International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 03 , March - 2026

Understanding and Implementing Concurrency in Python-based Systems

Adewole A.P, Adaeze Modesta Ezikeakukwu, Isaac Abayomi Fadupin, Bashir Adeyemi Amokomowo, Frank Adakole Ameh
University of Lagos, Nigeria

Abstract: - This paper is geared towards deep understanding and mastery of the implementation of concurrency in Python-based systems. One of the areas examined by the paper is the difference between threads and processes. Python's Global Interpreter Lock (GIL), which is a mutex in the CPython interpreter that allows only one thread to execute Python bytecode at a time, even on multi-core systems, has also been explored in the paper. The paper also explores the difference between CPU-bound tasks and I/O-bound tasks. The paper also examines Python concurrency models. Python concurrency models covered include threading, asyncio, multiprocessing, and structured concurrency. The implementation of these Python concurrency models is explained and illustrated by means of example codes. The paper also discusses some challenges that could be faced, as well as best practices, in the implementation of Python's concurrency models.

Keywords: concurrency, asyncio, threading, multiprocessing, Global Interpreter Lock (GIL), structured concurrency, coroutines, and synchronisation

1. INTRODUCTION

In contemporary software development, applications often need to manage multiple operations simultaneously, such as handling numerous user requests in a web server, processing large datasets, or managing I/O operations (Clarke, 2023). Concurrency allows programs to better utilize system resources and reduce idle time spent waiting for I/O operations (like network requests or database queries) to complete. Proper concurrency models enable applications to handle increasing workloads and a higher number of simultaneous connections. For user-facing applications, concurrency ensures that the application remains responsive to user input while background tasks are running (Kuyucu, 2024).

It is important to understand the distinction between concurrency (task management) and parallelism (task execution). They are often confused, but they represent distinct, complementary concepts in computer science. The key distinction is that concurrency is about managing many tasks (dealing with lots of things at once) through techniques like task switching, while parallelism is about actually executing tasks simultaneously (doing lots of things at once) on multiple processing units. Concurrency provides the *ability* to handle multiple tasks, while parallelism provides the *means* for simultaneous execution (Tommi, 2025). Concurrency is a structural, design-level approach that manages multiple tasks by having them overlap in time (interleaving) on a single processor, ideal for I/O-bound operations. Parallelism is the actual, simultaneous execution of multiple tasks, requiring multiple CPU cores to increase speed (Hadeli, 2025). On a single processor, concurrency is achieved by rapidly switching between tasks (context switching), creating the illusion that they are running simultaneously (Usa, 2025). Parallelism is the truly simultaneous execution of multiple tasks or subtasks on different CPU cores or processors, aiming to boost performance by reducing computation time. It requires multi-core hardware to run tasks side-by-side, making it ideal for heavy, CPU-bound workloads like video rendering, scientific simulations, and big data processing (Sweeton, 2025).

Python-based systems leverage the language's versatility, extensive libraries, and frameworks to build scalable applications across web development (Django, FastAPI), data science (Pandas, Scikit-learn), AI/ML (PyTorch, TensorFlow), and automation. These systems are widely adopted due to their simple syntax, extensive library ecosystem, and ability to transition seamlessly from rapid prototyping to production-ready, scalable applications. Its versatility makes it a leading choice for web platforms, scientific computing, and Internet of Things (IoT) automation (GrapesTech Solutions, 2026).

Concurrency in Python enables programs to improve performance, enhance responsiveness, and efficiently utilize system resources by managing multiple tasks through overlapping execution. In essence, concurrency aims to keep the CPU engaged and productive, whether it is distributing computations across cores or efficiently managing tasks that involve waiting for external

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 03 , March - 2026

resources. Through the strategic use of concurrency, developers can optimize their programs and achieve enhanced performance, responsiveness, and resource utilization (Pathirana, 2023).

Concurrency is necessary to maximize system efficiency by ensuring the CPU does not sit idle while waiting for slow operations. It handles I/O-bound tasks (e.g., network calls, database queries) by switching to other work during waiting periods and optimizes CPU-bound tasks (e.g., heavy computations) by running them in parallel across cores. Concurrency allows the CPU to process other tasks while one is blocked, preventing inefficient, idle server resources. Concurrency (parallelism) is needed to distribute tasks across multiple CPU cores to improve speed, rather than waiting for one core. The goal of concurrency is to prevent a single slow I/O operation from blocking the entire application, thereby enhancing responsiveness and throughput (*Pohasii, 2023*).

The rest of this paper includes a comparison of the different models of concurrency in Python, with emphasis on the impact of the Global Interpreter Lock (GIL). The paper also covers synchronization techniques and best practices for creating scalable, high-performance applications on multicore systems. The ultimate objective is to guide developers in choosing the correct tool for specific workloads and to evaluate the effectiveness of different approaches in reducing execution time and resource utilization.

2. THEORETICAL FOUNDATIONS

2.1 Threads vs. Processes

A process is an independent, heavyweight execution instance of a program with its own memory space, while a thread is a lightweight subset of a process that shares memory with other threads within the same process. Processes are isolated, whereas threads allow for efficient, concurrent communication (Xu, 2025).

