Automated Code-Smell Prioritization: An Expert Review of a Master's Thesis Research Plan

I. Analysis of the Proposed Research Plan: A Foundational Review

1.1. Strategic Cohesion and Iterative Design: A Robust Research Roadmap

The proposed research plan for "Automated Code-Smell Prioritization Integrated with Agile Platforms for Automatic Ticket Creation and Ranking" demonstrates a high degree of strategic cohesion. Its most notable strength is the adoption of a structured, iterative roadmap, which is a hallmark of a mature research methodology. The plan is systematically broken down into four distinct cycles, each approximately 3–4 months long, aligning logically with a standard academic timeline.[1] This phased approach is particularly well-suited for a complex master's thesis, as it effectively manages risk by systematically validating hypotheses and building upon successful increments.

The documents provided meticulously detail this approach, beginning with Cycle 1, which serves as a minimal viable research product.[1] The goals for this initial cycle are clearly defined: establish a baseline with SonarQube, identify initial weaknesses like false positives, and build a first ML/AI prototype to address these issues. The plan also specifies concrete deliverables for each cycle, from "Prototype v0.1" in the first phase to "Prototype v1.0 (Research System)" at the conclusion of the project.[1] The strategic choice of using well-supported, open-source projects like Spring Boot, Apache Commons Lang, and JUnit 5 further underpins this plan by ensuring a diverse and realistic codebase for analysis.[1] This foresight ensures that the research can be reproduced by others, a fundamental requirement for a strong academic publication.

The overall structure of the project mirrors a modern, agile software development lifecycle. Each cycle acts as a sprint, culminating in a demonstrable, evaluated increment. This methodical approach mitigates the inherent risks of a long-term research project, such as unforeseen technical challenges or data acquisition problems. By proving the most fundamental hypotheses in the initial cycle, the plan ensures a publishable result even if later cycles face unexpected difficulties, thereby demonstrating a sophisticated understanding of how to manage complexity and risk in a research context.

Cycle	Focus	Key Actions	Expected Increment/Deliverable
Cycle 1	Exploration & First Prototype: Detection Engine (false positives) & Prioritization Logic (static heuristics).[1]	Collect baseline data from SonarQube, implement a small ML prototype for false positive reduction, add a naive ranking adjustment.[1]	Prototype v0.1: ML-enhanced SonarQube with a limited scope.
Cycle 2	Refinement & Expansion: Improved Detection Engine (NLP embeddings) & Prioritization (feedback-driven ranking).[1]	Expand dataset to multiple repositories, simulate a developer feedback loop, and compare results to the baseline.	Prototype v0.2: A smarter detection and ranking system.
Cycle 3	Integration & Cross-Domain Synergy: Unified Detection + Prioritization & Defect Density Mapping.[1]	Integrate prior improvements into a unified module, add defect density correlation, and rank issues by likelihood of	Prototype v0.3: A multi-domain improvement.

		causing bugs.	
Cycle 4	Stabilization & Thesis: End-to-End Workflow & Behavioral ML Patterns.[1]	Integrate all improvements into a coherent tool, perform a large-scale evaluation, and write the dissertation.	Prototype v1.0 (Research System) + Thesis.

1.2. The Foundational Hypothesis: A Valid Research Problem

The project's foundational hypothesis is sound and targets a well-documented and important research gap in the field of software engineering. The user correctly identifies the fundamental limitations of traditional static analysis tools like SonarQube, which rely on static, hand-crafted rules to detect code smells.[1] These heuristic-based rules are known to produce a high rate of false positives and lack the context-aware, dynamic prioritization needed for a modern, complex codebase. The plan meticulously outlines these weaknesses across five core domains: a context-insensitive "Detection Engine," a "Prioritization Logic" based on static severity, a simplistic "Code Health Metric," a "Defect Density Overlay" that ignores historical context, and the absence of "Behavioral & ML Patterns" to create a feedback loop.[1]

The external academic literature corroborates these identified weaknesses. Research on code smell detection tools highlights the "diversity and inconsistency" and the "unclear results" that stem from varying definitions and a lack of contextual understanding.[2] The user's strategic decision to augment SonarQube's output rather than rewrite its core engine is a brilliant tactical choice that makes the project highly feasible within a master's thesis timeline.[1] This approach correctly identifies the core value of a tool like SonarQube (its robust AST parsing and rule engine) and focuses the research on the problem where a machine learning approach can provide the most value—namely, the post-processing and contextualization of the results. The proposed system acts as a sophisticated filter on SonarQube's output, analogous to a spam filter that refines an initial, noisy classification.[1] This demonstrates a clear understanding of the academic and technical landscape, positioning the research as an intelligent enhancement rather than an unattainable ground-up development effort.

