INSTITUT FÜR INFORMATIK
Ludwig-Maximilians-Universität München

A Web-First Software Analysis Tool for Visualizing Class Dependencies through Force-Directed Graph Layouts

Amos Mohr

Bachelor Thesis

Supervisor	Prof. Dr. Dirk Beyer
Mentor	Thomas Lemberger
Submission Date	October 4, 2025

Statement of Originality

English:

Declaration of Authorship

I hereby declare that I have written this work independently and only with the aids specified. All passages that I have taken from literature or other sources have been clearly marked as quotations with reference to the source. I used ChatGPT to improve wordings of single sentences and small paragraphs, and to suggest small snippets of code for evaluation purposes.
This printed copy is a printout of the submitted electronic copy.

Deutsch:

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig und nur mit den angegebenen Hilfsmitteln verfasst habe. Alle Passagen, die ich aus einer Literatur oder anderen Quellen übernommen habe, sind eindeutig als Zitate mit Quellenangabe gekennzeichnet. Ich habe ChatGPT verwendet, um Formulierungen für einzelne Sätze und kleine Absätze zu erstellen und zu verbessern, und um kleine Codefragmente für Auswertungen vorzuschlagen.
Diese gedruckte Fassung ist ein Ausdruck der eingereichten elektronischen Fassung.

München, October 4, 2025 Amos Mohr

Abstract

Visualizing class dependencies is essential for understanding complex software systems. While CCVisu provides powerful algorithms for clustering inter-class dependencies, it relies on outdated Java Swing interfaces, offers limited interactivity, and includes redundant functionality. This thesis presents CCVisu-Web, a modern web-first alternative that implements the same force-directed graph layouts using Next.js and sigma.js. Our solution features a completely redesigned user interface that improves accessibility and usability. For the sophisticated core layout algorithms, we employed generative AI to efficiently translate CCVisu's Java implementation to TypeScript. Using the CPAchecker project as a case study, we demonstrate that our TypeScript implementation produces layouts virtually identical to the original, with minor deviations attributable exclusively to floating-point precision differences. This work showcases the design decisions and the enhanced functionality of CCVisu-Web, illustrating how AI-assisted code translation can rapidly modernize legacy software while creating more accessible, web-native analysis tools.

Acknowledgements

First and foremost, I would like to thank Prof. Dr. Dirk Beyer for giving me the opportunity to write this thesis at the Software and Computational Systems Lab.

I am especially grateful to Dr. Thomas Lemberger, whose mentoring, feedback, and expertise greatly improved the quality of this work, and whose unwavering support and understanding made the journey truly rewarding.

Finally, I would like to thank my family and friends for their ongoing support, patience, and encouragement throughout this journey.

Contents

List of Figures

List of Tables

Listings

1 Introduction

Understanding the structure and dependencies of large software systems is essential for maintaining, extending, and debugging complex source code. As projects grow in size and complexity, developers, non-technical stakeholders, and researchers rely on visualization tools to gain insight into how classes and modules interact. CCVisu [7] provides a powerful workflow for visualizing such dependencies using force-directed layouts and energy-based clustering. However, despite its algorithmic strengths, CCVisu remains a legacy desktop application with substantial usability limitations.

The original Java Swing interface of CCVisu proves difficult to navigate and slows down the analysis workflows. For example, zooming requires manually drawing a selection rectangle, while zooming out is hidden deep in a submenu with no keyboard shortcut. Graph labels frequently overlap and become unreadable, and many interface elements are unintuitive or inconsistent. Users often find that the

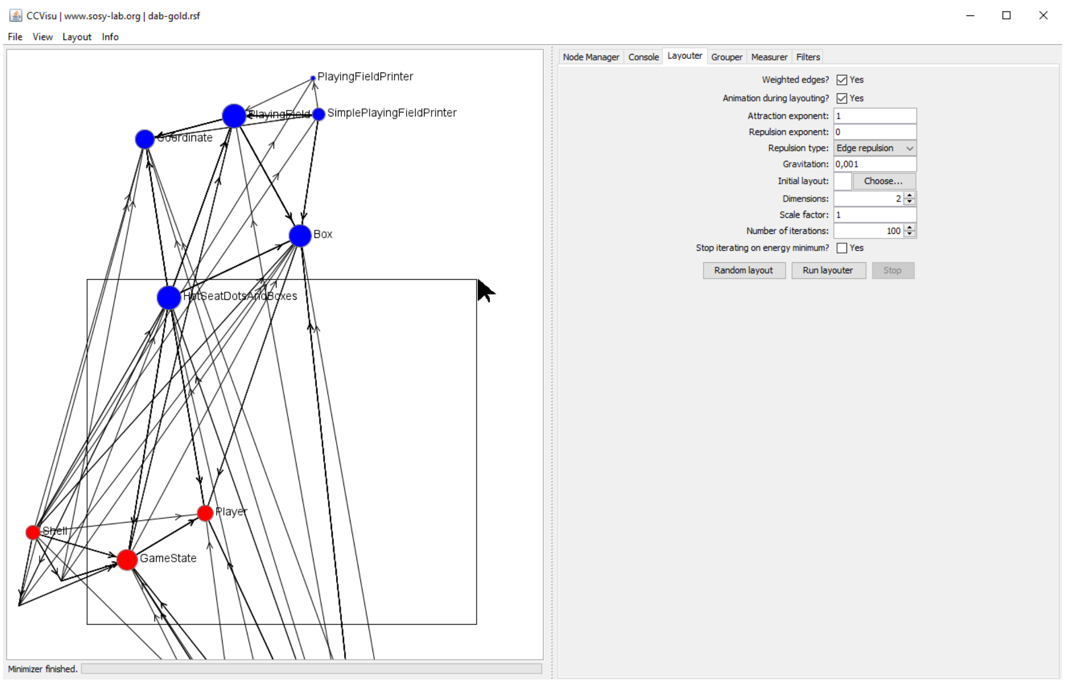

Figure 1.1: Old CCVisu 3.5 GUI demonstrating zoom

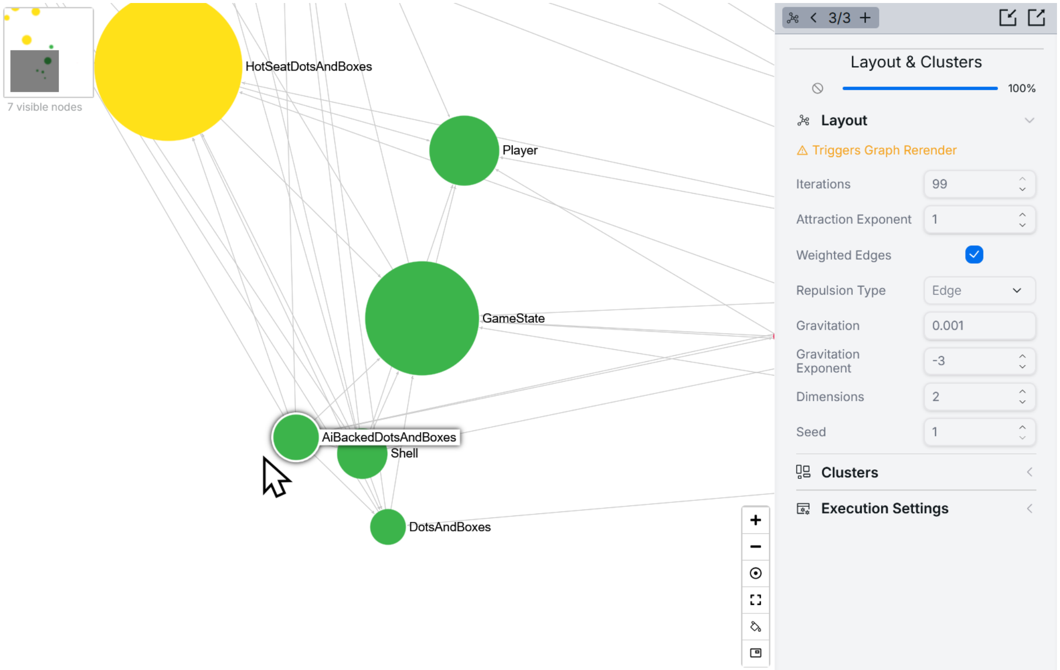

Figure 1.2: New CCVisu-Web showing node highlight on hover

fastest way to continue working is to simply restart the program when confronted with overlapping pop-out windows or unclear menu interactions. Additionally, redundant settings are distributed across different menus, forcing users to re-enter the same parameters multiple times. The overall workflow for generating and visualizing a source code graph in CCVisu is manual and non-trivial: users must first configure and invoke Doxygen [12] to produce XML files, convert them into the Relational Standard Format (RSF), and finally run CCVisu via the command line to generate the layout before viewing it in the GUI. Figure 1.1 shows CCVisu's original interface and highlights how zooming works in the graph viewer.

This thesis presents CCVisu-Web, a modern, web-first reimplementation of CCVisu that preserves its analytical strengths while offering a significantly improved user experience. We build the new application with Next.js [28], sigma.js [30], and Graphology [18] to deliver a fully interactive, browser-based interface. Users can analyze any Java project directly from a Git repository URL, requiring no manual setup or local installation. The interface synchronizes all layout settings with the displayed visualization in real time, removing the need for "Apply" dialogues and redundant prompts. Features such as smooth zoom and pan, node highlighting on hover, a minimap for navigation, and a snapshot system for managing multiple layouts make software exploration more intuitive and accessible. We introduce a random seed option that makes layouts reproducible and comparable, unlike the original CCVisu, which starts from the current or a new random configuration. Figure 1.2 shows the new frontend and highlights the improved readability when hovering a node.

