MinimapSignatures: A Multi-IDE Ecosystem for Analysis of Visual Source Code Signatures

Munif Gebara Jr.
Graduate Program in Computer Science, State University of Maringá, Maringá, Brazil
munif@munif.com.br

Jamily G. S. Souza
Department of Informatics, State University of Maringá, Maringá, Brazil
salesjamily0@gmail.com

Lucas B. N. Carvalho
University Center of Maringá - UNICESUMAR, Maringá, Brazil
lucasneo@outlook.com.br

Vitor T. Correa
Department of Informatics, State University of Maringá, Maringá, Brazil
vitorteodorodev@gmail.com

André F. R. Cordeiro
Department of Informatics, State University of Maringá, Maringá, Brazil
afrcordeiro@uem.br

Igor S. Wiese
Federal University of Technology - Paraná, Campo Mourão, Brazil
igor.wiese@gmail.com

Yandre M. G. Costa
Department of Informatics, State University of Maringá, Maringá, Brazil
yandre@din.uem.br

Abstract

Modern development environments provide powerful textual navigation, syntax highlighting, and rule-based static analysis, but developers still need complementary mechanisms to reason about structural and stylistic patterns across large codebases. Previous studies have shown that source code minimaps can act as compact visual signatures, enabling machine learning and computer vision models to analyze source code through image-like representations. This paper presents *MinimapSignatures*, a multi-IDE tool infrastructure that integrates plugins for IntelliJ IDEA and Visual Studio Code with a decoupled Python backend. The IDE clients generate anonymized 128×128 grayscale minimaps from source files being edited or inspected, while the backend applies trained models for *Type*, *Project*, and *Author* prediction. The architecture separates IDE-specific interaction from model inference, allowing plugins to remain lightweight while models can be updated, replaced, or extended independently. The current prototype supports file-level analysis through Markdown-based split-screen feedback in IntelliJ IDEA and a richer WebView-based interface in Visual Studio Code. A preliminary functional evaluation validates the complete workflow, including local minimap generation, anonymization, client–server communication, backend inference, and result presentation in both IDEs. The results indicate that source code minimaps can be integrated into real multi-IDE workflows, establishing reusable infrastructure for project-level model customization, software quality assessment, and AI-assisted program comprehension.

Demo video: https://zenodo.org/records/20275474 and https://youtu.be/BT7ucCpCrrk

Keywords

Software visualization, Source code minimaps, IDE plugins, Machine learning, Computer vision.

1 Introduction

Modern software systems are increasingly large, heterogeneous, and continuously evolving, making program comprehension, maintenance, and quality assessment progressively more difficult. Although integrated development environments provide powerful textual navigation, syntax highlighting, local warnings, and rule-based static analysis mechanisms, developers still spend substantial effort understanding structural patterns, architectural conventions, and relationships across large codebases. Traditional software visualization approaches commonly represent software systems through dependency graphs, UML diagrams, metrics dashboards, architectural views, and interactive visualizations of source code relations [7, 8, 11]. These approaches are valuable because they support human reasoning about structures and dependencies that are difficult to infer from raw textual code alone.

Source code minimaps provide a complementary representation for this problem. In this work, they are not treated as conventional software visualization mechanisms intended primarily for direct human exploration. Instead, a minimap is considered a compact, image-like representation of a source file that preserves spatial, textual, and structural regularities of the underlying code. Rather than encoding programs only as tokens, abstract syntax trees, graphs, or metric vectors, minimaps transform source code into machine-readable visual signatures that can be processed by machine learning techniques [3, 4]. This makes them suitable for lightweight and automated analysis scenarios in which source code artifacts need to be represented in a uniform format, while still reducing dependence on language-specific parsers in the first analysis layer.

Despite this potential, minimap-based analysis has mostly remained detached from the environments where developers write, navigate, and inspect code. Existing tools usually emphasize dependency graphs, architectural diagrams, metrics dashboards, fault localization views, repository-level summaries, or learning-based representations built from tokens, embeddings, graphs, and long code sequences [1, 2, 5, 10, 12]. Recently, source code has also been explored as visual data, showing that image-like representations can support code understanding tasks without relying exclusively on parsers or language-specific analysis [9]. However, tool support remains limited for generating, anonymizing, collecting, and analyzing source code minimaps directly within IDEs.

