swespoke: DECLARE WHAT YOU WANT, GET A BESPOKE SOFTWARE ENGINEERING BENCHMARK

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding benchmarks must rank agents, expose where they fail, and tell developers which agent to trust with their tasks. Early benchmarks were fixed task sets that humans defined, and current benchmarks mine tasks from an existing repository. Mining is slow and expensive, and at the end, users (software-engineering teams) get every task the repository yields. This process never asks what work users care about or which properties they want in tasks. Users, not benchmark authors, must be able to *declare task properties*. These declarations let users test the capability they care about, raise difficulty as agents improve, build an RL curriculum from easy to hard, and pay only for tasks that have the declared properties. These needs differ across users and change over time.

swespoke lets users declare the properties they want in tasks, and a query optimizer turns the declaration into an optimized plan that builds the benchmark to reduce execution time and cost. We call the process *task tailoring*. Unlike mining, tailoring checks the declaration before it builds a task, so selection and creation run as one step and users pay for no task they throw away. Each task comes with a runnable environment, a build, and tests, and swespoke verifies the task by running its tests. We also build NDI, a navigation difficulty index, as an example of a user-defined property. NDI estimates how hard it is to find the code a task must change. Users compose benchmarks at a chosen NDI level. Compared with the baselines, swespoke speeds up task creation by 7.27 to 12.48×, reduces LLM expense by 2.39 to 4.85×, and accelerates measurement collection by 14.43 to 19.94×. The average coding-agent pass rate falls from 68% overall to 46% on the hardest 10% of tasks by NDI.

1 THE CASE FOR USER-DECLARED BENCHMARKS

Coding benchmarks lead the development of coding agents. Agents now help developers explore codebases, plan changes, write code, and test software (Maddila et al., 2026). Benchmarks such as SWE-bench (Jimenez et al., 2024) rank agents, expose where they fail, and track progress from one release to the next. They also tell developers which agent to trust with their work. What a benchmark measures is what agents are judged on, and in time, what they get better at.

Each user needs a benchmark of their own, for their own purpose. A fixed task set does not reflect the work of a particular team, so teams need to make their own benchmark. For example, Cursor and Rakuten reportedly built CursorBench (Chan et al., 2026) and Rakuten-SWE-Bench (Anthropic, 2026) to assess agents based on their own engineering tasks. In other cases, a user has to test a particular *property* or capability. For instance, Amazon built SWE-InfraBench (Tarasova et al., 2026) to test how well agents make incremental edits to cloud infrastructure code. Google built GITS-Eval (Rondon et al., 2025) to test how well agents fix bugs described in text alone. Researchers need the same control to diagnose failures and to test design changes. In each case, no existing benchmark met the need, so the users built their own.

Benchmarks must also evolve as agents evolve. Once agents solve most of a benchmark's tasks, the benchmark can no longer rank them (Deng et al., 2026). Training has the same need from the other side: reinforcement learning curricula move agents from easy tasks to hard ones as learning progresses (Parashar et al., 2026; Chen et al., 2025). Evaluation and training both need to know how hard each task is, and task statistics such as code churn and file depth give that answer without

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

running any agent (Al-Haque & Johnson, 2026). Difficulty is therefore another task property that users need to measure and control.

Existing benchmarks give users tasks, not the choice of which properties those tasks have. First-generation benchmarks were fixed task sets that humans defined, at the cost of substantial expert effort. Building SWE-InfraBench by hand, with LLM assistance and expert review, yielded five high-quality tasks per engineer-day. The next generation removes that effort by mining tasks from existing repositories: SWE-smith (Yang et al., 2025) and SWE-bench-Live (Zhang et al., 2025b) supply many more tasks, and SWE-rebench V2 (Badertdinov et al., 2026), RepoLaunch (Li et al., 2026), and SWE-Factory (Guo et al., 2025) automate repository setup and test execution, so users can build tasks from new repositories. Mining is still slow and expensive. SWE-Bench++ reports about 67 minutes of build and test time per candidate and a working environment for one candidate in five (Wang et al., 2025). Mining also leaves users with every task the repository yields, whatever its properties. SWE-rebench V2 ships 12 metadata fields for selecting subsets by type and estimated difficulty, and SWE-smith builds difficulty-stratified subsets for its own experiments. Even so, the properties on offer are the ones the benchmark authors chose, and picking out tasks with the properties a user needs still means writing ad hoc code. Existing systems automate task creation, not the choice of task properties.

Users, not benchmark authors, must declare task properties such as difficulty. A team's own work, a particular capability, a difficulty that moves: these needs differ from user to user and change over time. No fixed menu of properties can anticipate the next one. The properties are also particular. SWE-InfraBench needs two at once: code written with the AWS Cloud Development Kit, and changes that modify existing infrastructure rather than create it. GITS-Eval excluded bugs whose reports required screenshots, so its tasks test fixing bugs described in text. A user must *declare* how to measure such properties, which combinations to include, and what difficulty to target. The user must also raise that difficulty as agents improve, with the capability held fixed. When no collection holds such tasks, the user must create them. So the declaration guides creation as well as selection. It also decides what the user pays for: tasks with the declared properties, not every task the repository yields. Two things make such a declaration hard to support. No single interface fits every user, because properties differ so much between users. And creating every task first and selecting afterward is expensive. The declaration must therefore be expressive enough for every user and act before a task is built, not after.

swespoke lets users declare the properties they want in tasks and tailors a benchmark to the declaration. It is an agentic framework that creates verified tasks from repository change history and composes benchmarks from user-defined measurements, selection criteria, and difficulty targets. Users write a declaration with three composable interfaces for declarative filtering: a *measurement* computes a task property, a *condition* tests its value, and an *action* says what to do with a task when the condition holds. For selection, users combine conditions on programming languages, codebase components, and task statistics such as change size. For revision, conditions trigger actions that complete underspecified descriptions, remove solution-revealing content, or mitigate the effects of training-data contamination. We call the process *task tailoring*. Unlike mining, tailoring checks the declaration as it creates tasks, so selection and creation run as one step, at less cost and more efficiently. Each task comes with a runnable environment, a build, and tests, and swespoke verifies the task by running its tests. The user writes the declaration, and swespoke does the rest.

