Make Your Own Stream Operators

Advanced stream processing in Rust

Willem Vanhulle

EuroRust 2025 • Paris, France

30 minutes + 10 minutes Q&A
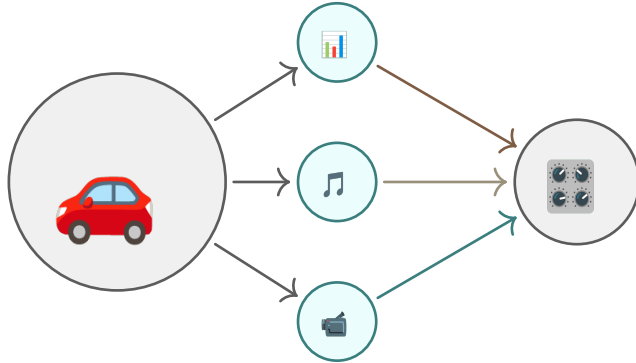
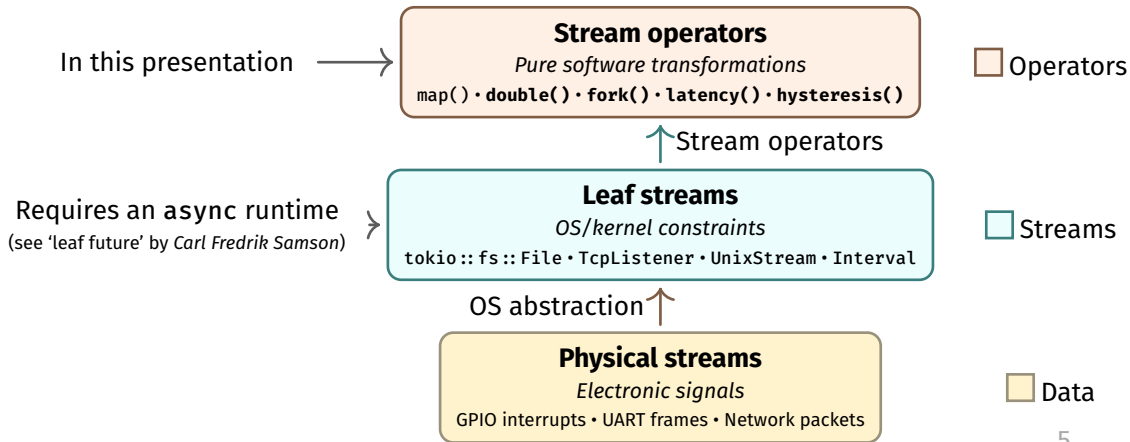Version with clickable links:
github.com/wvhulle/streams-eurorust-2025

Plan

Motivation

Processing data from moving vehicles

Kinds of streams

In this presentation →

Stream operators
Pure software transformations
map() · **double()** · **fork()** · **latency()** · **hysteresis()**

☐ Operators

↑ Stream operators

Requires an async runtime
(see 'leaf future' by *Carl Fredrik Samson*) →

Leaf streams
OS/kernel constraints
tokio :: fs :: File · TcpListener · UnixStream · Interval

☐ Streams

OS abstraction ↑

Physical streams
Electronic signals
GPIO interrupts · UART frames · Network packets

☐ Data

Naive stream processing

The challenge: Process TCP connections, filter messages, and collect 5 long ones

```
let mut filtered_messages = Vec::new(); let mut count = 0; let mut = 0;
let mut tcp_stream = tokio::net::TcpListener::bind("127.0.0.1:8080")
        .await?
        .incoming();
while let Some(connection) = tcp_stream.next().await {
    match connection {
        Ok(stream) ⇒ {
            if should_process(&stream) {
                // More nested logic needed ...
            }
        }
        Err(e) ⇒ {
            total_errors += 1;
            log_connection_error(e);
            if total_errors > 3 { break; }
        } } }
```

Complexity grows with each requirement

Inside the processing block, **even more nested logic:**

```
match process_stream(stream).await {
    Ok(msg) if msg.len() > 10 ⇒ {
        filtered_messages.push(msg);
        count += 1;
        if count ≥ 5 { break; }  // Break from outer loop!
    }
    Ok(_) ⇒ continue,  // Skip short messages
    Err(e) ⇒ {
        total_errors += 1;
        log_error(e);
        if total_errors > 3 { break; }  // Another outer break!
    }
}
```

Problems: hard to read, trace or test!

