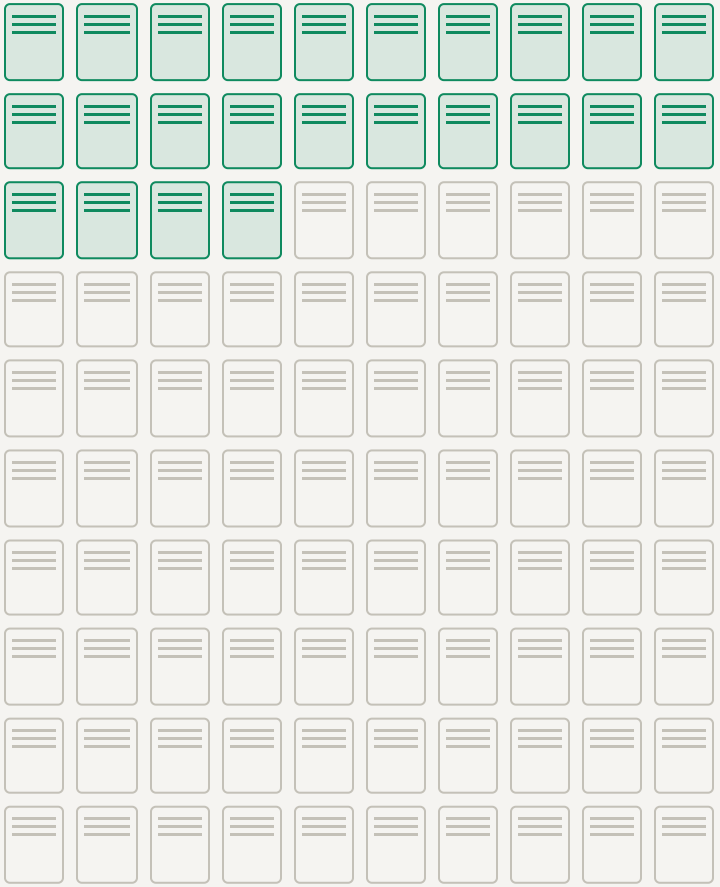
# While We Await

From Python threads to
asyncio, Tokio, and async Python-Rust interop

Terence Liu (𝕏) @ TNHC Data Systems

November 2026



## 01 · Python async intro

Threads, coroutines, and concurrency

## 02 · Inside Python asyncio

Event loop, I/O readiness, and task wakeups

## 03 · Inside Rust Tokio

Polling futures, wakers, and the reactor

## 04 · asyncio ↔ Tokio interop

Bridging runtimes, results, and cancellation

# Overlapping network waits with threads and asyncio.

### The workload

Fetching many pages

### Thread pools

Blocking I/O in worker threads

### asyncio

Cooperative task scheduling

# Coordinating network I/O examples

## Web server

Incoming requests

Serve many clients.

While one request waits on a database or another service, handle other requests.

## Fan-out utility

Outgoing requests

Fetch many resources for one job.

While one response is pending, make progress on other downloads.

# One request at a time

```python
def fetch(client: httpx.Client, url: str) -> bytes:
    response = client.get(url)
    response.raise_for_status()
    return response.content
```

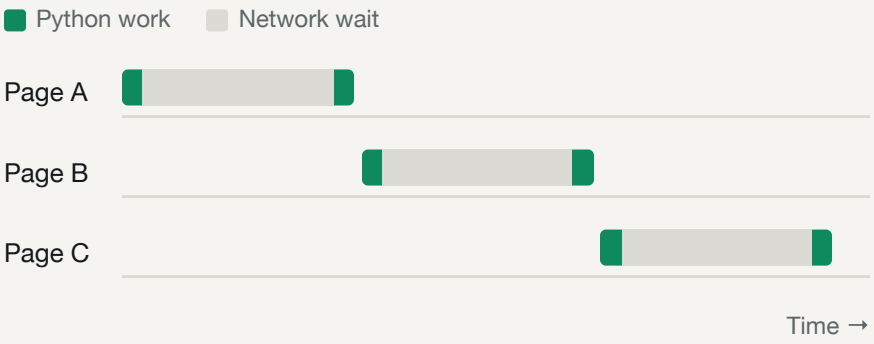
```python
def sequential(urls: list[str]) -> list[bytes]:
    with httpx.Client() as client:
        return [fetch(client, url) for url in urls]
```

Each call waits for the **complete response** before the next starts.

