MEASURING SOFTWARE INNOVATION WITH OPEN SOURCE SOFTWARE DEVELOPMENT DATA

Eva Maxfield Brown ● *
University of Washington
Seattle, WA, USA

Cailean Osborne ●
University of Oxford
Oxford, UK

Peter Cihon ●
GitHub
San Francisco, CA, USA

Moritz Böhmecke-Schwafert ●
Technical University of Berlin
Berlin, Germany

Kevin Xu ●
GitHub
San Francisco, CA, USA

Mirko Boehm ●
The Linux Foundation
Berlin, Germany

Knut Blind ●
Technical University of Berlin, Fraunhofer ISI
Berlin, Germany

ABSTRACT

This paper introduces a novel measure of software innovation based on open source software (OSS) development activity on GitHub. We examine the dependency growth and release complexity among ∼200,000 unique releases from 28,000 unique packages across the JavaScript, Python, and Ruby ecosystems over two years post-release. We find that major versions show differential, strong prediction of one-year lagged log change in dependencies. In addition, semantic versioning of OSS releases is correlated with their complexity and predict downstream adoption. We conclude that major releases of OSS packages count as a unit of innovation complementary to scientific publications, patents, and standards, offering applications for policymakers, managers, and researchers.

Keywords Open source software · innovation indicators · GitHub · dependency analysis · semantic versioning · release complexity · software ecosystems

1 Introduction

There is widespread, interdisciplinary research and policy interest in measuring innovation, with attention traditionally focusing on academic outputs [1], business outcomes [2], and their interactions [3]. Yet software innovation, particularly in open source software (OSS) innovation, often eludes conventional metrics like scientific publications and patents. While OSS are digital public goods that firms leverage as an "inward knowledge flow" to produce innovations [2], it also constitutes innovation in its own right. The OECD Oslo Manual [2] frames innovation through product and process advances, evaluating novelty either by comparing to the current state of the art or by assessing ex-post impact. This framework poses unique challenges for OSS, where individual contributions may appear incremental—often just a few lines of code—but collectively drive radical innovation and creative destruction [4]. This tension between granular contributions and systemic impact highlights the need for new approaches to measuring OSS innovation.

Previously, researchers have sought to measure OSS innovation through collaboration activity [5, 6], its economic impact [7, 8], and value [9, 10]. In this paper, we define and operationalize a novel method of measuring OSS

---

*Corresponding author: Eva Maxfield Brown, evamxb@uw.edu

innovation in discrete units, offering researchers and policymakers a software innovation metric that is complementary to publications, patents, and standards. Our method uses publicly available data from GitHub, leveraging software developers' publication of packages and others developers' decisions to adopt the software. Developers often use semantic versioning when publishing a change to a software package that denotes its significance as major, minor, or a patch [11]. In practice, then, versioning increments may represent inventive steps in software. By reviewing other developers' activity in response to new versions, the open source community can offer external validation of whether an increment is in fact innovative. Specifically, we assess package dependencies by examining how frequently new versions are integrated into other projects over time, providing a metric to evaluate the impact of these inventive steps.

Data & methods: *We analyze a dataset of approximately 200,000 unique releases from 28,000 unique packages across the JavaScript, Python, and Ruby ecosystems on GitHub, examining package dependencies and release complexity over time frames up to two years post-release. Our methodology combines quantitative analysis of dependency growth patterns with a machine learning approach to assess release complexity using GPT-4.*

Key finding 1: *At the one-year post-release mark, major releases are associated with significantly larger increases in package dependencies compared to minor or patch releases across the JavaScript, Python, and Ruby ecosystems, highlighting distinct patterns of dependency growth that emerge following significant package updates.*

Key finding 2: *Major releases continue to generate larger positive changes in log-difference compared to minor and patch releases, regardless of the package ecosystem, initial package utilization, or time frame examined up to two years post-release. However, the magnitude of the log-difference tends to diminish slightly in the second year.*

Key finding 3: *Release complexity, as rated by a prompted GPT-4 model, correlates slightly positively with dependency growth, particularly for JavaScript and Python packages. However, the Ruby ecosystem does not exhibit the same correlation, suggesting different dynamics in the Ruby community and a different handling of release notes.*

Limitations: *We do not account for software licensing effects on dependency relationships (e.g. GPL vs. Apache v2), geographic patterns of adoption, or the inherent variability in our LLM-based semantic analysis methodology.*

Implications statement: *The findings provide new insights into OSS innovation dynamics, offering potential applications for policymakers, practitioners, and researchers in evaluating and fostering innovation in OSS development.*

The paper is structured as follows. In § 2, we discuss prior work on innovation indicators and the gap for OSS innovation. In § 3, we discuss our approach to the measurement of units of software innovation using GitHub data. In § 3.3.3, we report our key findings. In § 4, we discuss the implications and limitations of our method, as well as future research directions.

2   Related Work

This section reviews prior work on innovation indicators and discusses the absence of indicators for OSS innovation. We then establish a conceptual foundation for developing OSS innovation indicators by reviewing prior OSS measurement approaches and available data on OSS development and innovation.

2.1   Innovation indicators and the open source gap

Innovation plays a fundamental role in driving long-term economic growth by fostering productivity improvements, technological advancements, and competitive advantages [12]. According to the 2018 OECD Oslo Manual [2], innovation refers to the "introduction of a new or significantly improved product or process that differs substantially from previous offerings". For products, this involves being made available to potential users, while for processes, it means being brought into use by the organization. Consequently, the accurate and efficient measurement of innovation is essential for both policymakers and organizations [2].

Innovation indicators are measurable values that provide insights into specific aspects of innovation or its overall status. As defined by the OECD [2], these indicators summarize observed innovation phenomena within a population over time and enable the analysis of behaviors and activities, including developmental, financial, and commercial measures taken by organizations. The evaluation of innovation is complex, with various methodologies discussed in the literature. Dziallas and Blind (2019) [1] identified 82 distinct indicators in their systematic review of innovation indicators. Based on this, we provide a summary of common innovation indicators in **Table 1**.

Innovation indicators constitute a fundamental metric for the evaluation of innovation performance. Companies utilize these metrics to allocate resources and evaluate the efficacy of innovation processes [13, 14]. For policymakers, innovation indicators are indispensable in shaping and implementing mission-oriented innovation policy [15]. They provide a means to rigorously evaluate regulatory impacts and the efficacy of funding initiatives [16], and facilitate

a data-driven approach to policy development and resource distribution [17]. Moreover, investors rely on innovation indicators to inform decisions on funding new ventures, underlining their significance for entrepreneurship [1].

Table 1: Characteristics of innovation output indicators adapted from Blind et al (2023) [18]

	Publications	**Patents**	**Standards**	**Start-ups**
Input	Scientific references (backward citations)	Patents, NPL references, standards	Standards, Scientific references	Patents, Standards, Open source
Process	Co-authors plus affiliations	Co-inventor	Standardization working group (limited visibility)	Start-up teams
Assessment	Peer review and acceptance by editors based on novelty and contribution	Granting by patent office based on novelty and inventive step	Publication based on stakeholder consensus and state of practice	Decision by founder/investors based on expected commercial success
Impact of output	Forward citations in publications, patents, standards	Forward citations in patents, publications, standards	Citations in standards, patents, publications	Investments, employment
Measurement of implementation	Indirectly via references in patents and standards	Patent-product database	Specific cases (e.g. laptops) or web mining	Revenue with product portfolio

Kleinknecht et al. [19] argue that widely used innovation indicators, such as R&D spending and patent applications, have significant limitations, including varying patent valuations and time lags. Consequently, using a broader range of data sources is essential for constructing more comprehensive indicators. The OECD criteria further emphasize this, focusing on four dimensions: Relevance, which ensures indicators meet user needs; Accuracy/Validity, requiring unbiased representations of innovation; Reliability/Precision, ensuring consistent results when repeated; and Timeliness, which stresses the availability of data for decision-making [2].

Additionally, underpinning indicators include methodologies for measuring innovation, such as the selection of units of innovation, with novelty being a key consideration. The OECD defines novelty as a criterion for determining whether a product or process is "significantly different" [2] from previous iterations, thus qualifying as an innovation. This evaluation can be approached in two ways: first, by comparing a firm's innovations with the state of the art within the firm and its market or industry; and second, by assessing the impact of an innovation in transforming or creating a new market, which may indicate radical or disruptive innovation. Unlike the first approach, assessing the impact of an innovation can only be done ex-post, often requiring several years to evaluate its effects.

Hence, the 2018 version of the OECD Oslo Manual advocates for the inclusion of novel digital data sources, emphasizing how automated data collection can expand the range of available data, thereby enhancing the development of more diverse and dynamic innovation indicators [2] [20]. Traditional indicators, in particular, often lack timeliness and accuracy and tend to underestimate innovation activities in some contexts [20]. In response, recent studies have begun to explore the potential of novel indicators, for example, based on web-based data sources such as startup databases and online repositories (see e.g. [21] [20]).

To date, OSS indicators are rarely discussed in prior innovation scholarship. For example, a systematic literature review of innovation indicator studies did not even mention "open source" [1]. In 2018, the OECD considered OSS only under "inward knowledge flows", but did not consider it as innovation per se [2]. However, (open source) software development is an act of innovation in its own right, which contributes to product and process innovations alike that generate significant economic value. This is particularly true in the context of emerging technologies, where traditional metrics such as patents are increasingly inadequate for capturing the full scope of innovative activity [22]. Hence, the rapid pace of development and the collaborative ethos inherent in OSS projects frequently result in innovations that elude conventional patenting processes, underscoring the necessity for alternative indicators to evaluate their impact.