Functional `Stream` *usage preview*

Same logic, much cleaner with stream operators:

```
let filtered_messages: Vec<String> = tcp_stream
    .filter_map(|connection| ready(connection.ok()))
    .filter(|stream| ready(should_process(stream)))
    .then(|stream| process_stream(stream))
    .filter_map(|result| ready(result.ok()))
    .filter(|msg| ready(msg.len() > 10))
    .take(5)
    .collect()
    .await;
```

"Programs must be written **for people to read**, and only incidentally for machines to execute." — *Harold Abelson & Gerald Jay Sussman*

Rust's `Stream` trait

A lazy interface

Similar to `Future`, but yields multiple items over time (when queried / **pulled**):

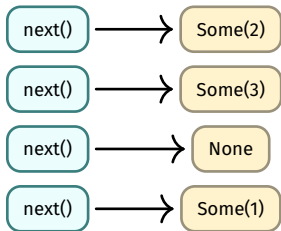```
trait Stream {
    type Item;

    fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
        → Poll<Option<Self::Item>>;
}
```

Returns `Poll` enum:

1. `Poll::Pending`: not ready (like `Future`)
2. `Poll::Ready(_)`:
   - `Ready(Some(item))`: new data is made available
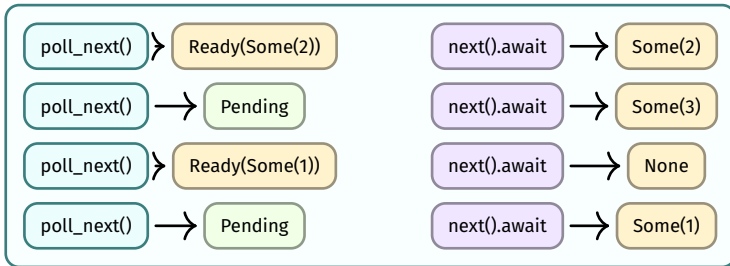   - `Ready(None)`: currently exhausted (not necessarily the end)

# *Moving from* `Iterator` *to* `Stream`
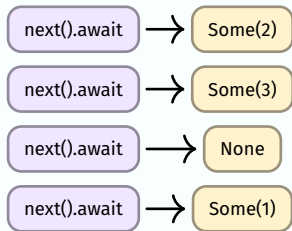
✓ Always returns immediately     ⚠️ May be Pending     ✓ Hides polling complexity

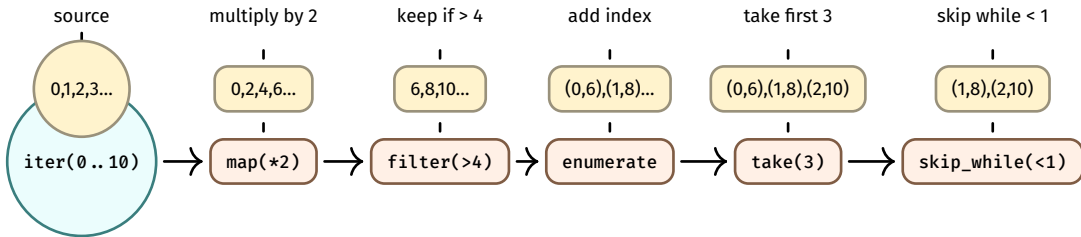| next() → Some(2) | poll_next() → Ready(Some(2)) | next().await → Some(2) |
| next() → Some(3) | poll_next() → Pending | next().await → Some(3) |
| next() → None | poll_next() → Ready(Some(1)) | next().await → None |
| next() → Some(1) | poll_next() → Pending | next().await → Some(1) |

**Iterator (sync)**      **Stream (low-level)**      **Stream (high-level)**

# Using the `Stream` API

## *Pipelines with* `futures::StreamExt`

All basic stream operators are in `futures::StreamExt`

| source | multiply by 2 | keep if > 4 | add index | take first 3 | skip while < 1 |
|--------|---------------|-------------|-----------|--------------|----------------|
| 0,1,2,3... | 0,2,4,6... | 6,8,10... | (0,6),(1,8)... | (0,6),(1,8),(2,10) | (1,8),(2,10) |
| `iter(0..10)` | `map(*2)` | `filter(>4)` | `enumerate` | `take(3)` | `skip_while(<1)` |

```
stream::iter(0..10)
  .map(|x| x * 2)
  .filter(|&x| ready(x > 4))
  .enumerate().take(3).skip_while(|&(i, _)| i < 1)
```

13

The handy `std::future::ready` *function*

The `futures::StreamExt::filter` expects an **async closure** (or closure returning `Future`):

Option 1: Async block (not `Unpin`!)

