arXiv:2608.18280v1 [cs.SE] 18 Aug 2026

What Makes Software Issue Resolution Tasks Difficult for Agents?

Ebtesam Al-Haque
Department of Computer Science, George Mason University, Fairfax, VA

Brittany Johnson
Department of Computer Science, George Mason University, Fairfax, VA

Abstract

Background. Advances in agentic systems are simultaneously, and rapidly, saturating benchmarks. Despite this often discussed phenomena, benchmark scores remain difficult to interpret due to the lack of control and characterization of task difficulty. More specifically, we currently have little understanding of what makes one task harder than another, and to what extent task difficulty is predictable from static task properties. **Aims.** We propose a measurement framework to investigate and systematically quantify what structural properties of software tasks correspond to agent success rates for issue resolution tasks. **Method.** We conducted a large scale empirical study on CoderForge-Preview, the largest open dataset of coding agent trajectories to date, by extracting features across task patch, repository and prompt. We evaluated the predictive power of each feature against task outcomes using ensemble methods, SHAP attribution, and effect size analysis. **Results** We found that task difficulty is substantially predictable from static features ($AUC = 0.863$) and is largely driven by patch fragmentation and repository scale. Prompt linguistic features become visible among top contributors for tasks in the mid-band, revealing a layered structure of difficulty. **Conclusion.** The difficulty of an issue resolution task is encoded in its structure. This enables static, pre-hoc difficulty estimation and lays the groundwork for difficulty-controlled benchmark construction for evaluation of agents.

2012 ACM Subject Classification Software and its engineering → Software creation and management; Software and its engineering → Software verification and validation; Computing methodologies → Model development and analysis; General and reference → Empirical studies

Keywords and phrases software, oss, agents, empirical studies

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.3

Category Technical Track Paper

1 Introduction

AI-based coding agents are rapidly advancing, with the best open-source systems now resolving more than half the tasks on SWE-bench Verified [28]. The aggregate solve rate, however, provides no account of *why* a task is hard. Understanding this in agent outcomes is critical as agentic decision-making systems become more widely deployed. Prior work has invested heavily in building larger and more diverse task collections [12, 37, 3] and in validating the correctness of existing ones [1, 28], but has not systematically characterized task difficulty from static properties. Item Response Theory (IRT) [17] offers a framework for difficulty estimation, and recent work applies it to the agentic coding setting using LLM embeddings and rubric scores [6]. However, their approach involves nondeterministic features that complicate reproducibility.

To address this gap, we conducted a large-scale empirical study on CoderForge-Preview [2] to answer the following research questions:

RQ1 Can static task features predict agent success?

RQ2 Which feature groups account for predictive performance?

© Ebtesam Al Haque and Brittany Johnson;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón; Article No. 3; pp. 3:1–3:20

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

RQ3 Which individual features are most strongly associated with agent success?

We represent task difficulty as agent success rate and engineer several features across three groups: the gold patch, the repository, and the natural-language prompt, grounded in prior literature on code change complexity, software navigation, and requirements engineering. We then train models, measure their predictive accuracy, ablate feature groups, and decompose predictions using SHAP values to identify which features drive difficulty and how.

To this end, our work makes the following contributions:

- We introduce a deterministic measurement framework for characterizing issue-resolution task difficulty from static patch, repository, and prompt properties.
- We provide large-scale evidence that agent success is predictable from static task structure.
- We identify factors that influence task success and briefly discuss how this varies across different levels of difficulty.

2 Related Work

2.1 Agentic Benchmarks for Issue Resolution

SWE-bench [14] established the standard task for evaluating LLM-based agents: given a repository and an issue description, produce a patch that passes a held-out test suite. Progress on the benchmark has been rapid, but so have concerns about what aggregate solve rates actually measure. SWEBench+ [1] and UTBoost [38] independently found that a large fraction of resolved instances involved solution leakage or tests too weak to discriminate correct from incorrect patches; once filtered, resolution rates drop by more than half. SWE-bench Verified [28] addressed this through manual developer vetting of 500 instances. Subsequent work has moved toward harder settings: long-horizon multi-file tasks in SWE-bench Pro [5], terminal-interaction tasks in Terminal Bench [24], and software performance optimization in GSO [31]. In parallel, training-oriented environments have scaled task supply: R2E-Gym [12], SWE-Smith [37], and SWE-Rebench [3] together generate tens of thousands of executable tasks from real commits. CoderForgePreview [2], the dataset we study, assembles 51K tasks from all three sources. What none of this work provides is an account of *why* some tasks are harder than others. Our study is the first to systematically quantify which structural properties of a task derived from the required patch, the repository, and the issue description drive agent success rates on a large scale.

2.2 Predicting Task Difficulty

Item Response Theory (IRT) has become the standard tool for modeling task-level difficulty in language-model evaluation [17], with recent work using it to reduce evaluation cost through adaptive task selection [29, 11, 32]. Ge et al. [6] bring IRT explicitly into the agentic coding setting, augmenting the Rasch model with LLM embeddings and LLM as-a-judge rubric scores to predict task difficulty on SWE-bench Verified, SWE-bench Pro, Terminal-Bench, and GSO.

Our features are fully deterministic measures grounded in empirical software engineering constructs rather than LLM encodings. This makes our difficulty scores reproducible without any model inference. The structural features we identify could serve as cost effective, interpretable inputs to an IRT difficulty predictor of the type they describe.

2.3 Code Change Complexity and Fault Localization

The patch and repository features we engineer are grounded in a long empirical literature on what makes software changes hard to understand and fix. Munson and Elbaum [26] and Nagappan and Ball [27] established code churn as a reliable predictor of defect density. The just-in-time defect prediction literature [16, 25, 39] further shows that it is not raw volume but *diffusion*, how scattered a change is across files and hunks, that most consistently distinguishes defect-inducing from clean commits. We carry these measures into the agentic setting and find similar patterns.

2.4 Linguistic Properties

The natural-language prompt an agent receives is functionally a software requirement. Two decades of work has discovered ways to measure specification quality and clarity. They identified coordination ambiguity, referential ambiguity, and attachment ambiguity as the three most consequential sources of misinterpretation in natural-language requirements [10, 30]. Recent empirical work shows that LLMs struggle to reliably identify ambiguous requirements [18] and that ambiguous specifications increase code generation uncertainty by introducing multiple valid interpretations [34]. What none of this work establishes is the *relative* importance of specification quality compared to structural task properties, or whether linguistic features carry independent signal once structural features are controlled for. We directly address this in our work.

3 Method

3.1 Dataset

CoderForge-Preview [2] is the largest open dataset of coding agent trajectories to date, consisting of 258K test-verified trajectories spanning 51K tasks across 1,655 repositories. Tasks are drawn from three sources: R2E-Gym (4,216 tasks) [12], SWE-Smith (37,221 tasks) [37], and SWE-Rebench (9,764 tasks) [3]. We generated trajectories using Qwen3-Coder-480B [35], one of the best-performing open-source models on agentic coding benchmarks, operating within an OpenHands v0.52.1 scaffold where the agent iteratively issues bash commands and file edits for up to 100 steps within an isolated Docker container. This makes CoderForge-Preview well-suited for studying the behavioral characteristics of capable coding agents rather than the failure modes of weaker models. They generated multiple trajectories per task (up to 8), and retained those whose final patches pass all repository tests as successful. For our work, we excluded R2E-Gym tasks due to lack of a compatible patch format. We also excluded tasks for which we could not map to the repository using the GitHub API. Our experiments therefore draw on 45,769 tasks from SWE-Smith and SWE-Rebench spanning 1,553 repositories, with a mean of 4.8 trajectories per task. 49.8% of tasks are solved by every trajectory (*always pass*), 32.5% by none (*always fail*), and 17.7% exhibit mixed outcomes where some but not all trajectories succeed.

