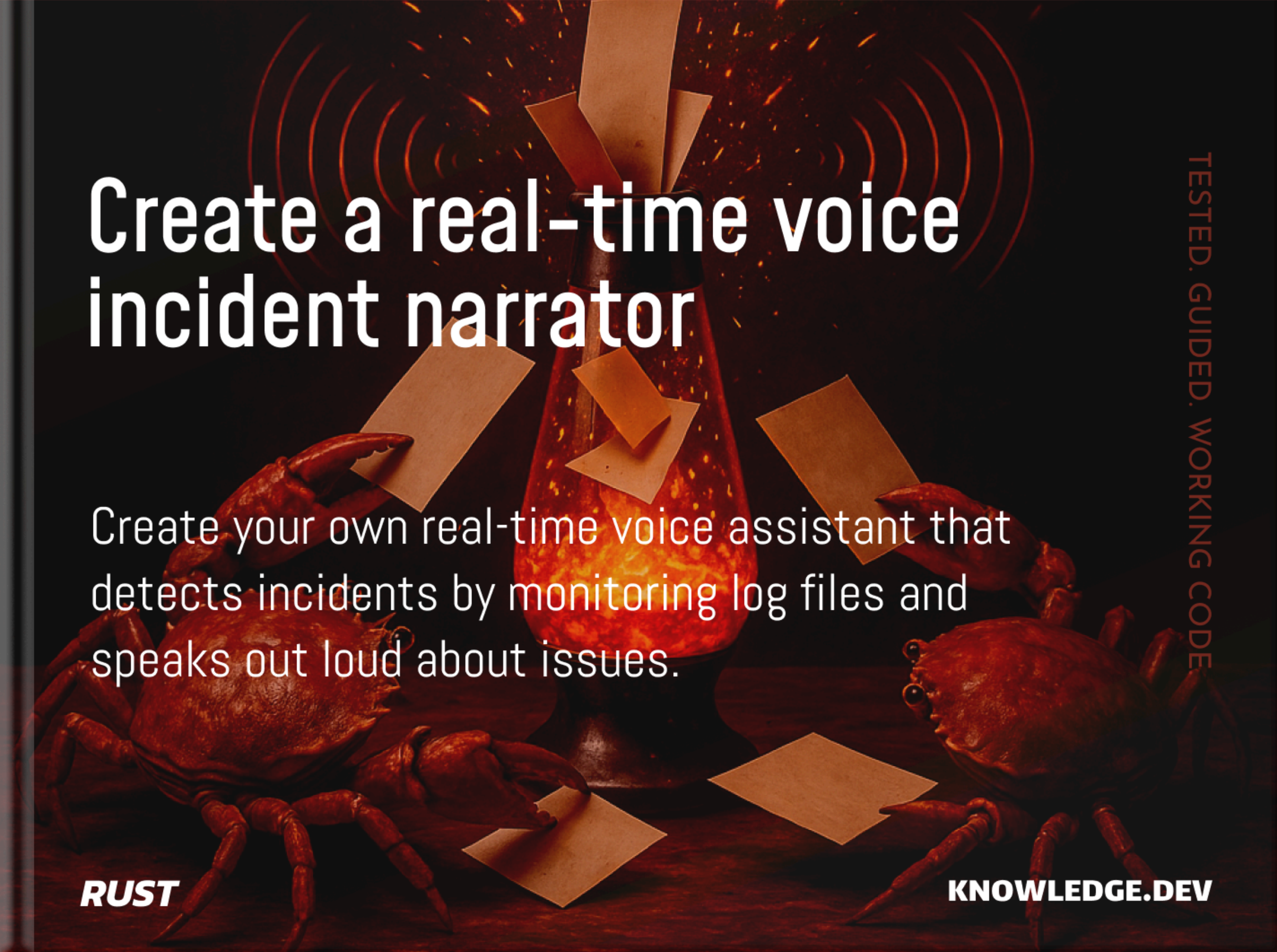
PRACTICAL LEARNING

SERIES: AGENTS



Create a real-time voice incident narrator

Create your own real-time voice assistant that detects incidents by monitoring log files and speaks out loud about issues.

TESTED. GUIDED. WORKING CODE.

RUST

KNOWLEDGE.DEV

Contents

About This Book

This section provides essential information about the book's structure, conventions, and how to get the most out of your learning experience.

How to Read This Book?

This book is a **step-by-step guide** to building a real Rust program from scratch. It's designed like an **instruction manual** for a construction kit — if you follow each step carefully, you'll assemble a working solution and gain hands-on development experience.

Step-by-Step Instructions

Each chapter breaks down the development process into clear, manageable steps. Every step shows you exactly what changes to make, which files to modify, and what the result should be. Think of it as building with LEGO blocks — one piece at a time, following the instructions, until you have a complete working program.

Adapted for Learning

Every step includes detailed explanations of all concepts involved. You learn by doing, similar to how AI models are trained — by repetition and practice. Your brain processes each pattern, each solution, building neural pathways through hands-on experience.

This approach ensures you don't just copy code — you understand **why** it works and **how** the pieces fit together.

Adapted for Vibe-Coding

This book embraces modern **vibe-coding** workflows where you work alongside AI assistants. The premium version includes carefully crafted **prompts** designed for coding agents like Claude, GitHub Copilot, and other AI tools.

These prompts look like this:

> (> *This is a prompt for a coding agent — copy and use it with AI assistants*

Just copy the prompt, paste it into your AI assistant, and get context-aware code suggestions. This makes development faster while maintaining the learning benefits — you see how AI interprets requirements and produces solutions.

File Changes with Guarantees

At each step, you'll see the **delta of file changes** — exactly what lines were added, modified, or removed. This approach guarantees that if you replicate everything as shown, your program will compile successfully.

No guesswork, no missing pieces — just follow the changes, and you'll have a working program. This makes the book perfect for both manual coding and AI-assisted development.

Who Is This Book For?

Whether you're learning Rust for the first time, exploring AI-assisted coding, or just want a structured project to build, this book provides a clear path from empty project to working software.

Formatting Conventions

This book uses consistent formatting conventions to help you navigate and understand the material more easily.

Code Blocks

Code examples appear in monospace font with syntax highlighting. Line numbers on the left help you reference specific lines:

```rust
fn main() {
    println!("Hello, world!");
}
```

File Changes

When modifying existing files, changed lines are highlighted to show exactly what was added or modified. This helps you track incremental changes throughout the book.

Command Prompts

Terminal commands are shown with a dollar sign prefix and include the current working directory:

```sh
cargo new my-project
cd my-project
cargo run
```

Emphasis and Formatting

Italic text indicates new terms or concepts being introduced for the first time.

Bold text highlights important library names, framework names, or critical concepts you should pay special attention to.

`Inline code` appears in monospace font for variable names, function names, file paths, and short code snippets within regular text.

Notes and Callouts

Important information, tips, and warnings appear in special formatted boxes to draw your attention to critical details or common pitfalls.

File Paths

File paths use forward slashes and are shown relative to the project root unless otherwise specified. For example: `src/main.rs` refers to the main source file.

Version Numbers

Code examples in this book are tested with specific versions of Rust and dependencies. While newer versions should generally work, check the book's introduction for the recommended version information.

Introduction

The tool we've built is like an incident radio for your backend: it listens to your logs, notices when things start going wrong, and tells you the story out loud. Instead of staring at scrolling text all day, you get short spoken updates about what's breaking and how it's evolving over time.

Normally, you might do something like:

```sh
tail -f sample.log | grep -E 'ERROR|WARN'
```

That shows every raw line but doesn't answer the real question: "What is actually happening to my system right now?" With this tool, you can run:

```sh
cargo run -- sample.log
```

and hear something like: "Login failures are rising, the cache is unstable, and traffic is falling back to the database."

Conceptually, it's a log companion that turns noisy error streams into calm, periodic status reports. It keeps a rolling memory of past problems for context, focuses on the newest failures, and turns that into a short, non-technical summary you could play over speakers while you work on something else. The idea is that you only drop back into dashboards or log searches when the radio tells you there's a real story unfolding.

We aim this at humans first, not machines, so the summaries are brief, plain-language, and easy to follow even if you're not deeply technical. That also makes it safe to treat as background audio: you can let it run alongside your usual tools and only react when it says something important.

Build Log Watcher

Initialize Log Watcher

Let's begin building a utility to reproduce errors from logs by creating a new binary crate with the `cargo new` command, specifying the `--bin` flag and the crate name.

.

SH

```sh
$ cargo new --bin incident-narrator
```

Cargo.toml

TOML

```toml
[package]
name = "incident-narrator"
version = "0.1.0"
edition = "2021"
[dependencies]
```

src/main.rs

RUST

```rust
fn main() {
}
```

Let's create the first module responsible for reading files and monitoring changes to them. To do this, create a `watcher` module and a struct with the same name inside it.

src/main.rs RUST

```rust
mod watcher;
```

src/watcher.rs RUST

```rust
pub struct Watcher {
}
```

An instance of `Watcher` will monitor exactly one file, but to find and read that file, we need its path.

The most convenient type to store the path inside the struct is `PathBuf`. Import it from the `std::path` module and add a `log_path` field to the struct.

src/watcher.rs RUST

```rust
use std::path::PathBuf;
pub struct Watcher {
    log_path: PathBuf,
}
```

Let's create a constructor for the `Watcher` structure: add a `new()` method that takes the path of the file to be watched as its parameter.

You can use the already imported `PathBuf` as the argument type.

src/watcher.rs RUST

```rust
impl Watcher {
    pub fn new(log_path: PathBuf) {
    }
}
```

Store the provided value in a field of the newly created `Watcher` struct instance, and return it from the `new()` method using `Self` as the return type—an alias for the type itself.

src/watcher.rs RUST

```rust
impl Watcher {
    pub fn new(log_path: PathBuf) -> Self {
        Self {
            log_path,
        }
    }
}
```

Now we need to get the filename from the command-line parameter and pass it to our `Watcher`.

To parse the provided arguments more conveniently, import the `clap` crate. Enable the `derive` feature, which lets us turn a struct (that we will add next) into an argument parser.

You can also enable the `env` feature right away so that later we can read the AI model's API key from an environment variable.

Cargo.toml TOML

```toml
[dependencies]
clap = { version = "4.5", features = ["derive", "env"] }
```

Import the `Parser` trait from the `clap` crate, declare a new `Args` struct, and enable parsing by adding a `derive` attribute with the imported type. This creates a command-line argument parser.

The created parser is empty and, for now, it does not accept any arguments.

src/main.rs RUST

```rust
use clap::Parser;
#[derive(Parser)]
struct Args {
}
```

We need the log file name as an argument, which we will watch; to add this field, import the `PathBuf` type we used earlier in the `Watcher` itself, and add the `log_file` field, which is a required positional argument.