```
stream.filter(|&x| async move {
  x % 2 == 0
})
```

Option 2: Async closure (not `Unpin`!)

```
stream.filter(async |&x| x % 2 == 0)
```
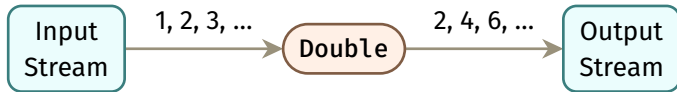
Option 3 (recommended): Wrap sync output with `std::future::ready()`

```
stream.filter(|&x| ready(x % 2 == 0))
```

- `ready(value)` creates a `Future` that immediately resolves to `value`.

- `ready(value)` is `Unpin` and **keeps pipelines `Unpin`**: *easier to work with*, see later.

14

Example 1: One-to-One Operator

Doubling stream operator

Input Stream → 1, 2, 3, … → Double → 2, 4, 6, … → Output Stream

Wrapping the original stream

All stream operators start by:

- **wrapping input stream by value**
- and being **generic over stream type**

(No trait bounds yet):

```
struct Double<InSt> { in_stream: InSt, }
```

And implementing the Stream trait for it (**with trait bounds**):

```
impl<InSt> Stream for Double<InSt> where InSt: Stream<Item = i32> {
  type Item = InSt::Item;

  fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>) →  Poll<Option<Self::Item>> {
            …
  }
}
```
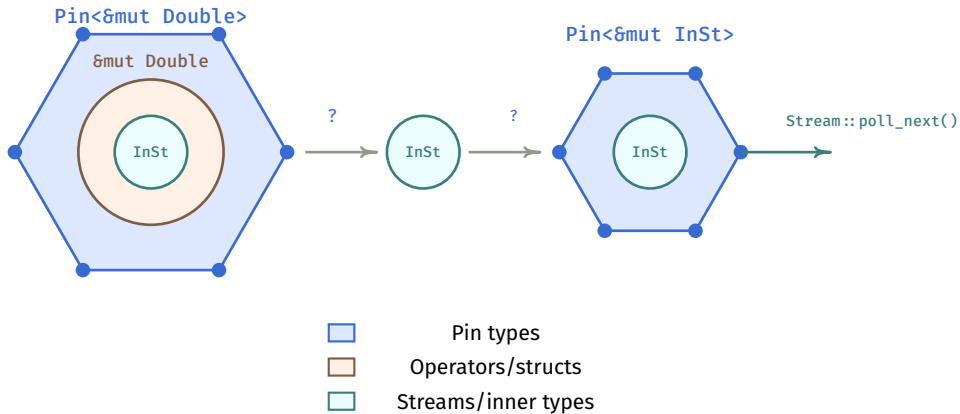
Naive implementation of `poll_next`

Focus on the implementation of the `poll_next` method

(Remember that `Self = Double<InSt>` with field `in_stream: InSt`):
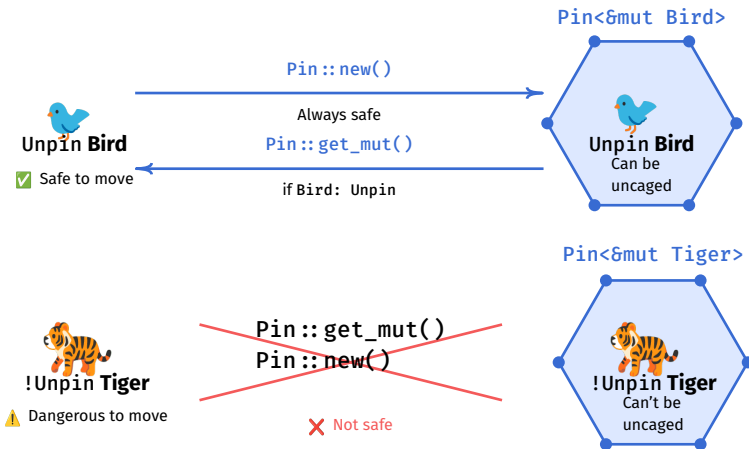
```
fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
    → Poll<Option<Self::Item>> {
        // Access field `self.in_stream`
        //   `self.in_stream` is of type `InSt`
        Pin::new(&mut self.in_stream) // Not possible!
            .poll_next(cx) // Unreachable ...
            .map(|x| x * 2)
}
```
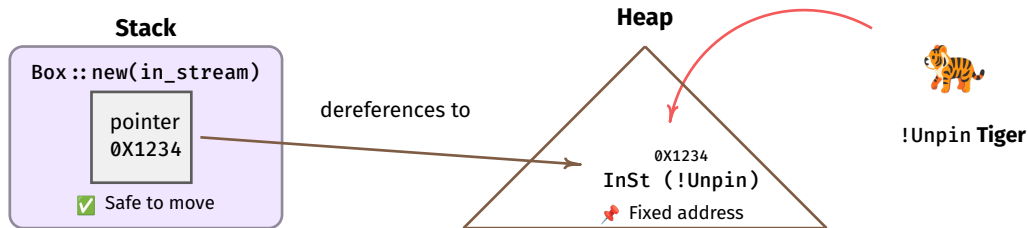
`Pin<&mut Self>` **blocks access to `self.in_stream`!**

How to access `self.in_stream`?

Pin<&mut Double>

&mut Double

InSt

?

InSt

?

Pin<&mut InSt>

InSt

Stream::poll_next()

- ☐ Pin types
- ☐ Operators/structs
- ☐ Streams/inner types

19

!Unpin *defends against unsafe moves*

Pin<&mut Bird>

Pin::new()
Always safe

Unpin **Bird**

✅ Safe to move

Pin::get_mut()
if Bird: Unpin

Unpin **Bird**
Can be
uncaged

Pin<&mut Tiger>

Pin::get_mut()
Pin::new()

!Unpin **Tiger**

⚠️ Dangerous to move

❌ Not safe

!Unpin **Tiger**
Can't be
uncaged

20

Put your `!Unpin` _type on the heap_

Stack

Box :: new(in_stream)

pointer
0X1234

✅ Safe to move

dereferences to

Heap

0X1234
InSt (!Unpin)
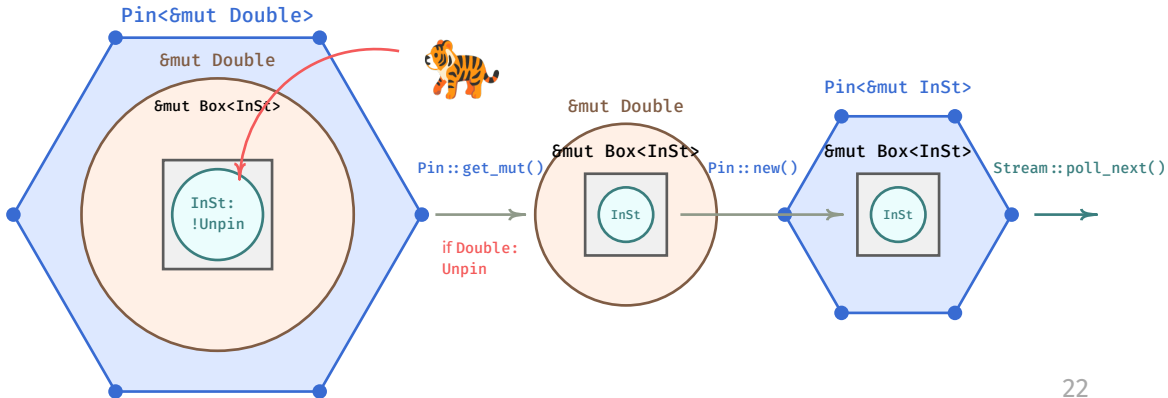📌 Fixed address

🐯

`!Unpin` **Tiger**

1. The output of `Box :: new(tiger)` is just a pointer
   Moving pointers is safe, so **Box: Unpin**
2. Box behaves like what it contains: **Box<X>: Deref<Target = X>**

Result:

```
struct Double {in_stream: Box<InSt>}: Unpin
```

Putting it all together visually

Mapping from `Pin<&mut Double>` to `&mut InSt` is called **projection**

Complete `Stream` *trait implementation*

We can call `get_mut()` to get `&mut Double<InSt>` safely:

```
impl<InSt> Stream for Double<InSt>
where InSt: Stream<Item = i32>
{
    fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
        → Poll<Option<Self::Item>>
    {
        // We can project because `Self: Unpin`
        let this: &mut Double<InSt> = self.get_mut();
        // `this` is a conventional name for projection
        Pin::new(&mut this.in_stream)
            .poll_next(cx)
            .map(|r| r.map(|x| x * 2))
    }
}
```

Distributing your operator

Define a constructor and turn it into a method of an **extension trait**:

```
trait DoubleStream: Stream {
    fn double(self) → Double<Self>
    where Self: Sized + Stream<Item = i32>,
    { Double::new(self) }
}
// A blanket implementation should be provided by you!
impl<S> DoubleStream for S where S: Stream<Item = i32> {}
```

Now, users **don't need to know how** Double is implemented, just

1. import your extension trait: DoubleStream
2. call .double() on any compatible stream

# Example 2: One-to-N Operator

Complexity 1-N operators

Challenges for `Stream` operators are combined from:

Inherent `Future` challenges:
- Clean up orphaned wakers
- Cleanup when tasks abort
- Task coordination complexity

Inherent `Iterator` challenges:
- Ordering guarantees across consumers
- Backpressure with slow consumers
- Sharing mutable state safely
- Avoiding duplicate items

Thousands of clones...

C1

C2

C3

C4

C5
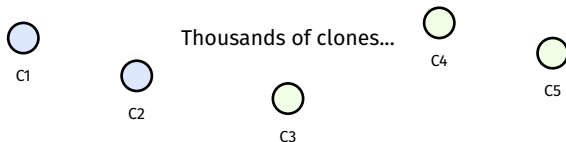
Sharing latency between tasks

Latency may need to processed by different async tasks:

```
let tcp_stream = TcpStream::connect("127.0.0.1:8080").await?;
let latency = tcp_stream.latency(); // Stream<Item = Duration>
spawn(async move { display_ui(latency).await; });
spawn(async move { engage_breaks(latency).await; }); // Error!
```

Error: `latency` is moved into the first task, so the second task can't access it.

Cloning streams with an operator

Solution: Create a **_stream operator_** `fork()` makes the input stream `Clone`.

```rust
let tcp_stream = TcpStream::connect("127.0.0.1:8080").await?;

// Fork makes the input stream cloneable
let ui_latency = tcp_stream.latency().fork();

let breaks_latency_clone = ui_latency.clone();
// Warning: `Clone` needs to be implemented!

spawn(async move { display_ui(ui_latency).await; });
spawn(async move { engage_breaks(breaks_latency_clone).await; });
```
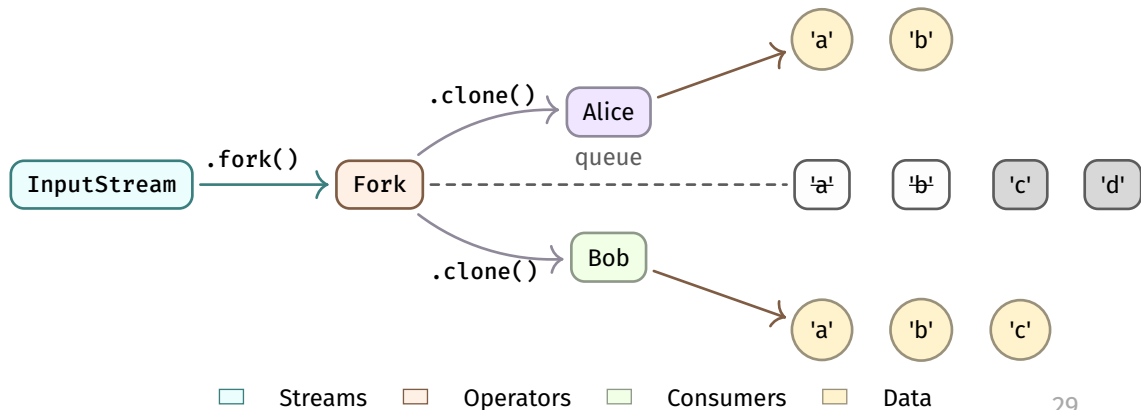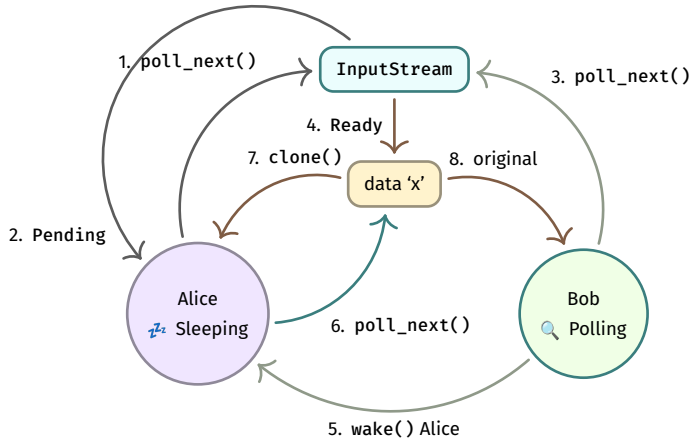
Requirement: `Stream<Item: Clone>`, so we can clone the items (`Duration` is `Clone`)

Rough architecture of `clone-stream`

Streams ☐ Operators ☐ Consumers ☐ Data ☐
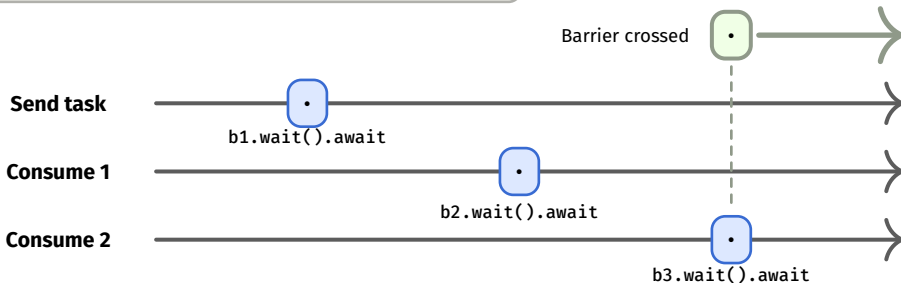
Polling and waking flow

`Barrier`*s for task synchronization*

For performance reasons, you may want to **ignore unpolled consumers** (init required) in 1-to-N stream operators.

Synchronisation after the "init" phase is done with a single `Barrier` ot type $N + 1$.

