International Journal of Science and Research (IJSR)
ISSN: 2319-7064
SJIF (2022): 7.942

SageMinder: AI-Driven Technical Debt Reduction in AWS-Based Data Science Pipelines

Sai Tarun Kaniganti

Abstract: ***This paper focuses on the difficulty of managing technical debt in large-scale software projects that use AI and Machine Learning for identification and resolution. The approach consists of systematic surveillance of technical debts, gradualistic refinancing techniques, emphasis on sustaining high-quality standards, and automatic tools for detecting technical debt. When implemented in the development lifecycle, such strategies will improve the software projects' stability and sustainability in the long run.***

Keywords: Technical Debt, Artificial Intelligence, Machine Learning, Amazon Web Service, Software Engineering, Debt, Code Quality, DevOps, Data Science, Cloud Computing

1. Introduction

Technical debt is quite popular in software development, mainly when specialists have worked on the project for many years. With long-term development, plenty of shortcuts and bad practices can be introduced. This buildup of debt can significantly influence sustainability and its ability to grow and evolve; thus, software systems' complex, inefficient, and expensive characteristics sunk costs. With time and the size of such projects, technical debt becomes a big issue that development teams must consider in their endeavours to provide tremendous and agility-designed software solutions.

A good solution for TD is to have a set of rules that would allow it to manage the current technical debt and the future one. In this respect, any contemporary methods, including AI and ML, can be regarded as ways to improve the identification and management of technical debt. Utilizing the opportunity AI and ML, the development team can enhance the identification of issues, establish hierarchies of debt repayment, and incorporate the continuous monitoring process into development. They offer state-of-the-art approaches to code quality analysis and system performance evaluation and ways to improve processes and, thus, enhance the vision-making process.

The following paper reviews different approaches to dealing with TD when projects have a long duration, mainly about AI and ML for enhancing debt recognition and management. Thus, by analyzing methods like code smell detection employing NLP, performance metrics anomaly detection, or code's complexity predictive analysis, the paper attempts to demonstrate the advantages of integrating AI technologies into tech debt management strategies. Therefore, by implementing these new ways of managing technical debt, organizations can avoid significant problems, which could create software production and attain more stable and efficient software systems.

Figure 1: Software development

Understanding Technical Debt

On the other hand, technical debt can be described as the cost incurred because of the extended time frame on the project due to the use of a poor approach that a better approach could have solved, although it is time-consuming (Spínola et al., 2019). They experience a higher cost to maintain such debt as it accumulates over long periods to decrease development speed and threaten the system's stability.

Types of Technical Debt

Code Debt

Concerns the formation of a low-quality and suboptimal code structure created because of shortcuts during code development (Gradišnik & Hericko, 2018). This kind of debt results in code that works but needs more design and comments and is challenging to maintain. For example, some literal could be used instead of the configuration and constant files, which is not viable. Code duplication is a scenario that arises when, instead of reusing code, similar procedures are written over and over again instead of encapsulated into functions. It's also typical to see poor organizations with many functions or classes performing multiple actions rather than using the single responsibility principle.

Architectural Debt

Originates from issues with the high-level design of a system, it affects the manner of scaling up, execution speed, and ease of upkeep. This type of debt might present as components heavily dependent on other components, which complicates changing or even expanding a part of the system without affecting other parts. Such a monolith architecture may also

lead to Architectural Ros, which is when one big application has tightly coupled components and faces problems in terms of scalability and deployment. Also, low SC arises from general and architectural solutions that will not allow it in the future, for instance, horizontal scaling or proper data management (Nasir et al., 2022).

Documentation Debt
Comes up when there is inadequate documentation, especially when documentation is not proportional to the changes in code or is lacking totally. It can cause issues with comprehension, care, and training. Some examples include current API documentation that may need to be updated regarding the API's exposure, thus causing confusion and errors. Sometimes, design documents need to adequately describe the system's architecture da, flow/interactions, or integration points (Glumich et al., 2018). In other cases, missing commentaries may represent critical parts of the code's logic or rationale.

Test Debt
Happens when testing practices fail because there are not enough tests, no automatized tests or low test quality. This can lead to different bugs being missed in the studies and the general confidence in the software's reliability being lowered (Rodríguez-Pérez et al., 2018). For instance, the absence of unit tests may mean that only some segments, such as individual components or functions, are not tested. This may occur when different components or services are not adequately tested for integration or when the tests are conducted in a way that might give inconsistent results, thus making them unbeneficial to quality analysis.

Infrastructure Debt
Concerns are tied to deployment and development environments that impact efficiency, reliability, and costs (Baier et al., 2019). Old deployment pipelines are from the CI/CD progression, which is time-consuming, error-prone, or needs up-to-date automation. Mismanagement of resources implies over-allocation, over-allocation, or under-allocation of resources in the cloud; hence, it sounds like an expense or lack of IT capacity. Also, only doing deployment, scaling, or monitoring through manual operations creates a lot of room for errors and ineffectiveness.

