

Conflict-Core Extraction for Explainable Package Version Planning with Language Models

Gabriela Costa Soares, Larissa Monteiro Silva

Department of Electrical Engineering, School of Engineering, Universidade Federal do Rio Grande do Sul, Porto Alegre, Rio Grande do Sul, Brazil

Abstract.

Modern software development relies heavily on open-source ecosystems, necessitating robust package management systems to handle complex dependency graphs. As applications scale, dependency networks often become deeply nested, leading to version conflicts that are computationally difficult to resolve and conceptually challenging for human developers to understand. Traditional package managers utilize Boolean satisfiability solvers to guarantee correct version planning; however, when constraints are unsatisfiable, these solvers typically output cryptic error messages. This lack of explainability creates a significant bottleneck in software engineering workflows. This paper proposes a novel framework for explainable package version planning by combining Conflict-Core Extraction with Large Language Models. By isolating the Minimal Unsatisfiable Subset of dependency constraints, the proposed system distills massive dependency graphs into a precise conflict core. This core is then translated into natural language prompts and processed by language models to generate highly contextual, human-readable explanations and actionable resolution strategies. Through extensive empirical analysis across multiple open-source ecosystems, we demonstrate that this methodology significantly reduces the cognitive load required to debug dependency conflicts. The results indicate a substantial improvement in diagnostic accuracy and a marked decrease in resolution time compared to traditional package management diagnostic outputs.

Keywords:

Package Management, Conflict Resolution, Large Language Models, Explainable Artificial Intelligence.

1. Introduction

The proliferation of open-source software has fundamentally transformed the landscape of software engineering, enabling developers to build complex applications rapidly by assembling pre-existing components. Package managers serve as the critical infrastructure in this paradigm, automating the process of installing, upgrading, configuring, and removing software packages. However, the convenience of component-based software engineering introduces the significant challenge of dependency management. Modern applications frequently depend on hundreds or thousands of external libraries, each of which may specify its own set of dependencies with strict

version constraints. Ensuring that all dependencies are mutually compatible requires solving a dependency resolution problem, which has been mathematically proven to be NP-hard [1].

When a package manager successfully finds a valid configuration, the underlying complexity remains hidden from the developer. The fundamental issue arises when no valid version configuration exists, a scenario commonly referred to as dependency hell [2]. In such cases, the constraints imposed by the application and its transitive dependencies form an unsatisfiable system. Contemporary package managers, equipped with advanced Boolean satisfiability solvers, can quickly determine that a conflict exists, but they struggle to explain why the conflict occurred [3, 4]. Standard error outputs often dump the entire trace of the solver's execution or provide a localized but incomplete picture of the failure. For a human developer, parsing thousands of lines of constraint evaluations to identify the root cause of a version conflict is an arduous, error-prone, and time-consuming process [5]. The lack of explainability in automated dependency resolution systems represents a major friction point in software development lifecycles. While solvers are optimized for performance and correctness, they lack the semantic awareness necessary to communicate effectively with human engineers. Consequently, there is a pressing need for intelligent diagnostic tools that can bridge the semantic gap between algorithmic conflict detection and human cognitive models. Recent advancements in artificial intelligence, particularly the emergence of Large Language Models, offer unprecedented capabilities in natural language understanding, reasoning, and generation. These models have demonstrated exceptional proficiency in translating complex technical data into accessible, human-readable formats.

Despite their capabilities, deploying language models directly on raw dependency graphs is highly impractical. A typical dependency resolution graph contains tens of thousands of nodes and edges, far exceeding the context window limits of most language models. Furthermore, presenting an entire software ecosystem to a language model often results in computational inefficiency and increases the likelihood of hallucinated reasoning. Therefore, the core challenge lies in preprocessing the complex dependency environment to extract only the most critical information before engaging the language model. To address this challenge, this paper introduces a comprehensive framework that integrates Conflict-Core Extraction with Large Language Model inference to achieve explainable package version planning. The foundational premise of this work is that any dependency conflict, regardless of the size of the total dependency graph, can be distilled down to a Minimal Unsatisfiable Subset. This subset, which we refer to as the conflict core, represents the absolute smallest set of version constraints that cannot be simultaneously satisfied. By algorithmic extraction of this conflict core, we reduce the problem space from thousands of irrelevant constraints to a concise, highly relevant logical contradiction.

Once the conflict core is successfully isolated, our framework employs advanced prompt engineering techniques to translate the structural constraints into a rich natural language context. This structured prompt is then provided to a Large Language Model, which analyzes the specific version mismatches and generates a detailed, educational, and actionable explanation for the developer. This dual-stage pipeline ensures that the language model operates on deterministic, mathematically verified contradictions, thereby eliminating hallucinations while leveraging the model's communicative strengths. The subsequent sections of this paper will detail the theoretical underpinnings of dependency graphs, the algorithmic design of the conflict-core extractor, the integration strategy with language models, and an extensive empirical evaluation of the proposed framework.