Recently, research interest in the economics of OSS has begun to increase, albeit not for OSS innovation indicators directly. Hoffman et al. (2024) [9] estimate the supply-side value of OSS at $4.15 billion, while the demand-side value is estimated to reach a significantly larger $8.8 trillion. The authors conclude that firms would need to spend 3.5

times more on software if OSS did not exist, highlighting not only its economic impact of OSS but also its innovative potential in software development. Meanwhile Korkmaz et al. (2024) [10] develop a framework for measuring OSS value using GitHub data, estimating US investment in OSS at $37.8 billion in 2019. Furthermore, Wright et al. (2023) [8] demonstrate a correlation between country-level GitHub participation and increased founding of innovative ventures, suggesting that OSS activity serves as a catalyst for innovation-driven entrepreneurship. Lin and Maruping (2021) [23] add nuance to this relationship, showing that the benefits of OSS engagement for startups vary across their lifecycle stages, with different advantages accruing from inbound and outbound OSS activities.

The way in which OSS is developed presents challenges for the measurement of innovation, with tensions between micro- and macro-level effects of OSS development activity. At the micro-level, individual contributions to an OSS repository, such as a few lines of code in a commit, may be too marginal to qualify as an inventive step. By comparison, patents require a significant inventive step and a certain level of novelty to be granted. Yet, when considered at scale, the OSS community contributes to radical innovation and creative destruction in the ICT sector [4] and OSS is used widely, making up 96% of software stacks according to recent surveys [24]. This underscores the discrepancy between the incremental contributions made by individuals at the micro-level and the substantial impact of OSS on innovation and its widespread incorporation into modern software stacks, an area that has yet to be thoroughly examined.

In addition, one must take into consideration that OSS represents a unique innovation model that blends elements of user innovation [25], co-creation [26], and open science [27]. This model is characterized by a complex ecosystem of contributors with diverse economic, social, and technological incentives, from individuals [28, 29] to research institutes and universities [30] to multinational corporations [31, 32]. Companies—both large and small—use, contribute to, and fund OSS development [7, 33, 34]. Thus, innovations in OSS may be the outcome of contributions across diverse organizations and developers. Due to this open collaboration model, value creation and value capture operate differently than in traditional R&D processes or team structures in university or industry labs [35].

Given the gap in OSS innovation indicators and the challenges presented by the nature of OSS development, there is a pressing need to develop new approaches for measuring units of innovation in this domain. The following sections explore approaches that have been proposed or may be adapted to address this challenge, drawing insights from (1) established innovation indicators not specifically related to OSS and (2) prior approaches to measuring OSS innovation using data from popular OSS hosting and development platforms such as GitHub. By examining these approaches, we aim to lay the groundwork for developing more comprehensive and precise indicators of OSS innovation that can capture both the incremental nature of individual contributions as well as the transformative potential of OSS at scale.

2.2 Learning from existing innovation indicators

In this section, we establish connections between established innovation indicators and potential OSS innovation indicators. Scientific publications offer a well-established benchmark for innovation measurement. Key performance indicators include the volume of output and forward citations received [36]. Brown et al. demonstrated a similar approach for measuring the impact of scientific OSS repositories and software projects by creating large networks of scientific publications with links to software used and mentioned in each article [37]. This approach aligns with traditional bibliometric analyses but applies them to software artifacts.

Patents provide another crucial external indicator. They can reference OSS projects, particularly in non-patent literature (NPL) citations, indicating the industrial relevance and potential commercial value of a repository. As Harhoff et al. (2003) [38] note, NPL citations are particularly important in fields like pharmacy and chemistry, while in the ICT sector standards are also frequently cited [39]. For OSS repositories, one can track the number of patents citing them and analyze the characteristics of the citing patents and their applicants.

Standards are another established indicator of innovation [40]. Standards play a crucial role in innovation diffusion, which is a measure of the success of innovative products and processes. Empirical evidence shows that involvement in standardization [41] and the implementation of standards [42] significantly correlate with companies' innovation performance. For OSS repositories, we can examine the number of links to repositories in standards documents and analyze how these standards are subsequently used or cited. The European Commission's recent Code of Practice on Standardization (2023) [43] recognizes standards as outputs in the context of research projects, further validating their importance as innovation indicators. However, it is worth noting that contributor information for standards is often limited, with only a few standard-setting organizations, such as AFNOR in France, releasing this data [44].

Start-up ecosystems provide another indicator of OSS innovation. As Autio et al. [45] argue, start-ups can be effective drivers for the implementation of innovation, playing a crucial role in economic development [46], technological advancement, and societal impact [8]. For OSS repositories, we can analyze the use of repositories by start-ups, examining factors such as VC funding and growth rates of start-ups referencing specific repositories.

Table 2: Characteristics of OSS repositories and comparison with innovation indicators

(a) Characteristics of OSS repositories

Indicators	Implications for Innovation
Commits	Input indicator; increasing commits suggest relevance and impact
Stars	Weak indication of relevance, not necessarily innovativeness
Watching	Indicator of relevance and impact, not direct innovation
Forks	Potential for innovation through combination and modification
Pull requests	Input indicator, not direct measure of innovation
Dependencies	Revealed-preference view of project's influence and use

(b) Comparison between patents, standards, and OSS repositories adapted from Blind et al (2023) [18]

	Publications	Patents	Standards	Open Source Repositories
Authors	Co-authors	Inventors and applicants	Not available (exceptions)	Contributors
Contributions	Not available (exceptions)	Not available (exceptions)	Not available (exceptions)	Commits or dependencies
Inputs to determine innovation	Scientific papers, data	Patents, non-patent literature, scientific papers, standards	Standards, scientific papers, patents (incl. SEP)	Existing code in repositories
References to outputs	Publications, patents, standards, OSS, products, start-ups	Patents, publications, standards, OSS, products, start-ups	Standards, publications, patents, products, start-ups	Other repositories, publications, patents, standards, products, start-ups

The implementation of innovations in practice is a critical aspect of their impact and success. While scientific publications may not be immediately implemented, they can indirectly influence practice through citations in patents [47] or standards [40]. For patents, recent research has begun to establish explicit links to specific products [48]. Standards implementation has been analyzed through case studies, such as Biddle et al.'s [49] examination of standards in laptops, and more recently through large-scale web mining approaches [42].

Given the wealth of data available on developer platforms such as GitHub, which includes real-time metrics on repository usage and collaboration patterns, integrating indicators derived from these platforms with established innovation metrics can establish a robust framework for measuring OSS innovation.

2.3 Prior approaches to measuring OSS innovation

The measurement of OSS innovation builds upon diverse methodological traditions and data sources. Early work focused on adapting traditional innovation metrics to software contexts. Edison et al. (2013) [50] developed a conceptual framework incorporating patents [51] and other indicators to evaluate innovation determinants, inputs, outputs, and performance in the software industry. Similarly, Nirjar (2008) [52] proposed an "index of innovativeness" for software SMEs, linking innovation capability to growth stages.

Recent studies have leveraged repository-level metrics to measure OSS innovation at the project or repository level [7, 53, 54, 6]. Fang et al. (2024) [55] examine atypical recombinations of software libraries, finding that higher levels of innovativeness correlate with higher GitHub star counts, suggesting a link between innovation and popularity. However, they also note that more innovative projects often involve smaller teams and face higher abandonment risks, highlighting the complex relationship between innovation and sustainability in OSS.

Network analysis and user adoption patterns offer complementary approaches to measuring OSS innovation. Banks et al. [56] demonstrate a positive correlation between the centrality of Python packages in dependency networks and download statistics. In addition, download data provide signals of user adoption and growth patterns. For instance, Wiggins et al. [57] propose a method to measure the size of an OSS project's user base and the level of potential user

interest that it generates based on daily download patterns, finding that active users rapidly download software after new releases, while potential new users download software irrespective of releases. However, adoption patterns vary across ecosystems and often exhibit significant lags, typically ranging from 3 months to a year [58, 59, 60, 61].

GitHub offers rich data for OSS innovation measurement. The platform provides various indicators for the performance and popularity of OSS repositories, including commits, stars [62], watching [63], forks [64], and pull requests [65]. Project metrics, in particular repository forks, have been shown to correlate with usage [66], offering a proxy variable for scarce usage and adoption data. The GitHub dependency graphs illustrates the projects a repository depends on and those that depend on it, offering a network view of a project's influence and reliance in the broader software ecosystem [56]. **Table 2a** discusses the merits and limitations of these internal indicators, and **Table 2b** shows key variables for indicators from OSS repositories, patents, and standards.

3 Methods & Data

This section outlines the methodology and data sources utilized in our study. § 3.1 provides a glossary of key terms essential to understanding the technical aspects of software dependencies, releases, and versioning. § 3.2 details the construction of our GitHub repository dataset and dependency analysis. § 3.3 describes our approach to analyzing the complexity of releases using LLM annotation.

3.1 Glossary of Key Terms

This study utilizes core concepts from OSS development.

Dependencies and Dependents   A *dependent* is a software project that relies on other software modules (known as *dependencies*). While software projects often form complex networks of dependencies, our analysis focuses solely on immediate, first-order relationships between dependents and dependencies.

