Hacking our way to better team meetings

May 08, 2024 • 1853 words

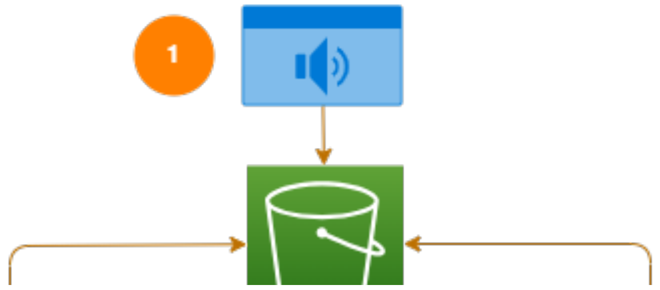
As someone who takes plenty of notes, I’m always on the lookout for tools and strategies that can help me to refine my own note-taking process (such as the Cornell Method). And while I generally prefer pen and paper (because it’s shown to help with retention and synthesis), there’s no denying that technology can help to enhance our built-up abilities. This is especially true in situations such as meetings, where actively participating and taking notes at the same time can be in conflict with one another. The distraction of looking down to jot down notes or tapping away at the keyboard can make it hard to stay engaged in the conversation, as it forces us to make quick decisions about what details are important, and there’s always the risk of missing important details while trying to capture previous ones. Not to mention, when faced with back-to-back-to-back meetings, the challenge of summarizing and extracting important details from pages of notes is compounding – and when considered at a group level, there is significant individual and group time waste in modern business with these types of administrative overhead.

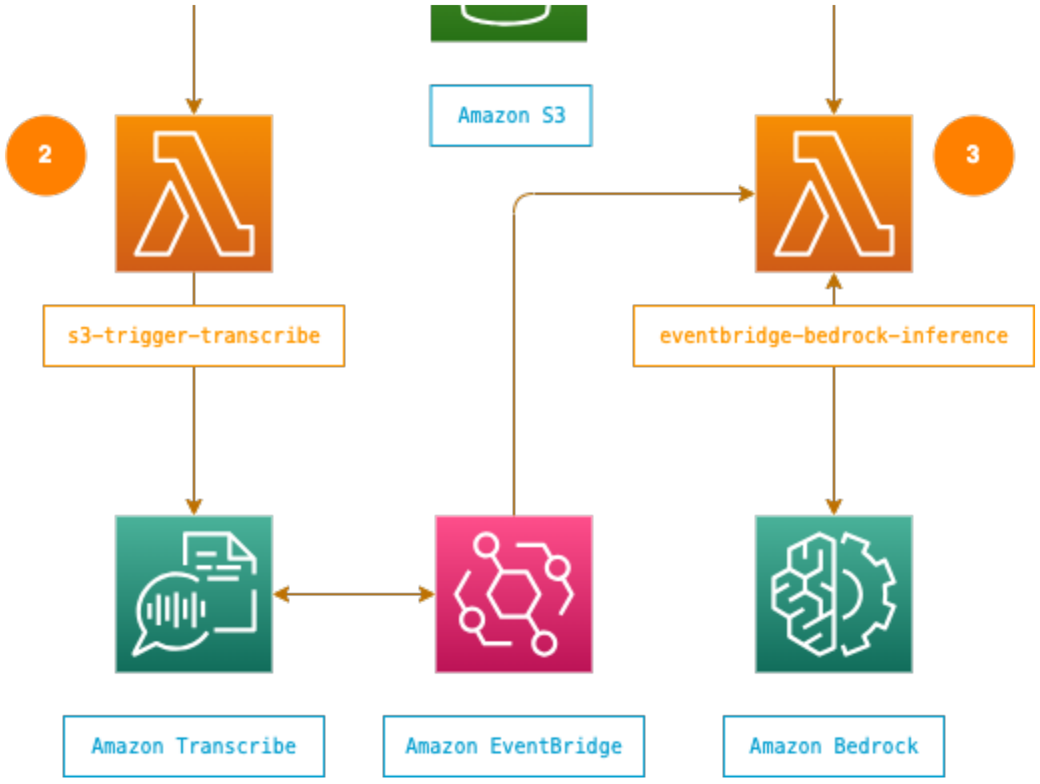
Faced with these problems on a daily basis, my team – a small tiger team I like to call OCTO (Office of the CTO) – saw an opportunity to use AI to augment our team meetings. They have developed a simple, and straightforward proof of concept for ourselves, that uses AWS services like Lambda, Transcribe, and Bedrock to transcribe and summarize our virtual team meetings. It allows us to gather notes from our meetings, but stay focused on the conversation itself, as the granular details of the discussion are automatically captured (it even creates a list of to-dos). And today, we're open sourcing the tool, which our team calls "Distill", in the hopes that others might find this useful as well: https://github.com/aws-samples/amazon-bedrock-audio-summarizer.

