Alma Mater Studiorum · Università di Bologna

Dipartimento di Informatica — Scienza e Ingegneria
Bachelor's Degree in Computer Science (Informatica)

CODE COMMENTING PRACTICES IN PYTHON PROJECTS ACROSS DEVELOPMENT ERAS: AN EMPIRICAL STUDY

Supervisor:
Prof. SIMONE MARTINI

Presented by:
MICHELE GARAVANI

March 2026 - Graduation Session IV
Academic Year 2024/2025

Abstract

Source code comments are a basic tool for documenting software systems, but how developers actually use them changes over time. As languages evolve, tools improve, and habits shift. This thesis looks at that change in Python over roughly twenty years.

Sixteen open-source Python repositories were selected, eight from the 2000s and eight from the 2020s. A custom pipeline extracted comment blocks and measured density, structure, function, and wording. Docstrings and test code were included to round out the picture of how each project handled documentation.

The differences between the two eras are fairly consistent. Modern repositories have fewer comment blocks per line of code, more inline comments, and less multi-line comment sections. `TODO` has largely taken over from `FIXME` and `XXX` as the annotation marker of choice. Comments meant to be processed by automated tools, such as linters, appear far more often in recent code. Linguistically, comments show a shift towards a more formal syntax.

Overall, the findings indicate that commenting practices in Python have changed. Specifically, comments are now shorter, more focused and formal, and increasingly written with tools in mind rather than just future readers.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Writing software is mainly regarded as a purely technical act. However, software can also be considered a medium for communication between developers. Comments occupy a small but important part of this medium: they explain reasoning and document decisions that the code itself cannot express. Their volume in any given codebase is modest, yet their absence is felt quickly, particularly when someone unfamiliar with the original context tries to understand, extend, or debug the work.

Over the past two decades, Python's development ecosystem has changed considerably with new tooling, coding standards, and collaborative norms. These changes have all influenced how developers write and structure code. Yet surprisingly little empirical research has looked at whether commenting practices have kept pace with these shifts, or even changed at all.

This thesis examines how commenting practices have shifted in Python open-source repositories, comparing projects from the late 2000s with those from the 2020s. Through systematic analysis of comment density, structural patterns, annotation markers, tooling directives, and linguistic features, it asks a straightforward question: do developers comment differently today than they did two decades ago?

1.1 Motivation and Context

Source code comments serve several practical functions: they explain complex logic, record design decisions, and give future maintainers the context that the code alone cannot provide. In collaborative settings, they also act as a quiet form of communication between developers.

Empirical research treats commenting as an integral part of the development process, not an afterthought. Arafat and Riehle [1] found that comments appear throughout the lifecycle of open-source projects and that their density holds relatively steady regardless of project size, suggesting that developers comment out of habit and necessity, not just when a codebase gets large enough to warrant documentation.

What remains understudied is change over time. Most research focuses on comment density, quality, or usefulness within individual projects or pooled datasets, treating commenting as static. Fewer studies ask how the way developers write comments has

shifted across different eras of software development.

This study will focus on such analysis, examining how commenting practices in Python open-source repositories have shifted between the late 2000s and the 2020s.

By analyzing the structural, functional, and linguistic properties of comment blocks across both periods, it aims to build a clearer picture of what has actually changed in how developers use comments today versus a generation of code ago.

1.2 Research Questions

As mentioned above, this study aims to understand how comments have changed in code over roughly two decades: from the late 2000s to the mid 2020s. Four research questions guide this investigation, as presented. This study will respond directly to these questions, reporting the findings in Chapter 5 and discussing them in Chapter 6.

RQ1: How does the density of comments differ between Python projects from the 2000s and those from the 2020s?

This question investigates whether modern Python repositories contain more or fewer comments relative to their size. The analysis will look at metrics such as the number of comment blocks per file and the number of comment blocks per 1,000 lines of code, for each repository. The goal is to establish whether commenting has become more or less common as the Python ecosystem has matured.

RQ2: How has the structural composition of comments changed between the two eras?

This question examines how different kinds of comment blocks are used in each time period. In particular, the analysis compares the distribution of inline comments, full-line comments, and multi-line full-line comment blocks across the repositories in the dataset. Another important aspect under examination is the block length for each kind of comment. Changes in this distribution through the two eras may reflect shifting stylistic conventions in how developers structure comments in code.

RQ3: How have annotation markers and tooling directives in comments evolved between the two eras?

Developers use comments not just to explain code, but to flag work in progress (`TODO`, `FIXME`, `HACK`) or to communicate with automated tools (`noqa`, `pragma`, `type: ignore`). This study will define these different comment categories. This question tracks how prevalent these patterns are in each era. Differences in frequency may reflect changes in development practices, partly as a window into how important tooling has become in modern development.

RQ4: What differences exist in the linguistic style of comments between the two eras?

This question explores stylistic and linguistic aspects of comments. The analysis considers patterns such as punctuation usage, capitalization style, the presence of separators or emoticons, and whether comments tend to begin with imperative or descriptive verb

forms. These patterns are small, but collectively they may reveal patterns about how developers use comments and who they imagine reading them.

Together, these four questions cover both the measurable side of commenting (density, frequency, structural type) and a side that might be harder to quantify: what developers actually write and how they write it. The goal is to provide a clear picture of how commenting has changed across two distinct periods in Python's history.

1.3 Contributions

This thesis makes three concrete contributions. It presents a reproducible pipeline for extracting and analyzing comment blocks from Python source code repositories. It introduces a curated dataset of sixteen open-source Python projects spanning two eras, selected to enable systematic comparison. And it provides an empirical analysis of how comments have changed across those periods, examining all the structural and linguistic characteristics mentioned in the research questions.

1.4 Thesis Structure

The remainder of this thesis is organized as follows.

Chapter 2 covers the relevant background on source code comments and reports selected prior work on comment analysis in software engineering.

Chapter 3 describes the dataset used in this study, covering how repositories were selected and what the collected snapshots look like.

Chapter 4 presents the design and implementation of the pipeline used to extract comment blocks and docstrings and compute the features examined in this study.

Chapter 5 reports the empirical results, working through each research question (comment density, structure, annotation markers, tooling directives, linguistic features). It also reports results for other comment related findings that were not included in the research questions (such as legal headers, docstrings and test code findings).

Chapter 6 interprets the findings and addresses threats to validity.

Finally, Chapter 7 summarizes the main contributions and outlines directions for future work.

Chapter 2

Background

This chapter builds the conceptual foundation for the comment analysis, situating the study within prior research. Since this study investigates how commenting practices in Python open-source projects have evolved between the 2000s and the 2020s, it is important to first clarify what constitutes a comment in Python source code, and how different types of comments are used by developers.

2.1 Code Comments: Definitions and Taxonomy

In computer programming, comments are non-executable text fragments inside executable source code. This means that comments will be ignored by a compiler or interpreter.

As defined in [2], a comment is generally an annotation intended to make the code easier for a programmer to understand, often explaining aspects that are not readily apparent from the source code itself. However, in modern programming, comments serve multiple purposes, as will be shown.

Comment syntax differs across programming languages. For example, in the Java programming language, comments can be delimited at both ends using the syntax "`/*`" to begin a comment and "`*/`" to terminate it. This syntax allows comments to span multiple lines, written with an additional "`*`" at the beginning of each line within the block. According to the Oracle Java documentation [3], Java also supports end-of-line comments, which are preceded by the two characters "`//`" and continue until the end of the line.

In Python, the language on which this analysis focuses, comment syntax is more limited. Python supports only single-line comments preceded by the hash character "`#`", which extend until the end of the line [4]. This means that multi-line comments are written as consecutive single-line comments.

Python comments can appear either on their own line or inline with code. The following example illustrates both cases:

```
# This is a full-line comment
def sum(a, b):
    res = a + b  # This is an inline comment
    return res
```

For the comment analysis, consecutive lines of full-line comments at the same indentation level are grouped and treated as a single comment block, as they typically refer to the same code element or convey a single conceptual explanation.

A special form of comment can occur at the start of many executable Python scripts: the *shebang* line. This line begins with the characters `#!` (and thus is syntactically treated as a comment) followed by the path to the Python interpreter [5, p. 95]. An example is `#!/usr/bin/env python`. In this study, shebang lines are treated as a distinct category of comments.

In addition to regular comments, Python provides a structured form of documentation known as docstrings. A docstring is a string literal that appears as the first statement in a module, class, or function definition and is used to document the behavior and purpose of the corresponding object. Unlike regular comments, docstrings are accessible at runtime through the `__doc__` attribute and can be extracted by documentation generation tools. The conventions for writing docstrings are described in PEP 257 [6], which recommends the use of triple-quoted strings to provide concise descriptions of program elements and their intended usage.

The following code snippet illustrates the use of docstrings:

```
"""
This is a module docstring.  It explains the purpose of this module.
"""

def sum(a, b):
    """
    This is a function docstring.  It explains the purpose of this function.
    """
    res = a + b
    return res
```

In this study, docstrings are analyzed separately from regular comments. This is because they represent structured documentation rather than annotations within the code.

The conventions for comments and docstrings in Python are formally documented in two style guides: PEP 8 [7], which provides general guidelines for Python code style, and PEP 257 [6], which specifies conventions for writing docstrings.

Having defined the main forms of comments in Python, the following section analyzes their role and importance in software development.

2.2 Role of Source Code Comments in Software Development

In software, comments play an important role helping developers to understand the source code at hand [8]. While programming languages define the syntax and semantics, comments add a natural-language layer that explains the purpose and context of the code to human readers. As a result, comments are widely regarded as a key component of software documentation and maintenance.

Software literature also emphasizes the importance of comments as a means of communicating information that might not be immediately apparent from the code itself. Ousterhout argues that comments should provide insights that cannot be easily inferred from the program structure, such as the rationale behind design decisions or the broader context in which code operates [9]. According to this perspective, comments are most valuable when they complement the code with high-level explanations. They should not only restate what the code already explicitly conveys.

Empirical evidence points out that not all comments are helpful. A study by Steidl, Hummel, and Juergens (2013) [10] conducted an analysis of comments in large open-source software projects to examine their classifications and quality. In their first research question (RQ1), the authors investigated how many comments actually contribute to system understanding. Their results showed that a substantial portion of comments do not improve documentation quality, as many comments consist of copyright notices, task annotations (e.g., TODO), or commented-out code. Across the analyzed projects, between 18% and 49% of comments were copyright statements, while up to 11% consisted of commented-out code. The study therefore demonstrated that simple comment ratio metrics can be misleading, since a high number of comments does not necessarily indicate better documentation quality. By classifying comment types, the authors showed that only a subset of comments meaningfully contributes to improving system understanding.

These findings imply that meaningful results in source code comment analysis require a distinction between different types of comments. Therefore, this study adopts a classification based approach and subdivides comments into various categories according to their purpose and use.

These categories are presented in the following section and will be analyzed in the comparison between source code from the late 2000s and source code from the mid 2020s.

2.3 Comment Categories Studied

As discussed in the previous section, comments can serve a variety of purposes beyond simple documentation. In order to analyze commenting practices with as little bias as possible, this study categorizes comments according to several functional and linguistic characteristics. Once the comments have been classified, a temporal analysis of the ratios of comments belonging to the various categories will also be performed.

Each comment block extracted from the source code is analyzed and annotated with a set of features describing some of its structural properties, its purpose, and certain linguistic characteristics of its text. The analysis focuses on four major feature families: license header indicators, annotation markers, tooling directives, and linguistic features.

These four categories were chosen because they capture important aspects of how comment blocks are used in practice, and enable the analysis to compare commenting practices across repositories and thus time periods.

License Header Indicators

License headers are comments that contain legal information regarding copyright ownership, usage permissions, and redistribution conditions. Such comments usually appear at the beginning of source files and could include copyright notices, references to software licenses, or explicit license identifiers.

In this study, various license-related signals are identified. Subcategories include SPDX identifiers, copyright statements, text with license keywords, warranty disclaimers, and other common legal phrases.

License header comment blocks do not communicate any meaningful information about the evolution of commenting practices, as they consist largely of standardized legal text.

License headers are also not ideal indicators of a trend based on their ratio compared to the total number of comments, as there might be projects that use a license header block in every source-code file, while other projects might not make use of them at all or simply have a file named `LICENSE` or `LICENSE.txt`, without needing license headers as comments in the individual. files. This means that the overall ratio of license headers in comment blocks might differ excessively depending on the project.

However, it is nonetheless important to classify license headers as such. In fact, license headers might otherwise alter comment block ratios on specific characteristics, if not properly grouped and labeled.

Annotation Markers

Previous research has shown that developers use task annotations embedded in source code comments to manage development tasks and coordinate work during software development [11].

Such annotations are typically expressed through standardized keywords. The most common example is `TODO`, which is usually a placeholder for developer tasks that need attention but haven't been addressed yet [12]. Other classes of annotation markers include `FIXME` (used to mark code that needs addressing), `XXX` (used to draw attention to potentially problematic code), `HACK` (used to indicate a temporary or suboptimal implementation), `BUG`, and `NOTE`.

This study therefore records the presence and type of annotation markers for each comment block in order to analyze how the use of task-related annotations varies across repositories and time periods.

Tooling Directives

In modern software development, comments are also used to communicate with automated tools such as linters, type checkers, or formatters. These comments function as configuration directives embedded directly in the source code.

Common examples include directives such as `noqa` (used by linting tools to suppress warnings), `pragma` statements, `type: ignore` annotations used by static type checkers, or tool-specific comments for linters such as `pylint`. Shebangs and encoding declarations

(comments used to specify the character encoding of Python source files) also fall into this category [13].

This study records the various types of tooling comments to enable comparisons of tool-related comments across repositories and time periods.

Linguistic Features

Beyond their functional roles, comments also simply communicate information or context to developers. Therefore, an important part of a comment analysis is exploring its linguistic and textual features.

The extracted features include punctuation patterns (such as comments ending with a period, question mark, or exclamation mark), capitalization patterns, and the presence of certain textual elements such as URLs, numbers, emoticons, or decorative separator lines. Additional features capture stylistic aspects such as whether a comment begins with an imperative verb (e.g., "Fix memory leak") or a descriptive phrase (e.g., "Fixes memory leak").

These linguistic indicators allow a layered analysis of comment structure and commenting practices across projects and time.

These four main feature families provide a structured representation of comments, in their content and purpose, and are a solid foundation for this empirical analysis.

2.4 Related Work

Source code comments have been studied in the software engineering literature, though they occupy a narrower slice of program analysis research than topics like static analysis or refactoring.

Empirical studies such as this one typically fall within the field of Mining Software Repositories (MSR), which applies data analysis techniques to large collections of software artifacts in order to study development practices.

Arafat and Riehle (2009) [1] studied a specific comment metric: density. Their analysis included a large dataset containing thousands of open-source projects, spanning across various programming languages. Their findings suggest that commenting is an ongoing and integrated part of open source development, not a one-off documentation phase or afterthought. They also found that density is very weakly correlated with project size in lines of code, so larger codebases are not less commented on average.

Another study on comment density by Hao He (2019) [14] found that project density varied greatly within his dataset. These differences were attributed to factors such as the programming languages used, the project's purpose, and the size of the development team.

When it comes to comment features, studies are much more sporadic.

For annotation markers, some studies have focused on understanding their usefulness and quality. Specifically, Wang et al. (2024) [15] analyze the characteristics of effective `TODO`

comments and identify factors that distinguish useful `TODO` annotations from low-quality ones. Among their results, it is worth mentioning that they conclude that many `TODO` comments persist long-term, linking to technical debt.

The linguistic features of comments have also been analyzed. Notably, Rabbi et al. (2020) [16] analyzed inconsistencies between code and its respective comments, showing that mismatches occur frequently and may negatively affect code comprehension and maintenance.

These works, while important and overlapping in scope with this study, do not provide a direct answer to the four research questions presented in Chapter 1.

Most treat individual projects or aggregated datasets as their unit of analysis, with little attention to how commenting behavior has shifted across different eras of software development.

This study compares commenting practices in Python open-source repositories across two periods: the 2000s and the 2020s. Using comment density, block structure, annotation markers, tooling directives, and linguistic features as measures, it asks whether and how developers write comments differently in the modern day than they did in the 2000s.

Chapter 3

Repository Selection and Dataset

This chapter focuses on clarifying the methodology for project selection, snapshot retrieval and the refinements necessary to produce a valid dataset to then analyze. Since the study analyzes repositories from a variety of time periods, it is important to clearly document how the projects were selected and how they align with the study goals.

Providing this information ensures that the resulting dataset forms a coherent and reproducible basis for this empirical analysis.

3.1 Selection Criteria

The dataset used in this study consists of sixteen Python open-source projects. Eight of these projects were originally created and developed during the 2000s, while the remaining eight were created in the early 2020s and are currently well-maintained and widely used. For each project, a specific repository snapshot was selected as the unit of analysis. The snapshots for the projects originating in the 2000s correspond to versions from approximately 2009, whereas the snapshots for the 2020s projects correspond to versions from around 2025. An equal number of projects from each period were selected in order to enable a balanced comparative analysis of commenting practices.

To make sure that the chosen projects are comparable, several aspects have been taken into consideration in the selection.

Project Domain

Firstly, the purpose of all the candidate projects has to be related. If two projects have broadly different purposes, their comments would have a different purpose as well, and might display semantic and syntactic differences that transcend the period in which their source code is written.

Comments of a scientific library such as *NumPy* might focus on mathematical concepts or explain formulas that cannot be easily inferred from the code alone. In contrast, comments of an infrastructure tool such as *pip* might tend to explain workflow, edge cases and system behavior.

