

Make Your Own Stream Operators

Transforming asynchronous data streams in Rust

Willem Vanhulle

EuroRust 2025 • Paris, France

30 minutes + 10 minutes Q&A

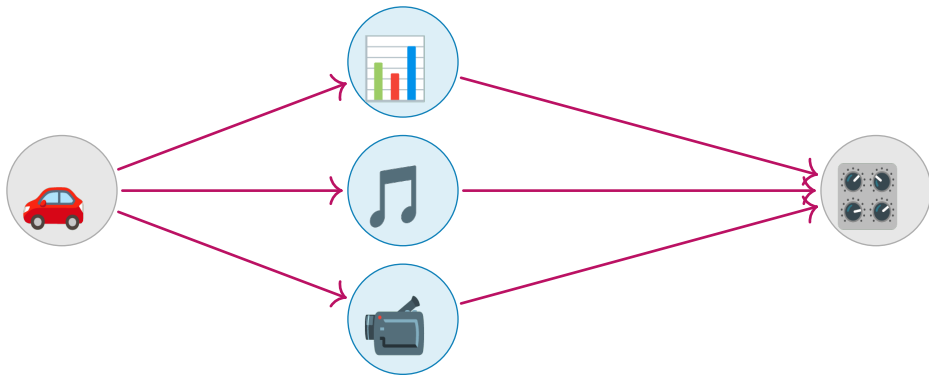
Version with clickable links:
github.com/wvhulle/streams-eurorust-2025

Plan

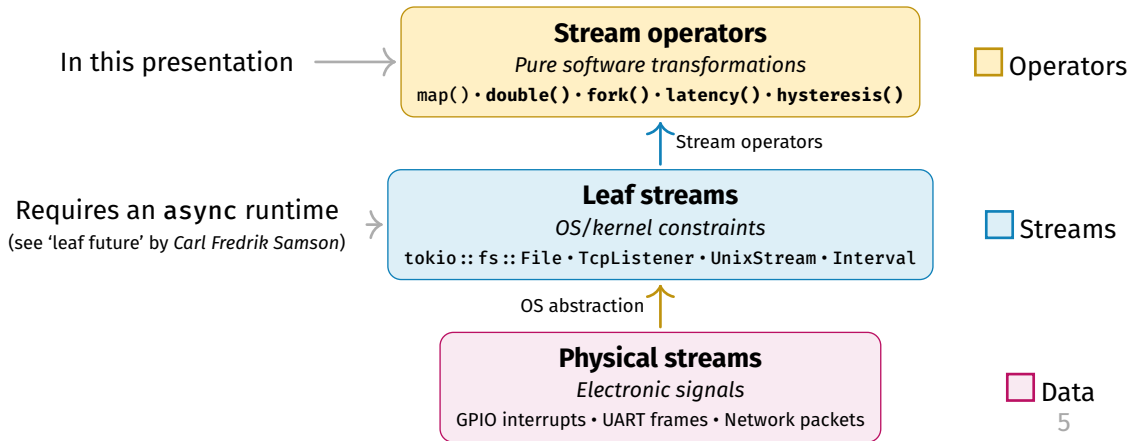
Motivation	3
Rust's <code>Stream</code> trait	8
Example 1: $1 \rightarrow 1$ Operator	12
Example 2: $1 \rightarrow N$ Operator	28
General principles	36
Bonus slides	39

Motivation

Processing data from moving vehicles



Kinds of streams



Process TCP connections and collect 5 long messages

```
let mut results = Vec::new(); let mut count = 0;

while let Some(connection) = tcp_stream.next().await {
    match connection {
        Ok(stream) if should_process(&stream) => {
            match process_stream(stream).await {
                Ok(msg) if msg.len() > 10 => {
                    results.push(msg);
                    count += 1;
                    if count ≥ 5 { break; }
                }
                Ok(_) => continue,
                Err(_) => continue,
            }
        }
        Ok(_) => continue,
        Err(_) => continue,
    }
}
```

Problems:

- Deeply nested
- Hard to read
- Cannot test pieces independently

Stream *operators: declarative & composable*

Same logic with stream operators:

```
let results: Vec<String> = tcp_stream
    .filter_map(|conn| ready(conn.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;
```

Benefits:

- Each operation is isolated
- Testable
- Reusable

“Programs must be written **for people to read**”

— Abelson & Sussman

Rust's Stream trait

The Stream *trait*: *async iterator*

Like Future, but yields **multiple items** over time when polled:

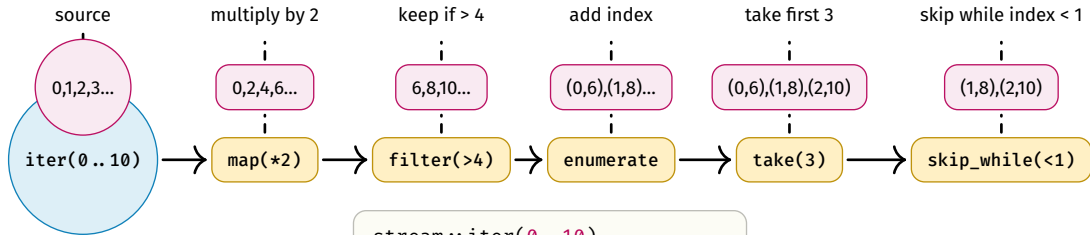
```
trait Stream {  
    type Item;  
  
    fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)  
        → Poll<Option<Self::Item>>;  
}
```

The Poll<Option<Item>> return type:

- Poll::Pending - not ready yet, try again later
- Poll::Ready(Some(item)) - here's the next item
- Poll::Ready(None) - stream is exhausted (no more items **right now**)

Pipelines with futures :: StreamExt

All basic stream operators are in futures :: StreamExt



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

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))
```

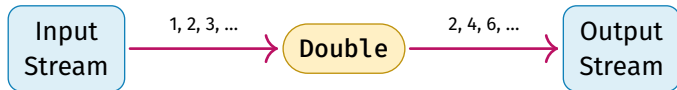
- `ready(value)` creates a `Future` that immediately resolves to `value`.
- `ready(value)` is `Unpin`

ready keeps pipelines `Unpin`: easier to work with

Example 1: $1 \rightarrow 1$ Operator

Doubling stream operator

Very simple Stream operator that **doubles every item** in an input stream:



Input stream **needs to yield integers**.

Building a stream operator: structure

Step 1: Define a struct that wraps the input stream

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

- Generic over stream type (works with any backend)
- Stores input stream by value

Step 2: Implement Stream trait with bounds

```
impl<InSt> Stream for Double<InSt>  
where  
    InSt: Stream<Item = i32>  
{  
    type Item = i32;  
  
    fn poll_next(self: Pin<&mut Self>,  
cx: &mut Context<'_>)  
        → Poll<Option<Self::Item>> {  
        // ... implementation goes here  
    }  
}
```

Naive implementation of poll_next

Focus on the implementation of the poll_next method

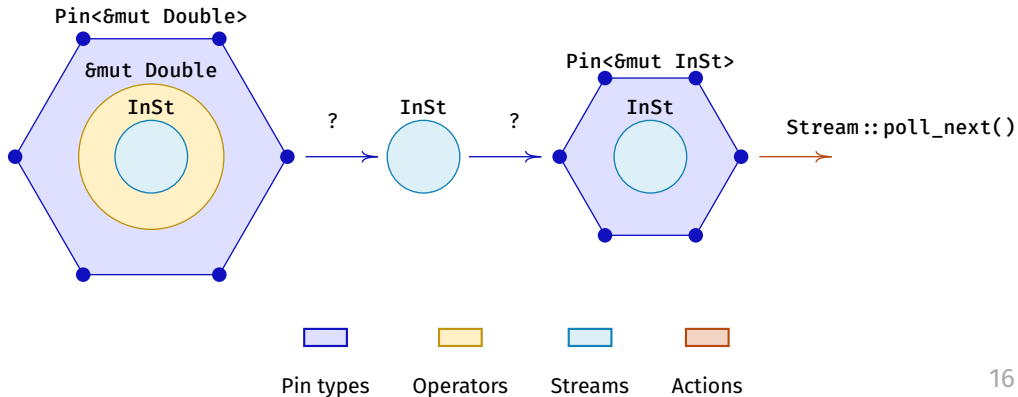
(Remember that Self = Double<InSt> with field in_stream: InSt):

