ChatGPT

# Python 3.11+ Features for Building a Subprocess-Isolated Execution Service

## Async/Await and Coroutine Management in Modern Python

**Structured Concurrency (Task Groups):** Python 3.11 introduced `asyncio.TaskGroup`, an asynchronous context manager to run and supervise groups of tasks. Task groups provide **structured concurrency** – all tasks spawned within a group are awaited collectively and cancelled together on errors. This pattern (inspired by Trio's *nurseries*) makes concurrent code safer and easier to reason about. For example, using `asyncio.TaskGroup` instead of `asyncio.gather` ensures that if any task errors, all remaining tasks in the group are canceled and their exceptions collected into an `ExceptionGroup`. Python's adoption of this pattern was directly influenced by Trio's design; the ExceptionGroup PEP even credits Trio's MultiError and nurseries as motivation. **ExceptionGroup** (PEP 654) is a new built-in exception type allowing multiple unrelated exceptions to propagate together, along with new syntax `except*` to catch them selectively. In practice, if two tasks in a TaskGroup raise errors, they bubble up as one ExceptionGroup, which can be handled with `except*` clauses. This is available in Python 3.11+ and is **fully production-ready**.

**Cancellation and Timeouts:** Hand-in-hand with TaskGroups, Python 3.11 added an `asyncio.timeout()` async context manager to limit task duration. This built-in uses cancellation internally to cancel a block if it exceeds the time limit, raising `TimeoutError` inside the block. It's a safer alternative to `asyncio.wait_for`, avoiding some edge-case pitfalls. Under the hood, both TaskGroups and timeouts rely on cooperative cancellation. (Notably, if a coroutine suppresses `CancelledError` without properly calling `task.uncancel()`, it can interfere with these mechanisms.) These features are stable in 3.11+ and mirror patterns from Trio/AnyIO, signaling Python's shift toward *async-first* concurrency.

**Coroutine Lifecycle and Introspection:** In Python 3.12, a new `inspect.markcoroutinefunction()` API was added to help frameworks treat certain callables as coroutines. This allows marking a regular sync function that returns an awaitable as a "coroutine function" so that `inspect.iscoroutinefunction` returns True. For example, a library can wrap an API that internally uses `asyncio.create_task` and mark it, improving compatibility with tools expecting native `async def` functions. This is primarily for framework authors (e.g. Django's async wrappers) and is **available in 3.12**. Other introspection improvements include more granular async stack tracing: Python 3.11 implemented PEP 657, adding fine-grained error locations in tracebacks. Now exception tracebacks pinpoint the exact expression or sub-expression that caused an error, not just the line. This is extremely useful for debugging complex expressions – the interpreter stores start/end column offsets for bytecode instructions, enabling precise traceback highlights. This feature is on by default in 3.11+ and improves developer experience (while being fully backwards-compatible).

**Exception Notes:** Python 3.11 also introduced a way to attach notes to exceptions (PEP 678). Every `BaseException` now has an `.add_note()` method to append context messages. While not specific to async, this can be helpful in an async execution service for enriching errors (for example, tagging exceptions with an execution ID or phase). This is production-ready in 3.11+. Additionally, Python 3.12's low-impact monitoring API (PEP 669) provides a new `sys.monitoring` framework for debuggers/profilers to track events with minimal overhead. This advanced feature enables nearly zero-cost tracing of calls, lines, etc., and could be relevant if the service needs to implement custom monitoring or timeout logic beyond `asyncio` facilities (PEP 669 is **available in 3.12** and used in PyCharm's debugger for performance).

## Top-Level `await` Support

**Background:** Normally, `await` can only be used inside `async def` functions. However, interactive environments (like Jupyter, IPython, or an execution service that evaluates user code on the fly) benefit from *top-level await* – allowing `await` at the session or script level. Python's solution is the compiler flag `PyCF_ALLOW_TOP_LEVEL_AWAIT`, which lets the `compile()` built-in accept top-level `await`, `async for`, and `async with` outside any function. This flag was added in Python 3.8 and is used internally by IPython and similar tools. In a SIES like Capsule, you can use this flag to compile user code so that an expression like `result = await fetch_data()` executes directly at top level without syntax errors.