Comment density would also be impacted. Scientific libraries might have a higher comment frequency around core algorithms, while package managers might have a higher comment frequency in edge cases, such as version mismatches. To support this, a study from 2019 by Hao He [14] found that the purpose of a project has an effect on its comment density, showing that, across his dataset, projects with educational purposes have the highest comment density, while projects which are ready-to-use applications have the lowest.

For this reason, a fixed domain for all the candidate projects has been established. To be considered, all candidate projects were required to belong to the terminal ecosystem. In practice, this means that the selected repositories implement command-line applications, packaging/build tools, development utilities, or related libraries used primarily in terminal workflows.

This ensures that the comments in their source code are more likely to be similar in both their purpose and their expected density.

Project Size

Project size was another factor considered during the project selection process. Instead of focusing on sixteen projects of homogeneous size, the approach that was taken in this study was to include projects of different sizes, as long as they align with the other selected projects and do not remain isolated outliers. Arafat and Riehle (2009) [1] showed that comment density is very weakly correlated with project size in lines of code, and thus selecting projects of various sizes will not excessively alter densities.

At the same time, projects that were either extremely small or too large have not been considered, to avoid obscuring comparability within the selected repositories.

It is important to note that the combined lines of code for the eight projects in each of the two eras are approximately balanced.

Tables 3.1 and 3.2 summarize project size by Python files and Python lines of code.

Project Maturity and Snapshot Policy

In the snapshot selection process for the sixteen repositories, another important factor was ensuring that all the projects were represented at a comparable stage in their development lifecycle.

In growing projects, comments often focus on the current intent of the code and tend to be concise and closely tied to ongoing work. Such comments may refer to current issues in the codebase, and they may evolve quickly, as the codebase changes.

In contrast, comments in legacy projects might become outdated or misleading. They may refer to systems or bugs that no longer exist in the current implementation.

Empirical studies support these observations, showing that as systems evolve, developers frequently fail to update comments when modifying code, leading to outdated and misleading comments in codebases that have been active for a long time [17].

Due to these effects, the sixteen repositories have been analyzed at a comparable stage of

their development. For each project, a snapshot was chosen approximately five years after its initial creation. In practice, this ensures that each project had already grown enough to be considered a mature tool, but not yet at a stage where evolution and maintenance had come to a stop.

For the eight repositories belonging to the 2000s group, snapshots were selected around the year 2009. For the repositories representing the 2020s group, snapshots were selected around 2025. A specific release closest to the target year was chosen as the reference snapshot for the analysis.

The repositories selected according to the criteria discussed in the previous paragraphs form the dataset that will be used across this study. The following section presents the selected repositories together with basic metadata.

3.2 Repository Inventory

Tables 3.1 and 3.2 list the sixteen repositories included in the dataset. For each project, the selected release snapshot, release date, and several structural metrics, are reported. These metrics include the number of Python source files, the number of Python test files, and the corresponding LOC[1]. Python source files (referred to as *core files* in this study) are Python files that are not classified as test files, vendored dependencies, or documentation-related code.

Table 3.1: Repositories from the 2000s group

Project	Release	Release Date	Core Files	Test Files	Core LOC	Test LOC
virtualenv	v1.4	2009-11-07	8	0	3,027	0
coverage	v3.0	2009-06-14	17	0	2,281	0
pip	v1.0.1	2011-04-30	33	25	8,140	3,082
setuptools	v0.6c11	2009-10-20	34	6	10,643	3,626
nose	v0.11.1	2009-05-13	53	187	12,778	7,482
pygments	v1.0	2008-11-23	69	13	22,005	972
ipython	v0.10.1	2010-10-12	202	71	56,696	8,536
scons	v1.0.0	2008-08-12	265	879	81,675	96,940
Total			**681**	**1,181**	**197,245**	**120,638**

The dataset contains repositories of varying sizes, ranging from smaller utilities with only a few thousand lines of core code to larger tools exceeding tens of thousands of lines of core code. Despite this variation, the overall scale of the projects remains broadly comparable across the two eras.

It is important to note that the amounts of files and LOC shown in the tables might slightly vary compared to the numbers that have been used in the study. This is because files with encoding issues have been discarded automatically.

Figures 3.1 and 3.2 visualize the distribution of Python files and lines of code across the repositories. It is important to note that only the core and test files will be analyzed

[1]Lines of Code (LOC).

Table 3.2: Repositories from the 2020s group

Project	Release	Release Date	Core Files	Test Files	Core LOC	Test LOC
ppe[2]	v1.10.0	2026-01-18	5	14	939	5,278
build	v1.4.0	2026-01-08	13	19	1,752	2,398
typer	v0.19.0	2025-09-20	20	326	5,401	13,104
pip-audit	v2.9.0	2025-04-07	26	27	3,637	3,436
pytask	v0.5.8	2025-12-30	65	0	12,585	0
rich	v14.0.0	2025-03-30	123	66	28,808	10,637
pdm	v2.26.0	2025-10-11	130	91	22,061	11,237
textual[3]	v1.0.0	2024-12-12	247	396	75,275	42,276
Total			**629**	**939**	**150,458**	**88,366**

in this study (shown in the figures in blue and orange respectively), as vendored and documentation-related code are not part of the primary software implementation.

3.3 Snapshot Retrieval via Software Heritage

To ensure long-term reproducibility of the dataset, all repository snapshots analyzed in this study were retrieved from the Software Heritage archive. Software Heritage is a large-scale initiative whose goal is to collect, preserve, and share all software that is publicly available in source code form [18]. By archiving source code and releases from platforms such as GitHub, GitLab, and other repositories, Software Heritage provides a persistent infrastructure for software preservation and research.

Each archived object in Software Heritage is identified using a *SoftWare Hash IDentifier* (*SWHID*) [19]. SWHIDs provide a cryptographically derived and content-addressed reference to a specific artifact in the archive, such as a revision, directory, or file. Because SWHIDs are derived from the content itself, they provide a stable and unambiguous reference to a specific version of the source code.

Using SWHIDs ensures that the exact repository snapshots analyzed in this study can always be retrieved in the future. This is particularly important for empirical software engineering research, where the ability to reproduce results depends on access to the precise version of the source code used during analysis.

A table with the *SWHID* used for each repository is presented in Appendix A.

For each repository included in the dataset, the snapshot corresponding to the selected release directory was identified and resolved through the Software Heritage API. The resulting snapshot directory was then retrieved using the Software Heritage Vault API [20], which provides downloadable archives of stored repository states. The retrieved snapshots form the input dataset for the comment extraction pipeline described in Chapter 4.

In the case of *virtualenv*, preprocessing steps were required to reconstruct the full source tree from the retrieved snapshot. In fact, the chosen release contained embedded files

[2] poetry-plugin-export

[3] Also referred to in this study as *textualize*

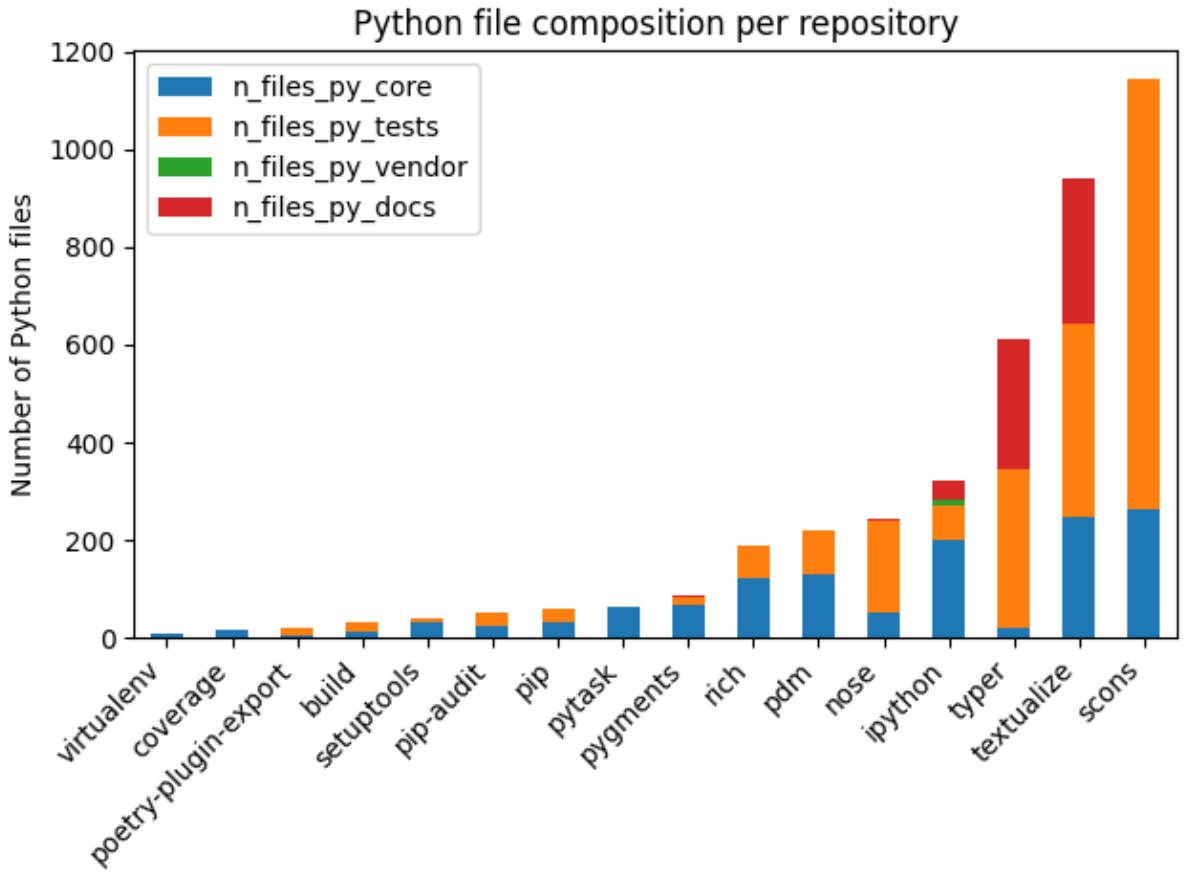

Figure 3.1: Number of Python files per repository in the dataset. The bars show the total number of Python source, test, vendor and documentation files with color coding for each project.

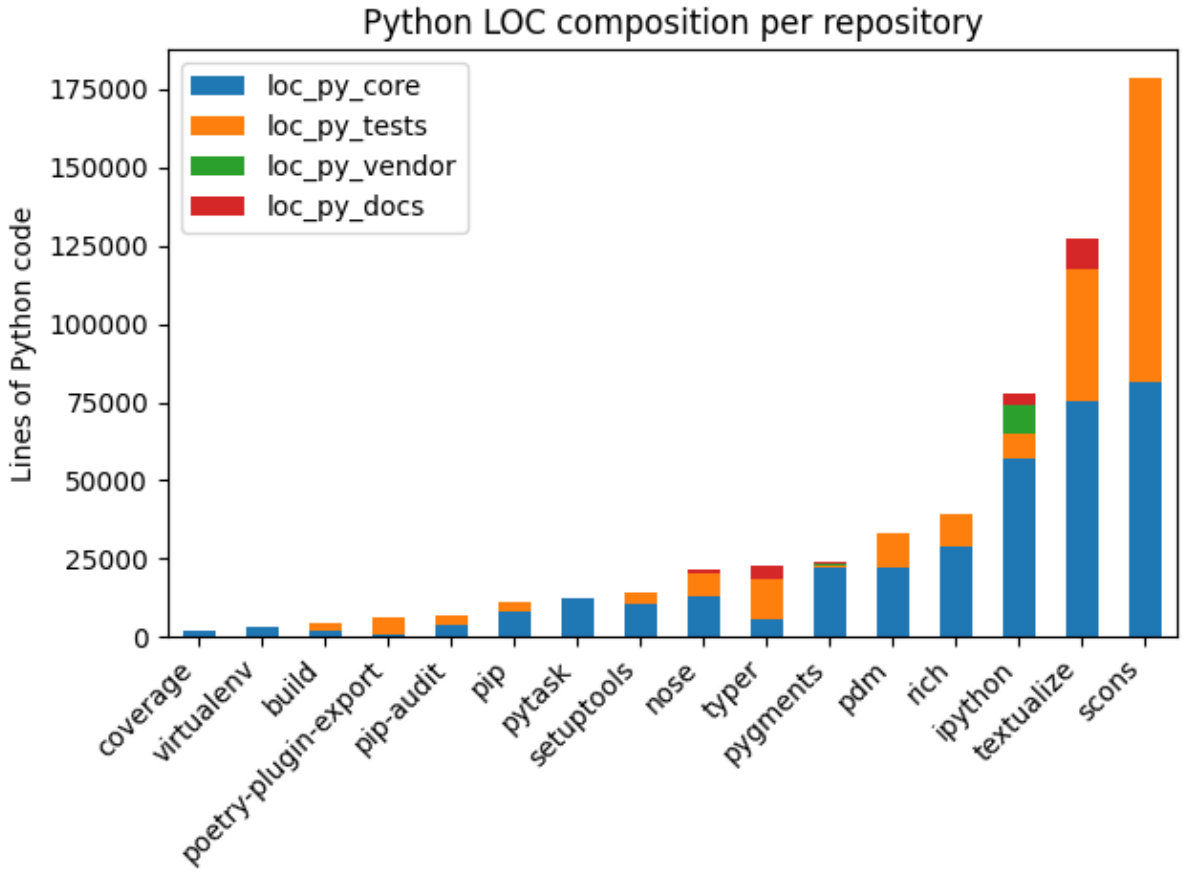

Figure 3.2: Number of Python lines of code per repository in the dataset. The bars show the total number of Python source, test, vendor and documentation files with color coding for each project.

that have been stored as compressed base64 payloads inside the main Python pipeline file. These payloads were unpacked at runtime by the project itself. In order to obtain the complete file structure for analysis, a small extraction script was implemented to decode and materialize these embedded files within the raw *virtualenv* directory.

3.4 File Exclusions and Subset Definitions

Not all files contained in a repository are relevant for analyzing commenting practices. Open-source repositories often include third-party dependencies, generated files, virtual environments, or documentation-related code that does not reflect the commenting behavior of the project's developers.

To ensure that the analysis focuses on the original source code of each project, a number of directories and files were excluded during dataset preparation. These exclusions remove content that is either automatically generated or imported from external sources. Examples include version-control metadata directories (such as `.git`), Python cache directories (such as `__pycache__`), virtual environment directories (such as `venv` and `.venv`), and dependency folders such as `site-packages` or `node_modules`.

In addition to the exclusions, a small number of files created during the snapshot retrieval process were also removed from the analysis. For example, the file `_MANIFEST.json`, generated during dataset collection, is excluded because it does not belong to the original repository snapshot.

After applying these exclusions, the remaining Python files were classified according to their role in the repository. The classification is based on common naming conventions used in Python projects. Directories containing terms such as `test`, `tests`, or `unit_tests` are classified as test code. Directories containing terms such as `docs`, `doc`, or `documentation` are classified as documentation. Similarly, directories such as `vendor`, `third_party`, or `external` are classified as third-party code.

Based on this classification, three subsets of Python files are defined for the analysis:

- **core**: Python source files belonging to the main implementation of the project, excluding tests, documentation, and third-party code.
- **tests**: Python files that belong exclusively to the project's test suite.
- **core plus tests**: the union of the core source files and all test files contained in the repository.

The primary analysis in this thesis focuses on the *core* subset, as it most accurately reflects the commenting practices used within the implementation of the software itself. The *tests* subset is used in additional comparisons to explore whether commenting behavior differs between production code and test code.

In summary, this chapter described the construction of the dataset used for the empirical analysis. Sixteen Python repositories were selected according to clearly defined criteria regarding project domain, size and maturity. Repository snapshots were retrieved through the Software Heritage archive, ensuring that the analyzed versions of the code can be uniquely identified and accessed in the future. Finally, the retrieved snapshots were refined through directory exclusions and path-based classification in order to isolate the relevant subsets of Python source and test files used in the analysis.

The resulting dataset provides the starting point for the automated comment extraction and feature analysis pipeline as described in the following chapter.

Chapter 4

Pipeline Design and Implementation

This chapter describes the design and implementation of the analysis pipeline used to extract and analyze comments from the repository snapshots that have been presented in Chapter 3. The goal of the pipeline is to process Python source files and transform them into structured data suitable for large-scale comment analysis.

The pipeline touches several stages of processing. First, the retrieved repository snapshots are fetched and their file structure is analyzed. Next, comments and docstrings are extracted from Python source files. Comments are then grouped into comment blocks. These blocks are then analyzed to compute a set of features describing their content, purpose, and linguistic characteristics. Various blocks datasets are produced, one for each role subset, as described in Section 3.4.

The following sections describe the individual stages of this pipeline in detail.

4.1 Pipeline Overview

The comment analysis is performed using a purpose-built processing pipeline that operates on the repository snapshots described in the previous chapter. The pipeline transforms raw Python source files into a structured dataset of comment blocks enriched with various features, together with a dataset of docstrings classified by type, as described in the following sections.

Figure 4.1 illustrates the overall architecture of the pipeline and the flow of data between its stages.

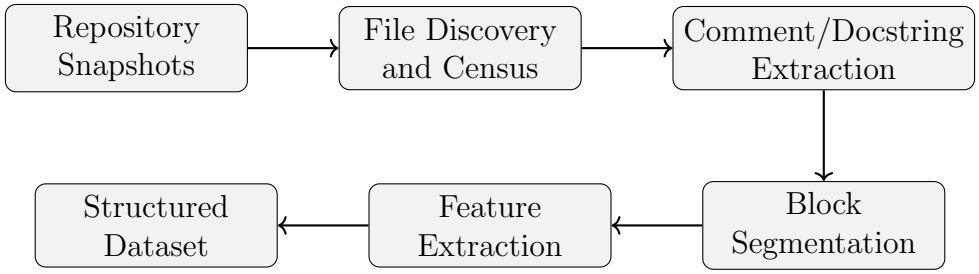

Figure 4.1: Overview of the comment analysis pipeline used in this study.

