The Evals That Made GitHub Copilot

58 MIN · YOUTUBE · HTTPS://WWW.YOUTUBE.COM/WATCH?V=LWLXLEWRTRA
https://www.youtube.com/watch?v=LwLxlEwrtRA

SUMMARY

John Bryman and Sean Simster discuss their experiences working on GitHub Copilot, focusing on the evaluation methods that contributed to its development. They highlight the importance of various evaluation techniques, including verifiable evaluations, A/B testing, and subjective quality assessments, which helped refine the product over time.

- *John Bryman and Sean Simster share their backgrounds and roles in developing GitHub Copilot.*
- *They emphasize the critical role of evaluations in improving Copilot's code completion and chat functionalities.*
- *Four main evaluation types are discussed: verifiable evaluations, A/B testing, subjective quality judgments (LLM as Judge), and algorithmic evaluations for tool use.*
- *The team learned that users preferred shorter, incremental code suggestions rather than long, complete solutions.*
- *They highlight the importance of using a mix of engineering and data science skills for effective evaluations.*
- *The conversation reveals the evolution of evaluation methods in response to user feedback and product changes.*
- *They stress the need for evaluations to be adaptable and integrated into the product development process without hindering innovation.*

All right, then let's kick it off. this talk is the eval that made GitHub copilot. and just to get things started, we'll go through quick intro to to so that you guys can know, who you're talking to, my name is John Bryman. very nice to make all your acquaintance. I worked at GitHub starting in 2019. but that was that was well before co-pilot was a thing. And so my my first thing at GitHub was search at at that time search was the world to me. It's it's search is my hammer and the the world is a nail. but over time I after I worked on the co-pilot I worked on the code search team I moved on to data science and then worked with HAML there actually it's the first time we got to meet and then from there moved into co-pilot because I had kind of a data background and an engineering background so it it was a it was a good fit for me my projects while I was there was code completions and then after oh about six seven months in that I moved into the chat but it was web chat so it's not the ID chat since then about a year ago I stepped away finally from GitHub to become an independent consultant following again in Hamilton's footsteps and I I wrote a book that was published last November called prompt engineering for LLMs also since he's given me the bully pull pit and Hamill if you wouldn't and like just posting a link from this that would be awesome. I do have a Maven course coming up soon about building LLM applications. so the idea is like we're getting lots of buzzwords and there's like a million different ways that you can build all the stuff, but when you get right down to it, there's a few underlying principles and if you know them, you can rearrange them to make to make an amazing diversity of things. So that's kind of the the that Sean, do you want to come in here?

Yeah. Hi, I posted John's course in the chat. Thanks. Hi, I'm Sean Simster. worked at GitHub for 3 years starting just after Copilot had come out as in beta. before that I was at Google working on some machine learning stuff,

some search engine stuff. and yeah, so at GitHub I took over some of the offline evaluation work building out we'll talk about that the frameworks that we're building out for offline evaluation and then I prototyped the first version of co-pilot chat and that led to a whole bunch of new evaluation techniques as well and so I'll talk about that later. and then yeah, just recently I've been working on some some personal research projects, explain prompt, ground rules. just really enjoying building with LLMs and and building developer tools and and thank you for John for inviting me to to reminisce about our our times on GitHub Copilot.

Yeah, this is really exciting. I mean from my perspective just you know I was around when GitHub copilot first started and when it first that effort first kicked off I was pretty skeptical you know they just partnered with OpenAI and it seemed like you know the first project they jumped into right away there's a whole like menu of different things one was you know code search which I thought okay that one seems like you could tackle it. you know, no one really ever had seen like a language model like that before. The one that in OpenAI privately shared with GitHub and you know I was really skeptical like the first version did not work well at all. was like barely functional if that but it's really like because of the evals that I saw the product improve like you could see like every other day all the engineers would make progress against evals and by the time you know first it was like vibe based eval like people on the GitHub team we would be using it be like okay this is cool and after a certain point like the evals you know like we would be moving the metrics on evals and after you know that provided a lot of signal into what wasn't and was was working and seemed like it was really unlocked the product. So I think from my perspective is like very critical. So and but I don't know the whole story. Y'all know the whole story cuz y'all have been there. Y'all worked on it for quite a long time. So we we know the whole story up until the point where we left and and it's and it's a really interesting story. So I guess let's just let's dive in.

first off, obviously quick slide, but make sure we have a shared foundation. Copilot, you guys all know what it is, right? Yeah. You you type in you type in some code and it automatically completes stuff. I think this is probably the one of the most common completions that I got while writing comments. it was always insulting my work. this was the first part that grabbed everyone's attention, but it wasn't the only part. both Sean and I worked on the code completions.

but Shawn was actually one of the very first people to work on the chat component to it. and here you can see kind of a screenshot of how that's evolved and to what it is today. And that's not the only part that there was. There's it's probably you're less familiar with it. but this is the part that I moved to after code completions. there is an online component that you still have access today. I think it's free to everyone.

