Automated Dependency Management

Project Proposal
To
Prof. Kenneth Kung

Submitted by
Sanjyot Satvi

Date

12/18/2024

CPSC 589
California State University Fullerton

Department of Computer Science

CPSC 589

To the graduate student:
1. Complete a project proposal, following the department guidelines.
2. Have this form signed by your advisor and reviewer / committee.
3. Submit it with the proposal attached to the Department of Computer Science.

☒ Project
☐ Thesis

Please print or type:

Student Name: Sanjyot Bharat Satvi Student ID: 6573195570

Address:= _____
Street City Fullerton Zip Code 92831

Home Phone: ____ Work Phone: 6573195570

E-Mail: satvi.sanjyot@csu.fullerton.edu Units: 3 Semester: Fall 2024

Are you a Classified graduate student?
☒ Yes
☐ No

Is this a group project?
☐ Yes
☒ No

Proposal Date: 12-18-2024

Tentative Date for Demonstration /Presentation/Oral Defense: _____

Completion Deadline: _____

Tentative Title: Automated Dependency Management

We recommend that this proposal be approved:

Faculty Advisor Kenneth Kung  12/19/2024
Printed name Signature Date

Faculty Reviewer _____
Printed name Signature Date

Faculty Reviewer _____
Printed name Signature Date

Table of Contents

1. ABSTRACT

In modern software development, third-party dependencies have become integral to accelerating growth and enhancing functionality. However, the complexity of managing these dependencies-ranging from version conflicts to security vulnerabilities-poses significant challenges for developers. Traditional dependency management approaches often fail to address these issues comprehensively, leading to outdated libraries, unresolved vulnerabilities, and technical debt.

This project proposes an Automated Dependency Management System designed to streamline and secure the process of dependency updates. The system integrates Software Composition Analysis (SCA) tools, such as Black Duck, with CI/CD pipelines to automate updates, analyze risks, and provide developers with actionable insights. By focusing on transparency, configurability, and trust mechanisms, the proposed solution aims to mitigate common issues such as notification fatigue, transitive dependency vulnerabilities, and update-induced regressions.

Key features of the system include:

- **Automated risk assessment for dependency updates using static and dynamic analysis tools.**
- **Enhanced support for managing transitive dependencies in large-scale projects.**
- **Continuous monitoring of dependency health and real-time integration with CI/CD workflows.**

Through this project, we aim to improve security, reduce technical lag, and foster developer confidence in automated tools. The deliverables include a fully functional prototype, evaluation metrics for update success rates, and comprehensive documentation for adoption in diverse software ecosystems.

2. INTRODUCTION

2.1 The Rise of Software Dependencies

In Modern software development relies heavily on third-party libraries and frameworks, allowing developers to reuse code efficiently and enhance productivity. Package ecosystems like npm for JavaScript, Maven for Java, and PyPI for Python have significantly streamlined this process, providing millions of reusable packages. This dependency-driven model accelerates innovation but introduces significant challenges in maintenance and security.

A typical software project may include hundreds of direct and transitive dependencies, creating intricate dependency trees. Transitive dependencies—dependencies of dependencies—add a layer of complexity, as developers often have limited visibility and control over them. This interconnectedness makes projects vulnerable to cascading failures when updates or vulnerabilities arise.

2.2 Challenges in Dependency Management

Security Vulnerabilities:

Open-source software ecosystems are inherently exposed to security risks due to their public nature and widespread adoption. Vulnerabilities in dependencies can propagate through entire ecosystems, affecting both direct and indirect dependents. For example, incidents like the **event-stream malware** in npm demonstrated how a single malicious dependency could compromise millions of projects. This risk is exacerbated by the lack of visibility into transitive dependencies, where vulnerabilities often go undetected.

Version Conflicts and Breaking Changes:

Dependencies are frequently updated to introduce new features or fix security vulnerabilities. However, updates can cause version conflicts and breaking changes that disrupt dependent projects. Developers often hesitate to apply updates, fearing that changes may break existing functionality. This delay in updating increases the project's exposure to vulnerabilities and technical debt.

Transitive Dependencies:

Managing transitive dependencies is a significant challenge, as these indirect

dependencies often introduce risks without the developer's knowledge. Current tools provide limited visibility into these dependencies, leaving projects vulnerable to unseen threats. The cascading effects of transitive dependency issues can compromise security, functionality, and system stability.