After fetching all the appropriate raw snapshot code for the projects, a small census was run to ensure that all the repositories exhibited the expected structure to detect potential anomalies or outliers. This step also produced several structural statistics about the repositories, including the distribution of Python files and lines of code across projects. Figures 3.1 and 3.2 are direct outcomes of the census.

The preliminary step was to build a dataset that would act as a universal single source of truth. The dataset consists of a list of every single file of the repositories in analysis, with additional important metadata. In the following steps, this file was used as the primary source of truth.

4.2 Comment Extraction

The first processing stage of the analysis pipeline extracts comments from Python source files contained in the repository snapshots. The goal of this step is to identify all occurrences of comments in the codebase while preserving their position and textual content, so that they can later be grouped into comment blocks and be analyzed further.

The primary mechanism used for comment extraction is Python's `tokenize` module from the standard library. This module performs lexical analysis of Python source code and produces a stream of tokens representing the syntactic elements of the program [21]. Among these tokens is the `COMMENT` token type, which corresponds directly to comments written in the source code.

Using the tokenizer provides comment identification according to Python's lexical rules. This prevents false positives that could arise from brute-force pattern matching approaches, for example, occurrences of the `#` character inside string literals. Each extracted comment is recorded together with its file path, line number, column position, and the full textual content of the source line in which it appears.

During extraction, comments are also classified as either *inline comments* or *full-line comments*. This distinction is made at this early stage of the pipeline so that the transition into comment blocks can be more straightforward. Such classification was determined by examining whether non-whitespace characters occur before the comment token on the same line of code. If the comment appears after executable code on the same line, it is classified as an inline comment, otherwise it is treated as a full-line comment, as discussed in Section 2.1.

In addition to the raw comment token, normalized textual variants of the comment are stored, together with basic lexical statistics, such as length in characters and words. These attributes form the basis for the next stages of block segmentation and feature extraction.

As a result, a second output dataset is created with a list of all the comments found, including their classification as inline or full-line, as well as all other attributes discussed.

4.3 Docstring Extraction

In parallel to the comment extraction pipeline described above, docstrings have also been extracted from the source code.

Python's tokenize module is not well suited for reliably extracting docstrings because it operates purely at the lexical token level, rather than at the syntactic-structure level of the program [21]. The pipeline therefore uses Python's `ast` module, which parses the source file into an abstract syntax tree. A custom AST visitor traverses the tree and identifies docstrings according to Python's official semantics, namely that the first statement in the body of a module, class, or function is a string literal expression [22]. With this approach, only genuine docstrings are extracted, rather than string literals that could appear anywhere in the code.

However, the AST parser is not able to process all the files: it will not work if it encounters syntax errors in them. For instance, the source code that needs to be parsed might be written in older versions of Python (such as Python 2), that at times use invalid syntax for the current Python version. This is especially the case when parsing files from old repositories that are part of the 2000s group. In such cases, the pipeline falls back to the `parso` parsing library. Parso provides a more tolerant parser that can recover structure from incomplete or syntactically inconsistent code [23]. The fallback extraction reproduces the same docstring detection rule by identifying string literals that appear as the first statement in a module, class, or function body.

Although docstrings are extracted during this stage, they are treated separately from regular comments in the subsequent analysis, since they represent structured documentation rather than inline annotations within the code.

The result of this stage is a dataset, recording each docstring's content, its scope (module, class, function) and its metadata, such as its position, length and content.

4.4 Block Segmentation

After individual comments have been extracted, the next stage of the pipeline groups them into *comment blocks*. A comment block is the basic unit of analysis used throughout this thesis.

The segmentation process is performed separately for each file. The input consists of the comment-level records produced during the extraction stage, ordered by file, line number, and column position. The pipeline will then construct block-level entries according to deterministic rules.

We define three distinct subtypes of comment blocks:

1. **Inline block:** A block consisting of a single inline comment. Since it occurs on the same physical line as executable code, it is interpreted as a standalone annotation attached to that specific line and is therefore not merged with neighboring comments.
2. **Full-line singleton block:** A block consisting of a full-line comment spanning a single line.

3. **Full-line block:** A block consisting of more than one full-line comment. To be part of the same block, multiple comments have to be immediately adjacent by line number and begin at the same indentation level.

As a result, isolated full-line comments become singleton blocks, whereas sequences of adjacent full-line comments at equal indentation are grouped into a single block.

The use of indentation as a segmentation criterion is important because it likely reflects the different purpose of adjacent comments. Two consecutive comment lines that appear at different indentation levels are likely to belong to different logical contexts and are therefore not merged.

A special case is the *shebang* line. If a full-line comment at the first line of a file begins with the prefix `#!`, it is always treated as a singleton block, even if other comment lines appear immediately afterwards.

For each resulting block, the pipeline records its start and end line, indentation level, number of lines, member comments, and concatenated textual content. This structured representation allows the subsequent feature extraction stage to operate on separated and classified comment units.

4.5 Feature Extraction

After comment blocks have been constructed, the next stage of the pipeline enriches each block with a set of descriptive features. These features capture structural, functional, and linguistic properties of the comment text and form the basis for some results presented in Chapter 5.

Feature extraction is performed at the block level. Each comment block is treated as a single unit, and the block text is analyzed to detect the presence of specific signals. The pipeline applies a sequence of feature extraction functions to the block dataset, with each function computing a family of related features and adding the resulting columns to the dataset. Features are identified in blocks through a series of regular expressions (also referred to as *regexes*). The Python library *re* was used.

The feature extraction stage focuses on four main category families: license header indicators, annotation markers, tooling directives, and linguistic features. These categories correspond to the comment types discussed in Chapter 2.

License Header Features

License header features detect blocks that contain legal or licensing information. Such blocks are commonly located at the beginning of source files and may include copyright notices, references to specific software licenses, or legal disclaimers.

The pipeline identifies a number of signals associated with license headers, including the presence of SPDX identifiers, copyright statements, license-related keywords, references to specific license families (such as MIT, Apache, or GPL), and warranty disclaimers. Each signal is detected using regular expression patterns applied to the comment block text.

For each block, boolean feature flags indicate the presence of individual legal signals. Additional aggregated features record whether any legal signal is present and how many distinct signals occur in the block.

A representative example of identification through a regex is as follows:

```
import re

_COPYRIGHT_RE =
    re.compile(r"\bcopyright\b|\(c\)|{©}", re.IGNORECASE)
```

This snippet of code is in charge of finding the strings "`copyright`", "`(c)`" or the copyright symbol in blocks (case insensitive). If one or more of these symbols is found, the block will then be classified as having a copyright identifier and thus as a legal header block.

Annotation Marker Features

Annotation marker features capture task-related comments used by developers during the development process. These comments typically contain standardized keywords.

The pipeline detects a set of common annotation markers, including `TODO`, `FIXME`, `XXX`, `HACK`, `BUG`, and `NOTE`. These markers are identified using case-insensitive regexes applied to the comment text.

For each block, individual boolean indicators record the presence of each marker type. Additional summary features capture whether any annotation marker is present in the block, the number of distinct markers detected, and the set of marker types appearing in the text.

Tooling Directive Features

Tooling directive features include comments used to communicate with automated development tools such as linters, type checkers, or code formatters.

Examples of such directives include `noqa` comments used to suppress linter warnings, `type: ignore` directives used by static type checkers, and tool-specific comments for systems such as `pylint` or `mypy`. The pipeline also detects encoding declarations and other formatting directives that follow established comment conventions.

As with the other feature families, each directive type is detected through regexes. The resulting features indicate which directives appear in the block, whether any directive is present, and how many distinct directive types occur.

Linguistic Features

The final feature family captures linguistic characteristics of comment text. These features are intended to provide insight into stylistic aspects of comments and the way developers communicate information within the source code.

Among the extracted block features are:

- **punctuation patterns**, as for example whether a comment ends with a question mark, an exclamation point or a period,
- **all-uppercase comments**, consisting of at least two capitalized letters and no lowercase ones,
- **comments with a non-capitalized start**,
- **empty comments**, consisting of only the `#` symbol,
- **separators** (non-empty comments having no alphanumeric content),
- **comments containing symbol-based emoticons**,
- **comments starting with an imperative verb** (e.g., *handle*),
- **comments starting with a descriptive verb** (e.g., *handles*)

Together, these linguistic signals provide a lightweight characterization of comment style that complements the more functional feature families described above.

4.6 Output Format

Once block segmentation and feature extraction are completed, the data is written as columnar *Parquet* files. Each row maps to a single comment block and carries both its structural metadata and the computed feature values.

The schema of the dataset can be divided into several conceptual groups of columns, as follows.

Repository and File Metadata

The first group of columns describes the repository context in which a comment block appears. These columns allow each block to be uniquely associated with a specific repository, release, and file.

- `subset`
- `repo`
- `group`
- `release`
- `release_date`
- `directory_id`
- `snapshot_root`

- path_rel
- file_id

Together, these attributes identify which repository snapshot each comment block came from.

Block Identifiers and Positional Metadata

The second group of columns describes the structure and position of the comment block within the source file.

- block_id
- block_kind
- start_lineno
- end_lineno
- indent_col
- n_lines
- is_singleton

The column block_id is a stable identifier derived from repository metadata and positional information.

Block Text Representation

The dataset also stores the textual content of the block and the original comment lines from which it was constructed.

- member_linenos
- member_cols
- member_texts
- member_raw_tokens
- member_line_texts
- block_text_raw
- block_text_stripped
- block_char_len
- block_word_len

These fields preserve both the raw textual representation of the comments and normalized forms used for feature extraction.

Feature Columns

The remaining columns hold the extracted features. Most are boolean flags marking whether a given signal appears in the block, with a few additional columns recording counts and lists of detected feature types.

The feature columns are organized according to the four feature families described in Section 4.5:

- **License header features** (`lh_*`)
- **Annotation marker features** (`am_*`)
- **Tooling directive features** (`td_*`)
- **Linguistic features** (`lf_*`)

An additional column, `sb_is_shebang`, records whether a block corresponds to a shebang directive. For the analysis, shebang blocks are grouped with the annotation markers.

Table 4.1 shows a simplified example of a comment block entry from the dataset.

Table 4.1: Example comment block record (only select columns shown)

Column	Example Value
repo	ipython
release	v0.10.1
path_rel	.../IPython/Extensions/ipy_completers.py
block_id	93cd761b7c75dd450cd86fad93387016d835da2e
block_kind	full_line_singleton
start_lineno	247
end_lineno	247
block_text_stripped	"the rest are probably file names"
block_char_len	32
am_has_todo	False
td_has_noqa	False
lf_is_all_caps	False

Choice of Storage Format

The dataset is stored using the Apache Parquet format. As stated in its documentation, the format provides high-performance compression and encoding schemes to handle complex data in bulk and is supported in many programming languages and analytics tools [24].

The reason for choosing the Parquet format was that its columnar layout allows analytical tools to read only the columns required for a specific analysis, and by doing so improving performance. It also integrates well with data analysis tools commonly used in the Python ecosystem, including the `pandas` library used in this study for handling the datasets.

4.7 Reproducibility

Reproducibility shaped how the analysis pipeline was built from the start. The dataset and results can be regenerated reliably because specific steps were taken at each stage to make that possible.

First, all repository snapshots were retrieved from the Software Heritage archive using *SoftWare Hash IDentifiers* (SWHIDs). Because these identifiers are content-based, they permanently and immutably point to the exact version of source code used in the analysis.

Furthermore, the data processing workflow was built as a deterministic pipeline of version-controlled scripts. Each stage outputs explicit intermediate datasets, stored as Parquet files, so individual stages can be rerun or inspected without reprocessing everything from scratch.

Finally, the pipeline uses configuration files and explicit feature definitions to control dataset preparation, comment segmentation, and feature extraction. Executing it on the same repository snapshots with the provided configuration and scripts reproduces the full analysis.

The project repository can be viewed and downloaded from
`https://github.com/mikegaravani/py-comment-evolution`.

Chapter 5

Results and Analysis

This chapter presents the empirical results and answers the research questions from Chapter 1. Using the dataset and pipeline described in Chapters 3 and 4, it compares commenting practices in Python repositories from the 2000s and the 2020s across several dimensions.

The chapter begins with comment block density and structural properties, then moves to functional categories (license headers, annotation markers, and tooling directives) and closes with an analysis of linguistic features, docstrings, and a mention of test code commenting patterns.

5.1 Comment Block Density

To address Research Question 1, which investigates how the density of comments differs between Python projects from the 2000s and those from the 2020s, comment block density was examined across the repositories in the dataset. This metric offers an approximate indication of how often developers comment relative to overall codebase size.

Table 5.1 shows each repository from the dataset together with two key indicators of block density: blocks per file (calculated as total number of blocks over total number of files in the repository) and blocks per 1000 lines of code ($KLOC$).

Figure 5.1 shows the distribution of comment block density across the two eras (blocks per KLOC). Repositories from the 2000s tend to have higher densities, with most projects falling between roughly 35 and 50 blocks per KLOC. Those from the 2020s cluster lower, generally between 20 and 35 blocks per KLOC.

Mean, median and aggregate density decreases noticeably between the two eras. Individual projects vary, but the overall trend points toward modern Python codebases containing fewer comment blocks relative to their size.

Figure 5.2 breaks this down by repository. The gap between eras holds across multiple projects. A few 2020s repositories, notably *rich* and *textualize*, sit at particularly low densities, while the 2000s repositories largely cluster toward the higher end. There is one outlier: *typer* (composed of only twenty core files and thus representing a small analyzed codebase) presents an unusually steep amount of blocks, of about 70 per 1000 LOC, which is significantly higher than that of all the other repositories in the 2020s era.

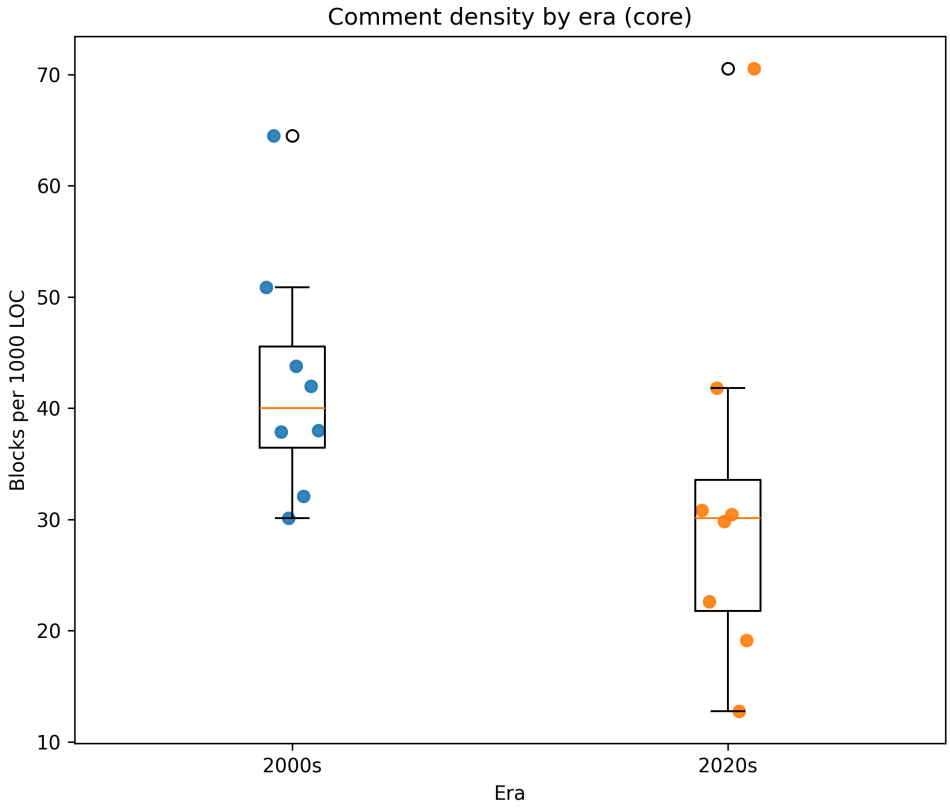

Figure 5.1: Box plot showing the block density for the two eras. One dot equals one repository.

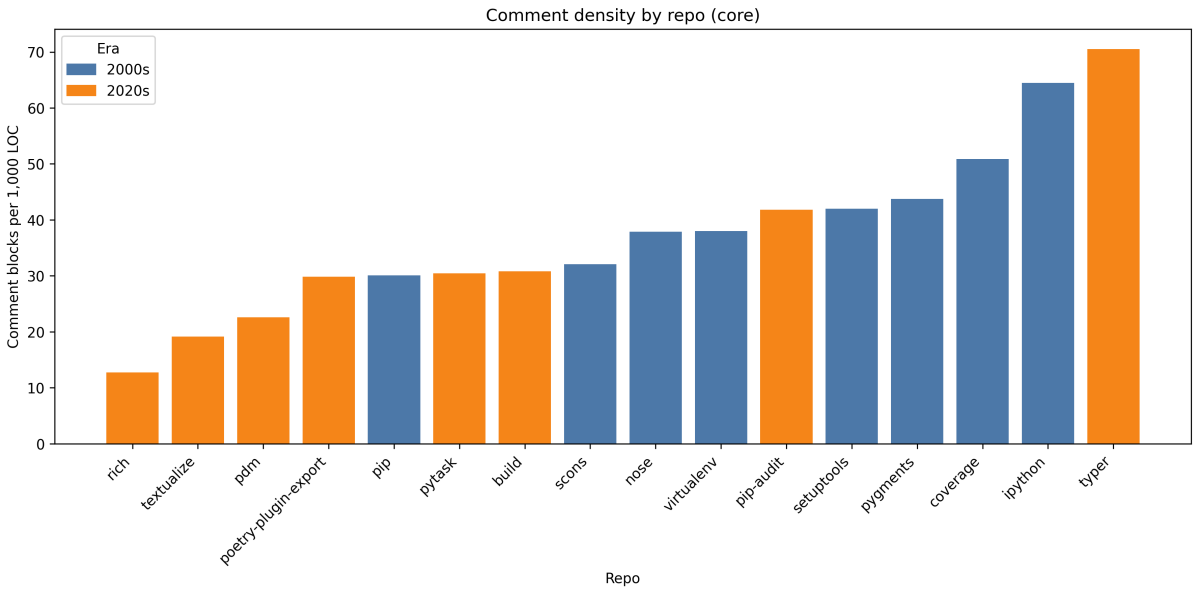

Figure 5.2: Bar chart showing each repository in ascending order of blocks/KLOC, color coded by era.

