

[We previously noted that, while it's easier than ever to hit a particular quality bar by having coding agents use effective test techniques, software quality seems to be getting worse](#), indicating that whatever defaults developers are using may not work very well. Here, we test if simple instructions to agents to use particular techniques or libraries improve implementation correctness, as a kind of test to see how effective agents are when guided by someone with no expertise in testing who's maybe heard that you should apply certain techniques or use certain libraries.

We'll re-use [the Zstd implementation eval discussed in this comparison of agentic programming language effectiveness](#) and, instead, compare different testing techniques and testing libraries when agents are given a prompt to implement Zstd with different addendums, such as "Use test-driven development", "Use Lean 4", "Use QuickCheck", "Use property-based testing", etc. I also ran some other evals, such as on the IMAP RFC, which are briefly discussed.

All implementations were in Rust. The 26 prompt conditions tested were ACL2, Alloy, "Audit and fuzz risky areas", "Audit first", Creusot, Default (no additional instructions), Differential testing, Fuzzing, Hegel, Insta, Judgement (agents asked to use the best technique), Kani, Lean 4, "Make no mistakes", Metamorphic testing, Mutation testing, Property-based testing, Proptest, QuickCheck, rtest, Rust built-in test framework, SMT solvers (with Z3, cvc5, and Yices, all available), Spin, TDD, TLA+, and Verus. Additional, 4 skills were tested: Hegel with [the official Hegel skill](#), the [ECC Rust test skill](#) (ECC is a collection of skills with 250k GitHub stars and 38k forks), the [Trail of Bits property test skill](#), and a test skill I wrote (I'm a luddite who uses prompts instead of skills and have no feel for how to write a good skill). Other than my skill, the skills were chosen because those were the top skills codex turned up when asked to find relevant skills.

Predictions

I pre-registered some guesses on how conditions will do:

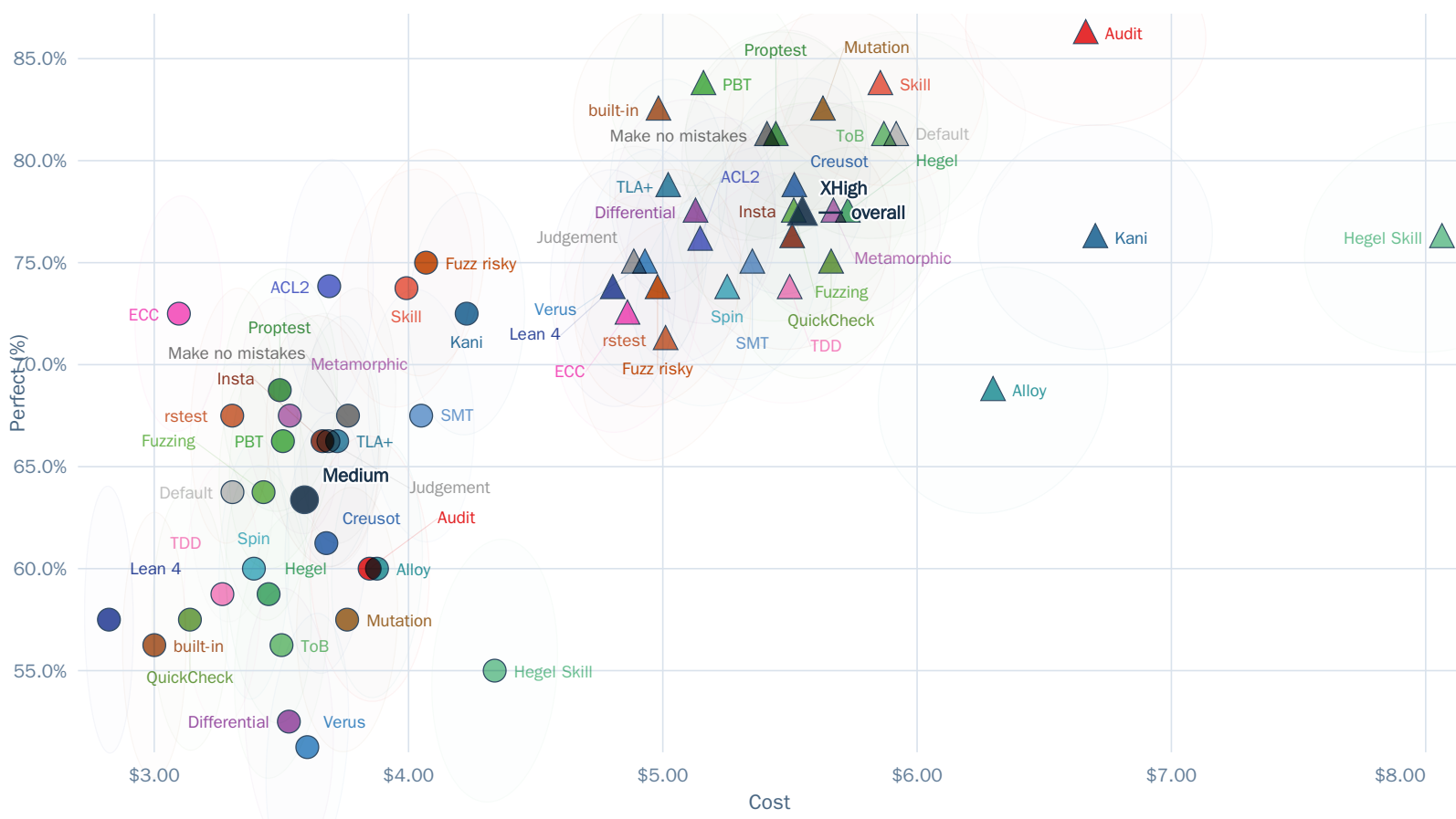
- TDD will underperform (55% confidence)
 - I actually added TDD specifically because I thought it would underperform
 - My confidence is low here because I don't know what agents will do when instructed to do TDD; perhaps agents won't do TDD and will do something that doesn't underperform (or perhaps I'm wrong about TDD underperformance)
- Formal methods will not overperform (52% confidence)
 - My thought here is that formal methods are effective and useful (more so now than ever), good test methods are also effective and useful and, on simple problems, formal methods shouldn't outperform if used at a similar level of competence
 - As with the above, but even more so, my confidence is low here because I don't know what agents will do when instructed to do anything, and formal methods have been more hyped than effective test techniques for agentic coding, so it's entirely plausible that labs have trained agents with RL environments with synthetic data which trains them to be very effective with formal methods without having trained agents to be effective with good test techniques (which I would expect to be easier to do, but not done because of how relatively untrendy effective test techniques are)
- Make no mistakes will not outperform no instructions (95% confidence)
 - It's a joke, and one that a lot of people have tried. If it worked, surely people would've noticed?
- The ECC test skill (with 250k stars and 38k forks) will not outperform (65% confidence)
 - It's somewhat big and doesn't have any information I'd expect to be useful. It instructs agents to use TDD; to the extent that it gets agents to use TDD, I'd expect this to make things worse (and it's more directive than the TDD condition and perhaps more likely to succeed, although for all I know

that makes it less likely to succeed); the rest of the information doesn't seem useful and has some cost

- All of my skill predictions are low confidence because I don't tend to use skills and don't know how to really evaluate them. I'm thinking of this like, "how effective would it be if I passed the text in as a prompt and had this thing floating around in the LLM's context window?"
- Hegel's skill will not outperform (65% confidence)
 - It's very big (the SKILL.md plus the linked Rust reference are over 20k tokens) and reads more like a tutorial than agent instructions
- The Trail of Bits test skill will not outperform (55% confidence)
 - It has what looks like it might be useful information, but it's also fairly big