# Overlap the waiting

## Sequential

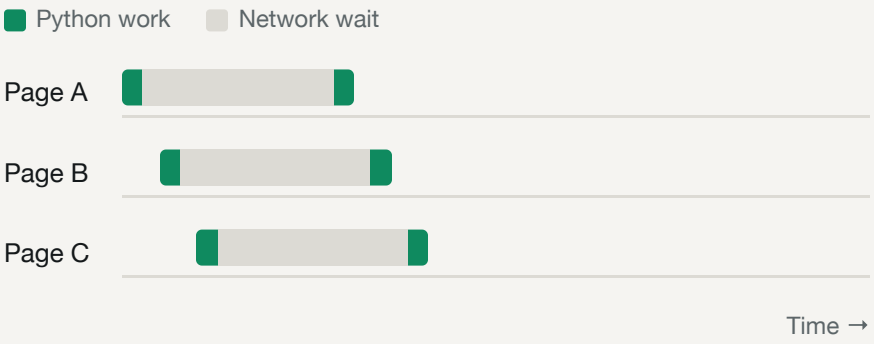
One outstanding request




Shared time scale · 3 of 100 requests · schematic

## Concurrent

Multiple outstanding requests




Shared time scale · 3 of 100 requests · schematic

# Downloading with a thread pool

```python
def threaded(urls: list[str], limit: int = 20) -> list[bytes]:
    with httpx.Client() as client:
        with ThreadPool(limit) as pool:
            return list(pool.imap_unordered(partial(fetch, client), urls))
```

`partial(fetch, client)` binds `client`, leaving `url` for the pool to supply. Equivalent here to `lambda url: fetch(client, url)`.

# Threads and asyncio tasks

## Thread pool

Blocking I/O

- Worker 1 → waiting inside a socket call
- Worker 2 → waiting inside a socket call
- Worker 3 → executing Python code

The OS schedules native threads.

## asyncio

Nonblocking socket I/O

- Event-loop thread → runs ready work
- Task A → suspended, waiting for I/O
- Task B → suspended, waiting for I/O

The loop schedules ready task steps.

# Synchronous and asynchronous requests

## Synchronous

Called by a worker thread

```
def fetch(
    client: httpx.Client, url: str
) -> bytes:
    response = client.get(url)
    response.raise_for_status()
    return response.content
```

## Asynchronous

Awaited within a task

```
async def fetch_async(
    client: httpx.AsyncClient, url: str
) -> bytes:
    response = await client.get(url)
    response.raise_for_status()
    return response.content
```

# Downloading with a concurrency limit

## Thread pool

Worker limit

```python
def threaded(
    urls: list[str], limit: int = 20
) -> list[bytes]:
    with httpx.Client() as client:
        with ThreadPool(limit) as pool:
            return list(pool.imap_unordered(
                partial(fetch, client), urls
            ))
```

## asyncio

Semaphore limit

```python
async def asynchronous(
    urls: list[str], limit: int = 20
) -> list[bytes]:
    async with httpx.AsyncClient() as client:
        slots = asyncio.Semaphore(limit)

        async def bounded(url: str) -> bytes:
            async with slots:
                return await fetch_async(client, url)

        return await asyncio.gather(
            *(bounded(url) for url in urls)
        )
```

# Async does not automatically mean concurrent

## Still sequential

One fetch at a time

```
pages = [
    await bounded(url) for url in urls
]
```

## Concurrent downloads

gather schedules each coroutine

```
pages = await asyncio.gather(
    *(bounded(url) for url in urls)
)
```

# Choosing between threads and asyncio

## Thread pool

20 workers

- Existing blocking libraries.
- Native stack per worker.
- Shared-state coordination.