A query optimizer turns the declaration into an optimized plan that builds the benchmark. It translates the declarative filters into execution plans that preserve the declared criteria. It jointly optimizes filter order and batching to cut execution time and cost, since filter order affects both the remaining workload and batching efficiency. The optimizer matters because the library it plans over is large and growing: swespoke already ships over 100 built-in measurements. The user declares what to measure, and the optimizer decides in what order and in what batches.

NDI shows what a user-defined difficulty measure looks like. Tailoring difficulty measures to evaluation targets is a key application of swespoke's declarative filters. We develop a built-in navigation difficulty index (NDI) to demonstrate the use case. NDI measures how hard it is to find the code a task must change. This is particularly relevant to systems software, where locating the code to change can require tracing behavior across multiple subsystems. Users compose benchmarks at a chosen NDI level to vary navigation difficulty across benchmarks. Alongside NDI, swespoke provides other built-in difficulty measurements, including agent task completion time and structural

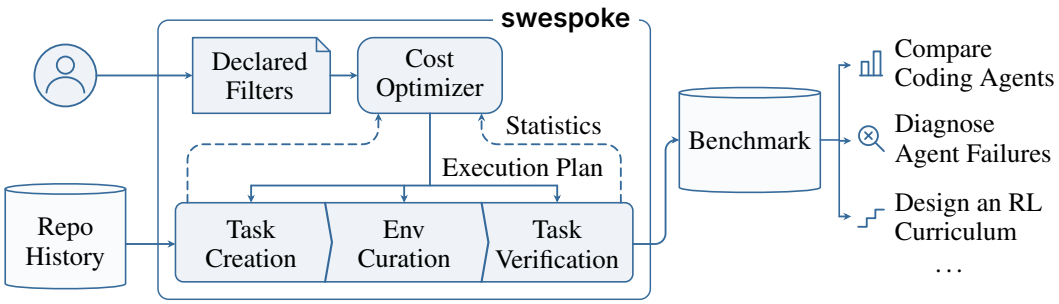

Figure 1: The swespoke workflow. Users declare filters. The optimizer compiles an execution plan that places filters throughout the pipeline. These filters prune tasks early and avoid costly downstream work.

properties such as patch fragmentation and repository size. Any of these can be declared, and any can be replaced by the user's own.

Experiments demonstrate efficient task tailoring and the selection of difficult tasks using NDI. Across three repositories spanning C++ and Rust, swespoke turns 223 of 31,677 candidate changes into verified tasks with the declared properties, without per-task expert effort. Task tailoring reduces creation time by 7.27 to 12.48× and LLM expense by 2.39 to 4.85×. The average coding-agent pass rate falls from 68% overall to 46% on the hardest 10% of tasks by NDI. The NDI result shows how swespoke turns a tailored difficulty measure into benchmarks that reveal differences in agent performance.

2 swespoke: Declarative Task Tailoring

swespoke turns a declaration of task properties and a repository history into a customized software engineering benchmark. Users declare what to measure and what to do with a task given its measured values. swespoke determines which task creation and selection jobs to run, in what order, and in what batches. It then compiles the declaration into an optimized pipeline and executes that pipeline. We first describe the workflow, then present the interfaces for declarative filters, and finally discuss the optimizer. We also show how users declare the navigation difficulty index (NDI) as a use case of a declared measurement.

2.1 swespoke Workflow

Given repository history and a filters declaration, swespoke creates candidate tasks, curates their environments, verifies them, and composes a benchmark from the tasks that the filters select. We illustrate this workflow in Figure 1. Task creation derives a statement, a base commit, a reference solution, and held-out tests from a historical change. Agentic environment curation installs dependencies and configures the build and test procedures. Task verification runs tests before and after applying the reference solution to identify the pass-to-pass (P2P) and fail-to-pass (F2P) test sets. This workflow follows established benchmark-construction practices used in RepoLaunch/SWE-bench-Live (Zhang et al., 2025b) and SWE-Factory (Guo et al., 2025).

swespoke's distinguishing design is to integrate user-declared filters throughout task creation, environment curation, and verification. Users declare which properties to measure and whether to select, report, or revise a task based on the results, without specifying optimizations like execution order or batching. The optimizer compiles these declarations into an execution plan that places filters according to their input dependencies and shares computation across measurements. By filtering candidates as the required evidence becomes available, swespoke avoids costly downstream work on unsuitable tasks while allowing revisions such as completing underspecified statements.

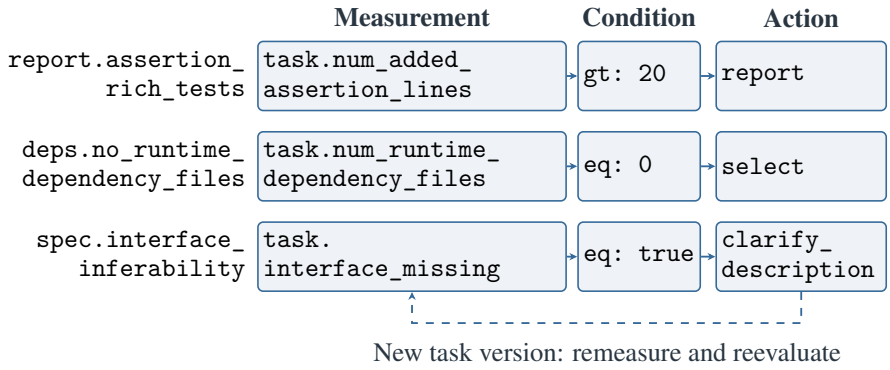

Figure 2: Three filters report assertion-rich tests, select tasks without runtime dependency edits, and clarify missing interfaces. Users declare these filters without specifying execution order or batching.