A notable technical contribution of this work is the AI-assisted translation of CCVisu's Barnes–Hut layout algorithm from Java to TypeScript, enabling layout computation directly in the browser. We use generative AI (ChatGPT and Le Chat) to translate over 800 lines of performance-optimized Java code with minimal manual intervention. The resulting implementation achieves near-identical layouts to the original Java algorithm, with deviations explained solely by floating-point precision differences. Our evaluation on the large open-source project CPAchecker confirms that the translated algorithm maintains correctness while improving responsiveness and workflow efficiency through client-side execution.

This work contributes both a modernized user interface and a validated AI-assisted code translation process. It demonstrates how generative AI can accelerate legacy software modernization, producing a web-native tool that enhances usability and accessibility.
Through our evaluation on CPAchecker, we can answer the following research questions with yes:

RQ 1. *Can CCVisu's layouting and minimisation Java algorithm be replicated in TypeScript with the help of AI translation?*

RQ 2. *Is client-side execution of the new layouting and clustering TypeScript algorithms faster than running the CCVisu CLI on the backend of the web app?*

RQ 3. *Can the AI-translated implementation be integrated into the existing system with only minimal manual changes?*

Our results show that less than 10% of the AI-generated code requires manual correction and that client-side execution reduces user wait times by over a factor of ten when enabling real-time layout updates. Together, these findings confirm the feasibility and advantages of using AI to revive and modernize legacy analysis tools for today's web ecosystem.

2 Related Work

Present software systems tend to grow in size and complexity over time [2], making static analysis and visualization increasingly vital. At the same time, developers often work across multiple projects and switch contexts frequently [38], which further supports the need for tools that provide clarity. Visualization of software projects has a long history, and many approaches exist that emphasize different aspects.
Our work sits at the intersection between class dependency extraction, force-directed layouting, and web-based visualization. In the following, we outline the alternative branches we could have taken but deliberately did not, and explain why.

Legacy Tools for Software Visualization. Our main inspiration is CCVisu [7], which demonstrates the value of energy-based clustering but suffers from an outdated Swing GUI and limited interactivity. Other legacy visualization systems include Rigi [34] and LSEdit [35], which focus on reverse engineering and system landscapes. While they certainly paved the way for structural analysis, they rely on desktop environments with steep learning curves. JArchitect [19] and Understand [31] are commercial successors that combine metric analysis with visualization, but their visual output remains close to UML diagrams rather than interactive layouts.

Alternative Metaphors. A large branch of work represents software as cities. CodeCity [39] pioneers this approach, representing classes as buildings and packages as districts. Inspired works, for example, explore virtual reality representations [21] and the addition of streets and different building types [10]. Other metaphors include forests [13], where trees encode classes with attributes such as size or complexity, and particle systems [29], which model software entities as interacting physical objects. Further visualization forms include treemaps [3] and calling context ring charts [22]. These techniques highlight the diversity of software visualization but differ fundamentally from our path, which emphasizes interactive force-directed layouts.

Layout Algorithms. General-purpose graph libraries such as graph-tool (Python) [37], Graphology [18], or Cosmos [27] (JavaScript) also offer built-in layouts, but not specifically optimized for software dependency graphs. We adopt the Barnes–Hut [4] minimization algorithm as implemented in CCVisu because it combines specialization for software graphs with interpretability and enables a direct comparison to CCVisu.

Web-Based Visualization. We use CCVisu with Doxygen [12] on the backend and sigma.js [30] together with Graphology [18] on the frontend. In this area there are a number of tools that implement similar techniques. General-purpose visualization libraries such as D3.js [8] and VisX [1] provide flexible charting components, but offer limited out-of-the-box support for large network graphs. ReaGraph [17] and Nivo [24] include network layouts but do not expose positional node attributes, making it unclear how to apply our own force-directed layout algorithm. Cosmos.gl [27] demonstrates high-performance particle-based graph visualization, yet its underlying simulation tightly couples with the layout generation. Finally, Gephi Lite [5] also relies on sigma.js for rendering but serves primarily as a general-purpose network exploration tool, without integrated support for extracting and visualizing source code graphs.

AI-Based Tools. More recently, CodeViz [9] and FalkorDB's Code Graph [14] use AI-based querying for software structure visualization through a chat interface, but they focus on Python and databases or are proprietary, with limited layout control compared to our approach. Also, they combine static analysis with visualizations that emphasize UML-like diagrams or tree structures rather than dynamic clustering.

3 Background

3.1 Doxygen

Doxygen [12] is an open source tool for generating documentation from source code comments across multiple programming languages. To this end, it extracts information such as dependency data about classes, functions and variables. Using its *Doxyfile* configuration file, Doxygen supports a number of different output formats, such as HTML, PDF and XML.

Basic Relations. The XML output from Doxygen contains relational data about inter- and intra-class dependencies between classes of the examined source code. CCVisu uses this data to construct a graph representation of the source code [11]. Doxygen recognizes the following relations:

- `COMPOUND <kind> <name>`
- `MEMBER <kind> <name>`
- `CONTAINEDIN <member> <compound>`
- `BASEDON <compound> <basecompound>`
- `REFERSTO <member> <referredmember>`
- `LOCATEDAT <compound|member> <file-path> <line-no>`

Listing 1 demonstrates how and where Doxygen extracts these basic relations from the source code.
First, `COMPOUND` identifies a compound of `kind` class, interface, enum, package, file or directory, with a given `name`. An example for this relation is: `COMPOUND class Coordinate`. On the other hand, `MEMBER` primarily refers to an occurrence of the programming construct of `kind` variable or function, for example: `MEMBER function toString`. The `CONTAINEDIN` relation expresses what `COMPOUND` a specific `MEMBER` belongs to. To continue our example:
`MEMBER toString Coordinate`. Furthermore, `BASEDON` represents inheritance, e.g. *basecompound* is the parent class of *compound*. A crucial relation is `REFERSTO`, which illustrates a function referring to another function or variable in an expression or statement. Last, `LOCATEDAT` expresses where in a file-context the compound or member was extracted from by Doxygen. `LOCATEDAT Coordinate org/example/Coordinate.java 1` for example expresses that Doxygen finds class `Coordinate` on the first line of file `org/example/Coordinate.java`.

```
// COMPOUND class Coordinate
// BASEDON Coordinate CoordinateBase
public class Coordinate implements CoordinateBase {
  // MEMBER variable row
  private final int row;
  // MEMBER variable column
  private final int column;

  // MEMBER function Coordinate()
  public Coordinate(int row, int column) {...}

  // MEMBER function row()
  public int row() {
    // REFERSTO Coordinate.row() Coordinate.row
    return this.row;
  }

  // MEMBER function column()
  public int column() {
    // REFERSTO Coordinate.column() Coordinate.column
    return this.column;
  }

  // MEMBER function toString()
  public String toString() {
    // REFERSTO Coordinate.toString() CoordinateBase.stringify()
    return this.stringify(this.row, this.column);
  }
}

/*
 Additional Relations:
 CONTAINEDIN Coordinate.row Coordinate
 CONTAINEDIN Coordinate.column Coordinate
 CONTAINEDIN Coordinate.Coordinate() Coordinate
 //...

 LOCATEDAT Coordinate.row Coordinate.java 5
 //...
*/
```

Listing 1: Doxygen relations using the example Coordinate class

3.2 CCVisu

CCVisu [7] is a tool for visualizing software graphs by structuring classes into separated, highly cohesive clusters. The generated graph layouts reveal the degree of relatedness between classes through interpretable distances.

CCVisu provides a command-line interface (CLI) as well as a graphical user interface (GUI) for dynamic interaction with graphs. The CLI in particular highlights CCVisu's design as a reusable software component, integrating well with other tools.

CCVisu uses the Relational Standard Format (RSF) for graph input. It converts source code to this format using Doxygen's XML output. CCVisu also supports co-change graphs, which incorporate version history data [6].

3.2.1 Relational Standard Format

The Relational Standard Format (RSF) represents graphs as a list of relations (edges). Each edge follows the following format:

```
<relation name> <source> <target> <weight>
```

For example, `REFERSTO FOO BAR 2.0` defines an edge with weight 2.0 between the nodes `FOO` and `BAR`.
CCVisu converts Doxygen's XML to RSF and derives additional relations:

- `REFvariable`
- `REFfunction`
- `refFile`
- `refClass`
- `implements`

Notably, `refClass` relation, derived from `REFERSTO`, expresses class dependencies. It is important to note that this approach does not capture every class usage. For example, it misses newer Java constructs such as records (introduced in Java 14) and partially overlooks references via method parameters. As a result, it does not captures all dependencies that tools like IntelliJ identify as usages. This remains an open issue. Despite these limitations, the RSF-based approach still provides a useful foundation for analyzing class dependencies because it captures inheritance, direct function calls, and field references, which form the core structural relationships in software graphs.

The obtained RSF file parses into a *Graphology* graph by first adding a node for each identified unique source or target, then registering the edges.

3.2.2 Barnes-Hut Minimizer

CCVisu layout algorithm begins with random node positions. Subsequently, with the goal to decrease the total energy of the system, every iteration, the layout improves according the the configured energy model by computing a direction and distance for the movement of each node. The procedure is based on real words gravity physics, which is why terms like attraction and repulsion forces apply. Opposed to approaches with quadratic time complexity, the Barnes-Hut algorithm [4] implemented in Java by CCVisu achieves this energy minimization in $O(|E| + |N| \log |N|)$,[1] which is crucial for computing the layout of large graphs in acceptable time.

3.2.3 Energy Model

CCVisu provides a general energy model which influences and affects layout quality. It supports models such as: *Fruchterman Reingold* [15], *vertex-repulsion LinLog* [25] or *edge-repulsion LinLog* [26] energy model and more, individual models.

