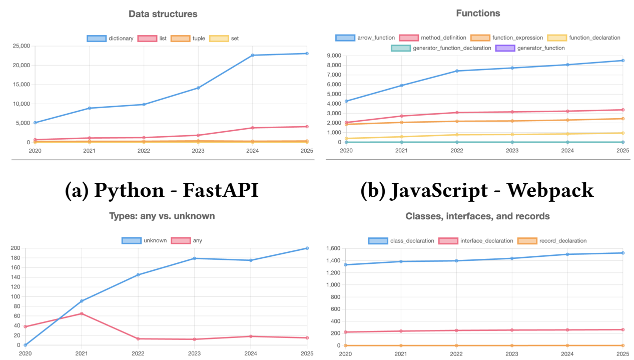
GitEvo: Code Evolution Analysis for Git Repositories

Andre Hora
Department of Computer Science, UFMG
Belo Horizonte, Brazil
andrehora@dcc.ufmg.br

Abstract

Analyzing the code evolution of software systems is relevant for practitioners, researchers, and educators. It can help practitioners identify design trends and maintenance challenges, provide researchers with empirical data to study changes over time, and give educators real-world examples that enhance the teaching of software evolution concepts. Unfortunately, we lack tools specifically designed to support code evolution analysis. In this paper, we propose GitEvo, a multi-language and extensible tool for analyzing code evolution in Git repositories. GitEvo leverages Git frameworks and code parsing tools to integrate both Git-level and code-level analysis. We conclude by describing how GitEvo can support the development of novel empirical studies on code evolution and act as a learning tool for educators and students. GitEvo is available at: https://github.com/andrehora/gitevo.



(a) Python - FastAPI **(b) JavaScript - Webpack**

(c) TypeScript - Docusaurus **(d) Java - Mockito**

Figure 1: Examples of code evolution reports by GitEvo.

CCS Concepts

• **Software and its engineering** → **Software maintenance tools**.

Keywords

Software Evolution, Git, Mining Software Repositories, Python

ACM Reference Format:
Andre Hora. 2026. GitEvo: Code Evolution Analysis for Git Repositories. In *23rd International Conference on Mining Software Repositories (MSR '26), April 13–14, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3793302.3793327

1 Introduction

Software evolution assessment provides multiple benefits for practitioners, researchers, and educators [3, 4, 11, 14, 15, 17]. Practitioners can identify design trends and potential maintenance challenges. Researchers can assess empirical data to investigate how software evolves over time. Educators can draw on real-world examples to enrich the teaching of software evolution concepts.

Nowadays, most real-world software systems are managed using the version control system Git [9, 27]. Thus, a common approach to exploring software evolution is to rely on tools that analyze Git repositories, such as GitPython [10] and PyDriller [22, 24] for Python, JGit [13] for Java, and isomorphic-git [12] for JavaScript, to name a few. These Git-level tools allow practitioners to assess repository changes over time, such as commits, authors, and file modifications. However, these tools do not support code-level analysis; they are limited to extracting Git-level data. As a result, they cannot provide insights into how the *source code* itself evolves.

On the other hand, code-level analysis can be performed using code parsing tools. There is a wide variety of such tools available. Some tools are more generic and focus on processing syntax trees, such as the AST module [23] for Python, JavaParser [25] and Spoon [20, 26] for Java, Babel parser [2] for JavaScript, and Tree-sitter [28] for multiple programming languages. Many other tools serve more specific purposes, for example, computing code complexity [16], LOC [5], metrics [1], and code changes [8, 19]. However, these code-level tools operate independently of the Git version control system and are therefore limited to analyzing isolated code snapshots or individual files, rather than capturing the evolution of code over time.

This scenario brings to light two main issues. *First*, there is a lack of tools specifically designed to support code evolution analysis, that is, tools that combine both Git-level *and* code-level assessment. *Second*, existing solutions are often language-dependent. For example, to analyze the evolution of Python code, one might use GitPython together with the AST module. However, performing a similar analysis for another language, such as Java, would require re-implementing the solution using JGit and Spoon.

