XLGym: API-Grounded Task Synthesis for Training and Evaluating Spreadsheet Agents

Anonymous authors
Paper under double-blind review

Abstract

Spreadsheet agents act on tables, charts, PivotTables and conditional formats, yet existing benchmarks are dominated by formula, computation and domain-oriented queries, which frontier models are getting increasingly good at. We introduce XLGym, a pipeline that turns Excel API documentation and real workbooks into hard tasks with reference solutions. Each task is grounded in a specific API member, verified by execution in Excel, and hardened until frontier models no longer solve it reliably. Because tasks are seeded per API member and per artifact, coverage and difficulty are set by construction. With XLGym we build XLBench, a 190-task benchmark that covers all eight major artifact types and gives every frontier model its lowest score among the benchmarks we evaluate (Opus 5 52.8%, GPT-5.6 35.4%), and XLTrain, a 20.6k-task corpus spanning 1,287 API members. Fine-tuning Qwen3.6-27B on XLTrain yields XLSage-27B, which raises the pass rate from 1.8% to 31.9% on XLBench, from 3.1% to 31.6% on WTM-Bench and from 2.5% to 22.2% on SpreadsheetBench. Because XLGym can regenerate tasks against newer models, XLBench can continually evolve.

1 Introduction

Spreadsheets are the most widely used end-user programming environment, with more than 750 million users (Bendre et al., 2019), and a growing share of spreadsheet work is now delegated to LLM agents that operate inside the workbook. Microsoft, Anthropic and OpenAI all ship agents that plan and edit across Excel workbooks (Microsoft, 2025; Anthropic, 2026; OpenAI, 2026), and AI assistance is estimated to save about 80% of the time on financial-analyst tasks (Tamkin & McCrory, 2025). Research systems have followed, from agents that plan a request as spreadsheet operations (Li et al., 2023) to agents trained with reinforcement learning (Chi et al., 2026).

Public benchmarks, however, measure a narrow and easy slice of this work, with two flaws (Section 2). *Artifact coverage*: real workbooks hold structured tables, charts, pivot tables, conditional formats, filters, sort states and hyperlinks alongside formulas (Uniyal et al., 2026), yet contemporary benchmarks oversample formula and filtering queries, cover only a few of the other artifact types (Section 4), and, following the frequency of real requests, rarely reach the long tail of operations each artifact supports. *Query hardness*: A fixed collection cannot get harder as models improve; keeping pace requires a pipeline that produces harder data as models get better. The coverage gap is partly a property of the interface. Most benchmarks are built on Python libraries that lack support for many native Excel artifacts and attributes (Gazoni & Clark, 2024) (Figure 2). Office.js, the JavaScript API that Excel's own agents use (Microsoft, 2026a), exposes every native workbook object and runs through Excel's calculation engine (Microsoft, 2026b); with it, frontier models score higher on most existing benchmarks than with Python (Table 3, Appendix B).

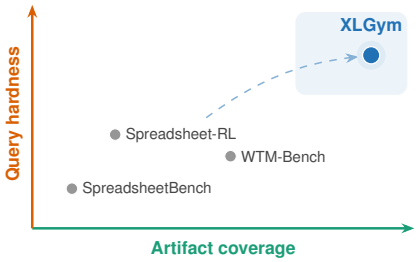


Figure 1: XLGym synthesises tasks that are both broad in artifact coverage and hard for frontier models.

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.

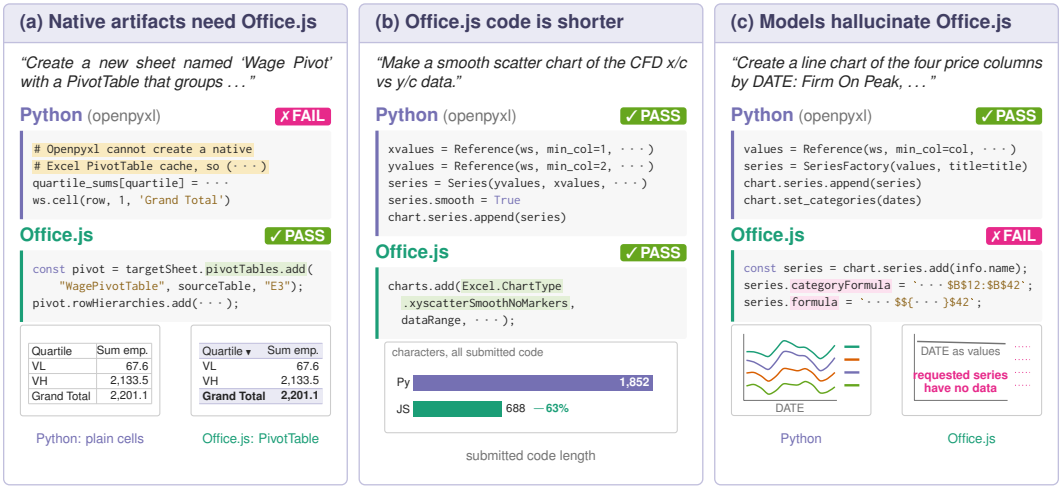


Figure 2: Python (openpyxl) vs. Office.js on three tasks. (a) openpyxl cannot create a native pivot table, so the Python solution writes plain cells. (b) Office.js code for the same chart is shorter. (c) Models also hallucinate Office.js members, which is where the Office.js solution fails. Complete code for each panel is in Appendix D.

Building a benchmark without these flaws is hard because frontier models are not fluent in Office.js. Although they reach higher pass rates with Office.js than with Python, their Office.js programs fail to execute more often (Table 5, Appendix B), mostly by misreading API members (Section 3.2).

We propose XLGym, a pipeline that generates diverse and hard Excel tasks with verified solutions. A frontier LLM, the *generator*, writes each task (a query, a reference Office.js program and an expected answer) from the documentation of one Office.js API member and a real workbook containing the artifact that API acts on. Because the generator is itself not great at Office.js, its drafts are often wrong, and tasks it solves easily are easy for other models too. XLGym wraps it in two feedback loops (Figure 3). A *consistency loop* helps it reach a correct solution when the query is hard: we run the reference program in Excel, and a verifier checks the resulting workbook against the query and expected answer, returning any mismatch. A *hardness loop* makes queries harder when they are too easy: a solver model attempts each verified task five times, and if three or more attempts pass, the generator is told why and rewrites the task, which re-enters the consistency loop. 91.8% of kept tasks were revised by at least one loop. Our contributions are:

- XLGym, an API-grounded pipeline for synthesising Excel tasks, whose two feedback loops make generated tasks both correct and hard (Section 3).
- XLBench, a 190-task benchmark stratified over eight artifact types. It gives frontier models their lowest score among the benchmarks we evaluate (Section 4).
- XLTrain, a training corpus of about 20.6k tasks. Fine-tuning Qwen3.6-27B on its own rewrites of the reference programs gives XLSage-27B, which raises the pass rate from 1.8% to 31.9% on XLBench and from 3.1% to 31.6% on WTM-Bench, above GPT-5.4 Mini and Haiku 4.5, and we ablate corpus size and training target (Section 5).

2 Related Work

Spreadsheet and office benchmarks. InstructExcel pairs over 10k instructions with Office Scripts recorded from user actions, but grades predictions by textual similarity rather than by execution (Payan et al., 2023). SpreadsheetBench (Ma et al., 2024) and SheetCopilot (Li et al., 2023) evaluate atomic manipulations drawn from user questions or written by hand, and WTM-Bench rebalances artifact types by reverse-engineering user-authored workbooks (Uniyal et al., 2026). A recent wave targets end-to-end professional workflows, mostly in finance (Zhu et al., 2026; Dong et al., 2025; Lau et al., 2026; Yen et al., 2026; Kundurthy et al., 2026b;a), and broader suites test

conversational data analysis and multi-application office work (Dutta et al., 2025; Wang et al., 2024; 2025a). OfficeEval finds that once execution errors are fixed, almost all remaining loss comes from API knowledge such as wrong property paths (Lv et al., 2026), the failure mode XLGym targets. These benchmarks reflect realistic usage, which makes them valuable for measuring how agents handle typical requests, but miss out on tail cases. Most are also built on Python, so their artifact distribution has been partly shaped by its capabilities, which cover few native Excel artifacts (Gazoni & Clark, 2024) (Figures 2 and 4). XLGym is complementary: it synthesises tasks from API documentation, so artifact coverage and difficulty are set by construction.

Spreadsheet agents. Most spreadsheet agents improve the scaffold, by planning over atomic operations or Python (Li et al., 2023; Chen et al., 2025; Zhu et al., 2025; Liang et al., 2025) or by better workbook encodings and retrieval (Dong et al., 2024; Wang et al., 2025b; Ren et al., 2026; He et al., 2026; Gulati et al., 2026; Cheng et al., 2026b). Fewer train the model itself, with supervised fine-tuning (Zhang et al., 2024) or reinforcement learning (Chi et al., 2026; Lau et al., 2026). Nearly all use Python or LibreOffice, limiting their artifact coverage; XLTrain supplies execution-verified data for Office.js, the interface of production Excel agents (Microsoft Excel Team, 2025).

