Unlocking Growth and Developer Productivity With Code Visibility:

A Practical Guide

GitKraken

50%
less time in code review

3K hours
saved in code review/year

$443.2K
saved in onboarding/year

Unlocking Growth and Developer Productivity With Code Visibility

A Practical Guide

Summary

Understanding code is the fundamental task of a developer. Developers spend 60% of their time reading and understanding the company's code before they can ship new features. To do this effectively they need a guide. Without one, they'll take longer onboarding, longer to write code, and longer to review. All of which leads to more costs and less growth. But with a guide, like code visibility, your team can learn, understand, and move quicker, leading to a more productive, happier team and more growth for your company.

Here's how you can implement code visibility with GitKraken at your company.

Code is written once, then read 100 times. Understanding code is the fundamental task of a developer. Developers spend 60% of their time reading and understanding the company's code before they can ship new features.

But this makes it sound easy, like you can just start on page one of your code and read through to the end. You can't do that. Code isn't a story, it's a system, with dozens of moving parts coming together to form the whole. And just like any system, you need a guide to understand it–a schematic, a blueprint, a map–that lets you see the interaction between each file, function, or module so you can grok that whole.

That's what code visibility does. It lets you see the system as a whole so you can then start your journey into understanding it better. Giving this ability to your developers, new and old, will increase their productivity and ability, leading to more growth and a happier team.

Let's walk through how code visibility does this and see how it works in practice.

3 barriers to confident execution

Complexity is a built-in feature of most modern codebases. Let's say you create a new NextJS app. Before you've written a single line of your own, you'll have 17 files and over 500 lines of code. By the time you've built something valuable, those numbers could easily be three digits and five digits, respectively.

This isn't bloat; this is necessary code. Actual functioning codebases are big. But the size and complexity of these codebases come with costs.

1. Increased preparation time for both junior and senior developers

Let's play a game. Here's the repo for our Open Source Hub (if you have an open source project, we'd love you to add it!). Let's say you were a new developer working with us and we said "take a look at the codebase and we'll chat later."

Where do you even start

- Where's the entry point
- What's the most important feature
- Where's the data model?

If you have some knowledge of Remix, you might know that the home page is going to be in app/routes/index.tsx, but even then you've got forty lines of imports to work back through. You'll spend the entire morning clicking through files, trying to traverse the code line by line. You won't be gaining understanding; you'll just be navigating.

When that chat comes around, there's nothing to chat about. There's no insight from the new developer, and even if they were given a small bug to fix or a feature to implement, they'd still have to go back and go through the whole process again. And again, and again whenever something needs doing as they are never gaining a high-level understanding of the codebase.

Senior devs aren't immune to this either–they might have better knowledge of parts of the codebase, but they still don't have a higher-level view. Every team will have insight into part of the system, but never the whole.

GitKraken

This costs money in two ways

1. Developer time is expensive. If you are losing hundreds of hours to just preparation, you are losing 10x that in costs
2. You aren't shipping. Every day in prep is a day that a feature isn't shipped or a bug isn't fixed. This means less revenue.

2. Inefficient code production

Even once you get through to writing code, the problems persist with a lack of code visibility. Because of the overall lack of understanding of the codebase, developers lose the ability to efficiently write code that works together.

Let's go back to our Open Source Hub repo. Say an engineer has to add a new component to the home page. To do so, they have to

- Understand the dependencies of their new componen
- Understand the dependencies of the other components on the pag
- Understand the data model and how props and state are passed

Doable, but it takes time. Probably what a dev would do is copy a similar component and start hacking from there. This leads to bugs, which leads to more time needed to fix them.

Again, these engineering problems become business problems

- Updates or features are delayed
- Bugs are more likely, leading to lower customer satisfaction and more
- engineering time to deal with failure demand.

3. Wasted time in code reviews

Code is done; PR is submitted. Let's push to production and call it a day.

Well, first we have code review. This is the first of those 100 times your code is going to be read. Here the lack of visibility starts to spread across the team:

- For the developer, they have to think about how to explain all the impacts of this PR to the reviewer. They have to go into the mind of the reviewer to try and intuit the reviewer’s understanding and write, in prose, the changes made to guide them through the PR
- For the reviewer, they now have completely new code they have to assimilate into their own understanding of the codebase. What does this new code do? How does this component change? How will it impact other modules this dev doesn’t know about at scale?

