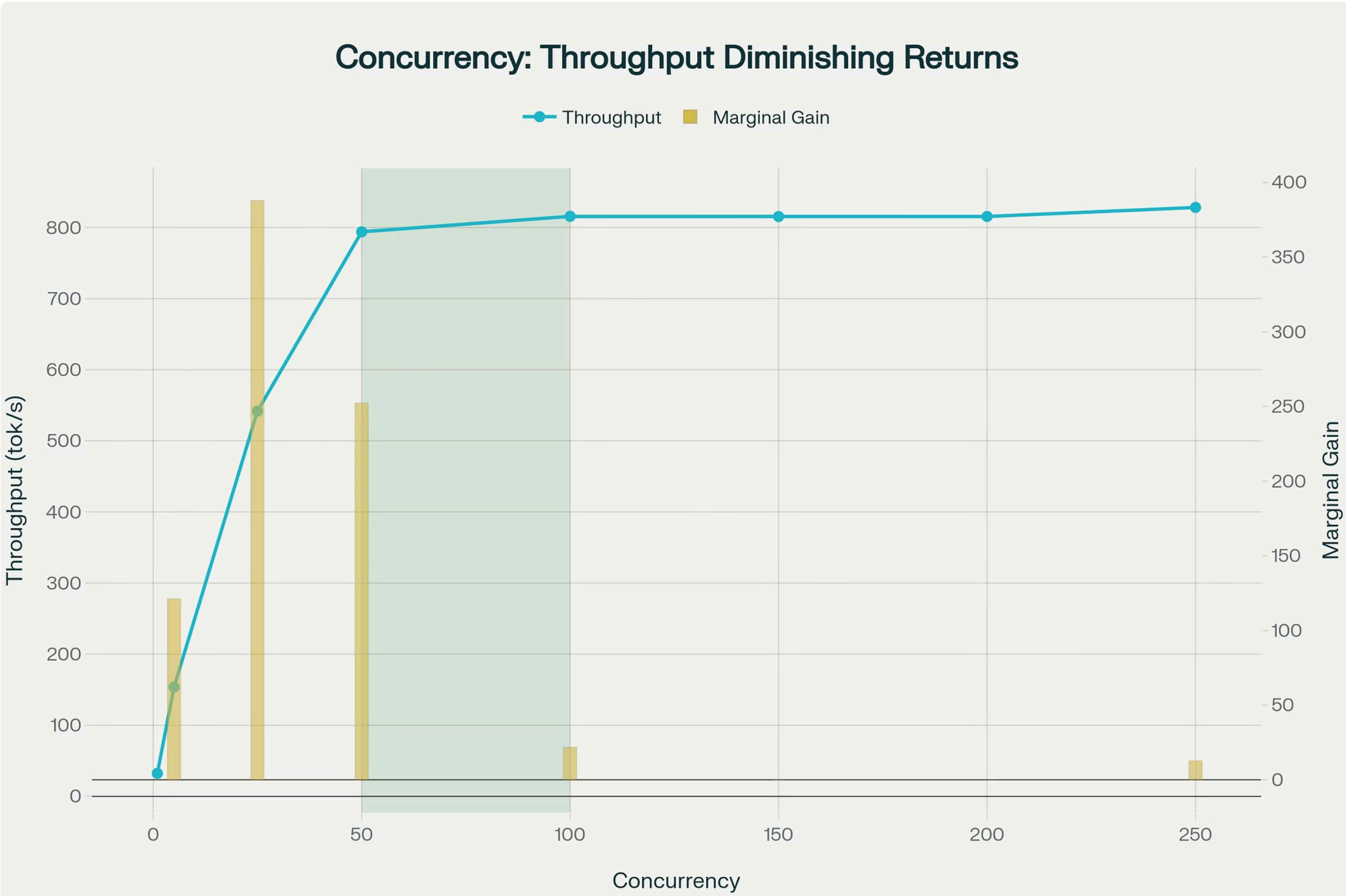
- Total time taken = ~**30ms * (Number of Output tokens) + 0.05ms * (Number of Input Tokens)**

- *Performance impact of the number of output tokens overshadows the number of input tokens.*