To address this gap, this paper presents *MinimapSignatures*, a multi-IDE tool infrastructure for applying source code minimap analysis within practical development workflows. The tool provides plugins for Visual Studio Code and IntelliJ IDEA connected to a

decoupled backend for visual source code analysis. This design allows the plugins to focus on source capture, local minimap generation, anonymization, and developer interaction, while the backend concentrates on preprocessing, model execution, and prediction formatting. This separation is particularly important for AI-based tools, since analysis models may evolve, be replaced, or be retrained more frequently than IDE integrations.

The current prototype supports file-level analysis through anonymized 128×128 grayscale minimaps and follows a shared client-server contract across both IDEs. Although the backend is initially designed around three prediction targets—*Type*, *Project*, and *Author*—these classification tasks serve primarily as initial benchmarks to validate the feasibility, stability, and performance of our multi-IDE infrastructure. The core value of this decoupled ecosystem lies in its extensibility: the architecture was engineered so that advanced downstream models—such as those targeting software quality assessment, code smell detection [6], or security and vulnerability auditing—can be seamlessly plugged into the backend. This deployment model ensures that as visual analysis models evolve, developers benefit from state-of-the-art software diagnostics in real time, without ever needing to update or reinstall their IDE plugins.

The main contributions of this paper are threefold. First, it presents a multi-IDE tool for generating anonymized source code minimaps directly from popular development environments, allowing the implementation to reuse mature editor capabilities rather than rebuild them. Second, it describes a decoupled backend service for receiving minimap representations and applying visual analysis models to software artifacts, supporting more flexible model updates and future retraining. Third, it demonstrates how minimap-based empirical research can be transformed into reusable tool infrastructure, supporting practical scenarios in software comprehension, project characterization, and author identification, and paving the way for future extensions toward quality assessment, code smell detection, and project-level model customization.

2 Background and Related Work

This work builds directly on previous studies on source code minimaps. Gebara Jr. et al. [3] introduced source code minimaps as visual signatures for software feature detection, transforming source files into image-like representations and analyzing them with Local Binary Patterns and supervised classifiers. Their results showed that minimaps can preserve discriminative spatial and structural patterns of source code, supporting software stereotype classification without relying exclusively on manually defined rules or language-specific parsers. A subsequent study extended this line of investigation to a larger setting, analyzing minimap representations extracted from multiple projects and using deep learning models for the targets *Type*, *Project*, and *Author* [4]. *MinimapSignatures* builds on these studies by moving minimap-based analysis from offline experimental pipelines to an IDE-integrated tool infrastructure.

Software visualization tools have also been widely used to support program comprehension, maintenance, debugging, and quality assessment [6]. Classical and recent approaches represent software systems through dependency graphs, UML diagrams, architectural views, metrics dashboards, fault localization views, repository-level summaries, or interactive visualizations of source code relations [1, 7, 8, 10, 11]. CodePanorama [2] is particularly related because it explores zoomed-out and language-agnostic visual inspection of source code. However, these tools are primarily designed to support human interpretation, whereas source code minimaps in this work are treated mainly as machine-readable visual signatures that can be submitted to computational models.

Another related direction concerns machine learning representations for source code. Several approaches encode programs as tokens, embeddings, graphs, or long sequences, enabling predictive tasks such as bug prediction, code understanding, and software classification [5, 12]. More recently, CV4Code [9] explored source code as visual data, showing that image-like representations can support code understanding without depending exclusively on parsing or language-specific analysis. This perspective aligns with minimap-based analysis, as both approaches treat visual representations of code as inputs to learning models. Additionally, predicting developmental patterns like the *Author* target directly connects this visual paradigm to the broader concepts of source code stylometry and software authorship attribution.

Although these studies demonstrate the relevance of visual, structural, and learning-based representations of source code, there is still limited support for operationalizing such representations inside development environments. Existing tools commonly focus either on visual inspection by developers or on offline machine learning pipelines. In contrast, *MinimapSignatures* connects IDE plugins to a decoupled backend that accepts anonymized minimap representations and invokes visual analysis models. This architecture addresses a practical gap between previous empirical results and reusable tool support, while also enabling future extensions such as model retraining, project-level analysis, and new prediction targets related to quality, code smells, and maintainability.

3 MinimapSignatures: Tool Overview

MinimapSignatures is a multi-IDE software analysis tool that operationalizes source code minimaps as visual signatures inside practical development workflows. Instead of providing a standalone experimental pipeline, the tool integrates minimap generation and analysis into popular development environments. This design choice allows the implementation to reuse mature IDE capabilities, such as file navigation, editor interaction, and workspace management, while concentrating the tool design on its functional requirements: generating anonymized minimaps, invoking trained models, and presenting analysis results to developers.