In this case, invoke the command by passing the file name.

src/main.rs RUST

```rust
use std::path::PathBuf;
#[derive(Parser)]
struct Args {
    #[arg(value_name = "LOG_FILE")]
    log_file: PathBuf,
}
```

The `Parser` trait that we imported and implemented for the `Args` structure provides the `parse()` function, which parses the command-line arguments and maps them to the structure, or displays help and the required parameters if anything is specified incorrectly. Call this function and store the result in the `args` variable.

Now that we have the log file name, create an instance of the `Watcher` component, import it from the module we created, call the `new()` function we implemented, passing the value of the `log_file` field extracted from the arguments structure as the parameter, and store the resulting instance in the `watcher` variable.

src/main.rs RUST

```rust
use watcher::Watcher;
fn main() {
    let args = Args::parse();
    let watcher = Watcher::new(args.log_file);
}
```

Now let's add the `run()` method to our `Watcher`, which will serve as the entry point for monitoring file changes and reading the file.

Although the method is still empty, we can already call it on our instance.

src/main.rs RUST

```rust
fn main() {
    let watcher = Watcher::new(args.log_file);
    watcher.run();
}
```

src/watcher.rs RUST

```rust
impl Watcher {
    pub fn run(self) {
    }
}
```

Even though the program reads log files, it's useful to add logging to the program itself to understand what's happening and when different components start. Later, we will have one task reading and watching the log file for changes, and another sending asynchronous requests to the model to generate summaries.

The `tracing` crate provides macros for logging, and the `tracing-subscriber` crate outputs those logs.

In the latter crate, it's helpful to enable the `env-filter` feature, which lets you control the logging level by setting a filter, for example via the `RUST_LOG` environment variable.

Cargo.toml TOML

```toml
[dependencies]
clap = { version = "4.5", features = ["derive", "env"] }
tracing = "0.1"
tracing-subscriber = { version = "0.3", features = ["env-filter"] }
```

The `tracing-subscriber` crate provides the `fmt()` function, which returns a `SubscriberBuilder` instance that lets you configure logging output.

We don't need to dive into configuration details for this logger and can use the default settings by calling `init()`.

Logs will now appear in the console, so indicate the program start using the `info!` macro from the `tracing` crate.

Also, in the `Watcher` component's `run()` method, use the same macro to report which file path the task is watching.

src/main.rs RUST

```rust
fn main() {
    tracing_subscriber::fmt().init();
    tracing::info!("Starting incident narrator...");
}
```

src/watcher.rs RUST

```rust
impl Watcher {
    pub fn run(self) {
        tracing::info!("Watching log file: {}", self.log_path.display());
    }
}
```

It would also be helpful to generate a sample log file to test the program.

Add sample log entries and fictional errors to the file `sample.log`.

You can use the version provided here, or generate a new file by prompting your favorite AI code assistant.

sample.log LOG

```
2035-11-16T10:30:15Z INFO Starting application server
2035-11-16T10:30:18Z ERROR Database connection timeout
2035-11-16T10:30:24Z CRITICAL Cannot start without database
2035-11-16T10:30:26Z INFO Connected to backup database
2035-11-16T10:30:27Z INFO HTTP server started on port 8080
2035-11-16T10:30:32Z ERROR Authentication failed for user 'admin'
2035-11-16T10:30:36Z ERROR Failed to process record: missing field 'email'
2035-11-16T10:30:45Z ERROR Redis connection lost
2035-11-16T10:30:46Z WARN Falling back to direct database queries
2035-11-16T10:30:47Z FATAL Out of memory
```

Try running the program with `cargo run`, passing the name of our sample log file `sample.log` as an argument, and remember to include the double dash `--` to separate `cargo` options from our program's arguments.

After startup, you will see an informational message indicating that the `Watcher` component has started, along with the name of the file it intends to watch.

```sh
.
$ cargo run -- sample.log
```

Implement Tail Reader

Let's implement the file reading functionality now. We need to regularly read the tail of the file as new log entries are added. In other words, we need to read the new parts of the log file that we have not yet processed.

There are several ways to implement this. One approach is to subscribe to filesystem events and read changes when an event arrives. However, in our case, periodic file reading is sufficient, since we expect entries to be added to the log regularly. Moreover, this approach is often used by utilities because it is simpler to implement and more portable across different systems —there is no need to rely on a system-specific API.

Add a new `read_tail()` method where we will implement the required reading functionality.

src/watcher.rs RUST

```rust
impl Watcher {
    fn read_tail(&mut self) {
    }
}
```

For working with the file system, it is convenient to use an asynchronous approach; this keeps the code concise. This functionality is fully provided by the `tokio` crate, so add it to the project.

Also enable the `full` feature, since we will rely not only on reading files with this crate but also on running tasks in its runtime.

> *You may know that Rust has no built-in runtime—it is as lean as C. What you write is what you get, with no extra magic, so to make an application asynchronous we must explicitly add a runtime that drives async tasks.*

Cargo.toml TOML

```toml
[dependencies]
clap = { version = "4.5", features = ["derive", "env"] }
tokio = { version = "1.48", features = ["full"] }
tracing = "0.1"
tracing-subscriber = { version = "0.3", features = ["env-filter"] }
```

To read a file's tail, we first need to open it. Use the asynchronous `open()` function of the `File` type, which we must import beforehand from the `fs` module provided by the `tokio` crate.

The function takes the file path as its only parameter; for this, we will use the path previously saved in the `log_path` field, so pass a reference to this field as the argument when calling `open`.

src/watcher.rs RUST

```rust
use std::path::PathBuf;
use tokio::{
    fs::File,
};
impl Watcher {
    fn read_tail(&mut self) {
        File::open(&self.log_path);
    }
}
```

As mentioned, the `open()` function is asynchronous, so you need to use the `await` operator to get its result.

You can also do this within an asynchronous block, so add the `async` keyword to our `read_tail()` function to make it asynchronous as well.

src/watcher.rs RUST

```rust
impl Watcher {
    async fn read_tail(&mut self) {
        File::open(&self.log_path).await;
    }
}
```

Reading a file can fail, as can many other operations the program performs, so we also need a convenient error-handling mechanism. A great solution for this is to use the `anyhow` crate.

Cargo.toml TOML

```toml
[dependencies]
anyhow = "1.0"
clap = { version = "4.5", features = ["derive", "env"] }
tokio = { version = "1.48", features = ["full"] }
tracing = "0.1"
tracing-subscriber = { version = "0.3", features = ["env-filter"] }
```

Import the `Result` type alias from the `anyhow` crate and use it as the return type of the `read_tail()` function.

Also use the `?` operator to handle the result of `open()`, since it returns a file handle wrapped in `Result`; on error, the function will return early.

src/watcher.rs RUST

```rust
use std::path::PathBuf;
use anyhow::Result;
impl Watcher {
    async fn read_tail(&mut self) -> Result<()> {
        File::open(&self.log_path).await?;
        Ok(())
    }
}
```

Log file entries are generally represented as separate lines. However, we cannot read individual lines directly from a `File` object. We need a special wrapper that stores and accumulates read data in a buffer until an end of line is detected.

This functionality is already provided by the `BufReader` type, so import it from the `io` module of the `tokio` crate.

Store the result of the `open()` function in the `file` variable, and use it to create a `BufReader` instance by calling `new()` and passing the saved `file` object as the parameter. Assign the result to the `reader` variable; we will now use it for operations on the opened file.

src/watcher.rs RUST

```rust
use std::path::PathBuf;
use tokio::{
    fs::File,
    io::BufReader,
};
impl Watcher {
    async fn read_tail(&mut self) -> Result<()> {
        let file = File::open(&self.log_path).await?;
        let reader = BufReader::new(file);
        Ok(())
    }
}
```

To read only the tail of the file instead of the whole file, record the position where you last stopped and continue from there.

Add a `last_position` field of type `usize` to the struct for this purpose, and initialize it to `0` in the associated `new()` function when creating a `Watcher` instance.

> *We use* `usize` *because the read operation returns the number of bytes read as a* `usize`*, so we can add it to a field of the same type without conversion.*

src/watcher.rs RUST

```rust
pub struct Watcher {
    log_path: PathBuf,
    last_position: usize,
}
impl Watcher {
    pub fn new(log_path: PathBuf) -> Self {
        Self {
            log_path,
            last_position: 0,
        }
    }
}
```

A file offset is specified not as a simple byte count from the beginning, but via the `SeekFrom` enumeration, which lets you set the offset relative to the start, end, or current position. This enables more flexible navigation within the file. We will always track the position from the very beginning of the file to discard the part already read.

Import the `SeekFrom` enumeration from the `io` module, construct the `Start` variant with the `last_position` value to set the offset from the start of the file, and store it in the `position` variable.

Since we store the position as a `usize`, we need to convert it to `u64`, which is expected as the offset type for the enumeration in use.

src/watcher.rs RUST

```rust
use std::path::PathBuf;
use tokio::{
    fs::File,
    io::{BufReader, SeekFrom},
};
impl Watcher {
    async fn read_tail(&mut self) -> Result<()> {
        let file = File::open(&self.log_path).await?;
        let reader = BufReader::new(file);
        let position = SeekFrom::Start(self.last_position as u64);
        Ok(())
    }
}
```

Now we can use the computed offset to move within the `BufReader`.

For this, the `AsyncSeekExt` trait provides the asynchronous `seek()` method to reposition within the file and expects a `SeekFrom` value as its argument.

Because the method is asynchronous, use `await` to run it, and the `?` operator to handle any errors that may occur, for example, if the end of the file is reached while seeking.

Since the `seek()` method will modify the `BufReader`, add `mut` to the declaration of the `reader` variable.

src/watcher.rs RUST

```rust
use std::path::PathBuf;
use tokio::{
    fs::File,
    io::{AsyncSeekExt, BufReader, SeekFrom},
};
impl Watcher {
    async fn read_tail(&mut self) -> Result<()> {
        let file = File::open(&self.log_path).await?;
        let mut reader = BufReader::new(file);
        let position = SeekFrom::Start(self.last_position as u64);
        reader.seek(position).await?;
        Ok(())
    }
}
```

We plan to call the `read_tail()` method in an infinite loop to regularly read updates from the file. To avoid overloading the system, we need to add an interval to give the log file time to grow.

Create an interval constant of type `Duration`, import `Duration` from the `time` module of the `tokio` crate, and set the interval to `500` milliseconds.

> *The `500`-millisecond value might still be too small, and you may want to set it to `2` or `3` seconds.*

src/watcher.rs RUST

```rust
use std::path::PathBuf;
use tokio::{
    fs::File,
    io::{AsyncSeekExt, BufReader, SeekFrom},
    time::Duration,
};
const INTERVAL: Duration = Duration::from_millis(500);
```

Then import the `sleep()` function from the same module. Call it at the end of the method, passing the created constant as the argument; since the function is asynchronous, use `await` to pause the method for the specified number of milliseconds.

