

ModelForge-AI @ An Intelligent Machine Learning Code Analysis, Optimization and Deployment Assistant

Dr. M.K. Jayanthi Kannan¹, Mukul Mishra², Akshara Singh³, Ananya Chaurasia⁴, Akshara Srivastava⁵,
Akshat Srivastava⁶, Nilabh Sharma⁷

¹*Professor, VIT Bhopal University, Bhopal-Indore Highway, Kothrikalan, Sehore, Madhya Pradesh*
^{2,3,4,5,6,7}*B.Tech., Students, School of Computing Science and Engineering, VIT Bhopal University, Bhopal-Indore Highway, Kothrikalan, Sehore, Madhya Pradesh*

Abstract—Machine learning applications increasingly depend on complex data-processing, model-training, and evaluation pipelines, making the maintainability and reliability of ML code an important engineering concern. However, existing development tools generally address individual aspects of the ML lifecycle, such as conventional code linting, hyperparameter optimization, model explainability, or deployment, without providing a unified understanding of the complete ML workflow. This paper presents ModelForge AI, a proposed intelligent platform for context-aware analysis, optimization, explainability, and deployment assistance for Python-based machine learning projects. The proposed approach uses Python Abstract Syntax Tree (AST) parsing to identify ML-related operations and construct a structured pipeline model representing stages such as data loading, preprocessing, feature engineering, data splitting, model training, and evaluation. A context-aware rule engine is then designed to identify potential ML-specific issues, including data leakage, inappropriate preprocessing order, missing validation, unsuitable evaluation practices, and reproducibility concerns. To improve the usability of detected findings, a Large Language Model (LLM), accessed through the Gemini API or locally using Ollama, is proposed to generate contextual explanations and improvement suggestions grounded in deterministic analysis results. The platform further incorporates optional Optuna-based hyperparameter optimization, SHAP-based model explainability, and automated generation of FastAPI and Docker deployment artefacts. The key contribution of the proposed framework is the integration of these capabilities through a shared ML pipeline representation, enabling a continuous workflow from code analysis to model improvement and deployment preparation. The current work focuses on system design, requirements, architecture, and conceptual methodology, while implementation and empirical evaluation are planned for the subsequent phase.

Index Terms—Machine Learning Code Analysis, Context-Aware Static Analysis, ML Code Smells, Abstract Syntax Tree, Data Leakage Detection, Large Language Models, Hyperparameter Optimization, Explainable AI, ML Pipeline Analysis, Automated ML Deployment.

I. INTRODUCTION

Machine learning has become an important part of many modern applications, but developing an ML project involves several connected stages such as data collection, preprocessing, feature engineering, model training, evaluation, and deployment. As the number of stages increases, it becomes difficult for developers to manually verify whether the complete workflow is correct and follows good ML practices [1]– [3]. Traditional code analysis and linting tools can identify common programming errors, but they are often not designed to understand ML-specific problems such as data leakage, incorrect preprocessing order, missing validation, or the use of unsuitable evaluation metrics [4]– [6]. At the same time, several tools are available for tasks such as hyperparameter optimization, model explainability, and deployment, but these tools generally work independently, requiring developers to manage different systems during the ML development process [7], [8]. Large Language Models (LLMs) can provide explanations and suggestions for improving code, but using an LLM alone may produce unreliable recommendations because it may not have a structured understanding of the complete ML pipeline. To address these limitations, ModelForge AI is proposed as a unified platform for analyzing and improving existing Python-based ML projects. The system uses Abstract Syntax Tree (AST)-based analysis to

examine the project structure and identify different stages of the ML pipeline, followed by context-aware rules to detect potential issues and ML code smells. An LLM can then be used to explain the identified problems in a more understandable way and suggest possible improvements. The platform also includes proposed support for hyperparameter optimization, model explainability, and deployment resources, allowing multiple development needs to be handled within a common framework. The main objective of

ModelForge AI is to provide a structured and integrated approach to ML project analysis by using a shared representation of the ML pipeline. The current phase focuses on the system design and methodology, while the actual implementation, testing, and empirical evaluation are planned for the subsequent phase.

II. LITERATURE REVIEW OF EXISTING SYSTEMS

TABLE I. Literature Review of Automatic Detection of Long Method and God Class Code Smells Through Neural Source Code Embeddings

Paper Details	Objectives	Technology Used	Methodology Used	Efficiency/Performance	Issues
<p>Title: Automatic Detection of Long Method and God Class Code Smells Through Neural Source Code Embeddings [9]</p> <p>Authors: Aleksandar Kovačević, Jelena Slivka, Dragan Vidaković, Katarina-Glorija Grujić, Nikola Luburić, Simona Prokić, Goran Sladić</p> <p>Journal: Expert Systems with Applications</p> <p>Year: 2022</p> <p>DOI: 10.1016/j.eswa.2022.117607</p>	<ol style="list-style-type: none"> 1. To automatically detect two common code smells: Long Method and God Class. 2. To compare machine-learning based detection with traditional metric-based methods. 3. To check whether neural source-code embeddings can improve code-smell detection. 4. To study whether knowledge from pre-trained code models can be transferred to code-smell detection 	<ol style="list-style-type: none"> 1. Code2vec, code2seq and CuBERT were used to represent source code. 2. Traditional source-code metrics were also used for comparison. 3. Machine-learning models were trained for detecting Long Method and God Class. 4. The MLCQ manually labelled dataset was used for evaluation. 5. The study compared neural embeddings with traditional metric-based heuristics. 	<ol style="list-style-type: none"> 1. The source code was represented using both traditional code metrics and neural code embeddings. 2. The researchers trained ML models to classify code as smelly or non-smelly. 3. Three neural representations, code2vec, code2seq and CuBERT, were tested. 4. The models were compared with traditional heuristic-based code-smell detectors. 5. The experiments were performed on a large manually labelled dataset, allowing the different approaches to be compared under the same conditions. 6. An error analysis was also performed to understand where the models worked well and where they failed. 	<ol style="list-style-type: none"> 1. CuBERT performed better than the other approaches for the two studied code smells. 2. The study showed that neural source-code embeddings can be useful for automated code-smell detection. 3. The results also showed that pre-trained code representations can provide useful information without depending only on manually designed code metrics. 4. The paper also found that code-smell detection performance can vary significantly depending on the type of smell being detected. 	<ol style="list-style-type: none"> 1. The study focuses mainly on Long Method and God Class, so it does not cover all types of code smells. 2. Some code smells are harder to detect because their characteristics cannot be fully understood from the code representation alone. 3. The quality of the manually labelled dataset can affect the results. 4. The approach mainly focuses on detecting code smells, rather than explaining the problem and suggesting complete solutions. 5. It does not provide ML-pipeline optimization, model explainability or deployment support

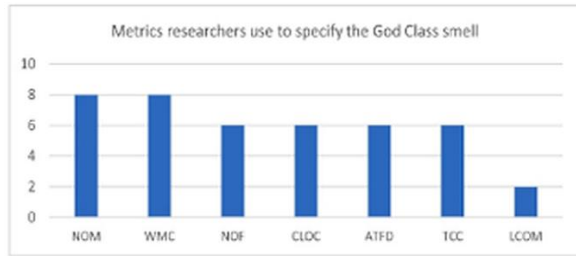


Fig. 1. Metrics researchers use to specify the God Class smell

The graph (figure 1) in the above table shows the frequency of different code metrics used by researchers to specify the God Class smell. NOM and WMC are the most frequently used metrics, followed by NOF, CLOC, ATFD and TCC. These metrics capture different characteristics such as class size, complexity, coupling and cohesion. The authors use such traditional code metrics as handcrafted features and later compare them with neural source-code embeddings for automatic God Class detection.

TABLE II. Literature Review of MLSmellHound: A Context-Aware Code Analysis Tool

Paper Details	Objectives	Technology Used	Methodology Used	Efficiency/Performance	Issues
Title: MLSmellHound: A Context-Aware Code Analysis Tool [6] Authors: Jai Kannan, Scott Barnett, Luís Cruz, Anj Simmons, Akash Agarwal Publication: ICSE-NIER 2022 Year: 2022	<ol style="list-style-type: none"> To improve code analysis for Machine Learning projects. To reduce irrelevant warnings produced by traditional code-analysis tools. To make code-analysis results more useful by considering the context of the ML project. To improve the maintainability and reliability of ML applications through context-aware analysis. 	<ol style="list-style-type: none"> Pylint was used as the main code-analysis tool. Static code analysis was used to identify coding issues. A context model was proposed for understanding ML code. The context included source-code purpose, technical domain, problem domain, team norms, operational environment and development stage. 	<ol style="list-style-type: none"> The source code is first given to the analysis system. The system checks the code and identifies its context. For the prototype, the presence of ML-library imports was used to identify whether a file was an ML module. Pylint warnings were then modified according to the identified context. Four types of context changes were proposed: subtraction, reprioritisation and remessaging. Finally, the modified warnings are presented to the developer as a more meaningful audit report. 	<ol style="list-style-type: none"> The prototype showed that existing code-analysis tools can be adapted to provide context-specific warnings. Some warnings that are less important for ML code can be given lower priority. Some general rules can be changed or disabled depending on the type of code. The approach is intended to reduce false positives and make warnings more useful for ML developers. However, the paper presents an initial demonstrator and research agenda, so it does not report a complete large-scale accuracy benchmark. 	<ol style="list-style-type: none"> The prototype mainly uses a simplified context, such as identifying a file as ML or non-ML. It mainly demonstrates the idea using Pylint and Python coding conventions. A complete catalogue of ML smells and their fixes was planned as future work. Large-scale testing with open-source ML repositories and developers was also planned for future work. It mainly focuses on context-aware code analysis and warnings and does not provide complete ML pipeline optimization, explainability or deployment assistance.

