iGent AI VIBECODEBENCH: SONNET 4.0

BENCHMARKS COMPILED 05/2025

Maxime Robeyns
maxime@igent.ai

Summary: We look at LLM code generation in the context of entirely autonomous, mid-scale (<10,000 LOC) codebase generation, with progressively revealed feature specifications. We focus not just on code functionality, but also notions of code maintainability, marginal feature implementation cost and user deception. We compare Sonnet 4.0 with Sonnet 3.7 and 3.5 v2; GPT 4o and 4.1; and Gemini 2.5 Flash and 2.5 Pro. Sonnet 4.0 achieves strong results in many categories in this eval, yet does not clearly surpass 3.7 or Gemini 2.5 Pro. The marginal feature implementation cost (measured in tokens) is not statistically different between the best models, after controlling for implementation success. Gemini 2.5 Pro (05-06) is able to successfully complete a comparable number of features as 3.7 and Sonnet 4.0, although at higher USD cost (assuming Sonnet 4.0 is priced no greater than 3.7). Sonnet 4.0 scores slightly higher than Sonnet 3.7 and 3.5 in our measures of code quality, and walks back on Sonnet 3.7's verbosity. We find Sonnet 4.0 slightly improved in comparison to 3.5 and 3.7 on measures of '*goalpost moving*' (i.e. making a task appear complete when it is in fact not).

1 Introduction and Setup

VibeCodeBench contains several programming mini-projects, each with 10 sequential feature specifications which are revealed to the agent one at a time.

This tests not just whether the LLM can implement functional code, but whether the code is maintainable over long horizons and can be adapted to changing requirements, while remaining easy to work with. The agent is entirely un-directed beyond the progressively disclosed feature specifications, so decisions relating to code structure and organisation are entirely down to the underlying LLM and its biases. The features and problem sets are intentionally designed so that poor code decisions made early on lead to later features becoming increasingly difficult to implement cleanly, frustrating progress.

More details of the problems in the dataset and representative feature progressions are listed in Appendix A.

Each model is tested using an identical agent framework - sharing prompts and tool sets. The test agent scaffold is a relatively minimal agent loop, with the only notable features being sub-agents invoked as tools to manage context window length and an oversight mechanism to catch errors and prevent pathological behaviours such as repeating failed approaches. High thinking budget reasoning models are exposed through tool calls.

The only scaffolding variation is in the tool calling mechanism; we use an un-structured tool calling mechanism for Sonnet 3.5, 3,7 and Sonnet 4.0 as well as Gemini 2.5 Pro. Weaker models using constrained decoding from their respective APIs are Gemini 2.5 Flash, and GPT 4o and 4.1.

1.1 Interpreting the Results

To reduce variance and gain more insight from the results, we sample agent trajectories $n = 5$ times for each model, problem pair. Unless stated otherwise, for each metric we report the mean $\pm$ 2 standard errors. Most results are presented as graphs over feature iterations: generally the feature number can be taken as a proxy for the complexity of the sub-task at hand, with later features generally being more sophisticated, and building upon a larger existing codebase where poor previous design decisions come to bear.

Also note that while the test agent framework is kept identical across runs and models (with the exception of the tool calling interface), as model behaviours diverge it isn't necessarily ideal to benchmark models with an identical scaffold. The scaffold was primarily developed using Sonnet and Gemini models for testing, and as such may offer a slight bias against the OpenAI models.

2 Marginal Cost Per Feature

We start by reporting the tokens used by each agent for each new feature request in Figure 1. We stress that this is not the token cost *to implement* each feature, since all models fell short of a full implementation of each feature. The Gemini models use the most tokens per feature by a large margin, with Gemini 2.5 Flash being a particularly strong outlier, explained by strong model agency paired with ineffective use of tools leading to many tokens being spent fixing indentation issues and the like. An optimal scaffold for these cheap models may simply do away with sophisticated tools and overwrite each file. On the other end, the GPT models spend a low number of tokens per feature since they only implement each feature partly, lacking the agency to complete each when working autonomously.