More delay. They think all this and then merge it with “lgtm.” More bugs.

Code visibility unlocks growth and productivity

"...If I'm interacting with a given product/given project, and I need visibility into what parts of the code are responsible for doing what I am doing at a particular time, GitKraken really helps with that."

Ryan
Senior Engineer, Stripe

How to get past these problems? **Code visibility.**

Think about what people are really doing in the issues above. Developers are reading and traversing the code and building a mental map of how it all fits together. But there is only so much you can fit in your head at one time.

Code visibility takes that mental strain off your developers and presents them with an actual map of the codebase. Instead of having to concentrate on the map, they can concentrate on reasoning about the code.

When you provide this map to your team, everything can move quicker and more smoothly:

- You can **onboard developers to your codebase quicker** as they have a natural entry point, the ability to quickly traverse the codebase. They can easily see how the system works together. For Distribute Aid, 72 hours saved in onboarding resulted in an estimated $43,200 in humanitarian aid delivered
- You can **plan new features easier**, as missing functionality is more evident and it's clearer how new features will impact older code
- You can **streamline your code review process** as reviewers can see how the new code or changes impact the codebase. Again for Distribute Aid, over 20 hours per month in code review saved translates to $19,500 in additional humanitarian aid delivered each month.
- You can **refactor legacy code** using visibility to find code that is too dependent on other modules or code that is stale. P.volve got complete alignment on a 13.5K refactor in 30 minutes.

From a business perspective, this means lower costs and more revenue because you get the flipside of the outcomes above

- Faster prep means less engineering time and lower costs
- More shipping means more features, quicker to market, and more revenue
- Fewer bugs means happier customers.

But really, this is about giving your team a better work environment–they're the ones who then become more productive, saving you money and making you money. As Taylor, co-founder of Distribute Aid told us:

"We're a globally distributed team relying on highly collaborative dev techniques. GitKraken has helped Distribute Aid devs collaborate and stay in sync no matter their location or time."

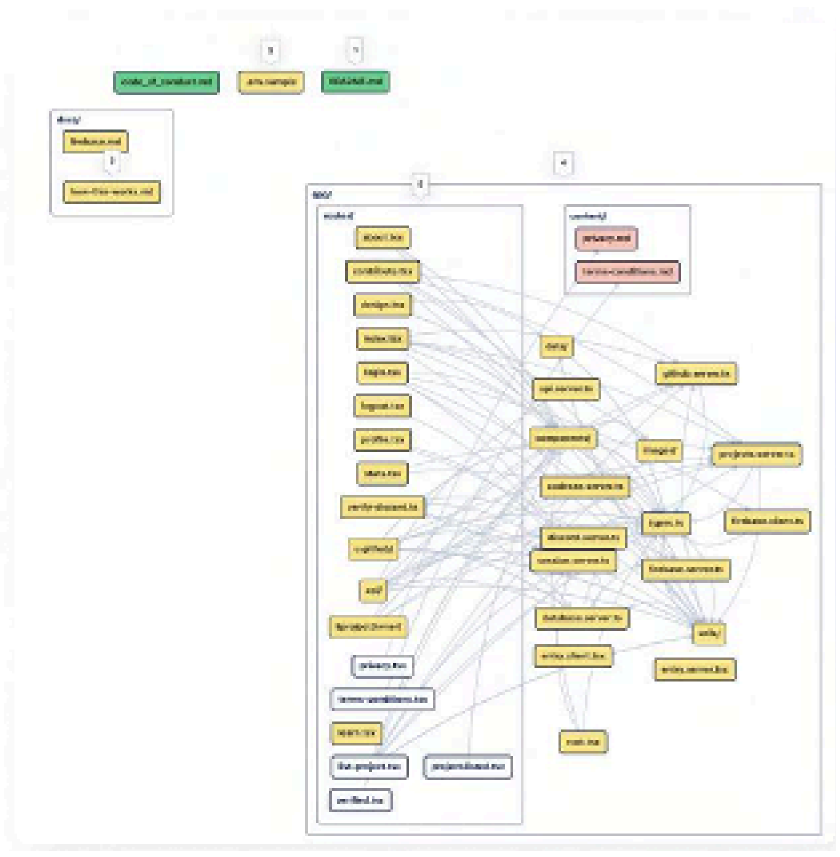
Taylor
Co-Founder, Distribute Aid

How to introduce code visibility into your dev process

Let's work through a couple of examples from above–onboarding new developers and refactoring legacy code–to show how this works.