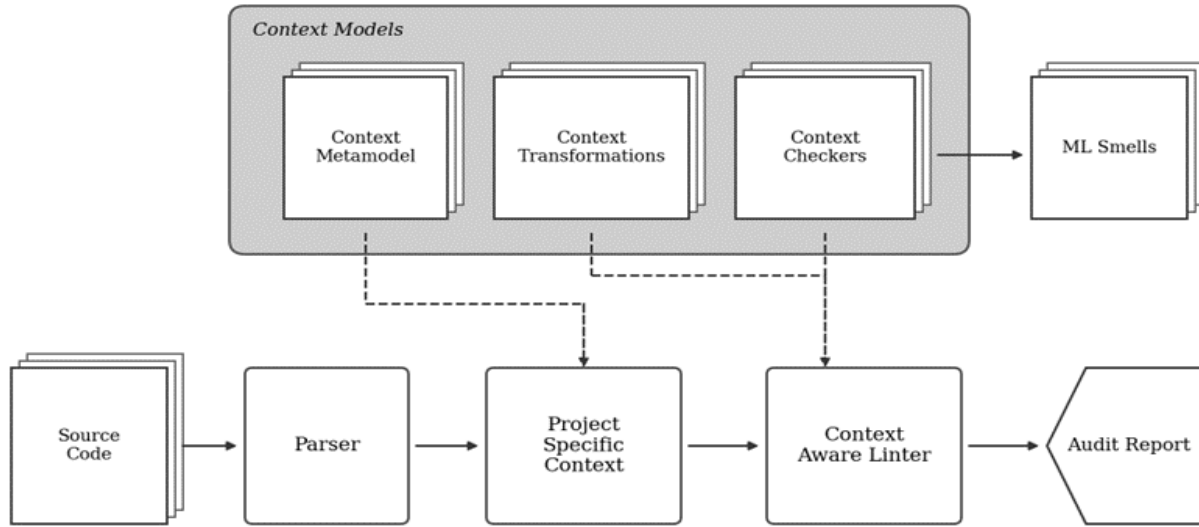


Fig. 2. Conceptual architecture of MLSmellHound

The figure (figure 2) presents the conceptual architecture of MLSmellHound. Source code is first passed through a parser to derive a project-specific context. Context models—comprising a context metamodel, context transformations, and context checkers—then guide the context-aware linter. The

linter identifies ML smells and produces an audit report. This approach addresses technical debt by making code analysis sensitive to the context of ML projects, helping reduce irrelevant warnings and improve the maintainability and robustness of ML software.

TABLE III. Literature Review of The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction

Paper Details	Objectives	Technology Used	Methodology Used	Efficiency/Performance	Issues
Title: The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction [2] Authors: Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, D. Sculley (Google, Inc.) Venue: Proceedings of IEEE Big Data 2017 (IEEE International Conference on Big Data)	1. Give teams a concrete, actionable way to judge how production-ready an ML system is, since model behavior is learned from data and can't be fully specified in advance. 2. Reduce ML technical debt through systematic testing and monitoring. 3. Provide a quantified score (the "ML Test Score") that	1. Tools: TensorFlow Debugger, TFX (pipeline platform), Facets (data visualization), Google Vizier/HyperTune (hyperparameter tuning), Sibyl (ML platform) 2. Techniques: grid search, Bayesian optimization, A/B tests, canary deployment, LIME, Daikon invariant detection, unit/integration tests	1. 28 tests in four groups of 7: Data, Model, ML Infrastructure, Monitoring 2. Scoring: 0.5 points per manually run test, 1 point per automated test. The final score is the minimum of the four section scores. 3. Validation: structured interviews with 36 Google	1. Interviews with 36 Google teams showed low adoption: no test was implemented by more than 80% of teams, and average section scores were only about 0.3 to 0.4 out of 1, with monitoring weakest. 2. Training/serving skew monitoring, which the authors call the most important test, was among the least implemented. Model code review and canarying were	The minimum-of-sections scoring hides individual progress, and equal test weights ignore differing importance. The evidence is self-reported from one company, with no link to actual incident rates. Many Data tests don't fit image or audio data, and the rubric doesn't cover expensive or heuristic-generated labels. Technical hurdles include non-deterministic training and slow

Paper Details	Objectives	Technology Used	Methodology Used	Efficiency/Performance	Issues
	teams can track over time, which also incentivizes better engineering practices.		teams, plus a hypothetical TFX-based system.	the most common. The interviews exposed real gaps, such as untested feature code and no per-country quality check. 3. Scope: continuously trained supervised models with low-...	model unit tests, and teams often wrongly assume upstream or downstream systems will catch their problems. It finds problems but says little about how to fix them.

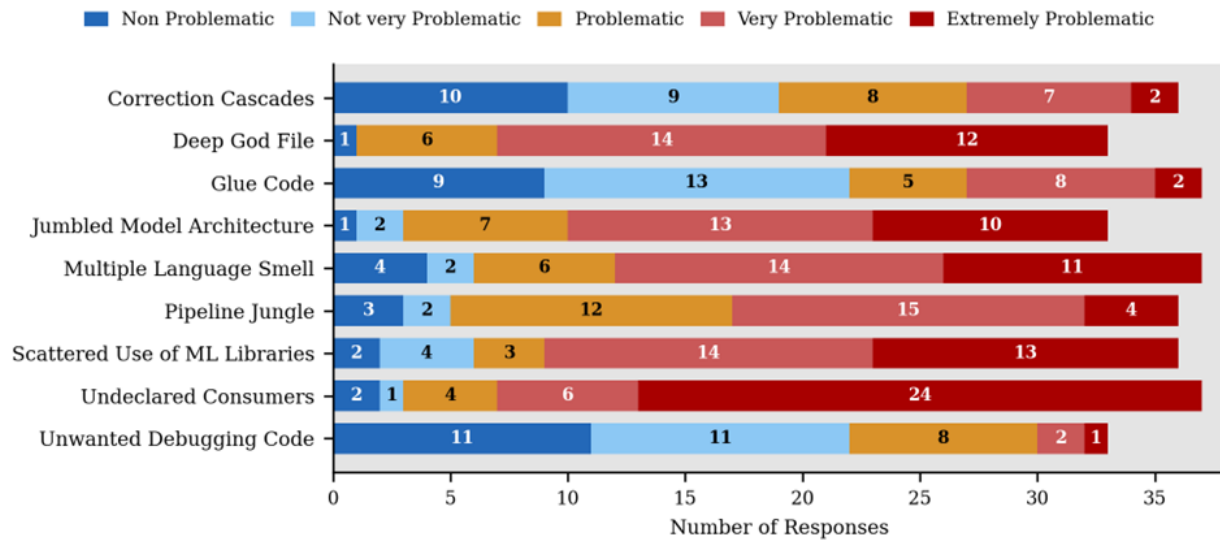


Fig. 3. Severity of AI technical debt (AITD) issues

The figure (figure 3) in the above table shows the severity of nine AI Technical Debt issues based on practitioners' responses. The key finding is that although these issues may not always occur frequently, several of them are considered highly problematic when they do occur. Undeclared Consumers is the most severe because of its security

risks, while issues such as Deep God File, Scattered ML Libraries, Jumbled Model Architecture and Pipeline Jungle mainly increase complexity and maintenance difficulty. This supports the paper's overall finding that AITD can have a high impact on the quality of AI-enabled systems.

