UNIVERSITY OF CALIFORNIA

SANTA CRUZ

TOWARD SUSTAINABLE AND SECURE OPEN SOURCE SOFTWARE: DISCOVERY, MEASUREMENT, AND DEFENSE

A dissertation submitted in partial satisfaction of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE AND ENGINEERING

by

Juanita Gómez

August 2026

The Dissertation of Juanita Gómez
is approved by:

_____
Álvaro Cárdenas, Chair

_____
James Davis

_____
Emily Lovell

_____-

Donald Smith
Interim Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

Abstract

Toward Sustainable and Secure Open Source Software:
Discovery, Measurement, and Defense

Juanita Gómez

In March 2024, a backdoor was discovered in xz Utils, a widely-used open source data compression library present in nearly every major Linux distribution. The attack was discovered days before merging into major distributions, and if this had happened, it would have allowed attackers to execute arbitrary code on millions of systems worldwide via SSH.

The success of this backdoor was enabled by two failures. The first was technical: weaknesses in the software supply chain allowed a malicious actor to inject code into a widely trusted release. The second was human: the project's only maintainer, overwhelmed and burned out after years of maintaining critical infrastructure alone, was the target of a multi-year social engineering campaign, in which a malicious actor built trust under a false identity and gradually obtained commit access to the project. This incident shows that software security failures and sustainability failures are not independent: an overburdened, unsupported maintainer is itself an attack surface.

Academic and scientific open source software (OSS) faces both of these crises simultaneously. Projects that critical infrastructure depends on are maintained by researchers, students, and faculty who contribute in their spare time, without dedicated security training or institutional support. Existing security frameworks such as NIST's SSDF, OWASP's SCVS, and SLSA were not designed with these communities in mind, and policy efforts such as the EU Cyber Resilience Act have shown that mandates developed without community input risk harming the ecosystems they are meant to protect.

This dissertation addresses the sustainability and security of academic open source software through two parallel empirical research tracks. The sustainability track com-

bines GitHub's REST API with LLM-based filtering to discover and characterize over 216,000 institutionally affiliated repositories across 32 academic and research institutions, finding that while 84% include a README, only 23.4% carry a detectable license and fewer than 2% include a Contributing Guide. Building on this dataset, we develop a maturity-staged sustainability framework that classifies projects into four lifecycle stages and generates targeted recommendations for Open Source Program Offices (OSPOs).

The security track examines whether post-9/11 trade security programs offer a workable model for OSS supply-chain policy, finding that effective frameworks require voluntary incentives and direct community engagement rather than top-down mandates. We further evaluate five large language models on the OWASP Benchmark for vulnerability triage, finding that o1-mini reduces false positives by 20% over the Semgrep baseline, demonstrating the potential for automation to reduce the security burden on individual maintainers.

Together, these contributions treat sustainability and security as interconnected problems. A project that cannot sustain itself cannot secure its code, and this dissertation takes steps toward closing both gaps.

Acknowledgments

The text of this dissertation includes reprints of the following previously published material:

Gomez, J., Muson, A., & Cardenas, A. ”With a Little Help from My (LLM) Friends: Enhancing Static Analysis with LLMs to Detect Software Vulnerabilities.” Published at the International Conference on Software Engineering (ICSE), LLM4Code Workshop, 2025.

Chapter 1

Introduction

Nearly all of the world's digital infrastructure depends on open source software. Operating systems, web servers, scientific computing platforms, financial systems, and medical devices all rely on open source components, most of which are maintained by small teams of volunteers working without dedicated funding or institutional support. A famous webcomic by Randall Munroe [145] captured this reality with precision: a towering structure labeled "all modern digital infrastructure" balanced on a single thin block labeled "a project some random person in Nebraska has been thanklessly maintaining since 2003." The comic is funny because it is accurate.

The consequences of this situation became obvious on March 29, 2024, when a Microsoft developer discovered a backdoor planted in xz Utils [73], a data compression library that most people have never heard of, but it is shipped inside nearly every major Linux distribution in the world. The attack was discovered days before merging into stable release channels; had it not been, it would have granted attackers the ability to execute arbitrary code on millions of systems worldwide via SSH [7]. The fragility of the dependency in the comic was not hypothetical; xz Utils was already inside the infrastructure, and almost no one knew.

What made this attack possible was not a single technical failure but two distinct but related ones. The first was a sustainability failure: the project had a bus factor of one. A single maintainer, working alone and largely without recognition or support, was responsible for a piece of software that critical global infrastructure depended on. Years of maintaining critical infrastructure alone, with little recognition or support, had left them burned out and susceptible to manipulation. The second was a security failure: a malicious actor recognized this vulnerability, spent two years building trust under a false identity, gradually obtained commit access, and injected a backdoor into a release that the package managers were prepared to distribute automatically. The sustainability failure created the conditions for the security one. An overburdened, unsupported maintainer is itself an attack surface.

1.1 The Sustainability Gap

A significant and largely overlooked portion of this critical open source infrastructure originates in academia. Jupyter, now the standard environment for data science and scientific computing across industry and research, started as a graduate student's tool for interactive Python work. PostgreSQL, which powers production databases at some of the world's largest organizations, grew out of a UC Berkeley research project. LLVM, the compiler infrastructure that Apple, Google, and most major technology companies depend on, was a PhD thesis at the University of Illinois. None of these projects were designed to become infrastructure; they became infrastructure because their authors invested in practices that allowed them to grow beyond academia: licensing the code openly, writing documentation, building contribution pathways. These practices allowed others to find, use, and eventually depend on them.

However, there are thousands of academic repositories that never realize their potential, not because the software is not valuable, but because sustainability practices are rarely part of the training or incentive structure of academic work. Hence, projects

are released publicly without the infrastructure that would allow others to find, build on, or sustain them. Academic OSS therefore has distinct needs that general purpose frameworks have not addressed. In recent years, universities have begun to recognize this gap through the creation of Open Source Program Offices (OSPOs) [205], institutional units in charge of supporting, coordinating, and promoting open source activity within their institutions. The CURIOSS network, a consortium of university OSPOs, now spans over 30 institutions and represents the leading edge of this effort. Yet even OSPOs face a foundational challenge: before they can support their institution's open source ecosystem, they must first know what it contains. Academic OSS is largely invisible, scattered across thousands of GitHub organizations and personal accounts with no central registry, no consistent metadata, and no systematic picture of who is maintaining what, at what stage of maturity, and with what needs. Without this picture, targeted institutional support is not possible.

Beyond discoverability, there is a deeper problem: the people who build and maintain this software are often doing so alone, without institutional support, without security training, and without the organizational scaffolding that would allow their work to survive past graduation. OSPOs need to understand its state: whether it is actively maintained, whether it has any contributors beyond its original author, whether it is approaching abandonment, and whether it has the basic sustainability infrastructure, licenses, documentation, governance that would allow it to survive long term. A project with a bus factor of one and no documentation is fragile. And fragile academic software, if it becomes depended upon, becomes the next xz Utils: critical infrastructure maintained by a single exhausted person with no institutional safety net. Measuring sustainability is therefore crucial in order to understand where projects are at risk and where institutional support can make the most difference.

1.2 The Security Gap

On the security side, governments and organizations such as NIST, OWASP, and the Linux Foundation have responded to the software supply-chain challenge with frameworks such as the Secure Software Development Framework (SSDF) [2], the Software Component Verification Standard (SCVS) [1], and Supply-chain Levels for Software Artifacts (SLSA) [5]. These efforts are valuable, but their assumptions about organizational resources, commercial incentives, and dedicated security teams do not map onto software built and maintained by researchers and students fitting open source contributions around their primary work. Other efforts have attempted to mandate security practices directly. The European Union's Cyber Resilience Act (CRA), for example, was designed to impose baseline security requirements on software products, but its initial drafts were alarming for open source communities by appearing to impose lifecycle security obligations on unpaid maintainers [174]. This illustrates a recurring failure: policy developed without continuous input from the communities risks not only being ineffective, but also harmful to the ecosystems it depends on.

Moreover, the growing use of AI-assisted development tools means more code is being written faster than human reviewers can keep up with, and the vulnerabilities introduced along the way are accumulating too [8]. AI tools are also generating a flood of low-quality, hallucinated vulnerability reports that land in maintainers' inboxes and consume the volunteer time that would otherwise go toward fixing real issues [121]. Addressing this at scale requires finding ways to reduce the burden on maintainers rather than adding to it. The answer cannot be to ask already-overburdened maintainers to do more, it has to be to make the tools better.

This dissertation addresses these gaps through two parallel research tracks: one focused on understanding and supporting the sustainability of scientific OSS, and one focused on securing it.

1.3 Dissertation Arc and Contributions

The dissertation is organized into two parts. Part I addresses the sustainability of OSS, and Part II addresses its security.

Part I: Sustainability of Academic Open Source Software

The sustainability track begins with understanding the academic landscape at scale and and moves toward developing frameworks for institutional support.

Chapter 2 situates this work within prior research on software sustainability, community health metrics, and institutional open source governance.

Chapter 3 addresses the foundational challenge of knowing what the academic OSS ecosystem contains. We build an automated pipeline that combines GitHub's REST API with LLM-based repository filtering and classification to identify over 216,000 institutionally affiliated repositories from an initial pool of more than 760,000 candidates across the 32 CURIOSS member institutions, achieving 90–92% accuracy in affiliation detection. We then characterize these repositories across licensing, documentation, community engagement, programming language, and project type, revealing a consistent and actionable gap: academic repositories are made public but rarely structured to invite or sustain external contribution. Only 23.4% carry a detectable license; fewer than 2% include a Contributing Guide; under 1% include a Code of Conduct. These findings reveal a consistent gap between what exists and what would be needed for these projects to survive and grow, motivating a deeper investigation into how sustainability can be assessed and improved at the institutional level.

Chapter 4 moves from characterization to diagnosis. Building on the data collected in Chapter 2 and grounded in a systematic review of prior work on OSS sustainability we develop a maturity-staged comparative framework that maps academic OSS projects onto three lifecycle stages adapted from the CNCF project maturity model:

Sandbox, Incubating, and Graduated. Using CNCF cloud-native projects as a sustainability benchmark, we compare eleven sustainability indicators across both ecosystems to provide a stage-aware picture of the sustainability landscape of academic OSS. This framework enables OSPOs to understand where their ecosystem stands relative to a mature reference population and intervene where it matters most.

Part II: Security of Academic Open Source Software

The security track examines where supply chain defenses fail and what can be done about it, at both the policy and technical levels.

Chapter 5 chapter provides a taxonomy of software supply chain security tools and prior work, establishing the landscape of existing defenses and their limitations.

Chapter 6 addresses a core challenge in OSS supply-chain security: existing policy frameworks were designed for commercial software producers, not volunteer-maintained communities, and have largely been developed without input from the people they are meant to govern. We examine whether the incentive-based frameworks developed to secure global trade after September 11, 2001, principally CTPAT and the AEO programs, offer a workable model for OSS policy. Through a comparative literature review and nine expert interviews with trade security veterans and OSS practitioners, we identify what transfers and what does not, and propose a framework built on voluntary "Trusted Source" certification, burden-sharing that places obligations on commercial beneficiaries rather than volunteer maintainers, and standards co-authored with open source communities.

Chapter 7 explores how automation can reduce the vulnerability triage burden on maintainers. We use large language models (LLMs) to triage the false positive that make static analysis tools impractical in volunteer-maintained projects. Using the OWASP Benchmark as an evaluation dataset, we evaluate five LLMs, GPT-4, GPT-4o Mini, Claude 3.5 Sonnet, o1-preview, and o1-mini, across three prompting strategies

(Zero Shot, Chain of Thought, and Tree of Thoughts). o1-mini substantially outperforms the Semgrep baseline in accuracy (87.04%), F1 score, and false positive reduction, though performance varies significantly by vulnerability type. These results demonstrate the growing potential of LLMs to complement static analysis and reduce the security burden on individual maintainers, while identifying the conditions under which current models still fall short.

1.4 Background

The research in this dissertation is informed by six years of active involvement in the open source community, including roles as a core contributor to Spyder, a community leader for the Scientific Python project, a member of the NumFOCUS Security Committee, and an organizer across SciPy, PyCon US, and the OpenSSF Community Day. This experience has shaped the research questions and contributions, since it has provided direct insight into the challenges that academic and scientific OSS maintainers face: the difficulty of sustaining projects beyond a single contributor, the gap between security best practices and what volunteer maintainers can realistically implement, and the absence of institutional frameworks that connect the work of individual researchers to the broader open source ecosystem.

This grounding extends directly into the research methodology. Throughout this work, I collaborated closely with the CURIOSS network of academic OSPOs, whose needs motivated the research questions in Part I and whose members provided feedback on findings and recommendations at key stages. I also presented this work at CURIOSS community gatherings and working group meetings, as well as at open source conferences including Open Source Summit, UC Open Summit, and FOSSY, seeking critical engagement from the communities the research is intended to serve. The goal throughout has been to ensure that the work is not only empirically rigorous but genuinely useful.

Part I

Sustainability

Chapter 2

Background and Literature Review: Sustainability

This chapter provides the background and prior work that situates the sustainability contributions of this dissertation. Section 2.1 defines the core concepts of open source software, sustainability, and project health. Section 2.2 reviews prior empirical work on measuring sustainability and health in open source projects.

2.1 Background

2.1.1 Open Source Software

Open source software (OSS) is software whose source code is made publicly available under a license that grants users the rights to use, study, modify, and distribute the software and its derivatives [163]. The Open Source Initiative (OSI) maintains the canonical definition of open source, requiring the license doesn't discriminate against people, groups, or fields of endeavor, and that it doesn't restrict other software distributed alongside it [163].

OSS development is typically collaborative and distributed. Contributors may include the original authors, external developers, researchers, students, and volunteers, coordinating asynchronously through version control platforms. GitHub, launched in 2008, has become the dominant platform for OSS hosting and collaboration, with over 180 million developers and more than 630 million repositories as of 2025 [82]. GitHub provides the infrastructure through which most modern OSS projects manage contributions: pull requests for code review, issues for bug tracking and feature requests, wikis and README files for documentation, and Actions for continuous integration. For the purposes of this dissertation, we focus on OSS hosted on GitHub, as it represents the largest and most systematically accessible body of open source activity.

OSS projects vary widely in their governance, contributor base, and institutional context. Some are maintained by large foundations such as the Linux Foundation, Cloud Native Computing Foundation or Apache Software Foundation, with dedicated staff and formal governance structures. Others are maintained by individual volunteers in their spare time. Academic OSS occupies a particular position in this landscape: it originates in research contexts, is often developed by students and faculty whose primary obligations are to their research rather than to software maintenance, and is frequently released publicly without the community infrastructure that would enable sustained external contribution [99].

2.1.2 Software Sustainability

The term sustainability is used across many domains with different meanings. In the context of software, sustainability refers broadly to the capacity of a software project to continue to exist, function, and evolve over time [24].

Becker et al. [24] identify five dimensions of sustainability applicable to software systems: technical, economic, environmental, social, and individual. In the context of OSS, the most relevant dimensions are technical sustainability, which concerns the abil-

ity of the software to continue functioning and being maintained as its dependencies and environment evolve, and social sustainability, which concerns the ability of the community of contributors to continue producing and maintaining the software over time. Economic sustainability, the availability of funding or resources to support development, is increasingly recognized as a critical factor for OSS projects that underpin commercial infrastructure [62].

A key challenge in studying OSS sustainability is that it is multidimensional and context-dependent. A project may be technically robust but socially fragile if it depends on a single maintainer; conversely, a project with many contributors may still suffer from poor code quality or architectural issues that limit its long-term viability. For academic OSS, sustainability is further complicated by the nature of academia: graduate students graduate, postdocs move on, and faculty shift research priorities, leaving projects without maintainers at any point in their lifecycle [112].

2.1.3 Project Health

Project health focuses on measurable signals of a project's current state and trajectory. Where sustainability is a long-term property, health captures the present condition of a project across dimensions such as activity, community engagement, governance, and documentation quality [84].

Several frameworks and tools have been developed to assess project health in practice. The CHAOSS project (Community Health Analytics for Open Source Software), an initiative of the Linux Foundation, has developed a structured set of metrics covering areas including contributor activity, project evolution, risk, and value, with examples such as contributor absence factor, release frequency, and license coverage [40]. The OpenSSF Scorecard provides a complementary set of security-focused health metrics [167]. Individual studies have proposed additional indicators including bus factor [48], response time to issues and pull requests [88], release frequency, and the presence of

community files such as a Contributing Guide or Code of Conduct [43].

For this dissertation, we use project health as a practical measure of sustainability, using a set of indicators derived from a systematic review of prior work that we describe in Chapter 4. This allows us to evaluate sustainability at large scale where individual assessment of long-term viability is not feasible.

2.2 Literature Review

The literature relevant to the sustainability contributions of this dissertation spans three overlapping areas: the measurement of OSS sustainability and health, the specific challenges of research and academic open source software, and the institutional structures that have emerged to support it. The methods used across these areas vary widely, including large-scale empirical studies mining software repositories, surveys of developers and maintainers, and qualitative interviews with practitioners. Rather than organizing the review by method, we organize it by theme, noting the methods used within each.

2.2.1 OSS Sustainability and Health Metrics

A substantial body of empirical work has aimed to identify the indicators that characterize healthy, sustainable open source projects and understand the factors that drive their long-term viability. These studies form the foundation for the measurement framework we develop in Chapter 4.

Understanding the risks that threaten open source project sustainability has been a central concern of empirical research. Coelho and Valente [43] surveyed the maintainers of 104 deprecated GitHub projects to identify nine reasons for project failure, finding that dependence on a small number of core developers is a recurring risk factor, and that the adoption of contributing guidelines and continuous integration are signifi-

cantly associated with whether a project succeeds or fails. Ehls [63] complemented this work through ten in-depth case studies of collapsed projects, identifying antecedents of developer departure and recurring patterns of project collapse. A follow-up study by Coelho et al. [44] proposed a machine learning approach to measure maintenance activity, finding that 16% of active GitHub projects become unmaintained within a single year.

Governance infrastructure has also been identified as a risk factor. Tourani et al. [204] conducted the first empirical study of codes of conduct in open source projects, finding that they emphasize diversity and welcoming communities, and that projects typically adopt them in response to prior experiences with negative behavior. Bosu and Sultana [32] documented the state of gender diversity in OSS projects, finding that women are significantly underrepresented and that gender biases in code review processes may discourage female participation, pointing to structural barriers that affect community health and sustainability.

Contributor concentration is a recurring theme in this literature. The bus factor, defined as the minimum number of contributors whose departure would jeopardize a project's continuity, has been studied empirically across large corpora of repositories. Avelino et al. [20] developed a method for computing bus factor from commit histories, finding that the majority of GitHub projects have a bus factor of one or two. A subsequent study by Avelino et al. [19] investigated 1,932 popular GitHub projects, finding that 89% had experienced losing their core development team at least once, and that in 70% of cases this happened within the first three years of a project's life.

Beyond individual project signals, several studies have examined sustainability at a broader scale, looking at ecosystems, evaluation frameworks, and the relationship between health metrics and software outcomes. Valiev et al. [220] conducted a mixed-methods study of 46,547 projects in the PyPI ecosystem, finding that a project's sustainability is shaped not only by internal factors but by its position within the broader software ecosystem, including its dependency relationships and the social ties of its

contributors. This ecosystem-level perspective is complemented by work on evaluation frameworks: Liao et al. [126] proposed a sustainability model built around four dimensions, openness, stability, activity, and extensibility, demonstrating through a case study of Stack Overflow that no single indicator captures the full picture and that all dimensions must be considered together. Ait et al. [11] reinforced this complexity by analyzing the survival rate of 1,127 GitHub repositories across four ecosystems, finding that a significant number of projects die within their first year and that the probability of surviving beyond five years is less than 50%, with activity patterns characterized by intensive coding periods followed by long stretches of inactivity. Alami et al. [12] examined how sustainability metrics relate to software quality in 217 Apache projects, finding that the relationship is not straightforward: sustainability and code quality are not consistently linked, and community growth can degrade certain quality metrics, suggesting that sustainability interventions and code quality practices may need to be pursued independently.

Goggins et al. [84] contributed the CHAOSS project's conceptual framing of community health, arguing that health is best understood as a multidimensional construct requiring multiple complementary metrics rather than a single composite score. Taken together, this body of work converges on a common finding: sustainability is multidimensional, context-dependent, and best assessed through a combination of activity, community, governance, and structural indicators. Linåker et al. [128] synthesized this prior empirical work through a snowball literature review of 146 publications, identifying 107 health characteristics across 15 themes and providing a structured overview of the dimensions that must be considered when evaluating OSS project health.

2.2.2 Research and Academic Open Source Software

Academic and research software presents sustainability challenges distinct from those of industry or community-driven OSS. As Howison and Herbsleb [99] showed in a foundational study of scientific software production, the incentives governing academic

software work are fundamentally different from those in other domains: publications and citations are the primary currency of academic recognition, leaving software development undervalued and under-resourced.

Survey-based studies have documented these challenges at scale. Carver et al. [37] conducted a large national survey of the state of the practice for research software in the United States as part of the US Research Software Sustainability Institute (URSSI) effort, finding widespread concerns about funding, training, career paths, and the absence of formal software engineering practices among researchers who develop software. Hettrick et al. [96] conducted a similar survey at the University of Southampton, documenting the gap between the central role software plays in research and the lack of institutional recognition and support for its development. Besser et al. [27] surveyed research software at the University of Illinois Urbana-Champaign, finding that many research software projects lack basic sustainability infrastructure even at institutions with dedicated computing support.

Qualitative work has complemented these surveys. Rosado de Souza et al. [178] interviewed research software engineers about what makes research software sustainable, identifying community, documentation, and long-term funding as the three most critical factors, and noting that most research software projects lack at least one of them. Mourão et al. [143] interviewed Brazilian computer science researchers about the success factors of research software, finding that visibility, community engagement, and institutional support were consistently cited. Hasselbring et al. [92] argued that research software should be both open and FAIR (Findable, Accessible, Interoperable, and Reusable), and that proper software engineering practices are a prerequisite for achieving both goals, situating sustainability within the broader open science agenda.

2.2.3 Institutional Support and OSPOs

Universities have increasingly acknowledged the growing need for structured support for open source software through the creation of Open Source Program Offices (OSPOs). The first academic OSPO in the United States was established at Johns Hopkins University in 2019 with support from the Alfred P. Sloan Foundation, and the SustainOSS academic ecosystem map currently reports approximately 25 academic OSPOs worldwide [198]. Young et al. [232] provide the first formal definition of an academic OSPO, distinguishing it from its industry counterpart by emphasizing the university's distinct goals around training students, securing grants, and preserving research products in the scholarly record. While academic OSPOs share some structural similarities with their industry counterparts [90, 162], universities operate under different incentives and constraints. A comprehensive study of university OSPOs by Ruediger et al. [179] documents the range of activities these offices undertake, including advising researchers on funding opportunities, helping institutions leverage open source for education and research, and educating the campus community about open source best practices.

Institutional efforts to map open source activity have taken several forms. Manual registries, including the Stanford Open Source Registry [166], the George Washington University OSPO Project Registry [200], and France's national research software catalogue [136], rely on self-reporting or curator input. They provide curated, actionable lists for OSPOs but do not scale and cannot surface projects whose maintainers have not proactively registered them. Semi-automated approaches have improved on this: the Institutional Innovation Grapher [209] identifies GitHub accounts that mention a university in their bios, and the University of Wisconsin Open Source Monitoring Playbook [87] links GitHub repositories to publications and authors via OpenAlex, but neither performs systematic repository affiliation detection from GitHub metadata. These efforts collectively illustrate both the demand for institutional open source discovery and the limitations of approaches that depend on self-reporting or partial automation.

Two prior surveys are particularly relevant to Part I of this dissertation. A survey con-

ducted by the OSPO at the University of Wisconsin-Madison [157] focused on open source usage, perceptions, and campus culture, providing the first publicly shared results from a university OSPO needs assessment. However, it relied on self-reported data and did not investigate the specific challenges that would inform targeted contributor support. The URSSI survey by Carver et al. [37] was broader and more comprehensive, but focused on research software development at the national scale rather than at the campus level, and did not distinguish between open source more broadly and research software specifically. More recently, Schwartz et al. [182] combined GitHub search with LLM-based filtering to identify research software engineering repositories across US universities.

Chapter 3

Discovering Academic Open Source: A Large-Scale Empirical Study of Institutional Repositories

3.1 Introduction

Some of the most impactful open source projects in the world began as academic work. Jupyter, the de facto platform for data science and scientific computing used by millions of researchers worldwide, evolved from a graduate student's side project at the University of Colorado and was developed into its current form at UC Berkeley [171]. PostgreSQL traces its roots to the POSTGRES research project at UC Berkeley [196] and is now one of the most widely deployed open source databases in existence. LLVM, the compiler infrastructure underpinning Apple, Google, and most major technology companies, began as a PhD thesis at the University of Illinois Urbana-Champaign [122]. What these projects share is not just their academic origins, but the fact that they were built with the practices that allow communities to grow around them: clear licenses that permit reuse, documentation that lowers the barrier to contribution, and governance in-

frastructure that signals the project is open to outside involvement. For every Jupyter or PostgreSQL, however, there are thousands of academic repositories that never realize their potential; not because the software is not valuable, but because they lack the basic infrastructure that enables discovery, reuse, and sustained collaboration. Building these practices into projects from the start is not administrative overhead; it is what separates a shared experiment from a sustainable community resource.

Institutions worldwide are increasingly recognizing this challenge through the formation of Open Source Program Offices (OSPOs), dedicated units that support researchers, students, and staff in developing open source software responsibly [180, 130]. CURIOSS (the Community for University and Research Institution OSPOs) brings together practitioners from 32 leading universities and research institutes spanning North America and Europe, including Carnegie Mellon University, ETH Zürich, Stanford, and Georgia Tech, all of which have explicitly committed resources to open source sustainability. Yet even these institutions face a fundamental obstacle: without a systematic, empirical picture of what their institution is actually building and under what practices, it is impossible to prioritize support, identify gaps, or demonstrate institutional impact. Without knowing what is being built, by whom, and how, OSPOs cannot act effectively.

In this chapter, we address that gap directly with a large-scale empirical study of open source practices across all 32 CURIOSS member institutions. Using an automated pipeline that combines GitHub's REST API with LLM-based repository filtering and project type classification, we discover over 760,000 candidate repositories and characterize approximately 217,000 affiliated repositories, examining their licensing, documentation, and community engagement infrastructure at institutional scale. Our findings reveal a consistent and actionable pattern: academic institutions produce a large volume of open source software, dominated by software development tools (45.6%) and educational projects (39.4%). Yet the infrastructure for sustainable, collaborative open source is largely absent. Only 23.4% of repositories carry a machine-detectable license; fewer than 2% include a Contributing Guide; and under 1% have a Code of Con-

duct, despite 84% having a README. A repository without a license cannot legally be reused or built upon; one without a contributing guide gives potential collaborators no clear pathway in. These gaps represent a systemic failure to convert individual academic effort into durable community assets, and they point directly to where OSPOs can intervene. To structure our investigation, we formulate three research questions:

- **RQ1**: How effectively can automated filtering identify institutionally affiliated repositories at scale from large, noisy repository collections?
- **RQ2**: What types of open source projects do academic institutions produce, and how are they distributed across institutions?
- **RQ3**: What open source practices characterize academic repositories, and how do they vary with project popularity?

Our contributions include:

- A large-scale empirical analysis of open source practices across the 32 CURIOSS institutions, producing actionable guidance for OSPOs on licensing, documentation, and community engagement gaps.
- A scalable method for discovering institutionally affiliated repositories on GitHub, yielding over 760,000 candidate repositories across all 32 CURIOSS member institutions.
- A validated six-category project type classification scheme with explicit precedence rules, achieving 90–92% accuracy.
- A dataset of around 217,000 affiliated repositories with associated metadata, licensing status, community file presence, and project type labels, shared with all CURIOSS member OSPOs to support institutional decision-making and made available to the research community.
- An interactive dashboard presenting per-institution breakdowns of repository types,

licensing status, and community file adoption, enabling OSPOs to identify gaps and prioritize targeted interventions for their specific institution: `https://juanis2112-repoexplorer.share.connect.posit.cloud`

- An open-source, configurable pipeline enabling any institution to replicate the study by editing a single configuration file. The code is available at: `https://github.com/UC-OSPO-Network/repofinder`.

3.2 Related Work

Understanding the institutional open source landscape requires integrating three distinct capabilities: *discovering* relevant repositories, *classifying* them by purpose, and *analyzing* their practices. Prior work has made progress in each area independently, but no single effort combines all three at institutional scale. Table 3.1 summarizes the works most similar to ours; the discussion below situates them within the broader literature.

Institutional Discovery and Registries: Several universities have developed tools or registries to map their open source activity. Manual registries; including the Stanford Open Source Registry [166], the George Washington University OSPO Project Registry [200], and France's national research software catalogue [136]; rely on self-reporting or curator input. They provide curated, OSPO-actionable lists but do not scale and cannot surface projects whose maintainers have not proactively registered them. Semi-automated approaches have improved on this: the Institutional Innovation Grapher [209] (now inactive) identifies GitHub accounts that mention a university in their bios, and the University of Wisconsin Open Source Monitoring Playbook [87] discovers open source contributions by linking GitHub repositories to publications and authors via OpenAlex and xDeepDive, but doesn't perform repository affiliation detection from GitHub metadata.

The most comprehensive prior discovery effort is Schwartz et al. [182], who combine

Table 3.1: Comparison of the most related works across six dimensions relevant to institutional open source governance. ✓ = fully addressed, ∼ = partially addressed, × = not addressed.

Work	Automated Discovery	Affil. Detection	Type Classif.	Practice Analysis	Multi-institution	OSPO-Actionable
Project Registries [166, 200, 136]	×	×	×	∼	×	✓
Innovation Grapher [209]	∼	∼	×	×	×	✓
UW Madison Playbook [87]	∼	×	×	×	×	✓
Schwartz [182]	✓	×	∼	∼	✓	∼
Munaiah [144]	✓	×	∼	×	✓	×
Balla [21]	×	×	✓	×	✓	×
Kalliamvakou [110]	✓	×	×	∼	✓	×
Alami [12]	×	×	×	✓	×	∼
This work	✓	✓	✓	✓	✓	✓

GitHub keyword search with university website scraping and LLM-based filtering to identify research software engineering (RSE) repositories across US research universities. Their approach is the closest precursor to ours: it is automated, operates at scale, and uses LLMs for filtering. Key differences are that Schwartz et al. focus specifically on RSE repositories rather than the full institutional open source landscape, do not study affiliation, do not classify project types, and do not produce a practice analysis targeted at OSPO governance needs. Their scope is also limited to US universities.

Repository Classification: Repository classification has been addressed from several angles. Munaiah et al. [144] implemented Reaper to distinguish engineered software projects using score-based heuristics and random forest classifiers, later extended

by Pickerill et al. [172] with PHANTOM, a time-series clustering method based on Git logs. Many efforts focus on topic and functionality classification via README topic modelling [187], keyword-driven hierarchical classification [238], multilingual source code classifiers [17], and application domain classification using third-party API calls [129]. More recent work has explored richer representations including BERT-style models [28], semantic knowledge graphs [108], and topic recommendation systems [57, 58, 222, 226, 107]. None of these works considers institutional affiliation as a classification dimension.

Analysis of Open Source Practices: Prior work has examined open source sustainability and community health from several angles. Ehls [63] analyzes sources of project failure to identify risks in open source development. Alami et al. [12] apply sustainability metrics to evaluate how community structure affects software quality. At the institutional level, the 2024 Open Source Survey at UW-Madison [157] provides insights into usage patterns and opportunities to improve institutional practices, but through voluntary survey responses rather than automated repository analysis.

GitHub Mining at Scale: Kalliamvakou et al. [110] documented the promises and perils of mining GitHub data, establishing key quality and bias considerations that we account for in our methodology. Ma et al. [133] introduced World of Code, an infrastructure for mining global VCS data and exploring cross-repository dependencies. Cosentino et al. [47] conducted a systematic mapping study of GitHub research methods. This work collectively informed our approach to mining open source software repository data.

Table 3.1 compares the most related works across six dimensions relevant to institutional open source governance. A recent systematic mapping study by Balla et al. [21] surveying 43 repository classification works confirms that none explicitly integrates institutional affiliation, project categorization, and actionable analysis at scale. The table makes this gap concrete: works with strong automation (Schwartz et al., Munaiah et al.) lack OSPO-actionable outputs, while tools built for OSPOs (registries, Innovation

Grapher) lack automation and scale. Our work is the only effort that addresses all six dimensions, enabling the large-scale empirical study of institutional open source practices we present in this chapter.

3.3 Stakeholder Personas and Use Cases

To motivate the study and demonstrate practical relevance, we identify three primary stakeholder personas for whom our pipeline and findings are actionable. These personas were developed with input from CURIOSS OSPO practitioners and reflect real needs articulated in CURIOSS community calls.

Persona 1: The OSPO Director - Demonstrating Impact and Securing Resources. OSPO directors must justify their program's existence and growth to university administrators, funding agencies, and external partners. Without a comprehensive inventory of institutional open source activity, impact communication relies on anecdote rather than evidence. Our pipeline provides the data for dashboards that quantify the scope of institutional open source contributions: number of affiliated projects, active contributors, license adoption rates, and community engagement metrics. This enables directors to report impact to administrators, make the case for additional staffing and budget, and support grant proposals that require evidence of existing open source capacity.

Persona 2: The OSPO Program Manager - Triage and Targeted Support. OSPOs have limited staff and cannot support every project equally. A program manager needs to identify which projects need more intervention and which are most likely to benefit from it. Our pipeline surfaces projects that are gaining traction (high stars, active contributors) but lack basic sustainability infrastructure (missing licenses, absent contributing guides). OSPOs can proactively reach out to high-impact projects with targeted support, improving the sustainability and compliance of the institution's most visible open source work.

Persona 3: The Research Community - Visibility and Connection. Researchers working on open source projects within the same institution are often unaware of each other's work, missing opportunities for collaboration, shared infrastructure, and joint grant proposals. Our pipeline produces an institutional inventory that can power developer connections between researchers doing similar work and identify opportunities for industrial collaborations.

3.4 Methodology

We identify and characterize open source repositories affiliated with CURIOSS member institutions through three sequential steps: discovering candidate repositories on GitHub, filtering out false positives, and categorizing retained repositories by project type.

3.4.1 Discovery

We query GitHub's REST API [79] using three complementary searches per institution. Each institution is represented by a set of lexical cues: its full name, acronym, email domain, and common alternate name. A *repository search* matches these cues against repository `name`, `description`, `README`, `tags`, and `email` fields via the `/search/repositories` endpoint. An *organization search* queries `/search/users?type=Organization` across organization `name`, `login`, `description`, `company`, and `blog` fields; all repositories owned by matched organizations are added to the candidate set. A *user search* queries `/search/users?type=User` over the same profile fields for individual developers; all repositories they own are similarly included. The three-phase workflow is illustrated in Figure 3.1 phase 1.

Beyond core repository metadata, we collect community files (README, code of conduct, contributing guide, security policy, issue and pull request templates) [78, 81];

engagement indicators (stars, forks, release downloads, subscriber count); contributor profiles (`username`, `name`, `bio`, `company`, `email`) for the top contributors of each repository 2; and, where the owner is a GitHub organization, organization metadata (`login`, `name`, `description`, `email`, `url`) 3. This strategy deliberately favors recall; false positives are removed in the next step.