Overall results

Below, we have a very messy graph which shows the results for the conditions tested (codex with GPT-5.6 Sol, with medium and xhigh efforts). When looking at data, I tend to prefer much denser and messier graphs than most people, [such as the first graph here](#). Because most people find these kinds of graphs unreadably messy, I tend to split information out into a series of graphs, each of which shows less information, when presenting information to others. For reasons discussed elow, I'm not going to do this here and am just going to present this extremely messy graph where we have cost on the x axis and the fraction of runs that passed 100% of the (hidden) tests on the y axis, average of 80 runs from each condition and effort (mousing over items shows bootstrap covariance, [50% uncertainty](#), and there's some attempt at making like things similar colors, e.g., blue-ish for formal methods, green-ish for property-based testing, etc.):



One thing we can see is that nothing really wildly outperforms. However, Default (no additional instructions) does well above average. Looking at xhigh, on average, the fuzzing and PBT-related conditions did a little better than formal methods on average, with the situation being a lot more mixed at medium. The testing-related skills codex recommended we try underperformed, although our quick custom skill did ok (a major difference is that our skill is designed to nudge away from their default behavior towards more productive

behaviors whereas the other skills seem more like tutorials). TDD didn't do well, as predicted (one skill also suggested that agents used TDD, and that skill also fared poorly in the cases where agents attempted to follow the instruction).

If we actually look at what agents did, it quickly becomes apparent that, in general, agents don't know how to use these tools or techniques very well. [As we noted here](#), and as everybody I've talked to has also noted, agents are really bad at testing and don't seem to understand how to test reasonably "by default". For example, [here's a comment by Gary Bernhardt](#):

AI agents' approach to testing, more or less:

1. Take the pathological cases dreamed up by someone objecting to mocks 15 years ago, without ever having actually used mocks. Naive dreams of excessive mocking.
2. Make those pathologies the backbone of your testing strategy.

It turns out, if you ask agents to use a particular test technique or test library, this approach doesn't change as much as you'd hope. We'll look at what happened in cases in more detail, but at a high level, with test techniques, agents tend to either just write the tests they would normally write, but inside a framework for a different type of test technique, or they'll use a technique superficially but not really do the things that get the value out of the technique. For the most part, when a technique was named, they did what Gary described, but with respect to that technique (for example, for formal methods, they mostly proved irrelevant properties and with property-based testing, agents would lean heavily on totally random inputs and heavily hit invalid/rejection cases or find a trivial property to check and run low-value random cases against the trivial property). Results weren't materially different on the IMAP RFC (where I tried 40 runs of each condition) or other random RFCs (where I tried a few individual runs). In general, regardless of the type of problem, whether it's some kind of bit manipulation problem like Zstd, a protocol like IMAP¹, or anything else, agents did not use formal methods or test libraries or techniques in an effective way.

On xhigh, agents were generally able to get the tests they wrote to pass, but they wrote poor tests (e.g., they'd submit four identical bitstreams into a test of a feature that uses four bitstreams and miss any bug that would occur because they transposed bitstreams). [And as we noted previously on the Zstd eval with respect to languages](#), running at a lower effort level in a naive loop gets worse results (agents do even more of this and stall out with lower correctness).

I'm curious why AI labs haven't created RL envs to get agents to learn how to test well since software not working reasonably seems important for coding agent adoption and it also seems like the kind of thing that's amenable to RL. [As we previously saw, agents have gotten quite good at bounded runtime optimization problems](#), which makes sense because that's exactly the kind of thing you cheaply create a ton of RL envs to train on. Maybe this is one of those things that's harder than it seems when you try it, but creating RL envs for effective testing and test techniques seems like it's in the same class of problem. Perhaps the limiting factor is just that knowledge of [effective test techniques](#) isn't very widespread, so no one's thought to try it and people are getting agents to test inefficiently (for example, by doing standard unit testing)², or maybe this problem is much harder to package up than runtime optimization for some reason? It's possible this will be a moot point soon if agents get so good that they can generally write correct code without testing or verification, but at least for the state of publicly available agents from inception until now (September 2026), it seems like agents having some idea how to test without being guided by a testing expert would've substantially increased agentic coding effectiveness.

Below we'll look at how agents did things for each condition, ordered from worst correctness to best, but I would caution anyone against drawing any kind of strong conclusions from the ordering.

A lot of the failures here seem analogous to the failures we saw when we looked at [the impact of programming language on token usage and correctness](#), in that the failures are often idiosyncratic. For example, with programming languages, we saw that agents had a fairly high rate of getting the semantics of byte conversion incorrect in Clojure but not Java, even though agents "should" (and probably sort of do) know that they can get Java byte conversion semantics by converting with unchecked-byte instead of byte.

Although people have all sorts of hand wave-y high-level explanations for why some languages are better for agents than others, when we look at what agents actually do and what the failure modes are, none of the explanations I've heard for why someone's pet language is suited for agentic coding, whether it's Elixir or Ocaml or J, are actually true (with the exception of comments about Rust's memory safety, [which were validated in the multi-language pandoc eval we tried by comparison memory safety issues between agent-written C, C++, and Rust](#)). Instead, we see a bunch of idiosyncratic failures that happen for unclear reasons³. With languages, because we can observe a moderate correlation between language popularity and performance (both lower cost and higher correctness), it seems reasonable to guess that the reason is because there was more training data (possibly synthetic data and not just human-written code) for more popular languages. Here, there isn't a clear pattern, other than that agents are mostly not very effective at applying test or verification techniques when all they have is the name of a library or technique (we'll discuss what works better afterwards). If you don't want to read about what happened in each condition, [click here to skip to the last item](#).

Verus

[Verus](#) uses an SMT solver and various types of reasoning to prove that the code matches specifications.

Although Verus can prove that code matches specifications, agents didn't do that. Instead, they made proofs about various abstract properties relating to Zstd. I've not used a tool like Verus myself, so I can't speak to what an expert or even a beginner user would normally do, but from reading the tutorial, I find it a bit odd that agents didn't attempt to use Verus to verify any of the actual code and only used it to do abstract reasoning, as it seems designed to make it easy to prove properties about the actual code.

Additionally, if we look at the properties proved, there were generally few properties proved and the properties that were proved were uninteresting. For example, agents would prove things like "given a valid cursor/index/distance, the resulting operation remains in bounds", which isn't bad to prove, but wasn't really a source of bugs. Also, agents would frequently write vacuous proofs that were effectively $A \Rightarrow A$. An actual Verus proof of this form was:

```
requires
  0 < a <= window,
  0 < b <= window,
  0 < c <= window,
ensures
  0 < c <= window,
  0 < a <= window,
  0 < b <= window,
```

In cases where agents actually proved something, they generally proved something relatively simple and avoided proving properties about the parts that were likely to have a bug (for example, agents often failed to reverse the bitstream order for encode and decode and would write tests that failed to detect this because the tests were palindromic; perhaps some kind of proof of reversal here might get agents to "think" about this in a different way).

It doesn't seem that agents were getting value out of Verus when just provided with Verus and the Verus docs.

If we look at the result, the aggregate xhigh Verus results are fine (slightly lower correctness than average, but much cheaper). The medium results had average cost and the lowest percentage of correct runs as well as the lowest average number of correct tests. Because agents didn't really get value out of Verus, what they actually did for correctness was mostly just traditional tests (built-in Rust `#[test]` functions with unit tests). When going from medium to xhigh, agents spend much more effort on traditional testing and only a bit more effort on using Verus, which allowed the xhigh result to be ok.