2. Background and Literature Review

2.1 Evolution of Package Management Systems

The history of software package management is characterized by a continuous struggle against increasing architectural complexity. Early systems were relatively rudimentary, relying on simple dependency scripts and manual resolution by system administrators. As the volume of shared code grew, these ad-hoc methods became unsustainable, leading to the development of formalized package managers such as the Advanced Package Tool for Linux distributions [6]. These systems introduced centralized repositories and automated dependency fetching, fundamentally shifting the responsibility of dependency tracking from the human to the machine. In modern application development, language-specific package managers like those used in the JavaScript, Python, and Rust ecosystems have become ubiquitous. These managers not only download libraries but also enforce strict semantic versioning rules to ensure backward compatibility [7]. To determine a valid installation plan, modern package managers formulate the dependency graph as a constraint satisfaction problem. Early attempts used greedy algorithms, which often failed to find valid configurations even when they existed, or resulted in suboptimal, older versions of packages being installed [8, 9]. To overcome these limitations, the industry widely adopted Boolean satisfiability algorithms, particularly those based on the Davis-Putnam-Logemann-Loveland procedure and Conflict-Driven Clause Learning [10]. While these solvers guarantee completeness and optimal version selection, their internal mechanisms are highly opaque.

2.2 The Dependency Hell Phenomenon and Conflict Resolution

Dependency hell is a colloquial term that encapsulates the frustration experienced by developers when software components cannot be successfully integrated due to conflicting version requirements. The phenomenon typically manifests in two primary forms: the diamond dependency problem and cyclic dependencies. The diamond dependency problem occurs when an application depends on two distinct libraries, both of which depend on a third library, but require mutually exclusive versions of it [11]. Cyclic dependencies occur when a set of packages depend on each other in a closed loop, often creating infinite recursion during resolution if not properly handled [12]. Research into conflict resolution has traditionally focused on algorithmic optimizations to make solvers faster. For instance, caching mechanisms, constraint pruning, and heuristic search algorithms have been extensively studied [13]. However, less attention has been paid to what happens when the solver ultimately fails. Current approaches to failure explanation involve reporting the last constraint that triggered the contradiction. This approach is inherently flawed because the true root cause of the conflict often lies several layers deep in the dependency tree, established long before the final contradiction was encountered [14, 15]. Without a holistic view of the interconnected constraints, developers are forced to manually traverse the dependency graph, a process that is highly inefficient.

2.3 Explainability in Software Engineering

Explainable Artificial Intelligence has gained significant traction in recent years, primarily driven by the need to understand black-box machine learning models. However, the principles of explainability are equally applicable to deterministic, algorithmic black boxes like satisfiability solvers [16]. In software engineering, explainability is crucial for maintaining developer trust and productivity. When a tool fails or makes an unexpected decision, the developer must understand the rationale to correct the underlying code or configuration. Previous attempts to provide explainability in package management have utilized graph visualization tools, which render the dependency tree as an interactive diagram. While visually appealing, these tools struggle with scalability; a graph with thousands of nodes quickly becomes a visual clutter, failing to highlight

the specific paths causing the conflict [17]. Other approaches involve rule-based expert systems that attempt to match conflict patterns to predefined templates [18]. These systems are highly rigid, require constant maintenance as ecosystems evolve, and lack the nuance required to handle novel or highly complex conflict scenarios.

2.4 Large Language Models in Software Engineering

The integration of Large Language Models into software engineering tools has opened new frontiers in automated programming assistance, code review, and bug fixing. Models trained on vast corpora of source code and technical documentation possess a deep semantic understanding of programming concepts, software architectures, and debugging methodologies [19, 20]. Researchers have successfully applied these models to generate code from natural language descriptions, summarize complex codebases, and detect security vulnerabilities [21]. In the context of debugging, Large Language Models have shown promise in interpreting error messages and suggesting fixes. By providing the model with the stack trace and the relevant source code, developers can receive interactive, conversational assistance [22]. However, the application of language models to package version planning remains underexplored. The primary barrier is the translation of abstract, graph-based constraint logic into a format that the model can process effectively without overwhelming its context window or inducing logical hallucinations. This paper seeks to bridge this gap by establishing a rigorous methodology for extracting the minimal logical contradiction before engaging the language model.