TABLE IV. Literature Review of Technical Debt in AI-Enabled Systems: On the Prevalence, Severity, Impact, and Management Strategies for Code and Architecture

Paper Details	Objectives	Technology Used	Methodology Used	Efficiency/Performance	Issues
Title: Technical Debt in AI-Enabled Systems: On the Prevalence, Severity, Impact, and Management Strategies for Code	1. To study AI Technical Debt (AITD) in AI-enabled software systems, especially problems related	1. Artificial Intelligence & Machine Learning – Focus on AI-enabled software	1. Surveyed 53 AI practitioners using an experimental vignette study to understand their experiences	1. AI Technical Debt was found to be less common, but when present, it can have a serious impact	The study has some limitations, including a small sample size of 53 practitioners, which may limit the

Paper Details	Objectives	Technology Used	Methodology Used	Efficiency/Performance	Issues
and Architecture [3] Authors: Gilberto Recupito, Fabiano Pecorelli, Gemma Catolino, Valentina Lenarduzzi, Davide Taibi, Dario Di Nucci, Fabio Palomba Journal: The Journal of Systems & Software Year: 2024 Volume: 216, Article 112151 DOI: 10.1016/j.jss.2024.112151	to code and architecture. 2. To determine the frequency, severity, impact, and management strategies of AITD from the perspective of AI practitioners. 3. To understand how developers currently identify and mitigate these technical debt issues.	systems and ML components. 2. Prolific – Online platform used to conduct the practitioner survey. 3. GQM & Likert Scale – Goal-Question-Metrics approach and Likert-scale questionnaires used for data collection. 4. Statistical & Qualitative Analysis – Non-parametric statistical analysis and qualitative analysis of open-ended responses.	with AI Technical Debt. 2. Evaluated 9 types of AITD based on their prevalence, severity, impact, identification, and refactoring effort. 3. Used statistical analysis and qualitative analysis of open-ended responses to identify common patterns and challenges.	on the security, maintenance, and overall quality of AI-based systems. 2. Identifying and fixing these issues is mainly done manually, which requires considerable time, effort, and technical knowledge from developers. 3. The study suggests that automated tools and better techniques are needed to detect and manage AI Technical Debt more efficiently.	generalizability of the findings. Participant recruitment through Prolific may also introduce selection bias, and participants had different levels of AI and software-engineering expertise. The paper also highlights the lack of standardized practices, metrics, datasets, and automated tools for effectively identifying and managing AI Technical Debt.

Research on cybersecurity has examined threats and countermeasures across networked systems, cloud platforms, and user authentication. Studies on protecting confidential data in distributed cloud storage [16] and on network security attacks and countermeasures [17] provide foundations for understanding attack models and defensive strategies.

Authentication research on image-based authentication via randomized selection [18] and performance analysis of bag-of-password authentication across Python, Java, and PHP [19] explores improved security and implementation efficiency.

TABLE V. Literature Review of Machine Learning-Based Methods for Code Smell Detection: A Survey

Paper Details	Objectives	Technology Used	Methodology Used	Efficiency/Performance	Issues
Title: "Machine Learning-Based Methods for Code Smell Detection: A Survey" [10] Authors: Pravin Singh Yadav, Rajwant Singh Rao, Alok Mishra, Manjari Gupta Journal: Applied Sciences Year: 2024 DOI: 10.3390/app14146149 The selected studies were analysed according to machine learning algorithms,	1. To study the application of machine learning techniques for detecting different types of code smells in software systems. 2. To identify and compare commonly used machine learning algorithms such as Decision Trees, Random Forest, SVM, Naive Bayes, neural networks, and	1. Machine Learning algorithms such as Decision Tree, Random Forest, SVM, Naive Bayes, Bayesian Learning, Ensemble Learning, and Neural Networks. 2. Rule-Based Learning techniques were also investigated for identifying code smell patterns. 3. Software metrics such as Lines of Code (LOC),	1. A Systematic Literature Review (SLR) was conducted to collect and analyse previous research on machine-learning-based code smell detection. 2. Research papers were searched from	1. Precision, recall, accuracy, and F-measure were the most commonly used performance measures for evaluating code smell detection models. 2. Decision Tree was one of the most frequently applied	1. There is no fully standardized definition of code smells, which makes comparison between different studies difficult. 2. Many existing studies focus mainly on Java and object-oriented software, creating a need

Paper Details	Objectives	Technology Used	Methodology Used	Efficiency/ Performance	Issues
code smells, software metrics, datasets, feature-selection techniques, and evaluation measures	<p>rule-based learning.</p> <p>3. To analyse the software metrics and features commonly used for code smell detection, including LOC, WMC, DIT, LCOM, CC, and NoC.</p> <p>4. To examine the datasets, evaluation metrics, feature-selection methods, and experimental approaches used in existing code smell detection studies.</p>	<p>Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Lack of Cohesion of Methods (LCOM), Cyclomatic Complexity (CC), and Number of Children (NoC).</p> <p>4. Common datasets and software projects included Xerces, ArgoUML, Gantt Project, Azureus, Eclipse, and other open-source Java-based systems.</p>	<p>multiple digital libraries using predefined search terms and selection criteria.</p> <p>3. Relevant papers were filtered through title, abstract, and conclusion screening, followed by full-text evaluation and quality assessment.</p>	<p>algorithms, appearing in 27 of the reviewed studies.</p> <p>3. Several studies reported high detection performance. For example, some Random Forest-based approaches achieved precision values above 98% and accuracy values above 96%.</p>	<p>for research on other programming languages.</p> <p>3. Selecting the most useful software metrics and features is challenging because different code smells may require different characteristics.</p>

■ ADABOOST
 ■ B-Random Forest
 ■ Gradient Boosting tree
 ■ Bagging
 ■ Ensemble Learners
 ■ Random Forest

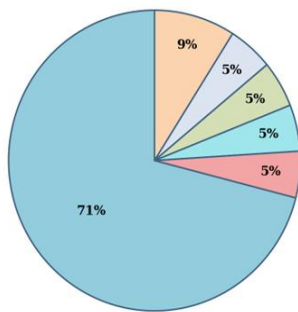


Fig. 4. Subclass distribution of machine learning algorithms used for code smell detection

The chart (figure 4) above illustrates the subclass distribution of machine learning algorithms used for code smell detection across the reviewed studies. The survey of computer-vision-based driver-monitoring systems [23] demonstrates visual intelligence for road safety and driver-risk identification, while the Python for data analytics textbook [24] supports programming-driven analytical problem-solving. Machine-learning malware detection [25] shows how predictive models strengthen threat identification, and comparative password-authentication implementations [26] emphasize evaluating security mechanisms across programming environments. Research on quantum computing for supply logistics [27] highlights emerging computational paradigms for complex logistics optimization.

TABLE VI. Literature Review of Leakage and the reproducibility crisis in machine-learning-based science

Paper Details	Objectives	Technology Used	Methodology Used	Efficiency/ Performance	Issues
<p>Title: Leakage and the reproducibility crisis in machine-learning-based science [5]</p> <p>Authors: Sayash Kapoor and Arvind Narayanan</p> <p>Journal: Patterns</p>	<p>1. To investigate reproducibility problems in scientific research that uses machine learning, with a major focus on data leakage.</p>	<p>1. Machine learning models including Logistic Regression, Random Forest, AdaBoost,</p>	<p>1. A cross-disciplinary survey was conducted to identify reproducibility problems caused by data leakage across different scientific fields.</p>	<p>1. The survey identified reproducibility problems in 22 papers across 17 scientific fields, collectively affecting 294 papers.</p>	<p>1. Data leakage can occur when training and test data are not properly separated during preprocessing, modelling, or</p>

Paper Details	Objectives	Technology Used	Methodology Used	Efficiency/Performance	Issues
Year: 2023 DOI: 10.1016/j.patter. 2023.100804	2. To study how data leakage affects the reliability and validity of ML-based scientific findings. 3. To develop a taxonomy of different types of data leakage and identify common causes of overoptimistic performance.	Gradient Boosted Trees, SVM, and Extra Trees were considered in the civil war prediction case study. 2. Train-test splitting, cross-validation, AUC, accuracy, confidence intervals, and statistical ...	2. A taxonomy of leakage was developed covering problems during data collection, preprocessing, modelling, and evaluation. 3. A systematic search of civil war prediction research identified 124 papers, which was narrowed to 12 papers with suitable train-test evaluation and available code and data. 4. The researchers reviewed the code and data of these studies, corrected identified errors.	2. Data leakage was found to be a common problem across all the surveyed fields and could produce highly overoptimistic model performance. 3. In the civil war prediction case study, 4 out of 12 reviewed papers contained leakage-related errors. 4. After correcting the errors, complex ML models no longer showed substantial improvement over Logistic Regression.	evaluation. 2. Using illegitimate or proxy features, temporal leakage, duplicate data, and non-independent samples can produce misleadingly high performance. 3. Poor data quality, incorrect evaluation metrics, missing code/data, and inconsistent reporting practices can reduce computational reproducibility.



Fig. 5. Number of papers containing the terms “civil war” and “machine learning” per year

The figure (figure 5) shows the number of papers containing the terms “civil war” and “machine learning” in the surveyed political-science literature from 2013 to 2022. The increasing trend illustrates the rapid adoption of ML methods in the field. The paper uses this growing body of ML-based research to investigate reproducibility problems caused by data leakage. The figure therefore provides context for why leakage-related technical debt is important: as ML

adoption increases, flawed preprocessing, sampling, or evaluation practices can produce inflated performance and make scientific results difficult to reproduce.

Work on cybercrime and open-source software [20], strategic information-security planning using decentralized identity [21], and machine-learning malware detection [22] highlights the need for layered security, proactive threat detection, and user-oriented protection.

A hybrid deep-learning intrusion-detection approach for AWS cloud [23] extends these ideas to cloud security, while the survey of confidential-data protection in distributed storage [16] emphasizes privacy and secure storage. Collectively, these studies establish a foundation for combining authentication, intrusion detection, machine learning, and secure cloud architecture against increasingly complex threats.