Made with GAMMA

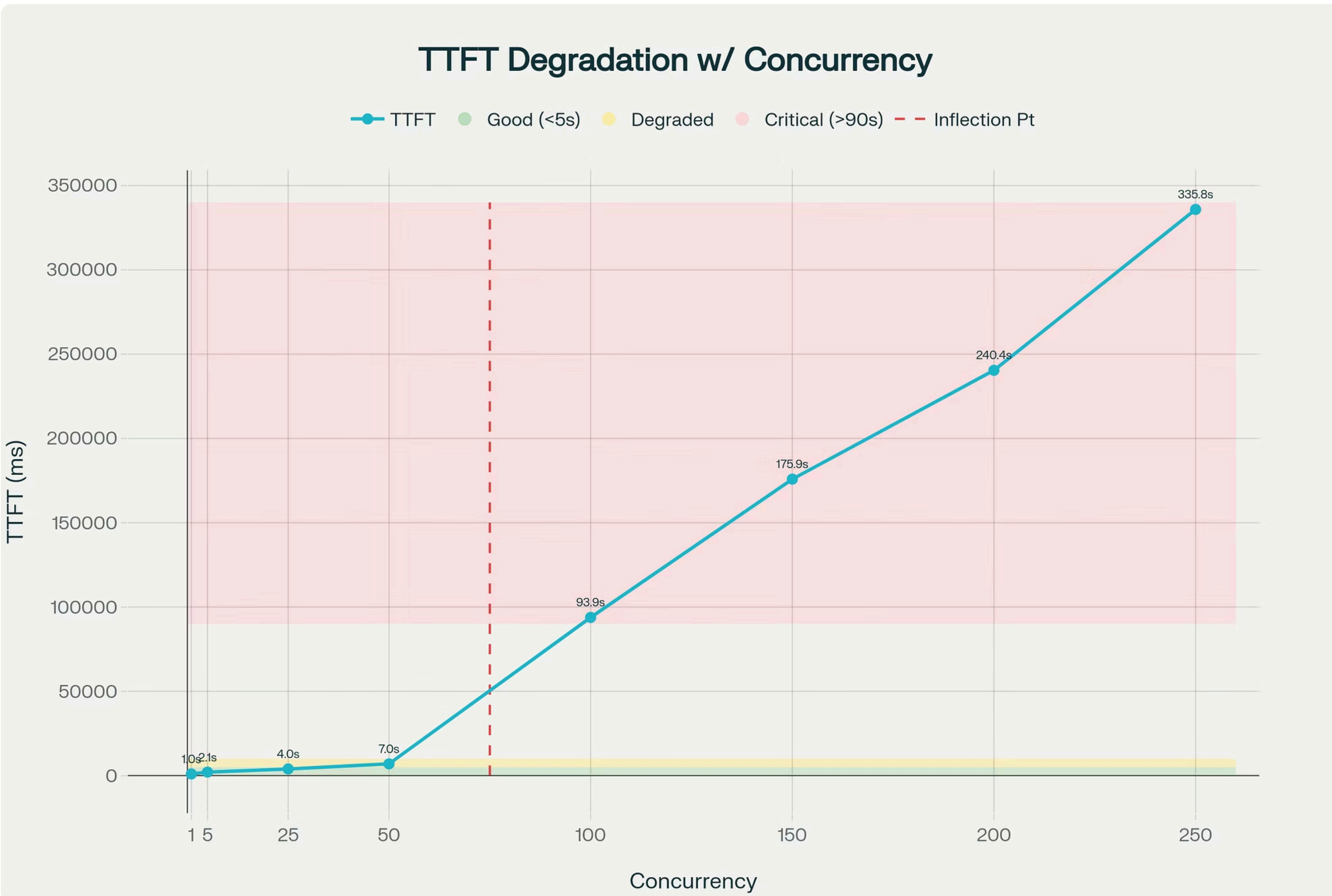
Do performance wins from concurrency last forever?

- Unfortunately, no. As this graph based on [data](#) from NVIDIA shows, as concurrency increases, throughput gains plateau.



Do performance wins from concurrency last forever?