The ecosystem is composed of two IDE clients, implemented for Visual Studio Code and IntelliJ IDEA, and a decoupled backend service responsible for visual source code analysis. The IDE clients transform source files into anonymized 128×128 grayscale minimaps and submit these visual signatures to the backend. The backend applies trained models associated with the prediction targets investigated in previous minimap-based studies: *Type*, *Project*, and *Author*. This separation keeps the IDE plugins lightweight and allows models to be updated, replaced, or extended independently from the user-facing integrations.

From the developer's perspective, *MinimapSignatures* acts as an auxiliary analysis layer. It does not replace existing IDE mechanisms such as syntax highlighting, static analysis warnings, dependency inspection, or code navigation. Instead, it complements them with model-based feedback derived from visual source code signatures.

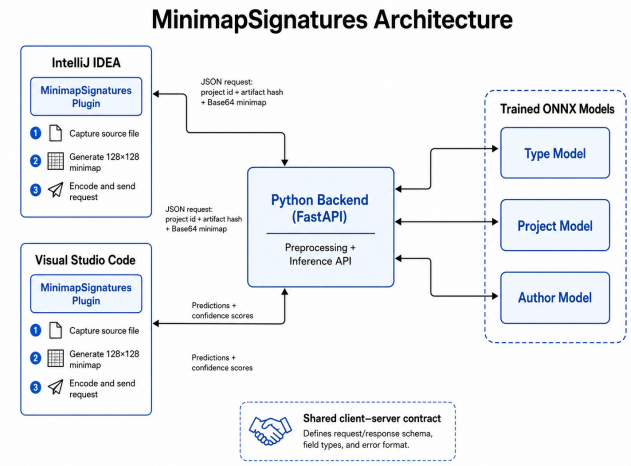

Figure 1: *MinimapSignatures* architecture. IntelliJ IDEA and Visual Studio Code plugins generate anonymized minimaps and submit them to a Python backend, which invokes the trained *Type*, *Project*, and *Author* models and returns prediction groups with confidence scores.

The current prototype focuses on file-level analysis, enabling developers to inspect individual source artifacts during maintenance, review, or exploratory comprehension activities.

The two IDE clients intentionally expose different interaction styles. The IntelliJ IDEA plugin provides Markdown-based split-screen feedback for quick inspections, while the Visual Studio Code extension provides a richer WebView-based interface for inspecting minimaps, prediction groups, and confidence scores. These modes support both routine file-level feedback and future analytical workflows involving multi-file, folder-level, or project-level analysis.

A central design principle of *MinimapSignatures* is extensibility. Additional IDEs can be supported by implementing the same minimap generation and communication protocol, while new models can be integrated into the backend without redesigning the clients. This makes the tool not only a practical implementation of minimap-based source code analysis but also a reusable infrastructure for future studies involving model customization, project-specific retraining, software quality assessment, code smell detection, and other visual or multimodal software engineering tasks.

4 Architecture and Implementation

The implementation of *MinimapSignatures* follows a client-server architecture composed of three components: a Python backend, an IntelliJ IDEA plugin, and a Visual Studio Code extension. The IDE clients capture source artifacts, generate minimap representations, and submit them to the backend, while the backend centralizes preprocessing, model execution, and prediction formatting (Figure 1). This separation keeps the plugins lightweight and allows analysis models to evolve independently from IDE integrations.

4.1 Backend Service

The backend service is implemented in Python using FastAPI. It exposes an HTTP endpoint that receives anonymized minimap representations from the IDE clients and returns structured predictions. FastAPI was adopted because it provides a lightweight API layer, request validation, automatic documentation, and straightforward deployment in containerized environments.

The main analysis endpoint receives a JSON payload containing the project identifier, the source artifact hash, and the Base64-encoded PNG image representing the source code minimap. When requested, the backend decodes the image, converts it to grayscale, normalizes pixel values, and reshapes the input according to the format expected by the deployed models. The current backend loads trained ONNX models using ONNX Runtime for the prediction targets investigated in previous minimap-based studies: *Type*, *Project*, and *Author*. ONNX was adopted to decouple model training from deployment, allowing models trained in different frameworks to be exported and executed through a common inference runtime.