So you can ask questions about your repo and stuff like that. but as Hamill was saying, evaluations were really a core piece of of what made Copilot and so let's dig into that. Now this is this kind of gets into how my own thinking has shaped since working at Copilot and I'm I'm going to attend Hamilton and Trader's course. so I'm excited to be corrected but this is the way I I kind of mentally cut up the the evaluations that that we were doing at GitHub. On the easy end of the spectrum, you have something that's like algorithmic things that you get the code completion and it's easy to just like check like an equalities check for what whatever you're testing. So does you know if I'm extracting structured content from particular website does this website lead to exactly this

output and you can have you know a bunch of those is like tests that you use in a val is does the response structure adhere to JSON and in particular the the schema that I'm looking at is is the length of the response within limit or does you know maybe it goes it's it's too verbose or something or too piffy.

is the code all in? Any of these things are things that are just really easy to write a normal programming check for. At the other end of the spectrum, we have LLM is judge. which is hugely important because so so much of the work that we're doing is subjective in nature and you kind of need a human to do it but humans don't scale. So element is judge is anything that is subjective. you ask you know what is a preference for you know this completion versus like the the canned response. am I hallucinating or is my response you know actually accurately using the the context provided those type things more more subjective things that you can't write an algorithm for.

Now, co-pilot was at this kind of interesting in between that I've come to call verifiable evaluations. There are things that it's like it's algorithmic in some respect, but it's not just so easy as looking at the output. it's things like I'm generating code and I want to make sure the code compiles. I want to make sure that the code passes unit test. Maybe it's SQL and I want to make sure that this the I don't care what query it writes but it needs to have the right answers come back so it can have some degree of freedom with that. So all these things the the code completion you can't immediately look at it but you can run it evaluate it somehow in a way that's more complicated than just the algorithm. And finally underlying everything is AB testing. there's nothing particularly special about this for for for a co-pilot, but it it was just an important part of our history. All right, so we're going to walk through each of these and tell you the piece of the puzzle and it lived itself out at at GitHub. So the first thing was exactly what HML was referring to a moment ago.

We called it harness lib. It was a a verifiable code completion evaluation and the idea was you know we we've got this magic box certainly was magic at a time that that will look at your code and write hopefully working code. so as we were working on making the completions better, pulling in more context data that was you know we hoped would be helpful, we had to test the changes before we go live with them. So this is an offline tool that that we did for this and it was one of the first things I was intro in introduced to when I started with co-pilot. The idea is simple but you got to admit it's a really cool idea. First off, the we would gather test samples. And here's the the rundown of how that works.

basically, you grab, you know, we we've got we've got all the repos. So, you grab a large set of open source repos. Specifically, in our case, Python and JavaScript. we had to narrow it down. you run their test. That's basically why we had to narrow it down. We could run Python and JavaScript tests pretty easily. you run their tests and you keep the repos where all the tests pass. So it's kind of a funnel. of those you use code coverage tools to associate the functions with your unit test. You had to make sure that you're actually testing something. And then of those functions that had some coverage, we we've kind of forgotten the exact criteria, but basically the candidate functions had to have one unit test had to with at least some percentage of its lines covered. It had to have you know no more than blah number of lines otherwise you know it was too it was too hard of a task for you know early days copilot and it needed a dock string. So we're looking for something that is the the function function name the signature the doc string and the context around this piece of code.

All those became candidate tests for evaluation. and then running it. You kind of see what we would do. U we would grab some of these one of these candidate functions kind of you know melon baller scoop out the the implementation of it generate the contents again and run the unit test again. And if we if we do this across lots and lots of functions across lots and lots of repos then we can start to get an idea of directionality like as a baseline we know what version X of of copilot did. It wouldn't succeed at all and I forgot what it was up to. It was like 40 or 50%. It wasn't even a huge number back then. but we talk a little bit about if I recall y'all selected these repos to be like you know you had a way of selecting or filtering ones are like higher quality and stuff like that. Can you talk a little bit about about that? maybe maybe I touch on to it a little bit in this lessons learned thing right here. But Sean's going to cut in here in a second. see we got these cute little pictures and he can probably address that a little bit too if I miss any points. but you know as far as quality is concerned that was one of the things that was kind of a lesson learned. we we had to make sure that the content itself wasn't trained into the model. We would get these models from OpenAI and guess what they were training the models on all all the code at at GitHub. So, when we had closer relations early on with, OpenAI because we were one of their first main main customers, we we could get a list of what went into the training and make sure that we had something that was sprained after that date. it's also important to make sure that your repos are consistent with the traffic that you expect in production. And I think we never quite I mean it was it was always very sub sub subjective of this but I think we always knew that we we wished it was a little bit more consistent. some of the models because they weren't in the training set were the newer models.