Data synthesis for Low-Resource Languages & APIs Code LLMs degrade on languages and libraries that are rare in pretraining. Prior work transfers knowledge through translated, test-verified data (Cassano et al., 2024; Wu et al., 2026), trains on language-agnostic or compiler-verified signals (Boruch-Gruszecki et al., 2025; Samaraweera et al., 2026), fine-tunes on the little data available (Giagnorio et al., 2026; Kier et al., 2026; Cheng et al., 2026a), retrieves documentation at inference time (Zhou et al., 2023; Patil et al., 2023; Wijaya et al., 2025; Li et al., 2026), or synthesises API-grounded training data (Qin et al., 2023; Liu et al., 2026b; Zhang et al., 2026). Models also emit outdated calls for evolving APIs (Wu et al., 2024; Liu et al., 2024; Wu et al., 2025; Liu et al., 2026a). Office.js sits in this regime: its syntax is ordinary JavaScript, but its object model is large and rare in public code. Beyond execution-graded benchmarks (Jimenez et al., 2024; Merrill et al., 2026), synthetic environments now calibrate difficulty against solvers: CalibForge revises terminal tasks until a strong solver passes and a weak one fails (Meng et al., 2026). XLGym combines documentation-grounded, execution-verified synthesis with a solver gate that also hardens tasks.

3 XLGym Pipeline

XLGym turns Office.js API documentation and real workbooks into spreadsheet tasks with verified solutions. Its design follows two principles. Tasks are *API-grounded*: each task starts from one Office.js API member and is solved cleanly by using it. Tasks are *hard*: a model that does not know the API well should fail, so a task is kept only when a strong solver model fails on it in most attempts.

3.1 Pipeline Overview

A task is a triple $(q, c^\star, a^\star)$: a natural-language query q, a reference Office.js program $c^\star$ that carries out q, and an expected answer $a^\star$ that describes the resulting workbook changes or printed output (Figure 3).

① **API–workbook matching.** Each API member in the Office.js documentation is tagged with the artifact it acts on (table, chart, PivotTable, filter, ...). We pair it with a real workbook from the Enron (Hermans & Murphy-Hill, 2015) or FUSE (Barik et al., 2015) corpus that already contains that kind of object, so the task has something realistic to act on.

② **Task generation and consistency check.** A *generator* LLM reads the documentation page and a structured summary of the workbook (sheets, used ranges, objects and a preview of cell values) and writes $(q, c^\star, a^\star)$. We then run $c^\star$ in Excel on a copy of the workbook and extract its state before and after the run (values, formulas, formats and object metadata) together with the printed output. A *query verifier* LLM checks that this evidence satisfies q and matches $a^\star$. If not, the verifier explains the mismatch and the generator revises the triple; if the triple fails three consistency checks in a round, it is discarded.

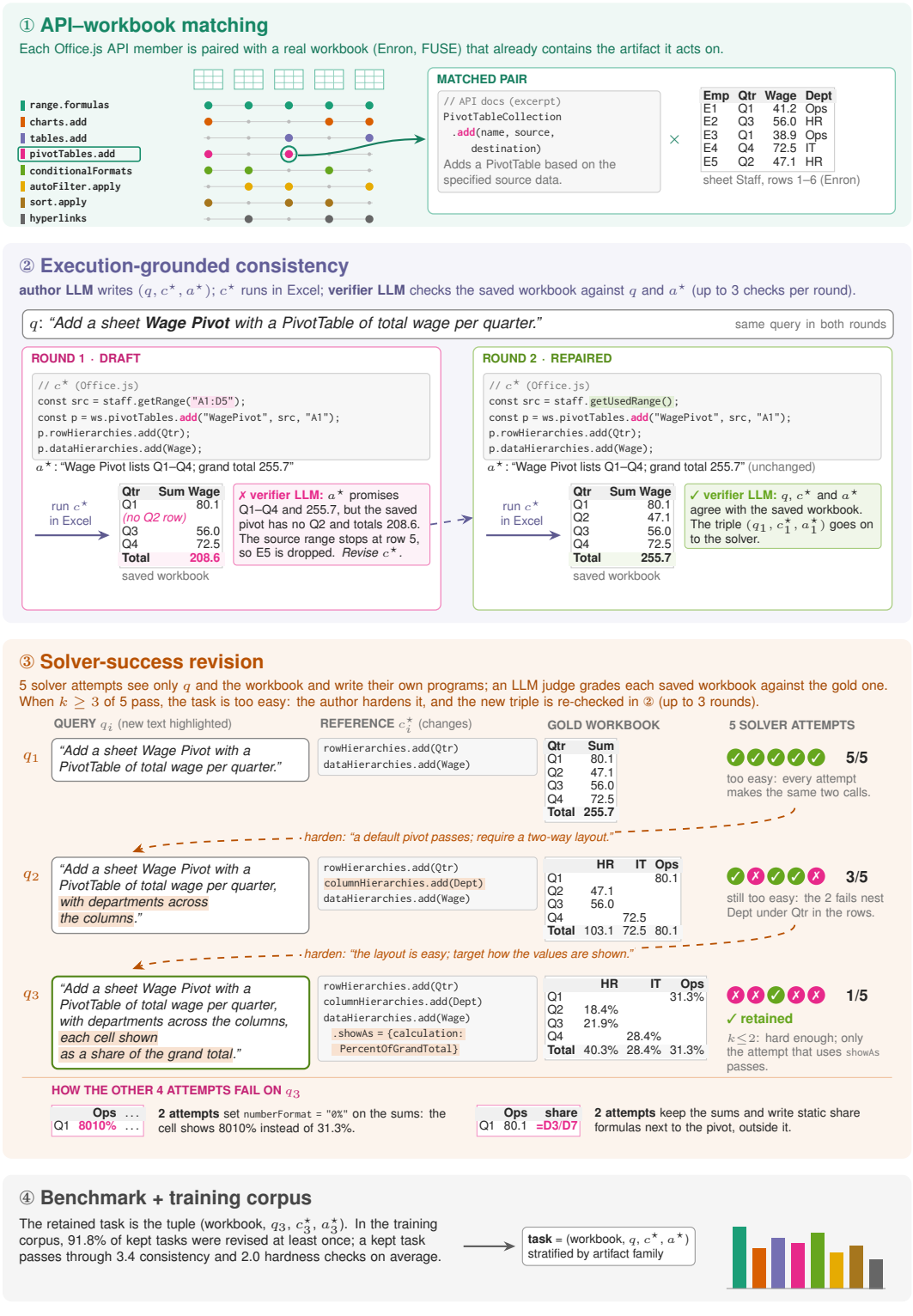


Figure 3: The XLGym pipeline on one illustrative task (code and queries abbreviated). ① The API member `pivotTables.add` is paired with a real workbook that has pivotable data. ② The draft reference program runs without error, but the saved workbook contradicts the expected answer, so the verifier sends it back and the author repairs it. ③ Solvers pass the first two versions of the query too often, so the author hardens the task twice; the third version is kept when only one of five solver attempts passes. ④ Retained tasks form the benchmark and training corpus.

③ **Hardness check.** A *solver* LLM (GPT-5.6 Reasoning Sol, the same model as the generator) makes five independent attempts, seeing only q and the workbook, never $c^\star$ or $a^\star$. Each attempt writes an Office.js program, which runs in Excel on its own copy of the workbook, with up to three retries after an execution error, as in execution-feedback agents (Yao et al., 2023; Chen et al., 2023). A *solution verifier* LLM compares each attempt's saved workbook and output with those of $c^\star$ and with q, and counts the k attempts that pass. If $k \leq 2$, the task is kept. If $k \geq 3$, it is too easy: the generator receives feedback on why the solver succeeded and makes the task harder, and the new triple goes through the consistency check again. We allow up to three hardness rounds per task (the first check and up to two rewrites).

3.2 Task Evolution

Few tasks survive their first draft. In our corpus, 91.8% of kept tasks were revised at least once and 70.4% were revised by both verifiers; a kept task went through 3.4 consistency checks and 2.0 hardness checks on average. Figure 3 shows both kinds of revision on one task. The query verifier mostly changes the reference and the expected answer. It catches programs that call functions Office.js does not provide, expected answers that contradict the saved workbook, and programs that run without error but leave the workbook wrong. The last kind is visible only after execution. The solution verifier changes the query. Typical revisions replace fixed cell addresses with conditions on the data, ask for additional outputs, or move the task to a behaviour of the API that is easy to misread. The added difficulty usually comes from the API or from a longer query. In another task, a draft asking whether a table was filtered could be answered from one flag, and four of five solver attempts passed. The revision asked which columns had an active filter. Office.js returns a placeholder rule of type `"BottomItems"` for a column with no filter, and all five attempts reported every column as filtered.

Generation mechanics. Table 1 follows the pipeline over the run to generate XLTrain and XLBench. Of 46.6k (API, workbook) seeds, 21.2k tasks (45.5%) pass both checks. We generated tasks for each artifact until it reached about 3.5k kept tasks or ran out of seeds (filters, sorts and hyperlinks), so the corpus is balanced by construction rather than by how often an artifact appears in the source workbooks. The cost of that balance depends on the artifact: a kept chart task costs 3.6 seeds, a hyperlink 1.2. Coverage takes two forms. Charts, conditional formats and PivotTables have large API surfaces (558, 292 and 224 members), and their tasks are spread thinly over them, at 6–16 tasks per member. Formulas and sorts expose few members (38 and 17), so each member yields many tasks (93 and 79), and their variety comes from the workbooks and from query evolution rather than from the API. For formulas the difficulty also lies mostly in Excel's formula language, which Office.js only passes through. In total the corpus exercises 1,287 distinct API members, the artifact depth that benchmarks derived from user edits lack.