Developer Trust and Notification Fatigue:

Dependency management tools like Dependabot and Renovate aim to automate updates, but they often overwhelm developers with excessive notifications. This **alert fatigue** leads developers to ignore or disable these tools, reducing their effectiveness. Furthermore, insufficient transparency about the risks and benefits of updates creates distrust, making developers reluctant to adopt automated solutions.

2.3 Existing Solutions and Limitations:

Software Composition Analysis (SCA) Tools

SCA tools, such as Snyk and Black Duck, are instrumental in identifying vulnerabilities within software dependencies. These tools conduct comprehensive scans of dependency trees, highlighting known issues and providing recommendations for remediation. By integrating with development workflows, SCA tools aim to enhance security and automate vulnerability management.

However, significant challenges remain:

- **Manual Intervention**: Despite their automation capabilities, developers often need to manually implement updates recommended by these tools. This process can be time-consuming, particularly in large projects with extensive dependency trees, and introduces potential delays in resolving critical vulnerabilities.

- **Workflow Integration**: SCA tools frequently face issues in seamlessly integrating with CI/CD pipelines and development workflows. This limits their ability to function as a fully automated solution, requiring developers to invest additional effort to align the tools with their processes.

- **Update Prioritization**: SCA tools often fail to prioritize updates based on the

specific context and requirements of a project. This results in inefficiencies, as developers may focus on less critical updates while more significant risks remain unaddressed.

Addressing these limitations is essential for SCA tools to become more effective and trusted components of modern software development pipelines.

.

Dependency Management Bots

Automated tools like Dependabot and Renovate streamline updates by generating pull requests for dependency changes. While effective in reducing technical lag, these bots face challenges such as:

- Inadequate handling of transitive dependencies.
- Dependabot's security repository lacks the depth and functionality of full-fledged Software Composition Analysis (SCA) tools, as it provides more limited insights and features for managing vulnerabilities in dependencies.

Semantic Versioning and Update Strategies

Semantic versioning (SemVer) provides a structured approach to dependency updates, categorizing them into major, minor, and patch updates. Despite its benefits, SemVer is not always consistently followed, leading to unexpected breaking changes. Developers often resort to restrictive update strategies, such as version locking, to avoid disruptions, further delaying necessary updates.

.

2.4 The Need for an Advanced Solution:

The complexity and limitations of existing tools highlight the need for a comprehensive solution that:

- Automates the management of direct and transitive dependencies.
- Integrates with CI/CD pipelines for streamlined testing and deployment.
- Employs advanced risk assessment techniques to minimize disruptions.
- Enhances transparency and developer trust through actionable insights.

The proposed **Automated Dependency Management System** aims to address these gaps by integrating cutting-edge dependency analysis tools, incorporating SCA capabilities, and prioritizing developer-centric features. This system seeks to redefine dependency management practices, ensuring security, stability, and efficiency in modern software ecosystems.

3. Problem Domain and Literature Review:

3.1 Problem Domain

Dependency management has become a cornerstone of modern software development, enabling faster, more efficient, and higher-quality application development. However, as the reliance on third-party libraries grows, the associated risks and challenges become increasingly evident. The key issues in dependency management include:

Security Risks

Dependencies often introduce security vulnerabilities that can compromise an entire application. Transitive dependencies pose significant risks, as their vulnerabilities are harder to detect and resolve. A study on npm and RubyGems ecosystems found that vulnerabilities in these ecosystems propagate through interconnected dependency trees, exposing millions of projects to risks. High-profile incidents, such as the event-stream malware attack, have demonstrated the devastating impact of compromised dependencies.

Version Conflicts

Frequent updates to libraries and frameworks, while necessary for adding features or fixing bugs, can lead to version conflicts. These conflicts arise when dependencies update their APIs or internal structures, causing breakages in dependent projects. Semantic versioning (SemVer) aims to mitigate these issues by categorizing updates into major, minor, and patch levels, but inconsistent adoption of SemVer often undermines its effectiveness.

Technical Debt and Update Hesitancy

Developers frequently delay updating dependencies due to fears of breaking changes or lack of awareness about available updates. This hesitancy accumulates technical debt,

leaving projects vulnerable to outdated libraries and unresolved security issues. Studies reveal that a significant percentage of projects across ecosystems like npm and Maven retain outdated dependencies, prioritizing short-term stability over long-term security.