Users configure the following parameters:

[1] E refers to the set of edges and N to the set of nodes of the graph

- useWeightedEdges: boolean
- attractionExponent: positive integer
- repulsionType: edge | vertex
- gravitation: float
- dimensions: 2 | 3

First, `useWeightedEdges` determines whether the edges between nodes (vertices) weigh their individual assigned weights or if all edges weigh the default value of 1. In the context of source code graphs, the weight of an edge corresponds to the number of usages between classes. The `attractionExponent` controls node density, with bigger values resulting in denser layouts. The `repulsionType` specifies whether repulsion and attraction forces emanate from edges or nodes. Additionally, this influences whether to regard node or edge weights in case of weighted calculations. The `gravitation` factor affects the Barnes-Hut procedure which originates in real words physics phenomena. Last, the `dimensions` option sets the layout to 2D or 3D, although 3D is not directly visible in 2D visualizations.

3.3 Web Technologies

3.3.1 Next.js

Next.js [28] is an open-source web development framework built on React. It enhances search engine optimization (SEO) and supports server-side rendering (SSR), pre-fetching, and static site generation. Unlike traditional React applications, which render exclusively on the client, Next.js uses Node.js to enable server-side rendering [36]. Due to these benefits, Next.js has become the 4th most-used web framework globally [33].

3.3.2 sigma.js

Sigma.js [30] is a JavaScript library for visualizing networks graphs in the browser. It works in conjunction with the graph manipulation library *Graphology* and uses WebGL for high-performance rendering of large graphs.

3.3.3 Graphology

Graphology [18] is JavaScript library providing a multipurpose Graph object with support for directed and parallel edges. It includes a extensive and comprehensive standard library of graph algorithms, generators, layouts, and iterators. Graphology facilitates browser interactivity through event-driven APIs.

3.3.4 Web workers

The *Web Workers API* [20], standardized in modern browsers since 2015, enables background thread execution for web content. This allows computationally intensive tasks, such as graph layout calculations, to run without blocking the main thread.

4 Contribution

We present the web-first *software analysis tool CCVisu-Web*, which enables visualization of class dependencies for Java source code through force-directed graph layouts. The web application utilizes the React framework Next.js for the development. The graph layouts render with the help of the `sigma.js` library and generate client-side using an AI-assisted port of the layout algorithm originally implemented by CCVisu. On the backend, Doxygen and CCVisu extract class dependency relations from the provided source code.

4.1 Usage

To execute the software analysis application, build the provided Docker container:

```
$ docker build -t software-analysis-tool .
```

Then, run the container:

```
$ docker run -p 3000:3000 software-analysis-tool
```

Open http://localhost:3000 in a web browser to access the tool.
Alternatively, deploy the application to any cloud provider that supports Docker containers, or follow the setup instructions in the Getting Started section of our repository's `README.md` to build and run the app locally.

4.2 Architecture

Figure 4.1 illustrates the core architecture of our web tool. Users provide source code either as a public Git repository URL or a zip file.

Import. The Import section on the left-hand side of the architecture diagram shows the server-side process that generates a displayable graph object for the standard git URL import method. First, the repository clones into a temporary directory, where Doxygen parses the source code. CCVisu then converts the resulting XML which contains all relevant class dependencies into a file in the Relational Standard Format. Finally, the system generates a `Graphology Graph` object compatible with the `sigma.js` library responsible for displaying the user-manipulable graph viewer. In the graph, each node represents a class, and each directed weighted edge represents a dependency, with the weight indicating the total number of usages.

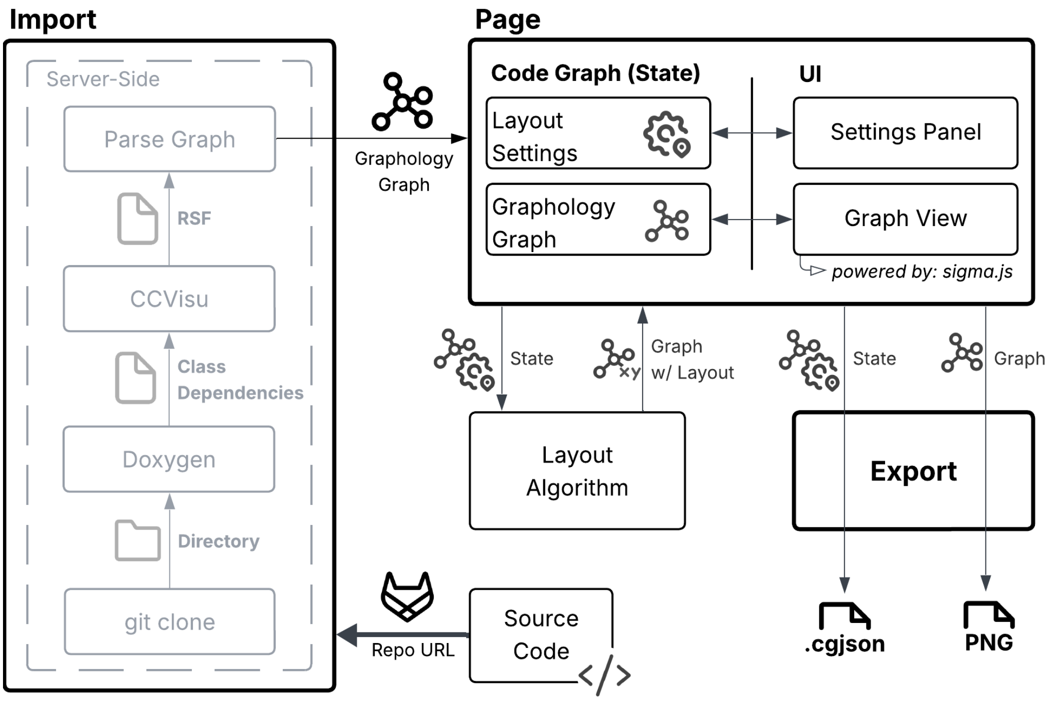

Figure 4.1: Software analysis application architecture diagram

Main Page. The right-hand side of the architecture diagram shows the Page section, which includes the rendering of the graph. The graph and layout settings form the page's state. In Figure 4.1 this combined state is referred to as "Code Graph". The internal state synchronized with the UI, which is indicated by the bi-directional arrows between the state and UI components. For example, regenerating the graph layout updates the graph viewer automatically, and user manipulations via the graph viewer or settings panel overwrite the internal state accordingly. The layout settings configure the generated graph layout, including the number of iterations, exponent values for the general energy model, and a gravity parameter.

Layout Algorithm. The layout algorithm generates a force-directed graph layout to reveal clusters of closely related classes. It incorporates the energy model configured via the layout settings and minimizes energy efficiently using the Barnes-Hut minimization algorithm. Users can execute the algorithm either client-side or server-side. The client-side TypeScript algorithm is an AI-assisted translation of CCVisu's original layout algorithm. Section 4.4 details our approach and insights into this translation.

Export. Users export the displayed graph layout as a PNG file or a `.cgjson` file, which saves the state of the page. The custom `.cgjson` file format allows users to resume work later by importing the file instead of a URL or zip.

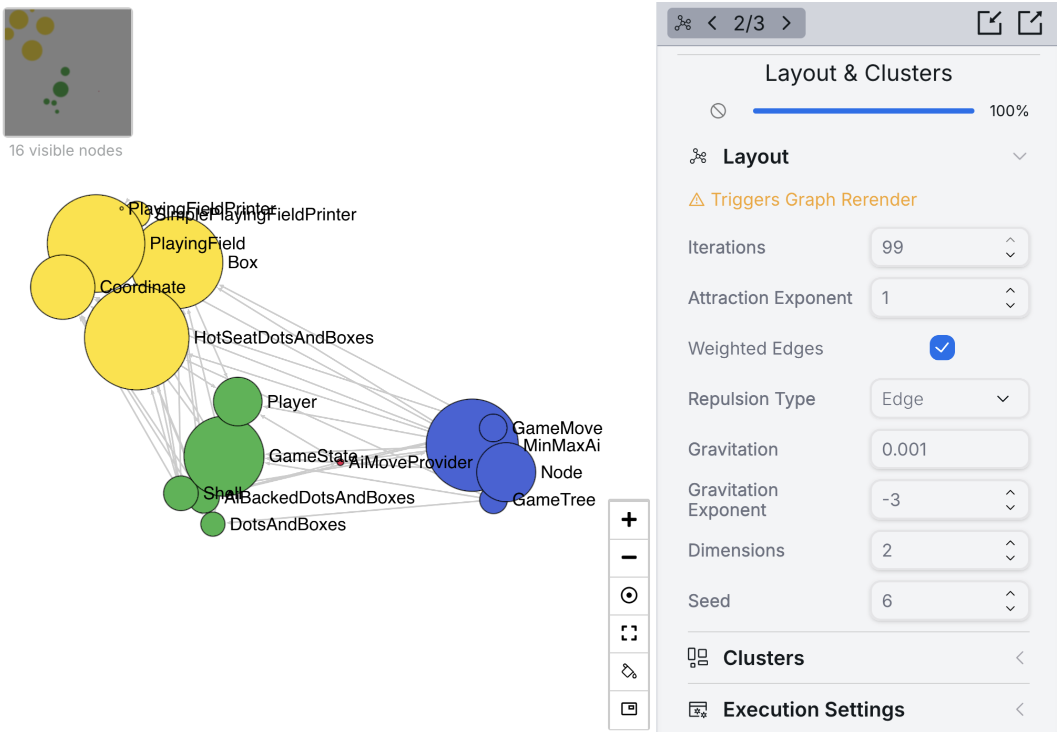

Figure 4.2: Screenshot of software analysis tool

4.3 Features

The Next.js app implements a single-page application, as shown in Figure 4.2. On the left-hand side is the *Graph View*, which renders the `Graphology Graph` object. Placed on the right is the *Settings Panel* with the *Top Bar* on top. The *Settings Panel* contains options to configure the generated graph layout and on the top the import and export button as well as buttons for creating and cycling through multiple graph layouts can be found.
Below, we detail the available features.

