RUST INDIA CONFERENCE - 2026

A Series of Unstoppable Events

Building Reliable Event Driven Systems in Rust

Shriram Balaji | Microsoft

About

- I work as a Senior Software Engineer at Microsoft, where I build distributed systems for the Microsoft 365 Data & Compute Platform.
- I'm interested in databases, linkers, compilers & distributed systems.
- I occasionally write about systems and rust on blog.shrirambalaji.com
- Find me on GitHub , X , LinkedIn as @shrirambalaji

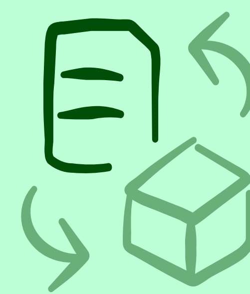
Martin Fowler, GOTO 2017

The *many meanings* of event-driven systems

Event Notification

Something happens, then we react.

State Transfer

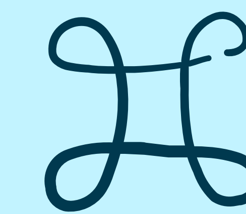


Events carry state for others to keep working.

Event Sourcing

State is rebuilt from the history of events ie. logs

CQRS



Segregate updates & reads to storage

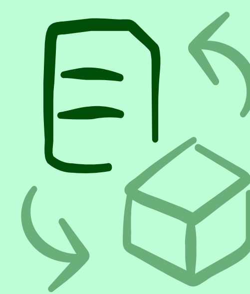
Martin Fowler, GOTO 2017

The *many meanings* of event-driven systems

Event Notification

Something happens, then we react.

State Transfer

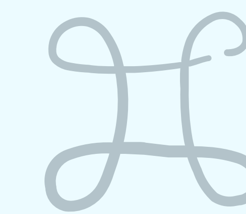


Events carry state for others to keep working.