Software Releases   A *release* represents the official distribution of a software version, typically comprising:

- A packaged version of the project's code
- Release notes documenting changes and new features
- Installation and integration instructions

Semantic Versioning   Modern software releases follow a three-part numerical scheme [11]:

- *Major version* (first number): Significant changes that may break backward compatibility
- *Minor version* (second number): New features that maintain compatibility
- *Patch version* (third number): Bug fixes and incremental improvements

Early-Stage Projects   Projects in early development phases often use a special versioning approach:

- *Zero-major* releases: When the major version remains at 0 during prototyping
- *Zero-minor* releases: Minor and patch changes during early development
- These early-stage releases may see different adoption patterns compared to mature projects

3.2 Analysis of GitHub Repository Data and Dependencies

3.2.1 Data Generation

We constructed our dataset from internal GitHub databases. Most of the data, such as repository stars, forks, and programming languages, can be accessed via the GitHub API. Additionally, dependent data can be sourced from third-party packages, such as `https://github.com/nvuillam/github-dependents-info` or from services such as `https://ecosyste.ms/`. However, results from these third-party options may vary in dependent counts due to differing counting criteria and the absence of look-ahead or look-behind metrics, which are essential for comprehensive analysis reproduction. While public sources support future use of our method, to our knowledge, no third-party has attempted a comprehensive collection of daily snapshots of repository metadata and dependency graph information for the time period considered in our study. Our dataset, available at `https://github.com/evamaxfield/dg-uoi-analysis`, is the combination of three datasets of innovation-related activities on the GitHub platform.

The first dataset includes *repository information*, which contains details such as the repository's description, topics, number of stars, and number of forks. This data provides insight into the popularity and engagement levels of various repositories (see **Table 2a**).

The second dataset focuses on *package releases*. This data provides key information about the versions of packages that are being updated, with each entry including a repository identifier alongside release notes describing changes for each version. For instance, an entry includes the account name, the repository name, and the release version of a Python package with a specific name, along with the corresponding date. For example, "my-account/my-repo" refers to the account and repository name, while "v1.2.3" indicates the package version, and the release date is September 25, 2024). This information is detected and logged daily through GitHub's automatic parsing of standard packaging files, such as pyproject.tom and setup.py for Python packages, and package.json for JavaScript packages.

The third dataset focuses on *dependents of a package*, which identifies the down-stream repositories that rely on specific packages along with their corresponding dependency versions. Each entry in this dataset includes a repository identifier for the dependent repository, the name of the package it depends on, and the version of that dependency. For instance, an entry indicates that "another-account/another-repo" depends on a Python package named "zyx," with the version "v1.2.2," as of September 25, 2024. Similarly, this information is detected and logged each day via GitHub's automatic parsing of standard packaging files (see above).

We combine and filter these datasets to create a comprehensive view of package releases and their dependencies. First, for each day of newly detected package releases, we join each package release with the corresponding repository information. Next, we implement a filtering mechanism to enhance data quality. Specifically, we remove package releases originating from repositories that do not have at least one star or are classified as "forked repositories". This step is crucial to eliminate packages that may be either extremely new, lacking community support, have not undergone sufficient development, or may be low-quality or automatically generated with little meaningful content (e.g. spam). This ensures that our dataset reflects more established and relevant packages, focusing on those with active development and community engagement. We apply another filter to exclude package releases that do not adhere to the *semantic versioning specification* (see § 3.1 above). During this process, we decompose the version string and categorize releases accordingly to major, minor, and patch. This classification allows us to systematically analyze the frequency and impact of updates based on the significance of the changes.

Next, we filter to ensure that only package releases from repositories whose names align with their respective package names are included in our dataset. Identifying the primary package for a repository is crucial because it simplifies the understanding of package dependencies and updates. For instance, the repository `https://github.com/pandas-dev/pandas` contains the source code for the widely used Python package "pandas," and thus, its package releases are included in our dataset. Conversely, if the repository were named "pandas-dev/pandas-subpkgs," where the repository name ("pandas-subpkgs") does not match the package name ("pandas"), those releases would be excluded. This filtering criterion helps avoid confusion that may arise when multiple packages are housed within a single repository, making it difficult to track dependencies accurately and understand which package is being referred to in analyses. By applying this criterion, we aim to focus on the packages that are directly associated with their respective repositories, thereby enhancing the quality and reliability of our dataset.

Lastly, we enrich our dataset by calculating several measures. First we integrate look-ahead metric information regarding repository stars and forks. This involves systematically retrieving and compiling repository data for each specified look-ahead day interval per repository. Specifically, for each repository, we collect data on the number of stars and forks over the defined intervals, which serves as an indicator of community engagement and interest. Second, we integrate the number of package dependents. This is accomplished by counting the instances of non-forked repositories that have received at least one star and include a dependency on a package sharing the same name, within the same ecosystem (e.g. Python, JavaScript), and aligned with the same major version number.

3.2.2 Data Processing

Our dataset, compiled from dependency graph package release detection (see § 3.2.3, in total includes 862,653 software releases from 148,628 unique packages from multiple different software ecosystems on GitHub. Among these, the most common ecosystems are JavaScript (NPM), Python (PyPI), and Ruby (RubyGems).

Following dataset compilation, we implemented a series of necessary data processing [2] steps to prepare the data for analysis. First, we removed any packages that had multiple releases on the same day. While it is common for packages to have multiple releases over longer time horizons, excluding same-day versions minimizes the risk of applying the

---

[2]All of these pre-processing activities were executed using the Polars library in Python [67], which facilitated efficient manipulation of large datasets and streamlined the data processing workflow

same set of metrics to different releases.[3] Second, we excluded packages not belonging to the JavaScript, Python, and Ruby ecosystems, as these languages consistently accounted for at least 5,000 package releases across all datasets (spanning six-month, one-year, and two-year intervals). Third, we also excluded any packages that did not have at least one dependent prior to the release. This criterion ensures that only actively utilized packages are included in our analysis, enhancing the focus on software that has demonstrated relevance within the ecosystem.

Following these steps, our filtered dataset 202,014 unique releases from 28,044 unique packages from the JavaScript (NPM), Python (PyPI), and Ruby (RubyGems) ecosystems.[4]

Subsequently, we stratified the dataset based on package popularity, measured by existing dependents. We categorized the packages into three bins: those in the less than 25th percentile, those between the 25th and 75th percentiles, and those in the greater than 75th percentile of package dependents prior to release. Additionally, we classified the packages by release series based on their versioning: zero-version, one-version, and two-plus-version (i.e., 0., 1., and 2+.*). This dual stratification reflects both the popularity and maturity of the packages.

Finally, we calculated the log differences between each metric at designated look-ahead points and the corresponding release date. For instance, to compute the 90-day log difference in dependents following a package release, we subtracted the natural logarithm of the dependents on the release date from the natural logarithm of the dependents at 90 days post-release. This log difference calculation was performed for each metric (stars, forks, and dependents) across all specified look-ahead intervals: six months (with measurements every 45 days), one year (every 90 days), and two years (every 180 days).

3.2.3 Data Analysis

Our data analysis focused on examining the characteristics of the generated dataset and assessing the impact of software releases on subsequent software utilization. The following steps outline our approach.

We began by analyzing the demographics of the dataset, aiming to understand its composition in terms of both programming ecosystems (i.e., JavaScript, Python, and Ruby) and the types of software releases. This demographic breakdown provided essential context for subsequent analyses by illuminating the distribution of releases across ecosystems and their respective release categories. When focusing on the NPM, PyPI, and RubyGems, we observe that a majority of our data comes from NPM (JavaScript) package releases. However, each ecosystem exhibits similar proportions for each release type which include major, minor, zero-major, and zero-minor releases (see **Figure 1**).

To explore software releases' effect on future utilization, we calculated log-differences in dependents from release date to one year after release. We select a one-year period because it is typically sufficient for users to integrate new features, adapt to changes, and evaluate implications of a release on their project [58, 59, 60, 61]. These log-differences are a measure of how much the software's dependency network expanded during this period. We categorized these measurements by release type—major, minor, and patch—and conducted an Analysis of Variance (ANOVA) to determine whether there were statistically significant differences between groups. We also performed post-hoc pairwise Welch's independent t-tests to identify which specific release types exhibited significantly different distributions in the one-year change in dependents. Both the ANOVA and t-tests were executed using the SciPy library [68].

Next, we compared log-differences in dependents for prevalent ecosystems in our dataset: JavaScript, Python, and Ruby. This comparison allowed us to observe variations in software utilization across these dominant programming ecosystems for understanding ecosystem-specific behaviors. Further stratification of the dataset was conducted to investigate how dependent growth changes when controlling for version series (i.e., zero-version, one-version, and two-plus-version) and initial software utilization. For this, we divided the initial number of dependents into three groups: bottom-25th percentile, middle-50th percentile, and top-25th percentile. By employing these subgroup analyses, we aimed to reveal how both software maturity and early-stage adoption influence long-term dependency growth. Data visualizations were generated using seaborn [69], matplotlib [70], opinionated [71], and colormaps [72].

---

[3]This scenario typically arises in two contexts: (i) a new major or minor release of a package which may be followed by a patch release to rectify bugs inadvertently introduced with new features, and (ii) mature software projects may maintain multiple versions of the same software, such as a "Long-Term Support" (LTS) version alongside a leading-edge version.