This could also be done at the end of the main loop, but in this implementation we prefer to wait for the interval only after a successful read, and on failure immediately attempt to read again.

src/watcher.rs RUST

```rust
use std::path::PathBuf;
use tokio::{
    fs::File,
    io::{AsyncSeekExt, BufReader, SeekFrom},
    time::{sleep, Duration},
};
impl Watcher {
    async fn read_tail(&mut self) -> Result<()> {
        let file = File::open(&self.log_path).await?;
        let mut reader = BufReader::new(file);
        let position = SeekFrom::Start(self.last_position as u64);
        reader.seek(position).await?;
        sleep(INTERVAL).await;
        Ok(())
    }
}
```

Integrate Async Watcher

At this point, the `read_tail()` method simply opens the file and moves the read position to the start of the unread section. We also need an additional method to process the remaining tail of the file. Add a new `process_lines` method for this task.

Because the read operation is not guaranteed to succeed, you can use `Result` as the return type to handle all errors conveniently later.

src/watcher.rs RUST

```rust
impl Watcher {
    async fn process_lines(&mut self) -> Result<()> {
        Ok(())
    }
}
```

Since we need to read the tail of the file, we can keep using the `BufReader` we have advanced to the required position, so just pass a mutable reference to it as a parameter to the new method.

Call this method from `read_tail()` right after calling `seek()`, but before we start waiting with the `sleep()` function. And remember that the new method is asynchronous and may return an error.

src/watcher.rs RUST

```rust
impl Watcher {
    async fn read_tail(&mut self) -> Result<()> {
        let file = File::open(&self.log_path).await?;
        let mut reader = BufReader::new(file);
        let position = SeekFrom::Start(self.last_position as u64);
        reader.seek(position).await?;
        self.process_lines(&mut reader).await?;
        sleep(INTERVAL).await;
        Ok(())
    }
    async fn process_lines(&mut self, reader: &mut BufReader<File>) -> Result<()> {
        Ok(())
    }
}
```

Earlier we discussed creating a general infinite file-reading loop that calls the `read_tail()` method. Add a `loop` to the `run` method and call the method from it.

Since the method will update the offset value, mark `self` as `mut`.

src/watcher.rs RUST

```rust
impl Watcher {
    pub fn run(mut self) {
        tracing::info!("Watching log file: {}", self.log_path.display());
        loop {
            self.read_tail();
        }
    }
}
```

The method invoked in the `loop` returns a `Result` and is asynchronous, which directly affects the `run()` method as well. Therefore, make it asynchronous too and return a `Result` to meet the same criteria.

src/watcher.rs RUST

```rust
impl Watcher {
    pub async fn run(mut self) -> Result<()> {
        tracing::info!("Watching log file: {}", self.log_path.display());
        loop {
            self.read_tail().await?;
        }
    }
}
```

But that's not all. Although we called the `run()` method, it is now asynchronous. Therefore, we also need to make the entire `main()` function asynchronous.

The easiest way to do this is to use the `main` macro from the `tokio` crate, which will start the runtime and call our asynchronous version of the function.

Just remember to add the `async` keyword to its declaration and use `await` to run the asynchronous `run()` method.

src/main.rs RUST

```rust
#[tokio::main]
async fn main() {
    let watcher = Watcher::new(args.log_file);
    watcher.run().await;
}
```

You also need to handle any error that may occur while running `Watcher`. In this implementation, we do not plan to retry reading the file if an error occurs during reading, so simply aborting execution is sufficient.

You can do this by handling the result returned by the `run()` method. Use the `?` operator, first importing the `Result` alias from the `anyhow` crate and returning it from the `main()` function. At the end of the function, return `Ok(())` if the read task completes successfully.

> *As you remember, our* `Watcher` *has an infinite loop, and the program will not actually exit until it is terminated entirely, but we will improve this a bit later.*

src/main.rs RUST

```rust
use anyhow::Result;
#[tokio::main]
async fn main() -> Result<()> {
    let watcher = Watcher::new(args.log_file);
    watcher.run().await?;
    Ok(())
}
```

Implement Line Processing

Let's return to implementing the `process_lines()` method, which is responsible for reading the tail of the log file.

The first step is to read the next line from the provided `BufReader`. As you recall, we wrapped the file in a `BufReader` precisely because it provides the `read_line()` method for line-by-line reading.

In fact, this method is provided by the `AsyncBufReadExt` trait, which should be imported from the `io` module of the `tokio` crate. This trait is implemented by the `BufReader` type, whose instance we stored in the `reader` variable.

The method takes a mutable reference to a buffer as its argument, where the line will be written. Create a new one of type `String` by calling its `new()` method; this produces an empty string into which `read_line()` will read the next line from the file.

src/watcher.rs RUST

```rust
use std::path::PathBuf;
use tokio::{
    fs::File,
    io::{AsyncBufReadExt, AsyncSeekExt, BufReader, SeekFrom},
    time::{sleep, Duration},
};
impl Watcher {
    async fn process_lines(&mut self, reader: &mut BufReader<File>) -> Result<()> {
        let mut line = String::new();
        reader.read_line(&mut line).await?;
        Ok(())
    }
}
```

We read one line once; to read all the remaining lines, we need to repeat the same operation multiple times.

To do this, move the line reading (the `read_line()` call) into a `loop`. To avoid creating a buffer on each iteration of the loop, reuse the existing one by clearing it to an empty string with the `clear()` method.

src/watcher.rs RUST

```rust
impl Watcher {
    async fn process_lines(&mut self, reader: &mut BufReader<File>) -> Result<()> {
        let mut line = String::new();
        loop {
            line.clear();
            reader.read_line(&mut line).await?;
        }
        Ok(())
    }
}
```

The loop will run indefinitely until a read error occurs; the compiler will even warn that the `Ok(())` result at the end of the `process_lines()` method will never be reached.

However, the `read_line()` method returns the number of bytes read, and we can stop further reading when no new data is read (it returns `0`).

Store the number of bytes read in the `bytes_read` variable, and if the value is `0`, break the loop using the `break` statement.

src/watcher.rs RUST

```rust
impl Watcher {
    async fn process_lines(&mut self, reader: &mut BufReader<File>) -> Result<()> {
        let mut line = String::new();
        loop {
            line.clear();
            let bytes_read = reader.read_line(&mut line).await?;
            if bytes_read == 0 {
                break;
            }
        }
        Ok(())
    }
}
```

But, as you recall, we also store the number of bytes read from the file in the `last_position` field.

It is time to update it—after reading a line, add the number of bytes read to this field, so that on the next iteration we start reading from the next line, since the read cursor will be moved using `seek()` before this method is called.

src/watcher.rs RUST

```rust
impl Watcher {
    async fn process_lines(&mut self, reader: &mut BufReader<File>) -> Result<()> {
        let mut line = String::new();
        loop {
            line.clear();
            let bytes_read = reader.read_line(&mut line).await?;
            if bytes_read == 0 {
                break;
            }
            self.last_position += bytes_read;
        }
        Ok(())
    }
}
```

Filter And Route Lines

We have read a line, and it is now stored in a buffer. What should we do with this line next? Obviously, the first step is to filter the lines: separate entries with errors and warnings, and discard the rest.

We can do this by comparing the line to a pattern; the simplest yet most flexible approach is to use *regular expressions*. This functionality is not included in the standard library, but is fully implemented by the `regex` crate—add it to the project.

Cargo.toml TOML

```toml
[dependencies]
anyhow = "1.0"
clap = { version = "4.5", features = ["derive", "env"] }
regex = "1.12"
tokio = { version = "1.48", features = ["full"] }
tracing = "0.1"
tracing-subscriber = { version = "0.3", features = ["env-filter"] }
```

Let's define a regular expression pattern that can detect lines with errors and warnings. For now, it can just be a string literal. Below is a detailed description of each part of the regular expression that detects any of the listed words:

Part	Description
`r"..."`	A raw string in Rust. This means backslashes (`\`) are not interpreted by Rust as escape sequences but are passed directly to the regular expression.
`(...)`	Capturing group. This parenthesized group combines the listed words.
`error`	Searches for the exact sequence of characters "error".
`warning`	Searches for "warning".
`warn`	Searches for "warn".
`fatal`	Searches for "fatal".
`critical`	Searches for "critical".
`|`	Alternation (OR).

src/watcher.rs RUST

```rust
const INTERVAL: Duration = Duration::from_millis(500);
const PATTERN: &str = r"(error|warning|warn|fatal|critical)";
```

To make the pattern more versatile, add case-insensitive matching and mark word boundaries by including the following elements in the pattern:

Part	Description
`(?i)`	A flag enabling case-insensitive matching. This means the expression will match error, `ERROR`, `Error`, `eRrOr`, etc.
`\b`	Word boundary. A zero-width assertion that matches the position between a word character (w, i.e., a letter, digit, or underscore) and a non-word character (W), or the start/end of the string. This ensures the match is a whole word, not part of another word (for example, it finds warn but not beware or warningly).
`\b`	Word boundary. Closes the pattern by ensuring that a non-word character or the end of the string follows the matched word.

src/watcher.rs RUST

```rust
const INTERVAL: Duration = Duration::from_millis(500);
const PATTERN: &str = r"(?i)\b(error|warning|warn|fatal|critical)\b";
```

Now we can construct a `Regex` instance from the generated pattern.

Import the `Regex` type from the `regex` crate and use the `new()` method, which takes the pattern as its only parameter; pass it the name of the constant we declared earlier. Store the result in the `regex` variable.

src/watcher.rs RUST

```rust
use std::path::PathBuf;
use regex::Regex;
impl Watcher {
    pub fn new(log_path: PathBuf) -> Self {
        let regex = Regex::new(PATTERN);
        Self {
            log_path,
            last_position: 0,
        }
    }
}
```

The `regex` variable indeed stores a value of type `Result`. Since we defined the pattern in code, it is enough to verify once that it compiles and then use `unwrap()` to discard the `Result` wrapper.

However, it is still good practice to check the result, since you might change the pattern from time to time, or it may even be provided by the user.

To do this, wrap the return value of the `new()` constructor in a `Result`, wrap the constructed `Self` instance in the successful `Ok` variant, and handle the result of constructing the regular expression with the `?` operator.

src/main.rs RUST

```rust
#[tokio::main]
async fn main() -> Result<()> {
    let watcher = Watcher::new(args.log_file)?;
    watcher.run().await?;
    Ok(())
}
```

src/watcher.rs RUST

```rust
impl Watcher {
    pub fn new(log_path: PathBuf) -> Result<Self> {
        let regex = Regex::new(PATTERN);
        Ok(Self {
            log_path,
            last_position: 0,
        })
    }
}
```

Move the constructed regular expression into a field of the `Watcher` struct.