2.2 Declarative Filters

A declarative filter connects a *measurement*, a *condition*, and an *action*, as illustrated in Figure 2. The measurement computes a task property, such as the number of changed source lines. The condition compares the measured value with a threshold or a set of values, and the action specifies how the result affects the task. Users can combine conditions with conjunctions or disjunctions to express more complex requirements. Separating these components lets users change selection criteria without rewriting measurements or implementing an execution pipeline.

Each measurement declares its inputs and returns a typed value. swespoke represents each historical item, such as a PR, with a structured entity model whose fields users can directly declare as measurement inputs. Measurements can characterize a historical change, analyze a task's statement or code, summarize test results, or combine other measurements. swespoke provides over 100 built-in measurements, and users can extend the library by defining new calculations and their input dependencies. These dependencies tell the framework what evidence a measurement needs and when it can be computed during task creation.

An action specifies what to do with a task when a filter's condition is satisfied. swespoke supports three action types: *select*, *report*, and *transform*. Selection retains tasks that satisfy the condition, while reporting records the measured value and condition outcome without changing benchmark membership. Transform is an abstract action that users instantiate with a concrete revision procedure, such as adding missing interface requirements or removing solution-revealing content from a statement. An accepted transformation creates a new task version that is evaluated again, recomputing measurements whose inputs have changed. Together, these actions let users specify both which tasks belong in a benchmark and how tasks should be revised to meet its requirements.

2.3 Filter Execution Optimizer

The filter execution optimizer compiles declarative filters into a plan that reduces computation and LLM expense. Using declared input dependencies and collected execution statistics, it jointly chooses measurement batches and filter order. This ordering prunes task candidates not satisfying the conditions before costly downstream stages. The plan also preserves the declared conditions and the required order of transformations, since revising a task can change subsequent measurement results.

The swespoke optimizer automatically exploits three opportunities in user-declared filters: (1) evaluating lower-cost filters earlier, such as checking file paths before repository analysis and LLM-based assessments; (2) batching compatible measurements that share inputs, like allowing patch-based measurements to share disk reads and parsing, and prompt-based measurements to share tokenization and NLP parsing; and (3) prioritizing filters with higher pruning rates among those with similar costs, hence reducing downstream work.

Table 1: NDI factors grouped by their effects on size in Equation 1.

Effect	Factor	Description	Weight
Size	L	Net changed source lines: lines added or removed in source files between the base commit and the reference solution	–
Scale (E)	F	Source files with a nonzero net change	s_F
	G	Directories that contain those files	s_G
Multipliers (M)	U	Largest number of functions or classes changed in one source file	w_U
	P	Largest number of changed lines outside any function or class in one source file	w_P
	T	Test cases that the task's test change adds or modifies	w_T
	Q	LLM-assessed difficulty on a user-defined ordered scale	$\{u_k\}_{k=2}^{K}$
Search cost (S)	R	Share of changed source files that contain an identifier from the statement	c_R
	D	Fewest import-graph hops to a changed source file from the files the statement mentions	c_D

2.4 Customizing Task Difficulty with NDI

The navigation difficulty index (NDI) illustrates how users define a difficulty measurement and use it to select tasks. NDI combines measurements of the reference change and task statement to estimate the difficulty of locating and understanding the code a task requires changing. Users can then use NDI to select tasks at desired difficulty levels or organize them into groups for evaluation and training curricula.

NDI uses changed source lines as its size term and adjusts it through three components: (1) scale effects that change the exponent of size; (2) multipliers that adjust the scaled size; and (3) an additive and independent search cost for locating the relevant code. The scale and multiplier components follow COCOMO II (Boehm et al., 2000), a software effort model based on size, scale effects, and effort multipliers. Once the weights are fitted, adding tasks to a benchmark does not change the NDI scores of existing tasks.

Formula. For a task with L changed source lines, NDI combines the scale exponent E, multiplier M, and search cost S:

$$\mathrm{NDI} = \underbrace{L^E}_{\text{size, scaled}} \cdot \underbrace{M}_{\text{multipliers}} + \underbrace{S}_{\text{extra search cost}}, \tag{1}$$

where

$$E = 1 + s_F \log_2 F + s_G \log_2 G, \tag{2}$$

$$M = (1+U)^{w_U} (1+P)^{w_P} (1+T)^{w_T} \cdot 2^{\,g(Q)}, \tag{3}$$

$$S = c_R (1-R) + c_D D. \tag{4}$$

Here $Q \in \{1, \ldots, K\}$ is an LLM-assessed difficulty rating on a user-defined scale ordered from easiest to hardest, with

$$g(Q) = \sum_{k=2}^{K} u_k \, \mathbf{1}[Q \geq k], \qquad u_k \geq 0.$$

Each fitted weight u_k specifies the increment between adjacent rating levels, allowing the multiplier to increase without assuming equally spaced levels. All index weights are non-negative and fitted to agent outcomes. Table 1 defines the factors in each component.

Fitting. We fit NDI weights by maximizing the likelihood of observed agent outcomes under a logistic model of task failure, with one observation per agent run:

$$\mathrm{logit}\,\Pr(\text{fail}) = a \log_2 \mathrm{NDI} + \beta_{\text{repository}} + \gamma_{\text{agent}}. \tag{5}$$

The slope a relates log-transformed NDI to failure log-odds, while repository intercepts β and agent offsets γ account for differences in overall pass rates. These calibration terms are separate from the index itself. We jointly fit the index weights and calibration terms using L-BFGS-B (Byrd et al., 1995), constraining the index weights to be non-negative. Because the objective is not concave in the weights, we use multiple random initializations and retain the fit with the highest likelihood.