We propose GitEvo, a multi-language and extensible Python-based tool for analyzing code evolution in Git repositories.[1] GitEvo currently supports the analysis of Python, JavaScript, TypeScript, and Java. It leverages Git and code parsing tools to integrate Git-level and code-level analysis. Figure 1 presents reports generated by GitEvo to analyze code evolution in multiple languages.

ACM ISBN 979-8-4007-2474-9/2026/04
https://doi.org/10.1145/3793302.3793327

[1] https://github.com/andrehora/gitevo

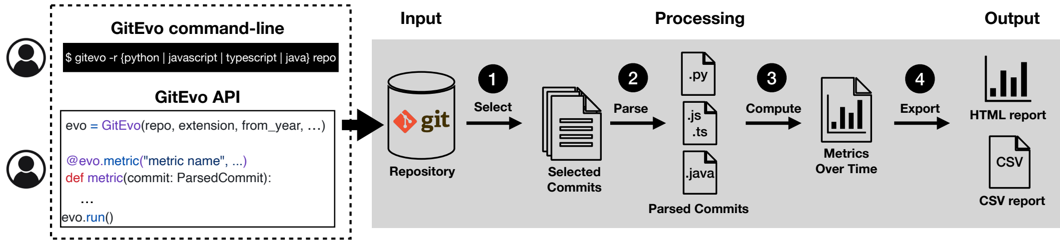


Figure 2: Overview of the GitEvo pipeline.

GitEvo relies on GitPython [10] and PyDriller [24] for Git-level analysis, and on Tree-sitter [28] for multi-language, code-level assessment (Section 2). It enables users to write *scripts* entirely in Python to compute code evolution metrics at the level of the concrete syntax tree (CST), leveraging Tree-sitter for parsing. For example, users can extract evolution metrics about various program constructs, such as classes, methods, functions, decorators, variables, parameters, types, and data structures, to name a few.

Finally, we showcase practical applications of GitEvo (Section 3). For instance, we describe how we employed GitEvo in two large-scale empirical studies and in an undergraduate Software Engineering course to support over 90 students exploring the evolution of real-world software systems.

Novelty. GitEvo's novelty lies in its ability to combine both Git-level and code-level analyses within a single tool, reducing the need for multiple tools or programming languages. In our experience, this approach reduces effort, allowing users to focus on what information needs to be analyzed over time rather than on which technologies should be adopted.

2 GitEvo

2.1 Overview

Figure 2 provides an overview of the GitEvo pipeline. GitEvo can be used from the command-line (Section 2.2) or programmatically via API (Section 2.3). When using via API, GitEvo is extensible, and users can define custom metrics. GitEvo receives as input one or more Git repositories and provides as output HTML and CSV reports of the code-level metrics over time. As an example, we make dozens of HTML reports available in `gitevo-examples`.[2]

2.2 Usage via Command-Line

The simplest way to use GitEvo is via the command-line. After installing GitEvo, we can analyze the evolution of a Git repository:

```
# Installing GitEvo
$ pip install gitevo
# Basic command-line usage
$ gitevo -r {python|javascript|typescript|java} repo
```

`repo` accepts (1) a Git URL, (2) a local repository, or (3) a directory containing multiple Git repositories. GitEvo provides as output HTML and CSV reports.

[2]https://github.com/andrehora/gitevo-examples

For example, the following commands analyze the source code evolution of Flask (Python),[3] Axios (JavaScript),[4] VueJS-core (TypeScript),[5] and Mockito (Java).[6] The resulting HTML reports are all available in our `gitevo-examples` repository.

```
# Using GitEvo to analyze Python, JS, TS, and Java code
$ gitevo -r python https://github.com/pallets/flask
$ gitevo -r javascript https://github.com/axios/axios
$ gitevo -r typescript https://github.com/vuejs/core
$ gitevo -r java https://github.com/mockito/mockito
```

When using GitEvo via the command-line, the computed code metrics are predefined within the tool.[7] For example, for Python, GitEvo computes lines of code, data structures, function parameters, and many more. To define custom metrics, users can use GitEvo through its API, as described next.