**How to Use:** You compile the code in *exec* mode with the flag, then evaluate it. If the code contains top-level `await`, the compiled code object evaluates to a coroutine that must be awaited; if not, it executes normally and returns `None`. For example:

```
code_str = "import asyncio\nawait asyncio.sleep(0.1)\nprint('done')"
code = compile(code_str, "<user-code>", "exec",
flags=ast.PyCF_ALLOW_TOP_LEVEL_AWAIT)
coro = eval(code)
if coro is not None:
    await coro  # run the top-level coroutine
```

This pattern is exactly how IPython and Jupyter implement `await` in cells. It allows multiple statements, not just a single await, to be handled as a unit of execution. **Capsule's design** embraces this: its "Native Top-Level Await" feature explicitly sets `PyCF_ALLOW_TOP_LEVEL_AWAIT = 0x1000000` in the code compiler. Pyodide (Python in the browser) similarly uses this flag in its `eval_code_async` API, compiling code so that it returns a coroutine if needed. Top-level await support is **stable in CPython** (no PEP needed beyond the flag), but it is not enabled by default in the main interpreter. In an execution service, you'll manually apply the flag for user code. This is forward-compatible: if future Python versions ever allow top-level await in scripts or via a `__future__` import, they will likely build on this same mechanism.

**Event Loop Integration:** With top-level coroutines, you need an event loop running to execute them. In an async-first service, you will typically run an `asyncio` event loop in each worker process or interpreter. Python 3.11 improved `asyncio` with `asyncio.run()` (added in 3.7) as the high-level API to start an event loop and run a main coroutine. There's also discussion of an `asyncio.Runner` context manager (proposed for easier loop management) – as of 3.12, this isn't in the stdlib, but third-party solutions exist (e.g. Trio has an open design for an async runner). For Capsule, you might not need a custom Runner; you can create an event loop per session and keep it running to execute submitted coroutines one after another. Jupyter, for example, keeps a persistent event loop in the kernel to handle multiple `await` calls across cell executions.

**Stability:** The `PyCF_ALLOW_TOP_LEVEL_AWAIT` flag and compile/eval trick is proven (Jupyter has used it for years). It's production-ready for interactive execution. Just be mindful that any top-level `await` will suspend the *entire* session execution until complete – which is usually fine in a dedicated async session. Also, if user code never awaits the returned coroutine (in our example above we do `await coro` if it's not None), it would not run – so your implementation should always detect and schedule the returned coroutine. Tools like IPython automate this by checking `if inspect.CO_COROUTINE & code.co_flags:` after compilation. Capsule likely does something similar under the hood.

## Subinterpreters and Interpreter Isolation (PEP 684, 554/734)

**Multi-Interpreter Architecture:** CPython has long had the concept of subinterpreters (multiple independent Python interpreters in one process), but until recently they were limited to the C API. Two major PEPs are changing that:

- **PEP 684 (Python 3.12)** – *Per-Interpreter GIL*. This foundational change refactors CPython's runtime so **each interpreter gets its own Global Interpreter Lock**. In Python <=3.11, even if you created subinterpreters via C, they all shared the single GIL, meaning they couldn't truly run in parallel. As of 3.12, the GIL is moved into the interpreter state, so subinterpreters can execute concurrently on separate threads and CPU cores. This involved moving many formerly global runtime structures into per-interpreter storage (e.g. the object allocator, interned strings, etc.) to avoid race conditions. The result is that Python 3.12 supports **true multi-core parallelism** *within one process* – if you use multiple interpreters. Critically, extension modules must be made compatible (PEP 684 defines that only multi-phase initialized extensions, per PEP 489, are considered *isolated-safe*). To enforce this, importing a legacy single-phase extension in a new interpreter can raise `ImportError` unless you allow a bypass. Popular libraries (NumPy, etc.) are being updated, but expect that in 3.12 some C extensions will not load under per-interpreter GIL. A context manager `importlib.util.allow_all_extensions()` is provided to temporarily relax this check if needed [1]. PEP 684 landed in 3.12 and is **enabled by default** for new interpreters (the C-API `Py_NewInterpreter()` default now gives each interpreter its own GIL). For backwards compatibility, the main interpreter and legacy API remain a single GIL unless opted-in. In short, Python 3.12 "unblocks" using subinterpreters for parallel tasks, which is a game-changer for CPU-bound workloads.