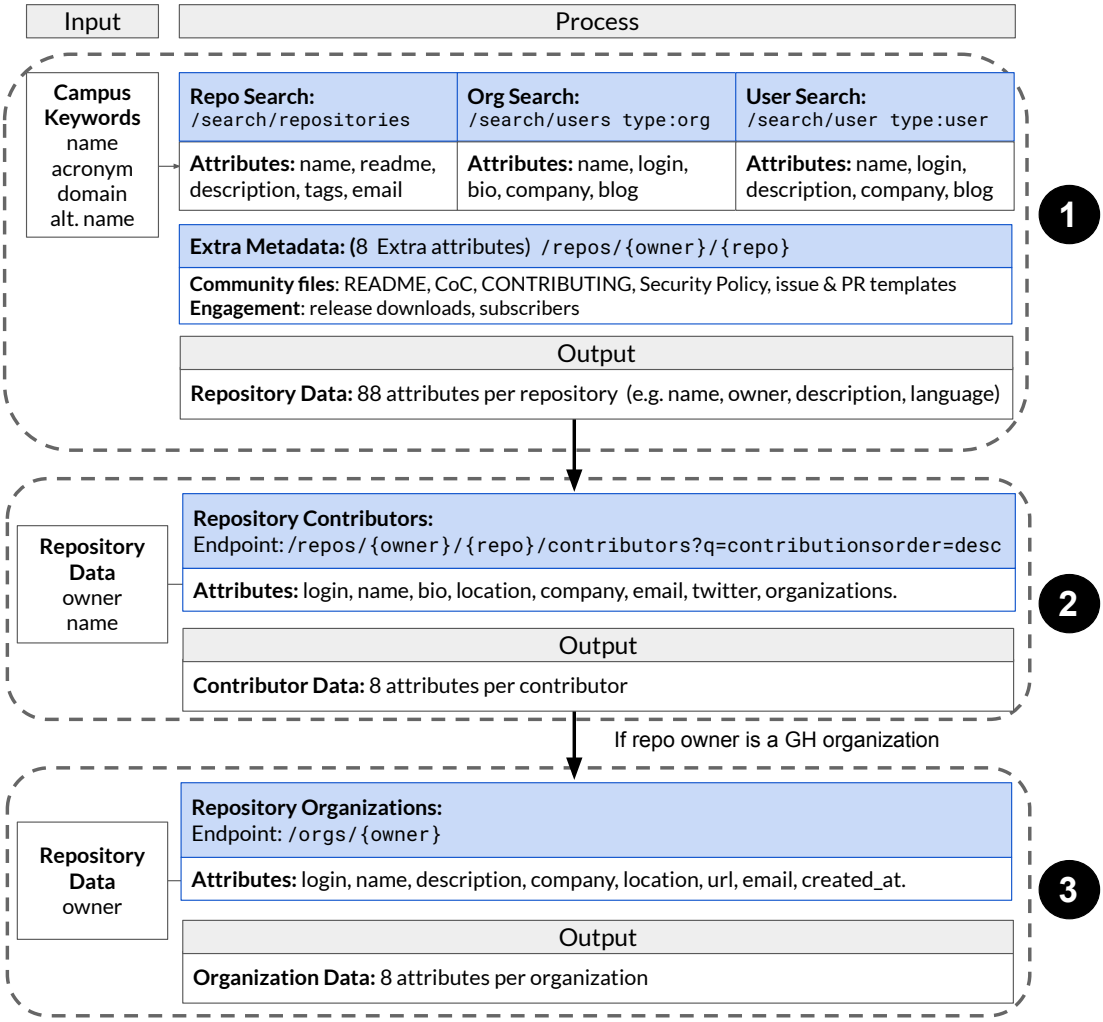

Figure 3.1: Data collection phases with GitHub's REST API.

3.4.2 Data Filtering

We reduce the candidate set to genuinely affiliated repositories in two steps: a cleaning phase to remove uninformative entries, followed by an affiliation classifier. In the cleaning phase we discard forks, archived repositories, templates, and repositories with zero reported size, as these do not represent original active development and bias affiliation and activity analyses.

Classifier Selection

To select the classifier, we evaluated three filtering strategies of increasing sophistication. A **score-based classifier** (SBC) assigns heuristic scores to keyword matches across repository, contributor, and organization fields; it is intuitive but cannot distinguish substantive institutional affiliation from incidental keyword overlap, yielding high false-positive rates. An **embedding-based machine learning** approach encodes aggregated repository metadata as text embeddings fed into standard classifiers (neural networks, random forests, SVMs); accuracy improved, but the approach requires a manually labeled training set. Our final approach uses **large language models**: we evaluated `gpt-3.5`, `gpt-4o`, and `gpt-5-mini`, with `gpt-5-mini` providing the best balance of classification quality and cost. All methods were validated on 100 manually labeled repositories from each of three CURIOSS institutions (UCSC, UCSD, UCSB). Figure 3.2 shows ROC curves for all five methods. More extensive descriptions of the alternative approaches appear in Appendix A.

Filtering using Selected Classifier: gpt-5-mini

To guide `gpt-5-mini`'s filtering decisions, we provide the model with an explicit definition of institutional affiliation

Definition 1 *A repository is considered affiliated with a university if there is clear, public evidence that its development, maintenance, or governance is connected to the university.*

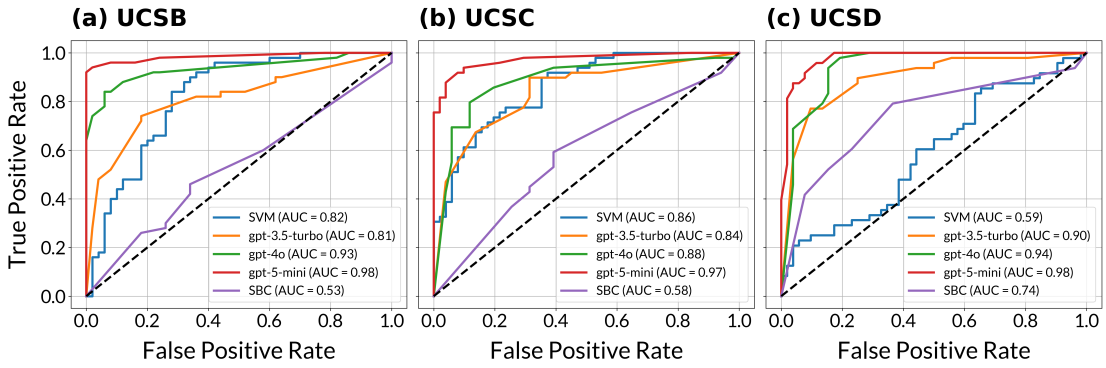

Figure 3.2: ROC curves for UCSB, UCSC, and UCSD for the five methods evaluated. `gpt-5-mini` achieves the highest area under the curve (AUC) for the three campuses, indicating strong classification performance.

This includes any of the following criteria:

***1. Research or Academic Unit Development**: Developed or maintained by a research group, lab, center, academic department, or institute that is formally part of the university.* Evidence *includes README statements such as "developed at . . . ", links to lab websites, or institutional pages under the university domain.*

***2. Contributor Affiliation (Explicit Evidence)**: One or more key contributors (maintainers, lead developers, or primary committers) are affiliated with the university.* Evidence *includes: (1) university email addresses, (2) public profiles or bios naming the university, or (3) README sections listing team members and their university affiliation.*

***3. Institutional or Administrative Development**: Developed or maintained by an official, non-academic university unit (e.g., libraries, Open Source Program Offices, IT departments, or research computing groups).*

***4. Official Ownership, Sponsorship, or Endorsement**: Owned, sponsored, or explicitly endorsed by the university.* Evidence *includes: (1) hosting under an official university GitHub organization, (2) README or website statements indicating university sponsorship or ownership, or (3) copyright notices naming the university.*

5. Documented Collaboration or Partnership*: The README or project website explicitly states a collaboration, partnership, or joint development effort with the university or one of its units.*

6. University-Linked Project Infrastructure*: The project's official homepage, documentation site, or primary web presence is hosted under the university's domain or a clearly associated subdomain.*

7. Educational Outreach and Online Courses*: Online learning initiatives affiliated with the university, including: (1) repositories linked to online specializations or courses offered on platforms such as Coursera or edX, and (2) course materials, code examples, or tools developed specifically for such offerings.*

The model receives repository metadata, organizational metadata, and profiles of the top two contributors, with text fields truncated to 20,000 characters (first 15,000 and final 5,000, preserving acknowledgement sections that frequently appear near the end of READMEs). It returns a probability between 0 and 1; repositories scoring above 0.5 are retained as affiliated. The full prompt is in shown below:

```
You are tasked with determining the likelihood that a GitHub repository
belongs to a university based on the following definition:
  • Definition: <Definition 1>
  • Here is the information about the repository: <repository
    information>
  • And here is the university context: <university information>
Based on this information and the definition above:
  • Estimate the probability between 0 and 1 (e.g., 0.87) representing
    how likely it is that the repository belongs to the university.
  • Provide a brief explanation (1--2 sentences) justifying your answer.
Your response must be formatted exactly like this:
  • Probability: <value between 0 and 1>
  • Explanation: <your explanation here>
```

3.4.3 Data Categorization

Retained repositories are classified by **project type** to enable granular analysis of institutional open source contributions. We designed six categories inspired by prior work [192], with concrete examples and keyword indicators added to improve LLM compatibility. The following definition is provided verbatim to the model:

Definition 2 *You must classify each GitHub repository into exactly one of the following categories based on its primary purpose. When multiple purposes are present follow the precedence rules described below.*

`DEV`*: A repository primarily used for the development and maintenance of a software artifact, including tools, libraries, components, applications, services, or APIs. The presence of documentation, a website, or example code does not override this classification if active software development is the main function.*

`EDU`*: A repository primarily used for educational purposes, including course-related materials (e.g., assignments, labs, class projects, or course websites), teaching infrastructure, or instructional content explicitly tied to a course, workshop, or training program. This category includes teaching demos and repositories created to support internships, tutorials, or learning exercises, provided their main goal is instruction rather than production use. If a repository contains software or an application developed as part of completing a course or academic requirement, it is classified as* `EDU`*, even if it resembles a standalone or production-style software project.*

`DOCS`*: A repository primarily used to store or track non-educational documents, such as reports, policies, white papers, specifications, or meeting notes.*

`WEB`*: A repository primarily used to host a public-facing website or informational page, such as a project homepage, documentation website, or static or CMS-based informational site. This includes repositories built with static site generators (e.g., Jekyll or Hugo), as well as site-specific styles, layouts, or templates, when the main purpose is presentation rather than software development. Reusable themes or design systems intended for general adoption are excluded and classified as* `DEV`*. Personal websites, portfolios, or "about me" pages are clas-*

sified as `OTHER`*.*

`DATA`*: A repository primarily used to store, curate, or distribute datasets, including research datasets, benchmarks, or data collections (including images). Lightweight scripts for data loading or inspection do not change this classification.*

`OTHER`*: A repository that does not clearly align with any of the above categories, such as empty repositories, personal experiments, configuration-only repositories, or miscellaneous content without a clear primary purpose.*

The model receives repository name, description, README, stars, forks, and contributor count, and returns the predicted category with a brief explanation. The full classification prompt is shown below:

```
You must classify each GitHub repository into exactly one of the
following categories: <Definition 2>
Choose the most appropriate category based on the repository information.
  • Here is the information about the repository: <repository
    information>
  • Your task: Return only the predicted category (one of: DEV, EDU,
    DOCS, WEB, DATA, OTHER), and a short explanation.
  • Your response must be formatted exactly as:
    -- Category: <one of the 6 categories>
    -- Explanation: <brief explanation>
```

3.5 Results

The three-step repository search identified **761,920** candidate repositories across all 32 CURIOSS member institutions. After data cleaning, **529,879** repositories remain for filtering. We present results for each research question: RQ1 evaluates the accuracy of the LLM-based filtering step; RQ2 characterizes the distribution of project types across institutions; RQ3 examines open source practices across the full affiliated dataset

(Section 3.6).

3.5.1 Filtering (RQ1)

We evaluate the accuracy of our LLM-based pipeline for identifying institutionally affiliated repositories, using manually labeled ground truth from three CURIOSS institutions (UCSD, UCSC, and UCSB) with 100 labeled repositories per institution.

Accuracy Analysis

As discussed in Section 3.4.2, `gpt-5-mini` achieved the best performance at filtering out false positives. We determine the optimal classification threshold for this model by maximizing Youden's J statistic [26], and evaluate its performance using accuracy and the F1 score. The results are summarized in Table 3.2.

University	F1-score	Accuracy	False Positive %	False Negative %
UCSB	0.90	0.91	0%	9%
UCSC	0.90	0.91	1%	8%
UCSD	0.93	0.94	2%	4%

Table 3.2: Filtering performance for `gpt-5-mini` across three CURIOSS institutions.

Both accuracy and F1 score exceed 90% across all three institutions. Most errors are false negatives, occurring when affiliation evidence is subtle; for example, a faculty mention buried at the end of a README, a brief single-line acknowledgment, or contributor profiles spanning multiple institutions. A smaller number of false positives occurred, typically involving repositories that reference a university domain without a substantive institutional connection.

Affiliated Repositories Per Campus

We apply `gpt-5-mini` to filter false positives across all 32 CURIOSS institutions; results are shown in Figure 3.3. The histogram presents the total number of repositories after data cleaning (529,879) and the number classified as affiliated (216,920).

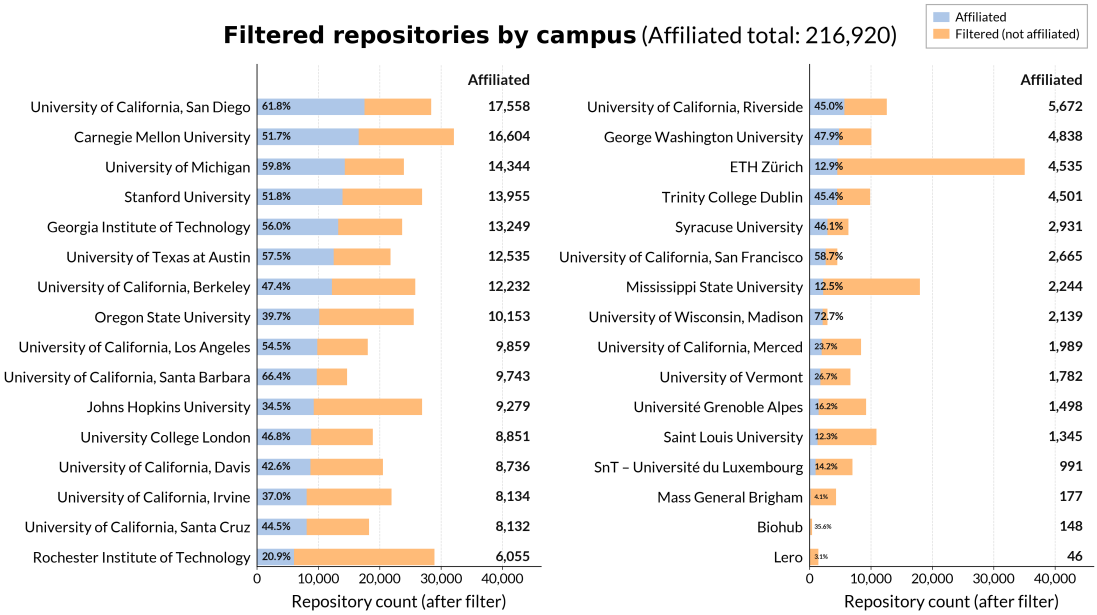

Figure 3.3: Affiliated repository counts per CURIOSS institution after LLM filtering. Each bar shows the total candidate repositories (after cleaning) and the affiliated subset.

Repository counts and affiliation rates vary considerably across the 32 institutions; out of approximately 760,000 total candidate repositories, only 28.5% are classified as affiliated (40.9% of the cleaned ones). Low affiliation rates often reflect search term ambiguity. ETH Zürich (12.9%) is the clearest case, as its acronym doubles as the Ethereum ticker symbol and appears in names like *Ethan* and ethical hacking content. Similarly, RIT (20.9%) appears as a substring in common words and tools, SLU (12.3%) false positives come from *Slurm* and *slugify*, and MSU (12.5%) is shared by at least three distinct universities. Conversely, longer and more distinctive acronyms such as UCSD and UCSB achieve affiliation rates above 50%. These results suggest that

keyword-based scraping effectiveness is inherently tied to the distinctiveness of each institution's search terms.

3.5.2 Project Type Classification (RQ2)

We first present the accuracy results of our project type classification model on the manually labeled test dataset to evaluate its overall performance and reliability. We then analyze the distribution of project types across all 32 CURIOSS institutions.

Accuracy Analysis

We evaluate `gpt-5-mini` on project type classification using 100 manually labeled repositories per campus across UCSB, UCSC, and UCSD, sampled from repositories previously predicted as affiliated. Table 3.3 reports accuracy per project type against these manual labels.

Project Type	**UCSB**		**UCSC**		**UCSD**	
	Count	**Acc.**	**Count**	**Acc.**	**Count**	**Acc.**
DATA	8	0.62	0	-	4	0.5
DEV	26	0.92	30	0.90	21	0.86
DOCS	0	-	3	0.33	1	0
EDU	51	0.98	50	0.96	65	0.98
OTHER	12	0.92	9	0.89	6	0.83
WEB	3	0.67	8	0.75	3	1.0
Total	100	0.92	100	0.90	100	0.92

Table 3.3: Project type classification results per university. Each cell shows the number of repos and accuracy.

Overall accuracy is high across all three campuses (90–92%). `EDU` and `DEV` repositories score highest, benefiting from explicit indicators such as course identifiers or deployment instructions. `DATA` and `WEB` are more challenging, as their READMEs can

resemble documentation or development repositories. DOCS repositories are rare in the test sets and show lower accuracy, reflecting overlap with WEB and DEV categories, particularly when documentation is colocated with code or hosted as a static site.

Distribution of Repositories by Project Type

We use gpt-5-mini to classify affiliated repositories across all 32 institutions; Table 3.4 summarizes the distribution by type.

	DEV	EDU	DATA	DOCS	WEB	OTHER	Total
Count	98,990	85,476	5,008	2,146	6,303	18,997	216,920
Percentage	45.6%	39.4%	2.3%	1.0%	2.9%	8.8%	100.0%

Table 3.4: Overall distribution of repository types across the 32 CURIOSS institutions.

Overall, the distribution is dominated by DEV and EDU repositories, which together account for approximately 85% of all affiliated projects (45.6% DEV and 39.4% EDU). This balance reflects GitHub's dual role across the academic system: a platform for active research software development and a central infrastructure for instructional use.

Figure 3.4 breaks down repository types by institution. A clear pattern emerges between institutional research orientation and the share of repositories classified as DEV. Pure research institutes and graduate-only universities rank highest: Mass General Brigham and Chan Zuckerberg Biohub both reach 81%, while UCSF, a graduate-only institution dedicated to health sciences research, reaches 63%. Technically oriented universities such as ETH Zürich (62%), Stanford (58%), Carnegie Mellon (53%), and Georgia Tech (52%) follow closely.

At the lower end, institutions such as Saint Louis University (29%), UC Santa Barbara (30%), and the University of Vermont (34%) have repository portfolios more heavily skewed toward EDU repositories. This does not necessarily indicate that these institutions produce less research software or fewer development tools. Rather, it

suggests that GitHub may be used more extensively for teaching and coursework. For example, UCSB includes many repositories associated with course organizations such as `ucsb-cs156-s24`, `ucsb-cs156-w24`, and `ucsb-cs56-projects`; GWU includes course-related owners such as `GWU-CSCI3411` and `gw-cs3410-s19`; and SLU includes teaching- or capstone-oriented organizations such as `slu-cs`, `slu-dss`, and `SLU-CS-Capstone`. Taken together, `DEV` share appears to be a useful proxy for research software intensity, while also reflecting student-body composition and institution-specific patterns of GitHub use.

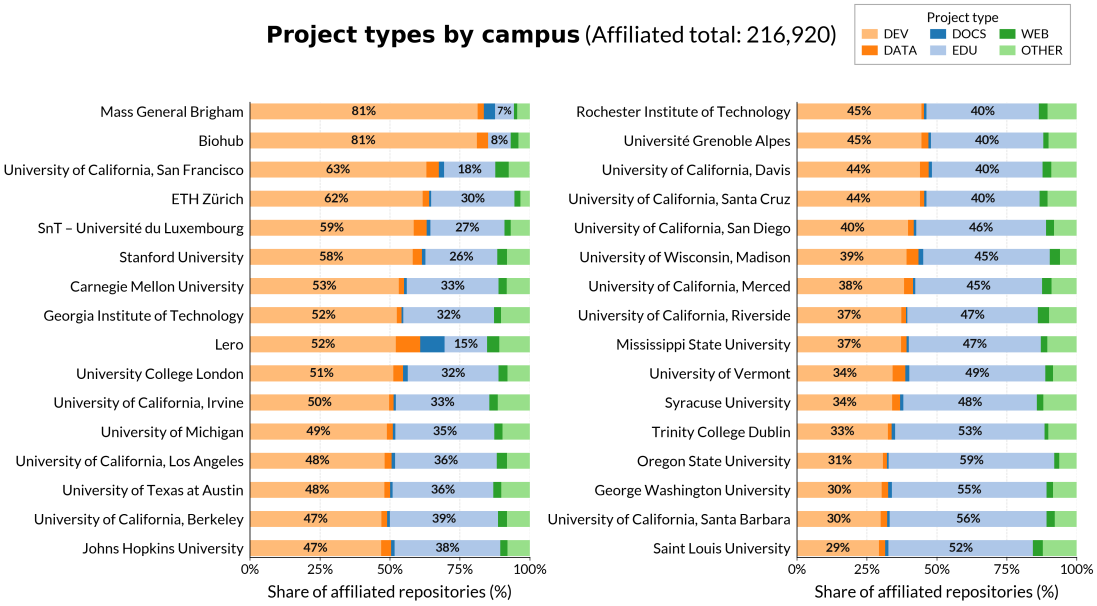

Figure 3.4: Project type distribution across CURIOSS institutions, ordered by `DEV` share. Research-focused institutions show the highest proportion of software development repositories.

3.6 Insights on Affiliated Repositories (RQ3)

We examine the open source practices of the 216,920 affiliated repositories across all 32 CURIOSS institutions. We analyze programming language and license distributions as

indicators of project composition, then turn to the adoption of GitHub's recommended community standards, the practices most directly tied to whether a project can attract and sustain external collaboration. The central finding is a consistent gap between basic discoverability (README, description) and collaborative infrastructure (licenses, contributing guides, codes of conduct), a pattern that holds across institutions and project types, with only partial improvement at higher visibility levels.

3.6.1 Programming Language Distribution

Figure 3.5 shows the aggregated distribution of programming languages in all affiliated repositories from all 32 CURIOSS institutions, broken down by project type.

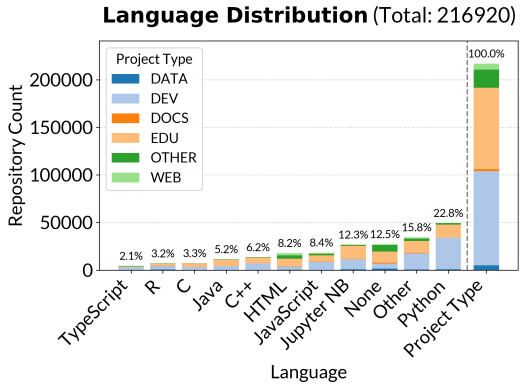

Figure 3.5: Programming language distribution across all 32 CURIOSS institutions. Languages representing less than 2% of total usage were aggregated under "Other." The bar on the right labeled "Project type" shows the overall distribution of repository types and serves as a baseline for interpreting language usage within each category.

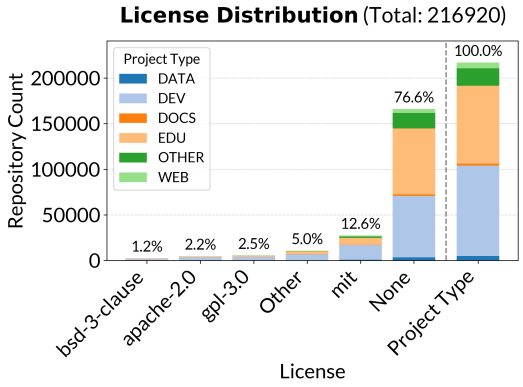

Figure 3.6: Software license usage across all 32 CURIOSS institutions. Licenses representing less than 1% of total usage were aggregated under "Other." The bar on the right labeled "Project type" shows the overall distribution of repository types and serves as a baseline for interpreting distribution within each type.

Python is the most widely used language overall (22.8% of all repositories), though this figure alone reflects the considerable linguistic diversity of academic open source: no

single language dominates, and the long tail of other languages is substantial. Jupyter Notebook is also highly prevalent (12.3%), particularly in educational (EDU) repositories, where it serves as a popular format for teaching and interactive coursework. HTML is the most frequently used language in web-related (WEB) repositories, closely followed by JavaScript. Other notable languages include Java, mainly appearing in educational projects; C++ and C, common across both DEV and EDU types; and R, reflecting the strong presence of statistical computing in academic research. Taken together, the concentration of Python and Jupyter Notebook points to a research-computing-oriented ecosystem, while Java, C++, and C reflect traditional computer science coursework and systems-level work; the distribution mirrors the full breadth of academic output from data science to infrastructure research.

3.6.2 License Distribution

Understanding the licensing choices of repositories provides insight into how openly these projects are intended to be shared and reused. Figure 3.6 shows the distribution of software licenses across all 32 CURIOSS institutions. A notable finding is that 76.6% of repositories appear to lack an explicit license (note that the GitHub API may fail to detect licenses stored in non-standard locations, so this figure is an upper bound on truly unlicensed repositories). Many of these unlicensed repositories are DEV projects, likely smaller research efforts or student works not intended for broad open source distribution. The large majority of OTHER repositories have no license, consistent with their varied and often informal content. Among repositories that do specify a license, MIT is by far the most popular, covering 12.6% of all repositories and representing more than half of those that carry any license; it is prevalent across both EDU and DEV projects as well as a smaller portion of web repositories. There is also notable license diversity beyond MIT, including GPL, Apache, and BSD licenses, with at least 5% using a variety of other licenses each representing less than 1% individually. From an OSPO perspective, the 76.6% unlicensed rate is the most immediately actionable

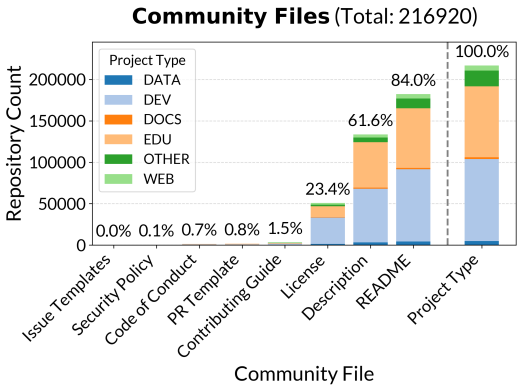

Figure 3.7: Presence of community files across all 32 CURIOSS institutions. The bar on the right labeled "Project type" shows the overall distribution of repository types and serves as a baseline for interpreting community file usage within each category.

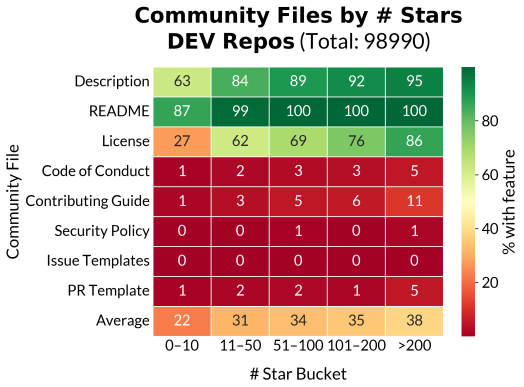

Figure 3.8: Presence of community files in `DEV` repositories across all 32 CURIOSS institutions, segmented by number of GitHub stars.

finding in this section: a repository without a license cannot legally be reused or built upon by others regardless of code quality, making license adoption the highest-impact single intervention an OSPO can make.

3.6.3 Open Source Sustainability and Community Engagement

A central question for institutional OSPOs is whether open source projects at their institutions exhibit the practices needed to sustain collaboration and community growth beyond the original developers. To address this, we analyze the presence of the community standards recommended by GitHub [78] (Description, README, License, Code of Conduct, Contributing Guide, Security Policy, Issue Templates, and Pull Request Templates) as concrete indicators of openness, collaboration readiness, and long-term project sustainability across all 32 CURIOSS institutions. Figure 3.7 illustrates the percentage of repositories that include each of GitHub's recommended community stan-

dards.

Most repositories include a README (84%) and a description (61.6%), making these the most commonly adopted community practices. These rates are notably higher than the GitHub-wide average of approximately 63% README adoption reported by the 2025 GitHub Octoverse [82], suggesting that academic repositories are comparatively well-documented at the basic level. However, when looking beyond basic documentation, the picture changes sharply. Only 23.4% of repositories include an explicit license. Adoption drops drastically for collaborative infrastructure: Contributing Guides appear in only 1.5% of repositories and Codes of Conduct in just 0.7%, both well below the already-low GitHub platform averages of 5.5% and 2%, respectively [82]. Moreover, Security Policies (0.1%), Issue Templates (0.0%), and Pull Request Templates (0.8%) are nearly absent across the board. This pattern, where basic discoverability is reasonably addressed but collaborative and governance infrastructure is largely absent, reflects a culture of **open code, closed doors**: repositories are made public, but not structured to invite or sustain external contribution. This gap represents a direct opportunity for OSPOs to intervene through targeted education and tooling support.

We are particularly interested in the repositories classified as `DEV`, as these are the most likely to be intended for collaborative software development. Additionally, we examine whether community file adoption scales with project visibility, under the assumption that more popular projects may attract more contributors and therefore require better governance practices. We use the number of GitHub stars as a proxy for popularity and focus our analysis on `DEV` repositories across all 32 CURIOSS institutions. While stars are an imperfect measure of popularity (they can reflect one-time interest, bookmarking behavior, or social media attention rather than sustained use), they are widely used in the literature as a practical signal of community engagement and provide a useful ordering for comparing adoption patterns across visibility tiers [31].

Figure 3.8 presents a heatmap showing the presence of community files in `DEV` repositories, segmented into five star buckets. As shown in the heatmap, README and

description adoption become nearly universal among more popular projects: all repositories with more than 50 stars include a README (100%), and description adoption reaches 92–95% in the top star tiers. Licensing shows a particularly strong correlation with popularity: 69% of repositories with 51–100 stars declare a license, rising to 86% for those with more than 200 stars. Despite this upward trend, the vast majority of lower-visibility projects (0–10 stars) show only 27% license adoption, meaning most academic open source software is shared without the legal clarity needed for reuse. Code of Conduct and Contributing Guide adoption also increase monotonically across star buckets; however, their overall levels remain critically low even at the highest popularity tier, with only 5% of the most starred `DEV` projects including a Code of Conduct and 11% including a Contributing Guide. Security Policies and Issue Templates remain effectively absent across all popularity levels. These results point to a systemic gap: even the most visible and widely used academic repositories have not adopted the collaborative infrastructure needed to grow into sustained open source communities, reinforcing the case for structured OSPO intervention at the institutional level.

3.7 Conclusions and Future Work

This chapter presented a large-scale empirical study of open source practices across 32 CURIOSS member institutions, enabled by an automated pipeline that combines GitHub's REST API with LLM-based repository filtering and project type classification. The pipeline discovers over 760,000 candidate repositories and characterizes approximately 217,000 affiliated projects, providing, to our knowledge, the first systematic, cross-institutional picture of academic open source activity at this scale.

RQ1 (Filtering). Our LLM-based filtering approach using `gpt-5-mini` achieves 90–92% accuracy across the three campuses evaluated, with errors concentrated in false negatives rather than false positives. Affiliation rates vary widely across institutions, due to how distinctive each institution's search terms are. Those with short or widely

shared acronyms, such as ETH, RIT, SLU, and MSU, suffer from severe keyword collision with unrelated content, while institutions with longer or more unique identifiers achieve higher precision.

RQ2 (Project types). Affiliated repositories are dominated by software development (`DEV`, 45.6%) and educational (`EDU`, 39.4%) projects, with smaller shares in `OTHER` (8.8%), `WEB` (2.9%), `DATA` (2.3%), and `DOCS` (1.0%). The balance between `DEV` and `EDU` varies substantially across institutions: the `DEV` share appears to capture one dimension of research software intensity, while also reflecting institutional GitHub use patterns, especially the extent to which GitHub is used for teaching and coursework.

RQ3 (Open source practices). The picture that emerges is one of widespread sharing without the infrastructure for collaboration. README adoption is high (84%) and exceeds the GitHub platform average (63%), but collaborative and governance infrastructure is nearly absent: only 23.4% of repositories carry a detectable license, 1.5% include a Contributing Guide, and 0.7% include a Code of Conduct; both below the already-low platform averages of 5.5% and 2%, respectively. Even among the most starred `DEV` repositories (above 200 stars), only 11% include a Contributing Guide and 5% a Code of Conduct.

3.7.1 Recommendations for OSPOs

These findings point to concrete interventions that OSPOs can prioritize according to project maturity, visibility, and practices. We organize them around the gaps identified in the data:

- **License adoption:** The most pervasive gap is the near-absence of machine-detectable licenses (76.6% of repositories). OSPOs can address this through: (1) providing ready-to-use license templates tailored to their institution's policies; (2) running targeted workshops for research groups and student organizations on the legal and reuse implications of open licensing; and (3) integrating automated

license checks at the organizational level to ensure repositories include proper licensing information early in the development process.

- **Education and awareness around open source practices:** The low adoption of Contributing Guides, Codes of Conduct, issue templates, and related community files should not be interpreted only as unwillingness to support contributors. In academic settings, many maintainers may simply be unaware of these practices, especially when repositories begin as research code, course projects, lab prototypes, or student assignments. OSPOs can play an important educational role by making these norms visible and actionable. This could include short training modules, onboarding materials for new research groups, and lightweight checklists that explain when and why to add files such as `CONTRIBUTING.md`, `CODE_OF_CONDUCT.md`, `SECURITY.md`, and issue or pull request templates. Rather than applying the same requirements to every repository, OSPOs could provide a maturity-based roadmap: early-stage projects might start with a license and README, projects with collaborators might add contribution guidelines and issue templates, and externally visible projects might adopt governance and security practices.

- **Contributing infrastructure:** With Contributing Guides at 1.5% and Codes of Conduct at 0.7%, most affiliated repositories are not structurally ready to welcome external contributors. OSPOs can lower this barrier by: (1) publishing institutional template repositories that pre-populate a `CONTRIBUTING.md`, a Code of Conduct (e.g., the Contributor Covenant), and a `SECURITY.md`; (2) integrating these templates into default starter projects used by institutional GitHub organizations; and (3) including community file checklists in any existing OSPO project support or grant reporting process.

- **Language-specific support and infrastructure:** The language distribution suggests that OSPO support should also be tailored to the technical ecosystems most common in academic repositories. Python is the most widely used language

overall and is especially prominent in `DEV` repositories, pointing to a need for reusable guidance around Python packaging, dependency management, testing, documentation, and release practices. At the same time, Jupyter Notebook is highly prevalent in `EDU` repositories, reflecting its central role in teaching and interactive coursework. OSPOs could therefore provide targeted support for instructors and researchers using notebooks, including templates for reproducible notebooks, guidance on environment files, version-control practices for notebooks, and workflows for publishing course materials. This kind of language-aware support would better match the actual technical profile of academic open source than generic open source guidance alone.

- **Priority targeting for high-impact projects:** OSPO support should be prioritized according to a repository's visibility, maturity, and likelihood of attracting external users or contributors. Our data shows that projects with more than 50 stars have significantly higher license adoption and community file presence than lower-visibility projects, yet even this group still has substantial gaps. OSPOs should use engagement signals, including stars, forks, and contributor counts, to identify projects that are gaining external traction but lack governance infrastructure, and proactively offer support to those maintainers. The pipeline developed in this study enables this kind of systematic prioritization at institutional scale.

3.7.2 Threats to Validity

Several limitations should be considered when interpreting our results.

1. Our study is limited to public repositories on GitHub. Academic open source activity may also occur on GitLab, Bitbucket, institutional self-hosted platforms, or in private repositories, so our dataset should be interpreted as a view of GitHub-visible institutional activity rather than the complete academic software ecosystem.

2. Repository discovery depends on institution-specific search terms, including names, acronyms, domains, and alternate names. Short or ambiguous acronyms can retrieve many unrelated repositories, while projects that do not mention the institution in searchable metadata may be missed. Although our LLM-based filtering step reduces false positives, it cannot fully eliminate discovery bias.

3. Affiliation and project type labels are produced by an LLM classifier. We validated the approach on manually labeled repositories from three institutions and observed high accuracy, but classification errors may remain, especially for repositories with sparse metadata, ambiguous ownership, or overlapping purposes.

4. Our analysis of licenses and community files relies on GitHub API metadata and standard file detection. Repositories may contain licenses, contribution instructions, or governance information in non-standard locations that are not detected by the API. As a result, reported absence rates should be interpreted as the absence of machine-detectable practices rather than definitive evidence that such information does not exist.

3.7.3 Data Sharing and Future Work

The full dataset produced by this study has been shared with all CURIOSS member OSPOs and made available through an interactive dashboard, allowing each institution to explore its own affiliated repositories, identify specific needs within the institution, and pinpoint projects that may benefit from targeted support.

Looking ahead, several directions call for further investigation. First, the pipeline currently relies on GitHub as the sole data source; extending it to GitLab, Bitbucket, and institutional self-hosted platforms would improve coverage. Second, another area for improvement is leveraging external signals such as publication data from sources like OpenAlex, both to surface additional repositories and to strengthen links between already identified projects and their scholarly outputs. Third, a longitudinal study track-

ing changes in community file adoption over time would clarify whether institutional OSPO programs lead to measurable improvements in open source practices. Finally, engaging directly with maintainers at CURIOSS institutions to validate classifications and gather qualitative context would strengthen both the methodology and the relevance of OSPO recommendations derived from it.

Chapter 4

Assessing Sustainability: A Maturity-Staged Framework for Academic OSS

4.1 Introduction

Open source software (OSS) is an important part of modern scientific infrastructure. Research software developed in universities supports work across multiple fields from bioinformatics to machine learning, but many of these projects are developed under conditions that make long-term maintenance difficult. Funding is often temporary, contributors may leave when students or researchers move to new positions, and academic incentives usually reward research outputs rather than proper software practices. As a result, projects that remain useful beyond the original research effort may not have the contributors, governance, documentation, or maintenance practices needed to support continued use [99]. These challenges are especially pronounced in academic settings: unlike industry open source, research software rarely benefits from dedicated engineering support, organizational backing, or the commercial incentives that sustain long-term

investment in code quality and community growth.

