# Evaluating Python’s Asyncio for a Proxy Herd

Elizabeth Flynn, UCLA

## Abstract

This report investigates the suitability of Python’s asyncio framework (tested with Python 3.11.2) as the core technology for building an “application server herd” to support a Wikimedia-style news service. We evaluate the ease of development and maintenance for asyncio-based applications, the performance implications for I/O-bound tasks, and challenges related to Python’s type checking, memory management, and multithreading compared to Java. A brief comparison with Node.js is also included. Our findings indicate that asyncio is particularly effective for applications dominated by asynchronous I/O and inter-server communication.

## 1. Introduction

An application server herd consists of multiple application servers that communicate directly with each other as well as with core databases and caches. A news service with frequent updates, multi-protocol access, and high user mobility demands a flexible, scalable architecture. Traditional synchronous models—such as those used by Wikimedia—can become bottlenecks under such dynamic data flows. Python’s asyncio, with its event-driven paradigm, offers a promising alternative for rapidly processing and propagating updates across a networked server cluster.

Concurrency involves managing multiple tasks by interleaving their execution; tasks do not run simultaneously but are switched effectively by an event loop. Python’s asyncio exemplifies this by scheduling asynchronous tasks on a single thread. In contrast, parallelism requires tasks to execute simultaneously on multiple cores. Efficient concurrency models are essential for high-performance applications, including web servers, real-time data systems, and distributed services. This report examines two approaches—Python’s asyncio and Java multithreading—addressing three key concerns:

- **Ease of Development and Maintainability:** Can asyncio simplify the construction of complex, intercommunicating servers?
- **Language Limitations:** How do Python’s type checking, memory management, and multithreading compare with Java’s mature ecosystem?
- **Comparative Approaches:** How does asyncio stack up against Node.js in design philosophy and performance?

Our research is based on documentation review, code prototyping, and comparative analysis.

## 2. Model Overview

Python’s asyncio is designed for single-threaded execution. It employs an event loop to manage asynchronous tasks, making it ideal for I/O-bound operations. Although it operates in a single thread, asyncio provides mechanisms such as run_in_executor to offload blocking or CPU-bound tasks to a separate thread pool when necessary.

Java multithreading is integrated into the language, offering preemptive multitasking where multiple threads run concurrently under the control of the Java Virtual Machine (JVM). This model is effective for both CPU-bound and I/O-bound tasks, though it requires careful synchronization to prevent race conditions.

## 3. Key Differences and comparison

In asyncio tasks yield control to implement the “cooperative multi-tasking”. Mechanisms such as await are used to allow the event loop to switch to another task.

In Java preemptive threading is where the operating system’s scheduler is able to interrupt and switch between threads at any time, which can lead to errors in the long run.

Asyncio is highly suited for I/O tasks because it can manage tasks that spend time waiting via the event loop. However, it is not as suited for CPU-bound tasks.
Java's threading can handle I/O tasks but less so when compared to asyncio because Java has overhead from managing multiple threads. Java is best for CPU works because it can leverage multiple cores through preemptive threading.

Asyncio is easy to use and implement. Because it is single-threaded there is no need for the managing of thread creation, locking, or shared memory issues. This reduces the possibility of certain errors and lessens the need for debugging, makes implementation easier, and makes maintenance easier. However, it can occasionally be harder to implement when trying to integrate with synchronous code. Often, errors are easier to catch in asynchronous tasks. Code can be wrapped with try/except blocks to catch errors while in multi-threading errors that occur in separate threads don’t automatically propagate back to the main thread, and each thread needs to handle its own exceptions, making error handling more decentralized and harder to track in java. In Java, it is easy to start but can grow in complexity due to synchronization, debugging, and maintenance

Asyncio is highly efficient at scaling in regards to I/O operations but not so efficient for CPU tasks. It reduces overhead and resource use because it avoids the overhead of creating and context-switching between many threads.