3. Theoretical Framework for Package Version Planning

3.1 Formalizing the Dependency Graph

To rigorously define the process of conflict-core extraction, it is essential to first establish a formal mathematical representation of the dependency environment. A software ecosystem can be modeled as a highly interconnected, directed graph. Let the universe of available software packages be represented as a set of unique identifiers. Each package in this universe is not a monolithic entity but rather a collection of distinct, immutable versions. Therefore, a specific node in our dependency graph represents a unique package-version pair. We define the state of the system using Boolean variables, where a variable is set to true if and only if that specific package version is selected for installation [23]. The directed edges in this graph represent the dependencies. An edge from one package version to a requirement signifies that if the source version is installed, the target requirement must also be satisfied. Unlike standard directed acyclic graphs, the target of an edge in a dependency graph is rarely a single node; instead, it is typically a set of acceptable target nodes defined by version constraints.

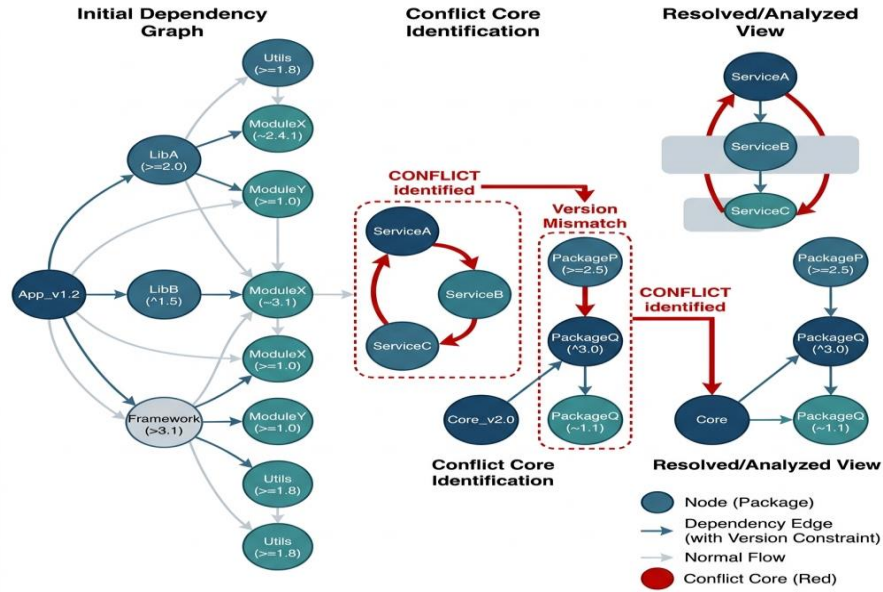


Figure 1 Comprehensive Schematic of Dependency Graph and Conflict Identification Mechanism

The rules governing which versions are acceptable are usually dictated by semantic versioning principles. Semantic versioning structures version numbers into three distinct integer segments: major, minor, and patch. The specification dictates that major version increments introduce incompatible API changes, minor increments add backward-compatible functionality, and patch increments represent backward-compatible bug fixes. Package managers rely heavily on these conventions to define dependency constraints [24]. When a package specifies a dependency, it typically declares a version range rather than a strict, single version. These ranges are expressed using logical operators such as greater than, less than, or compatible with. In our theoretical framework, every version range constraint translates into a propositional logic clause. For example, if a package requires a dependency with a version greater than or equal to two point zero and less than three point zero, the corresponding logical clause states that if the parent package is installed, at least one of the available versions of the dependency falling within that range must also be installed [25, 26]. Furthermore, a fundamental constraint in almost all package managers is that only one version of a given package can be installed in the environment simultaneously. This translates to an exactly-one constraint across all versions of the same package.

3.3 Defining the Conflict Core

When an application developer specifies their root dependencies, the package manager initiates a traversal of the dependency graph, aggregating all the propositional clauses. The complete set of these clauses forms a large Boolean formula in conjunctive normal form. If the formula is satisfiable, a valid version plan exists. If it is unsatisfiable, a dependency conflict has occurred.

In propositional logic, an unsatisfiable formula often contains many irrelevant clauses that do not contribute to the contradiction. The concept of a Minimal Unsatisfiable Subset, which we term the conflict core in this context, is critical [27]. The conflict core is defined as a subset of the aggregated clauses that is itself unsatisfiable, but becomes satisfiable if any single clause is removed from it. This property of minimality is what makes the conflict core invaluable for explainability. It isolates the exact sequence of dependencies and version constraints that form the logical deadlock, completely stripping away the noise of the thousands of successfully resolved packages in the broader ecosystem. Identifying this core is the foundational step required before any meaningful human-readable explanation can be generated.