Security Debt
Stem from failing to adhere to security best practices, resulting in weaknesses and threats. This can consist of the sequence strings and application credential codes such as passwords or API codes. Another one is the absence of data encryption during transmission and storage. Moreover, installing dependencies that have not been updated for some time and are exposed have specific vulnerable points that lead to security debt (Pashchenko et al., 2020).

Compliance Debt
Takes place every time the software does not conform to the current legislation or industrial standards, resulting in legal and financial consequences. This emerging risk category includes data privacy infringements, such as failure to meet data protection laws that include GDPR or CCPA. Lack of audit trails also falls under compliance debt; proper logging and tracking mechanisms must be included, and non-standard coding practices come in where industry regulations are disregarded.

Solving such technical debts requires a strategic approach to ensure that the software stays flexible, easy to manage, and secure throughout its life cycle.

Strategies for Managing Technical Debt

1) Continuous Identification and Monitoring
Continuously check code bases and designs for signs of code debt. Use static code analysis tools and organize periodic architecture reviews to stay aware of the system's health state.

2) TD Prioritization Model
It is thus essential to identify which technical debt to pay first when it comes to software projects. This framework should be designed to manage technical debt in interactions that improve the performance, stability and efficiency of the systems and the development processes (Lenarduzzi et al., 2021).

The following key factors should be considered:

Also entails assessing the impact of technical debt on the speed, the efficiency of the system, and the system's ability to respond to end-user needs. Worst-case, debt hampers performance, response time, resource consumption, usability and, in general, the quality of the application. As for assessing this kind of impact, one can use response time that reflects response times or latency for crucial functions, throughput that estimates the system's performance in handling concurrent requests or data processing, and resource consumption that quantifies the CPU, memory, and storage usage. Such debt should be targeted because it harms the users and worsens the system's performance.

Risk to System Stability
CBT requires an evaluation of its consequences in introducing risks that may lead to crashes, data corruption, and other crucial failures. Minimizing exposure to credit risk and finding out which debts, if left unpaid, could destabilize the system is vital. These are the error rate, which measures the rate of errors or exceptions that can be attributed to the technical debt; the failure rate, which considers the possibility of system failure or shutdown due to accumulation of the technical debt; and the rate of recovery which determines the extent to which debt has impact of the system's recovery in cases of failure or shut down. High-risk debts should, therefore, be prioritized to avoid failure on hazardous equipment and keep the operations going.

Effect on Development Velocity
Looks at how technical debts influence the development team's efficiency due to the extended time needed to deliver new functionality or resolve issues (Holvitie et al., 2018). The key metrics that must be measured are the time to provide new features that are consumed because of the feature debt, the time taken to rectify bugs and issues that have been made worse by the debt, and code complexity, which results from debt with the code base becoming hard to comprehend and modify. Debt that slows down the development velocity to a

large extent should be managed to optimize the delivery of new features.

Finally, the cost of repayment, which states the price that is incurred if a business delays its repayment:
It entails analyzing the costs of resolving the debt against the potential consequences of not so doing. A few examples are the repayment cost, which puts an approximation on the time and resources required to clear the outstanding; the delay cost of estimating what effect the outstanding will have on system predictability and development velocity; and the opportunity cost, or what someone loses, in terms of competitive advantage and other missed opportunities when they do not clear the outstanding. Prioritize debt where the benefits of repaying it are justified by performance. Therefore, choose an approach such as a cost-benefit analysis to prioritize debt for repayment based on the costs of delay.

Business Effects and Stakeholder Issues
Should be included in the conceptual prioritization framework because the satisfaction of obligations has to be reconciled with the achievement of business objectives and relevant stakeholders' attention. Metrics include customer impact, which measures technical debt's impact on a customer base satisfaction and retention, as well as its effect on total revenues; regulatory implications, which gauges the level of debt's compliance with regulatory requirements or standards; and stakeholder feedback, in which relevant stakeholders rank the importance and urgency of particular debt about others. Pay attention to high-value business debts or debts that are particularly important to stakeholders and customers to reflect the company's strategy and meet the marketplace needs.

Long-Term Sustainability
Concerns analyzing the impact of TD on the system's capacity to be upgraded and modified with the changes in requirements (Chen et al., 2020). Thus, the potential measures that should be taken into consideration include maintainability, which estimates how debt influences the possibility of further alteration and extension of the system; scalability, which determines how debt influences the possibility of additional enlargement of the system in the future; and technical flexibility, which express how debt influences the adaptability of the system to the new technologies or new requirements. Debt repayment is an order that threatens to destabilize the business in the long run, to keep the structure as flexible as possible.

2. Incremental Repayment

Integrate debt repayment into the regular development cycle. Allocate a percentage of each sprint or development phase to addressing identified technical debt.