- **PEP 554 / 734 (Python 3.13+)** – *Multiple Interpreters in the Stdlib*. This proposes a high-level Python `interpreters` **module** to create and manage subinterpreters directly from Python code. PEP 554 (now superseded by the cleaner PEP 734) was recently **accepted** (mid-2025) and slated for Python 3.14. The goal is to expose an API where you can do, e.g., `interp = interpreters.create()` to spin up a new interpreter (in the same process) and `interp.exec(code)` to run code in it [2] [3]. Each interpreter is isolated: it has its own `__main__` module, globals, module imports, etc., and (thanks to PEP 684) its own GIL. The module also defines lightweight communication primitives: you can **share data between interpreters** via special *shareable objects* and **channels**. For instance, `interpreters.create_channel()` yields a `(RecvChannel, SendChannel)` pair that acts like a FIFO pipe for basic types (bytes, str, int, etc.) [4] [5]. You can send data through a channel from one interpreter and receive it in another, with the Python runtime handling the necessary data copying or proxying [6] [7]. This is minimal by design – initially only a few immutable builtins are shareable (None, bool, numbers, str, bytes, plus memoryview for binary data) [5] [8]. The **Interpreter API** allows setting up an interpreter's environment via `interp.set_main_attrs()` (to prepopulate variables in its `__main__` namespace) and retrieving results via `interp.get_main_attr()` [9] [10]. There's also plan for `concurrent.futures.InterpreterPoolExecutor` to manage a pool of subinterpreters similar to a ProcessPool, making it easy to distribute tasks across cores without multiprocessing overhead.

*Status:* As of Python 3.13, PEP 734 is expected to be implemented (it may land late in 3.13 or in 3.14 if not ready sooner). RealPython notes that **PEP 554 was intended for 3.13** (with 3.12 only having the GIL groundwork). Indeed, the new interpreters module is highly anticipated as it "opens up true multi-core parallelism" in pure Python. If you are planning for forward-compatibility, design your service to optionally use subinterpreters when available (falling back to subprocesses on older versions).

**Interpreter Lifecycle and Isolation Semantics:** Each interpreter can be thought of as a *completely separate Python world* running on a shared process. They do not share Python objects or most state (other than a few truly global resources like open file descriptors and certain singletons) [11]. Notably, **memory is only isolated at the Python object level, not OS level** – a bug in one interpreter (or a malicious extension) could still crash the process affecting all interpreters. Thus, subinterpreters are lighter weight than OS processes but not a security sandbox. PEP 554 imposes some **restrictions** for safety: in interpreters created via the new API, you **cannot spawn OS threads marked as daemon, fork new processes, or exec a new program** (to prevent confusing interactions) [12]. The restriction on `os.fork()` means you cannot use the `multiprocessing` module from within a subinterpreter (since its default on Unix is fork) [12]. The rationale is that subinterpreters are an alternative to forking new processes, so forking *from inside one* is undesirable. (Launching a completely separate process via `subprocess` is still allowed, as it uses a fork+exec under the hood, which the runtime treats differently [12].) Also, an interpreter's **state is never reset** between `exec()` calls – variables, imports, and definitions persist in that interpreter's `__main__` module across multiple `interp.exec` calls [13] [14]. This is analogous to a Jupyter kernel session or a REPL: consecutive executions see each other's side effects. Capsule leverages this as "Namespace Persistence" – state preserved across executions. If you need to clear state, you would have to close the interpreter and start a fresh one (or manually del variables). The persistent state model is great for interactive sessions but means your service should monitor memory growth (unused objects lingering in a long-lived interpreter).

**Using Subinterpreters vs Subprocesses:** Once available, subinterpreters will enable an execution service to run many isolated sessions on one process with much lower overhead than full subprocesses. Context switches are faster (no OS process switch) and data exchange via channels is direct in-memory copying, avoiding serialization to bytes as with multiprocessing pipes. PEP 734 even suggests an interpreter-based Executor to parallelize Python code more efficiently than threads or processes. However, be cautious: if your use-case requires *strong isolation* (especially for untrusted code), OS processes or containers are still safer. A rogue infinite loop in a subinterpreter can be cancelled (since you can .close() or kill its thread), but a rogue C extension in a subinterpreter could corrupt the whole process. In practice, a mix might be ideal: use subinterpreters for parallel **compute** isolation (multiple tasks on cores), but keep a process boundary for **security isolation** when running arbitrary user code. (Capsule's approach is to use OS processes —"subprocess-isolated" by design—for security, augmented by async for concurrency within each process.)