To do this, add a new `regex` field of type `Regex`, and initialize it in the `new()` function by setting its value to the constructed regular expression stored in the `regex` variable.

src/watcher.rs RUST

```rust
pub struct Watcher {
    log_path: PathBuf,
    last_position: usize,
    regex: Regex,
}
impl Watcher {
    pub fn new(log_path: PathBuf) -> Result<Self> {
        let regex = Regex::new(PATTERN)?;
        Ok(Self {
            log_path,
            last_position: 0,
            regex,
        })
    }
}
```

Use the constructed regular expression in the `process_line()` method to determine whether the read line matches it.

For this purpose, the `Regex` type provides the `is_match()` method, which takes a string parameter to check whether it contains a match for the regular expression. Add an `if` block that checks this condition.

src/watcher.rs RUST

```rust
impl Watcher {
    async fn process_lines(&mut self, reader: &mut BufReader<File>) -> Result<()> {
        let mut line = String::new();
        loop {
            line.clear();
            let bytes_read = reader.read_line(&mut line).await?;
            if bytes_read == 0 {
                break;
            }
            self.last_position += bytes_read;
            if self.regex.is_match(&line) {
            }
        }
        Ok(())
    }
}
```

We have built a mechanism that detects lines with errors or warnings. Now we need to pass them to a component that can process them into a summary.

This should be a separate entity because we don't want to interrupt reading the update file while a request to the AI model is in progress.

For this purpose, the `mpsc` channel from the `tokio` crate is a perfect fit. Import it from the `sync` module.

And add an `UnboundedSender` as a parameter to the `new()` function, which will be used to send log lines.

src/watcher.rs RUST

```rust
use std::path::PathBuf;
use tokio::{
    fs::File,
    io::{AsyncBufReadExt, AsyncSeekExt, BufReader, SeekFrom},
    sync::mpsc,
    time::{sleep, Duration},
};
impl Watcher {
    pub fn new(log_path: PathBuf, tx: mpsc::UnboundedSender<String>) -> Result<Self> {
        let regex = Regex::new(PATTERN)?;
        Ok(Self {
            log_path,
            last_position: 0,
            regex,
        })
    }
}
```

Let's create a channel instance right away for `Watcher` to send the lines it reads.

Import the `mpsc` module and use its `unbounded_channel()` function, which returns a tuple with `UnboundedSender` and `UnboundedReceiver`.

Destructure the tuple into `log_tx` and `log_rx`, and pass the `UnboundedSender` stored in `log_tx` as the additional parameter to the `new()` function we just added.

src/main.rs RUST

```rust
use tokio::sync::mpsc;
#[tokio::main]
async fn main() -> Result<()> {
    let (log_tx, log_rx) = mpsc::unbounded_channel();
    let watcher = Watcher::new(args.log_file, log_tx)?;
    watcher.run().await?;
    Ok(())
}
```

Store the passed `UnboundedSender` in the `Watcher` struct by adding a `tx` field of the appropriate type and initializing it when creating the instance inside the `new()` function.

src/watcher.rs RUST

```rust
pub struct Watcher {
    log_path: PathBuf,
    last_position: usize,
    regex: Regex,
    tx: mpsc::UnboundedSender<String>,
}
impl Watcher {
    pub fn new(log_path: PathBuf, tx: mpsc::UnboundedSender<String>) -> Result<Self> {
        let regex = Regex::new(PATTERN)?;
        Ok(Self {
            log_path,
            last_position: 0,
            regex,
            tx,
        })
    }
}
```

We can now finish implementing the `process_lines()` method by using the channel sender stored in the struct's `tx` field.

If the read log entry matches our pattern (`if` block), call the sender's `send()` method, passing it the cloned string.

> *We clone the string because the `String` in `line` is reused as a buffer, which is efficient since not all lines need to be sent; therefore, we clone only the lines forwarded for processing, and we do not need to create a new string for reading (its capacity will grow automatically, and only when the next line is larger than the already allocated memory that could hold any of the previous lines).*

src/watcher.rs RUST

```rust
impl Watcher {
    async fn process_lines(&mut self, reader: &mut BufReader<File>) -> Result<()> {
        let mut line = String::new();
        loop {
            line.clear();
            let bytes_read = reader.read_line(&mut line).await?;
            if bytes_read == 0 {
                break;
            }
            self.last_position += bytes_read;
            if self.regex.is_match(&line) {
                self.tx.send(line.clone())?;
            }
        }
        Ok(())
    }
}
```

Generate Log Summaries

Initialize Log Summarizer

We will now create an asynchronous task that processes lines with errors and warnings, turning them into a summary.

Add the `summarizer` module to `main.rs` and create the corresponding empty module file.

Then add the `Summarizer` struct to this new file.

src/main.rs RUST

```rust
mod summarizer;
mod watcher;
```

src/summarizer.rs RUST

```rust
pub struct Summarizer {
}
```

Since we previously created a channel for sending messages for processing, add an `UnboundedReceiver` to our new structure to receive messages from this channel. Save it as the structure's `rx` field.

And of course, import the `mpsc` module from the `tokio` crate's `sync` module, which contains the type we need.

src/summarizer.rs RUST

```rust
use tokio::{
    sync::mpsc,
};
pub struct Summarizer {
    rx: mpsc::UnboundedReceiver<String>,
}
```

Add an implementation block for the `Summarizer` struct and declare the associated `new()` function in it to create an instance of the struct.

src/summarizer.rs RUST

```rust
impl Summarizer {
    pub fn new(
    ) {
    }
}
```

The method should take an `UnboundedReceiver` as a parameter and return an instance of the created structure; we can use the `Self` alias for this.

And in the function body, we should create a `Summarizer` instance, filling its single `rx` field with the passed value.

src/summarizer.rs RUST

```rust
impl Summarizer {
    pub fn new(
        rx: mpsc::UnboundedReceiver<String>,
    ) -> Self {
        Self {
            rx,
        }
    }
}
```

Now add an asynchronous `run()` method to the struct that will pull messages from the channel and process them.

It is also good practice to return a `Result` from the `anyhow` crate and return `Ok` on successful completion of the method. Using `Result` makes error handling convenient and is useful for entry points of asynchronous tasks.

src/summarizer.rs RUST

```rust
use anyhow::Result;
impl Summarizer {
    pub async fn run(mut self) -> Result<()> {
        Ok(())
    }
}
```

As you recall, we used the `tracing` crate to indicate the startup of the `Watcher` component. We can do the same for the new `Summarizer`.

Use the `info!` macro to report the start of the execution of the `run()` method. This way you will see that the message processing task has started successfully.

src/summarizer.rs RUST

```rust
impl Summarizer {
    pub async fn run(mut self) -> Result<()> {
        tracing::info!("Starting summarizer");
        Ok(())
    }
}
```

Run Tasks Concurrently

Since we now have two asynchronous tasks, `Watcher` and `Summarizer`, we need to run them together. The `futures` crate can help with this, so add it to your dependencies.

Cargo.toml TOML

```toml
[dependencies]
anyhow = "1.0"
clap = { version = "4.5", features = ["derive", "env"] }
futures = "0.3.31"
regex = "1.12"
tokio = { version = "1.48", features = ["full"] }
tracing = "0.1"
tracing-subscriber = { version = "0.3", features = ["env-filter"] }
```

Because Rust is strongly typed, to run multiple different tasks we need to bring them to a single type.

For an asynchronous task, we can use the `boxed()` method of the `FutureExt` trait from the `future` module of the `futures` crate. This method boxes the asynchronous task into a `Box` (i.e., allocates it on the heap), producing a value that implements the `Future` trait.

src/main.rs RUST

```rust
use futures::future::FutureExt;
#[tokio::main]
async fn main() -> Result<()> {
    let watcher = Watcher::new(args.log_file, log_tx)?;
    watcher.run().boxed().await?;
    Ok(())
}
```

To run a group of tasks concurrently, we need to collect them into a `Vec`. Create a new mutable vector `tasks` and add the boxed asynchronous task `Watcher` to this list.

By the way, you no longer need to wait for the task to complete with `await`, since we will do that next for the entire group of tasks at once.

src/main.rs RUST

```rust
#[tokio::main]
async fn main() -> Result<()> {
    let mut tasks = Vec::new();
    let watcher = Watcher::new(args.log_file, log_tx)?;
    tasks.push(watcher.run().boxed());
    Ok(())
}
```

Even though there is only one task in the list for now, let's execute the task list right away.

Import the `select_all()` function from the `future` module. Call it and pass our `tasks` list as the parameter. And since the function is asynchronous, apply `await` to execute it.

The `select_all()` function returns a tuple whose first element (index `0`) is the result of the earliest completed asynchronous task, and whose second element is all unfinished tasks that can continue running.

Since all our asynchronous tasks return a `Result`, we can immediately access index `0`—the result of the first completed task—and use the `?` operator to handle an error if the asynchronous task completed with one.

src/main.rs RUST

```rust
use futures::future::{select_all, FutureExt};
#[tokio::main]
async fn main() -> Result<()> {
    let mut tasks = Vec::new();
    select_all(tasks).await.0?;
    Ok(())
}
```

Now let's create an instance of the `Summarizer` task. Import this struct from the `summarizer` module and use the associated `new()` function, passing the `UnboundedReceiver` stored in the `log_rx` variable as a parameter so it can receive log file entries.

src/main.rs RUST

```rust
use summarizer::Summarizer;
#[tokio::main]
async fn main() -> Result<()> {
    let mut tasks = Vec::new();
    let summarizer = Summarizer::new(log_rx);
    select_all(tasks).await.0?;
    Ok(())
}
```

Call the `run()` method on the created structure to turn it into an asynchronous task. Wrap it in a `Box` using the `boxed()` method, and add it to the `tasks` list by calling its `push()` method.

As a result, the list contains two tasks, and both will run concurrently.

src/main.rs RUST

```rust
#[tokio::main]
async fn main() -> Result<()> {
    let mut tasks = Vec::new();
    let summarizer = Summarizer::new(log_rx);
    tasks.push(summarizer.run().boxed());
    select_all(tasks).await.0?;
    Ok(())
}
```

Handle Interrupt Signal

Let's also explicitly intercept the `SIGINT` interrupt signal, which can be sent from the terminal to the process with `Ctrl+C`. This will allow us to gracefully finish the `select_all()` call and shut down the program.

To do this, import the `ctrl_c()` function from the `signal` module of the `tokio` crate. Then call it to create an async future that listens for the interrupt signal, and store it in the `interrupt` variable.

src/main.rs RUST

```rust
use tokio::{signal::ctrl_c, sync::mpsc};
#[tokio::main]
async fn main() -> Result<()> {
    let mut tasks = Vec::new();
    let interrupt = ctrl_c();
    select_all(tasks).await.0?;
    Ok(())
}
```

Add the created task to the `tasks` list by first calling its `boxed()` method to convert it to the common type stored in the list: a `Future` inside a `Box`.