Previous work has proposed a wide range of characteristics and metrics for studying OSS health and sustainability. Linåker et al. [128] synthesized this literature into a taxonomy of 107 OSS health characteristics grouped into three dimensions: Actors, Software, and Orchestration. The taxonomy includes characteristics related to contributors and community structure, software quality and documentation, governance, project processes, and other aspects of project health. Other initiatives have developed more specific sets of metrics. CHAOSS defines metrics for areas such as contributor activity, organizational participation, responsiveness, and community development [40]. OpenSSF Scorecard measures repository practices related primarily to security [164], while OSS-Compass combines repository and community signals to study open source project health [169].

These efforts provide many potential measures of sustainability, but interpreting those measures is not straightforward. In particular, expectations for a project depend on its maturity. A recently created research repository is unlikely to have the same number of contributors, governance structures, release practices, or community processes as a project that has been maintained and adopted for several years. A low value for a particular metric may therefore indicate a sustainability concern in one project while simply reflecting an earlier stage of development in another. For this reason, we use the Cloud Native Computing Foundation (CNCF) project ecosystem as a reference point for comparing sustainability-related metrics. CNCF is a vendor-neutral open source foundation that hosts and stewards widely adopted cloud infrastructure projects, including Kubernetes, Prometheus, and Envoy, spanning container orchestration, observability, networking, and related domains. CNCF organizes its projects into lifecycle stages, including Sandbox, Incubating, Graduated [41], which reflect differences in project maturity across adoption, governance, community activity, and technical stability. CNCF therefore provides a set of projects for which maturity has been assessed independently of the repository metrics considered in this study.

Despite the richness of available frameworks and the well-documented challenges of research software maintenance, several important questions remain unanswered by prior work. How do the sustainability characteristics of academic OSS compare, at scale, to those of well-governed industry projects? Does maturity within the academic ecosystem correlate with better practices - and if so, how? And where, if anywhere, do academic and industry open source converge? No large-scale empirical study has examined these questions with academic OSS as its primary subject, nor has prior work done such a comparison by maturity stage. To address these gaps, in this paper, we study measurable sustainability characteristics across academic open source software from 32 institutions in the CURIOSS network [53], a community of university Open Source Program Offices whose member institutions support open source activity across their campuses.

Our dataset contains 35,336 active, development-oriented repositories. We first examine the 107 characteristics identified by Linåker et al. to determine which can be measured using GitHub-accessible data and identify appropriate formulas. We then compute these metrics for both the academic repositories and a set of CNCF projects spanning different lifecycle stages. By examining how the measures vary across CNCF maturity stages and comparing those distributions with the academic repository population, we characterize the sustainability-related practices and conditions present in academic OSS.

Our contributions are as follows.

- **C1: Operationalization of OSS sustainability characteristics.** We examine the 107 OSS health characteristics identified by Linåker et al., determine which can be measured using GitHub-accessible data, and define reproducible measures for large-scale analysis.
- **C2: Maturity classification of academic OSS.** We use CNCF lifecycle stages as a guide to classify academic repositories into maturity stages based on observable

project characteristics.

- **C3: Comparison with CNCF projects across maturity stages.** We measure the same sustainability characteristics for CNCF projects and examine how their distributions vary across lifecycle stages, providing a reference for interpreting the academic OSS population.
- **C4: Empirical characterization of academic OSS sustainability.** We measure sustainability-related characteristics across 34,629 repositories associated with 32 academic institutions and analyze how these characteristics vary across maturity stages in the academic OSS ecosystem.

The remainder of the paper is organized as follows. Section 4.2 reviews prior work on OSS health, sustainability, and project maturity. Section 7.3 describes the selection and operationalization of the metrics, the CNCF reference dataset, and the academic repository dataset. Section 7.4 presents the results of the CNCF and academic OSS analyses. Section 4.5 discusses the implications and limitations of the study, and Section 7.6 concludes.

4.2 Background and Related Work

This section provides background on the three organizational frameworks and datasets central to our study, followed by a review of three bodies of prior work. The background subsection introduces CURIOSS, the network of academic OSPOs whose repository data we analyze; CHAOSS, the open source health metrics initiative whose metric definitions we operationalize; and CNCF, the foundation whose project lifecycle stages we use as a maturity reference. The subsequent subsections review prior work on OSS sustainability metrics, OSS maturity models, and the sustainability of academic research software.

4.2.1 Background

CURIOSS. The CURIOSS network (Community for University and Research Institution OSPOs) is a consortium of university Open Source Program Offices (OSPOs) whose member institutions have demonstrated an organizational commitment to supporting and promoting open source activity on their campuses. We use data from all 32 CURIOSS member institutions because they represent a meaningful and well-scoped population of academic settings where open source is actively supported, making them a suitable subject for studying sustainability practices in academic OSS at scale. Our dataset spans 35,336 active, development-oriented repositories sourced from these institutions.

CHAOSS. CHAOSS (Community Health Analytics in Open Source Software) is a Linux Foundation project that brings together researchers, practitioners, and open source community members to define and standardize metrics for assessing open source project and community health. Founded in 2017, CHAOSS produces working group reports that define specific metrics, their rationale, and implementation guidance, covering dimensions such as contributor diversity and inclusion, project responsiveness, organizational involvement, project viability and risk, and software development activity. CHAOSS metrics have become a widely adopted reference for both practitioners evaluating project health and researchers studying open source communities at scale. We draw on CHAOSS metric definitions in our operationalization of sustainability characteristics, as they represent a practitioner-grounded consensus on measurable proxies for the conditions that support long-term project survival.

CNCF. The Cloud Native Computing Foundation (CNCF) is a vendor-neutral open source foundation under the Linux Foundation, established in 2016 to steward widely adopted cloud infrastructure projects. CNCF hosts projects spanning container orchestration, observability, networking, and related domains, including well-known projects such as Kubernetes, Prometheus, and Envoy. CNCF organizes its portfolio into a formal four-stage lifecycle: Sandbox, for early-stage projects being explored by the commu-

nity; Incubating, for projects that have demonstrated growth and adoption; Graduated, for mature projects with broad adoption, stable governance, and a diverse contributor base; and Archived, for projects that are no longer actively maintained. Stage transitions are governed by explicit criteria assessed through structured community review processes, making the CNCF lifecycle one of the few publicly available sources of externally validated project maturity labels in open source. This property is central to our study: we use CNCF stage labels as a maturity reference against which academic OSS repositories can be compared and classified.

4.2.2 OSS Sustainability Metrics

Although the literature frequently uses the terms *health* and *sustainability* interchangeably, we treat them as conceptually distinct throughout this paper. Sustainability refers to a project's capacity to be actively maintained and to continue serving its users over time, while health metrics capture current, observable conditions such as contributor activity, responsiveness, and governance practices. We retain sustainability as our primary focus but draw on health metrics because they reflect the conditions that support or threaten a project's long-term survival.

The measurement of open source project health has been an active research area for over two decades. Early work proposed multidimensional measures for assessing open source project success [51], while Schweik and English [183] studied the factors distinguishing successful from abandoned SourceForge projects. Colazo and Fang [45] examined the relationship between license choice and developer participation and development activity in OSS projects. Crowston et al. [52] developed a multidimensional framework for assessing OSS success, incorporating measures of project activity, developer participation, responsiveness, and software popularity.

Subsequent work expanded this foundation by studying project failure and survival at scale. Coelho and Valente [43] surveyed the maintainers of 104 failed GitHub

projects and identified nine recurring reasons for failure. They also found that contributing guidelines and continuous integration showed the largest differences in adoption between failed and highly popular projects. Their subsequent work [44] developed a machine-learning model to identify unmaintained GitHub projects and tracked the maintenance activity of 2,927 initially active repositories over one year, finding that 16% became unmaintained. Avelino et al. [20] developed an automated approach for estimating the Bus Factor—the minimum number of developers whose departure could incapacitate a project—and found that 65% of the 133 projects studied had a Bus Factor of two or fewer. A subsequent study [19] examined 1,932 popular GitHub projects and found that 16% experienced abandonment by their core developers; 41% of these projects subsequently survived after new core developers assumed their maintenance. Their survey further highlighted the important role of human and social factors in enabling project survival. At the ecosystem level, Valiev et al. [220] analyzed 46,547 PyPI projects and found that dependency-network position and some forms of organizational support were significantly associated with the risk of project dormancy. Ait et al. [11] extended this perspective through survival analysis of 1,127 GitHub repositories created in 2016 and tracked through 2021, finding that more than half died within their first four years and that organization-owned projects exhibited higher survival probabilities than individually owned projects.

More recently, the CHAOSS project [40] has systematized metric definitions and models spanning contributor engagement, organizational involvement, project viability and risk, and software development activity. Goggins et al. [85] grounded this effort in a four-year engaged field study, treating sustainability and survivability as distinct but related dimensions of project health and cautioning against metrics derived solely from trace data. The OpenSSF Scorecard [164] operationalizes security practices and supply-chain risk signals, including project maintenance, as automated checks scored from 0 to 10 and combined into an aggregate security score. Zahan et al. [234] evaluated Scorecard across the npm and PyPI ecosystems, finding that nine of the fifteen metrics available for large-scale analysis provided effective automated measures

of software security practices, while also identifying several metrics requiring improvement or broader industry consensus. Jansen [109] introduced the Open Source Ecosystem Health Operationalization (OSEHO), defining ecosystem health in terms of longevity and propensity for growth and operationalizing it across productivity, robustness, and niche creation—a three-dimensional structure later carried forward in OSS Compass [169]. Alami et al. [12] applied 16 sustainability metrics across four themes to 217 Apache Incubator projects, finding that community age was associated with improvements in several code-quality metrics while showing that sustainability and software quality are not consistently linked. However, existing frameworks generally do not calibrate the interpretation of health metrics to a project's age or developmental stage. Our work addresses this limitation by evaluating project health relative to stage-specific expectations rather than applying the same reference criteria across projects of different maturity.

4.2.3 OSS Maturity Models

Several researchers have proposed models for characterizing the maturity or lifecycle stage of OSS projects. Comino et al. [46] analyzed 88,192 SourceForge projects using six platform-defined development stages, from Planning to Mature, and an ordered probit model to examine how project characteristics relate to progression toward mature releases. Nakakoji et al. [146] proposed a layered model of OSS communities, later known as the onion model, comprising eight roles from passive users at the periphery to project leaders at the center. They further described an idealized progression in which contributors can move toward more influential roles through sustained participation and contributions.

The CNCF formalized a four-stage project lifecycle (Sandbox, Incubating, Graduated, and Archived) with progression and archival decisions governed through defined review processes [41]. Lu et al. [132] trained machine-learning classifiers on 24 CHAOSS-derived metrics for CNCF projects to predict Sandbox, Incubating, and Graduated life-

cycle stages.

4.2.4 Sustainability of Academic OSS

The sustainability of research software has received increasing attention. Goble [83] highlighted the fragility of scientific software developed and maintained by short-term academic personnel, emphasizing how loss of developer expertise and inadequate support can jeopardize its continued usability. Smith et al. [189] articulated six software citation principles (importance, credit and attribution, unique identification, persistence, accessibility, and specificity) providing a normative foundation for consistent software citation, attribution, identification, and subsequent bibliometric analysis. Katz and Chue Hong [111] extended this work by examining the practical implementation of software citation, emphasizing its role in ensuring that software contributions receive scholarly credit and recognition. Hasselbring et al. [93] examined how the FAIR principles (Findability, Accessibility, Interoperability, and Reusability) could be extended from research data to research software, emphasizing both archival for reproducibility and active maintenance for reuse. Barker et al. [23] subsequently introduced FAIR4RS, a set of 17 principles adapted specifically to research software and developed through a community working group jointly convened by ReSA, FORCE11, and RDA.

Linåker et al. [128] conducted a broad taxonomic review of OSS health, synthesizing 107 health characteristics from 146 studies into 15 themes spanning community actors, software, and project orchestration at both project and ecosystem levels. Their framework provides the health-characteristic vocabulary that we operationalize in this work. Wattanakriengkrai et al. [223] further illustrated the traceability gap between academic publications and research software. From more than 20,000 GitHub repositories containing references to academic papers, their qualitative analysis found that more than half of the referenced papers did not link back to any software repository. This limited bidirectional traceability complicates efforts to connect scholarly publications with their associated software artifacts. Our contribution is to stratify OSS health

characteristics by maturity stage and empirically calibrate stage-specific thresholds.

4.3 Methodology

Our methodology is divided into four parts, summarised in Figure 4.1. Part 1 filters an existing dataset of academic repositories, gathered from CURIOSS member institutions. Part 2 defines the set of sustainability metrics we measure and documents their computability. Part 3 classifies repositories into maturity stages by training a monotone ordinal classifier on CNCF projects with verified lifecycle labels and applying it to CURIOSS repositories via quantile normalization. Part 4 uses the CNCF dataset as a reference to contextualize CURIOSS sustainability metrics through comparative analysis. All GitHub activity data collected for this study corresponds to a five-year window spanning from August 2021 to August 2026.

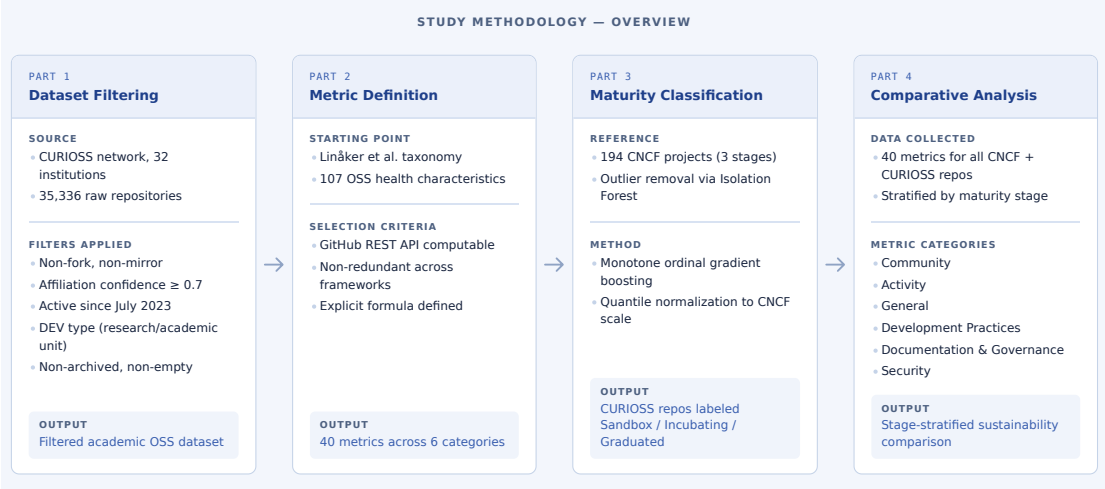

Figure 4.1: Overview of the four-part methodology. Part 1 filters the CURIOSS dataset to active, development-oriented repositories. Part 2 defines 40 sustainability metrics across six categories from the Linåker et al. taxonomy. Part 3 trains an ordinal classifier on CNCF projects and applies it via quantile normalisation to assign maturity stages to CURIOSS repositories. Part 4 compares the resulting distributions against the CNCF reference across all metric categories.

4.3.1 Part 1: Dataset Filtering

Source. Our starting point is the CURIOSS repository dataset, collected using the Repofinder pipeline [86] which identifies institutionally affiliated repositories across 32 academic institutions via keyword search and LLM-based classification.

Filtering criteria. From the raw data we apply the following filters to retain active, original development work:

- **Non-fork, non-mirror**: Exclude repositories flagged as forks or mirrors, ensuring the selection captures original institutional output rather than copies of upstream projects.
- **High institutional affiliation**: Retain only repositories with an affiliation confidence score ≥ 0.7 from the Repofinder classification pipeline, prioritizing precision over recall to minimize repositories with weak institutional ties.
- **Non-archived**: Exclude GitHub-archived repositories, which signal that a project has been explicitly marked as inactive or unmaintained by its owners.
- **Non-empty**: Exclude repositories reporting zero disk size, which correspond to placeholder or template repositories containing no committed source code.
- **Recently active**: Require at least one push event on or after 2023-07-27, restricting the dataset to projects with demonstrable activity within the 24 months preceding data collection.
- **Development type**: Retain only repositories classified as `DEV` by the Repofinder pipeline. The `DEV` category identifies repositories developed or maintained by a research group, lab, center, academic department, or institute that is formally part of the university, excluding course materials, documentation repositories, and datasets that do not constitute software development artifacts.

4.3.2 Part 2: Sustainability Metric Definition

Starting point: Linåker et al.'s taxonomy. Linåker et al. [128] surveyed 49 sources and identified 107 OSS health characteristics, of which 95 are externally visible. Each characteristic was assessed for computability from public signals (*yes, partial* or *no*) and for overlap with other frameworks (*duplicate, unique, similar* or *intersects with*).

Metric filtering. Our goal is to measure sustainability using only data accessible through the GitHub REST API, without requiring maintainer surveys or access to private signals. We therefore restrict our metric set to characteristics assessed as **yes** (fully computable). Where a characteristic is flagged as **duplicate of** or *similar to* another, we retain a single representative. Where characteristics overlap each other, we retain both only if they measure meaningfully different aspects of sustainability. After applying both filters, our final set contains **40 metrics**, each with an explicit GitHub API query or derived formula. The resulting metrics and their operationalizations are presented in Section 7.4.

4.3.3 Part 3: Maturity Stage Clustering

Repositories in the CURIOSS dataset do not have validated maturity labels. Rather than defining ad hoc categories, we adopt the CNCF lifecycle stages [41] (Sandbox, Incubating, Graduated) as our maturity taxonomy, leveraging an externally validated, industry-adopted framework. We download the full list of CNCF projects directly from the official CNCF website , which provides ground-truth maturity labels for each project. The dataset comprises 231 projects in total: 129 Sandbox, 38 Incubating, 38 Graduated, and 26 Archived. As described in Section 4.3.1, we require all repositories to have at least one push event on or after 2023-07-27 (recently active filter), which effectively excludes the archived projects: repositories that have been deprecated and are no longer maintained. Our analysis therefore considers only the three active maturity stages, yielding 205 CNCF projects with verified labels. Our approach is inspired by

Lu et al. [132], who trained a classifier on CNCF projects across these three stages using 11 repository-level CHAOSS features; as no replication package was available, we implement our own pipeline on the same feature set.

Classifier selection. We evaluate several approaches to predicting CNCF project maturity stage from sustainability indicators on CNCF data, including k-nearest neighbours, standard gradient boosting, and unsupervised clustering methods (k-means, agglomerative hierarchical clustering, Gaussian mixture models). The method that achieved best accuracy is a **monotone ordinal gradient boosting classifier** implemented via two binary
`HistGradientBoostingClassifier` models with monotonic constraints [113]: one separating Sandbox from the higher stages and one separating Graduated from the lower stages. Prior to training, we apply an outlier removal step using Isolation Forest (contamination = 5%) per maturity stage, reducing the dataset from 205 to 194 repositories. Trained and evaluated on 194 CNCF repositories with an 80/20 stratified split, the model achieves an accuracy of 80%.

Application to CURIOSS. Applying the CNCF-trained model directly to the 34,629 CURIOSS repositories assigns 99.9% of them to Sandbox (Table 4.1). CURIOSS repositories operate at substantially lower absolute values across nearly all features compared to even CNCF Sandbox projects, so the classifier's learned thresholds place virtually the entire CURIOSS dataset below the Sandbox boundary. The 48 repositories classified as Incubating or Graduated by the direct model are, by this measure, among the most mature academic OSS projects in the dataset. Applied to the full population, however, the model is uninformative, motivating the quantile normalization approach described below.

Table 4.1: Predicted maturity stage distribution for CURIOSS repositories ($n = 34{,}629$) using the CNCF-trained classifier applied directly to raw feature values.

Stage	Count	%
Sandbox	34,581	99.9%
Incubating	31	0.1%
Graduated	17	0.05%
Total	34,629	100%

Quantile normalization. To address the scale mismatch, we apply **quantile normalization** before classification. For each of the 11 features, we compute each CURIOSS repository's percentile rank *within the CURIOSS distribution* and map that rank to the value at the same percentile in the CNCF training distribution. For example, a CURIOSS repository at the 80th percentile of CURIOSS star counts is assigned the CNCF star count value at the 80th percentile. This transforms the classification question from "does this repository meet absolute CNCF thresholds?" to "how mature is this repository relative to its CURIOSS peers, assessed on the CNCF scale?"

4.3.4 Part 4: Comparative Baseline Analysis Against CNCF Projects

CNCF as a Sustainability Baseline

We use the 194 CNCF repositories retained after outlier removal in Part 3 as our sustainability reference. For each repository we compute the full set of 40 sustainability metrics defined in Part 2 using the same data collection pipeline applied to the CURIOSS dataset. The resulting CNCF distributions, stratified by maturity stage, provide a contextual reference against which academic OSS practices can be interpreted. While the substantial differences in scale between the two ecosystems preclude direct comparison of absolute metric values, the CNCF stage distributions help identify where academic projects lag behind industry practice and where, if at all, they converge.

4.4 Results

We present results for each stage of the methodology described in Section 7.3.

4.4.1 Filtered Data

Table 4.2 summarizes the repository counts at each stage of the filtering pipeline, which aims to retain only repositories with sufficient activity and data quality for sustainability analysis.

Table 4.2: CURIOSS repository counts at each stage of the collection and filtering pipeline.

Stage	**Repositories**
Candidate repositories	761,920
After cleaning (remove forks, archived, empty, zero-size)	529,879
After activity & affiliation filtering (score ≥ 0.7)	216,920
After keeping only `DEV`	34,629

Of the 216,920 repositories identified across 32 CURIOSS institutions after filtering, 34,629 are classified as `DEV` (research tools and development) repositories (Section 4.3.1) and form the dataset for the sustainability analysis.

4.4.2 Sustainability Metrics

After filtering Linåker et al.'s 107 health characteristics to those fully computable from the GitHub API and removing duplicates and redundant overlaps, we retain **40 metrics** organized into six categories: Community, Activity, Popularity, Development Practices, Docs, and Security. Table 4.3 summarizes each metric and its rationale; the precise computational formulas are provided in Appendix B.1.

Many metric definitions in the literature specify *that* a value should be measured over

a recent observation window without prescribing a fixed duration, phrasing such as "within a specified time frame" is common across CHAOSS metric specifications [40]. Where a source does define a specific window (e.g., Alami et al. [12] use 90-day periods for contributor-based metrics), we adopt it directly. For metrics whose sources leave the window unspecified, we standardize on **90 days**, consistent with the default observation period used across CHAOSS metric definitions [40]. Structural metrics that accumulate over a project's lifetime, such as unique contributors and bus factor are computed over the full five-year collection window (August 2021 – August 2026).

Table 4.3: Sustainability metrics by category.

Metric	**Description**	**Source**
Community		
Unique contributors	Distinct logins across commits, PRs, issues, and PR reviews (all-time)	[12]
Active contributors	Distinct commit authors active within W_{90}	[40]
Organizational diversity	Distinct non-personal email domains among commit authors in W_{90}	[40]
Contributor retention	% of contributors from prior W_{90} still active in current W_{90}	[12]
Contributor growth	% change in distinct contributor count between consecutive W_{90} windows	[12]
Bus factor	Smallest set of contributors whose commits exceed 50% of total	[20]

Continued on next page

Table 4.3 – *continued*

Metric	**Description**	**Source**
Activity		
Social activity	Sum of all interaction events (commits, issues, PRs, comments) in W_{90}	[40]
Commit frequency	Mean weekly commit count over W_{90}	[40]
Commit growth	% change in commit count between consecutive W_{90} windows	[40]
Issue closed ratio	Fraction of all issues that have been closed (all-time)	[40]
Issue response time	Mean days from issue creation to first response (all-time)	[40]
PR response time	Mean days from PR creation to first review (all-time)	[40]
PR merge time	Mean days from PR creation to merge (all-time)	[40]
Code review ratio	Fraction of all merged PRs that received at least one review (all-time)	[169]
Release frequency	Mean monthly release count over W_{90}	[40]
Contributors dev. activity	Total events by non-core contributors in W_{90}	[12]
Maintainers dev. activity	Total events by core contributors in W_{90}	[12]
Maintainer recency	Days since most recent activity by any core contributor	[128]

Continued on next page

Table 4.3 – *continued*

Metric	Description	Source
General		
Project age	Days since repository creation	[12]
Financial support	Presence of GitHub funding configuration (`FUNDING.yml`)	[128]
Community interest	Sum of GitHub stargazers, forks, and watchers	[12]
Development Practices		
CI/CD	Presence of continuous integration configuration	[43, 164]
Build environment	Number of recognized build system files present	[128]
Test presence	Presence of a test directory	[164, 40]
Platform OS support	Number of distinct OS targets across CI workflow definitions	[128]
Documentation & Governance		
README	Presence of top-level README	[43]
CONTRIBUTING	Presence of contribution guidelines	[43, 128]
Code of conduct	Presence of community code of conduct	[204]
Governance	Presence of governance document	[128, 161]

Continued on next page

Table 4.3 – *continued*

Metric	Description	Source
Issue template	Presence of issue templates	[43]
PR template	Presence of pull request template	[43]
Documentation completeness	Count of core documentation files present (0–3)	[43, 161]
Changelog	Presence of changelog file	[128]
Code review policy	Enforced code review via CODEOWNERS or branch protection	[164]
License	Repository has a recognized open source license	[128]
Security		
Security policy	Presence of a security vulnerability disclosure policy, as measured by OpenSSF Scorecard	[164]
Branch protection	Default branch has protection rules enabled, as measured by OpenSSF Scorecard	[164]
Signed releases	Releases are cryptographically signed (GPG or Sigstore), as measured by OpenSSF Scorecard	[164]
Dependency updates	Automated dependency update tool present, as measured by OpenSSF Scorecard	[164]
Known vulnerabilities	Number of open, unfixed vulnerabilities reported by OpenSSF Scorecard (via OSV)	[164]

4.4.3 Maturity Stage Clustering

Table 4.4 reports the resulting stage distribution after applying the CNCF-trained classifier to the quantile-normalized CURIOSS features. Approximately 18.5% of CURIOSS repositories are classified as Incubating or Graduated, representing projects that, relative to the broader academic ecosystem, display levels of activity and community engagement comparable to the corresponding CNCF lifecycle stages.

Table 4.4: Predicted maturity stage distribution for CURIOSS repositories ($n = 34{,}629$) using the CNCF-trained classifier applied to quantile-normalized feature values.

Stage	**Count**	**%**
Sandbox	28,222	81.5%
Incubating	3,067	8.9%
Graduated	3,340	9.6%
Total	34,629	100%

For interpretability, we retain the CNCF stage labels—*Sandbox*, *Incubating*, and *Graduated*—for the three resulting clusters. These labels are used solely as *relative maturity categories* within the academic open source ecosystem. They do not indicate, and should not be interpreted as equivalent to, the corresponding official CNCF project stages.

4.4.4 Comparative Sustainability Analysis: CURIOSS vs. CNCF

We compute all sustainability metrics for both the 194 CNCF reference projects (after outlier removal) and the 34,629 CURIOSS repositories, comparing their distributions across the three maturity stages (Sandbox, Incubating, and Graduated). Here we highlight the metrics that reveal the most relevant differences and convergences between academic open source and the CNCF ecosystem.

Community Metrics

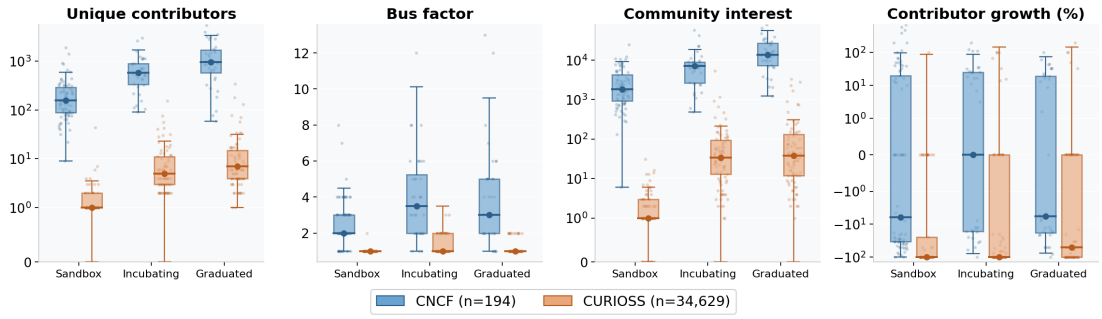

Figure 4.2: Box-and-whisker plots of four community metrics — unique contributors, bus factor, community interest, and contributor growth (90-day window) — for CNCF (blue) and CURIOSS (orange) repositories, grouped by maturity stage. Each box spans the IQR (25th–75th percentile); the horizontal line and dot indicate the median; whiskers use 1.5×IQR. Community interest and unique contributors use a symmetric-log scale to accommodate large dynamic range. Contributor growth uses a symmetric-log scale to accommodate both zero and negative values. Bus factor is shown on a linear scale.

Figure 4.2 presents the distribution of four community metrics across maturity stages.

Unique contributors. The gap between the two ecosystems is immediate and extreme. CURIOSS Sandbox repositories have a median of just 1 unique contributor (IQR: 1–2), compared to 158 for CNCF Sandbox (IQR: 89–290). Even at the Graduated tier, CURIOSS projects reach a median of only 7 unique contributors (IQR: 4–15), while CNCF Graduated projects have a median of 960 (IQR: 577–1,694). This order-of-magnitude difference reflects a fundamental structural distinction: most CURIOSS repositories are solo or small-team research projects and haven't reached broad community participation.

Bus factor. The bus factor behaves similarly. CURIOSS repositories have a median bus factor of 1 at every maturity stage, including Graduated, where only 22% of projects exceed a bus factor of 1. CNCF projects have a median bus factor of 2 at Sandbox (IQR: 2–3), 3.5 at Incubating (IQR: 2–5), and 3 at Graduated (IQR: 2–5). A bus factor of 1

means that only one developer is responsible for more than 50% of the project's development. That this value does not improve with maturity within the academic ecosystem is a particularly concerning finding: even the most active CURIOSS projects remain critically dependent on a single contributor, suggesting that contributor concentration is not self-correcting as projects age. Bus factor fragility is not entirely absent in CNCF either, a minority of projects, including some Graduated ones, also fall to a bus factor of 1, but it is the norm, rather than the exception, in the academic dataset.

Community interest. Measured as the sum of stars, forks, and watchers, community interest shows the largest absolute gap of any metric examined. CURIOSS Sandbox repositories have a median of 1 (IQR: 1–3) versus 1,823 for CNCF Sandbox; CURIOSS Graduated projects reach a median of 38 (IQR: 12–130) compared to 13,548 for CNCF Graduated. The comparison is not entirely fair, since CNCF membership functions as a visibility signal and foundation-backed projects tend to accumulate broad industry adoption. That said, only 0.7% of CURIOSS repositories (236 out of 34,629) exceed a community interest score of 1,000, yet these projects demonstrate that academic open-source software can achieve visibility and community engagement comparable to industry-backed projects, showing that broad adoption is an attainable outcome within the academic ecosystem.

Contributor growth. CURIOSS Sandbox and Incubating repositories have a median contributor growth rate of -100% ((i.e., no contributor from the prior window returned in the current window; IQR: -100% to -25% and -100% to 0% respectively), meaning the typical academic project is actively losing contributors over any given 90-day window. CURIOSS Graduated projects improve to a median of -50% (IQR: -100% to 0%). By contrast, CNCF medians range from -6.1% at Sandbox to -5.8% at Graduated, with IQRs that span from small decline to small growth. Notably, even CNCF projects trend slightly negative on this metric, suggesting that short-window contributor attrition is common across open source generally, but CURIOSS institution-affiliated projects experience it far more acutely.

Activity Metrics

Figure 4.3 presents the distribution of three activity metrics across maturity stages.

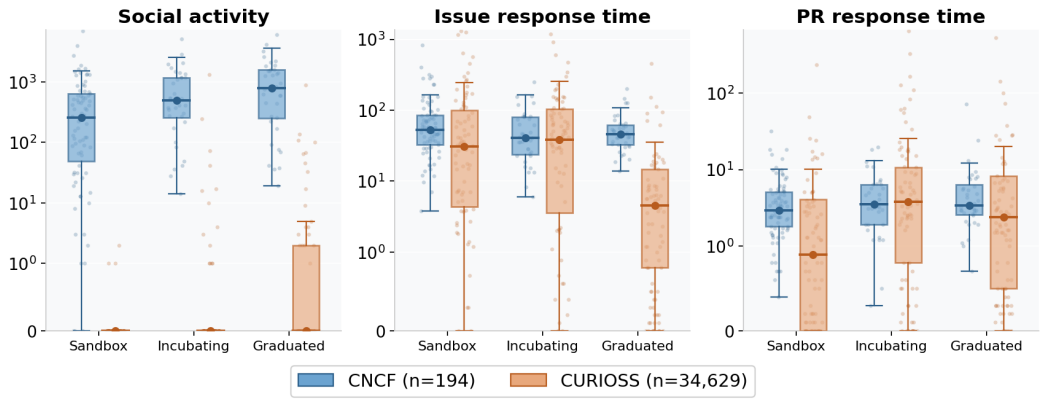

Figure 4.3: Box-and-whisker plots of three activity metrics — social activity, issue response time, and PR response time — for CNCF (blue) and CURIOSS (orange) repositories, grouped by maturity stage. All three metrics use a symmetric-log scale. Response times are measured in hours; lower values indicate faster responses. Boxes are drawn only for repositories with non-zero values; the fraction of repositories contributing to each box is noted in the text.

Social activity. Social activity, the total number of issues, pull requests, and comments within a 90-day window is near zero for the overwhelming majority of CURIOSS repositories at every maturity stage. The median is 0 across all three CURIOSS tiers, with only 6.3% of Sandbox, 18.8% of Incubating, and 31.7% of Graduated CURIOSS repositories recording any social interaction at all. Among those that do, activity remains low: the 75th percentile for CURIOSS Graduated reaches roughly 60–80 events, compared to CNCF medians of 257 (Sandbox), 488 (Incubating), and 787 (Graduated). The low rates of social activity reflect that most academic repositories are not community-facing projects; they function as code artifacts accompanying research, with no expectation of ongoing issue or pull request engagement.

Issue and PR response time. Among the minority of CURIOSS repositories that do have issue or pull request activity, response times are surprisingly competitive with CNCF. Issue response time (in hours) has a CNCF median of 53h, 41h, and 46h at

Sandbox, Incubating, and Graduated respectively. CURIOSS repositories with issue data respond in a median of 31h (Sandbox), 38h (Incubating), and just 4.5h (Graduated). PR response times follow a similar pattern: CNCF medians are 2.9h, 3.5h, and 3.4h; CURIOSS medians (among the subset with PR activity) are 0.9h, 3.8h, and 2.4h. These figures suggest that academic projects which do attract community engagement respond to it quickly, consistent with small, closely-knit teams monitoring a low volume of interactions. The critical caveat, however, is that these statistics characterise only a small active tail of the CURIOSS distribution: just 3.3% of CURIOSS Sandbox repositories and 73% of Graduated repositories have any issue response data at all, making these figures a property of the most engaged academic projects rather than a representative picture.

Documentation and Governance

Figure 4.4 shows the overall adoption rate of nine Documentation & Governance metrics across both datasets, collapsed across maturity stages because preliminary inspection revealed that the gap between CNCF and CURIOSS was consistently large regardless of stage, making a per-stage breakdown uninformative. The contrast is clear. README presence is the only metric where CURIOSS is comparable to CNCF (89% vs. 99%), confirming that basic file presence is a near-universal minimum even in academic repositories. Every other metric reveals a severe gap: contributing guides (CNCF 80%, CURIOSS 2%), code of conduct (CNCF 61%, CURIOSS 1%), governance documentation (CNCF 50%, CURIOSS 0%), issue templates (CNCF 80%, CURIOSS 2%), PR templates (CNCF 66%, CURIOSS 2%), changelog (CNCF 32%, CURIOSS 2%), and code review policy (CNCF 48%, CURIOSS 1%). These near-zero rates indicate that the governance and community-onboarding infrastructure that enables sustained collaboration is almost entirely absent in the academic ecosystem, across all maturity stages.