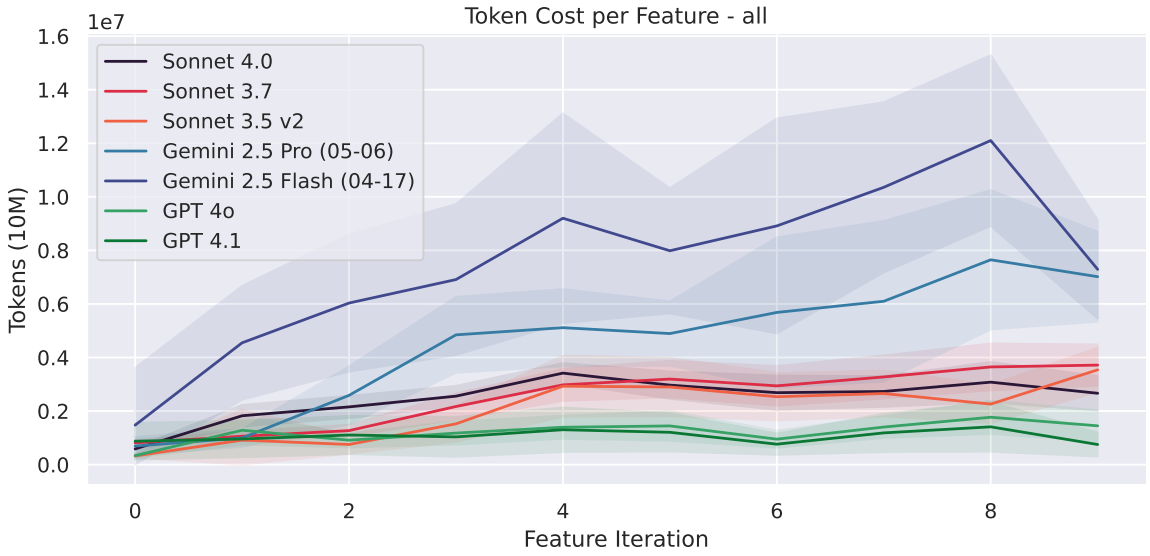

Figure 1: Total tokens spent on each feature, irrespective of feature implementation quality or completeness.

Stripping away data points where the feature implementation scored an average of below 50% on the held-out test set, we see the result in Figure 2:

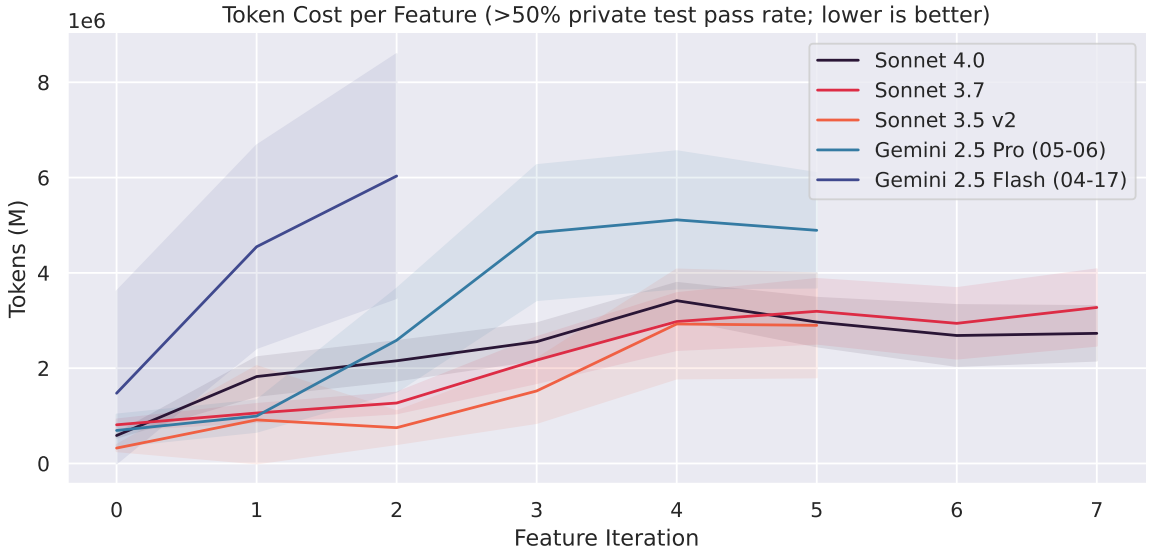

Figure 2: Tokens spent implementing each feature to at least 50% pass rate on the held-out test set.

