

Can Language Models Detect Backlog Technical Debt? Empirical Evidence from Agile Projects

Dewni Deraniyagala¹, Muhammad Shazil Khan¹, Sadia Hassan Chowdhury¹, Arshpreet Singh¹, Hung Cao²,
and Francis Palma¹

¹SE+AI Research Lab, Faculty of Computer Science, University of New Brunswick, NB, Canada

²Analytics Everywhere Lab, Faculty of Computer Science, University of New Brunswick, NB, Canada
{dewni.d, m.shazilkhan, sadia.chowdhury, arshpreet.singh, hung.cao, francis.palma}@unb.ca

Abstract—Agile software development is managed using backlog items. However, issue trackers often accumulate technical debt, in the form of duplicate, incomplete, and ambiguous issues. These problems can disrupt planning, development, and progress tracking. This study investigates how well language models can identify such issues using five open-source agile projects from the TAWOS dataset. We fine-tuned four encoder models: `all-mpnet-base-v2`, `BERT-base`, `SpanBERT`, and `ALBERT-base` to detect duplicate issues, and evaluated three open-weight large language models (LLMs): `Llama 3.1 8B`, `Qwen 2.5 14B`, and `Mistral 7B` to detect incomplete and ambiguous issues after applying four prompting strategies: *zero-shot*, *few-shot*, *chain-of-thought*, and *few-shot chain-of-thought*. We evaluated confidence-weighted and majority-vote ensemble models and compared two input types: *issue title only* vs. *issue title with description*. Among the encoder models, `all-mpnet-base-v2` detected duplicate issues best, achieving the highest F1-score on four out of five projects (up to 89%). Among the LLMs, `Qwen 2.5 14B` performed best for ambiguous and incomplete issue detection, reaching F1-scores up to 55% and 80%, respectively. Our findings offer practical guidance for choosing models, prompting methods, and input types to automate the detection of backlog technical debt in agile development.

Index Terms—Backlog technical debt, Duplicate issues, Ambiguous issues, Incomplete issues, Language models, Agile

I. INTRODUCTION

Technical debt is the long-term cost a software project pays when teams choose quick solutions over more reliable ones [1]. The concept now extends beyond source code. Requirements debt covers flaws in how teams specify, trace, and maintain requirements [2], while process debt covers inefficiencies accumulating in the development process [3]. Building on these, we define **backlog technical debt** (BTD) as “*the buildup of low-quality backlog items (duplicate, unclear, incomplete, inconsistent, or long-standing issues) that may increase the effort needed to implement and maintain software systems*”. Because agile teams plan and perform work from the backlog, BTD accumulates in the issue tracker and, if left unmanaged, harms planning and delivery [4].

BTD takes several forms, including (i) duplicate issues describe work items that already exist; (ii) ambiguous issues leave work items open to interpretation; and (iii) incomplete issues lack information developers need to proceed. Like other forms of technical debt, BTD grows when ignored, inflates planning effort, distorts prioritization, and slows delivery [1].

Detecting these problems early, before they affect subsequent sprints, is essential for maintaining a healthy backlog.

Teams record issues in tools such as Jira through free-form text comprising a short title and a longer description, among other fields. Because this free-form text is written by many developers with differing writing styles, it is often informal and inconsistent, making BTD difficult to detect manually [5]. The volume of issues compounds this challenge: active projects contain thousands of issues, far more than any agile team can manually review. Thus, automated detection of BTD is essential for maintaining backlogs efficiently.

Prior work has focused on duplicate requirements and bug-report detection [6], [7], ambiguous requirements [8], [9], and incomplete issue descriptions [10] using off-the-shelf models, lexicon-based heuristics, or handcrafted rules, and often on small or single-project datasets that limit generalizability [11], [12]. As a result, combined approaches to BTD and evidence on how fine-tuned embedding models and LLMs compare across multiple projects remain underexplored.

We address these gaps by evaluating language models for automated BTD detection. We fine-tune four embedding models for duplicate detection and evaluate three open-weight LLMs for ambiguous and incomplete issues, using the TAWOS dataset [13] across five open-source agile projects. We also evaluate a confidence-weighted ensemble (CWE) and a majority-vote ensemble (MVE), compare four prompting strategies for the LLMs, and contrast two levels of information fidelity: *issue title only* vs. *issue title with description*. We investigate the following research questions:

RQ1: *How effective are language models in the automated detection of BTD in agile software development?*

RQ2: *How effective is an ensemble of language models for the automated detection of BTD in agile software development?*

RQ3: *How do different large language models perform in detecting BTD using various prompting strategies?*

RQ4: *How does the best-performing model perform under varying levels of information fidelity?*

Our findings suggest that for duplicate issue detection, the fine-tuned embedding model `all-mpnet-base-v2` performed best, achieving an F1-score up to 89% outperforming both weaker encoders and the CWE model. `Qwen 2.5 14B`

was the most effective LLM for ambiguous and incomplete issue detection, with few-shot prompting generally performing best for ambiguous detection and zero-shot or few-shot prompting performing best for incomplete issue detection. Input fidelity affects the two tasks differently: including issue descriptions improved duplicate detection but significantly reduced LLM performance for ambiguous and incomplete issue detection. Our main contributions are as follows:

- We provide empirical evidence on the prevalence and automated detection of backlog technical debt, together with actionable insights for agile developers.
- We evaluate fine-tuned embedding models and open-weight LLMs for detecting three forms of BTD across five open-source agile projects from the TAWOS dataset.
- We assess the confidence-weighted and majority-vote ensemble models comparing with individual models.
- We compare four prompting strategies (zero-shot, few-shot, chain-of-thought, and few-shot chain-of-thought) and analyze their effects on LLM performance on ambiguous and incomplete issues.
- We investigate how input fidelity—issue title only vs. issue title with description—affects the detection performance of the best-performing models.

In the rest of the paper: Section II reviews related work. Section III describes our methodology, including dataset, models, and experimental setup. Section IV presents results, and Section V discusses findings and implications. Section VI concludes and outlines future work.

II. RELATED WORK

The quality of software requirements has a strong effect on system development, maintenance effort, and software quality. In Agile settings, these problems are known as requirements technical debt [2], and prior work has addressed them with lexical similarity metrics, rule-based systems, traditional machine learning, and deep learning models [6], [7], [12], [14], [15]. More recently, researchers have turned to transformers and language models to better capture the meaning of software artifacts [8], [16], [17]. Still, current methods vary widely in the types of BTD they target and often depend on fixed representations, handcrafted rules, or prompt-based inference, limiting their adaptability to the requirements domain.

Duplicate Detection: Duplicate detection finds requirements that describe the same functionality. Such duplicates create redundancy, burden maintenance teams, and can lead to inconsistent implementation decisions [6], [12], [14]. Early work used lexical similarity measures such as TF-IDF and cosine similarity, often with heuristic filtering and bag-of-words representations to reduce noise and improve matching [6], [14]. However, lexical overlap misses cases where different terms express the same intent. Later work adopted word and sentence embeddings to estimate similarity beyond exact word matching [12], [14], but it mostly relies on frozen, pre-trained embeddings that do not capture the characteristics of software requirement corpora. For detecting duplicate bugs, researchers widely apply

deep learning methods such as CNNs and Siamese networks to model report pairs, using word2vec embeddings and supervised learning to capture semantic interactions [7], [15]. These models depend on the structured form of bug reports and handcrafted features, which do not transfer well to free-text requirements. More recent work uses transformer-based embeddings such as SBERT for duplicate bug-report detection [17]. These learn a strong semantic space, but the encoders remain fixed and are not trained on backlog requirements, limiting their effectiveness.