4.3.1 Graph View

The *Graph View* renders the `Graphology Graph` object using the `sigma.js` library. Users interact with the graph in the following ways:

- Hovering a node with the mouse cursor highlights it.
- Dragging and Dropping manually displaces nodes.
- Pan and pinch gestures navigate the canvas.
- Zoom controls (plus/minus buttons in the right-hand corner) adjust the view. Alternatively, users can zoom using the scroll wheel or trackpad.
- The reset button (third control) returns the viewport to its starting position.
- The fullscreen button (third from bottom) expands the view.
- The background color button (second from bottom) changes the canvas background.

- The minimap toggle (bottom control) shows or hides the minimap in the upper left corner, which helps navigate large graphs by showing the current viewport's position.
- Below the minimap, the tool displays the number of nodes currently visible in the viewport.

We adopt many of these features from CCVisu's GUI but enhance usability and accessibility. The zoom gestures and controls make zooming more intuitive compared to CCVisu's limited options. The zoom reset button, originally hidden in a submenu, is now directly accessible, addressing a frequent need in the original GUI. New additions include the minimap, visible nodes display, and node highlighting on hover. The latter improves readability by addressing label overlap.

4.3.2 Settings Panel

The *Settings Panel* accommodates options for the configuration of the generated graph layout and includes the following sections:

Layout. This sections adjusts parameters for the energy model detailed in Section 3.2.3. Users can fine-tune and tweak the computed layout to explore and achieve desirable outcomes using these settings. Changing any setting triggers the layout algorithm for the specified number of iterations.
Users choose between server-side or client-side execution. Server-side execution runs the CCVisu CLI on the backend using the original Barnes-Hut minimization algorithm. Client-side execution uses a web worker to run the AI-assisted TypeScript reimplementation. During computation, the settings inputs disable, and a loading state appears.

Clusters. This collapsible section automatically partitions the graph into a user-specified number of clusters, each assigned a unique color from a contrasting palette for readability
Here, users again choose between client-side or server-side computation. The client-side clustering algorithm is an AI-assisted TypeScript reimplementation of CCVisu's original Java implementation.

Execution Settings. The *Execution Settings* collapsible contains options to enable server-side execution of CCVisu on the backend as opposed to the locally run layout algorithm. Furthermore, here, users can toggle real-time updates for client-side execution, allowing the observation of the layout optimization iteration by iteration.

Progress Bar. The blue *Progress Bar* below the *Layout & Clusters* header quantifies the layout algorithm's progress as a percentage of completed iterations. These live progress updates are possible due to the AI-assisted TypeScript translation of the layout algorithm. Another advantage of this is that users may interrupt the algorithm at any time using the stop button on the left side of the progress bar.

4.3.3 Top Bar

The *Top Bar* includes controls for managing graph snapshots, importing source code, and exporting results.

Code Graph Snapshots. On the left of the *Top Bar*, the *Code Graph Snapshot* controls manage multiple graph layouts without requiring additional browser tabs. In the context of our application, a *Snapshot* represents the state of the web page, including the `Graphology Graph` object and layout settings.
The center-left number indicates the currently selected *Snapshot*, while the number on the right shows the total number of *Snapshots*. The left and right caret buttons cycle thorough *Snapshots*. If the first or last *Snapshot* is selected, the *Add Snapshot* button appears dynamically, allowing users to create a copy of the selected *Snapshot*.

Export. An icon showing an arrow pointing out of a square in the very top right corner represents the *Export* button. It opens a pop-up modal for exporting to `PNG` or the custom `.cgjson` format. The `.cgjson` format saves the complete state of the graph viewer page as a *JSON-stringified* `CodeGraph` object, preserving:

- the current graph layout with all nodes, edges, and their positions (as a Graphology `MultiGraph`)
- layout parameters (including weighted edges and gravitation as detailed in Section 3.2.3) and a random seed for reproducible layouts
- Execution Settings controlling refresh behavior and server-/client-side execution mode

Import. The *Import* button is right next to the *Export* button and similarly opens a pop-up modal for importing a source code graph. The new import replaces the graph layout of the current *Snapshot*. Users provide source code in the form of a public git repository URL or a ZIP file containing Java code. In both cases, after extraction of the class dependency graph from the source code, the modal closes automatically and the layout algorithm executes with the default parameters. Alternatively, users can import a previously exported `.cgjson` file, replacing the *Code Graph* of the current *Snapshot*.

4.4 AI Translation of the Java Layouting Algorithm to TypeScript

A central role in our research questions plays the translation of the `MinimizerBarnesHut` class, that CCVisu uses for computing the graph layout, to TypeScript to directly incorporate it in our Next.js web app. Porting the layout algorithm over from Java to TypeScript enables us to move the computation from the backend to the client. This makes the frontend more independent and allows us to show real-time progress updates to the user each iteration as the minimisation algorithm is running.

The primary goals of this approach are:

- to attain a reimplementation in TypeScript of the layouting algorithm
- to mininize the manual work for translating the 872 lines of code of the `MinimizerBarnesHut.java` class
- to avoid reviewing and understanding every detail of the performance-optimised and, in parts, difficult to interpret implementation
- to gather valuable lessons and insights on how generative AI can be used to effectively and efficiently port legacy code

4.4.1 Approach

We adopt a text-prompt-based approach, communicating via a web interface with the AI assistants *Le Chat* and *ChatGPT*. Specifically, we use their respective free tiers, which, as of July 2025, include Mistral AI's most recent iteration of their conversational assistant and OpenAI's GPT-4o. To avoid limiting ourselves to a single assistant, we choose *ChatGPT* due to its widespread use in software development settings [32] and Mistral AI's *Le Chat* to evaluate a European alternative – following the spirit of the Go European initiative [16], which advocates for EU-based digital sovereignty. We consider a prompt successful if the generated code compiles and produces the same layout as the Java version for our example graph.

File Preparation. To prepare the resource for AI translation, we consolidate the main class and all important dependencies into a single file. For this, we include the `Positions` class, which represents a 3-dimensional vector and bundles methods for calculations using vectors, as well as inline attributes inherited from the parent class `Minimizer` directly. `MinimizerBarnesHut` already contains as a inner class the `Octree` class. We exclude the `GraphData` class, as it contains irrelevant code for our use case. Instead, we focus on providing clear instructions on how to incorporate the new graph library for the TypeScript implementation.
Finally, we replace custom logging statements with basic console logging and retain comments and docstrings in the code, hypothesizing that this improves the AI's comprehension.

General Strategy. We begin with a simple, minimal prompt and iteratively add more specific details and requirements to our instructions. This on the one hand gives us insight into what challenges the assistant and what aspects of the generated code work immediately. It also enables us to steer the results in the necessary direction while only providing the minimal information required by the assistant to fulfill the translation task.
If the generated code compiles but does not work as expected or, at times, runs in an infinite loop, we restart from scratch instead of debugging. This involves recycling previous prompts with additional information for the AI assistant, rather than continuing an extended debugging dialogue.

4.4.2 Used Prompts

This section provides an overview of the prompts that lead to a working solution. The prompts listed here omit pasted files, as they span upwards of 800 lines of code, which exceeds the scope of this paper. For the complete files and prompts, refer to the `drafts/ai-translation/` directory of our repository.

We begin with a basic initial prompt to assess how the assistants perform with minimal instructions. For this prompt, we provide only the main, unmodified `MinimizerBarnesHut.java` class to evaluate how much the AI infers from context alone.

Prompt 1: Minimal input

Translate this legacy Java code to TypeScript. The new algorithm should operate on a Graphology Graph object.
// MinimizerBarnesHut.java
...

Le Chat did not ask any questions for clarification and directly generated code. However the translation still represents a simplified version, which requires further adjustments. ChatGPT, on the other hand, immediately asks many clarifying questions about constructing a minimal working translation before generating any actual code. This code also requires further adjustments. Unsurprisingly, more information is required for a successful port of the 800+ lines of code. For this initial prompt both Le Chat and ChatGPT generate compilable code, except for the occasional import, type, or syntax errors that need fixing.

Considering the feedback, we address two main issues in the next prompt. First, the raw `MinimizerBarnesHut` class uses a number of external dependencies such as other classes, like `Position`, `Minimizer` and `GraphData`, and inherited fields and methods, which are difficult to reason about without knowing their specifications. The prepared file (described in 4.4.1) alleviates these uncertainties by including external classes, removing function calls that are irrelevant for our use case and other minor adjustments. This reduces ambiguity by providing the AI with complete definitions. Secondly, as our focus is on a simple translation of the Java Syntax to TypeScript rather than a more involved refactoring, we update our prompt to address ChatGPT's eagerness to simplify and change aspects of the implementation. These suggestions include changes like renaming variables, restructuring code, or reimplementing methods.

Prompt 2: Added dependencies

> I'm using Next.js with React-Sigma.js and Graphology to build a web app for software analysis using force-directed graph layouts. I need help porting the following legacy Java class MinimizerBarnesHut to TypeScript. The Java algorithm I want to port is responsible for layouting the nodes of a Graphology.Graph object using a specified set of layoutingOptions. Layouting in this context means: setting the x, y and z positional attributes of the nodes and iteratively improving the layout towards a optimum of minimal energy.
>
> Ask me for more information, if what I've given you isn't enough for you to be able to perfectly replicate the algorithm. Don't take any shortcuts. It's essential that the new TypeScript implementation is just as performant as the original Java code.
> ... <pasted Java code> ...