Looking at the actual tests, for one of the two features which agents using Verus did much worse on (the [four stream jump table](#)), Verus agents wrote a test for this in 89 out of 160 cases, coincidentally the exact same number as Default agents, but Verus agents were much more likely to write bad tests. They were more likely to encode incorrect results in the tests as well as make easy to pass tests that don't cover the space well, such as making all four streams identical. This kind of thing is what I meant when I said that the failures were idiosyncratic. There's nothing about Verus that necessarily makes one write poor tests when not using Verus and we wouldn't, in general, expect a human who's used Verus to write bad unit tests, in the same way that we wouldn't expect a human using Clojure to make more byte conversion mistakes, but this happened here for whatever reason (possibly a coincidence).

I don't know if folks inside AI labs can get access to better information on why things happened, but here on the outside it's generally quite difficult to tell why something like this happened (even when we formed a plausible hypothesis for the language issue, it required running many samples of many languages, and papers we looked at which studied the same thing didn't observe the language popularity / agentic effectiveness correlation because they either looked at too few languages to be able to reason about such a weak correlation or they looked at problems that were too small and too trivial).

Alloy

Alloy is often called a bounded model checker. This is maybe not quite right with Alloy 6 since that introduces some extra features, but this is way outside of my area of expertise. My understanding is that, with Alloy, you normally prove properties about your model (as opposed to proving that your code works).

Alloy got the 2nd worst correctness score and, unusually, scored generally poorly on both medium and xhigh. Although it isn't shown (because it doesn't seem to add anything), in general, results were highly correlated between max and xhigh, which were quite different from medium results.

As we saw with Verus, agents using Alloy pretty much relied on standard Rust `#[test]` for correctness and mostly fuffed about with Alloy. Once again, using a formal tool poorly did not help with correctness.

There were individual cases of Alloy use that were close to finding an issue or risk, but even then, only a small number. In one case, Alloy found a counterexample which then caused the agent to implement the Rust version with a mitigation for the potential bug. Unfortunately, the counterexample relied on an 8-bit overflow that couldn't happen in practice because the actual implementation used 64-bit `usize` with no possibility of overflow given the inputs, so it just made the code more complex without preventing an actual bug.

In another case, the Alloy specification was incorrect and a related test failed. After the test failed, the agent fixed the Alloy specification. Had the specification been correct, perhaps the agent would've written the correct code without the failure. There were some cases where it's possible the good version of this happened, but it's not clear if an actual potential bug was prevented.

Alloy agents did model things that were more closely related to the Zstd algorithm than Verus agents (which mostly checked things like arithmetic), but it was still the wrong modeling.

Differential testing

Differential testing is a technique where you give the same inputs to multiple implementations and then compare results to find issues. In principle, this seems like a reasonable thing to try with LLMs as we often get different results from different rolls of the dice, and [as we noted here](#), having an agent iterate more on an implementation (which might be only part of the entire thing, perhaps even only part of a function) often works worse than having the agent restart from scratch.

But this gave us the third worst results. In this case, we had slightly above average results on xhigh and far below average results on medium. None of the agents created two full implementations to compare. Out of 160 runs, 135 did something you might call differential testing, but like the other conditions we've seen, these were generally trivial and effectively useless. And, in the cases where differential testing might've caught a bug, instead of implementing things in independent ways, agents just did the same thing twice and encoded the same bug in both versions.

I sometimes tell agents to do things independently and get them to launch with separate contexts, but this was not done effectively for differential and agents would generally just write the same thing twice.

Hegel Skill

It makes sense to discuss how the official Hegel skill changes Hegel behavior, but in reverse correctness order, Hegel Skill appears above Hegel because the result was worse on correctness. See the Hegel section below for discussion of this skill.

Lean 4

Lean 4 can maybe be described as [an interactive theorem prover](#).

Although I didn't pre-register a guess about Lean, if I had pre-registered guesses on which formal tools would do well, I would've put Lean on the list of things I'd expect to do well because it's relatively hot/trendy and therefore seems relatively likely to have good performance due to synthetic data from RL envs.

The Lean agents did prove properties, like the Verus condition, agents mostly did arithmetic proofs that didn't hit the bug-prone or risk surface areas.

Like the other formal conditions, Lean agents relied heavily on standard Rust tests. As with the formal conditions so far, doing a few proofs of things that don't matter didn't help with correctness.

QuickCheck

QuickCheck is [a property-based testing library](#), probably the best known such library for a long time, although Hypothesis might currently hold that crown.

Unfortunately, agents were about as effective at using property-based testing as they were at using the formal tools we've seen so far. When using QuickCheck, agents mostly wrote very simple "smoke tests" that didn't check much. They also used random inputs, which, when fully randomized, are pretty poor for testing something like Zstd (because they just go down one of a few failure/rejection code paths).

Also, relatively few properties were checked. Although all agents used QuickCheck, 63 out of the 160 runs only checked a single property. Agents once again mostly relied on traditional testing, although they technically did use QuickCheck. For whatever reason, agents actually wrote more traditional tests than under the Default

condition or most other conditions, but did fewer test-fix iterations (which resulted in this condition coming in with below average cost).

TDD

TDD underperformed here as well as in the IMAP RFC eval.

The TDD prompt seemed to cause large changes to agent behavior. Agents produced twice as many tests, and worked in a much more iterative test-code-test-code-etc. workflow, although a TDD advocate would probably say that agents didn't actually use TDD. There were only a few instances of agents doing some kind of fine-grained iterative TDD.

Overall, agents wrote more tests up front; for example, agents had one or more failing tests in 67 of 160 cases before doing substantial (non-stub) implementation, vs. 0 of 160 for the Default condition.

For broad test classes, TDD had more tests of every kind. There were more small, trivial tests and there were also more integration and end-to-end tests. Any kind of obvious high-level "agents did too much or too little of X" doesn't seem to fit the data. If we look at specific failures and how they were missed by tests, we can observe that the TDD condition had a number of these. For example, Zstd uses something called a jump table when there are four Huffman streams.

TDD agents were more likely to fail the eval test for this although they wrote more tests that cover the general case. For whatever reason, TDD agents were more likely to write tests that don't cover hard cases (e.g., making all four streams identical and then also making them trivial, like we saw with Verus). This is another case where I'd be curious what kind of visibility people at AI labs have since it's not obvious from the outside why priming agents with TDD made them write worse tests and worse implementations.

If we only had TDD and a few test conditions to go on, a hypothesis might be that TDD'd code often seems to have a lot of small tests that aren't very good, so maybe priming agents with TDD causes them to write more of these sorts of ineffective tests. But it's not clear why we should see the same pattern with Verus. Maybe we could tell whether or not this is true for TDD if there's a shared reason for the Verus (or other) behavior by re-running the experiment on an open model and inspecting what's actually going on inside the model at some level?

Two of the skills also caused agents to run in a more iterative approach, perhaps on the theory that executing more frequently would give better results, and both of those skills also underperformed. In general, across all conditions, agents were able to get the tests they wrote to pass on xhigh and max (not shown, but max had slightly better correctness than xhigh at substantially better cost). Getting their own tests to pass more iteratively tended to get agents to write more incorrect tests that would enforce incorrect behavior.

Yossi Kreinin had this thought for why TDD might result in worse tests:

fwiw, I think if you write the tests before the code, it's harder to test the harder cases since you know less about what is going to be hard, and even if you do random testing which I don't think "tdd" is associated with, you are less likely to steer the distribution in the direction where the bugs are. if you wrote the code or at least can look at it, you know what seems trivially correct and what might or might not work since it's not easy to understand what it does. in other words, tdd steers you towards black box testing which for complicated machinery seems to me to be less effective than white box testing; pretty sure this is how it works with people, less sure about agents

Was my guess that TDD would underperform correct? Strictly on the result, the answer is yes. On my reasoning (not explicitly pre-registered in writing, but I do know what I was thinking), I think it's not clear. My thinking was

something like, as we've [recently discussed](#) in [a variety of contexts](#), getting agents to actually do something like the right thing and not just overfit is a key part of achieving good performance or correctness with agents. Speaking to the methodology in general and not how this instruction changed agent behavior, TDD seems primed to cause overfitting.

