AI Engineering at Jane Street - John Crepezzi

16 MIN · YOUTUBE · HTTPS://WWW.YOUTUBE.COM/WATCH?V=0ML7ZLMDCL4
https://www.youtube.com/watch?v=0ML7ZLMdcl4

SUMMARY

John Kzi from Jane Street discusses the integration of large language models (LLMs) into their development processes, specifically focusing on their unique use of the OCaml programming language. He outlines the challenges and innovations in training custom models, building editor integrations, and creating a robust evaluation system to enhance developer productivity.

- *Jane Street utilizes OCaml, which complicates the adoption of off-the-shelf AI tools due to its obscurity and specialized applications.*
- *The company has developed custom models to generate code diffs based on developer prompts, requiring tailored training data.*
- *Training data is sourced from unique methods like workspace snapshotting, capturing developer actions and build statuses.*
- *A Code Evaluation Service (CES) is employed to assess the quality of generated code, ensuring it compiles and passes tests.*
- *Integrations for editors like VS Code, Emacs, and Neovim are designed to be flexible and maintainable, allowing for easy updates and model swaps.*
- *The architecture supports A/B testing and metrics collection to evaluate the effectiveness of the AI tools.*
- *Future developments include expanding the use of retrieval-augmented generation (RAG) and multi-agent workflows within their systems.*

[Music] my name is John kzi and I work on a team at Jane Street called AI assistant our group roughly uh is there at Jan Street to try to maximize the value that Jan Street can get from large language models and I've spent my entire career

uh in Dev tools before I worked at Jan street I was a GitHub for a long time and then before that I worked at a variety of other Dev tools companies and llms kind of present this really amazing opportunity in that they're so open-ended that we can build kind of anything that we can imagine and it seems like right now the only thing moving faster than the progress of the models is kind of our creativity around

how to employ them uh at Jan street though we've made some choices that make adoption of off-the-shelf tooling a little bit more difficult than it is for other companies and kind of the biggest reason that we have this problem is that we use o camel as a development platform for those not familiar with oaml it is a a functional very powerful language but it's also incredibly obscure language uh it was built in France and its most common

applications are in things like theorem proving or formal verification it's also used to write programming

languages um we use oaml kind of for everything at change so just a couple quick examples when we write web applications of course web applications have to be written in JavaScript but instead we write oaml and we use a library called JS of oaml that is essentially a oaml bik code to JavaScript transpiler when we write plugins for Vim

those have to be written in Vim script uh but we actually use a library called vaml which again is oaml to vimscript transpiler and uh even people at the company that are working on fpga code they're not writing verog they're writing in an O camel Library called hard camel uh so why are the tools on the market available not good for working with oaml I think it kind of comes down to a few primary reasons the first and

the most important is that models themselves are just not very good at oaml and this isn't the fault of the AI labs this is just kind of a byproduct of the amount of data that exists for training so it's there's a really good chance that the amount of okl code that we have inside of J street it's just more than like the total combined amount of oam code that there exists in the world uh outside of our

walls the second is that we've made things really hard on ourselves partially as a byproduct of working in O camel we've had to build our own build systems we built our own distributed build environment we even built our own code review system which is called iron we develop all of our software on a giant monor repo application and just for fun instead of storing that monor repo in git we store it in mercurial and uh at last count 67% of the firm

uses emac instead of normal editors maybe like vs code uh we do have people using vs code but emac is the most popular and the last thing is we're dreamers I mean kind of everyone in this room hopefully is is a dreamer in a way uh and what I mean by this is we want the ability to kind of take llms and apply them to different parts of our development flow and light up different parts so maybe we want to use large

language models to resolve merge conflict or build better feature descriptions or figure out who reviewers for features be and we don't want to be hampered by the boundaries between different systems when we do that over the next 15 minutes I'm going to cover our approach to large language models at CH Street uh particularly when it comes to developer tools um I'm going to talk about custom models that we're building and how we build them I'm going

to talk about editor Integrations so these are the Integrations into uh to vs code emac and neovim and I will talk about uh the ability that we've built over time to evaluate models and figure out how to make them perform best and I guess at first glance it's not really obvious that training models at all is a good idea I mean it's very expensive proposition it takes a lot of time and it can go wrong in a ton of

different ways who here has trained a model before or tried to train something like a model maybe took a foundation model and trained on top of it cool we were more convinced after we read this paper this is a paper from meta about a project called code compose and in this paper they detail their results fine tuning a model specifically for use with hack uh hack is actually pretty similar to O camel uh not in its