Event Sourcing

State is rebuilt from the history of events ie. logs

CQRS

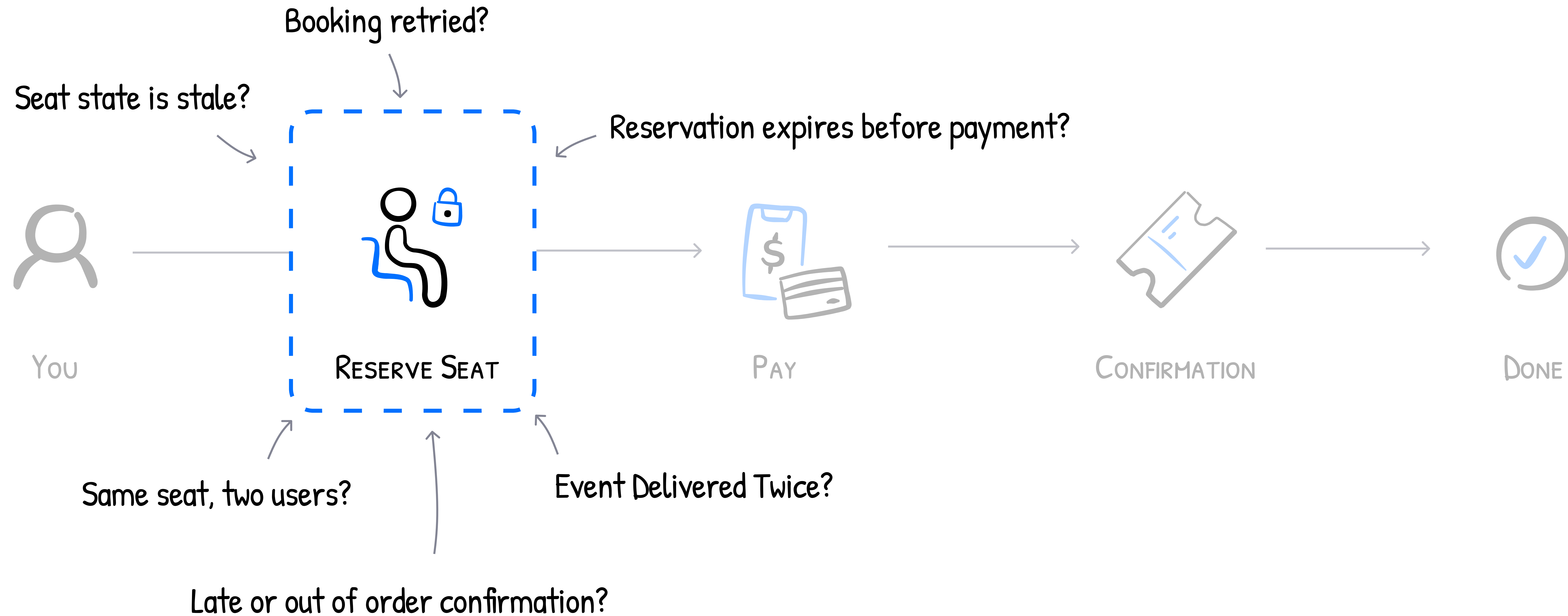


Segregate updates & reads to storage

Let's book some tickets!

Pretty simple right?

What It Takes to Reserve a Seat



What makes reliable event systems hard to build?

What happens under load?

When producers outrun consumers, queues grow and latency spikes.

What happens on retries?

The same work may be attempted again after partial failure.

What happens after a crash?

In-memory progress disappears, but the outside world keeps moving.

What happens when state must be rebuilt?

Recovery often means replaying history without repeating bad side effects.

What happens when events arrive out of order?

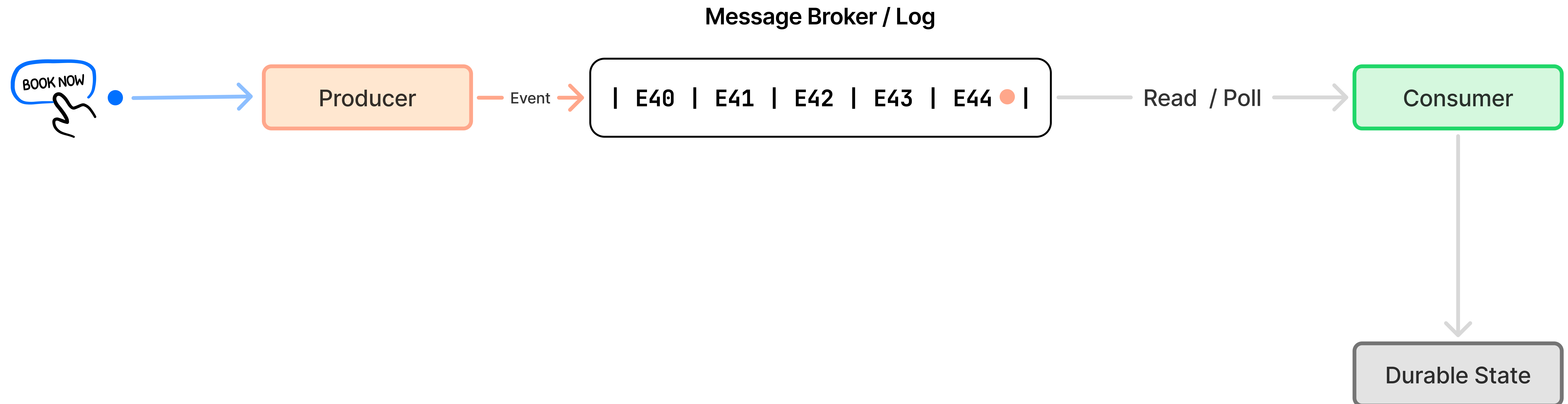
Real systems rarely preserve the order you hoped for.

What happens when we have duplicates?