Sorry some some of the code bases that weren't in the training data were newer code bases. And guess what? Newer code bases tend to be smaller tend to be certain type of thing. And so like a lot of our code bases were a little bit smaller than what you know you that see work work done in production. That's really interesting about the relationship with OpenAI. You mentioned like in the beginning it seemed really close like every week there was calls with Greg Brockman in in Ilia. I remember I would get on conference calls with just them and a few other people. It seemed like super close. Was it so it didn't stay that way? It wasn't like that like necessarily throughout the entire time. Yeah, it's interesting.

And by the way, Sean, you feel a free to cut in here at any point that you want. I you you had an earlier window than I did, even though we were working at GitHub at the same time, but you were in the middle of it. At that point, Sean got to see about a year before I I did. And then I think both me and Sean left at about a similar time last middle of last year. So yeah, at some point we we were the we were the big the big kid in the kitty pool, but eventually you know chat GBT landed and it was very important for them to the do the best thing for their own product. maybe a couple we've got a lot of ground to cover. So maybe just a couple more tidbits here. and then we've got the three other types of of vowels to go through. but a lesson that I think I learned, and the reason I've got our pictures here is because Sean almost learned the opposite lesson, and he makes a really great point, but I always felt like our test harness should have been more should have basically been a headless headless BS code, so we could absolutely test every single component of our codebase as it changed. And I still agree with myself. that ends up being really good for like integration tests to make sure that you're keeping the evaluations u high and now maybe Sean you can pop in here but I like the points that you're making about the flexibility of of the harness.

Yeah. Yeah. just there was this kind of these different phases of product development where you know

sometimes you've got a new model with new capabilities coming out and you're trying to imagine like oh how would people use this and how can we evaluate something that people have not even had a chance a chance to try yet. and so in that sense you can't just wrap the existing product in an evaluation harness and have that be the only way to test things. you do need this more experimental system where you can kind of prototype out new ways of using the AI, new ways of using the product.

and but then those are always less reliable. And so once you launch that, you kind of have to go back into making it more more reliable, more more easy to evaluate. So that covers harness lib. the next big chunk is so harness lib was verifiable test and it was probably the biggest chunk of our evaluations and also definitely one of the coolest. AB testing is what we did whenever it was time to ship our changes online.

the offline test that we had would would you know hopefully you know limit limit the exposure to you know shipping something that was surprisingly bad. And I I think it did a a good job of that and helped us really flesh out what was going to be a good product to ship. But once we finally did ship it, we had to make sure that the changes are appropriate that you know we'll start them start our users on like 10% of the traffic and kind of wrap it up. Keep keep an eye on how the product is doing and see if anything is changing.

and a really important lesson for us is it came from basically how how these things were set up. we had just a very few key metrics. and to to the best of our memory it's these three items on the screen here. complete completion acceptance rate. So you know a user gets the ghost text on the screen. Do they accept it or not? So how often do they accept it? characters retained.

a lot of times especially for users like me, I would like mindlessly accept everything and then edit. That was kind of the pattern that I used. So we also had to incorporate that type of user, that type of use case into it and make sure that you know people weren't zombie like accepting everything and then you know being did not liking any of it and having to you know go back and and and change it. And the last thing is kind of an obvious one is is latency. If we come come out with a recommended piece of code that is too late, then it's that's just bad for everything. Now, beyond the key metrics, we had several guardrail metrics. and by several, I mean high tens, maybe maybe even in low low in the the 100.

But it there were a few of these were like variations of these things above like characters retained after a certain number of minutes. but you know, anything that would give us a way of of kind of measuring how how how much it changed to see if if there's anything weird going on in the side. so I have a question about guardrails. Like a lot of times guard rails, people have guardrails, they're like binary past fails, like really catastrophic errors or something they want to like prevent against. Was there some of that too or something like that too or this? Was it mostly like these metrics you keep an eye on with the like drifting wildly?

Yeah, mostly we were looking at the key metrics that made the conversations really easy. just so we would focus on just that those few simple things. guardrails we didn't have like just a hard fail if something went wrong with them. they were they were part of the conversation if something was really going wrong or you know if we were trying to figure understand the metrics the guardrails were just a useful side conversation to to diagnose and

figure out what what was going on with the key metrics. So they all kind of fed back into that when in the cases where like let's say those key metrics like acceptance rate, latency, any everything we're like improving let's say and but then the other metrics were sort of going astray but did that ever happen and like did you do a bunch of analysis like data sciency type of work to like figure out why and how was that how was that conversation like surfaced let's say so stuff like that would happen occasionally and I you know I'm going to be super fuzzy on this because I wasn't really in the middle of this discussion but we did have like weird coupling in several of the metrics.

it's a second bullet here. Weird trade-offs. So, like if sometimes you would maximize one of the metrics and and it would just tank another metric and you'd have to think really hard about why those two were were trading off. like you know if if you really optimize for acceptance rates, well, one silly thing that would happen is like you have a bunch of little tiny completions and so the experience was was garbage when that happened. but you'd have to like you'd have to really dig dig around in some of the other metrics to figure out, you know, why a sudden departure happened from whatever your baseline was.