However, this *code will not compile* because the output type of this `Future` still differs from the others in our task list.

src/main.rs RUST

```rust
#[tokio::main]
async fn main() -> Result<()> {
    let mut tasks = Vec::new();
    let interrupt = ctrl_c();
    tasks.push(interrupt.boxed());
    select_all(tasks).await.0?;
    Ok(())
}
```

The mismatch is that our code returns `Error` from the `anyhow` crate, while `ctrl_c()` returns `std::io::Error`.

We can resolve this by explicitly importing `Error` from `anyhow` and using the `map_err()` method provided by the `TryFutureExt` trait, which should be imported from the `future` module.

This trait is a convenient layer for working with `Future` objects that return `Result` (specified via the `Future` trait's associated `Output` type).

The `map_err()` method takes a function that converts one error type to another. Therefore, we can directly use the `from()` function on the `Error` type from the `anyhow` crate.

Since the returned types are now fully compatible, the code compiles successfully.

src/main.rs RUST

```rust
use anyhow::{Error, Result};
use futures::future::{select_all, FutureExt, TryFutureExt};
#[tokio::main]
async fn main() -> Result<()> {
    let mut tasks = Vec::new();
    let interrupt = ctrl_c().map_err(Error::from);
    tasks.push(interrupt.boxed());
    select_all(tasks).await.0?;
    Ok(())
}
```

Handle Summarizer Events

Let's return to the `Summarizer` implementation and add an asynchronous `step()` method to it.

In that method, we will process one incoming event at a time and later call it in a loop inside the `run()` method.

src/summarizer.rs RUST

```rust
impl Summarizer {
    async fn step(&mut self) {
    }
}
```

In the `step()` method, we need to receive a message from an `UnboundedReceiver` instance; you can do this with the `recv()` method.

The method is asynchronous, so if we wait for it with `await`, we lose the ability to react to other events.

A better approach here is to use the `select!` macro, which allows us to handle multiple event streams. Import this macro from the `tokio` crate.

Use the macro to add a branch that reads messages from the receiver, and store the read message in the `message` variable.

src/summarizer.rs RUST

```rust
use tokio::{
    select,
    sync::mpsc,
};
impl Summarizer {
    async fn step(&mut self) {
        select! {
            message = self.rx.recv() => {
            }
        }
    }
}
```

You can process incoming messages directly in the `step()` method, but a cleaner approach is to define a dedicated method for each event type.

Therefore, add a `handle_message()` method that processes log messages forwarded by the `Watcher` worker on the exchange channel.

This method should take the message as a parameter (type `String`, the type we send through the channel).

However, because the channel may close, `recv()` returns an `Option` where `None` signals the end of transmission.

We must account for this to interrupt processing, so the parameter should be an `Option<String>`.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn handle_message(&mut self, message: Option<String>) {
    }
}
```

Call the new `handle_message()` method in the `select!` branch, passing the message received from the channel as its argument.

Also use `await` when calling `step()` because it is asynchronous and uses `select!` internally.

src/summarizer.rs RUST

```rust
impl Summarizer {
    pub async fn run(mut self) -> Result<()> {
        tracing::info!("Starting summarizer");
        self.step().await;
        Ok(())
    }
    async fn step(&mut self) {
        select! {
            message = self.rx.recv() => {
                self.handle_message(message);
            }
        }
    }
}
```

Let's now add a flag that will control the lifetime of the entire `Summarizer`.

Add an `active` flag of type `bool` to the struct, and initialize it to `true` in the `new()` function.

src/summarizer.rs RUST

```rust
pub struct Summarizer {
    rx: mpsc::UnboundedReceiver<String>,
    active: bool,
}
impl Summarizer {
    pub fn new(
        rx: mpsc::UnboundedReceiver<String>,
    ) -> Self {
        Self {
            rx,
            active: true,
        }
    }
}
```

Now add a `while` loop to the `run()` method that calls `step()` in its body.

The loop's exit condition will be checking the `active` flag, which is `true` at startup.

src/summarizer.rs RUST

```rust
impl Summarizer {
    pub async fn run(mut self) -> Result<()> {
        tracing::info!("Starting summarizer");
        while self.active {
            self.step().await;
        }
        Ok(())
    }
}
```

In the `handle_message()` method implementation, we first check whether a message actually arrived—that is, the `Option` variant is `Some`—and then extract it for processing using the `if let` construct.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn handle_message(&mut self, message: Option<String>) {
        if let Some(msg) = message {
        }
    }
}
```

Conversely, when the incoming message is `None`, we set the `active` flag to `false`. This will stop the main loop in the `run()` method.

That is, after we receive the last message from the log, the next loop iteration will check this flag and terminate execution.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn handle_message(&mut self, message: Option<String>) {
        if let Some(msg) = message {
        } else {
            self.active = false;
        }
    }
}
```

Implement Message Buffer

Now we need to accumulate messages coming from logs to build a summary not for a single message—since that makes no sense—but for an entire group.

At the same time, we need to implement a buffer that stores two categories of messages: those we have never processed and those previously processed.

We will use processed messages to remind the model of the overall context of all issues. And new messages will let us report specifically on new problems in the logs of the monitored application.

For this purpose, let's create a separate `buffer` module and declare the `MessageBuffer` struct in it.

src/buffer.rs RUST

```rust
pub struct MessageBuffer {
}
```

src/main.rs RUST

```rust
mod buffer;
mod summarizer;
mod watcher;
```

Add a `new()` function to the implementation of the `MessageBuffer` struct.

The `new()` method will return instances of the struct, and you can create it by constructing an empty struct using the `Self` alias.

src/buffer.rs RUST

```rust
impl MessageBuffer {
    pub fn new() -> Self {
        Self {
        }
    }
}
```

Although `MessageBuffer` does not yet provide useful functionality, we can already add it to our `Summarizer` so we can later decide which functionality (methods) we actually need.

Bring the `MessageBuffer` type into scope from the crate's `buffer` module, and add a `buffer` field of this type to the struct.

Also, in the `new()` function when creating the `Summarizer` struct, initialize this field by calling the `MessageBuffer` struct's `new()` method with no arguments.

src/summarizer.rs RUST

```rust
use crate::buffer::MessageBuffer;
pub struct Summarizer {
    rx: mpsc::UnboundedReceiver<String>,
    active: bool,
    buffer: MessageBuffer,
}
impl Summarizer {
    pub fn new(
        rx: mpsc::UnboundedReceiver<String>,
    ) -> Self {
        Self {
            rx,
            active: true,
            buffer: MessageBuffer::new(),
        }
    }
}
```

We will definitely need a `MessageBuffer` method to add a new message to the buffer.

Create a method `add_new()` for this purpose that takes a `String` message as its parameter.

src/buffer.rs RUST

```rust
impl MessageBuffer {
    pub fn add_new(&mut self, message: String) {
    }
}
```

An incoming message needs to be stored somewhere. For this purpose, add a `new_messages` field to the buffer struct, of type `Vec` containing `String` objects.

Initialize this field with an empty vector by calling its `new()` method in the `MessageBuffer` constructor.

And immediately add the incoming message in the `add_new()` method to the `new_messages` vector, using its `push()` method to append the message to the end of the list.

src/buffer.rs RUST

```rust
pub struct MessageBuffer {
    new_messages: Vec<String>,
}
impl MessageBuffer {
    pub fn new() -> Self {
        Self {
            new_messages: Vec::new(),
        }
    }
    pub fn add_new(&mut self, message: String) {
        self.new_messages.push(message);
    }
}
```

Immediately use the buffer's new `add_new()` method to add the incoming message within the `handle_message()` method.

Note that we did not limit the size of the new-messages buffer, so it will grow indefinitely, accumulating more and more messages.

To avoid overflow, we need to periodically process messages from this buffer.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn handle_message(&mut self, message: Option<String>) {
        if let Some(msg) = message {
            self.buffer.add_new(msg);
        } else {
            self.active = false;
        }
    }
}
```

Let's add a `Duration` constant `INTERVAL` that we'll use as the interval for sending the buffer.

Import the `Duration` type from the `time` module of the `tokio` crate.

Use the `from_secs()` function to set the constant's value. The function takes a number of seconds, so pass `5` as the argument; we will process records every five seconds.

src/summarizer.rs RUST

```rust
use tokio::{
    select,
    sync::mpsc,
    time::Duration,
};
const INTERVAL: Duration = Duration::from_secs(5);
```

Import from the `time` module the `Interval` type, a struct that measures time intervals and asynchronously waits for them to complete. Also import the `interval()` function that creates an instance of this type.

Add an `interval` field of type `Interval` to the struct, and initialize it by calling `interval()` with a `Duration` parameter—the interval length—using the constant we defined earlier as the value.

src/summarizer.rs RUST

```rust
use tokio::{
    select,
    sync::mpsc,
    time::{interval, Duration, Interval},
};
pub struct Summarizer {
    rx: mpsc::UnboundedReceiver<String>,
    active: bool,
    buffer: MessageBuffer,
    interval: Interval,
}
impl Summarizer {
    pub fn new(
        rx: mpsc::UnboundedReceiver<String>,
    ) -> Self {
        Self {
            rx,
            active: true,
            buffer: MessageBuffer::new(),
            interval: interval(INTERVAL),
        }
    }
}
```

In the `step()` method within the `select!` macro, add another branch that uses our `interval` field of type `Interval`, calling its `tick()` method.

The method waits for the interval to elapse. The countdown does not restart even if we did not wait for completion and called `tick()` again. This allows us to handle log events and react to the interval expiring simultaneously.

src/summarizer.rs — RUST

```rust
impl Summarizer {
    async fn step(&mut self) {
        select! {
            message = self.rx.recv() => {
                self.handle_message(message);
            }
            _ = self.interval.tick() => {
            }
        }
    }
}
```

To handle interval events, add a `handle_tick()` method and call it from the newly created branch in the `select!` macro.

In this method, we plan to check for new messages and send them for processing.

src/summarizer.rs RUST

```rust
impl Summarizer {
    async fn step(&mut self) {
        select! {
            message = self.rx.recv() => {
                self.handle_message(message);
            }
            _ = self.interval.tick() => {
                self.handle_tick();
            }
        }
    }
    fn handle_tick(&mut self) {
    }
}
```

To check for new messages in `MessageBuffer`, add a `has_new()` method that returns `bool`, so it can be used in `if` blocks.

In its implementation, call the `is_empty()` method provided by the `Vec` type; in our case, this is the `new_messages` field that holds new messages.

To know whether the buffer of new messages is not empty, add the negation operator `!` to the value returned by `is_empty()`.

src/buffer.rs RUST

```rust
impl MessageBuffer {
    pub fn has_new(&self) -> bool {
        !self.new_messages.is_empty()
    }
}
```

Use the newly added `has_new()` method on `MessageBuffer` within `handle_tick()` to run code when new messages arrive.