While throughput plateaus out, **time to first token gets significantly worse.**

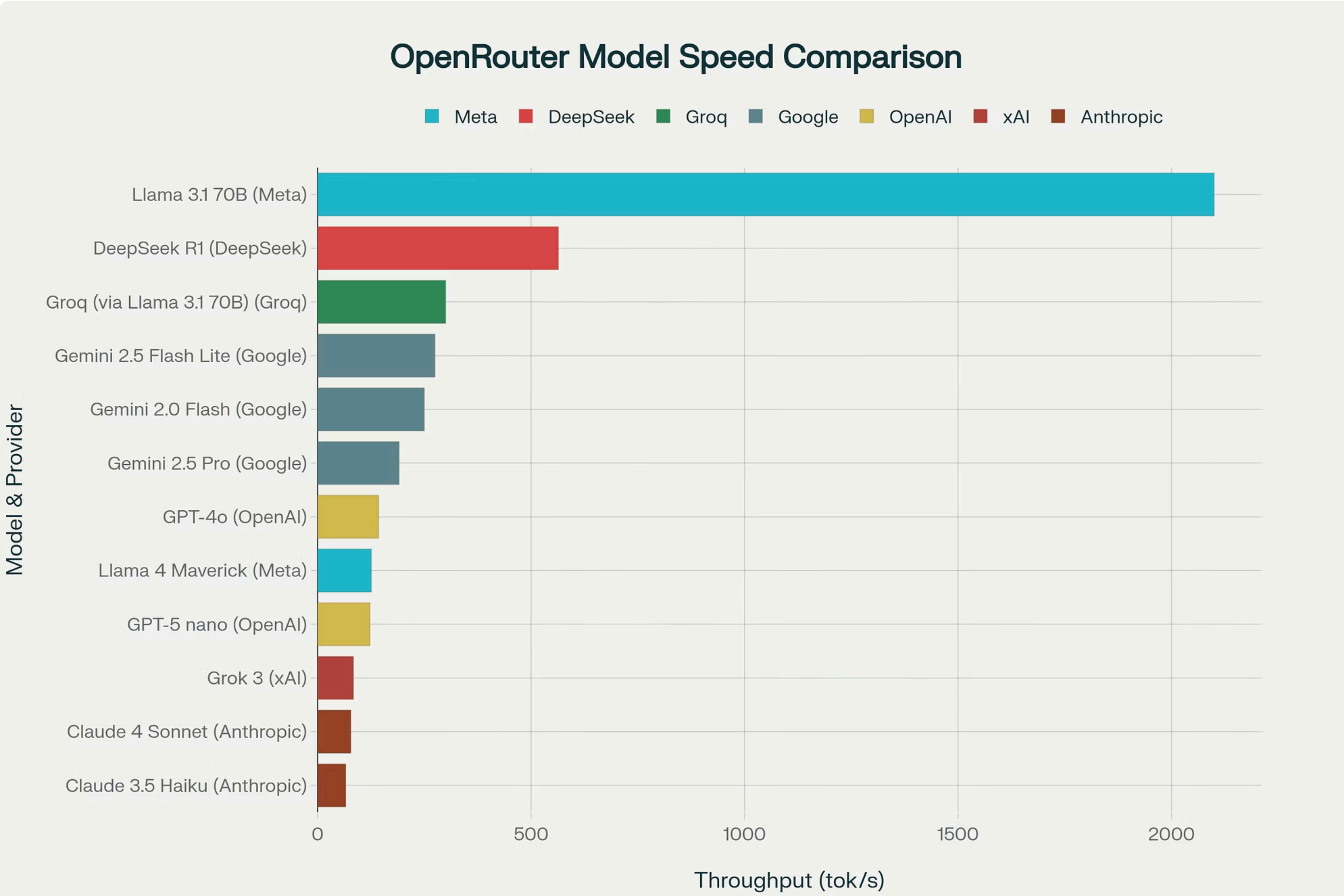


The reason for this is that the *prefill phase* of token generation is typically *compute bound.*

High concurrency means higher contention on the compute resources of the GPU.

On the other hand the *decode phase* is typically *memory bandwidth bound.* Extra compute is available, and throughput benefits from it.

Know your LLM model's output tokens/sec



Made with GAMMA

Jump Through Stages with Microservices

Transitioning to microservices boosts performance via parallelism and asynchronous processing.

Python Asyncio

Async handling for LLM API calls, preventing blocks

Parallel Execution

Concurrent file/network ops, reducing I/O wait

Event Queue Architecture

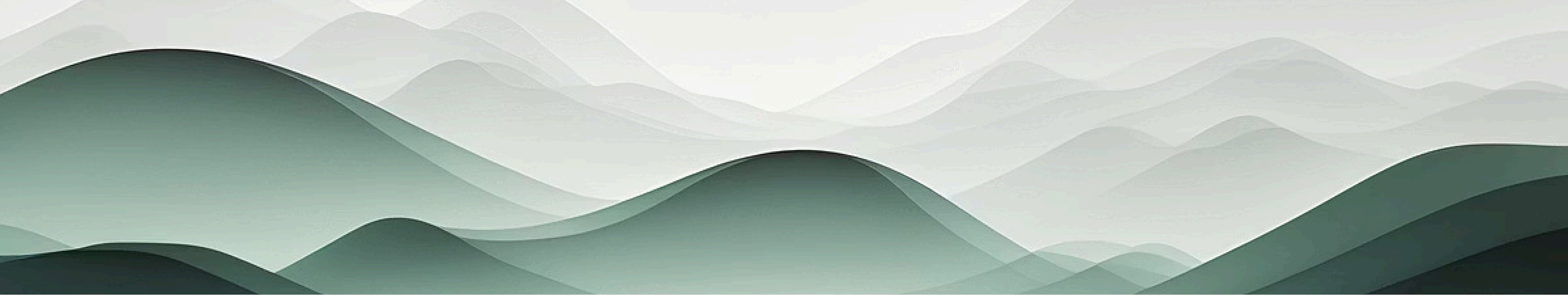
RabbitMQ enables independent worker processing