## asyncio

One event-loop thread

- Async-aware libraries.
- Coroutine state per task.
- Blocking code stalls the loop.

# Following an `await` through the event loop.

### Run ready work

Tasks and callbacks

### Wait for I/O

OS readiness notifications

### Resume the task

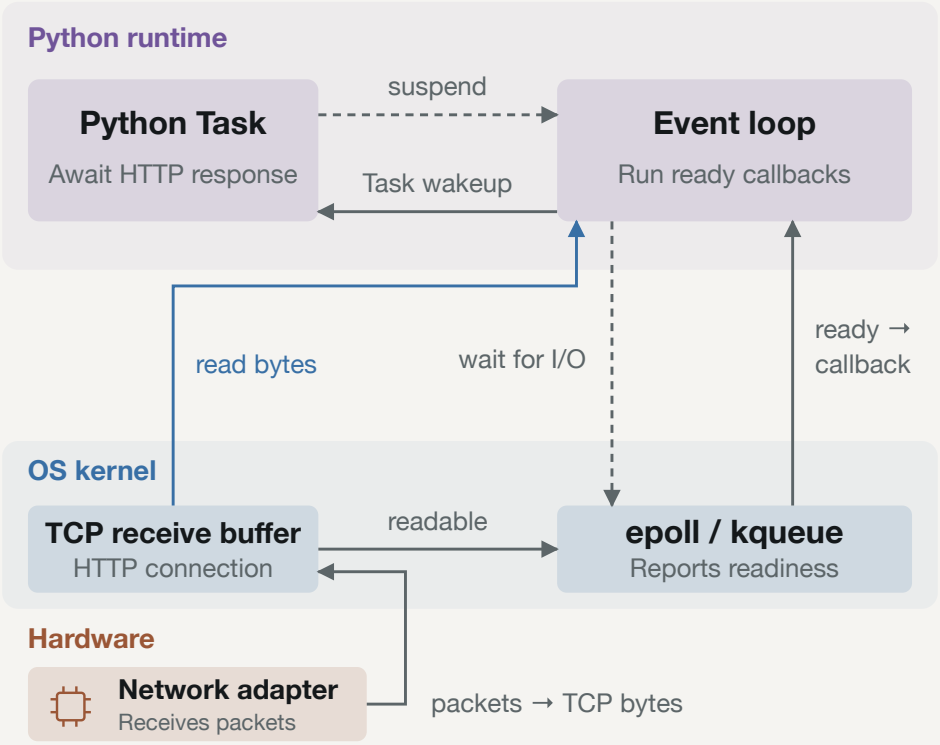
Future completion and wakeup

# Waiting for an HTTP response

```python
async def main():
    async with httpx.AsyncClient() as client:
        response = await client.get(
            "https://example.com"
        )
    return response.text

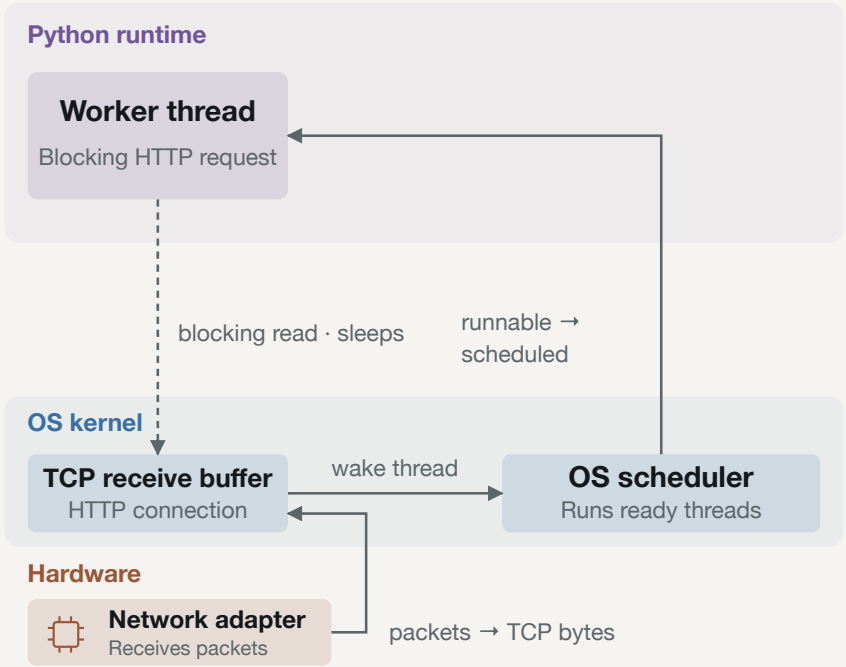

page = asyncio.run(main())
```