License. License adoption is measured here as whether the GitHub REST API was

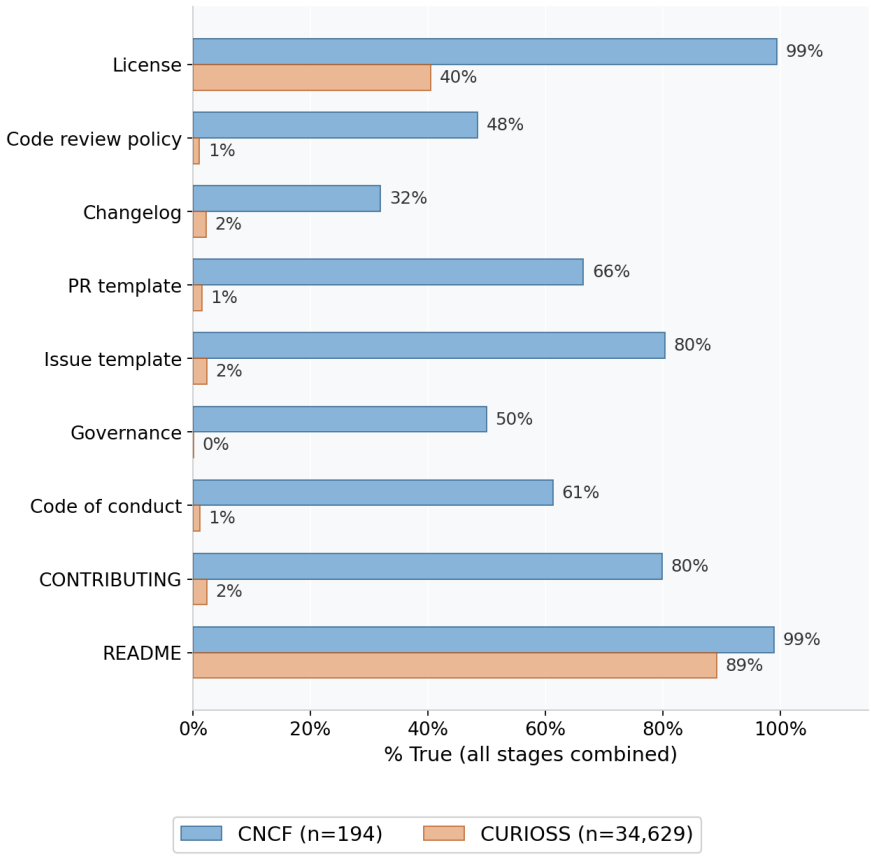

Figure 4.4: Percentage of repositories satisfying each Documentation & Governance metric across all maturity stages combined, for CNCF (blue) and CURIOSS (orange) projects. All metrics are binary (presence or absence).

able to detect a machine-readable license file in the repository; a project that describes its terms only in prose is counted as unlicensed by this metric. At 40% overall for CURIOSS (versus 99% for CNCF), license adoption warrants a closer look by maturity stage. As shown in Figure 4.5, license presence rises from 33% at CURIOSS Sandbox to 68% at Incubating and 75% at Graduated, the strongest maturity gradient of any metric in this category. This suggests that longer-lived and more active academic projects are substantially more likely to have addressed this baseline legal requirement, possibly driven by journal or conference requirements that mandate open licensing at the time of paper submission. It is worth noting that some academic repositories are not intended

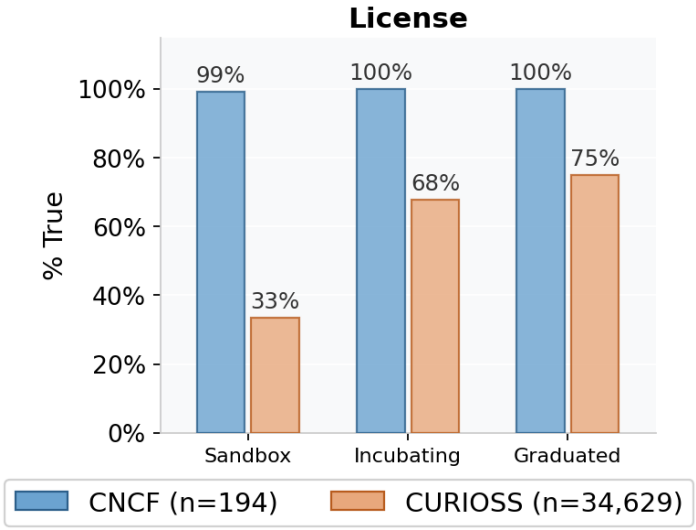

Figure 4.5: Percentage of repositories with a license file, broken down by maturity stage, for CNCF (blue) and CURIOSS (orange) projects.

as open-source distributions and the absence of a license may be deliberate rather than an oversight, which makes this metric less informative for the Sandbox population. The concern is more acute for Incubating and Graduated projects, which have already attracted external interest: among CURIOSS Graduated repositories without a detectable license file, several have accumulated substantial community interest (more than 1000 between stars, forks and watchers). In one of the cases, the license terms are mentioned in the README but no dedicated license file exists, so the GitHub API returns no license information and the terms remain invisible to automated dependency scanners and human contributors alike. This illustrates a broader point: a license should be placed in a standard, machine-readable location rather than embedded in prose, so that downstream users can resolve reuse rights without manual inspection. OSPO support could readily address this class of issue across the academic ecosystem.

Development Practices

Figure 4.6 shows development practice adoption across maturity stages for both ecosystems.

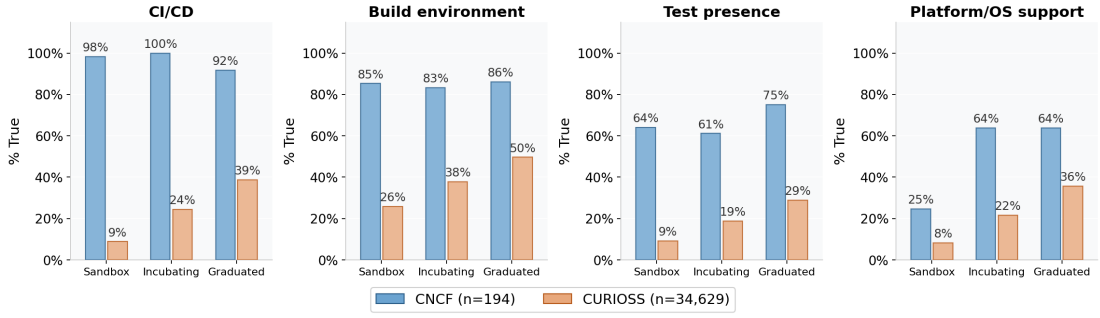

Figure 4.6: Adoption rates of development practice metrics by maturity stage for CNCF (blue) and CURIOSS (orange) repositories. CI/CD and test presence are binary (bars show % of repositories); build environment and platform OS support show the percentage of repositories with any configured support detected.

CI/CD. Continuous integration is configured in 98% of CNCF Sandbox repositories and 92% of CNCF Graduated repositories. CURIOSS Sandbox repositories show only 9% CI/CD adoption, rising to 24% at Incubating and 39% at Graduated — meaningful improvement but still far from the near-universal adoption seen in the CNCF ecosystem. The positive trend within CURIOSS suggests that more sustained projects do adopt automation practices over time, but the majority of academic repositories at every stage remain without automated pipelines.

Build environment and platform OS support. Build environment detection (whether a reproducible build configuration is present) shows a clearer maturity progression in CURIOSS: 26% at Sandbox, 38% at Incubating, and 50% at Graduated, against a near-flat CNCF rate of 83–86%. Platform OS support follows a similar trajectory in CURIOSS (8%, 22%, 36%) but with a more pronounced gap relative to CNCF, where it also rises with maturity (25%, 64%, 64%). The lower CNCF Sandbox rate for platform support (25%) relative to other CNCF metrics is worth noting: early-stage CNCF

projects have not yet necessarily standardised their build targets, and this settles by the Incubating stage.

Test presence. A detectable test suite is present in 64% of CNCF Sandbox, 61% of CNCF Incubating, and 75% of CNCF Graduated repositories. CURIOSS shows 9%, 19%, and 29% respectively. The absolute rates remain low even at Graduated, suggesting that test infrastructure is the development practice with the widest persistent gap — despite being, alongside CI/CD, one of the most directly impactful practices for long-term code maintainability.

Security

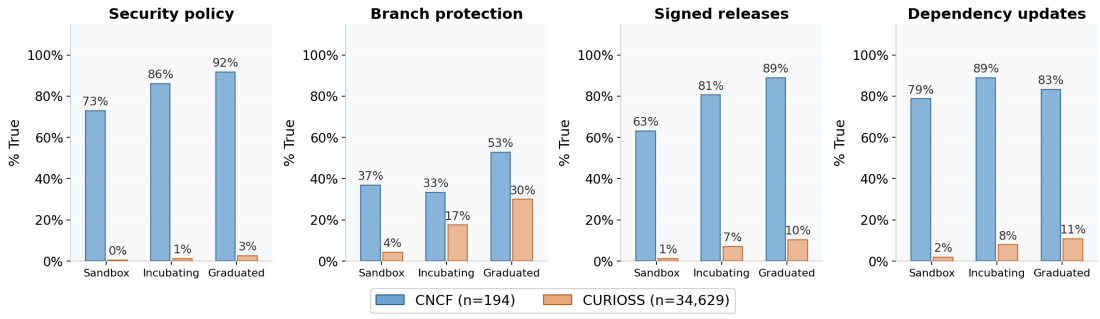

Figure 4.7: Adoption rates of four security metrics by maturity stage for CNCF (blue) and CURIOSS (orange) repositories. All metrics are binary (presence or absence, assessed via OpenSSF Scorecard).

Figure 4.7 reveals the largest and most uniform gaps of any category examined. Security practice adoption in CURIOSS is negligible at the Sandbox tier and only modestly better at Graduated.

Security policy. A documented security vulnerability reporting policy is present in 73–92% of CNCF repositories across stages but in only 0.4% (Sandbox), 1.2% (Incubating), and 2.7% (Graduated) of CURIOSS repositories. The near-zero rates at all stages suggest this is not a practice that emerges organically with maturity — it requires

an external prompt or requirement.

Branch protection. This is the security metric with the highest CURIOSS adoption, rising from 4% (Sandbox) to 18% (Incubating) and 30% (Graduated). CNCF rates are 37%, 33%, and 53% respectively. The convergence is partial: CURIOSS Graduated approaches the lower end of the CNCF range, suggesting that branch protection is a practice more likely to be adopted by active teams than the others.

Signed releases and dependency updates. Both practices remain rare in CURIOSS across all stages. Signed releases reach only 10% at CURIOSS Graduated versus 89% at CNCF Graduated. Automated dependency update tooling (e.g., Dependabot) is present in 1.9% of CURIOSS Sandbox repositories and only 10.7% at Graduated, compared to 79–89% across CNCF stages. Together, signed releases and dependency update tooling are the security baseline most universally required by CNCF, and the gap here likely reflects both the tooling investment required and the absence of external governance mandating them in academic settings.

4.5 Discussion

We frame our discussion around four research questions that synthesise the findings across all metric categories.

RQ D1: Is academic open-source software as sustainable as a well-governed industry baseline? The answer is a clear and consistent *no* across the large majority of metrics examined. CURIOSS repositories lag CNCF projects substantially in community size, development practices, governance, and security at every maturity stage. The most pronounced gaps are in governance infrastructure (code of conduct $<1\%$, governance documentation 0%) and security (security policy below 3% even at Graduated), areas where CNCF imposes explicit requirements on member projects. The comparison

makes explicit what is missing and where targeted intervention would have the highest leverage. The key takeaway is: *academic open-source repositories are, on average, far from self-sustaining, and the gap is structural, not incidental.*

RQ D2: Does project maturity within the academic ecosystem predict better sustainability practices? Yes, with important caveats. Across almost every metric, higher-maturity CURIOSS repositories outperform their academic Sandbox counterparts: CI/CD adoption rises from 9% to 39%; license presence from 33% to 75%; unique contributors from a median of 1 to 7; branch protection from 4% to 30%. The maturity progression is real and consistent. The caveat is that the absolute levels remain low: even CURIOSS Graduated projects, the top $\sim$10% of the distribution by activity, still have a median bus factor of 1, fewer than 10 unique contributors, and no security policy in 97% of cases. Reaching the Graduated tier within the academic ecosystem means being active and relatively well-managed by academic standards; it does not mean approaching CNCF-level sustainability.

The key takeaways are two. First, *maturity predicts improvement in absolute terms, but even the most active academic projects have not converged on the engineering baseline that industry considers standard.* Second, and perhaps more fundamentally, *applying a single sustainability benchmark across both ecosystems reveals the limits of a one-size-fits-all approach*: academic repositories operate under different incentive structures, staffing constraints, and lifecycle expectations than industry-backed projects, and a framework calibrated on CNCF projects will systematically classify the vast majority of academic software as immature, regardless of its actual value or fitness for purpose within its intended context.

RQ D3: Where do academic and industry open-source projects converge, and what does this reveal? Two areas stand out. First, *README presence* is near-universal in both ecosystems (CURIOSS 89%, CNCF 99%), suggesting that the prac-

tice of providing minimal project documentation is successfully adopted even without external mandates. Second, among the minority of CURIOSS repositories that generate any issue or pull request activity at all, *response times are competitive with CNCF* (CURIOSS Graduated: median 4.5h for issues vs. CNCF 46h; PR response: 2.4h vs. 3.4h). This counterintuitive finding reflects the dynamics of small, closely monitored teams: a project with few contributors and few issues may respond to each one more quickly than a large project where contributions queue across a distributed maintainer team. The key takeaway is: *convergence occurs precisely where practices are lowest-cost (a README) or where small-team dynamics are an advantage (fast response to the occasional issue), not where sustained investment or external governance is required.*

RQ D4: What should be the priority areas for OSPOs and institutions seeking to improve the sustainability of academic open source? The findings point to three priority areas differentiated by maturity stage. At the **Sandbox** level, the most tractable intervention is *documentation and licensing*: README adoption is already high, but contributing guides (2%), code of conduct (1%), and license files (33%) are achievable with near-zero marginal cost if project templates are mandated at the point of repository creation. At the **Incubating** level, the priority shifts to *development infrastructure*: CI/CD at 24% and test presence at 19% leave substantial room for tooling-level support such as pre-configured workflow templates or RSE office hours. At the **Graduated** level, the remaining gaps, bus factor, security practices, and governance documentation, require *sustained human investment*: contributor mentoring, structured onboarding, and the adoption of security tooling that does not emerge from project activity alone.

A uniform intervention across all repositories is unlikely to be efficient; the data suggest a tiered model in which lightweight template-based interventions target the long tail of Sandbox repositories, while more intensive OSPO support is reserved for the smaller population of Incubating and Graduated projects with demonstrated adoption potential. The key takeaway is that no single intervention fits all repositories. Documentation

templates and CI guides suit the majority; hands-on governance support is reserved for the mature few. Matching intervention to maturity stage is the most resource-efficient path forward.

4.6 Conclusions

This paper presented a large-scale empirical comparison of open-source software sustainability between the academic ecosystem, represented by 34,629 repositories from 32 CURIOSS member institutions, and the CNCF ecosystem spanning three maturity stages. We operationalized sustainability through 40 metrics drawn from the Linåker et al.[128] taxonomy, selected for their computability via the GitHub REST API and their non-redundancy across existing frameworks. Then we used a monotone ordinal gradient boosting classifier, trained on CNCF projects and calibrated through quantile normalisation, to assign CURIOSS repositories to Sandbox, Incubating, or Graduated maturity stages.

Results show that academic repositories lag behind their CNCF counterparts across basically every sustainability dimension, with the biggest gaps in governance documentation, security practices, and community scale. At the same time, the comparison reveals meaningful variation within the academic ecosystem: maturity stage is a genuine predictor of sustainability, with CI/CD adoption, license presence, unique contributors, and branch protection all rising substantially from Sandbox to Graduated.

Two broader conclusions emerge. First, open-source sustainability frameworks calibrated on a single reference ecosystem will systematically underrank software from other contexts, not because those projects are poorly maintained, but because they operate under fundamentally different incentive structures, staffing models, and lifecycle expectations. Second, the results point to actionable interventions: the practices where academic projects lag most severely, specifically license files, security policies, contributing guides, and code review policies, are low-effort additions that university

OSPOs are well-positioned to facilitate through templates, training, and policy. The gap is large, but the barrier to closing it for the most promising projects is not technical.

4.6.1 Limitations

Several limitations of this study should be acknowledged.

Data source and representativeness. The academic repositories analyzed in this study represent a sample of 32 institutions and should not be treated as a comprehensive census of academic open- source software. Conclusions about the broader academic ecosystem should be drawn with caution, as the sample may not capture the full range of institutional contexts, disciplines, or levels of open-source maturity present across universities worldwide.

GitHub API measurement constraints. All 40 metrics are computed from the GitHub REST API. License detection relies on the presence of a standard license file in the repository root; projects that document their license terms in a README or in a non-standard location are counted as unlicensed. Similarly, binary metrics such as governance documentation and code review policy detect the presence of specific files or fields, not the quality or completeness of their contents. A repository with a placeholder `CONTRIBUTING.md` is indistinguishable from one with detailed onboarding instructions in a different place such as a README.

CNCF as a reference benchmark. The CNCF lifecycle was designed for cloud-native infrastructure software with large, distributed contributor communities. Using it as a universal sustainability reference introduces a systematic bias: metrics that matter for cloud-native adoption, such as signed releases, dependency update automation, and governance documentation, may be less relevant for scientific software, simulation

tools, or data pipelines that are designed for a narrower user base. Future work should explore whether domain-specific maturity models produce more actionable assessments for academic software.

Classifier and quantile normalisation. Assigning a maturity stage to a software project is an inherently difficult problem with no universally accepted solution, and our classifier should be understood as one reasonable approximation rather than a ground truth. The model is trained on 194 CNCF projects, a relatively small and domain-specific sample, and the quantile normalisation step adjusts CURIOSS metrics to match CNCF distributions in ways that may hide real differences between the two ecosystems. The resulting stage assignments are best read as relative rankings within the academic ecosystem, indicating which projects are more or less mature compared to each other, rather than as definitive labels of sustainability.

Scope of metrics. The 40 metrics cover six categories but cannot capture every dimension of sustainability. Factors such as institutional funding stability, contributor diversity, or alignment with active grant cycles all affect the long-term viability of academic software and are not reflected in repository metadata alone.

Part II

Security

Chapter 5

Software Supply Chain Literature Review

Securing the software supply chain requires understanding both the research landscape and the tools available to practitioners. This chapter provides that grounding. We first define the software supply chain and its key stages, then survey the academic literature on supply chain security, organized by research objective and method. We conclude with a taxonomy of tools developed to address supply chain threats, grouped into four categories: component analysis, Software Bills of Materials (SBOMs), software integrity, and best practices. Together, these two surveys establish the context for the security contributions in Chapters 6 and 7.

5.1 Background

The software supply chain is made up of everything and everyone that touches code in the software development lifecycle (SDLC), from application development to the CI/CD pipeline and deployment [3]. Any tangible output or component produced during the SDLC is called an artifact and we say that the software supply chain is the

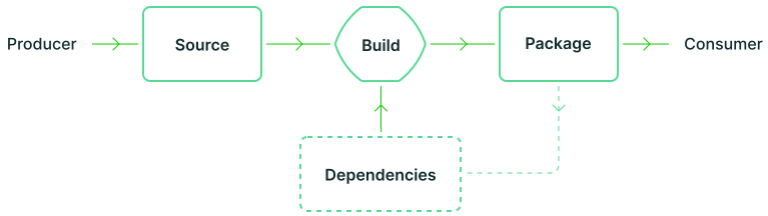

Figure 5.1: Software supply chain [6]

sequence of steps for creating them [6]. An artifact's supply chain is a combination of its dependencies' supply chains plus its own sources and builds.

Source: It is the beginning of the supply chain, and it refers to creating the "source code" for an artifact, which is usually stored in a repository hosted on a developer platform like GitHub.

Build: Refers to the process of transforming input artifacts (source code files) into output artifacts. For example the .travis.yml (which specifies the project's build configuration) is run by the Travis CI platform to build projects in GitHub.

Package: Refers to the artifact that results as an output of a build process and is "published" for use by others. These packages can be found in different package managers like PyPI (Python Package Index) for Python packages or NPM (Node Package Manager) for JavaScript.

5.2 Literature Review

Supply chain software security has become a prominent issue in recent years, prompting numerous studies that encompass multiple topics from risk assessment to Software Security Practices and even the impact of human factors in the supply chain. In this section, we aim to show the different topics within the supply chain that studies have presented, classified into nine different categories based on the objective of their stud-

ies, and six categories based on the methods used, as displayed in table 5.1. Although there are numerous works like [94] that discuss the security of supply chains in general, we only consider papers focused on software supply chain.

5.2.1 Risk Assessment

Multiple works study the software supply chain from the attack perspective in order to develop comprehensive risk assessments, which are crucial for identifying and mitigating potential vulnerabilities. These assessments help organizations understand and address the risks associated with supply chain attacks, ensuring the security and resilience of their software systems. [118] for example presented a taxonomy of open source supply chain attacks in the form of an attack tree and validated the results with expert surveys. This work can serve as reference point and common terminology regarding supply chain attacks. Other works instead, performed empirical studies that are valuable to understand the current risks in supply chain software. [231] for example, explored the attack surface of supply chain residual vulnerabilities in 50 open source projects while [120] went further into developing a visualization tool, *Risk Explorer*, to explore the taxonomy of attack vectors.

5.2.2 Supply Chain Security Overall

Numerous studies have explored the broader challenges and strategies for enhancing software supply chain security overall. [65] discussed software supply chain security challenges, summarizing information collected during three summits with 30 industry and government organizations. These include updating vulnerable dependencies, leveraging SBOMs for security, choosing trusted dependencies, securing build processes and achieving adoption of security tools and metrics. Similarly, [224] presented a qualitative study of the challenges of open source supply chain based on the results of surveys with software developers, architects, and engineers. However, this latter one focuses on

the implications for company stakeholders. [13] also presents a more enterprise-driven perspective. By performing a mining study on OSS repositories and survey a sample of project maintainers it aims to better understand the role of automated security activities in CI pipelines of enterprise open source software. [141] on the other hand proposes a tool that implements articulated trust, by introducing several new security techniques to improve supply chain security overall. These include per-package prioritization of repositories, multi-role delegations, multiple-repository consensus, and key pinning.

5.2.3 Vulnerability Management

Other works in software supply chain security focus on a specific part of the supply chain process. Vulnerability management for example, emphasizes identifying and addressing security flaws in software components. This is important because unpatched vulnerabilities can serve as entry points for attackers, potentially compromising the entire supply chain and leading to significant security breaches like the xz utils one. Some studies in this area targeted specific languages like [64] and [54] which present empirical studies on vulnerability scanners and vulnerability detection techniques on a Java respectively. Others, like [103] which studied vulnerability patches and [61] which presented a new tool to discover vulnerability fixes leveraging static analysis security testing (SAST) tools, presented a language agnostic approach. Finally, within this category, we also find development of datasets like the one in [173] which presented fixes to vulnerabilities in Java OSS as a open source vulnerability management solution.

5.2.4 Dependency Updates

A significant challenge in supply chain security is dependency management. When updating a dependency, it is crucial to ensure that the correct, uncompromised version is selected. [158] presents a Systematization of Knowledge (SoK) study that reviews academic papers discussing various approaches for dependency updates. The study

concludes that there is no one-size-fits-all solution and that none of the current methods provide a comprehensive solution to detect malicious code during dependency updates. Tools like Dependabot [77] have been developed to address this issue. However, as demonstrated in the empirical study by [95], Dependabot has limitations and missing features that lead developers to either deprecate it or restrict its notifications.

5.2.5 Software Bill of Materials (SBOMs)

Software Bill of Materials (SBOMs) have emerged as a critical solution for tracking dependencies and understanding software components. SBOMs function as detailed lists of ingredients that comprise software components [10]. The academic interest in SBOMs has significantly increased following Executive Order 14028 in May 2021, which mandated that software vendors to the US government list the components used to create their products using SBOMs. Various studies have highlighted the benefits of SBOMs [36, 16], their role in enhancing software supply chain security [134], and methods to improve their quality [202].

In addition, numerous user research and empirical studies have investigated the accuracy [22] and adoption [154] of SBOM tools, their role in software projects [29], their relevance for different customers [38], as well as adoption [230] and creation challenges [193]. A detailed study by [235], which reviewed 200 internet articles, concluded that as of 2022, SBOMs were inadequate for large-scale adoption. Furthermore, [15] developed a systematization of knowledge about SBOM tools and their selection based on user requirements, complemented by [184], who built a formal taxonomy on SBOM generation approaches.

5.2.6 Software Integrity

Ensuring software integrity is another crucial aspect of supply chain security. Tools like Sigstore [152] (for signing software), in-toto [203] (for verifying software's lifecycle),

and Scudo [142] (for securing vehicle software) have been developed to enhance the verification process and protect against tampering. We delve deeper into these tools in section 5.3.2.

5.2.7 Security Practices

Another approach to studying supply chain security involves examining the security practices of various software projects. Using the Scorecards tool, which will be further explained in section 5.3.4, [236] and [234] investigate the adoption of security practices in these projects. For instance, [237] introduces a novel technique for detecting weak link signals in the supply chain through an empirical study on JavaScript npm packages. Additionally, secure design properties, as discussed by [160], play a crucial role in maintaining a secure supply chain. Finally, [14] outlines strategies for integrating software supply chain security measures into CI/CD pipelines, providing a comprehensive approach to enhancing security throughout the software development lifecycle.

5.2.8 Human Factors

Another vital area of study within supply chain security is the impact of human factors, given that developers are integral to the supply chain and their actions affect a project's security. Some researchers argue that the usability of supply chain security measures, especially for developers, remains an under-investigated area [72]. They propose a research agenda to address this gap, highlighting the critical role of human-centered approaches in supply chain security. For example, [135] highlights the need for a common language to effectively communicate security requirements and [227] stresses the importance of educating developers about the risks associated with open source software and providing them with guidance and risk-based tools to mitigate these risks. This research area is very important since enhancing human factors is an essential step toward strengthening supply chain security in software development.

5.3 Tools Taxonomy

The development of tools for supply chain security has been largely driven by industry needs and it constitutes a significant effort to enhance the security of the software supply chain. These tools can be categorized into four main focus areas: component analysis, SBOMs (Software Bill of Materials), software integrity, and best practices. Component analysis tools focus on examining the individual parts of the software to identify vulnerabilities. SBOMs provide a detailed inventory of the components within a software product, facilitating transparency and traceability. Software integrity tools ensure that software has not been tampered with and maintain the trustworthiness of the software from development to deployment. Best practices cover guidelines and frameworks that help organizations implement effective security measures throughout the supply chain process.

In this Section we present our categorization of tools related to supply chain security. We grouped ten topics into these four main focus areas as shown on Figure 5.2. Each of these spans part of or all of the supply chain process described in 5.1

We describe the ten topics grouped in categories below:

5.3.1 Component Analysis

Component analysis tools are essential for ensuring the security of the software supply chain by addressing vulnerabilities within individual software components. The primary problem these tools aim to address is the identification and mitigation of security risks that may be introduced through third-party libraries and dependencies.

Databases

Several databases are used for vulnerability intelligence in supply chain security. They contain historically known vulnerabilities, and in component analysis, they are used as

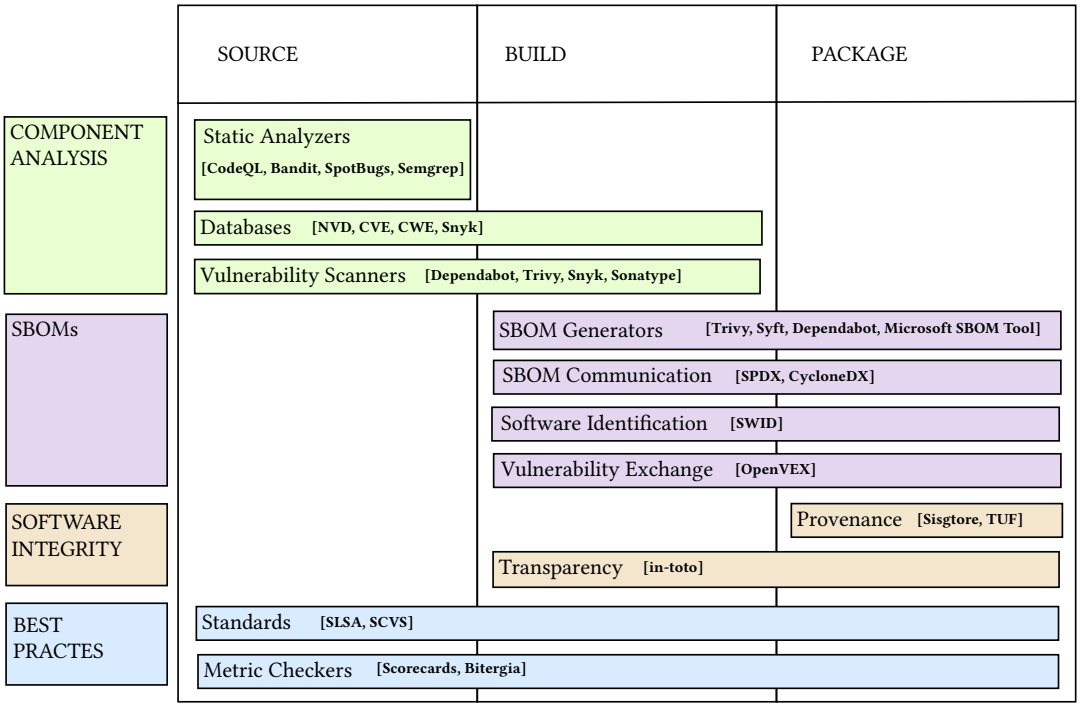

Figure 5.2: Tools taxonomy on Supply Chain Software Security

sources to identify risks in new software. The three commonly used databases are CVE, CWE, and NVD.

The CVE, **Common Vulnerabilities and Exposures** database contains known vulnerabilities in software and hardware identified by a unique number that helps track and share information about them[139]. This database is maintained by the MITRE Corporation, and vulnerabilities are assigned by **CVE Numbering authorities** [138].

The CWE, **Common Weakness Enumeration** [137], complements the CVE by identifying and categorizing common software weaknesses that can lead to vulnerabilities. It serves as a preventive measure during the development process.

The NVD, **National Vulnerability Database** [153] a government repository, contains

information from the CVE database along with impact metrics that help organizations understand the severity of a vulnerability as well as available patches and other external resources.

Another known database is the **Snyk** Database [191] developed by the company with the same name, which focuses its efforts on ensuring the accuracy of the CVE database while complementing it with more developer-friendly information such as overviews of vulnerable components, details about the vulnerability with examples and fixes if available. [199]

Security Analyzers

Security analyzers play a crucial role within the software supply chain since they automatically identify vulnerabilities present in the artifact's source code.

We classify security analyzers into two categories: Static Code Analysis and vulnerability scanners. In the first category, we find **CodeQL**, one of the most widely used tools integrated with GitHub that runs queries to find specific security issues in the codebase. [76] Other tools like this include **Bandit**, specific for Python which runs test plugins against the artifact files represented as AST nodes [56] and SpotBugs, specific for Java [80].

On the other hand, vulnerability scanning tools run the codebase against vulnerability databases like the CVE to identify risks in the code. The most used tool within this type is **Dependabot**, which identifies known vulnerabilities in a project's dependencies. [77] Its GitHub integration makes it advantageous since developers can quickly adopt it within their workflow. Other commercial tools that provide a broader scope of vulnerability detection are **Snyk**, which is more developer friendly [190]; **Trivy**, tailored for containers [131]; and **Sonatype**, which offers better integration. [104]

5.3.2 SBOMS

Much of the work done and the tools developed have been around identifying all the components of the software artifacts through SBOMs. The SBOM work started in 2018 and it's a community effort driven by the National Telecommunications and Information Administration (NTIA) [10]. We classify the tools and work around SBOMs into generation, communication and identification.

SBOM Generation

Several tools can be used to generate SBOMS automatically using container images and filesystems. **Syft** and **Trivy** offer an open source solution for SBOMS generation being the two of the most starred SBOM generators on GitHub. However, studies have shown that none of them are completely accurate.[202] The **Microsoft SBOM Tool** offers a solution that integrates within Microsoft's ecosystem of security and development tools. [4] Finally, a widely used SBOM generator, thanks to it's easy integration, is the **GitHub Dependabot**, which identifies dependencies in a GitHub repository making it very effective for updating dependencies but brings other problems like overwhelming number of notifications and lack of grouped update support. [95]

SBOM Communication

Multiple SBOM standards are used to share SBOM information with end users or customers in a specific way. The two widely used standards are **SPDX**, created by the Linux Foundation, and **CycloneDX**, created by OWASP. They both describe the components of a software artifact in a format that other tools can interpret but serve different needs. SPDX is well-suited for use cases requiring sharing detailed information about the components of a software system with humans along the supply chain since it provides detailed software component metadata, license compliance, copyright attribution. CycloneDX was developed with the purpose of generating SBOM documentation

which makes it a more efficient vulnerability management tool as it details all the standard components of a software product. [35] [91] Most SBOM generation tools support both standards since they are both widely used and unlikely to be unified.

Software Identification

In addition to SBOMs, there is an effort to identify each component of an artifact in a unique way since the use of different terminology and formats makes it hard to refer to these components outside of their ecosystem. SWID tags describe the information about a specific software product release, allowing organizations to track their software transparently. [194] [221]

Vulnerability Exchange

Often used to complement SBOMs, a Vulnerability Exploitability Exchange (VEX) is a vulnerability document that describes the applicability of one or more vulnerability findings in an artifact. These documents help users understand whether a vulnerability found is exploitable by reporting their status as "not affected," "affected," "fixed," or "under investigation." **OpenVEX** is an implementation for VEX developed in collaboration with CISA's VEX Working Group and is the first format that meets the VEX Minimum requirements. [9]

5.3.3 Software integrity

There are several interpretations of software integrity; in our case, we refer to software integrity as a measure of the source's code quality and correctness. In this area, we talk about works and tools that focus on aspects like signing, verification, and software provenance since these are critical for the integrity of software and, in the end, end-user trust.

Software Provenance

Tools like Sigstore [152] are used to identify software provenance. It provides a way for signing and verifying software with cryptographic signatures. The goal of this type of tool is to ensure that the software has not been tampered with before it is released to the final user. [97]

Software Transparency

Some other tools focus on making it transparent to the end user which steps are performed and by whom in the supply chain, such as **in-toto** or securing software updates by adding verifiable records of the files of a repository. [203]

5.3.4 Best Practices

The last area of focus that we have identified is related to best practices in open-source software development. This area spans the three steps we have identified as part of the software supply chain (source, build, package) since the tools and research here focus on a holistic analysis of the artifacts' development from the beginning of development to the release of the product to users. Some of the tools we identified are standards (or guidelines) that are meant for measuring and improving the assurance of the supply chain, and others are metric checkers that focus on project health.

Standards

Multiple standards help verify software components by measuring activities, controls, and practices in software artifacts development to reduce risks in the supply chain. The two primary standards used are SCVS, created by OWASP, and SLSA, implemented by the Linux Foundation. While SCVS establishes a framework for identifying activities, controls, and best practices for identifying and reducing risk in a software supply chain,

SLSA has a narrower scope as it focuses on integrity, ensuring that the consumed code has not been tampered. [118]

Metric checkers

Metric checkers assess software artifacts for security risks based on specific metrics related to the projects' health, security, and performance. They provide a score reflecting the project's risk level based on code vulnerabilities, maintenance, and continuous testing. Two known tools used for this purpose are **scorecards project** developed by the OpenSSF [167] and Bitergia Analytics, a paid projects health dashboard developed by Bitergia [30].

Objective \ Methods	Position Papers	User Research	Empirical Studies	Tool Development	Datasets	Survey / SoK
Risk Assessment			[231]	[120]		[118]
SCS Overall	[65]	[224]	[13]	[141]		
Vulnerability Management			[54, 103, 64]	[61]	[173]	
Dependency Updates		[95]	[95]			[158]
SBOMs	[202, 134] [16, 36]	[230, 193, 38]	[22, 29, 154]			[235, 15, 184]
Software Integrity			[152, 203, 142]			
Software Security Practices	[14]		[236, 234]	[237]		[160]
Human Factors	[135, 72, 227]					

Table 5.1: Taxonomy of Literature Review on Supply Chain Software Security

Chapter 6

Rethinking Policy: Trade Security Lessons for Open Source Supply Chains

6.1 Introduction

Over the past quarter century, two security challenges have arisen that, while distinct in nature, have comparative structures that could enable policymakers to leverage key lessons learned. The first, which was catalyzed by the attacks of September 11, 2001, saw a sudden and intense focus placed on the security of the global trade network and the concern that bad actors could exploit weak links and undermine the global flow of commerce. More recently, incidents like SolarWinds, Log4j, and XZ Utils have demonstrated the extent to which the global software supply chain can be exploited [69, 105, 176].

Software supply-chain security has moved from a specialist concern to a national-security policy problem. The SolarWinds compromise showed that a build pipeline can

become a distribution mechanism for malicious code. Log4Shell showed that a single vulnerable dependency can trigger a global remediation effort. The XZ Utils backdoor showed that social engineering against an overburdened maintainer can threaten widely deployed systems before an artifact reaches stable release channels [69, 105, 176, 73, 177]. These incidents differ technically, but they expose the same governance problem: downstream users depend on artifacts whose security properties are shaped by actors they rarely know, fund, or audit.