Table 1: Key Aspects and Strategies for Incremental Debt Repayment

Aspect	Description	Details
Defining Debt Repayment Goals	Establish clear goals and priorities for addressing technical debt.	Assess impact on performance, stability, and velocity. Set measurable objectives (e.g., reducing code smells, improving coverage). Regularly review and adapt goals.
Allocating Sprint Resources	Dedicate a percentage of resources to debt repayment each sprint or phase.	Allocate a portion of sprint capacity (e.g., 20%) for debt-related tasks. The remaining capacity (e.g., 80%) focuses on new features and bug fixes.
Prioritizing Debt Items	Use a prioritization framework to determine which debt items to address first.	Consider impact on performance, risk to stability, and effect on velocity. Regularly update priority based on assessments and feedback.
Monitoring and Reporting	Track progress and report outcomes on debt repayment efforts.	Monitor number of resolved debt items, code quality metrics, and system performance changes. Use sprint reviews or retrospectives for reporting and adjusting strategies.
Continuous Improvement	Refine debt repayment processes based on feedback and results.	Gather feedback from teams and stakeholders. Adjust resource allocation, prioritization criteria, and debt detection tools. Evaluate and improve strategies continuously.

3. Education and Culture

Managing technical debt means organizations must champion a culture that improves code quality and characteristics associated with lasting sustainability. It conceptually implies a process of integrating the principles of technical debt management into the team's beliefs and behaviours. Thus, the mindful approach to writing the code and paying attention to technical debt helps adopt a positive outlook when it comes to not accumulating the debt in the first place. It is recommended that training sessions and workshops be conducted regularly to enable the team members to gain knowledge on issues regarding code quality, including, but not limited to, standards of coding, code review procedures, and the need and procedure for code testing.

Discussions of technical debt within team meetings and retrospectives can keep it regularly at the top of the mind. Thus, the more an organization appreciates and compensates for creating value that aims to decrease technical debt metrics, the more it conveys the importance of such actions. Levelling out steps to ensure promotion, tolerance, and culture, change improvement to enhance the present quality of code and nurture a tactic plan for the long-term management of technical debt (Yang et al., 2023).

Figure 2: Technical Debt and How to Manage It

4. Automated Debt Detection

Use AI and ML patterns and best practices to identify possible issues causing technical debts. This can include:

The use of AI and ML to automatically identify technical debt can increase the performance of managing debt in software projects (Pandi et al., 2023). By using such advanced technologies, teams can analyze possible problems, minimize extensive scrutiny and concentrate on the high-risk debt segments. Several techniques illustrate the potential of AI and ML in automating debt detection:

Identification of the Code Smell with the Help of NLP

Some of the code smells may result in the formation of technical debt. Such smells can be identified with the help of NLP by analyzing the code comments, documentation, and actual code (Kokol et al., 2021). Such includes code comment analysis that entails using NLP techniques to analyze comments and documentation to look for areas of weakness that might indicate less than optimal code quality descriptions. In addition, conventional techniques of NLP can also be taught the most proven attributes of more frequent code smells like lengthy methods, a large number of global variables, or inadequate naming standards. To achieve this, the code is divided into tokens, and the tokens' syntactic analysis is performed before applying machine learning algorithms to identify the violations. For example, CodeNLP has a localized NLP approach to analyze the code comments to find the divergence between the documentation and the code or the code smell.

Figure 3: Understanding Code Smell Detection in Software Development

Digging into System Performance Metrics for Anomalies

The patterns of performance metrics can be analyzed using anomaly detection techniques, thus pointing towards the technical debt (Conejero et al., 2018). This process includes time series analysis, typically involving algorithms that monitor the performance over time to identify patterns of deviation from the standard. Measures like the response time, the rate of occurrences of errors, usage of CPU and other resources are then checked for any abnormality. Where TFQ has not been calculated, numeric techniques set threshold limits to identify unusual behaviour that could be put down to technical debt; the methods used include z-score or moving average. Other machine learning algorithms like Isolation Forests, Autoencoders, and Long Short-Term Memory (LSTM) networks can identify different patterns and outliers within the data that affect the performance. For example, Prometheus, alongside Grafana, could work with ML-based anomaly detection services showing the real-time status of performance issues.

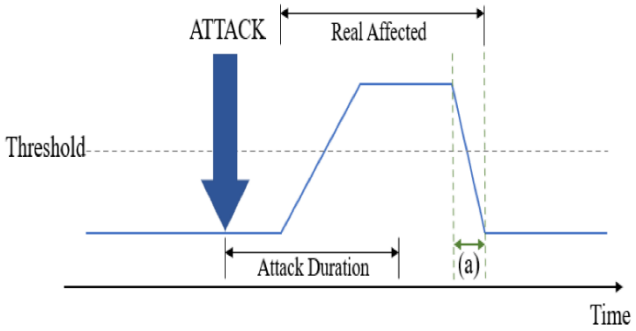


Figure 4: A Study on Performance Metrics for Anomaly Detection Based on Industrial Control System Operation Data

Code Complexity and Maintainability Index Prediction

Predictive analysis refers to employing algorithms along with machine learning to calculate future difficulties in complexity and maintainability by referring to current measurements of the code and the archive (Vallim Filho et al., 2022). This technique uses complexity measures involving cyclomatic complexity, code churn, and class coupling to identify the probable areas that will be an issue. The complexity of code can be predicted using machine learning models based on the historical data collected. In addition, the future maintainability of codes can be checked using ML algorithms by using historical maintainability data and code changes, whereby one can use regression or a classification method to determine probable regions that would require frequent maintenance. Systems like CodeClimate or SonarQube, which use ML algorithms, analyze the code's complexity and maintainability, thus helping the teams identify the code to be refactored.