Add an `if` block that wraps the summary-generation block when the buffer has new messages—that is, when `has_new()` returns `true`.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn handle_tick(&mut self) {
        if self.buffer.has_new() {
        }
    }
}
```

Extend Summarization Buffer

To generate a short message that describes the changes and errors mentioned in new log messages, add the `generate_summary()` method.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn generate_summary(&mut self) {
    }
}
```

In the `handle_tick()` method, call `generate_summary()` if there are new messages in the buffer.

It's also helpful to indicate in the logs that you are starting to generate the summary, so add an appropriate message using the `info!` macro from the `tracing` crate.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn handle_tick(&mut self) {
        if self.buffer.has_new() {
            tracing::info!("Sending logs for summarization...");
            self.generate_summary();
        }
    }
}
```

Now we need to add the `build_messages()` method, which will be responsible for constructing messages for the LLM.

In this case, these are different messages (not just from logs), and they are needed to prepare the context and obtain a response from the AI model.

You can also call the added `build_messages()` method directly from our `generate_summary()` method, thereby preparing the message for sending.

A separate method is needed here because LLM messages can have different, sometimes verbose types, and this is a fairly heavy structure that is convenient to isolate in its own method.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn generate_summary(&mut self) {
        self.build_messages();
    }
    fn build_messages(&mut self) {
    }
}
```

To build context, it helps to send not only new messages but also include a previously processed message in the context, making it easier for the model to understand what has been happening in the logs overall.

For such a buffer, we need to know its size—how many messages we want to store. So add a `DEPTH` constant of type `usize`.

The `usize` type is used to represent sizes in the system. Assign it a value; for example, start with `1024`. This is a fairly large window of log messages, but since these are prior logs, it will not make the summary large.

src/buffer.rs RUST

```rust
const DEPTH: usize = 1024;
```

Now let's add another collection to our `MessageBuffer` for processed messages.

A regular vector won't do; we should use `VecDeque` to efficiently implement a sliding window. We don't want to store all messages, only a limited number of them.

Import this collection from the standard library's `collections` module and add a `processed_messages` field of this type, containing strings, to the `MessageBuffer` struct.

`VecDeque` itself is not size-limited, but we can enforce this later by adding an explicit check that drops extra messages on insertion.

Also, in the `new()` function, include the `processed_messages` field when constructing the instance.

For its value, create the collection with `with_capacity()`, passing our constant as the parameter. This will preallocate space for that number of elements.

src/buffer.rs RUST

```rust
use std::collections::VecDeque;
pub struct MessageBuffer {
    new_messages: Vec<String>,
    processed_messages: VecDeque<String>,
}
impl MessageBuffer {
    pub fn new() -> Self {
        Self {
            new_messages: Vec::new(),
            processed_messages: VecDeque::with_capacity(DEPTH),
        }
    }
}
```

Add the `get_messages()` method, which we will use to retrieve slices of processed and new messages.

src/buffer.rs RUST

```rust
impl MessageBuffer {
    pub fn get_messages(&mut self) {
    }
}
```

Since `VecDeque` with processed messages (the `processed_messages` field) allows adding elements at both ends, to obtain a slice from such a collection you must call the `make_contiguous()` method, which turns the collection into a contiguous sequence.

This lets you get a slice, which is exactly what is returned as the result. Call this method and store the resulting slice in the `processed` variable.

src/buffer.rs RUST

```rust
impl MessageBuffer {
    pub fn get_messages(&mut self) {
        let processed = self.processed_messages.make_contiguous();
    }
}
```

For new messages, since this is a regular vector, you can simply take a slice with the `&` operator and assign it to the `new` variable.

src/buffer.rs RUST

```rust
impl MessageBuffer {
    pub fn get_messages(&mut self) {
        let processed =
self.processed_messages.make_contiguous();
        let new = &self.new_messages;
    }
}
```

We now have both slices for the processed and new messages, and we can return them as a pair by setting the method's return value accordingly.

That is, a tuple containing slices of strings—slices of log file entries (slice type `&[String]`).

src/buffer.rs RUST

```rust
impl MessageBuffer {
    pub fn get_messages(&mut self) -> (&[String], &[String]) {
        let processed = self.processed_messages.make_contiguous();
        let new = &self.new_messages;
        (processed, new)
    }
}
```

Construct Summary Messages

We will continue implementing the `build_messages()` method at the point where we called `get_messages()` to obtain two slices: decompose the resulting tuple into two variables, `processed` and `new`.

These contain, respectively, previously processed messages for which we already obtained a summary, and new messages whose processing should be prioritized by the model.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn build_messages(&mut self) {
        let (processed, new) = self.buffer.get_messages();
    }
}
```

To gauge how much information is being sent to generate the summary, you can add a debug message with the number of new and processed messages using the `debug!` macro from the `tracing` crate.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn build_messages(&mut self) {
        tracing::debug!(
            "Generating summary for {} new messages (context: {} processed)",
            new.len(),
            processed.len()
        );
    }
}
```

Build part of the context from the processed messages. Add an `if` block for this purpose in the `build_messages()` method.

As the condition of this block, check whether there are any messages in the `processed` slice using the `is_empty()` method.

Assign the result of the entire `if` block to the `processed_context` variable, and then we will construct the corresponding parts of the context in each branch of the conditional.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn build_messages(&mut self) {
        let processed_context = if processed.is_empty() {
        } else {
        };
    }
}
```

If the queue of previous messages is empty, we can simply create the string "No previous errors" by calling `to_string()`, which converts a string literal into a `String`, since that is exactly what the `format!` macro used in the other branch returns.

If we already have processed messages, we can use the slice method `join()` to combine the entries into text joined by the delimiter passed as the method's argument. In our case, this will be `\n`.

And with the `format!` macro, we will create a complete description by adding the prefix *"Previous errors (for context only)"* to indicate that this part consists of messages that have already been processed.

Use the `{}` placeholder to append the tail of text with the processed messages, separated by the delimiter.

Now the `proset_ontext` variable holds the message context, whose value depends on whether we have processed messages or not.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn build_messages(&mut self) {
        let processed_context = if processed.is_empty() {
            "No previous errors.".to_string()
        } else {
            format!(
                "Previous errors (for context only):\n{}",
                processed.join("\n")
            )
        };
    }
}
```

New log entries will always be present, since we already verified that this part of the buffer is not empty.

So also use the `format!` macro to create the second part of the context by adding text that indicates the intent *"New errors (focus on this)"* .

Also use the slice's `join()` method, passing a newline separator to gather all new messages into a single block of text.

Use the `{}` placeholder to insert the collected messages at the end of the formatted string.

And store the entire result in the `new_errors` variable.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn build_messages(&mut self) {
        let new_errors = format!("New errors (focus on these):\n{}", new.join("\n"));
    }
}
```

Now it's time to construct well-formed messages to send to the language model.

Import the `async-openai` crate by adding it to the `dependencies` section of the `Cargo.toml` configuration file.

It provides an OpenAI API client along with the necessary types.

Since functionality in different parts is hidden behind feature gates, you can simply enable them all with the `full` feature.

Cargo.toml TOML

```toml
[dependencies]
anyhow = "1.0"
async-openai = { version = "0.31.1", features = ["full"] }
clap = { version = "4.5", features = ["derive", "env"] }
futures = "0.3.31"
regex = "1.12"
tokio = { version = "1.48", features = ["full"] }
tracing = "0.1"
tracing-subscriber = { version = "0.3", features = ["env-filter"] }
```

We need to import many different types from this crate, and it's convenient to do so by importing everything from the `types` module using `*`.

We will need the `chat` namespace now and `audio` later, but we won't have to import the latter.

src/summarizer.rs RUST

```rust
use async_openai::types::*;
```

The first type we need from this crate is the system request arguments with the long name `ChatCompletionRequestSys` from the `chat` module.

The type implements the `Default` trait, so call the `default` method to get an empty list of arguments.

This arguments type effectively acts as a builder for constructing a system message for the AI model. It therefore provides the `build()` method to construct the message.

Call it and save the result in the `system_message` variable.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn build_messages(&mut self) {
        let system_message =
chat::ChatCompletionRequestSystemMessageArgs::default()
            .build();
    }
}
```

When constructing messages, an error may occur, so we should account for it and return the `Result` from the `build_messages()` method.

Also use the `?` operator to handle the result returned by the `build()` method.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn build_messages(&mut self) -> Result<()> {
        let system_message =
chat::ChatCompletionRequestSystemMessageArgs::default()
            .build()?;
        Ok(())
    }
}
```

Define System Prompt

Now we need to add the main prompt that defines the agent's functionality.

To do this, create a `system.md` file in the `prompt` folder.

Include it in the code using the `include_str!` macro by specifying the path to the created file, and assign it to the `SYSTEM_PROMPT` constant.

src/summarizer.rs RUST

```rust
const SYSTEM_PROMPT: &str = include_str!("../prompts/system.md");
```

Define the agent's role as a log analyzer, and specify its goal: to create a short 1–2 sentence summary.

Be sure to emphasize new messages.

prompts/system.md MARKDOWN

```markdown
You are a log analysis assistant. Generate a brief 1-2 sentence summary of the NEW errors only.
```

Although the summary is based on new log entries, you must note that older messages are also present and necessary for a full understanding of the situation. Indicate this.

prompts/system.md MARKDOWN

```markdown
You are a log analysis assistant. Generate a brief 1-2 sentence summary of the NEW errors only.
Use previous errors only for context to understand the situation.
```

Also note that the agent should describe the problem at a high level and avoid technical details, as we plan to convert this to audio and extracting technical details from an audio message will be difficult, and we only need the big picture of what is happening in the logs.

prompts/system.md MARKDOWN

```markdown
You are a log analysis assistant. Generate a brief 1-2 sentence summary of the NEW errors only.
Use previous errors only for context to understand the situation.
Focus on identifying the main problem without technical details.
```

It should also be noted that the text itself must be easy to understand, even for non-technical users. This ensures it is as concise and quick to grasp when listened to.

prompts/system.md MARKDOWN

```markdown
You are a log analysis assistant. Generate a brief 1-2 sentence summary of the NEW errors only.
Use previous errors only for context to understand the situation.
Focus on identifying the main problem without technical details.
Keep it simple and understandable for non-technical users.
```

Build Chat Messages

Specify the newly created prompt as the content for the system message using the `content()` method, passing the `SYSTEM_PROMPT` constant as the argument.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn build_messages(&mut self) -> Result<()> {
        let system_message =
chat::ChatCompletionRequestSystemMessageArgs::default()
            .content(SYSTEM_PROMPT)
            .build()?;
    }
}
```

We have defined the agent's overall purpose; now let's add a user message—that is, the logs currently read by the application.

For this, we need the `ChatCompletionReques` type from the `chat` module.

Use the `default()` method to create a builder for these arguments.