3 EXPERIMENTS

We evaluate the cost of tailoring benchmarks, the efficiency of collecting measurements, and NDI's ability to identify difficult tasks across RocksDB, Polars, and DuckDB using Claude Sonnet 5 in Claude Code 2.1.280, GPT-6-sol in Codex, and Kimi K3 in Kimi Code. swespoke's optimized early filtering reduces task-creation time by 7.27 to 12.48× and LLM expense by 2.39 to 4.85×. Optimizing measurement computation speeds up the collection of 63 measurements by 14.43 to 19.94×. NDI improves AUC over LOC by 0.07 and selects a hardest fifth with 7.9% higher failure rate.

3.1 SETUP

System environment. All experiments run on a server equipped with an Intel Xeon Platinum 8480CL processor, using 128 GiB memory. The software environment uses Rocky Linux 8.10, Python 3.12.14, spaCy 3.7.5 with its English model 3.7.1.

Datasets and agents. We follow the convention (Zhang et al., 2025b; Guo et al., 2025) to obtain 6,787 PRs from RocksDB, 13,521 from Polars, and 11,369 from DuckDB as the full set of candidates. We use the following agents: Claude Code 2.1.280 with Claude Sonnet 5 (high effort), Codex CLI 0.156.0 with GPT-6-sol (medium effort), and Kimi Code 2.1.0 with Kimi K3 (high effort) to solve the created tasks.

Baselines and implementation. Since no prior work supports user-defined task properties, we compare against baselines following typical implementations in the literature (Zhang et al., 2025b; Al-Haque & Johnson, 2026). For measurements, the baseline computes each feature independently in random order for each task. For filtering, the baseline only selects PRs with more than 50 changed source lines. swespoke further uses three NDI-based filters: >1 changed source file, >1 change location in at least one source file, and >2 assertions in at least one changed test. Each task in the creation pipeline receives two LLM assessments: one report action rates its difficulty (Badertdinov et al., 2026), and the other select action checks for mismatches between the test interfaces and the task description. For NDI, we compare against source lines of code (LOC), as used in SWE-bench Pro (Deng et al., 2026), and LLM rating of SWE-rebench V2 (Badertdinov et al., 2026).

Metrics. For efficiency, we report execution time and LLM expense. For difficulty, we report AUC (Hanley & McNeil, 1982) for predicting agent failure and failure rates among the hardest $k\%$ selected within each repository. Higher AUC indicates better concentration of difficult tasks.

3.2 MAIN RESULTS

Cost reduction by filtering. Figure 3(b) shows that early selection reduces candidates entering environment creation and verification from 2,055 to 202 in RocksDB, 4,542 to 341 in Polars, and 3,733 to 488 in DuckDB. As a result, time falls from 439 to 47 hours, 908 to 73 hours, and 798 to 110 hours, respectively; LLM expense falls from USD 1,409 to 590, 4,551 to 938, and 4,064 to 1,671.

Measurement efficiency. Figure 3(a) shows that collecting all 63 measurements in a difficulty study (Al-Haque & Johnson, 2026) with swespoke takes 68 seconds instead of 1,362 for RocksDB, 228 instead of 4,542 for Polars, and 761 instead of 10,980 for DuckDB. We break down the time in Table 2 and show that the main speedup comes from sharing NLP parsing, tokenization, and intermediate prompt statistics across measurements.

NDI effectiveness. Figure 4 shows that NDI concentrates agent failures more strongly than net LOC, with an aggregate AUC improvement of 0.07 (95% bootstrap interval: 0.02–0.12). The hardest 20% selected by NDI fail 7.9% more often than those selected by net LOC (0.8–13.9 points); the

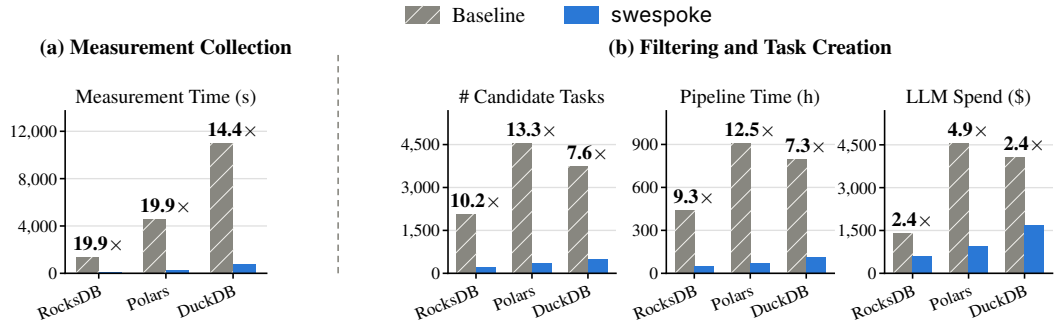

Figure 3: swespoke improves measurement collection speed and reduces task creation cost.

