```rust
fn main() {
    println!("hi there");
}
```

The main function doesn't have to return anything

```rust
fn main() -> Result<(), Error> {
    Ok(())

    Err(Error::other("asdf"))
}
```

Optionally, you can have it return a 'Result'

If you return an Ok variant, Rust won't do anything

If you return an Err variant, Rust will print the value in the Err variant

```rust
fn main() -> Result<(), Error> {
    let text = fs::read_to_string("logs.txt")?;

    Ok(())
}
```

'?' operator gets added onto functions that return a Result

If the function returns an Ok, the value inside is automatically extracted

If it contains an Err(), the Err() variant is automatically returned

Methods attached to the 'Option' enum

```
item.unwrap()
```

- If 'item' is a Some, returns the value in the Some
- If 'item' is a None, panics!
- Use for quick debugging or examples

```
item.expect("There should be
    a value here")
```

- If 'item' is a Some, returns the value in the Some
- If 'item' is a None, prints the provided debug message and panics!
- Use when we **want** to crash if there is no value

```
item.unwrap_or(&placeholder)
```

- If 'item' is a Some, returns the value in the Some
- If 'item' is a None, returns the provided default value
- Use when it makes sense to provide a fallback value

Many of the same methods work with Result

```
text.unwrap()
```

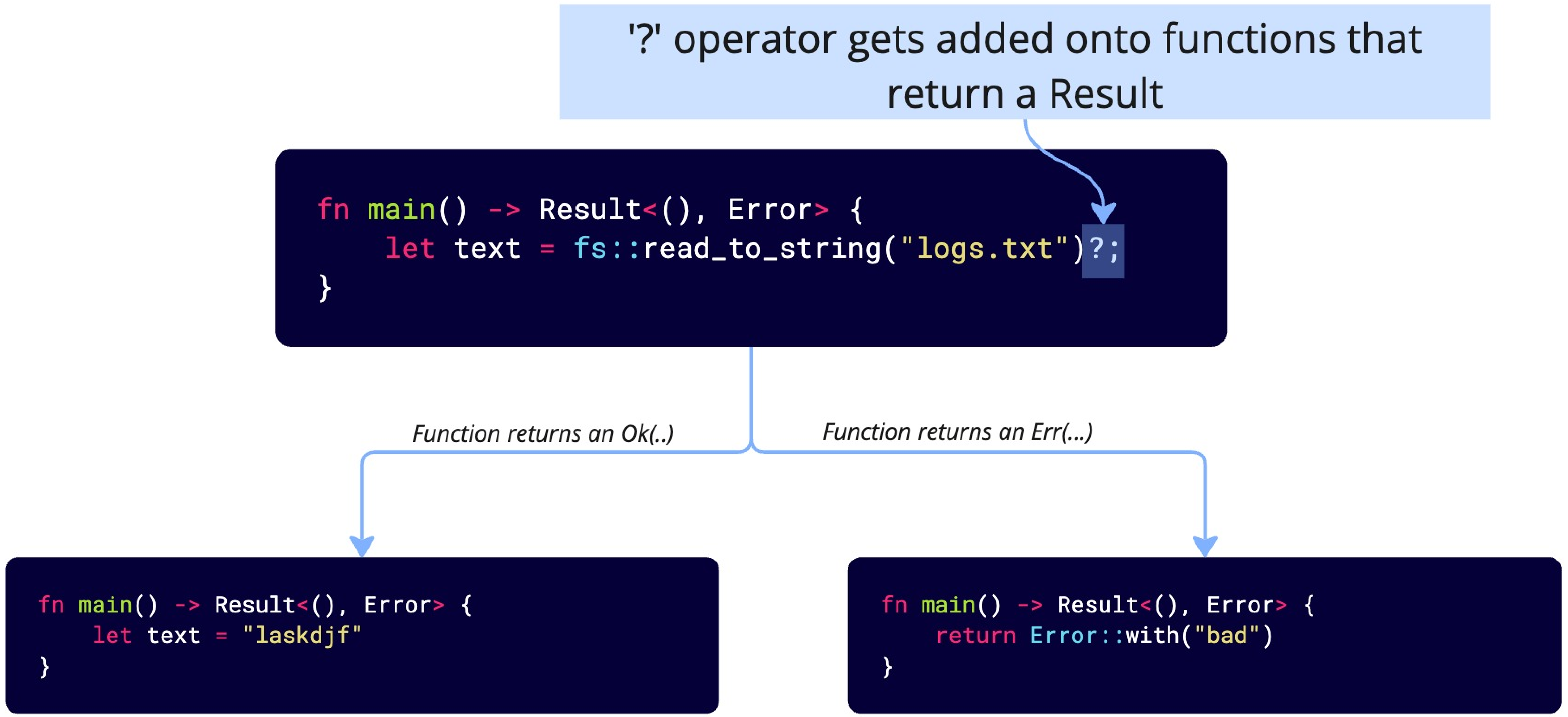
- If 'text' is an Ok, returns the value in the Ok
- If 'text' is an Err, panics!
- Use for quick debugging or examples

```
text.expect("couldnt open
    the file")
```