The backend response follows a target-oriented prediction format. Instead of returning a single label, the service returns a list of prediction groups, each corresponding to an analysis target. Each group contains ranked candidate classes and their corresponding confidence scores. This structure allows multiple models to be executed over the same minimap representation and enables extending the platform with new prediction targets, such as software quality indicators or code smells, without changing the IDE clients.

The service also includes a simple API key mechanism based on the `X-Api-Key` HTTP header. This mechanism provides a basic access-control layer for the prototype, although it is not intended to provide complete security guarantees. The backend is packaged with Docker to simplify deployment and improve reproducibility across local, laboratory, and server-based environments.

4.2 Minimap Generation

The source code minimap is generated as a 128×128 grayscale image. Each row corresponds to one line of source code, and each column corresponds to one character position. The current implementation captures up to 128 lines and 128 characters per line. Longer lines are truncated, and missing positions remain empty. This fixed-size representation provides a compact and standardized input for computer vision models. Figure 2 illustrates the minimap generation process from source code to its anonymized visual representation.

Character mapping anonymizes textual content while preserving its structure. Whitespace and control characters are mapped to black pixels. Digits, uppercase letters, and lowercase letters are each collapsed into fixed grayscale categories, removing their original lexical identity. Symbols and punctuation, however, are preserved as visual categories because they encode relevant structural cues, such

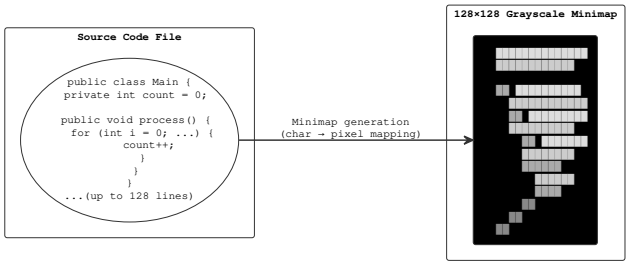

Figure 2: Source code minimap generation process. Source files are transformed into fixed-size 128×128 grayscale visual representations, preserving layout and structural cues while anonymizing lexical content.

as delimiters, operators, block boundaries, and expression patterns. This strategy prevents the minimap from being a readable screenshot of the source file and mitigates attempts to reconstruct the original code from the image. At the same time, it preserves visual regularities such as indentation, line density, block organization, symbol distribution, and structural rhythm.

The resulting representation is compact, language-independent at generation time, and suitable for computer vision models. Since the backend receives only the minimap image and the artifact identifier, raw source code does not need to be transmitted for inference. This design reduces the exposure of textual source content, although it should not be interpreted as a formal privacy guarantee.

While this character collapse and structural abstraction significantly mitigate the direct exposure of source code, they do not constitute formal, mathematically proven privacy guarantees. Important stylistic fingerprints, such as distinct indentation patterns (structural rhythm) and specific symbol distributions, are inherently preserved in the resulting visual signature. Consequently, deploying such an infrastructure in industrial settings or with proprietary code introduces potential risks, as advanced visual reverse-engineering or adversarial reconstruction attacks might still deduce architectural layouts. Therefore, future work must thoroughly evaluate these safety boundaries before integrating visual signatures into closed-source commercial workflows.

4.3 IDE Clients

The IDE clients implement the integration between *MinimapSignatures* and the developer environment. Although they target different platforms, both clients follow the same analysis workflow: they capture the active source file, generate a minimap locally, compute an artifact hash, encode the minimap as a Base64 PNG, submit the request to the backend, and present the predictions in the IDE.

The IntelliJ IDEA plugin is implemented in Java using the IntelliJ Platform SDK. It registers an action in the IDE interface, allowing developers to analyze the file currently opened in the editor. In the initial implementation, the backend response was displayed through IDE alert messages, providing immediate but limited feedback. As the interaction design evolved, the plugin adopted a Markdown-based presentation: after receiving the backend response, it generates a Markdown file with the prediction results and opens it in a split-screen view beside the editor window. This evolution preserves the goal of concise feedback while providing a more structured and readable representation of the analysis results during routine development activities.

The Visual Studio Code extension is implemented in TypeScript using the VS Code Extension API. It provides commands for analyzing the active file and for discovering source files in the workspace. After submitting the minimap to the backend, the extension opens a WebView panel beside the source editor. This panel displays the analyzed file, the generated minimap, the artifact hash, and prediction cards with confidence scores for each analysis target. This interaction style supports more detailed exploratory analysis and is aligned with future scenarios involving multiple files, folders, or complete projects.