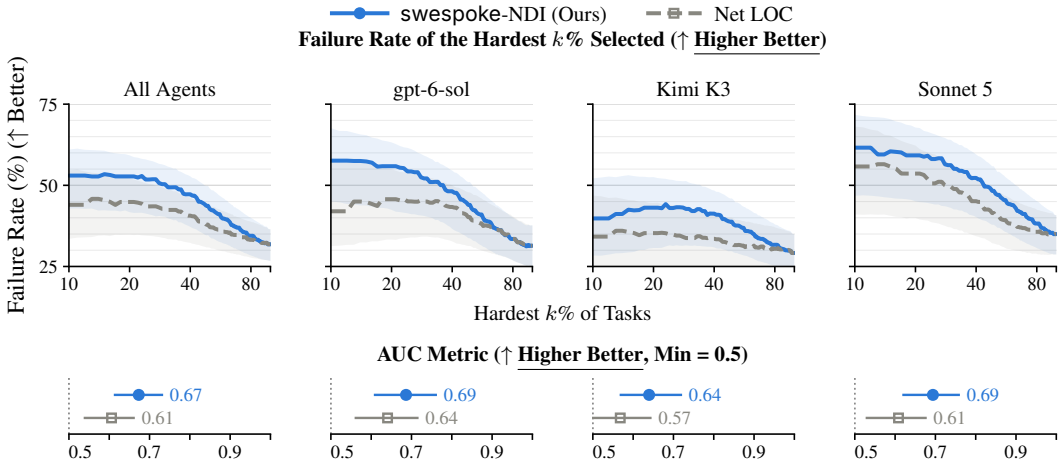

Figure 4: NDI identifies difficult tasks more effectively than LOC.

hardest-10% difference is 9.0 points, with a wider interval crossing zero (-0.4 to 16.0). Against the LLM rating alone, aggregate rounded AUC is equal at 0.67, but NDI also orders tasks within each rating level. More detailed results are in Appendix B.

3.3 Detailed Results

Measurement scaling. Figure 5(a) shows the aggregate speedup growing from 3.1× with 10 features to 16× with all 63; per-repository speedups reach 14 to 20×. The subsets are nested and balanced across patch, repository, and prompt categories as capacity permits. Each feature count uses different tasks, so changes between rows reflect both feature count and workload composition rather than feature count alone.

Measurement time breakdown. Table 2 identifies prompt calculation as the largest baseline cost. Sharing NLP parsing and prompt statistics reduces its time from 1,220 to 56 seconds for RocksDB, 4,075 to 186 for Polars, and 9,669 to 643 for DuckDB. The remaining rows include input/output, patch and repository calculation, and residual overhead.

Filtered task-creation cost breakdown. Figure 5(b) and Table 4 show that environment creation and verification dominate time, while both LLM assessments and environment setup contribute monetary expense. For Polars, early selection reduces environment creation from 579 to 43 hours and verification from 308 to 23 hours; environment-setup expense falls from USD 3,449 to 615 despite less reuse among the surviving tasks. Task counts distinguish the populations reaching the two assessments and the downstream stages.

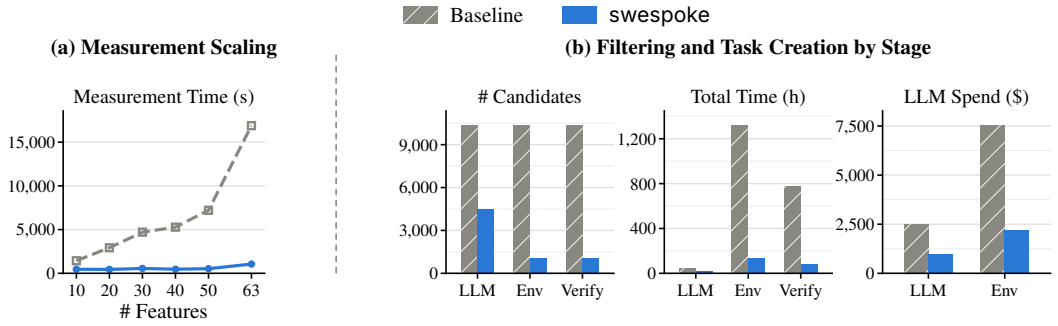

Figure 5: swespoke gains more from sharing as the number of measurements grows, and reduces all three downstream task creation costs through early filtering.

Table 2: Sharing prompt computation accounts for most of the measurement speedup.

Component	RocksDB		Polars		DuckDB	
	Base	Ours	Base	Ours	Base	Ours
Input/output	55.6	4.4	173.1	13.6	508.9	35.7
Patch-based measurements	20.3	1.6	55.2	4.4	25.5	2.2
Repository-based measurements	63.3	6.3	231.7	24.4	770.4	79.3
Prompt-based measurements	1,220	56	4,075	186	9,669	643
Other	2.8	0.2	6.7	0.5	6.7	0.4
Total	1,362	68	4,542	228	10,980	761

Fitted NDI. Fitting Equation 5 to a training split of 468 out of 669 agent runs gives

$$\mathrm{NDI} = L \cdot (1+T)^{3.16} \cdot 2^{4.41 H_M + 30 H} + 0.04\,(1-R). \tag{6}$$

Here H_M and H indicate medium-or-higher and hard LLM ratings. Due to the small training set, the other fitted factors have zero weight.

NDI ablation. Removing the LLM rating and refitting lowers AUC from 0.69 to 0.64 for Sonnet 5 and gpt-6-sol, and from 0.64 to 0.57 for Kimi K3 (Table 3). The hardest-10% pass rates rise from 37% to 40%, 41% to 51%, and 59% to 66%, respectively, showing that the rating contributes beyond structural features.

4 Related Work

Software engineering benchmarks. Coding agents combine language models with tools for inspecting repositories, editing code, and testing proposed changes. SWE-agent studies how interfaces for these operations affect an agent's ability to resolve repository issues (Yang et al., 2024). SWE-Gym provides executable repository tasks for training coding agents and verifiers, connecting task environments to both learning and inference-time selection (Pan et al., 2025). On the evaluation side, SWE-bench (Jimenez et al., 2024) pairs real issues with repository changes and execution-based tests, while SWE-Bench Pro (Deng et al., 2026) targets more demanding, long-horizon software engineering tasks.

Specialized benchmarks further distinguish evaluation targets: GITS-Eval (Rondon et al., 2025) draws on Google's internal bugs, and SWE-InfraBench (Tarasova et al., 2026) evaluates incremental changes to cloud infrastructure code. These efforts establish the value of executable, repository-level evaluation and of task populations chosen for particular capabilities or development settings. swespoke addresses these benchmark customization needs.