like syntax or function but really just in the fact that it's used primarily at one company and not really used much outside of that company even though it's open source so oh actually a fun fact hack is implemented in no camel I think that's just like a total coincidence but uh we were pretty naive early on we read this paper and we decided that it would be really cool if we could replicate the results we thought we

would just take a model off the shelf we would show it a bunch of our code and then we would get back a model that uh worked like the original model but knew about our libraries and idioms it turns out that's just not how it works uh it's not that easy in order to get good outcomes you have to have the model see a bunch of examples that are in the shape of the type of question

that you want to ask the model so we needed to First create a goal a thing that we wanted the model to be able to do and in our in our world the goal that we came up with was this we wanted to be able to generate diffs given a prompt so what that means is we wanted a user inside of an Editor to be able to write a description of what they wanted to happen and then have the model suggest a

potentially multifile diff so maybe you want to modify the test file an ml file and an mli which is kind of like a header file we wanted the diffs to apply cleanly and we wanted them to have a good likelihood of typechecking after they had been applied and we were kind of targeting this range of up to 100 lines as an ideal zone of what we thought llms would actually be capable of and in order for that to work we

needed to collect data like I was talking about before we needed data of the training shape that looked just like the test shape and this is what that shape looks like for this task you need to be able to collect a bunch of examples of what context the model would have had beforehand and then some prompt of what you want the model to do written hopefully in the same way that a human would write it and then some diff that

would accomplish that goal so context prompt diff and we need a bunch of these examples so how do we get these how do we get these training examples kind of the first place to look is features features is I mentioned a code review system that we built internally this is what it looks like it's called iron uh features are very similar to poll requests I think you can just you know swap that term in your head and features at first glance have

exactly the data they want on the description tab they have a human written description of a change and on the diff tab they have the code that accom is the goal of the developer but on closer look they're not exactly what you want right the way that you write a feature description or a p request description is really very different from what you might want to say inside of an editor so you're not writing multiple paragraphs in the

editor you're just saying something like fix that error that's happening right now and that's just not how we write feature descriptions another problem with these features or P requests is that they're really large right often it's a feature is 500 lines or a thousand lines so in order to use it as training data we would need to have an automated way to pull features apart into individual smaller components that we could train on so we need smaller things than

features what are those well maybe commits commits are smaller chunks than features uh this is what a typical commit log looks like at Jan street so this is not like a git short log this is literally just like an actual I want you to look at this as an actual git log and where it says summary Z that's my commit message we don't really use commits the same way the rest of the world use system so we use commits mostly As

checkpoints between different parts parts of a development cycle that you might want to revert back to commits don't have a description and they also have the same problem in that they're not isolated changes they're they're a collection of changes what we actually ended up with was a approach called workspace snapshotting and the way that that works is we take snapshots of developer workstations throughout the workday so you can think like every 20 seconds we just take a snapshot of what

the developer doing and as we take the snapshots we also take snapshots of the build status so the build that's running on the box we can see what the error is or whether the build is green and we can kind of notice these little patterns if you have a green to Red to Green that often corresponds to a place where a developer has made an isolated change right you start writing some code you break the build and then you get it back

to green and that's how you make a change maybe this one the red to Green this is the place where the developer encountered an error whether that's a type error or a compilation error and then they fixed it so if we capture the build error at the Red State and then the diff from red to Green we can use that as training data to help the model be able to recover from mistakes the next thing we need is a

description and the way that we did that we just used the large language model so we had a large language model write a really detailed description of a change in in as much words as it possibly could and then we just kept filtering it down until it was something that was around the right level of what a human would write so now we have this training data and training data is kind of only half the picture of training a model so you

you have the the supervised training data and then you need to do the second part which is the reinforcement learning this is really where models get a lot of their power right we we align the model's ability to what humans think is actually good code so what is good code I guess on the surface good code is I mean it's it's code it has the parse is code meaning if a piece of code doesn't go through the O camel parser and come

out with a green status that is that is not good code I would say by most definitions uh good code in oaml because it's statically typed is code that type checks so we want to have good code be code that when it is applied on top of a base revision can go through the type Checker and the type Checker agrees that the code is valid and of course the the gold standard is that good code is code that

compiles and passes tests so ideally in during the reinforcement learning phase of a model you could give the model a bunch of tasks that are like verifiable we have the model performs some some edit and then we check whether or not it actually passes the test when applied to the code so we did that uh we've done this as part of our our training cycle and we built this thing that is called uh CES it's the code evaluation service you can