[4]Release and package counts are based on the produced one-year look-ahead metric dataset. Different look-ahead time-span datasets resulted in different counts however we focused on the final counts for the one-year look-ahead metrics because our analysis also focuses on the same portion of the data.
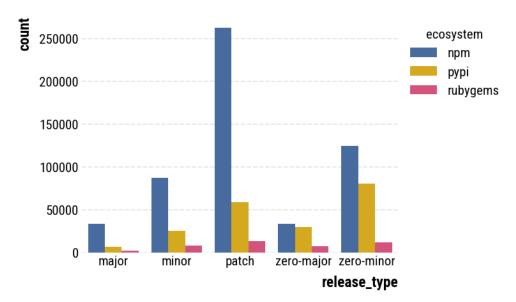
Figure 1: Release Counts by Release Type (Major, Minor, Patch, Zero-Major, and Zero-Minor and Ecosystem (NPM: JavaScript, PyPI: Python, RubyGems: Ruby)

3.3 Analysis of GitHub Releases and LLM Annotation of Complexity

3.3.1 Data Generation

For the analysis of GitHub releases and complexity, we closely follow the methodology established in § 3.2. We combine multiple private daily snapshot datasets encompassing repository information, package releases, and dependency details, ensuring consistency through uniform filtering and processing techniques. We perform two more steps to enrich our dataset. First, we focus on package releases that include an accompanying GitHub release, which allows us to capture the associated release notes. Second, we implement a stringent filtering criterion, removing releases lacking a minimum of 512 text characters in their release notes. This requirement ensures that the releases included in our dataset contain sufficient textual information, enabling both our prompted LLM and domain experts to effectively assess the complexity of each release. These notes potentially provide critical context and insights into the changes made in each release.

Aside from these additional steps, the fundamental data generation process remains unchanged from that described in § 3.2, allowing for a seamless integration of our augmented data approach and maintaining methodological rigor.

3.3.2 Data Processing

Our data processing methodology involved a novel and structured approach to assess the complexity of GitHub releases. After compiling a dataset of release notes and repository metadata, we moved to the task of rating the complexity of each release to better understand the effort required to implement the listed changes.

To achieve this, we employed a prompted GPT-4 model, which was iteratively refined to ensure accurate and consistent complexity ratings. The prompt, detailed in Appendix A, adhered to best practices in prompt engineering, incorporating clear task definitions, example input-output pairs, and a defined structure for input and response [73]. The model's responses were formatted in XML, facilitating structured data extraction for subsequent analysis. This ensured consistency in how the model evaluated the complexity of each GitHub Release.

The complexity ratings were designed to reflect the difficulty a core developer of the project might face in implementing the changes described in the release notes. We defined complexity on a 7-point scale, ranging from 1 (minimally complex, requiring minimal changes and low expertise) to 7 (maximally complex, involving substantial changes and significant expertise). This scale allowed us to capture variations in both the number of changes and the depth of knowledge required to implement them.

Given that this complexity-scoring approach is novel, we thoroughly examined its robustness. To validate the model's output, we compared its ratings with those provided by professional developers. A JavaScript developer evaluated the complexity of releases from JavaScript projects, while a Python developer did the same for Python projects. This

9

Table 3: Descriptive Statistics for Release Notes Complexity Ratings Across each Programming Language

Prog. Lang.	# Releases	Mean	StD	25%	50%	75%
Python	12157	4.52	1.16	4	5	5
JavaScript	11288	3.90	1.28	3	4	5
Ruby	2605	4.27	1.08	4	4	5

human annotation process served as a benchmark to assess the accuracy of the GPT-4 model's ratings. Our analysis revealed a positive correlation between the model and the developers' assessments, with a correlation coefficient of α = 0.39 for JavaScript (p ¡ 0.00001) and α = 0.36 for Python (p ¡ 0.001). Notably, more than 60% of the GPT-4 and developer ratings were within one complexity rank of each other (JavaScript: 68.5%, Python: 63.3%), indicating a strong degree of alignment between the model and human experts.

However, a more detailed analysis of the complexity ratings distributions shows a divergence between the GPT-4 model and the professional developers, with the GPT-4 model consistently assigning higher complexity scores. GPT-4 consistently assigns a score of 4 or higher to releases that professionals rate as 3, especially at the 25th and 50th percentiles of complexity, overestimating the complexity of releases that professionals consider less challenging. This discrepancy may stem from the model's sensitivity to detailed technical language or broader interpretation of the task's complexity, leading to a higher baseline rating for changes it encounters.

Despite this upward bias, the model demonstrated a strong degree of agreement with the professional developers' ratings within one complexity rank, particularly at the median and higher complexity levels. Given this high level of concordance, we proceeded to use the finalized GPT-4 prompt to generate complexity ratings for the entire dataset of GitHub Releases and their corresponding release notes. This approach allowed us to efficiently scale the complexity assessment process while maintaining reasonable accuracy and alignment with expert human judgment.

3.3.3 Data Analysis

Building upon the methodology described in § 3.2.3, we employed a structured approach to investigate the characteristics and impact of complexity ratings across different programming ecosystems.

We began by examining the descriptive statistics of the dataset, specifically focusing on the distributions of complexity ratings across different release types (i.e., major, minor, and patch) and ecosystems (JavaScript, Python, and Ruby). This initial step helped establish a comparative understanding of how complexity ratings varied by release type and ecosystem. **Table 3** summarizes the descriptive statistics of the dataset.

To evaluate differences in the distributions of complexity ratings between ecosystems, we conducted an ANOVA test. We then performed post-hoc pairwise Welch's independent t-tests to identify where significant differences occurred between specific ecosystems. Finally, we sought to understand how complexity ratings were related to the subsequent software utilization by calculating the Spearman correlation (ρ) between the complexity rating of each release and the one-year log-difference in dependents. This analysis was performed separately for each major ecosystem in our dataset. The goal was to assess whether more complex releases tend to result in greater or lesser growth in the number of dependents over time, providing insight into how release complexity may influence the broader software ecosystem's uptake and reliance on the software in question. ionResults In this section, we present the results of our analysis. We begin by reporting the results about dependency measures in § 3.4. Subsequently, we report the results on release complexity as annotated by LLMs in § 3.5.

3.4 Repository metrics and package dependencies

3.4.1 Major releases associated with significant increases in package dependencies

Key finding: *At the one-year post-release mark, major releases are associated with significantly larger increases in package dependencies compared to minor or patch releases across the JavaScript, Python, and Ruby ecosystems, highlighting distinct patterns of dependency growth that emerge following significant package updates.*

We find that packages experience a statistically significant positive change in their log-difference in direct dependents following a Major release compared to other release types (see **Figure 2**). **Table 4** provides a detailed overview of the log-difference in direct dependents associated with different release types across various programming languages. Specifically, we observe that JavaScript major releases are associated with an average increase of 2.32 in log-difference in dependents, while minor and patch releases show comparatively modest increases of only +0.15 and +0.16, respectively. In the Python ecosystem, major releases correspond to an average increase of 1.79 in log-difference in
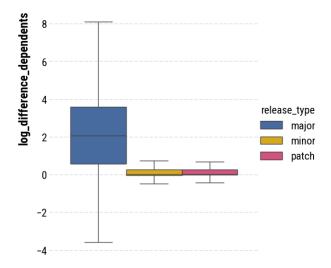
Figure 2: Log-Difference in Dependents after One-Year Post Software Release Split by Software Release Type

Table 4: Log-Difference in Dependents after One-Year Post Software Released Split by Language

Average Increase in Log-Difference in Dependents	JavaScript	Python	Ruby
Major release	2.32	1.79	2.43
Minor release	0.15	0.11	0.02
Patch release	0.16	0.08	0.08

dependents, in stark contrast to the minimal increases of +0.11 for minor releases and +0.08 for patch releases. Similarly, Ruby major releases demonstrate a strong impact, linked to an average increase of 2.43 in log-difference in dependents, whereas minor and patch releases exhibit negligible increases of +0.02 and +0.08, respectively.

Overall, in this one-year interval, these results indicate that major releases significantly enhance the log-difference in direct dependents across all three ecosystems, with JavaScript exhibiting the most pronounced effect. This underscores the critical role of major updates in driving package adoption and shaping dependency growth patterns in the short-term.

3.4.2 Consistent dependency growth patterns across ecosystems and time frames

Key finding: Major releases generate larger positive changes in log-difference compared to minor and patch releases, regardless of the package ecosystem, initial package utilization, or time frame examined up to two years post-release. However, the magnitude of the log-difference tends to diminish slightly in the second year.