Automated benchmark construction. Automated construction reduces the effort required to turn repository changes into executable tasks. SWE-bench-Live (Zhang et al., 2025b), SWE-Factory (Guo et al., 2025), SWE-Bench++ (Wang et al., 2025), and SWE-rebench V2 (Badertdinov

Table 3: LLM difficulty ratings improve NDI's failure prediction and selection of difficult tasks.

		Overall						AUC per repository		
			Pass rate (%) of the hardest							
Agent	Index	AUC	10%	20%	40%	60%	100%	duckdb	rocksdb	polars
Sonnet 5	NDI (ours)	0.69	37	41	48	56	65	0.73	0.66	0.72
	No LLM	0.64	40	46	53	59	65	0.67	0.61	0.64
gpt-6-sol	NDI (ours)	0.69	41	44	52	61	69	0.70	0.71	0.62
	No LLM	0.64	51	54	57	62	69	0.65	0.69	0.54
Kimi K3	NDI (ours)	0.64	59	57	59	64	71	0.63	0.61	0.72
	No LLM	0.57	66	65	65	68	71	0.58	0.55	0.62

et al., 2026) automate parts of collection, environment preparation, and task validation, with different methods for scaling across repositories and languages. RepoLaunch (Li et al., 2026) addresses the underlying build-and-test problem by automating dependency resolution, compilation, and test-result extraction across languages and platforms, and uses these capabilities in a task-generation pipeline. SWE-smith (Yang et al., 2025) provides a complementary route by introducing code changes that break existing tests, allowing many task instances to share a repository environment. These systems scale task creation, whereas swespoke uses user-declared filters to guide it.

swespoke's separation of filter declaration from execution also connects to declarative AI data processing: Palimpzest (Liu et al., 2025), LOTUS (Patel et al., 2025), and DocETL (Shankar et al., 2025) expose higher-level processing operations, while Abacus (Russo et al., 2026) and Predicate Pushdown (Yan et al., 2023) study execution optimization and earlier filtering. However, applying these principles to task creation is nontrivial because it introduces new optimization opportunities and constraints.

Task selection and difficulty estimation. Prior work supports both selecting existing tasks by difficulty and influencing difficulty during task generation. SWE-rebench V2 releases task metadata for downstream filtering, and SWE-smith studies subsets stratified by estimated human implementation time as well as generation strategies that produce different difficulty distributions. SWE-Flow constructs test-driven development tasks from runtime dependencies and supports adjusting their difficulty through function-call complexity (Zhang et al., 2025a). Empirical work by Al-Haque and Johnson (Al-Haque & Johnson, 2026) associates patch fragmentation and repository scale with agent outcomes, supporting the use of structural measurements to estimate task difficulty. swespoke exposes difficulty as a customizable measurement within the same filter interface as other task properties. NDI demonstrates this use through a composite measure calibrated against agent outcomes, which users can apply when composing benchmarks at different difficulty levels.

5 Conclusion and Future Work

We presented swespoke, a framework for creating verified software engineering benchmarks from user-declared task properties. It lets software engineering teams tailor evaluations to their workloads and researchers isolate capabilities or construct training curricula. Users express their requirements through declarative filters. The optimizer orders filter evaluation and batches measurement computation to avoid unnecessary work during task creation. Experiments across three large open-source repositories show that swespoke reduces task-creation time and LLM expense while accelerating measurement collection. NDI demonstrates how a user-defined difficulty measure can select tasks with higher failure rates across three coding agents. Together, these results show the value of letting users specify the benchmarks they need while the system determines how to build them efficiently.

Currently, swespoke creates each task from a single PR. Future work includes automating task creation across multiple PRs, as demonstrated by FeatureBench (Zhou et al., 2026). Another direction is creating tasks that test security and safety properties when suitable examples are scarce in PR histories.

AI USE STATEMENT

AI assistants were used to polish the wording of the manuscript, to identify relevant literature, and to implement small, well-scoped code and experiment scripts. All ideas, methods, and results are the authors' own. All generated text and code was reviewed and tested.

REFERENCES

Ebtesam Al-Haque and Brittany Johnson. What makes software issue resolution tasks difficult for agents? In *ESEM*, 2026.

Anthropic. Introducing Claude Opus 4.7. `https://www.anthropic.com/news/claude-opus-4-7`, 2026. April 16, 2026.

Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. SWE-rebench V2: language-agnostic SWE task collection at scale. In *ICML*, 2026.

Barry W. Boehm, Chris Abts, A. Winsor Brown, Sunita Chulani, Bradford K. Clark, Ellis Horowitz, Ray Madachy, Donald J. Reifer, and Bert Steece. *Software Cost Estimation with COCOMO II*. 2000.

Richard H. Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 1995.

Aaron Chan, Ahmed Shalaby, Alexander Wettig, Aman Sanger, Andrew Zhai, Anurag Ajay, Ashvin Nair, Charlie Snell, Chen Lu, Chen Shen, Emily Jia, Federico Cassano, Hanpeng Liu, Haoyu Chen, Henry Wildermuth, Jacob Jackson, Janet Li, Jediah Katz, Jiajun Yao, Joey Hejna, Josh Warner, Julius Vering, Kevin Frans, Lee Danilek, Less Wright, Lujing Cen, Luke Melas-Kyriazi, Michael Truell, Michiel de Jong, Naman Jain, Nate Schmidt, Nathan Wang, Niklas Muennighoff, Oleg Rybkin, Paul Loh, Phillip Kravtsov, Rishabh Yadav, Sahil Shah, Sam Kottler, Alexander M. Rush, Shengtong Zhang, Shomil Jain, Sriram Sankar, Stefan Heule, Stuart H. Sul, Sualeh Asif, Victor Rong, Wanqi Zhu, William Lin, Yuchen Wu, Yuri Volkov, Yury Zemlyanskiy, Zack Holbrook, and Zhiyuan Zhang. Composer 2 technical report. *arXiv:2603.24477*, 2026.