- If 'text' is an Ok, returns the value in the Ok
- If 'text' is an Err, prints the provided debug message and panics!
- Use when we **want** to crash if something goes wrong

```
text.unwrap_or(
  String::from("backup text")
)
```

- If 'text' is an Ok, returns the value in the Ok
- If 'text' is an Err, returns the provided default value
- Use when you want a fallback default value in case something goes wrong



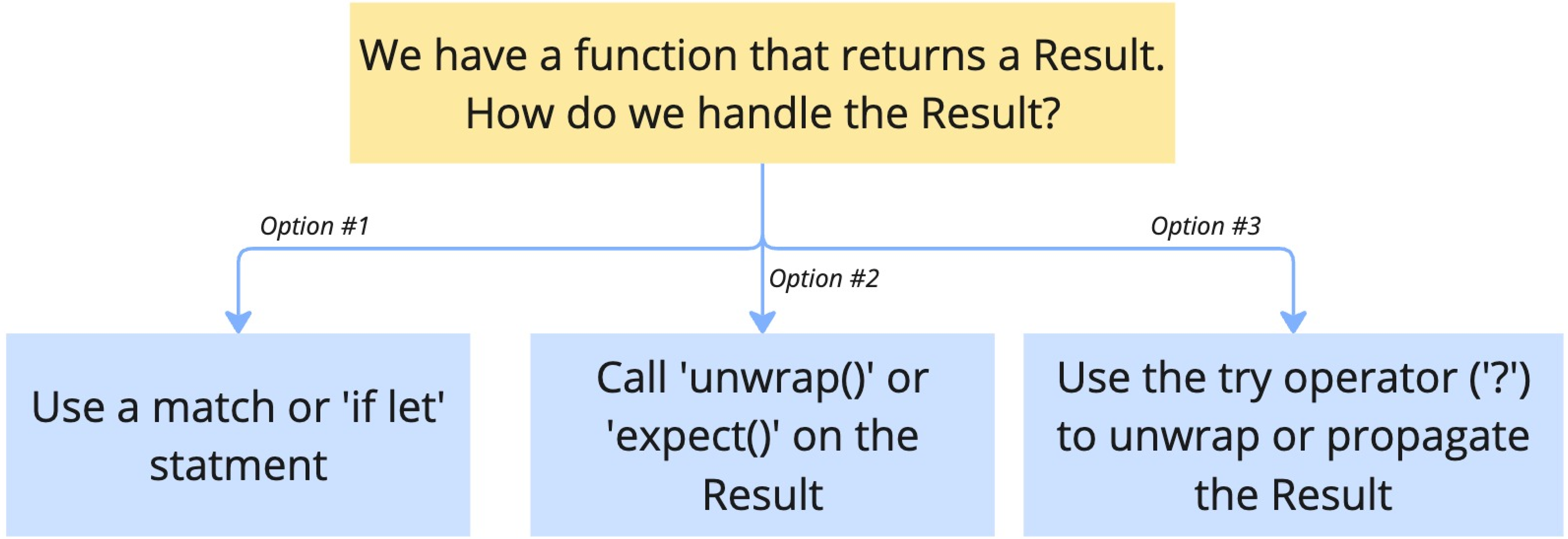
'?' operator gets added onto functions that return a Result

```
fn main() -> Result<(), Error> {
    let text = fs::read_to_string("logs.txt")?;
}
```

```
fn main() -> Result<(), Error> {
    let text = "laskdjf"
}
```

```
fn main() -> Result<(), Error> {
    return Error::with("bad")
}
```

```rust
fn do_something() -> Result<_, _> {}

fn my_function() {
    do_something() // A Result!
}
```