TABLE VII. Literature Review of General-purpose Hierarchical Optimization of Machine Learning Pipelines with Grammatical Evolution

Paper Details	Objectives	Technology Used	Methodology Used	Efficiency/Performance	Issues
<p>Title: General-purpose Hierarchical Optimization of Machine Learning Pipelines with Grammatical Evolution [7]</p> <p>Authors: Sumilan Estevez-Velarde, Yoan Gutiérrez, Yudivián Almeida-Cruz, Andrés Montoyo</p> <p>Journal: Information Sciences</p> <p>Year: 2020</p> <p>DOI: 10.1016/j.ins.2020.07.035</p>	<ol style="list-style-type: none"> 1. To introduce Hierarchical Machine Learning Optimisation (HML-Opt), an AutoML framework for automatically searching and optimising complete Machine Learning pipelines. 2. To provide a flexible framework where researchers can define a large and complex search space containing different algorithms, preprocessing techniques, feature representations, and hyperparameters. 	<ol style="list-style-type: none"> 1. The proposed framework is based on Automated Machine Learning (AutoML) and Grammatical Evolution. 2. It uses probabilistic grammars to represent the hierarchical search space of possible ML pipelines. 3. The framework can incorporate different Machine Learning algorithms, preprocessing techniques, feature-extraction methods, and ... 	<ol style="list-style-type: none"> 1. The researchers first define a context-free grammar representing the possible Machine Learning pipelines and their hierarchical structure. 2. Production probabilities are initialized and candidate pipelines are generated from the grammar. Each generated pipeline is evaluated using an appropriate performance metric. 3. The best-performing pipelines are selected and the production probabilities are updated using probabilistic grammatical evolution, increasing the likelihood of generating promising pipeline structures in later iterations. 	<ol style="list-style-type: none"> 1. HML-Opt was evaluated through multiple case studies and demonstrated that the framework can automatically explore complex hierarchical pipeline search spaces. 2. In one experiment, the optimisation process used 60 generations with a population of 50 pipelines per generation, with a selection factor of 20%. 3. The framework produced competitive results when compared with existing AutoML approaches across different benchmark problems. 	<ol style="list-style-type: none"> 1. The search space of ML pipelines can become extremely large because different algorithms, preprocessing techniques, features, and hyperparameters can be combined in many ways, making exhaustive search computationally impractical. 2. Different ML problems require different pipeline structures, so a single fixed AutoML framework may not be suitable for every scenario. HML-Opt addresses this by allowing researchers to define their own problem-specific grammar. 3. The optimisation process can still require significant computational ...

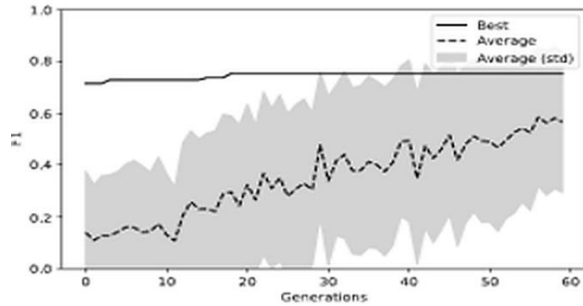


Fig. 6. Convergence of the grammatical-evolution-based HML-Opt optimisation (F1 over generations)

The graph (figure 6) in the above table illustrates the convergence of the grammatical-evolution-based HML-Opt search: as generations progress, the average

F1 of candidate pipelines improves, while the best solution remains at a high F1, indicating that the optimisation is progressively learning better ML pipelines. The confidential-data protection in distributed cloud storage [28] reinforces privacy and data integrity, soft clustering for gene-expression data [29] demonstrates computational intelligence in biomedical analysis, and digital health and medical-tourism innovations [30] illustrate connected healthcare services. Work on ethics and regulations in AI-driven ophthalmology [31] underscores responsible AI deployment and patient protection, and the review of NIST-recommended post-quantum cryptographic schemes [32] addresses preparation for quantum-computing threats.

TABLE VIII. Literature Review of The Prevalence of Code Smells in Machine Learning Projects

Paper Details	Objectives	Technology Used	Methodology Used	Efficiency/Performance	Issues
<p>Title: The Prevalence of Code Smells in Machine Learning Projects [4]</p> <p>Authors: Bart van Oort, Luis Cruz, Mauricio Aniche, Arie van Deursen</p> <p>Journal: arXiv preprint [cs.SE]</p> <p>Year: 2021</p> <p>arXiv: 2103.04146</p>	<ol style="list-style-type: none"> 1. To investigate the prevalence of code smells in Machine Learning projects. 2. To identify the most common code-quality issues affecting the maintainability and reproducibility of ML software. 3. To understand ML-specific software engineering challenges in real-world open-source projects. 	<ol style="list-style-type: none"> 1. Python was used as the primary programming language for analysing the projects. 2. Pylint 2.6.0 was used for static code analysis and detection of code smells. 3. Git repositories and Python dependency-management tools were used to collect projects and install their required dependencies. 	<ol style="list-style-type: none"> 1. The researchers collected 74 open-source ML projects containing 3,156 Python files and 511,018 lines of code. 2. Dependencies were installed in virtual environments and Pylint was executed on the projects. 3. Detected issues were collected and classified into Convention, Refactor, Warning, Error, and Fatal categories. 4. An automated tool was developed to clone repositories, create environments, install dependencies, run Pylint, and collect the detected code smells. 	<ol style="list-style-type: none"> 1. The study provided an empirical analysis specifically focused on ML application code rather than general Python software. 2. It identified recurring code-quality problems that can affect maintainability, reproducibility, and continuous integration. 3. The findings demonstrated that static analysis can help identify common problems in ML projects and highlight areas requiring software-engineering improvements. 	<ol style="list-style-type: none"> 1. Common code smells included unused wildcard imports, bad indentation, invalid naming, long lines, missing documentation, no-member errors, and duplicate code. 2. The study identified significant dependency-management problems in ML projects. 3. Pylint produced many false positives, especially when analysing ML libraries such as PyTorch. 4. These limitations make automated static analysis more challenging for ML projects and can affect maintainability and continuous integration.

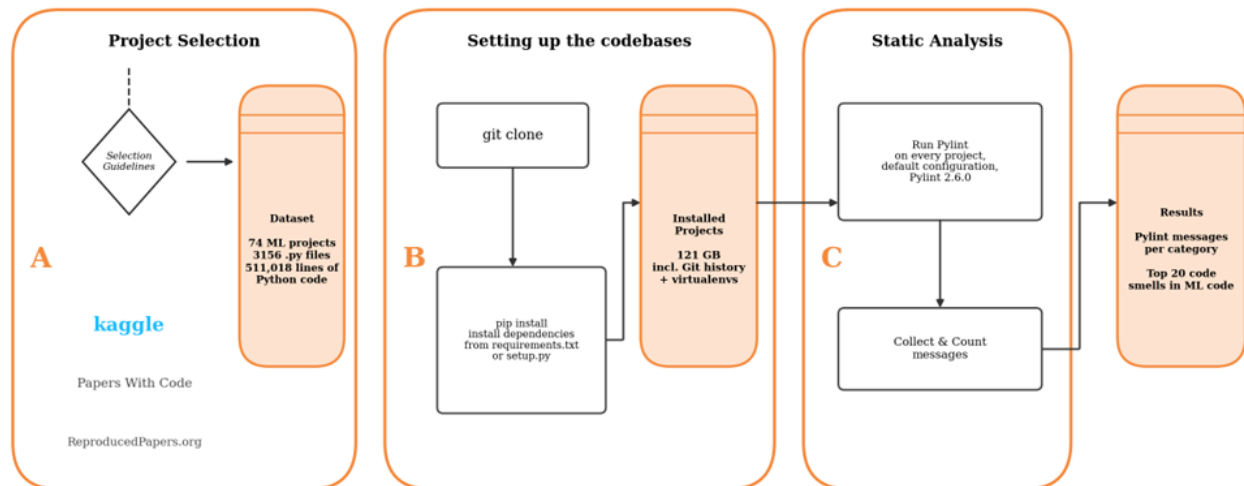


Fig. 7. Methodology used to analyse code smells in ML projects

The methodology diagram (figure 7) shows how the study analyzed ML projects for code smells. The researchers selected 74 open-source ML projects, cloned and configured their codebases with the required dependencies, and ran Pylint to collect and count static-analysis messages. UI design research on

users' viewing experience across static and dynamic websites [33] highlights presentation and content delivery for usability, while an automatic content creation and rearrangement mechanism [34] improves cloud storage utilization through efficient digital-content organization.