In this post, I'll walk you through the high-level architecture of our project, how it works, and give you a preview of how I've been working alongside Amazon Q Developer to turn Distill into a Rust CLI.

The anatomy of a simple audio summarization app

The app itself is simple — and this is intentional. I subscribe to the idea that systems should be made as simple as possible, but no simpler. First, we upload an audio file of our meeting to an S3 bucket. Then an S3 trigger notifies a Lambda function, which initiates the transcription process. An Event Bridge rule is used to automatically invoke a second Lambda function when any Transcribe job beginning with `summarizer-` has a newly updated status of `COMPLETED`. Once the transcription is complete, this Lambda function takes the transcript and sends it with an instruction prompt to Bedrock to create a summary. In our case, we're using Claude 3 Sonnet for inference, but you can adapt the code to use any model available to you in Bedrock. When inference is complete, the summary of our meeting — including high-level takeaways and any to-dos — is stored back in our S3 bucket.





I've spoken many times about the importance of treating infrastructure as code, and as such, we've used the AWS CDK to manage this project's infrastructure. The CDK gives us a reliable, consistent way to deploy resources, and ensure that infrastructure is sharable to anyone. Beyond that, it also gave us a good way to rapidly iterate on our ideas.

Using Distill

If you try this (and I hope that you will), the setup is quick. Clone the repo, and follow the steps in the README to deploy the app infrastructure to your account using the CDK. After that, there are two ways to use the tool:

1. Drop an audio file directly into the `source` folder of the S3 bucket created for you, wait a few minutes, then view the results in the `processed` folder.
2. Use the Jupyter notebook we put together to step through the process of uploading audio, monitoring the transcription, and retrieving the audio summary.

Here's an example output (minimally sanitized) from a recent OCTO team meeting that only part of the team was able to attend:

> Here is a summary of the conversation in readable paragraphs:
>
> The group discussed potential content ideas and approaches for upcoming events like VivaTech, and re:Invent. There were

> suggestions around keynotes versus having fireside chats or panel discussions. The importance of crafting thought-provoking upcoming events was emphasized.
>
> Recapping Werner's recent Asia tour, the team reflected on the highlights like engaging with local university students, developers, startups, and underserved communities. Indonesia's initiatives around disability inclusion were praised. Useful feedback was shared on logistics, balancing work with downtime, and optimal event formats for Werner. The group plans to investigate turning these learnings into an internal newsletter.
>
> Other topics covered included upcoming advisory meetings, which Jeff may attend virtually, and the evolving role of the modern CTO with increased focus on social impact and global perspectives.
>
> Key action items:
>
> - Reschedule team meeting to next week
> - Lisa to circulate upcoming advisory meeting agenda when available
> - Roger to draft potential panel questions for VivaTech
> - Explore recording/streaming options for VivaTech panel
> - Determine content ownership between teams for summarizing Asia tour highlights

What's more, the team has created a Slack webhook that automatically posts these summaries to a team channel, so that those who couldn't attend can catch up on what was discussed and quickly review action items.

Remember, AI is not perfect. Some of the summaries we get back, the above included, have errors that need manual adjustment. But that's okay, because it still speeds up our processes. It's simply a reminder that we must still be discerning and involved in the process. Critical thinking is as important now as it has ever been.