Ambiguity Detection: Ambiguity detection identifies unclear, vague, or multi-interpretable statements that can cause misunderstandings during design and implementation [8]. Early models rely on rule-based templates, manually defined ambiguity dictionaries, and TF-IDF features [18], [19]. These approaches depend on predefined rules and struggle to identify new types of ambiguity. Later approaches used pre-trained language models to resolve anaphoric ambiguity by linking pronouns to their references [20], or used syntactic parsing that ranks a sentence’s multiple parse trees by probability [11]. However, most target only limited ambiguity types and depend on third-party NLP pipelines, making them sensitive to parser errors and domain drift. More recently, researchers apply large language models with in-context learning and prompting to classify or explain ambiguity in requirements [8].

Incompleteness Detection: Incompleteness detection identifies missing, vague, or insufficient information in requirements that can lead to incorrect or partial implementation. Masked language modeling can detect missing elements by framing the task as predicting tokens from context, often with BERT-based models [10]. For bug reports, researchers define completeness through required fields such as expected behavior, observed behavior, and steps to reproduce and use supervised classification with template-based verification to find missing fields [21], [22]. These methods assume rigid structures and are poorly suited to free-text *issues*. More recent work uses LLMs to generate missing bug-report information iteratively [21], but still depends on predefined schemas or generation tools.

Overall, existing backlog technical debt detection relies on rule-based methods, frozen word embeddings, or prompt-based LLMs. In contrast, we fine-tune embedding models on real-world agile projects and empirically evaluate both embedding models and LLMs across input types in large agile projects.

III. RESEARCH METHODOLOGY

Figure 1 depicts the research methodology in three stages: data collection and cleaning, detection BTB, and evaluation.

A. Research Questions

We presented the four research questions (RQ1–RQ4) that guide this study in Section I. We further decompose RQ1 and RQ2 into the sub-questions addressed by our experiments.

For RQ1, we examine the effectiveness of individual models:

- **RQ1.1** *To what extent can embedding models accurately detect duplicate issues?*

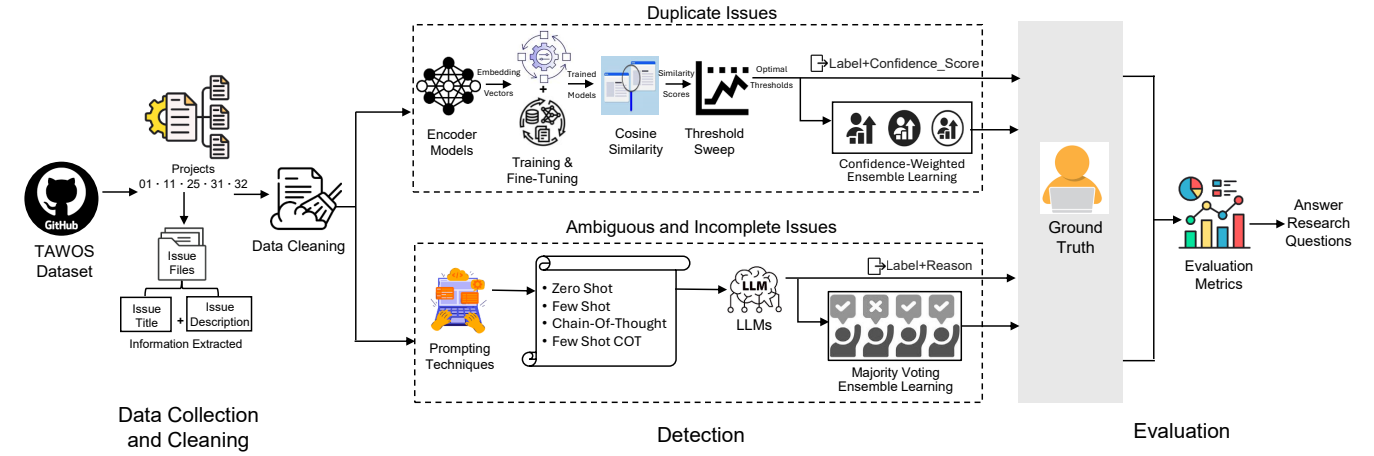


Fig. 1. The Overview of the Research Methodology

TABLE I
OVERVIEW OF THE FIVE OPEN-SOURCE AGILE PROJECTS FROM TAWOS

ID Project	Domain	Language	#Issues	Sample
01 Spring XD	Data Processing	Java	3,707	94
32 Mongo Java Driver	Database Driver	Java	3,560	93
11 Titanium Mobile Platform	Mobile Development	JavaScript	3,059	92
25 Hyperledger Indy Node	Blockchain	Python	2,321	91
31 Mongo C++ Driver	Database Driver	C++	2,032	92
Total			14,679	462

- **RQ1.2** To what extent can large language models accurately detect ambiguous issues?
- **RQ1.3** To what extent can large language models accurately detect incomplete issues?

For RQ2, we examine ensemble models:

- **RQ2.1** How effectively does an ensemble of embedding models detect duplicate issues?
- **RQ2.2** How effectively does an ensemble of large language models detect ambiguous issues?
- **RQ2.3** How effectively does an ensemble of large language models detect incomplete issues?

B. Data Collection and Cleaning

1) *TAWOS Dataset*: We used the TAWOS dataset [13], the largest publicly available dataset of agile projects. It contains 508,963 issues from 44 open-source projects across 13 repositories. For each issue, TAWOS records the title, description, type, status, issue links, sprint and version information, making it well-suited for studying backlog quality.

2) *Project Selection*: From 44 TAWOS projects, we selected five medium-sized projects (2,000–3,700 issues) to enable reliable evaluation while keeping manual annotation feasible, spanning four languages (Java, JavaScript, Python, C++) and domains (data-processing, mobile, blockchain, and database-driver systems) to reduce ecosystem-specific bias (Table I).

3) *Data Extraction and Cleaning*: For each issue, we extracted the identifier, title, description, and links from the TAWOS database using SQL. We removed issues with empty

titles, stripped code snippets, stack traces, and formatting from descriptions, and normalized whitespace and casing. For ambiguous and incomplete detection, we used the project files containing issues; for duplicate issue detection, we additionally used the Issue Links table as ground truth, keeping only the link types that denote duplication.