Next, we examine log-difference dynamics in direct package dependencies over a two year time frame. We observe that the trend of major releases maintaining a strong positive impact on dependency growth persists across all ecosystems during this period (see **Figure 3**). However, it is important to note that the magnitude of the log-difference tends to diminish slightly in the second year, in particular for the Ruby ecosystem, where we observe a strong decline in log-difference for major releases by the third year, resulting in significantly lower values compared to the other ecosystems. This suggests a potential saturation effect or a shift in package utilization patterns specific to Ruby. The Python ecosystem also experiences a reduction in log-difference over time, indicating a gradual decrease in the impact of major releases. Meanwhile, the JavaScript ecosystem demonstrates remarkable stability, with log-difference values remaining relatively consistent, highlighting its resilience in maintaining dependency growth from major releases.
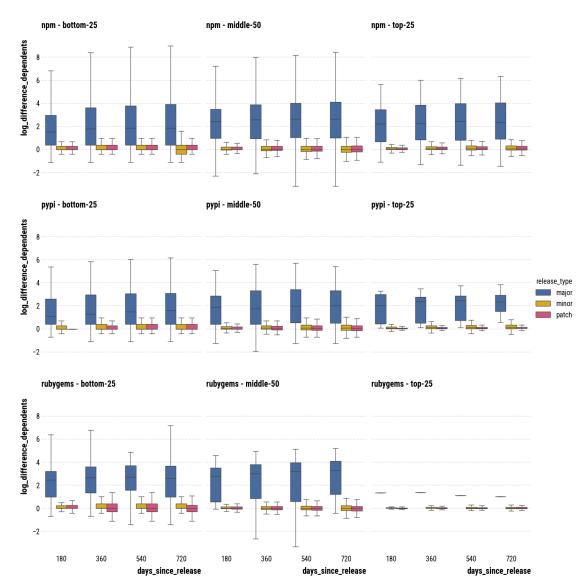
Figure 3: Distributions of Log-Difference in Dependents after 180, 360, 540, and 720 days since Software Release. Each row of plots are all the same ecosystem (JavaScript, Python, Ruby), while each column of plots are all the same initial package utilization bins (bottom 25th percentile, middle 50th percentile, and top 25th percentile).

Table 5: Programming Languages, Release Types, and Complexity Rating Comparisons

Prog. Lang.	Release Type 1	Release Type 2	Release Type 1 Mean	Release Type 2 Mean	pvalue
JavaScript	major	minor	4.83 (±1.07)	4.13 (±1.03)	¡ 0.0001
		patch	4.83 (±1.07)	3.38 (±1.18)	¡ 0.0001
	minor	patch	4.13 (±1.03)	3.38 (±1.18)	¡ 0.0001
Python	major	minor	5.19 (±1.23)	4.76 (±1.0)	¡ 0.01
		patch	5.19 (±1.23)	4.16 (±1.14)	¡ 0.0001
	minor	patch	4.76 (±1.0)	4.16 (±1.14)	¡ 0.0001
Ruby	major	minor	5.52 (±1.02)	4.54 (±0.87)	¡ 0.0001
		patch	5.52 (±1.02)	3.93 (±1.03)	¡ 0.0001
	minor	patch	4.54 (±0.87)	3.93 (±1.03)	¡ 0.0001

3.5 LLM Annotations of Release Complexity

3.5.1 Correlation between release complexity and dependency growth

Key finding: *Release complexity, as rated by a prompted GPT-4 model, correlates slightly positively with dependency growth, particularly for JavaScript and Python packages. However, the Ruby ecosystem does not exhibit the same correlation, suggesting different dynamics in the Ruby community and a different handling of release notes.*

We observe statistically significant differences in complexity ratings across the three main release types: major, minor, and patch. Major releases consistently show the highest complexity, as they typically introduce significant changes or breaking features. Patch releases, on the other hand, exhibit the lowest complexity ratings, reflecting their focus on bug fixes or minor adjustments. Minor releases lie between these two. This pattern holds consistent across all three ecosystems—JavaScript, Python, and Ruby. As depicted in **Figure 4a**, major releases display the highest complexity ratings, followed by minor and then patch releases, which is according to our expectations and serves as a robustness check of our analysis. The differences in complexity between these release types are statistically significant and can be attributed to the inherent nature of the changes introduced in each type. See **Table 5** for a summary of the statistical analysis, including means, standard deviations, and p-values for significance testing across ecosystems.

Next, we explore the relationship between release complexity and package dependency growth. The JavaScript and Python ecosystems hold statistically significant, slightly positive correlations between the complexity of releases and the log-difference in dependents after one year (see **Figure 4b**). Specifically, for JavaScript package releases, we observe a Spearman's correlation coefficient of 0.13 ($p \leq 0.00001$), suggesting that more complex releases tend to have marginally higher growth in dependent packages over time. Similarly, Python package releases show a Spearman's correlation coefficient of 0.08 ($p \leq 0.0001$), reflecting a similar trend, albeit weaker than in JavaScript. Curiously, the Ruby ecosystem does not exhibit the same statistical significance between release complexity and dependency growth. This points to different dynamics in the Ruby community and a different handling of release notes, leading to a divergence in how complexity is documented and perceived. It is also possible that other factors, such as community practices or the characteristics of dependencies, play a more significant role in influencing package adoption in Ruby.
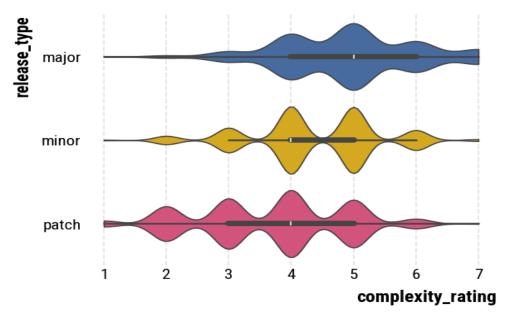
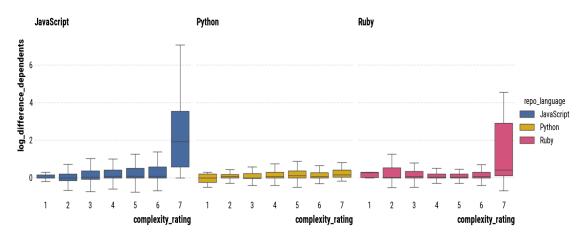## 4 Discussion

4.1 Theoretical Implications

4.1.1 Making visible OSS innovation

Our proposed measure of software innovation reduces a gap in the literature on innovation that has largely overlooked OSS outputs. Semantic versioning and the pronounced difference in downstream adoption of major versions compared to minor and patches may illustrate the concept of incremental versus radical innovation in a software context. This distinction is particularly important in the case of OSS, where contributions can range from small, incremental improvements to large, transformative updates.

Additionally, in many emerging countries, measuring innovation is challenging due to limited data [74] or incentives to patent [20]. At the same time, OSS innovation holds significant potential in these regions due to its cost-effectiveness and accessibility, which can drive technological advancement without expensive proprietary solutions. Since traditional innovation metrics fall short in capturing the potential of OSS in these regions, our work represents an initial

(a) Complexity Rating Distributions by Release Type (Major, Minor, or Patch)

(b) Complexity Ratings of Release Notes Correlate with Log-Difference of Dependents After One Year

Figure 4: Complexity Ratings of Release Notes

step toward developing new measurement methods, paving the way for further research that could enable country-level analyses and better assess OSS innovation, especially in areas where conventional metrics are inadequate.

4.1.2 Complementing innovation indicators with granular, cost-effective, and real-time innovation measurement

Established innovation indicators (discussed in § 2) face significant limitations in various contexts, particularly regarding timeliness and granularity. The Oslo Manual [2] highlights the challenge of measuring radical innovations due to delays in their effects, which can obscure the real-time dynamics of innovation. In contrast, our proposed measure enables more immediate evaluations of innovation–whether radical or incremental–even if allowing for a year to validate trends in dependents. Unlike patent applications, which are published only after an 18-month period from the earliest filing date, our methodology facilitates the analysis of how innovations are utilized, providing citation visibility akin to academic publications and practical implications similar to patents. Focusing on software innovation, our approach can also address insufficient sectoral and technological granularity of traditional metrics (e.g. with a focus on the ICT sector). Moreover, conventional indicators often incur high data collection costs, especially at scale. By leveraging accessible data from developer platforms such as GitHub, our methodology offers scalable access to data without substantial financial investment. Additionally, our approach employs established criteria for assessing software innovation, with only few qualitative aspects in comparison to e.g. survey-based innovation indicators. These qualitative aspects are primarily related to the documentation of release notes and the rigor of semantic versioning applied by contributors.

4.1.3 Transparency and visibility of individual contributions in the innovation ecosystem

Our measure of software innovation enables transparency in contributions: specific lines of code within an innovative package can be traced back to their creators. While not explored at length in this paper, this visibility could support further work to evaluate individual contributions to innovation over time and their relation to trends in academic publishing and other innovative fields. Building on this, social network analysis (SNA) presents an important avenue for future research in understanding the dynamics of OSS innovation ecosystems. Widely applied in innovation economics, SNA can offer a more in-depth exploration of how knowledge and innovations circulate within networks [75] As noted in the literature review, approaches to network analysis of OSS networks have already been undertaken (see, e.g. [56] or [57]). However, by incorporating dependency data and differentiating inventive steps through semantic versioning, SNA could provide a more granular map of interactions between contributors in OSS ecosystems and contribute to our understanding of how knowledge and innovation circulate within OSS projects. For example, this approach could reveal how collaborative efforts shape innovation, identifying key nodes or bottlenecks and informing policies aimed at fostering more effective knowledge exchange and innovation flow.

4.1.4 Potential self-fulfilling prophecy in release labeling and innovation adoption

Our measure may demonstrate a self-fulfilling prophecy that reflects the marketing of innovation. We find that the type of release is the strongest predictor of downstream adoption, and while it is (weakly) positively correlated with release complexity as assessed by release notes, developers may only pay attention to major releases due to their label or public sharing of a release in channels not observed in the GitHub data.This creates a risk of endogeneity: when a developer claims their new release is a major one, this assertion may spur downstream adoption based more on the label than the actual substance of the release.