**Current State:** PEP 684 (per-interpreter GIL) is **fully implemented in Python 3.12** – you can create multiple interpreters via C and run them on different threads in parallel. PEP 554/734 (interpreters module) is

**accepted and incoming**; an early preview exists as a PyPI module (`interpreters-pep-734`) for Python 3.12+ that can be used for experimentation. We expect by Python 3.14, the interpreters API will be a standard feature. Until then, the tried-and-true method for isolation is separate processes (see below).

## Multi-Process Execution and Orchestration Strategies

Even before subinterpreters become available to Python code, **subprocess-based isolation** has been the norm. There are a few patterns to consider:

- **Forking vs Spawning:** On POSIX, using `os.fork` (as `multiprocessing` does by default on Linux) is fast for process creation but can lead to issues if the parent process has threads (forking a multi-threaded process is unsafe). Python's `multiprocessing` defaults to *spawn* on Windows and macOS (as of Python 3.8) to avoid those issues. In a service like Capsule, **preforking** a pool of worker processes might minimize startup costs: you can fork a template process that has preloaded modules to amortize import overhead, then isolate each session in one fork. However, Python itself has moved away from forking in library code due to the aforementioned safety issues. Instead, you might run a master process that *spawns* child processes on demand or maintains a pool of idle workers.

- `subprocess` **Module:** The lowest-level approach is using the `subprocess.Popen` API to launch an external process (which could be `python` itself running a script or an interactive session). This gives complete OS isolation (each process has its own memory space). It's robust but incurs overhead of starting a Python interpreter each time. Capsule's GitHub project "capsule-run" addresses this by using a single static binary (written in Rust) that isolates commands with Linux namespaces and seccomp, achieving sub-50ms cold start times – effectively treating each command as a mini-container. In pure Python, achieving <50ms startup per process is hard (the Python interpreter alone typically takes ~20-30ms to start). To mitigate this, you can keep processes alive and reuse them for multiple executions (a pool of persistent workers). This is essentially what Jupyter does with its kernel process: the process stays running for the whole notebook session, executing successive inputs.

- `multiprocessing` **/** `concurrent.futures.ProcessPoolExecutor`**:** These provide a higher-level interface to launching worker processes and dispatching tasks. A ProcessPoolExecutor in Python 3.11/3.12 is quite mature and can manage a pool of workers for you. However, for an execution service where you want *persistent state per session*, a static pool is not enough – you likely need to *pin* each session to a specific process that persists its state (rather than stateless task workers). You could still use multiprocessing primitives (like `multiprocessing.Queue` or `Pipe`) to communicate between a session process and the manager process. Keep in mind that everything sent through those must be picklable. Complex user objects might not pickle well, so your protocol might need to stick to basic types or use a serialization like JSON or cloudpickle for more flexibility.

- **Communication Protocol:** Capsule's design mentions *"Protocol-Based IPC"* with structured message passing and promise-based correlation. This suggests that the manager and worker processes talk via an IPC channel (perhaps stdio pipes or sockets) using a message format (likely JSON as shown in capsule-run's JSON API). Each execution request/response is labeled with an execution ID (as in capsule-run's `--execution-id` option) so that responses can be matched to requests – essentially a promise/future system over IPC. Python's stdlib doesn't provide a full RPC mechanism out of the box, but you can build one with `asyncio.StreamReader/Writer` on a socket or by reading/writing JSON lines to a pipe. Using an **async IO** approach here is beneficial: e.g. one process can listen to multiple worker pipes concurrently using `asyncio` without blocking.