C. Detection of BTD

1) *Duplicate Issue Detection*: We framed duplicate issue detection as binary classification over issue pairs, *i.e.*, given two issues, the model decides whether they describe the same work item and assigns a label and a confidence score. We used four fine-tuned encoders and a cosine-similarity decision because duplicate issues are often phrased differently while sharing meaning, and semantic embeddings capture this better than lexical or rule-based matching.

a) *Encoder Models*: We applied four transformer-based encoders: all-mpnet-base-v2, BERT-base, SpanBERT, and ALBERT-base. We chose all-mpnet-base-v2 because it is trained on over a billion sentence pairs for semantic similarity [23], making it suitable to decide whether two texts refer to the same concept. We chose the other three because prior work found them effective for software requirements [24].

b) *Training and Fine-Tuning*: We fine-tuned each encoder with *CosineSimilarityLoss* [25], a contrastive objective that pulls duplicate pairs together and pushes non-duplicate pairs apart. We built training pairs from ground-truth duplicate links, sampling non-duplicates at a 1:2 positive-to-negative ratio. To offset the limited positive pairs (485 across five projects), we applied T5-based data augmentation [26], generating two paraphrases per positive pair strictly within each training fold to prevent leakage. We trained the models using a five-fold stratified cross-validation, where each fold uses 80% of the data for training and the remaining 20% for testing. We fine-tuned the encoder models using the AdamW optimizer [27] with a learning rate of 2×10^{-5} , batch size 8, 5 epochs, and 10% warm-up steps, and selected the optimal decision threshold by

sweeping from 0 to 1 in steps of 0.01 retaining the value that maximizes the F1-score.

c) *Cosine Similarity*: We measure the semantic closeness of two fine-tuned issue embeddings u and v , computed as:

$$\text{sim}(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \|\mathbf{v}\|} \quad (1)$$

Higher scores indicate greater semantic similarity. Each model produces a score in $[-1, 1]$, which we use as its confidence.

d) *Threshold Sweep*: Since cosine similarity is continuous, we need a threshold above which a pair counts as a duplicate. We set it empirically, sweeping candidate values on the validation set and choosing the one that maximizes F1 for each model and project. This calibration accounts for how differently each encoder distributes similarity scores, giving the best precision–recall trade-off rather than an arbitrary cutoff.

e) *Confidence-Weighted Ensemble Model (CWE)*: We combined the four encoders with a confidence-weighted ensemble that favors the better-performing models. We weighted each model’s confidence by its normalized fine-tuned F1-score, so stronger models contribute more. As shown in Equation (2), the ensemble computes a decision score as the weighted sum of the model confidences, and Equation (3) classifies a pair as duplicate when this score reaches 0.5:

$$\text{decision_score} = \sum_{i=1}^N \left(\frac{\text{acc}_i}{\sum_{j=1}^N \text{acc}_j} \times \text{sim}_i \right) \quad (2)$$

$$\hat{y} = \begin{cases} \text{duplicate} & \text{if decision_score} \geq 0.5 \\ \text{not duplicate} & \text{otherwise} \end{cases} \quad (3)$$

where, in Equation (2) and Equation (3):

- N is the number of models in the ensemble;
- acc_i is the fine-tuned F1-score of model i ;
- sim_i is the confidence score produced by model i for the given pair; and
- $\frac{\text{acc}_i}{\sum_{j=1}^N \text{acc}_j}$ is the normalized weight of model i .

2) *Ambiguous and Incomplete Issues Detection*: The performance of four prompting strategies—*zero-shot*, *few-shot*, *chain-of-thought* (CoT), and *few-shot chain-of-thought* (Few-Shot CoT) using three open-weight LLMs (Llama 3.1 8B, Qwen 2.5 14B, and Mistral 7B¹) was compared to examine the effect of prompt engineering on classification performance. The models were utilized to make binary decisions about the ambiguity and incompleteness of software issues.

a) *Selection of LLMs*: For fair comparison among different model families, as shown in Table II, the performance of three open-weight LLMs with different architectures and sizes is assessed in this study. These models were chosen to reflect architectural differences, training approaches, and the size of the model, which allowed for a more comprehensive examination of the impact of these differences on the detection of ambiguous and incomplete software issues.

¹Llama 3.1 8B, Qwen 2.5 14B and Mistral 7B are openly available open-weight LLMs. Their model weights are available for research and deployment, but their full training sets and training processes are not public.

TABLE II
LLM MODELS USED IN THE EVALUATION SETUP

Model	Parameter Size	Model Family	Provider	Runtime
Llama 3.1	8B	LLaMA family	Meta	Ollama
Qwen 2.5	14B	Qwen family	Alibaba	Ollama
Mistral	7B	Mistral family	Mistral AI	Ollama

TABLE III
PROMPTING STRATEGIES AND THEIR CHARACTERISTICS

Strategy	Examples	Reasoning	Output
Zero-Shot	No	No	Yes / No
Few-Shot	Yes	No	Yes / No
Chain-of-Thought	No	Yes	Final Answer: Yes / No
Few-Shot CoT	Yes	Yes	Final Answer: Yes / No

The models were executed in a local inference environment using the Ollama framework. To achieve reproducibility and deterministic behavior, the temperature parameter was set to zero for all experiments. Each model was assessed with the same prompting strategies to ensure consistency and enable fair comparisons. To guarantee the same generation constraints for all models during evaluation, the maximum length of the output was fixed to 512 tokens. To investigate the performance of inference for the selected models, the experimental evaluations were performed on a system with an NVIDIA RTX 4000 Pro GPU with adequate computational capacities for inference of the selected models.

b) *Prompting Strategies*: To examine how prompt engineering affects the detection of ambiguity and incompleteness, we evaluated four prompting strategies: (1) in *zero-shot prompting*, we gave the model task-specific instructions and the target issue with no examples, and asked it to classify the issue as ambiguous or not ambiguous, and as complete or not complete, (2) in *few-shot prompting*, we added a small set of handcrafted positive and negative examples to the task description, before presenting the target issue, (3) in *chain-of-thought prompting*, we instructed the model to reason step by step about possible signs of ambiguity or incompleteness, then output its decision as “Final Answer: Yes/No.”, and (4) in *few-shot chain-of-thought prompting*, we combined the two preceding strategies: the prompt supplies labeled examples together with their reasoning, and the model reasons about the target issue before producing its final classification. Figure 2 shows a representative few-shot chain-of-thought prompt for ambiguity detection; the other strategies and the incompleteness task follow the same structure with the corresponding instruction and examples.

Table III summarizes the four strategies along with whether each uses examples and reasoning. All four were assessed in the same experimental context to enable a fair comparison of their effectiveness in detecting ambiguity and incompleteness.

c) *Majority-Vote Ensemble Model (MVE)*: In addition to examining the performance of individual LLMs, we examined an ensemble approach to determine whether the predictions of several LLMs could be combined to improve the classification accuracy. The final prediction by the ensemble model was

You are an expert software requirements analyst. Your task is to determine whether a given software issue is AMBIGUOUS. An issue is AMBIGUOUS if it uses vague language, lacks clear context, has unclear scope, or can be interpreted in multiple ways. Think step-by-step before giving your final answer. End your response with 'Final Answer: Yes' or 'Final Answer: No'.

Example 1:

Title: Fix the issue #1
Description: Something is broken. Please fix it.
Reasoning: The issue is vague --- "something is broken" does not specify what component, what behavior is expected, or under what conditions the problem occurs.
Final Answer: Yes

Example 2:

Title: NullPointerException in UserService at line 142
Description: A NullPointerException is thrown in UserService.java at line 142 when userId is null during login.
Reasoning: The issue is clear --- it names the exact file, line, condition, and context.
Final Answer: No

Now classify the following issue:

Title: {issue_title}

Description: {issue_description}

Think step-by-step, then provide your Final Answer (Yes/No):

Fig. 2. Example few-shot chain-of-thought prompt for ambiguity detection (title-with-description mode). {issue_title} and {issue_description} are filled per issue at inference time.

obtained by gathering the predictions from three LLMs for each classification task, ambiguous or incomplete, and selecting the class with the most votes.