We have a function that returns a Result. How do we handle the Result?

Option #1

Use a match or 'if let' statment

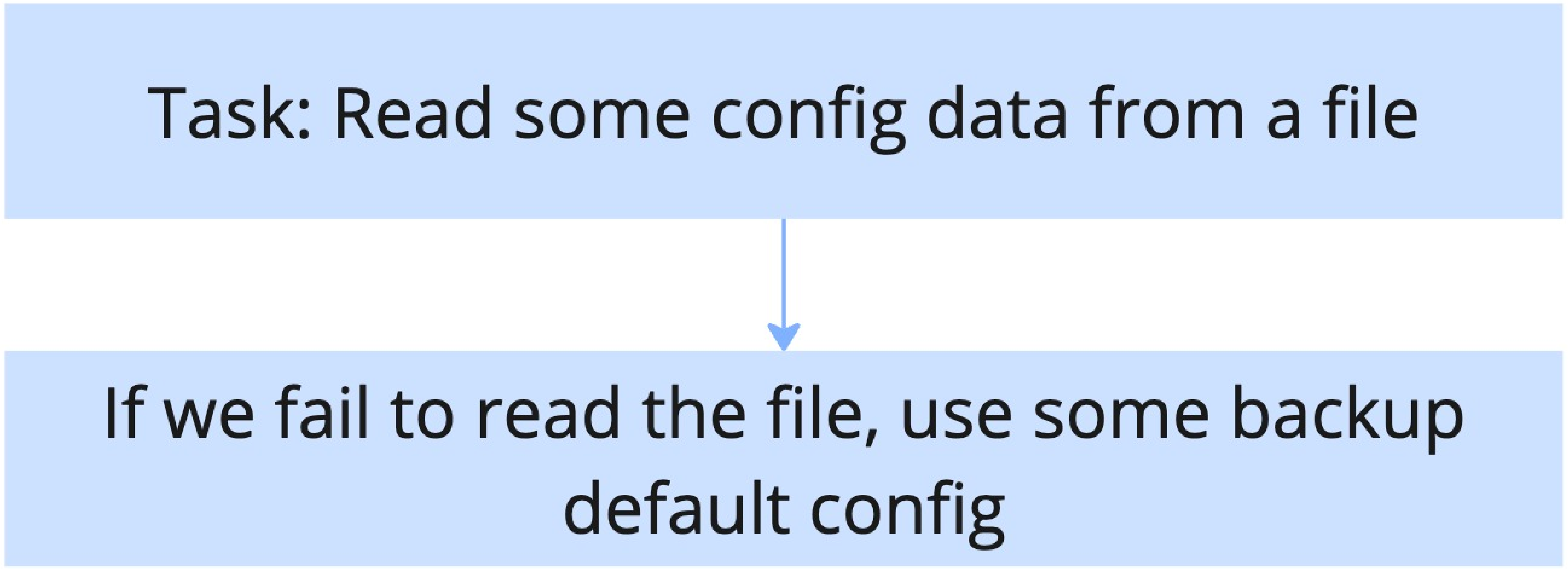
Option #2

Call 'unwrap()' or 'expect()' on the Result

Option #3

Use the try operator ('?') to unwrap or propagate the Result

1	Use a match or 'if let' statement	When you're ready to meaningfully deal with an error
2	Call 'unwrap()' or 'expect("why this paniced")' on the Result	Quick debugging, or if you want to crash on an Err()
3	Use the try operator ('?') to unwrap or propagate the Result	When you don't have any way to handle the error in the current function