Agents did write worse tests and sometimes used a relatively expensive and ineffective iterative workflow, but I don't know that the failure mode I'd expect from a human using TDD and then directing agents to implement was the real problem here, and that problem was where my intuition came from. I would rate the reasoning here as perhaps and perhaps not in the right vicinity; I think more evals and investigation would be necessary to decide this and I would guess that the result of additional data would be that my original reasoning is wrong.

Spin

Spin is a model checker.

Now we're getting into the range where results weren't far from average. Spin did moderately worse than average on both medium and xhigh, at below average cost. As we saw with the other formal tools, usage of Spin was generally ineffective. In this case specifically, using Spin to model a certain class of behavior had no correlation to passing or failing the hidden tests covering that behavior. Usage of Spin was superficial and not productive.

Hegel

Hegel is a property-based testing library based on [Hypothesis](#).

As we might expect by now, agents didn't use Hegel effectively. To the extent they used it, they used it superficially, and they generally used it after heavily relying on ordinary testing. Since just saying that agents didn't really meaningfully do the thing is repetitive, I'll make these sections short and only highlight particular curiosities.

The actual workflow agents used was generally

1. Read the RFC and API/contract
2. Implement Zstd
3. Run normal tests
4. Read Hegel docs
5. Use Hegel to write 1-4 simple property-based tests
6. Continue using normal built-in Rust tests

As noted above, the Hegel skill didn't improve correctness. Correctness was worse (though it was close enough that this could've been random). What was more striking was that cost was much higher (26% higher on medium and 41% on xhigh), for reasons which seem causal.

The skill caused agents to generate more tests. The additional tests were mostly checks that malformed inputs don't cause a panic and round-trip tests. The former is something that agents were already inclined to do an excessive amount of for all of the property-based and fuzzing conditions, so additional effort there wasn't useful. The latter doesn't seem like an inherently bad idea (I even often explicitly instruct agents to create round-trip tests and they seem to be useful to check specific properties), but it wasn't done in any of the most bug-prone areas. Without additional instruction, agents were inclined to create round-trip tests for relatively trivial properties that were already likely to be correct.

As for the cost, there are multiple reasons for the cost. One is that the skill is fairly large (34k characters for the skill, which also loads a 45k Rust-specific reference, which ends up being more than 20k tokens). This was loaded at the start of the run and was re-read on many subsequent actions. This resulted in an average additional dollar cost of 16% for medium and 18% for xhigh (by raw tokens, the average increase was 900k on medium and 1.8M on xhigh; although the cache hit rate on these was very high, 99.85% after the initial read, they were re-read enough that this was still a substantial fraction of total cost).

A multiplicative cost (this multiplier is included in the previous numbers) is that the skill also specified a structured set of operations that cause a lot more work to get done. This work didn't increase correctness, so this increased cost without a concomitant benefit.

One thing to note is that the skill was "only" used in 157 out of 160 cases. As is generally the case when using LLMs, the actions and results are random. If you have a skill available that you think an agent should use for a particular task, it may or may not use it depending on factors that seem opaque to people outside of AI labs.

ToB skill

In this case, only 108 out of 160 runs actually opened [the skill](#) to read it. The skill suggests using proptest in Rust, but the skill suggests approval is required to add a dependency and these were all single-turn autonomous runs, so this wasn't done.

As with the other property test cases seen so far, property testing was rudimentary and not done in a helpful way.

Rstest

Rstest is a [fixture-based](#) test library.

Agents effectively didn't use rstest. They technically did use it, but they pretty much just wrote standard unit tests inside rstest and didn't use rstest as intended, defeating the purpose of rstest. While this is arguably true at some high level for techniques seen so far, agents were at least superficially using some of the other techniques (such as writing some low-value property tests with Hegel), but here agents didn't use the thing that makes Rstest Rstest (the analogous behavior for the property-based testing libraries would be if they just wrote non-property-based unit tests with them).

Rust test

This is referring to the standard Rust built-in test framework that agents used in the Default condition and also very heavily relied on in the other conditions.

Explicitly asking agents to use the built-in test framework resulted in more tests (double normal on medium, 25% more on xhigh), but this didn't result in better correctness. When agents got things wrong, it was often because they didn't test significant behavior or implemented incorrect test behavior. Adding more tests didn't materially increase coverage of risky behaviors or reduce the fraction of runs with tests that encoded incorrect behavior.

Yossi Kreinin added:

i think the fixed input/output style of testing encourages this in machines and humans alike. if you generate inputs you need to then have code that classifies output as correct or incorrect, and while

this code itself might be buggy, it at least makes you think about what correct means and how to tell if something is correct more easily than running the code and assuming its output is the right answer. with fixed outputs you are quite likely to just encode the output of the code and convince yourself that it makes sense

Creusot

Creusot sits in the same space as Verus.

As we've seen with the other formal conditions, Creusot was not used effectively.

Mutation testing

[Mutation testing](#) involves modifying the code to determine how effective tests are and then adding tests to get good coverage. Although mutation testing is a standard programming term, agents generally didn't actually do mutation testing and instead did normal testing with some small amount of mutating things in a way that isn't really mutation testing, similar to how the TDD instruction modified behavior but didn't get agents to do TDD.

There were a few cases where mutation testing occurred, but only a small amount, and that was rare.

Judgement

This condition asked agents to adaptively use testing methods as appropriate based on their judgement. Given what we've seen so far, unsurprisingly, agents mostly used standard Rust unit tests. A few agents did some limited fuzzing. Agents had access to other test and formal libraries but didn't use them.

Fuzzing

Fuzzing involves randomizing test inputs in some way.

Agents relied heavily on sending random bytes in, which mostly resulted in going down the same code paths (invalid input). Agents also tried sending in random variations of valid inputs, which mostly also just repeatedly exercised input rejection paths.

On the rare occasion that agents generated random structured inputs (10 out of 160 cases), this found real bugs half the time, some of which were non-trivial cases. Using fuzzing a bit effectively in 5 out of 160 cases isn't exactly good, but this was one of the more effective uses of a technique that we've seen so far. This also seems to indicate that agents could be trained to do this better and also that that can be directed to do this better without changes in training. They do, in some sense, know how to do this; they just don't normally actually do it without being pushed into doing it.

Insta

Insta is a library for snapshot testing (sometimes called golden testing), where you compare results to a "snapshot" or "golden file" of correct results. Speaking generally, a snapshot is usually some kind of serialized data, e.g., it could be a JSON object of a data structure, a log of CLI output, etc.

As you might expect, snapshot testing was barely used and agents mostly relied on traditional tests. Agents did use Insta, but would often just write normal unit tests in Insta.

SMT

Agents were instructed to use an [SMT solver](#), with Z3, cvc5, and Yices installed.

Agents mostly used the SMT solver as a kind of scratchpad to compute things like FSE state ranges, header arithmetic, etc. Even when agents modeled something, they would generally not model the right thing to avoid a common mistake.

For example, there's a computation that should've been `byte1 + (byte2 << 8) + 0x7F00`. Many agents implemented `byte1 + (byte2 << 8) | 0x7F00` instead. Agents used SMT solvers to prove properties relating to this computation, but then still wrote the wrong code, making SMT use seemingly no better than Default (no instructions).