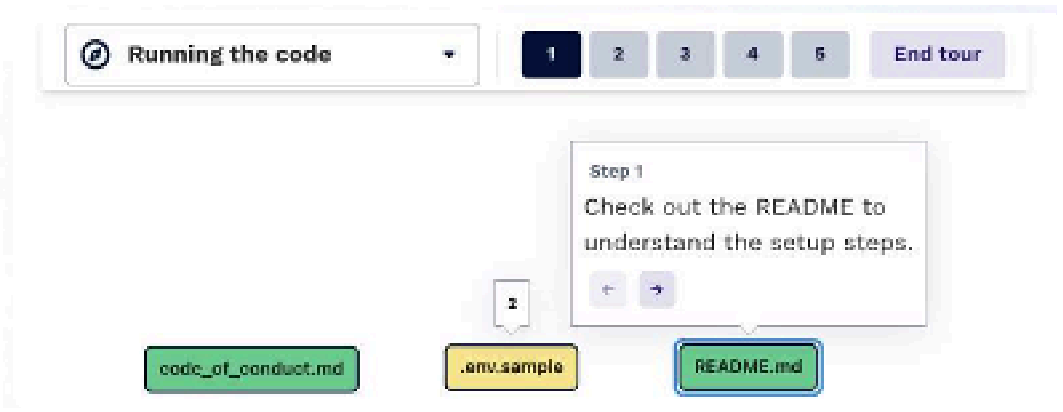
Let's go back to our new team member. Again, she needs to get up-to-date with the codebase. Without code visibility, we started by telling her -"take a look at the codebase and we'll chat later." Now, we can say "take a look at the code visualization and we'll chat later."

First, we'll send her a link to an onboarding code visualization. Immediately the new dev can see the entire codebase:



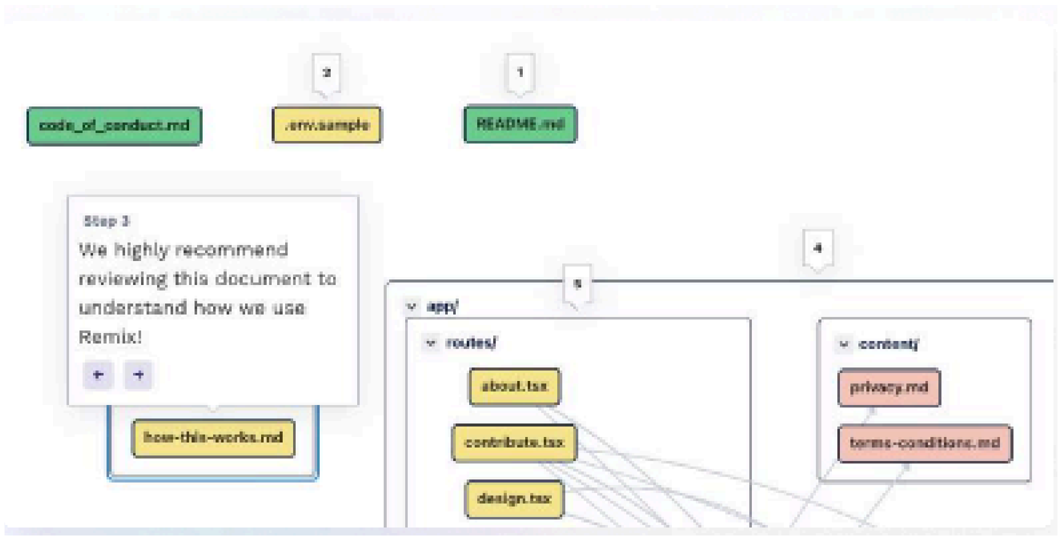
Even if this was it, it would still be more helpful. By visualizing the codebase, it lets the new dev see the structure and dependencies. But we can go deeper.

First, we can use a walkthrough to guide the onboarding developer through the code. If we select "1" in the walkthrough, we can see what the team think is the starting point for getting to know the code:



Step 1
Check out the README to understand the setup steps.