```rust
let b1 = Arc::new(Barrier::new(3)); // For input task
let b2 = b1.clone(); // First output
let b3 = b1.clone(); // For second output
```


Barrier crossed

Send task

`b1.wait().await`

Consume 1

`b2.wait().await`

Consume 2

`b3.wait().await`

31

Including `Barrier`_s in your unit tests_

When you build your own:

1. Pick a `Barrier` crate (tokio / async-lock).

2. Define synchronization points with `Barrier`:

```
let b1 = Arc::new(Barrier::new(3));
let b2 = b1.clone(); // Second output
let b3 = b1.clone(); // For input
```

3. Apply your custom operator

```
let out_stream1 =
create_test_stream(in_stream)
    .your_custom_operator();
let out_stream2 = out_stream1.clone();
```
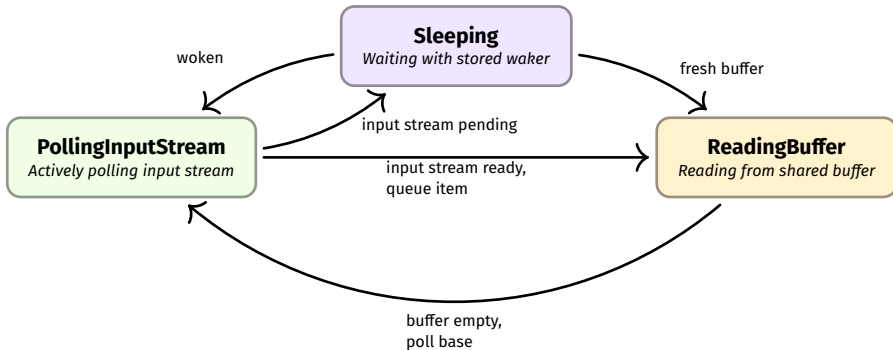
4. Send your inputs and outputs to separate tasks

5. Do not use `sleep` and await all tasks.

