Advanced Data Processing Algorithms and Structures for Technical Debt Management with Generative Artificial Intelligence

Adam Czyzewski
Institute of Information Technology/Lodz University of Technology
Lodz, Poland *adam.czyzewski@dokt.p.lodz.pl*

Aneta Poniszewska-Maranda
Institute of Information Technology/Lodz University of Technology
Lodz, Poland *aneta.poniszewska-maranda@p.lodz.pl*

Abstract

Technical debt management is increasingly critical in modern software systems, where organizations grapple with complex digital infrastructure. This paper aims to explore innovative data processing algorithms and frameworks that leverage generative AI to improve the diagnosis and management of technical debt problems. We conduct a literature review and synthesize findings on the application of generative AI in technical debt management, focusing on algorithmic approaches and frameworks designed for this purpose. Analysis reveals that generative AI-driven methods show promise in enabling more accurate diagnosis of technical debt, particularly in automating the identification of complex patterns and generating targeted remediation strategies.

Keywords: Technical debt, technical debt management, generative artificial intelligence, data processing algorithms, data processing structures.

1. Introduction

In rapidly changing technological environment, organizations must constantly adapt their IT systems to keep up with the growing demands of the market and users. This constant development is often associated with a phenomenon known as "technical debt", which occurs when design decisions made under time and cost pressure lead to suboptimal solutions. Technical debt, although inevitable in short-term projects, can lead to significant challenges in the long term, affecting the performance, scalability, and maintenance of systems [1, 2]. In response to these challenges, the development of advanced data processing algorithms and data structures is becoming increasingly important. In this context, the role of generative artificial intelligence (GAI) begins to play a key role. GAI, capable of learning and generating new patterns, offers innovative approaches to solving problems related to technical debt [1], [3]. Generative AI can be used to automate the processes of diagnosing and assessing technical debts, which contributes to faster and more precise detection of areas requiring intervention. By analyzing the current state of systems and predicting potential failure points, GAI supports the creation of more informed decisions regarding investments in development of IT infrastructure, also more complementary analysis of technical debt [2, 3]. This paper aims to explore the importance of modern data structures and processing algorithms in context of managing technical debt using generative AI.

2. Technical debt management as a software engineering process

Technical debt refers to the process in software engineering where developers make expedient decisions during software development – often to meet immediate deadlines or to quickly address urgent requirements – which may result in less optimal code structures or design choices [1]. These shortcuts and compromises can lead to future maintenance challenges, decreased software quality, and increased development costs, collectively termed as technical debt. Effective technical debt management involves identifying, measuring, prioritizing, and systematically addressing debt items to balance immediate benefits against future risks and costs.

Types of technical debt. Technical debt can be categorized into several distinct types, each requiring different management strategies [3]: (1) *Code debt*: arises from poorly written or unoptimized code, often characterized by complex structures, duplication, and lack of proper documentation. (2) *Architectural debt*: results from suboptimal architecture decisions that might hinder scalability, flexibility, or adaptability to future requirements. (3) *Design debt*: occurs due to inadequate or rushed design decisions that impact the maintainability and clarity of the software. (4) *Documentation debt*: refers to insufficient, outdated, or incorrect documentation, making the software difficult to understand and maintain. (5) *Testing debt*: develops when there is inadequate testing coverage, resulting in undiscovered bugs and vulnerabilities that complicate future changes.

Lifecycle of technical debt. Technical debt follows a lifecycle involving several distinct phases: (1) *Identification*: recognizing areas of technical debt through methods such as code reviews, static analysis tools, continuous integration systems, automated testing reports, and developer or stakeholder feedback. (2) *Assessment*: evaluating the significance, impact, and urgency of identified debt using various quantitative metrics and qualitative judgment (e.g., expert opinions, team surveys). (3) *Prioritization*: determining the sequence and importance of addressing specific debt items, based on their potential to negatively affect software quality, business objectives, project timelines, and available resources. (4) *Resolution*: implementing improvements through techniques such as refactoring, redesigning modules, rewriting problematic components, or improving documentation, always balancing immediate resource constraints against longer-term maintenance benefits. (5) *Monitoring and prevention*: continuously tracking technical debt to understand its evolution over time, ensuring new debt is recognized early, and promoting practices to minimize future debt accrual (e.g., enforcing coding standards).