Processes and threads have the following differences:

1. **Memory & Resources:** Processes are independent units of execution with dedicated memory space (code, data, heap), requiring inter-process communication (IPC) to interact. Threads, conversely, are lightweight subsets of a process that share the same address space, heap, and resources, enabling easier data sharing but risking data corruption if not synchronized (Singh, 2026).

.

2. **Creation & Overhead:** Creating a process is "heavyweight" and slow, requiring substantial OS overhead. Thread creation is "lightweight," faster, and requires less overhead. This difference in overhead makes threads a more efficient choice for tasks within a single application that require frequent communication or concurrency, while processes are better suited for independent, isolated tasks where stability and security are priorities (Andrieu, 2025).
3. **Context Switching:** Switching between threads is faster than switching between processes because threads share the same memory space, requiring less overhead. Thread context switches only require saving minimal state (registers, stack), whereas process switches require swapping memory maps, updating page tables, and flushing the Translation Lookaside Buffer (TLB), which is a high-latency operation (Premchandu, 2024).
4. **Isolation & Stability:** If one process crashes, it does not affect others. If one thread crashes, it can terminate the entire process (Mohdmohsinkhanee, 2023).

5. **Communication:** Threads communicate directly (via shared memory), whereas processes require complex Inter-Process Communication (IPC) mechanisms (Xenoss, 2026).
6. **Use Case:** Processes are best suited for independent, resource-intensive tasks because they possess their own private memory address space, ensuring high stability, security, and failure isolation. Threads are ideal for concurrent tasks within a single application, such as web browser rendering, because they are lightweight, share the same memory, and allow for efficient, fast communication (Andrieu, 2025).

2.2 The Global Interpreter Lock (GIL)

Python's Global Interpreter Lock (GIL) is a mutex in the CPython interpreter that allows only one thread to execute Python bytecode at a time, even on multi-core systems. It ensures thread safety for CPython’s memory management (reference counting) but prevents true parallel execution of CPU-bound threads. The GIL is best for I/O-bound tasks; CPU-bound tasks are better handled with multiprocessing (Humrich, 2019).

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 03 , March - 2026

The GIL was introduced to simplify memory management in CPython, making it easier to integrate with non-thread-safe C extensions. It restricts multi-threaded programs from utilizing multiple CPU cores simultaneously, often leading to reduced performance for CPU-intensive tasks. While CPU-bound programs suffer, I/O-bound programs (e.g., networking, file operations) can still benefit from threads because the GIL is released during I/O operations. To achieve true parallelism, developers should use the multiprocessing module, which creates separate memory spaces and interpreter instances for each process (Ramki, 2025).

The Python GIL limits multi-core utilization by preventing multiple threads from executing Python bytecode simultaneously within a single process, effectively making CPU-bound, multi-threaded programs single-threaded (Ranjanibabu, 2024). This design choice was made to simplify memory management and ensure thread safety. GIL restricts a Python process to using only one CPU core at a time for CPU-intensive tasks, regardless of how many cores are available on the machine (Drosopoulou, 2026).

Significant efforts have led to the GIL becoming optional through a special "free-threaded" build, allowing for better multi-core utilization. The default build of Python still includes the GIL, but users can opt into the no-GIL version. A "free threaded build of Python" is an official, optional build of the CPython interpreter where the GIL is disabled. This allows Python threads to run in true parallel on multi-core machines, which was previously not possible for CPU-bound tasks (Gomez, 2025; Aftab, 2025).

2.3 I/O-Bound vs. CPU-Bound Tasks

CPU-bound tasks are limited by processor speed, keeping CPU usage near 100% with intense calculations, while I/O-bound tasks are limited by external systems (disk, network, database) and spend most time waiting, leaving the CPU largely idle. CPU-bound tasks can be optimized via multi-threading/parallelism; I/O-bound tasks can be optimized via asynchronous, non-blocking I/O (Riyahi, 2026).

For CPU-Bound tasks, the bottleneck is the CPU. Examples include video processing, machine learning training, and complex mathematical simulations. For I/O-Bound tasks, the bottleneck is input/output operations. Examples include database queries, file reading/writing, and API calls. CPU-bound tasks need faster processors or better algorithms; I/O-bound tasks need faster storage or concurrent programming to handle waiting periods (Zhang, 2024). Table 2.1 shows the key differences between CPU-Bound tasks and I/O-Bound tasks.

Table 2.1: Key differences between CPU-Bound tasks and I/O-Bound tasks

Feature	CPU-Bound	I/O-Bound
Bottleneck	CPU Speed	Disk/Network/Database
CPU Usage	Very High (near 100%)	Low (Waiting)
Primary Need	Computational Power	High Throughput
Optimization	Parallelism, Multi-threading	Asynchronous I/O, Caching

If the application is slow but CPU usage is low, it is likely I/O-bound. If the application is slow and the CPU is maxed out, it is CPU-bound.

3. IMPLEMENTATION APPROACHES IN PYTHON

3.1 Threading (Multi-threading)

Python's threading library enables pre-emptive multitasking, where the OS scheduler pauses threads to run others, making it suitable for I/O-bound tasks. It excels in concurrently handling network requests or UI responsiveness but fails for CPU-bound parallelism due to the Global Interpreter Lock (GIL), which restricts execution to one core (Adekogbe, 2026).