```
try_join_all([
    spawn(async move {
        setup_task().await;
        b1.wait().await;
        out_stream1.collect().await;
    }),
    spawn(async move {
        setup_task().await;
        b2.wait().await;
        out_stream2.collect().await;
    }),
    spawn(async move {
        b3.wait().await;
        send_input(in_stream).await;
    })
]).await.unwrap();
```
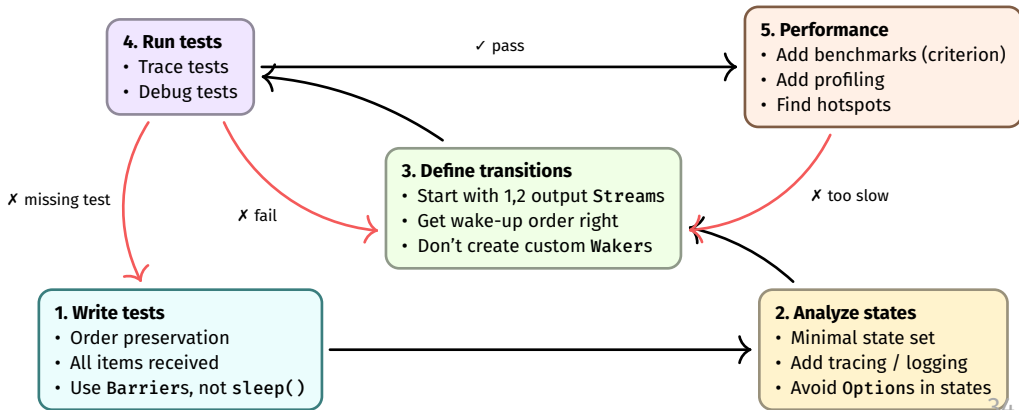
Simplified state machine of `clone-stream`

Enforcing simplicity, **correctness and performance**:

Each clone maintains its own <u>state</u>:

Steps for creating robust stream operators

4. Run tests
- Trace tests
- Debug tests

✓ pass

5. Performance
- Add benchmarks (criterion)
- Add profiling
- Find hotspots

3. Define transitions
- Start with 1,2 output `Streams`
- Get wake-up order right
- Don't create custom `Wakers`

✗ missing test

✗ fail

✗ too slow

1. Write tests
- Order preservation
- All items received
- Use `Barriers`, not `sleep()`

2. Analyze states
- Minimal state set
- Add tracing / logging
- Avoid `Options` in states

General principles

Principles

Don't overuse streams:
- Keep pipelines short
- Only _physical async data flow_

Meaningful objective targets:
- Simple, clear unit tests
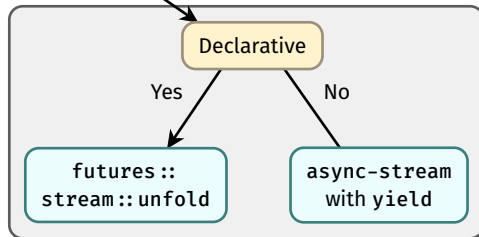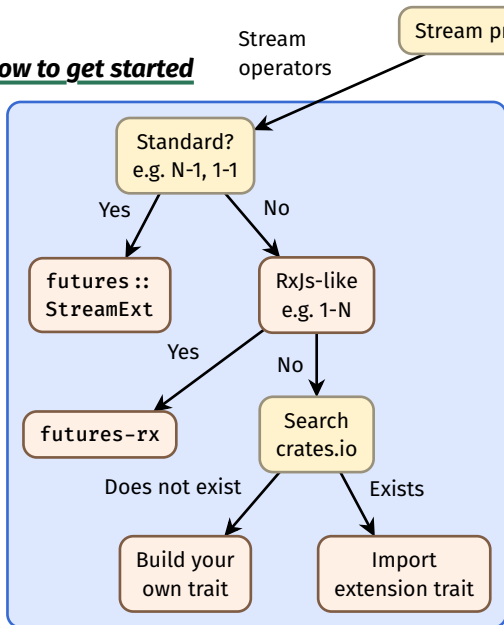- Relevant benchmarks

Separation of concerns:
- Modular functions
- Descriptive names
- Split long functions

Simple state machines:
1. Fewer `Options`
2. Fewer states

"Perfection is achieved, not when there is nothing more to add, but when there is **nothing left to take away**." — _Antoine de Saint-Exupéry_

Stream processing style

How to get started

Stream operators

Traditional control flow

```
Standard?
e.g. N-1, 1-1
```

Yes → `futures::StreamExt`

No → `RxJs-like e.g. 1-N`

Yes → `futures-rx`

No → `Search crates.io`

Does not exist → `Build your own trait`

Exists → `Import extension trait`

Declarative

Yes → `futures::stream::unfold`

No → `async-stream with yield`

Always requires `Box` to make `!Unpin` output `Unpin`

37

Any questions?

Thank you!

Want to learn Rust in-depth?

Join my 7-week course ***"Creating Safe Systems in Rust"*** in Ghent starting November 2025.

Register at willemvanhulle.tech (link at bottom of page)

- Contact me: willemvanhulle@protonmail.com
- These slides: github.com/wvhulle/streams-eurorust-2025

Bonus slides

`Stream`*s in Rust are not new*

2019
async/await stabilized in Rust
Stable async streams in std
RFC 2394, Rust 1.39.0

2009
Microsoft Reactive Extensions
ReactiveX brings streams to mainstream
Erik Meijer, Microsoft

1997
Functional Reactive Programming
Conal Elliott & Paul Hudak (Haskell)
ICFP '97, pp. 263-273

1978
Communicating Sequential Processes
Tony Hoare formalizes concurrent dataflow
CACM 21(8):666-677

1973
Unix Pipes
Douglas McIlroy creates `` `|` `` operator
Bell Labs, Unix v3-v4

1960s
Dataflow Programming
Hardware-level stream processing
Early dataflow architectures

The meaning of Ready(None)

Regular Stream

"No items **right now**"

(*Stream might yield more later*)

Fused Stream

"No items **ever again**"

(*Stream is permanently done*)

'Fusing' streams and futures

	Future	Stream	Meaning
Regular	——┼—→	+++→	May continue
Fused	`FusedFuture`	`FusedStream`	`is_terminated()` method
Fused	——┼—┃	+++┤┃	Done permanently
Fused value	`Pending`	`Ready(None)`	Final value

Flatten a **finite collection** of Streams

A finite collection of `Streams` = `IntoIterator<Item: Stream>`