How did that how did that process that detective work happen? Was it like a lot of data sciency type of work? Were you kind of doing a lot of data exploration or you know like Yeah. How did it who did that kind of ex like detective work? How did that how did that work? Yeah. So, as part of like these these AB experiments, you know, we we would ship out new new models, new prompt crafting techniques and then, you know, we would hope to see, you know, the key metrics going up. but then sometimes we would see something like a guardrail metric like number of lines generated would, you know, average number of lines generated would jump in some some weird direction either up or down. could be kind of alarming. and so then there were different scales of that is like you know we'd have the prompt engineering folks in the meeting and they might say oh yeah we you know we expect that because we improve the prompt in this way and it's giving more targeted suggestions so it's actually okay if the number of lines goes down because they're they're higher quality lines. and then like you said there are other times where everyone's like we don't know why the number of lines have gone up. this is concerning enough that it's worth, you know, delaying the launch for a week while someone digs into the logs and actually figures out what's going on. and then it becomes a data science project.

and so luckily, yeah, we had had a whole bunch of people on the co-pilot of Alice's team that had that skill set who could kind of go off and and dig through the data and figure out what was going on, come back the next week for ship room and say, "All right, we looked at it. It's okay. Here's why. Let's let's move forward." Hey Sean, you're you're doing the talking right now. I think this is one of the most interesting parts and you were right in the middle of this LLM is judge stuff. Do you want to take this next section? Just tell me when to progress the slides. Yeah. Yeah. So so yeah like we felt we had a really really good system with these this offline harness libal where we could run different prompt crafting experiments or you know fine-tune models and see how that affects the product in an offline setting. and then we could also kind of ground that in online AB experiments and make sure that you know the changes actually expected the way actually played out the way that we expected with with our real user base. so we're feeling pretty good about that and then chat GPT came out and it kind of changed how we thought about a val and I think it was obvious from the beginning that these these chat models were going to be a big deal. but we didn't quite realize at the time, you know, how much how much it was going to change our vows. and so yeah, this is how we kind of backed our way into LLM as Judge, I

guess.

and kind of started looking more at like subjective quality judgments instead of these purely kind of executionbased you know, does the code work or not? which was kind of one of the key ingredients to getting the co-pilot completion product to that point. So at first yeah thank you thank you John. At first I think when we looked at what people were doing with chat GPT and like kind of conversational programming I think we were calling it at the time it kind of looked like what we're already doing with copilot. you know, people would ask it, you know, make this code cleaner or, you know, write some unit tests for me or help me fix this bug and then it would, you know, go through multiple steps of conversation, but there still be some code output.

and so I I started building out the first version of co-pilot chat and thinking about how I would evaluate that so we could we could launch something eventually. And I started to realize like even though there was code coming out of chat, what we needed to evaluate wasn't just the code that was coming out of chat. What we needed to evaluate was the actual conversation, right? And people obviously cared about getting good code that came out, but they also wanted to feel like the conversation was productive, like the the assistant was listening to them and giving them good feedback and that sort of thing. And then you know we also had use cases in chat where there wasn't code coming out right so someone would say like explain to me how this code works or like you know why why is it using this API and not that API and those are really helpful answers but there's no code that we can run. and so I starting to think like maybe the only way to evaluate these is just to have like side by sides, right? Like you have your baseline like here's here's how chatt handles it and then then here's how our agent handles it. Which one is better? you just put it in front of a human and you ask them to, you know, to rate, you know, which one is better, which one is worse.

but obviously that's slow and it's expensive to you know hire people and and get a lot of those those kind of judgments. and you have to come up with a lot of guidelines too to help people. You can't just say which one is better. You have to explain what better means and they kind of read through and and judge which parts of it. and then so I I set up a a basic version of that in label studio so I could just kind of test the flow and and I found it difficult just to to say which one was better. and so I tried to make my life easier by passing it into GP4 and saying, "Okay, here's my grading criteria. Tell me which one you think is better." And then now I can just click through in label studio and and kind of judge the judge and say, you know, did did GPT4 make the right call on which one is better or worse? but even that was challenging.

you know, this this was this was years ago when when models weren't as as good at at following the system instructions. And so sometimes it would get distracted and it would say, you know, this the code on this conversation is more maintainable because, you know, they put their braces on a new line, whereas this one the braces are on the same line. So someone would have to go in and adjust that. And I I was like, no, that's not what I'm evaluating at all. You know, I'm I'm trying to ask like, did it write better unit tests? and so I started going in and actually just writing out in like bullet points what explicitly what I wanted it to evaluate. Like does it have a test for, you know, passing in null values? Does it have a test for passing in an empty list? and finally that got it to the point where where I could use this LM as judge and it would stick to my, you know, very granular list of things that I wanted it to evaluate on and give me good results and I could kind of click through quickly and

double check these. and then once I, you know, verified that it worked for for one model, then that could kind of become the baseline and I would only have to compare the new conversation to see if it matched the baseline. so that was kind of how we we backed our way into this this LMS judge. It wasn't something that we decided that we needed at the beginning.

