Dotty (Scala 3) Compiler Visualization

Cao Nguyen Pham (354716), Yaoyu Zhao (319801), Yingtian Tang (368634)

May 30, 2025

Abstract :

This project aims to visualize the code structure, development progress, and maintenance status of the Dotty (Scala 3) compiler. Dotty is the next-generation compiler for the Scala programming language, used by a wide range of developers—from large enterprises and academic institutions to individual hobbyists. As an open-source project hosted on GitHub, Dotty benefits from a large and diverse contributor base. While core development is led by the LAMP group and the Scala Center at EPFL, hundreds of external contributors—both individuals and industry professionals—have actively shaped the project. With hundreds of thousands of lines of code and multiple interconnected components (such as the core compiler, language server, and documentation generator), Dotty is a large and complex system. Maintaining such a project requires continuous collaboration among developers, who must simultaneously implement new features and respond to user-reported issues in a timely manner.

Our goal is to create an interactive, navigable visualization of the project's development and maintenance history—mapped across two key dimensions: contributors and functional modules (down to the level of individual source files). This visualization also captures the project's evolution over time and across official release milestones.

Modern development is a complex and collaborative endeavor, especially in the context of large-scale open source projects. Dotty, the reference implementation of the Scala 3 compiler, encounters these challenges. Designed as the successor to the Scala 2 compiler, Dotty introduces numerous innovations and simplifications to improve expressiveness, safety, and tooling support. As such, it serves not only as a critical tool for the Scala ecosystem but also as a platform for language research and experimentation.

As an open-source project hosted on GitHub, Dotty benefits from a large and diverse contributor base. While core development is led by the LAMP group and the Scala Center at EPFL, hundreds of external contributors, both individuals and industry professionals, have actively been maintaining the project.

The Dotty code base is extensive, consisting of hundreds of thousands of lines of code organized into multiple interdependent components. These include the core compiler, a language server for IDE integration, a documentation generator, various supporting tools and libraries, etc. Managing such a project involves more than just writing code; it requires tracking contributions, managing issue reports, reviewing and integrating pull requests, and maintaining consistency across releases. This model of collaboration has proven effective, but also introduces significant complexity in project management, coordination, and long-term maintenance.

Gaining a comprehensive understanding of its internal structure, contributor dynamics, and maintenance status remains a nontrivial task. Existing tools like GitHub Insights offer high-level metrics, but they lack granularity, customization, and are limited by project size.

To address this gap, this project presents an interactive visualization platform that maps the development and maintenance history of Dotty along multiple axes. Our visualization is structured to support following topics:

- Code structure: We will illustrate the connection between components in the compiler.
- Git Commits: Analysis will include commit history data such as author details, commit messages, timestamps, and code changes to track the development process.
- Issues and Pull Requests: We will extract issues and pull requests via GitHub API (JSON format) to analyze issue lifecycle, contributor engagement, and maintenance workflows.

Feature and module granularity: Which files and components have been actively developed or maintained? How does this activity correlate with the release cycle? Temporal dimensions: How has the project evolved chronologically? What are the patterns and rhythms of development around major releases?

In the following sections, we describe our methodology for extracting, processing, and visualizing the relevant data. We also present case studies and examples that illustrate the utility of our platform in understanding and maintaining a complex compiler project like Dotty.

Our Paths

The three of us share a deep passion for Scala, especially its next-generation Dotty compiler. Two members of our team are part of the Scala core development group, while the third brings a fresh perspective as an enthusiastic newcomer to the community.

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.

Dotty is open source under the Apache License Version 2.0, and all our data sources are publicly accessible. To collect this data, we use Git for commit history and the GitHub API for issues and pull requests. Once retrieved, the raw data is cleaned and preprocessed using custom scripts before visualization. For the source code, we wrote a Scala program to read the tasty files of the compiler, to figure out the dependencies.

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!

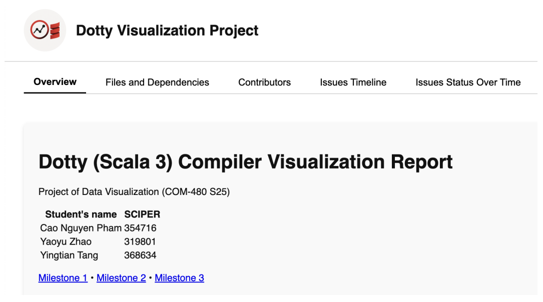
Once the design was finalized and tasks were split, we began development in parallel.

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.

Visualization Content

TODO: insert a screenshot of the main page.