In pre-emptive multitasking in Python, the Operating System (OS) scheduler decides when to pause a running thread and resume another, regardless of what the application is doing. Python threads switch execution based on the OS, though the GIL signals them

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 03 , March - 2026

to release control occasionally to allow others to run. Unlike asyncio, which requires tasks to yield control (await), pre-emptive multitasking forces switches arbitrarily (Adekogbe, 2026).

The Python threading module is part of the standard library and is used to create threading.Thread objects that operate within a single process. Threads share the same memory space and resources, making shared data management crucial to prevent race conditions and ensure thread safety. Unlike the multiprocessing module, threads in the threading module exist within the same process and thus share the same memory. When multiple threads access and modify shared data simultaneously without proper synchronization, it can lead to inconsistent and unpredictable results. In CPython (the default Python implementation), the GIL ensures that only one thread executes Python bytecode at a time. This limits true parallel execution for CPU-bound tasks but makes threading ideal for I/O-bound operations where the GIL is released during wait times (Python Documentation, 2001).

To manage shared data safely, synchronization primitives are essential. The most basic and common mechanism is a threading.Lock, which is a mutual exclusion (mutex) lock (Jain, 2025). A lock can be in one of two states: "locked" or "unlocked". A thread calls the acquire() method to lock the resource before accessing shared data. If the lock is already held by another thread, the calling thread blocks (waits) until it is released. Once finished with the shared data, the thread calls the release() method to free the lock for other threads to use. A recommended practice is to use locks with the with statement, which automatically acquires the lock upon entering the block and releases it upon exiting, even if errors occur.

To create and start a thread in Python, use the threading module, defining a target function that the thread will execute, and then calling the start() method on the thread object(Mimo, 2026).

Steps to Create and Start a Thread:

1. Import the threading module.
2. Define a function that contains the code the thread should run.
3. Create a threading.Thread object, specifying the target function and any arguments it needs.
4. Call the .start() method on the Thread object to begin its execution.
5. *(Optional)* Use the .join() method to wait for the thread to complete before the main program exits.

Example Code:

```
import threading
import time

def my_task(message, duration):
    """The function the thread will execute."""
    print(f"Thread starting: {message}")
    time.sleep(duration) # Simulate a time-consuming task
    print(f"Thread finishing: {message}")

# 1. Create a Thread object
# The 'target' is the function to run, and 'args' is a tuple of arguments
thread = threading.Thread(target=my_task, args=("Task A", 2))

# 2. Start the thread
thread.start()

print("Main thread continues to run concurrently...")

# 3. Wait for the thread to finish before exiting the main program
thread.join()
```

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 03 , March - 2026

```
print("Main thread: All done.")
```

The following are the limitations of Python Threading (zen8labs, 2025):

1. Global Interpreter Lock (GIL): Only one thread can execute Python bytecode at a time, preventing true CPU parallelism on multi-core systems.
2. Performance Bottlenecks: CPU-bound tasks (calculations, data processing) run slower or no faster than a single thread.
3. Complexity: Race conditions and deadlocks can occur, necessitating locks.
4. Overhead: Creating and switching many threads has a high cost.

The following are the use cases of Python Threading(Paes, 2024):

1. I/O-Bound Tasks: Web scraping, API requests, database queries, and GUI applications where waiting for data is common.
2. Long-Running Tasks: Keeping a user interface active while a background thread downloads data.
3. Simulated Parallelism: When I/O wait times can be hidden by doing other work concurrently.

3.2 Async IO (asyncio)

Python's asyncio library provides a single-threaded concurrency model based on cooperative multitasking, enabled by the async/await syntax and managed by an event loop. This approach is ideal for I/O-bound operations as it allows the program to perform other useful work while waiting for I/O tasks (like network requests or file access) to complete (Oluwatobi, 2025).

In cooperative multitasking, tasks voluntarily yield control of the CPU back to a central scheduler (the event loop) when they encounter an operation they must wait for, such as an I/O request. Unlike pre-emptive multitasking (used by threads, where the OS can interrupt a task at any time), the running coroutine in an asyncio program continues until it explicitly uses the await keyword or returns. This model avoids the overhead of creating and managing multiple threads or processes and eliminates the need for complex locking mechanisms to prevent race conditions, as only one coroutine runs at a time within a single thread (Zhang, 2025).

The async and await keywords are the modern Python syntax for defining and running asynchronous code. async def declares an asynchronous function, known as a coroutine. Calling an async def function does not run the code immediately; it returns a coroutine object that needs to be scheduled for execution. await pauses the execution of the current coroutine until the awaited operation (an "awaitable" object, like another coroutine, a Task, or a Future) is complete and returns control to the event loop. This allows the event loop to switch to another task that is ready to run. The await keyword can only be used inside an async def function (Jacobs, 2024).