Table 5.1: Block densities by repository

Repo	Era	Blocks/File	Blocks/1000 LOC
pip	2000s	7.42	30.10
scons	2000s	9.89	32.08
nose	2000s	9.13	37.88
virtualenv	2000s	16.43	37.99
setuptools	2000s	13.15	42.00
pygments	2000s	13.96	43.76
coverage	2000s	6.82	50.85
ipython	2000s	18.75	64.48
rich	2020s	3.01	12.74
textualize	2020s	5.93	19.13
pdm	2020s	4.09	22.62
poetry-plugin-export	2020s	5.60	29.82
build	2020s	4.50	30.82
pytask	2020s	5.89	30.43
pip-audit	2020s	5.85	41.79
typer	2020s	19.05	70.54

These results therefore provide evidence that comment block density has generally decreased in modern Python repositories, answering Research Question 1.

5.2 Block Structure and Length

Beyond overall comment density, it is also important to examine how comments are structured within the source code. As described in Chapter 4, comments were grouped into three structural block types: inline blocks, full-line singleton blocks, and multi-line full-line blocks. In this section, the distribution and length of these block types are analyzed across the two eras, addressing Research Question 2.

It is important to note that for this part of the analysis, blocks classified as license headers were excluded. License headers typically consist of standardized legal text and do not reflect the commenting practices of developers within the implementation of the code itself. They might also significantly inflate full-line block length, as license boilerplate might be added at the top of some files as a comment, spanning several lines.

Table 5.2 summarizes block structure statistics for each repository, including the proportion of inline comments, full-line singleton comments, and multi-line comment blocks, together with average comment lengths.

5.2.1 Block Type Distribution

Figure 5.3 shows the aggregate distribution of comment block types for the two eras. In both periods, full-line singleton comments represent the most common form of comment block, and their ratio has not significantly changed. However, the share of inline

Table 5.2: Block structure and length statistics by repository and era (license header blocks excluded).

Repo	Era	Inline (%)	Singleton (%)	Block (%)	Mean Chars	Mean Words
coverage	2000s	10.4	67.0	22.6	57.7	9.5
ipython	2000s	10.0	57.9	32.1	71.7	9.6
nose	2000s	10.6	56.9	32.5	67.6	10.7
pip	2000s	9.4	70.6	20.0	53.8	9.1
pygments	2000s	23.7	67.8	8.4	29.2	4.7
scons	2000s	8.2	45.9	45.8	115.6	18.5
setuptools	2000s	40.0	47.9	12.1	46.1	7.5
virtualenv	2000s	13.9	58.3	27.8	52.2	8.1
build	2020s	31.4	39.2	29.4	77.8	11.9
pdm	2020s	45.1	43.0	11.9	46.3	7.3
pip-audit	2020s	25.0	35.5	39.5	100.9	16.3
ppe	2020s	21.4	42.9	35.7	82.1	13.5
pytask	2020s	49.3	33.4	17.2	51.3	7.9
rich	2020s	42.8	45.0	12.3	45.0	7.2
textualize	2020s	22.4	56.3	21.3	58.4	10.0
typer	2020s	11.3	74.8	13.9	43.5	6.5

comments increases markedly in the 2020s.

Most repositories from the 2000s exhibit relatively low inline comment ratios, typically between 8% and 14%. Only a small number of projects, such as *setuptools*, exceed this range (visible in Table 5.2).

In contrast, many repositories from the 2020s show substantially higher inline comment proportions. Projects such as *pytask*, *pdm*, and *rich* approach or exceed 40% inline comments. This pattern reinforces the observation that inline comments have become significantly more common in modern Python projects.

At the same time, the share of multi-line full-line blocks decreases substantially. This suggests a shift in commenting practices: earlier projects relied more frequently on multi-line comment sections, whereas more recent projects tend to use more inline blocks. As will be shown, tooling directives (which are mostly found as inline blocks) play a role in this change.

5.2.2 Comment Length

Another structural aspect worth examining is the textual length of comment blocks. Figure 5.4 shows the distribution of average block lengths (in characters) for each block type across the two eras.

Inline comments are the shortest form of comments in both periods, with typical lengths between roughly 18 and 25 characters. Notably, inline comments in the 2020s tend to be slightly shorter on average than those in the 2000s.

Full-line singleton comments are moderately longer, typically ranging between 35 and 50 characters. A slight increase in average length can be observed in the 2020s.

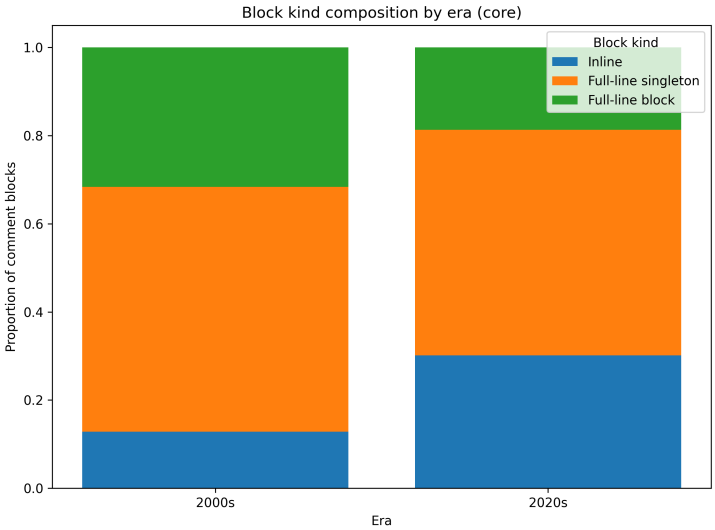

Figure 5.3: Distribution of comment block types by era (license header blocks excluded).

Multi-line full-line blocks are substantially longer than the other comment types, often exceeding 120 characters. The distribution of these blocks also shifts slightly upward in the 2020s, indicating that, in modern development, when longer comment sections are written, they tend to be more verbose.

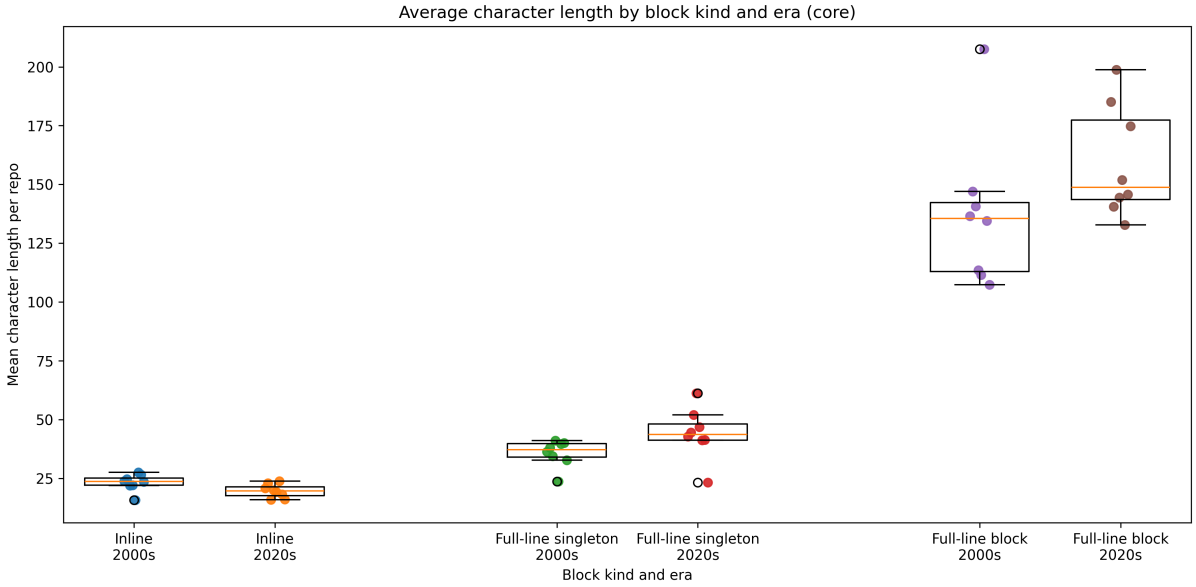

Figure 5.4: Average comment block length by block type and era (license header blocks excluded).

5.2.3 Interpretation

These findings address RQ2 directly. Full-line singleton comments remain the most common block type in both periods. Inline comments appear substantially more often in 2020s code, which hints at shorter comments that are more closely tied to specific lines of

code (such as tooling directives, as will be shown). In contrast, multi-line full-line blocks were comparatively more common in earlier projects.

The block length data point in the same direction: modern codebases use shorter, localized annotations more frequently, and longer explanatory blocks have declined, although they have become slightly more verbose. Taken together, comment structure in Python has moved toward concise, line-specific notation.

5.3 License Headers

License-related blocks typically contain legal information regarding copyright, licensing conditions, and redistribution permissions. Because these comments consist largely of standardized legal text, they do not necessarily reflect the commenting practices of developers.

Table 5.3 reports the number of detected license-related comment blocks for each repository, together with their proportion relative to the total number of comment blocks.

Table 5.3: Legal block counts by repository and era.

Repo	Era	Block Count	Legal Blocks	Legal Blocks (%)
coverage	2000s	116	1	0.9
ipython	2000s	3656	118	3.2
nose	2000s	484	1	0.2
pip	2000s	245	0	0.0
pygments	2000s	963	2	0.2
scons	2000s	2610	244	9.3
setuptools	2000s	447	0	0.0
virtualenv	2000s	115	0	0.0
build	2020s	54	3	5.6
pdm	2020s	499	4	0.8
pip-audit	2020s	152	0	0.0
poetry-plugin-export	2020s	28	0	0.0
pytask	2020s	383	0	0.0
rich	2020s	367	0	0.0
textualize	2020s	1440	1	0.1
typer	2020s	381	0	0.0

The results show that license-related comment blocks are relatively rare in most repositories. In many projects, no such blocks are present at all, while in others they appear only once or a few times. A notable exception is *scons*, where license headers are systematically included in many source files, resulting in a high proportion of legal blocks, accounting for 9.3% of total comments.

It is also worth noting that a small number of detected legal blocks in a repository does not necessarily mean they are false positives. In the case of *coverage*, a single legitimate legal header was found in the entire repository.

However, repositories such as *pdm*, *pygments*, and *textualize* contain blocks mentioning licensing concepts without constituting full license headers. An example is this block text, found in a *pygments* file and classified by the algorithm as a legal header: "`# check for copyright and license fields`".

License comment blocks are mostly a matter of repository policy, not commenting practices. For this reason, this study cannot infer trends in commenting behavior for license headers.

5.4 Annotation Markers

Research Question 3 examines the evolution of annotation markers and tooling directives in code. This section will address the annotation markers part, while the next section will focus on tooling directives.

Annotation markers are comments developers leave to flag things like pending work, known bugs, or implementation details worth remembering. Typical examples include markers such as `TODO`, `FIXME`, `XXX`, `HACK`, `BUG`, and `NOTE`. Unlike regular comments, annotation markers double as lightweight project management signals directly inside the source code.

Figure 5.5 shows how often annotation markers appeared in comment blocks across the two eras. Specifically, it highlights the share of blocks containing a given marker out of all comment blocks in that era.

The marker *ANY* represents the proportion of blocks containing at least one annotation marker. It is important to note that its value might be a little lower than the sum of the individual marker proportions. This is because a block might contain more than one marker.

Overall, annotation markers appear in a relatively small proportion of comment blocks across both eras, even when all marker types are aggregated. This proportion is a little above 4% of all blocks in the 2000s, while it is at about 2.6% in the 2020s.

That said, some notable differences between the two periods emerge. The `TODO` marker shows a clear increase in the 2020s, appearing in roughly three times as many comment blocks proportionally, compared to the 2000s. This suggests that modern projects lean more heavily on `TODO` comments as a lightweight way to track incomplete tasks directly in the source code.

Conversely, markers such as `FIXME` and `XXX` appear much more frequently in the older repositories. In the 2020s sample, `FIXME` does not appear at all, while `XXX` is extremely rare. These markers historically served a similar purpose to `TODO`, often signaling code that requires correction or attention. Their disappearance in modern repositories may reflect a shift toward more standardized annotation conventions.

Markers such as `HACK` and `BUG` remain rare in both periods (although more common in the 2000s), while `NOTE` appears at comparable levels across the two eras.

Taken together, these results suggest a modest decrease in annotation practices. Importantly, the preferred vocabulary has noticeably shifted. Modern projects seem to favor the more neutral `TODO` marker, while older markers such as `FIXME` and `XXX` have become

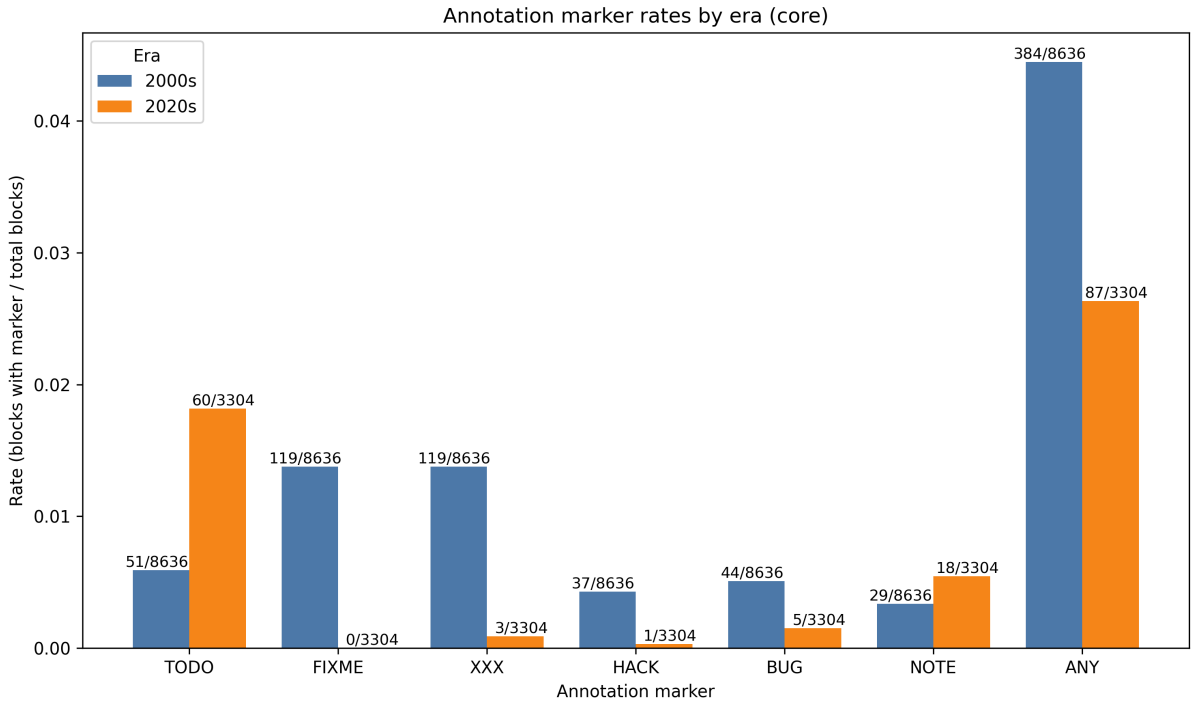

Figure 5.5: Annotation marker rates by era. Values above each bar show the absolute counts used to compute the rates.

much less common, or even disappeared completely.

This concludes the first half of RQ3, defining the change in the prevalence of these signals.

5.5 Tooling Directives

The second part of Research Question 3 concerns tooling directives. These comments are read by automated tools (linters, type checkers, code formatters), not by developers. Unlike annotation markers, which signal intent to human readers, tooling directives are picked up directly by external programs during static analysis or formatting passes.

Common examples include directives such as `noqa`, which suppresses linter warnings, `type: ignore`, which instructs type checkers to ignore specific lines, and comments targeting tools such as `pylint` or `mypy`. Formatting systems may also introduce directives such as `fmt` comments, while encoding declarations specify the character encoding of a source file.

Figure 5.6 shows the rate at which tooling directives appear in comment blocks across the two eras. The values represent the proportion of comment blocks containing a given directive relative to the total number of comment blocks in that era.

The marker *ANY* represents the proportion of blocks containing at least one tooling directive. As with the annotation markers, this value could be slightly lower than the sum of individual directive rates because a single block might contain more than one directive.