3.2 Feature Engineering

We engineered features across three groups, each capturing a distinct dimension of task difficulty. **Patch features** characterize what a correct solution looks like. **Repository features** characterize the structural environment the agent must navigate. **Prompt features** characterize the natural-language specification the agent must interpret. An overview of

the three groups and their sub-constructs is shown in Table 1. Tables 2–4 list every feature with a description. Features marked † were dropped during the feature selection process (Section 3.3); the 54 retained features are used in all models.

Table 1 Feature taxonomy overview. Each cell shows the sub-construct and its feature count

Patch (18)	**Repository** (14)	**Prompt** (31)
Edit volume (3)	Scale (5)	Component load (4)
Edit fragmentation (4)	Directory structure (4)	Coordinative complexity (6)
File scope (4)	Naming ambiguity (1)	Coordination ambiguity (5)
Churn distribution (3)	Test infrastructure (3)	Attachment ambiguity (7)
Spatial distribution (4)	Documentation (1)	Referential ambiguity (4)
		Cohesion (5)

3.2.1 Patch Features

The patch group captures *solution complexity*: how much the agent must find, understand, and change to resolve an issue. The features here draw on the JIT defect prediction constructs discussed in Section 2.3.

Table 2 Patch features

Sub-construct	**Feature**	**What it measures**
Edit volume	`patch_lines_added`	Lines inserted in the gold patch
	`patch_lines_deleted`	Lines removed in the gold patch
	`patch_lines_changed_total`†	Total churn
Edit fragmentation	`patch_num_hunks`	Total disjoint edit blocks
	`patch_edit_fragmentation`	Hunks per changed file
	`patch_hunk_gap_mean`	Mean unchanged lines between consecutive hunks
	`patch_test_files_changed`	Test files in the diff
File scope	`patch_src_files_changed`	Non-test source files changed
	`patch_python_files_changed`	Python files changed
	`patch_extensions_changed_count`	Distinct file types touched
	`patch_num_files_changed`†	Total files changed
Churn distribution	`patch_churn_gini_over_files`	Edit concentration across files
	`patch_is_single_file`	Binary: all edits in one file
	`patch_entropy_over_files`†	Edit entropy
Spatial distribution	`patch_primary_file_depth`	Directory depth of most-edited file
	`patch_num_unique_dirs_changed`	Distinct directories touched
	`patch_mean_file_depth`†	Mean depth of changed files
	`patch_max_file_depth`†	Max depth of changed files

3.2.2 Repository Features

The repository group captures *navigation difficulty*: how hard it is to locate the relevant code before making any edit. Fault localization is a prerequisite for patch generation; before an agent can write a fix, it must identify which files and lines are relevant to the issue. We represent navigation difficulty through two primary sub-constructs. *Scale* features (file count, codebase size, directory count) reflect the size of the search space the agent must traverse. *Directory structure* features (root-level breadth, maximum and mean nesting depth) reflect the structural complexity of that space. A *naming ambiguity* feature (basenames shared by two or more files) captures another layer of complexity in identifying the right file to edit. *Test infrastructure* and *documentation* features capture repository organization conventions that may provide or withhold contextual signals useful for localization.

Table 3 Repository features.

Sub-construct	Feature	What it measures
Scale	`repo_file_count`	Total files in the repository
	`repo_python_file_count`	Python files
	`repo_total_known_size`	Codebase size in bytes
	`repo_mean_known_file_size`	Mean file size in bytes
	`repo_dir_count`	Total directory count
Directory structure	`repo_top_level_dir_count`	Breadth at root level
	`repo_max_depth`	Maximum directory nesting depth
	`repo_mean_file_depth`	Mean file nesting depth
	`repo_test_ratio`	Fraction of files that are tests
Naming ambiguity	`repo_basename_collision_count`	Basenames shared by ≥2 files
Test infrastructure	`repo_test_file_count`	Number of test files
	`repo_has_tests_dir`	Dedicated `tests/` directory present
	`repo_has_examples_dir`	`examples/` directory present
Documentation	`repo_has_docs_dir`	`docs/` directory present

3.2.3 Prompt Features

The prompt group captures *specification interpretability*: whether the agent can derive an action plan from the issue text. We draw on the RE literature's taxonomy of natural-language defects to motivate four sub-constructs. *Component load* features measure how much information the agent must integrate. *Coordinative complexity* features represent structural processing difficulty at the sentence level, motivated by Gibson's Dependency Locality Theory [8] and the finding that syntactically complex prompts increase LLM code generation uncertainty [18, 34]. *Ambiguity* features represent the three defect types identified by the RE literature as most consequential for misinterpretation: coordination ambiguity (scope of conjunctions), attachment ambiguity (which constituent a modifier modifies), and referential ambiguity (which noun phrase a pronoun resolves to) [10, 36]. *Cohesion* features measure lexical and semantic connectivity between adjacent sentences [9, 22].

3.3 Feature Selection

We applied iterative Variance Inflation Factor (VIF) [4] filtering with threshold 10 [21], dropping the highest-VIF feature in each iteration until all scores fall below the threshold. This process dropped 9 features (marked † in Tables 2–4), leaving 54 features with a maximum VIF of 7.59. The same feature set is used for all three models.

3.4 Predictive Modeling

3.4.1 Outcome Variables.

For each task, we compute three outcome variables: **(1) pass_rate**: the fraction of runs that passed all tests (continuous, $[0, 1]$; mean = 0.593, SD = 0.455). **(2) any_success** (*pass@k*): an indicator of whether at least one run succeeded (positive rate: 67.5%). **(3) maj_success** (majority success): an indicator of whether the majority ($\geq 50\%$) of runs succeeded (positive rate: 61.7%).

3.4.2 Data Split.

We split the dataset at the task level into training and held-out test sets using an 80/20 split, resulting in 36,615 training tasks and 9,154 test tasks.

Table 4 Prompt features.

Sub-construct	Feature	What it measures
Component load	`prompt_len_bpe`	Prompt length in BPE tokens
	`prompt_sent_count`	Number of sentences
	`prompt_clauses_per_sentence`†	Clauses per sentence
	`prompt_content_tokens_per_sentence`†	Content tokens per sentence
Coordinative complexity	`prompt_dep_distance_mean`	Mean linear dependency distance
	`prompt_dep_distance_max`	Max linear dependency distance
	`prompt_branching_factor_mean`	Mean syntactic branching factor
	`prompt_clause_nesting_depth_mean`	Mean clause nesting depth
	`prompt_clause_length_mean`	Mean clause length
	`prompt_clausal_branching_factor`	Mean clause-level branching
Coordination ambiguity	`prompt_coordination_density`	Conjunctions per clause
	`prompt_mean_conj_chain_len`	Mean conjunction chain length
	`prompt_max_conj_chain_len`	Max conjunction chain length
	`prompt_verb_coordination_ratio`	Conjunctions with verb heads
	`prompt_noun_coordination_ratio`	Conjunctions with noun heads
Attachment ambiguity	`prompt_pp_density_per_noun`	PP count per noun
	`prompt_pp_density_per_clause`	PP count per clause
	`prompt_mean_pp_chain_depth`	Mean PP nesting depth
	`prompt_max_pp_chain_depth`	Max PP nesting depth
	`prompt_multi_pp_head_ratio`	Heads with $\geq$2 attached PPs
	`prompt_attachment_competition_ratio`	Heads with $\geq$2 competing dependents
	`prompt_mean_competing_dependents_per_head`	Mean competing dependents per head
Referential ambiguity	`prompt_pronouns_per_sentence`	Pronouns per sentence
	`prompt_pronoun_candidate_antecedent_density`	Candidate antecedents per pronoun
	`prompt_multi_antecedent_pronoun_ratio`	Pronouns with $\geq$2 antecedents
	`prompt_pronoun_ratio`†	Pronoun fraction
Cohesion	`prompt_adjacent_lemma_overlap_mean`	Lemma overlap between adjacent sentences
	`prompt_sentence_embedding_sim_mean`	Mean cosine sim of adjacent sentences
	`prompt_sentence_embedding_sim_min`	Min cosine sim of adjacent sentences
	`prompt_entity_reuse_ratio`	Entity reuse across sentences
	`prompt_coref_chain_len_mean`†	Coreference chain length