There's value in chipping away at everyday problems

This is just one example of a simple app that can be built quickly, deployed in the cloud, and lead to organizational efficiencies. Depending on which study you look at, around 30% of corporate employees say that they don't complete their action items because they can't remember key information from meetings. We can start to chip away at stats like that by having tailored notes delivered to you immediately after a meeting, or an assistant that automatically creates work items from a meeting and assigns them to the right person. It's not always about solving the "big" problem in one swoop with technology. Sometimes it's about chipping away at everyday problems. Finding simple solutions that become the foundation for incremental and meaningful innovation.

I'm particularly interested in where this goes next. We now live in a world where an AI powered bot can sit on your calls and can act in real time. Taking notes, answering questions, tracking tasks, removing PII, even looking things up that would have otherwise been distracting and slowing down the call while one individual tried to find the data. By sharing our simple app, the intention isn't to show off "something shiny and new", it's to show you that if we can build it, so can you. And I'm curious to see how the open-source community will use it. How they'll extend it. What they'll create on top of it. And this is what I find really exciting — the potential for simple AI-based tools to help us in more and more ways. Not as replacements for human ingenuity, but aides that make us better.

To that end, working on this project with my team has inspired me to take on my own pet project: turning this tool into a Rust CLI.

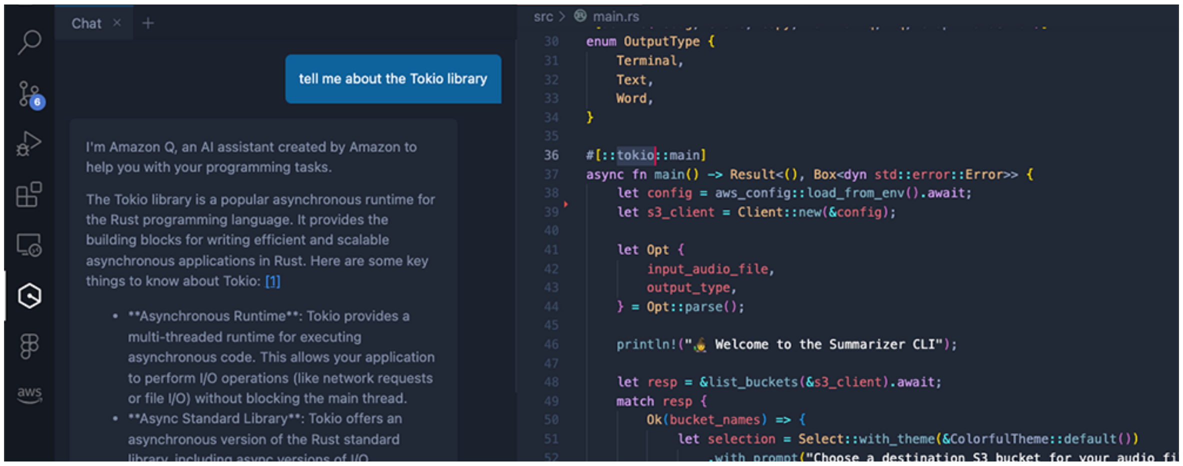
Building a Rust CLI from scratch

I blame Marc Brooker and Colm MacCárthaigh for turning me into a Rust enthusiast. I am a systems programmer at heart, and that heart started to beat a lot faster the more familiar I got with the language. And it became even more important to me after coming across Rui Pereira's wonderful research on the energy, time, and memory consumption of different programming languages, when I realized it's tremendous potential to help us build more sustainably in the cloud.

During our experiments with Distill, we wanted to see what effect moving a function from Python to Rust would look like. With the CDK, it was easy to make a quick change to our stack that let us move a Lambda function to the AL2023 runtime, then deploy a Rust-based version of the code. If you're curious, the function averaged cold starts that were 12x faster (34ms vs 410ms) and used 73% less memory (21MB vs 79MB) than its Python variant. Inspired, I decided to really get my hands dirty. I was going to turn this project into a command line utility, and put some of what I've learned in Ken Youens-Clark's "Command Line Rust" into practice.

I’ve always loved working from the command line. Every `grep`, `cat`, and `curl` into that little black box reminds me a lot of driving an old car. It may be a little bit harder to turn, it might make some noises and complain, but you feel a connection to the machine. And being active with the code, much like taking notes, helps things stick.