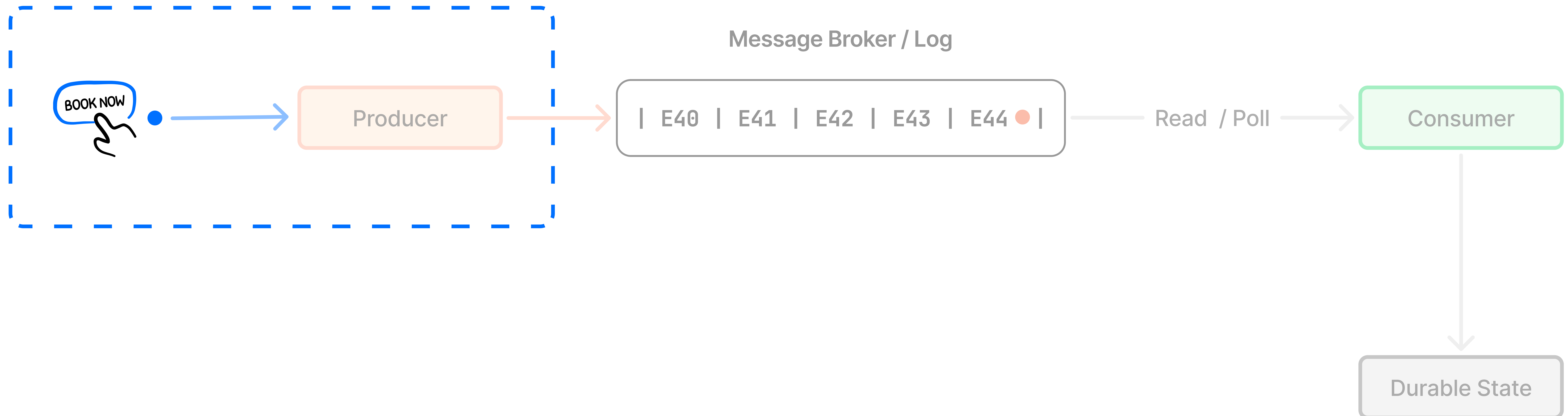
At-least-once delivery means the same event may arrive more than once.

and more . . .

The Anatomy of an Event-Driven System



Defining the Event



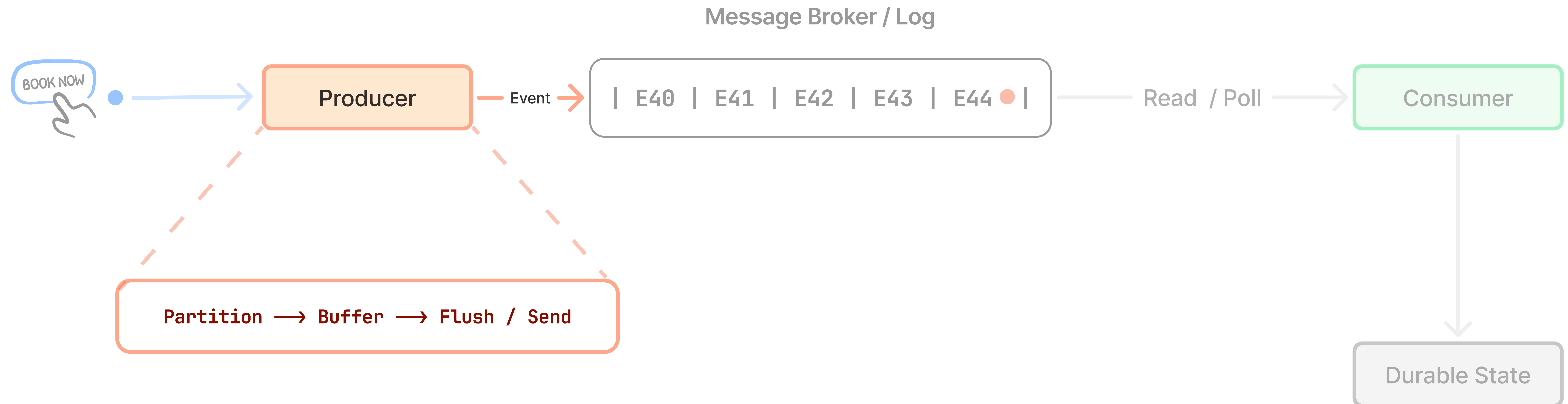
Defining the Event

```rust
pub enum BookingEvent {
    ReservationRequested {
        booking_id: Uuid,
        seat_id: String,
        user_id: Uuid,
    },
    PaymentReceived {
        booking_id: Uuid,
    },
}
```