Yang Chen, Zhuolin Yang, Zihan Liu, Chankyu Lee, Peng Xu, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. AceReason-Nemotron: Advancing math and code reasoning through reinforcement learning. In *NeurIPS*, 2025.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? In *ICML*, 2026.

Lianghong Guo, Yanlin Wang, Caihua Li, Pengyu Yang, Jiachi Chen, Wei Tao, Yingtian Zou, Duyu Tang, and Zibin Zheng. SWE-Factory: Your automated factory for issue resolution training data and evaluation benchmarks. *arXiv:2506.10954v1*, 2025.

James A. Hanley and Barbara J. McNeil. The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology*, 1982.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world Github issues? In *ICLR*, 2024.

Kenan Li, Rongzhi Li, Linghao Zhang, Qirui Jin, Liao Zhu, Xiaosong Huang, Geng Zhang, Yikai Zhang, Shilin He, Chengxing Xie, Xin Zhang, Zijian Jin, Bowen Li, Chaoyun Zhang, Yu Kang, Yufan Huang, Elsie Nallipogu, Saravan Rajmohan, Qingwei Lin, and Dongmei Zhang. RepoLaunch: Automating build&test pipeline of code repositories on ANY language and ANY platform. *arXiv:2603.05026*, 2026.

Chunwei Liu, Matthew Russo, Michael J. Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael J. Franklin, Tim Kraska, Samuel Madden, Rana Shahout, and Gerardo Vitagliano. Palimpzest: Optimizing AI-powered analytics with declarative query processing. In *CIDR*, 2025.

Chandra Maddila, Adam Tait, Claire Chang, Daniel Cheng, Nauman Ahmad, Vijayaraghavan Murali, Marshall Roch, Arnaud Avondet, Aaron Meltzer, Victor Montalvao, Michael Hopko, Chris Waterson, Parth Thakkar, Renuka Fernandez, Kristian Kristensen, Sivan Barzily, Sherry Chen, Rui Abreu, Nachiappan Nagappan, Payam Shodjai, Killian Murphy, James Everingham, Aparna Ramani, and Peter C. Rigby. Agentic program repair from test failures at scale: A neuro-symbolic approach with static analysis and test execution feedback. *IEEE TSE*, 2026.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym. In *ICML*, 2025.

Shubham Parashar, Shurui Gui, Xiner Li, Hongyi Ling, Sushil Vemuri, Blake Olson, Eric Li, Yu Zhang, James Caverlee, Dileep Kalathil, and Shuiwang Ji. Curriculum reinforcement learning from easy to hard tasks improves LLM reasoning. In *ICLR*, 2026.

Liana Patel, Siddharth Jha, Melissa Z. Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. Semantic operators and their optimization: Enabling LLM-based data processing with accuracy guarantees in LOTUS. *PVLDB*, 2025.

Pat Rondon, Renyao Wei, José Cambronero, Jürgen Cito, Aaron Sun, Siddhant Sanyam, Michele Tufano, and Satish Chandra. Evaluating agent-based program repair at Google. In *SEIP@ICSE*, 2025.

Matthew Russo, Chunwei Liu, Sivaprasad Sudhir, Gerardo Vitagliano, Michael J. Cafarella, Tim Kraska, and Samuel Madden. Abacus: A cost-based optimizer for semantic operator systems. *PVLDB*, 2026.

Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G. Parameswaran, and Eugene Wu. DocETL: Agentic query rewriting and evaluation for complex document processing. *PVLDB*, 2025.

Natalia Tarasova, Enrique Balp-Straffon, Aleksei Iancheruk, Yevhenii Sielskyi, Nikita Kozodoi, Liam H. Byrne, Jack Butler, Dayuan Jiang, Marcin Czelej, Andrew Ang, Yash Shah, Roi Blanco, and Sergei Ivanov. SWE-InfraBench: Evaluating language models on cloud infrastructure code. *arXiv:2606.05249*, 2026.

Lilin Wang, Lucas Ramalho, Alan Celestino, Phuc Anthony Pham, Yu Liu, Umang Kumar Sinha, Andres Portillo, Onassis Osunwa, and Gabriel Maduekwe. SWE-Bench++: A framework for the scalable generation of software engineering benchmarks from open-source repositories. *arXiv:2512.17419*, 2025.

Cong Yan, Yin Lin, and Yeye He. Predicate pushdown for data science pipelines. *PACMMOD*, 2023.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *NeurIPS*, 2024.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. In *NeurIPS*, 2025.

Lei Zhang, Jiaxi Yang, Min Yang, Jian Yang, Mouxiang Chen, Jiajun Zhang, Zeyu Cui, Binyuan Hui, and Junyang Lin. Synthesizing software engineering data in a test-driven manner. In *ICML*, 2025a.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. SWE-bench goes live! In *NeurIPS*, 2025b.

Qixing Zhou, Jiacheng Zhang, Haiyang Wang, Rui Hao, Jiahe Wang, Minghao Han, Yuxue Yang, Shuzhe Wu, Feiyang Pan, Lue Fan, Dandan Tu, and Zhaoxiang Zhang. FeatureBench: Benchmarking agentic coding for complex feature development. *arXiv:2602.10975*, 2026.

A Filtering Costs by Stage and Repository

Table 4 reports the per-repository stage counts, times, and expenses behind Figure 5(b).

Table 4: Task counts, stage time (hours), and LLM expense (USD). Early selection avoids environment and verification work; assessment counts are alignment/difficulty entrants.