The differences in the result presentation strategies between the two IDE clients are intentional. At this stage of the project, the

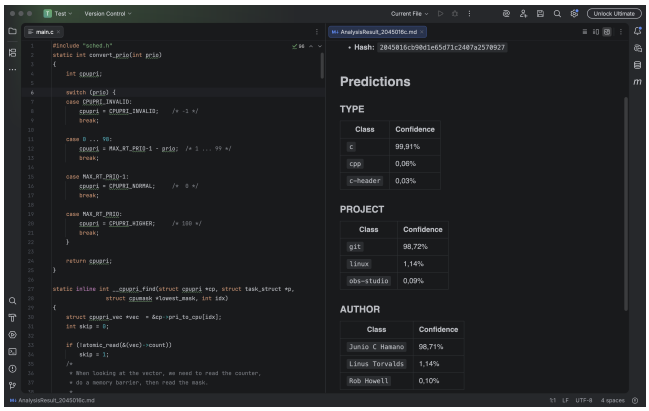

Figure 3: IntelliJ IDEA plugin showing Markdown-based feedback in split-screen generated from source code analysis.

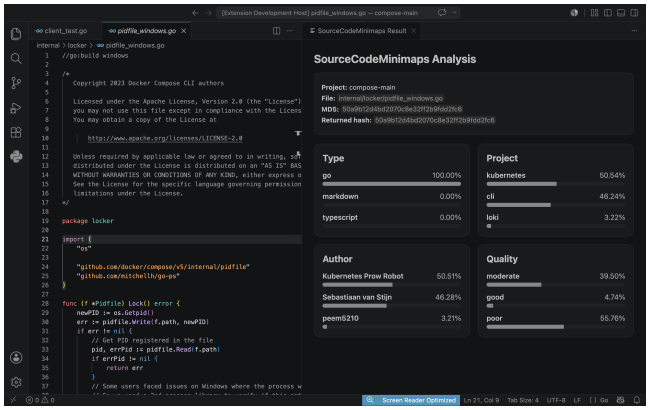

Figure 4: Visual Studio Code extension displaying the WebView interface with minimap visualization and prediction groups.

IntelliJ IDEA plugin (Figure 3) and the Visual Studio Code extension (Figure 4) adopt distinct user experience patterns to explore how developers interact with minimap-based feedback in different environments. These alternatives will support future evaluations comparing usability, clarity, interruption level, and developer preference. Based on these evaluations, we plan to define a more consistent look for the current plugins and for future IDE integrations.

Both IDE clients communicate with the backend through the same JSON-based contract. The request contains the project identifier, the source artifact hash, and the Base64-encoded minimap image. The response contains the analyzed hash and a list of prediction groups, each associated with an analysis target such as *Type*, *Project*, or *Author*. Each group returns ranked candidate classes with confidence values. This stable contract separates IDE-specific interaction from model-specific inference: the plugins only need to generate minimaps and submit requests using the agreed format, while the backend remains responsible for preprocessing, model execution, and prediction formatting. As a result, backend models can be replaced, retrained, or extended without requiring changes in the IDE clients, and additional IDE integrations can be developed by implementing the same protocol.

5 Usage Scenario and Demonstration

This section describes a representative use case for *MinimapSignatures* and summarizes the demonstration workflow for the tool. The scenario focuses on the current file-level analysis supported by the prototype and illustrates how minimap-based inference can be executed directly from the development environment, without requiring developers to export source files, execute external scripts, or interact with standalone experimental pipelines.

Consider a developer working on a multi-language software project during maintenance or code review. While inspecting a source file, the developer wants to obtain additional evidence to determine whether the file exhibits visual patterns compatible with a known source code type, project family, or authorial style. The developer triggers the analysis action from the active editor. The IDE client then captures the source artifact, generates the anonymized minimap locally, submits the visual signature to the backend, and presents the returned predictions inside the IDE.

In IntelliJ IDEA, the interaction is designed for structured and comprehensive feedback. The developer invokes the plugin action from the IDE interface, and the prediction result is displayed through a dynamically generated Markdown file opened in a split-screen view beside the editor window. This approach delivers a lightweight, native experience for quick, non-disruptive inspections during ordinary development. In Visual Studio Code, the same analysis pipeline is exposed through a WebView panel beside the source editor, showing the analyzed file, the generated image, the artifact hash, and prediction cards with confidence scores for each analysis target.