This chapter argues that OSS supply-chain security needs a more durable model of intervention. Government action is warranted because OSS failures can have national-security consequences. Post-9/11 trade security offers a useful precedent. Programs such as Customs Trade Partnership Against Terrorism (CTPAT), Authorized Economic Operator (AEO), Standards to Secure and Facilitate Global Trade (SAFE), and the Container Security Initiative (CSI) responded to national-security risk through trusted status, shared responsibility, upstream risk management, audit, incentives, and international recognition [208]. We examine how that design logic can inform OSS policy, with special attention to where the analogy fails.

Through a combination of literature review and interviews with trade security and OSS security practitioners, this research produces a framework for actionable recommendations focusing on collaborative efforts and positive incentives for developing and maintaining software security, recognizing that policies which center on investment in critical cyber-infrastructure, the development of risk assessment tools, and the establishment of international standards are likely to be more effective than those that aim to control, dictate to, and punish open source projects and their maintainers. To guide this analysis, we address the following research questions:

RQ1: Which design features helped post-9/11 trade-security programs gain adoption and persist?

RQ2: Which of those features translate to OSS supply-chain security, and where does

the analogy fail?

RQ3: What actionable framework for OSS supply-chain security follows from this comparison, and how should its incentives and obligations be distributed across commercial vendors, foundations, and volunteer maintainers?

Our contributions include:

- A cross-domain comparative analysis of trade security and OSS supply-chain security grounded in interviews with practitioners from both communities—to our knowledge, the first empirical test of this policy analogy with the people who built the trade programs.
- An account of which trade security success factors transfer to open source and which structural differences limit the analogy, including traceability as the unsolved technical problem on which certification frameworks depend.
- A framework for actionable recommendations tailored to the structure of open source communities focusing on collaborative efforts and positive incentives.

6.2 Related Work

Prior work has explored several component areas this chapter brings together. Much of the existing research focuses on technical contributions: taxonomies of attack vectors [119], analyses of security properties across the supply-chain lifecycle [160], and community-driven frameworks such as Supply-chain Levels for Software Artifacts (SLSA) [188], the OpenSSF Scorecard [168], and NIST Secure Software Development Framework (SSDF) [148] that define standards for artifact integrity and security hygiene.

Far less attention has been paid to whether these tools and standards are actually adopted in practice, and what organizational, cultural or economic factors affect adoption. Some

works that have begun to address this gap include Chakraborti et al. who find significant gaps between stated and enacted governance structures in OSS communities [39], and Wermke et al. who document how incentive mismatches undermine secure dependency management among practitioners [225].

Most directly related to our work, Neaher draws on 14 semi-structured expert interviews to evaluate voluntary standards for OSS security, finding that uptake is the central barrier and identifying financial incentives and community culture as the key determinants [150]. However, her work leaves room for further development at the institutional level: the question of how to structure financial incentives so that adoption is graduated and self-sustaining, and what a concrete program architecture for "government as convener" would look like in practice, remain open; and are precisely the questions this chapter takes up.

We argue that the missing element is a proven institutional model. Specifically, we draw on CTPAT, the post-9/11 voluntary tiered certification program that produces graduated operational benefits and achieves community buy-in through program architecture rather than outreach alone. Building on these principles, we propose a "Trusted Source" framework that derives its design principles from CTPAT and stress-tests them against practitioner knowledge from both domains.

6.3 Open Source Supply-Chain Security

It is commonly acknowledged that supply chains are only as strong as their weakest link; understanding how this axiom plays out across OSS ecosystems helps clarify why policy interventions modeled on trade security may prove effective. The software supply chain encompasses all the materials, components, processes, people and channels that it takes to develop a software product [123], including third-party libraries comprising a vast amount of OSS. Unlike compromises in the physical chains, where malicious intent can be more easily contained, software supply-chain attacks inject ma-

licious code into applications, allowing attackers to rapidly infect all downstream users [49].

Three recent security incidents help illustrate the diversity of supply-chain attack vectors. The **SolarWinds** attack attributed to a Russia-backed group, compromised the Orion build pipeline by inserting malicious code directly during the software compilation. Attackers distributed trojanized updates to roughly 18,000 customers, including government agencies and critical infrastructure providers, allowing them to spy on SolarWinds Orion customers and gain access to many critical IT systems in companies and government agencies using this platform [240]. The **Log4j** zero-day (Log4Shell) was a critical remote code execution flaw in one of the most widely used logging libraries, built into consumer endpoints, web applications, and cloud services, which made the impact of this attack massive [147]. The disclosure led to a worldwide effort to locate and patch vulnerable systems, a task the US Department of Homeland Security estimated could take a decade to complete because of how embedded this vulnerability is in the software supply chain [101]. The **XZ Utils** attack was a deliberate social engineering campaign in which an attacker gained the trust, obtained administrative access and introduced a backdoor into a compression library present in nearly all Linux systems, caught only days before integration into major distributions [73, 177].

6.3.1 Response to Incidents and Policy Changes

In response, the open source and security communities have made significant strides toward improving software supply-chain integrity and transparency. Community-led efforts have produced standards and tooling for artifact provenance (Sigstore [151], in-toto [201], TUF [18]), software composition transparency (SBOMs) [149], and tiered security attestation (SLSA, OpenSSF Scorecard). Organizations such as the OpenSSF have worked to harmonize these initiatives into coherent frameworks developers can adopt incrementally. However, adoption remains limited: many organizations—and especially volunteer-maintained OSS projects—lack the resources or incentives to in-

tegrate these practices at scale. On the policy side, an early version of the European Union's Cyber Resilience Act (CRA), announced in 2023, alarmed open source communities by appearing to impose lifecycle security obligations on unpaid maintainers [174]. These outcomes illustrate the risk of developing policy without early and sustained stakeholder involvement and provide independent validation of actor-type differentiation as a policy design principle, motivating the incentive-based framework this chapter develops.

6.4 Tangible Supply-Chain Security

The attacks of September 11, 2001, forced a rapid transformation in how governments and industry approached the security of global trade. The programs that emerged from that moment demonstrate that public-sector security goals and private-sector efficiency priorities can be aligned without resorting to punitive mandates. Governments feared the impact of terrorist organizations exploiting the weakest link in cargo-centered supply chains [215]. However, initial U.S. efforts focused on relatively impractical mandates, including 100% screening of all inbound cargo [155]. With increased input from industry, more pragmatic countermeasures quickly emerged that reflected the need to balance securing trade without adversely impacting global commerce, including incentive-based public-private partnerships such as **Customs Trade Partnership Against Terrorism (CTPAT)** and **Authorized Economic Operator (AEO)** programs, which were reinforced internationally through **Mutual Recognition Agreements (MRAs)** [214]. Other government-to-government efforts also saw an increase in information-sharing and pushed screening of cargo further up the supply chain and closer to the point-of-origin [217].

Launched in November 2001, the US-based CTPAT initiative is a government-industry collaborative venture [216, 117]. As a voluntary program, CTPAT is based on a principle of shared responsibility and incentive-based approach [218]. A company voluntarily

applies to join CTPAT and agrees to undertake a stringent assessment of the security of its own supply chain, identifying areas of risk or gaps and then formulating action plans to mitigate these risks [159]. In exchange for their commitment to enhancing global supply-chain security, companies were certified as CTPAT partners which streamlined importation of goods and shorter wait times at the border [216]. CTPAT's tiered structure offers progressively greater benefits to partners who demonstrate a greater level of commitment to trade security, incentivizing continuous improvement and the treatment of security as an opportunity to attain a competitive business advantage [218]. CTPAT is now largely institutionalized within key import-focused stakeholders and viewed as adding value to the services offered for import-dependent operations [212].

The Authorized Economic Operator (AEO) program is a parallel international effort launched by the World Customs Organization in 2005 [213]. AEO programs are operational in 90 countries, covering all of the EU, 15 countries in the Americas, 12 in Asia-Pacific, 9 in East and South Asia, 8 in the Middle East and North Africa, and 1 in West and Central Africa, while 7 additional programs are under development [228]. The AEO programs are further supported through Mutual Recognition Agreements (MRAs), under which an AEO holder can request that the foreign customs authorities grant them comparable benefits in the relevant jurisdiction [66].

6.4.1 Success Factors

The adoption and long-term institutionalization of these trade security programs—spanning multilateral, national, public, and private sector actors—was driven by several key factors. These factors include:

1. **Sense of shared urgency**: The impact of the 9/11 attacks was felt worldwide, particularly in the transportation sector with the grounding of flights, including air cargo [50]. Fear of further attacks drove action, and governments and the private sector alike expected that supply-chain security would demand sustained

attention [68].

2. **Incentives for key stakeholders**: Private sector actors are oriented toward profitability, and security compliance can look like a sunk cost. CTPAT and AEO answered this by letting participants streamline their importing processes at a moment when supply chains were globalizing rapidly and just-in-time logistics put a premium on speed of delivery [34]. The benefits spoke directly to importers' bottom line. Incentives also drove the international spread of CSI: partner governments gained access to shared intelligence, visible participation in anti-terrorism and nonproliferation goals, and streamlined access to a key export market [140].

3. **Clear guidance aligned with existing operations**: The programs added security-focused requirements without re-engineering the importation process. New data elements were added to import declarations to improve advance screening, but the declaration process itself was untouched; freight forwarders were asked to improve physical security measures such as fencing and lighting, not to build new warehouses.

4. **Auditing and monitoring**: CTPAT and AEO require ongoing auditing of security-specific records and operations [211]. Auditing was already familiar to the importing sector, since Customs periodically reviews the Harmonized System tariff codes importers self-declare to verify duty rates[1]. Periodic review kept CTPAT security practices from becoming one-off actions; they became part of continuous operations, analogous to quality control in product development.

These factors explain both the programs' early uptake and their durability: supply-chain security is now part of the shared vocabulary of the global trade community and its government counterparts. The sections that follow test whether the same principles can support a framework for OSS.

[1] Goods declared for importation are classified using the internationally accepted Harmonized System ("HS Codes"), and national customs authorities publish duty rates indexed by these codes.

6.4.2 Limitations of the Analogy

The frameworks developed since 2001 to manage risk in the movement of tangible goods provide useful models for thinking about the development and flow of software. However, the practical limits of the analogy between software and tangible goods deserve attention before any parallel framework is proposed.

An example of limits includes the use of the term SBOMs. A shipping Bill of Materials (BOM) provides a single layer inventory of shipment contents. However an SBOM must capture deeply nested dependencies that can run dozens of layers deep [70]. As one analyst notes, a software package "may incorporate [a third-party component] it in its entirety, or select individual files, or functions, or even lines of code.... There is no atomic unit of software" [74]. Unlike tangible goods which are static, software is dynamic, replicable and distributed globally nearly instantaneously with origins spread across geographically dispersed communities of volunteers. The demands placed on a software SBOM are therefore fundamentally different from its shipping counterpart.

The Trusted Trader program, by contrast, translates more directly. This voluntary program certifies that a company's internal security processes meet a defined standard in exchange for operational and business benefits. A system rewarding software projects that follow secure development best practices grounded in established standards such as the NIST SSDF, the SLSA framework, or the OpenSSF Scorecard would follow the same self-assessment, external validation logic [117].

Beyond the technical SBOM analogy, the political and commercial assumptions underlying CTPAT also require adaptation when applied to decentralized, volunteer-driven OSS ecosystems. The terrorist attacks of 2001 created rare alignment across government, industry, and the public that made rapid coalition-building possible, a condition that does not currently exist in OSS security. Additionally, the success of programs like CTPAT relies on a supplier having commercial interests and business demands, a framing much of the volunteer-driven OSS community actively rejects. Unlike the enti-

ties that are certified through CTPAT, many open source developers have no contractual relationship with downstream users, nor are they obligated or swayed by commercial interests. While these limits do not invalidate the analogies, they do require empirical grounding. The expert interviews from this study examine whether and how each of these limitations can be addressed in practice.

6.5 Research Methodology

6.5.1 Research Design

This study uses a qualitative research design based on semi-structured expert interviews and document analysis. We examine whether lessons from post-9/11 trade-security programs can inform OSS supply-chain security. Because the comparison spans two practitioner communities that rarely overlap, neither a literature review alone nor expert consultation within a single domain would be sufficient. We therefore combined document analysis with semi-structured expert interviews across both domains.

The research proceeded in three phases. First, we conducted parallel literature reviews of post-9/11 trade-security programs and the current OSS supply-chain security landscape, the findings of which were synthesized into a preliminary presented in November 2025 [206, 127]. This review informed a preliminary comparison between trade-security mechanisms, such as CTPAT, AEO, and CSI, and emerging software supply-chain security mechanisms, such as SBOMs, provenance frameworks, and secure-development standards. Second, we conducted expert interviews to test and refine that preliminary comparison. Third, we synthesized the interview evidence with the literature review to identify which lessons from trade security appear transferable to OSS security, which require adaptation, and where the analogy breaks down.

6.5.2 Participant Selection and Recruitment

We used purposive expert sampling to recruit participants with direct, practice-based knowledge of either software supply-chain security or post-9/11 trade-security programs. This sampling strategy was appropriate because the goal of the study was to understand mechanisms, assumptions, and implementation challenges across two specialized domains.

Participants were recruited through the research team's professional networks and conference-based recruitment. Candidate-selection criteria were documented in the interview protocols, which are hosted on an external platform and linked in Appendix C.2. For software supply-chain security participants, selection criteria emphasized hands-on experience in software supply-chain security or OSS security, including experience in corporate security, Open Source Program Offices, open source foundations, security tooling, standards, or academic research. For trade-security participants, selection criteria emphasized professional expertise in supply-chain security or customs security programs, ideally including direct experience with CTPAT, AEO, CSI, or related customs risk-management initiatives.

We conducted nine formal expert interviews: six with software supply-chain security or OSS security practitioners and three with trade-security practitioners (see Appendix C.1 for full expert profiles). Throughout the chapter, interviewees are referenced anonymously as **Experts [1–6]** for software supply-chain security and OSS security practitioners and **Experts [7–9]** for trade-security practitioners. The software-security interviews allowed us to assess how practitioners understood current OSS supply-chain security challenges and whether trade-security concepts appeared meaningful in the software context. The trade-security interviews allowed us to identify which features of CTPAT, AEO, CSI, and related programs practitioners viewed as central to their adoption and long-term institutionalization.

6.5.3 Interview Procedure

We conducted semi-structured expert interviews using two interview protocols, one for software supply-chain and OSS security practitioners and one for trade-security practitioners. Both protocols centered on the chapter's overarching question: what can open source and cybersecurity communities learn from post-9/11 trade-security efforts? The protocols differed in emphasis to reflect participants' domain expertise.

The software-security protocol asked participants about current OSS supply-chain security challenges, existing tools and standards, the role of government and regulation, and their reactions to the trade-security analogy. The trade-security protocol asked participants about the history, adoption, incentive structures, auditing mechanisms, and long-term institutionalization of programs such as CTPAT, AEO, and CSI. Trade-security participants were not expected to have software-security expertise. Software-security participants were given a brief explanation of the relevant trade-security programs before being asked to evaluate the analogy.

Interviews lasted approximately 30 to 60 minutes and were conducted through a mix of remote and in-person formats. With participants' consent, all interviews were audio-recorded and transcribed for analysis.

6.5.4 Analysis

We analyzed the interview materials using exploratory thematic analysis, a qualitative approach for identifying recurring patterns across interview data [33]. The goal was to identify recurring claims, tensions, examples, and points of disagreement relevant to the chapter's comparative argument.

The analysis proceeded in three steps. First, two researchers reviewed six interview transcripts to identify preliminary themes. The lead author reviewed the remaining three interview records. Second, the two researchers met to compare the themes they

had identified, discuss differences in emphasis, and refine the analytic categories. Third, the lead author finalized the themes that structure the chapter's findings and recommendations.

The analysis combined deductive and inductive reasoning. Deductively, the analysis was guided by the chapter's central comparison between trade-security programs and OSS supply-chain security, including concepts such as trusted trader programs, tiered certification, mutual recognition, incentives, and continuous validation. Inductively, the analysis remained open to themes that emerged from the interviews themselves, including trust dynamics in open source communities, resistance to labeling volunteer maintainers as suppliers, the need to distinguish commercial actors from community actors, and traceability as a technical frontier.

The framework in the next section shows adaptations based on three sources of evidence: the historical and policy literature on trade-security programs, interviews with trade-security practitioners, and interviews with software supply-chain security practitioners. This triangulation helps distinguish between mechanisms that appear meaningfully transferable, mechanisms that require modification, and limits of the analogy that should constrain policy design.

6.6 Framework for Action: Recommendations Grounded in Expert Evidence

The following presents actionable lessons and final recommendations informed by our expert interviews and grounded in the historical analysis of CTPAT and related programs presented earlier in this report. They are organized around four sequential stages that reflect the order in which a durable voluntary framework must be built (Figure 1).

- **Preconditions** (§6.6.1) establish the trust, risk awareness, and government-community literacy to facilitate collaboration.

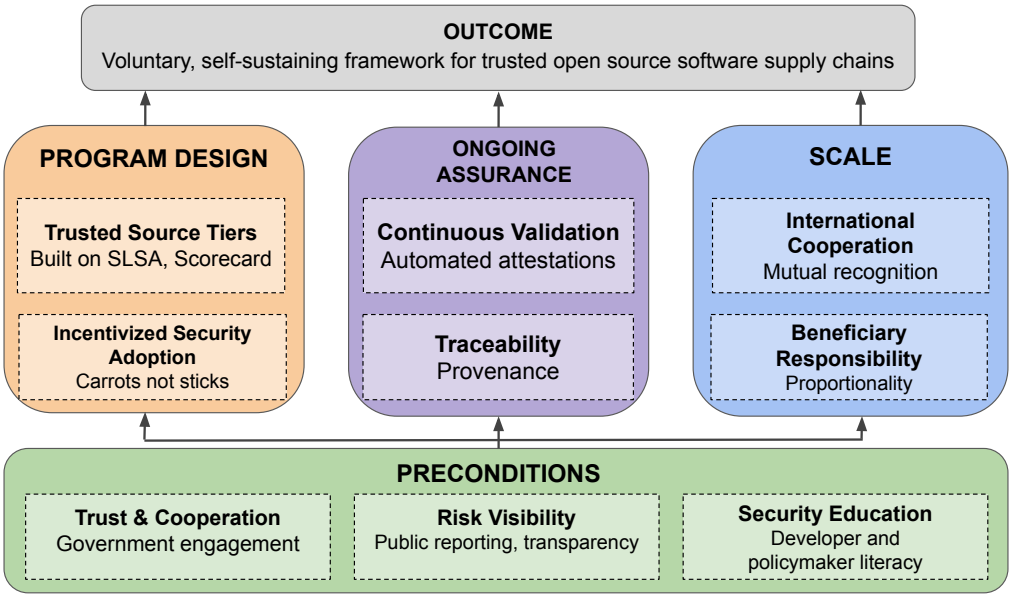

Figure 6.1: A layered framework for adapting trade-security logic to OSS supply-chain security.

- **Program Design** (§§6.6.2–6.6.3) defines the structure of the Trusted Source framework itself: its tiered voluntary certification architecture and the incentive logic that drives participation.
- **Ongoing Assurance** (§§6.6.4–6.6.5) addresses what the program must do to remain effective over time: continuous validation rather than one-time compliance snapshots, and the provenance infrastructure that is the technical foundation of the entire framework.
- **Scale** (§§6.6.6–6.6.7) examines the conditions for international expansion through mutual recognition and the burden-sharing mechanisms needed to put security investment where the value extraction is.

Together, these stages are designed to produce a single outcome: a voluntary, self-sustaining framework for trusted OSS supply chains, that does not depend on perpetual

government intervention, scales across jurisdictions, and gives the market a legible signal for OSS trustworthiness.

6.6.1 Preconditions for Success

Trust and cooperation before compliance

A key lesson from the trade security case study is that successful approaches did not emerge from a blank institutional slate. Before 9/11, the Customs Modernization Act of 1993 had shifted the Customs-industry from adversarial to cooperative. Expert 7 captures the program's spirit: "we can't search every container, so we give you [industry] more responsibility." Expert 8 reinforces this: the shift from a "gotcha" enforcement model to a client-based approach—with dedicated client managers assigned to major importers—meant that industry had already experienced government as a partner before it was asked to act like one. The same logic applies to OSS policy: regulatory discussions like the CRA proceeded with limited open source community consultation and were experienced as punitive impositions rather than collaborative investments. Expert 3 identifies the deeper process failure: the role of government should be to "become a meaningful stakeholder in open source software communities rather than try to transliterate policy frameworks that exist in closed-source environments." Expert 1 agrees: "the most important thing any government can do before writing laws and regulations is making sure they understand the circumstance, and they're talking to the folks who are directly involved." Expert 9, who was directly involved in the original design of CTPAT, illustrates why partnership had to precede requirements: in the earliest design discussions, a government agent demanded that one major retail company guarantee 100% security across 15,000 vendors in 84 countries. The lesson Expert 9 drew was clear: "if we leave them to themselves and we don't have input into the process, they're gonna make a program, but it's gonna be onerous and costly."

A Trusted Source program, or any certification-based framework, is unlikely to achieve

CTPAT-scale adoption unless government has first built genuine relationships with the open source ecosystem. As Expert 2 notes, engagement must begin as contribution and support, not compliance burden; the communities that need to be secured are already operating at the edge of their capacity. As Expert 3 observes, "you need a lot more meaningful understanding of the reputational and trust dynamics that lend credibility to work in the open source space"—a condition that cannot be satisfied by regulatory mandate alone.

This precondition operates at two levels in decentralized ecosystems. Government-to-community trust is necessary but not sufficient: in a federated ecosystem of largely independent projects, no government agency or certification body has a central point of contact that can commit the community to any standard. Adoption must instead be achieved through social consensus, propagated outward from a set of influential coordinating projects—analogous to how the Scientific Python project's SPEC process propagates technical recommendations across the ecosystem by building consensus among a set of influential core projects first, rather than mandating compliance from above. Ecosystem coordination bodies—cross-project governance structures, umbrella foundations, and standards intermediaries—are therefore not optional partners in a Trusted Source program; they are the mechanism through which government-to-community relationships become actionable at scale.

Risk Visibility

One precondition from trade security is harder to manufacture: the political urgency that made CTPAT and other programs possible. Expert 3 observed: "post 9/11, you have like a galvanizing moment. Everybody rallied around the whole sort of like 'let's fight these international terrorism criminals'... I don't think this is happening right now... not in a way that would allow... this very closely knit collaboration between the industry and governments." This view is consistent with a widespread organizational tendency to underestimate exposure. Expert 4 describes it: "there's too many

organizations that are just saying 'well, we're not the slowest gazelle in the pack. The cheetah is going to catch and eat someone else more likely than not. And anything we spend to be any faster than other people, it's wasted." Policymakers cannot manufacture a crisis, but they can make structural risk visible—through public reporting on supply-chain incident costs, coordinated risk assessments, and transparency about commercial dependence on volunteer-maintained infrastructure.

Security Education

Education and awareness emerged as one of the most consistently cited priorities across the interviews. Expert 1 identified it as the single most important shift the field needs: "start with the basics; in all the computer science curricula right now, there is a big lack of security awareness." Expert 2 echoed this, arguing for "minimum basic security awareness, uniformly taught to all computer science engineers." Multiple experts framed this as a "socio-technical problem": the tools to secure the software supply chain largely exist, but awareness of why and how to use them does not. Expert 8, from the trade security side, pointed to Congressional hearings on TikTok [98] as evidence of what happens when decision-makers lack basic literacy about the industries they regulate. Expert 5 reinforces the point from the software side: "there are still people every week that say that they've never heard about supply-chain security before"—a reminder that awareness-building is not a one-time effort but an ongoing condition for any framework's effectiveness. Expert 7's account of the Customs Modernization Act tells the same story from the other direction; Customs had to learn how industry operated before it could design requirements that industry would accept.

6.6.2 Trusted Source Tiers

The cornerstone of the framework is a voluntary "Trusted Source" program, the same logic that underlies CTPAT's Trusted Trader designation applied to software: voluntary adherence to defined security standards earns tangible operational benefits and a

reputational market signal.

Expert interviews converge on four design principles:

- **Tiered Structure**: Like CTPAT, the program should offer graduated certification levels corresponding to increasing security maturity, with progressively greater benefits at each tier.

- **Differentiation by actor type**: The program should explicitly distinguish the primary categories of participant: commercial software vendors and large enterprises; open source foundations and ecosystems managing widely-used projects; and individual maintainers. Each category requires different entry points, baseline requirements, and incentive structures. Compliance burdens must fall predominantly on commercial actors, not on unpaid volunteer maintainers. As Expert 1 notes, many OSS developers "really strongly object to the word 'supplier'... 'I gave this to you freely. Why do you think that you can demand things from me? For a gift I gave you with no contract?' " The program's language, governance, and entry points must reflect the voluntary, community-driven nature of OSS contribution rather than importing supply-chain assumptions.

- **Build on existing signals**: The program should be based on existing frameworks like SLSA levels, the OpenSSF Scorecard, and the NIST SSDF, which provide ready starting points for defining these tiers. The DoD's Cybersecurity Maturity Model Certification (CMMC) and NIST-800 standards, already operational in defense procurement, offer institutional models that do not need to be invented from scratch. Expert 5 observed that vendors "would really like to just be able to have this stamp and say, look, you can trust me because I'm SLSA level three." A formal framework should build on these organically emerging signals rather than replace them.

- **Open source community co-authorship**: The program should be administered by designated agencies (CISA in the US, ENISA in the EU) but overseen by

an international collaboration including government agencies, open source foundations, industry groups, and academic institutions. Open source foundations must be structural participants in program governance—not merely consultees—and standards must be developed through a process that gives the open source community genuine co-authorship, a lesson directly drawn from the CRA's compressed drafting process.

6.6.3 Incentivized Security Adoption

The Trusted Source program's reach depends on genuine participation—which, as the trade security case demonstrates, is best achieved through carrots, not sticks: positive incentives rather than punitive mandates. CTPAT reframed security investment as a source of competitive advantage—faster border crossings, fewer inspections, lower operational costs [197]. This carrot was reinforced by a soft stick: non-participants faced higher inspection rates and unpredictable delays, so the program created strong pull without a mandate. Expert 9 noted that one large retail company found enough internal efficiencies during CTPAT compliance that "the program was almost self-funded."

Expert 4 makes the parallel explicit for software: "SBOMs are being adopted not because the government says 'you must have an SBOM' but because 'you have to have an SBOM to sell to the government'." Access-based incentives create value rather than mere compliance obligations. A successful software supply-chain security program should therefore be built around a value proposition centered on positive incentives:

- **Liability safe harbors** for projects and adopters adhering to best practices, which would fundamentally alter the risk calculus for software producers [75].
- **Preferential treatment in public and private sector procurement**: governments and large enterprises should explicitly favor certified projects in their procurement decisions, creating a direct market incentive that extends beyond the public sector—already working in practice as companies adopt SBOMs to qual-

ify for US government contracts.

- **Reduced cyber insurance premiums** for certified projects and their adopters: as Expert 4 noted, "insurance companies are actually much better at dealing with risk and understanding risk" than government bodies, and cyber insurance rewards genuine security improvements rather than checkbox compliance.

For academic and research-community OSS, market-facing incentives are only part of the picture. Foundational scientific computing libraries and domain-specific research tools are often maintained by volunteers whose primary incentive systems run through federal grants (NSF, DOE, NIH) and academic reputation rather than commercial procurement or insurance premiums. For this population, grant-eligibility requirements and incentives—analogous to export-compliance and data-protection clauses already embedded in research funding—may be the most actionable pathway for Trusted Source adoption.

A clear market signal linking secure development practices to greater adoption can elevate the security posture of the entire ecosystem [75], though its impact will be uneven across projects where market forces play a limited role.

The incentive calculus is not uniform across actor types. Expert 8 found that the key question companies asked was: "can you tell us with confidence that, if we spend $1M on a CTPAT compliance program, our competitors won't have an advantage by not participating?" For large-volume importers the answer was yes; for mid-size companies ROI calculations were less certain. The same differentiation applies to software: large technology companies and major open source foundations are well-positioned to engage with a certification program; individual volunteer maintainers and smaller projects are not. Expert 3 recommends targeting at the project level, "where an open source product can be tagged in such a way that it receives favorable treatment," rather than applying uniform expectations across individuals and organizations.

6.6.4 Continuous Validation

Any certification framework for OSS security must be a continuous process, not a one-time snapshot. In CTPAT, regular risk assessments and re-evaluations were required to maintain certified status, creating a virtuous cycle of improvement and preventing security postures from becoming stale. The January 2026 GAO report on CTPAT found that 4% of participants were involved in security incidents over a five-year period, and that CBP had not consistently investigated those incidents or taken enforcement action—illustrating that participation alone does not guarantee compliance, and that ongoing auditing with clear enforcement criteria is essential even in a mature program [219]. Automated tools are necessary but not sufficient: spot checks and human review remain essential while verification infrastructure matures.

Priorities for automated continuous validation include:

- **Pipeline attestations** that provide, in Expert 4's words, "an accurate representation of what actually happened," generating verifiable records at every build stage as a continuous process output rather than a retrospective audit; Expert 6 describes these as records that "link all of the actions and attribute them to who did it and how."

- **Automated dependency management**, such as Dependabot and the built-in update mechanisms of Go and Rust, so that projects are alerted when a dependency becomes outdated or compromised rather than discovering it after the fact (Expert 5).

- **Human review of contributor activity** for high-tier certified projects: Expert 4 emphasizes that basic practices like code review before merge and branch protection rules remain far too rarely applied, and Expert 6's account of XZ Utils shows that automated tools alone would not have caught an attacker who built trust gradually over months of legitimate-looking commits.

Compliance burdens must not be prohibitive for volunteer-driven projects. Automated tooling that minimizes manual overhead is vital for equitable participation across project types. Continuous validation, however, is only as reliable as the underlying infrastructure that makes actions attributable; which brings us to the unsolved technical problem at the foundation of the entire framework.

6.6.5 Traceability as the Technical Frontier

Traceability emerged across both expert groups as an unsolved technical problem. Expert 8 observed that "the tech really hasn't evolved enough to get to true traceability" even in physical cargo; in software, where contributors are geographically dispersed and dependency chains run dozens of layers deep, the challenge is greater still. Yet traceability is the condition that would allow all other elements of this framework—Trusted Source certifications and continuous validation—to function. The most promising current approach is provenance-based: frameworks like SLSA and in-toto record not just what is in a software artifact but who performed each step in its creation and how, linking all build-pipeline actions to identifiable, verifiable actors. This approach captures process integrity, not just component lists. Expert 6 observed that the major supply-chain compromises of recent years—SolarWinds, Log4j, XZ Utils—all involved failures of process integrity that an SBOM alone would not have caught: "most of the major supply-chain compromises are about lack of transparency."

Policymakers, funders, and industry practitioners should treat investment in provenance tooling and traceability infrastructure as a priority, including:

- Funding development of automated provenance generation and verification tools that integrate into common build systems with minimal friction.
- Establishing provenance documentation as a requirement within the Trusted Source framework, with higher tiers corresponding to more complete traceability coverage.

- Supporting cross-ecosystem standards for provenance data formats, allowing provenance chains to be followed across language ecosystems, package managers, and organizational boundaries.

A forward-looking note on emerging threats is warranted. AI-assisted supply-chain attacks (autonomous agents conducting influence operations against maintainers, automated vulnerability chaining, and cache poisoning via compromised CI/CD workflows) are already documented as of 2026 [186, 89, 175]. These attacks are particularly dangerous in volunteer-maintained ecosystems where maintainer capacity is thin and security governance is informal. The provenance and traceability infrastructure described above is not merely useful for the attack classes of 2020; it is the baseline defense against AI-accelerated threats already emerging, and the technical foundation on which the long-term effectiveness of this entire framework depends.

6.6.6 International Cooperation

With the program's internal integrity established through continuous validation and traceability infrastructure, the final stage addresses how that framework can operate at global scale. The global nature of trade necessitated global security standards over the last two decades. CTPAT's success in the international arena is heavily reliant on its Mutual Recognition Arrangements (MRAs) with the Authorized Economic Operator (AEO) programs of other nations [210]. These agreements mean that those certified under one country's trusted trader program are recognized by another, allowing benefits to extend across borders and creating a harmonized international security framework. This lesson has far-reaching implications for OSS supply chains, which are arguably more globalized and interconnected than physical trade. Projects—particularly larger, widely adopted ones—are developed by global communities of contributors and are used worldwide. Security standards from a single jurisdiction are insufficient and reliance on standards not accepted more widely would be ineffective and potentially counterproductive, creating barriers to trade and innovation. Expert 3 identifies frag-

mentation and balkanization of norms across jurisdictions as one of the two primary challenges in OSS security today: diverging regulatory environments reduce the community engagement and resource pooling that are the backbone of open source's security model. Expert 8's practical experience guiding companies through both CTPAT and AEO compliance programs across the US, Europe, and Asia provides firsthand confirmation that cross-jurisdictional certification is achievable at scale—and that when it works, it creates significant efficiency gains for all parties. Conversely, the parallel and non-harmonized demands of multiple certification regimes create compliance costs that fall hardest on the smallest actors.

Policymakers must therefore pursue a strategy of international cooperation to align on foundational security principles. The trade security case provides a clear roadmap: countries, regional groupings, and international organizations should work toward harmonizing standards for secure development lifecycles, agreeing on common formats and data fields for SBOMs, and establishing shared protocols for vulnerability disclosure. The goal should be to create a global framework of mutual recognition for national Trusted Source programs, mirroring the success of the MRA framework that linked CTPAT and AEO programs internationally. Due to the community nature of open source, these efforts must include open source foundations and communities as active participants; their co-authorship of standards is as essential here as it is in the Trusted Source program design itself.

6.6.7 Beneficiary Responsibility

Open source security is structurally under-resourced because those who bear the burden — volunteer maintainers — are not those who capture most of the value. Expert 6, who observed the XZ Utils incident firsthand, describes the failure directly: when a major technology company's employee found the backdoor, the company published a blog post and walked away, leaving one volunteer maintainer alone to spend weeks inspecting every commit the attacker had made, with no sustained support from the

companies that depended on the library. As Expert 6 argues, those companies should either be required to help fix it or "give money to this poor guy who's maintaining like half of the compression libraries in the world so that he doesn't do this on the side and doesn't do it alone."

The CTPAT model offers a direct analog: the program asked the companies profiting from the supply chain to invest in securing it. Expert 8 observed that CTPAT-compliant companies gained not just regulatory benefits but supply-chain visibility—"spillover effects" that created private value, not just public goods. Adapted to software, commercial actors who depend on open source must bear a proportionate share of security investment. Governments occupy a symmetric position: as one of the world's largest OSS consumers, it is subject to the same free-rider critique as commercial actors. Expert 2 put it plainly: "I wouldn't make a difference between public and private sector... you're all consuming open source. You should give back to it... money, expertise, support." Table 6.1 summarizes these mechanisms, the actors they target, and their expected effects:

Proportionality is the governing principle: obligation should scale with the volume and criticality of OSS dependence, just as CTPAT compliance requirements scaled with import volume and supply-chain complexity. Companies whose core products depend heavily on specific OSS components bear the greatest responsibility for securing them.

6.7 Limitations

This study has several limitations. First, the interview sample is small and purposively selected. The interviews lasted between 30 and 60 minutes, and we wanted to capture in-depth discussions. The nine expert interviews were intended to capture practice-based knowledge across two specialized domains and were not necessarily aimed at producing statistically generalizable findings. The findings should therefore be read as expert-informed design guidance, and they do not represent claims about all open

Actor targeted	Mechanism	Example	Expected effect
Large commercial dependents	Enterprise Stewardship Obligations	Require companies above a dependency-use threshold to contribute resources to security of relied-upon OSS	Closes the gap between value extraction and security investment
Foundations and community projects	Funded Foundations or Consortia	Government and industry co-fund security infrastructure around critical OSS communities	Provides resources for security tooling and governance without displacing volunteer-led direction
Commercial actors earning revenue from OSS	Regulatory Leverage	Liability proportional to revenue derived from OSS [2]	Creates financial incentive for proactive commercial investment in OSS security
Entire ecosystem (commercial and public sector)	Reputational Accountability	Public scorecards rating companies on OSS security contributions relative to their consumption	Applies social and market pressure where direct regulation is infeasible

[2] Expert 6: "you're going to get sued for 1% of your last year's profit. Well, why don't you spend 0.5% of that money making sure that open source developers are actually complying?"

Table 6.1: Burden-sharing mechanisms: actor targeted, mechanism, example, and expected effect.

source maintainers, software security practitioners, trade-security professionals, or policymakers.

Second, recruitment through professional networks and conference-based outreach may have shaped the perspectives represented in the study. This approach was appropriate for reaching participants with direct experience in specialized programs and technical communities, but it may overrepresent experts connected to formal institutions, standards processes, or policy discussions. While some participants also served as unpaid volunteer maintainers, the sample was not designed to represent the broader popula-

tion of volunteer maintainers, whose experiences and constraints remain central to the policy problem.