The event loop is the core of every asyncio application, acting as the scheduler that manages the execution flow. It runs asynchronous tasks and callbacks, performs network I/O operations, and manages subprocesses. The event loop continuously monitors a queue of pending tasks. When a running task hits an await expression, it returns control to the loop. The loop then picks the next task that is ready to proceed from its internal queue and runs it. Application developers typically interact with the event loop using high-level functions like asyncio.run() (introduced in Python 3.7), which starts the loop, runs the main coroutine, and shuts the loop down when finished (Karn, 2024).

Here is a basic example of Python asyncio code that runs two tasks concurrently. This pattern is ideal for I/O-bound operations (like network requests or file I/O) where tasks can yield control while waiting, allowing other tasks to run on a single thread. This example defines a coroutine that waits for a specified delay and prints a message. The main coroutine creates two independent tasks and waits for both to complete. The total execution time is only about 2 seconds, the duration of the longest task, rather than 3 seconds if they ran sequentially (Farrag, 2025).

```
import asyncio
import time
```

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 03 , March - 2026

```
async def say_after(delay: float, what: str):
    """A coroutine that waits and then prints a message."""
    await asyncio.sleep(delay)  # Yield control to the event loop
    print(what)

async def main():
    print(f"started at {time.strftime('%X')}")

    # Create tasks to run concurrently
    task1 = asyncio.create_task(say_after(1, 'hello'))
    task2 = asyncio.create_task(say_after(2, 'world'))

    # Wait for both tasks to complete
    await task1
    await task2

    print(f"finished at {time.strftime('%X')}")

if __name__ == "__main__":
    # The entry point to run the top-level async function
    asyncio.run(main())
```

Expected Output:

```
started at <current time>
hello
world
finished at <time after ~2 seconds>
```

Explanation of the above code:

i. **async def**: Defines a coroutine function.
ii. **await**: Pauses the execution of the current coroutine and yields control back to the event loop, allowing other tasks to run until the awaited operation (e.g., asyncio.sleep, a network request) is complete.
iii. **asyncio.run(main())**: The modern (Python 3.7+) way to start the event loop and run the main asynchronous function.
iv. **asyncio.create_task()**: Schedules a coroutine to run on the event loop as an independent task.
v. **asyncio.sleep()**: A non-blocking sleep that simulates an I/O operation, unlike time.sleep() which blocks the entire thread.

3.3 Multiprocessing

Python's multiprocessing module is used to implement true parallelism, leveraging multiple CPU cores by creating separate processes, each with its own memory space and Python interpreter. It effectively bypasses the Global Interpreter Lock (GIL) (Olivieri, 2023).

Implementation can be achieved using the following methods:

1. **Using the Process Class**

   This method is suitable for running heterogeneous (different) tasks in separate processes.
   i. Define a target function: Create a function that the process will execute.
   ii. Create a Process instance: Instantiate the multiprocessing.Process class, passing the target function and its arguments (if any).

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 03 , March - 2026

iii. Start and join: Call .start() to run the process and .join() to make the main program wait for the process to complete.

```
import multiprocessing
import os

def task_function(name):
    print(f"Hi, I'm {name} in process ID: {os.getpid()}")

if __name__ == "__main__":
    print(f"Main process ID: {os.getpid()}")
    # Create processes
    p1 = multiprocessing.Process(target=task_function, args=("Process 1",))
    p2 = multiprocessing.Process(target=task_function, args=("Process 2",))

    # Start processes
    p1.start()
    p2.start()

    # Wait for processes to finish
    p1.join()
    p2.join()
    print("Both processes have finished execution!")
```

2. **Using a Pool of Workers**

The Pool class is ideal for data parallelism, where the same function is applied to multiple pieces of data simultaneously. It automatically manages a pool of worker processes and distributes tasks efficiently.

```
import multiprocessing

def calculate_square(number):
    """Function to calculate the square of a number."""
    return number * number

if __name__ == "__main__":
    numbers_list = [1, 2, 3, 4, 5]
    # Create a process pool with default number of cores
    with multiprocessing.Pool() as pool:
        # Map the function to the list of inputs
        results = pool.map(calculate_square, numbers_list)
    print(f"Results: {results}")
```

3. **Using the concurrent.futures Library**

This provides a higher-level interface for asynchronous execution, often simpler to use than managing Pool directly.

```
import concurrent.futures

def calculate_square(number):
    return number * number
```

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 03 , March - 2026

```
if __name__ == "__main__":
    numbers_list = [1, 2, 3, 4, 5]
    with concurrent.futures.ProcessPoolExecutor() as executor:
        # Submit tasks and get futures
        futures = [executor.submit(calculate_square, i) for i in numbers_list]
        # Collect results as they complete
        for future in concurrent.futures.as_completed(futures):
            print(f"Result: {future.result()}")
```

The following are key considerations concerning Python Multiprocessing:

1. if __name__ == "__main__": guard: This block is essential for cross-platform compatibility (especially on Windows) to prevent child processes from recursively spawning new processes.
2. Inter-process communication (IPC): Since processes have separate memory, data sharing requires specific mechanisms:
   i. Queue or Pipe: For message passing and exchanging objects.
   ii. Value or Array: For sharing simple C types allocated from shared memory.
   iii. Manager: To create shared Python objects like lists and dictionaries.
3. Synchronization: Use Lock, Semaphore, etc., to prevent race conditions when multiple processes access shared resources.