It was more just kind of realizing that harness lip wasn't going to be able to give us what we needed because copilot chat was in fact like a whole new product. It wasn't just the obvious next step for code completion. now were you able to measure like somehow the alignment between the judge and yourself or other people? Like how did you how did you go about instilling trust in the judge?

I would have loved to have done a much a much more extensive study proving that that our judgments were were being aligned with the judge. it it was a crazy time to be working on AI and everyone was shipping chat products as quickly as possible. and so, yeah, the the first version of Copilot Chat went out with, basically our gut feelings of, you know, here's what we think users will do with this product and here is an offline data set that kind of models that gut instinct. and then you know once we launch it then we can we can actually start to collect telemetry and see how users are using it and then adjust our offline benchmarks. But the first version was really just like a bet on how we thought this conversational programming would evolve.

So on balance like would you say the judge was helpful like more helpful than than wrong? Like you were able to see the outputs of the judge and find you know use case find examples you needed to work on or correct things in you know in the product. Yeah absolutely. So once once we kind of made the the adjustment to like micromanaging the judge where we're saying like you know is there a test for this? Is there a test for that?

Is there a test for that? Or like does the does the answer mention that you have to do this? Does the answer refer to this specific, you know, part of the API? then it it it was much more reliable. And I think that that helped people get over also this this weirdness with LLM as judge where it's like, okay, you're asking the LM to write some code for you and then you're asking the same LLM or like a slightly different LLM whether it did a good job. Like, isn't it always just going to say, yeah, I did a great job. but the the intuition here is that like that generative task of starting from, you know, the user's question and actually solving it is a much harder task than giving it this really like detailed list of requirements and then just getting it to check each one off. So once we got to that point, yeah, we were pretty confident that it was giving us good results. I mean obviously there's still sometimes would be hallucinations where you know you ask it to check it off on the list and it adds three more items to the list you know and people always like to ask okay when you're doing all this work on the LM as a judge prompt did anything from that prompt make it over into the that the code completion model or you know did you end up prompting another model with elements from the rubric of the judge? Did you end up, you know, making the main prompt more specific based upon like, you know, Yeah. Like just curious like did it kind of feed into each other at all?

I don't remember prompt model. Maybe it was just like a code completion model that was just, you know, but I'm just curious if there's anything like that going on. I don't remember any specific cases where it did. I mean like I said we we were much more familiar and confident with how we were evaluating the code completion

product at that point. and so much of the co copilot autocomplete was like can we get the right context in the prompt. and so and and we were also fine-tuning models but it wasn't like crafting the right system prompt the same way we were doing with with co-pilot chat. I know that we we had done a number of experiments with writing unit tests in in copilot completion and then we we did you know other experiments in chat with writing unit tests. but I don't think one ended up feeding into the other. it it ended up just mostly happening in in chat.

but one one of the interesting lessons that that we learned coming out of this was or that I learned personally building this was that there ended up being these two very different personas on the team for evaluation. there were folks who you know we're trying to ship things. We're trying to keep things from breaking. We're trying to prevent regressions and they would come in and they would want to run via Val. and so we would have to have, you know, really clean, reliable ways to run like via val, get a number out, compare it to the previous number, and make a shipping decision. and at the same time, there are a whole bunch of folks who wanted to pull everything apart into Lego blocks and reconfigure them and say, "What if the product worked this way? What if we used this model? What if we did that?" and so it was much more experimental and and trying new things that hadn't been built before.

and so yeah, it wasn't always clear that that should be all one eval pipeline. And I think over time we developed more and more evaluation tools, evaluation pipelines, and it felt a little bit overwhelming to some people to have more than one way to do a val. but I I believe it was necessary because there's just more than more than one type of a val. There's these kind of unit testing, experimental things, and the more regression testing, you know, shift decision things. What's like one of the more experimental things where you kind of took a took it apart, the modular pieces and whatever. That sounds really interesting.

actually John John was doing some of this stuff with like trying to Yeah, originally we just kind of chop up code by line numbers and shove it into the prompt. and John was trying to be a little bit smarter about that and and find ways to to treat code as code. Yeah, it's I it's hard because it's it's so I' I've forgotten a lot of the details, but I remember we used to have when we were running the evals, it was basically a notebook and in the notebook we had a pattern for like a typical harness a typical harness lial with you know where to get the data from and how to how to set up these individual tests. but and it was and it was like Lego blocks. You could just use the piece that you needed. but it's kind of nice because if if you make if you were to make something that fit in the same place that the old Lego LEGO block went, you could, you know, kind of mix and match and like Sean is saying, you could prepare for evaluation of a product that doesn't even exist yet rather than do like a regression test against something that's more rigid and frozen. And so I've I've forgotten the details. it's going to be uninteresting but some of some of the work I did was whereas initially we were like melon balls scooping out the the full contents of our candidate functions a lot of our important completions were the oneline completions and so I just made another way of harvesting candidates as the Lego block where the candidates were the function with a piece of the code missing or like the bottom half of the code missing something like that where it looked like you were editing a block instead of editing the you know here's the new function which is a little bit different than what our users typically did. So yeah, it was re it was really neat. the the harness li just how modular everything. you mentioned that some of these evals run in notebooks. Did those not do you run the notebooks programmatically? Was it more like ad hoc? You kind of open a notebook and run each cell. How did how that's really interesting I think to a lot of people. You you could have done it either way. and so like for the