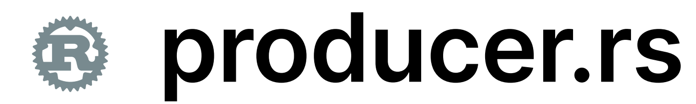
Defining the Event *over the wire* with Protobufs

```
message BookingEvent {
  oneof kind {
    SeatReservationRequested seat_reservation_requested = 10;
    PaymentReceived payment_received = 11;
    ReservationExpired reservation_expired = 12;
  }
}
message SeatReservationRequested {
  string booking_id = 1;
  string user_id = 2;
  string seat_id = 3;
}
```

The Producer



The Producer




```rust
struct Producer {
    topic: Topic,
    partitioner: Partitioner,
    buffers: PartitionBuffers,
    client: BrokerClient,
}
impl Producer {
    async fn send(&self, event: BookingEvent) -> Result<()>;
}
```

What's a Topic?

```rust
struct Producer {
    topic: Topic,
    partitioner: Partitioner,
    buffers: PartitionBuffers,
    client: BrokerClient,
}
impl Producer {
    async fn send(&self, event: BookingEvent) -> Result<()>;
}
```

```rust
struct Topic {
    name: String,
    partition_count: usize,
}
```

Why Partition?

Partitioning scales a topic by splitting one ordered sequence into multiple lanes.

Choosing a partition

```rust
// conceptual
fn partition_for(key: &[u8], n: u32) -> u32 {
    hash(key) % n
}
```

Buffers

```rust
struct Producer {
    topic: Topic,
    partitioner: Partitioner,
    buffers: PartitionBuffers,
    client: BrokerClient,
}
impl Producer {
    async fn send(&self, event: BookingE
}
```

```rust
struct PartitionBuffers {
    batches: HashMap<PartitionId, Batch>,
    // one batch per lane
}

struct Batch {
    events: Vec<Event>,
    bytes: usize,
    created_at: Instant,
}
```

Buffers & Flush

```rust
impl Producer {
    async fn send(&self, event: Event) -> Result<()> {
        let partition = self.partitioner.choose(&self.topic, &event);
        self.buffers.push(partition, event)?;
        if self.buffers.should_flush(partition) {
            self.client.flush(partition, &self.buffers).await?;
        }

        Ok(())
    }
}
```

Buffers & Flush

```rust
impl Producer {
    async fn send(&self, event: Event) -> Result<()> {
        let partition = self.partitioner.choose(&self.topic, &event);
        self.buffers.push(partition, event)?;
        if self.buffers.should_flush(partition) {
            self.client.flush(partition, &self.buffers).await?;
        }

        Ok(())
    }
}
```

Buffers & Flush

```rust
impl Producer {
    async fn send(&self, event: Event) -> Result<()> {
        let partition = self.partitioner.choose(&self.topic, &event);
        self.buffers.push(partition, event)?;
        if self.buffers.should_flush(partition) {
            self.client.flush(partition, &self.buffers).await?;
        }

        Ok(())
    }
}
```

When Buffers Fill: Backpressure

```rust
enum BackpressureMode {
    Block,                       // wait for space
    BlockWithTimeout(Duration), // wait, but bounded
    FailImmediately,             // return Err(Full) now
}
```

Backpressure turns load into latency, loss, or a crash. Pick one.