TABLE IX. Literature Review of Architecting ML-enabled systems: Challenges, best practices, and design decisions

Paper Details	Objectives	Technology Used	Methodology Used	Efficiency/Performance	Issues
Title: Architecting ML-enabled systems: Challenges, best practices, and design decisions [11] Authors: Roger Nazir, Alessio Bucaioni, Patrizio Pelliccione Journal: The Journal of Systems and Software Year: 2023 (published online October 2023; Vol. 207, 2024 issue) DOI: 10.1016/j.jss.2023.111860	<ol style="list-style-type: none"> To identify the most common software architecture design challenges faced when building ML-enabled systems. To collect the best practices used by researchers and practitioners when architecting these systems. To identify the main architectural design decisions taken for ML-enabled systems. To show how challenges, best practices and design decisions are related to each other. 	<ol style="list-style-type: none"> Four databases and indexing systems (IEEE Xplore, ACM Digital Library, Scopus, Web of Science) were used for the literature search. Online interviews were held on Zoom, and participants were found through LinkedIn and the authors' contacts. Coding, thematic synthesis, narrative 	<ol style="list-style-type: none"> A mixed-method study combined a systematic literature review with expert interviews The literature search returned 3038 studies, which were filtered to 34 and then extended by snowballing to 41 primary studies. A data extraction form with three parts (challenges, best practices, design decisions) was used to analyse each primary 	<ol style="list-style-type: none"> The study identified 35 design challenges, 42 best practices and 27 design decisions. Architecture and data were the most frequent challenge categories (29% each). Architecture was the most mentioned category for both best practices (32%) and design decisions 	<ol style="list-style-type: none"> Practitioners and the literature often disagree. For example, no architecture challenges were raised by practitioners, and their architecture best practices differed from those in the papers. Some challenges, such as data dependencies, data accuracy and model versioning, have no matching best practice or

Paper Details	Objectives	Technology Used	Methodology Used	Efficiency/Performance	Issues
		synthesis and contingency tables were used to analyse the data. 4. Technologies discussed in the findings include TensorFlow, Jupyter Notebook, microservices, client-server architecture, cloud platforms, Infrastructure as Code and federated learning.	study. Twelve semi-structured interviews (30 to 45 minutes each, 15 open-ended questions) were conducted with ML engineers, researchers and a manager from 9 countries. 4. Results from the literature and the interviews were compared, and the links among challenges ...	(25%). 4. Microservices suit NLP systems, the Siemens four-view architecture suits robot navigation, and client-server suits image processing. 5. TensorFlow and cloud technologies were found effective for scalability and performance.	design decision. 3. The literature search was done in 2021, so recent trends may be missing. 4. Only 12 interviews were conducted, so the results may not generalise, and respondents may have biased their answers.

The figure (figure 8) in the above table represents the research methodology followed in the paper. The authors divided their study into three phases: planning, conducting, and documenting. During planning, they defined the motivation, research questions, and research protocol. During conducting, they performed a systematic literature review and expert interviews, followed by data extraction and synthesis

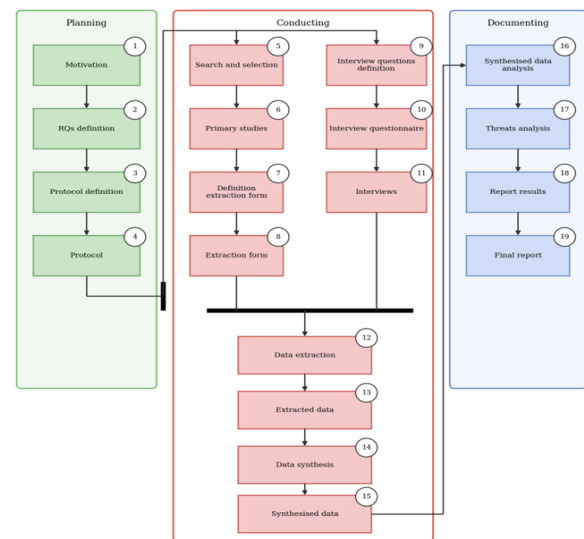


Fig. 8. Research methodology of the study on architecting ML-enabled systems

TABLE X. Literature Review of Taxonomy of Real Faults in Deep Learning Systems

Paper Details	Objectives	Technology Used	Methodology Used	Efficiency/Performance	Issues
Title: Taxonomy of Real Faults in Deep Learning Systems [12] Authors: Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio,	1. To build a taxonomy of real faults that occur in deep learning (DL) systems, since faults in DL	1. Deep learning frameworks studied: TensorFlow, Keras and PyTorch, with Python as the	1. Mining GitHub: 564 projects were retained. From the top 100 per framework, the authors filtered	1. Generalisation is limited because the study covers only three frameworks and a limited	1. Limited Scope: Examines only three frameworks (TensorFlow, Keras, PyTorch),

Paper Details	Objectives	Technology Used	Methodology Used	Efficiency/Performance	Issues
Andrea Stocco, Paolo Tonella Report: Technical Report TR-Precrime-2019-01, USI Università della Svizzera Italiana Published as: Proceedings of the ACM International Conference on Software Engineering (ICSE), Seoul, South Korea Year: 2019 (technical report, August 2019); ICSE version 2020 DOI: 10.1145/3377811.3380395	are more complex than in traditional software. 2. To help developers avoid common pitfalls and give testers a checklist for defining test scenarios. 3. To provide a realistic basis for fault seeding and for designing new mutation operators for DL testing. 4. To validate the taxonomy through a survey of developers and compare it with earlier work.	main language. 2. GitHub search API and the GitHub API were used to find repositories and to collect issues, pull requests and commits. 3. StackExchange Data Explorer was used to collect Stack Overflow posts. 4. A custom web application was used for manual labelling of artefacts and for resolving conflicts between authors.	issues, pull requests and commits using a vocabulary of 105 DL-related words. 2. Mining Stack Overflow: the top 3,000 most-viewed posts that had an accepted answer were selected. 3. Manual labelling: 1,059 artefacts were labelled by all authors using open coding over six rounds. Every artefact was labelled by at least two people. 4. Interviews: 20 researchers and practitioners (10 each) took part in semi-structured interviews of 26 to 52 minutes.	set of artefacts and interviewees. 2. Tagging of artefacts and interviews may be biased, and the taxonomy structure may reflect the authors' own point of view. 3. The interview questions could have been influenced by the initial taxonomy built from Stack Overflow and GitHub. 4. Stack Overflow questions are not fully representative of real-world DL problems. Interviewees called them "two worlds", with many beginner questions.	restricting how well the findings generalize. 2. Subjectivity: Manual tagging of artifacts and interviews risks introducing the authors' personal biases. 3. Biased Interviews: Interview questions may have been skewed by the authors' initially drafted taxonomy. 4. Skewed Data: Stack Overflow data heavily overrepresents beginner issues rather than complex, real-world faults. 5. Ignores generic programming errors and framework bugs, focusing strictly on DL-specific faults.

TABLE XI. Manual labelling rounds used to build the taxonomy of real deep-learning faults

Round	Analyzed Artifacts	Conflicts	Relevant to DL	New Inner Categories (1st lvl/2nd lvl/3d lvl.)	New Leaf Categories
1	29	3	5	-/-/-	5
2	110	11	16	-/-/-	11
3	134	8	14	-/-/-	5
4	126	11	20	5/7/7	10
5	345	31	46	0/2/1	21
6	315	47	48	0/0/0	13
Sub Total	1059	111	149	5/9/8	65
Interviews	297	6	226	0/2/0	27
Total	1356	117	375	5/11/8	92

The table (figure 9) summarizes the manual labelling process used to construct the taxonomy of real deep-learning faults. Across five rounds, the researchers

progressively analyzed more software artefacts and identified conflicts, relevant deep-learning issues, and new fault categories, ultimately covering 1,059

manually analyzed artefacts. The figure demonstrates the systematic effort required to identify and classify real DL faults, supporting the paper's broader finding

that faults can occur across different parts of deep-learning systems and become sources of maintenance and reliability debt.

TABLE XII. Literature Review of Code Smell Detection by Deep Direct-learning and Transfer-learning

Paper Details	Objectives	Technology Used	Methodology Used	Efficiency/ Performance	Issues
<p>Title: Code Smell Detection by Deep Direct-learning and Transfer-learning [13]</p> <p>Authors: Tushar Sharma, Vasiliki Efstathiou, Panos Louridas, Diomidis Spinellis</p> <p>Journal: Journal of Systems and Software</p> <p>Year: 2021</p> <p>DOI: 10.1016/j.jss.2021.110936</p>	<ol style="list-style-type: none"> 1. To check whether deep learning models can detect code smells without extensive manual feature engineering. 2. To compare CNN, RNN and autoencoder models and find which one works best for which smell. 3. To study transfer-learning: train a model on C# code and test it on Java code, and the other way round. 4. To share the tools, scripts and data openly so that other researchers can repeat and extend the work. 	<ol style="list-style-type: none"> 1. CNN (1D and 2D), RNN with LSTM layers, and autoencoders were built using Keras. 2. Designite (C#) and DesigniteJava were used to find smells and label the data. CodeSplit was used to split code into methods and classes. 3. A Tokenizer converted source code into integer tokens. GitHub repositories were selected using RepoReapers quality scores. 	<ol style="list-style-type: none"> 1. Code was split into methods and classes. Each one was labelled "smelly" (positive) or "not smelly" (negative) using the smell detection tools. 2. Four smells were studied: complex method, complex conditional, feature envy and multifaceted abstraction. Duplicates were removed. 3. The same language was used for training and testing. 144 CNN and 18 RNN configurations were tried. Autoencoders were trained only on non-smelly code, and a high reconstruction loss was treated as a smell. 4. Experiments ran on the GRNET supercomputer with NVIDIA V100 GPUs. 	<ol style="list-style-type: none"> 1. In direct-learning, the best F1 scores were 0.64 for complex method (autoencoder) and 0.63 (RNN). Complex conditional and feature envy were low, at about 0.20. 2. Autoencoders performed best overall. RNN was better than CNN only for complex method, and CNN-1D and CNN-2D were similar to each other. 3. Transfer-learning gave results close to direct-learning. For complex method, CNN-2D reached an F1 of 0.50, and the correlation between the two settings was 0.88. 4. The models performed far better than random classifiers, except for multifaceted abstraction. 	<ol style="list-style-type: none"> 1. Design smells were detected very poorly. Token-level input cannot capture class cohesion and method interactions. 2. Smelly samples were very few and the data was highly imbalanced, up to 588 clean samples per smelly one. This gave low precision. 3. The ground truth came from tools, not from developers' opinions, so context was not considered. There is also no standard gold dataset to compare with existing machine learning methods. 4. No single model worked for all smells. RNN training was extremely slow (over 1,000 seconds per epoch, against 1 to 4 seconds for the others).