```rust
let streams = vec![
    stream::iter(1..=3),
    stream::iter(4..=6),
    stream::iter(7..=9),
];

let merged = stream::select_all(streams);
```

1. Creates a `FuturesUnordered` of the streams
2. Polls all streams concurrently
3. Yields items as they arrive

Flattening an infinite stream

Beware!: `flatten()` on a stream of infinite streams will never complete!

```
let infinite_streams = stream::unfold(0, |id| async move {
    Some((stream::iter(id..), id + 1))
});
let flat = infinite_streams.flatten();
```

Instead, **buffer streams** concurrently with `flatten_unordered()`.

```
let requests = stream::unfold(0, |id| async move {
    Some((fetch_stream(format!("/api/data/{}", id)), id + 1))
});
let flat = requests.flatten_unordered(Some(10));
```

More `Stream` *features to explore*

Many more advanced topics await:

- **Boolean operations**: any, all
- **Async operations**: then
- **Sinks**: The write-side counterpart to `Stream`s

```
Stream ——————— 'a' — 'b' — 'c' ——.forward()——→ Sink
```

Read side   Write side
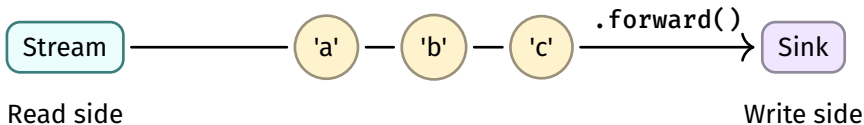
The `Stream` *trait: a lazy query interface*

The `Stream` trait is NOT the stream itself - it's just a lazy frontend to query data.
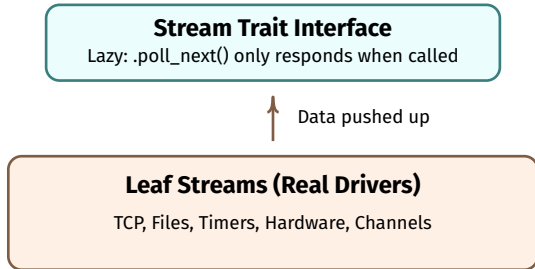
What `Stream` trait does:
- Provides uniform `.poll_next()` interface
- Lazy: only responds when asked
- Doesn't drive or produce data itself
- Just queries whatever backend exists

What actually drives streams:
- TCP connections receiving packets
- File I/O completing reads
- Timers firing
- Hardware signals
- Channel senders pushing data

The 'real' stream drivers

Stream Trait Interface
Lazy: .poll_next() only responds when called

↑ Data pushed up

Leaf Streams (Real Drivers)
TCP, Files, Timers, Hardware, Channels

Stream trait just provides a **uniform way to query** - it doesn't create or drive data flow.

Possible inconsistency

```
trait Stream {
    type Item;

    fn poll_next(self: Pin<&mut Self>, cx: &mut Context)
        → Poll<Option<Self::Item>>
}
```

What about Rust rule `self` needs to be `Deref<Target=Self>`?

`Pin<&mut Self>` only implements `Deref<Target=Self>` for `Self: Unpin`.

Problem? No, `Pin` is an exception in the compiler.

Why does Rust need special treatment?

- Stream operators must wrap and own their input by value
- Combining `Future` (waker cleanup, coordination) and `Iterator` (ordering, backpressure) complexity
- Sharing mutable state safely across async boundaries requires careful design
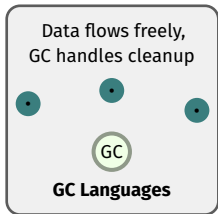
| GC Languages | vs | Rust |
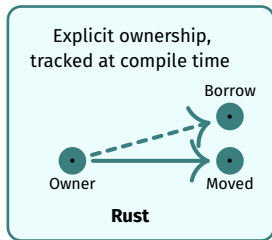
Reactive patterns from GC languages require rethinking in Rust's ownership model

<ins>The end</ins>