1.3. Evaluation of Technical Feasibility: A Realistic First Step

The technical plan for Cycle 1 is concrete, well-justified, and highly feasible. The proposed approach leverages a precise set of open-source tools and projects, which provides a solid foundation for a reproducible and impactful study.[1] The selection of Java projects—specifically Spring Boot, Apache Commons Lang, and JUnit 5—is justified by their diverse and realistic codebases, which have mature SonarQube support, providing a rich dataset for analysis and ML model training.[1] The visual workflow diagram [1] further clarifies the process, from baseline analysis and dataset preparation to prototype development and evaluation.

The proposed ML approach, using well-understood and reliable algorithms like Random Forest or XGBoost, is appropriate for a first prototype and avoids the pitfalls of overly complex or unproven models.[1] The plan even specifies the exact metrics for initial evaluation, such as precision and recall for false positive reduction and Spearman correlation for ranking improvement.[1] The technical viability of the project's final step, automated ticket creation, is also confirmed by the existence of robust APIs for both Jira and GitHub.[3] This demonstrates that the end-to-end pipeline is not a conceptual hurdle but a practical integration task. By proactively selecting well-known, actively maintained repositories, the user is ensuring that their baseline and data collection methods are not tied to a private, inaccessible codebase, which is a crucial aspect of academic rigor and publication potential.

II. Core Research Contributions and Novelty Assessment

2.1. The Synthesis of Domains: Bridging the Gap Between Research and Practice

The project's primary contribution to the field is its successful synthesis of three distinct but interconnected domains: static code analysis, machine learning for software engineering, and agile/DevOps practices. The research is not merely a theoretical exercise in ML; it is an end-to-end system designed to deliver tangible, practical value in a professional software development context.[1] This synthesis is a key differentiator from academic work that focuses on just one of these aspects.

The plan explicitly positions this work as an advancement in the state-of-the-art by transforming a passive, rule-based tool into a dynamic, adaptive system.[1] Traditional static analysis provides a snapshot of a codebase at a single point in time. The user's proposal, with its feedback loops and integration of historical bug data, turns this snapshot into a living, context-aware system that learns from developer behavior and project evolution.[1] The final output is no longer a static report but a dynamically ranked list of actionable tasks pushed directly into an agile platform like Jira or GitHub Issues. This transformation from a passive artifact to an active, intelligent agent in the software development process elevates the work from a purely academic exercise to a piece of research with direct, practical implications.

2.2. The Feedback Loop: The Central Pillar of Novelty

As correctly identified in the plan, the feedback-driven ML system is the central pillar of this research's novelty.[1] This approach directly addresses a major gap in current static analysis tools and aligns perfectly with cutting-edge research in the field of software engineering. The core idea is to train a machine learning model that re-prioritizes code smells based on validation and feedback from human developers, thereby transforming a static system into a living, adaptive one.[1]

This approach does more than simply retrain a model; it provides a crucial, missing layer of information that cannot be captured by static rules: developer intent and perceived criticality. A static rule for a "Long Method" might flag a method that exceeds a simple line-count threshold. A human developer, however, may know that this particular method, while long, is an acceptable design decision or a necessary trade-off for performance. The feedback loop enables the system to learn this context-specific distinction. By having developers re-rank tickets or mark false positives, the model learns the nuanced, context-dependent nature of technical debt within a specific project. This ability to unlock developer intent and use it to refine the model's understanding is a powerful and publishable contribution. As noted in existing research, this type of developer-driven code smell prioritization is a crucial aspect of effective software maintenance and is a topic of significant interest in the academic community.[5]

Dimension	User's Approach	Key Related Works
ML Approach	Hybrid approach	Employs ensemble models

	combining a static analysis engine (SonarQube) with a machine learning post-processor.[1] Initially using traditional classifiers (Random Forest, XGBoost) and later exploring code embeddings and NLP.[1]	like Gradient Boost and Max Voting for Python code smell detection.[7] Proposes using CodeBERT embeddings with Bi-LSTM networks for prioritization.[5]
Data Source	Utilizes SonarQube's output as a baseline, augments it with historical data from Git (commits, churn) and Agile platforms (Jira/GitHub bug history).[1]	Relies on manually annotated datasets (DACOS, SmellyCodeDataset) or social coding metrics (GitHub followers) to train and validate models.[8]
Contribution Type	Develops a novel, end-to-end workflow that not only detects and prioritizes smells but also automatically creates and ranks tickets in agile platforms.[1]	Focuses on specific aspects of the problem, such as improving detection accuracy [7], benchmarking detection tools [11], or proposing new prioritization techniques.[5]
Novelty	The core novelty is a feedback-driven ML system that learns from developer behavior, allowing for context-aware and dynamic prioritization that evolves over time.[1]	Prioritization models based on mixed features (developer, process, code metrics).[6] Use of machine learning to detect smells in ML-specific projects.[2]