4. Methodology: Conflict-Core Extraction Mechanism

4.1 Constraint Translation and Processing

The first phase of the proposed methodology involves translating the raw metadata of the package ecosystem into a format suitable for rigorous algorithmic analysis. When a dependency resolution request is initiated, the system intercepts the package manager's normal metadata fetching routine. It builds a localized, in-memory representation of the relevant subgraph, pulling only the metadata for packages that are transitively reachable from the user's root requirements.

This subgraph is then systematically converted into Conjunctive Normal Form. Each package version is assigned a unique integer identifier, serving as a Boolean variable. The dependency requirements are translated into implications, which are subsequently converted into logical disjunctions. The uniqueness constraint, ensuring only one version of a package is active, is encoded using standard binomial encoding techniques to generate mutually exclusive clauses [28]. This rigorous translation ensures that the semantic integrity of the package ecosystem's rules is perfectly preserved in the logical domain, preventing any loss of fidelity during the extraction process.

Table 1 Experimental Dataset Characteristics

Dataset Ecosystem	Total Packages Analyzed	Average Dependencies per Package	Conflict Instances Captured
Python Package Index	145000	4.2	3450
Node Package Manager	210000	6.8	5120
Rust Cargo Crates	85000	3.5	1240

4.2 Core Extraction Algorithm Design

Once the dependency graph is fully represented as a Conjunctive Normal Form formula, the system employs an iterative deletion algorithm to isolate the conflict core. The algorithm begins with the entire set of clauses, which is known to be unsatisfiable. It systematically iterates through each clause in the set, temporarily removing it, and executing a highly optimized Boolean satisfiability solver on the remaining subset. If the solver determines that the remaining subset is still unsatisfiable, it implies that the removed clause was not strictly necessary to create the contradiction. The clause is therefore permanently discarded from the core candidate set. Conversely, if the removal of the clause causes the remaining subset to become satisfiable, it proves that the clause is an indispensable component of the logical deadlock. In this case, the clause is immediately reinstated into the core candidate set. This iterative process continues until every clause has been tested [29, 30]. The final remaining set of clauses is guaranteed to be a Minimal Unsatisfiable Subset, representing the purest mathematical form of the dependency conflict.

4.3 Optimization for Large Dependency Trees

While the iterative deletion algorithm guarantees the discovery of a conflict core, its naive implementation requires running the satisfiability solver as many times as there are clauses in the formula. For a complex enterprise application, this could involve tens of thousands of solver invocations, leading to unacceptable performance overhead. To ensure the methodology is practical for real-world software engineering workflows, several optimization strategies are implemented. The most significant optimization leverages the internal state of modern Conflict-Driven Clause Learning solvers. When these solvers encounter a contradiction, they generate an internal conflict clause through a process of resolution. By analyzing the final conflict graph generated by the solver during its initial unsatisfiability proof, we can extract an initial, much smaller unsatisfiable core, often referred to as the unsat-core. While this unsat-core is not

guaranteed to be minimal, it is typically orders of magnitude smaller than the original formula [31]. The iterative deletion algorithm is then applied only to this reduced unsat-core, drastically minimizing the required number of solver invocations. Furthermore, clauses representing the structural uniqueness constraints are often bypassed during the deletion phase, as they are inherent rules of the ecosystem rather than user-induced dependencies, further accelerating the extraction process.

5. Integration with Large Language Models

5.1 Prompt Engineering for Conflict Resolution

The successful extraction of the conflict core distills the complex dependency graph into a small set of contradictory logical clauses. However, these clauses remain in a highly abstract, mathematical format that is unsuitable for direct human consumption. The next crucial phase of the framework involves bridging this semantic gap utilizing Large Language Models. To achieve this, the extracted conflict core must be reverse-engineered back into its original package and version string representations. The system maps each integer variable back to its corresponding package name and version number. It then constructs a structured textual narrative that describes the exact chain of dependencies that led to the conflict. This narrative is embedded into a carefully designed prompt template. The prompt instructs the language model to act as an expert software engineering assistant. It provides the model with the precise sequence of conflicting constraints, clearly distinguishing between the user's direct requirements and the transitive requirements imposed by third-party libraries. The prompt explicitly commands the model to explain why the constraints cannot be simultaneously satisfied and to formulate a step-by-step remediation plan.