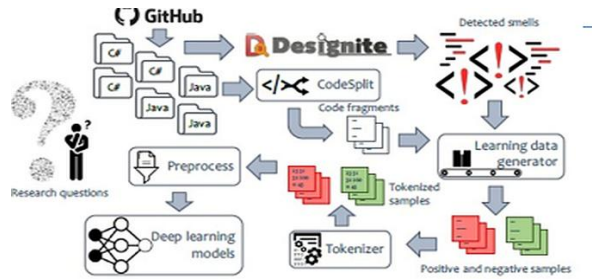


Fig. 9. Proposed pipeline for code smell detection by deep direct-learning and transfer-learning

The figure (figure 10) in the above table illustrates the proposed pipeline in which C# and Java code from GitHub is split into code fragments, labeled as smelly or non-smelly using Designite, converted into numerical tokens, preprocessed, and finally used to train deep-learning models for code-smell detection

TABLE XIII. Literature Review of Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities

Paper Details	Objectives	Technology Used	Methodology Used	Efficiency/Performance	Issues
<p>Title: Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities [8] Authors: Doris Xin, Hui Miao, Aditya G. Parameswaran, Neoklis Polyzotis Journal/Conference: ACM SIGMOD 2021 – International Conference on Management of Data Year: 2021 DOI: 10.1145/3448016.3457566 A Random Forest classifier was trained to predict whether a pipeline execution was likely to result in a deployed model. To identify optimization opportunities for making ML pipelines more efficient and resource-conscious.</p>	<ol style="list-style-type: none"> 1. To empirically analyze the lifespan, architecture, complexity, and resource consumption of production ML pipelines. 2. To understand how production pipelines execute repeatedly and identify inefficient or unnecessary computations. 3. To study pipeline characteristics such as model updates, data reuse, execution frequency, and deployment behaviour. 	<ol style="list-style-type: none"> 1. The researchers analyzed provenance information from approximately 3,000 production TFX pipelines containing more than 450,000 trained models. 2. Large provenance graphs were divided into smaller model graphlets, with each graphlet representing the execution associated with a particular model. 3. Graphlet features such as structure, model type, input-data overlap, and code-change history were extracted for analysis. 	<ol style="list-style-type: none"> 1. A mixed-method study combined a systematic literature review with expert interviews 2. The literature search returned 3038 studies, which were filtered to 34 and then extended by snowballing to 41 primary studies. 3. A data extraction form with three parts (challenges, best practices, design decisions) was used to analyse each primary study. 	<ol style="list-style-type: none"> 1. The study found that training accounts for less than one-third of total pipeline computation, with significant resources spent on data and model analysis and validation. 2. Approximately 80% of trained models were not deployed, indicating considerable unnecessary computation in repeated pipeline executions. 	<ol style="list-style-type: none"> 1. Production ML pipelines perform repeated computations, even when the resulting models are unlikely to be deployed. 2. A significant amount of computational resources is consumed by data processing, model analysis, and validation rather than training alone. 3. Large and interconnected provenance graphs make it difficult to identify which individual executions are useful or unnecessary.

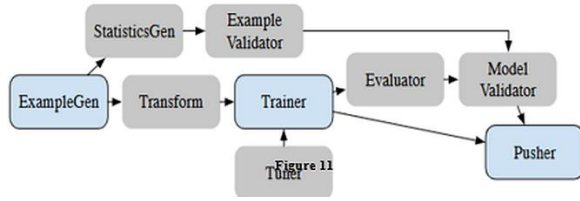


Fig. 10. Major stages of a production ML pipeline

The figure (figure 11) illustrates the major stages of a production ML pipeline, including example generation, statistics generation and validation, transformation, tuning, training, evaluation, model validation, and model pushing/deployment. It demonstrates that production ML systems contain substantial infrastructure beyond the model-training step. The additional stages and their repeated execution can create complexity, redundant computation, and resource costs, contributing to operational and architectural technical debt.

III. PROPOSED SYSTEM DESIGN

The proposed ModelForge AI system is designed as a modular platform that analyses existing Python-based machine learning projects and provides suggestions for improving them. The system accepts .py and .ipynb files through a React-based frontend and processes them using a FastAPI backend. The Code Analysis Module extracts notebook code, parses the source using Abstract Syntax Tree (AST) analysis, and identifies different stages of the ML workflow to create a common pipeline model. This model is then used by the Best Practice Checker to identify issues such as data leakage, incorrect preprocessing, and other ML-specific problems. The system follows a hybrid approach in which deterministic analysis identifies and traces potential problems, while an LLM is used to explain the detected findings and provide improvement suggestions. The architecture consists of six main modules covering frontend and backend services, code analysis, AI recommendations, ML optimization, explainability, and deployment/reporting. The optional optimization module uses Optuna and runs user code in a sandboxed environment, while the explainability module is designed to use SHAP for model interpretation. The system can also generate deployment-related artefacts based on the pipeline information. All results are combined into a final report and displayed through the

analysis dashboard. A key design feature is the use of a shared pipeline representation, which allows different modules to work with the same understanding of the ML project instead of operating as separate tools.

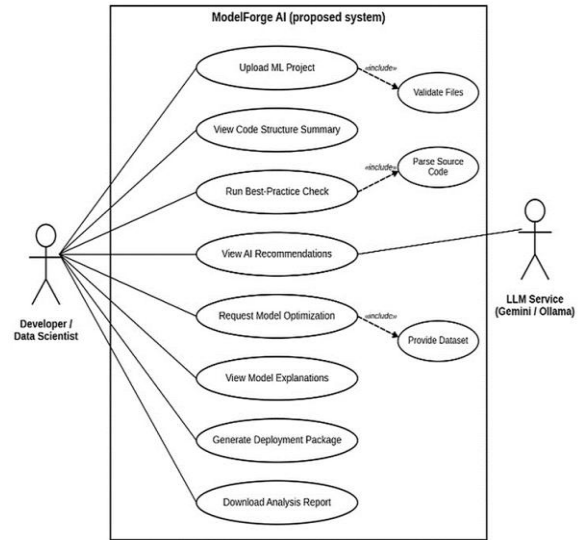


Figure 13: Proposed System: Proposed overall system design of ModelForge AI

The ML category carries the highest weight because it is the main focus of the system. A cap is also applied when critical issues exist, so that code with two critical issues cannot score above 68 regardless of its other scores. *D. Output Generation.* Each analysis produces five dark-themed Matplotlib charts saved as PNG files. One of them, the ML pipeline coverage chart (Figure X), shows which of the nine stages were detected or missing, with their line or cell numbers and the overall coverage. The results can also be exported as a Markdown report. Analysis history is currently stored in JSON files. *E. Current Limitations.* The prototype is limited to the deterministic analysis path. The LLM recommendation module (M3) exists only as a placeholder, and the Optuna optimization module (M4), SHAP explainability and the deployment generator (M6) have not been started. Authentication, a proper database and export formats other than Markdown are also not yet available, and only five of the planned ML-specific rules exist. All of these are planned for Phase-II together with the quantitative evaluation. In its current state the prototype covers about 70–75% of the planned analysis-only system, and about 20–25% of the full proposed design. The workflow identifier then compares the base name of every function call (for

example, fit for model.fit()) against predefined indicator sets. Each call is assigned to one of nine stages: dataset loading, preprocessing, train-test split, model creation, training, prediction, evaluation, cross-validation and hyperparameter tuning. The line number of every detected stage is recorded. For notebooks, a line-to-cell mapping also gives the cell number, so findings point to the exact location in the user's file. B. Rule Engine. Fifteen deterministic rules were implemented in five categories, as summarised in table below.