Transitive Dependencies

Indirect dependencies, or transitive dependencies, are a persistent challenge. Developers often have little visibility or control over these dependencies, making it difficult to assess their security or compatibility. Tools like Dependabot focus primarily on direct dependencies, leaving transitive dependencies under-addressed, further increasing the risks to projects.

Alert Fatigue

Automated tools for dependency updates, such as Dependabot and Renovate, generate numerous alerts and pull requests. While these tools aim to simplify the update process, they frequently overwhelm developers with excessive notifications. This alert fatigue discourages developers from adopting updates, negating the benefits of automation.

3.2 Literature Review

Security Vulnerabilities and Mitigation Strategies

Research has highlighted the growing prevalence of vulnerabilities in software ecosystems. A study on npm and RubyGems found that vulnerabilities often remain undisclosed for extended periods, exposing projects to risks. Tools like Snyk and Black Duck have emerged to identify and address these vulnerabilities by scanning dependency trees and integrating remediation strategies. However, these tools often lack mechanisms for effectively managing transitive dependencies and prioritizing updates.

Dependency Smells and Maintenance Issues

Dependency smells, such as bloated dependencies, missing dependencies, and restrictive version constraints, have been identified as significant contributors to project inefficiencies. Empirical studies on JavaScript and Python ecosystems have cataloged these smells, highlighting their impact on project health and maintenance. Dependency smells often arise due to poor configuration management or asynchronous updates,

leading to long-term technical debt.

Automated Dependency Management Tools

Tools like Dependabot and Renovate have gained popularity for automating dependency updates. An exploratory study on Dependabot revealed that while it effectively reduces technical lag, developers often mistrust its updates due to limited transparency and reliance on insufficient test coverage. Dependabot's inability to handle transitive dependencies comprehensively further limits its effectiveness.

Semantic Versioning and Update Strategies

Semantic versioning provides a structured approach to managing dependency updates, enabling developers to assess the potential impact of changes. However, inconsistent adherence to SemVer standards and the complexity of dependency trees often hinder its utility. Studies on update strategies in npm ecosystems show that developers frequently adopt restrictive policies, locking versions to avoid breaking changes but inadvertently increasing technical debt.

Software Composition Analysis (SCA) Tools

SCA tools like Snyk and Black Duck focus on identifying and addressing vulnerabilities in dependencies. These tools integrate with CI/CD pipelines to automate vulnerability scans and provide actionable insights. However, their effectiveness depends on comprehensive test coverage and developer adoption. Alert fatigue and limited support for transitive dependencies remain significant barriers to their widespread use.

3.3 Identified Gaps in Current Solutions

Despite the availability of multiple tools and strategies, critical gaps remain in current dependency management approaches:

Lack of a Unified Tool: Existing tools address specific aspects of dependency management but fail to offer a comprehensive solution. There is a critical need for a single, unified platform that integrates automated updates, security scanning, and

transitive dependency management while prioritizing developer-centric features.

Comprehensive Transitive: Dependency Management: Current tools lack robust mechanisms for managing transitive dependencies, which are often the source of hidden vulnerabilities.

Integration with Developer Workflows: Seamless integration into CI/CD pipelines is essential for automating updates and ensuring compatibility, but many tools provide only partial support.

Developer Trust and Usability: Limited transparency and excessive notifications reduce developer trust and hinder adoption.

Risk Assessment and Prioritization: Tools often fail to prioritize updates based on project-specific contexts, leading to inefficient update processes.

The proposed Automated Dependency Management System aims to fill these gaps by providing a unified, developer-friendly platform that automates updates, ensures security, and integrates seamlessly with development workflows. This comprehensive approach is designed to redefine dependency management practices and address the challenges faced by modern software ecosystems.

4. Proposed Solution:

4.1 Overview

To address the challenges and gaps identified in the previous sections, we propose an Automated Dependency Management System (ADMS) that combines automated updates, security analysis, and seamless CI/CD integration into a unified, developer-friendly platform. This solution focuses on reducing security risks, technical debt, and developer intervention while enhancing the transparency and reliability of dependency updates.