The difference between the two eras is striking. In the repositories from the 2000s, tooling

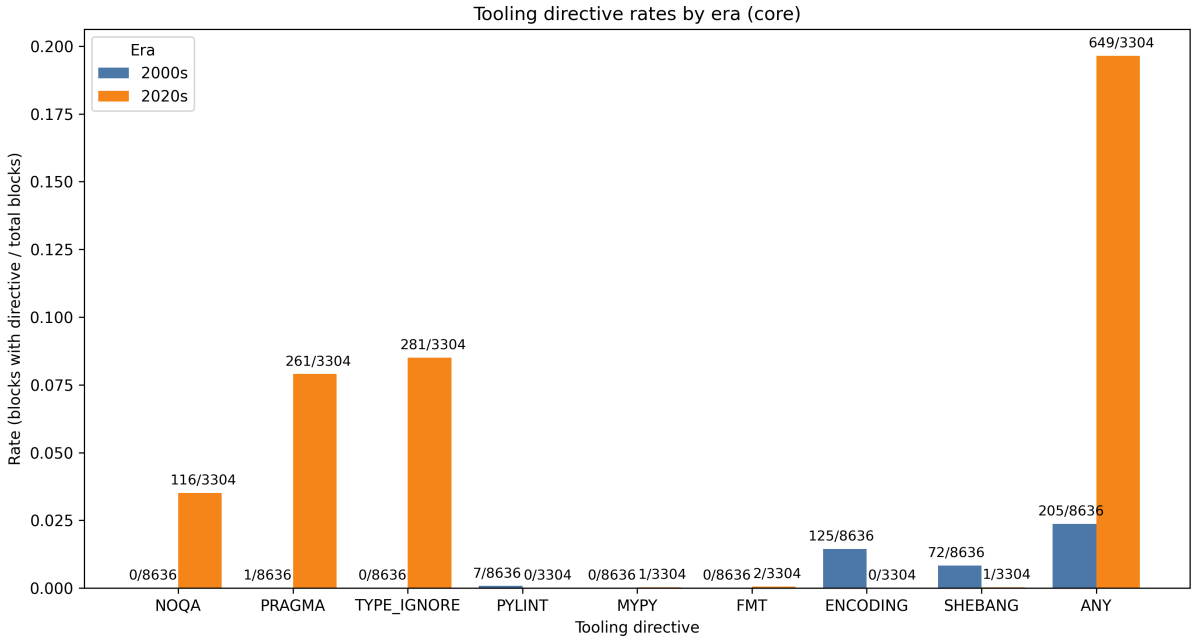

Figure 5.6: Tooling directive rates by era. Values above each bar show the absolute counts used to compute the rates.

directives are almost entirely absent. Only a small number of comments related to tools such as `pylint`, shebangs, or encoding declarations appear, resulting in a total rate of roughly 1.5% of comment blocks.

It is noteworthy that the one instance of a *pragma* block in the 2000s era is not a false positive.

Tooling directives are far more common in 2020s repositories. Close to 20% of all comment blocks contain at least one directive, compared to under 2% in the earlier period.

Several specific directive types contribute to this increase. The most common are `type: ignore` and `pragma`, which appear frequently in the modern repositories. These comments typically interact with static analysis tools such as type checkers and linters. The prevalence of `noqa` comments reflects how standard automated linting has become common practice in modern Python workflows.

Encoding declarations appear exclusively in the 2000s repositories. Python 2 sometimes required explicit encoding comments in source files, but the practice was made redundant when Python 3 adopted UTF-8 as its default encoding [25]. Similarly, shebang lines appear predominantly in the 2000s repositories. While historically used to specify the interpreter for executable scripts, their usage has declined in more recent repositories.

These results indicate a clear shift in how comments function within Python codebases. Earlier projects used comments primarily to explain code to human readers. In contrast, modern repositories use them increasingly to communicate with automated tools. This change reflects the expanding role of static analysis, code formatting, and automated quality checks in contemporary software development practice.

5.6 Linguistic Features

The last Research Question (RQ4) revolves around understanding linguistic and stylistic differences between eras.

For this part of the study, the focus shifts exclusively on narrative blocks. This means that comment blocks from the categories discussed in the previous sections are excluded from the block dataset. In practice, only comments identified as **not** being **legal headers**, **annotation markers** or **tooling directives** (including shebang comments) have been analyzed.

These exclusions ensure that the rates of blocks with specific linguistic and stylistic properties are not altered by these types of comments.

As previously stated, this part of the analysis focuses on several stylistic signals, including sentence structure, punctuation usage, and grammatical patterns at the beginning of comments. These features characterize how developers formulate explanations within source code.

5.6.1 Imperative and Descriptive Comment Starts

One stylistic choice examined is the grammatical form used at the beginning of comments. In particular, comments were classified according to whether they begin with an *imperative* verb form or a *descriptive* verb form (eventually preceded by a pronoun). Here are examples for both patterns:

- **Imperative form:** "*handle errors*", "*return the result*"
- **Descriptive form:** "*handles errors*", "*returns the result*", "*this handles errors*", "*it returns the result*"

Figure 5.7 compares the proportion of comment blocks beginning with each grammatical structure across the two eras.

Imperative-style comments appear frequently in both eras, but their usage increases in the modern repositories. Approximately 10.8% of comment blocks in the 2000s begin with an imperative verb, compared to about 14.0% in the 2020s, marking an approximate increase of 30% in the popularity of imperative comments.

In contrast, descriptive comment starts decrease over time. In the earlier repositories, roughly 4.2% of comments begin with descriptive verb forms, but this proportion drops to approximately 2.4% in the 2020s, marking about a 43% decrease in the popularity of descriptive comments.

This shift suggests a gradual stylistic preference for imperative phrasing when writing explanatory comments. This phrasing mirrors conventions found in programming manuals and API documentation, where actions are typically described in directive form.

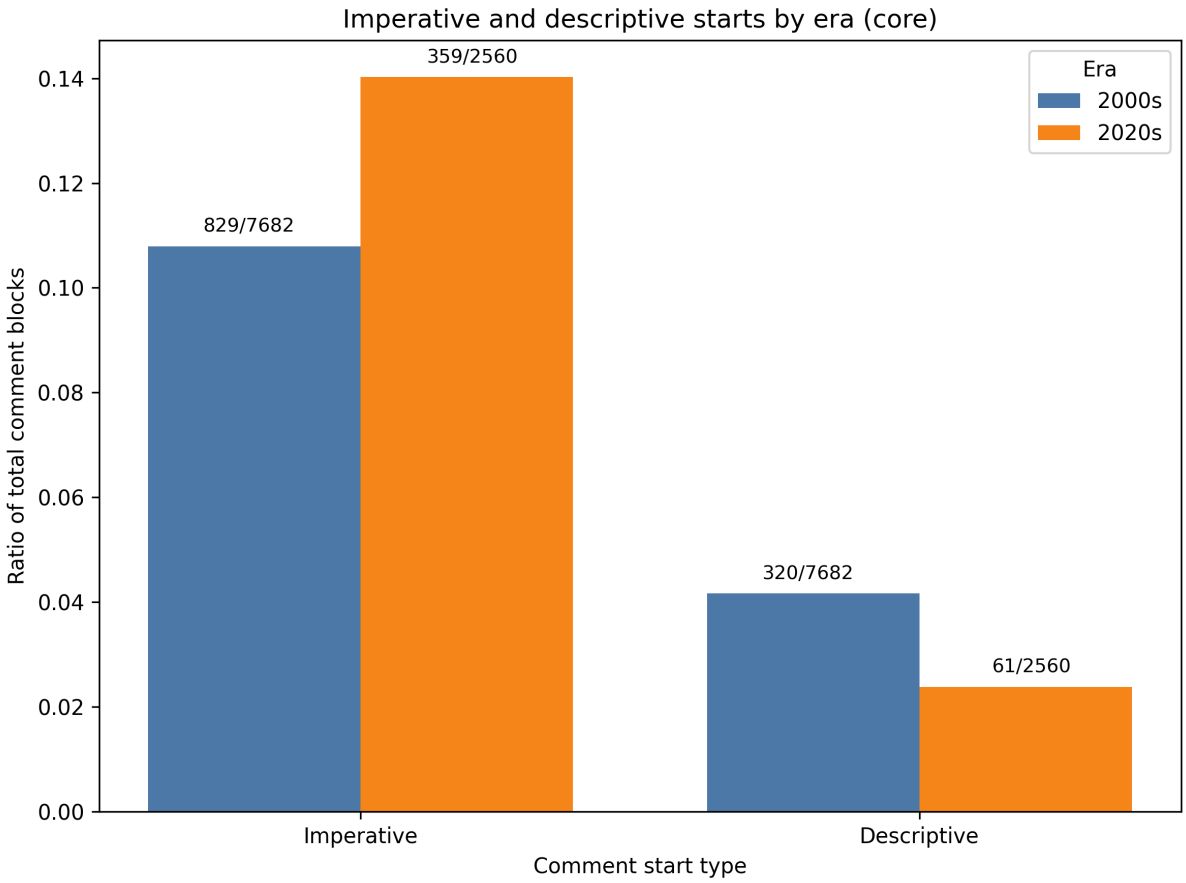

Figure 5.7: Imperative and descriptive comment starts by era.

5.6.2 Punctuation Patterns

A further stylistic feature examined is punctuation at the end of blocks: whether comments end with a period, question mark, or exclamation mark.

Figure 5.8 shows the distribution of punctuation patterns across the two eras.

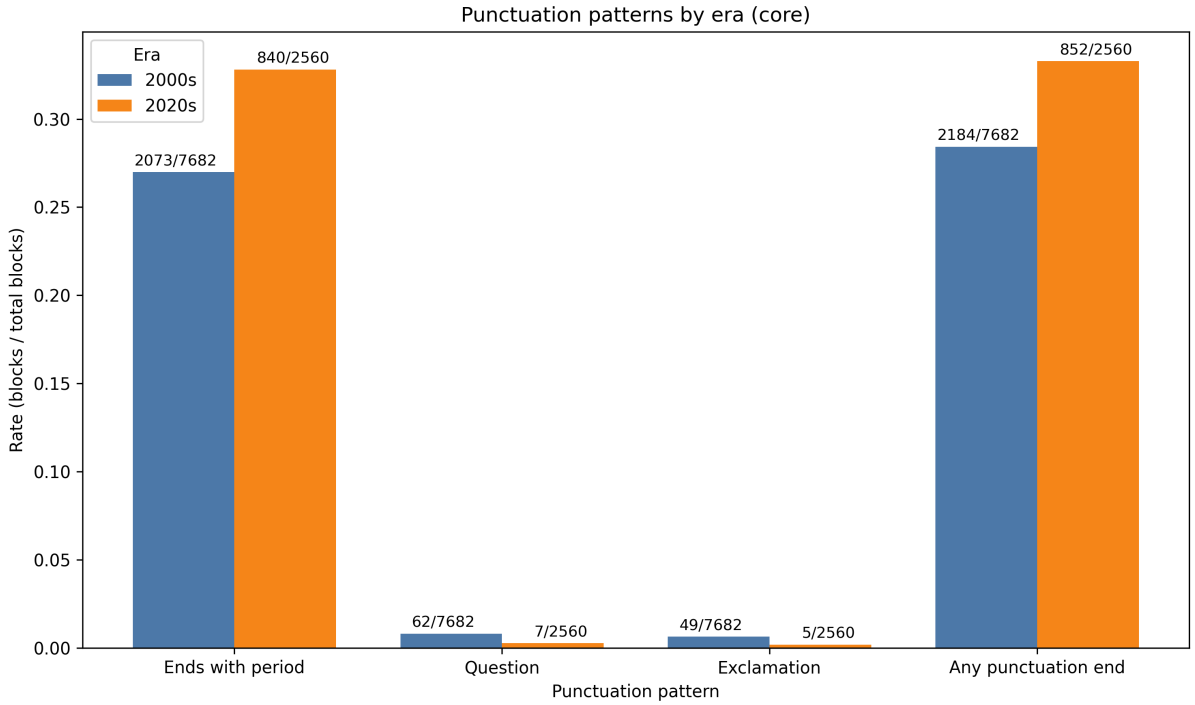

Figure 5.8: Punctuation patterns in block endings by era.

Comments ending with a period become noticeably more common in the 2020s repositories. Roughly 27% of comment blocks in the 2000s end with a period, whereas this proportion rises to approximately 33% in the modern repositories.

In contrast, comments ending with question marks or exclamation marks are rare in both periods. These forms typically represent informal or rhetorical comments rather than structured explanations, which likely explains their limited presence in production codebases.

It is important to note, however, that, while never common, these two types of punctuation endings have become even more sporadic in the 2020s. 62 out of 7682 blocks from the 2000s have a question mark at the end, representing a ratio of 0.81% of blocks. In the 2020s, the number is only 7 blocks out of 2560, translating to a ratio of merely 0.27% of blocks. Similar patterns can be found for the ratios of exclamation marks.

The increased use of sentence-ending punctuation in the 2020s may indicate a gradual shift toward writing more informative comments as complete sentences rather than brief fragments.

Furthermore, the decrease in use of "colloquial" punctuation marks, such as question marks and exclamation points, hints at a more formal and systematic use of comments in code.

5.6.3 Additional Linguistic Features

Beyond grammatical structure and punctuation, several additional stylistic signals were examined. These include features such as capitalization patterns, the presence of numbers or URLs, and comments consisting primarily of separator symbols. Table 5.4 summarizes the distribution of some of these additional linguistic characteristics across the dataset, per era.

Table 5.4: Linguistic features per era

Era	2000s	2020s
Ends with punctuation	28.43%	33.28%
Ends with period	26.99%	32.81%
Ends with question mark	0.81%	0.27%
Ends with exclamation point	0.64%	0.20%
Written in all capital letters	0.26%	0.78%
Starts with a lowercase letter	41.49%	13.01%
Starts with an imperative verb	10.79%	14.02%
Starts with a descriptive verb	4.17%	2.38%
Empty blocks	0.52%	2.73%
Separator blocks	1.13%	0.04%
Contains a URL	0.44%	2.03%
Contains a number	7.49%	8.36%
Contains an emoticon	0.27%	0.04%
Contains parentheses	15.80%	10.39%

A stylistic feature worth highlighting in the table is the ratio of blocks containing an emoticon. The ratio of these blocks has decreased from 0.27% of blocks in the 2000s (21 occurrences), to just 0.04% in the 2020s (1 occurrence), once again highlighting the trend of prioritizing formal and systematic commenting practices. Table 5.5 shows the various types of emoticons found in blocks. The total number of emoticons might amount to slightly more than the total number of emoticon occurrences, because the same block might contain more than one emoticon.

These signals capture stylistic differences in how developers structure comments. Individually, the features vary in prevalence, but read together, they offer a more complete picture of comment-writing conventions across codebases.

Table 5.5: Emoticons in blocks

Emoticon	Count
`:(`	8
`:)`	5
`:-/`	3
`:-)`	3
`:-(`	2
`:P`	2
`;-)`	1

5.6.4 Interpretation

Taken together, the linguistic results suggest that the stylistic form of explanatory comments has evolved modestly over time. Modern repositories

exhibit a slightly stronger tendency toward imperative phrasing and sentence-like structures, as reflected by the increased presence of periods at the end of comments.

These trends are consistent with the broader structural changes observed earlier in this chapter. As comments become more localized, they also appear to adopt clearer grammatical forms resembling short documentation statements.

Although smaller in magnitude than the structural changes discussed earlier, these linguistic shifts indicate that comment-writing conventions are not static, they respond to changes in tooling, language standards, and development workflows.

5.7 Docstrings

In a separate dataset from comment blocks, docstrings have also been collected and analyzed. Unlike regular comments, docstrings are part of the language syntax and can be accessed programmatically through Python's introspection mechanisms. They are therefore commonly used to provide documentation for APIs and library interfaces.

Docstrings have been divided into three groups based on their formal scope: *module*, *class* and *function*. Their density has been analyzed with metrics similar in nature to the ones used for comment blocks in Section 5.1. Moreover, their scope ratios across repositories and eras are also analyzed.

Table 5.6 summarizes docstring densities for each repository, measured both as docstrings per file and docstrings per thousand lines of code (KLOC). The table also reports the distribution of docstrings across different scopes: module, class, and function.

Table 5.6: Docstring (*DS*) densities by repository and era

Repo	Era	DS/File	DS/KLOC	Module DS (%)	Class DS (%)	Function DS (%)
pip	2000s	4.09	16.58	7.41	12.59	80.00
scons	2000s	7.15	23.19	8.64	11.23	80.13
nose	2000s	7.98	33.10	9.93	17.02	73.05
virtualenv	2000s	6.71	15.53	10.64	6.38	82.98
setuptools	2000s	8.24	26.31	4.29	16.79	78.93
pygments	2000s	4.93	15.45	19.41	54.41	26.18
coverage	2000s	7.18	53.49	13.93	12.30	73.77
ipython	2000s	9.39	32.30	9.50	14.69	75.81
2000s Aggregate		**7.54**	**25.72**	**9.65**	**16.19**	**74.16**
rich	2020s	5.92	25.06	4.02	21.75	74.24
textualize	2020s	13.42	43.33	2.42	18.95	78.63
pdm	2020s	3.36	18.58	1.46	26.83	71.71
poetry-plugin-export	2020s	0.40	2.13	0.00	50.00	50.00
build	2020s	2.92	19.98	2.86	22.86	74.29
pytask	2020s	8.14	42.03	10.78	15.50	73.72
pip-audit	2020s	6.00	42.89	16.67	31.41	51.92
typer	2020s	1.30	4.81	3.85	7.69	88.46
2020s Aggregate		**8.36**	**34.18**	**3.87**	**19.97**	**76.16**

Across the dataset, docstrings appear frequently in both eras. The aggregated results

show a moderate increase in docstring density, rising from an aggregate of 7.54 docstrings per file in the 2000s repositories to 8.36 in the 2020s repositories. A similar increase can be observed when measuring docstrings relative to code size: the density grows from 25.72 to 34.18 docstrings per KLOC.