To illustrate this workflow with a concrete example, when a developer triggers the analysis on a C source file (such as the `main.c` file shown in Figure 3), the IDE client immediately reduces it to an anonymized 128×128 visual signature. Once received by the Python backend, this minimap is processed by the trained classification models. The backend then returns structured predictions that are rendered back in the editor. As shown in the IntelliJ IDEA Markdown interface (Figure 3), the system displays the prediction groups along with their respective confidence scores, outputting "Type: c" with 99.91% confidence, "Project: git" with 98.72%, and "Author: Junio C Hamano" with 98.71%. This explicit feedback allows developers to quickly cross-check if the visual signature of their active file aligns with expected project and authorial patterns.

The demonstration video follows this end-to-end workflow. First, it introduces the intuition behind source code minimaps as compact visual signatures. Then, it presents the backend service and both IDE clients. Finally, it demonstrates the same source code analysis process in IntelliJ IDEA and Visual Studio Code, showing how the plugins generate minimaps, communicate with the backend, and display the returned predictions.

This scenario also exposes an important design insight. Document-based feedback, as implemented via Markdown in the IntelliJ IDEA plugin, provides a clean, native-feeling interface for code-level inspection. In contrast, the graphical Visual Studio Code WebView interface suggests a more visual analysis space, using prediction cards for distinct analysis targets. Thus, the demonstration does not only show that the same backend can serve multiple IDEs; it also motivates future work on interaction design and user preference between textual and graphical presentation formats in software engineering tools.

Table 1: Functional validation of the current and planned *MinimapSignatures* prototype.

Component or feature	Status
Backend receives Base64 minimaps	Validated
Backend invokes trained models	Validated
Backend returns prediction groups	Validated
IntelliJ plugin submits minimaps	Validated
IntelliJ plugin displays Markdown results	Validated
VS Code extension submits minimaps	Validated
VS Code extension displays WebView cards	Validated
Shared client–server contract	Validated
Multi-file and project-level analysis	Future
Project-specific model update	Future

6 Preliminary Functional Evaluation

The preliminary evaluation of *MinimapSignatures* focuses on functional validation and integration feasibility. At this stage, the goal is not to conduct a controlled user study or to report a new empirical assessment of model accuracy. Instead, the evaluation verifies whether the proposed ecosystem can execute the complete minimap-based analysis workflow across different IDEs: source file capture, local minimap generation, anonymization, client-server communication, backend inference, and result presentation inside the development environment.

The first level of evidence comes from previous studies on source code minimaps. Gebara Jr. et al. [3] showed that minimaps combined with texture descriptors and supervised classifiers can support software feature detection and stereotype classification, with Random Forest reaching up to 93% accuracy in the reported experimental setting. A subsequent large-scale study extended minimap-based analysis to hundreds of thousands of source code representations extracted from multiple projects and programming languages, using deep learning models for the targets *Type*, *Project*, and *Author* [4]. These results support the main assumption behind the tool: minimaps preserve discriminative spatial and structural information that can be exploited by computational models.

The second level of evaluation concerns the current prototype. We validated the complete workflow using the Python backend, the IntelliJ IDEA plugin, and the Visual Studio Code extension. In both IDEs, the client successfully captures the active source file, generates a 128×128 grayscale minimap, computes an artifact hash, encodes the minimap as a Base64 PNG image, submits the request to the backend, receives structured predictions, and presents the results to the user. The same backend endpoint and JSON contract were used by both clients, confirming that the architecture supports multiple IDE integrations without changing the inference service.

Table 1 summarizes the validation status. Backend validation covered request handling, preprocessing, model invocation, and response formatting, while IDE validation confirmed minimap generation, submission, and result presentation in both clients.

Overall, the preliminary functional evaluation indicates that *MinimapSignatures* successfully moves minimap-based analysis from offline scripts to an integrated multi-IDE workflow. The prototype

demonstrates that source code minimaps can be generated locally, anonymized, transmitted to a backend, processed by trained models, and returned as IDE feedback. This provides a functional basis for future evaluations involving larger datasets, additional prediction targets, multi-file analysis, project-level model customization, and controlled studies with developers.