The system leverages state-of-the-art techniques such as static and dynamic analysis, Software Composition Analysis (SCA) tools, and advanced dependency management algorithms to achieve the following objectives:

- **Automation:** Automatically update dependencies, including transitive ones, with minimal developer intervention.
- **Risk Assessment:** Evaluate the impact of updates on security, compatibility, and functionality using comprehensive risk metrics.
- **Workflow Integration:** Integrate seamlessly with CI/CD pipelines to streamline testing, deployment, and remediation.
- **Transparency and Usability:** Provide developers with clear, actionable insights to foster trust and encourage adoption.

4.2 Key Features

Comprehensive Dependency Management

The system offers end-to-end dependency management, covering both direct and transitive dependencies. It uses advanced dependency tree analysis to identify hidden vulnerabilities and conflicts, ensuring that updates are applied effectively and securely.

Integrated Security Analysis

The platform integrates with leading SCA tools such as Snyk and Black Duck to perform vulnerability scans. These scans are complemented by custom algorithms that prioritize vulnerabilities based on their severity, exploitability, and relevance to the project.

CI/CD Pipeline Integration

To ensure smooth updates and rigorous testing, the system integrates directly with popular CI/CD tools like Jenkins, GitHub Actions, and Azure DevOps. This integration automates the testing of updated dependencies, reducing the risk of regressions and breaking changes.

Risk Assessment and Decision Support

The ADMS employs static and dynamic analysis to assess the impact of updates. Features such as change impact analysis and compatibility scoring provide developers with a clear understanding of the risks and benefits of each update.

Alert Optimization

To mitigate notification fatigue, the system prioritizes updates and alerts based on project-specific contexts. Developers receive fewer, more relevant notifications, ensuring that critical updates are not overlooked.

Developer-Centric Design

Transparency and usability are central to the platform's design. Detailed update reports, visual dependency trees, and interactive dashboards empower developers to make informed decisions while fostering trust in the system.

5. Objectives and Deliverables

5.1 Objectives

The primary objective of this project is to develop an Automated Dependency Management System (ADMS) that addresses the challenges of modern dependency management while promoting security, usability, and developer trust. The specific objectives are outlined below:

Automate Dependency Updates

- Enable automated updates for both direct and transitive dependencies, minimizing the need for manual intervention.
- Provide intelligent prioritization of updates to ensure critical vulnerabilities are addressed promptly.

Improve Security Posture

- Integrate with SCA tool Black Duck to identify and mitigate vulnerabilities.
- Conduct risk assessments to evaluate the impact of updates on the security and stability of the software.

Enhance Workflow Integration

- Seamlessly integrate with CI/CD pipelines to automate the testing and deployment of updated dependencies.
- Minimize disruptions to developer workflows while maintaining rigorous

compatibility checks.

Provide Transparency and Usability

- Offer an intuitive dashboard with visual dependency trees, risk scores, and update recommendations.
- Emphasize developer-centric design to encourage adoption and trust in automated tools.

Support Long-Term Sustainability

- Reduce technical debt by encouraging timely updates and providing tools for dependency maintenance.
- Promote best practices for dependency management in diverse software ecosystems.

5.2 Deliverables

The project will produce the following tangible and intangible outcomes:

Functional Prototype

- A working prototype of the Automated Dependency Management System (ADMS) that integrates dependency analysis, security scanning, and CI/CD pipeline automation.

Software Components

- **Dependency Analyzer Module:** A tool to identify outdated, vulnerable, or conflicting dependencies.
- **Update Evaluator Module:** A component to assess and prioritize updates using risk metrics.
- **CI/CD Integrator:** A plug-in for popular CI/CD tools such as Jenkins and GitHub Actions.
- **Dashboard Interface:** A user-friendly interface for developers to visualize dependencies, risks, and updates.

Comprehensive Documentation

- Technical documentation detailing the architecture, implementation, and integration of the system.
- User manuals and onboarding materials for developers and organizations adopting the platform.

6. System Design and Architecture

6.1 Overview

The Automated Dependency Management System (ADMS) is designed as a modular, extensible, and scalable platform to address the challenges of dependency management comprehensively. The system architecture integrates various components to automate dependency updates, assess risks, and provide seamless integration into existing workflows.

The system operates as a unified tool, combining features such as dependency analysis, risk evaluation, security scanning, and developer-centric dashboards. It leverages advanced algorithms and tools like SCA platforms to ensure robust and efficient management of both direct and transitive dependencies.