IV. ARCHITECTURE DIAGRAM

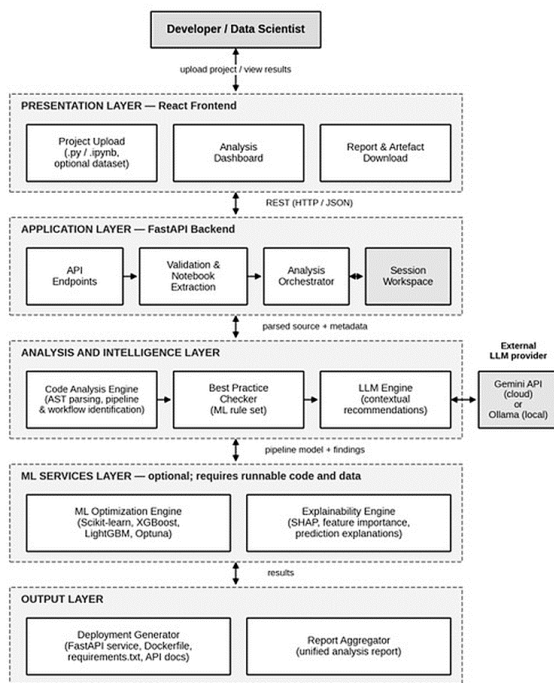


Figure 13: Architecture Diagram: Proposed overall system architecture of ModelForge AI

V. PROPOSED METHODOLOGY AND ALGORITHMS USED

The system accepts Python .py files and Jupyter Notebook. ipynb files as project input, then extracts and prepares the source code for analysis. It uses Abstract Syntax Tree (AST) parsing to understand the structure and flow of the code and identify key ML pipeline stages such as data loading, preprocessing, training, and evaluation. Predefined rules are then applied to detect issues like data leakage, incorrect preprocessing, and missing validation. To make these findings actionable, an LLM (either Gemini or a

locally hosted Ollama model) explains the detected problems and suggests improvements. When runnable code and data are available, Optuna [14] can optionally be used for hyperparameter optimization, and SHAP [15] provides explanations of model predictions. The system also generates deployment-related resources, including requirements files and FastAPI/Docker resources. Overall, it follows a hybrid approach that combines deterministic static analysis with LLM-based recommendations for a more complete and reliable analysis.

Module	Architectural components
M1 Frontend and Backend Module	React frontend; FastAPI REST API; input validation; analysis orchestrator; session workspace
M2 Code Analysis Module	Notebook extractor; AST parser; pipeline parser; workflow identifier; context classifier
M3 AI Recommendation Module	LLM engine; prompt builder; provider adapters (Gemini, Ollama); response validator
M4 ML Optimization Module	Search-space registry; Optuna study manager; sandboxed executor
M5 Explainability and Best Practice Module	Best Practice Checker (rule engine); SHAP-based explainability engine
M6 Deployment and Reporting Module	Deployment generator (templates); report aggregator; export

VI. IMPLEMENTATION PROTOTYPE

In Phase I, a working prototype was built for the analysis path of the proposed architecture. It covers M1 (frontend and backend), M2 (code analysis), the rule-based part of M5, and a basic form of the reporting function of M6.

The backend exposes eight REST endpoints through Fast API for submitting analyses, retrieving history, charts, and reports. Its data structures, such as issues, scores and workflow components, are defined as twelve Pedantic models. The React frontend, written in TypeScript, has eleven pages. Nine are fully functional: the landing, analysis, results, dashboard, chart gallery, report viewer and three informational pages. The Projects and Settings pages are interface shells only. A. *Library and Pipeline-Stage Detection*. After AST parsing, each import is reduced to its root module and

matched against a dictionary of 26 common ML and data science libraries, so that sklearn. model_selection is reported as scikit-learn.

Category	Rules implemented	Highest severity
Data leakage	Preprocessor fitted before the split (ML-LEAK-001); fitting on test or validation data (ML-LEAK-002)	Critical
ML practice	Missing train-test split, missing random_state, missing evaluation	High
Security	Hardcoded credentials, hardcoded paths, insecure pickle deserialization	Critical
Code quality	Unused imports, wildcard imports, bare except	Medium
Maintainability	Long functions (>50 lines), deep nesting (>4 levels), large classes (>200 lines), inefficient ML loops	High

ML-LEAK-001 works by comparing line numbers. It reports an issue when a scaler, encoder or imputer is fitted on a line before the train_test_split call. ML-LEAK-002 inspects the arguments passed to fit() and fit_transform() and flags names that refer to test or validation data. Since ML-LEAK-001 depends on line order rather than full data-flow tracking, it is an approximation and can miss leakage that occurs through reassigned variables or across functions. *C. Scoring.* The results are summarised as a weighted composite score, which was not part of the original design. Each category starts at 100 points, and a penalty is deducted for every issue according to its severity: 18 for critical, for high, 4 for medium and 1.5 for low. The ML score is the average of the ML-practice and data-leakage scores. The overall score is calculated as $0.35 \times \text{ML} + 0.25 \times \text{Security} + 0.25 \times \text{Code Quality} + 0.15 \times \text{Maintainability}$

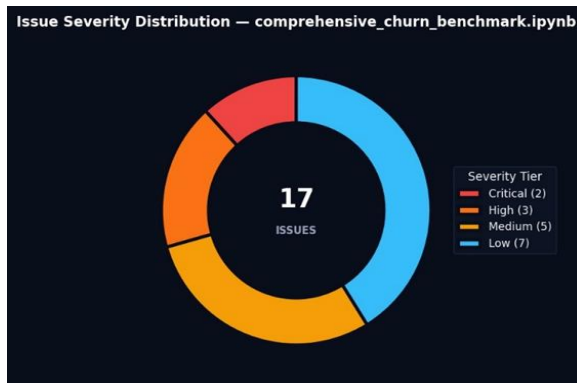


Figure 14: Severity Distribution Chart

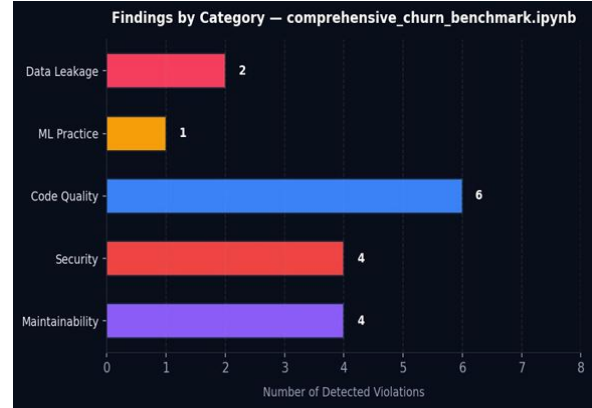


Figure 15: Violation Detection Chart

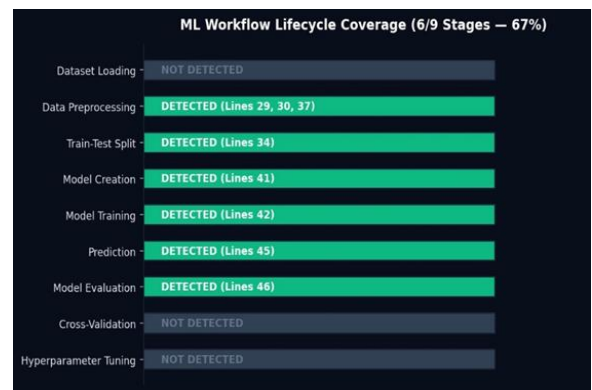


Figure 16: ML Workflow Detection

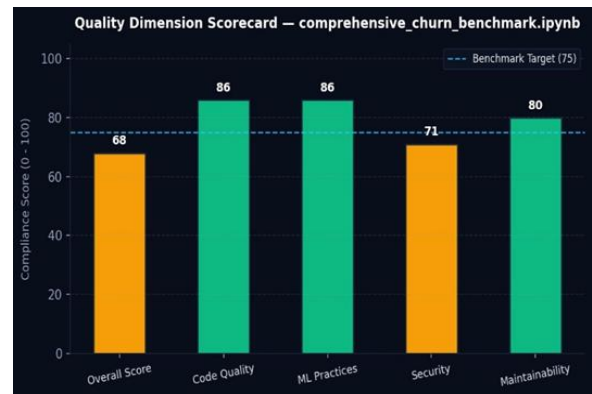


Figure 17: Quality Scorecard Chart

VII. CONTRIBUTION AND FINDINGS

The main contribution of ModelForge AI is the integration of ML code analysis, best-practice checking, AI-based recommendations, optimization, explainability, and deployment support into a single platform. The system uses a shared ML pipeline representation to connect these different functions and provide more consistent analysis. The findings from

Phase-I show that combining deterministic static analysis with LLM-based assistance can provide a structured way to identify potential ML-specific issues and explain them to users. However, the complete system implementation and quantitative evaluation are planned for Phase-II, so no experimental performance results are reported at this stage.

VIII. CONCLUSION

ModelForge AI presents a unified approach for analyzing and improving machine learning projects. The proposed system combines AST-based code analysis, rule-based best-practice checks, LLM recommendations, optimization, explainability, and deployment support within a single platform. Its shared ML pipeline representation helps connect these functions and provide structured feedback to developers. Phase-I establishes the system architecture and methodology, while complete implementation, testing, and quantitative evaluation are planned for Phase-II.

