Assignment 2

My research

My PhD work is focused on techniques for building cloud-based compilers & static analysis tools. Building language tooling can be challenging, and making those tools run well in a cloud environment adds an extra layer of complexity. I am trying to find solutions that allow you to write language tooling that runs on multiple machines, without having to think about network-related issues like latency, serialization, synchronization, etc.

More concretely, I am working with Reference Attribute Grammars (RAGs), a declarative technique for specifying functionality on top of an abstract syntax tree (AST). When creating RAGs you specify functionality, but not necessarily "when" or "how" your functions are evaluated. That decision is left to the system that evaluates the RAG specification. I am working with such a system which is called JastAdd. JastAdd already has a few evaluation strategies that adds automatic concurrent evaluations, caching with automatic dependency tracking (for incremental evaluations), and more. I want to create a new evaluation strategy that works across multiple machines in a network.

Topic 1: Continuous Deployment

Continuous deployment feels like the natural progression of software engineering. We started with waterfall, where everything was planned in advance and you had a very lengthy QA/testing process before release. Then we got agile, where the development process itself became more flexible, but the release process was, for many areas, still lengthy and complicated. Later, as the internet became more prevalent, it was possible to update devices and services even after release, which introduced "release" as a natural part of the agile cycle. At my previous job, we actually released/deployed our product for *every single commit* to the master branch. It required an extensive testing pipeline, and to be honest it often lead to issues, but it shows how far we have come from waterfall.

One of the reasons that continuous deployment is possible is because everything is versioned (usually in Git), and deployment is automated with software-defined infrastructure (for example CloudFormation in AWS). Mistakes are bound to happen, but thanks to the versioning and automation, it is always possible to roll back a change. When it comes to AI/ML, this is not necessarily true. Versioning can be really hard to do for a system that gradually changes every single day and whose models can be extremely large. That is, I think, an area that will require more work in the future.

Looking forward, the next (somwhat related) step after continuous deployment is automated, cloud-based developer environments. Humans themselves are one of the most expensive resources during the software development process. Onboarding new people is a lengthy, complicated process. At my previous job, we usually told people that during your first ~2 months you weren't expected to be "productive", since there is so much to learn. Add to that the fact that many people don't stay more than a few years at any given company, then you spent a significant part of your entire career just being onboarded. With cloud-based environments like Github Codespaces and Gitpod, you can remove some of the onboarding cost since you get a fully working environment with a single click. Also, automated developer environments reduce the maintenance cost that every developer otherwise has to pay to keep their local environment up to date.

Topic 2: Automated Software Testing

This is a related topic to Continuous Deployment. In order to continuously deploy, you must (should) have an extensive automated test suite. There are many classes of tests available. Unit tests are small, focused on a single piece of functionality. Smoke tests are larger, black-box like tests where you just want to see that the system runs at all, without checking details. Integration tests are used to ensure that different systems integrate properly, and so on. Tests are sometimes refered to as being "flaky", if they randomly fail due to e.g race conditions or other random factors. Having flaky tests in a project is concerning, but tests involving AI/ML are all "flaky" by their nature. Rather than binary pass/fail, AI-related test suites will need to measure success as a fraction instead.

There is an open source project called "Dolphin Emulator", which is an emulator for two games consoles (GameCube & Wii). Testing for regressions is extremely time consuming due to the large library of games the emulator supports. To address this problem they have developed a graphics testing pipeline stage they call "FifoCI" (see https://github.com/dolphin-emu/fifoci). FifoCI loads up games at a predetermined state and renders a frame. Then it compares the rendered frame to the frame from the previous commit, and notifies the developer when a change is detected. It also compares rendering results across multiple renderer backends (OpenGL, Vulkan, etc). The test pipeline itself doesn't necessarily make judgements on whether a test passed or failed, it just notifies developers when changes happen. Sometimes the change is good, sometimes bad, each developer has to determine that by themselves when a notification comes in. A similar strategy can probably be applied to AI/ML, where a model is continuously evaluated, and whenever a significant change is detected, the pipeline sends a notification to developers.

Topic 3: Human-Computer Interaction

The field of Human-Computer Interaction (HCI) has some connections to my PhD topic. One of the reasons why you would want cloud-based developer environments is because it reduces the barrier to entry for beginners to start programming. I believe that in the future, more and more people will need to program as part of their jobs, and a majority of people must at the very least be somewhat familiar with programming, even if they don't write code every day. Now, how do we teach programming to people who don't actively seek it out or aren't interested in it?

When I started learning programming I downloaded an IDE called Code::Blocks, and I wanted to write a simple hello world program. To run my program, I had to configure my compiler to find header files on my computer. I had no idea what a header file was! It was quite intimidating. I felt like the UI I was using had been designed with professionals in mind. Eventually I solved it, but if I wasn't interested in programming, then it would have been very easy to just give up.

IDEs today are much better designed for beginners, but I think we aren't finished yet. The ultimate goal for teaching purposes is to have a url students can click on, say "www.myschool.com/programming-course/editor". When they get there, they find a repository has set up, and there is a fully configured IDE available right in the browser. There should also be a big "play" button available to run the code. Creating your first program should not be harder than writing `printf("Hello World!\n");` and pressing play.

Future trends in Software Engineering and AI/ML

The connection between my PhD topic and AI/ML is quite weak. I started my PhD because I was working on a compiler in my spare time and found it fascinating. I think we are a very long way from having AI used when *compiling* code. *Analyzing* code, however, is a different matter. Github Copilot has shown us that AI can be very good at recognizing patterns in code. From a simple comment like "// Calculates X based on Y", Copilot can often synthesize a method that often works pretty well.

I feel like Github has a golden opportunity for "Copilot v2" not only to synthesize code, but look at and identify bugs in code you have written manually. Github probably has access to the largest set of commits/pull requests in the world. With some lightweight analysis on labels and comments in changes, you can determine which of them are bug fixes. You could then take all the bug-fixes and feed that to a ML model for recognizing bugs.

While many analyses are fast enough that ML isn't needed, some are very slow. I met a group at a conference earlier this year that did concurrency analysis for C code. They wanted to detect instances where mutex locks were taken in an order that could cause deadlocks. Their analysis took roughly 20 minutes to run on a project with 2k lines of code. A ML model might not detect everything, but I would expect it to be several orders of magnitude faster, and probably still detect a lot of issues.

We are living in the midst of an AI/ML revolution, and it is easy to get caught up in the enthusiasm and think that AI will solve everything. I think it is important to sometimes consider that "normal" algorithms, customer surveys, etc sometimes work much better, and at a significantly lower cost. At a previous employer I added probes to our product to collect various bits of information. Much of the information seemed irrelevant to me, so I challenged the project owners on why this was necessary. They justified it by saying something like "more data is better, we can decide what to do with it later". Maybe it was the correct decision in theory, but I know for a fact that we collected a lot of data that nobody ever looked at. The energy/carbon cost of collecting and storing all that unused data seems so wasteful.

Almost all ML models today are presented in terms of their accuracy and number of parameters. For either metric, more is better. Very rarely do people mention the cost of training the models. I recently learned that GPT-3 is estimated to have cost ~5 million dollars to train. In the context of large silicon valley companies, that is not very much, but it is much higher than I had anticipated. I believe that in the future we will see more and more models mention their training/inference cost as well, especially with all the debate around electricity costs we have had recently.