2.3 Usage via API: Defining Custom Metrics

Overview. The most powerful way to use GitEvo is programmatically through its API. In this mode, users can implement Python scripts to define custom code evolution metrics at the level of the concrete syntax tree (CST). These scripts are executed by GitEvo, which automatically generates HTML and CSV reports. Behind the scenes, the script performs four major steps: (1) selects representative commits, (2) parses source files, (3) computes metrics, and (4) export reports, as illustrated in Figure 2.

Example 1 (basic evolution metrics). Figure 3 presents an example of a GitEvo script that computes four metrics: Lines of Code, Python files, Test files, and LOC per Python file.[8] The Lines of Code metric (lines 6–8) produces the chart shown in Figure 4.

API. GitEvo provides three key classes that can be used in the scripts: `GitEvo`, `ParsedCommit`, and `ParsedFile`. `GitEvo` is the main class, the entry point to use the tool. It receives as input the repository, file extension, date unit for analysis, and start/end year for analysis. GitEvo can analyze code evolution yearly (first day of each year) and monthly (first day of each month). By default, GitEvo analyzes the last five years.

Metrics are defined in functions decorated with `@evo.metric()`. The metric decorator can include the metric name and other arguments to customize the metric output. Notice that the metric function receives as an argument a `ParsedCommit`.

[3]https://andrehora.github.io/gitevo-examples/python/flask.html
[4]https://andrehora.github.io/gitevo-examples/javascript/axios.html
[5]https://andrehora.github.io/gitevo-examples/typescript/vuejs-core.html
[6]https://andrehora.github.io/gitevo-examples/java/mockito.html
[7]https://github.com/andrehora/gitevo/tree/main/gitevo/reports
[8]https://github.com/andrehora/gitevo/blob/main/examples/ex_basic.py

```python
from gitevo import GitEvo, ParsedCommit

git_url = 'https://github.com/pallets/flask'
evo = GitEvo(repo=git_url, extension='.py')

@evo.metric('Lines of code (LOC)')
def loc(commit: ParsedCommit):
    return commit.loc

@evo.metric('Python files')
def python_files(commit: ParsedCommit):
    return len(commit.parsed_files)

@evo.metric('Test files')
def test_files(commit: ParsedCommit):
    test_files = [f for f in commit.parsed_files if 'test_' in f.name.lower()]
    return len(test_files)

@evo.metric('LOC / Python files')
def loc_per_file(commit: ParsedCommit):
    python_files = len(commit.parsed_files)
    if python_files == 0: return 0
    return commit.loc / python_files

evo.run()
```

Figure 3: Example of a basic GitEvo script.

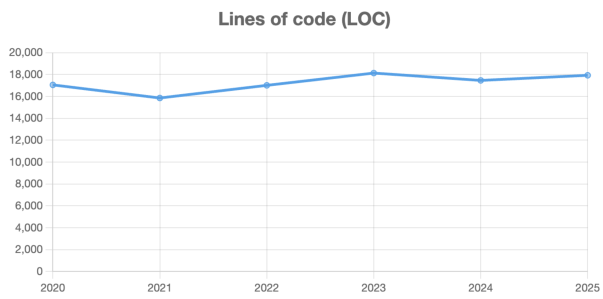


Figure 4: Lines of code over time (Flask).

The ParsedCommit class represents a parsed commit and contains (1) a list of ParsedFile and (2) a list of tree_sitter.Node.[9] ParsedCommit also contains multiple methods (e.g., find_node_types and loc_by_type) and properties (e.g., commit hash and loc). A ParsedFile represents a parsed file in a commit, including properties as name, path, and tree-sitter nodes. A tree_sitter.Node is a single node within the CST, including type, parent, and children.

Example 2 (metrics based on node types). Figure 5 presents two metrics (Data structures and Loop) that use find_node_types to count nodes by type.[10] The Data structures metric (lines 1–4) produces the chart shown in Figure 6.