6.2 High-Level Architecture

The architecture of ADMS is structured into four core layers:

1. **Input Layer**
    - **Dependency Data Collection**: Extracts metadata about direct and transitive dependencies from package managers (e.g., npm, Maven, PyPI).
    - **Vulnerability Scanning**: Interfaces with SCA tools like Snyk and Black Duck to identify vulnerabilities and risks in dependencies.
2. **Processing Layer**
    - **Dependency Analyzer**: Analyzes dependency trees to detect outdated, conflicting, or vulnerable dependencies.
    - **Update Evaluator**: Evaluates the impact of updates by conducting risk assessments and compatibility tests using static and dynamic analysis.

- **Alert Manager**: Optimizes alerts and notifications by consolidating and prioritizing updates based on project-specific requirements.

3. **Integration Layer**
    - **Version Control Systems**: Supports Git-based workflows for creating pull requests and tracking updates.
4. **Presentation Layer**
    - **Developer Dashboard**: A user-friendly interface that provides Visual representations of dependency trees, Risk scores and actionable insights and Historical logs of updates and their impacts.
    - **Reporting Tools**: Generates detailed reports on vulnerabilities, updates, and system performance.

6.3 Workflow of the System

The ADMS operates in the following steps:

1. **Dependency Scanning:**
    - The system scans the dependency tree of the project, identifying direct and transitive dependencies.
    - Metadata is collected and vulnerabilities are flagged using integrated SCA tools.
2. **Risk Assessment:**
    - Each dependency is analyzed for potential risks, including security vulnerabilities, version conflicts, and transitive dependency issues.
    - Risk scores are generated based on factors such as severity, exploitability, and compatibility.
3. **Update Prioritization:**
    - Updates are categorized and prioritized based on the project's context, focusing on critical vulnerabilities and compatibility with existing systems.
4. **Developer Feedback:**
    - Developers receive detailed insights via the dashboard, including visual dependency trees, risk evaluations, and actionable recommendations.
    - Alerts and notifications are optimized to avoid fatigue and encourage timely action.

5. **Deployment:**
    - Once updates are approved, the system automatically generates pull requests or directly integrates the updates into the project.

6.4 Modular Components

The system is built on modular components that ensure scalability and ease of integration:

Dependency Analyzer

Extracts and visualizes the complete dependency tree, highlighting outdated and vulnerable dependencies.

Supports multiple ecosystems, including npm, Maven, and PyPI.

Update Evaluator

Conducts risk assessments using static and dynamic analysis techniques.

Employs change impact analysis to predict potential regressions.

CI/CD Integrator

Automates testing and deployment workflows.

Ensures compatibility with popular CI/CD tools and cloud environments.

Dashboard and Reporting

Provides an intuitive user interface for developers to monitor dependencies and updates.

Generates reports for stakeholders, summarizing system performance and update outcomes.

6.5 Functional Requirements

Automated Dependency Management

Automatically update direct and transitive dependencies with minimal manual intervention.

Integrate with CI/CD pipelines for seamless testing and deployment.

Provide risk assessments for updates, ensuring security and stability.

Risk Assessment and Analysis

Perform static and dynamic analysis for evaluating updates.

Prioritize updates based on severity, project context, and exploitability.

Vulnerability Detection and Mitigation

Use tools like Black Duck or Snyk to scan for vulnerabilities.

Provide actionable remediation strategies for security risks.

Alert Optimization

Minimize alert fatigue by prioritizing critical updates.

Consolidate notifications to focus on project-specific risks.

Developer-Centric Features

Visualize dependency trees for clarity on updates and risks.

Provide an interactive dashboard for decision-making with detailed reports.

6.6 Non-Functional Requirements

Performance

Ensure scalability to handle large projects with extensive dependency trees.

Optimize the system to minimize CI/CD pipeline delays during testing and deployment.

Reliability

Maintain system availability during dependency updates.

Guarantee consistent vulnerability detection with integrated SCA tools.

Usability

Provide an intuitive user interface with clear dashboards and reports.

Reduce manual interventions through automation.

Security

Secure the update process to prevent tampering with dependency updates.

Safeguard data exchanged with third-party tools like SCA platforms.

Maintainability

Ensure modular design for easy integration and future enhancements.