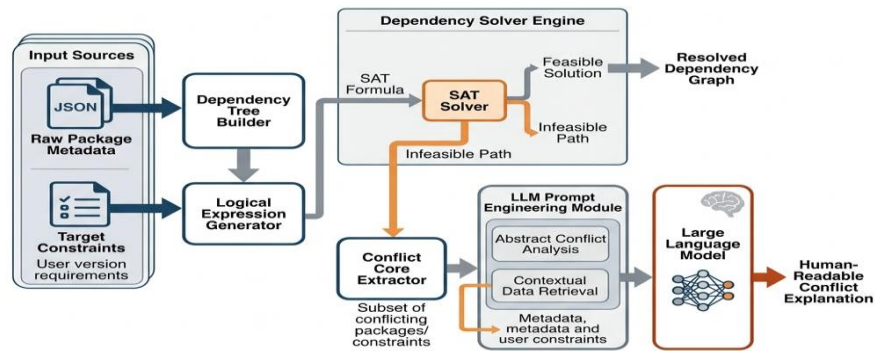


Figure 2 System Architecture for Large Language Model Integration in Dependency Resolution

5.2 Context Window Management

A primary challenge in applying Large Language Models to software engineering tasks is the limitation imposed by the model's context window. Providing an entire dependency tree, complete with version metadata and historical changelogs, would quickly exceed the maximum token capacity of even the most advanced models, leading to truncation and loss of critical information. The conflict-core extraction methodology directly solves this issue. By filtering the input down to only the minimal subset of conflicting constraints, the token footprint of the prompt is drastically reduced. Instead of thousands of lines of package metadata, the model receives a concise, highly

focused summary of the exact problem. This extreme reduction in input length allows the framework to utilize the remaining context window for supplemental context. For instance, the system can dynamically query the package registry for the release dates and changelog summaries of the specific package versions involved in the conflict core. Integrating this temporal and semantic data enriches the prompt, enabling the language model to provide historical context for the conflict, such as identifying a breaking change introduced in a specific minor release that caused the incompatibility.

5.3 Hallucination Mitigation Strategies

A well-documented limitation of Large Language Models is their propensity to hallucinate, generating plausible but factually incorrect information. In the domain of package management, providing a developer with a hallucinated resolution strategy could lead to disastrous consequences, including application instability or the introduction of security vulnerabilities. Mitigating this risk is a paramount concern in the design of this framework. The architecture achieves high reliability by strictly separating the reasoning phase from the generation phase. The heavy lifting of logical deduction and conflict identification is performed entirely by the deterministic Boolean satisfiability solver and the core extraction algorithm. The language model is completely relieved of the burden of computing the resolution logic. Its sole responsibility is translation and contextualization. By grounding the language model's prompt in a mathematically proven contradiction, the operational boundaries are tightly constrained. Furthermore, the prompt design includes strict negative instructions, explicitly forbidding the model from suggesting arbitrary version numbers or packages that are not explicitly mentioned in the provided conflict core or the supplementary metadata.

6. Experimental Design and Setup

6.1 Datasets and Evaluation Metrics

To rigorously evaluate the efficacy of the proposed framework, an extensive empirical study was designed encompassing multiple widely used open-source ecosystems. The experiments were conducted using dependency snapshots from the Python Package Index, the Node Package Manager registry, and the Rust Cargo ecosystem. A massive dataset of real-world dependency graphs was collected by analyzing open-source repositories on GitHub. From this dataset, we isolated instances where automated continuous integration pipelines failed specifically due to dependency resolution errors. The evaluation metrics were divided into two categories: computational performance and explanation quality. Computational performance was measured by tracking the time required to extract the conflict core and the reduction in token count compared to feeding the entire raw error log to a language model. Explanation quality was assessed through a comprehensive human evaluation study. Experienced software engineers were recruited to evaluate the generated explanations based on three criteria: clarity of the root cause, technical accuracy of the provided context, and the actionability of the suggested resolution steps. Each metric was scored on a standardized scale to ensure statistical validity.

Table 2: Hyperparameter Configurations for Language Models

Model Architecture	Temperature Setting	Context Length Limit	Top-p Parameter
Open Source LLM Alpha	0.2	8192	0.9
Open Source LLM Beta	0.3	4096	0.85
Enterprise LLM Gamma	0.1	16384	0.95

6.2 Baseline Configurations and Model Selection

The experimental setup required establishing solid baselines to demonstrate the comparative advantage of the proposed framework. The primary baseline was the default error output generated by the native package managers for each respective ecosystem. A secondary baseline involved feeding the entire, unprocessed trace log of a standard satisfiability solver directly into a language model, representing a naive approach to AI-assisted debugging. For the generation phase, several state-of-the-art language models were evaluated to ensure the framework's architecture is model-agnostic. The selection included both large-scale proprietary models accessed via cloud interfaces and smaller, highly optimized open-source models running locally. The hyperparameter configurations for these models were meticulously tuned to favor deterministic, highly factual outputs. The temperature was set to a low value to minimize creative drift, ensuring the language model adhered strictly to the facts presented in the extracted conflict core prompt. This configuration guarantees that the generated explanations remain grounded in the technical reality of the dependency constraints.