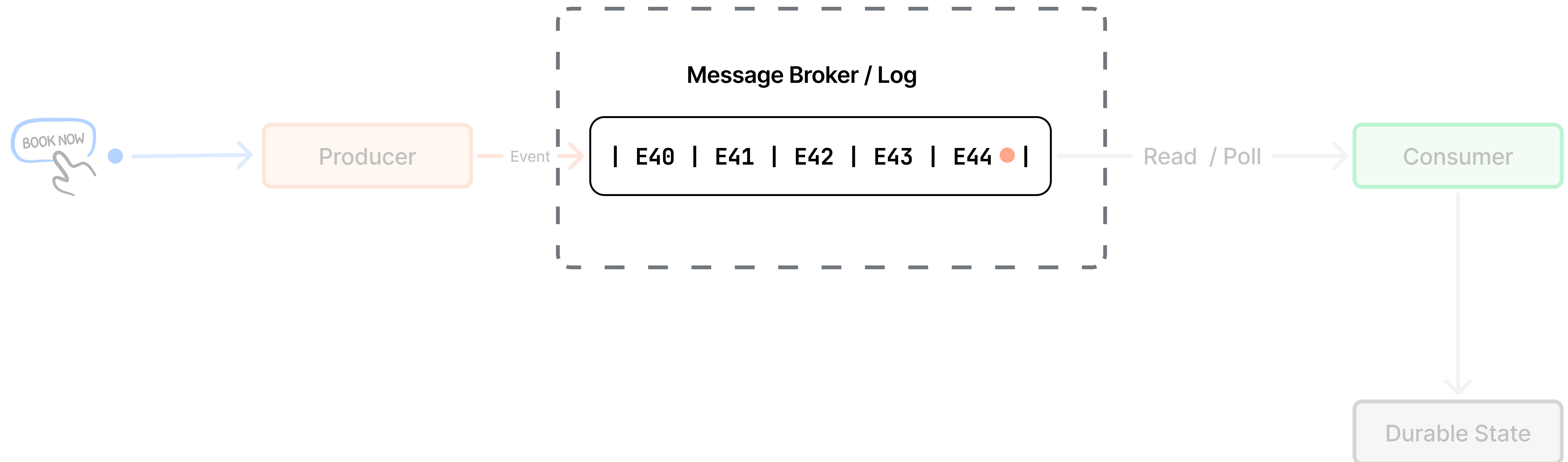
When Buffers Fill: Backpressure

```rust
enum BackpressureMode {
    Block,                           // wait for space
    BlockWithTimeout(Duration),      // wait, but bounded
    FailImmediately,                 // return Err(Full) now
}
impl Producer {
    async fn send(&self, event: Event) -> Result<()> {
        match self.buffers.push(partition, event)?; } {
            Ok(())        => {}
            Err(Full(e)) => self.mode.apply(e).await?,
```

Sending to the Broker

```rust
let event = BookingEvent::ReservationRequested { ... };
producer.send(event).await?;
```