This phenomenon parallels challenges seen in other innovation metrics, such as those for patents or R&D expenditures. In the patent landscape, the sheer number of filings does not inherently correlate with the quality or impact of the innovations. A high volume of patents might mask a lack of substantive progress, while truly innovative ideas could be underrepresented if they are not effectively communicated or marketed [76]. Similarly, with R&D expenditures, what is labeled as R&D can encompass a wide range of activities that do not necessarily lead to innovation; for instance, a significant portion of R&D spending might be directed towards marketing or administrative tasks rather than genuine innovative processes. This variability complicates the standardization of R&D as a measure of innovation [76].

In the context of our research, the label of a "major release" could similarly drive adoption regardless of the technical merit or innovative substance of the release. This highlights the importance of considering how the communication surrounding releases can shape perceptions and behaviors in the OSS community. The influence of promotional language and activity may significantly affect how innovation is recognized and valued, leading to the potential misrepresentation of the innovation landscape. Further research on this possible phenomenon with off-platform data would support related research tracking promotional language and activity associated with academic research [77, 78].

4.1.5   Supporting comparisons of software language innovation ecosystems

Finally the measure of software innovation can be used to create a baseline by which adoption is expected for a major, minor, or patch version in a particular language. This could support research on language evolution and comparative analysis between programming language ecosystems, building on our observed variation between JavaScript, Python, and Ruby in growth of dependents two years after a release. Within a particular language, the expected baseline can also be used to identify especially innovative or impact releases early on, based on their divergent growth trajectory. Future work could describe this innovative subset and compare such a measure of 'innovativeness' to existing approaches like package citation combination novelty [55].

4.2   Practical Implications

4.2.1   Informing national measurement and policy

Beyond the innovation literature, our measure of software innovation has practical implications for policymakers, practitioners, and researchers. The measure may be aggregated by jurisdiction, as has been done to-date with software development activities in the GitHub Innovation Graph and inform international comparisons of innovation [79]. Governments may look to the measure as one data point to inform the evaluation of national industrial strategies, technology-specific, e.g. AI, strategies, and other initiatives that aim to spur innovation. For practitioners in the private sector, our approach can support the evaluation of investment decisions and outcomes. By analyzing software innovation metrics, organizations can gain insights into market trends, competitive positioning, and potential return on investment. This can enable more informed decision-making regarding research and development initiatives, partnerships, and resource allocation.

4.2.2   Refining OSS innovation metrics

Researchers and practitioners focusing on OSS can leverage the proposed measure of software innovation to advance their understanding of how innovation unfolds within different OSS ecosystems. Previous literature [7] measuring the impact of OSS has primarily relied on commits as proxies for OSS activity, correlating these with broader economic and social outcomes. However, commits may not adequately capture innovation since they do not distinguish between incremental updates and substantial, innovative changes.

For future research, we may consider the ratio of dependents to dependencies as a measure of a repository's innovativeness, similar to the forward-to-backward citation ratio used in bibliometric and patent analysis [80]. This approach aligns with the concept of revealed preference, suggesting that when developers choose to depend on a project rather than writing code from scratch, they signal that the dependency represents a non-trivial inventive step. Our preliminary exploration of the top 1,000 GitHub repositories, ranked by stars, forks, and issue authors, has shown a positive correlation between the dependent-dependency ratio and the number of dependents. This suggests that repositories with a high ratio of dependents to dependencies can be considered more innovative, as they exert a greater influence on the OSS ecosystem relative to their reliance on other projects. Using major version releases as output units of innovation provides a clearer view of substantial changes in software development that could potentially have wider economic and technological impacts. While directly linking these releases to specific economic outcomes—such as increased demand for technical services, job creation, or shifts in industry productivity—remains subject to further research, this measure may serve as an early indicator of such impacts. For example, downstream adoption patterns following a major release could signal broader trends in the uptake of new technologies or the influence of OSS on industry innovation.

However, in this work, we chose not to rely on dependency data as the primary unit of innovation, as it appeared less useful for comparing individual releases within a project. Outside of major releases, projects rarely change their dependencies significantly, making this ratio less informative for release-to-release comparisons. Nonetheless, project-to-project comparisons using the dependent-dependency ratio remain a promising avenue for identifying innovative software. Future studies should further explore these methods to connect the proposed OSS innovation metrics to broader economic or sectoral trends.

While our measure of OSS innovation is primarily designed to track innovation within the software development ecosystem itself, future research could investigate potential links to broader outcomes, such as industry adoption rates or collaborative dynamics across programming communities. In this sense, the measure provides a foundation for more granular analysis of OSS innovation, without yet venturing into economic impact analysis.

4.3 Limitations

This work faces the following limitations. First, we analyze only direct dependents of a given package. However, this undervalues packages that have become central to software ecosystems, where their direct dependencies are in turn relied upon widely. Future work could seek to remedy this limitation by analyzing indirect dependencies, albeit at significantly higher computational costs.

Second, our analysis does not account for the role of software licenses in shaping innovation patterns. Different license types (e.g. GPL vs. Apache v2) may systematically influence both developer choices and dependency relationships, as licenses can either enable or constrain the ways packages build upon each other. Furthermore, we do not address the geographic dimension of software innovation diffusion, which could reveal important regional variations in OSS adoption and development patterns.

Third, mature software packages often have their own platforms for communicating and distributing releases that are not captured in GitHub data. For example, some version releases on GitHub may use release notes that link to further detail off-platform. Thus, our method may not accurately account for the complexity of mature packages.

Fourth, and relatedly, our analysis of release complexity examines the notes describing the release only. Further analysis could examine the code associated with each release directly as another, perhaps more direct, indicator of complexity. Fourth, we examine only three software languages. Future work could expand the language coverage of the measure. However, some languages, such as C and C++, do not have centralized package ecosystems or rely on GitHub's Releases feature, which undermines the universal application of our measure.

Finally, while our use of LLMs to assess semantic content offers novel analytical possibilities, this approach has inherent limitations. Recent research suggests that LLM responses can vary significantly even with identical prompts [81, 82]. Future work could enhance reliability by implementing multiple prompt iterations and employing modal or mean responses as final classifications, alongside expanded human validation procedures.

5 Conclusion

This paper has introduced a novel method for measuring innovation in software, complementing traditional indicators such as scientific publications, patents, and standards. By analyzing GitHub data on package releases and dependency adoption, we make three empirical contributions. First, we observed that major releases are associated with significantly larger increases in package dependencies compared to minor or patch releases across JavaScript, Python, and Ruby ecosystems. Second, this trend holds true regardless of package ecosystem, initial package utilization, or time frame examined, up to two years post-release. Third, we found that release complexity, as rated by a prompted GPT-4 model, correlates slightly positively with dependency growth, particularly for JavaScript and Python packages. These findings offer new insights into OSS innovation dynamics and raise important questions for future research and discussion.

References

[1] M. Dziallas and K. Blind, "Innovation indicators throughout the innovation process: An extensive literature analysis," *Technovation*, vol. 80, pp. 3–29, 2019.

[2] OECD and Eurostat, *Oslo Manual 2018*. 2018.

[3] K. Blind, B. Krieger, and M. Pellens, "The interplay between product innovation, publishing, patenting and developing standards," *Research Policy*, vol. 51, no. 7, p. 104556, 2022.

[4] M. Böhm and D. Eisape, "Standard setting organizations and open source communities: Partners or competitors?," *First Monday*, vol. 26, June 2021.

[5] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in github: transparency and collaboration in an open software repository," in *Proceedings of the ACM 2012 conference on computer supported cooperative work*, pp. 1277–1286, 2012.

[6] C. Osborne, J. Ding, and H. R. Kirk, "The AI community building the future? A quantitative analysis of development activity on Hugging Face Hub," *Journal of Computational Social Science*, June 2024.

[7] K. Blind, S. Pätsch, S. Muto, M. Böhm, T. Schubert, P. Grzegorzewska, and A. Katz, *The impact of open source software and hardware on technological independence, competitiveness and innovation in the EU economy – Final study report*. Publications Office, 2021.

[8] N. L. Wright, F. Nagle, and S. Greenstein, "Open source software and global entrepreneurship," *Research Policy*, vol. 52, no. 9, 2023.

[9] M. Hoffmann, F. Nagle, and Y. Zhou, "The value of open source software," Strategy Unit Working Paper 24-038, Harvard Business School, January 2024.

[10] G. Korkmaz, J. B. S. Calderón, B. L. Kramer, L. Guci, and C. A. Robbins, "From github to gdp: A framework for measuring open source software innovation," *Research Policy*, vol. 53, no. 3, p. 104954, 2024.

[11] T. Preston-Werner, "Semantic Versioning 2.0.0 — semver.org." `https://semver.org/`. [Accessed 14-10-2024].

[12] P. Aghion, U. Akcigit, and P. Howitt, "What do we learn from schumpeterian growth theory?," *Handbook of Economic Growth*, vol. 2, pp. 515–563, 2014.

[13] H. Evanschitzky, M. Eisend, R. J. Calantone, and Y. Jiang, "Success factors of product innovation: An updated meta-analysis," *Journal of product innovation management*, vol. 29, pp. 21–37, 2012.

[14] V. Dewangan and M. Godse, "Towards a holistic enterprise innovation performance measurement system," *Technovation*, vol. 34, no. 9, pp. 536–545, 2014.