However, it doesn't use multiple cores for parallelism (unless combined with other techniques). Asyncio uses the event loop to manage many concurrent tasks using coroutines, but bottlenecks can occur if a coroutine performance blocking operations.
Because Java can distribute tasks across multiple CPU cores it is good for CPU-intensive tasks. Java threads also have a higher resource cost compared to a coroutine. When handling numerous threads, managing shared resources becomes challenging. Developers must implement proper synchronization to prevent race conditions and contention, which adds complexity and can affect scalability if not designed carefully.

## 4. Pros and Cons of the Models

Python's asyncio offers a very simple readable structure with async/await syntax. Running in a single thread minimizes risks like race conditions since tasks share the same thread context. Since it avoids spawning multiple threads, it reduces context-switching overhead and memory consumption. Additionally, there is less management due to not having to manage multiple threads. Asyncio scales efficiently because it does not incur costs due to thread creation and management.

In Java, the complexity of managing thread synchronization, race conditions, and debugging can be challenging and consuming in larger applications. Java's multi-threading also has a robust ecosystem with extensive libraries and frameworks support multithreaded programming, along with mature debugging and profiling tools. Java can also scale due to its ability to take advance of multiple cores. However, it can offer parallelism.

### 5. Use Cases and Tradeoffs

Python's best use cases as previously mentioned are for I/O intensive tasks. Applications such as web servers with aiohttp or systems that require real-time responsiveness with concurrent I/O are ideal. Generally, coroutines are best when you spend time waiting rather than computing. Given that the example Wikimedia project involves web servers that are heavily I/O intensive it is best to use asyncio rather than Java in this case.

Java's best use case as previously mentioned are when there are computation heavy processes. It thrives in environments where it can use its ability to run in parallel using multiple cores.

Tradeoffs that asyncio has is that it's often not well suited for CPU-bound work because it is single threaded and cannot utilize multiple cores easily. Java's tradeoff is that it often is complex to manage and can have high overhead due to thread management. It requires careful resource management (often through thread pools) to avoid contention and excessive memory usage. This management makes Java have hard "ease of development" because ensuring thread safety adds complexity.

## Problems

In the course of creating the project, several challenges emerged. Initially, there were difficulties in developing the server architecture, extracting client data, and establishing a storage mechanism for processing data. Basic functionalities, such as error checking of client-supplied input, were implemented by abstracting the data collection to a central function responsible for input processing. However, integrating the retrieved client data with other functional components proved problematic.

Obstacles were encountered with the integration of the Google API. Considerable effort was devoted to utilizing the legacy version of the API, only to discover that it was no longer supported. However, even the regular version of the API was throwing errors. There were multiple ways to get an API key, and only one of the ways actually worked for this project. This issue was compounded by logistical challenges, as I had finished a substantial portion of the code ahead of schedule, and was waiting for a consensus on what the Google API implementation should look like, along with waiting for the auto-grader to work.

## Conclusions and Recommendations

Our research indicates that Python's asyncio is the best suited viable and effective framework for implementing an application server herd in a news-oriented service. Its asynchronous nature provides the scalability required for handling frequent updates and multiple protocols, and its integration with libraries such as aiohttp facilitates non-blocking HTTP communications. Compared to a Java-based implementation, asyncio offers a more straightforward approach for I/O-bound tasks while maintaining acceptable performance and maintainability.

References

- Python Software Foundation. *Official Python Documentation for asyncio*. Available at: https://docs.python.org/3/library/asyncio.html
- Node.js Foundation. *Node.js Documentation*. Available at: https://nodejs.org/en/docs/
- Ramalho, L. (2021). *Fluent Python* (2nd ed.). O'Reilly Media.
- Goetz, B., et al. (2006). *Java Concurrency in Practice*. Addison-Wesley Professional.
- Official Python Documentation for asyncio (Python 3.11.2).
- Google Places API Documentation.