Impact on software quality and sustainability. Unchecked technical debt can substantially degrade software quality, leading to increased complexity, higher defect rates, reduced productivity, and greater difficulty in accommodating future changes or enhancements. Over time, technical debt accumulation can also hinder a project's responsiveness, and adaptability to new user requirements. Furthermore, persistent technical debt can create significant barriers to innovation, diverting resources away from development of new features toward maintenance and refactoring. This paper focuses specifically on code and architectural debt, as these types are among the most prevalent and impactful in industry practices.

Metrics and measurement techniques. Effective measurement of technical debt relies on capturing both code-level and architectural properties that directly impact maintainability and system resilience [3]. Code metrics reflect internal complexity, size, quality of source code: (1) *Complexity and effort*: cyclomatic complexity quantifies branching logic and test effort, while Halstead metrics assess code volume, difficulty, and estimated comprehension time. High values indicate hotspots requiring review or refactoring. (2) *Size and churn*: lines of code (LOC) serve as a basic measure of system size; sudden spikes or modules with sustained high churn often signal rushed or unstable code that may harbor hidden debt. (3) *Duplication*: clone detection finds identical or near-duplicate code fragments, which inflate maintenance effort and risk inconsistent bug fixes. (4) *Coupling and cohesion*: coupling metrics (e.g. dependencies between modules) reveal tightly interconnected components that are hard to change in isolation; cohesion metrics (e.g. how closely methods within a class relate) indicate whether modules have clear, focused responsibilities. (5) *Quality indicators*: test coverage percentages expose untested regions prone to regressions, and code smell counts highlight common anti-patterns (such as long methods or large classes) with an associated remediation cost.

Available tools for technical debt measurement. While custom scripts can compute individual metrics, several mature platforms offer integrated measurement and reporting. However, two major platforms commonly used: (1) *SonarQube*: provides static code analysis for multiple languages, computing complexity, duplication, coverage, and smell counts. It allows threshold configuration, pull-request decoration, and historical trend charts. SonarQube's SQALE model also estimates remediation costs, producing a consolidated debt report [3]. (2) *CAST Software Engineering Platform*: focuses on architectural and structural analysis, building dependency graphs and measuring coupling, cohesion, and layering violations. CAST also delivers detailed architectural health reports and remediation roadmaps. These tools integrate into CI/CD pipelines, enabling continuous measurement, automated gating, and visualization through dashboards or IDE plugins. Selecting the right combination depends on language support, budget, desired metric depth, and integration requirements.

3. Algorithms and data structures for measuring and representing technical debt

Graphs. Graphs model software components as nodes and their relationships as edges, enabling holistic analysis of architecture and code interdependencies. Two common graph types: (1) *Dependency graphs*: nodes represent modules, packages, or classes, with directed edges indicating compile-time or run-time dependencies. Analysis of dependency graphs can reveal tightly coupled clusters, cyclical dependencies, and architectural violations. Algorithms such as Tarjan's strongly connected components detect cycles, while centrality measures (e.g., betweenness, eigenvector centrality) identify modules that act as critical bridges. Debt hotspots correspond to highly central or cyclically related nodes where changes propagate widely [3]. (2) *Call graphs*: nodes represent functions or methods, with edges for invocation relationships. Call graphs support detection of overly complex routines (dense subgraphs) and the analysis of execution paths. Path-based metrics, like longest call chains and fan-in/fan-out counts, highlight methods with high maintenance risk. Graph traversal algorithms (depth-first search, breadth-first search) aid in impact analysis: determining the ripple effect of changes and prioritizing refactoring targets [1].

Abstract Syntax Trees. Abstract Syntax Trees [2] provide a structured, hierarchical representation of source code, breaking down programs into their constituent syntactic elements such as expressions, statements, and declarations. By capturing the grammatical structure of code, ASTs enable tools and algorithms to understand not just the textual content, but also the relationships and nesting that define how a program operates. Processing an AST typically involves traversing its nodes in pre-order or post-order fashion, which allows static analysis tools to identify patterns indicative of technical debt. For instance, deeply nested conditional blocks or excessively large method subtrees often correspond to complex or error-prone code segments. By measuring tree depth and counting the number of nodes in specific subtrees, analysts can pinpoint "God classes" or long methods that are candidates for refactoring. AST traversal thus serves as a powerful mechanism to extract detailed code metrics beyond simple line counts.