- **Thread Offloading for Blocking I/O:** Within each process (or interpreter), if you run user code that does blocking I/O (like a database call or reading a file), it will block the async event loop unless addressed. Capsule's roadmap indicates a **ThreadedExecutor for blocking operations**. In practice, this can be implemented by using `loop.run_in_executor()` to offload blocking functions to a thread pool. Python's default loop has a thread pool (`ThreadPoolExecutor`) accessible via `asyncio.get_running_loop().run_in_executor(None, blocking_func)`. You might maintain a small pool of threads in each session for I/O tasks. This way, your async tasks can offload, say, a `requests.get()` (which is blocking) to a thread and meanwhile the event loop can schedule other tasks. This hybrid model (async for high-level orchestration, threads for CPU-bound or blocking calls) is quite common in high-performance async services.

**Durability and Recovery:** Using separate processes has the advantage that if a worker process crashes (due to a segmentation fault or `kill -9` or out-of-memory), it doesn't take down the manager or other sessions. The manager can detect the death (via process exit status) and restart a new process, then use persisted state (if any) to recover. One approach to persistence is **checkpointing**: at intervals, serialize the session state (variables, results, etc.) to disk or an external store so it can be reloaded. Python does not have a built-in "checkpoint interpreter" facility, but one can imagine pickling critical objects. In practice, full automatic checkpointing of an interpreter (including stack frames) is very complex. Instead, systems like **Resonate SDK** and **Temporal.io** take a different approach: they model long-running functions as state machines that periodically save their progress (e.g. the function yields at checkpoints and the state is saved as a record). Resonate, in particular, provides **"promises"** that survive process crashes – if a worker goes down, another can resume the promise's computation from the last checkpoint. Capsule's *DurableSession* example shows executing code with a `checkpoint_interval=10` seconds. Likely, under the hood, the session periodically serializes some global state or at least the instruction pointer (perhaps by splitting the code into chunks or using asyncio checkpoints).

If implementing your own, you might restrict what can be checkpointed (e.g. only data in a certain structure). Alternatively, one can use **persistent storage for state**: for example, store variables in a database or in shared memory, so if a new process starts, it can re-import that state. Python's `pickle` or third-party `dill` can serialize many (not all) Python objects, so one strategy is to pickle the `globals()` of the session after each execution. This would allow restoring most simple objects. However, things like open file handles, network connections, or running coroutines can't be pickled and would need special handling (capability-based design helps here: encapsulate I/O in managed objects that know how to reconnect or cannot be persisted, forcing a reconnect on recovery).

**Concurrency vs Isolation Trade-offs:** A SIES has to balance *concurrency* (running tasks in parallel) with *isolation* (keeping tasks from interfering). In Capsule's architecture, each session is isolated in its own process (for security and state segregation), and within that process, the session can execute code concurrently via async (tasks on an event loop) or threads. With Python 3.12+, an alternative is to have one OS process with multiple interpreters: you gain parallelism and some isolation (separate GILs and module namespaces) without the overhead of multiple OS processes. But, as noted, this is a newer and less battle-tested approach. Initially, you might continue to use processes (especially if using containers or sandboxing like capsule-run for security) and gradually adopt subinterpreters in contexts where OS process overhead is a bottleneck and security is less of a concern.

## Free-Threaded Python (No GIL) – The Future of Concurrency

Perhaps the most radical upcoming change is **PEP 703**, which proposes making the GIL optional (i.e. allowing a build of CPython with **no global lock**). This PEP has been **accepted** by Python's Steering Council in 2023, with the caveat that it will be a gradual, opt-in rollout. The plan is to introduce a compile-time flag `--disable-gil` that produces a **GIL-free Python build**. In such a build, multiple threads can truly execute Python bytecode in parallel. Sam Gross's "nogil" prototype (which PEP 703 is based on) showed that it's possible to remove the GIL with manageable impact to single-thread performance (using techniques like biased reference counting and per-object locks) [15]. The steering council mandates that the standard (GIL-enabled) build remains the default for now, and the no-GIL mode will debut as an experimental alternative. This means in Python 3.13 or 3.14, we might see **two distributions** of CPython: one regular, one GIL-free. It's expected that the no-GIL build will require extension modules to be rebuilt and perhaps modified (PEP 703 introduces a `PyThreadState_Enter/Exit` API and changes refcount implementation). So adoption will take time.