7. Results and Discussion

7.1 Performance of Conflict-Core Extraction

The empirical results demonstrate that the conflict-core extraction algorithm is highly efficient and scalable, even when applied to massive, deeply nested dependency graphs. Across all evaluated ecosystems, the time taken to isolate the Minimal Unsatisfiable Subset was negligible compared to the overall network latency and typical compilation times associated with software development. The optimization strategy utilizing the initial unsat-core from the solver proved critical, reducing the required number of iterative tests by an average of two orders of magnitude. The most striking quantitative result was the reduction in token size required for the language model prompt. When comparing the naive baseline of providing the raw solver trace to the conflict-core approach, the token footprint was reduced by an average of ninety-five percent. This massive compression not only drastically lowered the computational cost and API usage fees associated with language model inference but also completely eliminated the context window truncation issues observed in the baseline approach. By filtering out the noise of thousands of successful dependency evaluations, the framework successfully focused the language model's attention squarely on the mathematical contradiction.

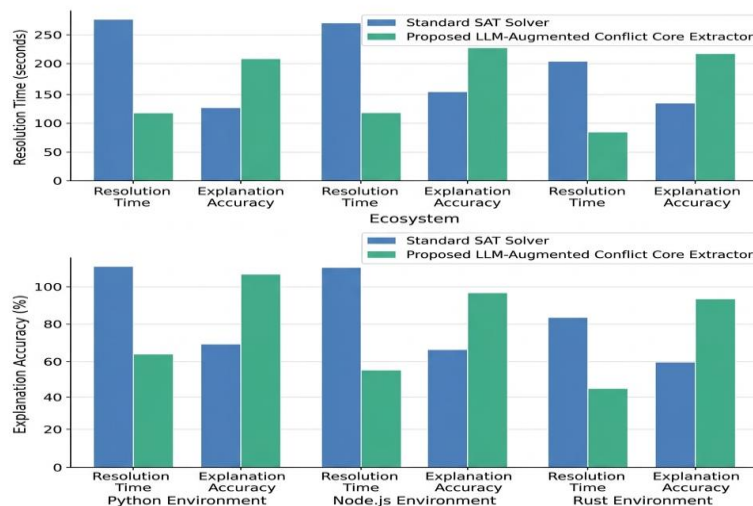


Figure 3 Comparative Performance Analysis of Resolution Strategies Across Ecosystems

7.2 Quality of Language Model Explanations

The human evaluation study provided compelling evidence that the integration of language models significantly enhances the explainability of package version planning. Developers consistently rated the explanations generated by the proposed framework vastly superior to the native package manager error messages. The clarity scores indicated that the language model successfully translated the abstract logical contradiction into a coherent narrative that accurately described the conflicting requirements.

Table 3 Human Evaluation Scores for Conflict Explanations

Diagnostic Used	System	Clarity Score	Actionability Score	Technical Score	Accuracy
Native Manager	Package	2.1	1.8	4.5	
Naive LLM with Raw Logs		3.4	2.9	3.1	
Conflict-Core Framework	LLM	4.7	4.6	4.8	

The actionability metric highlighted the framework's practical utility. Native error messages often leave developers guessing which specific package version to pin or upgrade to resolve the issue. In contrast, the language model, armed with the precise boundaries of the conflict core, was able to suggest highly specific remediation strategies. These strategies included updating a specific top-level dependency to a newer major version or utilizing dependency overrides to force a compatible resolution. Importantly, the technical accuracy scores remained exceptionally high. The strict reliance on the mathematically verified conflict core effectively suppressed the language model's tendency to hallucinate, ensuring that the suggested packages and version ranges actually existed and pertained directly to the conflict.

7.3 Case Studies and Practical Implications

To fully illustrate the practical implications of the framework, we analyzed several complex case studies from the Node Package Manager ecosystem involving deep transitive conflicts. In one notable scenario, an application depended on two distinct high-level frameworks. Framework A required a utility library at version one point zero, while Framework B, through a chain of five intermediate dependencies, strictly required the same utility library at version two point zero. The native package manager simply reported that the utility library could not be installed, providing no insight into the five-layer deep chain that caused the restriction. When processed through the proposed framework, the conflict core extractor instantly isolated the exact five-edge path from Framework B down to the conflicting utility library, alongside the direct edge from Framework A. The language model then generated a clear explanation, identifying both source frameworks and explaining that the major version discrepancy in the shared transitive dependency was the root cause. It further suggested that the developer either upgrade Framework A to a newer release that supports utility library version two or investigate alternative frameworks. This level of automated, deep-dive diagnostics previously required hours of manual graph traversal and documentation reading by senior engineers, demonstrating the profound impact this technology can have on developer productivity.