3.4 Structured Concurrency Concepts

Structured concurrency is an approach to concurrent programming where the lifetime of a concurrent task is explicitly tied to a clear, syntactical scope. This ensures that all spawned tasks complete (either successfully or with an error) before the parent scope exits, which makes concurrent code easier to reason about, safer, and less prone to errors like leaked tasks or silent failures (Sheth, 2025).

These are the core concepts of structured concurrency (Sheth, 2025):

1. **Scoped Lifetimes**: Concurrent tasks (e.g., coroutines) are started within a defined scope, such as an async with block in Python. The program cannot exit this block until all tasks within it are complete.
2. **Hierarchical Structure**: Tasks form a parent-child hierarchy, much like standard function calls. The parent is responsible for managing its children, making the flow of control and potential side effects predictable.
3. **Deterministic Error Propagation**: If any task within a scope fails, the parent scope is immediately aware of the error. It can then take appropriate action, such as cancelling the remaining sibling tasks to prevent them from doing unnecessary work or ending up in an invalid state.
4. **Guaranteed Cleanup and Cancellation**: The scoped nature guarantees that resources are cleaned up properly. Cancellation requests are propagated down the hierarchy, ensuring that long-running or blocking operations can be stopped gracefully.
5. **Local Reasoning**: Developers can reason about the behavior of concurrent code locally within the scope, without having to worry about tasks running indefinitely in the background or their effects manifesting in seemingly unrelated parts of the program.

In Python, structured concurrency is primarily implemented using the asyncio library (specifically with the introduction of TaskGroup in Python 3.11) and third-party libraries like Trio and AnyIO (Plesnik, 2025).

1. asyncio.TaskGroup (Python 3.11+): This class, added to the standard library following PEP 654, provides a built-in way to use structured concurrency.

```
import asyncio

async def scrape(url):
    # do network I/O here
```

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 03 , March - 2026

```
    print(f"got {url}")

async def main(urls):
    async with asyncio.TaskGroup() as tg:
        for u in urls:
            tg.create_task(scrape(u))
    # The 'async with' block exits only after all tasks are done.

# asyncio.run(main(["https://a", "https://b", "https://c"]))
```

2. Trio (Library): A pioneering library in Python structured concurrency, which popularized the "nursery" pattern. Its open_nursery() function provides the same scoping guarantees as TaskGroup.

3. AnyIO (Compatibility Library): This library provides a universal API for structured concurrency that can run on top of either asyncio or Trio, offering flexibility across different async backends. It also uses TaskGroup for a consistent experience.

4. CHALLENGES AND BEST PRACTICES

4.1 Synchronization Issues

Synchronization issues in Python's structured concurrency arise when multiple concurrent tasks access shared mutable state. Race conditions occur when operations aren't atomic, leading to data corruption. Deadlocks happen when tasks wait indefinitely for each other, and fairness issues arise if tasks are starved (Ahmad, 2025).

These are key synchronization issues (Tencent, 2025):

1. **Race Conditions**: These occur when the outcome depends on the unpredictable order of execution. Even with structured concurrency, if two tasks read a variable, calculate a new value, and write it back, they can overwrite each other. The solution to this problem is to use locks (asyncio.Lock or threading.Lock) to create atomic critical sections, ensuring only one task modifies shared data at a time.
2. **Deadlocks**: These occur when tasks are stuck, each holding a lock that the other needs (Circular Wait). Structured concurrency reduces this by enforcing clear scope, but improperly nested locks can still cause issues. The solution to this problem is to acquire locks in a consistent order, minimize the scope of locks (keep critical sections small), and avoid holding multiple locks simultaneously whenever possible.
3. **Fairness & Starvation**: If high-priority tasks monopolize a resource or CPU, other tasks may starve, never getting a turn to execute. The solution to this problem is to use synchronization primitives designed for fairness (e.g., in asyncio, locks generally grant access in a FIFO manner to waiting coroutines).
4. **Structured Concurrency Specifics**:
    i. Using asyncio.TaskGroup provides clear boundaries for task lifecycles, ensuring all tasks in a block are complete before moving on, which reduces "orphaned" tasks causing issues.
    ii. In asyncio, switching only happens at await points (cooperative multitasking), which reduces race conditions but requires careful handling of shared state between those points.
5. **Best Practices**:
    i. Minimize Shared State: Prefer passing data through queues or returning values from tasks instead of modifying shared global variables.
    ii. Context Managers: Use async with lock: to guarantee that locks are released automatically, reducing deadlocks caused by unreleased locks.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 03 , March - 2026

4.2 Modern Python Concurrency

odern Python concurrency relies on the asyncio library for high-efficiency, I/O-bound tasks, leveraging the async/await syntax. Key best practices include using structured concurrency with asyncio.TaskGroup (Python 3.11+) and offloading CPU-intensive or blocking operations to a separate thread or process to prevent blocking the event loop (Ozer, 2026).

