## Introduction & Motivation (1.5 mins)

Good morning everyone, I’m Matthew Britton and today I’ll be presenting my research on *Automated and Verified Extract Method Refactoring in Rust*, supervised by Dr Alex Potanin and Sasha Pak, who I believe is present today. **SLIDE**

Over the next 10 minutes, I’ll take you through what extract method refactoring is, why it’s so challenging in Rust, how I expanded the REM toolchain to handle these challenges, how we verify that the refactorings are correct, and finally, the results we’ve obtained so far. **SLIDE**

Before diving into extract method refactoring itself, I want to set the scene.

This wasn’t just about building a tool — it’s about creating a *research platform*.  
The goal was to design a modular system where each component and algorithm could be swapped out, studied and tested in isolation.

Over the course of the project, this has led to three aims:

* To build a system that supports experimental evaluation of automated Rust refactoring
* To bridge research and practice by producing a usable developer tool
* And to investigate how lifetimes, ownership, and control flow interact with automation and correctness

In other words — not just to refactor code, but to *research* how safe, automated refactoring can be achieved in Rust. **SLIDE**

## Extract Method & Rusts Challenges (2 mins)

So what is Extract Method Refactoring? **SLIDE**

At its core, Extract Method refactoring is simple — take a block of code, turn it into a function, and replace it with a call, as the example on the left shows.

Developers do this all the time: a recent JetBrains survey found nearly 80 percent of developers refactor weekly or daily — yet they still perform extract-method refactorings manually. **SLIDE**

What then makes this refactoring so much harder in Rust?

The key challenge is Rust’s *safety model*. The compiler acts like a bodyguard — it’s constantly checking who owns what, and for how long. Every value in Rust has exactly one owner. That ownership can be *moved*, or temporarily *borrowed* — either immutably or mutably — but those borrows cannot overlap. On the left, we can see how the compiler prevents multiple mutable borrows and enforces these rules automatically. On the right, we see *lifetimes* — every reference has a defined scope during which it’s valid. The compiler checks that no reference ever outlives the data it points to.

So when we try to pull code out into a new function, we’re effectively breaking those ownership and lifetime relationships — and that’s what makes automated extraction in Rust so difficult. **SLIDE**

Here’s what happens when we try a naïve extraction in Rust.

On the left, we’re taking a small code block — just splitting a string — and pulling it out into a new function. But as soon as we do that, the extracted function takes ownership of text, rather than just borrowing it. The compiler rightly complains that we’ve moved text and can’t use it afterwards — shown by the red error on the right. However, what we actually wanted was for the show\_first function to borrow the text immutably, keeping the original behaviour. This is exactly the kind of subtle lifetime and ownership repair that our extraction tool needs to handle automatically. **SLIDE**

The original REM toolchain tackled these issues — fixing control flow, analysing ownership, and repairing lifetimes through compiler feedback. These stages are shown in blue on the diagram on screen.   
In short, REM made Extract Method *possible* in Rust, but it had many crippling limitations and remained a purely research prototype without real-world usability. **SLIDE**

## Expanding REM (2.5 mins)

The first major aspect to this project has been expanding the REM prototype to meet the demands of modern rust programming, whilst maintaining the ability to rapidly iterate and evaluate new techniques and concepts. **SLIDE**

This work builds directly on two prior projects — *Adventure of a Lifetime* and *Borrowing Without Sorrowing*. The version of REM developed in those projects could repair lifetimes automatically, and indeed its core logic has been integrated into the new toolchain, but it relied incredibly heavily on an outdated IntelliJ platform and wasn’t suitable for real-world experimentation.

My contribution has been to completely re-architect that toolchain.

The new design introduces an Incremental Intermediate Representation Server that keeps a cached representation of the program’s syntax and intermediate form — the AST and a form of Mid level IR. The program watches the filesystem, and whenever a file changes, only the affected regions are rebuilt, which keeps the system incredibly responsive.

Sitting after that is the Extraction engine, which is going to be covered off on very shortly. As I mentioned earlier, components of REM then come after the extraction engine to address lifetime issues specifically. All these pieces are tied together with whats known as a CLI – which is one unified command line entry point. From there a VSCode extension allows everyday developers to benefit directly from this research. **SLIDE**

The next slide shows how this architecture behaves as a full experimental framework. The IR Server manages incremental updates; the Extraction Engine manages the analysis; and allows us to plug in the new composable and modular algorithms, like *Generic Extraction* and *Asynchronous Extraction*. Crucially, preceding each of these algorithms is a heuristic that determines if it applies, and also allows us to easily toggle an algorithm on or off to assist with the research process. After each extraction, results are fed back into the IR cache for immediate re-analysis.

The result of this is that the system isn’t just fast, it helps us research the best way to achieve each small goal. Every stage in this process can be measured, easily modified and compared without limiting its functionality. This helped us achieve the aim of designing a research tool that can be used by developers. **SLIDE**

## Verification Toolchain (2 mins)

The second major aspect of this project has been coming up with a way to verify that the complex transformations we are executing don’t change the code. **SLIDE**

Just because Rust code compiles doesn’t mean it’s correct. The Rust compiler guarantees *type safety* and *memory correctness*, but that’s where its job ends — it only proves the code runs safely, not that it behaves the same. After extraction, our toolchain rewrites lifetimes, restructures control flow, and introduces new wrapper types. These transformations can subtly change program behaviour, even though the compiler still accepts the code.