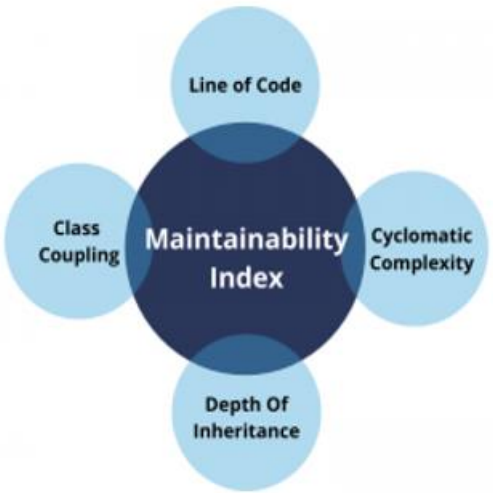


Figure 5: A Simple Understanding of Code Complexity

Automated Code Review Systems

Automated code review points to the use of AI and ML in assessing code changes and the presence of problems. These systems combine the static code analysis tools, which are AI-based, by recognizing the code smells about vulnerabilities

and other deviations from set coding standards (Hamfelt, 2023). The different AI models can understand patterns related to low code quality. Secondly, dynamic analysis tools work during the code execution to look for some problems that static analysis tools cannot find. ML models then analyze the execution pattern to determine which areas are problematic. For example, GitHub Copilot or Amazon Code Guru apply artificial intelligence solutions to provide code recommendations and detect code review problems to help developers optimize source code (Sarkar et al., 2022).

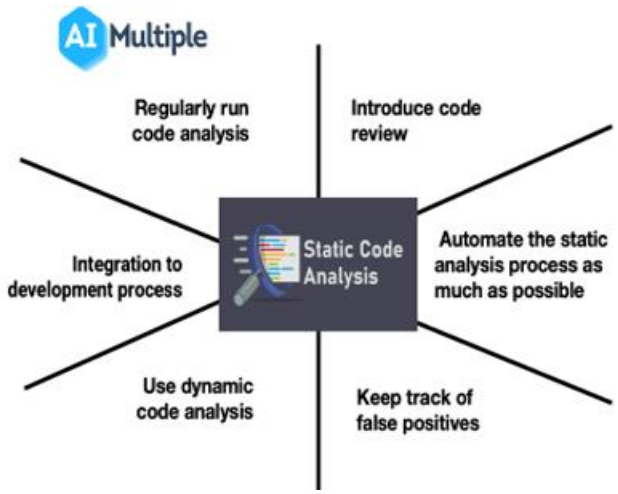
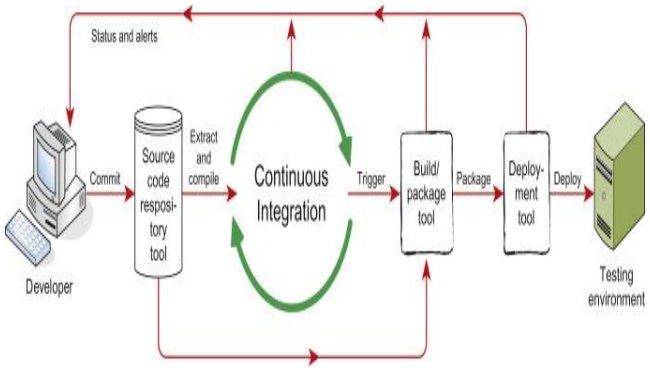


