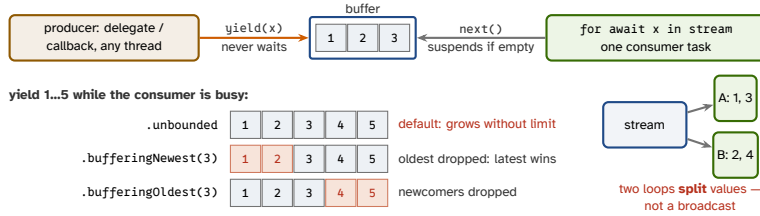
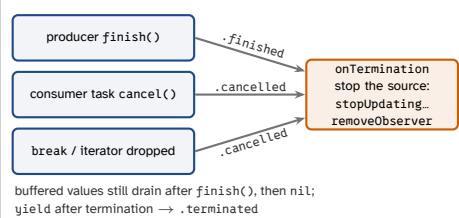


In one line: An AsyncSequence is a **pull**-based sequence whose iterator's next() is async: for try await suspends until the next element or nil (end). AsyncStream turns a **push** source (delegate, callback, notification) into one: the producer yields into a **continuation**, values wait in a **buffer** (default **unbounded, no backpressure**), one consumer task pulls them, and onTermination tears the source down.

Push in, pull out



Who ends it — onTermination runs once



How it works

- AsyncSequence.makeAsyncIterator() returns an iterator (AsyncIteratorProtocol): mutating func next() async throws -> Element?; nil ends it. for try await x in s = var it = s.makeAsyncIterator(); while let x = try await it.next(). Swift 6 (SE-0421) adds a Failure type and next(isolation:); non-throwing sequences drop the try.
- Lazy, pull-driven:** map, filter, compactMap, flatMap, prefix, dropFirst, first(where:), reduce run per element as the consumer asks.
- AsyncStream(Int.self, bufferingPolicy:) { cont in } — the build closure runs **once, immediately**. Better (Swift 5.9, SE-0388): let (stream, cont) = AsyncStream.makeStream(of: Int.self).
- The **continuation** is Sendable: yield from any thread. yield returns .enqueued(remaining:) / .dropped(x) / .terminated. finish() ends; AsyncThrowingStream's finish(throwing:) makes the loop throw.
- No backpressure:** yield never suspends the producer. Need it? A custom iterator, or AsyncChannel (await send waits for the consumer). Pull-style producer: AsyncStream(unfolding: { await read() } {}).
- Cancellation:** cancel the consuming task -> the stream's next() returns nil (loop just ends) -> onTermination(.cancelled). A hand-written iterator must check Task.isCancelled itself.

Example — delegate -> stream

```
final class LocationFeed: NSObject, CLLocationManagerDelegate {
    private let manager = CLLocationManager()
    private var cont: AsyncStream<CLLocation>.Continuation?
    func updates() -> AsyncStream<CLLocation> {
        let (stream, cont) = AsyncStream.makeStream(
            of: CLLocation.self, bufferingPolicy: .bufferingNewest(1))
        cont.onTermination = { [weak self] _ in // runs once
            self?.manager.stopUpdatingLocation() }
        self?.cont = cont; manager.delegate = self
        manager.startUpdatingLocation(); return stream }
    func locationManager(_ m: CLLocationManager,
        didUpdateLocations l: [CLLocation]) {
        l.forEach { cont?.yield($0) } // push
    }
}
let t = Task { for await loc in feed.updates() { show(loc) } }
t.cancel() // loop ends, GPS stops via onTermination
```

Bridging — which tool

Source	Bridge
one-shot callback delegate, repeated callback NotificationCenter Combine publisher	withCheckedThrowingContinuation — not a stream AsyncStream / AsyncThrowingStream + onTermination .notifications(named:) (iOS 15) .values (iOS 15) — demand 1 at a time: a subject's sends during the loop body are dropped ; add .buffer(size:prefetch:whenFull:) URLSession.bytes(from:), url.lines, FileHandle.bytes

swift-async-algorithms (Apple package)

merge(a, b) (interleave) · combineLatest(a, b) · zip(a, b) · chain(a, b) · .debounce(for:) (search box) · .chunks(ofCount:) / .chunked(by:) (batch uploads) · .removeDuplicates() · AsyncChannel (back-pressured) · AsyncTimerSequence.

Interview traps

- ! Default .unbounded + fast producer + slow consumer = memory grows; pick .bufferingNewest(n) for "latest state".
- ! Two for await loops on one stream: each value goes to **one** of them. Fan-out = one stream (continuation) per subscriber.
- ! No finish() and no cancel -> the consuming task waits forever.
- ! A stored Task you never cancel keeps the source running; onTermination capturing self strongly -> cycle.
- ! break ends *your* loop; a hand-rolled producer that ignores cancellation keeps working.

Remember

Yield pushes, next pulls, the buffer decides what's lost, onTermination cleans up.

Likely questions

1. AsyncStream vs AsyncThrowingStream? — only the latter can end with an error.
2. Backpressure? — none; buffer or drop. AsyncChannel suspends the sender.
3. vs Combine? — pull + task cancellation, no AnyCancellable; Combine = push with demand, multicast, more operators.
4. Stop the GPS when the view goes away? — cancel the task -> onTermination.