To address that, we formalise what it means for refactoring to be correct. We prove that for every input, both the original and refactored programs produce identical results — expressed as these equivalence formulas — and we automatically discharge those proofs using Coq, as shown here in the bottom right. Whilst this guarantee is very straightforward to write down, proving it for any arbitrary program without even knowing what the program does is less so. **SLIDE**

Once both versions of the program are extracted, the verification pipeline begins. The extraction toolchain produces two working programs — the original, and its refactored counterpart — and REM fixes any lifetime issues in the refactored version using compiler feedback. From there, CHARON translates each program into a structured, low-level bytecode representation called *LLBC*, which captures Rust’s semantics precisely. AENEAS then converts that bytecode into Coq definitions, giving us a formal model of both programs.

Finally, the Coqsolver automatically proves that, under the same observable behaviour, both caller functions are equivalent — effectively guaranteeing that our refactoring hasn’t changed what the code does. A key aspect of the research up to this point has been ensuring that these proofs are completely annotation-free — the developer doesn’t need to write or modify any formal specifications, and everything is inferred automatically from the code itself. **SLIDE**

## Results & Evaluation (2-3 mins)

So, did it work? *(pause for effect)* **SLIDE**

We evaluated the extraction toolchain on 20 of the top real-world Rust projects drawn from Evan Li’s GitHub rankings — including Deno, Tauri, and Zed. From here we were able to extract 40 case studies where developers manually performed an extract method, alongside a few more arbitrary attempts to extract code from these projects by ourselves. The goal with finding these cases was to specifically stress new capabilities of this toolchain, such as the ability to refactor generic and asynchronous code. 26 of these cases extracted successfully, with another 4 blocked only by minor formatting bugs.

Importantly, the toolchain was also able to replicate the 37 successful cases in the original REM evaluation, whilst adding substantial new refactoring capabilities.

Performance-wise, extraction is now fast enough to feel instantaneous. The original REM took on average 0.95 seconds after IntelliJ performed its extraction; ours performs complete end-to-end extraction in on average 0.52 seconds — and the IR query itself returns in under 10 milliseconds. Crucially, we have chosen to measure the total time including VSCode overhead, so this time reflects exactly how responsive the toolchain would be for a developer. **SLIDE**

We also evaluated the verifier on 20 cases — 10 real-world and 10 synthetic examples.  
Every case successfully discharged its proof of equivalence. The verifications all took between 2 – 5 seconds to discharge. Due to their simplicity, the synthetic examples all discharged on the shorter end of this spectrum, so in the real world we are looking at closer to the 5 second span. It must also be acknowledged that CHARON is still in its early development phase and can just hang on a conversion for upwards of 30 seconds. To keep the results sensible I have chosen to omit these cases but they do affect the user experience.

In that vein, as the data shows, CHARON’s LLBC generation accounts for 80–90 percent of the total runtime; the Coq conversion and subsequent proofs themselves are practically instantaneous. And that’s OK - verification can run in the background while the developer continues to work, so the latency in this instance isn’t user-visible.

Additionally, this work was recently presented at the 2025 SPLASH conference, where feedback implored us to start exploring edge cases such as recursive functions, loops, and asynchronous contexts — to determine where the verifier breaks down, so the hope is that when this project is finished in a month we will be able to accurately show exactly what this verifier is capable of. **SLIDE**

## Key Takeaways & Future work (1.5 – 2 mins)

What do we takeaway from all of this? **SLIDE**

First lets recap what these results show and what we have taken away:

* Extraction is fast and usable, completing in under half a second.
* The incremental IR server works, delivering sub-10 millisecond queries.
* Verification isn’t slow — translation is the bottleneck – which is out of the scope of this project. But in the future, verification could be as instantaneous as the actual extraction
* Every verified case proved behavioural equivalence. Whilst we haven’t found counter examples at this stage, they must exist, and we need them to determine the limits of our endeavours
* The architecture itself enables new research, letting us isolate and study each stage.
* And finally, this demonstrates that extract method refactoring can be both research friendly and developer-friendly.

**SLIDE**

Looking ahead, there are four major directions:

* First, bringing incremental compilation to LLBC, which could reduce verification latency to near-instant levels.
* Second, improving diagnostics, so the tool can explain why extraction or verification failed.
* Third, expanding coverage to handle unsafe code, concurrency, and higher-order borrows.
* And long-term, integrating with RustAnalyzer, which is the official Rust front-end, to make verified refactoring part of the everyday developer toolkit

In closing — this project shows that Rust refactoring can be both fast and trustworthy.  
We’ve gone from a theoretical prototype to a usable, verified toolchain that bridges research and real development.

Thank you for listening — I’m happy to take any questions.