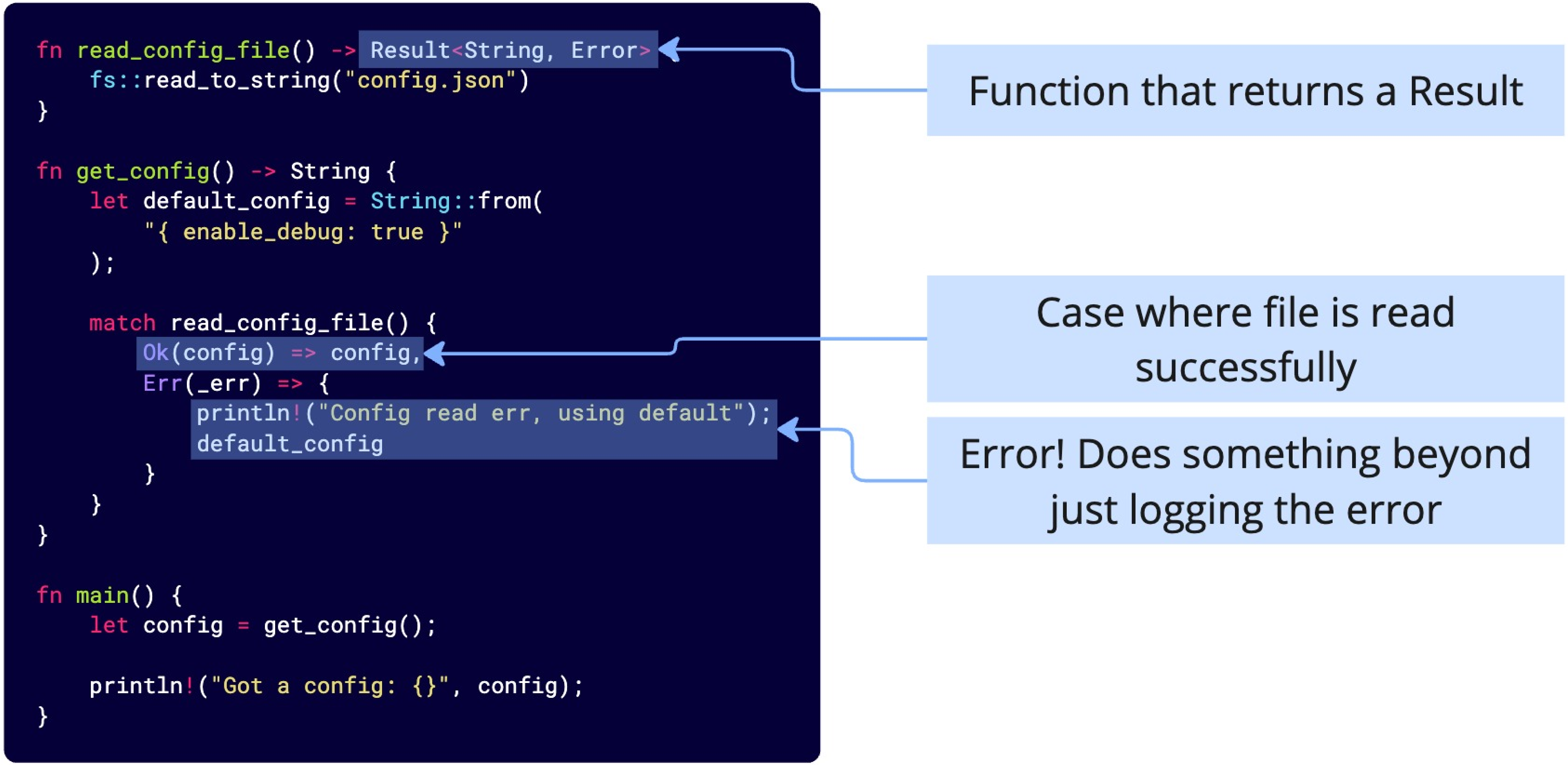


Task: Read some config data from a file

If we fail to read the file, use some backup default config