HTTPX manages the connection and I/O.

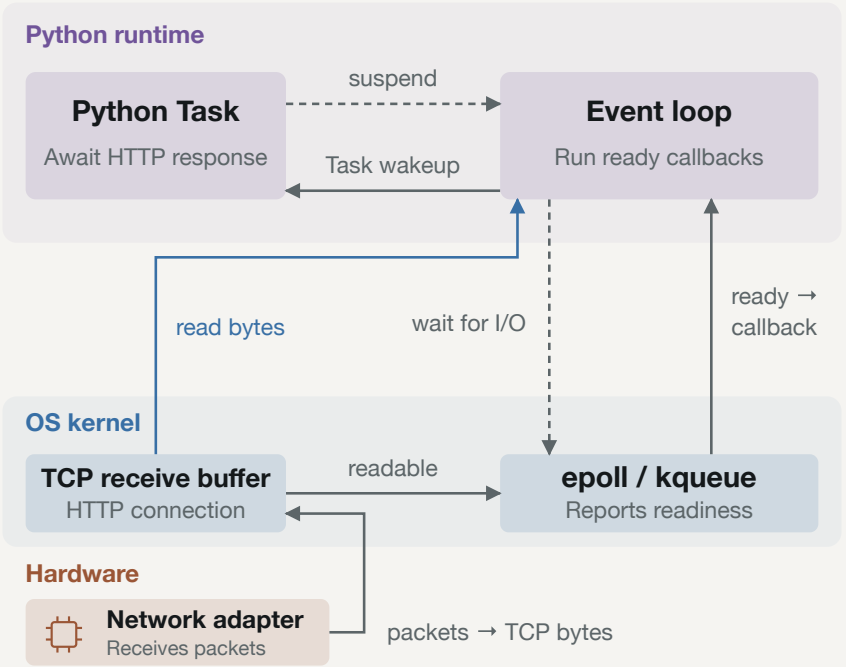


# Blocking threads vs. event loop

## Threads · block a worker




## asyncio · suspend a Task





# Following an `.await` through Tokio.

### Futures

Polling async work

### Scheduling

Executor and Waker

### I/O readiness

Reactor and Mio

# Async Rust: the same request

## Python · asyncio + HTTPX

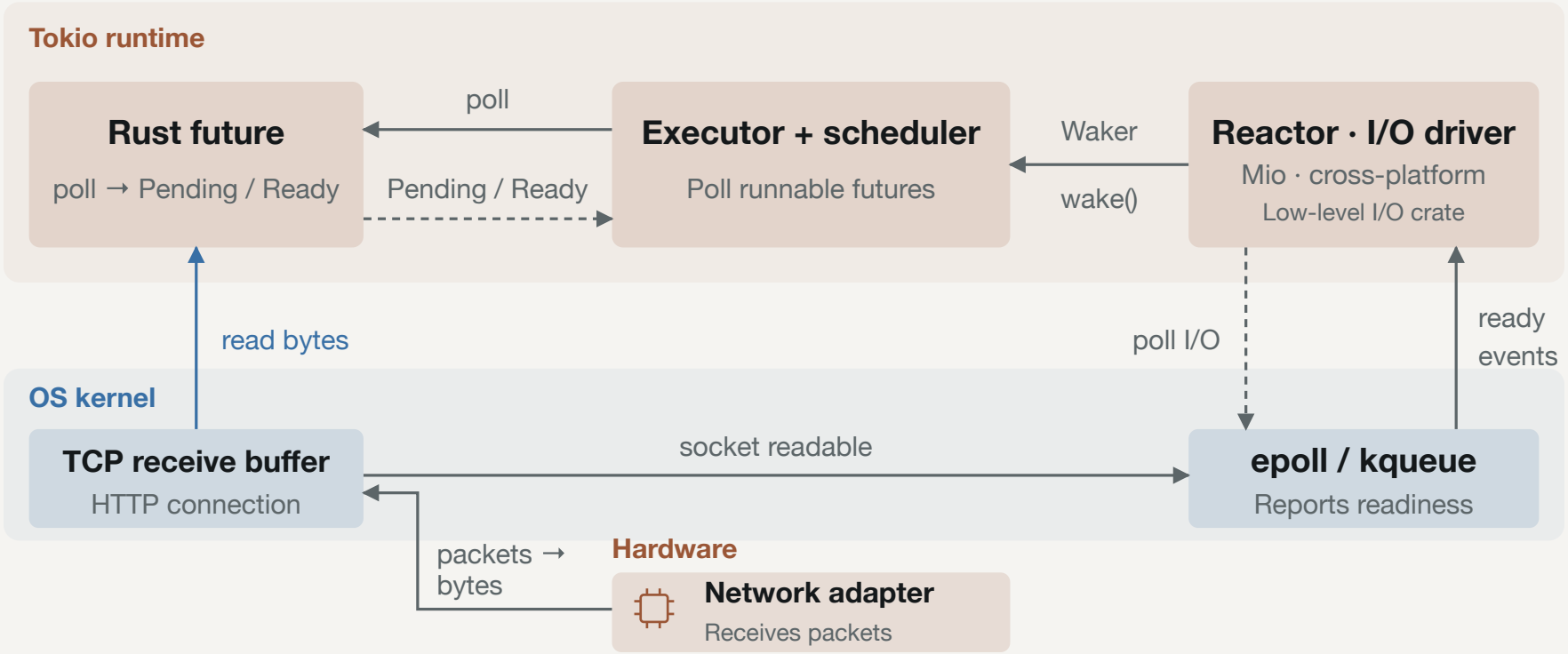
```python
async def main():
    async with httpx.AsyncClient() as client:
        response = await client.get(
            "https://example.com"
        )
    return response.text


page = asyncio.run(main())
```

## Rust · Tokio + reqwest

```rust
#[tokio::main(flavor = "current_thread")]
async fn main() -> Result<(), reqwest::Error> {
    let client = reqwest::Client::new();
    let response = client
        .get("https://example.com")
        .send()
        .await?;
    let _page = response.text().await?;
    Ok(())
}
```

# Inside the Tokio runtime





# Connecting the runtimes with `pyo3-async-runtimes`.

### Python calls Rust

A Python awaitable for Rust work

### Rust calls Python

A Rust future for Python work

### Runtime handoffs

Scheduling and completion

# Python awaits Rust

## Python · application

```python
async def main():
    result = await bridge.rust_sleep()
    print(result)


asyncio.run(main())
```

## Rust · extension

```rust
#[pyfunction]
fn rust_sleep<'py>(
    py: Python<'py>,
) -> PyResult<Bound<'py, PyAny>> {
    future_into_py(py, async move {
        sleep(Duration::from_millis(100)).await;
        Ok("done")
    })
}
```

# Rust awaits Python

## Rust · application

```rust
#[pyo3_async_runtimes::tokio::main]
async fn main() -> PyResult<()> {
    let future = Python::attach(|py| {
        let demo = py.import("demo")?;
        let coro = demo.call_method0("py_sleep")?;
        into_future(coro)
    })?;
    let result = future.await?;
    Python::attach(|py| {
        let text = result.extract::<String>(py)?;
        println!("{text}");
        Ok(())
    })
}
```