Le Chat immediately delivers a convincing translation of the `Position` class included in the prompt. We decide to paste the completed TypeScript translation for subsequent prompts instead of requesting a new translation each time. After tweaking a minor detail of one function, we add the following to our prompt:

> The Position class has already been translated. See the pasted Position.ts file below.

Through trial and error, we identify another problem: many of the classes `MinimizerBarnesHut` depends on, such as `GraphData`, `GraphVertex` and `Options`, are tightly coupled with the rest of the CCVisu codebase. Providing their implementation to the AI to improve its understanding would require including many additional indirect dependencies, most of which are irrelevant for our use case. Instead, we opt to add examples to our prompt for translating these occurrences to TypeScript in a way that integrates well with our existing code. Mostly, we add examples to our prompt, on how to use the Graphology Graph object and its helper functions in place of the custom `GraphData` Java class.

Prompt 3: Included examples for Graphology

... <Previous instructions> ...
Note the following for replacing occurrences of the GraphData class:

- use Graphology.Graph (from now on referred to as "graph")
- use graph.size; instead of javaGraph.getVertices().size();
- use graph.forEachNode((node, attributes) => ...); instead of "for (int i = 0; i < vertexCount; ++i) " (and such)
- use graph.forEachEdge((edge, attributes, source, target) => ...); instead of "for (GraphEdge edge : graph.getEdges()) " (and such)
- in this case, the degree of a node is synonymous to it's size. So use attributes.size instead of graphVertex.getDegree();
- the id (sometimes also var name 'index') of a vertex of the Java Graph object corresponds to the key of a node of the Graphology.Graph object

This yields a promising translation by Le Chat, though the `Octree` class implementation hits the recursion limit imposed by the JavaScript execution environment due to the quickly growing depth of the tree structure. ChatGPT succeeds in producing the final missing piece: an iterative reimplementation of the `Octree` class in response to Prompt 4. In comparison, Le Chat struggles with this task, as all its proposed iterative implementations result in a stack overflow or an endless loop.

Prompt 4

Remove all recursion from this TypeScript class. Feel free to consolidate addNode with addNode2.
// Octree.ts
. . .

4.4.3 AI Translation Retrospective

This section discusses what works well and what works less effectively in using generative AI to port legacy code efficiently.

Code Preparation. To provide the AI assistant with a clear and straightforward goal, preparing and cleaning the code proves beneficial – especially for a large translation task as we required it. Simplifying the code in advance reduces ambiguity, particularly when the source code contains unnecessary external dependencies that complicate the AI's comprehension. For example, we replace code like:
`statusMessage("Starting minimizer.”);`
with: `System.out.println("Starting minimizer.”).`
Similarily, if a parent class (in our case `Minimizer`) can easily be removed, we inline its inherited fields and methods directly.

One-to-One Translation. For a straightforward one-to-one translation between programming languages, we emphasize preserving the logic and structure of the code while removing obstacles. This allows the AI assistant to focus on translating the syntax directly. As a result, comparing and identifying mistakes between the original and translated code becomes straightforward, since the same classes and methods are implemented in the same order and structure.

Interface Mappings. Defining mappings from one interface in the source language to another in the target language proves essential. In our case, a problem emerges: the AI struggles to adapt the code from the `GraphData` class to the `GraphologyGraph` class. See 4.4.2 (Prompt 3) for the provided mappings. Hypothetically, the AI could also retrieve up-to-date information itself, but we overlook this possibility during our interaction.

Le Chat. A notable strength of Mistral AI's assistant is its ability to provide near-instant responses. Additionally, it executes tasks more directly, with fewer detours and without apparent character or line limits.
However, Le Chat does not quite match OpenAI's assistant in performance. Particularly, it fails to refactor the `Octree` class while ChatGPT succeeds with relative ease. This difference may stem from differences in the models' training data or optimization for refactoring tasks. On the other hand, Le Chat's speed enables us to iterate quickly, while ChatGPT's confirmations slow down the process at times.

ChatGPT. Our approach focuses on the free tier of ChatGPT. During its use, we encounter instances where a paid subscription could offer more possibilities or a more efficient workflow. For one, ChatGPT can perform a deep analysis of attached files, but this feature quickly exceeds the free tier's quota. Also, ChatGPT tends to engage in iterative dialogue and often asks for excessive clarifications. This behavior sometimes feels unproductive, as if to intentionally prolong the time to reach a solution, potentially discouraging free-tier users.
Furthermore, the free tier imposes a 1,000-line limit on prompts, which restricts our workflow. In some cases, we must split prompts to accommodate this limitation.

5 Evaluation

This section covers the experiments we conduct to answer the following research questions:

RQ 1. *Can CCVisu's layouting and minimisation Java algorithm be replicated in TypeScript with the help of AI translation?*
RQ 2. *Is client-side execution of the new layouting and clustering TypeScript algorithms faster than running the CCVisu CLI on the backend of the web app?*
RQ 3. *Can the AI-translated implementation be integrated into the existing system with only minimal manual changes?*

To evaluate the research questions, we compare the node positions generated by the original Java algorithm and the AI-translated TypeScript implementation for the CPAchecker graph, measuring average and median Euclidean deviations after 1, 10, and 100 iterations. We assess the impact of 64-bit precision by comparing the original 32-bit Java implementation with a refactored 64-bit version.

For performance evaluation, we measure execution and response times of both the layouting and clustering algorithms under different configurations. We test the local TypeScript execution against the remote CCVisu CLI execution on Google Cloud Run, focusing on performance metrics at 1, 10, and 100 iterations. Our measurements account for real-world conditions by including network latency and JVM startup overhead.

The integration effort for the AI-generated code is quantified by analyzing the manual changes required, using `git diff` to track lines of code added or removed, and valuate the complexity of these adjustments.

5.1 Experimental Setup

To ensure the replicability of our results, we provide a TypeScript script that automatically collects the relevant data points for our research questions. The script automates data collection by running both the Java and TypeScript implementations on the same input graph and recording execution times and response times. All reported values in the tables represent the average of 5 independent measurements, rounded to 2 significant figures for clarity and consistency.
To run the TypeScript code locally, we use Node.js alongside the tsx package. Since the experiments involve comparing the original Java algorithm with the AI-aided

TypeScript reimplementation, this setup allows us to directly access and execute both.
We conduct the experiments on a MacBook Air running macOS 15.5, with an Apple M1 chip and 8 GB of RAM using the following tools:

- Node.js: v22.14.0
- Python: Version 3.12
- Java: OpenJDK 22.0.1
- Google Cloud SDK: 532.0.0

For RQ 2 in particular, to simulate a realistic production environment, we set up a Docker image and deploy our Next.js app to Google Cloud Run (GCR) in the Frankfurt region. We configure the CPU and memory limit of the active revision to 8 cores and 8 GB respectively. To avoid cold-start delays, we run tests on a warm GCR instance. This prevents significantly worse response times due to a cold start of a new container instance to serve the requests, by sending a preliminary request before running the tests. The TypeScript script and Docker configuration are available in our repository, along with instructions for deployment.

For our case study, we use CPAchecker and extract an example graph containing 2422 classes from the latest version of its source code at the time of running the evaluation script.
For our experiments we used the latest version of our web application (`ec00168f`), CCVisu version 3.5 (`85394b33`) and the latest version on main of CPAchecker (`5abc57c8`) at the time of beginning our evaluation work.

5.2 Quantitative Analysis

5.2.1 Layout Deviations in Java vs. TypeScript

Research question 1 investigates whether the AI-aided TypeScript implementation produces layouts comparable to the original Java algorithm. To assess this, we compare the layouts calculated by the original Java algorithm in CCVisu and our new TypeScript implementation. Both run with identical starting parameters: the source code graph representation of CPAchecker with random initial node positions and the following layouting settings for 1, 10 and 100 iterations.

Default layouting settings by CCVisu:
- useWeightedEdges: true
- attractionExponent: 1
- repulsionType: edge
- gravitation: 0.001
- dimensions: 2

Table 5.1 shows that both the average and median deviations increases with the number of iterations used for the layouting algorithm. The term deviation in this context refers to the euclidian distance between the two final positions calculated for a certain node by algorithms 1 and 2. The nodes of the graph passed to both algorithms each have a unique key for identification.

Table 5.1: Comparison of node position deviations for Algorithm 1 and Algorithm 2.

Algorithm 1	Algorithm 2	Iterations	Avg Deviation	Median Deviation
Java 32-bit	TypeScript	1	$3.6 \cdot 10^{-7}$	$2.6 \cdot 10^{-7}$
Java 64-bit	TypeScript	1	$1.1 \cdot 10^{-16}$	$6.9 \cdot 10^{-18}$
Java 32-bit	TypeScript	10	$3.7 \cdot 10^{-4}$	$2 \cdot 10^{-4}$
Java 64-bit	TypeScript	10	$4.8 \cdot 10^{-13}$	$2.5 \cdot 10^{-13}$
Java 32-bit	TypeScript	100	5.1	2.7
Java 64-bit	TypeScript	100	3.9	2.3

We also run our comparison algorithm a second time using an adapted version of CCVisu (see `scripts/ccvisu3.5-64bit.jar`) referred to as `Java 64-bit` in Table 5.1. With a straightforward refactoring, we replace the 32-bit primitive data types, such as `float` and `int`, used in the original Java implementation with their 64-bit counterparts. As JavaScript's `number` type is 64-bit, this serves as an attempt to reduce deviations caused by differences in number precision.