D. Evaluation

1) *Ground Truth*: To detect duplicate issues, we derived the ground truth from the Issue Links table in TAWOS. However, no ground truth existed for ambiguous and incomplete issue detection; thus, we created a ground truth through manual annotation by drawing a representative sample from the issues in the selected projects with a 10% margin of error computed at 95% confidence ($Z = 1.96$) and $p = 0.5$ using $n_0 = Z^2 p(1-p)/e^2$, adjusted for each project's finite population size N via $n = n_0/(1+(n_0-1)/N)$; issues were then sampled uniformly at random within each project, as shown in Table I. Four annotators labeled each issue as incomplete, ambiguous, or neither, using both the title-only and title+description inputs.

2) *Evaluation Metrics*: We evaluated the performance of all models using precision, recall, and F1-score per project to show their variation across the projects.

IV. EXPERIMENTS AND RESULTS

Table IV summarizes the experimental setup for the four research questions, including the models, techniques, detection goals, input representations, and evaluation metrics. Across all experiments, we used the issue title as the default input and evaluate performance using precision, recall, and F1-score.

A. Prevalence of Backlog Technical Debt in Agile Projects

Building the annotated dataset allowed us to examine: *to what extent real-world agile projects contain backlog technical debt?* Four annotators (three graduate students and one professional) independently assigned labels for their ambiguity and incompleteness under both input modes (i.e., title-only and title+description) following a *conservative* approach. We assigned 'non-ambiguous' or 'complete' only when at least

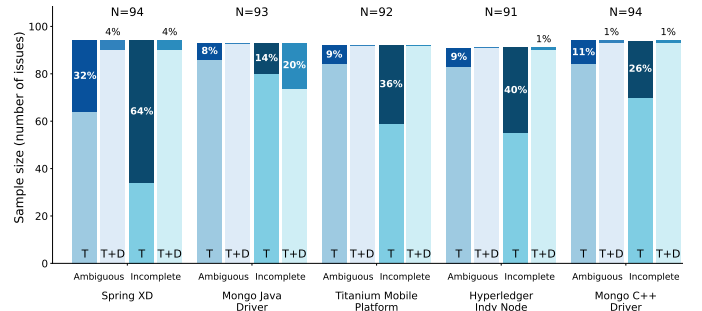


Fig. 3. Ratio of ambiguous and incomplete issues across five projects with title only (T) and with title and description (T+D). N : sample size.

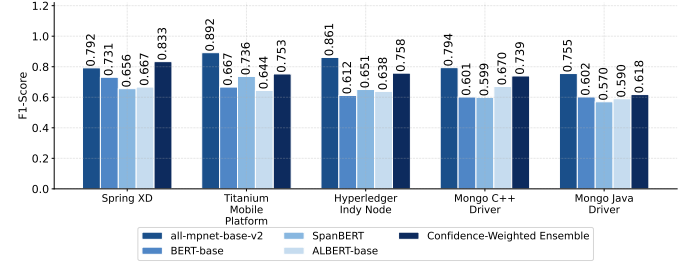


Fig. 4. Duplicate Detection F1-Score Results for the Embedding Models

three out of four (75%) annotators agreed; an issue was labeled 'ambiguous' or 'incomplete' when at least two out of four (50%) annotators agreed. Thus, we assigned a label only when there is sufficiently strong and clear evidence.

Figure 3 presents the prevalence of ambiguous and incomplete issues when annotators considered either the issue title alone (T) or with its description (T+D). Across the five agile projects, when only the issue title is considered, between 8% and 32% of the issues appeared to be ambiguous, with an average of 13.8%. However, with the descriptions added to the context, the issues become easier to understand reducing their ambiguity to 4%, with an average of 1%. This suggests that only the 'title' is not sufficiently representative to understand an issue well, thus, often agile developers need to read the 'description' for better comprehending the issue.

Concerning the 'completeness' of issue titles, between 14% and 64% of titles (36% on average) are labeled as incomplete. This suggests that a significant proportion of issue titles are insufficiently specified. With issue descriptions, the information deficiency improves, i.e., on average only 5.2% of issues were labeled as 'incomplete'.

Titles alone frequently lacked sufficient contextual information, causing more issues to be identified as ambiguous or incomplete. When descriptions were included, prevalence generally declined substantially. The implication is clear: writing clear, self-contained titles, and populating descriptions directly reduces how much BTM an agile project carries.

B. Effectiveness of Individual Language Models (RQ1)

Table V and Figure 4 show the performance of duplicate issue detection across the five projects for the four embedding models.

TABLE IV
EXPERIMENTAL SETUP

RQ	Models / Techniques	Goal	Input	Metrics
RQ1	all-mpnet-base-v2	Duplicate	Title	Precision, Recall, F1
	BERT-base			
	SpanBERT			
	ALBERT-base			
RQ2	Llama 3.1 8B	Ambiguous, Incomplete	Title	
	Qwen 2.5 14B			
	Mistral 7B			
	CWE Model			
RQ3	MVE Model	Ambiguous, Incomplete	Title	
	Zero Shot			
	Few Shot			
	Chain-of-Thought			
RQ4	Few Shot CoT	Ambiguous, Incomplete	Title	
	Best Model (all-mpnet-base-v2)			
	Best Model (Qwen 2.5 14B+Few-Shot)			

TABLE V
PERFORMANCE OF INDIVIDUAL MODELS AND ENSEMBLES ACROSS THE FIVE PROJECTS FOR BTD DETECTION. **BOLD** INDICATES BEST F1.

Project	Model	Detection	Precision	Recall	F1
Spring XD	all-mpnet-base-v2	Duplicate	0.955	0.677	0.793
	BERT-base		0.905	0.613	0.731
	SpanBERT		0.636	0.677	0.656
	ALBERT-base		0.629	0.710	0.667
	CWE Model		0.862	0.807	0.833
	Llama 3.1 8B	Ambiguous	0.333	0.967	0.496
		Incomplete	0.652	1.000	0.790
	Qwen 2.5 14B	Ambiguous	0.434	0.767	0.554
		Incomplete	0.682	0.967	0.800
	Mistral 7B	Ambiguous	0.319	1.000	0.484
		Incomplete	0.652	1.000	0.790
	MVE Model	Ambiguous	0.391	0.833	0.532
		Incomplete	0.645	1.000	0.784
Mongo Java Driver	all-mpnet-base-v2	Duplicate	0.646	0.910	0.755
	BERT-base		0.477	0.814	0.602
	SpanBERT		0.408	0.949	0.570
	ALBERT-base		0.446	0.872	0.590
	CWE Model		0.454	0.968	0.618
	Llama 3.1 8B	Ambiguous	0.118	0.571	0.195
		Incomplete	0.171	1.000	0.292
	Qwen 2.5 14B	Ambiguous	0.235	0.571	0.333
		Incomplete	0.194	0.923	0.320
	Mistral 7B	Ambiguous	0.076	1.000	0.141
		Incomplete	0.149	1.000	0.260
	MVE Model	Ambiguous	0.081	0.857	0.148
		Incomplete	0.153	1.000	0.265

all-mpnet-base-v2 achieved the highest F1-score on four out of five projects, from 76% on the Mongo Java Driver to 89% on Titanium Mobile, outperforming BERT-base, SpanBERT, and ALBERT-base across all projects; the other three encoders cluster around 64%. These scores are conditional on the 1:2 positive-to-negative evaluation pairs used for training and testing and do not reflect the much sparser duplicate rate among all candidate pairs in an unfiltered backlog. Across all models, recall is higher while precision varies between projects; thus, the encoders reliably retrieve true duplicates but differ in the false matches they produce, which is a common challenge in agile because many distinct issues are worded similarly. For ambiguity and incompleteness, Figures 5 and 6 show each model’s best strategy. Overall, Qwen 2.5 14B performs best, at 33%–55% (for ambiguity detection) and 32%–