more regression testing like flow, I guess these things were pretty pretty canned. it was in a Python script and you you could just like run the thing.

the but as especially as you're getting you know I Sean had been working on this for a year by the time that I I would I was introduced to it and to get intro to get an understanding of how these things work. It was just so easy to go to a notebook which was the same stuff as in this, you know, can script, but you could see how each each bit of it worked and you could, you know, it's a notebook. You could muck around with it and and play with the Lego blocks, change out the valves and and kind of get an idea for what you could do in the future. It was kind of neat. Yeah. Yeah.

A lot of like quick and easy idea or experiment ideas started off in a notebook and then would kind of move into just a regular Python script. So you could have an aval that that people could run easily. and then some of those would move actually into like u you know Azure pipelines that would just run automatically. but notebook was mostly just for experimentation. Yep. A quick note on time. we're 15 minutes past time. there's a couple of slides left. No, we can keep going.

This should have been an hour. I don't know why I made it 30 minutes is that's my mistake. So whatever. Yeah, it's it's cuz we just had me be in the beginning and then we have a whole year's worth of knowledge prior to me that joined after. well, so let's keep going and then u but we we can obviously stick around for a bit. The conversation's been fantastic so far. all right, so the last little bit, you remember in the the first slide we had the the four things we used.

this is just kind of a footnote to be perfectly honest. It's it's it's a a a much smaller type of evaluation. It might as far as I know it might have grown and become a really important thing after I left, but it was coming into existence about the time that I left and I probably used it a solid twice. But it is it is a demonstration of the third the fourth type of evaluation in in my pipe hierarchy algorithmic evaluation. And what we were doing with this is you know as the the chat models you know gave way to chat with functions you know these more interesting capabilities it was important to double check that the function was making the correct pool calls or sorry that the the LLM was making the correct tool calls and the way you could do that is you you provide the context user message you know all the tools that the the the model is going to have access to. How many tools approximately did Copilot have roughly at the time?

I could probably speak to the web interface and Sean could probably speak to the VS Code, but it's kind of funny. It's this is a hack. You you can go to the web interface in GitHub right now homework assignment and you can ask it what its tools are and it'll tell you. so we don't we don't like hide that very well if we're supposed to, but it didn't matter. And I think there was probably when I was there, just a handful, like five, six. I think since then, it's kind of moved up to about about double that, but I haven't checked in a bit. so not just a whole ton, u, but more than more than a couple.

yeah, but the the cool the cool thing you could do is you know given given that expected kind of trial input, you know what the output should be and you can you can besides just doing like you know checking for accuracy by

seeing if this is an exact match for the tool call, you could do cool stuff like you do confusion matrices like this. I mean, it's a toy problem I showed you on the right, but you know, you you'd say, "Here's all the the tools that my model has access to, and we call we attempted to call, you know, them this number of times. Here's what actually got called instead." And so that's that that provides you with a lot of good information there. You can say, you know, if you got any tools that get called that get mis mistook a lot, you you can kind of step back and say, oh, you know, I need to figure out how to make these things more distinct that your tools ideally need to partition the space that they're they're operating in so that there isn't, you know, silly overlaps that will confuse the model.

So, that's just a just kind of a a tip. I'm glad to see the confusion matrix actually. in an earlier lightning lesson, Brian went over very similar confusion matrix for a evaluating agents like agent handoffs. So, yeah, this is pretty Yeah, right. It's pretty cool. It makes sense. Yeah, you can see how often they Yeah, it's really important, right? Because if if you're that just a very similar task. if you're handing it off to the wrong agent, then probably what it means is is there's just too much overlap with your agents and the orchestrator model doesn't know how to decide between them. So yeah, confusion matrices are are great for that type of thing. All right, so we're back to this slide again and we have our bases covered. verifiable vowels was harness code with a code completion.