Figure 6: Static Code Analysis Best Practices

CI/CD Integration Continuous Integration and Deployment

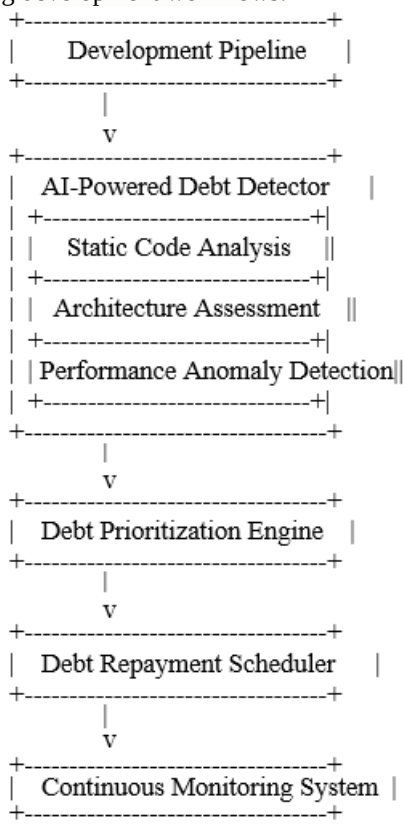
Combining automated debt detection with CI and CD processes implies that debt detection is continuous instead of a manual effort conducted once (Biazotto et al., 2023). This integration entails the utilization of automated testing frameworks that employ AI to consider claims on probable technical debt during each build or deployment process, including unit tests, integration tests, and performance tests. Integration of some form of reporting system within CI/CD pipelines to report and track technical debt put into CI/CD pipeline, issues suspected of technical debt during the automated tests ensure that technical debts are priced and solved before becoming a cost to the business. Hence, with popular technologies like Jenkins or GitLab CI, the outcome of code analysis can be provided instantly and, combined with AI-based tools, revealing the issue of technical debt.



By applying the suggested AI and ML approaches to automated debt detection, companies can advance their capacities for technical debt identification and remediation, promoting the development of more sustainable, flexible, and efficient software systems.

Proposed Architecture for Technical Debt Management

To effectively manage technical debt in long-term projects, we propose an architecture that integrates AI/ML capabilities with existing development workflows:



This architecture incorporates the following components:

1) AI-Powered Debt Detector: Utilizes machine learning models to analyze code, architecture, and system performance for potential debt indicators.
2) Debt Prioritization Engine: Applies predefined rules and ML-based recommendations to prioritize identified debt items.
3) Debt Repayment Scheduler: Integrates with project management tools to schedule debt repayment tasks alongside feature development.
4) Continuous Monitoring System: Tracks debt metrics over time and provides insights on the effectiveness of repayment efforts.

Table 2: Components and Features of the Technical Debt Management Architecture

Component	Function	Key Features
AI-Powered Debt Detector	Utilizes ML models to analyze code, architecture, and performance for identifying technical debt.	- Code Analysis: Detects code smells, complexity, maintainability.
		- Architecture Assessment: Identifies design flaws and scalability issues.
		- Performance Monitoring: Detects anomalies in performance metrics.
		- Integration: Works with version control systems and build pipelines.
Debt Prioritization Engine	Applies rules and ML-based recommendations	- Rule-Based Prioritization: Ranks debt items by impact on performance, stability, and development velocity.

	to prioritize debt items.	- ML Recommendations: Predicts long-term impact and refines prioritization.
		- Customizable Metrics: Aligns prioritization with project goals. - Integration: Syncs with project management tools.
Debt Repayment Scheduler	Schedules and allocates resources for debt repayment tasks alongside feature development.	- Sprint Integration: Allocates a portion of each sprint to debt repayment.
		- Resource Allocation: Manages developer time for debt tasks.
		- Task Management: Coordinates with task tracking systems.
		- Feedback Mechanism: Adjusts scheduling based on progress and feedback.
Continuous Monitoring System	Tracks debt metrics over time and provides insights on debt repayment effectiveness.	- Metric Tracking: Monitors debt-related metrics (complexity, test coverage, performance).
		- Effectiveness Analysis: Analyzes impact of repayment activities.
		- Reporting and Alerts: Provides updates on debt management progress.
		- Adaptive Insights: Recommends improvements based on historical data.

Leveraging AI and ML for Technical Debt Management

Artificial intelligence and machine learning offer significant potential for enhancing technical debt management processes. Here are some specific applications:

1. Automated Code Review

Implement ML models trained on large codebases to identify potential code smells and suggest improvements. This can be integrated into the development workflow through IDE plugins or code review tools.

Example code snippet for a simple code smell detector using Python and scikit-learn:

```
python
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
import numpy as np

# Training data (simplified example)
code_samples = [
    "def function():\n    pass",
    "def long_function():\n    # Many lines of code...",
    # ... more samples ...
]
labels = [0, 1]  # 0: Good, 1: Potential smell

# Create feature vectors
vectorizer = TfidfVectorizer(token_pattern=r'\b\w+\b')
X = vectorizer.fit_transform(code_samples)

# Train classifier
clf = MultinomialNB()
clf.fit(X, labels)

# Function to predict code smells
def predict_code_smell(code):
    X_new = vectorizer.transform([code])
    return clf.predict(X_new)[0]

# Example usage
new_code = "def very_long_function():\n    # Lots of complex logic..."
if predict_code_smell(new_code):
    print("Potential code smell detected!")
```

2. Architecture Debt Detection

Determining possible technical debt can be automated using computer learning techniques, especially the ML models applied to system architecture. These models can be trained to learn from good examples and examples of adverse outcomes. Thus, these models are capable of identifying flaws and inefficient structures. Suppose the models superficially calculate aspects like modularity, scalability, and coupling. In that case, PF can identify design patterns that will result in future technical debt, such as high interdependence and improper data flow design patterns (Varga, 2018). This proactive detection contributes to solving architectural problems at the early stages of development when, for instance, such issues do not turn into severe ones.