TLA+

TLA+ is a language and tool for modeling behaviors.

We're into the set of above average results (but still worse than Default) but, as noted above, I wouldn't take the actual ordering too seriously. Though this isn't necessarily significant, TLA+ did score a bit above average on medium and more above average on xhigh.

159/160 agents created some kind of TLA+ model, generally a state-machine model of Zstd. For particular coverage, 30 modeled Huffman/FSE/entropy (areas that often had bugs). As with the other formal cases, TLA+ modeling happened relatively late in the flow (after a lot of standard tests and implementation). Agents sometimes found and fixed errors in the TLA+ model, but I didn't find an instance of a TLA+ issue resulting in an actual change in the Rust code.

Although there was some real looking TLA+ modelling happening, if this improved correctness, it did so in a small way that was difficult to observe. In general, runs that had more sophisticated TLA+ modeling did not have better correctness.

Metamorphic testing

With metamorphic testing, we check that related inputs produce outputs with the expected relationship. For example, you could check that, for a sort function, changing the order of unequal inputs doesn't change the order of the outputs, or for addition, adding a value to an input adds the value to the output modulo overflow.

As we've seen for the other conditions, Metamorphic testing wasn't done very usefully with respect to correctness. Some actually reasonable properties were checked (e.g., inserting a skippable frame at a frame boundary shouldn't change the output, legal block repartitioning shouldn't change outputs, etc.), but these didn't hit the areas that agents got wrong relatively frequently so checking these properties didn't help. In general, agents seemed to be fans of the old joke:

A policeman sees a drunk man searching for something under a streetlight and asks what the drunk has lost. He says he lost his keys and they both look under the streetlight together. After a few minutes the policeman asks if he is sure he lost them here, and the drunk replies, no, and that he lost them in the park. The policeman asks why he is searching here, and the drunk replies, "this is where the light is".

Curiously, metamorphic testing was used less on xhigh than on medium.

ECC

The [ECC Rust test skill](#) did ok, but mostly because large parts of the skill were ignored. Agents generally opened and read the skill (153/160 read it) and this seemed to cause them to generate more tests. Not only

did agents generate more tests in this condition, if we look at when agents read the skill (earlier vs. later vs. never), there's an exposure-based gradient in how many tests were added.

Although ECC scored almost as well as Default, based on how agents did when more exposed to the skill, I would guess that this is random. The earlier an agent looked at the skill, the more its behavior was impacted and the worse the correctness result.

ECC seemed to do ok in terms of raw score because the 7 agents that didn't read the skill did unusually well and got a 100% correct result, and then the 9 agents that looked at ECC late and were only barely influenced also did well and had 100% correctness. This also explains the unusual ECC result that medium had the same score as xhigh (all but one of these runs where agents didn't really look at the skill happened on medium). While it's true that there may be some kind of bias in when the skill gets invoked or not, the overall pattern would indicate that ECC is not effective unless you think ECC acts as a good luck charm that improves results, but only when the skill isn't really used, which is more likely to happen at lower effort levels.

Of course, agents shouldn't be influenced by a skill they didn't look at and we should score this based on the cases where the skill was used. If we look at the cases where the skill actually influenced agents, ECC scores below average (between Rust built-in framework and Creusot), with a very similar failure mode to Rust built-in framework of having a large number of small and not meaningful tests. The skill tells agents to use red-green TDD. The agent behavior probably isn't what a TDD practitioner would call TDD, but agents do write a small test before implementing functionality, which results in a large number of tests. As noted above, this isn't an effective way for agents to develop, so the result is worse than no instruction and no skill.

BTW, [as we noted when we tried out Caveman mode](#), there's quite a bit of variance and people are often misled into thinking a skill is useful by a few small runs. In this case, we tried 160 runs of a skill, a fairly large number, more than any reasonable person would do. And yet, superficially, if we just look at the score, ECC seems ok.

We would need a much larger number of runs to average out the noise inherent when using an LLM. We can do what we did here, and inspect the results and use our human brains a little bit, but I rarely see this done when people are talking about public LLM benchmarks, whether it's for skills or anything else (I did try having LLMs analyze the results but, as usual, even with current public SOTA models, the analysis was poor and full of basic reasoning errors). Instead, I mostly see people pass around the top-line number, even when it's not meaningful for boring statistical reasons or, worse yet, [the benchmark is fatally flawed, as we saw with Senior SWE-Bench](#).

Default

Default gave the agent no test or verification instructions.

Given what we've seen so far, it's not surprising that Default scored above average. Agents generally did things that were not useful when asked to use particular libraries or use particular test techniques. It stands to reason that not telling agents to do things that will make them do useless work does better than telling them to do things that will make them do useless work.

Audit

Audit asked agents to audit the code after implementation. 152/160 agents actually did this and 151 agents claimed find an issue and then made a change as a result of the audit. Agents generally picked reasonable areas to audit, but usually didn't do an independent audit with a fresh context (which I will often ask agents to do) and often just made the same mistake in the audit that they had already made.

42 used an independent agent, but these runs actually scored worse (it's possible this isn't causal and agents decided to spin off an independent audit because they were in a worse or harder situation). Audit ended up with the best correctness on xhigh, but below average correctness on medium, and all of this auditing substantially increased cost, especially on xhigh. On average, Audit did about as well as Default and it's not clear if it's really better on xhigh and worse on medium. That would be plausible, but I don't think we have enough evidence to tell.

Em Chu had the following comment:

The results here are consistent with my experience. Auditing code is where most of my tokens go at the moment because I find it quite useful. I always give the two instructions though:

- Don't spawn subagents; read and understand the code/diff yourself
- Don't execute any of the code

because I find the LLM to be significantly dumber if you let it do either of those (though of course I haven't measured...). It really doesn't read or reason about code by default, even if I'm never making a change big enough to exceed its context window.

I also usually include some BS like "be adversarial" "consider all possible combinations of features" "consider the entire input space" but I'm less sure that helps at all.

It would be interesting to try that, but as I've noted in my recent posts, I'm trying to go into less detail in posts, so maybe that will be a topic for another post.

Audit and fuzz risky areas

For Zstd, when this instruction was followed, it caused agents to focus heavily on [FSE](#), [Huffman](#), bit readers, and state. These were areas where, in general, agents often missed issues, so agents were correct to think that these areas were risky. The areas that were targeted for fuzzing were better choices than the plain Fuzzing condition.

On medium effort, agents mostly ignored the instruction and didn't do it, but they did follow instructions on xhigh. While this condition didn't perform poorly, it didn't seem to do better than no instructions.

When looking at what agents actually did, one issue was that agents often just generated a bunch of random inputs which were generally invalid and wouldn't test any interesting condition.

When a human tester generates randomized tests, they'll generally try to target the randomization in a way that generates "interesting" inputs and agents failed to do that. Agents also didn't check outputs very effectively and, in many cases, only looked for crashes. Fuzzing is often associated with only checking for crashes and not checking for properties, so this is maybe not too surprising, but it's probably not what a human would want if they were testing a Zstd implementation.

Make no mistakes

Although this technically scored higher than Default, behavior didn't seem to be meaningfully different and the scores are quite close; I would guess that this is due to random variation. At every level at which I looked at the results, they were indistinguishable from random draws of Default.

Kani

Kani is a Rust [model checking](#) library.

In terms of "actually using a formal method on the code that will execute", Kani had the best coverage in that Kani was actually used on the Zstd code. However, that only happened occasionally and most use was superficial.

There was one case where real Kani use caught a non-trivial bug that caused Rust code to change. 1 out of 160 isn't amazing, but it does indicate that agents can stumble into using Kani reasonably sometimes (which, I would guess, means that, if used in an RL env, models could learn how to use Kani more effectively).