Ellen's judge was that really interesting conversation with Shawn then that revolved around chat completion. ship room was our AB testing and that's you know whenever we change the model or you know tokenization or the prompt or anything like that we made sure that we we we put that through AB test and then finally just a footnote but tool use evaluations was was an example of where we're also using algorithmic evaluations. So, that is it for the content. Thank you guys for sticking around for 22 minutes longer than we' set this up for, but I guess we can still Yeah, there's some there's some questions if you don't mind. one from Jesse is asking, "What are your approaches with respect to caching? Like if the inputs, the prompts, the model is the same. do you do you somehow leverage caching you know with all of these tests like you're running them in mass you know how did that I remember I'll I'll pump this over to Sean shortly but I remember wishing that we had done something more intelligent with with with caching for harness lib because some of those tests took a while to run and I I felt like there was still room to do more caching but Sean do you do you remember some of the specific It's a horrible thing I'm doing to you.

Yeah. No, no, I I don't remember any any specific like prompt response caching. I mean, obviously the model hosting had had some optimizations to to speed up you know, reusing the the same the same context, that sort of thing. but most of the caching approaches we we had in in harness were just sort of the the preparing the code you know so you know John talked about how you know we would select these these repositories and then we would go through and run the tests and then we would parse out the function bodies and that sort of thing.

You don't have to do that every time you run the eval harness, right? And so, actually touching on another question we have here from Pastor, you know, we could run all of that analysis once for this fixed set of repos, store the list of functions that we had and the offsets into those files because, you know, they weren't changing in a SQL database and then just refer to that SQL database every time we had to rerun the the harness. And then did you build like your own dashboards and some kind of other alerting like what kind of thing did you build on

top of that eval data that you know every time you run tests you get all these all this telemetry?

I'm thinking did you have something on top of that that you use? So for the offline eval you know this was just something that we were mostly kicking off manually and then we would you know build a pretty simple little well I don't even know if you could call it a scorecard but just a list of metrics you know showing pass rates so there wasn't there wasn't a lot of of dashboards there but for the the shiproom decisions there there was a lot more and so the teams we worked with at Azure had already built a lot more tooling around you know displaying the results of of the AB experiments and kind of making sound decisions based on you know how much data we'd collected and whether they were you know representative samples and that sort of thing and so that that was really helpful to have that that expertise brought in into the product.

did you find that, you know, when it came to troubleshooting errors that you saw or kind of digging into the results of the evals, was it, you know, was it necessary to hand things off to a separate data science team a or what were like engineers like able to figure things out or what's your intuition on like can this be done by one set of skills or do you need a distinct set of skills to do it. you know, that that's one of the questions that always comes up with evals because a lot of people are doing eval there people of different stripes doing eval. There's like, you know, a lot of software engineers with no data science experience. So I'm just curious like how did it unfold at when you one thing that I've often heard you say Haml is is that and I started parodying it myself is that whenever you get someone to evaluate these things you get the people that are actually good at this stuff. You get you know content curator that know that what the heck they're they're talking about when they're reviewing rag or you know whatever it is. In this case it was great because I was an engineer. I was using this stuff and when I and and so and the the harness li was really engineering heavy test bed. I was using you know Python notebook a lot of times when I was just running kicking these things off. and I was the best and I was the one working on the product that was running through the valves.

so maybe it's, you know, there's so many fortunate things about this very spec specific niche of codegen that that doesn't exist in other places where you really do have to have the other person that's not the engineer. But a a fortunate thing that we had is like we could do it all. And it made it I think a lot easier in that circumstance to say like this about went sideways, but I I think I know what's happening. and you didn't have to have that communication and the breakdown. What about what about like for situations where code completions for programming languages that you are not familiar with? I don't know which programming language you're not as familiar with. Like maybe you're not comfortable with Haskell. Maybe you're not comfortable with maybe Haskell is like a little bit lower resource but like I don't know maybe not comfortable with C something maybe you are but pick a language like how would you would you feel okay with like trying to debug like hey what's going on with this code completion like why is it not you know working the way I want it to.

You can take this Sean or me? I mean I I was surprised when we looked at the internal numbers and saw you know hey we're running harnesslib on JavaScript and Python and we're getting you know roughly similar acceptance rates for you know hasll and you know visual basic or whatever people are writing in this thing and it's like it was really encouraging to see that the val that we were doing translated over to other languages that we weren't necessarily evaluating on or comfortable with. I'm not familiar with examples where we we had to do

a deep dive into, you know, one of those those less familiar languages. and I think a big part of that is like anytime you're trying to diagnose one of these problems, especially in production, you want to kind of go back and be like, okay, what's our baseline? You know, where is the point in the past where this worked? and so you're always kind of building on the the eval runs that you've done in the past. And so if you don't have those evals for like hey how did HA hasll work in copilot a year ago then it's kind of hard to diagnose that in the present and so okay yeah that makes sense. It sounds like the the high resource languages Python JavaScript are a good proxy sounds like for even the low resource ones surprisingly so. Yeah. okay I'm going to put you a little bit on the on the spot here with the question but maybe it'll work out. What is like a surprising thing that y'all learned?