The Message Broker



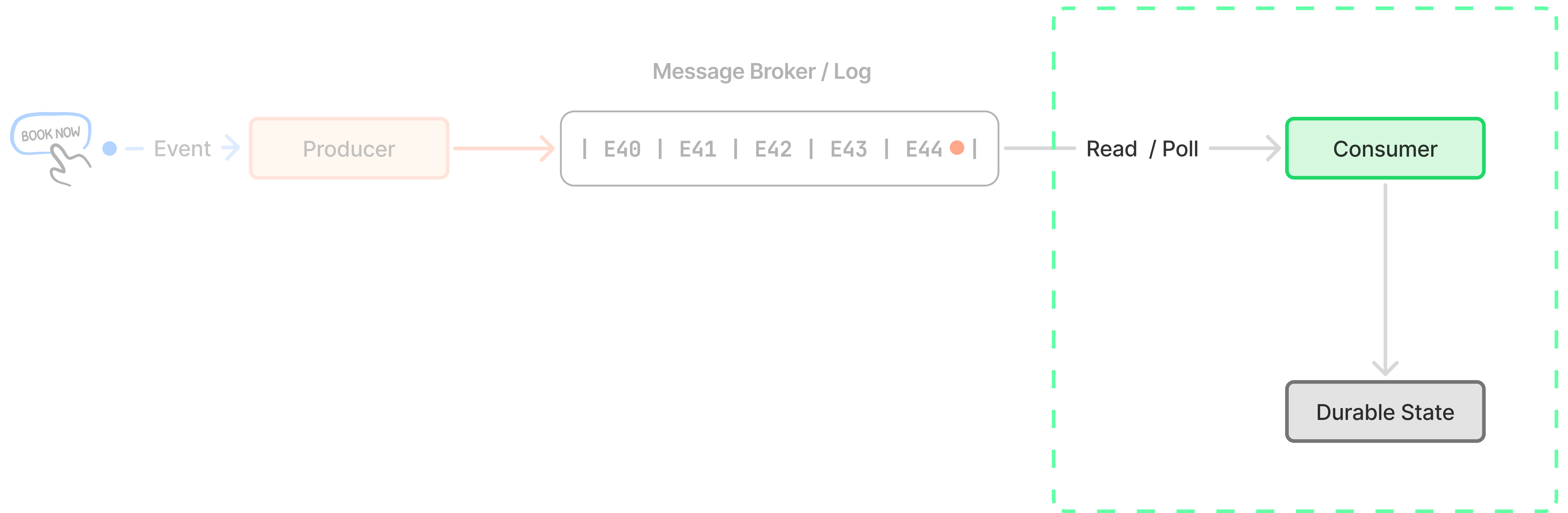
“A Message Broker is essentially a kind of database optimized for handling message streams. Producers write messages to the broker, and consumers recieve them by reading from the broker.

Martin Kleppman

Message brokers are hard.

Durable, Ordered, Replayable are three distributed systems problems in a trench coat.

The Consumer



Consumers are Streams

```rust
pub Stream for Consumer {
    type Item = Result<Event, Error>;
    fn poll_next(self: Pin<&mut Self>, cx: &mut Context) -> Poll<Option<Self::Item>>;
}
while let Some(event) = consumer.next().await {
    handle(event?).await?;
}
```

What’s a Stream?

```rust
pub trait Stream {
  type Item;

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

What is a Future?

```rust
pub trait Future {
    type Output;

    fn poll(self: Pin<&mut Self>, cx: &mut Context) -> Poll<Self::Output>;
}
```

What is a Future?

future.rs

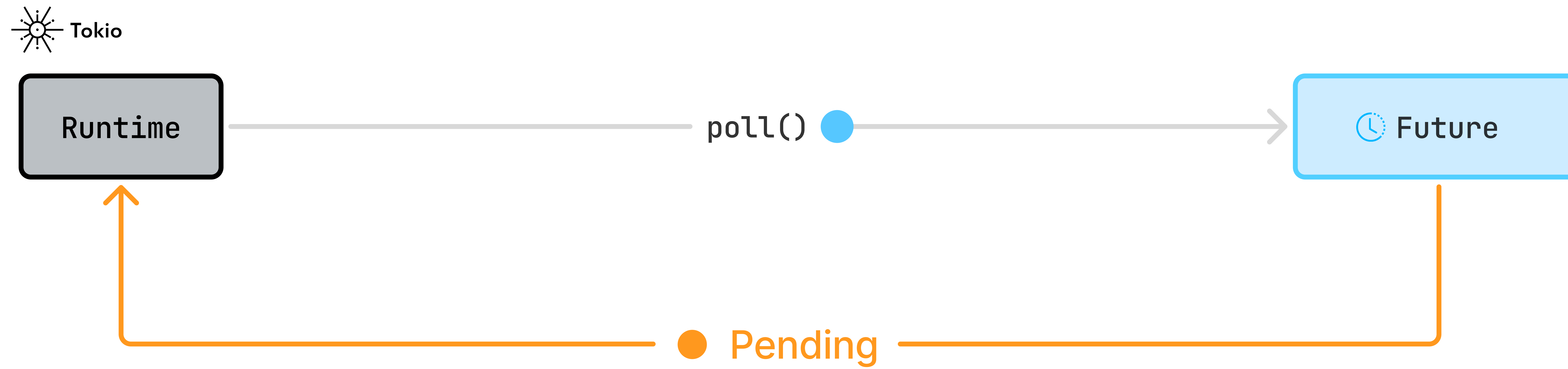
```rust
pub trait Future {
    type Output;