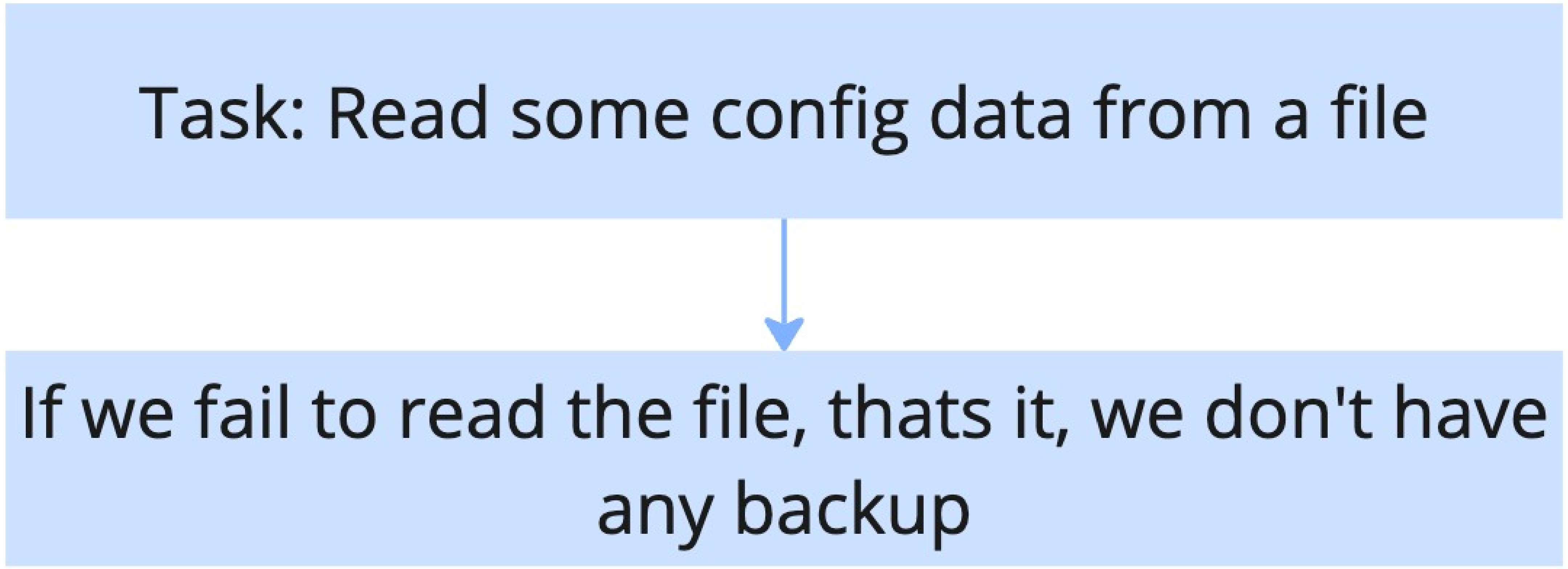
If we get an error, we have a workaround - a meaningful way of dealing with the error besides just logging it