7 Limitations

The current version of *MinimapSignatures* has limitations that should be considered when interpreting the results. First, the evaluation presented in this paper is functional and integration-oriented. It verifies whether the tool workflow operates correctly across the backend and the two IDE clients, including minimap generation, anonymization, client-server communication, model invocation, and result presentation. However, it does not yet measure usefulness, usability, cognitive effort, trust, or impact on developer productivity through a controlled user study.

Second, although the backend is designed to support the three prediction targets derived from previous minimap-based studies – *Type*, *Project*, and *Author* – the present evaluation focuses on the operational integration of these models into the tool ecosystem rather than on a new empirical assessment of their predictive performance. Third, as discussed in Section 4.2, the structural nature of collapsed representations does not guarantee formal privacy against reverse-engineering, introducing potential cybersecurity and industrial risks if public models are exploited.

Finally, the current prototype performs file-level analysis using fixed-size 128×128 minimaps. This representation is compact, but it may discard information from long files. Additionally, our functional validation relied on a limited scope of programming languages and project sizes, which may not fully represent diverse industrial environments. Furthermore, while the integration with the *minimap-learner* repository is documented, the actual technical effort required for third-party developers to train and deploy custom models remains to be formally quantified. Future versions should therefore investigate multi-file analysis, alternative resolutions, and mechanisms to support model customization and retraining.

8 Conclusion and Future Work

This paper presented *MinimapSignatures*, a multi-IDE ecosystem for source code analysis based on minimap visual signatures. The tool integrates plugins for IntelliJ IDEA and Visual Studio Code with a decoupled Python backend for model inference, allowing source files to be transformed locally into anonymized 128×128 grayscale minimaps and analyzed through machine learning models. By bringing minimap generation into popular development environments, the implementation could focus on the functional requirements of the proposed approach instead of rebuilding basic editor features already provided by mature IDEs.

The architectural separation between lightweight IDE plugins and backend services isolates source capture from model execution. This decoupling enables continuous model updates and extensions without requiring frequent changes to the user interface.

The prototype also provided initial insights into interaction design for minimap-based feedback. In the IntelliJ IDEA plugin, the evolution from simple alert messages to a Markdown-based split-screen view suggests that lightweight textual reports can support non-disruptive, side-by-side inspections during routine development activities. In contrast, the richer WebView interface implemented in the Visual Studio Code extension explores a more graphical alternative, suggesting that detailed visual feedback is better suited for exploratory scenarios involving multiple files, folders, or entire projects. This indicates that future versions should support both interaction modes: a lightweight textual format for routine coding and a richer graphical view for exploratory analysis.

A further direction concerns model customization, evolution, and the expansion of prediction targets. Allowing organizations or developers to update models with project-specific minimap collections may provide substantial value, since each company, team, or repository may exhibit particular architectural conventions, coding styles, and recurring visual patterns. Therefore, a natural next step is to provide mechanisms for submitting complete projects or selected project fragments to expand the training base, enabling continuous dataset growth, model retraining, and organization-specific adaptation. In addition to the current targets – *Type*, *Project*, and *Author* – future studies will investigate new analysis targets, such as software quality indicators, code smells, architectural deviations, and other maintainability-related properties. These targets may require not only file-level classification but also more fine-grained strategies, such as region-based analysis or segmentation of minimap regions associated with relevant structural patterns.

Future work will proceed in this direction by extending *MinimapSignatures* from single-file analysis to multi-file and project-level workflows, integrating complete trained models for project identification and author identification, and improving the IDE interfaces based on developer feedback. We also plan to incorporate additional model architectures, including convolutional neural networks, Vision Transformers, segmentation-based models, and explainable AI techniques for highlighting relevant minimap regions. Overall, *MinimapSignatures* establishes a reusable infrastructure for bringing visual source code signatures into practical development environments and for supporting future research on AI-assisted and multimodal software analysis.

Artifact Availability

The artifact package includes the complete source code, deployment instructions, and full demonstration workflow for the Visual Studio Code extension, the IntelliJ IDEA plugin, and the Python backend service. The primary tool ecosystem code is hosted at https://github.com/minimap-project, and the model training pipeline is available at https://github.com/munifgebara/minimap-learner. The benchmark datasets used for model evaluation and the archived artifact accompanying this paper are deposited on Zenodo at https://zenodo.org/records/16929672 and https://doi.org/10.5281/zenodo.22151359, respectively. The repositories also provide detailed documentation describing how custom minimap datasets can be produced from user-provided projects. Software is released under the MIT License, and the accompanying documentation is available under a Creative Commons (CC) license.