Despite this increase in density, the distribution of docstring types remains relatively stable. In both eras, the majority of docstrings are associated with functions, representing roughly three quarters of all docstrings. Class-level docstrings account for a smaller share, while module-level docstrings are comparatively rare.

This is due to the fact that a codebase naturally contains more functions than classes and modules.

Some variation can be observed between individual repositories. For example, projects such as *textualize* and *pytask* contain relatively high docstring densities, whereas others (e.g. *typer*) use them much more sparingly. These differences likely reflect project-specific documentation practices rather than broader historical trends.

It is worth noting that *typer*, which shows little docstring presence, presented the highest density of comment blocks out of all the sixteen repositories, as shown in Section 5.1. This dual divergence demonstrates that *typer* does not conform to the trends defining 2020s Python commenting practices.

Overall, the results on docstrings suggest that while regular comment blocks have become somewhat less dense over time, docstrings have remained a consistent and widely used mechanism for documenting Python code. Their slightly increased density in the modern repositories may reflect the growing importance of structured documentation in contemporary Python projects.

5.8 Test Code

All analyses presented so far were conducted on the *core* subset of the dataset, meaning files belonging to the main source code of each repository. Test files were excluded from these analyses because they serve a different purpose within the codebase and might have altered the metrics if included in the core analysis.

Nevertheless, the same metrics were computed for the subset of files identified as test files. Table 5.7 summarizes the aggregate densities of comment blocks and docstrings per thousand lines of code (KLOC) for both subsets and eras.

Table 5.7: Aggregate densities per era and subset.

Metric	core 2000s	core 2020s	tests 2000s	tests 2020s
blocks/KLOC	43.85	21.95	31.37	20.87
docstrings/KLOC	25.72	34.18	4.70	13.18

Overall, the test subset largely mirrors the core dataset. Across most metrics, commenting practices shifted between eras in the same direction, suggesting these trends extend beyond production code.

One notable difference, however, concerns docstring usage. While docstrings are relatively

common in the core source files, they appear much less frequently in the test subset. Even in the modern repositories, docstring density in test files remains noticeably lower than in the core code. This likely reflects the different role of test code, which typically involves less structured documentation and does not need to be maintained as much.

Another important aspect is that test code from the 2000s contains an exceptionally high number of shebangs. About 23% of all blocks in 2000s test code are identified as shebang directives. This proportion is reasonable, since test files are usually run directly and shebangs ensure the correct interpreter for these executable files.

This metric completely alters the rates of tooling directives between the 2000s and 2020s. In fact, the 2000s show a higher ratio of tooling directives than the 2020s, entirely due to the large number of shebangs. Figure B.6 illustrates the tooling directives shift for test files.

Excluding shebangs, the upward trend in tooling directives still holds for test files as well.

The detailed figures corresponding to the test subset analyses (similar to all the figures presented in the previous sections of this chapter) are included in Appendix B.

5.9 Cross-Cutting Observations

Taken together, the results in this chapter point to a consistent pattern: commenting practices in Python repositories shifted in several distinct ways between the 2000s and the 2020s.

The most obvious change is density. Modern repositories tend to have fewer regular comment blocks overall, suggesting developers no longer lean on them as a primary documentation mechanism the way they once did.

Structure has changed too. Full-line singleton blocks remain the most common block type across both eras, but modern code has noticeably more inline comments and fewer multi-line full-line blocks. This means that the favored trend is short remarks attached directly to specific lines, rather than larger explanatory sections sitting apart from the code.

Functionally, the shifts are equally clear. Annotation markers like FIXME and XXX have almost disappeared, while TODO increasingly takes their place, a small but real sign of vocabulary converging on a shared convention.

More dramatically, tooling directives have surged in the 2020s. Comments no longer address only human readers, but they also speak to linters, type checkers, and formatters.

Linguistic form has moved as well, though more gradually. Narrative comments more often open with an imperative verb and close with a period. Informal signals (emoticons, question marks, exclamation marks) are rarer. These are modest shifts compared to the structural and functional ones, but they point in the same direction: toward writing that reads less like a note-to-self and more like deliberate documentation.

Docstrings tell a different story. Regular block comment density falls, while docstrings become more popular, suggesting that modern Python projects might favor docstrings as a more structured form of documentation.

None of this necessarily means simply "less commenting." What the data shows is a reorganization: comments become less dense, more localized, more standardized, and more machine-friendly (tooling directives). Furthermore, docstrings grow more prominent.

The following chapter discusses these findings in greater depth, considering their limitations and implications.

Chapter 6

Discussion

This chapter focuses on interpreting the results from Chapter 5 rather than merely restating them. The empirical patterns observed across the repositories raise questions about why commenting practices shifted, how those shifts relate to broader changes in the Python ecosystem, and what they suggest for developers and researchers. The discussion works through comment density, structure, function, and linguistic form in turn, before addressing threats to validity, comparing the findings with related work, and drawing out implications for development practice and future empirical research on code documentation.

6.1 Interpretation of Results

Chapter 5 reveals consistent differences between Python repositories from the 2000s and those from the 2020s, differences that run deeper than comment volume alone. Structure, function, and linguistic form all shifted between the two periods. This pattern is difficult to attribute to chance. Rather, it suggests that commenting conventions changed in step with the broader evolution of Python development workflows, tooling ecosystems, and documentation standards.

Instead of indicating a simple decline in commenting activity, the findings show a gradual reorganization of how comments are used inside Python codebases. The following subsections discuss the most important patterns observed in the dataset.

6.1.1 Changing Density of Comment Blocks

Among the clearest findings is a decline in comment block density. Repositories from the 2000s consistently contain more comment blocks per thousand lines of code than those from the 2020s, a gap that holds across the dataset rather than being driven by a handful of outliers.

A drop in comment density might seem to indicate that modern developers simply comment less. That reading is too narrow. The remaining findings suggest documentation effort has not declined but has migrated toward different mechanisms, such as docstrings, a shift that comment-block counts alone do not capture.

Docstrings have structured machine-readable properties that plain comment blocks lack. It is reasonable to expect that developers have shifted some explanatory writing into docstrings precisely because the format documents modules, classes and functions in a more formal and complete way.

Python itself has also matured since the early 2000s. PEP 8 [7] and the conventions that formed around it set explicit expectations for readable, self-documenting code, which might not necessarily need a comment explaining what the next lines do. When the language and its ecosystem actively discourage obscure constructs, some of what comments once had to explain simply disappears.

In conclusion, it is important to note that fewer comment blocks does not necessarily relate to worse documentation. What the findings suggest is a redistribution of documentation work toward mechanisms that the language and its tooling recognize directly.

6.1.2 Localization of Comments

Density aside, the structural composition of comments has also changed. Inline comments are substantially more common in the 2020s repositories than in those from the 2000s, while multi-line full-line comment blocks have become rarer. The shift is not just quantitative but points to a different relationship between code and comments, one in which explanation sits closer to the line it describes rather than gathered into blocks above or below it.

This shift suggests that modern comments are increasingly attached to individual lines rather than grouped into longer explanatory passages. Inline comments are by nature brief, and their growing prevalence suggests that when developers do comment, they are more often flagging something specific than narrating a block of logic from the outside.

Firstly, modern software development often emphasizes smaller, well-defined functions and modular architectures. When functions are shorter and responsibilities are more clearly separated, explanations can often be added directly to the relevant line of code rather than requiring broader comment sections.

Additionally, tooling also plays a role. As comments increasingly function as signals to automated tools, they also increasingly appear inline. Directives are not inline by convention, but rather they are inline because that is the only place they work. For example, a linter suppression comment from a tool such as `noqa` on its own line has no effect. It needs to be placed as an inline comment after the line of code to target.

6.1.3 Evolution of the Functional Role of Comments

Comments have also shifted in how they function within a codebase. Older repositories used them almost exclusively to explain things to whoever was reading the code. Now, they are increasingly written with automated tools in mind (flags, hints, directives), as much for machines as for people.

This is most visible in how often tooling directives show up in 2020s code. Tags like `noqa`, `type: ignore`, and linter-specific annotations were essentially nonexistent in 2000s projects, but now they have gained traction. As shown, about 20% of comment

blocks in the 2020s era core code are a tooling directive.

These comments form part of a broader ecosystem of static analysis, type checking, formatting, and automated quality assurance tools that have become standard components of contemporary Python development workflows.

As a result, comments now serve another purpose, providing value not only to developers but also to automated systems.

6.1.4 Standardization of Annotation Markers

Annotation markers also illustrate a subtle form of evolution in commenting practices. Although their overall use appears to have decreased, their vocabulary changes noticeably across the two eras.

Markers such as `FIXME` and `XXX`, which appear in several of the older repositories, are almost completely absent in the modern ones. In contrast, the marker `TODO`, which was already in use in the 2000s dataset, but not the most popular marker, becomes the most common marker type in the 2020s dataset. While the intended purposes of the annotation markers differ slightly, in practice `TODO` markers have effectively absorbed the role of `FIXME` markers.

The vocabulary seems to be consolidating with fewer terms that get more consistently used. A shared set of conventions makes code easier to read and simpler to pick up with search tools, issue trackers, or static analysis.

Fewer annotation terms, used more consistently, reflect a wider push toward standardization in how software gets written.

6.1.5 Changes in Linguistic Style

Looking at narrative comment blocks more closely, the stylistic differences between the two eras are more subtle, but they still exist.

First, comments in the modern repositories more frequently begin with imperative verb forms. Imperative constructions such as "*handle errors*" or "*return the result*" resemble the phrasing commonly used in technical documentation and programming manuals.

Second, comments in modern repositories more frequently end with periods, indicating a tendency toward complete sentences rather than brief fragments.

Third, informal textual features decline across the two periods. Emoticons, question marks, and exclamation marks are all on average less frequent in the 2020s dataset. Informal textual features appear to have given way to more neutral, standardized comment language.

Although these stylistic differences are smaller in magnitude than the structural or functional changes observed elsewhere in the analysis, they nevertheless point toward a general trend: comments are becoming more similar to formal documentation.

6.1.6 The Role of Docstrings in Modern Python Projects

Docstrings tell a different story from regular comments. Block comment density generally falls across the two eras, while docstring density moderately increases.

Docstrings hold a distinct place in Python's documentation system. Unlike regular comments, they are part of the language syntax and accessible at runtime through Python's introspection mechanisms, which means tools like documentation generators can extract them automatically to produce API documentation.

The uptick in docstring density likely reflects how central automated documentation and package distribution have become. As Python projects are more widely shared through package repositories and documentation portals, structured documentation carries more practical weight.

The stable distribution of docstrings across modules, classes, and functions also indicates that developers maintain a relatively consistent documentation hierarchy across eras.

6.1.7 Consistency Across Core and Test Code

The test file analysis offers useful context for the main findings. Though test files were excluded from the core analysis to avoid skewing the metrics, applying the same measurements to that subset produces broadly similar patterns across both eras.

The trends seen in the core dataset largely hold in the test subset, suggesting that shifts in commenting practices are not confined to production code but reflect wider changes in how developers write software in general.

One difference does stand out: test files contain far fewer docstrings on average than core source files in both periods. This is unsurprising, as test code is written to verify behavior, not to expose public interfaces that need formal documentation.

It is also worth noting that 2000s test files contain a disproportionate number of shebangs, a tooling directive that has since become less common. This is expected, as shebangs are particularly needed for executable files, such as most test files. This does not, however, contradict the broader finding that tooling directives are generally on the rise.

That said, most structural and stylistic patterns hold across both subsets, which reinforces the interpretation that these changes reflect general shifts in Python development practice rather than artifacts of the sample.

6.1.8 Summary

Taken together, the results point to commenting practices evolving along several related dimensions. Modern repositories tend to have fewer traditional comment blocks, but make greater use of inline comments, structured docstrings, and comments directed at automated tooling.

Comment language has also grown somewhat more standardized and oriented towards documentation over the same period. These shifts mirror broader changes in the Python ecosystem over the past two decades: the rise of automated tooling, the spread of shared

coding standards, and greater reliance on structured documentation.

6.2 Threats to Validity

Like any empirical study, this work is subject to factors that could affect the reliability or generalizability of the findings. Consistent with established practice in empirical software engineering research, the discussion is structured around three validity dimensions: internal, construct, and external.

6.2.1 Internal Validity

Internal validity concerns whether the observed results might not be completely produced accurately inside the analysis process itself.

Parts of the analysis depend on heuristic methods to detect linguistic patterns and classify comment categories. Annotation markers and tooling directives were identified using regex patterns built around predefined keyword lists. This is a pragmatic choice with known limits. Some instances have been misclassified, and others were almost certainly missed (for example, there might have been other rarer types of annotation markers that were not appropriately captured).

The same applies to linguistic features like imperative or descriptive comment openings, which were identified by inspecting the first tokens of each comment block. Natural language is ambiguous enough that no simplified heuristic will handle every grammatical case cleanly.

License header detection introduces its own classification uncertainty. The implemented rules identified most legal headers correctly, but a small number of comment blocks that merely referenced licensing concepts were flagged as full license declarations when they were not. These cases were rare, though they still contributed to altering the rates.

That said, the trends reported in the results are drawn from large comment block counts across multiple repositories. At that scale, occasional misclassifications are unlikely to meaningfully distort the aggregate findings.

6.2.2 Construct Validity

Construct validity concerns whether the chosen metrics actually measure what the study intends.

Here, commenting practices are operationalized through indicators such as comment block density, block structure, annotation marker usage, and linguistic patterns. These capture several meaningful dimensions of commenting behavior, but they do not account for everything that falls under software documentation.

Comment density is a case in point: a low count says nothing about whether the comments that exist are useful, clear, or worth reading. A repository with sparse inline comments might still be well-documented through careful naming, external documentation, or thorough docstrings.

The linguistic analysis has similar constraints. It covers a limited set of stylistic features (most notably punctuation and verb forms) and leaves out dimensions like vocabulary complexity, semantic content, or narrative structure, which were not the subject of this study.

The results should therefore be read as describing observable patterns in comment structure and style, not as a comprehensive account of documentation quality.

6.2.3 External Validity

External validity refers to the degree to which the findings can be generalized beyond the analyzed dataset.

The study examines sixteen Python repositories, divided evenly between projects from the 2000s and the 2020s. Although these repositories represent real-world open-source projects, they cover only a small portion of the Python ecosystem.

Furthermore, the repositories were selected within a relatively specific domain of Python software, primarily involving command-line tools and developer-oriented utilities. Commenting practices in other domains, such as web frameworks, scientific libraries, or educational codebases, might present a completely different distribution and evolution of comments.

The study is also limited to Python. Other languages come with their own commenting conventions, documentation systems, and tooling ecosystems, so the trends observed here do not necessarily extend to software development practices beyond the Python ecosystem.

That said, the dataset covers multiple independent projects and development teams, which limits the extent to which any single project's conventions can skew the results.

6.2.4 Temporal Validity

A separate concern involves interpreting the temporal comparisons. The study contrasts projects from the 2000s against those from the 2020s, but differences between the two groups may not be due to the passage of time alone.

Individual project characteristics, such as coding standards, maturity, contributor communities, and documentation philosophies, can all shape commenting practices independently of era. Repositories were selected to share similar purposes and ecosystems, but fully controlling for every contextual factor is not realistic.

Python itself has also changed considerably over the period the dataset covers. Many projects in the 2000s used Python 2, while in the 2020s Python 3 covers the entirety of the codebases. This might also have influenced how developers write and structure comments, independently of any broader cultural shift in practice.

For these reasons, the differences observed between the two eras are best read as correlations rather than causal relationships.

6.2.5 Summary

The trends reported here are reasonably consistent, but they should be read against the limitations discussed above.

That said, the analysis spans thousands of comment blocks across multiple independent repositories, and similar patterns emerge consistently across different metrics. That consistency is reassuring, suggesting that the observed trends reflect something real about how Python commenting practices have shifted, rather than being an artifact of the analysis itself.

6.3 Comparison with Related Work

The findings of this thesis complement and extend several strands of prior research on source code comments.

Previous empirical studies have frequently focused on the *density* of comments. Arafat and Riehle (2009) [1] analyzed comment density across a large dataset of open-source projects and concluded that commenting is a continuous activity throughout the development lifecycle. The results of the present study are broadly consistent with this observation. Comments appear throughout all analyzed repositories regardless of their age or size, confirming that commenting remains an integral part of software development practice.

However, this study's comparison adds something that earlier density studies did not address. Comments remain present in modern repositories, but their density relative to code size tends to decline, while docstring density increases. This, however, does not contradict Arafat and Riehle's claim of comments being a continuous activity in development.

He (2019) [14] also found that comment density varies substantially across projects and is influenced by factors such as programming language, project purpose, and team structure. The variation observed among the repositories in this study is consistent with those findings. Some projects in the dataset exhibit much higher comment densities than others, indicating that project-specific practices continue to play a significant role in shaping commenting behavior.

Research on annotation markers has primarily focused on their usefulness and lifecycle rather than their historical evolution. For example, Wang et al. (2024) [15] investigated characteristics of effective `TODO` comments and found that many of them persist for long periods and are often associated with technical debt. The results of this thesis support the continued relevance of such annotations while revealing a shift in their vocabulary. Markers such as `FIXME` and `XXX`, which appear in older repositories, are nearly absent in the modern projects, while `TODO` becomes the dominant annotation marker.

Together, these comparisons show how the present study builds on rather than merely replicates earlier work. While previous research has examined comment density and annotation markers in isolation, this study brings a cross-era perspective that combines structural, functional, and linguistic analysis.

6.4 Implications

The results of this study have several implications for both software development practice and future research on source code documentation.