Hardening and correctness pull against each other. A kept task passes 3.44 consistency checks over 1.98 hardness rounds, about 1.7 per round: a harder query needs a new reference program, and that program often fails the consistency check again. Charts need the most repairs per round (1.93). The generator (GPT-5.6 Reasoning Sol) is also the solver, and as a generator it makes the same kind of API errors that it makes when solving. Yield also tracks where models are weak. Across the eight artifacts, yield correlates with the mean pass rate of the four frontier models on XLBench (Spearman $\rho = 0.79$, $p = 0.02$; Section 4). Sorts and hyperlinks are easy to generate, need the fewest hardening rounds (1.39 and 1.60) and are the easiest to solve; tables and PivotTables have the lowest yields after charts (37.0% and 47.7%) and are the two hardest artifacts when averaged over models. Charts are the exception: their low yield reflects the breadth of the chart API more than its difficulty.

4 Data Diversity and Benchmarking

XLBench is a stratified set of 190 tasks from the generation run described in Section 3, on workbooks that XLTrain does not use: 25 tasks per artifact type and 15 for structured tables. We compare it with the verified split of SpreadsheetBench (Ma et al., 2024), WTM-Bench (Uniyal et al., 2026) and Spreadsheet-RL (Chi et al., 2026), and evaluate Opus 5, GPT-5.6, GPT-5.4 Mini and Haiku 4.5 on all four. The model receives the query and a summary of the workbook (its artifacts, data ranges and a preview of values) and writes one Office.js program, which we run in Excel and grade with

Table 1: XLGym generation statistics by artifact family. Each seed is an (API, workbook) pair; a task is retained if its reference program passes execution-grounded consistency checking and at most two of five solver candidates pass. Tasks/API and tasks/WB are computed over retained tasks. †Mean per retained task; the budgets are three consistency checks per hardness round and three hardness rounds (i.e., at most nine checks).

	Funnel			Coverage				Effort†	
Artifact	Seeds	Kept	Yield (%)	APIs	WBs	Tasks /API	Tasks /WB	Ref. checks	Hard. rounds
Formula	6,055	3,517	58.1	38	55	92.6	63.9	3.72	2.21
Chart	12,723	3,520	27.7	558	139	6.3	25.3	3.77	1.95
Table	9,479	3,505	37.0	110	745	31.9	4.7	3.65	2.24
PivotTable	7,351	3,509	47.7	224	526	15.7	6.7	3.13	1.90
Cond. format	5,885	3,512	59.7	292	191	12.0	18.4	3.51	2.00
Filter	2,925	1,995	68.2	39	46	51.2	43.4	3.11	1.70
Sort	1,819	1,350	74.2	17	92	79.4	14.7	2.53	1.39
Hyperlink	378	316	83.6	9	32	35.1	9.9	2.97	1.60
Total	46,615	21,224	45.5	1,287	853	16.5	24.9	3.44	1.98

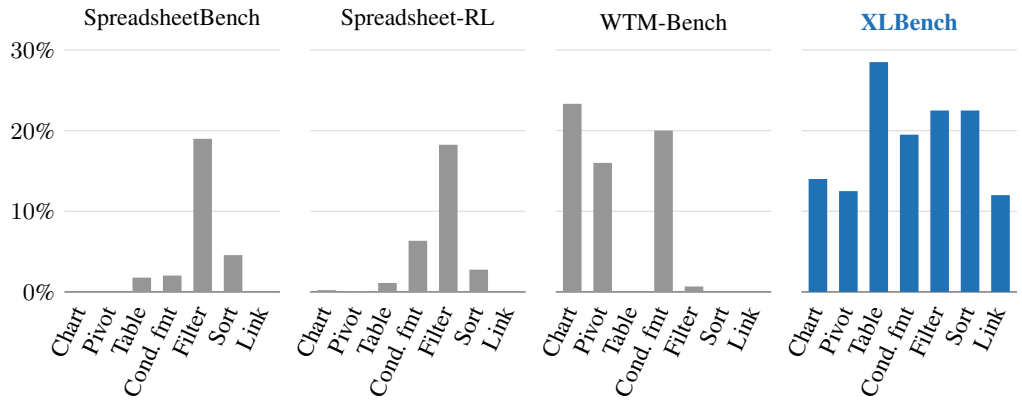


Figure 4: Share of queries that create, modify or inspect each non-formula artifact type.

the solution verifier of Section 3.1. We set reasoning effort to none for all models, proprietary and open-weight, and report the execution rate and the pass rate. Every result in the paper is a mean over three runs. We focus on a single step rather than a multi-step loop with execution feedback (Appendix A details both) because it measures what the model knows about Office.js before any error message can correct it. For the two frontier models we compare in both harnesses, it gives the same ranking as the multi-step harness on all three external benchmarks, although multi-step scores are higher (Table 4, Appendix B).

Coverage. We annotate each query with the artifacts it asks to create, modify or inspect, using an LLM that sees only the query text. We validate our annotation results against the original benchmark papers where available. Every benchmark covers formulas well. Beyond formulas, each existing benchmark covers a narrow slice of Excel (Figure 4; we leave formulas out of this figure as it skews the other artifact types). SpreadsheetBench and Spreadsheet-RL are dominated by formulas (58% and 53% of queries) and filters (19% and 18%); at most 0.2% of their queries involve charts, PivotTables or hyperlinks. WTM-Bench covers formulas (40%), charts, conditional formats and PivotTables (16–23%), the artifact types it was built around, but none of its queries involve structured tables, sort states or hyperlinks. XLBench covers all artifact types well, each in 12–29% of tasks.

Difficulty. XLBench gives every frontier model its lowest pass rate and its lowest execution rate (Table 2). Opus 5 solves 52.8% of its tasks, against 55.8–71.4% on the other benchmarks; GPT-5.6

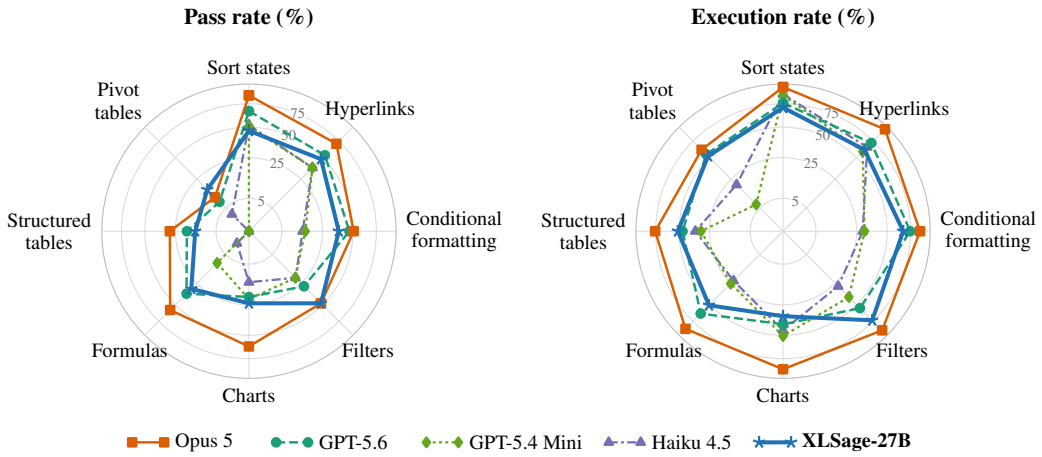


Figure 5: Per-artifact pass rate (left) and execution rate (right) on XLBench for four frontier models and for XLSage-27B (Qwen3.6-27B fine-tuned on XLTrain; Section 5). Radius on a square-root scale; rings mark 5, 25, 50 and 75%.

Table 2: Office.js execution rate and task success (pass rate) per model under the single-step harness, in %, with reasoning effort set to none for all models. XLSage-27B is Qwen3.6-27B fine-tuned on the 20.6k teacher corpus of Section 5.3; all results are means over three runs.

	XLBench		SpreadsheetBench		WTM-Bench		Spreadsheet-RL	
Model	Exec.	Pass	Exec.	Pass	Exec.	Pass	Exec.	Pass
Frontier models								
GPT-5.6	60.88	35.4	81.77	57.9	72.00	50.4	82.21	45.7
Opus 5	85.79	52.8	95.95	71.4	88.44	60.2	95.10	55.8
GPT-5.4 Mini	41.58	20.2	74.09	34.1	57.11	23.8	75.75	28.2
Haiku 4.5	41.93	18.3	68.35	37.2	58.22	28.0	73.89	31.4
Open-source models								
Qwen3.6-27B	5.61	1.8	6.84	2.5	5.56	3.1	22.01	5.2
XLSage-27B	52.28	31.9	57.64	22.2	63.11	31.6	63.01	16.8

solves 35.4%, against 45.7–57.9%; and the two small models solve only about a fifth. Part of this is by design, since the pipeline keeps only tasks that the solver, GPT-5.6 Reasoning Sol, fails in most of its five attempts.

Where models fail. PivotTables and structured tables are among the three hardest artifact types for every model (Figure 5): the best frontier model solves 10.7% and 28.9% of them, and the small models almost none. These are also the types existing benchmarks barely cover. Execution and pass rates show two kinds of failure. Opus 5 and GPT-5.6 write PivotTable programs that mostly run (61% and 55%) but are rarely right (17% and 15% of the programs that run): they misread what an API member does, as in the filter example of Section 3.2. GPT-5.4 Mini fails earlier: only 7% of its PivotTable programs run. Both reflect missing Office.js knowledge, which Section 5 shows can be taught.

5 Teaching Office.js with XLTrain

Section 4 showed that frontier models fail on XLBench mostly for lack of Office.js knowledge. We now ask whether fine-tuning on XLGym tasks can teach it to an open-weight model.