    fn poll(self: Pin<&mut Self>, cx: &mut Context) -> Poll<Self::Output>;
}
```

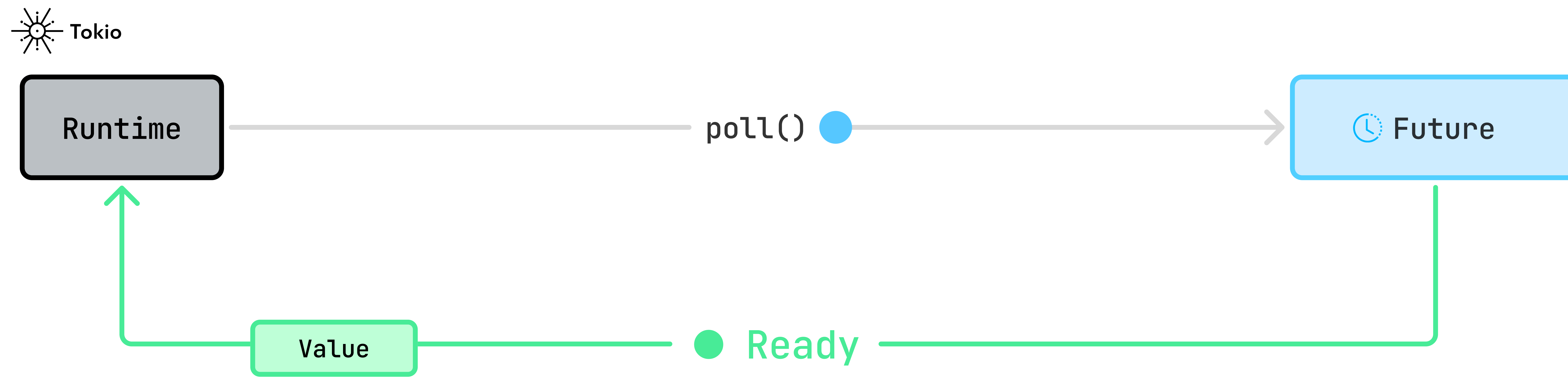
```rust
pub enum Poll<T> {
    Ready(T),
    Pending,
}
```

Poll indicates whether a value is available or if the task is scheduled to wakeup

What is a Future?



What is a Future?



Iterator

```rust
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}

// Iterating over a collection that impl Iterator
iter.next() -> Option<Item>
```

Stream is an Async Iterator

Consumers are Streams

