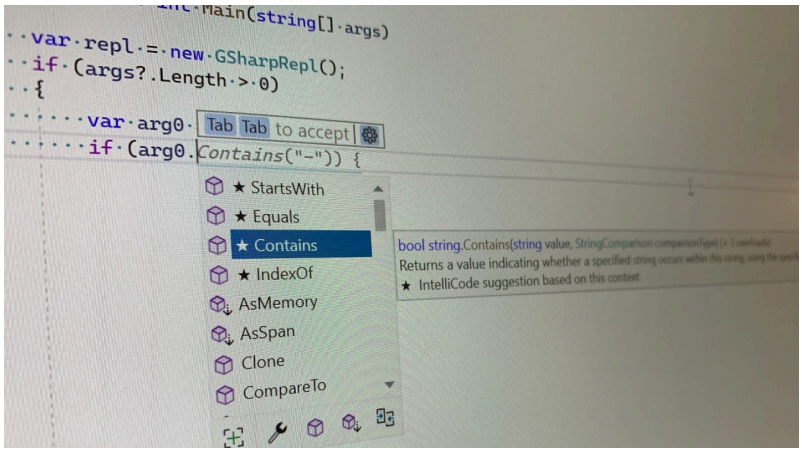

Screenshot of IntelliCode features in Visual Studio 2022

The IntelliCode story 2018-2022

David Obando
Core AI / Principal Software Engineering Manager at Microsoft

June 18, 2023

Gather 'round the fire folks, we're going to talk about the Visual Studio IntelliCode story (and a bit of GitHub Copilot's) from my vantage point. I began the project as an individual contributor and was eventually promoted to software engineering manager for IntelliCode. During the four years I was there, I was part of an amazing team that took this general idea of using AI to help coders be more productive, and helped it crystalize into a product with a story worth telling. My role enabled me to invent new ways of doing things, interact with our customers, plan and execute on our engineering needs, measure the impact of our work, and learn a ton about shipping AI tools to our customers.

As I'm starting to write this in the middle of 2023, we're seeing an "AI spring" of sorts with the proliferation of AI startups and unicorns post ChatGPT. This "AI spring" has been years in the making, with some sunny days along the last "AI winter" that tipped me personally in the direction of going deeper into Artificial Intelligence. By mid-2018 I had been working on a few AI projects at home to refresh my skills around deep learning for computer vision (and separately robotics, but that's another story), specifically convolutional neural networks (CNNs). Along the way, I started experimenting with recurrent neural networks (RNNs), doing toy projects on very limited natural language generation. This was enough to get me excited about what could be done by experts in the field and by datasets larger than what I could scramble during weekend work spikes. It's at this point that I was invited to join the recently formed IntelliCode team. I had been socializing my AI adventures with colleagues and I was eager to jump to an opportunity to have impact in the AI space.

The beginning

The IntelliCode project originated in DevDiv's Data Science team and they were deeply involved throughout IntelliCode's early years, from demos and prototypes to production-quality AI models, they partnered with engineering the whole way. The IntelliCode engineering team had very humble beginnings, with just three individual contributors, one

manager and one PM. We were working on our first attempt to help developers increase productivity through an AI model that would suggest the most likely entry in a completions list when one was available. For example, say you have "if (string." In your code. The dot in the statement would trigger a completions list, and IntelliCode would show starred ⭐ items at the very top of the list with entries such as "Equals" or "IsNullOrEmpty", based on the next phrase that was most likely. Of course, the starred suggestion could be ignored, as the user could still freely scroll the list and select the entry they're looking for.

This first model was based on Markov chains and worked remarkably well; for reference, think of this Markov chain model as a dictionary of words where each word has weighted connections to the next word based on the context. Our data set to train this model was modest, comprising mostly of code publicly available in GitHub under an MIT license (which we accessed the same way anyone else would, pulling the repos into our own cache) and some internal Microsoft repositories, focusing on learning about the use of common types such as the .NET base class library. We had to deal with significant complexities – for example to know what type was being used at any time we had to parse the code and do type binding, an exercise that you can imagine isn't trivial when grabbing random code from the internet which may have weird build steps, weird package imports, and even code that simply doesn't compile. This didn't stop the data science team, and they delivered a very capable model. It enabled the starred items experience for multiple languages such as C#, VB.NET, Python, TypeScript/JavaScript, C++, SQL and Java, but it had one disadvantage: it only worked for types it knew about and based on contexts it knew about. For example, it had great examples for the C# string class, with different suggestions for methods, constructors, or loops. But it didn't have any suggestions for types defined by your own code. We shipped this first AI model for VSCode and Visual Studio 2017 and found mostly positive feedback from our users, which gave the project wings and the political oxygen to keep growing.