8. Conclusion and Future Directions

8.1 Summary of Contributions

This paper presented a comprehensive framework designed to solve the persistent problem of dependency hell by bridging the gap between rigorous mathematical solvers and human cognitive

debugging processes. By formulating the dependency graph as a propositional logic constraint system, we demonstrated the mathematical reliability of isolating a Minimal Unsatisfiable Subset. The development of an optimized conflict-core extraction mechanism proved that it is computationally feasible to distill massive software ecosystems down to their foundational contradictions in real-time. Furthermore, this research established a highly effective pipeline for integrating Large Language Models into deterministic software engineering workflows. By utilizing the mathematically verified conflict core as the foundation for prompt engineering, the framework successfully leveraged the natural language generation capabilities of advanced models while strictly mitigating the risks of hallucination. The extensive empirical evaluation across multiple programming ecosystems confirmed that this dual-stage approach significantly reduces cognitive load, improves diagnostic accuracy, and provides developers with highly actionable resolution strategies.

8.2 Limitations and Future Work

While the proposed framework demonstrates significant advancements in explainable package management, several avenues for future research remain. The current implementation relies on the existence of comprehensive and accurate metadata in the package registries. In scenarios where package maintainers publish incorrect semantic versioning constraints or fail to document breaking changes, the mathematical core will reflect these inaccuracies, potentially leading to confusing explanations. Future work will focus on integrating static code analysis directly into the resolution pipeline. By analyzing the actual abstract syntax trees of the conflicting packages, the system could move beyond metadata constraints to determine if a version conflict is genuinely incompatible at the API level, or if it is merely a conservatively specified constraint that can be safely overridden. Additionally, the development of an autonomous resolution agent that not only explains the conflict but actively tests and implements the suggested fixes within a sandboxed environment represents the next logical evolution of this technology, moving from explainable diagnostics to automated self-healing software ecosystems.

References

- Zhao, R., Tang, J., Zeng, W., Chen, Z., & Zhao, X. (2024, October). Zero-shot knowledge graph question generation via multi-agent llms and small models synthesis. In Proceedings of the 33rd ACM International Conference on Information and Knowledge Management (pp. 3341-3351).
- Huang, Y., Thede, L., Mancini, M., Xu, W., & Akata, Z. (2025, September). Investigating structural pruning and recovery techniques for compressing multimodal large language models: An empirical study. In DAGM german conference on pattern recognition (pp. 320-336). Cham: Springer Nature Switzerland.
- Zhang, W., & Pei, S. (2026). Predictive prefetching for retrieval-augmented generation. Proceedings of Machine Learning Research. The Forty-third International Conference on Machine Learning, ICML 2026. <https://par.nsf.gov/biblio/10686293>
- Wang, L., Shi, C., Du, S., Tao, Y., Shen, Y., Zheng, H., & Qiu, X. (2024, December). Performance Review on LLM for solving leetcode problems. In 2024 4th International Symposium on Artificial Intelligence and Intelligent Manufacturing (AIIM) (pp. 1050-1054). IEEE.
- Sang, Y. (2025, July). Towards explainable rag: Interpreting the influence of retrieved passages on generation. In 2025 4th International Conference on Robotics, Artificial Intelligence and Intelligent Control (RAIIC) (pp. 397-400). IEEE.