Kani had noticeably higher cost than other conditions. This seemed to be because reading Kani output repeatedly was expensive, which resulted in a high input token cost.

ACL2

ACL2 is a theorem prover. One thing to note about the result here is that, in many cases, ACL2 OOM'd (192 GiB limit). OOM results weren't counted, which biases the results in some opaque way.

Although ACL2 scored higher than Default, I think it would be surprising if this was causal and significant. As we saw with almost all of the other formal methods, ACL2 was mostly used to prove things that didn't significantly impact correctness, so it's not clear why this would improve correctness.

With many different conditions, we wouldn't expect Default or the seemingly equivalent Make no mistakes to be at the top unless other conditions had severely degraded performance.

Proptest

Proptest is a property-based testing library.

Just as we saw with the other randomized testing, most tests weren't very interesting, and a too-heavy reliance on randomness caused poor coverage.

Despite generally poor use of property-based testing, proptest's shrinking (finding a simpler input that causes a test failure) did sometimes provide some value, which is better than the little to no value we saw in most other cases.

Property-based testing

As with the other technique-based approaches, agents had a container with all options installed. Every agent chose to use proptest, so this effectively became a 2nd proptest condition.

As with the proptest condition, tests were mostly not very good but they did sometimes find bugs and shrinking seemed to generate some wins.

I find it mildly interesting that this second "accidental" proptest arm also scored well above average, just like proptest.

Skill

Here, Skill refers to the skill I wrote to test having a simple skill (as opposed to the large/complex skills that were what I found when I asked an agent to find relevant testing skills).

Maybe I should use skills, but I generally don't and instead rely on prompting and seeing what happened and then prompting some more. As a result, I have no intuition for what makes a good skill since I don't have any practice at it, but Max Bittker suggested that it would be interesting to see the result with a test skill that attempts to encode some information I have in my head about testing. On seeing the result of this, he had an "I told you so" reaction.

I didn't write a pre-registered guess about this, but the guess in my head was that this wouldn't work well. [From my attempt at conveying this to humans in 2015](#), which I would say pretty much failed, I don't think I'm good at explicitly laying out how someone should test in writing. I've sat down with people and showed them what to do, which has generally converted them for life and turned them into [way above average](#) bug finders, but [being able to convey something by showing someone is a different \(and easier\) skill than conveying it by writing down how to do it](#).

In this case, the skill was:

- Think about areas likely to have subtle bugs before implementing; for each, state likely mistakes and plausible alternative interpretations, then come up with a check where the results differ (prefer asymmetric / boundary examples on both sides of the boundary)
- After implementing, for high risk areas, independently re-derive the result without context on production code and compare (fresh context, do not re-use helper functions)
- When feasible, use property-based testing or randomized inputs to try to explore the space, minimizing effort on no-panic or no-crash randomization
- When randomizing, lean towards inputs that will explore interesting state and code paths (don't just naively randomize inputs that all fall into the same error paths); this may require structured random inputs
- If you're unsure about details, use independent reasoning to check what's correct (fresh context, do not re-use helper functions)

This got the highest score, but didn't work as intended. It didn't really do the fresh context thing almost ever, so it was pointless to have that in there and we don't know if that's something that's effective that needs to be refined to force agents to do it more frequently or if it's something that should be removed (while it's technically possible it's happening at the optimal frequency, I highly doubt it).

We noted in "Audit and fuzz risky" that agents seemed to know how to identify risky areas. This was true here as well, but this didn't necessarily mean that agents did the right thing. For example, agents identified bitstreams being reversed for encoding vs. decoding in Zstd as being risky, but agents didn't do better on tests that exercised this. If we look at specific examples, for medium run #35, an agent identified this as risky, did independent derivations and an audit, but still failed. It had a relevant test, but the input was palindromic, so reversing the order gave the same result, allowing for a failing implementation that had this backwards.

Another issue, if we can call it that, is that all of the fuzzing / property-based testing was done "by hand". Given that agents seem ok at using proptest and that proptest has some useful machinery to lean on, this skill could probably trivially be improved by instructing agents to use proptest. The instructions to agents that were intended to minimize the standard failure mode of generating many useless "too random" tests directionally worked and a larger fraction of agents generated somewhat meaningful tests, but the tests were still worse than I'd expect a human to write (or an agent with active human guidance). Without iterating on this, I'm not sure what generic guidance would be good (as opposed to spending a few minutes looking at the structure of Zstd and giving Zstd-specific guidance, which is one kind of thing that's worked well for me on other problems).

As a first draft for a skill to iterate on, I don't think this is horrible, but I don't think it's really ready to use either. I could see an improved version of this working if it were tried with many more examples to make sure

there isn't overfitting to RFC-like problems, bit-manipulation-intensive problems, etc., but, since I don't normally make skills and haven't ever tried to iterate on one, this fails to capture what I or another human would do if really driving an agent.

Since I'm used to prompting and then looking at the result (not necessarily the code, but at least what agents say they did and some kind of agentic summary of what happened, and parts of actual results for some kinds of experimental work) and then re-prompting based on that, I'm not used to front-loading information, which is a fairly different problem than reacting to information. From previous fuzzing work, I've seen failure modes that agents often fall into and the skill was intended to prevent those failure modes, but it's easier to do this if you check back in even occasionally than to do it fully up front, and the up front instructions weren't sufficient to stop the standard failure modes, though they did mitigate them somewhat.

General comments

As we noted above, I didn't try to break down the data in a nice, easy to look at way. I didn't do that because, once we look at what agents actually did, it seems like they were mostly pretty ineffective and I don't think it's particularly interesting to see how well "agents using Verus badly" do compared to "agents using QuickCheck badly". One thing that I find a bit interesting is that, when asked to identify areas that are risky or prone to subtle bugs, agents were able to do that.

But, in general, regardless of the library or technique suggested, agents failed to use the technique. As [previously discussed](#), just asking agents to "test" or repeatedly asking them to test more results in poor testing. It turns out that asking them to use test techniques (some of which I've personally found to be highly effective) also results in poor testing. The quick and dirty skill I wrote seems like it could improve things a bit, but would need more than the 2 minutes I spent on it to be actually useful. Yossi Kreinin made the comment that the state of software testing is atrocious, therefore we should expect poor results if agents fall back to their training, so to speak, which is what we saw.

For whatever reason, agents seemed to be somewhat better at using proptest, although the level of testing was well below what I'd expect out of a reasonable human who's read the proptest manual and is given some direction on how to test. I'd be curious if agents that are given more direction are more effective with proptest than with other libraries, but that's a topic for another post as I've been trying to get posts out in half an hour and we're approaching 9000 words here, which is beyond a reasonable amount to try to type in half an hour.

How do you get agents to write good tests?

My experience has been, if you guide agents to set up a reasonable test and triage structure, getting agents to add to that effectively without a huge amount of supervision works ok-ish. Because of my background (bias), the kind of testing I tend to lean on is some form of randomized testing / fuzzing / property-based testing.

I talked to Jamie Brandon about this, and he's found the same with snapshot testing. He mentioned that, on one project, when he asked agents (using a variety of models) to do snapshot testing, they would say that they were doing it and then just wouldn't do it (they would write a unit test and then say they wrote a snapshot test). On a different project, he was able to get them to write reasonable end-to-end tests with mocked IO, but only after moving the tests into a separate crate and putting instructions in AGENTS.md to keep tests in the crate and not modify the public interface.

At least to date, I've been leaning more heavily on getting agents to write the test code than Jamie (my tendency has been to type to agents in a CLI; at least for now, he favors writing code by hand a lot more than I do), but it doesn't seem to matter how you do it as long as you set up some kind of reasonable structure.