For example, when comparing the the 64-bit Java layout algorithm and the new TypeScript implementation, the final calculated graph layouts are almost identical for 1 and 10 iterations but diverge noticeably for 100 iterations. This divergence results primarily from nodes highlighted in Table 5.2. These nodes all have in common that they connect to only two other nodes. For the broader context of the layout of the graph, this means that algorithm 1 finds a completely different optimal neighbourhood for nodes like `JArrayInitializer` than Algorithm 2.

Table 5.2: Deviation and degree of main outlier nodes.

Deviation	Degree	Name
340	2	cfa/ast/java/JArrayInitializer.java
180	2	core/defaults/GenericReducer.java
360	2	core/interfaces/WrapperPrecision.java

Figure 5.1 highlights the differences in node x and y positions between the two algorithms. Each arrow indicates where Algorithm 1 places a node relative to Algorithm 2. The base of an arrow represents the position determined by the first algorithm, while the tip indicates the position determined by the second algorithm. A long line indicates a high positional deviation for a node, whereas small points show that the positions determined by the first and second algorithm are very close together.

Since each iteration builds directly on the previous one, it is not surprising that the final node positions diverge increasingly with a growing number of iterations. However, the three extreme outliers from 5.1, represented by the three long red

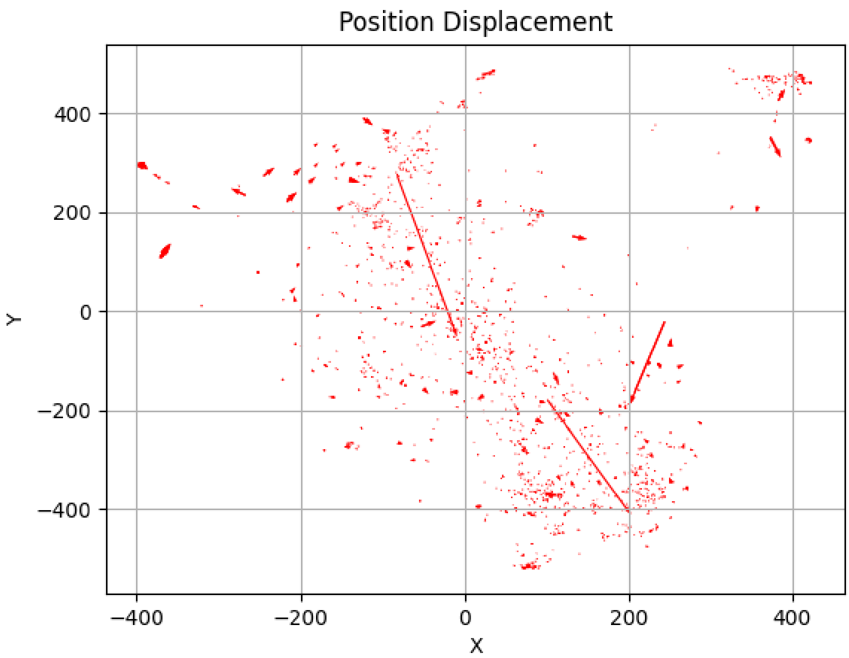

Figure 5.1: Quiver plot highlighting node position deviations.

arrows (one mainly in the NW quadrant and two in the SE quadrant) in Figure 5.1, stand out as unexpected. We specifically will examine the outlier represented by the line from point `(-82,272)` to `(-9,-52)`.

5.2.2 Client-Side vs. Server-Side Layout Computation Performance

To address research question 2, we measure the execution and response times of the layouting and clustering algorithms, both client-side and remotely on the server, using the adapted 64-bit version of CCVisu. In a real-world scenario, this metric represents the time required for final graph layout to appear after a user modifies a layouting or clustering setting. We send requests to the remote GCR instance to test the API under realistic conditions.

Table 5.3 presents the measured elapsed times for different configurations. The first column differentiates the layouting and clustering algorithms. `Layouting` refers to the Barnes-Hut minimization algorithm to compute a layout for the graph. `Clustering` refers to the algorithm that partitions the graph into a specified number of clusters, where nodes in a particular partition share a color.

Our measurements show that the client-side implementation computes the layout of the CPAchecker source code graph in less than 1 second for 1 iteration, whereas the Java backend requires 2 seconds. Comparing layout times for 100 iterations draws a more complex picture: the client-side TypeScript algorithm allows us to directly display progress updates with the most recent graph layout to the user after reach iteration completes. Here, the algorithm runs in a worker thread and, after each

Table 5.3: Execution and response times of execution of layout and clustering algorithms on client and backend.

Algorithm	Endpoint	Iterations / Cluster	Time (ms)	
Layouting	Local (Typescript)	1	260	
Layouting	API (CCVisu - 64-bit)	1	2000	
Layouting	Local (Typescript)	100	33000	
Layouting	API (CCVisu - 64-bit)	100	11000	*
Layouting	API (CCVisu - 64-bit)	100	200000	**
Clustering	Local (Typescript)	2	3400	
Clustering	API (CCVisu - 64-bit)	2	8200	
Clustering	Local (Typescript)	100	3200	
Clustering	API (CCVisu - 64-bit)	100	8100	

iteration, sends the current graph layout to the main thread, which updates the UI and displays the new state of the graph. The server-side execution of CCVisu currently lacks real-time update functionality. Thus the time in row four (*) accounts for the time required to send the requests, run the layout algorihtm on the backend, and receive the final graph layout without any in-between progress updates. To match the client-side functionality, we must send a separate request to the server for each iteration, totaling 100 requests. We approximate this scenario in row five (**) of Table 5.3. For clustering, the algorithm operates by iteratively merging clusters. It starts with each node in its own cluster and continues merging until it reaches the desired number of clusters. This explains why partitioning the nodes into 100 clusters takes less time than partitioning them into two clusters: the algorithm requires fewer merges to reach a higher target number of clusters. Our results demonstrate that local execution of the clustering algorithm on the CPAchecker source code graph reduces user wait time by approximately 10 seconds.

5.2.3 Required Manual Changes for AI-assisted Translation

To address research question 3, this section quantifies the manual effort required to finalize the AI-translated TypeScript layout algorithm.

We use `git diff` to analyze the manual effort needed to achieve a functional state for the AI-translated TypeScript layout algorithm. Specifically, the initial unaltered file, `app/lib/Layout.ts` from commit `2a228021`, contains 702 lines of code (LOC), including comments and whitespace. The subsequent commit, `91a7afac`, includes all necessary adjustments to integrate the translated algorithm into the existing code.

The 67 manual line changes correct imports, adjust and rename passed parameters, and fix small errors caused by our specification. For example, we replace `this.graph.size` with `this.graph.order` and ensure random initial positions are set for the graph nodes. In Prompt 3 (4.4.2), we explicitly instruct the AI to use `attributes.size` instead of `graphVertex.getDegree()`.

Other than syntactic changes, we also address a critical runtime issue. Since stack overflow errors occur, we replace the recursive private `Octree` class with an equivalent iterative, fully AI-generated implementation. This replacement requires 65 line changes and 25 additions. The final working implementation totals 727 LOC, encompassing the AI-assisted translations of CCVisu's Java classes: `MinimizerBarnesHut`, `Octree` and `Position`. We avoid premature automatic code formatting, which could introduce many non-functional modifications. For consistency, we maintain object-oriented constructs throughout for this comparison, rather than adapting the more functional style used in our Next.js web application.

Overall 67 out of the 727 LOC of AI-generated code require manual adjustments to ensure compilation and full functionality.

Less than 10% of the generated code requires manual retouching. These changes are straightforward, slight adjustments that we implement easily using a global replace tool. They do not involve changing the logic of the inner workings of the algorithm. Manual adjustments prove faster than going through an extended dialogue to have the AI assistant fix them. We do not prioritize keeping the metric of manually changed lines of code (LOC) as low as possible. Instead, we focus on achieving a correct and functional translation of the Java classes efficiently and swiftly.

5.3 Qualitative Analysis

5.3.1 Interpretation of Node Position Deviations

The following section addresses RQ 1 by interpreting the relatively high positional deviations observed after 100 iterations, as outlined in Section 5.2.1.

The outliers in Figure 5.1 share a common characteristic: they connect directly to only two other nodes. Nodes with only two connections are more sensitive to small differences in energy calculations because their positions depend on fewer factors. In the context of gravity-based repulsion and attraction in force-directed layout computation, this means that the tendency to pull a node toward one connected neightbor or the other overweighs in different directions, depending on the algorithm in use. This results from slight differences in the computed energies of the used energy model, which arise from differing implementations of floating-point arithmetic across languages [23].

To illustrate this, Figure 5.2 shows the isolated positions of the `JArrayInitializer` class and its connected nodes `AbstractExpressionValueVisitor` and `CFAUtils`. The red dots represent the graph layout generated by the original Java implementation, while the yellow dots represent the layout generated by the new TypeScript algorithm. This isolated example clarifies the tendency for `JArrayInitializer` to cluster with either one of its direct neighbors.

Analyzing several layouts generated by the original Java algorithm based on different random initial node positions reveals that the connected node with which `JArrayInitializer` clusters varies. This indicates that the original implementation can also produce a layout with the same relative positions for the three considered nodes as our TypeScript implementation.