```
fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)  
    → Poll<Option<Self::Item>> {  
    // Cannot access self.in_stream!  
    Pin::new(&mut self.in_stream) // Not possible!  
        .poll_next(cx)  
        .map(|x| x * 2)  
}
```

Pin<&mut Self> **blocks access to self.in_stream** (when Self: !Unpin)!

The projection problem: accessing pinned fields

We have `Pin<&mut Double>`, but need `Pin<&mut InSt>` to call `poll_next()`. How?



The naive solution fails

Can we use `Pin::get_mut()` to unwrap and re-wrap?

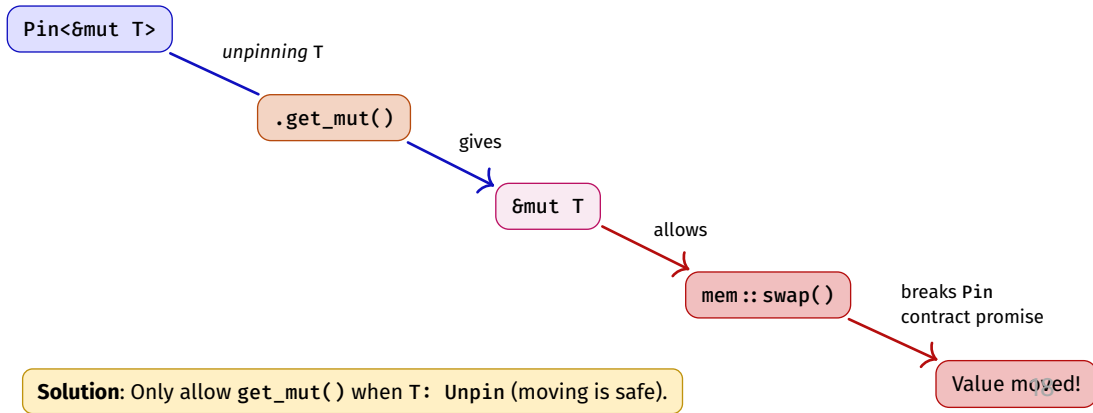
```
impl<InSt> Stream for Double<InSt> where InSt: Stream<Item = i32> {  
    type Item = InSt::Item;  
  
    fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)  
        → Poll<Option<Self::Item>> {  
        let this = self.get_mut(); // Error!  
        let pinned_in = Pin::new(&mut this.in_stream);  
        pinned_in.poll_next(cx).map(|p| p.map(|x| x * 2))  
    }  
}
```

Problem: `Pin::get_mut()` requires `Double<InSt>: Unpin`

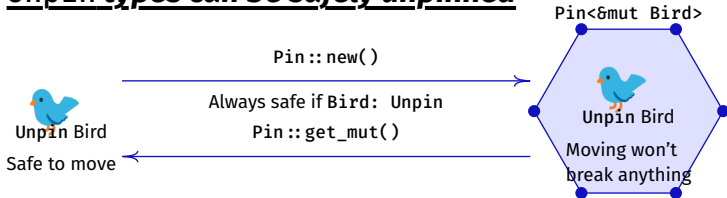
But `Double<InSt> is !Unpin` when `InSt: !Unpin`!

Why does `Pin::get_mut()` require `Unpin`?

`Pin<P>` makes a promise: **the pointee will never move again.**



Unpin types can be safely unpinned



If `T: Unpin`, then `Pin::get_mut()` is safe because moving `T` doesn't cause UB.

Examples of Unpin types:

- `i32`, `String`, `Vec<T>` - all primitive and standard types
- `Box<T>` - pointers are safe to move
- `&T`, `&mut T` - references are safe to move

Why safe?

These types don't have self-referential pointers. Moving them in memory doesn't invalidate any internal references.

Almost all types are `Unpin` by default!

!Unpin types cannot be safely unpinned

Pin<&mut Tiger>



!Unpin Tiger

Dangerous to move

~~Pin::get_mut()
gives &mut T~~

Would break
pin promise!



Examples of !Unpin types:

- PhantomPinned - explicitly opts out of Unpin
- Most Future types (self-ref. state machines)
- Types with self-referential pointers
- Double<InSt> where InSt: !Unpin

Why unsafe?

These types may contain pointers to their own fields. Moving them in memory would invalidate those internal pointers, causing use-after-free.

!Unpin is rare and usually intentional for async/self-referential types.

One workaround: add the Unpin bound

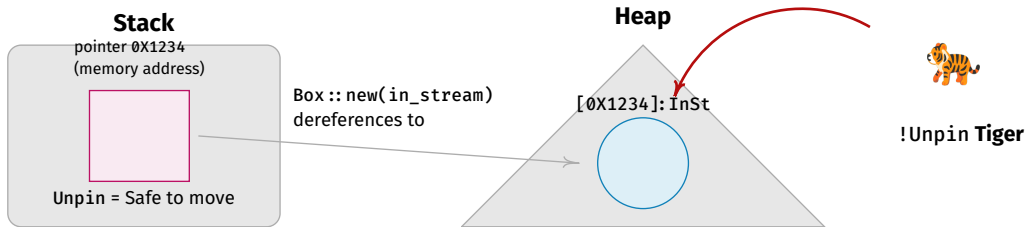
The compiler error suggests adding `InSt: Unpin`:

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

We don't want to impose `InSt: Unpin` on users of `Double`!

How to support `InSt: !Unpin` streams? ...

Turning !Unpin into Unpin with boxing



Nice to have:

1. `Box :: new(tiger)` produces just a pointer on the stack
 - Moving pointers is always safe
 - Therefore: **`Box<Tiger>: Unpin`**
2. Box dereferences to its contents
 - **`Box<X>: Deref<Target = X>`**

Problem: Need `Pin<&mut InSt>`, but `Box<InSt>` requires `InSt: Unpin` to create it

Solution: Use `Pin<Box<InSt>>` to project from `Pin<&mut Double>` to `Pin<&mut InSt>` via `Pin :: as_mut()`

Applying the solution: Pin<Box<InSt>>

Change the struct definition to store Pin<Box<InSt>>:

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

Why this works:

- Box<InSt> is always Unpin (pointers are safe to move)
- Pin<Box<InSt>> can hold !Unpin streams safely on the heap

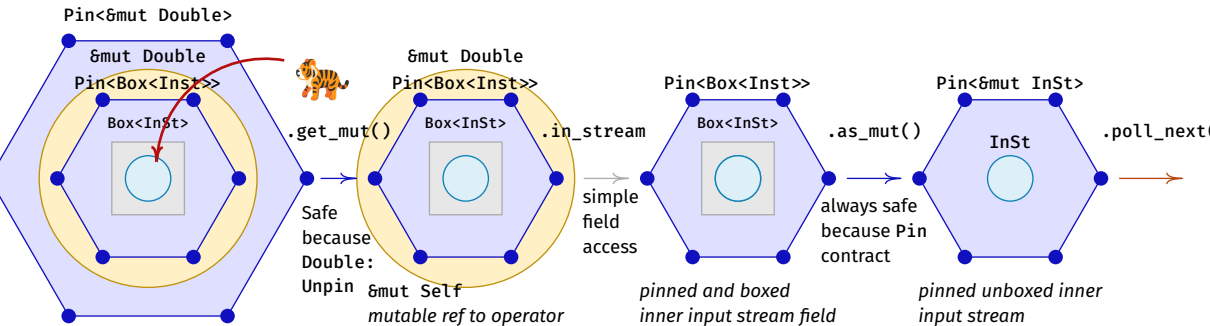
Projection in poll_next:

```
fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)  
    → Poll<Option<Self::Item>> {  
    let this = self.get_mut(); // Safe: Double is Unpin now  
    this.in_stream.as_mut()    // Project to Pin<&mut InSt>  
        .poll_next(cx)  
        .map(|opt| opt.map(|x| x * 2))  
}
```

This works **without requiring InSt: Unpin!**

Projecting visually

From `Pin<&mut Double>` to `Pin<&mut InSt>` in a few **safe steps**:



Complete boxed Stream *implementation*

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

```
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.in_stream.as_mut()
            .poll_next(cx)
            .map(|r| r.map(|x| x * 2))
    }
}
```

Two ways to handle !Unpin fields

Approach 1: Use Box<_>

```
struct Double<InSt> {  
    in_stream: Pin<Box<InSt>>  
}  
  
impl<InSt> Stream for Double<InSt>  
    where InSt: Stream
```

✓ Works with any InSt, also !Unpin

... or, use pin-project crate

Approach 2: Require Unpin

```
struct Double<InSt> {  
    in_stream: InSt  
}  
  
impl<InSt> Stream for Double<InSt>  
    where InSt: Stream + Unpin
```

✗ Imposes Unpin constraint on users

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: $1 \rightarrow N$ Operator

Complexity 1 \rightarrow N operators

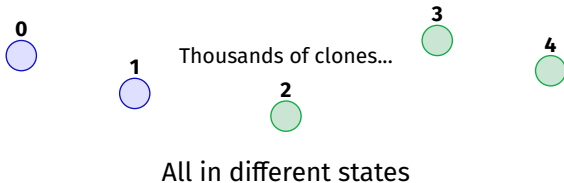
Challenges for Stream operators are combined from:

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



Sharing latency between tasks

Latency may need to be 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.

We need a way to clone the latency stream!

Cloning streams with an operator

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

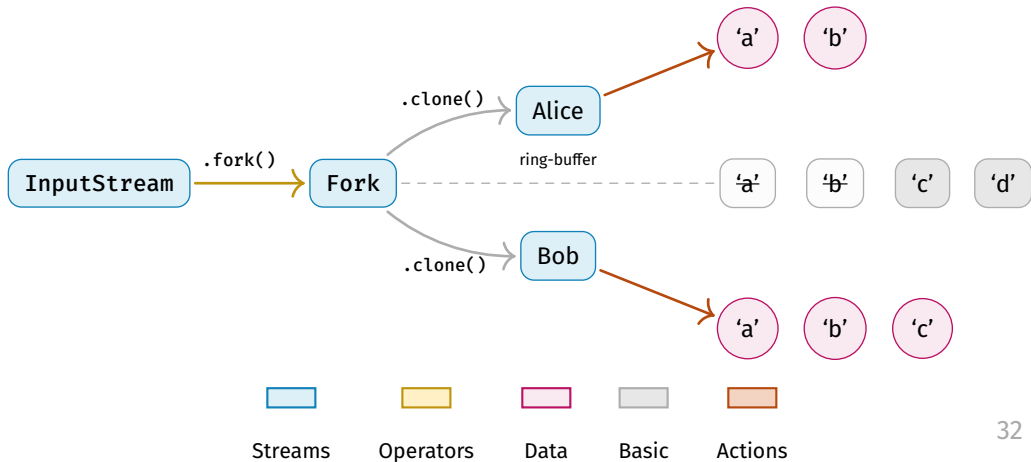
```
let ui_latency = tcp_stream.latency().fork();