Table 5 Outcome variable distributions (n = 45,7695 tasks).

(a) Continuous outcome

Outcome	Mean	SD
`pass_rate`	0.593	0.455

(b) Binary outcomes

Outcome	Positive rate	Imbalance ratio
`any_success`	67.5%	2.08
`maj_success`	61.7%	1.61

3.4.3 Model.

We trained and three model classes for each outcome. We selected hyperparameters using `RandomizedSearchCV` with 4-fold inner cross-validation and 28 iterations per model group, using ROC-AUC as the classification scoring criterion and negative mean squared error for regression. All hyperparameters reported below reflect the tuned values found by this procedure.

- **Binary classification models.** For `any_success` and `maj_success`, we train: (i) Logistic Regression with L2 regularization and standardized features ($C = 0.013$ for `any_success`; $C = 0.037$ for `maj_success`); (ii) Random Forest with 500 trees, maximum depth 16, `max_features`= 0.25, and minimum 8 samples per leaf; and (iii) XGBoost with 400 trees, maximum depth 8, learning rate 0.03, and minimum child weight 2. Ensemble methods, particularly Random Forest and gradient boosting, consistently outperform linear classifiers in software defect prediction tasks [19, 7], motivating their inclusion alongside the interpretable linear baseline.
- **Regression models.** For `pass_rate`, we train: (i) Ridge Regression with $\alpha = 115.67$ and standardized features; (ii) Random Forest Regressor with 500 trees, unlimited depth, `max_features`= 0.25, and minimum 4 samples per leaf; and (iii) XGBoost Regressor with a squared-error objective and the same hyperparameters as the classifiers.

3.4.4 Evaluation

Output Metrics. We evaluated the models on the held-out test set. Classification performance is reported using AUC–ROC, Precision–Recall AUC (PR–AUC), Brier score, F1, and Matthews Correlation Coefficient (MCC). Regression performance is reported using RMSE, MAE, and R^2. We additionally perform 10-fold cross-validation on the training set for the XGBoost model and report mean ± SD to confirm stability.

Feature Importance. Beyond predictive accuracy, we quantify the direction and magnitude of each feature's association with task success.

- **Component ablation.** We trained XGBoost models using each feature group alone, all pairwise combinations, and the full three-way combination. We report test AUC and R^2 for each subset to estimate how much predictive signal is contributed by patch complexity, repository structure, and issue text.
- **SHAP feature importance.** We compute SHAP values for the best-performing XGBoost model and rank features by their mean absolute contribution to model predictions. SHAP additive explanations provide a theoretically grounded decomposition of model predictions [20], making them suitable for interpreting the relative importance of features in a non-linear ensemble model.

4 Results

4.1 Task-Level Feature Agent Success Prediction Accuracy (RQ1)

Static task features derived from the gold patch, repository structure, and agent prompt yield strong predictive performance across all three outcome targets. Table 6 reports held-out test performance for all models alongside a majority-class baseline that assigns the positive label to every task. The baseline achieves AUC = 0.500 and MCC = 0.000, confirming that it carries no discriminative information; its comparatively high F1 score of 0.806 is due to the 67.5% positive rate in the `any_success` target and should not be interpreted as predictive signal.

Among the three classifiers, XGBoost and Random Forest reach near-identical performance on `any_success` (XGBoost: AUC = 0.863, MCC = 0.549, Brier = 0.129; Random Forest: AUC = 0.863, MCC = 0.538, Brier = 0.128).

Table 6 Held-out test performance. Majority denotes the always-positive baseline. Bold denotes best non-baseline result per metric per target.

Target	Model	AUC	PR-AUC	Brier	F1	MCC
`any_success`	Majority	0.500	0.675	0.219	0.806	0.000
	Logistic	0.750	0.849	0.183	0.741	0.341
	Random Forest	**0.863**	0.914	**0.128**	0.832	0.538
	XGBoost	**0.863**	**0.916**	0.129	**0.848**	**0.549**
`maj_success`	Majority	0.500	0.617	0.383	0.763	0.000
	Logistic	0.738	0.809	0.198	0.722	0.330
	Random Forest	**0.848**	**0.885**	**0.147**	0.797	0.508
	XGBoost	0.845	0.884	0.148	**0.811**	**0.517**

`pass_rate` regression			
Model	**RMSE**	**MAE**	**R^2**
Ridge	0.415	0.374	0.169
Random Forest	**0.350**	**0.276**	**0.409**
XGBoost	**0.350**	**0.276**	0.408

The substantially lower performance of Logistic Regression (AUC = 0.750, MCC = 0.341) indicates that the relationship between task features and agent success is *non-linear* and cannot be adequately captured by a linear decision boundary.

Predicting the continuous `pass_rate` outcome is a harder problem. Ridge regression accounts for only 16.9% of variance ($R^2 = 0.169$), whereas Random Forest and XGBoost reach $R^2 = 0.409$ and $R^2 = 0.408$ respectively, showing a 2.4× improvement over the linear baseline. The two ensemble methods are essentially tied on this target. A portion of the remaining variance is likely irreducible: 17.7% of tasks show mixed outcomes across repeated runs, and no static task-level feature can account for the stochastic component of agent behavior within a fixed task.

Ten-fold cross-validation on the XGBoost training set confirms that held-out test performance is stable. For `any_success`, mean CV AUC is 0.851 ± 0.009 and mean CV MCC is 0.558 ± 0.017, both in close agreement with held-out test values. For `maj_success`, mean CV AUC is 0.838 ± 0.006 and mean CV MCC is 0.530 ± 0.008. For the `pass_rate` regression, mean CV R^2 is 0.386 ± 0.008, consistent with the held-out result of 0.408.

Calibration reliability diagrams for all three classifiers are shown in Figure 1. XGBoost is

Table 7 Component ablation using XGBoost on the held-out test set. AUC and Brier score are reported for the two binary outcomes; R^2 and MAE for the continuous `pass_rate` outcome. Bold denotes the best value in each column.

		any_success		maj_success		pass_rate	
Feature set	**N**	**AUC**	**Brier**	**AUC**	**Brier**	**R^2**	**MAE**
Patch only	13	0.846	0.135	0.830	0.154	0.375	0.285
Repo only	14	0.839	0.137	0.820	0.158	0.350	0.291
Prompt only	27	0.599	0.214	0.593	0.231	0.025	0.417
Patch + Repo	27	0.861	0.129	0.841	0.149	0.401	0.275
Patch + Prompt	40	0.850	0.135	0.832	0.154	0.380	0.290
Repo + Prompt	41	0.839	0.138	0.823	0.158	0.361	0.295
All features	54	**0.863**	**0.129**	**0.846**	**0.148**	**0.406**	**0.277**

the best-calibrated model across both outcomes, with small deviations from the diagonal. Random Forest is well-calibrated in the upper probability range but exhibits overconfidence in the low-to-mid range, where predicted probabilities consistently exceed observed success rates. Logistic Regression shows the largest deviations across both outcomes, consistent with its compressed prediction range and weaker discriminative performance. The well-calibrated XGBoost probability estimates make it the preferred model for downstream applications such as task stratification and difficulty-weighted evaluation.