These are the core concepts and features of modern Python concurrency (ITLearningAi, 2026):

1. **Event Loop**: The single-threaded "heart" of asyncio that manages and switches between multiple tasks as they wait for I/O operations (e.g., network requests, database calls), allowing for high concurrency.
2. **Coroutines**: Functions defined with async def that can pause execution at await points, yielding control back to the event loop. Calling an async function creates a coroutine object; it must be awaited or scheduled as a task to run.
3. **Tasks**: Coroutines wrapped and scheduled on the event loop using asyncio.create_task() or implicitly via asyncio.TaskGroup.
4. **asyncio.run()**: The recommended entry point for an asyncio application, handling the setup and teardown of the event loop and task cleanup.

These are the best practices for modern asyncio (Majidbasharat, 2026):

1. **Avoid Blocking Operations**: Never use blocking synchronous code (like time.sleep() or the requests library) inside an async function. This will block the entire event loop. Instead, use async-compatible libraries (e.g., httpx for HTTP requests, aiosqlite for databases) or offload the blocking work.
2. **Use asyncio.TaskGroup for Structured Concurrency**: For managing groups of related tasks, use async with asyncio.TaskGroup() as tg: (Python 3.11+). This provides safer error handling and ensures all tasks are completed or properly cancelled when the context manager exits, which is more robust than asyncio.gather().
3. **Manage Concurrency Volume**: To prevent overwhelming external services or running out of memory (OOM) when handling large numbers of tasks (e.g., 100,000 URLs), use an asyncio.Semaphore to limit the number of concurrent operations or implement a Producer-Consumer pattern with asyncio.Queue.
4. **Implement Robust Error Handling and Timeouts**:
    i. Wrap awaited calls in asyncio.wait_for() to prevent tasks from hanging indefinitely due to network issues or slow responses.
    ii. Use except* with ExceptionGroup (Python 3.11+) to handle multiple exceptions from concurrent tasks in a clean way.
    iii. Implement retry logic with exponential backoff for transient errors, especially when interacting with APIs.
5. **Prefer Async Context Managers**: Use async with for resource management (e.g., HTTP sessions, database connections) to ensure proper and timely cleanup, even in the event of a cancellation or error.

For long-running tasks, especially those that are CPU-bound, asyncio is not the right tool for true parallelism because it runs on a single thread. The best approach is to offload these tasks (Rahman, 2025):

1. **Offload CPU-Bound Tasks**: Use the event loop's loop.run_in_executor() method with a ThreadPoolExecutor (for I/O-bound synchronous tasks) or ProcessPoolExecutor (for CPU-bound tasks) to run the blocking code in a separate thread or process. The main asyncio loop can then continue running other tasks without being blocked.
2. **Handle Cancellation Gracefully**: Design your long-running tasks to be aware of cancellation. Wrap critical cleanup code in try...finally blocks to ensure resources are released when a asyncio.CancelledError is raised.

4.3 Hybrid Approaches

Combining asyncio and multiprocessing is a powerful hybrid approach used to achieve maximum performance for applications involving both I/O-bound and CPU-bound tasks. The core idea is to use asyncio for efficient, concurrent handling of I/O operations (like network requests) and multiprocessing to offload CPU-intensive work to separate CPU cores in parallel (Mahmud, 2026).

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 03 , March - 2026

A common implementation pattern involves a main process that manages the asyncio event loop and a pool of worker processes to handle the heavy computations (Wagh, 2025):

1. **asyncio for I/O-bound tasks**: The main process, using asyncio, fetches data (e.g., hundreds of files over a network). While it waits for one network request, the event loop switches to another, maximizing concurrency.
2. **multiprocessing for CPU-bound tasks**: Once the data is fetched, the CPU-intensive processing is sent to a concurrent.futures.ProcessPoolExecutor, which runs in separate processes.

The primary method to combine these techniques effectively in Python is to use the run_in_executor function provided by the asyncio event loop (Dano, 2015).

1. **loop.run_in_executor**: This built-in function allows you to execute a regular blocking or CPU-bound function in a separate executor (either a ThreadPoolExecutor or ProcessPoolExecutor) without blocking the main asyncio event loop.
2. **Inter-process communication (IPC)**: To share data between the asyncio event loop in the main process and the worker processes, you need to use multiprocessing primitives like Queue, Pipe, or Manager objects. Standard asyncio data structures are single-process only.
3. **Third-party libraries**: Libraries such as aioprocessing (though potentially unmaintained) provide asyncio-compatible versions of multiprocessing data structures, which can simplify communication between the event loop and child processes.

5. CONCLUSION

Concurrency allows programs to better manage multiple operations simultaneously, such as handling numerous user requests in a web server, processing large datasets, or managing I/O operations. This paper has looked at the difference between threads and processes. A Process means a program is in execution. A Thread is the smallest unit of execution within a process. Python's Global Interpreter Lock (GIL), which is a mutex in the CPython interpreter that allows only one thread to execute Python bytecode at a time, even on multi-core systems, has also been looked into in the paper. The paper has also looked at the difference between CPU-bound tasks and I/O-bound tasks. The paper also looked at approaches to the implementation of concurrency in Python. Python concurrency models covered include threading, asyncio, multiprocessing, and structured concurrency. The implementation of these Python concurrency models has been explained and illustrated by means of example codes. The paper has also discussed some challenges that could be faced, as well as best practices, in the implementation of Python's concurrency models.