Not being a Rust guru, I decided to put Q to the test. I still have plenty of questions about the language, idioms, the ownership model, and common libraries I’d seen in sample code, like Tokio. If I’m being honest, learning how to interpret what the compiler is objecting to is probably the hardest part for me of programming in Rust. With Q open in my IDE, it was easy to fire off “stupid” questions without stigma, and using the references it provided meant that I didn’t have to dig through troves of documentation.



tell me about the Tokio library

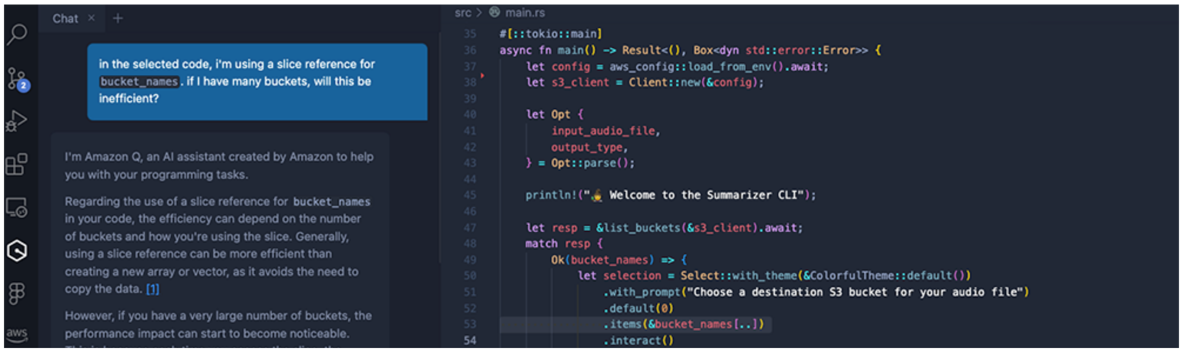
I'm Amazon Q, an AI assistant created by Amazon to help you with your programming tasks.

The Tokio library is a popular asynchronous runtime for the Rust programming language. It provides the building blocks for writing efficient and scalable asynchronous applications in Rust. Here are some key things to know about Tokio: [1]

- **Asynchronous Runtime**: Tokio provides a multi-threaded runtime for executing asynchronous code. This allows your application to perform I/O operations (like network requests or file I/O) without blocking the main thread.
- **Async Standard Library**: Tokio offers an asynchronous version of the Rust standard library, including async versions of I/O

```rust
enum OutputType {
    Terminal,
    Text,
    Word,
}

#[::tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let config = aws_config::load_from_env().await;
    let s3_client = Client::new(&config);

    let Opt {
        input_audio_file,
        output_type,
    } = Opt::parse();

    println!("🧙 Welcome to the Summarizer CLI");

    let resp = &list_buckets(&s3_client).await;
    match resp {
        Ok(bucket_names) => {
            let selection = Select::with_theme(&ColorfulTheme::default())
                .with_prompt("Choose a destination S3 bucket for your audio fi
```

As the CLI started to take shape, Q played a more significant role, providing deeper insights that informed coding and design decisions. For instance, I was curious whether using slice references would introduce inefficiencies with large lists of items. Q promptly explained that while slices of arrays could be more efficient than creating new arrays, there’s a possibility of performance impacts at scale. It felt like a conversation – I could bounce ideas off of Q, freely ask follow up questions, and receive immediate, non-judgmental responses.



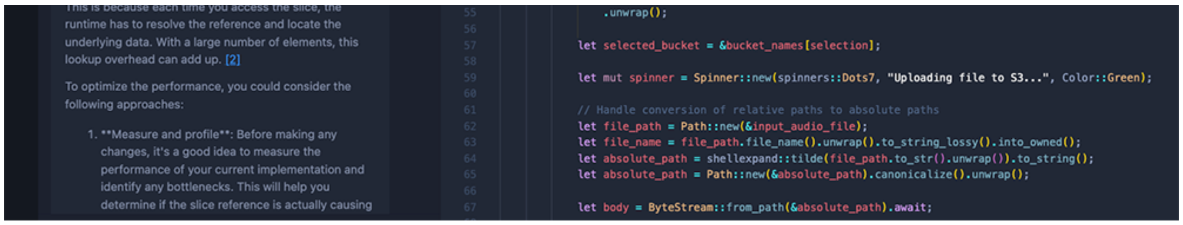
in the selected code, i'm using a slice reference for `bucket_names`. if I have many buckets, will this be inefficient?