[Similar to this earlier problem we looked at](#), it seems like doing anything remotely reasonable works. If you "talk to" an agent and give it light instructions like we did for the Zstd or IMAP evals, the agent will do poor work. But if you look at what it does and type a few more sentences, you can often get it to a good place pretty quickly (or that's what my experience has been on other problems, anyway). Without having ever attempted to understand what makes models work, my made up and probably completely wrong guess for why it seems like doing something remotely reasonable generally works is that, somewhere inside the model is some kind of understanding how to do this stuff. It's not the default, and it's not even near enough to the default that naming what technique to use works, but if the model is sufficiently primed, the knowledge for how to do this stuff will actually get put into practice.

I'd be curious if this can be effectively packaged up into skills, or if AI labs are going to start training models to get better at testing or formal methods, or if whatever they're doing that doesn't directly improve those things will still improve those indirectly enough that agents will write decent tests without much supervision or structure.

Prediction accuracy

- TDD underperforms (55% confidence)
 - True
- Formal methods do not outperform (52% confidence)
 - True, but not for the reason I expected. Agents failed to use them remotely effectively, so of course they couldn't outperform
- Make no mistakes doesn't outperform no instructions (95% confidence)
 - True; outperformed most conditions because a no-op is better than getting agents to do ineffective things
- ECC skill will not outperform
 - True; I think I would've had more confidence in this if I used skills more, since most of the skill text seems like it won't do much of anything, and the text that seems like it will do something looks counterproductive
- Hegel skill will not outperform
 - True; another one where I would've had higher confidence if I'd used skills more, since this didn't work well for the reason I guessed; I just didn't have any confidence in my feeling about this
- ToB skill will not outperform
 - True

Guesses I didn't register, but I could tell I held implicitly because I was surprised when I saw the result:

- Lean will do relatively well among formal methods
 - False
- My skill will be mediocre to bad
 - False, even though Max Bittker correctly guessed what would happen, told me this in advance, and named a reason that's consistent with what happened; it feels quite silly when someone names the reason something is going to happen and you don't believe it, and then it happens for the exact reason they named

Skills

At various times, I've felt like I have a bad/antiquated/ineffective workflow because I hear people are doing something and I've been too lazy to try it out. I've felt this way about skills for a while since I don't really use

skills. Instead, I keep a large scratchpad of things that I sometimes copy+paste in as prompts, which sort of feels like the equivalent of commenting out blocks of code to save them instead of using version control.

But then [I saw this talk by Thorsten Ball](#), where he mentions he doesn't rely heavily on skills, and I talked to a couple people who seem relatively effective with LLMs who also don't really use skills, and it made me wonder if I'm not missing out on much.

Then I tried this experiment, where my feeling was that the skills I looked at weren't going to help and are probably actually going to hurt, with low confidence since I don't know anything about skills. The skills did pretty much what I thought they would do, so it turns out the intuition I have from just seeing how agents respond to things and running a bunch of little experiments seems to hold up ok for skills. I also looked at a number of other skills that allegedly improve testing which I didn't include in this experiment that looked like they would have the same failure modes as the skills we tested.

I also ran two experiments (details not discussed here, perhaps in another 10k word post another time) on some other skills that are "official" skills that companies have to support their product. One is from a big AI lab and the other is from a "small" few billion dollar company, but in both cases, the skills made results worse, just like we saw here. Funnily enough, after these experiments, I'm actually more bullish on skills for personal use than I was before since the failure modes seem predictable and therefore fixable without a huge amount of costly experimentation. Creating a publicly released skill that's intended to be really good, work well across different models and harnesses, etc., seems like it might be hard (claude and codex seem to "want" different styles of prompting, so of course that should be true for skills as well), but just addressing the issues that cause a lot of skills to be worse than no skill for personal use seems quite doable?

Naive thoughts on skill writing

I don't know enough about skills to say how to write a good skill, but with all the skills we looked at in this post (except for the one I wrote in a minute or two) and the skills from these other two experiments, the skills seemed written like they're human tutorial instructions, in that the goal of the skill seems to be to explain how to do something. My naive thought as someone who's written all of one skill is, I'd guess that this isn't optimal when working with a model that should already have some knowledge of the topic (which was the case here and in the other experiments as well). The model is already going to have some kind of default behavior distribution, so I feel like the more natural thing to do is to give statements that will modify that behavior, not write instructions that would allow a human or non-knowledgeable agent to do the behavior at all.

One obvious problem is that we get different default behaviors from different harnesses, models, and effort levels, but throwing a bunch of text into a prompt or a skill doesn't actually change this; that's just a longer way to push the agent away from its default, with a lot of text that may do some kind of unintentional pushing. As we saw here, much of the text just gets ignored (and what gets ignored and when is of course harness, model, and effort dependent). For example, for the ECC skill, even when it was read, had most instructions ignored, and although the TDD instructions were influential (which made results worse), the instructions to do TDD were still not really followed despite them being laid out clearly.

Since what kind of prompting is effective changes enough between model releases and effort levels, for something general like "testing code well", it's not clear to me how these skills are supposed to work across so many models and efforts. Just going from GPT-5.5 to GPT-5.6 changed how I worked substantially because a number of things that worked fairly reliably with GPT-5.5 either stopped working or became much less reliable (even though, overall, the level of capability seems higher). In the same way that I don't prompt GPT-5.6 the same way I prompted GPT-5.5, I don't think I'd want to use the same skills.

I'm not sure who, other than someone at an AI lab, would actually go through the trouble of running evals on skills to see what's effective for each model and effort level and then create a portfolio of skills that are differentiated by model and effort, and I wouldn't expect AI labs to have skills that are optimized for their competitors' harnesses and models, so I don't know about things like generic "testing" skills (as noted above, quite a few testing skills that I looked at but didn't test here looked like they would have the exact same failure modes as the pre-existing skills we tested), but I could see having a few skills that work for my own use cases with the specific harness/model/efforts that I tend to reach for.

Just thinking about testing, while there are particular pitfalls that certain models fall into at certain effort levels that I want to nudge them away from, there isn't really a generic test workflow that I want to give agents that's independent of the thing being tested and the level of quality I want from the thing and the dimensions in which I want quality, so I don't think I'd want a generic test skill that lays out a set of testing steps that agents should, in general, do. Something like this goes for a lot of task that I do, which I want done in task-specific way and not a generic way. I could imagine some kind of skill that asks me questions and then emits the correct instructions to agents, but given how fast models are improving, if I'm making something for personal use, I don't think it makes sense to spend time tweaking a skill like that until it's useful. If I was working on an agentic product and wanted more people to use it, that might be a different story, but the skills I've tried have had the same failure modes as the skills we tested here, so it seems fairly easy to make a skill that turns out to not be that effective.

I could see skills being generically useful for things like teaching agents how to execute workflows or how to interact with APIs/interfaces, such as [Sawyer Hood's skill that helps agents drive a web browser](#). Since I haven't tried that skill, I'm not endorsing it, but from reading through it, it seems like the kind of thing that would work well and save me a lot of hassle when I'm trying to get an agent to drive a web browser. However, if you read the actual skill (and scripts), it has a very different style than the testing skills we tried here.

Thanks to Max Bittker, Yossi Kreinin, Em Chu, Dennis Snell, @panoramic.blue, and Jamie Brandon for comments/corrections/discussion.