Then call the `build()` method to construct the message, handling any potential errors. Store the result in the `user_message` variable.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn build_messages(&mut self) -> Result<()> {
        let user_message =
chat::ChatCompletionRequestUserMessageArgs::default()
            .build()?;
    }
}
```

For the message itself, simply pass the processed logs stored in the `processed_context` variable.

And the new entries are already concatenated into a text representation and stored in the `new_errors` variable.

Combine all of this using the `format!` macro and pass it as an argument to the `content()` method, which we also used earlier to populate the system message.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn build_messages(&mut self) -> Result<()> {
        let user_message = chat::ChatCompletionRequestUserMessageArgs::default()
            .content(format!("{}\n\n{}", processed_context, new_errors))
            .build()?;
    }
}
```

Although we constructed different message types, all of them must be converted to the common `ChatCompletionRequestMessage` type.

And return the constructed set of messages as a `Vec` from the `build_messages()` function.

You can immediately use the `vec!` macro to create an empty message vector, assign it to the `messages` variable, and return it as the result.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn build_messages(&mut self) ->
Result<Vec<chat::ChatCompletionRequestMessage>> {
        let messages = vec![
        ];
        Ok(messages)
    }
}
```

`ChatCompletionRequestMessage` is an enum with different variants.

For a system message, use `System`, and for a user message, use the `User` variant. Use these variants to add the messages to the vector you created.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn build_messages(&mut self) ->
Result<Vec<chat::ChatCompletionRequestMessage>> {
        let messages = vec![
chat::ChatCompletionRequestMessage::System(system_message),
chat::ChatCompletionRequestMessage::User(user_message),
        ];
    }
}
```

Since `build_messages()` now returns a vector of messages wrapped in a `Result`, we also need to handle errors, so `generate_summary()` should return a `Result` type as well. Add it as the return type.

And return `Ok` as the successful result.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn generate_summary(&mut self) -> Result<()> {
        self.build_messages()?;
        Ok(())
    }
}
```

Move up the call stack, since `generate_summary()` now also returns a `Result`. On error, we must propagate it further from the `handle_tick` method. There, also declare the return type as `Result`, and return `Ok` for the success case.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn handle_tick(&mut self) -> Result<()> {
        if self.buffer.has_new() {
            tracing::info!("Sending logs for summarization...");
            self.generate_summary()?;
        }
        Ok(())
    }
}
```

You've seen a good example of how to handle errors in a convenient, simple, and elegant way. Just pass them upward as a `Result` value and try to avoid custom errors or any intermediate handling unless it's truly necessary.

> *It may seem that you might miss some important details, but even the generic `Error` from the `anyhow` crate is more than sufficient. If necessary, you can always add additional context to the returned result when it's an error.*

Accordingly, `handle_tick()` can now also fail, so the `step()` method itself must return a `Result` as well. Add `Ok` at the end as the success value.

When calling your `handle_tick()`, use the question mark operator to handle the results returned by this method.

src/summarizer.rs RUST

```rust
impl Summarizer {
    async fn step(&mut self) -> Result<()> {
        select! {
            message = self.rx.recv() => {
                self.handle_message(message);
            }
            _ = self.interval.tick() => {
                self.handle_tick()?;
            }
        }
        Ok(())
    }
}
```

In the `run()` method, we do not want to interrupt the agent if something goes wrong while constructing or sending the `summary`. Therefore, we will not propagate the error upward here—though we will technically keep that option—and will handle it instead.

To do this, add an `if let` construct and check whether the result is the `Err` variant. If it is, print the error to the console and let the loop continue without breaking.

src/summarizer.rs RUST

```rust
impl Summarizer {
    pub async fn run(mut self) -> Result<()> {
        tracing::info!("Starting summarizer");
        while self.active {
            if let Err(err) = self.step().await {
                tracing::error!("Error during processing: {err}");
            }
        }
        Ok(())
    }
}
```

Configure Completion Request

So far, we have only assembled a set of messages for the AI model. To turn this into a summary, we need to make a request to the model by passing these messages.

Add a new `request_completion()` method for this purpose, and have it accept as its argument a vector of messages of type `ChatCompletionRequestMessage`, exactly those we returned from the `build_messages()` method.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn request_completion(
        &self,
        messages: Vec<chat::ChatCompletionRequestMessage>,
    ) {
    }
}
```

Now we can pass messages from one method to another.

In the `generate_summary()` method, we called the `build_messages()` method. Store the return value in the `messages` variable, then call the `request_completion()` method, passing the message vector as an argument to this new method.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn generate_summary(&mut self) -> Result<()> {
        let messages = self.build_messages()?;
        self.request_completion(messages);
        Ok(())
    }
}
```

We will now use a builder to prepare a request for the LLM model. The `async-openai` crate provides the `CreateCha` type—this is a builder for creating a completion request to the model.

Call its `default()` method to create the builder, then call the `build()` method to construct the request.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn request_completion(
        &self,
        messages: Vec<chat::ChatCompletionRequestMessage>,
    ) {
        let request = chat::CreateChatCompletionRequestArgs::default()
            .build();
    }
}
```

As with the previous builders, the build can fail, so the `request_completion()` method must also return a `Result`.

The result returned by the `build()` method should be handled using the `?` operator, and on success we should return `Ok`.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn request_completion(
        &self,
        messages: Vec<chat::ChatCompletionRequestMessage>,
    ) -> Result<()> {
        let request = chat::CreateChatCompletionRequestArgs::default()
            .build()?;
        Ok(())
    }
}
```

To send a completion request, you must specify the model to use. For this purpose, we will choose `gpt-4.1-mini`.

Save this value in the `MODEL` constant and use the `model()` method of the `ChatComp` builder, passing the value of `MODEL` as the parameter.

The *gpt-4.1-mini* model was chosen for a reason, as it supports fine-tuning: you can adjust parameters such as temperature and set limits.

src/summarizer.rs RUST

```rust
const MODEL: &str = "gpt-4.1-mini";
impl Summarizer {
    fn request_completion(
        &self,
        messages: Vec<chat::ChatCompletionRequestMessage>,
    ) -> Result<()> {
        let request =
chat::CreateChatCompletionRequestArgs::default()
            .model(MODEL)
            .build()?;
    }
}
```

Add the message passed in the `messages` parameter to the request. To do this, use the `messages()` method of the same name and pass the received vector as the argument.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn request_completion(
        &self,
        messages: Vec<chat::ChatCompletionRequestMessage>,
    ) -> Result<()> {
        let request = chat::CreateChatCompletionRequestArgs::default()
            .model(MODEL)
            .messages(messages)
            .build()?;
    }
}
```

You can also set the number of output tokens for this model using the `max_completion_tokens()` method.

As the method parameter, pass the number of tokens the model may generate; it will not exceed this during generation.

We will use the `TOKEN_LIMIT` constant as the value, declared separately with the `u32` type the method expects, and set to `150` tokens.

> *As you likely already know, a token is not the same as a word. It is the smallest unit of text processed by the model and is usually part of a word. It is impossible to know in advance how many words will result from a given number of tokens.*

src/summarizer.rs RUST

```rust
const TOKENS_LIMIT: u32 = 150;
impl Summarizer {
    fn request_completion(
        &self,
        messages: Vec<chat::ChatCompletionRequestMessage>,
    ) -> Result<()> {
        let request =
chat::CreateChatCompletionRequestArgs::default()
            .model(MODEL)
            .messages(messages)
            .max_completion_tokens(TOKENS_LIMIT)
            .build()?;
    }
}
```

We can also use the `temperature()` method to set the temperature—the model's level of creativity or strictness.

For our case, set a lower temperature; for example, a value of `0.3` makes the output less creative and more formal and strict.

Store this value in a `TEMPERATURE` constant of type `f32`, since the temperature is a floating-point number, and pass this constant as an argument to the `temperature()` method.

src/summarizer.rs RUST

```rust
const TEMPERATURE: f32 = 0.3;
impl Summarizer {
    fn request_completion(
        &self,
        messages: Vec<chat::ChatCompletionRequestMessage>,
    ) -> Result<()> {
        let request = chat::CreateChatCompletionRequestArgs::default()
            .model(MODEL)
            .messages(messages)
            .max_completion_tokens(TOKENS_LIMIT)
            .temperature(TEMPERATURE)
            .build()?;
    }
}
```

Configure OpenAI Client

We now have a constructed request for the AI model, and we can create and initialize an API client to send it.

First, we need the configuration provided by the `OpenAIConfig` struct from the `config` module of the `async-openai` crate.

Import it and create a default instance using the `new()` method, storing the result in the `config` variable.

src/summarizer.rs RUST

```rust
use async_openai::{config::OpenAIConfig, types::chat};
impl Summarizer {
    pub fn new(
        rx: mpsc::UnboundedReceiver<String>,
    ) -> Self {
        let config = OpenAIConfig::new();
        Self {
            rx,
            active: true,
            buffer: MessageBuffer::new(),
            interval: interval(INTERVAL),
        }
    }
}
```

However, the API cannot be used without a key, so we need to extend the configuration with an API key for accessing the model, which can be set using the `with_api_key()` method by passing it as a parameter.

To make the `Summarizer` universal, add an `API_KEY` parameter to the `new()` function and use this parameter when calling the aforementioned method.

src/summarizer.rs RUST

```rust
impl Summarizer {
    pub fn new(
        api_key: String,
        rx: mpsc::UnboundedReceiver<String>,
    ) -> Self {
        let config = OpenAIConfig::new().with_api_key(api_key);
        Self {
            rx,
            active: true,
            buffer: MessageBuffer::new(),
            interval: interval(INTERVAL),
        }
    }
}
```

It's convenient to obtain the API key from the command-line arguments. To do this, add an `api_key` field of type String to the Args structure.

Have this argument support `short` (short form) and `long` (long form), specified in the `arg` attribute.

You can also allow this parameter to be populated from an environment variable by adding the `env` attribute with the variable name.

Specify `OPENAI_API_KEY` as the environment variable.

Now your program can accept the key for the OpenAI client; pass it from the parsed `args` structure as an argument to the `new()` function that creates the `Summarizer` structure.

src/main.rs RUST

```rust
#[derive(Parser)]
struct Args {
    #[arg(value_name = "LOG_FILE")]
    log_file: PathBuf,
    #[arg(short, long, env = "OPENAI_API_KEY")]
    api_key: String,
}
#[tokio::main]
async fn main() -> Result<()> {
    let mut tasks = Vec::new();
    let summarizer = Summarizer::new(args.api_key, log_rx);
    tasks.push(summarizer.run().boxed());
    select_all(tasks).await.0?;
    Ok(())
}
```

We now have the configuration we created in the `config` variable, suitable for creating an OpenAI API client.

Now import the `Client` type from the `async-openai` crate.

Then use the `with_config()` function to create a client instance and store it in the `client` variable.

Pass the configuration we created, stored in the `config` variable, as the argument to the `with_config()` function.

src/summarizer.rs RUST