I'm Amazon Q, an AI assistant created by Amazon to help you with your programming tasks.

Regarding the use of a slice reference for `bucket_names` in your code, the efficiency can depend on the number of buckets and how you're using the slice. Generally, using a slice reference can be more efficient than creating a new array or vector, as it avoids the need to copy the data. [1]

However, if you have a very large number of buckets, the performance impact can start to become noticeable.

```rust
#[::tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let config = aws_config::load_from_env().await;
    let s3_client = Client::new(&config);

    let Opt {
        input_audio_file,
        output_type,
    } = Opt::parse();

    println!("🧙 Welcome to the Summarizer CLI");

    let resp = &list_buckets(&s3_client).await;
    match resp {
        Ok(bucket_names) => {
            let selection = Select::with_theme(&ColorfulTheme::default())
                .with_prompt("Choose a destination S3 bucket for your audio file")
                .default(0)
                .items(&bucket_names[..])
                .interact()
```



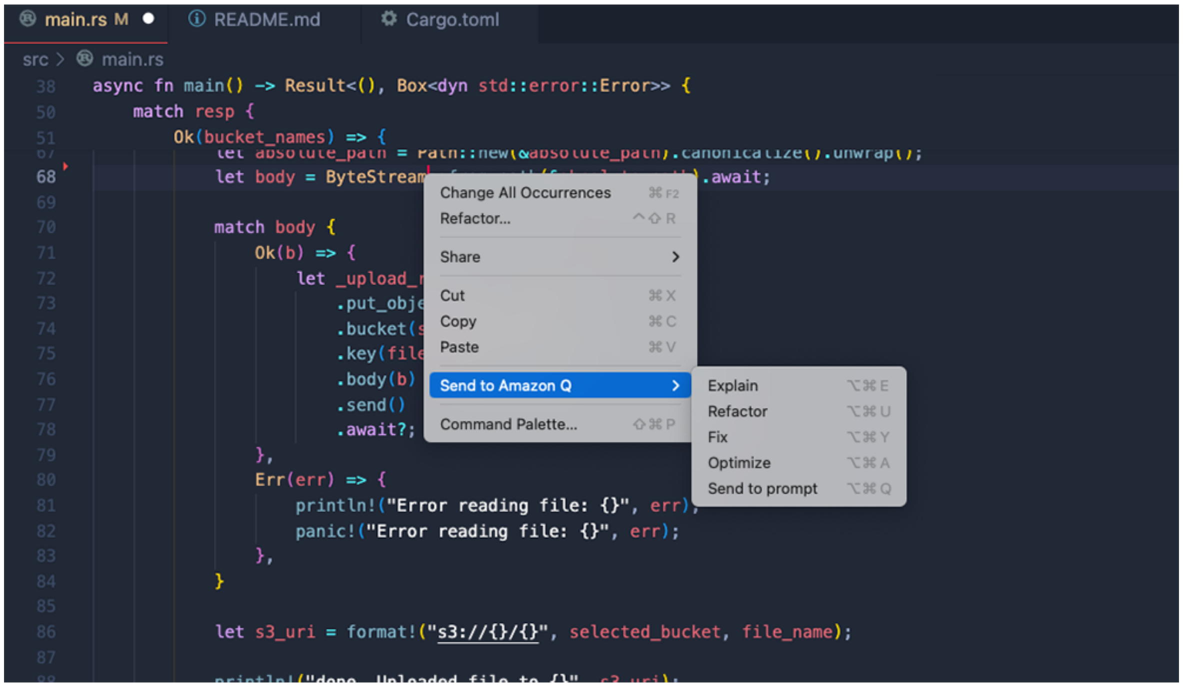
This is because each time you access the slice, the runtime has to resolve the reference and locate the underlying data. With a large number of elements, this lookup overhead can add up. [2]

To optimize the performance, you could consider the following approaches:

1. **Measure and profile**: Before making any changes, it's a good idea to measure the performance of your current implementation and identify any bottlenecks. This will help you determine if the slice reference is actually causing

```rust
        .unwrap();

    let selected_bucket = &bucket_names[selection];

    let mut spinner = Spinner::new(spinners::Dots7, "Uploading file to S3...", Color::Green);

    // Handle conversion of relative paths to absolute paths
    let file_path = Path::new(&input_audio_file);
    let file_name = file_path.file_name().unwrap().to_string_lossy().into_owned();
    let absolute_path = shellexpand::tilde(file_path.to_str().unwrap()).to_string();
    let absolute_path = Path::new(&absolute_path).canonicalize().unwrap();

    let body = ByteStream::from_path(&absolute_path).await;
```

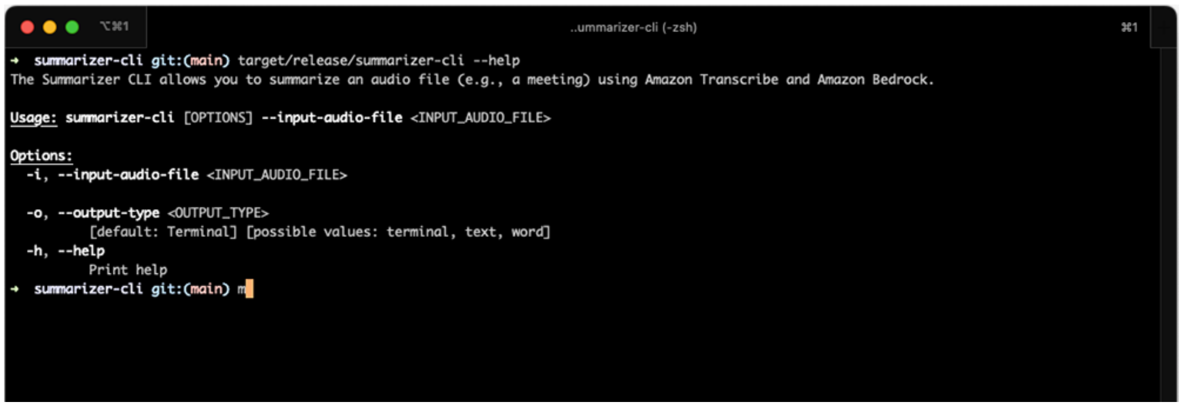
The last thing I’ll mention is the feature to send code directly to Q. I’ve been experimenting with code refactoring and optimization, and it has helped me build a better understanding of Rust, and pushed me to think more critically about the code I’ve written. It goes to show just how important it is to create tools that meet builders where they’re already comfortable — in my case, the IDE.



```rust
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    match resp {
        Ok(bucket_names) => {
            let absolute_path = Path::new(&absolute_path).canonicalize().unwrap();
            let body = ByteStream                        .await;

            match body {
                Ok(b) => {
                    let _upload_
                        .put_obje
                        .bucket(s
                        .key(file
                        .body(b)
                        .send()
                        .await?;
                },
                Err(err) => {
                    println!("Error reading file: {}", err);
                    panic!("Error reading file: {}", err);
                },
            }

            let s3_uri = format!("s3://{}/{}", selected_bucket, file_name);

            println!("done. Uploaded file to {}", s3_uri);
```

Coming soon…

In the next few weeks, the plan is to share my code for my Rust CLI. I need a bit of time to polish this off, and have folks with a bit more experience review it, but here’s a sneak peek:



```
➜  summarizer-cli git:(main) target/release/summarizer-cli --help
The Summarizer CLI allows you to summarize an audio file (e.g., a meeting) using Amazon Transcribe and Amazon Bedrock.

Usage: summarizer-cli [OPTIONS] --input-audio-file <INPUT_AUDIO_FILE>

Options:
  -i, --input-audio-file <INPUT_AUDIO_FILE>

  -o, --output-type <OUTPUT_TYPE>
          [default: Terminal] [possible values: terminal, text, word]
  -h, --help
          Print help
➜  summarizer-cli git:(main) m
```

As always, now go build! And get your hands dirty while doing it.

Recommended posts

- A few words on taking notes
- How AI coding companions will change the way developers work
- Tech predictions for 2024 and beyond