[15] U. Cantner and S. Vannuccini, "Elements of a Schumpeterian catalytic research and innovation policy," *Industrial and Corporate Change*, vol. 27, pp. 833–850, 08 2018.

[16] N. Bloom, J. Van Reenen, and H. Williams, "A toolkit of policies to promote innovation," *Journal of economic perspectives*, vol. 33, no. 3, pp. 163–184, 2019.

[17] P. Markianidou, V. Peter, P. Shauchuk, H. Bernard, K. Blind, and P. Neuhäusler, "Operational guide on the quantification of innovation impacts," tech. rep., Technopolis Group and Fraunhofer-Institut für System- und Innovationsforschung ISI, 2022.

[18] K. Blind, M. Boehm, K. Xu, and D. Flick, "Open source in the context of innovation (indicators)," in *OFA Symposium 2023, The Social, Political and Economic Impacts of Open Source*, 2023.

[19] A. Kleinknecht, K. Van Montfort, and E. Brouwer, "The non-trivial choice between innovation indicators," *Economics of Innovation and New Technology*, vol. 11, no. 2, pp. 109–121, 2002.

[20] M. Böhmecke-Schwafert and C. Dörries, "Measuring innovation in mauritius' ict sector using unsupervised machine learning: A web mining and topic modeling approach," *Journal of the Knowledge Economy*, 2023.

[21] J. Kinne and J. Axenbeck, "Web mining for innovation ecosystem mapping: A framework and a large-scale pilot study," *Scientometrics*, vol. 125, no. 12, pp. 2011–2041, 2020.

[22] M. Böhmecke-Schwafert and E. García Moreno, "Exploring blockchain-based innovations for economic and sustainable development in the global south: A mixed-method approach based on web mining and topic modeling," *Technological Forecasting and Social Change*, vol. 191, p. 122446, 2023.

[23] Y.-K. Lin and L. M. Maruping, "Open source collaboration in digital entrepreneurship," *Organization Science*, vol. 33, no. 1, pp. 212–230, 2022.

[24] Synopsys, "Open Source Security and Analysis Report," tech. rep., Mountain View, CA, USA, 2023.

[25] E. von Hippel, "Innovation by user communities: Learning from open-source software," *MIT Sloan Management Review*, vol. 42, pp. 82–86, 2001.

[26] V. Zwass, "Co-Creation: Toward a Taxonomy and an Integrated Research Perspective," *International Journal of Electronic Commerce*, vol. 15, no. 1, pp. 11–48, 2010. Publisher: Routledge _eprint: https://doi.org/10.2753/JEC1086-4415150101.

[27] G. Von Krogh and S. Spaeth, "The open source software phenomenon: Characteristics that promote research," *The Journal of Strategic Information Systems*, vol. 16, no. 3, pp. 236–253, 2007.

[28] G. Von Krogh, S. Haefliger, S. Spaeth, and M. W. Wallin, "Carrots and rainbows: Motivation and social practice in open source software development," *MIS quarterly*, pp. 649–676, 2012.

[29] M. Gerosa, I. Wiese, B. Trinkenreich, G. Link, G. Robles, C. Treude, I. Steinmacher, and A. Sarma, "The shifting sands of motivation: Revisiting what drives contributors in open source," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1046–1058, IEEE, 2021.

[30] X. Cheng, Z. Zhang, Y. Yang, and Z. Yan, "Open collaboration between universities and enterprises: a case study on GitHub," *Internet Research*, vol. 30, pp. 1251–1279, Jan. 2020. Publisher: Emerald Publishing Limited.

[31] X. Li, Y. Zhang, C. Osborne, M. Zhou, Z. Jin, and H. Liu, "Systematic literature review of commercial participation in open source software," *ACM Transactions on Software Engineering and Methodology*, 2024.

[32] C. Osborne, F. Daneshyan, R. He, H. Ye, Y. Zhang, and M. Zhou, "Characterising Open Source Co-opetition in Company-hosted Open Source Software Projects: The Cases of PyTorch, TensorFlow, and Transformers," *arXiv preprint: 2410.18241*, 2024.

[33] K. Blind and T. Schubert, "Estimating the GDP effect of Open Source Software and its complementarities with R&D and patents: evidence and policy implications," *The Journal of Technology Transfer*, vol. 49, pp. 466–491, Apr. 2024.

[34] C. Osborne, "Public-private funding models in open source software development: A case study on scikit-learn," *arXiv preprint arXiv:2404.06484*, 2024.

[35] M. Böhm, "Economics of open source," in *Open Source Law, Policy and Practice* (A. Brock, ed.), Oxford: Oxford University Press, 2 ed., 2022. Online edition, Oxford Academic, accessed 6 Nov. 2024.

[36] H. F. Moed, "New developments in the use of citation analysis in research evaluation," *Archivum immunologiae et therapiae experimentalis*, vol. 57, no. 1, pp. 13–18, 2009.

[37] E. M. Brown, A. Nesbitt, L. Hébert-Dufresne, B. Veytsman, J. F. Pimentel, S. Druskat, D. Mietchen, and J. Howison, "Exploring the dependencies of the CZI mentions dataset," Oct. 2023.

[38] D. Harhoff, F. M. Scherer, and K. Vopel, "Citations, family size, opposition and the value of patent rights," *Research policy*, vol. 32, no. 8, pp. 1343–1363, 2003.

[39] R. Bekkers, A. Martinelli, and F. Tamagni, "The impact of including standards-related documentation in patent prior art: Evidence from an epo policy change," *Research Policy*, vol. 49, no. 7, p. 104007, 2020.

[40] K. Blind, "Standardization and Standards as Science and Innovation Indicators," in *Springer Handbook of Science and Technology Indicators* (W. Glänzel, H. F. Moed, U. Schmoch, and M. Thelwall, eds.), pp. 1057–1068, Cham: Springer International Publishing, 2019.

[41] K. Blind and A. Fenton, "Standard-relevant publications: evidence, processes and influencing factors," *Scientometrics*, vol. 127, pp. 577–602, Jan. 2022.

[42] M. Mirtsch, J. Kinne, and K. Blind, "Exploring the adoption of the international information security management system standard iso/iec 27001: A web mining-based analysis," *IEEE Transactions on Engineering Management*, vol. PP, pp. 1–14, 04 2020.

[43] E. Commission, "Commission recommendation (eu) 2023/498 of 1 march 2023 on a code of practice on standardisation in the european research area," 2023.

[44] P. Heß and M. Buggenhagen, "Coordination, consensus, and innovation: a knowledge network perspective on standard-setting," *Available at SSRN 4292241*, 2022.

[45] E. Autio, M. Kenney, P. Mustar, D. Siegel, and M. Wright, "Entrepreneurial innovation: The importance of context," *Research policy*, vol. 43, no. 7, pp. 1097–1108, 2014.

[46] W. J. Baumol and R. J. Strom, "Entrepreneurship and economic growth," *Strategic entrepreneurship journal*, vol. 1, no. 3-4, pp. 233–237, 2007.

[47] O. Sorenson and L. Fleming, "Science and the diffusion of knowledge." Available at SSRN, May 2001.

[48] W.-H. Uhlbach, L. Zhou, and G. De Rassenfosse, "Revealing Commercial Potential and Follow-on Inventions: Evidence from Patent Marking," *Academy of Management Proceedings*, vol. 2023, no. 1, p. 16701, 2023. _eprint: https://doi.org/10.5465/AMPROC.2023.16701abstract.

[49] B. Biddle, A. White, and S. Woods, *How many standards in a laptop?(And other empirical questions)*. IEEE, 2010.

[50] H. Edison, N. Bin Ali, and R. Torkar, "Towards innovation measurement in the software industry," *Journal of systems and software*, vol. 86, no. 5, pp. 1390–1407, 2013.

[51] Z. J. Acs, L. Anselin, and A. Varga, "Patents and innovation counts as measures of regional production of new knowledge," *Research policy*, vol. 31, no. 7, pp. 1069–1085, 2002.

[52] A. Nirjar, "Innovations and Evolution of Software SMEs: Exploring the Trajectories for Sustainable Growth," *Vision*, vol. 12, no. 2, pp. 47–59, 2008. _eprint: https://doi.org/10.1177/097226290801200205.

[53] L. Bao, X. Xia, D. Lo, and G. C. Murphy, "A large scale study of long-time contributor prediction for github projects," *IEEE Transactions on Software Engineering*, vol. 47, no. 6, pp. 1277–1298, 2019.

[54] V. Cosentino, J. L. C. Izquierdo, and J. Cabot, "A systematic mapping study of software development with github," *Ieee access*, vol. 5, pp. 7173–7192, 2017.

[55] H. Fang, J. Herbsleb, and B. Vasilescu, "Novelty begets long-term popularity, but curbs participation," 2024.

[56] D. Banks, C. Leonard, S. Narayan, N. Thompson, B. Kramer, and G. Korkmaz, "Measuring the impact of open source software innovation using network analysis on github hosted python packages," in *2022 Systems and Information Engineering Design Symposium (SIEDS)*, pp. 110–115, IEEE, 2022.

[57] A. Wiggins, J. Howison, and K. Crowston, "Heartbeat: Measuring Active User Base and Potential User Interest in FLOSS Projects," in *Open Source Ecosystems: Diverse Communities Interacting* (C. Boldyreff, K. Crowston, B. Lundell, and A. I. Wasserman, eds.), (Berlin, Heidelberg), pp. 94–104, Springer Berlin Heidelberg, 2009.