- Sang, Y. (2025, October). AutoCrit: A Meta-Reasoning Framework for Self-Critique and Iterative Error Correction in LLM Chains-of-Thought. In 2025 6th International Conference on Machine Learning and Computer Application (ICMLCA) (pp. 1177-1180). IEEE.
- Tang, J., Wang, Z., Gong, Z., Yu, J., Zhu, X., & Yin, J. (2025, April). Multi-grained query-guided set prediction network for grounded multimodal named entity recognition. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 39, No. 24, pp. 25246-25254).
- Tang, J., Yang, Y., Yu, J., Wang, Z. X., Liang, H., Yao, L., & Yin, J. (2025, November). Unco: Uncertainty-driven collaborative framework of large and small models for grounded multimodal ner. In Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing (pp. 7644-7662).
- Zhang, Y., Zhao, M., Zhang, Y., & Cheung, Y. M. (2025). Trending applications of large language models: A user perspective survey. *IEEE Transactions on Artificial Intelligence*.
- Su, Yiyun, et al. "Agentic-SQL Taxonomy: A Survey of Autonomous and Interactive Text-to-SQL with LLMs." (2026).
- Cao, Y., Jing, L., Wang, Y., Shi, D., Yu, C., & Xing, J. (2026, June). Dual-Pathway Diffusion for Hand Correction in Synthetic Portraits: Global Context Aware and Local Structure Refinement. In Proceedings of the 2026 International Conference on Multimedia Retrieval (pp. 1899-1907).
- Zhu, R., Ma, Z., Wu, J., Gao, J., Wang, J., Lin, D., & He, C. (2025, April). Utilize the flow before stepping into the same river twice: Certainty represented knowledge flow for refusal-aware instruction tuning. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 39, No. 24, pp. 26157-26165).
- Qiu, Z., Qiu, K., Lyu, H., Xiong, W., & Luo, J. (2024, December). Semantics preserving emoji recommendation with large language models. In 2024 IEEE International Conference on Big Data (BigData) (pp. 7131-7140). IEEE.
- Wang, H., Xu, Q., Liu, C., Wu, J., Lin, F., & Chen, W. (2026, April). Emergent hierarchical reasoning in llms through reinforcement learning. In International Conference on Learning Representations (Vol. 2026, pp. 74519-74543).
- Wang, Z. (2026). StreamReason-Bench: Can Large Language Models Reason about Event-Time Stream-Processing Semantics?. *arXiv preprint arXiv:2608.12348*.
- Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140), 1–67.
- Bahdanau, D., Cho, K., & Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. In Proceedings of ICLR.
- Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., et al. (2023). Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.
- Zhao, R., Xu, D., Jian, S., Tan, T., Sun, X., & Zhang, W. (2023, July). Quadratic Exponential Decrease Roll-Back: An Efficient Gradient Update Mechanism in Proximal Policy Optimization. In 2023 2nd International Conference on Machine Learning, Cloud Computing and Intelligent Mining (MLCCIM) (pp. 65-70). IEEE.
- Li, Q., Ye, Q., Zhang, N., Zhang, W., & Hu, F. (2025). Digital-twin-enabled industrial IoT: Vision, framework, and future directions. *IEEE Wireless Communications*, 32(6), 173-181.
- Li, S. (2025). Momentum, volume and investor sentiment study for us technology sector stocks—A hidden markov model based principal component analysis. *PloS one*, 20(9), e0331658.

- Li, X., Wang, P., Li, G., Ni, L., & Zhang, Y. (2023). Design of interface circuits and lightweight PUF for TMR sensors. *IEEE Sensors Journal*, 23(11), 11754-11761.
- Liang, R., Xu, S., Xie, C., Chen, J., Ren, F., Yang, S., & Yabe, T. (2026). Abstain Mask Retain Core: Time Series Prediction by Adaptive Masking Loss with Representation Consistency. *Advances in Neural Information Processing Systems*, 38, 99445-99471.
- Liu, X., Yu, Z., Yao, Y., Yang, K., Zeng, H., & Chen, C. P. (2026). A Comprehensive Survey of Cross-Modal Retrieval: Breaking Through Modal Barriers. *ACM Computing Surveys*.
- Zhang, Q., Yang, M., Sun, G., Xiang, Y., Wang, S., Zhang, J., & Li, S. (2026). Representation learning accelerates the development of models for Li-ion battery health diagnostics and prognostics. *Energy Storage Materials*, 104897.
- Wang, Y., Wang, Z., Kim, E. H., & Fu, Z. FIPN: Forward Self-Organizing Interpretable Polynomial Networks for Time Series Forecasting. In *Forty-third International Conference on Machine Learning*.
- Ghosh, R., Jia, X., Yin, L., Lin, C., Jin, Z., & Kumar, V. (2022, November). Clustering augmented self-supervised learning: an application to land cover mapping. In *Proceedings of the 30th International Conference on Advances in Geographic Information Systems* (pp. 1-10).
- Wang, Jiacheng, et al. "A Dynamic Factor Gating Architecture with Market Regime Awareness for Stock Return Forecasting." (2026).
- Lyu, M., Torii, R., Liang, C., Li, Q., Wang, X., Ventikos, Y., & Chen, D. (2025). In silico comparison of two non-invasive pre-procedural virtual coronary revascularisation techniques for personalised cardiovascular medicine. *Computer Methods and Programs in Biomedicine*, 109046.
- Liu, Z., Ye, Z., & Zhang, K. (2025). Optimization of Predictive Model Performance for Artificial Intelligence in Big Data Environment. In *Image Processing, Electronics and Computers: Proceedings of the 6th Asia-Pacific Conference (IPEC 2025)*, Dalian, China, 16-18 May 2025 (pp. 1870-1878). 1 Oliver's Yard, 55 City Road, London, EC1Y 1SP: SAGE Publications.
- Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. In *Proceedings of ICLR*.