Match statements: Good for dealing with an error



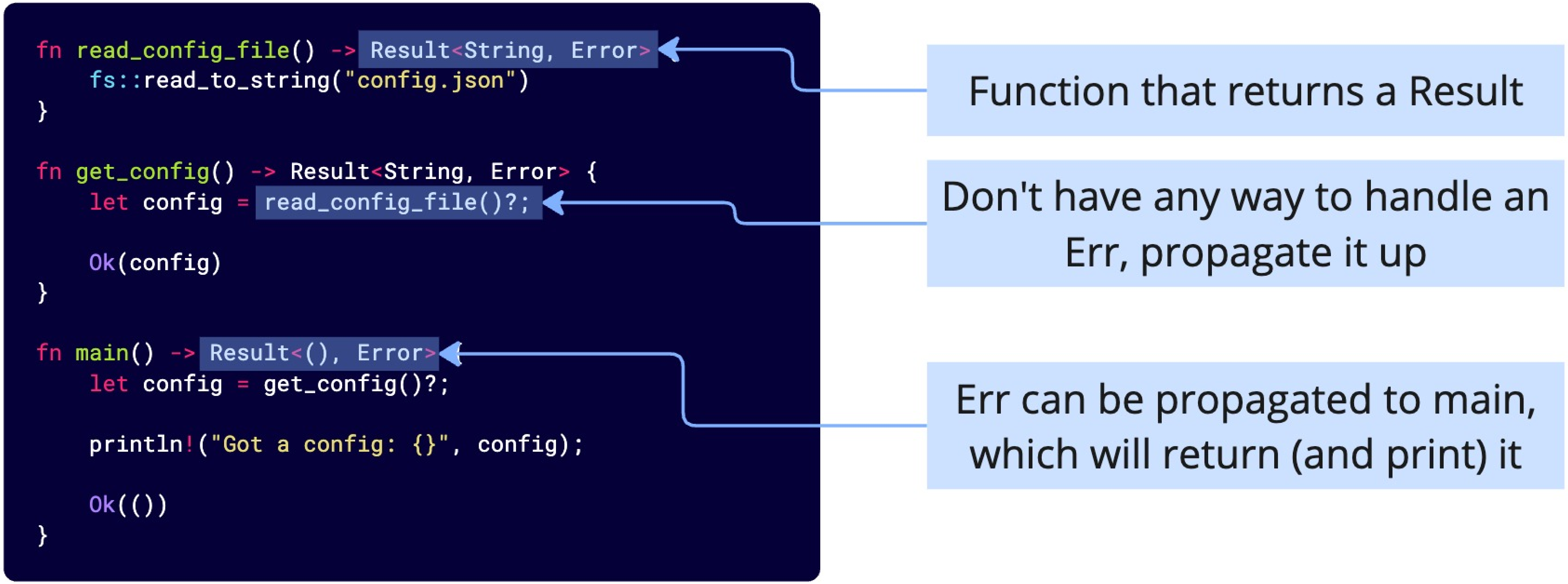
```rust
fn read_config_file() -> Result<String, Error>
    fs::read_to_string("config.json")
}

fn get_config() -> String {
    let default_config = String::from(
        "{ enable_debug: true }"
    );

    match read_config_file() {
        Ok(config) => config,
        Err(_err) => {
            println!("Config read err, using default");
            default_config
        }
    }
}

fn main() {
    let config = get_config();

    println!("Got a config: {}", config);
}
```

Function that returns a Result
Case where file is read successfully
Error! Does something beyond just logging the error


Task: Read some config data from a file
If we fail to read the file, thats it, we don't have any backup

If we get an error, we don't have any workaround

Try Operator: Propagate errors when you just don't know how to handle them



```rust
fn read_config_file() -> Result<String, Error>
    fs::read_to_string("config.json")
}

fn get_config() -> Result<String, Error> {
    let config = read_config_file()?;

    Ok(config)
}

fn main() -> Result<(), Error>{
    let config = get_config()?;

    println!("Got a config: {}", config);

    Ok(())
}
```

Function that returns a Result

Don't have any way to handle an Err, propagate it up

Err can be propagated to main, which will return (and print) it