6.8 Conclusion: Securing the Future of Code

The complex challenge of tracking software dependencies and vulnerabilities across global digital infrastructure is a source of profound risk to international security. The incidents examined in this chapter (SolarWinds, Log4j, and XZ Utils) are among the most visible examples of how structural weaknesses in the software supply chain can cascade into real-world consequences, illustrating the persistent "weakest link" problem. Strengthening security across the full development lifecycle is therefore a shared priority for governments, industry, and open source communities.

When the world faced a similar challenge in securing physical supply chains, stakeholders—led by the US and European governments—opted for public–private partnerships that emphasized proactive risk management and incentive-based security rather than purely punitive enforcement. The core principles that guided those trade-security efforts—trust through verification, transparency through documentation, and shared responsibility balanced against efficiency—also apply to complex, distributed software ecosystems. Expert interviews spanning both software-security practitioners and trade-security veterans confirm that these principles translate: incentives work better than punishment, cooperative relationships must precede compliance demands, and the burden of security cannot fall on the volunteers who build infrastructure that others profit from.

The largest gap between the trade and software contexts is institutional. Existing OSS security tools and practices are not yet connected to resources, incentives, and responsibility commensurate with the value extracted by large commercial beneficiaries. The framework developed in this chapter translates our findings into concrete recommendations:

- Building cooperative and educational foundations before introducing certification or regulatory machinery is the clearest precondition drawn from the trade-security case and the one most consistently identified by practitioners as missing from current OSS policy discussions.

- Trusted Source programs, tiered by actor type and grounded in existing community tools, offer an actionable path toward recognizing and rewarding good security practice without imposing impossible burdens on volunteer maintainers.

- Continuous validation, rather than one-time compliance, should be the foundation of any certification framework. At the same time, traceability remains the key unsolved technical challenge that limits how far such a framework can go, making investment in provenance infrastructure the highest-leverage technical priority.

- International harmonization and rebalanced commercial responsibility toward companies that benefit most from OSS are the remaining structural requirements for a framework that can operate at scale.

Securing the software supply chain requires a practical understanding of both the useful analogies and the critical differences between tangible and digital supply chains. It demands an approach that leverages automation, distinguishes between commercial vendors and OSS communities, and invests in the trust, governance, and security practices that underpin resilient digital infrastructure. This is not a task for government or industry alone, but a shared responsibility. By drawing on hard-won lessons from trade security while respecting the distinctive structure of decentralized, volunteer-driven ecosystems, we can build software supply chains that are more secure, trustworthy, and sustainable.

6.9 Ethical Considerations

This research involved semi-structured interviews with human participants and was reviewed and determined to be exempt (Category 2) by our Institutional Review Board prior to any data collection. Participants were recruited through the research team's professional networks spanning the trade-security and open-source-security communities. Before each interview, participants were informed of the study's purpose, the non-attribution policy for all published outputs, and were given the opportunity to ask questions before deciding to participate. Each participant provided verbal informed consent before any study procedures began. Interviews were audio recorded with participants' explicit consent, which was obtained as part of the verbal informed consent process before the interview began. Participation was voluntary, uncompensated, and there was no direct benefit to participants.

Chapter 7

Automating Defenses: Enhancing Vulnerability Detection with Large Language Models

7.1 Introduction

Java is one of the most widely used programming languages, powering applications across a broad spectrum, from enterprise software to mobile applications. According to Sonatype's State of the Software Supply Chain report [104], through the first 7 months of 2024, 828 billion Java components were requested from the Maven Central Repository. While Java provides memory safety, various other vulnerabilities may creep into production code and open source repositories, affecting millions of users worldwide. On average, 13 Critical or High-Severity Security vulnerabilities are discovered each year, per application [104]. These numbers underscore the urgent need for better tools to improve security and enhance vulnerability detection.

Static analysis tools such as CodeQL [42] and Semgrep [185] are capable of detect-

ing common security weaknesses in Java code; however, they often produce a large number of false positives and have limited triaging support to help developers address vulnerabilities more effectively. Furthermore, the output formats of these tools are not always easy to interpret, requiring developers to analyze large amounts of information to distinguish true vulnerabilities from noise and extract meaningful security insights.

This chapter studies how Large Language Models (LLMs) can enhance vulnerability triaging by using information from static analysis tools. Our approach combines the strengths of LLMs and static analysis, using LLMs to better categorize vulnerabilities identified by static analyzers. Our experiments show that by using LLMs to process findings from static analysis, we improve triaging accuracy for vulnerabilities such as injection attacks, weak cryptographic algorithms, and path traversal. However, this performance increase is not uniform for all types of vulnerabilities, and for some types, the performance decreases.

With this study, we answer the following research questions:

- **RQ1:** How does integrating LLMs with static analysis tools like Semgrep improve the accuracy, efficiency, and triaging of code vulnerabilities compared to using static analysis tools alone?
- **RQ2:** How effective are different LLMs in enhancing the triaging of code vulnerabilities within a Java codebase, particularly in terms of accuracy, efficiency, and reduction of false positives?
- **RQ3:** How do variations in prompt engineering influence the performance of LLMs in triaging code vulnerabilities, specifically in terms of detection accuracy and false-positive reduction?

Our contributions are as follows.

1. We conduct an empirical study of how different LLMs and prompting techniques perform when detecting a diverse set of Java vulnerabilities by triaging results

from a static analysis tool.

2. We test the performance of the latest (at the time of submission) OpenAI o1 "reasoning" models to detect software vulnerabilities.

3. We discuss and provide insights on why some configurations perform better than others for different software vulnerabilities or prompts.

7.2 Background

7.2.1 Vulnerability Detection

Software vulnerability detection refers to the process of identifying weaknesses that could be exploited by attackers. Some key resources for understanding and classifying vulnerabilities include databases like the Common Vulnerabilities and Exposures (CVE) system, which provides a publicly available list of specific known vulnerabilities in software (e.g., a vulnerability in a particular software package), and the Common Weakness Enumeration (CWE) database, which provides general definitions of software weaknesses (e.g., SQL injection).

Static Analysis

Static Code Analysis refers to code review to detect potential vulnerabilities in non-executing source code, often through automated tools. Although these tools aim to find security flaws automatically, they primarily help developers by narrowing down security sensitive areas of the code for a more detailed manual review [71].

Static code analysis tools have limitations, particularly with false positives and false negatives. False positives occur when the tool identifies non-existent vulnerabilities due to limited visibility into data flow integrity, particularly when analyzing applications that interact with closed-source components or external systems. False negatives occur

when real vulnerabilities go undetected, often because the tool lacks knowledge of new vulnerabilities in external components or details about the security configuration of the running environment [71].

Semgrep is one of the most popular open source static analysis tools designed to identify bugs. Unlike traditional string matching tools, Semgrep enables semantic code searches, allowing pattern matching within code syntax, though it is limited to the analysis of single functions or files [185]. We use Semgrep in our study as a baseline for evaluating the performance of vulnerability detection tools.

OWASP Benchmark

The Open Web Application Security Project (OWASP) is a non-profit organization providing resources for securing web applications. One of their projects is the OWASP benchmark, a free Java test suite that evaluates the accuracy, speed, and coverage of automated software vulnerability detection tools.

The benchmark provides a standardized dataset containing both vulnerable and non-vulnerable Java code samples, classified by CWEs, which we use for our study since it allows for reproducibility and objective evaluation of the tools' performance across a variety of security issues derived from real code patterns. We use version v1.2 of the benchmark. There are 2740 total test cases in the benchmark dataset, 1,415 of which contain a true vulnerability and 1,325 do not. Some of the weaknesses in this dataset include:

- **Injection vulnerabilities** (e.g., SQL Injection, Command Injection, LDAP Injection, XPath Injection, and Cross-Site Scripting) exploit improper handling of user inputs, allowing attackers to execute arbitrary commands or manipulate queries.
- **Cryptographic weaknesses** (e.g., Weak Hashing, Weak Cryptography, and Weak Randomness) arise from using insecure algorithms or poorly implemented cryptographic operations.

- **Access and trust boundary issues** (e.g., Path Traversal and Trust Boundary Violations) occur when there is insufficient input validation or improper path construction limitation.
- **Cookie and session vulnerabilities** (e.g., Missing Secure Cookie Flags) involve mishandling session management.

The complete list of code samples divided by CWE can be found in Table 7.1.

Table 7.1: OWASP Vulnerable Testcases

CWE	Name	Positive Cases	Negative Cases
22	Path Traversal	133	135
78	Command Injection	126	125
79	XSS	246	209
89	SQL Injection	272	232
90	LDAP Injection	27	32
327	Weak Cryptography	130	116
328	Weak Hashing	129	107
330	Weak Randomness	218	275
501	Trust Boundary	83	43
614	Secure Cookie Flag	36	31
643	XPATH Injection	15	20

Each test case in the benchmark is composed of a Java source code file, e.g. texttttBenchmarkTest00001.java, and its corresponding XML file `BenchmarkTest00001.xml` containing metadata and the ground truth. The test cases in the OWASP benchmark have either one type of vulnerability (e.g., CWE 22) or no vulnerability at all. However,

non-vulnerable code files include *traps*: code patterns that resemble real vulnerabilities of a specific type but are actually safe.

The XML files contain the benchmark version, vulnerability category, CWE number, test file number, and whether the vulnerability exists. An example of this ground-truth description is illustrated in Figure 7.1.

```
<test-metadata>
    <benchmark-version>1.2</benchmark-version>
    <category>pathtraver</category>
    <test-number>00001</test-number>
    <vulnerability>true</vulnerability>
    <cwe>22</cwe>
</test-metadata>
```

Figure 7.1: XML Labels for the OWASP dataset. This test case shows the file has a CWE 22 vulnerability.

7.2.2 Large Language Models

LLMs are deep learning models designed to process and generate human language and pre-trained on vast amounts of data. The foundational structure of many LLMs is the Transformer architecture, which processes input data through multiple layers of "self-attention" to capture context and relevance within text sequences.

The key components of LLMs include tokens, parameters, and temperature. Tokens are text units, like words or subwords, that the model processes to understand and generate responses. The parameters, on the other hand, represent the internal configuration of the model, consisting of weights and biases that influence its decision making. A model with more parameters generally has greater capacity to understand complex patterns, although this also requires substantial computational resources. The temperature parameter controls the randomness of the model output: Lower values produce more deterministic responses, while higher values increase response diversity and creativity.

For this study, we used five different models: GPT-4, GPT-4o Mini, Claude 3.5 Sonnet, o1-preview, and o1-mini. Each of these has a different base model, maximum number of tokens, and cut-off date indicating when the model's training was last updated, as illustrated in Table 7.2. At the time of the experiments, both o1-preview and o1-mini are beta models. This means that they are previews of unreleased models and do not have all of the features from models such as GPT-4 and GPT-4o Mini.

Table 7.2: LLM model details

Model API	Base Model	Max Tokens	Cut Off
gpt-4-0613	GPT-4	8,192	2021-09
gpt-4o-mini-2024-07-18	GPT-4o Mini	128,000	2023-10
o1-preview-2024-09-12	o1-preview	128,000	2023-10
o1-mini-2024-09-12	o1-mini	128,000	2023-10
claude-3-5-sonnet-20240620	Claude 3.5 Sonnet	200,000	2024-04

Prompt Engineering techniques

LLMs rely on an input prompt from a user to elicit a specific language model behavior [181]. Prompt techniques can be as straightforward as directly asking an LLM to complete a task without any examples (Zero Shot), to performing in-context learning by providing examples of the task (e.g., Few Shot) to thought generation (e.g., Chain of Thought) and decomposition (e.g., Tree of Thoughts) techniques to prompt the LLM to articulate its reasoning while solving a problem and decompose the problem into simpler sub-questions.

7.3 Methodology

We initially test the OWASP benchmark using Semgrep, and the results are stored in a SARIF file (SARIF is an OASIS standard defining the output file format of static analysis tools).

These SARIF results are imported into a pipeline where we can select an LLM and create prompts for evaluation. Each prompt includes the code from a dataset file and the corresponding SARIF file. The LLM then assesses whether the code contains a specific vulnerability, marking each finding as true or false.

The LLMs are compared to the ground truth in the OWASP dataset and are classified as true positives, false positives, true negatives, or false negatives. To ensure consistency and enable a fair comparison of different LLMs, the same SARIF file is used for all pipeline runs, eliminating variability from rerunning Semgrep.

7.3.1 Experiment Prompts

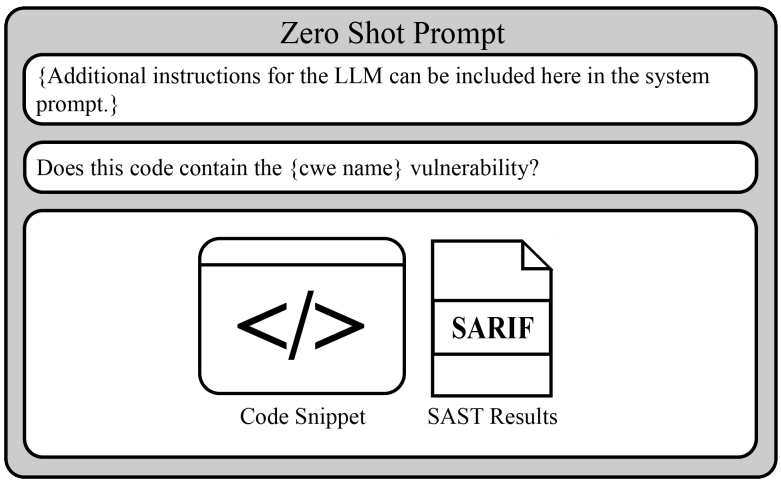

Figure 7.2: Zero Shot Prompt Template

We employ three prompting strategies to evaluate LLMs for vulnerability detection:

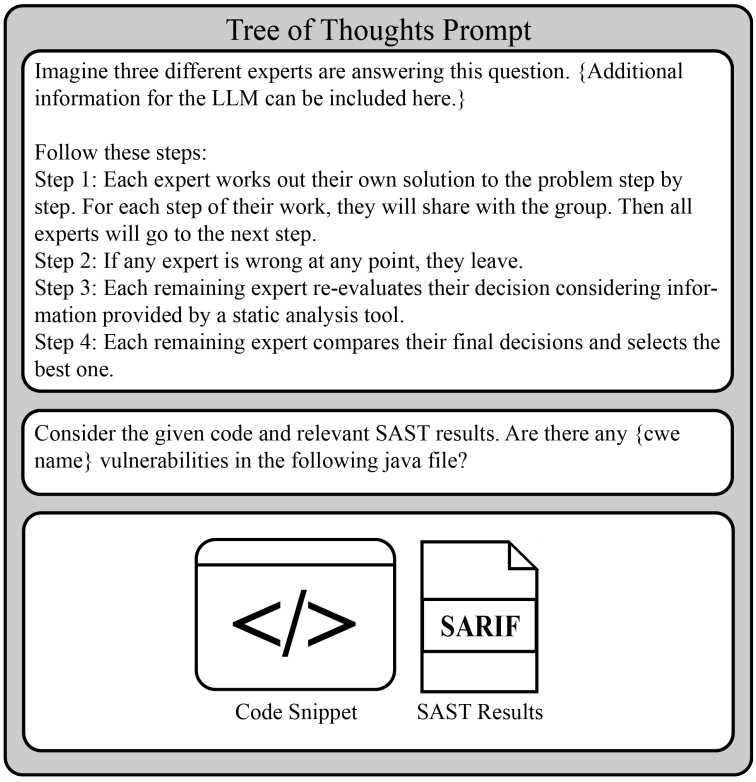

Figure 7.3: Tree of Thoughts Prompt Template

Zero Shot (ZS), Tree of Thoughts (ToT), and Chain of Thought (CoT). The Zero Shot prompt, based on [207], uses minimal context, relying on the pre-trained capabilities of the LLM. The Tree of Thoughts approach, inspired by [100], decomposes the problem into exploratory paths to improve reasoning. Finally, the Chain of Thought prompt guides the LLM through step-by-step reasoning for structured analysis.

The format of these prompts and the additional information are summarized in Figures 7.2-7.4. The `cwe name` in the prompt corresponds to the CWE in the ground truth of the test case, as in Figure 7.1.

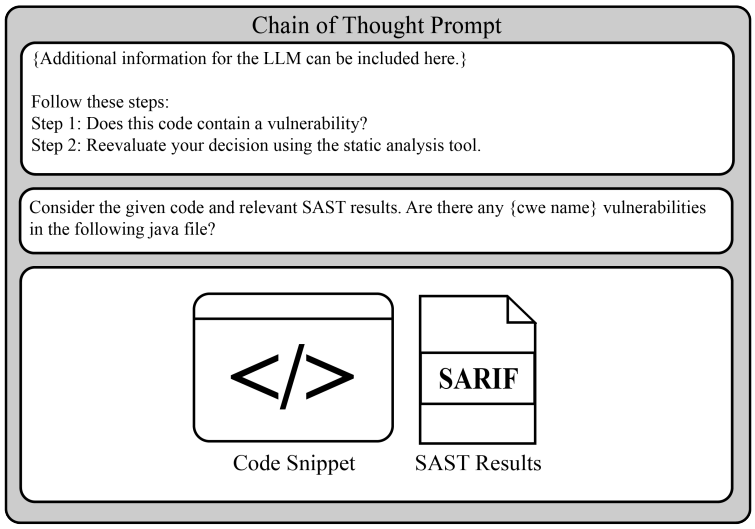

Figure 7.4: Chain of Thought Prompt Template

7.3.2 Evaluation Metrics

To evaluate the performance of LLMs, we use various metrics from classification and machine learning. These metrics provide insights into how accurately the models detect vulnerabilities in the code snippets provided [102]:

- **True Positives (TP)**: Number of correct predictions for the positive class.
- **False Positives (FP)**: Negative-class instances incorrectly identified as positive cases.
- **False Negatives (FN)**: Positive instances erroneously predicted as negative.
- **True Negatives (TN)**: Negative class instances accurately predicted as negative.

Out of these core metrics, we compute summary statistics of the performance of the classifier, such as the Accuracy (the proportion of correct predictions over the total number of predictions), the F1 Score (the harmonic mean of Precision and Recall), the ROC Curve (the tradeoff between the True Positive Rate (TPR) and the False Positive

Rate (FPR)) and the Area Under the Curve (AUC) (the probability that the model ranks a randomly chosen positive instance higher than a randomly chosen negative one).

Table 7.3: LLM Performance Comparison

Model	TP	FP	TN	FN	TPR	FPR	Accuracy	F1	AUC
Semgrep	1139	518	807	276	80.49%	39.09%	71.02%	74.15%	70.8%
GPT-4	1296	632	693	119	91.59%	47.70%	72.59%	71.20%	79.03%
GPT-4o Mini	1370	787	538	45	96.82%	59.40%	69.64%	66.55%	68.99%
Claude 3.5 Sonnet	1362	412	913	52	96.32%	31.09%	83.06%	82.59%	82.61%
o1-preview	1396	478	847	19	**98.66%**	36.08%	81.86%	81.10%	81.29%
o1-mini	1326	266	1059	89	93.71%	**20.08%**	**87.04%**	**86.92%**	**86.82%**

7.3.3 Experiments

We evaluate the performance of LLMs in triaging vulnerabilities through three different experiments based on our research questions:

Comparing Different LLMs

In this experiment, we compare the performance of five LLMs previously introduced in analyzing security findings.

To ensure comparability, each model analyzes the complete benchmark dataset using a standardized Chain of Thought prompt shown in Figure 7.4, with a fixed temperature setting of $T = 0$.

Comparing LLM and Semgrep Performance

The second experiment focuses on a performance comparison between Semgrep with and without the assistance of a LLM. The tools evaluate the OWASP dataset, using the same chain-of-thought prompt as in Experiment 1.

Comparing Different Prompts

In the third experiment, we examine the effects of using different prompts on an LLM's ability to triage results from Semgrep. We tested several types of prompts while maintaining the same temperature $T = 0$, and sample size (200) constant.

7.4 Results

Experiment 1

Table 7.3 displays the results for the Semgrep baseline and then the five LLMs tested.

The results show that the baseline model, Semgrep, has a true positive rate of 80.49% and a false positive rate of 39.09%. Notably, all LLMs increase the true positive rate while also reducing false positive rates in most cases.

We performed our first experiments in the summer of 2024 before the release of o1-preview and o1-mini by OpenAI in September 2024, and in our initial results, the best-performing model was Claude 3.5 Sonnet. However, as we can see in the table, the new o1-mini significantly increases the performance of vulnerability detection showing higher accuracy, F1-score, and AUC. Although other models have a higher TPR, the reduction in the number of false positives by o1-mini results in a superior overall performance. Given the typically high false positive rates of static analysis tools, reducing false positive rates is one of the most important objectives for triaging.

Claude 3.5 Sonnet ranks second, with high accuracy, F1-score, and AUC, closely followed by o1-preview.

GPT-4o mini achieves a high true positive rate, but also records the highest false positive rate, resulting in the lowest overall accuracy, F1 score, and AUC.

The o1-mini model outperforms o1-preview despite being a smaller model. o1-preview was trained on a more general dataset with broader knowledge, while o1-mini's training was geared toward STEM and coding tasks. This difference might be the reason o1-mini is better in our experiments.

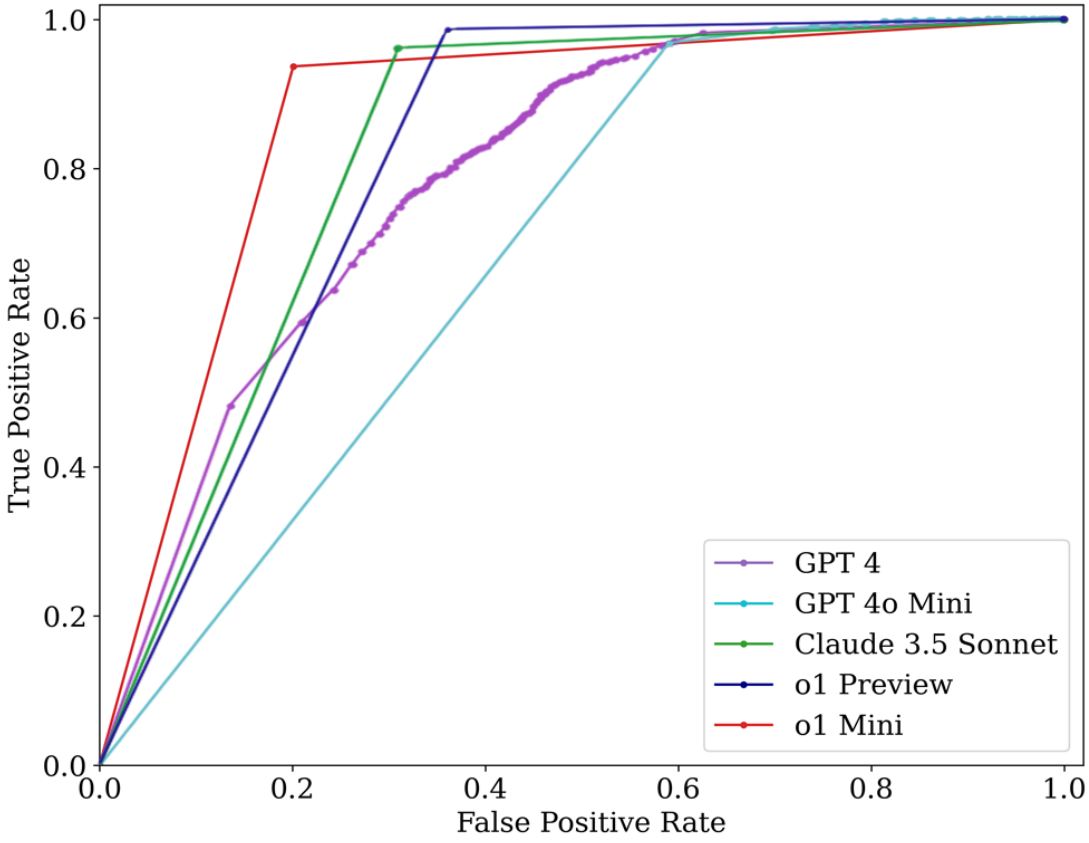

Figure 7.5: Different LLMs' ROC Curves for Triaging

To report the results in Table 7.3, we selected the classifier confidence threshold to maximize the difference between TPR and FPR.

For a more general understanding of the performance of classifiers under different thresholds, we now turn to ROC curves. Figure 7.5 presents the ROC curves for the five models tested. The ROC curves for o1-mini, o1-preview, and Claude 3.5 Sonnet all consistently sit above GPT-4 and GPT-4o mini due to their superior TPRs and FPRs. Although Claude 3.5 Sonnet and o1-preview have points that sit higher on the graph, o1-mini's curve has a lower false positive rate and a visibly larger AUC as shown in table 7.3.

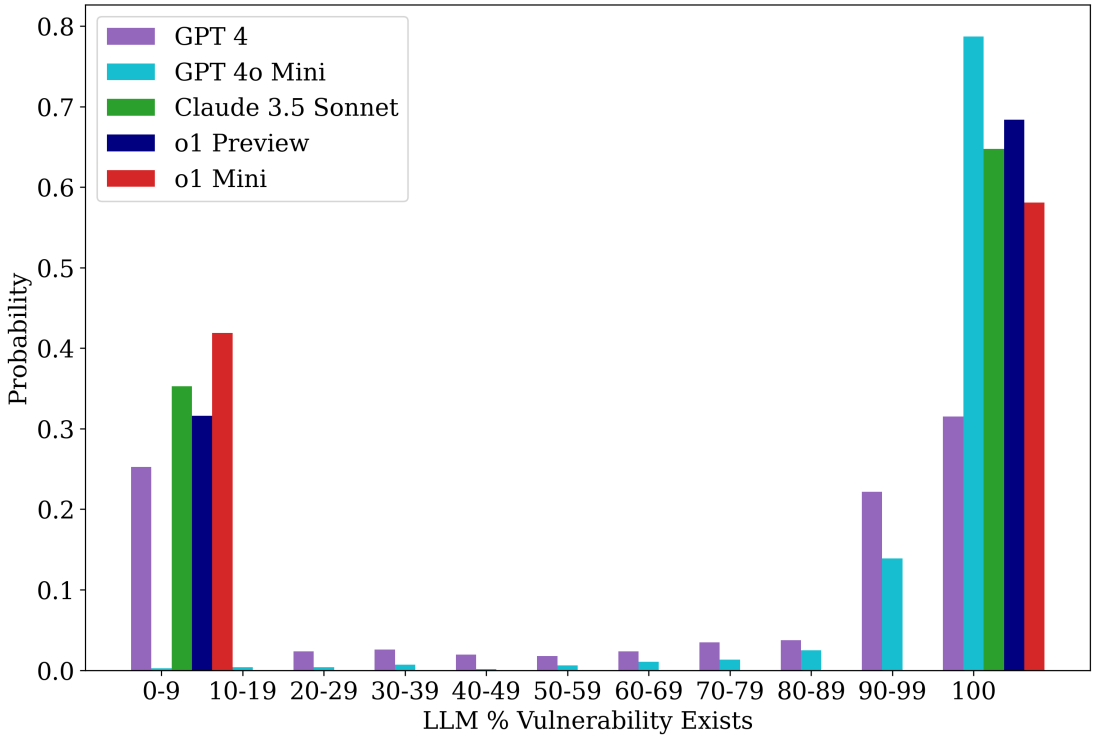

Figure 7.6: Probability Mass Function of Percent Probability that a Vulnerability Exists for different LLMs

The graph also reveals a noticeable variation in the number of points in the ROC curves

for different models. These points correspond to different TPRs and FPRs as the threshold for the vulnerability probability changes. In particular, o1-preview and o1-mini only have a single point on their ROC curves. This is because they are both beta models without support for features like logprobs, and consequently they do not return any vulnerability probability. Claude 3.5 Sonnet also does not return logprobs like most OpenAI models do, and so it has minimal points as well. The more varied vulnerability probability percentages seen when using GPT-4 result in more visible points in its ROC curve as the threshold for identifying a positive test case is shifted. Figure 7.6 illustrates the probability of each LLM certainty score for the different models and explains why some LLM models have a more continuous ROC curve trade off than others.

Experiment 2

Now we turn our attention to how Semgrep and LLMs classify individual CWEs to determine whether some CWE vulnerabilities are easier to detect than others. For this we use o1-mini as it performed the best in the previous experiment, and we select GPT-4 as a baseline to illustrate how LLMs are improving over time.

Figure 7.7 compares the performance of GPT-4 and o1-mini against Semgrep in identifying individual CWEs. The legend on the right calculates a score for each CWE category by subtracting the FPR from the TPR. The red dotted line represents random guessing, and the shaded area indicates performance levels below random guessing. GPT-4 improves the most in detecting CWEs 327, which correspond to the weak encryption algorithm vulnerability. Although GPT-4 has a true positive rate of 100%, Semgrep does not label any files as containing a weak encryption algorithm within the benchmark.

While GPT-4 performs better in the CWE 327 and 328 categories, it underperforms for the majority of CWEs. In most cases, it detects more true positives than Semgrep, but also reports a higher false positive rate. GPT-4's score for CWE 501 is notable as it falls below both Semgrep and the random guessing threshold.

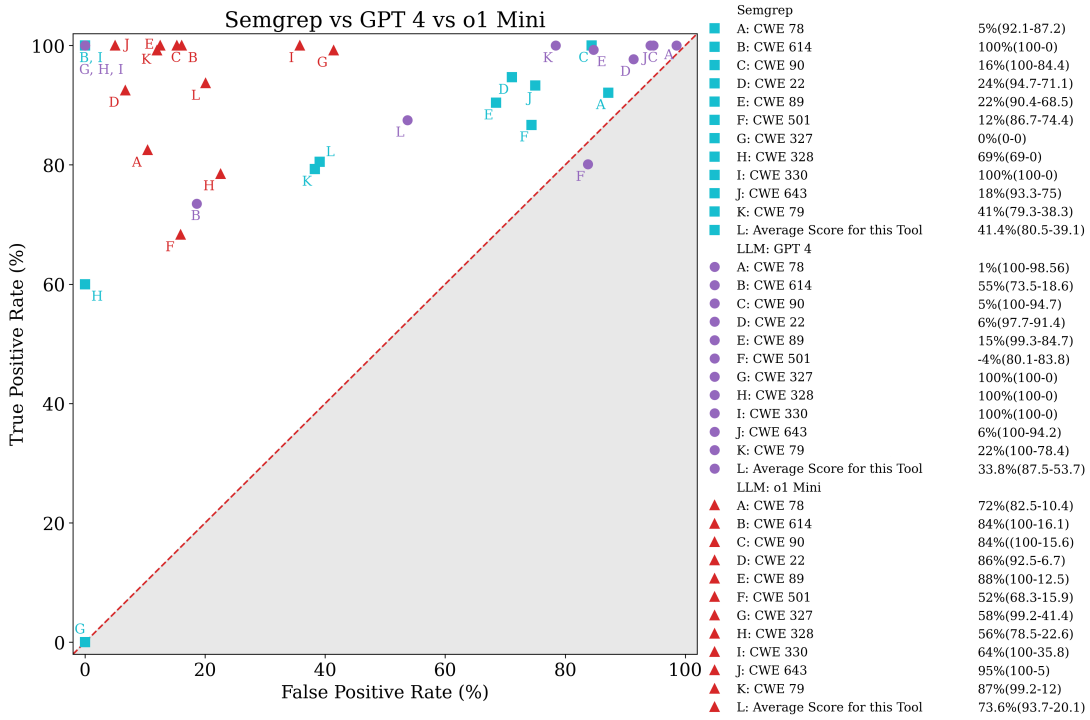

Figure 7.7: Performance of Semgrep vs Different LLMs (Using Semgrep)

The o1-mini model, however, performs better than Semgrep in all but three categories (CWE 614, 328, and 330). The o1-mini average performance in identifying vulnerabilities is also much higher than both GPT-4 and Semgrep.

To summarize these results, we focus on how better (or worse) the o1-mini model performs on individual vulnerabilities when compared to Semgrep.

Table 7.4, shows a ranked list based on the F1 score difference between the performance of Semgrep ($F1_S$) and the performance of o1-mini ($F1_o$) for each individual vulnerability. We can see how o1-mini has a higher F1-score than Semgrep in most CWE categories. The highest increase in the F1-score from Semgrep to o1-mini is for CWE 327 (weak cryptography). In this case, Semgrep failed to identify most of the weak cryptography cases, so this is an example of the LLM overriding the assessment

Table 7.4: Comparing Semgrep and o1-mini F1-Scores by CWE

Rank	CWE	$F1_S$	$F1_o$	$F1_o - F1_S$
1	327	0.0	0.83	0.83
2	643	0.71	0.98	0.27
3	89	0.71	0.94	0.23
4	90	0.717	0.93	0.22
5	22	0.72	0.93	0.21
6	79	0.74	0.94	0.20
7	78	0.67	0.86	0.19
8	501	0.68	0.75	0.07
9	328	0.82	0.79	-0.03
10	614	1.0	0.93	-0.07
11	330	1.0	0.86	-0.14

of the SARIF file and indicating that there is indeed a vulnerability, even if the SARIF file states that there is none.

There are three vulnerabilities where o1-mini performs worse than Semgrep: CWE 328, 614, and 330. The significantly higher false positive rate for all three of these categories when evaluated by o1-mini significantly impacts these scores even though the true positive rate remains the same or improves. This highlights an area for improvement in LLM-based detection, particularly for weak randomness and weak hashing vulnerabilities.

o1-mini's improved F1-scores for CWEs such as CWE 22, 79, and 89 in contrast to the lower scores for CWEs 501, 328, and 327 indicate that the LLM is better able to identify vulnerabilities that do not require additional context.

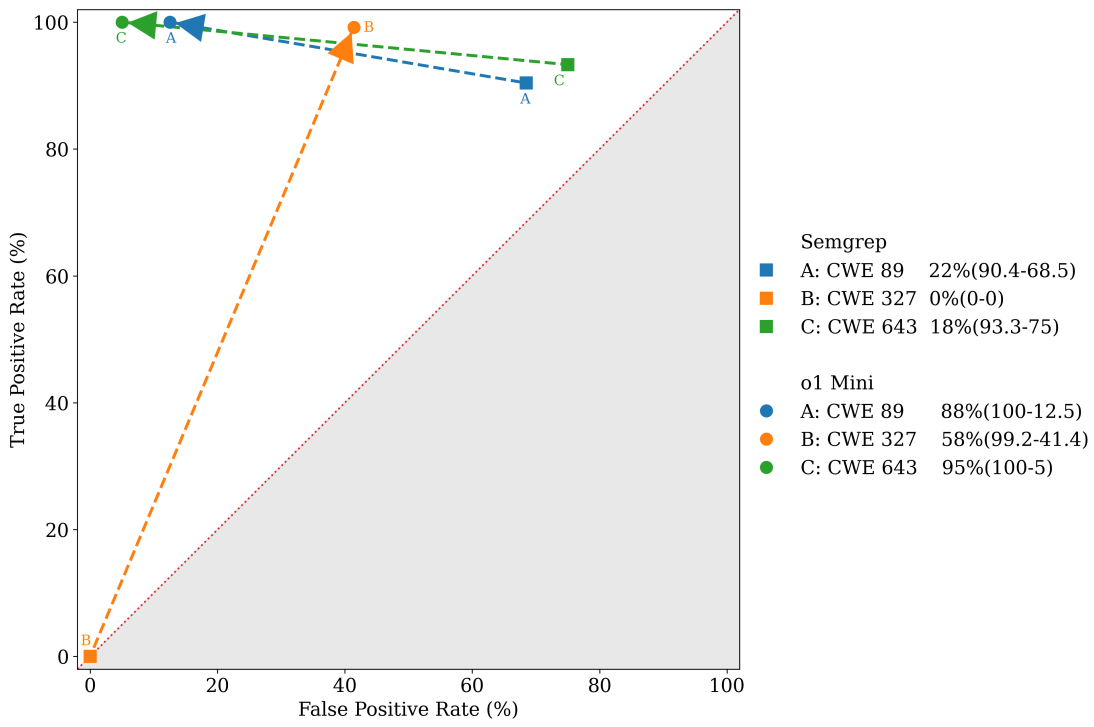

Figure 7.8: Semgrep vs o1-mini Performance for Best CWEs by F1-Score Improvement

Figure 7.8 shows the three most significant improvements of o1-mini over Semgrep (from table 7.4). We can see that for CWE 327 (weak cryptography), the LLM increases the detection rate at the cost of some false positives. However, for CWE 89 (SQL injection) and CWE 643 (XPATH injection), the LLM significantly reduces the false positive rate, while slightly increasing the true positive rates.

Figure 7.9 similarly shows the three cases in which o1-mini underperformed compared to Semgrep. Although the detection rate either improved or remained the same, the increased false positive rate negatively affected the overall F1 score. In a high-criticality setting, where missing vulnerabilities poses a greater risk than handling false positives, this trade-off might be acceptable. Therefore, we argue that o1-mini generally outperforms Semgrep if minimizing false negatives is prioritized over reducing false alerts.