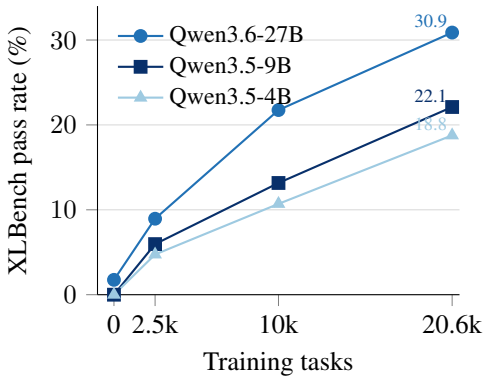


Figure 6: XLBench pass rate of Qwen models fine-tuned on 2.5k, 10k and 20.6k reference programs (0 = base model). Every model improves at every step.

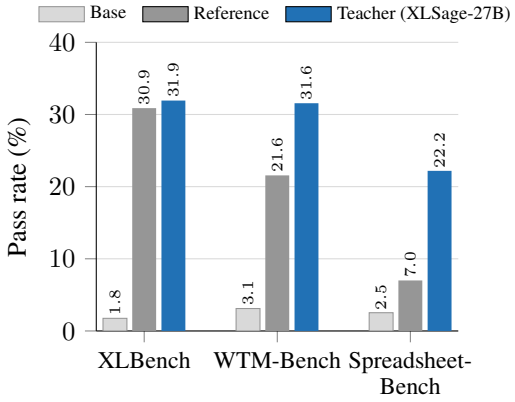


Figure 7: Qwen3.6-27B fine-tuned on 20.6k tasks with reference or teacher-rewritten targets (the latter is XLSage-27B). Teacher targets match or beat the references in-domain and generalise far better out-of-domain.

5.1 Setup and Main Result

Training data. XLTrain holds 20,554 tasks from the pipeline run of Table 1, all on workbooks that XLBench does not use, so the two share no workbook (hence fewer than the 21.2k kept tasks). Each example pairs the evaluation input, a query and a workbook summary, with an Office.js program as the target. We fine-tune with supervised learning and evaluate in the single-step harness of Section 4, on XLBench (in-domain) and on WTM-Bench, SpreadsheetBench and Spreadsheet-RL (out-of-domain). We fine-tune all parameters for two epochs with AdamW, a global batch of 32 sequences and a peak learning rate of 10^{-5} with cosine decay; full hyperparameters are in Table 6 (Appendix C).

Fine-tuning teaches Office.js. Table 2 reports our main model, XLSage-27B: Qwen3.6-27B fine-tuned on the 20.6k teacher corpus, in which the model rewrites each reference program as its own solution (Section 5.3). The base model almost never writes Office.js that runs: 5.6% of its XLBench programs execute and 1.8% pass. After fine-tuning, 52.3% execute and 31.9% pass, within 3.5 points of GPT-5.6 (35.4%) and well above GPT-5.4 Mini (20.2%) and Haiku 4.5 (18.3%). The gains transfer out of domain. On WTM-Bench the pass rate rises from 3.1% to 31.6%, above both small frontier models. On SpreadsheetBench it rises from 2.5% to 22.2%, but stays below them. SpreadsheetBench is dominated by formula tasks (Section 4), which make up only 17% of XLTrain, and grades with a static checker that compares cell values in fixed answer ranges, so it rewards writing values into the right cells rather than using native artifacts.

Single-step training transfers to multi-step use. We train only on single programs, but agents in practice iterate: they run code, read the error and try again. Figure 8 (Appendix C) evaluates Qwen3.6-27B and XLSage-27B in both harnesses, i.e. before and after fine-tuning on the 20.6k teacher corpus (Section 5.3). With feedback, even the base model does far better than in a single step (19.3% on XLBench against 1.8%), because it can repair code that fails. Fine-tuning adds to this rather than replacing it: the multi-step pass rate nearly doubles on XLBench, from 19.3% to 36.5%, and rises on WTM-Bench from 29.7% to 44.5%. SpreadsheetBench is again the exception: its multi-step pass rate falls by about four points, for the reasons above.

5.2 Scaling Data and Model Parameters

We fine-tune Qwen3.6-27B, Qwen3.5-9B and Qwen3.5-4B on 2.5k, 10k and 20.6k tasks (Qwen3.6 has no smaller variants, so we use Qwen3.5 for 9B and 4B), using the reference programs $c^\star$ as SFT targets (Figure 6). XLBench pass rate rises with data for every model, with no sign of saturation at

20.6k: from 1.8% to 30.9% for the 27B model, and from 0% to 22.1% and 18.8% for the 9B and 4B models. Since XLGym generates tasks on demand, more data is a matter of compute. At 20.6k tasks the 9B model already passes GPT-5.4 Mini on XLBench (22.1% against 20.2%) and the 4B model matches Haiku 4.5 (18.8% against 18.3%).

5.3 Training on the Model's Own Solutions

The reference programs are written by the generator, GPT-5.6 Reasoning Sol. For the model being fine-tuned they are off-policy and unlike what the model would write itself. Supervised fine-tuning on off-policy targets moves a model further from its base distribution than training on its own outputs, and the size of that shift predicts how much it forgets elsewhere (Shenfeld et al., 2025; Agarwal et al., 2024). We therefore let the model rewrite the references before training on them, following self-distillation fine-tuning (Yang et al., 2024): Qwen3.6-27B receives the query, the workbook summary, the documentation page of the task's API member and the reference program as privileged context, and writes its own solution. The same idea of a model teaching itself from a privileged reference underlies rationalisation in STaR (Zelikman et al., 2022) and recent on-policy self-distillation methods (Shenfeld et al., 2026; Zhao et al., 2026; Hübotter et al., 2026); we use its simplest, offline form. Conditioning on the reference lets the teacher reliably reproduce its solution, while the rewrite is expressed in the model's own style, so the corpus keeps the reference's content but becomes in-distribution, or on-policy, for the model being trained. The teacher corpus holds 20,554 examples, and we compare it with training directly on the references.

Figure 7 shows the effect at 20.6k. Teacher targets keep the in-domain gains and generalise far better out of domain. Compared with the references, XLBench pass rate rises from 30.9% to 31.9%, WTM-Bench from 21.6% to 31.6% and SpreadsheetBench from 7.0% to 22.2%. The same holds at every corpus size (Table 7, Appendix C): the teacher corpus gives the higher pass rate on all three benchmarks at 2.5k, 10k and 20.6k. The gap is widest out of domain and grows with data. On SpreadsheetBench, training on the references lowers the pass rate as the corpus grows (11.7%, 10.4%, 7.0%), consistent with forgetting under off-policy targets, while training on the teacher corpus raises it (16.5%, 18.4%, 22.2%). We use the 20.6k teacher model as our main model.

6 Limitations and Conclusion

Limitations. Task construction and evaluation rely on LLM verifiers; grounding them in executed workbooks removes many errors a judge reading code alone would miss, but we have not validated them against human annotators. A task is kept when at most two of five solver attempts pass, so XLBench is hard by construction for the solver (GPT-5.6 Reasoning Sol) and similar models, although the per-artifact differences in Figure 5 are not explained by the filter alone. Each task targets one API member and is solved by one program, so XLGym does not yet produce long-horizon tasks; workbooks come from two corpora and all references from one generator (GPT-5.6 Reasoning Sol). We train with offline SFT only; on-policy self-distillation (Shenfeld et al., 2026; Zhao et al., 2026) and RL in the XLGym environment are natural next steps, as is evaluating XLSage in production agent scaffolds with their own tools and prompts.

Conclusion. Spreadsheet agents increasingly act on rich artifacts, yet the benchmarks used to measure them cover little beyond formulas and are rarely hard for current models. XLGym turns Office.js API documentation and real workbooks into tasks that are grounded in specific API members, verified by execution and filtered for difficulty, and lets us control artifact coverage, depth and hardness together. The resulting benchmark, XLBench, is the only one we examined that covers every major Excel artifact, and it shows that frontier models are far from fluent: Opus 5 solves 52.8% of its tasks, and PivotTables and structured tables remain hard for every model, mostly because models misread or invent API members. The same pipeline produces a training corpus, XLTrain, that teaches this knowledge: fine-tuning Qwen3.6-27B into XLSage-27B raises the XLBench pass rate from 1.8% to 31.9%, approaching GPT-5.6, with gains that carry over to benchmarks it was not trained on. We release the pipeline, XLBench, XLTrain and the XLSage models. Because XLGym can regenerate tasks against newer models, XLBench can continually evolve rather than saturate. More broadly, we see XLGym as a basis for teaching models any API they are not yet fluent in, given its documentation and an environment to execute it.

AI USE STATEMENT

LLMs are part of the method studied in this paper: GPT-5.6 Reasoning Sol generates the XLGym tasks and reference programs and also serves as the solver that gates task hardness, and Qwen3.6-27B writes its own training targets (Sections 3 and 5). Separately, the authors used LLM assistants to help write code for the pipeline and experiments and to refine the wording of the paper. All research ideas, experimental design and conclusions are the authors' own, and the authors reviewed all code and text and take full responsibility for the content.

ETHICS STATEMENT

XLGym builds tasks from two public sources: the Office.js API documentation (Microsoft, 2026b), which we scrape from Microsoft's public documentation pages and use only as context for generating tasks and training targets, and workbooks from the public Enron (Hermans & Murphy-Hill, 2015) and FUSE (Barik et al., 2015) spreadsheet corpora, which are widely used in spreadsheet research; we do not collect new data from people, and no human subjects were involved. The Enron workbooks originate from a real company and may contain names or business details; we use them only as released by their authors. The models we train automate spreadsheet editing, and, like any code-writing agent, may produce incorrect edits, so their output should be reviewed before use on important data.