When it comes to implementing the ML-based architecture analysis, these generated models have to be connected to the design review tools and the CI/CD pipelines. The practical work of consumer committees provides an opportunity for constant assessment of architectural changes and immediate identification of possible problems in the field of debt. There is also the capacity to look at the architecture and how the enemy's strategies can enhance system design when historical and real-time data is collected. This approach guarantees that architecture is adequately preserved and that technical debt is dealt with properly, creating more 'architecturally sound' systems (Soliman et al., 2021).

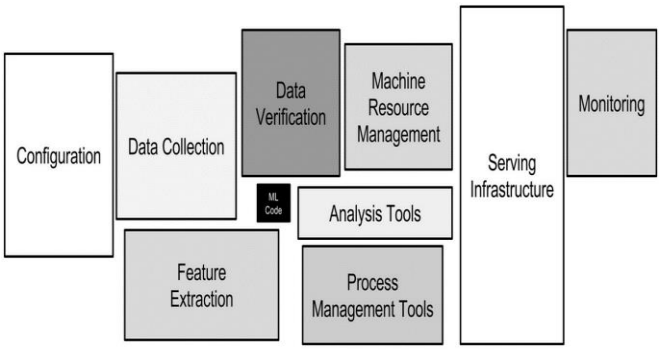


Figure 7: Technical Debt in ML systems

3. Performance Anomaly Detection

Use time series analysis and anomaly detection algorithms to identify potential performance-related technical debt by monitoring system metrics over time.

Example using Python and Facebook's Prophet library for time series forecasting:

```
python
from fbprophet import Prophet
import pandas as pd
```

```
# Load historical performance data
df = pd.read_csv('performance_metrics.csv')
df['ds'] = pd.to_datetime(df['date'])
df['y'] = df['response_time']

# Create and fit the model
model = Prophet()
model.fit(df)

# Make future predictions
future = model.make_future_dataframe(periods=30)
forecast = model.predict(future)

# Identify anomalies
threshold = 1.5  # Standard deviations
anomalies = forecast[abs(forecast['yhat'] - forecast['y']) > threshold * forecast['yhat'].std()]

print("Potential performance anomalies detected:")
print(anomalies[['ds', 'y', 'yhat']])
```

Case Study: Managing Technical Debt in an AWS-based Data Science Project

To illustrate the application of these principles in a real-world scenario, let's consider a hypothetical long-term data science project deployed on Amazon Web Services (AWS).

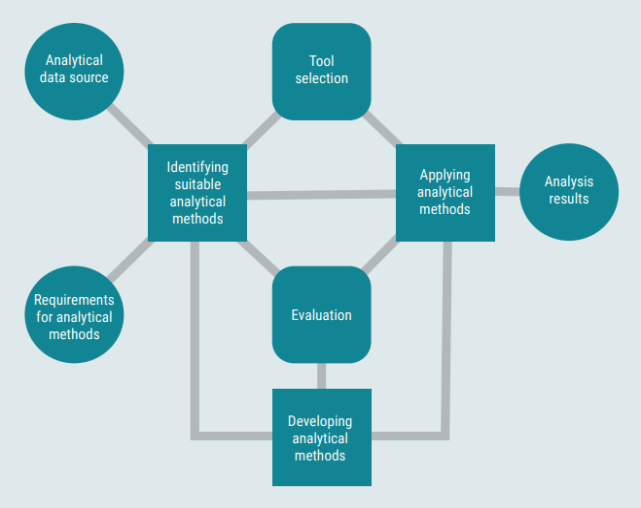


Figure 8: Applying a Data Science Process Model to a Real-World Scenario

Project Overview

The area is the huge Machine Learning data pipeline that will be processing terabytes of data each day. This pipeline consists of several components: data ingestion, data preprocessing, data feature engineering, model training, and finally, prediction service through a RESTful API. It is implemented to perform large computations to provide solutions and forecasts for critical applications in the business environment. Due to the broad usage and evolution of the system, the existing technical dept has grown, which applies pressure to the system, which affects its performance, evolution and scalability.

The technical debt appears in different shapes and forms, such as old architectures of ML models, ineffective data preparation and processing workflows, non-uniform code styling, and unused AWS services. These issues have gradually resulted in increased maintenance overhead, slower development cycles, and performance degradation. Over time, with the change in requirements, eradicating this technical debt has emerged as necessary for improved and effective system functionality. The project aims to develop and establish an all-encompassing technical debt management plan to improve the system's effectiveness by utilizing contemporary instruments and approaches.