[58] A. Decan, T. Mens, and E. Constantinou, "On the evolution of technical lag in the npm package dependency network," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 404–414, IEEE, 2018.

[59] J. D. Scarsbrook, M. Utting, and R. K. L. Ko, "Typescript's evolution: An analysis of feature adoption over time," 2023.

[60] A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. Gonzalez-Barahona, "An empirical analysis of technical lag in npm package dependencies," 04 2018.

[61] J. Stringer, A. Tahir, K. Blincoe, and J. Dietrich, "Technical lag of dependencies in major package managers," 10 2020.

[62] H. Borges and M. T. Valente, "What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform," *Journal of Systems and Software*, vol. 146, pp. 112–129, 2018.

[63] J. Sheoran, K. Blincoe, E. Kalliamvakou, D. Damian, and J. Ell, "Understanding "watchers" on github," 06 2014.

[64] J. Jiang, D. Lo, J. He, X. Xia, P. S. Kochhar, and L. Zhang, "Why and how developers fork what from whom in GitHub," *Empirical Software Engineering*, vol. 22, pp. 547–578, Feb. 2017.

[65] M. A. Batoun, K. L. Yung, Y. Tian, and M. Sayagh, "An empirical study on github pull requests' reactions," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, sep 2023.

[66] S. Vargas, G. Link, and J. Lee, "Estimating usage of open source projects," in *Proceedings of the 21st International Conference on Mining Software Repositories*, pp. 652–653, 2024.

[67] R. Vink, S. d. Gooijer, A. Beedie, M. E. Gorelli, W. Guo, J. v. Zundert, O. Peters, nameexhaustion, G. Hulselmans, C. Grinstead, G. Burghoorn, Marshall, chielP, I. Turner-Trauring, M. Santamaria, L. Mitchell, D. Heres, J. Magarick, ibENPC, K. Genockey, H. Harbeck, M. Wilksch, deanm0000, J. Leitao, M. v. Gelderen, P. Barbagiannis, J. Haag, O. Borchert, and I. Koutsouris, "pola-rs/polars: Python Polars 1.6.0," Aug. 2024.

[68] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[69] M. Waskom, O. Botvinnik, D. O'Kane, P. Hobson, S. Lukauskas, D. C. Gemperline, T. Augspurger, Y. Halchenko, J. B. Cole, J. Warmenhoven, J. de Ruiter, C. Pye, S. Hoyer, J. Vanderplas, S. Villalba, G. Kunter, E. Quintero, P. Bachant, M. Martin, K. Meyer, A. Miles, Y. Ram, T. Yarkoni, M. L. Williams, C. Evans, C. Fitzgerald, Brian, C. Fonnesbeck, A. Lee, and A. Qalieh, "mwaskom/seaborn: v0.8.1 (september 2017)," Sept. 2017.

[70] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in science & engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[71] M. Noichl, "Opinionated: Simple, Clean Stylesheets for Plotting with Matplotlib and Seaborn," Aug. 2023.

[72] P. Patel, "colormaps: a collection of colormaps and color palettes for Python.."

[73] Z. Lin, "How to write effective prompts for large language models," *Nature Human Behaviour*, vol. 8, no. 4, pp. 611–615, 2024.

[74] L. Marins, "The challenge of measuring innovation in emerging economies' firms: A proposal of a new set of indicators on innovation," MERIT Working Papers 2008-044, United Nations University - Maastricht Economic and Social Research Institute on Innovation and Technology (MERIT), 2008.

[75] U. Cantner and H. Graf, "The network of innovators in jena: An application of social network analysis," *Research Policy*, vol. 35, no. 4, pp. 463–480, 2006.

[76] M. Squicciarini, H. Dernis, and C. Criscuolo, "Measuring patent quality: Indicators of technological and economic value," OECD Science, Technology and Industry Working Papers 2013/3, OECD Publishing, 2013.

[77] H. Peng, H. S. Qiu, H. B. Fosse, and B. Uzzi, "Promotional language and the adoption of innovative ideas in science," *Proceedings of the National Academy of Sciences*, vol. 121, no. 25, p. e2320066121, 2024.

[78] T. A. Branch, I. M. Côté, S. R. David, J. A. Drew, M. LaRue, M. C. Márquez, E. C. M. Parsons, D. Rabaiotti, D. Shiffman, D. A. Steen, *et al.*, "Controlled experiment finds no detectable citation bump from twitter promotion," *Plos one*, vol. 19, no. 3, p. e0292201, 2024.

[79] S. Dutta, B. Lanvin, L. Rivera León, and S. Wunsch-Vincent, *Global Innovation Index 2024: Unlocking the Promise of Social Entrepreneurship*. WIPO, 2024.

[80] L. Dahlander and D. M. Gann, "How open is innovation?," *Research policy*, vol. 39, no. 6, pp. 699–709, 2010.

[81] C. S. Arlinghaus, C. Wulff, G. W. Maier, C. Arlinghaus, C. Wulff, and G. Maier, "Inductive coding with chatgpt-an evaluation of different gpt models clustering qualitative data into categories," 2024.

[82] R. H. Tai, L. R. Bentley, X. Xia, J. M. Sitt, S. C. Fankhauser, A. M. Chicas-Mosier, and B. G. Monteith, "An examination of the use of large language models to aid analysis of textual data," *International Journal of Qualitative Methods*, vol. 23, p. 16094069241231168, 2024.

A Complexity rating LLM prompt

SYSTEM_PROMPT = """ You are computer science grad student. As a computer science grad student, you are extremely knowledgeable in software development, development lifecycles, and release patterns. Further, you also know and use multiple programming languages and frameworks. Currently you are tasked with rating the complexity of a software release given the release notes. As this is a research project, you will be provided an annotation procedure and the example to annotate. Be sure to follow the procedure and always respond with the XML formatted rating information. """

USER_PROMPT = """ ##Task

Rate the complexity of a software release given the software release's release notes. Specifically, you should rate the "complexity to implement" (i.e. how difficult does the feature, or bug, or change seem to be to implement).

For your "complexity to implement" rating, take on the persona of a core developer for the library. That is, some things may be easier or harder in different languages and those differences should be taken into account. I know you will do great! Just try your best!

###Rating Scale for Complexity to Implement

Use the following scale and criteria to rate the complexity to implement:

- 1. Almost no changes.

If any, they may be purely for documentation, project administration, or very minor bugfixes such as a typo or a small formatting issue.

- 2. Very few changes.

They may involve a new feature or a minor bugfix. But the changes are entirely minor. The changes are so small that they may not require any new documentation outside of a very brief mention in the release notes.

- 3. A few changes.

Changes involve some basic understanding of the library. They may involve a new feature or a minor bugfix. But the feature itself shouldn't be major. For example, it may be the addition of a new parameter to a function, or a new method to a class. It may require some new documentation but not a lot.

- 4. A small number of changes.

Changes involve some moderate level of understanding the library. They may involve a new major feature or a major bugfix. They may require some refactoring or changes to existing code but that isn't the main focus of the release. They likely require some additional documentation to announce the new feature, bugfix, or new behavior.

- 5. A moderate number of changes.

Changes involve a decently-high level of understanding the library. This may include multiple new features, major bugfixes or changes, or a moderate amount of refactoring. They should require extensive documentation to announce the new features, bugfixes, or new behavior.

- 6. A large number of changes.

Changes involve a high level of understanding the library. There are multiple new features, major bugfixes or changes, and/or a large amount of refactoring. Each change may be interacting with multiple systems or modules of the library or tool.

- 7. Extensive changes.

The release includes new features, refactoring. Complex interactions between multiple systems. To implement these changes would require extensive knowledge of the whole library to fully understand the effects of each change. Further, it would require

22

extensive testing to ensure that the changes are correct and do not break existing
functionality. This also requires extensive documentation to explain the changes to
users.

 ##Input Structure

You will be provided with an XML object with the following structure:
```
<release-notes-information>
    <repository-name>...</repository-name>
    <repository-description>...</repository-description>
    <repository-topics>...</repository-topics>
    <repository-language>...</repository-language>
    <release-notes>...</release-notes>
</release-notes-information>
```

##Response Structure

 - 'required-skills': A list of short (less than a sentence) semi-colon separated
notes, of the skills required to implement the changes in the release notes. I.e. what
knowledge (e.g. asynchrony, data structures, etc.) would be required to implement the
changes.

 - 'reasoning': A list of short (less than a sentence) semi-colon separated notes, that
justify the rating you are providing.

 - 'complexity-rating': The complexity rating you are providing. This should be a
number between 1 and 7, inclusive, where 1 is "very low complexity" and 7 is "very high
complexity". Always try to rate the release on the scale from 1 to 7, however, if there
is almost no information in the release notes, rate the complexity as "null".

Your response should have the following structure:

```
<rating-response>
    <required-skills>...</required-skills>
    <reasoning>...</reasoning>
    <complexity-rating>...</complexity-rating>
</classification-response>
```

Provide only the XML response, without any additional text or formatting.

 ##Release Information

```
<release-notes-information>
    <repository-name>{repo_name}</repository-name>
    <repository-description>{repo_description}</repository-description>
    <repository-topics>{repo_topics}</repository-topics>
    <repository-language>{repo_language}</repository-language>
    <release-notes>{release_notes}</release-notes>
</release-notes-information>
```

 ##Rating Response """