For an execution service, **free-threading could eventually simplify certain things**: e.g. you could run multiple sessions as threads in one process rather than separate processes or interpreters, without GIL contention. However, without additional isolation, threads in one interpreter still share all memory, so this doesn't provide safety – you'd likely still want separate interpreters or processes for separate user sessions. No-GIL Python will shine for data-parallel computation (using threads within one task). For Capsule's use-cases (isolated sessions), no-GIL might allow each session process to handle more threads concurrently (for example, a single session could spawn true parallel worker threads for a data task). It could also make thread offloading of I/O in an async loop more efficient (since those threads can run Python code concurrently).

**Status:** PEP 703 is **in progress** with partial implementations. It probably won't be the default in any 3.x release; it will be opt-in. The acceptance of PEP 703 means the core devs are committed to exploring it, but they have left open the possibility of reverting if it proves too problematic. In planning forward, design your service to not *assume* the GIL. For instance, with PEP 684 and 703 combined, you could have a single process with multiple interpreters each on separate OS threads, and if the build is no-GIL, even threads in the same interpreter could run in parallel. This leads to interesting scenarios: e.g. a single session (one interpreter) with multiple threads doing work could parallelize CPU-bound tasks (something not beneficial today due to GIL). If Capsule integrates with scientific computing or AI workloads, a no-GIL build could allow using threads instead of processes for CPU parallelism, avoiding serialization overhead (PEP 703 specifically cites AI training and bioinformatics as beneficiaries).

**Recommendation:** Keep an eye on the no-GIL build progress. It's likely to land behind a configure flag in 3.13+, so one could run a Capsule server on a no-GIL CPython for potentially better throughput. However, extensive testing would be needed – the ecosystem's C extensions need to catch up to ensure thread safety. For now, PEP 703 is *emerging* and experimental; do not use it in production until further notice.

## Durable Execution and State Management

One key feature of Capsule is **durable sessions** – the ability to recover state after crashes and even distribute execution across machines (via Resonate promises). Achieving durability involves carefully managing state:

- **Persistent Namespace:** In a long-running session, the user's variables and imports live in memory (globals in the interpreter). To persist these, you can periodically snapshot them. A simple approach is to store the session's `globals()` dict to disk (perhaps using `pickle` or JSON for simpler objects). More advanced, you might only log *changes* to state (an operation log or event sourcing). For example, record each execution's inputs and outputs so you can replay them on a fresh interpreter. PEP 554 actually notes that `Interpreter.exec()` is like concatenating code in one environment [14] [16] – so replay is literally re-running all past code. That's a brute-force recovery (like how a notebook can be "Restart & Run All"). For automated recovery, a checkpointing interval is friendlier: e.g. every N commands or seconds, serialize the key state. This can be tricky if the state includes open network connections or file handles. Capsule's *capability-based architecture* suggests isolating external interactions (I/O, network) behind injected functions. This way, the core state is mostly pure data and easier to serialize, and external resources can be reinitialized on restore.

- **External Storage and Promises:** Resonate SDK provides a model where function calls (promises) are logged to an external server. If a worker dies, another can fetch the last promise state and continue. The **Resonate Python SDK** allows marking functions as "durable" – you call `foo.run(id, args)` to execute locally with a given invocation ID, and the SDK ensures that if it's called again with the same ID it resumes rather than duplicates work. Under the hood, Resonate likely checkpoints the function's progress and stores results in a backing store (like a database or their own service). In Capsule's context, Resonate integration means you could offload durability to Resonate: each execution block might be wrapped in a promise that automatically saves results and can restart on a new host if needed. This is an emerging pattern (similar to Azure Durable Functions or Temporal workflows) applied to Python.

- **Transaction and Rollback:** Capsule mentions *Transaction Support* with rollback policies. This implies that if an execution fails mid-way, the state can be rolled back to a previous checkpoint. Implementing rollback might involve keeping multiple snapshots (before and after execution) or using an append-only state log and ignoring entries from a failed transaction. Python doesn't provide this natively, but you can impose it at the application level: e.g., before running a code block, copy the `globals()` dict (shallow copy of keys to values) and if the block fails, restore that copy. This won't undo side-effects in external systems, but for in-memory state it works. Another approach is running each user code block in a fresh *temporary interpreter or process*, then only applying changes to the main session state if it succeeds. PEP 554 could assist here: you could create a scratch subinterpreter, run the code, then transfer objects (that are shareable) back to the main session interpreter. Non-shareable objects would need special handling though. Alternatively, if using OS processes, you could simply discard a process on failure and respawn a new one with last good state.