2.3. Advancing Context-Aware Prioritization

Beyond the feedback loop, the plan to integrate historical context and bug data represents a significant step forward in prioritization research. Current prioritization logic is often static and heuristic-based, failing to consider how a codebase has evolved or where real problems have historically occurred.[1] The proposed approach will be dynamic, a major step toward a more intelligent and useful system.

By using historical commit data and bug reports from platforms like Jira and GitHub, the system can learn which code smells in which parts of the codebase are most likely to lead to actual defects.[1] This establishes a probabilistic, data-driven link between a static symptom (the code smell) and a real-world, negative outcome (a bug). For example, a traditional static analysis tool might flag a "God Class" and give it a "Major" severity. The user's system, by looking at the bug history for that project, could identify that God Classes have a high correlation with bug reports and dynamically re-rank that specific issue as "Critical" to reflect its true impact. This causal link is a highly compelling and publishable finding. The availability of public datasets that include developer and process metrics such as

ChurnLOC (lines of code added/deleted) and NoC (number of commits) provides the raw material needed to implement this part of the plan, moving the idea from a theoretical concept to a practical one.[10]

III. Rigor, Metrics, and Publication Potential

3.1. Establishing a Rigorous Research Methodology

The plan correctly identifies the need for empirical evaluation and a manually annotated ground truth dataset to validate the ML model.[1] However, this is also the most challenging and potentially fragile part of the methodology due to the inherent subjectivity of code smells. The plan for manually annotating 200–500 samples is a good starting point, but a top-tier publication will require a larger, more robust, and demonstrably consistent dataset.[1]

A significant weakness in many code smell papers is a small, privately-annotated dataset that is not reproducible. To mitigate this, the user should consider leveraging existing public, annotated datasets as a benchmark. Datasets such as DACOS and SmellyCodeDataset are intentionally curated to contain subjective code snippets, which are the most valuable for training and benchmarking ML models.[8] By adopting a formal annotation methodology, potentially involving multiple annotators and using inter-rater reliability metrics like Cohen's Kappa, the user can dramatically increase the methodological rigor and the validity of their conclusions. Creating a reproducible corpus of their annotated data would also significantly increase the impact and publication potential of the research.

Dataset Name	Description	Primary Use Case	Source
DACOS (DAtaset of COde Smells)	A manually annotated dataset of 10,267 annotations for 5,192 Java code snippets, specifically filtered for subjective smells like multifaceted abstraction, complex method, and long parameter list.[9]	Training and validating machine learning models to detect subjective code smells.	9
SmellyCodeDataset	A collection of intentionally smelly code snippets in multiple languages (Java, Python, C++, JavaScript) with labels for various smell types (e.g., Long Method, Feature Envy).[8]	Evaluating the capability of Large Language Models (LLMs) to detect and suggest refactorings for common code smells.	8
GitHub Developer Metrics	A dataset containing over 700 anonymized developers from 17 open-source projects, associated with 24 software metrics (e.g., ChurnLOC, DiP - Days in Project, NoC - Number of Commits).[10]	Performing empirical studies on developer characteristics and correlating code quality with developer metrics.	10

3.2. A Framework for Evaluation Metrics

The user's proposed metrics—Precision, Recall, and Spearman correlation—are correct and necessary for evaluating the ML model.[1] However, for a high-impact publication, a more robust set of metrics is required, especially given the likelihood of imbalanced datasets (i.e., true positives will likely be a small fraction of all reported issues). As noted in external research on ML-based code smell detection, metrics like Cohen's Kappa and the Matthews Correlation Coefficient (MCC) are particularly useful in such settings as they provide a more balanced evaluation than accuracy alone.[7]

A nuanced, third-order analysis reveals a distinction between measuring the technical efficacy of the ML model and measuring the real-world impact of the entire system. A strong paper must show both. While precision and recall are excellent for assessing the model's performance on the detection task, they do not tell the whole story. The user's ultimate goal is to "help developers fix the right problems faster".[1] To demonstrate this, the user must link the ML model's improved metrics to broader software quality metrics, such as a reduction in defect density over time or a decrease in Mean Time to Recovery (MTTR) for a given codebase.[12] The causal chain would be: the ML model's improved precision leads to fewer false positive tickets; this improved signal-to-noise ratio enhances developer focus, which in turn increases ticket resolution rates and ultimately reduces the overall defect density of the codebase. This causal narrative is what transforms a good paper into a great one.