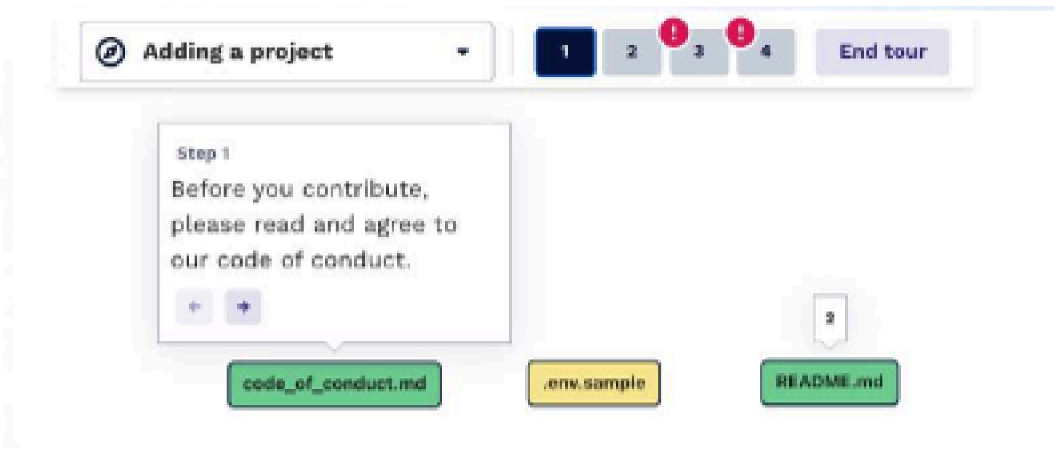
So now the new developer knows where to start to run the code. More than that, if she clicks through the walkthrough a little longer, she'll get to:



Step 3
We highly recommend reviewing this document to understand how we use Remix!

Awesome. Before, this was hidden in a subdirectory. With CodeSee, the onboarding developer can not only see this file more clearly, you can literally point her directly to it! She is now in a much better position for the chat this afternoon.

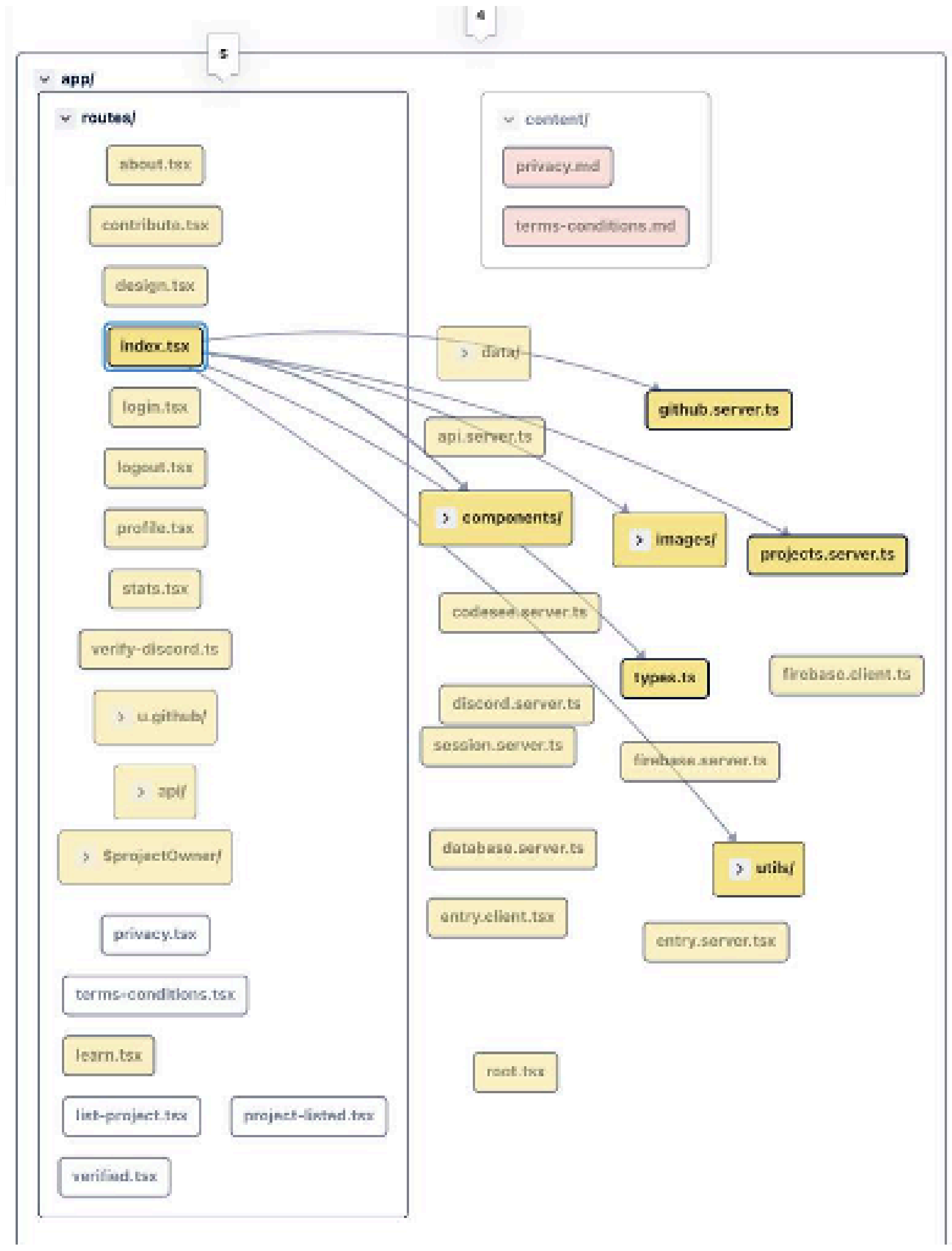
This is just the “Running the code” walkthrough. We can also set up different step- by-step guides for other workflows, such as adding a project:



Before you contribute, please read and agree to our code of conduct.

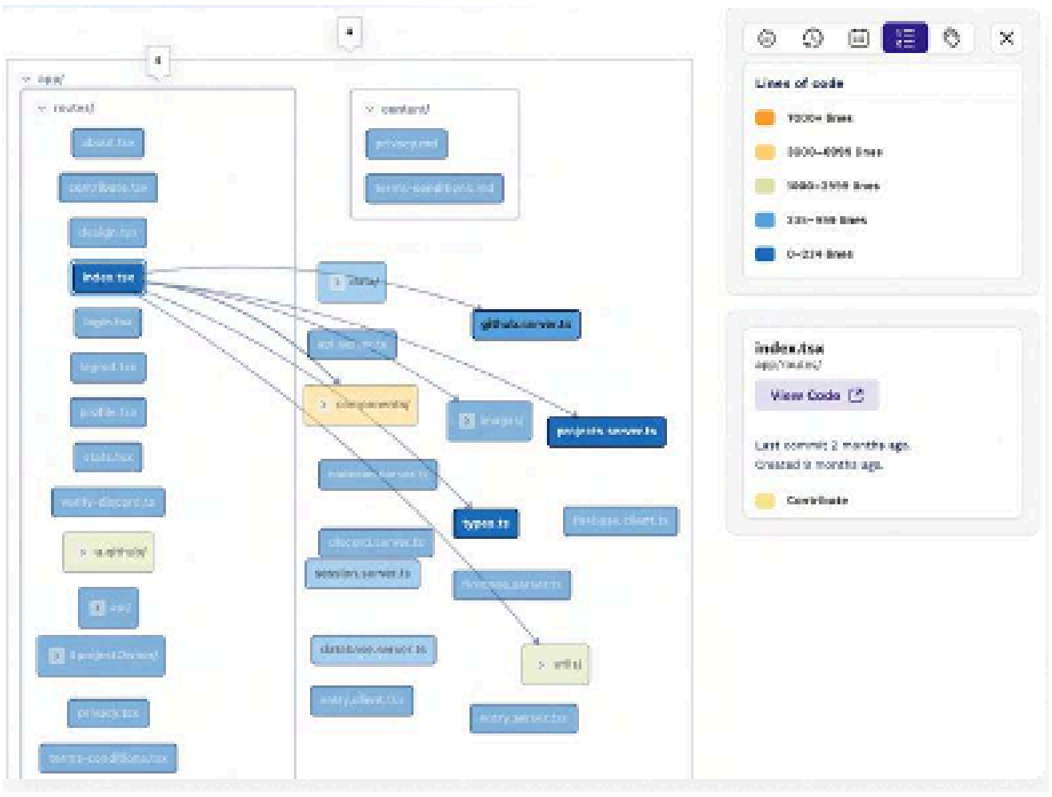
(Why the exclamation point? That tells you that walkthrough step no longer aligns with the code in your main branch.)