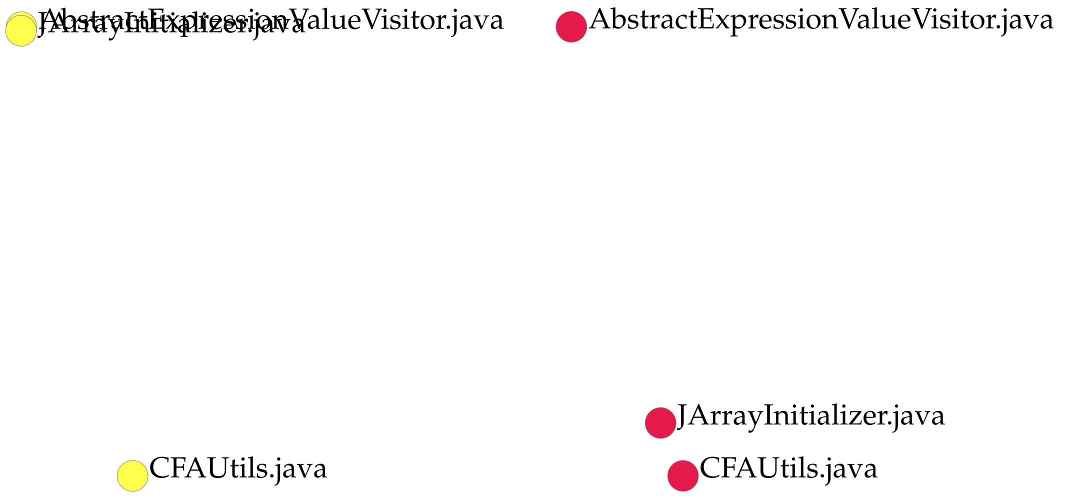

Figure 5.2: Isolated outlier 'JArray-Initializer' and its connected nodes.
Red: Java layout, yellow: TypeScript layout

This underlines the fact that, at least for large graphs, no single local optimum exists. Instead, many different optima are possible. This explains why CCVisu and our web application expose layouting options to users: to allow them to refine and improve the computed layouts. The outliers demonstrate how, in this complex system, small individual differences can accumulate into vastly different outcomes as the number of nodes and iterations increases.

Execution Time Comparison. Table 5.3 shows that the new algorithm performs significantly faster for a small number of iterations, or more precisely, for layout computations that complete in fewer than several seconds. Beyond this threshold, the JVM startup time is quickly offset by the Java implementation, which then outperforms the JavaScript algorithm in speed.
Nevertheless, despite being considerable slower in terms of raw computation, the reimplementation represents the better choice in practice. It enables real-time progress updates, which provide immediate visual feedback and gives users a grasp how the layout evolves over many iterations, which is useful for fine-tuning.

Impact of 64-bit Precision. Our results demonstrate that comparing the 64-bit Java algorithm to the TypeScript implementation significantly reduces the deviation in node positions. The issue with 32-bit floating-point numbers is their limited precision, which leads to accumulated errors over many iterations. For example, a small rounding error in the energy or position of a node in one iteration can propagate and amplify in subsequent iterations.
For a single iteration, the deviation is almost not observable anymore, as we now operate close to the highest precision that JavaScript can capture for fractional numbers. Thus, we conclude that this eliminates the uncertainty regarding the extent to which differences in floating point precision contributed to the observed

rounding errors. We attribute the remaining observed positional deviations to a cumulative effect caused by differing implementations of floating-point arithmetic.

Considering this, we conclude that research question 1 is positively answered.

5.3.2 Client vs Server Execution Time Analysis

The difference in execution times result partly from added latency when communicating with the remote GCR instance and the lower single-core performance of the CPU allotted to the GCR instance compared to the M1 MacBook used for local execution of the TypeScript algorithm.

Our measurements show that CCVisu on the server completes 100 uninterrupted iterations about 10 seconds faster (including latency) than our local TypeScript implementation, which updates the graph view after each iteration with the newly calculated layout. However, when we account for the time subsequent requests to the server would take to enable real time updates, the client-side solution reduces the theoretical execution duration by more than a factor of ten, as shown in Table 5.3. Hence, when we require real-time updates, the client-side implementation outperforms the server-side approach because it avoids network latency and repeated JVM startup times

We conclude that offloading the execution of the layout algorithm from the backend to the client is indeed faster for our expected use cases. A lower response time directly impacts user perception of responsiveness when interacting with the settings panel.

5.3.3 Manual Changes in AI-Generated Code

This subsection examines selected examples of manual changes required to correct the AI-generated TypeScript code. We highlight what the AI misinterpreted and propose hypotheses for why these errors occurred.

```
1 - import { Graph } from ’graphology’;
2 - import { LayoutingOptions, GraphJson } from ’./types’;
3 + import Graph from ’graphology’;
4 + import { LayoutingOptions } from ’types/LayoutingOptions’;
5 + import { setRandomPositions } from ’./Graph’;
6 //...
7 +   setRandomPositions(this.graph, options.randomSeed);
8 -   const vertexCount = this.graph.size;
9 +   const vertexCount = this.graph.order;
10 //...
11      this.graph.forEachEdge((edge, attributes, source, target) => {
12        if (source !== target) {
13 -         const sourceIndex = parseInt(source);
14 -         const targetIndex = parseInt(target);
15 +         const sourceIndex = graph.getNodeAttribute(source, "id");
16 +         const targetIndex = graph.getNodeAttribute(target, "id");
17           attrCounter[sourceIndex]++;
18           attrCounter[targetIndex]++;
19        }
20 //...
21 -    if (this.options.energyDiffThreshold === 0.0) {
22 +    if (this.energyDiffThreshold === 0.0) {
```

Listing 2: Selection of edited LOC of raw AI-translation

Listing 2 shows selected modifications we make to achieve a functional state for the AI-translated TypeScript layout algorithm. First, we correct the imports in lines 1-5. Our prompts to the AI lack context about the target project’s structure, so it cannot determine the correct import paths for our specific module setup. In lines, 7-9 we replace `this.graph.size` with `this.graph.order` and ensure random initial positions are set for the graph nodes. We fail to inform the AI-assistant about the initial layout requirements in Prompt 3 4.4.2, where we wrongly instruct the AI to use `attributes.size` instead of `graphVertex.getDegree()`. In lines 13-16 the AI misunderstands how the Graphology library handles node attributes. It incorrectly assumes that `source` and `target` keys contain stringified indices. The AI lacks context about Graphology’s attribute-based node identification system, which differs from the original Java implementation’s direct index access (`edge.getSource().getId()`). Finally, the AI confuses parameter names and object properties, likely due to inconsistent naming in our prompts. Instead of changing the usages, we could also adjust the declaration in the constructor.

5.4 Threats to Validity

5.4.1 Internal Validity

Layout Algorithm Translation using AI. It is not inconceivable that the AI-assisted translation of the Java implementation of the Barnes-Hut minimisation algorithm to TypeScript might contain subtle bugs or inaccuracies, particularly for edges cases not covered by our case study. To mitigate this possibility, we focus on easily comparable syntax-based translation by the AI and compare the output of the TypeScript implementation with the original Java version for identical input parameters. The close alignment of node positions suggests that the translation is functionally equivalent.

The identified outliers align with expectations and can be explained by differences in floating-point arithmetic. They do not indicate fundamental flaws in the translation. Also, the consistency of results across multiple iterations further supports the translation's validity.

Measurement Accuracy. External factors could influence the measured execution and response times in our experiments, such as fluctuating network latency, spikes in background CPU consumption on our test machine, or variability in the GCR environment. To ensure accuracy, we use a controlled and consistent hardware configuration and a warm GCR instance to minimize differences caused by cold starts. Additionally, our use of a TypeScript script to automate data collection reduces the probability of human error in timing measurements. While minor fluctuations in execution times are possible, we observe significantly large differences to suggest that the core findings are robust.

64-bit Reimplementation. Our 64-bit reimplementation of the Java algorithm might introduce subtle changes in behaviour compared to the original 32-bit version. However, since our refactoring involves almost exclusively replacing number types, we consider this to be highly unlikely. More importantly, we verify functional equivalence using our script to evaluate layouts of the TypeScript and Java implementation by comparing the results of the 32-bit and 64-bit version.

5.4.2 External Validity

Applicability of the AI-Assisted Translation. Our approach to translating Java code to TypeScript using AI assistants may not generalize to other codebases or programming languages. Different codebases or programming languages may be less suitable for direct AI-assisted translation. Specifically, translating the singular `MinimizerBarnesHut` class involves object-oriented constructs, well documented code, recursion, tree and vector algebra, as well as relatively few external dependencies.
We focus on a widely-reproducible procedure using the free and web-interface-based assistants Le Chat and ChatGPT. Our detailed account of our preparation, the encountered challenges and strategies used to address them, allows others to assess the applicability our methods to their own projects. While the effort required for translation may vary, our core insights, such as the importance of code preparation and the value of iterative prompting, are likely to apply broadly.

Case Study Source Code Graph. Our case study focuses on the source code graph of the CPAchecker project, which may not represent other input graphs with different sizes and connective patterns. We aim to include a relevant real-world example by using a complex software project. The observed trends in execution time and deviation are based on fundamental properties of the layout algorithm, which are likely to hold across different graphs.

6 Future Work

Below, we outline potential future work, to highlight possibilities for further improvement and optimization.

6.1 Unimplemented Features

Due to time constraints, some planned features remain unimplemented. Implementing these features would significantly improve usability and functionality:

Advanced Settings for Customization. The settings panel can be expanded to include input types like checkboxes, dropdowns, color pickers, sliders, and range selections for fine-grained control over node, edge and label appearance for attributes such as visibility, size and color. Support for non-deterministic states would clarify when selected nodes have conflicting settings, which for example a half-filled checkbox can communicate. This mirrors and builds upon CCVisu's *Node Manager* tab and enables users to create more expressive and meaningful graph layouts.

