Dotty (Scala 3) Compiler Visualization

Cao Nguyen Pham (354716)

Yaoyu Zhao (319801)

Yingtian Tang (368634)

May 29, 2025

This project aims to visualize the development progress, maintenance status, and adoption of new language features within the Dotty (Scala 3) compiler. Dotty is a compiler implementation of a mainstream programming language with hundreds of thousands of users, ranging from enterprise companies to academics to personal hobbyists. Unlike many other mainstream programming languages, Dotty is completely developed in the open: while a sizeable portion of the maintenance and experimental feature developement are done by LAMP (EPFL), the compiler welcomes and hugely benefits from external contributors: from ideas, proposals to implementation and bugfixes. On the other hand, Dotty is a complex project with multiple parts (a code compiler with multiple advanced features, a comprehensive language test framework, documentation tools, IDE integration tools,...) evolving rapidly over time.

Our goal is to provide an navigable interactive visualization of development and maintenance history across both axes: developers and feature modules (with granularity down to source files); as well as over the evolution of the project by both time and release timelines.

Our Paths

The three of us share a deep passion for Scala, especially its next-generation Dotty compiler; two are core members of the Scala team while one is an enthusiastic rookie. When we learned of this project we saw an ideal opportunity to dive deeper into our favorite subject by mapping its modular architecture, charting commit and review patterns, and uncovering the forces that drive its evolution. Combining veteran expertise with fresh curiosity, we brainstormed dependency graphs, debated which Git metrics would yield the most telling insights, and sketched interactive prototypes that transform raw version-control data into an immersive exploration of Dotty.

The First Step: Data

Our data sources come directly from the public Scala 3 GitHub repository (https://github.com/scala/scala3), focusing on three primary areas:

- Codebase: We analyze the code structure at the file level to compute metrics and infer relationships between different modules.
- Git Commits: We examine commit history, including author metadata, messages, timestamps, and code diffs, to trace the evolution of the codebase and identify contributor patterns.
- Issues and Pull Requests: Using the GitHub API (in JSON format), we collect information about issues and PRs to study lifecycle stages, contributor engagement, and maintenance workflows.

The Second Step: Statistics and Design

With the data in hand, we moved on to analysis and visualization design. Our first step involved extracting essential features using Python scripts, and storing the processed data in simple CSV format. This allowed for quick inspection and exploratory analysis using tools like Excel or pandas.

One of the main challenges we faced was the sheer volume of data. The raw GitHub API export, which includes full details of all issues and pull requests, amounted to several hundred megabytes and took considerable time to download and process. This size not only made analysis frustrating, but also exceeded Git's file size limits, making version control impractical. To address this, we implemented multi-stage filtering and transformation pipelines, ultimately reducing the dataset to a manageable and relevant subset.

Informed by our statistical findings and experience working with the Scala ecosystem, we sketched several visualization concepts aimed at serving the community. These included a network graph illustrating internal module dependencies within the compiler and a treemap that visualizes the structure and relative size of different source files in the Scala 3 codebase.

The final step: Making the visualization!

Once the design was finalized and tasks were split, we began development in parallel.

We chose to build our visualization platform using npm and Parcel, which allowed us to quickly develop responsive, interactive web pages using only JavaScript and HTML. Parcel's live-reload feature enabled rapid feedback during development by instantly reflecting code changes in the browser.

For the visualizations themselves, we primarily relied on D3.js for custom, data-driven diagrams and Chart.js for more standard charts. Together, these tools gave us the flexibility to create both exploratory visualizations for developers and polished summaries for broader audiences.

Challenges

While our pipeline and dashboard now offer deep insights into the Dotty compiler's evolution, we faced several significant challenges during development:

Temporal Complexity: Designing intuitive time controls that synchronized across multiple panels—each with its own visualization type—proved non-trivial. We iterated on the granularity of time sliders, playback speed settings, and transition easing functions to strike a balance between precision and usability.

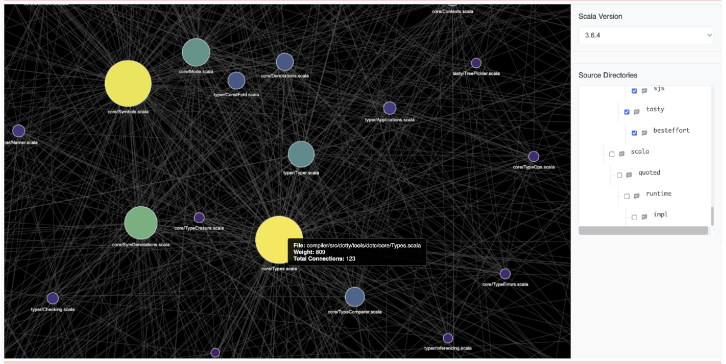
Data Quality and Linking: Analyzing commits to GitHub issues and pull requests based solely on commit messages introduced noise due to inconsistent conventions. We developed a hybrid matching approach combining regex patterns, commit metadata, and manual tag mappings to improve accuracy.

Maintaining Developer-Centric UX: Balancing the inclusion of advanced features—such as animated playback and deep drill-downs—with a clean, uncluttered interface required user testing and iterative design reviews. We prioritized feature discoverability and provided contextual help to guide new users through the dashboard's capabilities.

Visualization Content

Our visualization is composed of five panels: a main overview panel that provides project descriptions and useful links, along with four interactive panels corresponding to distinct diagram types. Each panel offers a different lens into the structure and history of the Scala 3 (Dotty) compiler project.