REPRODUCIBILITY STATEMENT

We release the XLGym pipeline, XLBench, XLTrain and the XLSage models. Section 3 describes each pipeline stage, and Section 4 the benchmark construction and evaluation harness. Training data, procedure and hyperparameters are given in Section 5 and Table 6 (Appendix C); all models are evaluated with reasoning effort `none`, and all reported results are means over three runs.

REFERENCES

Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos Garea, Matthieu Geist, and Olivier Bachem. On-policy distillation of language models: Learning from self-generated mistakes. In *International Conference on Learning Representations (ICLR)*, 2024.

Anthropic. Claude for excel, 2026. URL `https://claude.com/claude-for-excel`.

Titus Barik, Kevin Lubick, Justin Smith, John Slankas, and Emerson Murphy-Hill. Fuse: A reproducible, extendable, internet-scale corpus of spreadsheets. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 486–489. IEEE, 2015.

Mangesh Bendre, Tana Wattanawaroon, Sajjadur Rahman, Kelly Mack, Yuyang Liu, Shichu Zhu, Yu Lu, Ping-Jing Yang, Xinyan Zhou, Kevin Chen-Chuan Chang, et al. Faster, higher, stronger: Redesigning spreadsheets for scale. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 1972–1975. IEEE, 2019.

Aleksander Boruch-Gruszecki et al. Agnostics: Learning to code in any programming language via reinforcement with a universal learning environment, 2025.

Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. Knowledge transfer from high-resource to low-resource programming languages for code LLMs. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2), 2024. doi: 10.1145/3689735.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

Yibin Chen, Yifu Yuan, Zeyu Zhang, Yan Zheng, Jinyi Liu, Fei Ni, Jianye Hao, Hangyu Mao, and Fuzheng Zhang. SheetAgent: Towards a generalist agent for spreadsheet reasoning and manipulation via large language models. In *Proceedings of the ACM on Web Conference 2025*, pp. 158–177. ACM, 2025. doi: 10.1145/3696410.3714962.

Junhang Cheng, Fang Liu, Jia Li, Chengru Wu, Nanxiang Jiang, and Li Zhang. CangjieBench: Benchmarking LLMs on a low-resource general-purpose programming language, 2026a.

Mingyue Cheng, Shuo Yu, Daoyu Wang, Qingchuan Li, Xiaoyu Tao, Qingyang Mao, Yitong Zhou, and Qi Liu. TabClaw: An interactive and self-evolving agent for spreadsheet manipulation and table reasoning, 2026b.

Banghao Chi, Yining Xie, Mingyuan Wu, Jingcheng Yang, Jize Jiang, Zhaoheng Li, Shengyi Qian, Minjia Zhang, Klara Nahrstedt, Rui Hou, Xiangjun Fan, and Hanchao Yu. Spreadsheet-RL: Advancing large language model agents on realistic spreadsheet tasks via reinforcement learning, 2026. URL https://arxiv.org/abs/2605.22642.

Haoyu Dong, Jianbo Zhao, Yuzhang Tian, Junyu Xiong, Shiyu Xia, Mengyu Zhou, Yun Lin, José Cambronero, Yeye He, Shi Han, et al. Spreadsheetllm: Encoding spreadsheets for large language models. *arXiv preprint arXiv:2407.09025*, 2024.

Haoyu Dong, Pengkun Zhang, Yan Gao, Xuanyu Dong, Yilin Cheng, Mingzhe Lu, Zikun Zhu, Adina Yakefu, and Shuxin Zheng. Finch: Benchmarking finance & accounting across spreadsheet-centric enterprise workflows, 2025.

Avik Dutta, Priyanshu Gupta, Hosein Hasanbeig, Rahul Pratap Singh, Harshit Nigam, Sumit Gulwani, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. Condabench: Interactive evaluation of language models for data analysis, 2025.

Eric Gazoni and Charlie Clark. openpyxl documentation: Pivot tables. openpyxl 3.1 documentation, 2024. URL https://openpyxl.readthedocs.io/en/stable/pivot.html.

Alessandro Giagnorio, Alberto Martin-Lopez, and Gabriele Bavota. No resource, no benchmarks, no problem? evaluating and improving LLMs for code generation in no-resource languages, 2026.

Anmol Gulati, Sahil Sen, Waqar Sarguroh, and Kevin Paul. Beyond rows to reasoning: Agentic retrieval for multimodal spreadsheet understanding and editing, 2026.

Panjing He, Mingyue Cheng, Yucong Luo, Li Li, and Xiaohan Zhang. SheetCompass: Hierarchical relation graphs for agentic spreadsheet reasoning, 2026.

Felienne Hermans and Emerson Murphy-Hill. Enron's spreadsheets and related emails: A dataset and analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pp. 7–16. IEEE, 2015.

Jonas Hübotter, Frederike Lübeck, Lejs Behric, Anton Baumann, Marco Bagatella, Daniel Marta, Ido Hakimi, Idan Shenfeld, Thomas Kleine Buening, Carlos Guestrin, and Andreas Krause. Reinforcement learning via self-distillation. *arXiv preprint arXiv:2601.20802*, 2026.

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 *International Conference on Learning Representations*, 2024.

Kilian Kier, Alessandro Giagnorio, Omar AbedelKader, Robert Peharz, Oleksandr Zaitsev, Stéphane Ducasse, Romain Robbes, and Gabriele Bavota. Teaching LLMs a low-resource language: Enhancing code completion in Pharo, 2026.

Srivatsa Kundurthy, Clara Na, Michael Handley, Zach Kirshner, Chen Bo Calvin Zhang, Manasi Sharma, Emma Strubell, and John Ling. SpreadsheetArena: Decomposing preference in LLM generation of spreadsheet workbooks. In *International Conference on Machine Learning (ICML)*, 2026a.

Srivatsa Kundurthy, Clara Na, Colton Moraine, Anoushka Mohta, Case Winter, George Fang, John Ling, Emma Strubell, and Zach Kirshner. BlueFin: Benchmarking LLM agents on financial spreadsheets, 2026b.

Elaine Lau, Markus Dücker, Ronak Chaudhary, et al. BankerToolBench: Evaluating AI agents in end-to-end investment banking workflows, 2026.

Hongxin Li, Jingran Su, Yuntao Chen, Qing Li, and Zhaoxiang Zhang. SheetCopilot: Bringing software productivity to the next level through large language models. In *Advances in Neural Information Processing Systems*, volume 36, pp. 4952–4984. Curran Associates, Inc., 2023.

Mofei Li, Taozhi Chen, Guowei Yang, and Jia Li. Learning from execution: Self-evolving memory for private-library code generation, 2026.

Jenny T. Liang, Aayush Kumar, Yasharth Bajpai, Sumit Gulwani, Vu Le, Chris Parnin, Arjun Radhakrishna, Ashish Tiwari, Emerson Murphy-Hill, and Gustavo Soares. TableTalk: Scaffolding spreadsheet development with a language agent. *ACM Transactions on Computer-Human Interaction*, 32(6), 2025. doi: 10.1145/3765286.

Jinnuo Liu, Yue Peng, Jinhan Niu, and Hongyi Wen. Diagnosing knowledge gaps in LLM tool use: An agentic benchmark for novel API acquisition, 2026a.

Mingwei Liu, Zheng Pei, Yanlin Wang, Zihao Wang, Zikang Li, Enci Lin, Xin Peng, and Zibin Zheng. Knowledge-graph-driven data synthesis for low-resource software development: A HarmonyOS case study. *Proceedings of the ACM on Software Engineering*, 3(FSE), 2026b. doi: 10.1145/3808133.

Zeyu Leo Liu, Shrey Pandit, Xi Ye, Eunsol Choi, and Greg Durrett. CodeUpdateArena: Benchmarking knowledge editing on API updates, 2024.

Tengchao Lv, Dongdong Zhang, Jiayu Ding, Yilin Jia, Yuzhong Zhao, Yupan Huang, Wenshan Wu, Xiangyang Zhou, Shaohan Huang, Nan Yang, Li Dong, Lei Cui, and Furu Wei. Mind the gap: Can frontier LLMs pass a standardized office proficiency exam?, 2026.

Zeyao Ma, Bohan Zhang, Jing Zhang, Jifan Yu, Xiaokang Zhang, Xiaohan Zhang, Sijia Luo, Xi Wang, and Jie Tang. SpreadsheetBench: Towards challenging real world spreadsheet manipulation. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, volume 37, 2024.

Fanzhe Meng, Guoxin Chen, Jiale Zhao, Shuang Sun, Zhiyu Lin, Wayne Xin Zhao, Ruihua Song, Ji-Rong Wen, and Kai Jia. CalibForge: Adversarial solver calibration for scaling learnable terminal tasks, 2026.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026.

Microsoft. Agent mode in excel is now generally available on desktop, 2025. URL https://techcommunity.microsoft.com/blog/excelblog/agent-mode-in-excel-is-now-generally-available-on-desktop/4457408.

Microsoft. Overview of Copilot skills for Excel (preview). Microsoft Learn, 2026a. URL https://learn.microsoft.com/en-us/office/dev/add-ins/excel/excel-skills.

Microsoft. Excel JavaScript API overview. Microsoft Learn, 2026b. URL https://learn.microsoft.com/en-us/office/dev/add-ins/reference/overview/excel-add-ins-reference-overview.

Microsoft Excel Team. Building agent mode in Excel. Microsoft Excel Blog, 2025. URL https://techcommunity.microsoft.com/blog/excelblog/building-agent-mode-in-excel/4457320.