We note that Sonnet 4.0 spends more tokens than Sonnet 3.5 and 3.7 on early features, reflecting an increased propensity to refactor. This may be what allows it to spend fewer tokens during later iterations, undercutting Sonnet 3.7 by about 500k tokens on average while achieving comparable test outcomes.

3 Code Correctness

For each feature in each benchmark problem, we create a comprehensive test suite that rigorously tests the public API surface defined in each feature specification. This runs through both the happy path as well as many edge cases, using fuzz-testing to catch defects. It also tests for regressions from previous feature iterations, if the new feature is not intentionally breaking. Crucially, these tests are hidden from the agent, which is left to write its own unit tests to verify its implementation of the feature specification before completing.

In Figure 3 we show the average pass rate of the rigorous held-out test set after each feature iteration. While the sample size of only 20 agent runs per line in the graph does not afford us much statistical power, it seems that Gemini 2.5 Pro is strongest during the simpler early iterations, closely followed by Sonnet 3.7 and then Sonnet 4.0.

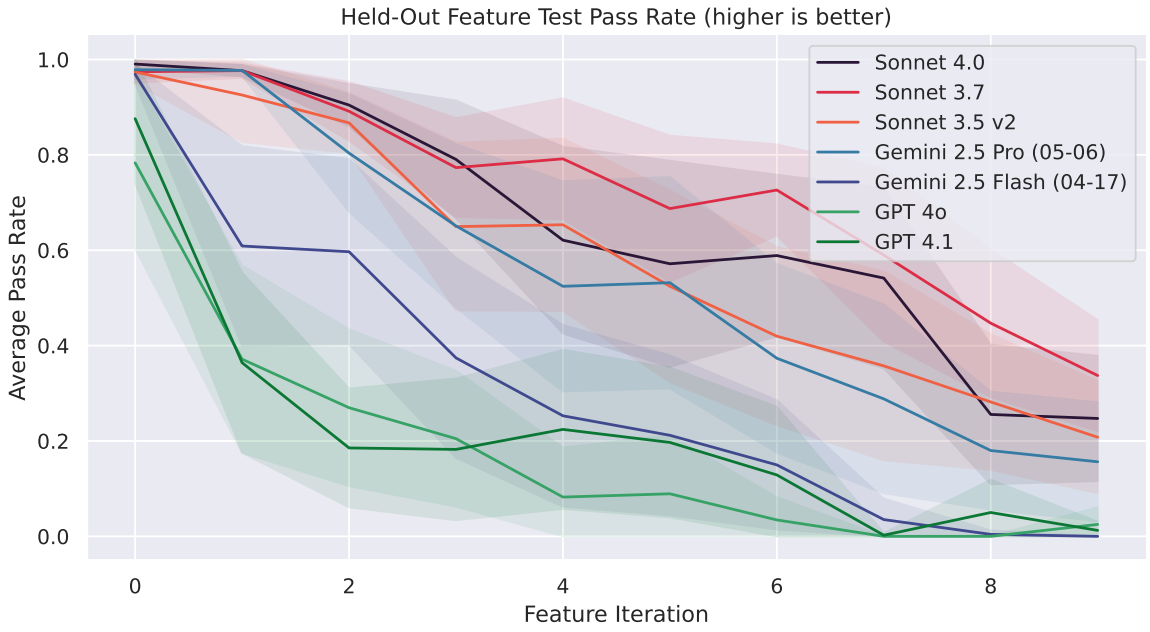

Figure 3: Test pass rate of the rigorous, held-out test set. Measures feature implementation quality.

We also show the agent's pass rate on its own self-written tests for each feature in Figure 4. While these are generally slightly higher than the held-out test pass rate, it is surprising that they are not closer to 1.0. This might be seen as a proxy for 'perseverance, since failing tests here indicate the agent concluded the run without having fixed its own failing test cases.

4 Goalpost-Moving

We've recently seen an increase in agent claiming "*The feature has now been implemented successfully!*" only for closer inspection to reveal that some terminal output or other execution feedback being disregarded, hallucinated (depending on the scaffold), or outright fabricated by writing a shell script with the test outputs the agent wants to see among other devious schemes. This may be an artifact of RL post-training where the model learns that it is more important to *show* that a feature is complete than to actually complete it.

We term this *goalpost moving* and attempt to quantify it here. Upon finishing a feature implementation request, each agent is asked to report the degree to which it estimates the feature is complete, as a number between 0 and 1. It may use its self-authored tests to help it come up with this number, or any other information or heuristics it has available.