Support multiple ecosystems (npm, Maven, PyPI, etc.).

6.7 Technical Diagram

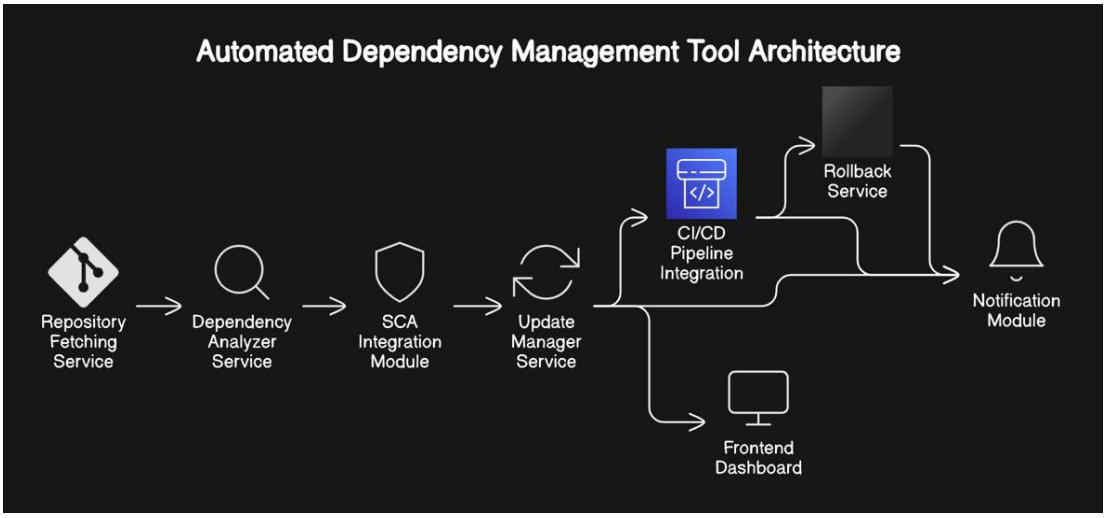

Fig – 1 Architecture Diagram

7. Methodology and Implementation

7.1 Development Methodology

To ensure a robust, scalable, and developer-friendly Automated Dependency Management System (ADMS), we will adopt an iterative and modular development approach. This methodology emphasizes flexibility, continuous improvement, and collaboration, enabling incremental delivery of features while addressing real-world challenges.

Agile Development Process

The project will follow an Agile methodology with the following principles:

Incremental Development:

- Deliver core functionalities in iterative sprints, starting with basic dependency

analysis and expanding to advanced features like risk assessment and CI/CD integration.

Continuous Feedback:

- Involve developers and stakeholders throughout the process to validate features and prioritize updates.

Adaptability:

- Allow flexibility to incorporate new tools, techniques, and feedback into the development process.

7.2 Implementation Phases

The implementation will be structured into four key phases:

Phase 1: Requirement Analysis and System Design

- Conduct a detailed analysis of user requirements and dependency management challenges.
- Design the high-level architecture, defining modular components such as the Dependency Analyzer, Update Evaluator, CI/CD Integrator, and Developer Dashboard.
- Select tools and technologies for implementation, including SCA platforms (e.g., Snyk, Black Duck) and CI/CD tools (e.g., Jenkins, GitHub Actions).

Deliverables:

- System requirements specification.
- Architectural design documents.

Phase 2: Core Component Development

- Dependency Analyzer: Implement algorithms to parse and analyze dependency trees, identifying outdated and vulnerable dependencies.
- Update Evaluator: Develop modules for risk assessment, change impact analysis, and compatibility scoring.
- Alert Manager: Design notification systems to optimize alert prioritization and relevance.

Deliverables:

- Functional modules for dependency analysis, risk evaluation, and alert management.
- Initial integration with a sample package manager (e.g., npm).

Phase 3: Integration and Automation

- CI/CD Pipeline Integration: Build connectors for Jenkins, GitHub Actions, and other popular CI/CD tools to automate testing and deployment workflows.
- SCA Tool Integration: Integrate with SCA platforms to enhance security analysis and vulnerability detection.

Deliverables:

- Fully automated CI/CD workflows for testing and deploying dependency updates.
- Reports on dependency vulnerabilities and update impacts.