Dotty Visualization Project

Dotty (Scala 3) Compiler Visualization Report

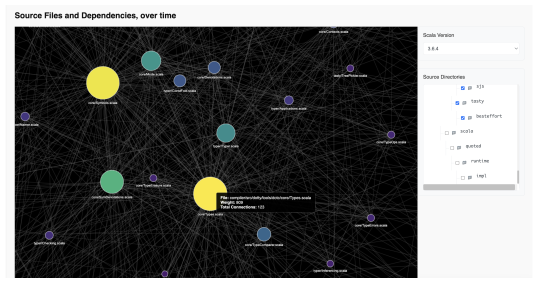
Project of Data Visualization (COM-480 S25)

Student's name	SCIPER
Cao Nguyen Pham	354716
Yaoyu Zhao	319801
Yingtian Tang	368634

Milestone 1 • Milestone 2 • Milestone 3

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



Source Files and Dependencies, over time

File: compiler/src/dotty/tools/dotc/core/Types.scala
Weight: 809
Total Connections: 123

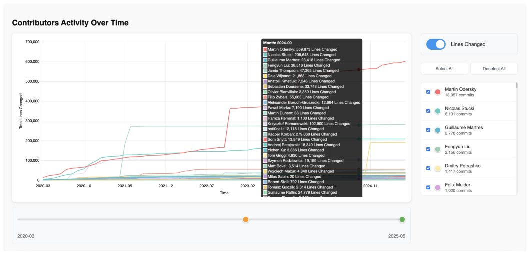
This panel features a network graph that visualizes the dependencies among a selected subset of source files within the compiler. Each node represents an individual file, with its size proportional to the number of incoming references—i.e., how frequently it is used by other files.

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



Contributors Activity Over Time

Month: 2024-09
Martin Odersky: 559,873 Lines Changed
Nicolas Stucki: 208,648 Lines Changed
Guillaume Martres: 23,418 Lines Changed
Fengyun Liu: 38,516 Lines Changed
Jamie Thompson: 47,365 Lines Changed
Dale Wijnand: 21,868 Lines Changed
Anatolii Kmetiuk: 7,246 Lines Changed
Sébastien Doeraene: 33,748 Lines Changed
Olivier Blanvillain: 3,350 Lines Changed
Filip Zybała: 55,663 Lines Changed
Aleksander Boruch-Gruszecki: 12,664 Lines Changed
Paweł Marks: 7,190 Lines Changed
Martin Duhem: 38 Lines Changed
Hamza Remmal: 1,130 Lines Changed
Krzysztof Romanowski: 102,900 Lines Changed
noti0na1: 12,118 Lines Changed
Kacper Korban: 279,068 Lines Changed
Som Snytt: 13,849 Lines Changed
Andrzej Ratajczak: 18,340 Lines Changed
Yichen Xu: 3,886 Lines Changed
Tom Grigg: 4,930 Lines Changed
Szymon Rodziewicz: 19,199 Lines Changed
Matt Bovel: 3,514 Lines Changed
Wojciech Mazur: 4,840 Lines Changed
Miles Sabin: 20 Lines Changed
Robert Stoll: 792 Lines Changed
Tomasz Godzik: 2,314 Lines Changed
Guillaume Raffin: 24,779 Lines Changed

Martin Odersky 13,057 commits
Nicolas Stucki 6,131 commits
Guillaume Martres 2,778 commits
Fengyun Liu 2,156 commits
Dmitry Petrashko 1,417 commits
Felix Mulder 1,020 commits

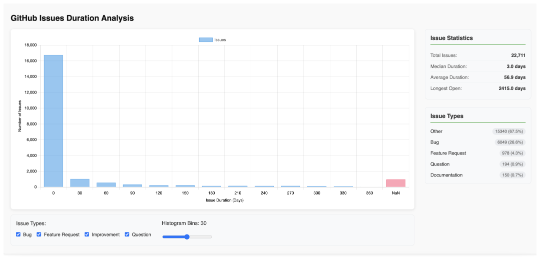
This panel presents an accumulated line plot showing contributor activity over the full history of the project. It visualizes either the number of commits or the number of lines changed per contributor over time.

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 final two panels focus on project maintenance and responsiveness—key indicators of the project's overall health.



GitHub Issues Duration Analysis

Issue Statistics

Total Issues:	22,711
Median Duration:	3.0 days
Average Duration:	56.9 days
Longest Open:	2415.0 days

Issue Types

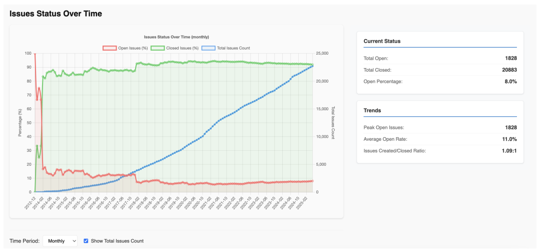
Other	15340 (67.5%)
Bug	6049 (26.6%)
Feature Request	978 (4.3%)
Question	194 (0.9%)
Documentation	150 (0.7%)

Issue Types: ✓ Bug ✓ Feature Request ✓ Improvement ✓ Question

Histogram Bins: 30

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.



Issues Status Over Time

Current Status

Total Open:	1828
Total Closed:	20883
Open Percentage:	8.0%

Trends

Peak Open Issues:	1828
Average Open Rate:	11.0%
Issues Created/Closed Ratio:	1.09:1

Time Period: Monthly ✓ Show Total Issues Count

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.

1 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.