- **Structured Error Handling:** With ExceptionGroups, if a code execution raises multiple errors (e.g. from parallel tasks), you can catch them, log or store them, and decide how to proceed. Python 3.11's except* could let you separate, say, a benign error from a fatal one. For durability, likely any uncaught exception means the execution failed and should trigger a rollback or retry. But

ExceptionGroup could be useful if, for instance, your service runs multiple user code snippets in parallel (maybe in different interpreters or threads) and you want to aggregate their results/errors.

**Production Readiness:** Features like checkpointing and durable promises are not part of stdlib, but proven patterns exist in industry (Resonate, Temporal, Prefect, etc.). These usually rely on an external system (database or scheduler) to store state. If Capsule integrates Resonate, a lot of heavy lifting (like promise correlation, state storage, crash detection) is handled by Resonate's server. Without it, you'd implement a simpler version: e.g. have a manager process monitor worker heartbeat; if a worker dies, launch a new one, then use a saved globals snapshot to restore variables. Python's built-in `marshal` or `pickle` can save basic data quickly, but be mindful of security (never unpickle untrusted data) and versioning (code changes might break pickles).

## Protocol-Level Improvements and Best Practices

Finally, a grab-bag of other relevant enhancements from Python 3.11–3.13 that bolster an async execution service:

- **Improved Introspection of Code Objects:** Python 3.11 made code objects carry rich info like end-line and column offsets (for PEP 657). Also, Python 3.11 added `types.TracebackException` which makes it easier to convert traceback info to a serializable form for logging or sending over IPC [17]. In an IPC-based service, you typically don't send raw exception objects across (they may not pickle), but you can send the `traceback.TracebackException.format()` output or the Exception's `__str__`. The PEP 554 spec itself notes that when an uncaught exception occurs in a subinterpreter, the `interpreters` module will raise a `RunFailedError` in the caller, with the original exception's info attached (as a surrogate) [17]. This prevents cross-interpreter tracebacks from leaking objects, instead providing a cleaned error representation. This design could inspire how you propagate errors from a worker process back to the client: don't try to send the live exception object; rather, serialize its type, message, and traceback. Python's standard `traceback` module or `traceback.TracebackException` (added in 3.5) can help format that in a nice way.

- **Exception Groups and Task Groups for Structured Results:** We touched on ExceptionGroup for errors. Similarly, if your service executes multiple pieces of code concurrently (say, fan-out to multiple shards), Python now has better primitives to handle their completion. `asyncio.gather(..., return_exceptions=True)` was the old way to collect errors without stopping others; with TaskGroup, you would not use `return_exceptions` but rather catch ExceptionGroup. This gives you structured access to each exception. You can then send a combined error report to the user. This is especially relevant if Capsule ever supports running user code in parallel (maybe not typical in a single session, but possible in a distributed scenario).

- **Inspection of Coroutines:** Python 3.11 adds `inspect.getcoroutinelocals(coro)` and `inspect.getcoroutinestate(coro)` which can introspect a suspended coroutine object's local variables and state. This could be useful for debugging or even for persistence (in theory, one could serialize a suspended coroutine's locals, though restoring is not trivial). There's also `inspect.getframeinfo` improvements that, combined with PEP 657, let you get finer detail on where in user code something is executing.

- **Async Generator Enhancements:** If your service uses async generators (e.g. streaming output via `yield` in async context), note that Python 3.12 introduces some performance improvements and 3.11 made their finalization more predictable. Notably, PEP 533 (completed earlier) ensured that `asyncgen.aclose()` is called on garbage collection to avoid warnings. This is low-level, but in a long-running service, you want to avoid resource leaks from generators left open.

- **AnyIO and Trio patterns:** While not standard library, these projects influence best practices. For example, *structured concurrency* as an idea means cleaning up child tasks automatically – Capsule likely uses TaskGroups internally to run user code tasks so that it can cancel them on timeout or session close. The service might also implement **timeouts** for each execution (to avoid runaway code). Python 3.11's `asyncio.timeout()` makes that straightforward: e.g. `async with asyncio.timeout(5): await session.execute(code)` will raise TimeoutError if it takes more than 5 seconds. Under the hood, Capsule could wrap user code execution in such a timeout to enforce resource limits (as capsule-run does via a watchdog in Rust, but at Python level you can do similarly albeit with less precision).