think of it kind of like a build service except with a slight modification to make it much faster and that's that first we pre-warm a build it sits at a a revision and is green and then we have these workers that all day just take diffs from the model they apply them and then we determine whether the build status turns red or green and then we report that error or or success back up to the build function and through

continued use of this service over the course of like months we're able to better align the model to write code that actually does compile and past tests it turns out this exact same setup is the one that you would want for evaluation so if you just hold out some of the RL data you can also use it to evaluate model's ability to write code kind of looks like this you give the model a problem you let it write some

code and then you evaluate whether or not the code that it writes actually works and training is hard and it can have kind of uh catastrophic but hilarious results so at one point we were training a code review model and this is a totally separate model but the idea was we want to be able to give some code to this model and have it do a first passive code review just like a human would do to try to save some of

the toil of of code review we train this model we put a bunch of dat dat into it we worked on it for months we're real excited and we put our first code in for uh for code review through the automated agent it spun for a bit and it came back with something along the lines of um I'll do it tomorrow and like of course it did that because it's trained on a bunch of human examples and humans write things like

I'll do things or I'll do this tomorrow uh so it's it's you know not very surprising so having evaluations that are meaningful is kind of a Cornerstone of making sure that models don't go off the rails like this and you don't waste a bunch of your time and money in the end though the real test of models is whether or not they work for humans so I'm going to talk a little bit about the editor Integrations that we've

built to expose these models to developers at Jan Street kind of when we were starting building these Integrations we had three ideas in mind the first idea was wow we support three editors we have neovim vs code and emac and we really don't want to write the same thing three times so ideally we don't want to write all the same context building strategies and all of the same prompting strategies we want to just write at once the second is that

we wanted to maintain flexibility so we had a model that we were using at the time uh that was not a fine tuned model we were pretty convinced that a fine tuned model was in our future we wanted the ability to do things like swap the model or swap the prompting strategy out and lastly we wanted to be able to collect metrics so in a developer uh in their in their editor developers care about latency and they care about

whether or not the diffs actually apply so we wanted to get kind of on the ground real experience of whether or not the diffs really were meaningful for people this is the simplified version of the architecture that we settled on for this service the AI development environment essentially you have llms on one side and then Aid handles all of the uh ability to construct prompts and to construct context and to see the build status and then we are able to just

write these really thin layers on top of Aid uh for each of the individual editors and what's really neat about this is that Aid sits as a sidecar application on the Developers machine which means that we when we want to make changes to Aid we don't have to make changes to the individual editors and hope that people restart their editors we can just restart the Aid Service on all of the boxes so we restart Aid and

then everyone gets the most recent copy uh this is an example of Aid working inside of vs code so this is the sidebar in vs code very similar to something like co-pilot except this thing allows you to uh ask for it and get back multifile diffs uh and you can see it kind of looks like what you'd expect in VSS code it's it's you know a visual interface that lays things out really nicely this is what we built in emac

though so in emac developers are used to working in text buffers they move around files they want to be able to copy things the normal way that they copy things so we actually built the eight experience in emac into a markdown buffer so users can move around inside this markdown buffer they can ask questions and then there are key binds that essentially append extra content to the bottom of the markdown buffer AIDS architecture lets us plug

various pieces in and out like I mentioned uh so we can swap in new models we can uh make changes to the context building we can add support for new editors which I think probably sounds far-fetched but this is something we're actually just doing right now uh and we can even add domain specific tools so different areas of the company can supply tools that are available inside of the editors and they kind of end up in all the editors

without having to write individual Integrations eight also allows us to AB test different approaches so we can do something like send 50% of the company to one model and 50% to another and then determine which one gets the higher acceptance rate a is kind of a an investment that pays off over time every time something changes in large language models we're able to change it in one place Downstream of the editors and then have it available

everywhere and things change like really often and we need to be ready uh when things change what I what I've had time to show you today is only a small portion of what my team is doing we've got a lot of other things going on so we're finding new ways to apply rag inside of the editors we're applying similar approaches to what you've seen here to large scale uh multi-agent workflows we are working with reasoning models more and more but the approach is

the same through all of these we keep things pluggable we lay a strong Foundation to build on top of and we build the ways for the rest of the company to add to our experience by adding more domain specific tooling on top of it uh if you think what I've said is interesting and you want to talk more about this I would love to hear from you you can just find me outside and thank you for your time

[Music] [Applause] [Music]