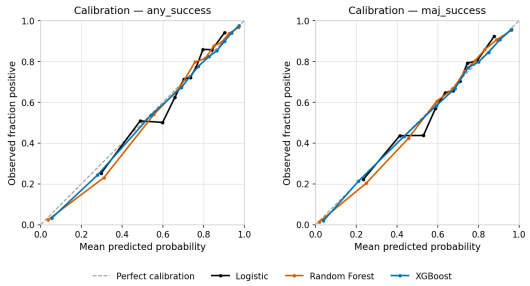


Figure 1 Calibration reliability diagrams

Finding 1. Task-level features derived from the patch, repository, and agent prompt predict agent success with AUC = 0.863 and MCC = 0.549 (XGBoost), explaining 41% of pass-rate variance ($R^2 = 0.408$), with well-calibrated probability estimates across the full prediction range.

4.2 Feature Groups Driving Predictive Performance (RQ2)

To determine the independent contribution of each feature group, we trained XGBoost models using each group alone and in all pairwise and three-way combinations, evaluating on the held-out test set across all three outcome definitions. Table 7 presents the results.

Patch features alone achieve AUC = 0.846 on `any_success` and $R^2 = 0.375$ on `pass_rate`. Repository features alone perform comparably, with AUC = 0.839 and $R^2 = 0.350$. Combining these two groups yields AUC = 0.861 and $R^2 = 0.401$, within 0.002 AUC and 0.006 R^2 of the full 54-feature model (AUC = 0.863, $R^2 = 0.406$).

Considered in isolation, the prompt group performs near chance on both binary outcomes (AUC = 0.599 for `any_success`; AUC = 0.593 for `maj_success`) and explains almost none of the continuous variance ($R^2 = 0.025$). Adding prompt features to patch alone yields marginal gains (ΔAUC ≤ 0.004, $\Delta R^2 = 0.005$ on `pass_rate`), and adding them to repository alone is similarly modest (ΔAUC ≤ 0.004, $\Delta R^2 = 0.012$ on `pass_rate`). Critically, once both patch and repository features are included, adding all prompt features produces negligible further gain: ΔAUC ≤ 0.002 and $\Delta R^2 < 0.01$ across all three outcome definitions. This pattern is consistent across both discrimination (AUC) and calibration (Brier score) measures.

These results indicate that the difficulty of a task, as measured by agent success rates, is determined almost entirely by the structural properties of the required solution. Specifically, how complex and scattered the patch is, and how large and complex the repository environment is are the strongest indicators. The linguistic and syntactic properties of the agent prompt, as operationalized by our feature set, carry negligible independent signal beyond these structural factors. Whether prompt properties matter conditionally, for specific tasks or in specific circumstances, is a question this ablation study alone cannot answer; we examine individual feature contributions in RQ3.

Finding 2. Patch complexity and repository structure together account for virtually all of the predictable variance in agent success across all outcome definitions, achieving AUC = 0.861 and $R^2 = 0.401$ with 27 features – within 0.002 AUC of the full model.

4.3 Individual Feature Association with Agent Success (RQ3)

We use SHAP values computed via `TreeExplainer` on the held-out test set to identify which features drive predictions and in what direction [20]. We report mean absolute SHAP value as the importance measure, SHAP dependence plots for directional effects, and SHAP interaction values for joint effects between feature pairs. All analyses use the tuned XGBoost model; we compare feature rankings between `any_success` and `pass_rate` to assess whether findings generalize across outcome definitions.

4.3.1 Feature Importance and Directional Effects

Figure 3 shows the top 20 features by mean absolute SHAP value for each outcome. Figure 4 shows the corresponding SHAP summary plot for `any_success`, which reveals the direction of each effect. The same three features occupy the top positions across all outcomes: `patch_lines_deleted`, `patch_num_hunks`, and `patch_hunk_gap_mean` (ranks 1–3, with only a 1/2 swap between targets, visible as near-flat connecting lines in Figure 3). For `any_success`, `patch_lines_deleted` has a mean absolute SHAP of 0.406– 2.65× larger than the fourth-ranked feature
(`repo_top_level_dir_count`, 0.153)—and the top three features together account for 29% of total mean absolute SHAP across all 54 features.

In the summary, all three plots show a clear monotone negative pattern: high feature values push predictions toward failure. `patch_hunk_gap_mean` additionally exhibits a small number of extreme outliers (SHAP ≈ -2.5), indicating that tasks with unusually large inter-hunk distances are especially penalized. SHAP values decay steeply below the leading group: the 20th-ranked feature (`prompt_len_bpe`) has a mean absolute SHAP 9× smaller than the top feature, and below rank 10 distributions are tightly clustered near zero with no directional gradient. Ranks 4–14 are occupied entirely by repository structural and patch features, all showing negative directional effects in Figure 4. These contribute independently

of the top patch trio, suggesting that the navigational complexity of the repository the agent must traverse matters beyond the properties of the edit itself.

The waterfall plots (Figures 5a–5b) illustrate how these effects compose at the task level. All values are in log-odds space; the population baseline is $E[f(x)] = 0.73$. The easiest task ($\hat{p} = 0.997$, `pygments/pygments`) has a compact two-site edit: every structural contributor pushes the prediction upward from baseline. The hardest task ($\hat{p} = 0.013$, `getnikola/nikola`) has moderate patch volume but high `repo_top_level_dir_count` as the second-largest negative contributor—illustrating that the combination of moderate fragmentation and a wide repository suffices to drive the prediction to near-zero success. The average task ($\hat{p} \approx 0.68$, `kurtmckee/feedparser`, Figure 6a) sits just above baseline with mixed structural signals, a pattern examined further below.

Feature interactions are dominated by patch × repo pairs; Figure 2 shows the top four by mean absolute interaction value averaged across outcomes. The two leading pairs (`patch_hunk_gap_mean` × `lines_deleted` and `num_hunks` × `top_level_dir_count`) swap ranks between `any_success` and the other two targets but remain the top interactions throughout.

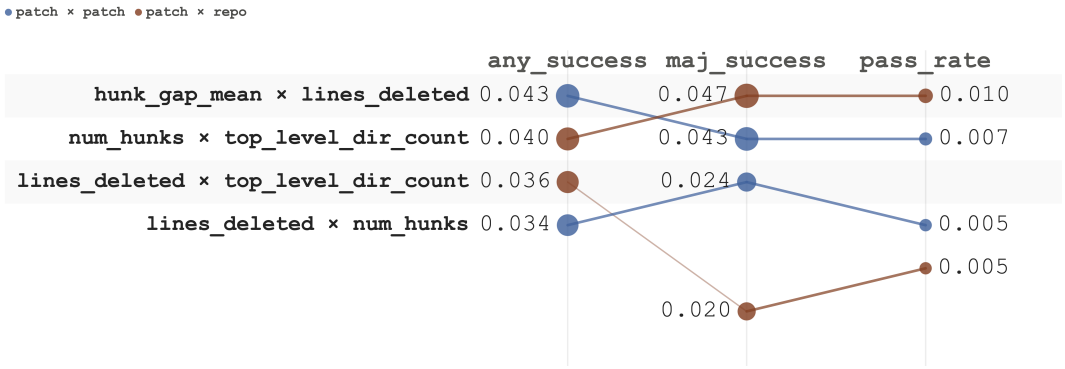


Figure 2 Rank–magnitude bump chart for the top four SHAP interaction pairs by mean absolute interaction value averaged across outcome targets.