Example 3 (metrics based on node content). Figure 5 presents two metrics (Async functions and @pytest decorated functions) that are computed from the node content, leveraging the tree_sitter.Node properties. The "@pytest decorated functions" metric (lines 7–12) produces the chart shown in Figure 8.

[9]https://tree-sitter.github.io/py-tree-sitter/classes/tree_sitter.Node.html
[10]https://github.com/tree-sitter/tree-sitter-python/blob/master/src/node-types.json

```python
1  @evo.metric('Data structures', categorical=True)
2  def data_structures(commit: ParsedCommit):
3      data_structure_types = ['dictionary', 'list', 'set', 'tuple']
4      return commit.find_node_types(data_structure_types)
5
6  @evo.metric('Loops', categorical=True)
7  def loops(commit: ParsedCommit):
8      loop_types = ['for_statement', 'while_statement', 'for_in_clause']
9      return commit.find_node_types(loop_types)
```

Figure 5: Metrics based on node types.

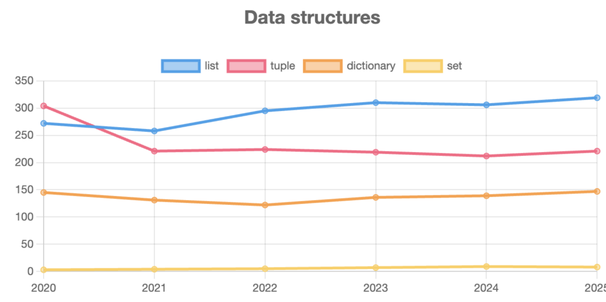


Figure 6: Data structures over time (Flask).

```python
1  @evo.metric('Async functions')
2  def async_functions(commit: ParsedCommit):
3      functions = commit.find_nodes_by_type(['function_definition'])
4      async_functions = [f for f in functions if as_str(f.child(0).text) == 'async']
5      return len(async_functions)
6
7  @evo.metric('@pytest decorated functions')
8  def decorated_functions(commit: ParsedCommit):
9      decorators = commit.find_nodes_by_type(['decorated_definition'])
10     decorated_functions = [d for d in decorators if d.child_by_field_name('definition').type == 'function_definition']
11     pytest_decorated = [dc for dc in decorated_functions if as_str(dc.child(0).text).startswith('@pytest')]
12     return len(pytest_decorated)
```

Figure 7: Metrics based on node content.

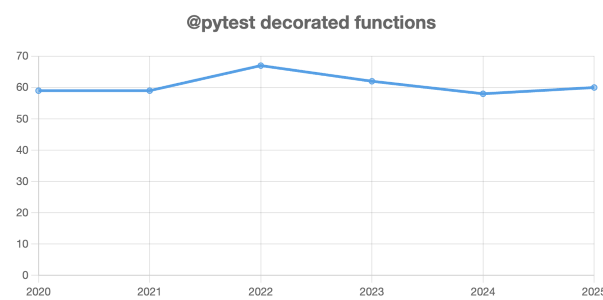


Figure 8: @pytest decorated functions over time (Flask).

2.4 Implementation Notes

GitEvo leverages GitPython [10] and PyDriller [24] for Git-level analysis, and Tree-sitter [28] for multi-language, code-level assessment. Behind the scenes, GitEvo orchestrates the analysis by creating a ParsedCommit object for each analyzed commit and injecting this object into the metric function. This way, the user simply defines each metric once (in the metric function), and its value is automatically computed over time across all analyzed commits.

GitPython and PyDriller. GitEvo relies on GitPython and PyDriller to clone/open repositories, iterate over commits, and retrieve the source code in each commit. However, these tools do not perform code analysis. To analyze source code across multiple programming languages, a parser such as Tree-sitter is required.
Tree-sitter. GitEvo relies on Tree-sitter to parse the source code of each target commit. GitEvo identifies the target programming language and parses the source code using the corresponding Tree-sitter grammar. The parsed code can be accessed by the user through the classes `ParsedCommit` and `ParsedFile`. Currently, GitEvo supports four grammars: tree-sitter-python [31], tree-sitter-js [30], tree-sitter-ts [32], and tree-sitter-java [29].