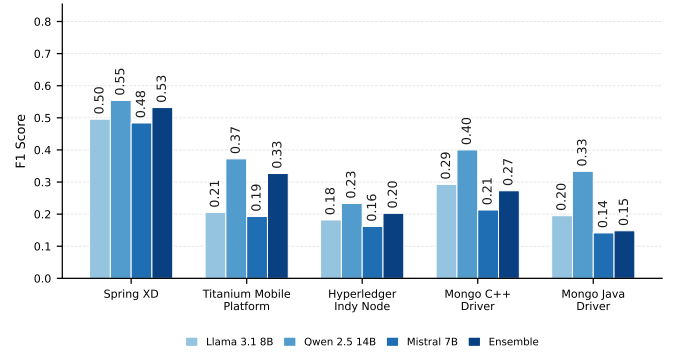


Fig. 5. Ambiguity Detection F1-Score Results Per Individual LLMs

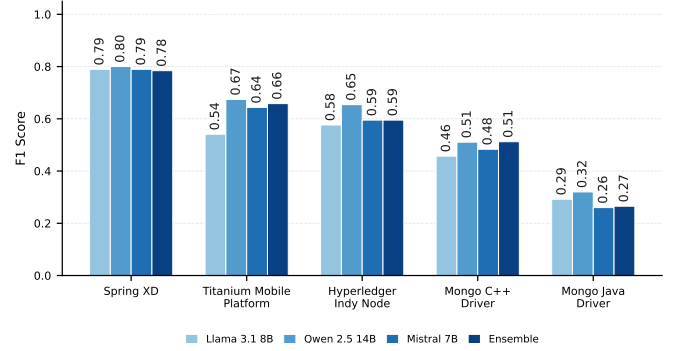


Fig. 6. Incompleteness Detection F1-Score Results Per Individual LLMs

80% (for incompleteness detection). The model performance varies across the projects, performing best on Spring XD, while under-performing on the Mongo Java Driver.

C. Ensemble of Models (RQ2)

Table V and Figure 4 present the performance of confidence-weighted ensemble (CWE) model with the individual encoders. The CWE model had the best F1-score on Spring XD (83%). In other projects, all-mpnet-base-v2 dominates and the weaker encoders inject noisy confidences that reduce the weighted decision score. The MVE ensemble model exhibits a similar pattern. As shown in Figures 5 and 6, the MVE model consistently under-performs Qwen 2.5 14B, achieving F1-scores of only 53% for ambiguity detection and 78% for incompleteness detection on Spring XD. Since Llama 3.1 8B and Mistral 7B have high recall and low precision, their votes outweighed the stronger Qwen 2.5 14B in many cases, affecting the performance of MVE model (inferior to the best performing individual model). In summary, the combination of individual models helps only when models have comparable performance and a confidence-aware or selective scheme is needed when an individual model clearly leads.

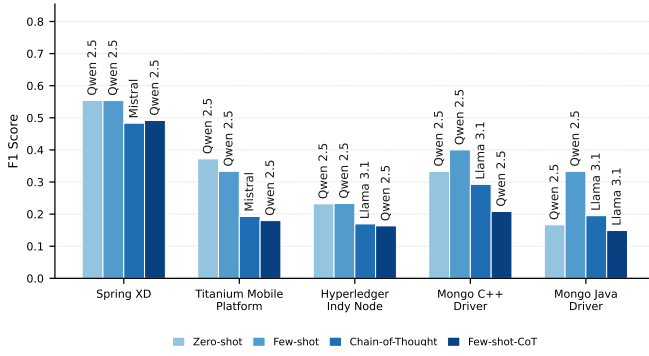


Fig. 7. Various Prompting Strategies for Ambiguity Detection

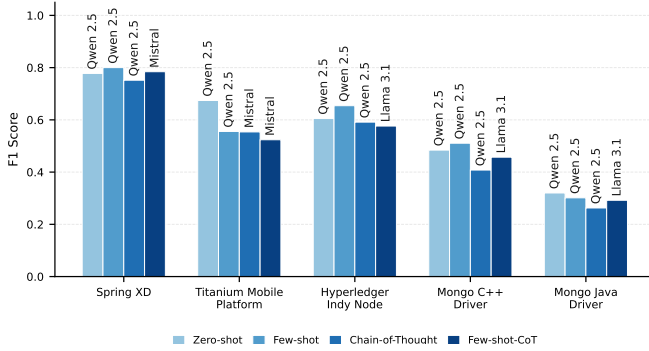


Fig. 8. Various Prompting Strategies for Incompleteness Detection

D. LLMs Under Varying Prompting Strategies (RQ3)

Table VI and Figures 7 and 8 show the results for ambiguous and incomplete issue detection. For ambiguous issue detection, Qwen 2.5 14B performs best under zero-shot and few-shot prompting, with F1-scores from 16.7% to 55.4% across five projects. For incomplete issue detection, the model performs best with these prompting strategies achieving F1-scores from 30.1% to 80%. Notably, the reasoning-based strategies, including CoT and few-shot CoT, under-perform for simpler prompts. Explicit step-by-step reasoning appears to lead models to over-predict the positive class, increasing recall but substantially reducing precision. Few-shot prompting was the strongest approach overall, *i.e.*, a few labeled examples benefit these models more than free-form reasoning.

E. Input Fidelity (RQ4)

Table VII and Figure 9 compare title-only and title-with-description input for all-mpnet-base-v2 on duplicate issue detection. Adding the description improves F1 on every project, between 1% (Mongo Java Driver) and 14% (Mongo C++ Driver, Spring XD), raising the largest cases from approximately 77% and 91%, because the additional information strengthens the semantic-similarity signal. For the LLM tasks, Figures 10 and 11 show a decrease in the F1-score of Qwen 2.5 14B when issue descriptions are included for both ambiguous and incomplete detection. This is consistent with the ground-truth distribution (Figure 3) where far fewer issues are labeled ambiguous or incomplete when the description is

TABLE VI
PERFORMANCE OF LLMs UNDER FOUR PROMPTING STRATEGIES FOR AMBIGUOUS AND INCOMPLETE DETECTION. **BOLD** INDICATES BEST F1.