4.3.2 Prompt Features in the Mid-Band

At the population level, prompt features first appear at rank 16 and carry an order of magnitude less SHAP mass than the leading patch features (Figure 3), consistent with the RQ2 ablation. These aggregate results, however, can suppress changes in the relative importance of feature groups across the prediction range.

We stratify tasks by the predicted probability $\hat{p}$ for `any_success`: *easy* tasks fall in the top decile of $\hat{p}$ (n=893), *hard* tasks in the bottom decile (n=886), and *mid-band* tasks have predictions within 0.05 of the mean prediction ($\bar{\hat{p}}$=0.685; n=575). This group has a median summed SHAP contribution is +0.09 log-odds, compared with +2.84 for easy tasks and −3.76 for hard tasks.

Prompt features become substantially more prominent in the mid-band. At least one prompt feature ranks among the five largest contributors for **70.3**% of near-baseline tasks (404/575), **26.8**% of easy tasks (239/893), and **6.8**% of hard tasks (60/886). The same ordering holds for `pass_rate` and `maj_success`.

We observe that the gap between prompt and structural contributions begin to narrow here. We define $[R(x) = \frac{\max_{j \in \text{prompt}} |\phi_j(x)|}{\max_{j \in \text{structural}} |\phi_j(x)|}]$. Median R rises to 0.404 near mid-band,

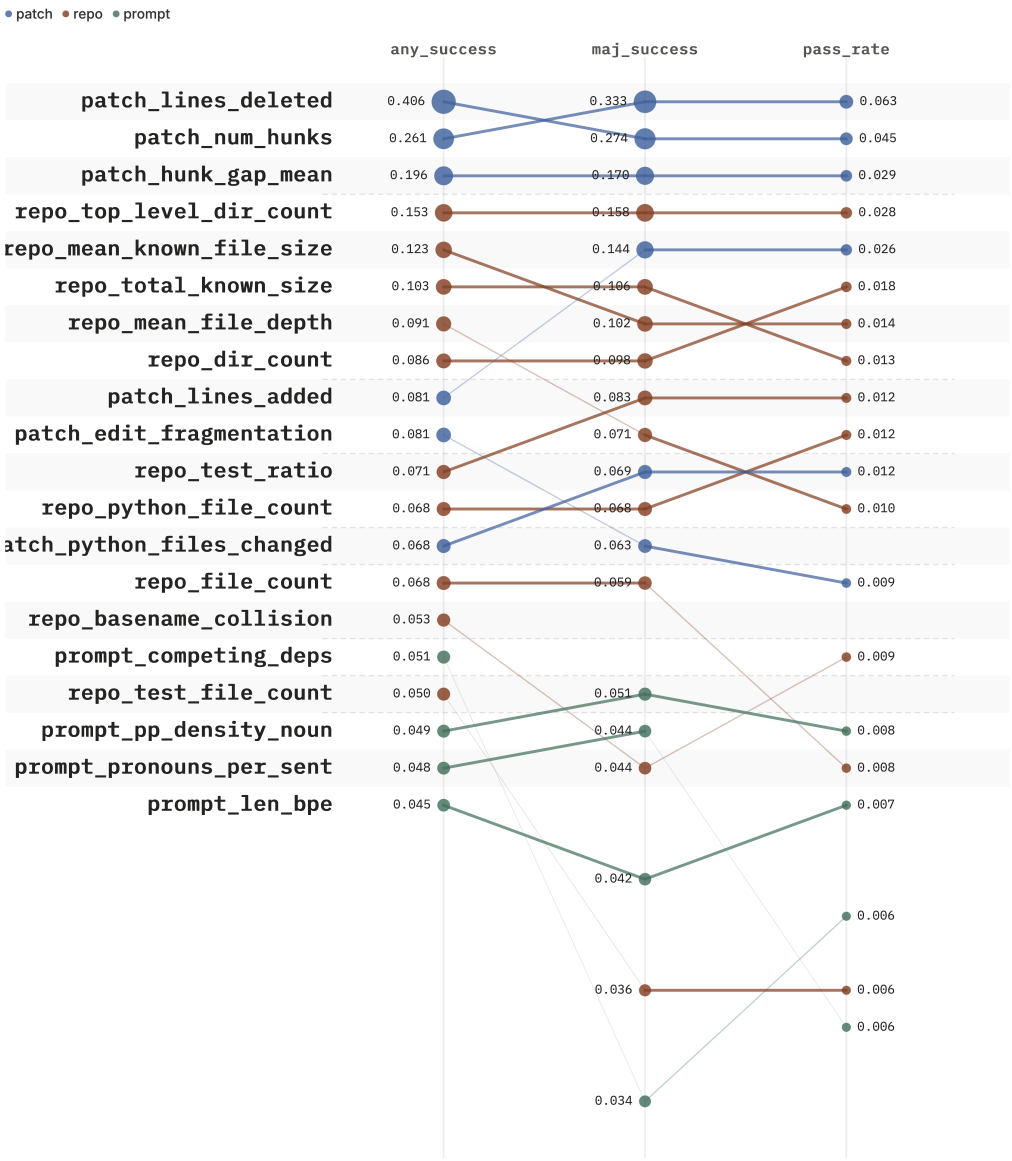


Figure 3 Rank–magnitude bump chart for the top 20 features by mean absolute SHAP value across three outcome targets (XGBoost, held-out test set). Each dot is one feature–outcome pair; dot area is proportional to mean absolute SHAP. Connecting lines trace rank shifts across outcomes; steep slopes indicate instability. Color encodes feature family: patch (blue), repo (red-brown), prompt (green). The top six ranks are stable across all three outcomes (near-flat lines); prompt features first appear at rank 16 with visibly smaller dots and show greater rank instability below rank 16.