Node Information Table. A node information table could be added to the settings panel which lists the graph nodes with columns for color (cluster), name, in-degree and out-degree. The HeroUI `Table` component provides a viable option for this integration. This table represents the basis for node searching and filtering and can give access to detailed metadata about nodes. Also, users should be able to select nodes in the table and be able to "pin" them to keep them selected even when filtering or searching.
Furthermore, node, edge and label settings should only apply to and be synchronized with the current selection of nodes.
A intermediate state for all inputs, such as checkboxes and color pickers, could be introduced, which communicates if the node selection does not share the same distinct value for a specific setting. For example, if the selected nodes include visible as well as hidden nodes, the visibility checkbox would display the intermediate to reflect the non-boolean state. Clicking the checkbox would show all nodes and give the checkbox a definite checked (visible) state.

Node Searching and Filtering. A search bar for pattern-matching node labels would enable quick navigation and easy compilation of a selection of nodes from the node table. Filtering options based on properties like node color (cluster), in-degree, or out-degree would allow users to focus on specific subsets of the graph. Additionally, searching and filtering should automatically update the graph view, with the selection being highlighted. Like explained above for a selection of nodes, the settings inputs would reflect the shared state of the filtered selection.

Export and Sharing. A SVG export feature would allow users to save high-quality vector graphics of the graph, with an option to export only the subgraph visible in the viewport.
The PNG export currently offers limited options, providing only width and height adjustments. Adding a DPI setting, layer-specific rendering (nodes, edges, labels, etc.), and frame selection would significantly improve this feature's usability and output quality.
Furthermore, shareable URLs could store source code graphs on a server, enabling users to share and come back to their work using a URL which includes the reference to the stored work in query parameters. A `dirty=true` parameter would warn users if the shared state differs from the original (e.g., due to manual node adjustments or filtering). Only the currently selected snapshot would be shared, with warnings for unsaved changes (e.g., hidden nodes or modified layouts).

6.2 Technical Optimizations

Multi-Language Support. Doxygen's capability to parse a variety of programming languages and generate XML for can be used to extend the class dependency extraction pipeline. This enables the analysis of software projects based on programming languages other than the only currently supported language Java. This requires the investigation of language-specific dependency relations to ensure accurate graph representations.

Performance Improvements. The current implementation gives several opportunities for performance optimization and reduced load times. By leveraging GPU acceleration through WebGL or WASM, we can substantially improve layout computation speeds, particularly for complex graphs. Additionally, leveraging Next.js caching mechanisms for server requests would minimize redundant computations and improve response times for repeated operations. Implementing incremental data loading techniques would also boost performance for very large graphs by loading only the immediately necessary portions of the graph data.

6.3 Explore AI Opportunities

Future work could explore additional porting efforts, this time examining paid services or IDE-integrated AI. Also interesting are automated test for self-verification of the AI's work. Last, we can explore context-aware AI to improve reasoning about dependencies by incorporating library and framework documentation using tools such as Context7 or MCP servers.

7 Conclusion

This thesis demonstrates the practicality of reviving legacy software through AI-assisted translation and web-first design. By porting CCVisu's Java layout algorithm to TypeScript and integrating it into a Next.js-based web application, we created CCVisu-Web, a modern, browser-based software analysis tool that preserves CCVisu's analytical strengths while offering a significantly improved user experience. The new frontend automates the previously manual workflow and introduces an intuitive, synchronized interface with smooth zooming, node highlighting, a minimap, and reproducible layouts via a random-seed option. These features make software exploration more accessible and efficient than in the original desktop tool. Our evaluation using the CPAchecker project confirms that the TypeScript reimplementation generates layouts comparable to the original algorithm, with observed deviations being attributable to differences in floating-point precision. Moreover, client-side execution improves responsiveness by eliminating backend round-trips. In summary, this work delivers both a modernized, web-native frontend and a validated AI-assisted translation process, demonstrating how legacy analysis tools can be effectively revived for today's web ecosystem.

Bibliography

[1] Airbnb. visx a collection of expressive, low-level visualization primitives for react, 2025. Last Accessed: 2025-10-03.

[2] M. Alenezi and K. Almustafa. Empirical analysis of the complexity evolution in open-source software systems. *International Journal of Hybrid Information Technology*, 8(2):257–266, Feb. 2015.

[3] M. Balzer, O. Deussen, and C. Lewerentz. Voronoi treemaps for the visualization of software metrics. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, SoftVis '05, page 165–172, New York, NY, USA, 2005. Association for Computing Machinery.

[4] J. Barnes and P. Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324(6096):446–449, Dec. 1986.

[5] M. Bastian, S. Heymann, and M. Jacomy. Gephi: An open source software for exploring and manipulating networks. In E. Adar, M. Hurst, T. Finin, N. S. Glance, N. Nicolov, and B. L. Tseng, editors, *Proceedings of the Third International Conference on Weblogs and Social Media, ICWSM 2009, San Jose, California, USA, May 17-20, 2009*. The AAAI Press, 2009.

[6] D. Beyer. Co-change visualization applied to postgresql and argouml: (msr challenge report). In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, page 165–166, New York, NY, USA, 2006. Association for Computing Machinery.

[7] D. Beyer. Ccvisu: automatic visual software decomposition. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, page 967–968, New York, NY, USA, 2008. Association for Computing Machinery.

[8] M. Bostock, V. Ogievetsky, and J. Heer. D^3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, Dec. 2011.

[9] CodeViz.ai. Codeviz: Instant codebase clarity, 2025. Last Accessed: 2025-10-03.

[10] P. Collet, J. Mortara, Y. Brault, and A. Dery-Pinna. The varicity ecosystem: City visualization of object-oriented variability in java and typescript. *Sci. Comput. Program.*, 240:103210, 2025.

[11] Dirk Beyer. Ccvisu 3.0 introduction and reference manual, 2010. Last Accessed: 2025-09-30.

[12] Doxygen. Code documentation. automated. free, open source, cross-platform., 2025. Last Accessed: 2025-09-30.

[13] U. Erra, G. Scanniello, and N. Capece. Visualizing the evolution of software systems using the forest metaphor. In *2012 16th International Conference on Information Visualisation*, pages 87–92, 2012.

[14] FalkorDB. Code graph: From visualization to integration, 2025. Last Accessed: 2025-10-03.

[15] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, Nov. 1991.

[16] Go European. Discover European products and services, 2025. Last Accessed: 2025-09-24.

[17] Good Code. Reagraph, 2025. Last Accessed: 2025-10-03.

[18] Graphology. Graphology documentation, 2025. Last Accessed: 2025-09-30.

[19] JArchitect. Jarchitect documentation, 2025. Last Accessed: 2025-09-30.

[20] MDN contributors. Web workers api, 2024. Last Accessed: 2025-09-30.

[21] D. Moreno-Lumbreras, R. Minelli, A. Villaverde, J. M. González-Barahona, and M. Lanza. Codecity: A comparison of on-screen and virtual reality. *Inf. Softw. Technol.*, 153:107064, 2023.

[22] P. Moret, W. Binder, D. Ansaloni, and A. Villazon. Visualizing calling context profiles with ring charts. In *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 33–36, 2009.

[23] J. Muller, N. Brunie, F. de Dinechin, C. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres. *Handbook of Floating-Point Arithmetic (2nd Ed.)*. Springer, 2018.

[24] nivo. Nivo network, 2025. Last Accessed: 2025-10-03.

[25] A. Noack. *An Energy Model for Visual Graph Clustering*, page 425–436. Springer Berlin Heidelberg, 2004.

[26] A. Noack. *Energy-Based Clustering of Graphs with Nonuniform Degrees*, page 309–320. Springer Berlin Heidelberg, 2006.

[27] OpenJS Foundation. Cosmos.gl, 2025. Last Accessed: 2025-10-03.

[28] S. Pati and Y. Zaki. Evaluating the efficacy of next.js: A comparative analysis with react.js on performance, seo, and global network equity. In *Companion Proceedings of the ACM on Web Conference 2025*, WWW '25, page 1239–1243, New York, NY, USA, 2025. Association for Computing Machinery.

[29] S. Scarle and N. Walkinshaw. Visualising software as a particle system. In *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, pages 66–75, 2015.

[30] Sciences-Po médialab and OuestWare. sigma.js homepage, 2025. Last Accessed: 2025-09-30.

[31] SciTools. Understand, 2025. Last Accessed: 2025-09-30.

[32] A. Sergeyuk, Y. Golubev, T. Bryksin, and I. Ahmed. Using ai-based coding assistants in practice: State of affairs, perceptions, and ways forward. 2024.

[33] Statista. Most used web frameworks among developers worldwide, as of 2024, 2024. Last Accessed: 2025-09-30.

[34] M. D. Storey, K. Wong, and H. A. Müller. Rigi: A visualization environment for reverse engineering. In W. R. Adrion, A. Fuggetta, R. N. Taylor, and A. I. Wasserman, editors, *Pulling Together, Proceedings of the 19th International Conference on Software Engineering, Boston, Massachusetts, USA, May 17-23, 1997*, pages 606–607. ACM, 1997.

[35] N. Synytskyy, R. C. Holt, and I. J. Davis. Browsing software architectures with lsedit. In *13th International Workshop on Program Comprehension (IWPC 2005), 15-16 May 2005, St. Louis, MO, USA*, pages 176–178. IEEE Computer Society, 2005.

[36] M. Thakkar. *Next.js*, page 93–137. Apress, 2020.

[37] Tiago P. Peixoto. graph-tool, 2025. Last Accessed: 2025-10-03.

[38] B. Vasilescu, K. Blincoe, Q. Xuan, C. Casalnuovo, D. Damian, P. Devanbu, and V. Filkov. The sky is not the limit: multitasking across github projects. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 994–1005. ACM, May 2016.

[39] R. Wettel and M. Lanza. Visualizing software systems as cities. In *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 92–99, 2007.