REFERENCES

[1] Adekogbe F. (2026): Python Parallelization - Threads vs. Processes. https://bytewax.io/blog/threads-vs-processes Accessed: 15th March, 2026.
[2] Aftab (2025): Finally Optional (And Why This Changes Everything). https://medium.com/@aftab001x/pythons-liberation-the-gil-is-finally-optional-and-why-this-changes-everything-5579b43e969c Accessed: 12th March, 2026.
[3] Ahmad A. (2025): Mastering Multithreading Patterns: Race Conditions, Deadlocks, and Producer-Consumer. https://designgurus.substack.com/p/mastering-multithreading-patterns Accessed: 17th March, 2026.
[4] Andrieu E. (2025): Processes vs. Threads: How to Choose the Right Concurrency Model for Your Application. https://medium.com/@eugenio.andrieu_63440/processes-vs-threads-how-to-choose-the-right-concurrency-model-for-your-application-2bffdbade814 Accessed: 14th March, 2026.
[5] Clarke H. (2023): Mastering Concurrency: A Guide for Software Engineers. https://www.harrisonclarke.com/blog/mastering-concurrency-a-guide-for-software-engineers#. Accessed: 14th March, 2026.
[6] Dano (2015): What kind of problems (if any) would there be combining asyncio with multiprocessing? https://stackoverflow.com/questions/21159103/what-kind-of-problems-if-any-would-there-be-combining-asyncio-with-multiproces Accessed: 17th March, 2026.
[7] Drosopoulou E. (2026): The Python GIL Controversy: Why Multi-Core Parallelism Remains Broken (And Why It Might Not Matter). https://www.javacodegeeks.com/2026/01/the-python-gil-controversy-why-multi-core-parallelism-remains-broken-and-why-it-might-not-
[8] Farrag M. (2025): Python asyncio: All you need to know with Examples. https://mafarrag.medium.com/python-asyncio-all-you-need-to-know-with-examples-b18e99d35a2a Accessed: 16th March, 2026.
[9] Gomez K. (2025): Parallel with Python 3.14 Free-Threaded Version. https://medium.com/@kyeg/executing-multiple-agents-in-parallel-with-python-3-14-free-threaded-version-4d857d9c4639 Accessed: 12th March, 2026.
[10] GrapesTech Solutions (2026): Python in IoT: Why It's the Go-To Language for Smart Solutions. https://www.grapestechsolutions.com/blog/python-in-iot/. Accessed: 14th March, 2026.
[11] Hadeli M. (2025): Concurrency vs. Parallelism in .NET: A Practical Guide. https://mehdihadeli.com/blog/concurrency-parallelism Accessed: 11th March, 2026.
[12] Humrich N. (2019): Let's Talk about Python's GIL. https://medium.com/pyslackers/lets-talk-about-python-s-gil-ade59022bc83 Accessed: 12th March, 2026.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 03 , March - 2026