OpenAI. Chatgpt for spreadsheets, 2026. URL https://chatgpt.com/apps/spreadsheets.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive APIs, 2023.

Justin Payan, Swaroop Mishra, Mukul Singh, Carina Negreanu, Christian Poelitz, Chitta Baral, Subhro Roy, Rasika Chakravarthy, Benjamin Van Durme, and Elnaz Nouri. InstructExcel: A benchmark for natural language instruction in Excel. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs, 2023.

Houxing Ren, Zimu Lu, Ke Wang, Yunqiao Yang, Mingjie Zhan, Haotian Hou, and Hongsheng Li. Towards robust real-world spreadsheet understanding with multi-agent multi-format reasoning. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2026.

Didula Samaraweera, Anjana Supun, and Srinath Perera. Selective left-shift: Turning test-time compute and difficulty-based curation into training data for low-resource code generation, 2026.

Idan Shenfeld, Jyothish Pari, and Pulkit Agrawal. RL's razor: Why online reinforcement learning forgets less. *arXiv preprint arXiv:2509.04259*, 2025.

Idan Shenfeld, Mehul Damani, Jonas Hübotter, and Pulkit Agrawal. Self-distillation enables continual learning. *arXiv preprint arXiv:2601.19897*, 2026.

Alex Tamkin and Peter McCrory. Estimating AI productivity gains from Claude conversations. Anthropic Research, 2025. URL https://www.anthropic.com/research/estimating-productivity-gains.

Mansi Uniyal, Agamdeep Singh, Ananya Singha, Priyanshu Gupta, Mukul Singh, Gust Verbruggen, Vu Le, and Sumit Gulwani. Back to the future: A workbook time machine for spreadsheet creation benchmarks. In *Third Conference on Language Modeling*, 2026. URL https://arxiv.org/abs/2608.07873.

Weixuan Wang, Dongge Han, Daniel Madrigal Diaz, Jin Xu, Victor Rühle, and Saravan Rajmohan. OdysseyBench: Evaluating LLM agents on long-horizon complex office application workflows, 2025a.

Zilong Wang et al. OfficeBench: Benchmarking language agents across multiple applications for office automation, 2024.

Ziwei Wang, Jiayuan Su, Mengyu Zhou, Huaxing Zeng, Mengni Jia, Xiao Lv, Haoyu Dong, Xiaojun Ma, Shi Han, and Dongmei Zhang. SheetBrain: A neuro-symbolic agent for accurate reasoning over complex and large spreadsheets, 2025b.

Sandya Wijaya, Jacob Bolano, Alejandro Gomez Soteres, Shriyanshu Kode, Yue Huang, and Anant Sahai. ReadMe.LLM: A framework to help LLMs understand your library, 2025.

Haoze Wu et al. ReCode: Updating code API knowledge with reinforcement learning, 2025.

Tongtong Wu et al. VersiCode: Towards version-controllable code generation, 2024.

Zhaofeng Wu, Shiqi Wang, Emma Peng, Anuj Goyal, Melanie Kambadur, Sebastian Ruder, Yoon Kim, and Chloe Bi. Parallel-SFT: Improving zero-shot cross-programming-language transfer for code RL. In *Findings of the Association for Computational Linguistics: ACL 2026*, 2026.

Zhaorui Yang, Tianyu Pang, Haozhe Feng, Han Wang, Wei Chen, Minfeng Zhu, and Qian Liu. Self-distillation bridges distribution gap in language model fine-tuning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024. arXiv:2402.13669.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.

Thomson Yen, Julian Poeltl, Harshith Srinivas Gear, Yilin Meng, Joshua Fan, Adam Shen, Yili Liu, Ali Bauyrzhan, Siri Du, Haoyang Liu, Daniel Guetta, and Hongseok Namkoong. MBABench: Evaluating LLM agents on end-to-end spreadsheet tasks in finance, 2026.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. STaR: Bootstrapping reasoning with reasoning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Xiaokang Zhang et al. TableLLM: Enabling tabular data manipulation by LLMs in real office usage scenarios, 2024.

Yitong Zhang, Chengze Li, Ruize Chen, Guowei Yang, Xiaoran Jia, Yijie Ren, and Jia Li. To see is not to master: Teaching LLMs to use private libraries for code generation, 2026.

Siyan Zhao, Zhihui Xie, Mengchen Liu, Jing Huang, Guan Pang, Feiyu Chen, and Aditya Grover. Self-distilled reasoner: On-policy self-distillation for large language models. *arXiv preprint arXiv:2601.18734*, 2026.

Shuyan Zhou, Uri Alon, Frank F. Xu, Zhiruo Wang, Zhengbao Jiang, and Graham Neubig. DocPrompting: Generating code by retrieving the docs. In *International Conference on Learning Representations*, 2023.

Jian Zhu, Yuzheng Zhang, Zeyao Ma, Bohan Zhang, Armin Schoepf, Daniel Woloch, Peter Yiliu Wang, Guangyu Robert Yang, Samuel Jacob, Siddharth Nagisetty, Abhiram Chundru, Jean Lin, Spencer Mateega, and Jing Zhang. SpreadsheetBench 2: Evaluating agents on end-to-end business spreadsheet workflows, 2026.

Ruiyan Zhu, Xi Cheng, Ke Liu, Brian Zhu, Daniel Jin, Neeraj Parihar, Zhoutian Xu, and Oliver Gao. SheetMind: An end-to-end LLM-powered multi-agent framework for spreadsheet automation, 2025.

A Evaluation Harnesses

Both harnesses give the model the query and a preview of the workbook (the summary of Section 4: sheets, artifacts, data ranges and a truncated preview of cell values).

Single-step. The model answers in one turn with a single Office.js program. We open a fresh copy of the workbook in Excel, run the program inside `Excel.run`, and export the saved workbook and console output for grading. There is no execution feedback, repair or second turn; infrastructure failures are retried and do not count as model attempts.

Multi-step. A ReAct-style loop (Yao et al., 2023): the model calls an `execute_officejs` tool on a live workbook, sees the console output or error, and may inspect, verify and repair for up to eight turns before it stops. We grade the workbook after its last successful call.

Python. For the comparison in Table 3, the model instead writes openpyxl code that modifies the workbook. The modified workbook is saved and recalculated in Excel before grading.

In every harness the solution verifier of Section 3.1 grades the saved workbook and output separately from execution success, and its verdict never enters the solving loop.

B Python vs. Office.js and Harness Comparison

Table 3 compares Python and Office.js in the multi-step harness (Section 1). Table 4 compares the single-step harness used throughout the paper with the multi-step harness. As in the main paper, all models run with reasoning effort `none`.

Table 3: Pass rate (%) with Python vs. Office.js in the multi-step harness. Bold marks the better language for each model and benchmark. Means over three runs.

	SpreadsheetBench		WTM-Bench		Spreadsheet-RL	
Model	Python	Office.js	Python	Office.js	Python	Office.js
GPT-5.6	57.0	**68.5**	65.6	**72.7**	62.0	**66.3**
Opus 5	74.4	**82.8**	68.4	**80.2**	**71.2**	69.5

Table 4: Office.js pass rate (%) in the single-step and multi-step harnesses. Multi-step scores are 11–22 points higher, but the two models rank the same in both harnesses on every benchmark. Means over three runs.

	SpreadsheetBench		WTM-Bench		Spreadsheet-RL	
Model	Single	Multi	Single	Multi	Single	Multi
Opus 5	71.4	82.8	60.2	80.2	55.8	69.5
GPT-5.6	57.9	68.5	50.4	72.7	45.7	66.3

Table 5 breaks down execution errors in the multi-step harness. Office.js trajectories hit an execution error two to three times as often as Python trajectories for both models, yet end with higher pass rates (Table 3): models recover from many of these errors when they see the error message. The single-step harness removes this recovery, and so measures what a model knows about the API before any feedback.

Table 5: Execution errors in the multi-step harness. Error: share of trajectories in which at least one program fails to run. Steps: mean number of programs per trajectory. Recovery: share of multi-step trajectories that recover from an error and then pass. Means over three runs.

		Python			Office.js		
Benchmark	Model	Error (%)	Steps	Recovery (%)	Error (%)	Steps	Recovery (%)
SpreadsheetBench	GPT-5.6	5.7	1.10	61.3	17.0	1.41	59.4
	Opus 5	3.0	2.72	3.6	7.3	2.26	8.6
WTM-Bench	GPT-5.6	11.1	1.55	33.1	32.7	1.98	58.1
	Opus 5	5.3	2.47	7.2	15.8	2.57	18.1

C Additional Training Results

Table 6 lists the fine-tuning hyperparameters. Table 7 reports every training target at every corpus size for Qwen3.6-27B (Section 5.3), in- and out-of-domain, for completeness. Figure 8 evaluates Qwen3.6-27B and XLSage-27B in both harnesses of Appendix A (Section 5.1).

Table 6: Supervised fine-tuning hyperparameters for XLSage-27B (base: Qwen3.6-27B).