Identified Technical Debt

1) Outdated ML model architecture
2) Inefficient data processing pipelines
3) Lack of comprehensive unit and integration tests
4) Inconsistent code style and documentation
5) Overprovisioned and underutilized AWS resources

Table 3: Identified Technical Debt and Suggested Actions

Technical Debt	Description	Impact	Suggested Action
Outdated ML Model Architecture	Use of deprecated or inefficient model architectures in machine learning systems.	Decreased performance, limited scalability, and maintainability issues.	Update or refactor model architecture to utilize modern frameworks and techniques.
Inefficient Data Processing Pipelines	Data pipelines that are slow, poorly optimized, or difficult to maintain.	Increased processing time, higher costs, and potential data quality issues.	Optimize pipelines, adopt modern processing frameworks, and improve pipeline design.
Lack of Comprehensive Unit and Integration Tests	Insufficient testing coverage for code and system integration.	Increased risk of bugs, lower code quality, and higher maintenance costs.	Develop and implement comprehensive unit and integration tests to improve reliability.
Inconsistent Code Style and Documentation	Variability in coding practices and documentation standards across the codebase.	Reduced readability, maintainability issues, and increased onboarding time.	Standardize code style and documentation practices, and enforce them through tools and guidelines.
Overprovisioned and Underutilized AWS Resources	AWS resources that are provisioned beyond current needs or not fully utilized.	Increased costs and inefficient resource management.	Perform resource optimization, scale resources according to actual needs, and implement cost management practices.

Debt Management Strategy

1) Implement automated code review using Amazon CodeGuru
2) Refactor ML pipeline using AWS Step Functions for better orchestration
3) Introduce AWS DevOps tools (CodeBuild, CodePipeline) for CI/CD

4) Utilize Amazon SageMaker for model versioning and deployment
5) Implement infrastructure-as-code using AWS CloudFormation

Table 4: Debt Management Strategy and Implementation

Debt Management Strategy	Description	Expected Benefits	Implementation Steps
Implement Automated Code Review Using Amazon CodeGuru	Use Amazon CodeGuru to automatically review code, detect issues, and provide recommendations.	Improved code quality, early detection of issues, and reduced manual review effort.	1. Integrate CodeGuru with your version control system. 2. Configure analysis settings and review results. 3. Act on recommendations to improve code quality.
Refactor ML Pipeline Using AWS Step Functions for Better Orchestration	Redesign data processing pipelines with AWS Step Functions to improve orchestration and management.	Enhanced pipeline reliability, better scalability, and easier maintenance.	1. Assess current pipeline architecture. 2. Design and implement new pipeline workflows using Step Functions. 3. Test and deploy the refactored pipeline.
Introduce AWS DevOps Tools (CodeBuild, CodePipeline) for CI/CD	Implement CI/CD pipelines using AWS CodeBuild and CodePipeline to automate build, test, and deployment processes.	Streamlined development workflow, faster delivery of features, and improved code quality.	1. Set up CodeBuild projects for building and testing code. 2. Create CodePipeline workflows for automating deployments. 3. Monitor and optimize CI/CD pipelines.
Utilize Amazon SageMaker for Model Versioning and Deployment	Leverage Amazon SageMaker to manage model versioning, training, and deployment processes.	Efficient model management, improved deployment practices, and streamlined training workflows.	1. Set up SageMaker environments for model training and deployment. 2. Implement versioning and monitoring of models. 3. Integrate SageMaker with existing ML workflows.
Implement Infrastructure-as-Code Using AWS Cloud Formation	Use AWS CloudFormation to define and manage infrastructure using code, ensuring consistency and scalability.	Simplified infrastructure management, improved reproducibility, and automated provisioning.	1. Develop CloudFormation templates for infrastructure components. 2. Deploy and manage resources using these

AI/ML-Enhanced Debt Management

a) Use Amazon Comprehend to analyze code comments and documentation for clarity and completeness
b) Implement custom ML models to predict potential performance bottlenecks based on code changes and infrastructure metrics
c) Leverage Amazon Forecast to optimize resource allocation and reduce infrastructure debt

Example CloudFormation template snippet for defining an optimized SageMaker endpoint:

```
yaml
Resources:
  OptimizedEndpoint:
    Type: "AWS::SageMaker::Endpoint"
    Properties:
      EndpointName: !Ref EndpointName
      EndpointConfigName: !GetAtt
EndpointConfig.EndpointConfigName

  EndpointConfig:
    Type: "AWS::SageMaker::EndpointConfig"
    Properties:
      ProductionVariants:
        - InitialInstanceCount: 1
          InstanceType: "ml.t2.medium"
          ModelName: !Ref ModelName
          VariantName: "AllTraffic"
      DataCaptureConfig:
        EnableCapture: true
        InitialSamplingPercentage: 100
        DestinationS3Uri: !Sub
"s3://${DataCaptureBucket}/endpoint-data-capture"
        CaptureOptions:
          - CaptureMode: Input
          - CaptureMode: Output
```

5. Conclusion

Applying the concept of technical debt in long-term projects requires active and strategic management. The combination of AI and ML can be considered a secure approach to mitigate and prioritize technical debt to prevent it from deteriorating the project's outcome in the future. The described debt management architecture, including AI debt detection and the Prioritization component, presents a clear concept of introducing debt management into the development process. This makes it easier to keep software projects healthy and sustainable because debt is addressed, and the overall quality of code is gradually enhanced.