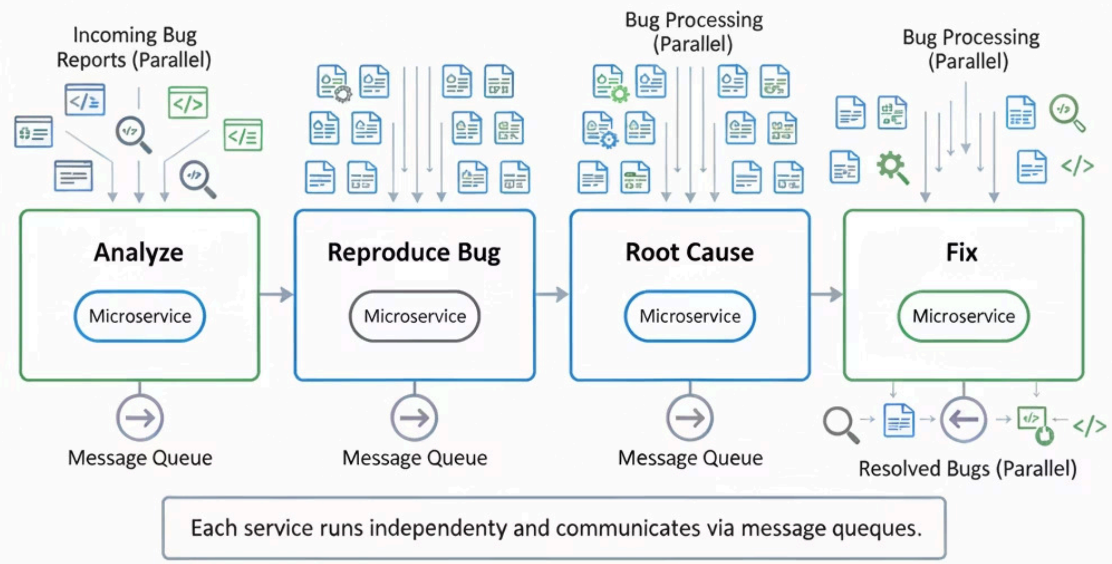
Benefits

- 6× faster bug fixes
- Concurrent job processing
- Decoupled producers and consumers

Drawbacks

- Higher complexity
- Increased monitoring
- Django async ORM limitations

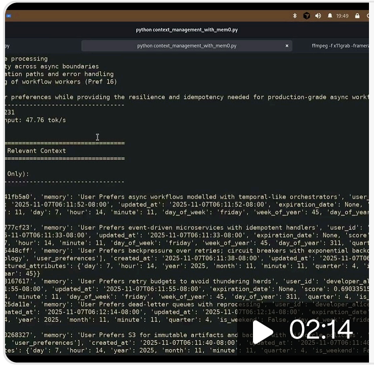




Modulo's bug processing pipeline allows stages to operate independently for incoming bugs.

Analysis	Parse and compress bug report and artifacts
Reproduce Bug	Reproduce bug and write test case
Root Cause	Identify root cause
Fix	Generate fix
Validate	Validate fix with reproduction

Context Management with Mem0



YouTube

PyDelhi Talk Demo – Context Ma...

This demo shows the token efficiency gains from using Mem0. Code for this...

- Mem0 processes a prompt to extract relevant facts from previous conversation history.
- Its internal architecture combines a vector database (for memory embeddings) and a graph database.
- Similar to RAG, Mem0 fetches only pertinent facts from a knowledge base.
- In the demo we show how token throughput improves from **60.51 tok/s to 216.06 tok/s** in the full context v/s Mem0 case.

1

Know Your Workload

Understand dependencies, parallelization, and atomicity requirements.

2

Tokens are the key

Reduce token usage for lower cost and better performance; use cost-effective models for non-critical tasks.

3

Embrace Parallelism

Leverage parallel processing for LLM applications.

4

Use Flexible Architectures

Microservices offer independent scaling, fault isolation, and pipelining.

5

Expect Failures

Implement retries universally to handle unpredictable behavior.


THANK YOU!