Files and Dependencies



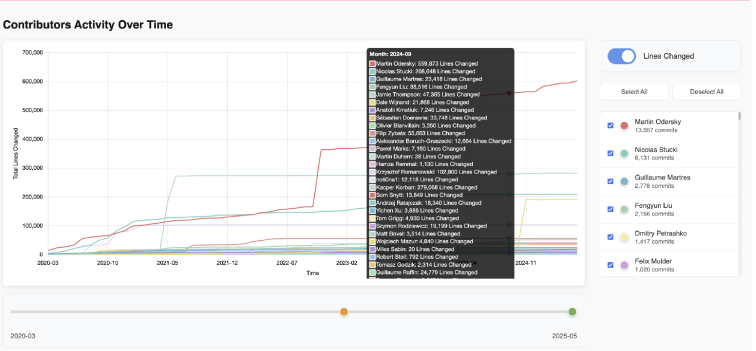
This panel features a network graph that visualizes the dependencies among a selected subset of source files within the compiler. Each node represents an individual file, with its size proportional to the number of incoming references—i.e., how frequently it is used by other files.

On the right-hand side, users can interactively select which files to include in the graph. Utility files can be filtered out to avoid clutter. A dropdown allows users to choose a specific release version, enabling exploration of the code structure at different points in time.

By focusing on a specific module, users gain insights into the internal structure and coupling between files. Hovering over any node displays a tooltip and provides information to the corresponding source file for in-depth examination.

An interesting example is the capture checking module (compiler/src/dotty/tools/dotc/cc), a relatively recent addition to the compiler. In earlier versions, this module is absent from the graph. As development progresses, its files begin to appear and grow, illustrating how a new language feature gradually takes shape within the codebase, also reflecting the priorities of the Scala development team.

Contributors Activity

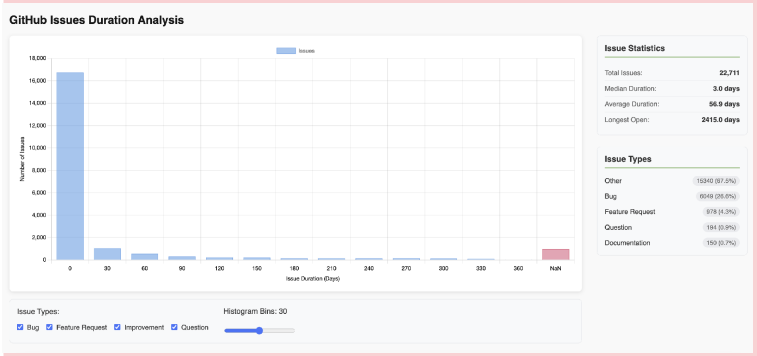


This panel presents an accumulated line plot showing contributor activity over the full history of the project. It visualizes either the number of commits or the number of lines changed per contributor over time.

Each contributor is represented by a uniquely colored line. A slider at the bottom allows users to adjust the visible time window interactively. The plot updates dynamically as the window is modified. A toggle switch on the right lets users choose between viewing commit counts or line changes. An additional menu allows users to include or exclude specific contributors from the plot. Hovering over the chart reveals a tooltip that displays the distribution of contributions at a given point in time.

Notably, Professor Martin Odersky, the creator of Scala, consistently appears at the top of the contributor list. Switching to the "line changes" view reveals sudden "cliffs" that typically correspond to major syntactic updates or the integration of new features into the compiler.

Issues Duration and Status

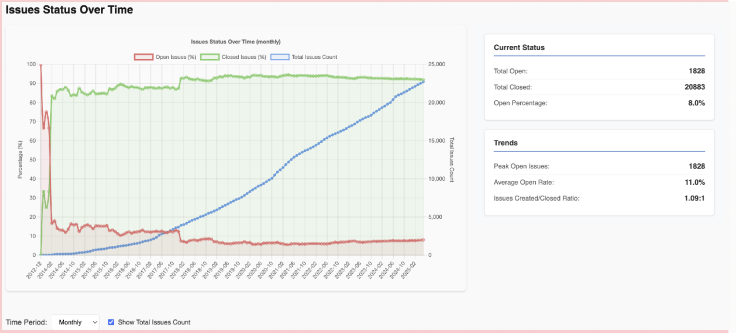


The first chart is a histogram that shows the duration of issue resolution—from when an issue is opened to when it is closed. Users can configure the bin size to analyze responsiveness across different time scales (e.g., weekly, monthly). This view helps illustrate how quickly the development team addresses user-reported issues.

In our analysis, we observe that over 50% of issues are resolved within the first week, and more than 70% within the first month, reflecting a high degree of responsiveness from the development team.

The second chart tracks the proportion of open versus closed issues over time, segmented by month, quarter, or year. This time series offers a broader view of the project's maintenance lifecycle. A consistently low ratio of open issues, despite a growing total number of reports, indicates ongoing attention to user feedback and technical debt.

Together, these two charts offer strong evidence of the Scala compiler's sustained health and developer engagement over time.



Peer Assessment

Cao Nguyen Pham initiated the project by analyzing file changes and author statistics, revealing key development patterns. He then designed and implemented the Files and Dependencies network visualization. Leveraging his Scala knowledge, he wrote tools extract useful information from the compiled code. He also refactored the original web framework into a scalable parcel project for the final deployment.

Yaoyu Zhao designed and implemented the Contributors, Issue Timeline, and Issue Status Over Time visualizations. These panels offer an interactive view of contributor dynamics and issue management. He also developed a data pipeline that converts Git logs into a structured format, enabling efficient analysis.

Yingtian Tang built the initial web framework that supported all sketch visualization panels. He analyzed module-level interactions within the codebase, laying the foundation for the Files and Dependencies graph. Although new to Scala, his outside perspective helped make the visualizations more accessible to a broader audience. He also compiled the final project report, including annotated figures, and summaries.