Like if you can think of any example that you learned from doing evals like you know that turned into a improvement of some kind like you know that like something counterintuitive or something that you remember that sticks out in your mind. I well just kind of go back to to Sean's part honestly with the ele judge. I really this is well after the fact but in our conversation to to put this talk together I thought it was just neat to hear him walk through the learning process that necessitated L judge. We I mean this process has been done at several places at this point but what you kind of saw right then was the birth of L judge like this needs humans humans don't scale here's how like we break this down into something that an that is easier for humans we like have it list its criteria so we can digest it why not just turn that over to an LLM it's easier to do like you know true or false that's that's gonna be a lot easier to to deal with than like something that suggest subjective as as a human. So I thought that was kind of a neat lesson that that came out of everything.

Yeah. I think I think one of the the things I remember is being kind of interesting is you know we started off with this harness that was ripping the bodies out of functions and trying to synthesize them again. and also this intuition like John was saying of like, you know, the more characters that are accepted and retained in the code that that that's a signal that we did something good. and I I remember we kind of discovered over time that users were getting frustrated when Copilot was trying to write too much code for them at at one time. and they would, you know, reluctantly accept it and then delete, you know, the last half of it and then get a better completion for the the second half of it. and so I I remember we we did a number of experiments and and changes to the prompt to actually generate shorter completions or even just, you know, chop off the completion and let the user kind of naturally tab their way through the the solution instead of trying to to oneshot it. and and so that was something that was kind of counter to our our initial intuition of like copilot should write as much code as possible for you and then it should all be you know good quality code on the first try and this was more like no people actually want to you know incrementally build up a good solution.

Makes sense. If y'all were going to build a product today, like some other kind of AI product, where would you still do evals? And like where would you do it? Like at what point would you start thinking about it if if if at all? I don't want to lead the answer. Well, it it is it is interesting because I the good the good student answer is well, you should have a vow always from from day one. And I bet Sean comes closer to that than I would. I I think it's super important to have a vow if you have any type of product, but there is a point there is a point in time to introduce them. A lot of a lot of times when you're just really early on trying to figure out what the product does, good old vibe vibe check is is is we'll get you a long way.

You can say is this working at all? Is you know what do I need to do with this? And and I I would be interested honestly Hamilton and hearing your opinion as I go through your course here in a few weeks about like what are the limits of you know how much evaluation you do because you know we've all seen companies that are like we're supposed to have a valve then they go a little bit whole hog into that and now they're in a val company but they're not really measuring the right right thing. So, I I I feel like there's it's it's fairly early on, but it's not at the very beginning. At some point in the beginning, you need to be just making prototypes and seeing what works.

Yeah. Yeah. Totally agree. yeah, like I say, like the purpose of of building a vow is not to to have a vow, it's to to build products, right? And so that should still be the the number one priority. you shouldn't not build the product because you can't think of the eval yet. and so you should be listening to your users and and doing these AB experiments to understand what your users want and then going out and building those products. but then thinking in the back of your mind, okay, eventually this product is going to be successful. How are we going to keep it from aggressing? How are we going to evaluate it? but it shouldn't it shouldn't block you from doing something wild and crazy and and different from your competitors. That makes sense. I agree with that. And what what we'll teach in the course is you should you know you should build evals after you've built a prototype. you don't need to start it too early. You need to have something first. And even while you're doing evals like if you notice an error in the process like you don't there's a tension on like when do you write an email and when you don't. It's a judgment call on if there's an ROI to writing that email. Some some errors you'll find like okay you know exactly how to fix it. just go fix it. Like, you know, you don't necessarily need to write an email for every single thing.

but yeah, that makes sense. That kind of experience. I mean, on that harness or on the AB experiment, so many of our guardrail metrics, I believe, came out of, you know, things that broke in the past, right? It was like this accumulation of like, oh, we tried something, it broke, someone dug spent a week digging into it and then found this this metric after the fact that would have, you know, caught it earlier. and so a a lot of that stuff you kind of only learned by building it. I have a really fun question for you before I wrap up. That LM as a judge prompt, do you think like that would be that would make for good cursor rule like that the one that you have? have you it seems interesting, right? Like if you have all these bulleted points of like, hey, this is good code, this is bad code, whatever. it seems like the perfect prompt for yourself in a way.

Yeah, maybe. I mean it's you always get to cheat a little bit when you're building your own eval set and kind of say like here's what I think programming is and here's you know the tasks that are representative of what our users will be doing and I'll only evaluate those tasks. that the scary thing whenever you're doing kind of especially like a chatbased interface it's like it's just wide open and people can ask anything and do anything with it. And so, yeah, I I would hope that the the the prompt that I did for for LMS Judge would generalize to more just kind of general code quality. but I also got to kind of pick and choose which use cases I covered and and tailor it to those.

Makes sense. I kind of want to see this. It was pretty obvious. I want to see the problem. Nice try. Nice try. Yeah. no, thank you so much. This is a really great conversation. It was really interesting. So, thanks for coming on and teaching everybody about how you all build evals. Yeah. Thank you for having us. This has been great. Yeah.