3 Practical Applications

3.1 Empirical Studies and Datasets

GitEvo can support the development of novel studies on code evolution across multiple programming languages. This allows researchers to analyze repositories written in different languages while writing analysis scripts in Python. As an example, we detail two large-scale empirical studies in which we employed GitEvo.
Mocking practices over time. Developers can use mocking in tests to isolate dependencies, making the test fast and repeatable [18, 21]. We analyzed more than 1.2 million commits across 2,168 TypeScript, JavaScript, and Python repositories, identifying 44,900 commits that add mocks to tests (see Table 1). Although multiple programming languages were analyzed, all scripts were implemented in Python itself with the support of GitEvo.

Table 1: Dataset of commits adding mocks to tests.

Commits	TS	JS	Python	Total
All commits	835,781	98,389	320,708	1,254,878
Mock commits	23,838	1,561	19,501	44,900

Functional features over time. We assessed 10 years of functional Python feature usage across three major open-source projects: CPython, Pandas, and Django. We analyzed six functional features: lambdas, yield statements, generator expressions, list comprehensions, dictionary comprehensions, and set comprehensions [33]. We found that *lambdas* are the most frequently used functional feature, with 9,988 cases, followed by *yield* statements (5,552) and *list comprehensions* (5,130). In contrast, generator expressions (2,454), dictionary comprehensions (657), and set comprehensions (288) are less frequently used. The metrics were computed entirely using GitEvo scripts.[11]

Practical Application 1: GitEvo can support the development of novel empirical studies on code evolution across multiple programming languages.

3.2 Custom Reports for Practitioners

GitEvo can also be used by practitioners to monitor the trends of specific metrics in their repositories. As an example, we created a custom report for the FastAPI web framework [7], which includes several metrics related to web development.[12]

Moreover, we presented GitEvo to practitioners via GitHub Discussions and received several positive feedback comments. For instance, one developer stated: "*really great to take a step back and see some of the work we have done at this level!*".[13] Other developers suggested new features,[14] improvements,[15] and clarifications.[16] Although GitEvo is still a prototype, these interactions demonstrate its potential to attract practitioners' interest.

Practical Application 2: GitEvo can help practitioners visualize and understand how their software evolves.

3.3 Education in Software Evolution

We envision GitEvo as a learning tool to help students gain a deeper understanding of software evolution. In this context, we employed GitEvo in an undergraduate Software Engineering course to support students exploring the evolution of real-world software systems. The exercise was conducted with over 90 students and is publicly available [6] (each `fork` corresponds to an individual student's answer). The exercise consisted of four main steps: (1) select a repository to analyze, (2) install and run the GitEvo tool, (3) explore the code evolution report, and (4) explain a code evolution chart.

The students identified many explanations for the observed code evolution patterns by exploring project documentation and inspecting the source code. For example, they detected real cases of increasing code complexity, rising proportion of tests, and decreasing ratio of code comments. They also identified several project-specific design decisions, such as the preference for `const` over `var` and `let` (in JS) and the adoption of `record` (in Java), to name a few.

Practical Application 3: GitEvo can serve as a learning tool for educators and students to gain deeper insights into real software evolution and its associated challenges.

4 Conclusion

In this paper, we presented GitEvo, a Python-based tool that analyzes the evolution of code metrics across Git repository history. We also showcased practical applications of GitEvo for researchers, practitioners, and educators.

As future work, we plan to extend support to any programming language available in Tree-sitter, only requiring the user to install the proper grammar. We also plan to enhance the GitEvo API by adding more convenient methods for accessing common source code constructs, as classes, methods, functions, and decorators.

Acknowledgments

This research was supported by CNPq (process 403304/2025-3), CAPES, and FAPEMIG. This work was partially supported by INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence), www.ines.org.br, CNPq grant 408817/2024-0.