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

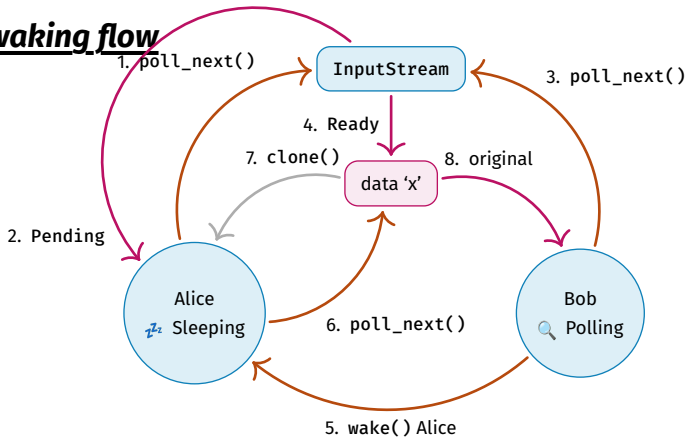
spawn(async move { display_ui(ui_latency).await; });
spawn(async move { engage_breaks(breaks_latency_clone).await; });
```

Requirement: `Stream<Item: Clone>`, so we can clone the items (`Duration` is `Clone`)

Rough architecture of clone-stream



Polling and waking flow

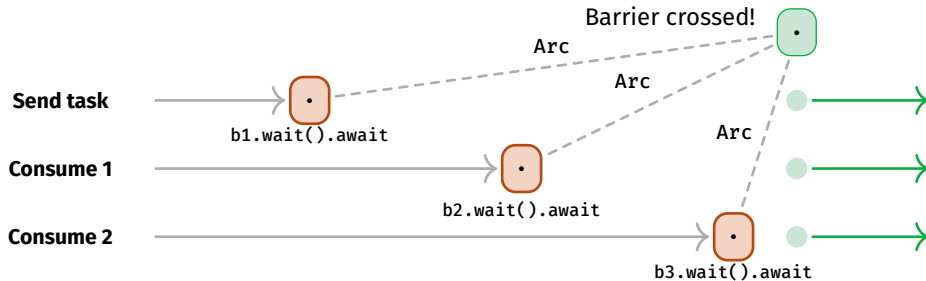


Barriers for task synchronization checkpoints

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

Each consumer needs to signal when it is ready to receive items.

Synchronisation after the “init” phase is done with a single Barrier of type $N + 1$.



Multiple **synchronization points** are possible with multiple Barriers.

Including Barriers 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();
```

General principles

Rules of thumb

Don't overuse streams:

- Keep pipelines short
- Only *physical async data flow*

Meaningful objective targets:

- Simple, clear unit tests
- Relevant benchmarks (criterion)

Separation of concerns:

- Modular functions
- Descriptive names
- Split long functions

Simple state machines:

1. Fewer Options
2. More states

Any questions?

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

Want to learn more in-depth?

Join my 7-week course ***“Creating Safe Systems in Rust”***

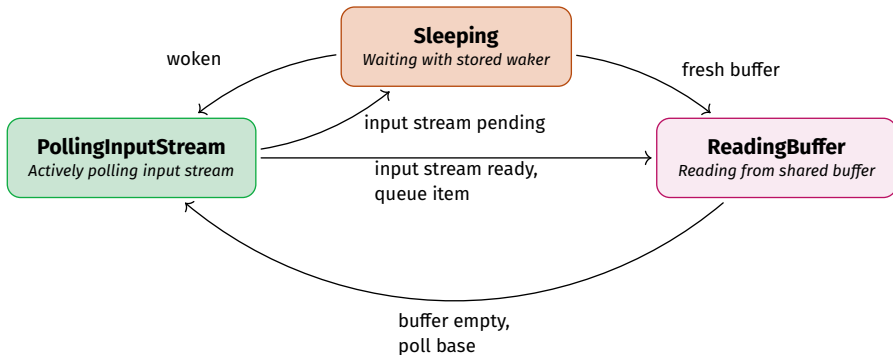
- Location: Ghent (Belgium)
- Date: starting 4th of November 2025.