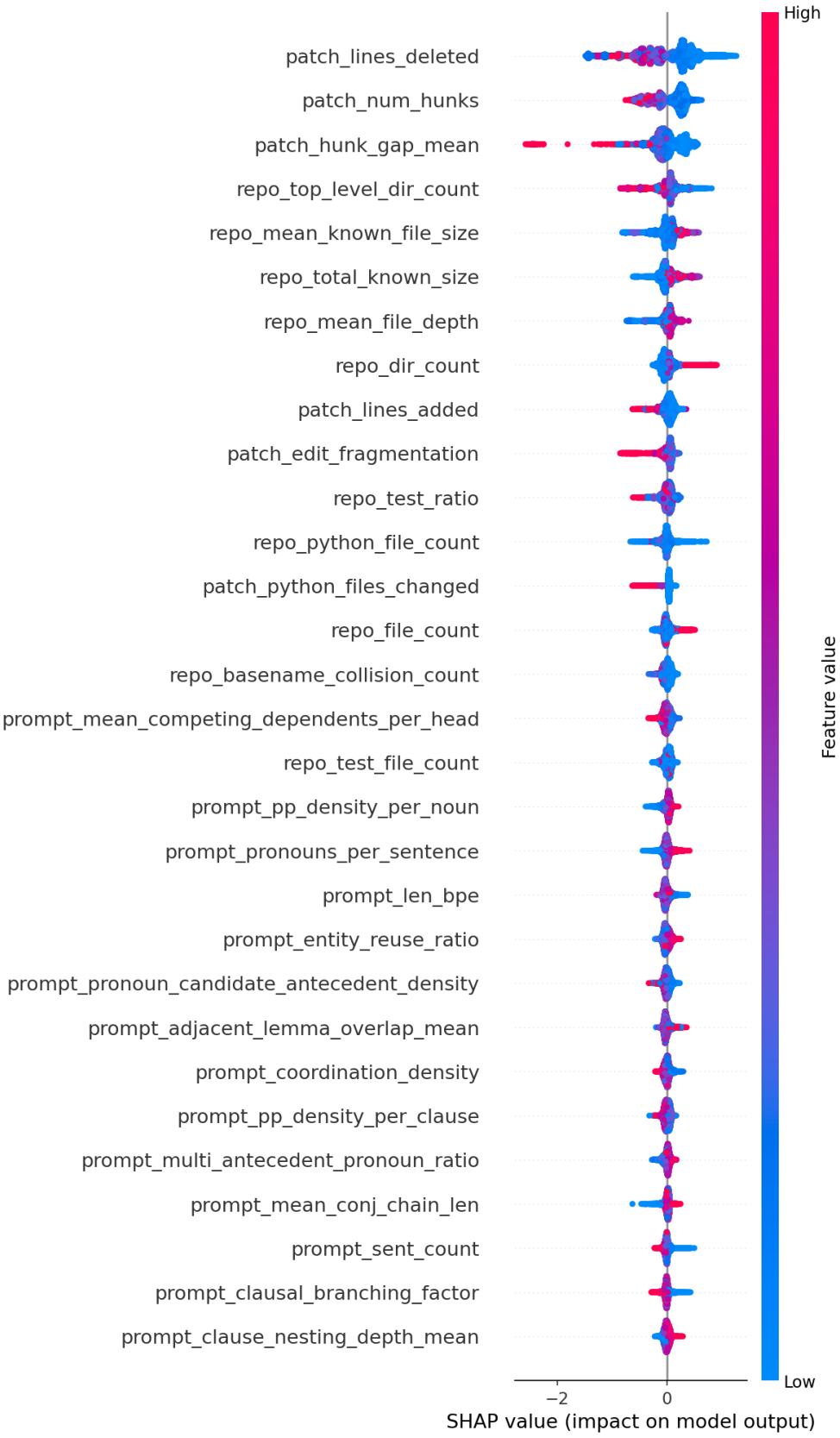
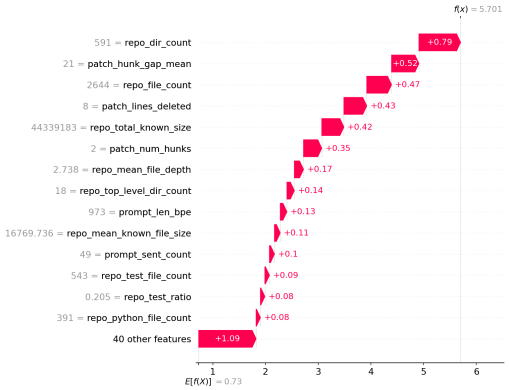
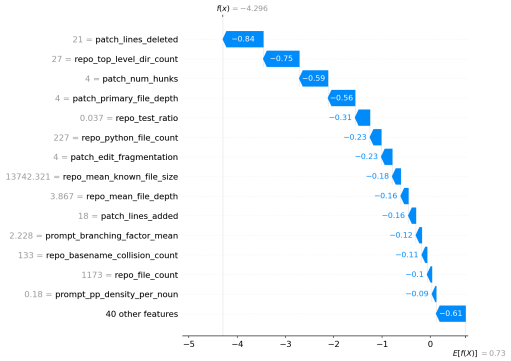


Figure 4 SHAP summary plot for `any_success`.

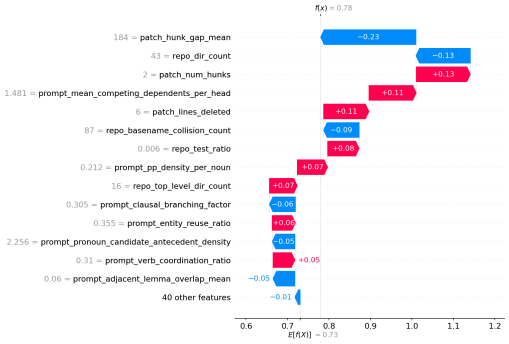


(a) Easiest task (`pygments/pygments`); $\hat{p} = 0.997$, $E[f(x)] = 0.73$, observed: success. Every structural contributor pushes the prediction upward from baseline.

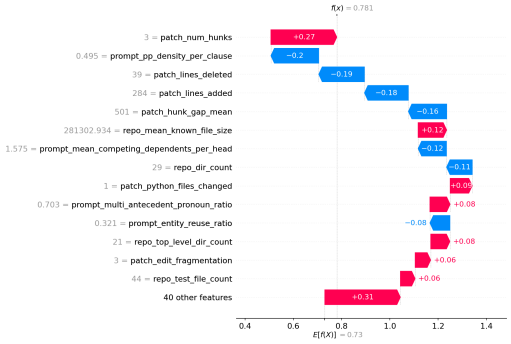


(b) Hardest task (`getnikola/nikola`); $\hat{p} = 0.013$, $E[f(x)] = 0.73$, observed: failure. Despite only moderate patch volume, high `repo_top_level_dir_count` is the second-largest negative contributor, illustrating the patch–repository interaction.

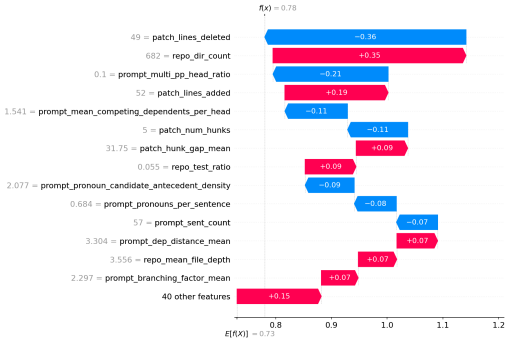
Figure 5 SHAP waterfalls for extreme-difficulty tasks (XGBoost, `any_success` target). All values in log-odds space; $E[f(x)] = 0.73$ for both.



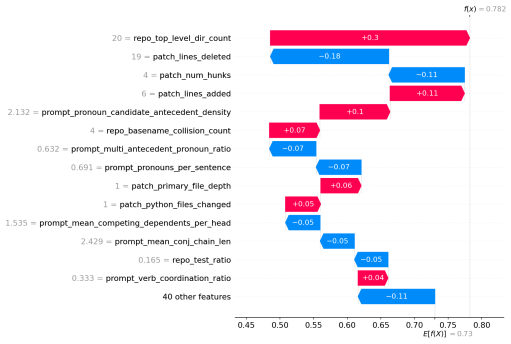
(a) `kurtmckee/feedparser`; $\hat{p} = 0.684$, observed: success. Mixed structural signals sit just above baseline.



(b) `sqlglot`; $\hat{p} = 0.685$, observed: failure. `prompt_pp_density_per_clause` (-0.20) is the second-largest contributor.