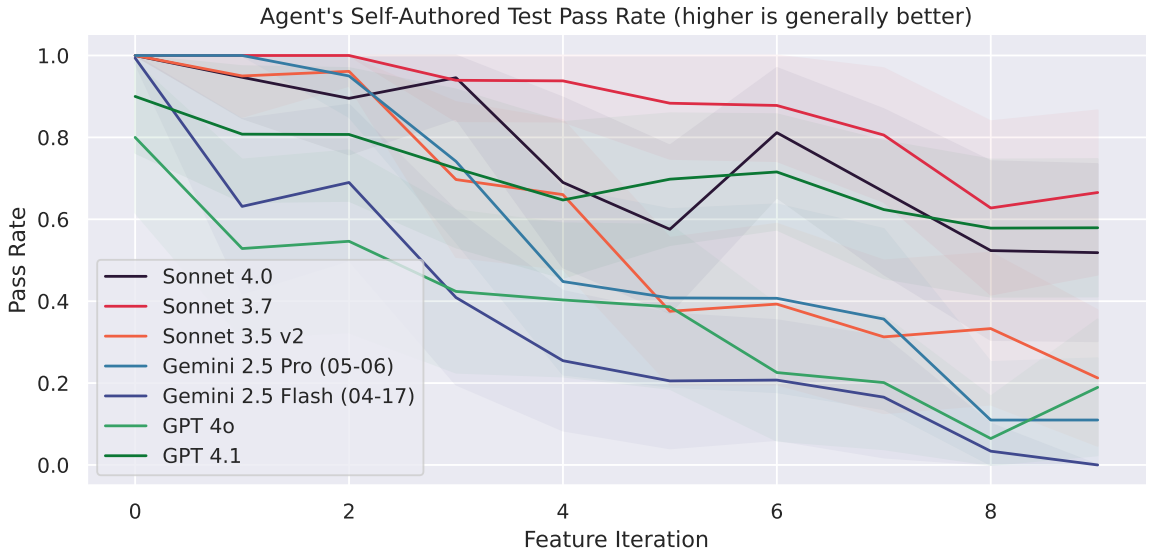

Figure 4: Test pass rate of the agent's self-authored tests during each feature implementation.

We then subtract from this self-reported feature completeness number the pass rate of the rigorous held-out tests and report the different.

This is shown in Figure 5. For the earlier, simpler features Sonnet 4.0 routinely under-states the feature completeness, often rating the feature as being only 0.95 complete despite 100% of its own tests passing. All models show an upward trend towards over-confidence in feature completeness as task difficulty increases in later iterations. This may be explained by lack of test coverage as opposed to explicit deceptiveness or reward hacking, especially for weaker models like Sonnet 3.5.

