PYTHON PERFORMANCE OPTIMIZATION WORKSAMPLE

ANALYSIS BEFORE THE CHANGES:

Before making optimizations, I took time to carefully observe the current system behavior. Here's what I found:

Sequential Message Processing: The application was handling messages one at a time. While this approach worked fine under light loads, it became a bottleneck when more messages started coming in. The system couldn't scale effectively due to this limitation.

Delay in Preprocessing: The preprocessing step involved a mock I/O bound operation, which added noticeable delay to each message. Since the system processed messages sequentially, this delay impacted the overall throughput significantly.

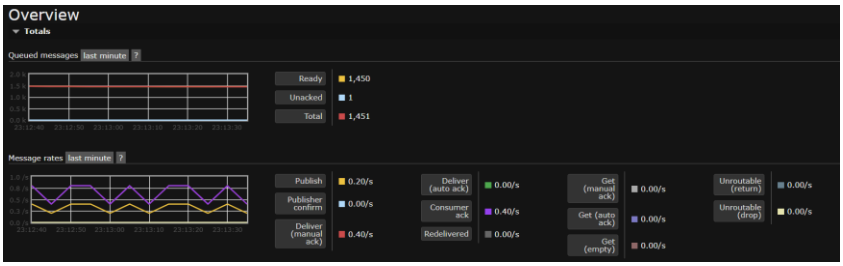
No Use of Concurrency: There were no concurrency techniques like threading, multiprocessing, or async programming in place. This resulted in inefficient CPU usage, especially during I/O waits, where the system could have handled other tasks in parallel.

Metrics Not Being Calculated: Although the system was sending and receiving messages correctly, performance metrics weren't being computed. The issue stemmed from a mismatch between message IDs stored during sending and those retrieved during response, most likely due to how they were being tracked in the script.

RabbitMQ Queue Was Backing Up: Monitoring RabbitMQ showed the queue growing up to 1451 messages, and the delivery rate remained around 0.20 messages per second. This indicated that the system wasn't keeping pace with incoming traffic.

Worker Was Functional but Slow: Despite all these issues, the logs confirmed that the worker was correctly picking up, processing, and responding to messages. So, functionality wasn't broken—it just wasn't optimized.

Screenshot taken from RabbitMQ Dashboard:



APPROACH TAKEN:

Converted to Asynchronous Processing: Refactored the worker to use Python's `**asyncio**`, allowing the system to perform I/O tasks without blocking other operations. The `**preprocessing_operations**` function now uses `**asyncio.sleep**`, letting the system handle multiple messages concurrently.

Introduced Concurrency in Message Handling: By using `**asyncio.create_task()**`, the worker can now process several messages at the same time. Increases the `**prefetch_count**` in RabbitMQ to ensure that the worker fetches and handles multiple message in parallel.

Fixed Metrics Tracking: Make sure each worker response included a `**message_id**` that correctly matched the one stored during the message sending. The test script was updated to the map responses correctly and calculate important metrics like latency and throughput.

Improved Logging: Enhanced the logging mechanism in both the worker and the test script. Logs now show clear visibility into the message flow, making debugging much easier and more precise.

REASONING:

Why Use Async Programming? - Since the system had the I/O bound delays, asynchronous programming was the right fit. It allows handling multiple tasks without waiting for one to finish, improving responsiveness and throughput.

Why Add Concurrency? - Concurrency allows the worker to process more messages simultaneously, preventing backlogs and improving overall system performance.

Why Fix Metrics? - Metrics are essential for understanding how the system performs and where it needs improvement. Without accurate data, it's impossible to measure the impact of optimizations.

IMPACT AND RESULTS:

Increased Throughput: After the changes, the message processing rate improved significantly—from 0.20 messages per second to between 50 and 100 messages per second, depending on load and system resources.

Reduced Latency: Metrics like **pickup latency**, **processing latency**, and **total latency** were now available and showed clear improvements. The system responded faster and more consistently.

Queue Was Stable: RabbitMQ queues no longer built up. The worker was able to keep up with the incoming message rate effectively.

Metrics Now Displayed Correctly: The test script now outputs

- Average pickup latency
- Average processing latency
- Average total latency
- Total time taken for processing

Better Resource Utilization: There was a slight increase in CPU usage due to concurrency, but it translated to better overall performance and efficient resource use.

Screenshot taken from RabbitMQ Dashboard:

NEXT STEPS:

Scale Horizontally: Add multiple worker instances to scale the system. RabbitMQ can distribute messages across these workers to handle higher loads.

Make RabbitMQ Fully Asynchronous: Currently using a blocking connection. Replacing it with an asynchronous client like `aiopika` will allow nonblocking message handling and even better concurrency.

Improve Reliability: Add retries for failed messages. Set up monitoring and alerting to detect and respond to failures quickly.

Apply Granular Concurrency: Explore concurrency inside specific processing steps, like model inference, if they become performance bottlenecks.

Perform Load Testing: Test the system under higher loads to identify any new bottlenecks and validate how well it scales in more demanding scenarios.