(c) `faker`; $\hat{p} = 0.684$, observed: success. `prompt_multi_pp_head_ratio` (-0.21) is the third-largest contributor.



(d) `parso`; $\hat{p} = 0.685$, observed: success. `prompt_pronoun_candidate_antecedent_density` ($+0.10$) ranks fifth among contributors.

Figure 6 SHAP waterfalls for four representative average-difficulty tasks ($\hat{p} \approx 0.68$, $E[f(x)] = 0.73$, XGBoost, `any_success`). In three of four cases a prompt feature ranks among the top contributors, with magnitudes comparable to the structural features for those tasks.

compared with 0.205 (1.97×) for easy tasks and 0.108 (3.75×) for hard tasks.

No single prompt feature accounts for the pattern. The three most frequent contributors are `prompt_pronouns_per_sentence` (referential ambiguity, 11.1% of near-baseline tasks; 64/575), `prompt_mean_conj_chain_len` (coordination ambiguity, 10.1%; 58/575), and `prompt_mean_competing_dependents_per_head` (attachment ambiguity, 9.4%; 54/575), all established markers of processing difficulty in computational linguistics [8].

Figure 6 illustrates this at the task level with three representative examples:

`prompt_pp_density_per_clause`, another attachment-ambiguity feature, is the second-largest contributor in a `sqlglot` task (−0.20); `prompt_multi_pp_head_ratio` ranks third in a `faker` task (−0.21); `prompt_pronoun_candidate_antecedent_density` ranks fifth in a `parso` task (+0.10).

Finding 3. The composition of task-difficulty signals changes across the prediction range. Patch fragmentation and repository scale dominate at the extremes, reliably separating easy from hard tasks across all three outcome definitions. Near the mid-band, prompt features enter the top-5 SHAP contributors for 70.3% of tasks, versus 26.8% (easy) and 6.8% (hard). No single prompt feature displaces patch or repository features in aggregate importance.

5 Discussion

Our results make the structural composition of a benchmark directly observable. This helps reason about whether two benchmarks are measuring the same thing or whether an improvement in solve rate reflects a genuine capability gain in targeted dimensions.

One practical application for our findings is in benchmark auditing [33]. Computing difficulty scores for each task in an existing benchmark reveals whether the difficulty distribution is structurally balanced or concentrated in a narrow region of the task space. A benchmark heavily populated with compact, single-hunk patches in shallow repositories would systematically under-represent tasks that require broader navigation or multi-site coordination, which would be invisible in aggregate solve rates. This can also provide opportunities for more intentional benchmark design, where tasks structure and difficulty are explicit considerations.

A second application area is in difficulty-stratified evaluation. An agent that improves primarily on tasks that are already easy for the reference agent represents a qualitatively different kind of progress from one that improves on structurally difficult tasks. Reporting performance within difficulty quartiles would make this distinction visible. The model produces well-calibrated probability estimates across the full range, so predicted probabilities can be used directly as continuous difficulty scores without post-hoc recalibration.

The features required for these applications are entirely deterministic. This is a meaningful practical advantage over IRT-based difficulty estimation approaches such as that of Ge et al. [6], which require LLM embeddings and rubric scoring per task.

5.1 Toward Agent-Specific Difficulty Profiles

The difficulty patterns we found are relative to a specific reference agent, and that framing opens a natural extension of the work. Applying our framework separately to different agents would make per-agent difficulty profiles observable. Prior work in software engineering prediction shows why this matters. Menzies et al. found that global rules learned across heterogeneous populations were consistently outperformed by rules learned from neighboring

clusters [23], and Jiang et al. showed that developer-specific defect models substantially outperformed pooled ones because developers differ too much in experience and coding style to be well-described by a single model [13]. The same principle likely applies here. What is hard for Qwen3 with this scaffold (due to, for example, its approach to navigating wide directory structures) may be easier for an agent with a different localization strategy. As trajectory data accumulates across frontier agents, this form of per-agent difficulty diagnosis becomes a practical step toward understanding and improving the specific structural bottlenecks each system faces.

5.2 Toward Calibrated Expectations in Development Tools

Though not the focus of our study, our efforts provide valuable insights towards trustworthy agentic developer solutions. Our findings connect to how developers decide how much to trust an agent's output on a given task. According to the PICSE [15] framework, trust in AI-assisted software tools is shaped in part by the expectations a developer brings to an interaction and by how well the system's reliability conditions are made explicit. A single aggregate benchmark score does not support this well as it hides the conditions under which the agent is more or less likely to succeed.

Difficulty scores give developers a more grounded basis for calibrating reliance. A developer who has a sense of how structurally complex a task is can look at an agent's performance within the corresponding difficulty band on a benchmark, rather than its overall solve rate, to form a more accurate expectation of whether the agent will succeed. This shifts the question from "how good is this agent" to "how good is this agent on tasks like this one," which is the question that matters most for deciding when and how to use an AI-assisted tool.

6 Threats to Validity

6.1 Internal

Agent pretraining on benchmark repositories. Qwen3-Coder-480B was pretrained on public code, and CoderForge-Preview draws from public GitHub repositories. Tasks from repositories present in pretraining may be solved at elevated rates unrelated to structural features.

Outcome reliability and agent stochasticity. Pass rate is estimated from trajectories generated at nonzero temperature, so outcomes conflate genuine task difficulty with sampling variance intrinsic to the agent's decoding process. No static feature can predict this stochastic component, which partially explains the irreducible variance in the regression target.

6.2 External

Single model group. All trajectories in CoderForge-Preview were generated by Qwen3-Coder-480B operating within an OpenHands scaffold. The features that predict difficulty for this agent may not generalize to agents with different architectures, context window sizes, or scaffolding strategies. In particular, our finding that repository scale and directory breadth are important predictors implicitly reflects this agent's navigation strategy; an agent with a different file-localization approach might be less sensitive to these features. Replication with trajectories from other frontier agents would strengthen generalizability claims.

Python-centric sample. The underlying repositories are predominantly Python projects. Features related to Python file counts and test infrastructure reflect conventions of the Python ecosystem (e.g., `pytest`-based test organization). The absolute importance values for these features may not transfer directly to repositories in other languages, though the underlying constructs (test density, repository scale, edit fragmentation) are language-agnostic.

6.3 Construct

Gold patch as ground truth. Patch features are computed from the gold (oracle) patch provided with each task. In practice, an agent generates a candidate patch that may differ structurally from the gold. A task could have a simple gold patch but require complex agent reasoning, or vice versa. We use the gold patch as the best available proxy for "what a correct solution looks like," consistent with prior just-in-time defect prediction work. The strong predictive performance suggests this proxy is informative, but we acknowledge that gold-patch features do not capture all sources of task complexity.

Prompt features. Our linguistic prompt features operationalize structural conditions for ambiguity rather than ambiguity itself. The RE literature distinguishes *nocuous* ambiguity, where different readers reach genuinely different interpretations that lead to different implementations, from *innocuous* ambiguity, where the structure is syntactically ambiguous but context resolves it to a single intended meaning [36]. Our features cannot make this distinction. A syntactically complex issue description may still be pragmatically unambiguous as the domain, the surrounding code context, or simple common sense may resolve the attachment uniquely. Conversely, a structurally simple sentence can be genuinely nocuous if the two readings lead to different code changes. The consequence is that our prompt features measure a necessary but not sufficient condition for the kind of specification ambiguity that would actually mislead an agent. This upper-bounds their predictive power and is consistent with the low global SHAP signal we observe for prompt features. Whether finer-grained ambiguity detection, for instance, using LLM-as-judge to classify issues as nocuous versus innocuous—would recover additional predictive signal is an open question for future work.