Hyperparameter	Value
Fine-tuning	Full parameter (no LoRA)
Epochs	2
Validation split	5% of the training corpus, split by workbook
Global batch size	32 sequences (1 per GPU $\times$ 4 accumulation $\times$ 8 GPUs)
Peak learning rate	1×10^{-5}
Learning-rate schedule	Cosine decay to 0, 3% linear warmup
Optimizer	AdamW (β_1=0.9, β_2=0.999, ϵ=10^{-8}), CPU offload
Weight decay	0
Gradient clipping	1.0 (max norm)
Maximum sequence length	10,240 tokens (prompt + completion)
Hardware	8 A100 GPUs

Table 7: Effect of training data source on Qwen3.6-27B. Each cell reports pass rate / execution rate, both in % and averaged over three runs. Execution rate is the fraction of evaluated attempts whose code executed successfully, regardless of task correctness. *Reference*: training on the XLGym reference solutions. *Teacher*: the same model acts as a teacher, receiving the query plus API documentation and reference code as privileged information, and writing its own solution. The better pass rate within each corpus size on each benchmark is in bold.

Source corpus	Training data	XLBench Pass / Exec.	WTM-Bench Pass / Exec.	SpreadsheetBench Pass / Exec.
2.5k	Reference	8.95 / 34.04	15.78 / 57.78	11.65 / 49.96
	Teacher	**11.58** / 26.14	**21.11** / 52.89	**16.54** / 44.73
10k	Reference	21.75 / 37.89	24.89 / 64.22	10.38 / 45.99
	Teacher	**22.63** / 37.54	**31.11** / 65.78	**18.40** / 51.39
20.6k	Reference	30.88 / 55.61	21.56 / 50.89	7.00 / 35.53
	Teacher	**31.93** / 52.28	**31.56** / 63.11	**22.19** / 57.64

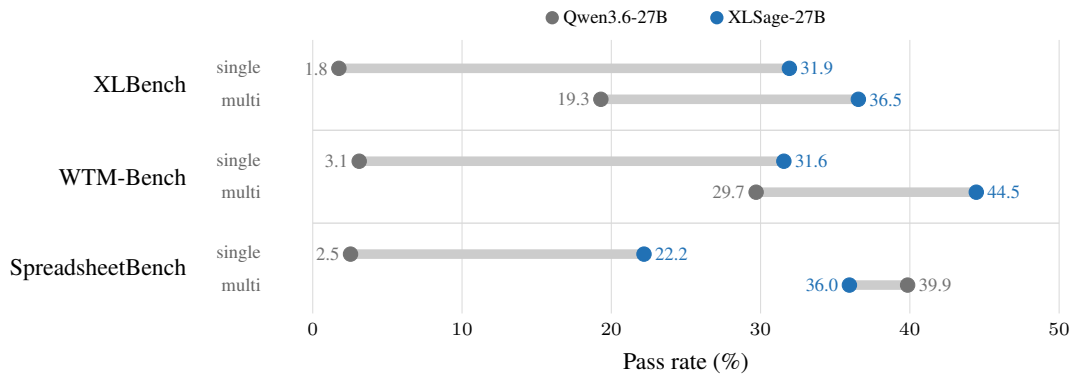


Figure 8: Pass rate of Qwen3.6-27B (grey) and XLSage-27B (blue), i.e. before and after fine-tuning on the 20.6k teacher corpus, in the single-step and multi-step harnesses. On XLBench and WTM-Bench, training on single programs improves both harnesses.

D PYTHON VS. OFFICE.JS: COMPLETE CODE EXAMPLES

Figures 9–11 give the complete submitted code behind the three panels of Figure 2: a native PivotTable (Figure 9), a shorter chart program (Figure 10) and a hallucinated API member (Figure 11). Each pair uses the same task, workbook and repeat, run with GPT-5.6 at reasoning effort none in the multi-step harness. Pass/fail labels are the recorded grader verdicts. Code is shown verbatim except for line wrapping and a few blocks marked as elided.

(a.1) Native artifacts: Wage Pivot

✗FAIL ✓PASS

Task: *"Create a new sheet named "Wage Pivot" with a PivotTable that groups Chief executives, General and operations managers, and Legislators by their median annual wage quartile and sums their 2008 employment. Include the grand total."*

XLBench task 3654, repeat 1 · GPT-5.6 (reasoning effort none)

Python (openpyxl)77 lines · 3,580 chars · no PivotTable in saved file ✗FA

```python
def modify(wb, wb_values):
    from openpyxl.styles import Font, PatternFill, Border, Side,
        Alignment
    from openpyxl.worksheet.table import Table, TableStyleInfo

    src = wb['Table 1.6']
    targets = {'Chief executives', 'General and operations
        managers', 'Legislators'}
    records = []
    for r in range(1, src.max_row + 1):
        occupation = src.cell(r, 2).value
        if occupation in targets:
            employment = src.cell(r, 3).value
            quartile = src.cell(r, 10).value
            records.append((occupation, quartile, employment, r))
            print('Found:', occupation, 'quartile=', quartile,
                'employment=', employment, 'row=',
                r)

    if len(records) != 3:
        raise ValueError(f'Expected 3 target occupations, found
            {len(records)}: {records}')

    if 'Wage Pivot' in wb.sheetnames:
        del wb['Wage Pivot']
    ws = wb.create_sheet('Wage Pivot')

    # Build a compact pivot-style report. Openpyxl cannot create
        a native Excel PivotTable cache,
    # so this sheet materializes the requested grouped
        aggregation and grand total.
    ws['A1'] = 'Median annual wage quartile'
    ws['B1'] = 'Sum of 2008 employment'

    quartile_sums = {}
    for occupation, quartile, employment, _ in records:
        quartile_sums[quartile] = quartile_sums.get(quartile, 0)
            + float(employment)

    order = {'VL': 1, 'L': 2, 'M': 3, 'H': 4, 'VH': 5}
    sorted_items = sorted(quartile_sums.items(), key=lambda x: (
        order.get(x[0], 99), str(x[0])))
    row = 2
    for quartile, total in sorted_items:
        ws.cell(row, 1, quartile)
        ws.cell(row, 2, total)
        row += 1
    ws.cell(row, 1, 'Grand Total')
    ws.cell(row, 2, f'=SUM(B2:B{row-1})')

    # Add supporting source records to the right for transparency.
    ws['D1'] = 'Pivot source records'
    ws['D2'] = 'Occupation'
    ws['E2'] = 'Median annual wage quartile'
    ws['F2'] = '2008 employment'
    for i, (occupation, quartile, employment, _) in enumerate(
        records, start=3):
        ws.cell(i, 4, occupation)
        ws.cell(i, 5, quartile)
        ws.cell(i, 6, employment)
    tab = Table(displayName='WagePivotSource', ref=f'D2:F{2+len(
        records)}')
    tab.tableStyleInfo = TableStyleInfo(name='TableStyleMedium2',
        showRowStripes=True, showColumnStripes=False)
    ws.add_table(tab)

· · · cell styling: fills, fonts, borders, widths, number formats (26 lines elided) · · ·

    print('Pivot groups:', sorted_items)
    print('Grand total formula:', ws.cell(row, 2).value)
    print('Expected grand total:', sum(v for _, v in
        sorted_items))
    print('Created sheet:', ws.title)
    return wb
```

Office.js 48 lines · 2,370 chars · execution 2 of 2 ✓PASS

Execution 1 (53 lines, not shown) failed with *"The number of rows or columns in the input array doesn't match the size or dimensions of the range."* after writing one value into the merged range `A1:C1`. Execution 2 writes the title to `A1` and keeps the same PivotTable construction.

```javascript
await Excel.run(async (context) => {
  const workbook = context.workbook;
  const sourceSheet = workbook.worksheets.getItem("Table 1.6");
  const usedRange = sourceSheet.getUsedRange();
  usedRange.load("values");
  const sheets = workbook.worksheets;
  sheets.load("items/name");
  await context.sync();

  const wanted = ["Chief executives", "General and operations
      managers", "Legislators"];
  const wantedSet = new Set(wanted);
  const found = new Map();
  for (const row of usedRange.values) {
    const occupation = String(row[1] ?? "").trim();
    if (wantedSet.has(occupation)) {
      const rawEmployment = row[2];
      const employment = typeof rawEmployment === "number" ?
          rawEmployment : Number(String(rawEmployment).
          replace(/,/g, ""));
      const quartile = String(row[9] ?? "").trim();
      found.set(occupation, [occupation, quartile, employment]);
    }
  }
  if (found.size !== 3) throw new Error(`Expected 3 matching
      occupations, but found ${found.size}.`);
  const rows = wanted.map(name => found.get(name));

  const oldSheet = sheets.items.find(s => s.name === "Wage
      Pivot");
  if (oldSheet) {
    oldSheet.delete();
    await context.sync();
  }

  const targetSheet = workbook.worksheets.add("Wage Pivot");
  targetSheet.getRange("A1").values = [["2008 Employment by
      Median Annual Wage Quartile"]];
  targetSheet.getRange("A1").format.font.bold = true;
  targetSheet.getRange("A1").format.font.size = 14;

  const data = [["Occupation", "Median annual wage quartile",
      "2008 employment"], ...rows];
  const sourceRange = targetSheet.getRangeByIndexes(2, 0, data.
      length, 3);
  sourceRange.values = data;
  const sourceTable = targetSheet.tables.add(sourceRange, true);
  sourceTable.name = "WagePivotSource";

  const pivot = targetSheet.pivotTables.add("WagePivotTable",
      sourceTable, "E3");
  pivot.rowHierarchies.add(pivot.hierarchies.getItem("Median
      annual wage quartile"));
  const dataHierarchy = pivot.dataHierarchies.add(pivot.
      hierarchies.getItem("2008 employment"));
  dataHierarchy.summarizeBy = Excel.AggregationFunction.sum;
  pivot.layout.layoutType = Excel.PivotLayoutType.tabular;
  pivot.layout.showRowGrandTotals = true;
  pivot.layout.showColumnGrandTotals = true;

  targetSheet.getRange("A:C").format.autofitColumns();
  targetSheet.getRange("E:H").format.autofitColumns();
  targetSheet.activate();
  await context.sync();
});
```

Python grader: *"Although the aggregated values and grand total are correct, the agent created only a static pivot-style summary and explicitly did not create a native Excel PivotTable. The requested PivotTable structure is absent from the final workbook, so the task was not fully completed.*

Office.js grader: *"The workbook contains a new "Wage Pivot" sheet with an actual PivotTable grouping the requested occupations by wage quartile and producing VH = 2133.5, VL = 67.6, and Grand Total = 2201.1. Its placement at E3:F6 and inclusion of source data on the same sheet are valid alternatives to the gold layout.*

Figure 9: Complete submitted code for Figure 2(a). Both agents compute the same totals (grey); amber: the Python agent's stated reason for a static summary; green: native PivotTable construction. openpyxl cannot create native PivotTables (Gazoni & Clark, 2024), so the Python agent can only write a static summary.