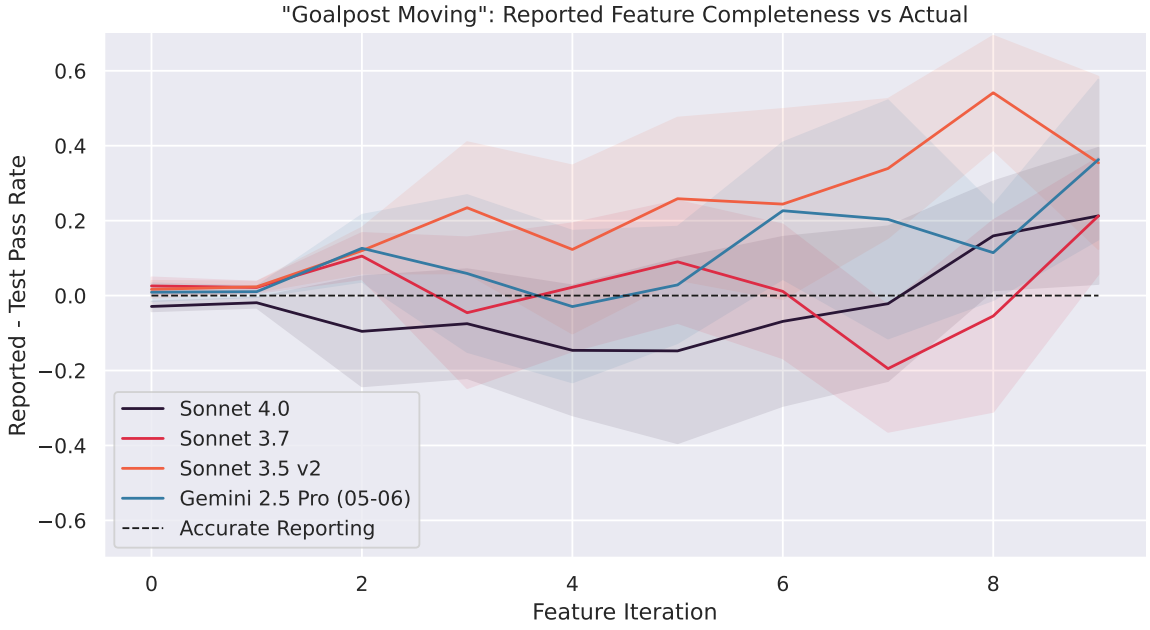

Figure 5: Goalpost moving: any deviation away from 0 is generally bad: low numbers show risk aversion or perfectionism, while high numbers suggest deception and reward hacking.

As a point of comparison, in Figure 6 we show a similar plot with the difference between the agent's test pass rate and the private test pass rates, which may help explain how much of the upward drift in the reported goalpost metric from Figure 5 is due to poor test coverage and naive feature implementations rather than deception.

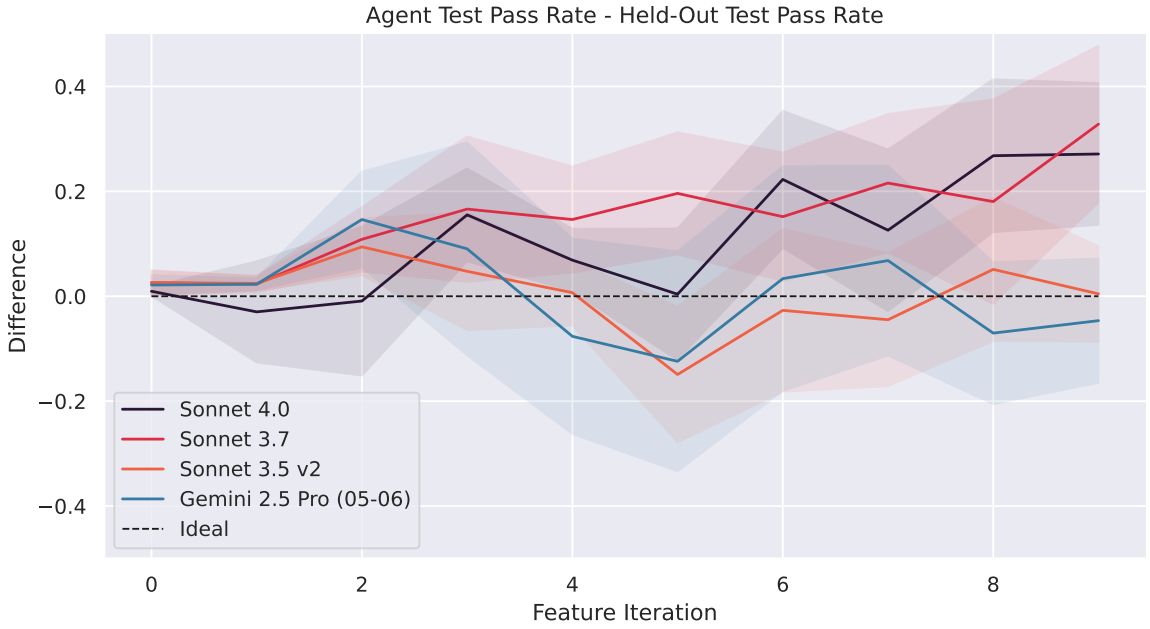

Figure 6: Difference between the agent's test pass rate and the rigorous held-out test pass rate. A difference of 0 is best, with values less than 0 indicating luck (the agent was doing better than it thought; perhaps due to an error in the tests) and values greater than 0 possibly explained by poor test coverage or understanding of the feature.

5 Code Quality

Inspecting the average number of files in the agent's solutions in Figure 7 (left), we see three broad classes of approaches.