The example of the case with a data science project based on AWS infrastructure is informative regarding applying these principles. If adequately harnessed by utilizing tools such as cloud-native systems and AI/ML capabilities, this advantage can lead to enhanced systems performance, maintenance, and scalability. Such steps as automated code reviews to refactoring ML pipelines show that managing technical debts can improve the appropriation of project results and the shift of existing capacities in the context of systems development.

This is why ongoing surveillance and selective extinguishment of debts are relevant to guarantee long-term

project flexibility. Suppose these three practices are incorporated into the development cycle. In that case, adequate measures to help teams manage technical debt are provided, enabling them to remain more flexible regarding changed requirements or new technologies. Besides preserving the project's overall health, this strategy also prepares teams for further successful work in the context of constant technological evolution.

References

[1] Spínola, R. O., Zazworka, N., Vetro, A., Shull, F., & Seaman, C. (2019). Understanding automated and human-based technical debt identification approaches-a two-phase study. *Journal of the Brazilian Computer Society*, *25*, 1-21.

[2] Gradišnik, M. I. T. J. A., & Hericko, M. (2018). Impact of code smells on the rate of defects in software: A literature review. In *CEUR Workshop Proceedings* (Vol. 2217, pp. 27-30).

[3] Nasir, M. H., Arshad, J., Khan, M. M., Fatima, M., Salah, K., & Jayaraman, R. (2022). Scalable blockchains—A systematic review. *Future generation computer systems*, *126*, 136-162.

[4] Glumich, S., Riley, J., Ratazzi, P., & Ozanam, A. BP: Integrating Cyber Vulnerability Assessments Earlier into the Systems Development Lifecycle. In *2018 IEEE Secure Development Conference*.

[5] Rodríguez-Pérez, G., Robles, G., & González-Barahona, J. M. (2018). Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm. *Information and Software Technology*, *99*, 164-176.

[6] Baier, L., Jöhren, F., & Seebacher, S. (2019, June). Challenges in the Deployment and Operation of Machine Learning in Practice. In *ECIS* (Vol. 1).

[7] Pashchenko, I., Vu, D. L., & Massacci, F. (2020, October). A qualitative study of dependency management and its security implications. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security* (pp. 1513-1531).

[8] Lenarduzzi, V., Besker, T., Taibi, D., Martini, A., & Fontana, F. A. (2021). A systematic literature review on technical debt prioritization: Strategies, processes, factors, and tools. *Journal of Systems and Software*, *171*, 110827.

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

[10] Chen, S., Liang, Y. C., Sun, S., Kang, S., Cheng, W., & Peng, M. (2020). Vision, requirements, and technology trend of 6G: How to tackle the challenges of system coverage, capacity, user data-rate and movement speed. *IEEE Wireless Communications*, *27*(2), 218-228.

[11] Yang, Y., Verma, D., & Anton, P. S. (2023). Technical debt in the engineering of complex systems. *Systems Engineering*, *26*(5), 590-603.

[12] Pandi, S. B., Binta, S. A., & Kaushal, S. (2023). Artificial intelligence for technical debt management in software development. *arXiv preprint arXiv:2306.10194*.

[13] Kokol, P., Kokol, M., & Zagoranski, S. (2021). Code smells: A synthetic narrative review. *arXiv preprint arXiv:2103.01088*.

[14] Conejero, J. M., Rodríguez-Echeverría, R., Hernández, J., Clemente, P. J., Ortiz-Caraballo, C., Jurado, E., & Sánchez-Figueroa, F. (2018). Early evaluation of technical debt impact on maintainability. *Journal of Systems and Software*, *142*, 92-114.

[15] Vallim Filho, A. R. D. A., Farina Moraes, D., Bhering de Aguiar Vallim, M. V., Santos da Silva, L., & da Silva, L. A. (2022). A machine learning modeling framework for predictive maintenance based on equipment load cycle: An application in a real world case. *Energies*, *15*(10), 3724.

[16] Hamfelt, P. (2023). MLpylint: Automating the Identification of Machine Learning-Specific Code Smells.

[17] Biazotto, J. P., Feitosa, D., Avgeriou, P., & Nakagawa, E. Y. (2023). Technical debt management automation: State of the art and future perspectives. *Information and Software Technology*, 107375.

[18] Varga, E. (2018). *Unraveling Software Maintenance and Evolution*. Springer International Publishing.

[19] Soliman, M., Avgeriou, P., & Li, Y. (2021). Architectural design decisions that incur technical debt—An industrial case study. *Information and Software Technology*, *139*, 106669.