Even without the walkthrough, a new developer can get value out of code visualization. By selecting different files, they can see the dependencies that file has on others:



Like we said above, this is the homepage for a Remix project. Now the new developer can see everything that's being called from this entry point and has the map of interactions and dependencies laid out in front of her.

Another option for the new developer is to look at lines of code per file to see where the bulk of the work is:



The same goes for looking for engineering hotspots or latest activity to find where the momentum is in the project.

So, by telling a new team member "hey check out our code viz" instead of "check out our repo" we've gone from her scrambling around trying to understand the codebase to:

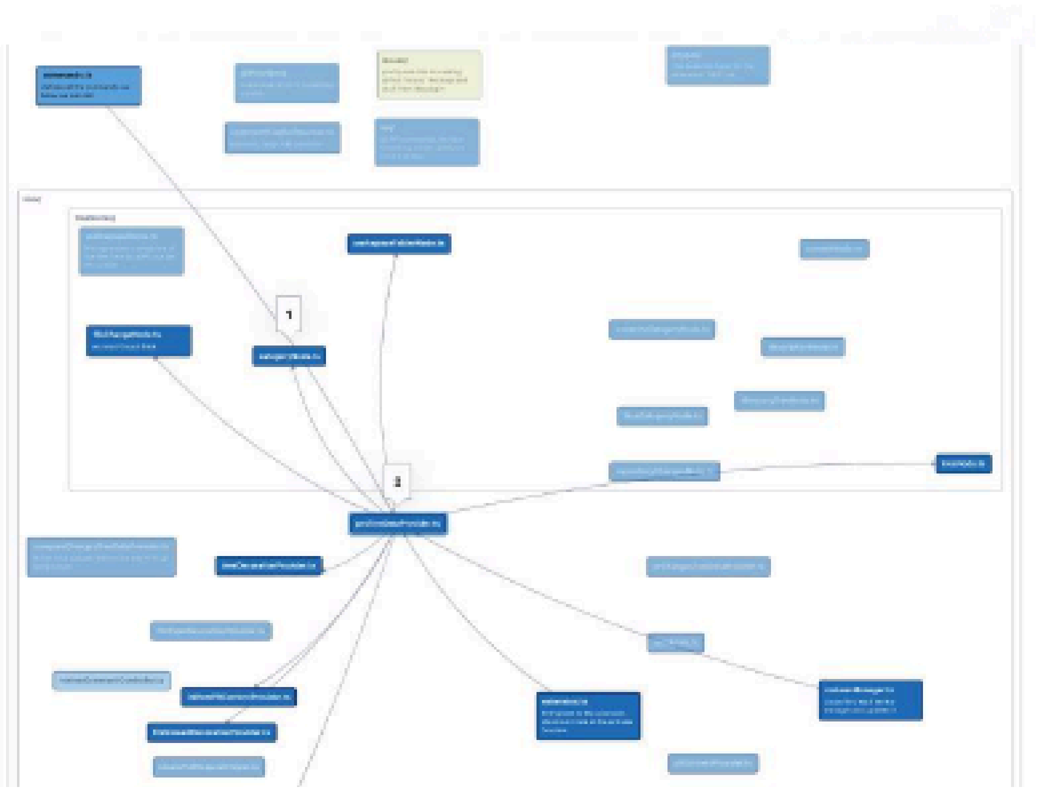
1. Immediately understanding the structure of the codebase
2. Having multiple walkthroughs for different workflows in the cod
3. Seeing individual dependencies within the cod
4. See what the team is currently working on

In this case, the new team member could even have contributed a new project by the end of their first day with minimal oversight. This is ideal for remote teams where you can't have someone sitting at your shoulder and guiding you through the code.

Let's take a look at another example. Now we have a more established team that has to refactor the code-base. Where to start?