One feature is out, now what?

At this point we knew we had to start measuring the impact IntelliCode had on a developer's performance to learn how to improve and what direction to go next. We knew it was important to be data driven, but at the same time we struggled to find a way to measure IntelliCode's impact. Was it fewer keystrokes? Was it a faster flow on completing a line of code? Was it reduced bugs? We had no idea. And thankfully we didn't stop here long enough to ponder, because years later we discovered that it didn't matter. IntelliCode can make a developer feel more productive by reducing the cognitive burden, which is close to impossible to measure. But I digress, let's go back to our next challenge: those missing suggestions for types defined by your own code.

We envisioned two avenues of attack to go fix the issue with starred items for types defined by your own code. The first approach was to train a Markov chain model on your own code that IntelliCode would then use to provide suggestions, we called this "custom models". The second approach was to generalize the problem and use deep learning techniques to produce suggestions on types we've never seen before as part of the base model we shipped in the box. We worked on both approaches in parallel, delivering custom models (using the Markov chain technique) relatively quickly. This first approach had the advantage of "weathered technology" which made it easier to implement and quick

to deliver, we had all the plumbing in place to make it work but the model training still required a wee bit of Azure data center pixie dust: a user would initiate a custom model training of their code from Visual Studio, allowing IntelliCode to gather metadata from their types, and then sending the metadata to our servers for final processing. This reliance on web services made the custom models approach not great for people who thought Microsoft would see their code and use it in some fashion other than to simply produce the promised IntelliCode model, which of course was never the case; we were ourselves very focused on privacy and compliance and shipped this solution that it was a temporary solution while we brewed a more general model that didn't require training on types defined by your own code. If I recall correctly, at a certain point we were training close to 14,000 custom models per day, based on our server-side metrics. As soon as a customer downloaded their newly trained model, we'd delete all assets on the service. But still, we knew this wasn't an ideal solution.

The second approach, based on deep learning, took a bit longer to cook, and was first prototyped and shipped for Python on VSCode in early 2020. This was a groundbreaking moment for the team as it required several new pieces of fundamental machinery to be put in place: 1) a pipeline for a larger corpus of code to be studied, type-analyzed, and put into model training mode, 2) a deep learning model runtime for inference that was powerful and efficient enough to be run in your laptop without draining your battery down too fast, and 3) the right techniques to shrink the model down and run it in our recommended minimum specs and within the time budget for a completions list, roughly under 60 milliseconds per shot in low power machines. That last point required novel quantization techniques that yielded great results, and paired with the ONNX runtime we found a way to ship this as a viable product. If you wish to learn more about this specific model go and read **The making of Visual Studio IntelliCode's first deep learning model: a research journey - Visual Studio Blog (microsoft.com)**. At this point we started generalizing this deep learning model to other languages, and eventually landed it in Visual Studio 2022 with support for both C# and VB.NET. With this new technology you don't need to train a custom model on your code, you simply use a completions list on your types and voila, starred items appear! There are, of course, limitations to this approach. For example, the corpus of training data is strongly skewed towards using the English language to name classes, methods, and variables, which means that you may not get stars from this deep learning model for your "Estudiante" class. Bummer, but still, we're inching closer to a more generic AI assistant for your coding needs. We celebrated this as a big win for IntelliCode.

In parallel to the development of our deep learning completion list model, we started partnering with the PROSE team from DevDiv to bring program synthesis technologies into the IntelliCode offering. They had produced the successful "flash fill" feature in Excel and we were ready to adopt this technology to power our "repeated edits" experience. If you've never used Visual Studio you'd be surprised when doing code refactorings, seeing how IntelliCode suddenly suggests every single edit you set yourself to do. PROSE works by analyzing the abstract syntax tree of your code, and seeing it evolve over time with your edits. Once a repeated pattern is recognized, a program for that pattern is synthesized and offered as a Code Action in the editor, enabling the developer to simply accept or ignore the patterns we're picking up for them. It's a beautiful experience that has evolved drastically since its introduction in Visual Studio 2019.