```rust
use async_openai::{config::OpenAIConfig, types::*, Client};
impl Summarizer {
    pub fn new(
        api_key: String,
        rx: mpsc::UnboundedReceiver<String>,
    ) -> Self {
        let config = OpenAIConfig::new().with_api_key(api_key);
        let client = Client::with_config(config);
        Self {
            rx,
            active: true,
            buffer: MessageBuffer::new(),
            interval: interval(INTERVAL),
        }
    }
}
```

Also add a `client` field of type `Client<OpenAiConfig>`, since the client's `new` requires you to specify the configuration as a type parameter.

Add this field to the `Summarizer` struct, and in the `new()` function initialize it with the client you created earlier and stored in the `client` variable.

src/summarizer.rs RUST

```rust
pub struct Summarizer {
    rx: mpsc::UnboundedReceiver<String>,
    active: bool,
    buffer: MessageBuffer,
    interval: Interval,
    client: Client<OpenAIConfig>,
}
impl Summarizer {
    pub fn new(
        api_key: String,
        rx: mpsc::UnboundedReceiver<String>,
    ) -> Self {
        let config = OpenAIConfig::new().with_api_key(api_key);
        let client = Client::with_config(config);
        Self {
            rx,
            active: true,
            buffer: MessageBuffer::new(),
            interval: interval(INTERVAL),
            client,
        }
    }
}
```

Wire AI Summarization Flow

Now we can use the client to interact with the AI model by accessing the `Client` field of the struct.

The `Client` type provides the `chat()` method, which returns an object for working with the chat completion API.

src/summarizer.rs RUST

```rust
impl Summarizer {
    fn request_completion(
        &self,
        messages: Vec<chat::ChatCompletionRequestMessage>,
    ) -> Result<()> {
        self.client.chat();
    }
}
```

The Chat Completion API provides the `create()` method, which generates a model response from the request passed as an argument.

Pass the `request` we created as the parameter. Because the method is asynchronous, use `await` to wait for the result. However, the function must be marked `async` to use this operator.

Store the result in the `response` variable. Also remember to handle the returned value, since it is wrapped in a `Result`.

src/summarizer.rs RUST

```rust
impl Summarizer {
    async fn request_completion(
        &self,
        messages: Vec<chat::ChatCompletionRequestMessage>,
    ) -> Result<()> {
        let response = self.client.chat().create(request).await?;
    }
}
```

We should now extract the summary from the response.

The model may return multiple answers, so the response includes a `choices` field, which is a vector.

Access this field and store the result in a new `summary` variable.

src/summarizer.rs RUST

```rust
impl Summarizer {
    async fn request_completion(
        &self,
        messages: Vec<chat::ChatCompletionRequestMessage>,
    ) -> Result<()> {
        let summary = response
            .choices;
    }
}
```

Since `choices` is a `Vec` of `ChatChoice` elements, it may contain many items, but we only need the first one.

You can specify this number in the request, but since we didn't, we will have only a single option anyway.

To obtain the first element of the vector, we first turn it into an iterator with `into_iter()`, then extract the first item with `next()`. Because a vector does not guarantee that it contains at least one element, the result is returned wrapped in an `Option`.

src/summarizer.rs RUST

```rust
impl Summarizer {
    async fn request_completion(
        &self,
        messages: Vec<chat::ChatCompletionRequestMessage>,
    ) -> Result<()> {
        let summary = response
            .choices
            .into_iter()
            .next();
    }
}
```

Therefore, we use `and_then()` and pass a closure that, from this `ChatChoice`, accesses the `message` field, which is of type `ChatCompletionResponseMessage`, and returns the `content` field, which is also optional and thus wrapped in `Option`.

Thus, by combining `and_then()` with the optional content inside, we handle the nested `Option` in one go.

src/summarizer.rs RUST

```rust
impl Summarizer {
    async fn request_completion(
        &self,
        messages: Vec<chat::ChatCompletionRequestMessage>,
    ) -> Result<()> {
        let summary = response
            .choices
            .into_iter()
            .next()
            .and_then(|choice| choice.message.content);
    }
}
```

We should always expect a value, so convert this `Option` into a `Result` using the `context()` method from the `Context` trait in the crate `anyhow`, and provide a message indicating that the request has no response, which will serve as the error message.

We can now propagate any error in the result using the `?` operator.

src/summarizer.rs RUST

```rust
use anyhow::{Context, Result};
impl Summarizer {
    async fn request_completion(
        &self,
        messages: Vec<chat::ChatCompletionRequestMessage>,
    ) -> Result<()> {
        let summary = response
            .choices
            .into_iter()
            .next()
            .and_then(|choice| choice.message.content)
            .context("No content in OpenAI response")?;
    }
}
```

Now we can change the return type of the `request_completion()` method to `String` and, at the end of the method, replace the returned value with the `summary` variable.

src/summarizer.rs RUST

```rust
impl Summarizer {
    async fn request_completion(
        &self,
        messages: Vec<chat::ChatCompletionRequestMessage>,
    ) -> Result<String> {
        Ok(summary)
    }
}
```

Since the `request_completion()` method is asynchronous, you need to use the `await` operator to execute it. Therefore, make the `generate_summary()` method that calls it asynchronous as well using the `async` keyword.

src/summarizer.rs RUST

```rust
impl Summarizer {
    async fn generate_summary(&mut self) -> Result<()> {
        let messages = self.build_messages()?;
        self.request_completion(messages).await?;
        Ok(())
    }
}
```

Let's propagate this asynchrony further up. Make the `handle_tick()` method asynchronous as well, and use the `await` operator at the point where it calls the `generate_summary()` method.

src/summarizer.rs RUST

```rust
impl Summarizer {
    async fn handle_tick(&mut self) -> Result<()> {
        if self.buffer.has_new() {
            tracing::info!("Sending logs for summarization...");
            self.generate_summary().await?;
        }
        Ok(())
    }
}
```

Since the `step()` method was originally asynchronous, we only need to use the `await` operator when calling `handle_tick()`. This executes the entire chain of asynchronous methods.

src/summarizer.rs RUST

```rust
impl Summarizer {
    async fn step(&mut self) -> Result<()> {
        select! {
            message = self.rx.recv() => {
                self.handle_message(message);
            }
            _ = self.interval.tick() => {
                self.handle_tick().await?;
            }
        }
        Ok(())
    }
}
```

Now we can store the returned value without calling the `request_completion()` method.

Add a `summary` variable, change the return type of `generate_summary()` to `String`, and use the created variable as the return value.

src/summarizer.rs RUST

```rust
impl Summarizer {
    async fn generate_summary(&mut self) -> Result<String> {
        let messages = self.build_messages()?;
        let summary = self.request_completion(messages).await?;
        Ok(summary)
    }
}
```

Also store the `summary` returned by the `generate_summary()` method in a variable. You can log the summary using the `info` macro. This confirms that it was generated successfully.

src/summarizer.rs RUST

```rust
impl Summarizer {
    async fn handle_tick(&mut self) -> Result<()> {
        if self.buffer.has_new() {
            tracing::info!("Sending logs for summarization...");
                let summary = self.generate_summary().await?;
                tracing::info!("Summary: {}", summary);
        }
        Ok(())
    }
}
```

Manage Processed Messages

Now we need to improve the buffer. Add a `mark_as_processed()` method whose purpose is to move messages from the list of new ones to the list of processed ones.

src/buffer.rs RUST

```rust
impl MessageBuffer {
    pub fn mark_as_processed(&mut self) {
    }
}
```

With the `drain()` method we can obtain an iterator that removes elements. The method takes as its parameter a range specified by the `..` operator.

If you do not specify the start and end of the range, the entire array will be drained. Use this method by adding it to a `for` loop and assigning each element's value to the loop variable `message`.

src/buffer.rs RUST

```rust
impl MessageBuffer {
    pub fn mark_as_processed(&mut self) {
        for message in self.new_messages.drain(..) {
        }
    }
}
```

Messages retrieved in the loop from the new messages queue should be added to the processed list using the `push_back()` method, passing the extracted message as an argument.

src/buffer.rs RUST

```rust
impl MessageBuffer {
    pub fn mark_as_processed(&mut self) {
        for message in self.new_messages.drain(..) {
            self.processed_messages.push_back(message);
        }
    }
}
```

As you remember, we planned to store only a limited number of processed messages.

We already declared the `DEPTH` constant, which limits the number of messages, and we used it when initializing the vector size.

Now add an extra condition before adding a message to ensure that the queue length does not exceed this limit.

In other words, compare the value returned by the `len()` method with the limiting `DEPTH` constant.

If the limit is reached or exceeded, meaning our message queue is already full, use the `pop_front()` method to remove the oldest message.

This will discard the extra message.

src/buffer.rs RUST

```rust
impl MessageBuffer {
    pub fn mark_as_processed(&mut self) {
        for message in self.new_messages.drain(..) {
            if self.processed_messages.len() >= DEPTH {
                self.processed_messages.pop_front();
            }
            self.processed_messages.push_back(message);
        }
    }
}
```

We implemented the `mark_as_processed()` method, which moves new messages into the processed ones.

We will call this method after successfully generating a summary for the current set of processed and new log messages.

So call the `mark_as_processed()` method on the `buffer` field of the `Summarizer` struct in the `handle_tick()` method.

src/summarizer.rs RUST

```rust
impl Summarizer {
    async fn handle_tick(&mut self) -> Result<()> {
        if self.buffer.has_new() {
            tracing::info!("Sending logs for summarization...");
                let summary = self.generate_summary().await?;
                tracing::info!("Summary: {}", summary);
                self.buffer.mark_as_processed();
        }
        Ok(())
    }
}
```

Our minimal implementation is ready; it reads a file and generates a summary.

Let's test it. To do this, we need to set an OpenAI API key, which you can obtain on the OpenAI service management page.

After you set the environment variable, run the `cargo run` command, passing the name of the sample log file we created earlier (the `sample.log` file) as a parameter.

To pass parameters not to the `run` command but to our application itself, you need to use a double dash `--` as a separator between the command's parameters and our program's parameters.

```sh
.
$ export OPENAI_API_KEY="<your-key>"
```

```sh
.
$ cargo run -- sample.log
```

Ready for More?

You've already built a log companion that watches your files, pulls out important errors and warnings, and turns them into short, clear summaries. It keeps a rolling memory of past problems, focuses on the newest issues, and uses an AI prompt designed for simple, non-technical explanations that are easy to understand as audio.

In the full version of the book, you'll hook those summaries up to text-to-speech, turning each update into MP3 audio and logging how much data you generate. You'll add a dedicated audio worker that receives these audio chunks over channels, plays them through the system output using **Rodio**, and wire everything into your existing async task system so text and sound stay in sync.

By the end, you'll have a fully automated incident "radio" that listens to your logs and speaks out what's going wrong in real time.