Acknowledgements

The authors thank CAPES - Finance Code 001 and CNPq for their support. Igor Wiese thanks CNPq (409359/2024-6, 444802/2024-0) and Fundação Araucária/Governo do Paraná (PRD2023361000043).

References

[1] Ra'Fat Al-Msie'deen. 2025. ScaMaha: A Tool for Parsing, Analyzing, and Visualizing Object-Oriented Software Systems. *International Journal of Computing and Digital Systems* 17, 1 (2025), 1–20. doi:10.12785/ijcds/1571046420

[2] Marc Etter and Farhad Mehta. 2022. CodePanorama: A Language Agnostic Tool for Visual Code Inspection. In *Proceedings of the 30th International Conference on Program Comprehension (ICPC '22)*. ACM, New York, NY, USA, 4 pages. doi:10.1145/3524610.3527874

[3] Munif Gebara Jr., Igor S. Wiese, and Yandre M. G. Costa. 2025. Software Feature Detection using Source Code Minimaps as Visual Signatures. In *Proceedings of the 32nd International Conference on Systems, Signals and Image Processing (IWSSIP '25)*. IEEE. doi:10.1109/IWSSIP66997.2025.11151888

[4] Munif Gebara Jr., Igor S. Wiese, Hua-Liang Wei, and Yandre M. G. Costa. 2025. Source Code Minimaps at Scale: Feature-Oriented Analysis of 100 Projects. In *Proceedings of the 28th Iberoamerican Congress on Pattern Recognition (CIARP '25)*. Bogotá, Colombia. To appear.

[5] Md Nadim, Debajyoti Mondal, and Chanchal K. Roy. 2022. Leveraging Structural Properties of Source Code Graphs for Just-in-Time Bug Prediction. *Automated Software Engineering* 29, 1, Article 27 (2022). doi:10.1007/s10515-022-00326-0

[6] José Pereira dos Reis, Fernando Brito e Abreu, Glauco de Figueiredo Carneiro, and Craig Anslow. 2022. Code Smells Detection and Visualization: A Systematic Literature Review. *Archives of Computational Methods in Engineering* 29, 1 (2022), 47–94. doi:10.1007/s11831-021-09566-x

[7] Martin Pinzger, Katja Gräfenhain, Patrick Knab, and Harald C. Gall. 2008. A Tool for Visual Understanding of Source Code Dependencies. In *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC '08)*. IEEE Computer Society, Los Alamitos, CA, USA, 254–259. doi:10.1109/ICPC.2008.23

[8] Abdul Qayum, Saif Ur Rehman Khan, Inayat-Ur-Rehman, and Adnan Akhunzada. 2022. FineCodeAnalyzer: Multi-Perspective Source Code Analysis Support for Software Developer Through Fine-Granular Level Interactive Code Visualization. *IEEE Access* 10 (2022), 20496–20513. doi:10.1109/ACCESS.2022.3151395

[9] Ruibo Shi, Lili Tao, Rohan Saphal, Fran Silavong, and Sean Moran. 2023. CV4Code: Sourcecode Understanding via Visual Code Representations. In *Computer Vision – ACCV 2022 (Lecture Notes in Computer Science, Vol. 13842)*. Springer, Cham, 291–306. doi:10.1007/978-3-031-26284-5_18

[10] Harsda Shrivastava, Lohitha Kanisettypalli, Jiya Borikar, Adhikari Bhavishya, and Aiswariya Milan K. 2025. CodeSketch: A Real-Time Multi-Language Code Visualization Tool with UML and Animation Support—A Comparative Study. In *Proceedings of the 2025 9th International Conference on Computational System and Information Technology for Sustainable Solutions (CSITSS '25)*. IEEE. doi:10.1109/CSITSS67709.2025.11294623

[11] Desheng Sun, Xiaoqi Yue, Chao Liu, Hongxing Qin, and Haibo Hu. 2024. SFLVis: Visual Analysis of Software Fault Localization. *Journal of Visualization* 27, 4 (2024), 585–602. doi:10.1007/s12650-024-00979-x

[12] Xueqi Yang, Mariusz Jakubowski, Li Kang, Haojie Yu, and Tim Menzies. 2025. SparseCoder: Advancing Source Code Analysis with Sparse Attention and Learned Token Pruning. *Empirical Software Engineering* 30, Article 38 (2025). doi:10.1007/s10664-024-10558-1