Project	LLM	Detection	Prompt	Precision	Recall	F1
Llama 3.1 8B	Ambiguous	Zero Shot	Zero Shot	0.250	0.267	0.258
			Few Shot	0.333	0.967	0.496
			CoT	0.318	0.467	0.378
			Few Shot CoT	0.323	1.000	0.488
	Incomplete	Zero Shot	Zero Shot	0.667	0.033	0.064
			Few Shot	0.652	1.000	0.790
			CoT	0.633	0.644	0.639
			Few Shot CoT	0.663	0.950	0.781
Spring XD	Qwen 2.5 14B	Ambiguous	Zero Shot	0.434	0.767	0.554
			Few Shot	0.514	0.600	0.554
			CoT	0.302	0.897	0.452
			Few Shot CoT	0.326	1.000	0.492
	Incomplete	Zero Shot	Zero Shot	0.667	0.933	0.778
			Few Shot	0.682	0.967	0.800
			CoT	0.654	0.883	0.752
			Few Shot CoT	0.638	1.000	0.779
Mistral 7B	Ambiguous	Zero Shot	Zero Shot	0.322	0.967	0.483
			Few Shot	0.319	1.000	0.484
			CoT	0.322	0.967	0.483
			Few Shot CoT	0.315	0.967	0.475
	Incomplete	Zero Shot	Zero Shot	0.648	0.767	0.702
			Few Shot	0.652	1.000	0.790
			CoT	0.625	0.333	0.435
			Few Shot CoT	0.645	1.000	0.784
Llama 3.1 8B	Ambiguous	Zero Shot	Zero Shot	0.037	0.143	0.059
			Few Shot	0.081	0.857	0.148
			CoT	0.118	0.571	0.195
			Few Shot CoT	0.081	1.000	0.149
	Incomplete	Zero Shot	Zero Shot	0.333	0.154	0.211
			Few Shot	0.151	1.000	0.263
			CoT	0.127	0.583	0.209
			Few Shot CoT	0.171	1.000	0.292
Mongo Java Driver	Qwen 2.5 14B	Ambiguous	Zero Shot	0.103	0.429	0.167
			Few Shot	0.235	0.571	0.333
			CoT	0.061	0.714	0.112
			Few Shot CoT	0.072	0.857	0.133
	Incomplete	Zero Shot	Zero Shot	0.194	0.923	0.320
			Few Shot	0.183	0.846	0.301
			CoT	0.151	1.000	0.263
			Few Shot CoT	0.138	1.000	0.243
Mistral 7B	Ambiguous	Zero Shot	Zero Shot	0.069	0.857	0.128
			Few Shot	0.076	1.000	0.141
			CoT	0.066	0.857	0.122
			Few Shot CoT	0.076	1.000	0.141
	Incomplete	Zero Shot	Zero Shot	0.134	0.692	0.225
			Few Shot	0.149	1.000	0.260
			CoT	0.171	0.538	0.259
			Few Shot CoT	0.143	1.000	0.250

included because the description resolves much of the BTD that is apparent from the title alone. Because label prevalence itself shifts substantially between conditions (Figure 3), this F1 comparison conflates a change in model behavior with a change in class balance; Table VII’s precision and recall values should be read alongside F1 for this reason. Due to space constraints, we report results for two representative projects. Complete results for all five projects, along with our code, datasets, and model implementations, are available online.²

V. DISCUSSION

A. RQ1: Effectiveness of Individual Language Models

For duplicate issue detection, all-mpnet-base-v2 consistently outperformed the other encoders, achieving the highest

²<https://zenodo.org/records/20805120>

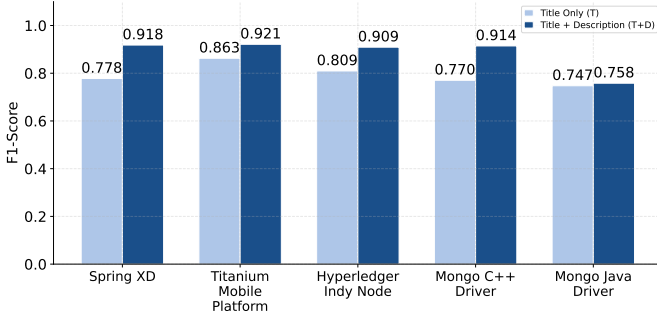


Fig. 9. Duplicate Detection F1-Score Results for the Various Input Modes Using the Best Performing Model.

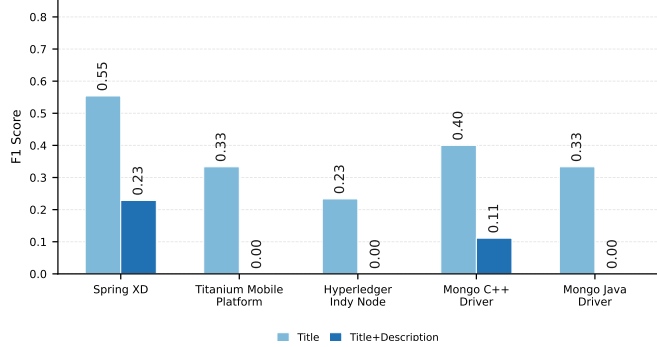


Fig. 10. Ambiguity Detection F1-Score Results for the Various Input Modes Using the Best Performing Model

TABLE VII
PERFORMANCE OF THE BEST-PERFORMING MODELS UNDER TWO INPUT REPRESENTATIONS FOR BTD DETECTION. **BOLD** INDICATES BEST F1.

Project	Best Model	Detection	Input	Precision	Recall	F1
Spring XD	all-mpnet-base-v2	Duplicate	Title	0.913	0.677	0.778
			Title+Description	0.933	0.903	0.918
	Best LLM+Prompt	Ambiguous	Title	0.434	0.767	0.554
			Title+Description	0.129	1.000	0.229
		Incomplete	Title	0.682	0.967	0.800
			Title+Description	0.118	0.500	0.191
Mongo Java Driver	all-mpnet-base-v2	Duplicate	Title	0.634	0.910	0.747
			Title+Description	0.620	0.974	0.758
	Best LLM+Prompt	Ambiguous	Title	0.235	0.571	0.333
			Title+Description	0.000	0.000	0.000
		Incomplete	Title	0.194	0.923	0.320
			Title+Description	0.240	1.000	0.388

F1-score in four of the five projects, with a maximum of 89% on Titanium Mobile. Its superior performance is primarily due to pre-training on over a billion semantically related sentence-pairs using a contrastive objective optimized for semantic similarity, which closely aligns with duplicate issue detection. In contrast, BERT-base, SpanBERT, and ALBERT-base, although effective in software engineering tasks, are not specifically optimized for sentence-level similarity and consistently achieved lower precision scores. Overall, the encoders maintained high recall but varied in precision, indicating that semantically similar yet distinct backlog items often led to false positives.