Register at pretix.eu/devlab/rust-course/

Bonus slides

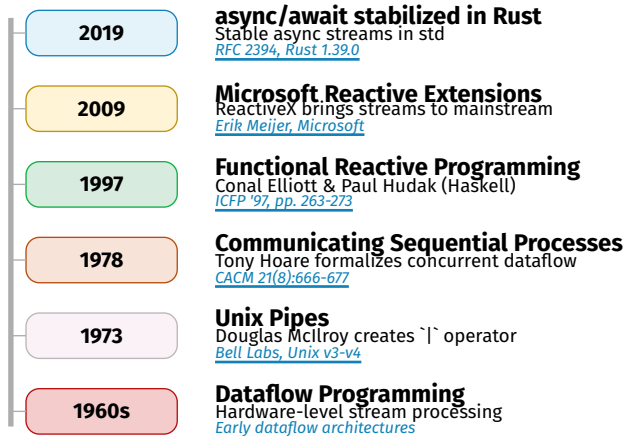
Simplified state machine of clone-stream

Enforcing simplicity, **correctness** and **performance**:



Each clone maintains its own state

Streams in Rust are not new



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	<code>is_terminated()</code> 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>`

```
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();
```

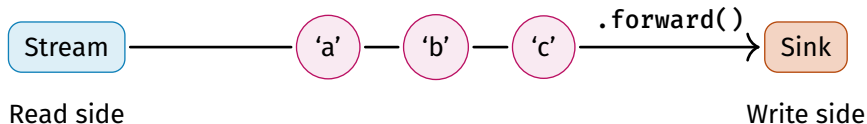
Instead, **buffer streams** concurrently with `flatten_unordered()`.

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

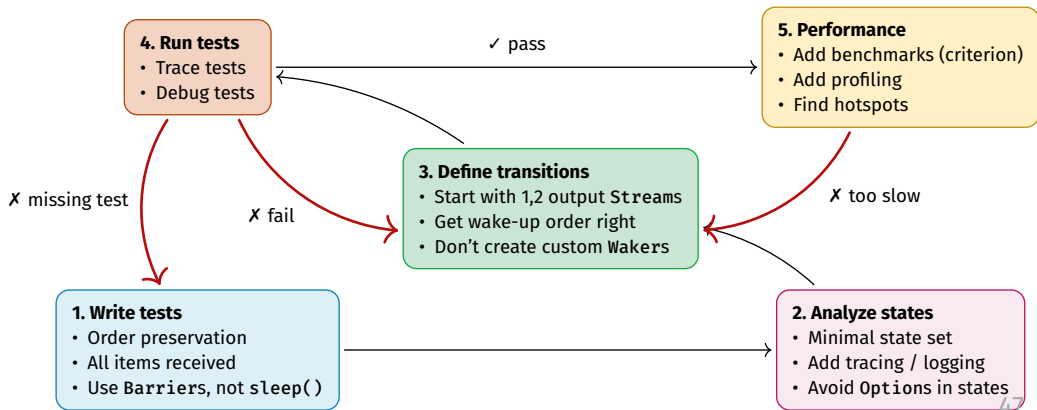
More Stream *features to explore*

Many more advanced topics await:

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



Steps for creating robust stream operators



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:

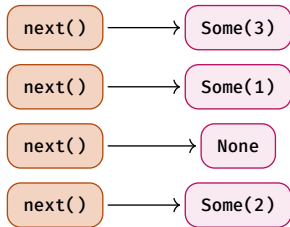
- Provides uniform `.poll_next()` interface
- Lazy: only responds when asked
- Doesn't drive or produce data itself
- Just queries whatever backend exists

What actually drives streams:

- TCP connections receiving packets
- File I/O completing reads
- Timers firing
- Hardware signals
- Channel senders pushing data