REFERENCES

- [1] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 28, 2015.
- [2] E. Breck, S. Cai, E. Nielsen, M. Salib, and D. Sculley, "The ML test score: A rubric for ML production readiness and technical debt reduction," in *Proc. IEEE Int. Conf. Big Data*, 2017, pp. 1123–1132.
- [3] G. Recupito, F. Pecorelli, G. Catolino, V. Lenarduzzi, D. Taibi, D. Di Nucci, and F. Palomba, "Technical debt in AI-enabled systems: On the prevalence, severity, impact, and management strategies for code and architecture," *J. Syst. Softw.*, vol. 216, Art. no. 112151, Oct. 2024, doi: 10.1016/j.jss.2024.112151.
- [4] B. van Oort, L. Cruz, M. Aniche, and A. van Deursen, "The prevalence of code smells in machine learning projects," *arXiv preprint arXiv:2103.04146 [cs.SE]*, 2021.
- [5] S. Kapoor and A. Narayanan, "Leakage and the reproducibility crisis in machine-learning-based science," *Patterns*, vol. 4, no. 9, Art. no. 100804, Sep. 2023, doi: 10.1016/j.patter.2023.100804.
- [6] J. Kannan, S. Barnett, L. Cruz, A. Simmons, and A. Agarwal, "MLSmellHound: A context-aware code analysis tool," in *Proc. 44th Int. Conf. Softw. Eng.: New Ideas and Emerging Results (ICSE-NIER)*, 2022, pp. 66–70.
- [7] S. Estevez-Velarde, Y. Gutiérrez, Y. Almeida-Cruz, and A. Montoyo, "General-purpose hierarchical optimisation of machine learning pipelines with grammatical evolution," *Inf. Sci.*, vol. 543, pp. 58–71, Jan. 2021, doi: 10.1016/j.ins.2020.07.035.
- [8] D. Xin, H. Miao, A. Parameswaran, and N. Polyzotis, "Production machine learning pipelines: Empirical analysis and optimization opportunities," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2021, pp. 2639–2652, doi: 10.1145/3448016.3457566.
- [9] A. Kovačević, J. Slivka, D. Vidaković, K.-G. Grujić, N. Luburić, S. Prokić, and G. Sladić, "Automatic detection of Long Method and God Class code smells through neural source code embeddings," *Expert Syst. Appl.*, vol. 204, Art. no. 117607, Oct. 2022, doi: 10.1016/j.eswa.2022.117607.
- [10] P. S. Yadav, R. S. Rao, A. Mishra, and M. Gupta, "Machine learning-based methods for code smell detection: A survey," *Appl. Sci.*, vol. 14, no. 14, Art. no. 6149, Jul. 2024, doi: 10.3390/app14146149.
- [11] R. Nazir, A. Bucaioni, and P. Pelliccione, "Architecting ML-enabled systems: Challenges, best practices, and design decisions," *J. Syst. Softw.*, vol. 207, Art. no. 111860, Jan. 2024, doi: 10.1016/j.jss.2023.111860.
- [12] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng. (ICSE)*, Seoul, South Korea, 2020, pp. 1110–1121, doi: 10.1145/3377811.3380395.
- [13] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, "Code smell detection by deep direct-learning and transfer-learning," *J. Syst. Softw.*, vol. 176, Art. no. 110936, Jun. 2021, doi: 10.1016/j.jss.2021.110936.
- [14] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation

- hyperparameter optimization framework,” in Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining, 2019, pp. 2623–2631.
- [15] S. M. Lundberg and S.-I. Lee, “A unified approach to interpreting model predictions,” in Proc. Adv. Neural Inf. Process. Syst. (NeurIPS), vol. 30, pp. 4765–4774, 2017.
- [16] G. D. K. Singh and M. K. Jayanthi, Eds., Network Security Attacks and Countermeasures. IGI Global, 2016, doi: 10.4018/978-1-4666-8761-5.
- [17] R. M. Balajee, M. K. Jayanthi Kannan, and V. Murali Mohan, “Image-based authentication security improvement by randomized selection approach,” in Inventive Computation and Information Technologies. Springer, Singapore, 2022, pp. 61–71.
- [18] M. K. J. Kannan, N. Aaijaz, K. G. Mani, and V. Tewari, The Future of Innovation and Technology in Education: Trends and Opportunities. S&M Publications, 2025.
- [19] B. R. M. and J. K. M. K., “Intrusion detection on AWS cloud through hybrid deep learning algorithm,” Electronics, vol. 12, Art. no. 1423, 2023, doi: 10.3390/electronics12061423.
- [20] K. L. S. Kumar and M. K. J. Kannan, “A survey on driver monitoring system using computer vision techniques,” in Innovative Computing and Communications, A. E. Hassanien, S. Anand, A. Jaiswal, and P. Kumar, Eds., Lecture Notes in Networks and Systems, vol. 1021. Springer, Singapore, 2024, doi: 10.1007/978-981-97-3591-4_21.
- [21] M. K. J. Kannan, “A bird's eye view of cyber-crimes and free and open-source software's to detoxify cyber-crime attacks—an end user perspective,” in Proc. 2nd Int. Conf. Anti-Cyber Crimes (ICACC), Abha, Saudi Arabia, 2017, pp. 232–237, doi: 10.1109/Anti-Cybercrime.2017.7905297.
- [22] M. K. J. Kannan, S. Kumar, P. T. Kalaivaani, and G. Tripathi, Artificial Intelligence and Blockchain Technology for Human Resource Management. Scientific International Publishing House, 2025.
- [23] M. K. Jayanthi, “Strategic planning for information security—DID mechanism to befriend the cyber criminals to assure cyber freedom,” in Proc. 2nd Int. Conf. Anti-Cyber Crimes (ICACC), Abha, Saudi Arabia, 2017, pp. 142–147, doi: 10.1109/Anti-Cybercrime.2017.7905280.
- [24] S. K. Shukla, U. Dwivedi, M. K. J. Kannan, and C. Sarvani, Python for Data Analytics: Practical Techniques and Applications. JSR Publications, 2024.
- [25] P. Jain, I. Rajvaidya, K. K. Sah, and J. Kannan, “Machine learning techniques for malware detection—a research review,” in Proc. IEEE Int. Students' Conf. Electr., Electron. Comput. Sci. (SCEECS), Bhopal, India, 2022, pp. 1–6, doi: 10.1109/SCEECS54111.2022.9740918.
- [26] B. R. M., M. M. V., and J. K. M. K., “Performance analysis of bag of password authentication using Python, Java and PHP implementation,” in Proc. 6th Int. Conf. Commun. Electron. Syst. (ICCES), Coimbatore, India, 2021, pp. 1032–1039, doi: 10.1109/ICCES51350.2021.9489233.
- [27] M. K. J. Kannan and T. R. Shree Nee, “Qubits unveiled: A deep dive into quantum computing and its revolutionary potential for supply logistics,” in Qubits Unveiled: Quantum Computing Solutions for Efficient Supply Logistics, P. Gaba, A. Panwar, V. Jain, and R. Kannan, Eds. Nova Science Publishers, 2025, pp. 273–293, doi: 10.52305/WSXW8884.
- [28] H. Naik and M. K. J. Kannan, “A survey on protecting confidential data over distributed storage in cloud,” SSRN, Dec. 1, 2020. [Online]. Available: <https://ssrn.com/abstract=3740465>
- [29] E. Kavitha, R. Tamilarasan, A. Baladhandapani, and M. K. J. Kannan, “A novel soft clustering approach for gene expression data,” Computer Systems Science and Engineering, vol. 43, no. 3, pp. 871–886, 2022, doi: 10.32604/csse.2022.021215.
- [30] T. R. Shree Nee, M. K. J. Kannan, and K. Mariyappan, “Digital health and medical tourism innovations for digitally enabled care for future medicine: The real-time project's success stories,” in Navigating Innovations and Challenges in Travel Medicine and Digital Health. IGI Global Scientific Publishing, 2025, pp. 325–344, doi: 10.4018/979-8-3693-8774-0.ch016.
- [31] M. K. J. Kannan, S. N. T. Ramesh, and K. Mariyappan, “Ethics and regulations in AI-driven ophthalmology,” in Generative Artificial Intelligence in Ophthalmology, B. K. Mishra, A. Kumar, K. Mariyappan, V. Tiwari, P. S. Rathore, and G. H. Das, Eds., ch. 34. Scrivener Publishing,

2026. [Online]. Available:
<https://www.scrivenerpublishing.com/cart/title.php?id=1341>

- [32] K. Mariyappan, B. K. Mishra, S. A. Harini, and M. K. J. Kannan, "A review on NIST-recommended cryptographic schemes for the post-quantum computer world era," in Proc. 8th EAI Int. Conf. Big Data Innovation for Sustainable Cognitive Computing (BDCC 2025). Springer Nature, 2025. [Online]. Available: <https://link.springer.com/book/9783032392329>
- [33] R. M. Balajee, M. K. J. Kannan, and V. Murali Mohan, "Web design focusing on users viewing experience with respect to static and dynamic nature of web sites," in Inventive Computation and Information Technologies, S. Smys, V. E. Balas, and R. Palanisamy, Eds., Lecture Notes in Networks and Systems, vol. 336. Springer, Singapore, 2022, doi: 10.1007/978-981-16-6723-7_5.
- [34] R. M. Balajee, M. K. J. Kannan, and V. Murali Mohan, "Automatic content creation mechanism and rearranging technique to improve cloud storage space," in Inventive Computation and Information Technologies, S. Smys, V. E. Balas, and R. Palanisamy, Eds., Lecture Notes in Networks and Systems, vol. 336. Springer, Singapore, 2022, doi: 10.1007/978-981-16-6723-7_7.