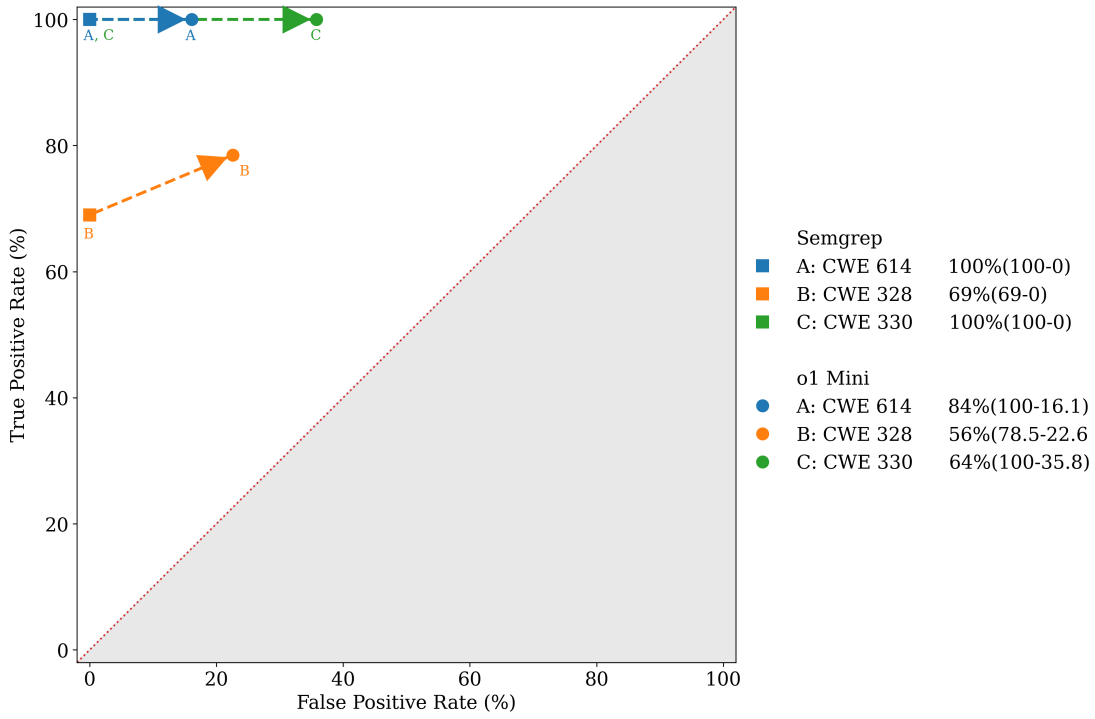

Figure 7.9: Semgrep vs o1-mini Performance for Worst CWEs by F1-Score Improvement

Experiment 3

We now discuss the performance of different prompts. Figure 7.10 illustrates the performance of different prompts with the same LLM models tested in Experiment 2.

For GPT-4 we can clearly see that the Tree of Thoughts prompt curve clearly lies above all other GPT-4 curves, showing that more efforts in prompt engineering may increase the performance of the models.

Notice, however, that despite being the best performing prompt for GPT-4, the Tree of Thoughts prompt curve for GPT-4 is still in the convex hull of the worst performing o1-mini prompt (Zero Shot).

Table 7.5 shows more detailed results for the o1-mini case. We can see that the o1-mini

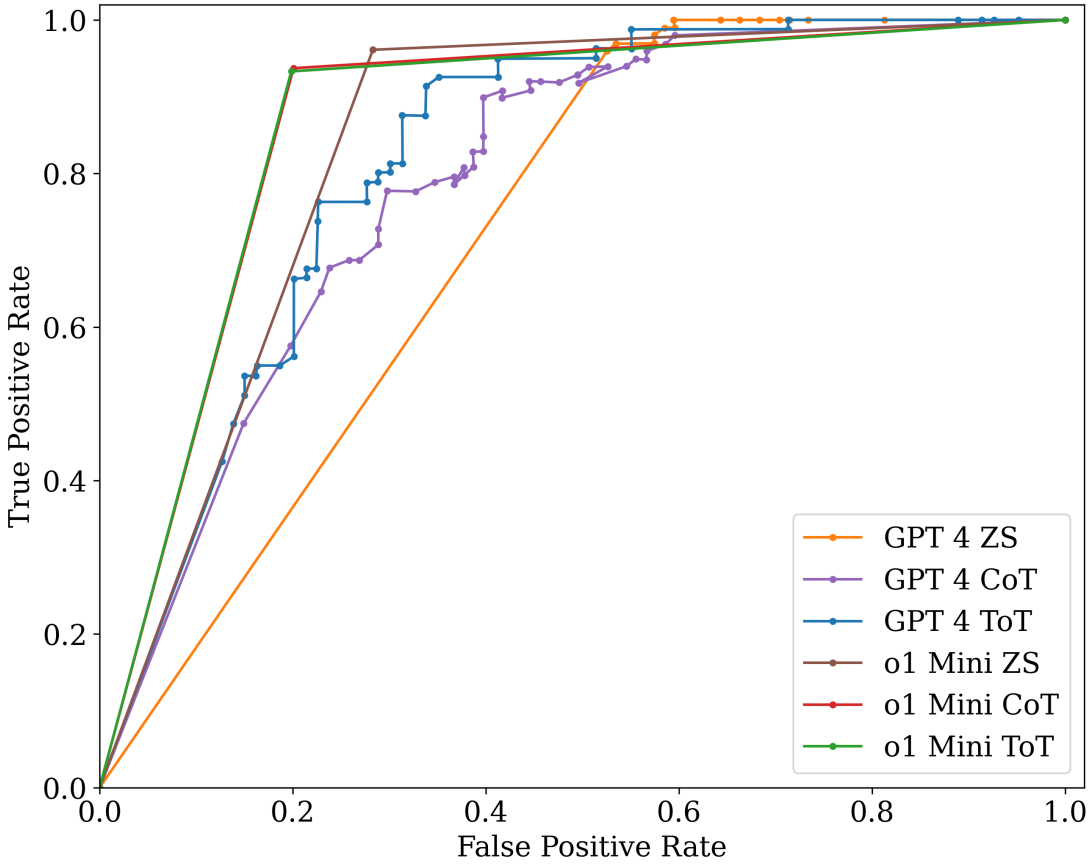

Figure 7.10: ROC Curves for Different Prompt Styles and LLMs

model, the more complex prompts performed similarly to even the simple Zero Shot prompt. While a more complicated Chain of Thought or Tree of Thoughts prompt was still a noticeable improvement over the Zero Shot prompt (particularly in regard to the FPR), the difference in their accuracy, F1-score, and AUC were all under 0.3%.

7.4.1 Summary and Analysis

Our study demonstrates the potential of combining LLMs with static analysis tools such as Semgrep for vulnerability detection. The results indicate that while GPT-4, which

Table 7.5: LLM and Prompt Performance Comparison

Model	Prompt	TP	FP	TN	FN	TPR	FPR	Accuracy	F1	AUC
o1-mini	Zero Shot	1360	375	950	55	**96.11%**	28.30%	84.31%	83.95%	83.91%
o1-mini	Chain of Thought	1326	266	1059	89	93.71%	20.08%	**87.04%**	**86.92%**	**86.82%**
o1-mini	Tree of Thoughts	1320	263	1062	95	93.29%	**19.85%**	86.93%	86.82%	86.72%

was state-of-the-art a couple of months ago and proved to be impressive on a variety of tasks, fell short on vulnerability detection. However, newer "reasoning" models, such as o1-mini, surpass the performance of previous models, achieving significantly better accuracy and lower false positive rates. For example, o1-mini exhibited the highest accuracy (87.04%) and F1 score, suggesting that newer models are progressing toward achieving reliable vulnerability detection and triaging.

We also show that the performance of LLMs is significantly dependent on the type of vulnerability. Weak cryptography seems to be a strong suit for LLMs, while finding weak randomness seems to be hard for them. In future work, we will attempt to find the reasons for this behavior.

Our findings also show that, depending on the model, prompt engineering can play a critical role in using LLMs full capabilities. Models like GPT-4 benefit greatly from better prompt design, while newer models like o1-mini require less prompt engineering effort since they were designed to perform complex *reasoning* by "thinking" before answering, so while a CoT or ToT prompt might improve their performance, the gains are not as significant when applying these prompts to earlier models.

Limitations One of the limitations of our study is that the evaluation setup is highly specific, relying on a fixed benchmark and a limited set of CWEs, which may not fully represent real-world vulnerability distributions. Our study assumes that each vulnerability corresponds to a single CWE, whereas, in practice, vulnerabilities often exhibit

multiple CWE classifications. This suggests that the problem may be more appropriately framed as a multi-label classification task, requiring different evaluation metrics. Finally, another potential problem is data contamination, as the models may have been trained on the OWASP benchmark; this raises the need to regularly update benchmarks to test newer LLM models with tests that could not have been seen by the LLMs before.

7.5 Related work

LLMs have been explored for a range of complex tasks within software engineering. Wu et al. [229] highlights LLMs strengths in code generation and reasoning tasks, and Feng and Chen [67] use LLMs to automate bug reproduction, outperforming existing methods in speed and accuracy. Yuan et al. [233] show that fine-tuning LLMs improves their code comprehension and generation abilities, while Zhang et al. [239] apply LLMs to resolve GitHub issues, achieving better results than traditional tools.

In terms of software security, LLMs have been used in vulnerability detection. Ullah et al. [207] and Ding et al. [59] highlight the limitations of LLMs in reliably identifying security vulnerabilities. In particular, LLMs such as GPT-4 and PaLM2 struggle with accurately detecting bugs [207]. Other approaches like improved labeling [59], customized prompts [195], and retrieval-augmented generation [60] have been attempted but still show that current LLMs appear to be unreliable for such tasks.

Other works study the use of LLMs for vulnerability repair. Pearce et al. [170] explore the use of LLMs for zero-shot vulnerability repair, highlighting both the potential and challenges in generating functionally correct code while addressing security bugs. Islam et al. [106] introduce SecRepair, which combines reinforcement learning with LLMs to generate secure code fixes and vulnerability descriptions. Similarly, Fitero-Dominguez et al. [55] present a fine-tuning approach with LLMs to improve accuracy in vulnerability repair. Although these methods show potential, they still face challenges in producing accurate, secure patches, often requiring further human refinement

and contextual understanding.

Despite these challenges, LLMs can be effective in detecting certain types of vulnerabilities with advanced prompting techniques, pointing to the potential of LLMs when combined with neuro-symbolic approaches [114, 156]. Symbolic tools, such as static analyzers, have the ability to assist in code development. Previous work has focused on comparing static analysis tools for vulnerability detection [25, 115] and web application security testing [116], but comparatively less emphasis has been placed on enhancing the capabilities of static analysis tools through artificial intelligence for this particular purpose. LLift [124] combines static analysis with LLMs for bug detection in the Linux kernel, demonstrating the potential of AI to enhance traditional analysis, and IRIS [125] uses a neuro-symbolic approach that integrates LLMs with CodeQL to infer taint specifications. Although promising, these approaches are domain-specific, focusing on particular systems or vulnerability classes.

Our work addresses this gap by combining LLMs with Semgrep for a more comprehensive investigation on how LLMs can improve the performance of static analysis tools with different prompt techniques, models, and vulnerability classes.

7.6 Conclusions and Future Work

This study shows promising progress towards the practical use of LLMs to improve software vulnerability detection. LLMs have the potential to reduce false positives, accelerate the triaging processes, and eventually eliminate the long and arduous task of triaging vulnerabilities in software.

In future work, we plan to evaluate LLM-based vulnerability detection on more complex and realistic datasets. The OWASP Benchmark consists of Java code examples, each one corresponding to a single CWE and are contained within a single file. In practice, real-world vulnerabilities often span multiple files, involve multiple CWEs,

and appear in various programming languages. Expanding our evaluation to include datasets that reflect these complexities will provide deeper insight into the strengths and weaknesses of LLMs in enhancing static analysis tools. Furthermore, since our findings suggest that LLMs introduce a trade-off between improved detection and higher false-positive rates, future work should explore hybrid approaches that combine LLM reasoning with rule-based static analysis to balance precision and recall in security analysis.

7.7 Acknowledgments

We acknowledge AppSecAI Inc. for providing the platform and support essential to conduct the high volume of experiments that underpin this chapter. Research was sponsored by the Army Research Office and was accomplished under Grant Number W911NF-23-1-0373. For this research, we also collaborated with OpenAI through the Cybergrant program, using their API credits and funding to advance our research in AI-driven cybersecurity solutions.

Appendix A

Alternative Repository Affiliation Classification Methods

In addition to the approach presented in the main text, we explored several alternative filtering methods. While these methods provided useful insights, they were ultimately not used in our final analysis due to lower performance or practical limitations.

A.1 Score-Based and Embedding Methods

A.1.1 Method 1: Score Based Classifier (SBC)

Our baseline system is inspired by previous work on feature engineering; where a feature of the repository, such as finding a keyword associated with a university in the metadata, is assigned a score. The intuition is that whenever we detect a feature contributing to the repository's signature of affiliation, we increase the probability that the SBC model labels it as a positive case.

In particular, to estimate the probability p_i that a repository r_i is affiliated with a uni-

versity, we apply a rule-based scoring system consisting of three main steps. First, we perform a keyword search across multiple data attributes. This includes scanning the repository's metadata component repo_i including its name, description, homepage, and README, for institution-specific keywords (full name, acronym, email domain, and alternate name). We also examine the organization metadata org_i. If the owner is an organization, we inspect its attributes: name, description, email, URL and company for matches. Additionally, we analyze the top two contributors $\text{contrib}_i^{(1)}$ and $\text{contrib}_i^{(2)}$, examining attributes such as email, name, bio, and company to determine potential university affiliation. Each attribute a has an associated probability score $s_a \in [0,1]$ representing how likely a match in that attribute indicates university affiliation. Let:

- $m_{i,a} \in \{0,1\}$ indicate whether repository r_i has a match in attribute a,
- p_i be the total probability estimate for repository r_i.

The total probability is computed as the sum of individual attribute probabilities, capped at 1:

$$p_i = \min\left(\sum_{a\in\text{repo}_i} s_a m_{i,a} + \sum_{a\in\text{org}_i} s_a m_{i,a} + \sum_{j=1}^{2}\sum_{a\in\text{contrib}_i^{(j)}} s_a m_{i,a},\ 1\right)$$

The specific scores are heuristically tuned using high-confidence attributes and are detailed in Table A.1.

A.1.2 Method 2: Text Embeddings & Machine Learning

Instead of relying only on keyword matches, our next approach creates a feature vector by using an embedding with the repository's signature, and then uses that feature vector as an input to a machine-learning classifier. The embedding captures the broader context in which university-related terms appear. For example, the phrase "developed by UC Santa Cruz" carries much stronger affiliation than an incidental mention of the campus name.

Metadata	Attribute Checked (a)	Matched Criteria	Probability Score (s_a)
repo_i	`homepage`	University domain	1.0
	`readme`, `description`, `name`	University name, acronym, or alt. queries	0.20 per match per attribute
org_i	`url`	University domain	1.0
	`email`	University domain	1.0
	`name`, `description`, `company`	University name, acronym, or alt. queries	0.30 per match per attribute
$\text{contrib}_i^{(j)}$	`email`, `name`, `bio`, `company`	University domain	0.50 per match per attribute
	`email`, `name`, `bio`, `company`	University name, acronym, or alt. queries	0.20 per match per attribute

Table A.1: Scoring system for estimating university affiliation.

Embedding: We aggregate the repository attributes into a single text block. For each repository r_i, we define:
$\mathbf{m}_i = \left(\text{repo}_i, \text{org}_i, \text{contrib}_i^{(1)}, \text{contrib}_i^{(2)}\right)$, where each component is a set of textual fields. We concatenate these fields into a single raw text string $t_i \in \mathbb{T}$, such that $t_i = \text{concat}(\mathbf{m}_i)$. We finally transform each t_i into a vector representation $\mathbf{v}_i \in \mathbb{R}^d$ using OpenAI's text embedding model [165] $\mathcal{E} : \mathbb{T} \to \mathbb{R}^d$. Given the input size constraints of the small embedding model, we truncate each text field to a maximum of 20,000 characters. Since the README is typically the longest field, this truncation primarily affects its content.

Training: We randomly sample 200 repositories and manually assign a binary label to each one, $y_i \in \{0,1\}$, where $y_i = 1$ indicates university affiliation. We intentionally avoid balancing the dataset to preserve the natural class distribution.

We tested several machine learning models, including neural networks, random forests, and Support Vector Machines (SVM). The classification results were very similar among all models, but SVMs gave slightly better results, so we showcase them for the rest of this paper. After training, for each repository, the model outputs a probability per class: $p_i = f_{\text{prob}}(\mathbf{v}_i)$, where p_i is the predicted probability that r_i is affiliated with the target university.

This approach requires a manually labeled training dataset, which introduces significant overhead in terms of human effort and time.

A.2 Cost and Runtime Comparison

Given the complexity of detecting affiliated repositories, it is also important to consider the practical trade offs in cost and runtime across different modeling approaches. Table A.2 shows a cost comparison for different methods of processing GitHub repository data. For the SVM approach the cost reflects transforming the text into vector embeddings. The total time is computed by adding the embeddings time to the model training and prediction time, which in most cases is around 0.70 seconds. For the other methods, costs are based on using the OpenAI API with prompts whose length includes both dynamic repository data and a static prompt component. We compute average input sizes by concatenating all relevant data fields, resulting in approximately 2,900 characters of dynamic text plus about 1,600 characters of static prompt text for the API models. The cost estimates are scaled to 236K repositories, which corresponds to the total number of repositories found for the UC System. The time is computed for `gpt-4o` and `gpt-3.5` For `gpt-5-mini`, which is much slower, we report the single-threaded runtime estimated from multithreaded benchmarks.

Model	Chars	In Tok.	Out Tok.	Price I/O	$/Repo	$/236K	Min.
SBC	–	–	–	$0 / $0	$0	$0	18
SVM	2,900	725	–	$0.00002 / –	$0.0000145	$3.42	1,022
gpt-3.5	4,500	1,125	100	$0.50 / $1.50	$0.00071	$167.6	3,708
gpt-4o	4,500	1,125	100	$2.50 / $10.00	$0.00381	$899.75	4,330
gpt-5-mini	4,500	1,125	100	$0.25 / $2.00	$0.00048	$113.46	30,342

Table A.2: Cost and average time comparison for 236,000 inputs using OpenAI models, based on average input size and token pricing. Single-threaded time is used for `gpt-5-mini`.

While `gpt-5-mini` achieves competitive classification accuracy and is relatively inexpensive compared to `gpt-4o`, its runtime presents a significant practical limitation for large-scale datasets. As shown in Table A.2, processing 236,000 repositories with `gpt-5-mini` costs just over $100, making it a cost-effective option. However, the single-threaded runtime is estimated to exceed 21 days, reflecting its much slower processing speed compared to `gpt-4o` or `gpt-3.5`. These results illustrate a key tradeoff between cost, accuracy, and efficiency, emphasizing that model selection should consider not only predictive performance but also practical constraints such as dataset size and processing time.

Appendix B

Sustainability Metrics for Maturity-Staged Framework

B.1 Sustainability metrics: definitions and formulas

Table B.1: Sustainability metrics: definitions and formulas. t denotes the data collection date; $t-90$, $t-180$ denote 90 and 180 days before collection. W_{90} = 90-day window. C = commits, I = issues, P = pull requests, R = reviews, M = merges, A = authors (distinct logins), τ = days elapsed.

Metric	**Formula**	**Source**
Community		
Unique contributors	$\lvert A_C \cup A_P \cup A_I \cup A_R \rvert$	[12]
Active contributors (W_{90})	$\lvert A_C \cap W_{90} \rvert$	[40]
Organizational diversity (W_{90})	$\left\lvert \{ d(a) : a \in A_C(W_{90}),\ d(a) \notin D_{\text{personal}} \} \right\rvert$	[40]
Contributor retention (W_{90})	$\frac{\lvert A_{t-90 \to t} \cap A_{t-180 \to t-90} \rvert}{\lvert A_{t-180 \to t-90} \rvert} \times 100$	[12]
Contributor growth (W_{90})	$\frac{\lvert A_{t-90 \to t} \rvert - \lvert A_{t-180 \to t-90} \rvert}{\lvert A_{t-180 \to t-90} \rvert} \times 100$	[12]
Bus factor	$\min k : \sum_{i=1}^{k} c_i > 0.5 \sum c$	[20]
Community interest	$\text{stars} + \text{forks} + \text{watchers}$	[12]
Activity		
Social activity (W_{90})	$\lvert C_{W_{90}} \rvert + \lvert I_{W_{90}} \rvert + \lvert P_{W_{90}} \rvert + \lvert \text{comments}_{W_{90}} \rvert$	[40]
Commit frequency (W_{90})	$\lvert C_{W_{90}} \rvert / 12.857$	[40]
Commit growth (W_{90})	$\frac{\lvert C_{t-90 \to t} \rvert - \lvert C_{t-180 \to t-90} \rvert}{\lvert C_{t-180 \to t-90} \rvert} \times 100$	[40]
Issue closed ratio	$\frac{\lvert \{ i \in I : \text{state}(i) = \text{closed} \} \rvert}{\lvert I \rvert}$	[40]
Issue response time	$\overline{\tau(\text{first\_response} - \text{created})}_I$	[40]
PR response time	$\overline{\tau(\text{first\_review} - \text{created})}_P$	[40]
PR merge time	$\overline{\tau(\text{merged\_at} - \text{created})}_P$	[40]
Code review ratio	$\frac{\lvert \{ p \in P_{\text{merged}} : N_{\text{reviews}}(p) \geq 1 \} \rvert}{\lvert P_{\text{merged}} \rvert}$	[169]
Release frequency (W_{90})	$\lvert \text{releases}_{W_{90}} \rvert / 3$	[40]
Contributors dev. activity (W_{90})	$\lvert C^{\text{non-core}}_{W_{90}} \rvert + \lvert I^{\text{non-core}}_{W_{90}} \rvert + \lvert P^{\text{non-core}}_{W_{90}} \rvert + \lvert \text{comments}^{\text{non-core}}_{W_{90}} \rvert$	[12]
Maintainers dev. activity (W_{90})	$\lvert C^{\text{core}}_{W_{90}} \rvert + \lvert I^{\text{core}}_{W_{90}} \rvert + \lvert P^{\text{core}}_{W_{90}} \rvert + \lvert \text{comments}^{\text{core}}_{W_{90}} \rvert + \lvert M^{\text{core}}_{W_{90}} \rvert$	[12]
Maintainer recency	$\tau(\text{today} - \max_{a \in \text{core}} \text{last\_activity}(a))$	[128]
General		
Project age	$\tau(\text{today} - \text{created\_at})$	[12]
Financial support	`FUNDING.yml`	[128]
Development Practices		
CI/CD	`.github/workflows/` or CI config file	[43, 164]
Build environment	Count of build system files (Makefile, pom.xml, etc.)	[128]

Appendix C

Supply Chain Security Expert Interview Profiles and Protocols

C.1 Interview Expert Descriptions

The expert findings cited throughout chapter 6 draw on nine interviews conducted between September 2025 and April 2026. All experts are referenced anonymously as Experts 1 through 9. Experts 1–6 are software supply-chain security practitioners; Experts 7–9 are trade security practitioners with direct experience implementing CTPAT and AEO programs.

Expert 1—Software Supply-Chain Security Practitioner and Educator
A director-level practitioner leading open source supply-chain security at a major international open source foundation, with nearly fifty years of experience in software development and security. Author of a widely-used secure development course with over 60,000 enrollees, with direct engagement with government bodies and standards organizations including the EU Cyber Resilience Act process.

Expert 2—Open Source Security Lab Leader

A senior leader running security research operations at a major software development platform, responsible for securing the open source supply chain at scale. Brings approximately thirty years of software engineering experience, including a decade focused on supply-chain security.

Expert 3—Technology Policy Researcher, AI and Open Source Governance
A researcher at an international policy think tank working on AI governance, OSS standards, and technology policy, with a focus on the Global South and voluntary governance mechanisms. Examines how OSS governance frameworks function across different regulatory environments and how fragmentation of international norms affects open source security and sustainability.

Expert 4—Academic Researcher and Supply-Chain Security Tool Builder
A university professor whose research focuses on building widely-used software supply-chain security tools, all freely available under major open source foundations. Holds a strong view that government liability frameworks are a necessary complement to voluntary security practices.

Expert 5—Research Scientist in Data Center and Cloud Security
A research scientist specializing in cloud and data center security, applying hardware and software security technologies to supply-chain integrity for approximately six years. Expertise in key management, cryptographic signing, and dependency management tooling.

Expert 6—Open Source Security Researcher and Long-Tenured Package Maintainer
A security researcher with over a decade of experience maintaining packages in a major Linux distribution, combined with research focused on supply-chain security. Brings a socio-technical framing to the field and advocates for provenance-based approaches and greater commercial responsibility for open source dependencies.

Expert 7—Retired US Customs Special Agent and Trade Security Consultant

A retired Special Agent of the US Customs Service who was serving on a joint FBI-Customs counter-terrorism task force on September 11, 2001, and directly observed the development and early implementation of CTPAT. Subsequently implemented Customs Mutual Assistance Agreements in Canada and worked as a private sector trade security consultant.

Expert 8—Retired US Customs Analyst, Trade Compliance Consultant, and Author

A former US Customs analyst and investigator with over a decade of government service, followed by a career as a trade compliance consultant at a major professional services firm. Guided large companies through CTPAT compliance in the US and AEO programs in Europe and Asia, with a published record on technology, trade, and geopolitics.

Expert 9—Corporate Trade Security Leader and CTPAT Charter Member

A senior trade compliance professional with decades of international trade experience, including roles as a licensed customs house broker and as a trade compliance leader at a major US retail corporation. One of the original seven charter members of CTPAT, with direct involvement in the program's foundational design discussions with US Customs leadership in the weeks following September 11, 2001.

C.2 Interview Instruments

Our interview protocols are available at `https://osf.io/jdwsa/overview?view_only=5afe09c9127e4d908ab7cf6cff7efe0c`. Raw interview transcripts are withheld to protect participant confidentiality given the small, identifiable expert pool.

Bibliography

[1] Owasp software component verification standard. URL: https://owasp.org/www-project-software-component-verification-standard/.

[2] Secure software development framework, 2021. URL: https://csrc.nist.gov/Projects/ssdf.

[3] What is software supply chain security?, 2022. URL: https://www.redhat.com/en/topics/security/what-is-software-supply-chain-security.

[4] microsoft/sbom-tool, 2024. URL: https://github.com/microsoft/sbom-tool.

[5] Safeguarding artifact integrity across any software supply chain, 2024. URL: https://slsa.dev/.

[6] Slsa: Terminology, 2024. URL: https://slsa.dev/spec/v1.0/terminology.

[7] Urgent security alert for fedora linux 40 and fedora rawhide users, 2024. URL: https://www.redhat.com/en/blog/urgent-security-alert-fedora-41-and-rawhide-users.

[8] Security findings increase 10x in fortune 50 enterprises. https://labs.cloudsecurityalliance.org/research/

csa-research-note-ai-generated-code-security-vibe-coding-202/, 2025. Cloud Security Alliance. Accessed: 2025.

[9] Chainguard Academy. What is openvex?, January 2023. URL: https://edu.chainguard.dev/open-source/sbom/what-is-openvex/.

[10] Cybersecurity & Infrastructure Security Agency. Software bill of materials (sbom), 2024. URL: https://www.cisa.gov/sbom.

[11] Adem Ait, Javier Luis Cánovas Izquierdo, and Jordi Cabot. An empirical study on the survival rate of GitHub projects. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*, pages 365–375, 2022. doi:10.1145/3524842.3527941.

[12] Adam Alami, Raúl Pardo, and Johan Linåker. Free open source communities sustainability: Does it make a difference in software quality? *Empirical Software Engineering*, 29:114, 2024. doi:10.1007/s10664-024-10529-6.

[13] Florian Angermeir, Markus Voggenreiter, Fabiola Moyón, and Daniel Mendez. Enterprise-driven open source software: A case study on security automation. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 278–287, 2021. doi:10.1109/ICSE-SEIP52600.2021.00037.

[14] Ramaswamy Chandramouli Frederick Kautz Santiago Torres Arias. Strategies for the integration of software supply chain security in devsecops ci/cd pipelines. 2024. doi:10.6028/NIST.SP.800-204D.ipd.

[15] Arushi Arora, Virginia L. Wright, and Christina Garman. Sok: A framework for and analysis of software bill of materials tools. 4 2022. URL: https://www.osti.gov/biblio/2204407, doi:10.2172/2204407.

[16] Arushi Arora, Virginia L Wright, and Christina Garman. Strengthening the

security of operational technology: Understanding contemporary bill of materials. *Journal of Critical Infrastructure Policy*, 3(1), 6 2022. URL: https://www.osti.gov/biblio/1888299, doi:10.18278/jcip.3.1.8.

[17] Muhammad Muneeb Aslam, Muhammad Farhan, Sana Yaseen, Ahmad Raza, Muhammad Javed Iqbal, and Muhammad Munwar Iqbal. A smart model for categorization of github repositories. *KIET Journal of Computing and Information Sciences*, 6(1):50–64, 2022. URL: https://kjcis.kiet.edu.pk/index.php/kjcis/article/view/149, doi:10.51153/kjcis.v6i1.149.

[18] The Update Framework authors. The update framework, 2024. URL: https://theupdateframework.io/overview/.

[19] Guilherme Avelino, Eleni Constantinou, Marco Tulio Valente, and Alexander Serebrenik. On the abandonment and survival of open source projects: An empirical investigation. In *Proceedings of the 13th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, 2019. doi:10.1109/ESEM.2019.8870181.

[20] Guilherme Avelino, Leonardo Passos, Andre Hora, and Marco Tulio Valente. A novel approach for estimating truck factors. In *Proceedings of the 24th IEEE International Conference on Program Comprehension (ICPC)*, pages 1–10, 2016. doi:10.1109/ICPC.2016.7503718.

[21] Stefano Balla, Thomas Degueule, Romain Robbes, Jean-Rémy Falleri, and Stefano Zacchiroli. Automatic Classification of Software Repositories: a Systematic Mapping Study. In *International Conference on Evaluation and Assessment in Software Engineering (EASE 2025)*, Istanbul, Turkey, June 2025. URL: https://hal.science/hal-05049757.

[22] Musard Balliu, Benoit Baudry, Sofia Bobadilla, Mathias Ekstedt, Martin Monperrus, Javier Ron, Aman Sharma, Gabriel Skoglund, César Soto-Valero,

and Martin Wittlinger. Challenges of producing software bill of materials for java. *IEEE Security & Privacy*, 21(6):12–23, November 2023. URL: http://dx.doi.org/10.1109/MSEC.2023.3302956, doi:10.1109/msec.2023.3302956.

[23] Michelle Barker, Neil P. Chue Hong, Daniel S. Katz, Anna-Lena Lamprecht, Carlos Martinez-Ortiz, Fotis Psomopoulos, Jennifer Harrow, Leyla Jael Castro, Morane Gruenpeter, Paula Andrea Martinez, and Tom Honeyman. Introducing the FAIR principles for research software. *Scientific Data*, 9:622, 2022. doi:10.1038/s41597-022-01710-x.

[24] Christoph Becker, Ruzanna Chitchyan, Leticia Duboc, Steve Easterbrook, Birgit Penzenstadler, Norbert Seyff, and Colin C. Venters. Sustainability design and software: The karlskrona manifesto. In *Proceedings of the 37th International Conference on Software Engineering*, pages 467–476. IEEE, 2015.

[25] Gareth Bennett, Tracy Hall, Emily Winter, and Steve Counsell. Semgrep*: Improving the limited performance of static application security testing (sast) tools. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, EASE '24, page 614–623, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3661167.3661262.

[26] Daniel Berrar. Performance measures for binary classification. In Shoba Ranganathan, Michael Gribskov, Kenta Nakai, and Christian Schönbach, editors, *Encyclopedia of Bioinformatics and Computational Biology*, pages 546–560. Academic Press, Oxford, 2019. URL: https://www.sciencedirect.com/science/article/pii/B9780128096338203518, doi:10.1016/B978-0-12-809633-8.20351-8.

[27] Samuel A. Besser, Boyana Norris, and Stephen Fickas. Research software at

the university of illinois urbana-champaign: A mixed methods survey dataset. Zenodo, 2025. doi:10.5281/zenodo.13368203.

[28] Shikhar Bharadwaj and Tushar Kadam. Github issue classification using bert-style models. In *2022 IEEE/ACM 1st International Workshop on Natural Language-Based Software Engineering (NLBSE)*, pages 40–43, 2022. doi:10.1145/3528588.3528663.

[29] Tingting Bi, Boming Xia, Zhenchang Xing, Qinghua Lu, and Liming Zhu. On the way to sboms: Investigating design issues and solutions in practice, 2023. arXiv:2304.13261.

[30] Bitergia. Bitergia analytics, 2024. URL: https://bitergia.com/bitergia-analytics/.

[31] Hudson Borges and Marco Tulio Valente. What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112–129, 2018. doi:10.1016/j.jss.2018.09.016.

[32] Amiangshu Bosu and Kazi Zakia Sultana. Diversity and inclusion in open source software (OSS) projects: Where do we stand and where do we need to go? In *Proceedings of the 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2019.

[33] Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3(2):77–101, 2006. arXiv:https://doi.org/10.1191/1478088706qp063oa, doi:10.1191/1478088706qp063oa.

[34] Christian Brezing. Just in time delivery – success stories and practical examples. https://www.lcmd.io/en/blog/just-in-time-delivery-success-stories-and-practical-examples, 2025.

[35] Barak Brudo. Spdx vs. cyclonedx: Sbom formats com-

pared, 2024. URL: https://scribesecurity.com/blog/spdx-vs-cyclonedx-sbom-formats-compared/.

[36] Drew Buttner and Bob Martin. The cybersecurity benefits of leveraging a software bill of materials. 2022.

[37] Jeffrey C. Carver, Nicholas Weber, Karthik Ram, Sandra Gesing, and Daniel S. Katz. A survey of the state of the practice for research software in the United States. *PeerJ Computer Science*, 8:e963, 2022.

[38] Peter Caven and L. Jean Camp. Towards a more secure ecosystem: Implications for cybersecurity labels and sboms. *SSRN Electronic Journal*, 2023. doi:10.2139/ssrn.4527526.

[39] Mahasweta Chakraborti, Curtis Atkisson, Ştefan Stănciulescu, Vladimir Filkov, and Seth Frey. Do we run how we say we run? Formalization and practice of governance in OSS communities. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, CHI '24. ACM, 2024. doi:10.48550/arXiv.2309.14245.

[40] CHAOSS Project. Community health analytics for open source software. https://chaoss.community, 2024. Accessed: 2024.

[41] Cloud Native Computing Foundation. CNCF project lifecycle and process. https://contribute.cncf.io/projects/lifecycle/, 2024. Accessed: 2026.

[42] CodeQL. Codeql. URL: https://codeql.github.com.

[43] Jailton Coelho and Marco Tulio Valente. Why modern open source projects fail. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 186–196, 2017. doi:10.1145/3106237.3106246.

[44] Jailton Coelho, Marco Tulio Valente, Luciano Milen, and Luciana L. Silva. Is

this GitHub project maintained? measuring the level of maintenance activity of open-source projects. *Information and Software Technology*, 122:106274, 2020. doi:10.1016/j.infsof.2020.106274.

[45] Jorge Colazo and Yulin Fang. Impact of license choice on open source software development activity. *Journal of the American Society for Information Science and Technology*, 60(5):997–1011, 2009. doi:10.1002/asi.21039.

[46] Stefano Comino, Fabio M. Manenti, and Maria Laura Parisi. From planning to mature: On the success of open source projects. *Research Policy*, 36(10):1575–1586, 2007. doi:10.1016/j.respol.2007.08.003.

[47] Valerio Cosentino, Javier L. Cánovas Izquierdo, and Jordi Cabot. A systematic mapping study of software development with github. *IEEE Access*, 5:7173–7192, 2017. doi:10.1109/ACCESS.2017.2682323.

[48] Valerio Cosentino, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Assessing the bus factor of Git repositories. In *Proceedings of the 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015. doi:10.1109/SANER.2015.7081864.

[49] CrowdStrike. What is a supply chain attack? URL: https://www.crowdstrike.com/en-us/cybersecurity-101/cyberattacks/supply-chain-attack/.

[50] P. J. Crowley and Bruce Butterworth. Keeping bombs off planes: Securing air cargo, aviation's soft underbelly. Center for American Progress, May 2007.

[51] Kevin Crowston, Hala Annabi, and James Howison. Defining open source software project success. In *Proceedings of the 24th International Conference on Information Systems (ICIS)*, pages 327–340, 2003.

[52] Kevin Crowston, James Howison, and Hala Annabi. Information systems suc-

cess in free and open source software development: Theory and measures. *Software Process: Improvement and Practice*, 11(2):123–148, 2006. doi:10.1002/spip.259.

[53] CURIOSS. CURIOSS: Community for University and Research Institution OSPOs. https://curioss.org/, 2023. Accessed: 2026-08-17.