Peeking into the next AI wave: language models

At this point of the IntelliCode story, we take a breather. We weren't sure that life inside a completions box and code actions was going to cut it for much longer. We had a hunch that natural language processing (NLP) models were an interesting and promising research avenue, with examples such as Gmail Smart Compose leading the way in the productization of NLP experiences since ~2018. In the summer of 2020 OpenAI unveiled their Generative Pre-trained Transformer 3 (GPT-3) a large language model (LLM) which caused an immediate sensation in the AI community. Very early on, a lot of folks noted that GPT-3 was able to produce some code snippets at a higher-than-chance level of accuracy, even though it had been trained on non-specialized data sets. This key insight into GPT-3 inspired a race by several organizations, us included, to find ways to force a transformer model into producing high quality source code; we noticed that if you asked GPT-3 programming questions it would give you an answer and generate some basic code. Its Python skills were the best of all we tested, and even though it displayed syntactic understanding, its semantic understanding was lacking. Yes, we love to code, but also, it's painful sometimes. We saw that IntelliCode's humble beginnings, with stars in the completions list and repeated edits code actions, reduced the burden ever so slightly, and we were hopeful that an NPL approach to helping developers would bring even more lightness and joy to the coding experience.

Our data science team jumped into action, and we quickly formulated our first approach to the problem. Note that all along, this was intended to be an IntelliCode capability, and as a brand IntelliCode wants to be useful, unobtrusive, and trustworthy. This means that from the get-go, our goal was to run the model locally on your machine. You could get IntelliCode's features and goodness in Visual Studio even if you programmed in an air-gapped offline work environment. By the autumn of 2020 we had our first iteration of the model working, called GPT-C, an AI transformer model that was able to produce high quality code predictions. You can read more about this model's operation at **IntelliCode Compose: Code Generation using Transformer (arxiv.org)** and **Typing Less, Coding More: How we delivered IntelliCode whole line completions with a transformer model - Visual Studio Blog (microsoft.com)**. At this time the model wasn't mature enough to become a product; its high accuracy came at the expense of its size and long inference time cost. The data science team continued iterating and had a product-ready model by late 2021. This model was trained on about 52,000 repositories from GitHub containing 1.2 billion lines of code, with the training data modified so it would only look at the code structure and semantics: no comments, string literals, character literals or numeric literals were used to train GPT-C. Our recent experience in training deep learning models based on code was foundational to this effort, and the team did a formidable job at scaling the model training process. We knew that it was critical for this model to learn code structure but not comments or literals that could contain sensitive or malicious data, and that the resulting model would have high quality predictions without direct memorization.

But the model was just one layer of this onion. When we started to do our early prototypes of the UX for predictive code, it was not obvious that the Gmail Smart Compose model of showing predictive grey text would translate well to programming. We tried it to promising results, but we quickly found that there were big rocks to be moved before we could ship to customers.

UX problem 1: quality of the predicted text.

The biggest UX rock was the quality of the prediction. The model was very good, but about half of the time it needed minor corrections to make the code compile. When we had early users try the suggestions, they quickly brought pitchforks to us and demanded a refund. We learned that we needed a filtering and processing system for these predictions so that the code that was produced wouldn't introduce new compilation errors. Think of the complexity of this problem: you're typing a program that's not in a compilable state yet, and you need to check that the code you're producing doesn't negatively affect the compilability of the code. This means that we had to leverage the might of advanced compiler tools (particularly the .NET compiler aka Roslyn, and TreeSitter) to deduce whether the predicted code change introduced any compilation errors, either inside of the prediction or elsewhere in the code, and if so, edit or discard said prediction. We also ran a comprehensive statistical analysis based on thousands of hours of use of the unfiltered predictions internally and we were able to determine that not all errors were created equal. An easy example of this is that we could ignore missing semicolon errors introduced by our prediction, as those are trivial and recoverable by users or by the model itself. Some errors would have us toss the entire prediction, other errors simply trim the prediction to the point of error, and so on. Kudos to Roslyn for being such an awesome tool here. If I recall correctly, this resulted in about 40% of our predictions being edited or discarded, but the early users flipped from hating us to loving us. Don't believe that they loved us? If you're a C# developer and are used to IntelliCode line completions, try using the product with the feature turned off for a week. It's like the oxygen is gone now! It's one of those things where you only know how good it is when you don't have it anymore.

UX problem 2: presenting the predicted text.

The second UX rock was the presentation of the predicted text. We experimented with showing the text in multiple ways, including a hideous one where we'd use green squiggles under the suggested text. It was bad. We quickly realized grey text with a slightly different look to your regular code would be beneficial, but this in turn presented three separate emerging problems: 1) How do you present this to a developer with low vision? After all, Visual Studio has excellent support for developers with vision impairment to be successful and productive, 2) How do you accept or ignore a predicted suggestion? And 3) What happens when there are completion lists in the way?

To tackle the first presentation problem when building a novel UX that is accessible to developers with low vision, we spent hundreds of hours following best design patterns for the code and doing A/B tests with developers with low vision, using multiple voice assistance tools and ensuring that the velocity, tone, and length of predictions are all working well for them. It was interesting to do this testing ourselves, especially when we combined localization and voice assistance, hearing Visual Studio in Spanish was a trip.