Phase 4: Testing, Evaluation, and Deployment

- Conduct unit, integration, and end-to-end testing of all components to ensure reliability and scalability.
- Deploy the system in a production-like environment and evaluate its performance using real-world projects.
- Refine features based on developer feedback and usage data.

Deliverables:

- Comprehensive test reports.
- Deployment-ready system.
- Final project documentation.

7.3 Key Technologies and Tools

The implementation will leverage the following tools and technologies:

Programming Languages

- **Python**: For core algorithm development.
- **JavaScript**: For frontend and dashboard development.

- **Node.js**: For backend services and API development.

Software Composition Analysis (SCA) Tools

- **Snyk**: For vulnerability detection and remediation.
- **Black Duck**: For security analysis and license compliance.

CI/CD Tools

- **Jenkins**: For automated testing and deployment pipelines.
- **GitHub Actions**: For seamless integration with Git-based workflows.

Visualization Tools

- **D3.js**: For rendering interactive dependency trees.
- **Tableau/Matplotlib**: For generating detailed reports and visualizations.

Storage and Databases

- **MongoDB**: For storing dependency metadata, vulnerability reports, and system logs.

7.4 Challenges and Mitigation Strategies

Challenge: Handling Complex Dependency Trees

Mitigation: Employ advanced algorithms for transitive dependency analysis and change impact assessment.

Challenge: Developer Adoption

Mitigation: Focus on usability, transparency, and seamless integration with existing workflows to encourage adoption.

Challenge: Managing Large-Scale Projects

Mitigation: Optimize the system for scalability and performance, ensuring it can handle large projects with extensive dependency trees.

8. Project Schedule and Risks

The project schedule outlines the timeline for the development, testing, and deployment of the Automated Dependency Management System (ADMS). The plan is divided into multiple phases, with milestones set to track progress and ensure timely delivery of project deliverables.

8.1 Timeline

The revised project timeline spans 16 weeks (4 months), focusing on delivering the essential components of the Automated Dependency Management System (ADMS). Below is the breakdown of activities and their respective deliverables:

Week 1–2: Requirement Analysis and Architectural Design

Conduct a detailed analysis of the system requirements, including key functionalities and user needs (8 hours/week).

Design the high-level architecture of the system, outlining its core modules and workflows (7 hours/week).

Deliverables: System requirements specification and high-level architectural design.

Week 3–6: Development of Dependency Analyzer and Risk Evaluator

Build the Dependency Analyzer module to scan, analyze, and visualize dependency trees (10 hours/week).

Develop the Risk Evaluator module to assess vulnerabilities and prioritize updates based on risk scores (10 hours/week).

Deliverables: Functional Dependency Analyzer and Update Evaluator modules.

Week 7–10: Development of Dashboard and Alert Manager

Create a user-friendly Developer Dashboard that provides interactive visualizations of dependency trees and detailed risk assessments (9 hours/week)

Implement the Alert Manager to optimize notifications, ensuring relevance and reducing alert fatigue (8 hours/week).

Deliverables: Developer Dashboard and optimized notification system.

Week 11–13: Testing and Debugging of System Components

Conduct rigorous unit and integration testing to validate the functionality and reliability of the system (6 hours/week).

Debug any identified issues to ensure seamless operation across all modules (5 hours/week).

Deliverables: Comprehensive unit and integration test reports.

Week 14–16: Deployment, Documentation, and Final Evaluation

Deploy the system in a production-like environment for final validation (3 hours/week).

Prepare comprehensive documentation, including user manuals, technical guides, and system reports (4 hours/week).

Deliverables: Deployment-ready system and complete project documentation.

8.2 Risk Management in Schedule

Risk: Challenges in completing core modules within the condensed timeline.

Mitigation**:** Focus on essential functionalities and reduce the scope for less critical features to ensure timely delivery of core modules.

Risk: Delays in dashboard development or integration.

Mitigation**:** Simplify dashboard features to prioritize usability and deliver core insights effectively.

Risk: Integration with SCA tools may encounter compatibility issues or require additional development time.

Mitigation**:** Begin integration efforts early in the development cycle and limit the initial integration to a single SCA tool, such as Black Duck, to reduce complexity and ensure feasibility within the timeline.