[54] Andreas Dann, Henrik Plate, Ben Hermann, Serena Elisa Ponta, and Eric Bodden. Identifying challenges for oss vulnerability scanners - a study & test suite. *IEEE Transactions on Software Engineering*, 48(9):3613–3625, September 2022. doi:10.1109/tse.2021.3101739.

[55] David de Fitero-Dominguez, Eva Garcia-Lopez, Antonio Garcia-Cabot, and Jose-Javier Martinez-Herraiz. Enhanced automated code vulnerability repair using large language models. *Engineering Applications of Artificial Intelligence*, 138:109291, December 2024. URL: http://dx.doi.org/10.1016/j.engappai.2024.109291, doi:10.1016/j.engappai.2024.109291.

[56] Bandit Developers. Bandit, 2024. URL: https://bandit.readthedocs.io/en/latest/.

[57] Juri Di Rocco, Davide Di Ruscio, Claudio Di Sipio, Phuong Nguyen, and Riccardo Rubei. Topfilter: An approach to recommend relevant github topics. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3382494.3410690.

[58] Claudio Di Sipio, Riccardo Rubei, Davide Di Ruscio, and Phuong T. Nguyen. A multinomial naïve bayesian (mnb) network to automatically recommend topics for github repositories. In *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*, EASE '20, pages 71–

80, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3383219.3383227.

[59] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability detection with code language models: How far are we?, 2024. URL: https://arxiv.org/abs/2403.18624, arXiv:2403.18624.

[60] Xueying Du, Geng Zheng, Kaixin Wang, Jiayi Feng, Wentai Deng, Mingwei Liu, Bihuan Chen, Xin Peng, Tao Ma, and Yiling Lou. Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag, 2024. URL: https://arxiv.org/abs/2406.11147, arXiv:2406.11147.

[61] Trevor Dunlap, Seaver Thorn, William Enck, and Bradley Reaves. Finding fixed vulnerabilities with off-the-shelf static analysis. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, pages 489–505, 2023. doi:10.1109/EuroSP57164.2023.00036.

[62] Nadia Eghbal. Roads and bridges: The unseen labor behind our digital infrastructure. https://www.fordfoundation.org/work/learning/research-reports/roads-and-bridges-the-unseen-labor-behind-our-digital-infrastructure/, 2016. Ford Foundation. Accessed: 2024.

[63] Daniel Ehls. Open source project collapse – sources and patterns of failure. In *Proceedings of the 50th Hawaii International Conference on System Sciences (HICSS)*, 2017. URL: https://aisel.aisnet.org/hicss-50/os/open_source/3/.

[64] Sarah Elder, Nusrat Zahan, Rui Shu, Monica Metro, Valeri Kozarev, Tim Menzies, and Laurie Williams. Do i really need all this work to find vulnera-

bilities? *Empirical Software Engineering*, 27(6):154, 2022. doi:10.1007/s10664-022-10179-6.

[65] William Enck and Laurie Williams. Top five challenges in software supply chain security: Observations from 30 industry and government organizations. *IEEE Security & Privacy*, 20(2):96–100, 2022. doi:10.1109/MSEC.2022.3142338.

[66] European Commission Taxation and Customs Union. Mutual recognition of aeos. URL: https://taxation-customs.ec.europa.eu/customs/authorised-economic-operator/mutual-recognition_en.

[67] Sidong Feng and Chunyang Chen. Prompting is all you need: Automated android bug replay with large language models, 2024. URL: https://arxiv.org/abs/2306.01987, arXiv:2306.01987.

[68] Stephen E. Flynn and Daniel B. Prieto. Neglected defense: Mobilizing the private sector to support homeland security. Council on Foreign Relations, CSR No. 13, March 2006.

[69] Fortinet. Solarwinds supply chain attack. Fortinet Cyber Glossary. URL: https://www.fortinet.com/resources/cyberglossary/solarwinds-cyber-attack.

[70] FOSSA. The complete guide to software supply chain security. URL: https://fossa.com/learn/software-supply-chain-security/.

[71] OWASP Foundation. Owasp static code analysis, 2024. URL: https://owasp.org/www-community/controls/Static_Code_Analysis.

[72] M. Fourne, D. Wermke, S. Fahl, and Y. Acar. A viewpoint on human factors in software supply chain security: A research agenda. *IEEE Security & Privacy*, 21(06):59–63, nov 2023. doi:10.1109/MSEC.2023.3316569.

[73] Andres Freund. backdoor in upstream xz/liblzma leading to SSH server compro-

mise. https://www.openwall.com/lists/oss-security/2024/03/29/4, 2024. Openwall mailing list. Accessed: 2024.

[74] Alex Gantman. Sbom: Good intentions, bad analogies, ugly outcomes. https://www.linkedin.com/pulse/sbom-good-intentions-bad-analogies-uglyoutcomes-alex-gantman/, n.d. Accessed: 2025-10-16.

[75] George Mason University Costello College of Business. Creating market incentives to improve cybersecurity. https://business.gmu.edu/news/2025-01/creating-market-incentives-improve-cybersecurity, January 2025.

[76] GitHub. Codeql, 2021. URL: https://codeql.github.com.

[77] GitHub. Dependabot, 2024. URL: https://github.com/dependabot.

[78] GitHub. Github community guidelines. https://docs.github.com/en/site-policy/github-terms/github-community-guidelines, 2024. Accessed: 2025-06-11.

[79] GitHub. Github rest api. https://api.github.com/, 2024. Accessed: 2025-06-11.

[80] GitHub. Spotbugs, 2024. URL: https://spotbugs.github.io.

[81] GitHub. Building communities documentation. https://docs.github.com/en/communities, 2025. Accessed: 2025-10-19.

[82] GitHub Staff. Octoverse: A new developer joins GitHub every second as AI leads TypeScript to #1. https://github.blog/news-insights/octoverse/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to
October 2025. Updated February 28, 2026. Accessed: 2026-05-23.

[83] Carole Goble. Better software, better research. *IEEE Internet Computing*, 18(5):4–8, 2014. doi:10.1109/MIC.2014.88.

[84] Sean Goggins, Matt Germonprez, and Kevin Lumbard. Making open source project health transparent. *Computer*, 54(8):104–111, 2021.

[85] Sean Goggins, Kevin Lumbard, and Matt Germonprez. Open source community health: Analytical metrics and their corresponding narratives. In *Proceedings of the 4th IEEE/ACM International Workshop on Software Health in Projects, Ecosystems and Communities (SoHeal)*, pages 25–33, 2021. doi:10.1109/SoHeal52568.2021.00010.

[86] Juanita Gomez, Emily Lovell, Stephanie Lieggi, Alvaro A. Cardenas, and James Davis. Recipe for discovery: A pipeline for institutional open source activity, 2026. URL: https://arxiv.org/abs/2506.18359, arXiv:2506.18359.

[87] Simon Goring. University of wisconsin open source monitoring project. https://simongoring-ospo.curve.space/, 2023. Accessed: 2025-06-18.

[88] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355, 2014.

[89] Grith. Clinejection: When your AI tool installs another. https://grith.ai/blog/clinejection-when-your-ai-tool-installs-another, 2026. Accessed: 2026-04-15.

[90] Ibrahim Haddad. A deep dive into open source program offices: Structure, roles, responsibilities, and challenges. https://www.linuxfoundation.org, 2022. Linux Foundation.

[91] B M Raihanul Haque. An analysis of sbom in the context of software supply-chain risk management, 2023.

[92] Wilhelm Hasselbring, Leslie Carr, Simon Hettrick, Heather Packer, and Thanas-

sis Tiropanis. FAIR and open computer science research software. *arXiv preprint arXiv:1908.05986*, 2020.

[93] Wilhelm Hasselbring, Leslie Carr, Simon Hettrick, Heather Packer, and Thanassis Tiropanis. From FAIR research data toward FAIR and open research software. *it – Information Technology*, 62(1):39–47, 2020. doi:10.1515/itit-2019-0040.

[94] Vikas Hassija, Vinay Chamola, Vatsal Gupta, Sarthak Jain, and Nadra Guizani. A survey on supply chain security: Application areas, security threats, and solution architectures. *IEEE Internet of Things Journal*, 8:6222–6246, 2021. URL: https://api.semanticscholar.org/CorpusID:226767829.

[95] Runzhi He, Hao He, Yuxia Zhang, and Minghui Zhou. Automating dependency updates in practice: An exploratory study on github dependabot. *IEEE Transactions on Software Engineering*, pages 1–18, 2023. doi:10.1109/tse.2023.3278129.

[96] Simon Hettrick, Alasdair Brown, Stephen Crouch, James Graham, Patrick J. Grylls, and Stuart W. Mangham. Research software at the university of southampton. Technical report, University of Southampton, 2019.

[97] Luke Hinds. What is software provenance, and how can it keep your software secure?, January 2024. URL: https://stacklok.com/blog/what-is-software-provenance-and-why-is-it-important.

[98] House Energy and Commerce Committee. TikTok: How Congress Can Safeguard American Data Privacy and Protect Children from Online Harms. https://www.congress.gov/event/118th-congress/house-event/115519, March 2023.

[99] James Howison and James D. Herbsleb. Scientific software production: Incen-

tives and collaboration. *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 513–522, 2011.

[100] Dave Hulbert. Using tree-of-thought prompting to boost chatgpt's reasoning. https://github.com/dave1010/tree-of-thought-prompting, May 2023. URL: https://doi.org/10.5281/zenodo.10323452, doi:10.5281/ZENODO.10323452.

[101] IBM. What is the log4j vulnerability? URL: https://www.ibm.com/think/topics/log4j.

[102] IBM. What is a confusion matrix?, 2024. URL: https://www.ibm.com/topics/confusion-matrix.

[103] Nasif Imtiaz, Aniqa Khanom, and Laurie Williams. Open or sneaky? fast or slow? light or heavy?: Investigating security releases of open source packages. *IEEE Transactions on Software Engineering*, 49(4):1540–1560, April 2023. doi:10.1109/tse.2022.3181010.

[104] Sonatyoe Inc. Sonatype, 2024. URL: https://www.sonatype.com.

[105] IronNet. Log4j: New software supply chain vulnerability unfolding as this holiday's cyber nightmare. URL: https://www.ironnet.com/blog/log4j-new-software-supply-chain-vulnerability-unfolding-as-this-holidays-cybe

[106] Nafis Tanveer Islam, Joseph Khoury, Andrew Seong, Mohammad Bahrami Karkevandi, Gonzalo De La Torre Parra, Elias Bou-Harb, and Peyman Najafirad. Llm-powered code vulnerability repair with reinforcement learning and semantic reward, 2024. URL: https://arxiv.org/abs/2401.03374, arXiv:2401.03374.

[107] Maliheh Izadi, Abbas Heydarnoori, and Georgios Gousios. Topic recommenda-

tion for software repositories using multi-label classification algorithms, 2021. URL: https://arxiv.org/abs/2010.09116, arXiv:2010.09116.

[108] Mohammad Izadi, Mehran Nejati, and Abbas Heydarnoori. Semantically-enhanced topic recommendation systems for software projects. *Empirical Software Engineering*, 28(50):1–37, 2023. doi:10.1007/s10664-022-10272-w.

[109] Slinger Jansen. Measuring the health of open source software ecosystems: Beyond the scope of project health. *Information and Software Technology*, 56(11):1508–1519, 2014. doi:10.1016/j.infsof.2014.04.006.

[110] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 92–101, 2014. doi:10.1145/2597073.2597074.

[111] Daniel S. Katz and Neil P. Chue Hong. Software citation in theory and practice. In *Mathematical Software – ICMS 2018*, volume 10931 of *Lecture Notes in Computer Science*, pages 289–296. Springer, 2018. doi:10.1007/978-3-319-96418-8_34.

[112] Daniel S. Katz, Lois Curfman McInnes, David E. Bernholdt, et al. Community organizations: Changing the culture in which research software is developed and sustained. *Computing in Science and Engineering*, 21(2):8–24, 2019. doi:10.1109/MCSE.2018.2883051.

[113] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: a highly efficient gradient boosting decision tree. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 3149–3157, Red Hook, NY, USA, 2017. Curran Associates Inc.

[114] Avishree Khare, Saikat Dutta, Ziyang Li, Alaia Solko-Breslin, Rajeev Alur, and

Mayur Naik. Understanding the effectiveness of large language models in detecting security vulnerabilities, 2024. URL: https://arxiv.org/abs/2311.16169, arXiv:2311.16169.

[115] Lukas Kree, René Helmke, and Eugen Winter. Using semgrep oss to find owasp top 10 weaknesses in php applications: A case study. In Federico Maggi, Manuel Egele, Mathias Payer, and Michele Carminati, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 64–83, Cham, 2024. Springer Nature Switzerland.

[116] Kajetan Kuszczyński and Michał Walkowski. Comparative analysis of open-source tools for conducting static code analysis. *Sensors*, 23(18), 2023. URL: https://www.mdpi.com/1424-8220/23/18/7978, doi:10.3390/s23187978.

[117] M. D. Laden. The genesis of the us c-tpat program: Lessons learned and earned by the government and trade. *World Customs Journal*, 1(2):76, 2007.

[118] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. Sok: Taxonomy of attacks on open-source software supply chains. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1509–1526, 2023. doi:10.1109/SP46215.2023.10179304.

[119] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. Sok: Taxonomy of attacks on open-source software supply chains. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1509–1526, 2023. doi:10.1109/SP46215.2023.10179304.

[120] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, Olivier Barais, and Serena Elisa Ponta. Risk explorer for software supply chains: Understanding the attack surface of open-source based software development. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and*

Ecosystem Defenses, SCORED'22, page 35–36, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3560835.3564546.

[121] Seth Larson. New era of slop security reports for open source. https://sethmlarson.dev/slop-security-reports, December 2024. Accessed: 2025.

[122] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 75–86. IEEE, 2004. doi:10.1109/CGO.2004.1281665.

[123] Legit Security. Software supply chain security 101. URL: https://www.legitsecurity.com/software-supply-chain-security-101.

[124] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proc. ACM Program. Lang.*, 8(OOPSLA1), April 2024. doi:10.1145/3649828.

[125] Ziyang Li, Saikat Dutta, and Mayur Naik. Llm-assisted static analysis for detecting security vulnerabilities, 2024. URL: https://arxiv.org/abs/2405.17238, arXiv:2405.17238.

[126] Zhifang Liao, Libing Deng, Xiaoping Fan, Yan Zhang, Hui Liu, Xiaofei Qi, and Yun Zhou. Empirical research on the evaluation model and method of sustainability of the open source ecosystem. *Symmetry*, 10(12):747, 2018. doi:10.3390/sym10120747.

[127] Stephanie Lieggi et al. From containers to code: Applying post-9/11 trade security lessons to the software supply chain. link available at https://ucsc-ospo.github.io/post/2025ofapresentation/.

[128] Johan Linåker, Thomas Olsson, and Efi Papatheocharous. How to assess and

communicate the sustainability of open source software projects: A practitioner perspective. *Information and Software Technology*, 2022.

[129] Mario Linares-Vásquez, Collin McMillan, Denys Poshyvanyk, and Jairo Aponte. On using machine learning to automatically classify software applications into domain categories. *Empirical Software Engineering*, 19(3):582–618, 2014. doi:10.1007/s10664-012-9230-z.

[130] Linux Foundation Research, TODO Group, CNCF, FossID, and FinOps Foundation. The 2025 state of OSPOs and open source management. https://www.linuxfoundation.org/research/ospo-2025, 2025. Accessed: 2026-05-23.

[131] Aqua Security Software Ltd. Trivy, 2024. URL: https://trivy.dev.

[132] Wenyi Lu, Enock Kasaadah, S M Rakib Ul Karim, Matt Germonprez, and Sean Goggins. Open source software lifecycle classification: Developing wrangling techniques for complex sociotechnical systems, 2025. URL: https://arxiv.org/abs/2504.16670, arXiv:2504.16670.

[133] Yuxing Ma, Chris Bogart, Sadika Amreen, Russell Zaretzki, and Audris Mockus. World of code: An infrastructure for mining the universe of open source vcs data. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 143–154, 2019. doi:10.1109/MSR.2019.00031.

[134] Robert Alan Martin. Visibility & control: Addressing supply chain challenges to trustworthy software-enabled things. In *2020 IEEE Systems Security Symposium (SSS)*, pages 1–4, 2020. doi:10.1109/SSS47320.2020.9174365.

[135] Marcela S. Melara and Santiago Torres-Arias. A viewpoint on software supply chain security: Are we getting lost in translation? *IEEE Security & Privacy*, 21(6):55–58, November 2023. doi:10.1109/msec.2023.3316568.

[136] Ministère de l'Éducation nationale, de l'Enseignement supérieur et de la

Recherche. Catalogue des logiciels — enseignement supérieur & recherche. https://logiciels.catalogue-esr.fr/, 2025. Accessed: 2025-06-19.

[137] MITRE. Common weakness enumeration, February 2024. URL: https://cwe.mitre.org.

[138] MITRE. Cve numbering authorities (cnas), 2024. URL: https://www.cve.org/ResourcesSupport/Resources#CVENumberingAuthorities.

[139] MITRE. Cve program mission, 2024. URL: https://www.cve.org.

[140] Nick Monacelli. Improving maritime transportation security in response to industry consolidation. Homeland Security Affairs 14, Article 2, page 4, January 2018.

[141] Marina Moore, Trishank Karthik Kuppusamy, and Justin Cappos. Artemis: Defanging software supply chain attacks in multi-repository update systems. In *Proceedings of the 39th Annual Computer Security Applications Conference*, ACSAC '23, page 83–97, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3627106.3627129.

[142] Marina Moore, Aditya Sirish A Yelgundhalli, Trishank Karthik Kuppusamy, Santiago Torres-Arias, Lois Anne DeLong, and Justin Cappos. Scudo: A proposal for resolving software supply chain insecurities in vehicles. 2022.

[143] Erika Mourão, Daniela Trevisan, and José Viterbo. Understanding the success factors of research software. In *Information Technology and Systems*. Springer, 2023.

[144] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22(6):3219–3253, 2017. doi:10.1007/s10664-017-9512-6.

[145] Randall Munroe. Dependency. https://xkcd.com/2347/, 2020. xkcd webcomic. Accessed: 2024.

[146] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwen Ye. Evolution patterns of open-source software systems and communities. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE)*, pages 76–85, 2002. doi:10.1145/512035.512055.

[147] National Cyber Security Centre. Log4j vulnerability: What everyone needs to know. URL: https://www.ncsc.gov.uk/information/log4j-vulnerability-what-everyone-needs-to-know.

[148] National Institute of Standards and Technology. Secure software development framework (ssdf). https://csrc.nist.gov/projects/ssdf, 2025. Last updated: February 27, 2025; Accessed: 2025-10-16.

[149] National Telecommunications and Information Administration. Survey of existing sbom formats and standards. Technical report, NTIA, 2019. URL: https://www.ntia.gov/files/ntia/publications/ntia_sbom_formats_and_standards_whitepaper_-_version_20191025.pdf.

[150] Giulia Neaher. Securing the open frontier: Voluntary standards for open-source security, October 2025. URL: https://www.stimson.org/2025/securing-the-open-frontier/.

[151] Z. Newman et al. Sigstore: Software signing for everybody. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022. URL: https://dl.acm.org/doi/10.1145/3548606.3560596.

[152] Zachary Newman, John Speed Meyers, and Santiago Torres-Arias. Sigstore: Software signing for everybody. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 2353–2367,

New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3548606.3560596.

[153] NIST. National vulnerability database, February 2024. URL: https://nvd.nist.gov.

[154] Sabato Nocera, Simone Romano, Massimiliano Di Penta, Rita Francese, and Giuseppe Scanniello. Software bill of materials adoption: A mining study from github. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 39–49, 2023. doi:10.1109/ICSME58846.2023.00016.

[155] Christopher Nolan. The rubik's cube of cargo screening: Is 100% screening of all u.s.-bound cargo containers prudent? *American Bar Association Brief*, 42(3), 2013. Spring 2013.

[156] Yu Nong, Mohammed Aldeen, Long Cheng, Hongxin Hu, Feng Chen, and Haipeng Cai. Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities, 2024. URL: https://arxiv.org/abs/2402.17230, arXiv:2402.17230.

[157] University of Wisconsin-Madison Open Source Program Office. 2024 open source survey results, 2024. Accessed: 2025-10-19. URL: https://uw-madison-dsi.github.io/open_source_survey_results/.

[158] Marc Ohm and Charlene Stuke. Sok: Practical detection of software supply chain attacks. In *Proceedings of the 18th International Conference on Availability, Reliability and Security*, ARES '23, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3600160.3600162.

[159] M. Ojah. Securing and facilitating trade through u.s. land borders: Critical analysis of c-tpat and fast programs. *Transportation Research Record*, 1938(1):32, 2005. doi:10.1177/0361198105193800105.

[160] Chinenye Okafor, Taylor R. Schorlemmer, Santiago Torres-Arias, and James C. Davis. Sok: Analysis of software supply chain security by establishing secure design properties. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, SCORED'22, page 15–24, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3560835.3564556.

[161] Pedro Oliveira, Tayana Conte, Marco Gerosa, and Igor Steinmacher. Governance in practice: How open source projects define and document roles. In *Proceedings of the 19th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE '26)*, 2026. doi:10.1145/3794860.3794911.

[162] Erin Omier, Chris Aniszczyk, and Ana Jiménez Santamaría. The business value of the OSPO. https://www.linuxfoundation.org, 2022. Linux Foundation.

[163] Open Source Initiative. The open source definition. https://opensource.org/osd, 2024. Accessed: 2024.

[164] Open Source Security Foundation. OpenSSF Scorecard. https://securityscorecards.dev/, 2024. Accessed: 2024.

[165] OpenAI. Openai platform documentation: Embeddings, 2024. Accessed: 2025-06-12. URL: https://platform.openai.com/docs/guides/embeddings.

[166] OpenSource@Stanford. Opensource@stanford projects registry. https://opensource.stanford.edu/projects-registry, 2025. Accessed: 2025-06-19.

[167] OpenSSF. Scorecard, 2024. URL: https://github.com/ossf/scorecard.

[168] OpenSSF. Openssf scorecard. https://scorecard.dev/, 2025. Assess open source projects for security risks via automated checks.

[169] OSS Compass Community. OSS Compass: Open source ecosystem evaluation system. https://compass.gitee.com/. Accessed: 2026-08-17.

[170] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models, 2022. URL: https://arxiv.org/abs/2112.02125, arXiv:2112.02125.

[171] Fernando Pérez and Brian E. Granger. IPython: A system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, 2007. doi:10.1109/MCSE.2007.53.

[172] Patrick Pickerill, Henrik J. Jungen, Miroslaw Ochodek, and Meiyappan Nagappan. PHANTOM: Curating GitHub for engineered software projects using time-series clustering. *Empirical Software Engineering*, 25(4):2897–2929, 2020. doi:10.1007/s10664-020-09825-8.

[173] Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *Proceedings of the 16th International Conference on Mining Software Repositories*, MSR '19, page 383–387, Montreal, Quebec, Canada, 2019. IEEE Press. doi:10.1109/MSR.2019.00064.

[174] Chandra Ramaswami. The EU Cyber Resilience Act could put Open Source at risk. https://www.linuxfoundation.org/blog/the-eu-cyber-resilience-act-could-put-open-source-at-risk, 2023. Linux Foundation Blog. Accessed: 2024.

[175] Rapid7. Ai-driven vulnerability discovery and supply chain risk. https://www.rapid7.com/blog/post/ai-changing-vulnerability-discovery-software-supply-chain-strategy/, 2026. Accessed: 2026-05-03.

[176] L. Ravid. The xz utils backdoor: Everything you need to know. Wired. URL: https://www.wired.com/story/xz-backdoor-everything-you-need-to-know/.

[177] Red Hat. Urgent security alert for fedora 40 and fedora rawhide users. Red Hat Blog, 2024. URL: https://www.redhat.com/en/blog/urgent-security-alert-fedora-40-and-rawhide-users.

[178] Mário Rosado de Souza, Robert Haines, Markel Vigo, and Caroline Jay. What makes research software sustainable? an interview study with research software engineers. In *Proceedings of the 2019 IEEE/ACM 12th International Workshop on Cooperative and Human Aspects of Software Engineering*. IEEE, 2019.

[179] Dylan Ruediger, Can Baytas, Robin MacDougall, and Clare McCracken. University open source program offices. https://sr.ithaka.org/, 2025. Ithaka S+R.

[180] Dylan Ruediger, Claire Baytas, Rosie MacDougall, and Claire McCracken. University open source program offices. Technical report, Ithaka S+R, 2025. Accessed: 2026-05-23. URL: https://sr.ithaka.org/publications/university-open-source-program-offices/, doi:10.18665/sr.323392.

[181] Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, HyoJung Han, Sevien Schulhoff, et al. The prompt report: A systematic survey of prompting techniques. *arXiv preprint arXiv:2406.06608*, 2024.

[182] Seth D. Schwartz, Boyana Norris, and Stephen Fickas. An empirical survey of GitHub repositories at U.S. research universities. https://doi.org/10.5281/zenodo.13368203, 2024.

[183] Charles M. Schweik and Robert C. English. *Internet Success: A Study of Open-Source Software Commons*. MIT Press, Cambridge, MA, 2012.

[184] Vandana Verma Sehgal and P. S. Ambili. A taxonomy and survey of software bill of materials (sbom) generation approaches. In Suparna Dhar, Sanjay Goswami, U. Dinesh Kumar, Indranil Bose, Rameshwar Dubey, and Chandan Mazumdar, editors, *AGC 2023*, pages 40–51, Cham, 2024. Springer Nature Switzerland.

[185] Inc Semgrep. Semgrep, 2024. URL: `https://semgrep.dev/`.

[186] Scott Shambaugh. An AI agent published a hit piece on me. `https://theshamblog.com/an-ai-agent-published-a-hit-piece-on-me/`, 2026.

[187] Abhishek Sharma, Ferdian Thung, Pavneet Singh Kochhar, Agus Sulistya, and David Lo. Cataloging github repositories. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, pages 314–319, 2017. `doi:10.1145/3084226.3084287`.

[188] SLSA. Supply-chain levels for software artifacts (slsa). https://slsa.dev/, 2025. Accessed: 2025-10-16.

[189] Arfon M. Smith, Daniel S. Katz, Kyle E. Niemeyer, and FORCE11 Software Citation Working Group. Software citation principles. *PeerJ Computer Science*, 2:e86, 2016. `doi:10.7717/peerj-cs.86`.

[190] Snyk. Snyk, 2024. URL: `https://snyk.io`.

[191] Snyk. Snyk vulnerability database, 2024. URL: `https://security.snyk.io`.

[192] Marcus Soll and Malte Vosgerau. Classifyhub: An algorithm to classify github repositories. In Gabriele Kern-Isberner, Johannes Fürnkranz, and Matthias Thimm, editors, *KI 2017: Advances in Artificial Intelligence*, pages 373–379, Cham, 2017. Springer International Publishing.

[193] T. Stalnaker, N. Wintersgill, O. Chaparro, M. Di Penta, D. German, and D. Poshyvanyk. Boms away! inside the minds of stakeholders: A comprehensive study of bills of materials for software systems. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 506–518, Los Alamitos, CA, USA, apr 2024. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/.

[194] Standards and Formats Working Group. Survey of existing sbom formats and standards. October 2019.

[195] Benjamin Steenhoek, Md Mahbubur Rahman, Monoshi Kumar Roy, Mirza Sanjida Alam, Earl T. Barr, and Wei Le. A comprehensive study of the capabilities of large language models for vulnerability detection, 2024. URL: https://arxiv.org/abs/2403.17218, arXiv:2403.17218.

[196] Michael Stonebraker and Lawrence A. Rowe. The design of POSTGRES. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 340–355. ACM, 1986. doi:10.1145/16894.16888.

[197] S. Sullivan and D. Garza. C-tpat and supply chain effectiveness. In *Proceedings of the Scientia Moralitas Conference, Research Association for Interdisciplinary Studies*, 2021.

[198] SustainOSS. Universities: Academic ecosystem map. https://sustainoss.org/academic-map/universities/index.html. Accessed: 2024.

[199] Liran Tal. Scoring security vulnerabilities 101: Introducing cvss for cves, May 2019. URL: https://snyk.io/blog/scoring-security-vulnerabilities-101-introducing-cvss-for-cve/.

[200] The George Washington University Open Source Program Office. Project registry, 2025. Accessed: 2025-10-19. URL: https://ospo.gwu.edu/project-registry.

[201] S. Torres-Arias et al. in-toto: Providing farm-to-table guarantees for bits and bytes. In *Proceedings of the 28th USENIX Security Symposium*, 2019. URL: https://www.usenix.org/system/files/sec19-torres-arias.pdf.

[202] S. Torres-Arias, D. Geer, and J. Meyers. A viewpoint on knowing software: Bill of materials quality when you see it. *IEEE Security & Privacy*, 21(06):50–54, nov 2023. doi:10.1109/MSEC.2023.3315887.

[203] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. in-toto: Providing farm-to-table guarantees for bits and bytes. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1393–1410, Santa Clara, CA, August 2019. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias.

[204] Parastou Tourani, Bram Adams, and Alexander Serebrenik. Code of conduct in open source projects. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 24–33, 2017. doi:10.1109/SANER.2017.7884606.

[205] UC OSPO Network. University of california open source programs office (uc ospo) network. https://ucospo.net/, 2025. Accessed: 2025-06-19.

[206] UCSC Open Source Program Office. Ucsc ospo participation in the open forum academy symposium. URL: https://ucsc-ospo.github.io/post/2025ofapresentation/.

[207] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks, 2024. URL: https://arxiv.org/abs/2312.12575, arXiv:2312.12575.

[208] United Nations Conference on Trade and Development. Container se-

curity: Major initiatives and related international developments. Technical report, UNCTAD, 2004. URL: https://unctad.org/system/files/official-document/sdtetlb20041_en.pdf.

[209] University of Texas OSPO. Institutional Innovation Grapher. https://github.com/UT-OSPO/institutional-innovation-grapher, 2023. Accessed: 2025-06-18.

[210] U.S. Customs and Border Protection. Mutual recognition, c-tpat program. https://www.cbp.gov/border-security/ports-entry/cargo-security/c-tpat-customs-trade-partnership-against-terrorism/mutual-recognition.

[211] U.S. Customs and Border Protection. Accountability and transparency features in the united states customs trade partnership against terrorism (ctpat) programme. WCO News 104, Issue 2 / 2024, June 25 2024. https://mag.wcoomd.org/magazine/wco-news-104-issue-2-2024/accountability-and-transparency-features-ctpat/.

[212] U.S. Customs and Border Protection. Customs trade partnership against terrorism (ctpat), 2025. Accessed October 5, 2025. URL: https://www.cbp.gov/border-security/ports-entry/cargo-security/CTPAT.

[213] U.S. Customs and Border Protection. Securing and facilitating trade in north america, 2025. URL: https://www.cbp.gov/border-security/ports-entry/cargo-security/c-tpat-customs-trade-partnership-against-terrorism/mutual-recognition/aeo-programs.

[214] U.S. Department of Homeland Security. Dhs/cbp/pia-013 customs-trade partnership against terrorism (c-tpat). Technical report, CBP, August 2021. URL: https://www.dhs.gov/sites/default/files/publications/privacy-pia-cbp013-ctpat-august2021.pdf.

[215] U.S. Government Accountability Office. Container security: Expansion of key customs programs will require greater attention to critical success factors. Technical Report GAO-03-770, GAO, July 2003.

[216] U.S. Government Accountability Office. Partnership program grants importers reduced scrutiny with limited assurance of improved security. Technical Report GAO-05-404, GAO, 2005.

[217] U.S. Government Accountability Office. Maritime security: One year later: A progress report on the safe port act. Technical Report GAO-08-171T, GAO, October 2007. URL: https://www.gao.gov/assets/a118059.html.

[218] U.S. Government Accountability Office. Supply chain security: U.s. customs and border protection has enhanced its partnership with import trade sectors, but challenges remain in verifying security practices. Technical Report GAO-08-240, GAO, 2008.

[219] US Government Accountability Office. Supply chain security: Actions needed to improve cbp management of the customs trade partnership against terrorism program. Technical report, GAO, 2026. URL: https://files.gao.gov/reports/GAO-26-107893/index.html.

[220] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 644–655, 2018. doi:10.1145/3236024.3236062.

[221] David Waltermire, Brant Cheikes, Larry Feldman, and Gregory Witte. Guidelines for the creation of interoperable software identification (swid) tags, 2016. doi:10.6028/NIST.IR.8060.

[222] Tao Wang, Gang Yin, Xiang Li, and Huaimin Wang. Labeled topic detection of open source software from mining mass textual project profiles. In *Proceedings of the First International Workshop on Software Mining*, SoftwareMining '12, pages 17–24, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2384416.2384419.

[223] Supatsara Wattanakriengkrai, Bodin Chinthanet, Hideaki Hata, Raula Gaikovina Kula, Christoph Treude, Jin Guo, and Kenichi Matsumoto. GitHub repositories with links to academic papers: Public access, traceability, and evolution. *Journal of Systems and Software*, 183:111117, 2022. doi:10.1016/j.jss.2021.111117.

[224] Dominik Wermke, Jan H. Klemmer, Noah Wöhler, Juliane Schmüser, Harshini Sri Ramulu, Yasemin Acar, and Sascha Fahl. "always contribute back": A qualitative study on security challenges of the open source supply chain. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1545–1560, 2023. doi:10.1109/SP46215.2023.10179378.

[225] Dominik Wermke, Noah Wöhler, Jan H. Klemmer, Michael Friedman, Yasemin Acar, and Sascha Fahl. A qualitative study of dependency management and its security implications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22. ACM, 2022. doi:10.1145/3372297.3417232.

[226] Ratnadira Widyasari, Zhipeng Zhao, Thanh Le Cong, Hong Jin Kang, and David Lo. Topic recommendation for github repositories: How far can extreme multi-label learning go? In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 167–178, 2023. doi:10.1109/SANER56733.2023.00025.

[227] Laurie Williams. Trusting trust: Humans in the software supply chain loop.

IEEE Security & Privacy, 20(5):7–10, September 2022. doi:10.1109/msec.2022.3173123.

[228] World Customs Organization. Aeo — home. URL: https://aeo.wcoomd.org/.

[229] Fangzhou Wu, Qingzhao Zhang, Ati Priya Bajaj, Tiffany Bao, Ning Zhang, Ruoyu "Fish" Wang, and Chaowei Xiao. Exploring the limits of chatgpt in software security applications, 2023. URL: https://arxiv.org/abs/2312.05275, arXiv:2312.05275.

[230] Boming Xia, Tingting Bi, Zhenchang Xing, Qinghua Lu, and Liming Zhu. An empirical study on software bill of materials: Where we stand and the road ahead. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2630–2642, 2023. URL: https://api.semanticscholar.org/CorpusID:255825791.

[231] Dapeng Yan, Yuqing Niu, Kui Liu, Zhe Liu, Zhiming Liu, and Tegawendé F. Bissyandé. Estimating the attack surface from residual vulnerabilities in open source software supply chain. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 493–502, 2021. doi:10.1109/QRS54544.2021.00060.

[232] Josh Young, Lorena A. Barba, Sayeed Choudhury, Carly Flanagan, David Lippert, and Richard Littauer. A definition of an academic OSPO. https://doi.org/10.5281/zenodo.13910683, 2024. Zenodo.

[233] Zhiqiang Yuan, Junwei Liu, Qiancheng Zi, Mingwei Liu, Xin Peng, and Yiling Lou. Evaluating instruction-tuned large language models on code comprehension and generation, 2023. URL: https://arxiv.org/abs/2308.01240, arXiv:2308.01240.

[234] Nusrat Zahan, Parth Kanakiya, Brian Hambleton, Shohanuzzaman Shohan, and

Laurie Williams. Openssf scorecard: On the path toward ecosystem-wide automated security metrics. *IEEE Security & Privacy*, 21(6):76–88, November 2023. doi:10.1109/msec.2023.3279773.

[235] Nusrat Zahan, Elizabeth Lin, Mahzabin Tamanna, William Enck, and Laurie Williams. Software bills of materials are required. are we there yet? *IEEE Security & Privacy*, 21(2):82–88, March 2023. doi:10.1109/msec.2023.3237100.

[236] Nusrat Zahan, Shohanuzzaman Shohan, Dan Harris, and Laurie Williams. Do software security practices yield fewer vulnerabilities? In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, May 2023. doi:10.1109/icse-seip58684.2023.00032.

[237] Nusrat Zahan, Thomas Thomas, Pavithra Kannan, Emad Hassan, Laurie Williams, and William Enck. What are weak links in the npm supply chain? In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. ACM, 2022.

[238] Yu Zhang, Frank F. Xu, Sha Li, Yu Meng, Xuan Wang, Qi Li, and Jiawei Han. Higitclass: Keyword-driven hierarchical classification of github repositories, 2021. URL: https://arxiv.org/abs/1910.07115, arXiv:1910.07115.

[239] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement, 2024. URL: https://arxiv.org/abs/2404.05427, arXiv:2404.05427.

[240] Zscaler. What is the solarwinds cyberattack? URL: https://www.zscaler.com/resources/security-terms-glossary/what-is-the-solarwinds-cyberattack.