perplexity

explain me clearly what actually happen

future is not `Send` as this value is used across an await

The error "future is not `Send` as this value is used across an await" happens because Rust requires futures that may be sent between threads to be thread-safe, i.e., implement the `Send` trait. Here is what actually happens step-by-step:

What happens under the hood?

- An `async` function or block in Rust compiles into a state machine that implements the `Future` trait.
- When you `.await` inside an async block, the compiler generates code that can suspend and resume the future.
- If the async runtime is multithreaded (e.g., Tokio's multi-threaded executor), the future may be moved between threads at suspension points (`.await`).
- For this to be safe, **all data held across** `.await` **points must be** `Send` so it can be safely transferred across threads.

Why does the error occur?

- The compiler detects that some value (variable, reference, or captured environment) is held across an `.await` point.
- That value or one of its components does **not implement** `Send`.
- Because the future holds that non-`Send` value across `.await`, the entire future is considered **not** `Send`.
- Since the async runtime requires futures to be `Send` to run them on thread pools, the compiler emits an error.

Common causes

- Holding non-thread-safe types like `Rc`, `RefCell`, or non-`Send` trait objects across `.await`.
- Awaiting on a future that itself is not `Send`.
- Capturing a non-`Send` reference or guard (e.g., `MutexGuard`) across an `.await`.
- Returning a trait object like `Box<dyn std::error::Error>` without `Send` bounds.

Example from the Rust Users Forum

```
Box::pin(async {
    match message {
        WorldRequest::HelloRequest(hello_request) =&gt;
            WorldResponse::HelloResponse(self.hello(hello_request.name).await),
    }
})
```

Here, `self.hello(hello_request.name).await` returns a future that is **not** `Send`. Because this `.await` is inside the async block, the entire future is not `Send`, causing the error.

How to fix it?

- Make sure all types held across `.await` implement `Send`.
- Add `+ Send + Sync + 'static` bounds to trait objects like `dyn std::error::Error`.
- Avoid holding non-`Send` values (like `Rc`, `RefCell`, or `MutexGuard`) across `.await`.
- Ensure futures you `.await` are themselves `Send`.
- If implementing async traits, add `Send` bounds on the returned futures.

Summary

The key point is that **any value held across an `.await` suspension point must be safe to send between threads** because the future might be resumed on a different thread. If not, the compiler refuses to compile, producing the "future is not `Send` as this value is used across an await" error.

This is Rust's way of guaranteeing thread safety in asynchronous code running on multithreaded executors, preventing subtle data races or undefined behavior[1] [2] [3] [4].

⁂

1. https://users.rust-lang.org/t/future-cannot-be-sent-between-threads-safely-but-only-with-generic-impl/100326
2. https://users.rust-lang.org/t/why-is-the-future-not-implemented-send/89238
3. https://users.rust-lang.org/t/async-await-does-rust-ensure-thread-safety-when-using-multithreaded-executor/40585
4. https://rust-lang.github.io/async-book/07_workarounds/03_send_approximation.html