[11] https://github.com/andrehora/gitevo/blob/main/examples/python_functional.py

[12] https://andrehora.github.io/gitevo-examples/fastapi/dispatch.html

[13] https://github.com/pypa/pipenv/discussions/6376

[14] https://github.com/andrehora/gitevo/issues/1

[15] https://github.com/sveltejs/svelte/discussions/15784

[16] https://github.com/mkdocs/mkdocs/discussions/3966

References

[1] Maurício Aniche. 2015. *Java code metrics calculator (CK)*. Available in https://github.com/mauricioaniche/ck/.
[2] Babel parser. January, 2026. https://babeljs.io/docs/babel-parser.
[3] Aline Brito, Andre Hora, and Marco Tulio Valente. 2020. Refactoring graphs: Assessing refactoring over time. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 367–377.
[4] Aline Brito, Andre Hora, and Marco Tulio Valente. 2022. Understanding refactoring tasks over time: A study using refactoring graphs. In *Congresso Ibero-Americano em Engenharia de Software (CIbSE)*. 330–344.
[5] cloc. January, 2026. https://github.com/AlDanial/cloc.
[6] Exploring code evolution with GitEvo. January, 2026. https://github.com/andrehora/exploring-code-evolution.
[7] FastAPI. January, 2026. https://github.com/fastapi/fastapi.
[8] Daniel M German, Bram Adams, and Kate Stewart. 2019. cregit: Token-level blame information in git version control repositories. *Empirical Software Engineering* 24, 4 (2019), 2725–2763.
[9] Git. January, 2026. https://git-scm.com.
[10] GitPython. January, 2026. https://github.com/gitpython-developers/GitPython.
[11] Andre Hora, Danilo Silva, Marco Tulio Valente, and Romain Robbes. 2018. Assessing the threat of untracked changes in software evolution. In *International Conference on Software Engineering*. 1102–1113.
[12] isomorphic-git. January, 2026. https://github.com/isomorphic-git/isomorphic-git.
[13] JGit. January, 2026. https://github.com/eclipse-jgit/jgit.
[14] Meir Lehman and Juan C Fernáandez-Ramil. 2006. Software evolution. *Software evolution and feedback: Theory and practice* (2006), 7–40.
[15] Manny M Lehman. 1996. Laws of software evolution revisited. In *European workshop on software process technology*. Springer, 108–124.
[16] Lizard. January, 2026. https://github.com/terryyin/lizard.
[17] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. 2005. Challenges in software evolution. In *International Workshop on Principles of Software Evolution (IWPSE)*. 13–22.
[18] Gerard Meszaros. 2007. *xUnit test patterns: Refactoring test code*. Pearson Education.
[19] Jakub Narębski, Mikołaj Fejzer, Krzysztof Stencel, and Piotr Przymus. 2025. PatchScope–A Modular Tool for Annotating and Analyzing Contributions. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1–5.
[20] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46 (2015), 1155–1179.
[21] Gustavo Pereira and Andre Hora. 2020. Assessing mock classes: An empirical study. In *International Conference on Software Maintenance and Evolution (ICSME)*. 453–463.
[22] PyDriller. January, 2026. https://github.com/ishepard/pydriller.
[23] Python AST Module. January, 2026. https://docs.python.org/3/library/ast.html.
[24] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. Pydriller: Python framework for mining software repositories. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 908–911.
[25] Spoon. January, 2026. https://github.com/javaparser/javaparser.
[26] Spoon. January, 2026. https://spoon.gforge.inria.fr.
[27] Stack Overflow Developer Survey. January, 2026. https://survey.stackoverflow.co/2025.
[28] Tree-sitter. January, 2026. https://github.com/tree-sitter/tree-sitter.
[29] /tree-sitter-java. January, 2026. https://github.com/tree-sitter/tree-sitter-java.
[30] tree-sitter-javascript. January, 2026. https://github.com/tree-sitter/tree-sitter-javascript.
[31] tree-sitter-python. January, 2026. https://github.com/tree-sitter/tree-sitter-python.
[32] tree-sitter-typescript. January, 2026. https://github.com/tree-sitter/tree-sitter-typescript.
[33] Yi Yang, Ana Milanova, and Martin Hirzel. 2022. Complex Python features in the wild. In *International Conference on Mining Software Repositories (MSR)*. 282–293.