Metric	Definition	Purpose	Causal Link to Project Goal
Accuracy	The proportion of correctly predicted instances out of the total instances: A=TP+TN+FP+FNTP +TN.7	Measures the overall correctness of the classifier.	A basic measure of the model's performance on the detection task.
Precision	The proportion of correctly predicted positive instances among all predicted positives:	Reflects the model's ability to avoid false positives.	Directly measures the reduction in false alarms, a core project goal.[1] Higher precision

	P=TP+FPTP.7		means fewer wasted developer hours on non-issues.
Recall	The proportion of correctly predicted positive instances among all actual positives: R=TP+FNTP.7	Reflects the model's ability to find all actual positive instances.	Measures the model's ability to identify all true code smells, ensuring critical issues are not missed.
F-measure	The harmonic mean of precision and recall: F=2×P+RP×R.7	Provides a single score that balances both precision and recall.	A combined measure of the model's effectiveness, useful for comparing against other models or a baseline.
Matthews Correlation Coefficient (MCC)	A balanced metric that considers all four confusion matrix values (TP,TN,FP,FN) and is useful for imbalanced datasets: MCC=(TP+FP)(TP+FN)(TN+FP)(TN+FN)TP×TN−(FP×FN).7	A reliable measure of classifier performance, especially when dealing with imbalanced data.	Provides a more balanced evaluation of the model's performance, validating the rigor of the research.
Cohen's Kappa	Evaluates the agreement between predicted and actual classifications, adjusted for chance: Cohen’s	Measures the level of agreement between annotators or between the model and a human, beyond what would be expected by	Crucial for validating the consistency of the manually annotated ground truth dataset, addressing the inherent subjectivity of code

	Kappa=1−PeP0−Pe. 7	chance.	smells.

3.3. Identifying Suitable Publication Venues

The project's interdisciplinary nature makes it an excellent candidate for high-impact publication in top-tier academic venues. The work is not a pure machine learning paper, nor is it a pure software engineering paper; its novelty is the synthesis of these two fields. The plan's explicit focus on a "novel ML/AI integration" and a "working prototype tool" [1] makes it a perfect fit for conferences and journals that value applied and empirical research.

The research is a prime candidate for top-tier software engineering venues such as the International Conference on Software Engineering (ICSE), the Foundations of Software Engineering (FSE), and the International Conference on Automated Software Engineering (ASE). It also fits well within journals such as the IEEE Transactions on Software Engineering (TSE) or the Journal of Systems and Software. The strong emphasis on a working, integrated system and its direct implications for DevOps practices and software quality are a perfect match for the applied research focus of these conferences. A paper that demonstrates a reproducible, end-to-end system that reduces false positives, enhances prioritization, and seamlessly integrates with a professional workflow is highly likely to be well-received by the academic community.

IV. Actionable Recommendations for Implementation

4.1. Refining the Cycle 1 Plan

To ensure the success of Cycle 1, the following refinements to the data acquisition and labeling methodology are recommended:

- **Leverage Public Datasets:** Instead of relying solely on a small, manually annotated subset of their chosen projects, the user should use an existing public dataset like DACOS or SmellyCodeDataset as a benchmark.[8] This provides an immediate, proven baseline for model training and allows for a direct comparison of the user's work against other research.

- **Formalize Annotation Methodology:** To address the inherent subjectivity of code smells, the user should formalize their manual labeling process. This could involve using a web-based annotation tool (similar to TAGMAN mentioned in DACOS [9]) and employing multiple annotators to establish inter-rater reliability.
- **Create a Reproducible Corpus:** The research plan should include the goal of publishing the manually annotated dataset and the scripts used to extract it. This is a significant contribution in itself and ensures that the research is reproducible, a major factor in publication decisions.