[13] ITLearningAi (2026): Asyncio in Python Explained: Master Asynchronous Programming (2025 Guide). https://medium.com/@itlearningai/asyncio-in-python-explained-master-asynchronous-programming-2025-guide-b0dd493ffaed Accessed: 17th March, 2026.
[14] Jacobs A. (2024): Deep Dive into Python Async Programming. https://alex-jacobs.com/posts/pythonasync/ Accessed: 16th March, 2026.
[15] Jain Y. (2025): When Threads Collide: Understanding Locks in Python. https://medium.com/algomart/when-threads-collide-understanding-locks-in-python-9e3faab0dacc Accessed: 15th March, 2026.
[16] Karn A. R. (2024): Concurrency in Python: Understanding Threading, Multiprocessing, and Asyncio. https://medium.com/@ark.iitkgp/concurrency-in-python-understanding-threading-multiprocessing-and-asyncio-03bd92ca298b Accessed: 16th March, 2026.
[17] Kuyucu A. K. (2024): Concurrency in Python: Advanced Patterns and Techniques (Part 16). https://python.plainenglish.io/concurrency-in-python-advanced-patterns-and-techniques-part-16-8610dfa02c37 Accessed: 14th March, 2026.
[18] Mahmud S. (2026): Python Concurrency Showdown: AsyncIO vs Threading vs Multiprocessing — Which Should You Choose in 2026? https://medium.com/@sizanmahmud08/python-concurrency-showdown-asyncio-vs-threading-vs-multiprocessing-which-should-you-choose-in-31205161899a#. Accessed: 17th March, 2026.
[19] Majidbasharat (2026): Asyncio Deep Dive: Building High-Performance Python Applications. https://medium.com/@majidbasharat21/asyncio-deep-dive-building-high-performance-python-applications-f77cfc44c463 Accessed: 17th March, 2026.
[20] Mohdmohsinkhanee (2023): Threads and Process. https://www.emblogic.com/forum/discussion/1050/threads-and-process/p1 Accessed: 14th March, 2026.
[21] Olivieri R. (2023): How to use a Python multiprocessing module. https://developers.redhat.com/articles/2023/07/27/how-use-python-multiprocessing-module Accessed: 16th March, 2026.
[22] Oluwatobi R. (2025): Python Concurrency Model Comparison For CPU And IO Bound Execution: Asyncio vs Threads vs Sync. https://medium.com/@romualdoluwatobi/python-concurrency-model-comparison-for-cpu-and-io-bound-execution-asyncio-vs-threads-vs-sync-35c114fc0045 Accessed: 16th March, 2026.
[23] Ozer C. B. (2026): What Is the Safe Way to Handle Concurrency in Python? https://medium.com/softtechas/what-is-the-safe-way-to-handle-concurrency-in-python-906807ba100c Accessed: 17th March, 2026.
[24] Paes E. (2024): Understanding Threading in Python Programming: When to Use It and How to Address Challenges. https://python.plainenglish.io/understanding-threading-in-python-programming-when-to-use-it-and-how-to-address-challenges-0f6863c89819 Accessed: 16th March, 2026.
[25] Pathirana V. G. (2023): Python Concurrency Demystified: Unlocking Speed and Scalability. https://medium.com/@ravinvishwa123/python-concurrency-demystified-unlocking-speed-and-scalability-287aca579f16 Accessed: 14th March, 2026.
[26] Plesnik J. (2025): Python: Guide to structured concurrency. https://applifting.io/blog/python-structured-concurrency Accessed: 17th March, 2026.
[27] Premchandu (2024): Process vs Thread. https://medium.com/@premchandu.in/process-vs-thread-12e0c8768a4a Accessed: 14th March, 2026.
[28] Python Documentation (2001): threading — Thread-based parallelism. https://docs.python.org/3/library/threading.html Accessed: 15th March, 2026.
[29] Rahman S. S. (2025): What's the Best Way to Handle Concurrency in Python: ThreadPoolExecutor or asyncio? https://towardsdev.com/whats-the-best-way-to-handle-concurrency-in-python-threadpoolexecutor-or-asyncio-85da1be58557 Accessed: 17th March, 2026.
[30] Ramki (2025): Multiprocessing in Python and its start methods (fork vs spawn). https://medium.com/@ramkit.r.1/multiprocessing-in-python-and-its-start-methods-fork-vs-spawn-81c51d50ebe2 Accessed: 12th March, 2026.
[31] Ranjanibabu (2024): Understanding Python's GIL: Overcoming Limitations for Multi-threading. https://medium.com/top-python-libraries/understanding-pythons-gil-overcoming-limitations-for-multi-threading-f16a7e382f76 Accessed: 12th March, 2026.
[32] Riyahi S. (2026): I/O-bound vs CPU-bound: Understanding Performance Bottlenecks. https://www.linkedin.com/posts/sina-riyahi_iobound-vs-cpubound-iobound-a-task-activity-7410366483924520960-LzyG Accessed: 15th March, 2026.
[33] Sheth H. (2025): Python concurrency: gevent had it right. https://harshal.sheth.io/2025/09/12/python-async.html Accessed: 17th March, 2026.
[34] Singh A. P. (2026): Processes vs Threads. https://algomaster.io/learn/concurrency-interview/processes-vs-threads Accessed: 14th March, 2026.
[35] Sweeton D. (2025): Parallelism — When Tasks Truly Run at the Same Time. https://medium.com/@sweetondonie/parallelism-when-tasks-truly-run-at-the-same-time-ca70fcaf735f Accessed: 11th March, 2026.
[36] Tencent (2025): How to solve the race condition problem in vulnerability repair? https://www.tencentcloud.com/techpedia/124103 Accessed: 17th March, 2026.
[37] Tommi V. (2025): Concurrency vs Parallelism: Understanding the Difference with Examples day 54 of system design. https://dev.to/vincenttommi/concurrency-vs-parallelism-understanding-the-difference-with-examples-day-54-of-system-design-27bh#. Accessed: 14th March, 2026.
[38] Usa W. (2025): Concurrency vs. Parallelism: What's the Difference and Why Should You Care? https://www.freecodecamp.org/news/concurrency-vs-parallelism-whats-the-difference-and-why-should-you-care/ Accessed: 11th March, 2026.
[39] Wagh P. (2025): Efficiently Process Concurrent Tasks with a Worker Pool. https://www.linkedin.com/pulse/efficiently-process-concurrent-tasks-worker-pool-pawan-wagh-shgnc#. Accessed: 17th March, 2026.
[40] Xenoss (2026): Concurrency. https://xenoss.io/ai-and-data-glossary/concurrency#. Accessed: 14th March, 2026.
[41] Xu A. (2025): Popular interview question: What is the difference between **Process** and **Thread**? https://www.linkedin.com/posts/alexxubyte_systemdesign-coding-interviewtips-activity-7383176861406633984-TrYt/ Accessed: 14th March, 2026.
[42] Zhang S. (2024): Demystifying Python Concurrency: IO-Bound vs. CPU-Bound Tasks. https://medium.com/@shane-zhang/demystifying-python-concurrency-io-bound-vs-cpu-bound-tasks-64016db696c7#. Accessed: 15th March, 2026.
[43] Zhang S. (2025): Cooperative Multitasking: The Core of Python Asyncio. https://shanechang.com/p/python-asyncio-cooperative-multitasking/ Accessed: 16th March, 2026.