perplexity

try without Pin box dyn future, just async fn

Okay, let's explore why you can't directly use `async fn` in a trait without encountering the auto-trait issues (specifically related to `Send`) and why the `Pin<Box<dyn Future...>>` or `async_trait` solutions are necessary.

The Core Problem (Revisited):

The fundamental issue is that when you use `async fn` in a public trait, you are *not* specifying the concrete type of the `Future` that the function returns. The compiler infers this type based on the contents of the `async fn` block.

1. **Unknown `Future` Type:** Because the trait doesn't specify the `Future` type, you can't add trait bounds to it. You can't say, "This `Future` *must* implement `Send`."
2. **Auto-Trait Inference:** Rust's auto traits (like `Send`, `Sync`, `UnwindSafe`) are automatically implemented for types based on their fields. If the compiler-generated `Future` contains any non-`Send` types (e.g., raw pointers, `Rc`), the `Future` itself will *not* be `Send`.
3. **Concurrency Issues:** This becomes a problem when you want to use the `Future` in a concurrent context (e.g., spawning a task on a thread pool), which requires the `Future` to be `Send`.
4. **Object Safety:** Traits with `async fn` are not object safe, which means they can't be used with trait objects.
5. **Breaking Changes:** You're not guaranteeing that the returned type will be `Send`, and a future change could cause things to break.

Why `Pin<Box<dyn Future + Send>>` or `async_trait` Work:

- `Pin<Box<...>>`**:** By returning a boxed trait object (`dyn Future + Send`), you're explicitly saying that the function returns *any* type that implements both the `Future` trait and the `Send` trait. The `Box` puts the `Future` on the heap, and the `Pin` ensures that it won't be moved in memory while it's being awaited. `async_trait` does this for you under the hood.
- `async_trait`**:** This crate essentially does the same thing, but it hides the complexity of manual boxing and pinning. It transforms your `async fn` into a function that returns a boxed and pinned `Future` that is guaranteed to be `Send`.

Why You Can't Just Use `async fn` Directly:

Let's illustrate with code what happens if you naively try to use `async fn` in a trait:

```
// This will cause a warning!
pub trait MyTrait {
    async fn mymethod(&mut self) -> &mut Self;
    async fn another_method(&mut self) -> &mut Self;
```

```
}

#[derive(Debug)]
pub struct MyStruct {
    state: i32,
}

impl MyStruct {
    pub fn new() -> Self {
        MyStruct { state: 0 }
    }
}

impl MyTrait for MyStruct {
    async fn mymethod(&mut self) -> &mut Self {
        self.state += 1;
        self
    }

    async fn another_method(&mut self) -> &mut Self {
        self.state *= 2;
        self
    }
}

#[tokio::main]
async fn main() {
    let mut instance = MyStruct::new();
    instance.mymethod().await;
    instance.another_method().await;
}
```

The compiler will issue the warning:

```
warning: `async fn` in public traits is discouraged as auto trait bounds cannot be specif
   --> src/main.rs:2:5
    |
2   |     async fn mymethod(&mut self) -> &mut Self;
    |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    |
note: the return type of this `async fn` is a `Future` that may not be `Send`
   --> src/main.rs:2:5
    |
2   |     async fn mymethod(&mut self) -> &mut Self;
    |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
help: consider using `async_trait` to automatically make the `Future` `Send`
    |
2   + #[async_trait::async_trait]
3   |
```

The Warning is the Key:

The warning explicitly states that the return type (the `Future`) *may not* be `Send`. The compiler *cannot guarantee* that it's `Send`, and you have no way to enforce it. If you try to use this trait in a

concurrent context, you might get runtime errors or undefined behavior.

In Summary:

You *can* technically write the code with just `async fn` in the trait, but it's highly discouraged for public traits because:

1. You lose control over the `Future` type.
2. You can't guarantee `Send`, which is crucial for concurrency.
3. It can lead to subtle and difficult-to-debug errors.

The `Pin<Box<dyn Future + Send>>` pattern (or the `async_trait` crate, which does it for you) is the standard and recommended way to solve these problems.

Why Not Just Make Everything `Send`?

Rust's type system is designed to be precise. It doesn't automatically make everything `Send` because that could introduce unnecessary overhead and restrict the types of data you can use in your `Future`s. The `Send` bound is a contract: it guarantees that the type is safe to move between threads. Rust forces you to be explicit about this contract to ensure that you don't accidentally introduce concurrency bugs.

For Kernel/HPC:

In your kernel/HPC work, the need for concurrency safety is even more critical. You should *always* use `async_trait` or the manual `Pin<Box<...>>` approach to ensure that your `Future`s are `Send` when working with public traits. Ignoring the warning can lead to very serious issues in these environments. You don't want race conditions or data corruption in a kernel or HPC application.

Simplified Analogy:

Imagine you have a box. You want to send this box to someone else via a delivery service.

- **Without `Send`:** You're saying, "Here's a box. I don't know what's inside, and I don't know if it's safe to send it through the delivery service." The delivery service might refuse to handle it, or it might get damaged in transit.
- **With `Send`:** You're saying, "Here's a box. I guarantee that everything inside is packaged safely and can be sent through the delivery service without any problems." The delivery service knows it's safe to handle the box.

The `Send` trait is like that guarantee for concurrent code.