4.2. Enhancing the End-to-End Automation

The plan for automated ticket creation is a critical component of the project's practical value. The user should think beyond simple issue creation to truly enhance the end-to-end workflow:

- **Implement a Dynamic Re-ranking API:** The system should not only create tickets but also implement a mechanism to dynamically *update* their priority or status based on new commits, pull requests, or developer feedback. This would require using the Jira/GitHub APIs for editing, not just for creation.[3] The ML model could be scheduled to re-assess the priority of open tickets, ensuring the agile board always reflects the most critical and up-to-date information.
- **Add Rich Context to Tickets:** The created tickets should not just contain a summary. They should include the relevant code snippet, a link to the file, and, if possible, a visualization of the issue. Furthermore, the model's prioritization could be explained using interpretability techniques, providing developers with the reasoning behind the recommendation.[5] This ensures the tickets are truly "actionable" and do not require additional developer effort to understand the context.

4.3. Mitigation of Anticipated Challenges

A forward-looking analysis of potential hurdles is essential for a successful project. The user should be prepared for challenges related to data scarcity, computational cost, and the subjectivity of ground truth.

- **Address Data Scarcity for Rare Smells:** The plan to use a few projects is excellent for an initial study but may not provide enough training data for less common code smells. To mitigate this, the user should consider using synthetic datasets or data augmentation techniques in later cycles.

- **Manage Computational Cost:** The plan mentions using AST embeddings and NLP in later cycles, which can be computationally expensive.[1] The user should plan for this, either by leveraging cloud-based ML platforms or by focusing on lightweight models that can run efficiently on a local machine.
- **Validate Ground Truth with Inter-Rater Agreement:** To address the subjectivity of manual labeling, the user should calculate inter-rater reliability metrics (e.g., Cohen's Kappa) to ensure their annotations are consistent. This metric is a standard for validating manually labeled datasets and would be a strong addition to the methodology section of their paper.

V. Conclusion

The user's research plan is exceptionally strong, well-structured, and highly promising. It is not only good and feasible but also possesses a clear, compelling, and demonstrable degree of novelty. The plan to transform a static analysis tool into a dynamic, context-aware, feedback-driven system is a significant and publishable contribution. The project successfully bridges the gap between theoretical research and practical software development by focusing on a problem that is both academically interesting and directly relevant to industry. With the actionable recommendations provided—particularly those related to refining the ground truth methodology, expanding the evaluation metrics, and enhancing the end-to-end automation—the user is well-positioned to produce a high-impact thesis and a strong paper suitable for publication in top-tier software engineering venues.

Works cited

1. Cycle 1 workflow Diagram-Automated Code-Smell Prioritization.docx
2. Automatic Identification of Machine Learning-Specific Code Smells - arXiv, accessed August 28, 2025, https://arxiv.org/pdf/2508.02541?
3. Jira REST API examples - developer Atlassian., accessed August 28, 2025, https://developer.atlassian.com/server/jira/platform/jira-rest-api-examples/
4. Creating an issue - GitHub Docs, accessed August 28, 2025, https://docs.github.com/articles/creating-an-issue
5. Developer-Driven Code Smell Prioritization | Request PDF - ResearchGate, accessed August 28, 2025, https://www.researchgate.net/publication/347307000_Developer-Driven_Code_Smell_Prioritization
6. On The Effectiveness of Developer Features in Code Smell Prioritization: A Replication Study, accessed August 28, 2025, https://huang.zj.cn/pdf/J25.pdf
7. An Empirical Evaluation of Ensemble Models for Python Code Smell Detection -

MDPI, accessed August 28, 2025, https://www.mdpi.com/2076-3417/15/13/7472
8. HRI-EU/SmellyCodeDataset - GitHub, accessed August 28, 2025, https://github.com/HRI-EU/SmellyCodeDataset
9. DACOS—A Manually Annotated Dataset of Code Smells - Tushar Sharma, accessed August 28, 2025, https://www.tusharma.in/preprints/DacosMSR2023.pdf
10. Dataset of open-source software developers labeled by their experience level in the project and their associated software metrics - PMC, accessed August 28, 2025, https://pmc.ncbi.nlm.nih.gov/articles/PMC9813504/
11. ON THE EVALUATION OF CODE SMELLS AND DETECTION TOOLS., accessed August 28, 2025, https://repositorio.ufmg.br/bitstream/1843/JCES-AREGGR/1/thanis_paiva.pdf
12. The 8 software quality metrics that actually matter - GetDX, accessed August 28, 2025, https://getdx.com/blog/software-quality-metrics/
13. Top 12 Software Quality Metrics to Measure and Why | LinearB Blog, accessed August 28, 2025, https://linearb.io/blog/software-quality-metrics