On the practical side, the findings suggest that comments in modern Python projects serve an increasingly varied set of purposes. Beyond explaining code to human readers, they now routinely carry instructions for automated tools: linters, type checkers, formatters. This blurring of roles is worth acknowledging explicitly: developers and project maintainers may find it useful to draw a clearer line between human-oriented comments and tool-oriented directives in their coding guidelines.

The results also point to the growing role of structured documentation mechanisms like docstrings. Traditional comment block density tends to decline, but docstrings remain widespread and are even slightly more common in the modern repositories. This suggests that Python projects have shifted toward language-supported documentation rather than free-form inline comments.

Finally, the linguistic analysis suggests that comment style is a dimension that deserves more attention than it typically receives. The shifts observed here are modest, but they point to a gradual change in how developers formulate explanations within source code. More fine-grained linguistic analysis could shed further light on how programming documentation practices have evolved over time.

Chapter 7

Conclusions

7.1 Summary of Contributions

This thesis examined how commenting practices in Python open-source repositories have changed between the late 2000s and the 2020s. To support the analysis, a reproducible pipeline was developed to extract and analyze comment blocks from source code snapshots of sixteen curated repositories: eight for each era.

The first contribution is the design and implementation of that pipeline. It automatically extracts comments from Python source files, segments them into structural blocks, and computes a set of feature flags describing their structure, function, and linguistic properties. The pipeline was built with reproducibility in mind and operates on repository snapshots retrieved through Software Heritage identifiers.

The second contribution is a reproducible dataset of comment blocks derived from those repositories. Using it, the study measures multiple aspects of commenting practice: comment density, block structure, annotation markers, tooling directives, and linguistic characteristics.

The third contribution is the empirical cross-era comparison itself. Several consistent trends emerge: modern repositories tend to contain fewer traditional comment blocks, but make greater use of inline comments, docstrings, and tooling directives. Annotation marker vocabulary has decreased and has become more standardized, and the linguistic form of comments has shifted slightly toward more structured, documentation-like phrasing.

The souce code of the project can be viewed and downloaded from
`https://github.com/mikegaravani/py-comment-evolution`.

7.2 Future Work

Several directions could extend this work. The dataset could be broadened to cover more repositories and a wider range of domains within the Python ecosystem, which would strengthen the generalizability of the findings.

Applying similar analyses to other programming languages would also help clarify whether

the trends observed here are specific to Python or reflect something more general.

Furthermore, natural language processing techniques could be used to analyze the semantic content and intent of comments, going beyond the rule-based regexes used in this study.

And rather than comparing snapshots across eras, an analysis tracking individual repositories throughout their development history would give a more specific picture of how commenting practices shift within a single project over time.

Acknowledgements

I would like to thank my supervisor, Prof. Simone Martini, for his constructive feedback and support throughout this project.

Special thanks are due to the maintainers of the open-source Python projects used in this study. Their work provides the foundation that made this research possible. Similarly, I would like to thank the Software Heritage platform for preserving and archiving source code, providing a universal source of code for analysis.

Finally, I would like to thank my family and my friends for their continued support during the preparation of this thesis.

I also dedicate a special thought to my cat Limone, who kept me company during the writing of this thesis and is dearly missed.

Use of Generative AI

Generative AI tools were used in a limited capacity during the development of the software described in this thesis. These tools were consulted on occasion to discuss programming approaches, assist with debugging, aid in the creation of testing scripts, and improve the readability of code inside the Python analysis pipeline.

All design decisions, implementation choices, experimental procedures, and data analyses were carried out by the author.

Bibliography

[1] Oliver Arafat and Dirk Riehle. The commenting practice of open source. In *Companion to the 24th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 2009.

[2] Penny Grubb and Armstrong Takang. *Software Maintenance: Concepts and Practice*. World Scientific, 2003.

[3] Oracle. Code conventions for the java programming language. `https://www.oracle.com/java/technologies/javase/codeconventions-comments.html`, 1999. Accessed: 2026-03.

[4] Python Software Foundation. An informal introduction to python. `https://docs.python.org/3/tutorial/introduction.html`. Accessed: 2026-03.

[5] Mark Lutz. *Learning Python*. O'Reilly Media, 4 edition, September 2009.

[6] David Goodger and Guido van Rossum. Pep 257 - docstring conventions. `https://peps.python.org/pep-0257/`, 2001. Python Enhancement Proposal, Accessed: 2026-03.

[7] Guido van Rossum, Barry Warsaw, and Alyssa Coghlan. Pep 8 - style guide for python code. `https://peps.python.org/pep-8/`, 2001. Python Enhancement Proposal, Accessed: 2026-03.

[8] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A study of the documentation essential to software maintenance. In *Proceedings of the International Conference on Design of Communication*, pages 68–75, 2005.

[9] John Ousterhout. *A Philosophy of Software Design*. Yaknyam Press, 2018.

[10] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. Quality analysis of source code comments. In *Proceedings of the 21st International Conference on Program Comprehension (ICPC)*, pages 83–92, 2013.

[11] Margaret-Anne Storey, Jody Ryall, R. Ian Bull, Del Myers, and Janice Singer. Todo or to bug: Exploring how task annotations play a role in the work practices of software developers. In *Proceedings of the 30th International Conference on Software Engineering*, pages 251–260, 2008.

[12] JetBrains. Todo comments. `https://www.jetbrains.com/help/pycharm/using-todo.html`, 2026. Accessed: 2026-03.

[13] Marc-André Lemburg and Martin von Löwis. Pep 263 - defining python source code encodings. https://peps.python.org/pep-0263/, 2001. Python Enhancement Proposal, Accessed: 2026-03.

[14] Hao He. Understanding source code comments at large-scale. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*, pages 1217–1219. ACM, 2019.

[15] Haoye Wang, Zhipeng Gao, Tingting Bi, John Grundy, Xinyu Wang, Minghui Wu, and Xiaohu Yang. What makes a good todo comment? *ACM Transactions on Software Engineering and Methodology*, 33(6):1–30, 2024.

[16] F. Rabbi et al. An ensemble approach to detect code comment inconsistency. In *Proceedings of the 32nd International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2020.

[17] Yuan Huang, Yinan Chen, Xiangping Chen, and Xiaocong Zhou. Are your comments outdated? towards automatically detecting code-comment inconsistency, 2024.

[18] Roberto Di Cosmo and Stefano Zacchiroli. Software heritage: Why and how to preserve software source code. In *iPRES*, 2017.

[19] Software Heritage. Software heritage archive. https://www.softwareheritage.org/, 2016. Accessed: 2026-03.

[20] Software Heritage Documentation. Vault api reference. https://docs.softwareheritage.org/devel/swh-vault/api.html. Accessed: 2026-03.

[21] Python Software Foundation. tokenize - tokenizer for python source. https://docs.python.org/3/library/tokenize.html. Python documentation, Accessed: 2026-03.

[22] Python Software Foundation. ast - abstract syntax trees. https://docs.python.org/3/library/ast.html. Python documentation, Accessed: 2026-03.

[23] Parso Developers. parso - a python parser. https://parso.readthedocs.io. Accessed: 2026-03.

[24] Apache Parquet Developers. Parquet documentation. https://parquet.apache.org/docs/overview/. Accessed: 2026-03.

[25] Martin von Löwis. Pep 3120 - using utf-8 as the default source encoding. https://peps.python.org/pep-3120/, 2007. Python Enhancement Proposal, Accessed: 2026-03.

Appendix A

Repository Table with SoftWare Hash IDentifiers

This appendix presents table A.1, where each repository also showcases its *SWHID* (SoftWare Hash IDentifier).

Table A.1: Repository table containing *SWHIDs*.

Project	Release	Release Date	SWH Directory ID
virtualenv	v1.4	2009-11-07	5262c2c114b4e591bd12c840195609def687860f
coverage	v3.0	2009-06-14	4e6a8ee64b61f84b596b5b0c9531e8e83bd1e064
pip	v1.0.1	2011-04-30	f41c2246cb66fe1aac134195fe788bd72baf2853
setuptools	v0.6c11	2009-10-20	e2e477a90f4f061ff2ddc2713cfadc46fdc24071
nose	v0.11.1	2009-05-13	7f6051bfe88ae470efb6abda2994e30c84694843
pygments	v1.0	2008-11-23	a82251310eff87d529dbe2502e07bdfd07c9a48b
ipython	v0.10.1	2010-10-12	90353396ee140362f37d8e7bee19b693ea16438a
scons	v1.0.0	2008-08-12	c5a9aef04201dc9299862d67cd3bdcf41887a653
ppe	v1.10.0	2026-01-18	9a00228714b7969724277b1cb38538330ca374b6
build	v1.4.0	2026-01-08	e09f3ac0100e897a4ee2341844f59515bf97e692
typer	v0.19.0	2025-09-20	47fa9af9614741f2c1b9a69f134203055d89c1b0
pip-audit	v2.9.0	2025-04-07	03f6d430cc3a76d56be02f6d49fd9b2914a2208b
pytask	v0.5.8	2025-12-30	7aa1bba9a44b5eec1ea81f82cf6846805c1510f1
rich	v14.0.0	2025-03-30	17252f25ff08d6d1a33ab14b8d6ccecdc26addd9
pdm	v2.26.0	2025-10-11	6ba48fc6c930bb3f20784ab065164f8a82d764e4
textual	v1.0.0	2024-12-12	27fb020b7dae4683ef7b6a41460b0c3456c5a258

Appendix B

Test Subset Figures

This appendix presents additional figures produced from the analysis of the *test* subset of the dataset. While the main results in Chapter 5 focus on the *core* source code files, the same analysis pipeline was also applied to files identified as tests. The figures included here replicate the visualizations shown in Chapter 5, but applied to the test files of the repositories.

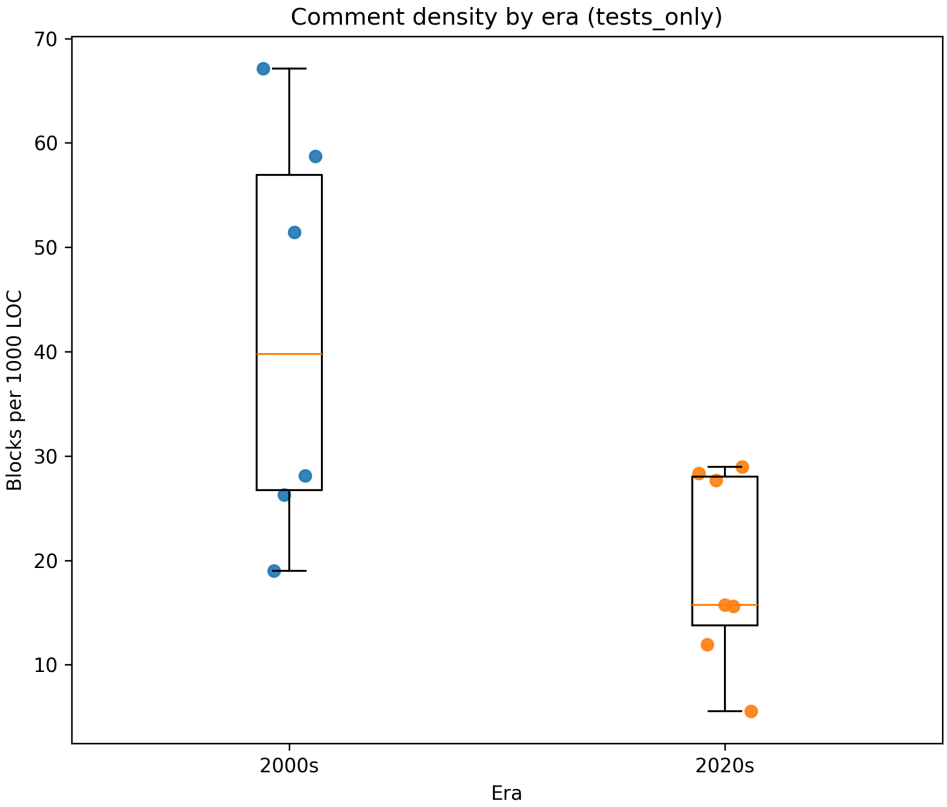

Figure B.1: Box plot showing the block density for the two eras. One dot equals one repository. [TESTS]

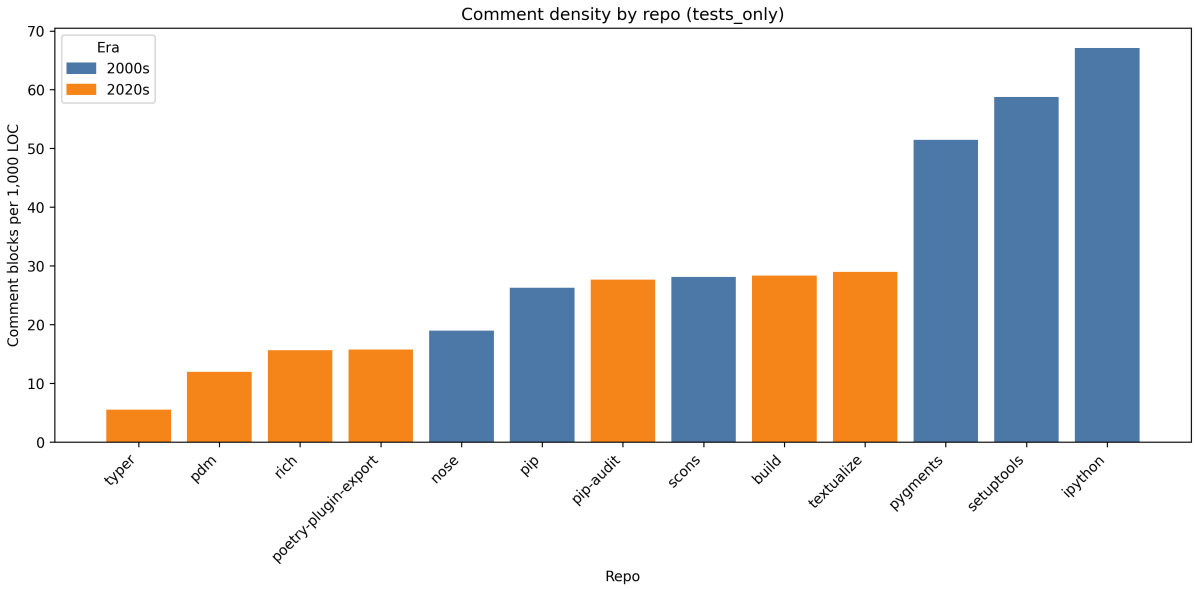

Figure B.2: Bar chart showing each repository in ascending order of blocks/KLOC, color coded by era. [TESTS]

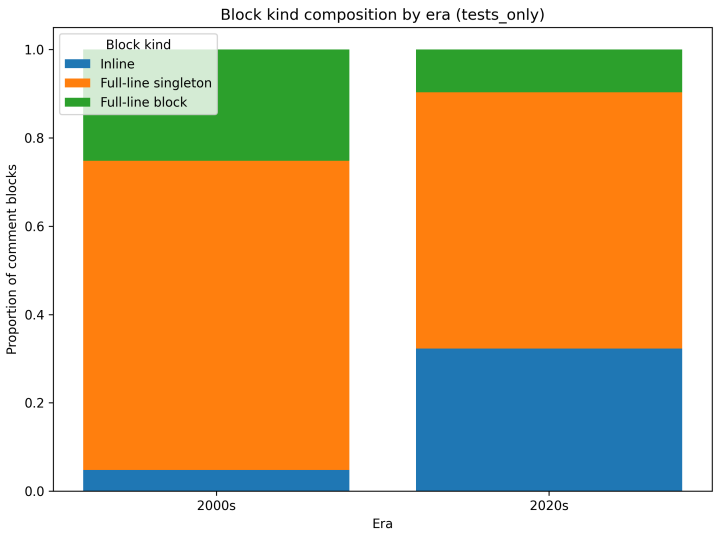

Figure B.3: Distribution of comment block types by era (license header blocks excluded). [TESTS]

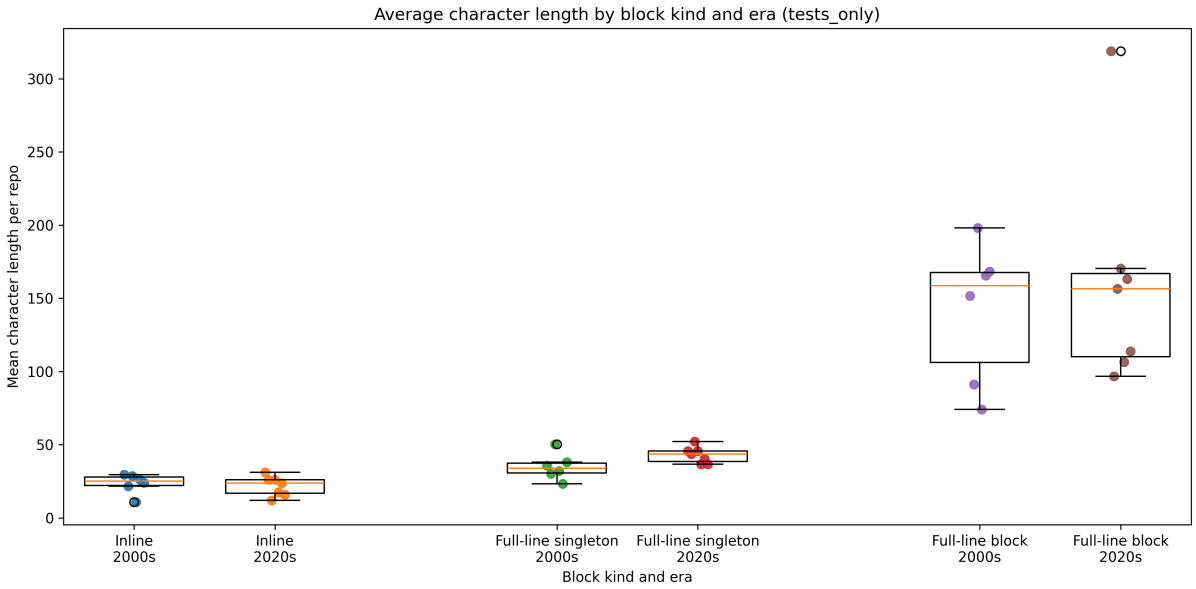

Figure B.4: Average comment block length by block type and era (license header blocks excluded). [TESTS]