9. Expected Results and Impact

9.1 Expected Results

The successful implementation of the Automated Dependency Management System

(ADMS) will result in the following tangible and intangible outcomes:

1. **A Comprehensive Automated Dependency Management System**

   A fully functional platform capable of automating the identification, evaluation, and update of both direct and transitive dependencies.

   Seamless integration with CI/CD pipelines and SCA tools, enabling real-time testing and deployment of updates.

2. **Improved Dependency Security**

   Significant reduction in the number of vulnerabilities within dependency trees through proactive scanning and prioritization.

   Enhanced ability to manage transitive dependencies, mitigating risks that are often overlooked by existing tools.

3. **Reduced Technical Debt**

   Timely updates of dependencies will prevent the accumulation of outdated libraries, reducing long-term maintenance challenges.

   System-generated recommendations will ensure that updates align with project-specific requirements, balancing stability and innovation.

4. **Enhanced Developer Productivity**

   Simplified workflows and minimized manual intervention will allow developers to focus on core application development.

   Reduced alert fatigue through optimized notifications, improving responsiveness to critical updates.

5. **Increased Trust in Automation Tools**

   Transparent risk assessments, detailed update reports, and an intuitive dashboard will build developer confidence in the system.

   Adoption of the system as a best practice for managing dependencies in software development.

9.2 Broader Impact

The ADMS has the potential to create significant value for both individual developers and organizations by addressing longstanding challenges in dependency management.

Industry Adoption and Standards

The system can serve as a model for unified dependency management tools, encouraging adoption across diverse ecosystems such as npm, Maven, and PyPI.

By incorporating best practices for security and automation, the project can influence industry standards and practices in software development.

Enhanced Software Supply Chain Security

With its focus on transitive dependencies and proactive vulnerability management, the ADMS will contribute to securing the software supply chain, a critical concern in modern development.

Contribution to Open-Source Ecosystems

The project's findings and tools can be shared with the open-source community, fostering collaboration and further advancements in dependency management.

By addressing dependency smells and inefficiencies, the ADMS will improve the health and sustainability of open-source projects.

Scalability for Large Enterprises

Large organizations with complex dependency trees can benefit from the system's scalability, reducing the overhead of manual dependency management across multiple projects.

Improved integration with enterprise-level CI/CD systems will enhance the overall efficiency of development workflows.

10. References

[1] A. J. Jafari, D. E. Costa, E. Shihab, and R. Abdalkareem, "Dependency Update Strategies and Package Characteristics," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, Article 149, 2023. [Online]. Available: https://doi-org.lib-proxy.fullerton.edu/10.1145/3603110.

[2] R. He, H. He, Y. Zhang, and M. Zhou, "Automating Dependency Updates in Practice: An Exploratory Study on GitHub Dependabot," *IEEE Transactions on Software Engineering*, vol. 49, no. 8, pp. 4004-4022, Aug. 2023. DOI: 10.1109/TSE.2023.3278129.

[3] A. Zerouali, T. Mens, A. Decan, et al., "On the impact of security vulnerabilities in the npm and RubyGems dependency networks," *Empirical Software Engineering*, vol. 27, no. 107, 2022. [Online]. Available: https://doi-org.lib-proxy.fullerton.edu/10.1007/s10664-022-10154-1.

[4] A. J. Jafari, D. E. Costa, R. Abdalkareem, E. Shihab, and N. Tsantalis, "Dependency Smells in JavaScript Projects," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 3790-3807, Oct. 2022. DOI: 10.1109/TSE.2021.3106247.

[5] Y. Cao, L. Chen, W. Ma, Y. Li, Y. Zhou, and L. Wang, "Towards Better Dependency Management: A First Look at Dependency Smells in Python Projects," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1741-1765, Apr. 2023. DOI: 10.1109/TSE.2022.3191353.

[6] I. Pashchenko, D.-L. Vu, and F. Massacci, "A Qualitative Study of Dependency Management and Its Security Implications," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20)*, New York, NY, USA:

ACM, 2020, pp. 1513–1531. [Online]. Available: https://doi-org.lib-proxy.fullerton.edu/10.1145/3372297.3417232.

[7] J. Hejderup and G. Gousios, "Can we trust tests to automate dependency updates? A case study of Java Projects," *Journal of Systems and Software*, vol. 183, pp. 111097, 2022. [Online]. Available: https://doi.org/10.1016/j.jss.2021.111097