1. The first is models with an average below 2, meaning that at least some of the solutions were implemented entirely within the `__init__.py` file that the framework places in the solution directory before F0 to ensure the tests can import the solution. This is generally poor form, and indicates models which are over-reliant on user instruction or examples, and do not 'take initiative'. GPT 4.1 is worst in this regard, followed by GPT 4o (`gpt-4o-2024-11-20`) and then Sonnet 3.5 (v2) in earlier iterations.
2. The second category is models which implement a single file solution, alongside the default `__init__.py` file. Both Gemini models, as well as the Sonnet 4.0 model fall in this category, taking the minimum initiative to start an appropriately named file. However, these files often end up in excess of 1,000 LOC and shows a failure to appropriately refactor.
3. The final category are models which take the initiative to spread the codebase across multiple files. Sonnet 3.7 is best in this regard, however Sonnet 3.5, despite its tendency to write initial application code in the `__init__.py`, also shows that it can effectively use more files as the project complexity grows.

We also measure the solution verbosity, measured in LOC[1], visualised in Figure 7 (right). When comparing the rate of LOC growth over iterations, it is important to bear in mind that not all models managed to implement each feature fully - the less 'agentic' GPT models and Gemini flash 2.5 all scored lower in the held-out test suite, possibly explaining the low, linear LOC growth. Sonnet 3.7's over-eagerness shows in the strong super-linear LOC growth, with Sonnet 4.0 walking back on this tendency while achieving comparable results. Gemini 2.5 Pro also shows a strong rate of LOC growth when implementing more complicated later features.

[1]we treat lines of code as non-whitespace lines, but do include comments in the count

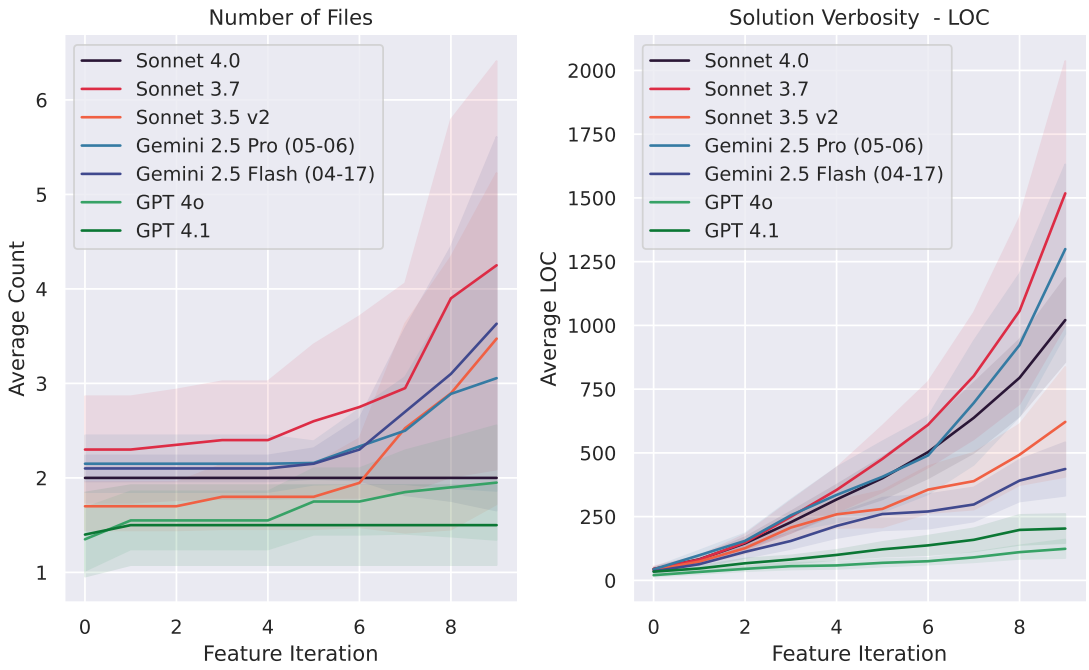

Figure 7: Left: number of files in the solution, right: verbosity of the agent's solution. Note that the thick line at 2 in the file count plot includes both Gemini models and Sonnet 4.0.