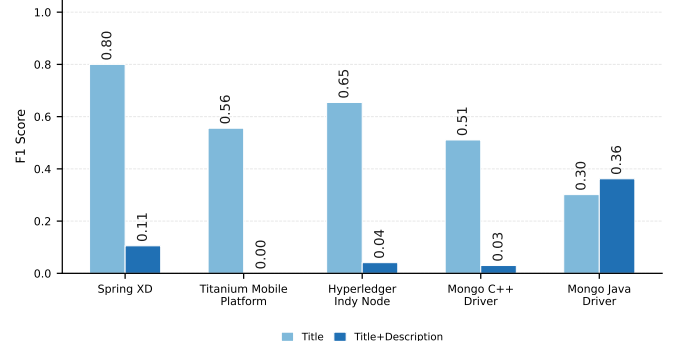


Fig. 11. Incompleteness Detection F1-Score Results for the Various Input Modes Using the Best Performing Model

TABLE VIII
EXAMPLES OF BTD DETECTION ACROSS THE THREE TASKS

Task	Issue Title(s)	True Label	Predicted Label
Duplicate	Support pagination in <code>list()</code> command for streams vs. <code>StreamsController</code> to return paged results for <code>list()</code>	Duplicate	Duplicate
	Decouple messagebus dependencies vs. Decouple transport from DIRT	Duplicate	Not Duplicate
Ambiguous	CompositeModuleDefinition should contain more than List	Ambiguous	Ambiguous
	Error in correlation strategy in <code>aggregator.xml</code>	Unambiguous	Ambiguous
Incomplete	Rename metrics repositories <code>setValue(x)</code>	Incomplete	Incomplete
	Change URL to access UI from browser	Incomplete	Complete

For ambiguous and incomplete issue detection, model performance was task-dependent. Using title-only input, Qwen 2.5 14B achieved the highest mean F1-scores for both ambiguous (29%) and incomplete (53%) issue detection, outperforming Llama 3.1 8B and Mistral 7B. Incomplete issues were detected more effectively than ambiguous issues, *i.e.*, missing information is more explicitly reflected in issue titles, whereas ambiguity detection requires deeper contextual interpretation. Nevertheless, the F1 scores remained between 55% and 80%, indicating that current LLMs are not yet sufficiently reliable as standalone classifiers. As shown in Table VIII, errors primarily arose from misinterpreting project-specific terminology as ambiguous and failing to recognize omitted implementation details when only titles were available.

B. RQ2: Ensemble of Models

For duplicate issue detection, the CWE model did not consistently outperform the best individual model. Although the ensemble weighted each encoder's prediction according to its fine-tuned F1-score, it surpassed all-mpnet-base-v2 only on the Spring XD dataset, where the individual encoders exhibited more comparable performance. This suggests that when one model substantially outperforms the others, as all-mpnet-base-v2 did, combining it with lower-performing models can reduce its predictive performance by incorporating less reliable confidence estimates. Consequently,

confidence-weighted ensembling appears most effective when the constituent models demonstrate similar predictive performance. A similar trend was observed for the MVE model as it failed to improve upon the best individual model, with an average F1-score of 52% for incomplete issue detection compared with 53% for Qwen 2.5 14B. The limited effectiveness of the ensemble can be attributed to the lower precision of the weaker LLMs, whose predictions diluted the performance gains achieved by the stronger model. These findings indicate that simple majority voting is insufficient when the constituent models differ substantially in predictive reliability.

C. RQ3: LLMs Under Varying Prompting Strategies

For ambiguous issue detection, Qwen 2.5 14B achieves the best performance under both zero-shot and few-shot prompting across all five projects. Under chain-of-thought prompting, Mistral 7B performs best on Spring XD, while Llama 3.1 8B performs best on Hyperledger Indy Node, Mongo C++ Driver, and Mongo Java Driver. Under few-shot chain-of-thought prompting, Qwen 2.5 14B achieves the best performance on four projects, with Llama 3.1 8B performing best on Mongo Java Driver. For incomplete issue detection under zero-shot and few-shot prompting, Qwen 2.5 14B consistently performs best across all projects. Under chain-of-thought prompting, Qwen 2.5 14B achieves the best performance on four projects and ties with Mistral 7B on the Titanium Mobile Platform dataset. Overall, Mistral 7B performs better on Spring XD and Titanium Mobile Platform, while Llama 3.1 8B performs better on Hyperledger Indy Node, Mongo C++ Driver, and Mongo Java Driver.

With title-only input, few-shot prompting achieves the best overall performance, with the highest mean F1-scores for ambiguous (28%) and incomplete (53%) issue detection. Contrary to expectations, chain-of-thought and few-shot chain-of-thought prompting do not consistently improve performance and often yield lower F1-scores than less complex prompting strategies, which may be attributed to over-prediction of the positive class. Furthermore, including labeled examples has a greater effect on ambiguity detection than on incompleteness detection, suggesting that exemplars are particularly beneficial for the inherently subjective task of identifying ambiguity.

D. RQ4: Input Representation

For duplicate issue detection, including the issue description alongside the title improves detection performance. On the Mongo Java Driver, `all-mpnet-base-v2` increases from an F1-score of 74.7% (title only) to 75.8% (title and description), and the gains are larger elsewhere, for example, Spring XD rises from 77.8% to 91.8%. The description gives the encoder extra context, which helps it tell real duplicates apart from issues that only share a similar title. The LLM tasks show the opposite: descriptions made performance worse. Thus, the value of extra input depends on the task; more context helps similarity matching but distracts models that must make finer judgments. For BTD detection with LLMs, titles alone yielded better performance than titles combined with descriptions.

E. Implications for Researchers and Developers

The literature on backlog technical debt in agile projects lacks manually labeled data for ambiguous and incomplete issue detection, which has limited empirical study. Our work contributes a manually annotated dataset of ambiguous and incomplete issues across five projects, which may facilitate further research on automated BTD detection. Our findings show that detecting BTD requires models that go beyond keyword overlap and capture semantic meaning, project context, and developer intent, since the embedding model pre-trained for semantic similarity clearly outperformed the others. Model performance varies across projects, which indicates that BTD detection should be evaluated per project rather than combined, as combined scores hide this variation. For developers, our analysis shows that a substantial proportion of issues appear ambiguous or incomplete when read from the title alone, but that this share drops sharply once a description is present (for incomplete issues, from 64% to 4% in Spring XD). We therefore recommend that developers provide self-contained, informative titles and consistently populate issue descriptions, as this directly reduces how much technical debt a backlog appears to carry and how much must be resolved manually. Finally, the ‘*ambiguous*’ and ‘*incomplete*’ labels in our study are inherently interpretive, and annotators base their judgments on their individual interpretations of the issue text, which may introduce annotation bias and inter-annotator variability. It is therefore important to account for these factors when developing models from manually annotated datasets.

F. Threats to Validity

Internal validity is influenced by two decisions in our experimental setup: (1) the encoder’s decision threshold is tuned for each project and model, and the chosen value directly affects F1. To mitigate this threat, we selected the threshold using a separate validation fold, rather than the test data. However, it still affects the reported scores; (2) the LLM results depend on prompt formulation. Prompt phrasing strongly affects model outputs, so our findings are tied to the specific prompts used in this study, and alternative prompts could lead to different results. *External validity* is limited by the projects we studied. All five are open-source projects whose issues are managed using Jira, so they may follow conventions that differ from those of industrial or closed-source backlogs. Such backlogs differ in development processes, review practices, and writing style, and our prevalence estimates and model rankings may not generalize directly to these settings. *Construct validity* depends on how we built the ground truth. For duplicate issue detection, we relied on developer-assigned issue links in TAWOS, which record only the duplicate issues developers noticed. Genuine duplicates that were never linked are counted as non-duplicates, which makes precision look higher and recall lower than they truly are. For ambiguous and incomplete issues, the labels are subjective: each reflects the annotator’s interpretation rather than the author’s intended meaning. Further, our title-only condition treats a title as a proxy for the full backlog item, which may conflate

under-specified titles with genuinely incomplete backlog items whose descriptions were simply not yet considered.