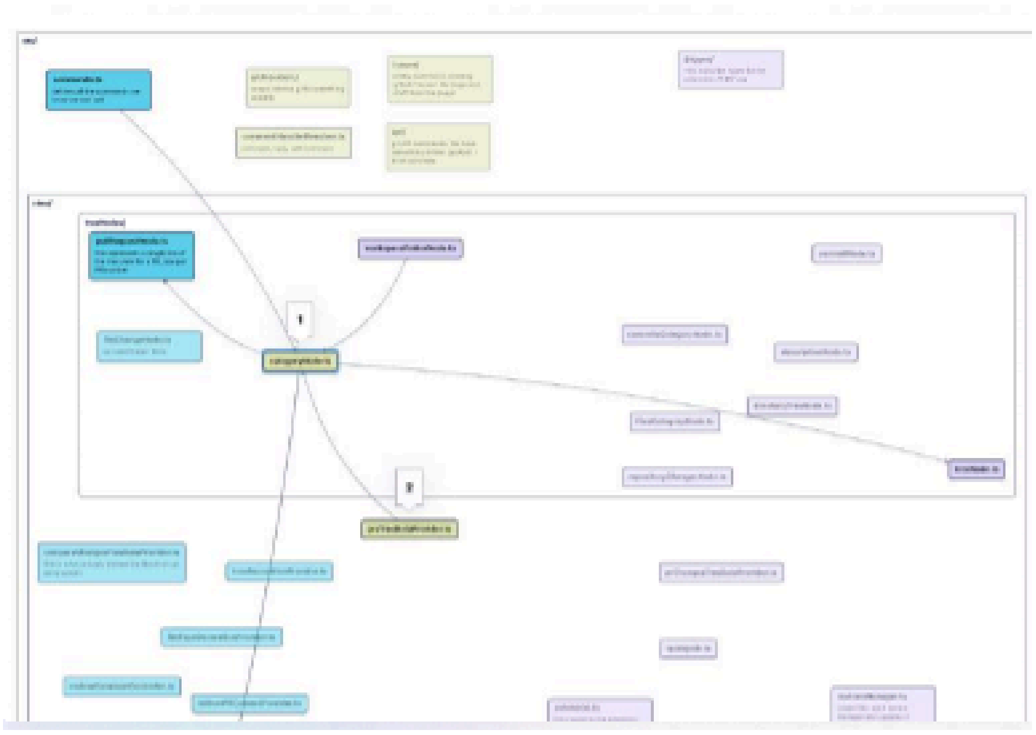
Without code visibility, it almost comes down to feels. Every engineer will probably have a different idea of what needs to be done. It'll come down to reading a lot of code to understand where the bottlenecks are.

With code visibility in a code refactor project, you can immediately see the potential issues within the codebase:



You can see the dependencies between each module so you can visualize the choke points. Here we have prsTreeDataProvider.ts which touches on a lot of other code all across the codebase. It's outside /treeNodes but touches four files within. It's throughout the files in /view, but also /common and /src. So we can have an immediate starting point for our refactoring.

Here's a few other ways we can help this process. The first is by using the visual aspects of code visibility further. In this codebase there are two main features, Pluto and Neptune. These features shouldn't be coupled. By color-coding by feature we can see this isn't true in our codebase:



categoryNode.ts is used by both so we should set it as a refactoring target.

Secondly, we can leave comments for each file to share our knowledge easily with other members of the team. As each member of the team will have different knowledge, kept in their head, in their notes, in their comments, or in their PRs, having a central repository for these more general notes is helpful. Then, the engineering manager can start to get a full understanding of the needs and assign the tasks for the refactoring.

Other options for finding good refactor targets are:

- Lines of code, looking for larger modules that can be split
- Creation date, looking for older files that can be remove
- Engineering hotspots and latest activity, looking for files that might be stale and can be updated

Code visibility gives time and energy back to your team to do the important work

Your developers don't want to spend hours going line by line through your codebase. They want to be productive. Code visibility gives them that opportunity.

When they don't have to read tons of code to get even the smallest understanding, and don't have to keep using the mental models of your codebase in their heads, they can spend their time thinking about the new feature and writing the new feature. When the weight of remembering how function A calls function B is gone, they'll have more energy to implement function C.

For the company, this only leads to good. Faster onboarding, faster shipping, better code, and happier engineers. All of which leads to happier customers and more growth.