```rust
impl Stream for Consumer {
    type Item = Result<Event, Error>;
    fn poll_next(self: Pin<&mut Self>, cx: &mut Context) -> Poll<Option<Self::Item>>;
}
while let Some(event) = consumer.next().await {
    handle(event?).await?;
}
```

Handling Events & State in the Consumer

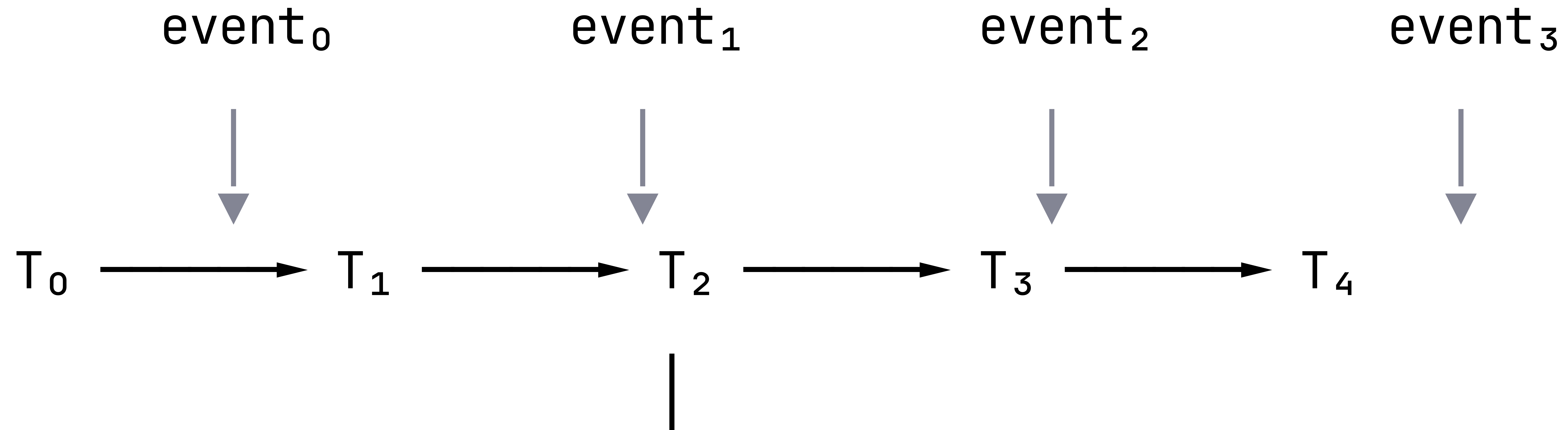
```rust
async fn handle(event: Event) -> Result<()> {
    let booking = decode(&event.payload)?;      // Received  -> Decoded
    db.insert(&booking).await?;                 // Decoded   -> Persisted
    metrics.bump();                             // Persisted -> Applied
    commit(event.offset).await?;                // Applied   -> Acked
    Ok(())
}
```

Offsets are watermarks

handle.rs

```rust
async fn handle(event: Event) -> Result<()> {
    let booking = decode(&event.payload)?;      // Received  → Decoded
    db.insert(&booking).await?;                 // Decoded   → Persisted
    metrics.bump();                             // Persisted → Applied
    commit(event.offset).await?;                // Applied   → Acked
    Ok(())
}
```

Offsets are watermarks



(durable up to T_2)

Watermarks enable at-least once delivery

The Gap between Effect & Commit

handle.rs

```rust
async fn handle(event: Event) → Result<()> {
    let booking = decode(&event.payload)?;      // Received  → Decoded
    db.insert(&booking).await?;                 // ← Crash Here
    metrics.bump();
    commit(event.offset).await?;
    Ok(())
}
```

Recovery & Replay

```rust
async fn replay_from(last_committed: Offset) -> Result<()> {
    let mut stream = consumer.stream_from(last_committed.next()).await?;
    while let Some(event) = stream.next().await {
        handle(event).await?;
        commit(event.offset).await?;
    }
    Ok(())
}
```

What makes reliable event systems hard to build?

What happens under load?

Backpressure slows producers when consumers fall behind.

What happens on retries?

The same work may be attempted again after partial failure.

What happens after a crash?

Recovery restarts from the last committed offset.

What happens when state must be rebuilt?

Replay reconstructs state from the log.

What happens when events arrive out of order?

Ordering is usually preserved only within a key or partition.

What happens when we have duplicates?

Idempotent event handlers

and more . . .

Thank You

Find me on GitHub , X , LinkedIn as @shrirambalaji