To tackle the second presentation problem when accepting or ignoring predictions, we quickly ran into people suggesting opposing solutions, and it became apparent that it was a design decision rather than one based on data. We settled for escape key to dismiss the prediction, tab key to accept the prediction, and the ability to "walk the prediction" by typing the text that was there, supporting backspace without dismissal in the process. This worked out well for most people.

Finally, for the third presentation problem when dealing with completions lists, this became an opportunity for us to “steer” the model into different directions. We were able to make the inference fast enough to show a new prediction based on which item in the completion list a user is currently selecting. If a user doesn’t see a prediction, it’s because it didn’t meet our display criteria.

UX problem 3: length of the predicted text.

The third UX rock was the length of the completion we would show. It became apparent that shorter completions were better than longer ones, from the perspective of reducing the cognitive load and saving keystrokes. Not only was this interesting, but it also helped us reduce the inference time by requesting fewer sub-tokens from the model. Our early users also strongly expressed a distaste for suggestions that spanned multiple lines. So, we decided to make short predictions, and never more than would fit in the line that’s being worked on.

This wrapped up the largest bulk of UX challenges, and we were able to ship IntelliCode line completions for C# in Visual Studio 2022, released in the fall of 2021. This was, to my knowledge, the first GPT model to be shipped to customers of any product category, that ran completely on their machine without the need for a web connection to a data center.

GitHub Copilot

While we were busy building the first version of IntelliCode line completions, our friends at GitHub were collaborating with OpenAI on their GPT-3 based model for coding, called Codex, which is built to run the inference in powerful datacenter computers, and as such was trained in a larger and broader data set, producing results that IntelliCode line completions could never produce. Both GitHub Copilot and IntelliCode line completions show the predictions in grey text, but that’s about the only thing they have in common. Both your car and an F-16 fighter jet have a seat, and both can get you from A to B, but that’s about the extent of their similarities. IntelliCode focuses on productivity and effective work within your IDE, whereas GitHub Copilot is a truly versatile programmer that can be nudged to generate complex code in seconds. They serve different and somewhat compatible purposes.

When GitHub first started working on Copilot, they had very little expertise in shipping extensions for Visual Studio, and our partnership started naturally as we knew it would be mutually beneficial to collaborate. I led the engineers that enabled the Visual Studio APIs that Copilot consumes to produce the code prediction experience, and we also shared a lot of our UX expertise with them in the process. Most importantly for me, it allowed for very early dogfooding and bug bashing for Copilot in Visual Studio and painted a clear path of differentiation between our two offerings. I learned a lot about how collaborating with external teams can elevate both groups, especially when working together to a win-win scenario for GitHub and Visual Studio.

It's amazing what can be done today with AI-assisted programming. If you haven't tried it yet, stop what you're doing and take GitHub Copilot for a spin, hopefully on a challenging problem or on a technology you're not very familiar with. I hope it blows your mind the way it keeps blowing mine. If you'd prefer to run models locally in your machine, give IntelliCode a try. Both IntelliCode and GitHub Copilot are available in Visual Studio and in VSCode.

AI spring 2023, and smaller models

Again, back in the middle of 2023, as I sit and reflect on the current "AI spring" and my own journey through producing multiple kinds of AI-based products, it doesn't surprise me that everyone is trying to pivot and produce generative AI models for everything and in every product; it's surprising how much of an "iPhone moment" ChatGPT was, and how the snappy and fast movements of OpenAI have enabled this moment; I can see how people would want to produce features such as IntelliCode's repeated edits using a large language model such as GPT-4, which I'm sure would work, but at a cost completely disproportional to what makes IntelliCode a viable product. I know that smaller and more targeted models can do wonders with a fraction of the cost to produce and ship to users. The cost of these models matters: training, productization, inference, all these costs must be assessed before committing to an expensive AI investment path. Organically growing capabilities based on your available data set and your customers' problems should always yield interesting results. It may take years for these investments to pay off, as was the case with IntelliCode, so the right mindset must accompany the right execution, focused on the value that you may be trying to achieve.

Enjoyed this article?

Follow to never miss an update.

David Obando

Core AI / Principal Software Engineering Manager at Microsoft

+ Follow

More articles for you

QLORA: Enabling Efficient Finetuning of Quantized Language Models (Summary)

Oluwatobi Adefami

2

This Week in AI (February 21 – March 2)

This Week in AI

8 · 2 comments · 1 repost

AI/ML Digest | Issue 26

AI/ML innovations digest

2