VI. CONCLUSION AND FUTURE WORK

This paper presents findings on the automated detection of BTD, focusing on duplicate, ambiguous, and incomplete issues in agile software development. Using the TAWOS dataset from five open-source agile projects, we fine-tuned four embedding models for duplicate detection and evaluated three open-weight large language models for the detection of ambiguous and incomplete issues under various configurations, including individual models, ensembles, prompting strategies, and input representations. For duplicate detection (RQ1), `all-mpnet-base-v2` was the most effective model, achieving the highest F1-score on four of the five projects and outperforming BERT-base, SpanBERT, and ALBERT-base. For ambiguity and incompleteness detection, Qwen 2.5 14B achieved the strongest overall performance, although its absolute performance remained modest. The confidence-weighted ensemble did not consistently outperform the best individual encoder (RQ2). Regarding prompting strategies (RQ3), few-shot and zero-shot prompting yielded the best overall performance. In terms of input representations (RQ4), performance was task-dependent: incorporating descriptions improved duplicate detection but reduced performance in ambiguity and incompleteness detection. Overall, these findings provide practical guidance for selecting models, prompting strategies, and input representations for automated BTD detection. Future work will extend the evaluation to additional open-source datasets, investigate domain adaptation and fine-tuning approaches to improve LLM performance on ambiguity and incompleteness detection, and explore adaptive ensemble and input selection strategies for both intra- and cross-project generalization.

REFERENCES

- [1] N. S. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, "Identification and management of technical debt: A systematic mapping study," *Information and Software Technology*, vol. 70, pp. 100–121, 2016.
- [2] A. Melo, R. Fagundes, V. Lenarduzzi, and W. B. Santos, "Identification and measurement of requirements technical debt in software development: A systematic literature review," *Journal of Systems and Software*, vol. 194, p. 111483, 2022.
- [3] A. Martini, T. Besker, and J. Bosch, "Process debt: a first exploration," in *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, 2020, pp. 316–325.
- [4] T. Dingsoyr, S. Nerur, V. Balijepally, and N. B. Moe, "A decade of agile methodologies: Towards explaining agile software development," *Journal of Systems and Software*, vol. 85, no. 6, pp. 1213–1221, 2012, special Issue: Agile Development.
- [5] D. Bijlsma, M. A. Ferreira, B. Luijten, and J. Visser, "Faster issue resolution with higher technical quality of software," *Software Quality Journal*, vol. 20, no. 2, pp. 265–285, 2012.
- [6] S. Saleem, M. N. Asim, and A. Dengel, "Passionnet: An innovative framework for duplicate and conflicting requirements identification," *Expert Systems with Applications*, vol. 293, p. 128684, 2025.
- [7] J. He, L. Xu, M. Yan, X. Xia, and Y. Lei, "Duplicate bug report detection using dual-channel convolutional neural networks," in *Proceedings of the 28th International Conference on Program Comprehension*, ser. ICPC 20. New York, NY, USA: Association for Computing Machinery, 2020, p. 117–127.
- [8] S. Bashir, A. Ferrari, A. Khan, P. E. Strandberg, Z. Haider, M. Saadatmand, and M. Bohlin, "Requirements ambiguity detection and explanation with llms: An industrial study," in *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2025, pp. 620–631.
- [9] M. Weiss, A. G. Gómez, and P. Tonella, "Generating and detecting true ambiguity: a forgotten danger in DNN supervision testing," *Empirical Software Engineering*, vol. 28, no. 6, p. 146, 2023.
- [10] D. Luitel, S. Hassani, and M. Sabetzadeh, "Improving requirements completeness: automated assistance through large language models," *Requirements Engineering*, vol. 29, no. 1, pp. 73–95, 2024.
- [11] M. Osama, A. Zaki-Ismail, M. Abdelrazek, J. Grundy, and A. Ibrahim, "Score-based automatic detection and resolution of syntactic ambiguity in natural language requirements," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 651–661.
- [12] G. Malik, M. Cevik, A. Başar, and D. Parikh, "Supervised semantic similarity-based conflict detection algorithm: S3cda," in *2025 IEEE International Conference on Collaborative Advances in Software and COmputiNg (CASCON)*, 2025, pp. 1–10.
- [13] V. Tawosi, A. Al-Subaihini, R. Moussa, and F. Sarro, "A versatile dataset of agile open source software projects," in *Proceedings of the 19th International Conference on Mining Software Repositories*, ser. MSR '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 707–711.
- [14] S. Reddivari and J. Wolbert, "Calculating requirements similarity using word embeddings," in *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2022, pp. 438–439.
- [15] A. Kukkar, R. Mohana, Y. Kumar, A. Nayyar, M. Bilal, and K.-S. Kwak, "Duplicate bug report detection and classification system based on deep learning technique," *IEEE Access*, vol. 8, pp. 200 749–200 763, 2020.
- [16] C. E. Laney, A. Barovic, and A. Moin, "Automated duplicate bug report detection in large open bug repositories," in *2025 IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2025, pp. 450–458.
- [17] M. Götharsson, K. Stahre, G. Gay, and F. G. de Oliveira Neto, "Exploring the role of automation in duplicate bug report detection: An industrial case study," in *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*, ser. AST 24. New York, NY, USA: Association for Computing Machinery, 2024, p. 193–203.
- [18] R. Izhar, S. N. Bhatti, and S. A. Alharthi, "Bridging precision and complexity: A novel machine learning approach for ambiguity detection in software requirements," *IEEE Access*, vol. 13, pp. 12 014–12 031, 2025.
- [19] M. R. Asadabadi, M. Saberi, O. Zwikael, and E. Chang, "Ambiguous requirements: A semi-automated approach to identify and clarify ambiguity in large-scale projects," *Computers and Industrial Engineering*, vol. 149, p. 106828, 2020.
- [20] S. Ezzini, S. Abualhaija, C. Arora, and M. Sabetzadeh, "Automated handling of anaphoric ambiguity in requirements: a multi-solution study," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE 22. New York, NY, USA: Association for Computing Machinery, 2022, p. 187–199.
- [21] L. Bo, W. Ji, X. Sun, T. Zhang, X. Wu, and Y. Wei, "ChatBR: Automated assessment and improvement of bug report quality using ChatGPT," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2024, pp. 1472–1483.
- [22] A. Patil and A. Jadon, "Next-generation bug reporting: Enhancing development with AI automation," in *Proceedings of the 10th International Conference on Signal Processing and Communication (ICSC)*, 2025, pp. 487–493.
- [23] Sentence-Transformers, "all-mpnet-base-v2 model card," <https://huggingface.co/sentence-transformers/all-mpnet-base-v2>, accessed: 2026-06-20.
- [24] Q. Zhang, C. Fang, Y. Xie, Y. Zhang, S. Yu, W. Sun, Y. Yang, and Z. Chen, "A survey on large language models for software engineering," *Science China Information Sciences*, vol. 69, no. 4, p. 141102, 2026.
- [25] A. Draganov, S. Vadgama, and E. J. Bekkers, "The hidden pitfalls of the cosine similarity loss," in *ICML 2024 Workshop on High-dimensional Learning Dynamics (HiLD)*, 2024.
- [26] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," vol. 21, no. 140, 2020, pp. 1–67.
- [27] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *International Conference on Learning Representations (ICLR)*, 2019.