Machine Learning algorithms. Machine learning supplements structural representations by learning patterns of technical debt from historical data: (1) *Classification*: supervised models (e.g. decision trees, support vector machines, random forests) classify code modules or commits as "debt" or "non-debt" based on feature vectors derived from metrics and embeddings. Training labels originate from past refactoring records or expert annotations. Classification aids in automatic detection of debt hotspots and generates precision/recall statistics to validate metric thresholds [1]. (2) *Clustering*: unsupervised algorithms group similar code components based on metric profiles or embedding similarities. Clusters often correspond to functional areas or debt domains; outlier clusters identify modules with unusual complexity or duplication patterns [1]. (3) *Ranking*: graph-inspired ranking algorithms, such as PageRank, can prioritize debt items by treating modules as nodes with weighted edges representing dependencies. Modules with high "debt rank" scores—due to centrality or frequent modification—are surfaced first for remediation. Learning-to-rank approaches can refine ranking using supervised signals (e.g., historical fix order) to optimize prioritization [1].

Data preparation and input formats. Transforming raw software artifacts into meaningful inputs for generative AI begins with careful data curation. The process starts by sanitizing source code, configuration files, and architecture descriptions—removing irrelevant comments, standardizing naming conventions, and stripping out proprietary or non-essential content to reduce noise. Once cleaned, the codebase is segmented into coherent units – such as individual classes, modules, or microservices – so that each unit preserves contextual integrity and conforms to the token limits of target AI models. Each segment is then enriched with metadata annotations: cyclomatic complexity scores reveal branching hotspots, duplication percentages highlight clone-related debt, and dependency counts map coupling intensity. These annotations guide the AI's focus toward areas most in need of attention. Finally, to provide historical and rationale context, each code snippet is packaged alongside related artifacts – documentation excerpts, commit messages, or issue tracker entries – helping the model understand why certain shortcuts were taken and how debt accumulated over time.

Embedding techniques. Translating code and architecture elements into continuous vector spaces enables AI models to understand syntactic and semantic relationships: (1) *CodeBERT and GraphCodeBERT*: pretrained on large code corpora, these transformer-based models produce token embeddings that capture local syntax and some semantic patterns. Graph-CodeBERT further incorporates data flow and control flow graphs, enriching embeddings with structural context [3]. (2) *AST-based embeddings*: by applying tree-based convolutional networks or graph neural networks directly to ASTs, embeddings can encode fine-grained syntactic patterns and variable scopes. These embeddings are particularly useful for capturing nuanced debt indicators like nested complexity or unconventional control structures. (3) *Hybrid embeddings*: combining token-level embeddings from CodeBERT with graph embeddings from dependency graphs or call graphs yields richer representations that fuse code semantics with architectural context, enabling generative AI to reason across multiple abstraction levels [3].

4. Conclusions

This paper has examined the multifaceted process of technical debt management—from its measurement and representation to leveraging generative AI for intelligent remediation. By defining clear, quantitative code and architectural metrics and integrating them with qualitative assessments, we establish a solid foundation for identifying and prioritizing debt hotspots. Data structures such as dependency graphs and ASTs enable rich modeling of software artifacts, while machine learning algorithms classify, cluster, and rank debt items based on their complexity, coupling, and historical maintenance data.

References

[1] Holvitie, J., Leppänen, V., Hyrynsalmi, S.: Technical Debt and the Effect of Agile Software Development Practices on It - An Industry Practitioner Survey. In: Proceedings of 6th International Workshop on Managing Technical Debt, Canada, pp. 35–42, IEEE (2014)

[2] Holvitie, J., Licorish, S.A., Spínola, R.O., Hyrynsalmi, S., MacDonell, S.G., Mendes, T.S., Buchan, J., Leppänen, V.: Technical debt and agile software development practices and processes: An industry practitioner survey. In: Information and Software Technology, Vol. 96, pp. 141–160 (2018)

[3] Ramasubbu, N., Kemerer, C.F.: Integrating Technical Debt Management and Software Quality Management Processes: A Normative Framework and Field Tests. In: IEEE Transactions on Software Engineering, Vol. 45(3), pp. 285–300 (2019)