## Python · embedded module

```python
async def py_sleep():
    await asyncio.sleep(0.1)
    return "done"
```

# Connecting asyncio and Tokio



# Python attachment and the GIL

## Rust · attachment scope

```rust
let receiver = Python::attach(|py| {
    let demo = py.import("demo")?;
    let coro = demo.call_method0("py_sleep")?;
    into_future(coro)
})?;
let result: Py<PyAny> = receiver.await?;
```

Keep the highlighted scope brief. Await after it ends.

## Accessing Python

- `Python<'py>` proves attachment.
- `Bound<'py, T>` stays within that lifetime.
- `receiver.await` yields an owned `Py<PyAny>`.

While holding the GIL, don't synchronously wait for work that needs Python to run.

# Python receives Rust failures

## Python · application

```python
async def main():
    try:
        await bridge.rust_error()
    except ValueError as error:
        print(error)


asyncio.run(main())
```

## Rust · extension

```rust
#[pyfunction]
fn rust_error<'py>(
    py: Python<'py>,
) -> PyResult<Bound<'py, PyAny>> {
    future_into_py(py, async {
        Err::<(), _>(
            PyValueError::new_err("bad input")
        )
    })
}
```

Shown: `Err(PyValueError)` → `ValueError`.

A separate `panic!` inside this async block would become `RustPanic`.

# Rust receives Python failures

## Rust · application

```rust
#[pyo3_async_runtimes::tokio::main]
async fn main() -> PyResult<()> {
    let pending = Python::attach(|py| {
        let demo = py.import("demo")?;
        into_future(demo.call_method0("py_fail")?)
    })?;
    match pending.await {
        Ok(_) => println!("completed"),
        Err(err) => eprintln!("{err}"),
    }
    Ok(())
}
```

## Python · embedded module

```python
async def py_fail():
    await asyncio.sleep(0.1)
    raise ValueError("bad input")
```

The `raise ValueError(...)` reaches Rust's `Err(err)` branch as a `PyErr`.

# Python cancels Rust work

## Python · application

```python
async def main():
    pending: asyncio.Future = bridge.rust_wait()
    await asyncio.sleep(0.05)
    pending.cancel()
    try:
        await pending
    except asyncio.CancelledError:
        print("Python stopped waiting")


asyncio.run(main())
```

## Rust · spawned work

- The Python Future signals cancellation.
- Rust drops the pending future when it observes cancellation.
- Blocking code and separately spawned work are not interrupted.

# Rust stops waiting for Python

## Rust · result receiver

```rust
let receiver = Python::attach(|py| {
    let demo = py.import("demo")?;
    into_future(demo.call_method0("py_sleep")?)
})?;

let result = timeout(
    Duration::from_millis(20),
    receiver,
).await;

assert!(result.is_err());
```

No new Rust task is spawned here.

Timeout, select, or scope exit can drop this future.

## Python · scheduled Task

- `into_future` returns a Rust future, not a Python Task handle.
- Dropping it stops Rust waiting; the Python Task is not cancelled.
- Cancelling Python requires its Task handle and loop.

# Python → Rust → Python

## Python · application + callback

```python
async def decorate():
    await asyncio.sleep(0.1)
    return "hello"


async def main():
    text = await bridge.roundtrip(decorate)
    print(text)


asyncio.run(main())
```

## Rust · extension

```rust
#[pyfunction]
fn roundtrip<'py>(
    py: Python<'py>,
    callback: Py<PyAny>,
) -> PyResult<Bound<'py, PyAny>> {
    future_into_py(py, async move {
        let pending = Python::attach(|py| {
            into_future(callback.bind(py).call0()?)
        })?;
        let value = pending.await?;
        Python::attach(|py| {
            value.bind(py).extract::<String>()
        })
    })
}
```



# Questions?