Hi everyone!
I'm Chaitrika, and I'm thrilled to be spending my summer interning at Chubb. I've just completed my third year of B.Tech at IIIT Hyderabad, and it's been an exciting journey so far.

Today, I'm excited to share what I have been working on — **Project Synapse**.
I picked this name intentionally: just like a synapse connects neurons in the brain, this project is all about connecting the dots across our enterprise software — making it easier to understand, navigate, and build upon the existing codebase and documentation.

—----------

So, the challenge we're tackling is this:

Imagine you're a new developer joining the team, or you're trying to fix a bug in a system you haven't touched in a while. The information you need — the code, where it lives, what it does, and how it connects — is scattered across different GitHub repositories.

And it's not just about finding the code — it's about understanding **how it all fits together**. What depends on what? What's safe to change, and what could break things?

Without clear context, this leads to wasted time, confusion, and risk.

(Pause – transition to solution)

our goal is to build a **smart, connected map of all our repositories**. (Point to LLM Knowledge Base Image)

We're doing this by generating **ASTs — Abstract Syntax Trees** — for each code repository.

Why ASTs?
Because they give us **a structured, detailed understanding of the code**, beyond just plain text. With ASTs, we can extract useful metadata like:

- Function definitions and calls
- Class structures
- Imports and dependencies
- Variable scopes

Basically, **everything the code is doing, and how it's connected**.

Then, we store this structured information in a **Neo4j database** as a **knowledge graph**.

Neo4j is **graph-native**, meaning it's perfect for modeling relationships — like which function calls which, which module depends on which, or which repo imports what.

With Neo4j, we can **query relationships intuitively**, like:
"Show me all the downstream functions affected by a change in module X."
Or:
"Which services depend on this library?"

(Pause – transition to LLM integration)

Finally, we feed this graph data into an **LLM-powered service** — but not as raw code.

Instead, we're giving the LLM **context** — a head start.

So when someone asks,
"What does this module do?"
or
"What breaks if I change this function?"

The LLM can answer based on a **structured understanding** of the code — not just token-matching from a flat file.

(Pause – transition to benefits)

This unlocks huge value:

(Point to "Faster Onboarding")
New developers can understand unfamiliar systems quickly — without having to read every file manually.

(Point to "Fast, Deep Insights")
Teams can trace bugs, understand impact, and make smarter decisions — all backed by a **connected understanding** of the codebase.

—----------

Now, you might be thinking — *"But we already have Prodigy. Doesn't it do the same thing?"*

And that's a **fair question**. (Pause)

The key difference is this: **we're not just throwing code at the model and hoping it figures things out.**

We're giving it **context**. (Pause and emphasize "context")

We're feeding the model a **smart, structured map** of the entire codebase — so it understands **how everything is connected.**

Think of it like this:
Prodigy is great when you're looking at a single piece of code — like reading **one page in a book**.

It can help you understand what that page says.

But it **can't tell you how that page relates to the rest of the book** — or to other books in the library.

Our approach, using a **code knowledge graph**, is like having the **entire library indexed and cross-referenced**.

—----------

What you're seeing here is the **interface we've built** to interact with our **code knowledge graph**.

(Point to "I am new to marketplace / I know what I am doing" buttons)
We've designed this to be helpful for **everyone** — whether you're new to the system or already familiar with the codebase.

🔹 If you're new, you can click **"I am new to marketplace"** — this will guide you through some common queries and help you get started easily.

🔹 If you already know your way around, you can use the **"I know what I'm doing"** option to enter **free-form natural language queries**.

(Pause and transition to the graph schema)
Before I run any queries, let me briefly walk you through the **graph schema** on the left side of the screen. (Point to schema)

This schema shows the **types of entities** and **relationships** we're tracking in our knowledge graph — things like:

- `Repository`
- `Class`
- `Method`
- `FunctionCall`

And the **edges** between them — such as:

- `DEPENDS_ON`

- `CONTAINS`
- `CALLS`

For example, you can see that one repository can **`DEPENDS_ON`** another — this reflects the **code dependencies** we talked about earlier.

Understanding this schema can help users ask **more precise and insightful questions**.

(Smooth transition to upcoming feature)
One more thing I want to quickly mention —
we're also working on a **Class Diagram feature**, which will provide a **visual overview of the codebase structure**.

It's still a **work in progress**, but once it's fully developed, it'll offer a powerful way to **explore and understand complex class relationships visually** — especially helpful for onboarding and refactoring tasks.

—----------

Smooth transition from the demo)
So, what's next for **Project Synapse**?

As you've seen, we've made great progress in **understanding and mapping our codebase** through structured analysis and dependency tracking.

(Point to the diagram on the slide)
But our **ultimate vision** goes beyond just code.

We aim to **unify all the knowledge about our software** — code, documentation, architecture diagrams, how-to guides — into a **single, connected source of truth**.

Think of it as building a **living, evolving encyclopedia** of our systems — one that any developer, new or experienced, can rely on.

Right now, we've focused on understanding the **technical meaning of the code** — its structure, relationships, and dependencies.

But to truly get to the **functional or business-level meaning**, we need to integrate the **documentation** — because that's where a lot of context and rationale lives.

We're actively working on bringing this documentation into the knowledge graph, **just like we did with code**.

And to make this effective, we need **structure** — especially in places like README files.

So one of the main insight i found was , we need to define **a consistent format for READMEs** that makes them easier to parse and integrate into our graph.

Long term, we envision this analysis being **automated and embedded directly into our pipelines**, so knowledge stays fresh and always reflects the current state of the system.

—----------

READ FROM THE SLIDE(challenges/progress)
READ FROM THE SLIDE(thank you)