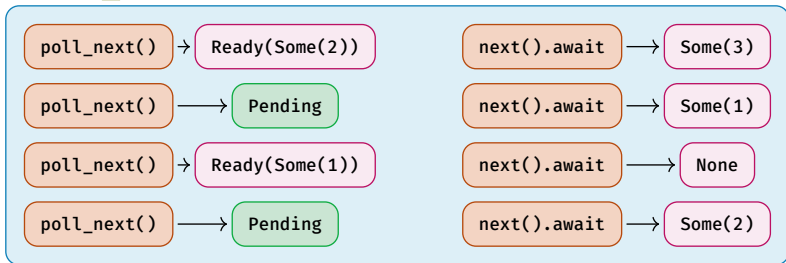
Moving from Iterator to Stream

✓ Always returns immediately



Iterator (sync)

⚠ May be Pending

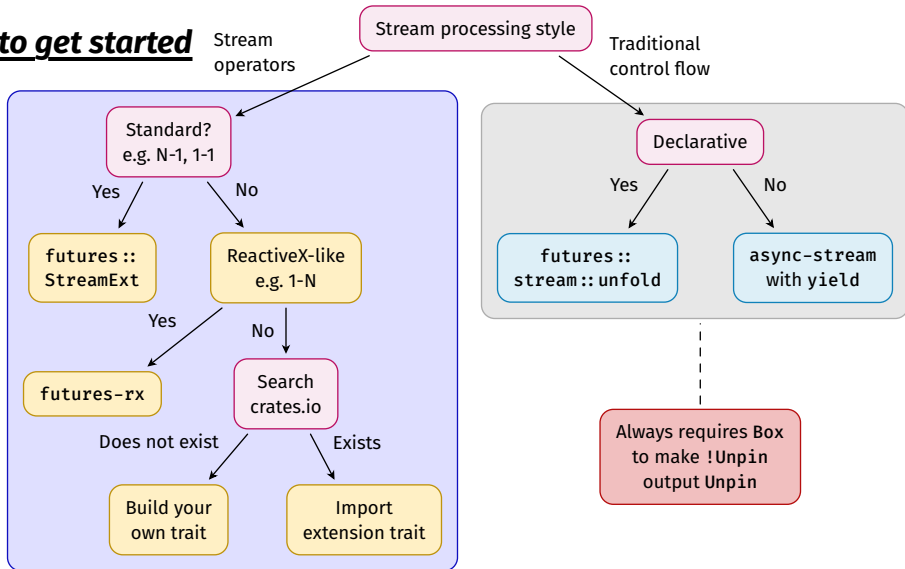


Stream (low-level)

Stream (high-level)



How to get started



Approach 3: Projection with pin-project

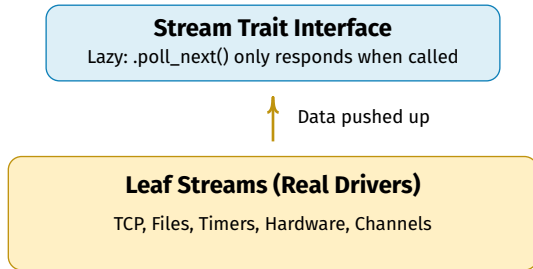
Projects like Tokio use the pin-project crate:

```
#[pin_project]
struct Double<InSt> {
    #[pin]
    in_stream: InSt,
}

impl<InSt: Stream> Stream for Double<InSt> {
    fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
        → Poll<Option<Self::Item>>
    {
        // pin-project generates a safe projection method `project()`
        self.project().in_stream.poll_next(cx)
            .map(|r| r.map(|x| x * 2))
    }
}
```

Uses a lot of macros underneath (a bit out-of-scope).

The 'real' stream drivers



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>>  
}
```

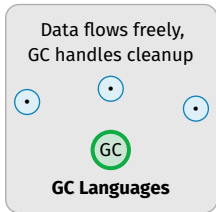
What about Rust rule `self` needs to be `Deref<Target=Self>`?

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

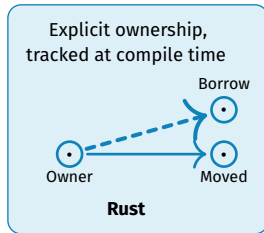
Problem? No, `Pin` is an exception in the compiler.

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



vs



Reactive patterns from GC languages require rethinking in Rust's ownership model

The end