(b.1) Concision: smooth scatter chart

✓PASS ✓PASS

Task: *"Make a smooth scatter chart of the CFD x/c vs y/c data."*

WTM-Bench task 98, repeat 2 · GPT-5.6 (reasoning effort none)

Python (openpyxl) — 46 lines · 1,852 chars ✓PASS

```python
def modify(wb, wb_values):
    from openpyxl.chart import ScatterChart, Series, Reference
    from openpyxl.chart.label import DataLabelList

    ws = wb['Sheet1']

    # Find the last row containing both CFD x/c and y/c values.
    last_row = 2
    for r in range(3, ws.max_row + 1):
        if ws.cell(r, 1).value is not None and ws.cell(r, 2).
            value is not None:
            last_row = r

    # Remove an existing chart with the same title to avoid
        duplicates on reruns.
    retained = []
    for chart in ws._charts:
        title_text = ''
        try:
            title_text = chart.title.tx.rich.p[0].r[0].t
        except Exception:
            pass
        if title_text != 'CFD x/c vs y/c':
            retained.append(chart)
    ws._charts = retained

    chart = ScatterChart()
    chart.title = 'CFD x/c vs y/c'
    chart.style = 13
    chart.height = 12
    chart.width = 20
    chart.scatterStyle = 'smooth'
    chart.x_axis.title = 'x/c [⌄]'
    chart.y_axis.title = 'y/c [⌄]'
    chart.legend = None

    xvalues = Reference(ws, min_col=1, min_row=3,
        max_row=last_row)
    yvalues = Reference(ws, min_col=2, min_row=3,
        max_row=last_row)
    series = Series(yvalues, xvalues, title='CFD')
    series.graphicalProperties.line.solidFill = '4472C4'
    series.graphicalProperties.line.width = 24000
    series.graphicalProperties.noFill = True
    series.marker.symbol = 'none'
    series.smooth = True
    chart.series.append(series)

    chart.x_axis.scaling.min = 0
    chart.x_axis.scaling.max = 1
    chart.x_axis.majorGridlines = None

    ws.add_chart(chart, 'G2')

    print(f'Created smooth scatter chart from A3:A{last_row} vs
        B3:B{last_row}.')
    print(f'First point: ({wb_values["Sheet1"]["A3"].value},
        {wb_values["Sheet1"]["B3"].value})')
    print(f'Chart title: CFD x/c vs y/c; chart count: {len(ws.
        _charts)}')
    return wb
```

Office.js — 18 lines · 688 chars ✓PASS

```javascript
await Excel.run(async (context) => {
  const sheet = context.workbook.worksheets.getItem("Sheet1");
  const dataRange = sheet.getRange("A3:B803");

  const charts = sheet.charts;
  const chart = charts.add(Excel.ChartType.
      xyscatterSmoothNoMarkers, dataRange, Excel.
      ChartSeriesBy.columns);
  chart.name = "CFD_xc_vs_yc";
  chart.title.text = "CFD x/c vs y/c";
  chart.title.visible = true;
  chart.legend.visible = false;
  chart.setPosition("G2", "N20");

  const xAxis = chart.axes.categoryAxis;
  xAxis.title.text = "x/c [⌄]";
  xAxis.title.visible = true;

  const yAxis = chart.axes.valueAxis;
  yAxis.title.text = "y/c [⌄]";
  yAxis.title.visible = true;

  await context.sync();
});
```

Figure 10: Complete submitted code for Figure 2(b). Amber: explicit reference and series construction in openpyxl; green: the native chart type that replaces it in Office.js; grey: optional last-row discovery, rerun de-duplication and diagnostics, which also count towards the Python length.

(c.1) Hallucination: line chart of four price series

✓ PASS ✗ FAIL

Task: *"Create a line chart of the four price columns by DATE: Firm On Peak, Firm Off Peak, Non Firm On Peak, and Non Firm Off Peak."*

WTM-Bench task 58, repeat 3 · GPT-5.6 (reasoning effort none)

Python (openpyxl) — 52 lines · 2,201 chars — ✓ PASS

```python
def modify(wb, wb_values):
    from openpyxl.chart import LineChart, Reference
    from openpyxl.chart.axis import DateAxis
    from openpyxl.chart.label import DataLabelList
    from openpyxl.chart.marker import DataPoint
    from openpyxl.chart.series_factory import SeriesFactory

    ws = wb["MeadMktplace"]

    # Remove any existing charts to avoid duplicates on reruns.
    ws._charts = []

    # Build a line chart using the DATE column and the four price columns.
    chart = LineChart()
    chart.title = "Electricity Prices by Date"
    chart.style = 13
    chart.y_axis.title = "Price"
    chart.x_axis.title = "DATE"
    chart.height = 10
    chart.width = 19
    chart.legend.position = "r"

    dates = Reference(ws, min_col=2, min_row=12, max_row=42)
    series_specs = [
        (4, "Firm On Peak"),
        (6, "Firm Off Peak"),
        (8, "Non Firm On Peak"),
        (10, "Non Firm Off Peak"),
    ]

    colors = ["4472C4", "ED7D31", "70AD47", "A5A5A5"]
    for (col, title), color in zip(series_specs, colors):
        values = Reference(ws, min_col=col, min_row=12, max_row=42)
        series = SeriesFactory(values, title=title)
        series.graphicalProperties.line.solidFill = color
        series.graphicalProperties.line.width = 24000
        series.marker.symbol = "circle"
        series.marker.size = 5
        series.graphicalProperties.noFill = False
        chart.series.append(series)

    chart.set_categories(dates)
    # Display dates in chronological order; source data is reverse chronological.
    chart.x_axis.scaling.orientation = "maxMin"
    chart.x_axis.number_format = "mmm d"
    chart.x_axis.tickLblPos = "low"
    chart.y_axis.numFmt = '$0.00'

    ws.add_chart(chart, "M9")

    print("Created line chart on sheet:", ws.title)
    print("Chart anchor: M9")
    print("Categories: B12:B42")
    print("Series:", [title for _, title in series_specs])
    print("Chart count:", len(ws._charts))
    for row in [12, 13, 42]:
        print("Row", row, "date/formula value:", ws.cell(row, 2).value,
              "cached:", wb_values[ws.title].cell(row, 2).value,
              "prices:", [ws.cell(row, c).value for c, _ in series_specs])

    return wb
```

Office.js — 32 lines · 1,300 chars · no runtime error — ✗ FAIL

```javascript
await Excel.run(async (context) => {
  const sheet = context.workbook.worksheets.getItem("MeadMktplace");
  const charts = sheet.charts;
  charts.load("items/name");
  await context.sync();

  const chartName = "PriceByDate";
  const existing = charts.items.find(c => c.name === chartName);
  if (existing) {
    existing.delete();
    await context.sync();
  }

  const chart = charts.add(Excel.ChartType.line, sheet.getRange("B12:B42"), Excel.ChartSeriesBy.columns);
  chart.name = chartName;
  chart.title.text = "Prices by Date";
  chart.hasLegend = true;
  chart.legend.position = Excel.ChartLegendPosition.bottom;

  const seriesInfo = [
    { name: "Firm On Peak", range: "D12:D42" },
    { name: "Firm Off Peak", range: "F12:F42" },
    { name: "Non Firm On Peak", range: "H12:H42" },
    { name: "Non Firm Off Peak", range: "J12:J42" }
  ];

  for (const info of seriesInfo) {
    const series = chart.series.add(info.name);
    series.categoryFormula = `='MeadMktplace'!$B$12:$B$42`;
    series.formula = `='MeadMktplace'!$${info.range.split(":")[0].replace(/[0-9]/g, "")}$12:$${info.range.split(":")[1].replace(/[0-9]/g, "")}$42`;
  }

  chart.axes.categoryAxis.title.text = "DATE";
  chart.axes.valueAxis.title.text = "Price";
  chart.setPosition("M9", "V28");
  await context.sync();
});
```

Python grader: *"The candidate contains a line chart with DATE categories and exactly the four requested price series, each linked to the correct source range. No source cells were changed.*

Office.js grader: *"The workbook contains a line chart with the requested labels, but its series bindings are incorrect: DATE is plotted as an extra value series, while the four requested price series have no data or category formulas.*

Figure 11: Complete submitted code for Figure 2(c). Red: `ChartSeries.categoryFormula` and `ChartSeries.formula` do not exist (the real methods are `setValues(range)` and `setXAxisValues(range)`); JavaScript accepts the assignments silently, so the call reports success. Amber: the chart is initialised from the DATE column, which becomes an extra value series. The saved `chart1.xml` has four named series with no category or value references.