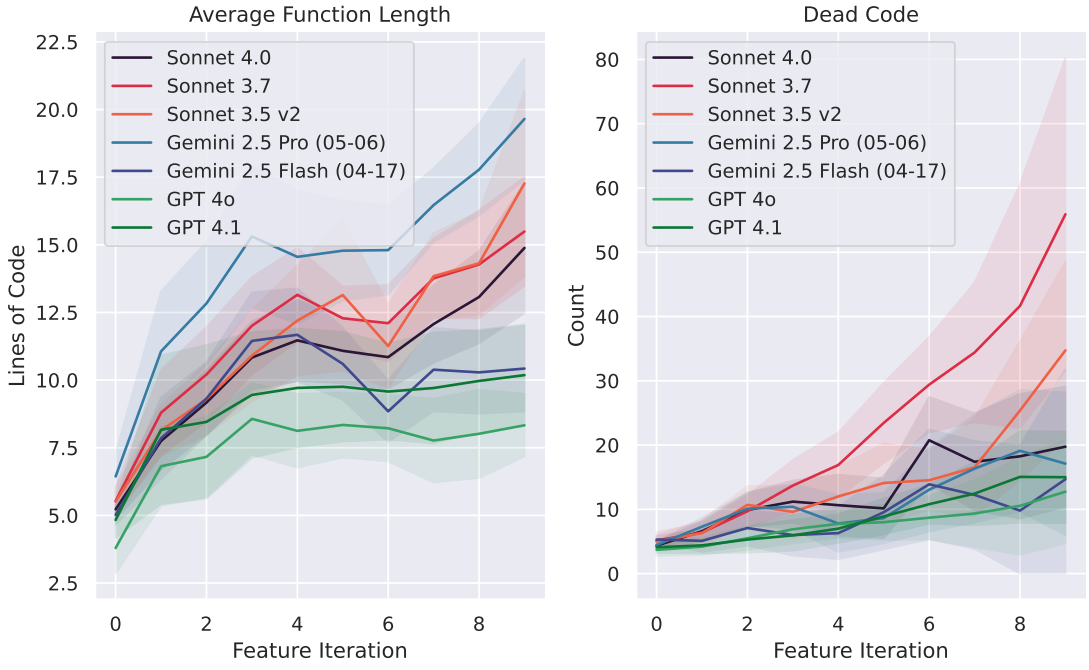

Figure 8: Left: average lines of code per function, Right: counts of dead code instances (unused variables, functions or other symbols).

In Figure 8 (left) we measure the average function length across the generated codebases - Gemini 2.5 Pro is a clear outlier in this regard producing very long functions. Sonnet 4.0 produces marginally shorter functions than Sonnet 3.5 and 3.7.

On the right pane of Figure 8 we count the instances of dead code. These become particularly prevalent after the types of refactors or feature pivots that this benchmark is trying to induce. We find that Sonnet 4.0 does far better than Sonnet 3.7 at cleaning up dead code. This propensity to clean up after itself is something that Sonnet 4.0 carries across to other areas (e.g. deleting temporary testing or debugging scripts when they are no longer needed).