Repository / stage	Tasks		Time (h)		Cost (USD)	
	Base	Ours	Base	Ours	Base	Ours
RocksDB						
Two LLM assessments	2,055/2,055	1,156/202	9.13	4.48	498.5	253.8
Environment creation	2,055	202	261.98	25.77	910.6	336.4
Task verification	2,055	202	167.80	16.51	0.0	0.0
Polars						
Two LLM assessments	4,542/4,542	1,457/341	20.19	5.70	1,102.1	322.4
Environment creation	4,542	341	579.08	43.48	3,449.0	615.3
Task verification	4,542	341	307.83	23.11	0.0	0.0
DuckDB						
Two LLM assessments	3,733/3,733	1,860/488	16.59	7.32	906.2	412.9
Environment creation	3,733	488	475.96	62.23	3,157.9	1,257.7
Task verification	3,733	488	304.87	39.86	0.0	0.0

B Full NDI Results

Table 5 reports the held-out AUC and the pass rate of the hardest $k\%$ of tasks for every score, every agent, and every repository. Every agent ran all 223 tasks.

Table 5: Held-out AUC and pass rate (%) of the hardest $k\%$ of tasks, by repository, agent, and score. The sets are cumulative, with 100% equal to the agent's overall pass rate in the repository; ties under the LLM rating are broken at random and counted by their expected pass rate. The hardest 10% holds 5 to 10 tasks per agent and repository.

Agent	Score	AUC	Pass rate (%) of the hardest 10%	20%	40%	60%	100%
duckdb							
Sonnet 5	NDI (ours)	0.73	48	46	55	65	74
	Net LOC	0.69	41	51	60	68	74
	LLM rating	0.73	52	52	53	64	74
gpt-6-sol	NDI (ours)	0.70	68	61	66	74	80
	Net LOC	0.70	61	65	67	73	80
	LLM rating	0.73	62	62	63	73	80
Kimi K3	NDI (ours)	0.63	70	67	72	78	82
	Net LOC	0.65	75	75	72	76	82
	LLM rating	0.67	69	69	70	76	82
rocksdb							
Sonnet 5	NDI (ours)	0.66	39	44	48	55	63
	Net LOC	0.54	53	48	57	63	63
	LLM rating	0.62	46	46	49	57	63
gpt-6-sol	NDI (ours)	0.71	27	35	44	54	64
	Net LOC	0.64	61	50	51	57	64
	LLM rating	0.66	40	40	45	57	64
Kimi K3	NDI (ours)	0.61	59	55	52	56	63
	Net LOC	0.49	70	64	63	64	63
	LLM rating	0.60	51	51	53	58	63
polars							
Sonnet 5	NDI (ours)	0.72	19	25	35	45	55
	Net LOC	0.62	29	37	44	50	55
	LLM rating	0.68	25	25	33	46	55
gpt-6-sol	NDI (ours)	0.62	25	35	47	55	60
	Net LOC	0.56	49	49	53	58	60
	LLM rating	0.68	38	38	42	50	60
Kimi K3	NDI (ours)	0.72	45	44	52	60	70
	Net LOC	0.60	40	51	65	68	70
	LLM rating	0.72	44	44	50	61	70

Table 6 reports the agents' mean cost per run over the same hardest-$k\%$ sets. Costs are list-price equivalents of each run's token usage: gpt-6-sol at the standard-tier price, twice the flex price at which it ran; Sonnet 5 at its list price, excluding the 5,894 tokens per request that the evaluation environment added; and Kimi K3 at the Fireworks list price. For every agent and repository, runs on the hardest 10% by NDI cost 1.5 to 2.0 times the agent's average run; net LOC and the LLM rating select tasks of similar cost.

Table 6: Mean agent cost (USD per run) of the hardest $k\%$ of tasks, by repository, agent, and score, on the same held-out sets as Table 5. The 100% column is the agent's mean cost over all its runs in the repository.

Agent	Score	Mean cost (USD per run) of the hardest 10%	20%	40%	60%	100%
duckdb						
Sonnet 5	NDI (ours)	6.73	6.48	5.52	4.54	3.67
	Net LOC	5.26	5.50	5.26	4.36	3.67
	LLM rating	5.85	5.85	5.74	4.62	3.67
gpt-6-sol	NDI (ours)	1.07	1.02	0.96	0.85	0.72
	Net LOC	1.04	1.01	0.89	0.80	0.72
	LLM rating	1.01	1.01	1.00	0.85	0.72
Kimi K3	NDI (ours)	6.97	6.61	5.73	4.90	3.95
	Net LOC	7.44	6.62	5.29	4.55	3.95
	LLM rating	5.91	5.91	5.81	4.82	3.95
rocksdb						
Sonnet 5	NDI (ours)	4.64	4.16	3.79	3.25	2.62
	Net LOC	4.83	4.30	3.90	3.33	2.62
	LLM rating	4.20	4.20	3.92	3.24	2.62
gpt-6-sol	NDI (ours)	1.40	1.28	1.13	1.00	0.83
	Net LOC	1.44	1.29	1.10	0.94	0.83
	LLM rating	1.24	1.24	1.17	0.99	0.83
Kimi K3	NDI (ours)	6.81	6.02	5.32	4.51	3.58
	Net LOC	6.37	5.65	5.05	4.29	3.58
	LLM rating	5.87	5.87	5.45	4.47	3.58
polars						
Sonnet 5	NDI (ours)	7.09	6.72	5.35	4.54	3.53
	Net LOC	8.50	7.15	5.38	4.58	3.53
	LLM rating	5.78	5.78	5.29	4.43	3.53
gpt-6-sol	NDI (ours)	1.00	1.04	0.94	0.81	0.65
	Net LOC	1.01	1.01	0.84	0.76	0.65
	LLM rating	1.05	1.05	0.96	0.80	0.65
Kimi K3	NDI (ours)	7.59	7.58	6.67	5.72	4.49
	Net LOC	8.15	7.57	6.26	5.67	4.49
	LLM rating	7.63	7.63	6.95	5.78	4.49