- **Resource Cleanup on Interpreter Shutdown:** If using subinterpreters, note that when you `.close()` an interpreter, it will destroy all objects in it. PEP 554 defines that it's an error to close an interpreter that is running code [18]. So you must ensure no code is executing (or explicitly kill its threads) before closing. Similarly, when terminating a process, you'd want to clean up (or just let the OS clean up). Python's `atexit` handlers run on normal termination, but if you `os._exit` or crash, they won't. For durability, you often intentionally *don't* rely on atexit (instead, the manager reconstructs state from logs).

- **Security and Capabilities:** Although not a direct Python language feature, it's worth noting that Python removed the old "restricted execution" mechanism long ago. Instead, Capsule's approach (capability-based injection) means you give the executed code limited functions for I/O, etc. In Python, you can achieve this by controlling the `globals` you pass to `exec`. For instance, supply a `{'__builtins__': minimal_builtins, 'safe_open': safe_open_func, ...}` dictionary to restrict what the code can call. There is no built-in sandbox, so this is by construction. One can use libraries like `restricted_python` or run code in a container (Capsule uses Linux sandboxing via capsule-run for true isolation [19]). At the Python level, ensure your execution service never `exec`s untrusted code with full access to the real `builtins` or `os` – always curate the environment. This ties into the design of being language-agnostic (Capsule is 70% language-independent, suggesting it could swap out Python for another runtime).

In summary, **Python 3.11–3.13 bring a wealth of features that align with Capsule's goals**: first-class async/await with structured concurrency for managing execution *within* a session; interpreter-level isolation and no-GIL efforts for maximizing parallelism *between* sessions; and better introspection, error handling, and debug support to build robust, maintainable systems. Most of these features are already stable (TaskGroup, ExceptionGroup, top-level await via `compile`, per-interpreter GIL in 3.12, etc.), whereas the multi-interpreter stdlib API and no-GIL are on the horizon for 3.14 and beyond (experimental in 3.13). By architecting with these in mind – e.g. designing your system to leverage channels or to easily switch to subinterpreters – you can ensure Capsule remains cutting-edge and forward-compatible with the Python concurrency landscape.

**Sources:**

- PEP 684 – Per-Interpreter GIL (Python 3.12)
- PEP 554/734 – Multiple Interpreters and `interpreters` Module
- Python 3.11 AsyncIO TaskGroup and ExceptionGroup docs
- Stack Overflow – using `PyCF_ALLOW_TOP_LEVEL_AWAIT` for top-level `await`
- Pyodide documentation – `eval_code_async` uses top-level await flag
- PEP 654 – Exception Groups motivation (Trio influence)
- Python 3.12 Inspect – `markcoroutinefunction` added
- PEP 669 – Low-impact monitoring (sys.monitoring) in 3.12
- Capsule README – SIES design and features
- Capsule Roadmap – plans for AsyncExecutor, Resonate integration, top-level await, etc.
- Resonate SDK docs – durable async application model
- Real Python Preview of Python 3.12/3.13 – subinterpreters and GIL changes
- Python Steering Council acceptance of PEP 703 (No GIL)
- PEP 554 draft – interpreter API and restrictions (no fork, etc.) [12] [20]
- asyncio.timeout and TaskGroup usage (Python docs)

---

[1] PEP 684 – A Per-Interpreter GIL | peps.python.org
https://peps.python.org/pep-0684/

[2] [3] [4] [5] [6] [7] [8] [9] [10] [12] [13] [14] [16] [17] [18] [20] PEP 554 – Multiple Interpreters in the Stdlib | peps.python.org
https://peps.python.org/pep-0554/

[11] PEP 734 – Multiple Interpreters in the Stdlib | peps.python.org
https://peps.python.org/pep-0734/

[15] PEP 703 – Making the Global Interpreter Lock Optional in CPython | peps.python.org
https://peps.python.org/pep-0703/

[19] GitHub - haasonsaas/capsule-run: Lightweight, secure sandboxed command execution for AI agents
https://github.com/haasonsaas/capsule-run