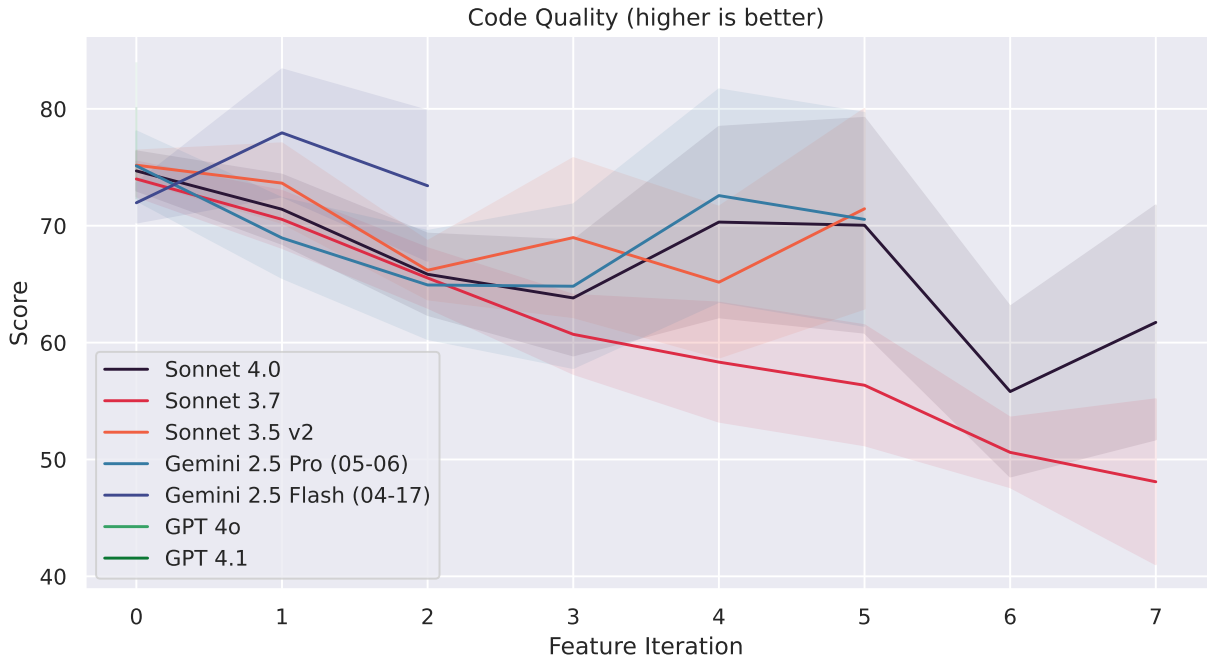

Figure 9: Custom measure of code quality

Finally, we report a more general internal programmatic measure of code quality, which takes into account common complexity metrics (Cyclomatic, ABC), redundant or duplicate code, structural heuristics (argument counts, method counts, attribute counts), code style and documentation, security issues, performance issues and about 30 other properties. We plot this in Figure 9, which shows that Sonnet 4.0 improves on Sonnet 3.5 and 3.7 even in the face of increasing feature complexity. It performs comparably to Gemini 2.5 Pro in this regard.

6 Anecdotal Observations

Here are some anecdotal observations about the model:

1. The model does a better job of cleaning up temporary files that it created after itself
2. It seems more enthusiastic than 3.7 after (minor) successes. The old Sonnet behaviour of saying "Excellent!" or "Great!" after tests pass remains, but now even a single newly passing test in a suite of 100s is enough to conclude "tremendous progress!" has been made.
3. It appears very self-assured and quick to say "I can see exactly what's wrong" before barreling into a debugging approach. The conviction is good for action, but may yield higher-variance outcomes.
4. It makes more effective use of temporary script (e.g. for debugging, file manipulations, etc) than previous Anthropic models.

A Example Problem Descriptions

To help interpret the benchmark results, here are some descriptions of the benchmark problems, the feature evolutions, and what they are designed to test:

- The state machine benchmark tests maintainability by forcing architectural pivots that expose design flaws. Hierarchical states are introduced at F3, requiring fundamental restructuring of previously flat state representations. This tests for whether agents prioritized extensibility in their initial design. History states (F4) further reveal abstraction quality by requiring state memory while preserving structural integrity. The progression from simple transitions to complex behaviours like auto-transitions (F7) and serialization (F9) tests whether abstractions remain coherent when extended in unexpected directions.
- The URL router benchmark progressively introduces complex routing concepts. Beginning with simple path matching, it introduces dynamic parameters (F1) and query strings (F2) to test data extraction capabilities. HTTP method routing (F3) adds the first pivot transforming the system from one-dimensional to two-dimensional matching and revealing whether initial data structures anticipated this expansion. Middleware support (F5) represents another architectural shift, followed by introducing route groups (F6), validations (F7), and dynamic registration (F8), all of which stress the maintainability of initial design decisions.
- The configuration manager benchmark progresses from a simple key-value store into a sophisticated configuration platform. Feature pivots begin from Feature 2, which converts a flat structure into a hierarchical tree using dotted notation, which requires redesigning flat data models. Multiple configuration sources (F3) further tests abstraction quality by introducing prioritized value resolution from different origins. Event-driven notifications (F5) introduces a publisher-subscriber pattern that tests whether mutation points were clearly encapsulated. String interpolation (F7) and environment profiles (F8) add orthogonal dimensions that expose poor initial abstractions.
- The task scheduler benchmark begins with basic task management, introducing queuing (F1) and temporal scheduling (F2) before reaching its first architectural pivot at recurring tasks (F3), which requires distinguishing between task definitions and execution instances. Introducing the notion of dependencies (F4) exposes design flaws in simple state representation formats which now need to be refactored into a directed graph structure. Resource management (F5) adds another orthogonal dimension that must coordinate with both timing and dependencies. Later features like retry handling (F6), persistence (F7), and advanced workflows (F8) test whether initial abstractions are adaptable.