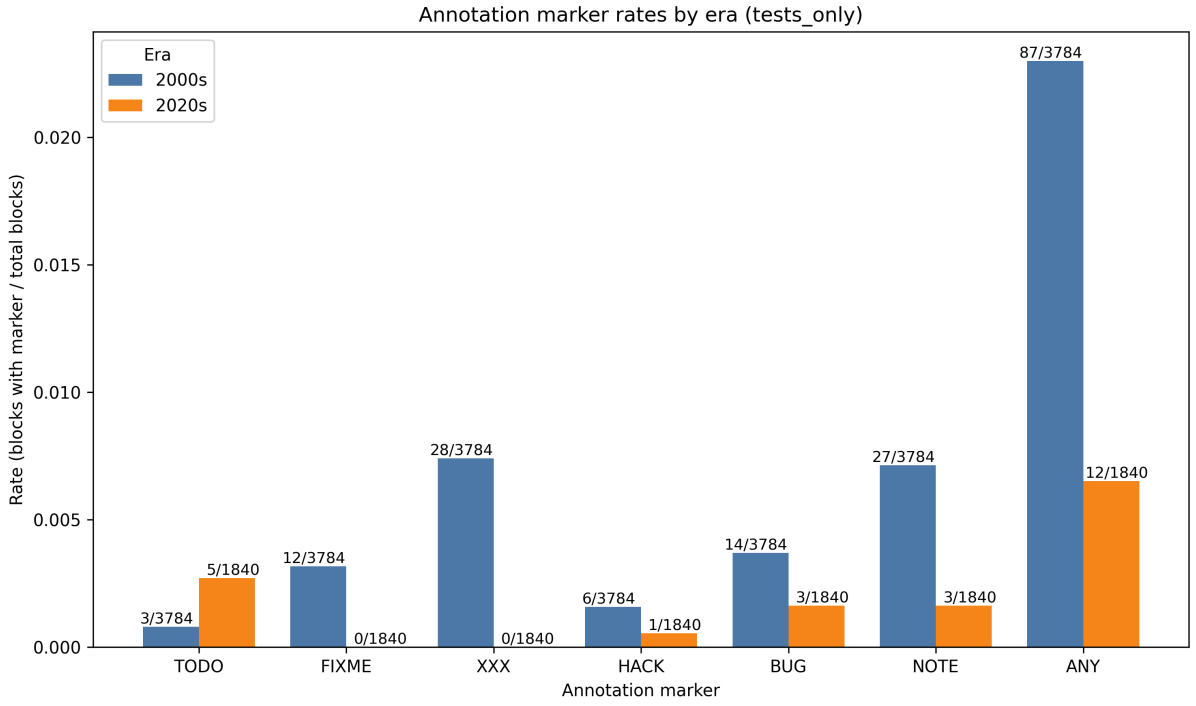

Figure B.5: Annotation marker rates by era. Values above each bar show the absolute counts used to compute the rates. [TESTS]

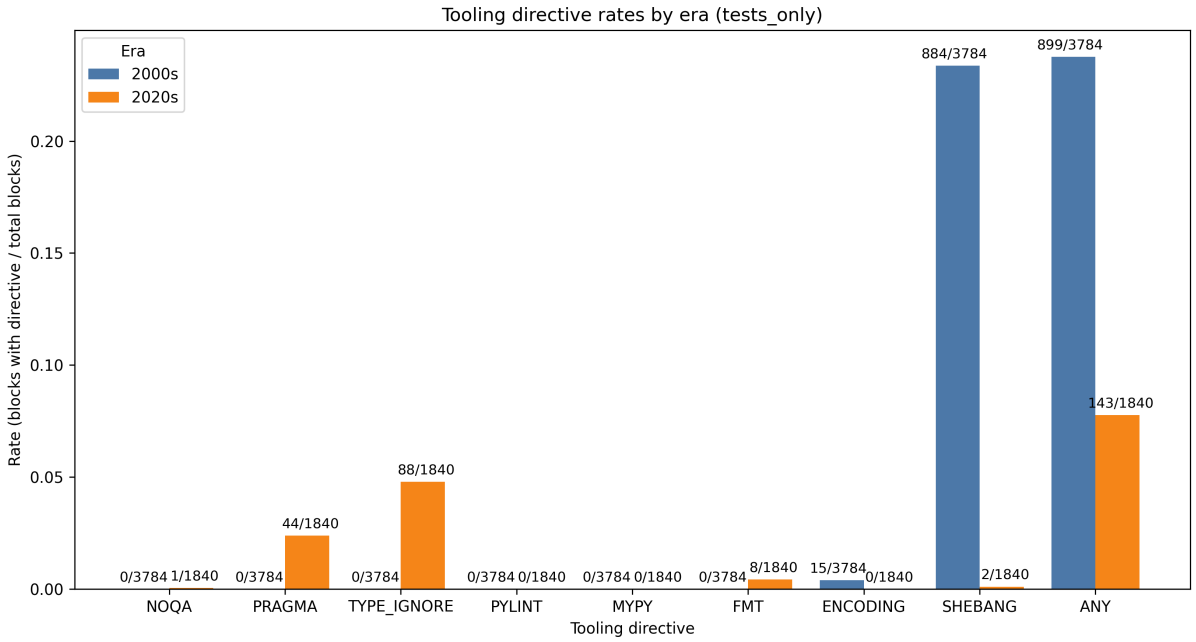

Figure B.6: Tooling directive rates by era. Values above each bar show the absolute counts used to compute the rates. [TESTS]

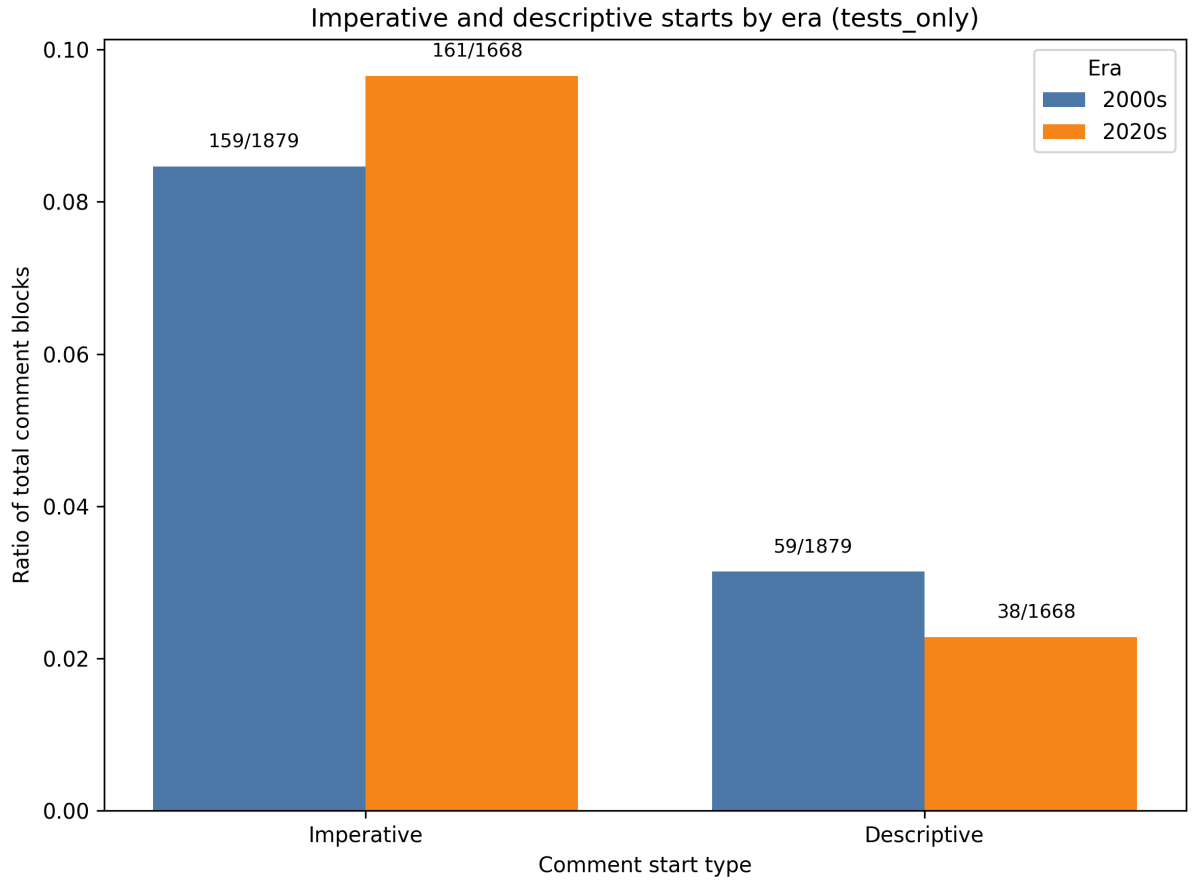

Figure B.7: Imperative and descriptive comment starts by era (only narrative blocks: no legal headers, no annotation markers, no tooling directives). [TESTS]

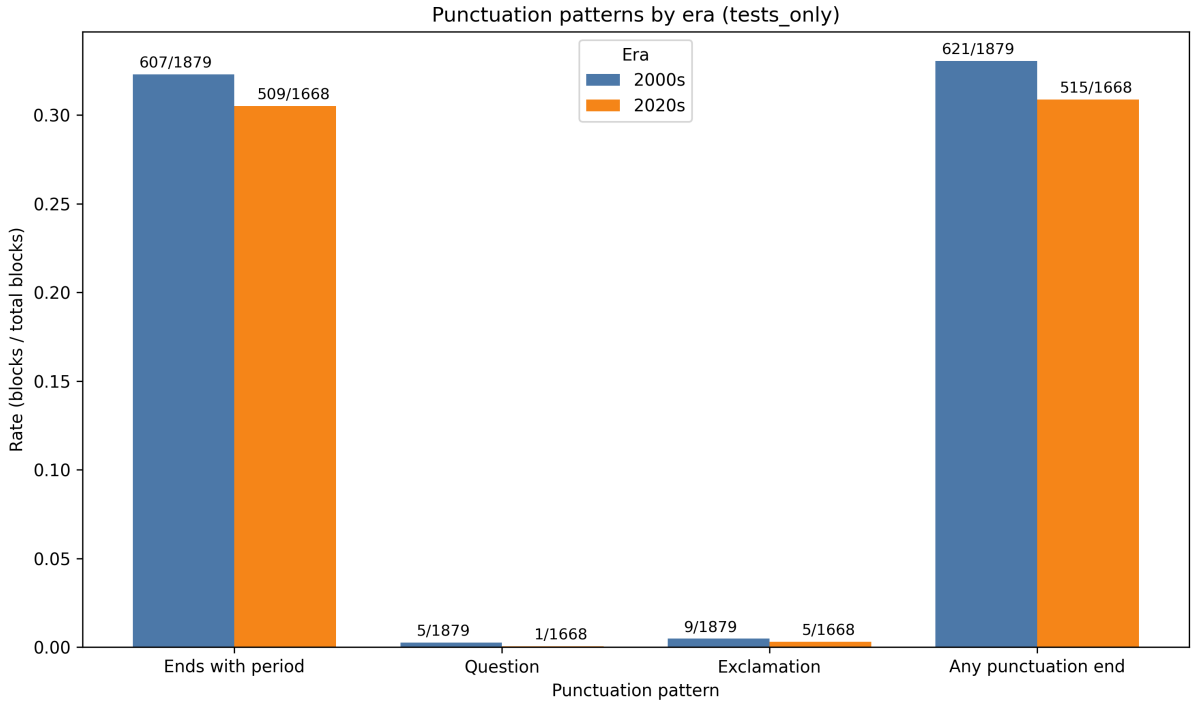

Figure B.8: Punctuation patterns in block endings by era (only narrative blocks: no legal headers, no annotation markers, no tooling directives). [TESTS]

Appendix C

Summary in Italian

Questo lavoro si concentra sull'analisi di differenze nei commenti di codice sorgente di progetti open-source. In particolare, sono stati selezionati sedici progetti scritti in Python, otto dei quali negli anni 2000, e gli altri otto negli ultimi anni a partire da attorno al 2020. In tal modo, i progetti rappresentano due epoche distinte di sviluppo software. Nell'analisi, sono emerse diverse tendenze nella popolarità dei commenti, la loro densità e varie altre differenze nelle loro caratteristiche strutturali e linguistiche.

La scelta dei sedici progetti è stata pensata per fare in modo che tali progetti possano essere direttamente comparabili sotto varie metriche. Ad esempio, tutte le repository sono strumenti CLI Python ampiamente utilizzati e attivamente mantenuti al momento dello snapshot della versione specifica analizzata.

Le versioni analizzate sono state recuperate tramite l'archivio Software Heritage, che consente di identificare snapshot specifici del codice sorgente tramite identificatori SWHID, garantendo la riproducibilità dell'analisi. Complessivamente, il dataset comprende diverse migliaia di file Python e centinaia di migliaia di righe di codice.

Per analizzare le repository è stata progettata una pipeline automatica in grado di estrarre e classificare i commenti presenti nel codice sorgente. Il processo si articola in più fasi: l'ingestione degli snapshot, l'estrazione dei commenti tramite il modulo `tokenize`, la segmentazione in blocchi, e l'estrazione di caratteristiche strutturali e linguistiche. I commenti vengono raggruppati in blocchi secondo regole semplici: un commento inline costituisce sempre un blocco a sé, mentre sequenze di commenti su righe consecutive con la stessa indentazione vengono trattate come un unico blocco. La pipeline estrae anche le docstring, gestite separatamente in quanto parte della sintassi del linguaggio.

L'analisi pone quattro domande di ricerca (**Research Questions**). Tutte e quattro le domande vengono poste sull'evoluzione tra le due epoche di date caratteristiche dei commenti nel dataset in analisi. La prima riguarda la densità dei commenti, misurata tramite indicatori come il numero di blocchi per file e per mille righe di codice. La seconda riguarda la struttura, distinguendo tra commenti inline, commenti su riga singola e blocchi multi-riga. La terza esamina l'uso di marcatori specifici come `TODO`, `FIXME`, `XXX` e `HACK`, con cui gli sviluppatori segnalano lavoro in sospeso o problemi noti. Inoltre, essa guarda anche allo sviluppo di commenti specifici che interagiscono con strumenti automatici (ad esempio, commenti `# noqa` per sistemi di linting). La quarta analizza alcune caratteristiche linguistiche, tra cui la punteggiatura, la forma grammaticale delle

frasi e la presenza di elementi informali come emoticon o separatori decorativi.

I risultati mostrano diversi cambiamenti tra le due epoche. La densità complessiva tende a diminuire nei repository più recenti: i progetti degli anni 2020 contengono mediamente meno blocchi di commento per unità di codice rispetto a quelli degli anni 2000. Cambia però la struttura: nei progetti recenti i commenti inline sono più frequenti, mentre i blocchi multi-riga risultano meno comuni. Gli sviluppatori, in altre parole, sembrano preferire annotazioni brevi e localizzate rispetto a spiegazioni estese.

Le docstring invece diventano più popolari, sottolineando l'importanza nel codice moderno di una documentazione formale per le spiegazioni di parti del codice come funzioni e moduli.

Un'altra differenza rilevante riguarda l'uso dei commenti come mezzo di comunicazione con strumenti automatici. Nei progetti degli anni 2000 queste direttive sono quasi assenti (tranne per il caso di *shebangs* e *encoding declarations*), mentre nei progetti moderni compaiono in una percentuale significativa dei commenti. Questo risultato riflette il crescente ruolo degli strumenti automatici nel processo di sviluppo software moderno.

Per quanto riguarda i marcatori di annotazione, lo studio mostra una decrescita di utilizzo totale ma anche un cambiamento nella terminologia utilizzata dagli sviluppatori. Alcuni marcatori storicamente diffusi, come `FIXME` e `XXX`, compaiono molto più spesso nelle repository più vecchie, mentre nei progetti moderni tende a prevalere `TODO`, che si è affermato come convenzione dominante per segnalare attività incomplete o miglioramenti futuri.

L'analisi linguistica evidenzia cambiamenti stilistici più sottili. Nelle repository moderne è più comune trovare commenti formulati come frasi complete con punto finale, mentre in quelli più vecchi compaiono più spesso forme informali come domande ed esclamazioni che implicano toni più colloquiali. I commenti moderni tendono inoltre ad aprirsi con verbi all'imperativo, uno stile che richiama le convenzioni della documentazione tecnica e delle linee guida di programmazione.

Nel complesso, i risultati suggeriscono che i commenti nel codice Python non sono semplicemente aumentati o diminuiti nel tempo, ma hanno cambiato forma e funzione. Nei progetti moderni tendono a essere più brevi, più localizzati e più standardizzati, usati non solo per spiegare il codice ad altri sviluppatori, ma sempre più spesso per comunicare con strumenti automatici di analisi e formattazione.

Questo lavoro offre quindi una visione empirica dell'evoluzione delle pratiche di commento nel software Python open-source. Oltre ai risultati in sé, la tesi mette a disposizione una pipeline di analisi riproducibile e un dataset di repository selezionati che potranno essere riutilizzati in studi futuri sulle pratiche di sviluppo Python.