P.S. I've had this note on my last handful of posts indicating that I'm trying an experiment where I write up half-baked (barely fixed/audited/cleaned up) results as quickly as possible because agents let you run experiments so quickly that I otherwise wouldn't write anything up at all. I actually ran this experiment immediately after [I ran the programming language token cost / correctness experiment](#), but I haven't had time to write this up because I wanted to write up [this creation of a regex engine with an interpreter and a native code compiler](#), this [experiment with running a forked version of ripgrep that uses the native code compiler on codex's ripgrep queries](#), and a few other things I haven't had time to write up; I've had a goal to do each of these write-ups in half an hour, but I'm still falling pretty far behind in terms of experiments I've run vs. what I've written up.

For a couple years, I was running experiments like this and just telling a few friends about curious results and then moving on without really ever talking about these things publicly. If you have opinions on these quicker (and lower quality) experiments and write-ups, let me know what you think ([X Bsky Mastodon](#))!

Appendix: some responses

[David R. MacIver said](#):

Regrettably, @danluu is right. The Hegel skill sortof sucks right now.

I think the common problem with a lot of agent skills is that agents suck at writing agent skills and also everyone (including us) uses an agent to write their skills.

A mutual friend mentioned that Maclver (later?) set up a benchmark and confirmed the deficiency in the Hegel Skill and is presumably working on either improving the skill or tweaking the Hegel comments/docs so the skill isn't necessary. If the only thing that comes out of this post is that Hegel gets an improved skill, I think that would already be pretty awesome. [As previously discussed, I think measuring and benchmarking are underrated](#), in large part because just publishing a measurement can often point a problem people didn't know existed and motivate some changes. I've had more than a few posts that have driven some kind of change just by showing where there's a gap. It's nice that this is another one of those posts.

Appendix: agent silliness

After asking an agent to do a simple lookup of something for an analysis, it exec'd a perl process that ran for 2 hours and 20 minutes before I killed it (I really need to have something that automatically catches things like this, because it's fairly common).

A subagent used perl to do a regex search over a relatively small file (44kB, 1364 lines), but the expression was degenerate with PCRE and did a combinatorially large amount of work. I tried re-running this with [the FRED regex engine we tried building in a few minutes here](#) and it finished matching in 0.7s (Rust regex was 0.6s).

You never know what's going to happen when agents are off doing things, but there are three reasons this never should've happened in the first place. First, the agent shouldn't have invoked a regex engine that can give you a combinatorial explosion like this; there's no reason not to use a safer regex engine for this (such as ripgrep at default settings). Second, the expression was wrong; the actual regex returns a uselessly large capture and doesn't do what's intended when it succeeds. Third, why did the subagent (or the harness) not automatically kill this after the subagent finished? Of course this shouldn't happen for everything a subagent runs, but agents often leave runaway processes like this lying around.

I actually have a process that goes around cleaning up after things agents leave lying around (agents that, themselves, leak memory, temporary build artifacts that consume space, etc.), but it wasn't looking for runaway perl processes. That's another one to add, but surely I'm not the only person who's run into this problem. I guess I could open source my silly tool for this, but it should become obsolete once the major harnesses fix this, so there doesn't seem to be any good reason for anyone to even pick up my thing in the first place were I to open source it.

Appendix: experimental details

In the interest of writing this quickly, I'm going to punt on this (sorry!). The distribution across conditions [wasn't fundamentally different than we saw here when we looked at how language impacts correctness and token cost](#). Somehow, this post I wanted to write up quickly in half an hour is almost 10k words, which is definitely more than half an hour of writing (10k words in half an hour would be over 300 words per minute).

One thing I'll note is, [just like with this post on programming languages](#), there were a couple of things that looked like really interesting/compelling results (at least from the standpoint of just looking at the top-level graph and seeing if anything stands out), but on looking more closely at those, they were due to an experimental error caused by giving a short prompt to agents to set up the experiment. On fixing those errors, we got a much more boring negative result, except for the part where the skills codex suggested might be useful seemed to be counterproductive.

-
1. Overall, I think the IMAP results were less interesting. I thought it might be more interesting to try since it's more of a "business logic" problem, but not as huge and expensive to run as the [pandoc eval we look at when we looked at languages](#), which costs over \$1k in API costs per run for some of the less effective

languages, and was still \$700/run in Rust to get to only 30% passing tests (note that this eval uses a much harder metric, of fraction of runs that achieve a perfect score). In the IMAP eval, only a single run achieved a perfect score ("Make no mistakes", on medium). In general, there were some bits of logic that were, apparently, too tricky for 5.6 Sol to one-shot. I suppose one way to look at it would be that Make no mistakes dominated with a 2.5% score on medium vs 0% for all other conditions. Finally, evidence that "Make no mistakes" works!

If we look at the actual results, as before, all methods weren't very effective. Even things where you might expect a good fit. For example, you might think TLA+ would do well because it's a natural fit for IMAP state and concurrency; in principle, it should be good at modeling a lot of the higher-level parts of the protocol. However, as we saw with the Zstd results, agents "decided to" do most of the implementation in Rust, then whip out whatever tool they had been told to use in the initial prompt (5 out of 80 agents actually used TLA+ before coding, but 75/80 did it the other way around). For whatever reason, agents almost always decided to use TLA+ to model a small mailbox mutation. None used TLA+ to model things like multiple observers, event queues, UIDVALIDITY and mailbox epochs, etc., where agents made many mistakes in implementation. Instead, agents modeled things like CONDSTORE and QRESYNC, where the Default condition had a more than 99.6% test pass rate, and the TLA+ agents actually had a lower pass rate despite modeling these things.

One thing that's true of all the evals I tried for this is that they're RFCs, which are highly unrealistic, but the way in which they're highly unrealistic is that the specs are much clearer, more detailed, and less ambiguous than the specs virtually all programmers give to agents when they ask an agent to implement something. I would expect that the failure modes we've seen here are the same or worse on most real-world problems.

[\[return\]](#)

2. Funnily enough, when I asked ChatGPT to fact check this, it told me that this paragraph was wrong because there are papers that show that people have used RL environments to train agents to test, and then linked to three papers that trained agents to write poor tests by training them to write unit tests like most programmers do. That's exactly the kind of thing that I would expect to lead to the kind of poor testing we see LLMs do today, where it takes a human who understands more effective test techniques to steer the agent. In multiple independent subfields where people care about correctness, folks have independently converged to a few sets of related techniques that are generally the opposite of writing small unit tests. Of course training agents to do this thing that's the opposite of what people do when they're serious about correctness isn't likely to result in good correctness.

There is some work related to RL environments and randomized testing, [such as this paper](#), but based on how ineffective models are at any of {PBT, fuzzing, randomized testing, etc.} without specific guidance, it doesn't seem that this has made it into the training of the models from the big AI labs in a serious way.

[\[return\]](#)

3. Maybe people's stated reasons for their language supremacy will become true after models get much better, but there doesn't seem to be any reason to think that should be the case. If I had to guess, I'd guess it will look more like poker, where people had all sorts of thoughts about what's most effective that were invalidated once simulation became powerful enough that computers could outplay humans in many situations. Even "modern" concepts like "range advantage", that people often say is consistent with optimal solver play, [actually don't fall out of solver data at all](#) once you look closely. They're just [nice cocktail party concepts](#) that are easy to understand and sound compelling.

It probably still makes sense to use concepts like that if we're talking about humans learning how to play poker against humans because, for casual play, humans are generally not going to want to study solver lines enough to understand what's near-optimal, but if we're talking about what coding agents are good and bad at, I don't see a reason to throw around cocktail party ideas when we can run experiments to see what works and the experiments that have been run to date don't support most of these abstract reasons X is good or bad that are bandied around.

[\[return\]](#)

[Ed Zitron's AI prediction track record →](#)

[Archive](#)

[Patreon](#)

[Mastodon](#)

[Bluesky](#)

[Twitter](#)

[RSS](#)