7 Conclusion

The difficulty of a software issue resolution task is encoded in its structure. We showed that static, deterministic features derived from the gold patch, repository, and issue prompt predict agent success with AUC = 0.863 without any model inference. This enables pre-hoc difficulty estimation and opens the door to structurally controlled benchmark construction, difficulty-stratified evaluation, and more grounded developer reliance on agent tools. Future work will extend the framework across multiple agents to expose per-agent difficulty profiles and, in aggregate, a global difficulty landscape that characterizes the structural frontier of agentic software engineering. More broadly, the measurement framework generalizes naturally to other agentic SE tasks such as feature implementation, test generation, where analogous structural properties of the target artifact and the natural-language specification can be represented as static difficulty signals.

8 Data Availability

The replication package is available *here*.

References

1 Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. Swe-bench+: Enhanced coding benchmark for llms. *arXiv preprint arXiv:2410.06992*, 2024.

2 Alpay Ariyak, Junda Zhang, Junxiong Wang, Shang Zhu, Federico Bianchi, Sanjana Srivastava, Ashwinee Panda, Siddhant Bharti, Chenfeng Xu, John Heo, Xiaoxia Shirley Wu, James Zou, Percy Liang, Leon Song, Ce Zhang, Ben Athiwaratkun, Zhongzhu Zhou, and Qingyang Wu. Coderforge-preview: Sota open dataset for training efficient agents, February 2026. Project core leads: Alpay Ariyak; Zhongzhu Zhou; Qingyang Wu. URL: `https://www.together.ai/blog/coderforge-preview`.

3 Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026.

4 David A Belsley, Edwin Kuh, and Roy E Welsch. *Regression diagnostics: Identifying influential data and sources of collinearity*. John Wiley & Sons, 2005.

5 Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

6 Chris Ge, Daria Kryvosheieva, Daniel Fried, Uzay Girit, and Kaivalya Hariharan. Agent psychometrics: Task-level performance prediction in agentic coding benchmarks. *arXiv preprint arXiv:2604.00594*, 2026.

7 Baljinder Ghotra, Shane McIntosh, and Ahmed E Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 789–800. IEEE, 2015.

8 Edward Gibson et al. The dependency locality theory: A distance-based theory of linguistic complexity. *Image, language, brain*, 2000:95–126, 2000.

9 M A K Halliday and Ruqaiya Hasan. *Cohesion in English*. Routledge, London, England, January 2014.

10 A Handbook. From contract drafting to software specification: Linguistic sources of ambiguity, 2003.

11 Valentin Hofmann, David Heineman, Ian Magnusson, Kyle Lo, Jesse Dodge, Maarten Sap, Pang Wei Koh, Chun Wang, Hannaneh Hajishirzi, and Noah A Smith. Fluid language model benchmarking. *arXiv preprint arXiv:2509.11106*, 2025.

12 Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. *arXiv preprint arXiv:2504.07164*, 2025.

13 Tian Jiang, Lin Tan, and Sunghun Kim. Personalized defect prediction. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 279–289. Ieee, 2013.

14 Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pages 54107–54157, 2024.

15 Brittany Johnson, Christian Bird, Denae Ford, Ebtesam Al Haque, Nicole Forsgren, and Thomas Zimmermann. Facilitating trust in ai-assisted software tools. *ACM Trans. Softw. Eng. Methodol.*, October 2025. Just Accepted. `doi:10.1145/3772370`.

16 Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E Hassan. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering*, 21(5):2072–2106, 2016.

17 John P Lalor, Hao Wu, and Hong Yu. Building an evaluation scale using item response theory. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 648–657, 2016.

18 Maya Larbi, Amal Akli, Mike Papadakis, Rihab Bouyousfi, Maxime Cordy, Federica Sarro, and Yves Le Traon. When prompts go wrong: Evaluating code model robustness to ambiguous, contradictory, and incomplete task descriptions. *arXiv preprint arXiv:2507.20439*, 2025.

19 Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE transactions on software engineering*, 34(4):485–496, 2008.

20 Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *Advances in neural information processing systems*, 30, 2017.

21 Donald W Marquardt. Generalized inverses, ridge regression, biased linear estimation, and nonlinear estimation. *Technometrics*, 12(3):591–612, 1970.

22 Danielle S McNamara, Eileen Kintsch, Nancy Butler Songer, and Walter Kintsch. Are good texts always better? interactions of text coherence, background knowledge, and levels of understanding in learning from text. *Cognition and instruction*, 14(1):1–43, 1996.

23 Tim Menzies, Andrew Butcher, David Cok, Andrian Marcus, Lucas Layman, Forrest Shull, Burak Turhan, and Thomas Zimmermann. Local versus global lessons for defect prediction and effort estimation. *IEEE Transactions on software engineering*, 39(6):822–834, 2012.

24 Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

25 Audris Mockus and David M Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

26 John C Munson and Sebastian G Elbaum. Code churn: A measure for estimating the impact of code change. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 24–31. IEEE, 1998.

27 Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 284–292, 2005.

28 OpenAI. Introducing SWE-bench Verified. `https://openai.com/index/introducing-swe-bench-verified/`, 2024.

29 Felipe Maia Polo, Lucas Weber, Leshem Choshen, Yuekai Sun, Gongjun Xu, and Mikhail Yurochkin. tinybenchmarks: evaluating llms with fewer examples. *arXiv preprint arXiv:2402.14992*, 2024.

30 Adwait Ratnaparkhi, Jeff Reynar, and Salim Roukos. A maximum entropy model for prepositional phrase attachment. In *Human Language Technology: Proceedings of a Workshop held at Plainsboro, New Jersey, March 8-11, 1994*, 1994.

31 Manish Shetty, Naman Jain, Jinjian Liu, Vijay Kethanaboyina, Koushik Sen, and Ion Stoica. Gso: Challenging software optimization tasks for evaluating swe-agents. *Advances in Neural Information Processing Systems*, 38, 2026.

32 Sang Truong, Yuheng Tu, Percy Liang, Bo Li, and Sanmi Koyejo. Reliable and efficient amortized model-based evaluation. *arXiv preprint arXiv:2503.13335*, 2025.

33 Xinming Tu, Tianze Wang, Kexin Huang, Yuanhao Qu, Sara Mostafavi, et al. Benchguard: Who guards the benchmarks? automated auditing of llm agent benchmarks. *arXiv preprint arXiv:2604.24955*, 2026.

34 Sanidhya Vijayvargiya, Xuhui Zhou, Akhila Yerukola, Maarten Sap, and Graham Neubig. Interactive agents to overcome ambiguity in software engineering. *arXiv preprint arXiv:2502.13069*, 2025.

35 An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

36 Hui Yang, Anne De Roeck, Alistair Willis, and Bashar Nuseibeh. A methodology for automatic identification of nocuous ambiguity. In *Proceedings of the 23rd International Conference on Computational Linguistics (Coling 2010)*, pages 1218–1226, 2010.

37 John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026.

38 Boxi Yu, Yuxuan Zhu, Pinjia He, and Daniel Kang. Utboost: Rigorous evaluation of coding agents on swe-bench. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3762–3774, 2025.

39 Yunhua Zhao, Kostadin Damevski, and Hui Chen. A systematic survey of just-in-time software defect prediction. *ACM Computing Surveys*, 55(10):1–35, 2023.