My immediate response to seeing the project was “oh, hey, this is pretty simple, I can just do this in AWS” and as you saw I went about figuring out what that would look like. I actually got pretty close, I got all the pieces connected in AWS, but deploying it would have been a pain. I probably would have spent my entire time just fiddling with CloudFormation and that doesn’t serve much purpose. I’m happy that I came up with the “product” version, but I think I let it get in the way of just building something to showcase. On the flip side, a lot of my conversation with Jonathan revolved around those ideas and it helped guide me in a better direction (plus I learned some new stuff in the process).

The Python server didn’t take me particularly long at all to build. I probably spent more time writing the API document and fiddling with Python 3 (everything at Amazon was 2.7) than I did on anything else. I’m happy with the way the server came out. It’s simple and Python’s base HTTPServer is a bit slow, but it gets the job done, is fairly extensible in terms of supporting API, and is easy to start and communicate with. It has hard expectations about how the data is supposed to be formatted that would make it impossible to use in a more general purpose situation, but those expectations made it so I could get things running without much fuss.

As for my API design, I’m a bit iffy on it. A lot of the design is the way it is because I started with this vision with SQS, which inadvertently turned the pipeline into an eventing system. An event has a name and a payload, right? So that’s how I designed the API: a singular endpoint where everything is defined in the request’s body. This *works* and made the implementation a little bit easier, but it isn’t very RESTful and would make supporting other tools (like say a web UI) more difficult or impossible.

In particular, this means I have a GET request that has a body, which while is not strictly prohibited in HTTP,  it is bad form and straight up not supported by .NET. I spent a lot of time trying to work around that and ended up needing to pull in a nuget package for a specific HTTP handler from Microsoft. In theory, I could have just changed GetScores to be a POST method, but that felt even more wrong.

There are also a few quirks when it comes to error handling. Pretty much every failure is chalked up to a bad request, even when that might not be the case. For example, the server treats a DeleteScore call that had nothing to delete as a bad request even if the request syntax was correct. It’s not the end of the word, but feels a little bit weird to interact with.

Were I to make Leaderboard v2, I’d definitely take a more traditional RESTful approach to the API. GetScores definitely should have supported multiple endpoints (/getscores/, /getscores/{username}, /getscores/{username}?param1=x, etc).

A few other areas of improvement are supporting multiple leaderboards and regions, bulk score submission, bulk delete score from a specific user, better error handling, as well as having a web view of the leaderboard.

The visualizer tool turned out more or less as I expected it to. WPF isn't exactly the most relevant framework anymore and building a cool web UI likely would have been the better choice, but I'm well versed in WPF and it meant I could get everything up and running *really* quickly. The tool reinvents the wheel a little bit by allowing you to make API calls to the server (which is completely doable in Postman), but I didn't want it to just end up being a mostly uninteractive tool. It lets you view the leaderboard from GetScores, see exactly what is in the database file, as well as start the server and monitor its output.

I did make use of Xceed's Extended WPF toolkit for this. They provide a PropertyGrid and a few other controls that I think make using the tool a lot better. I also have some code in the tool's Main that lets it support embedded dependencies that I took from a blog (it's sourced in the Program.cs). I did this because I wanted to be able to provide a singular exe if for some reason you weren't able to build it, and dealing with dependencies in C#/.NET can be a literal hell.

In a more robust pipeline, this tool would need to be a bit more involved in terms of connecting to varying services and requesting data, and it's current form isn't necessarily extensible in that regard. I did, however, maintain MVVM wherever I could, so the UI could be stripped out and replaced with something else if necessary. I'd also have liked to provide a nicer looking Leaderboard UI.

There are a couple things to keep an eye out for when using the viewer tool. I know I don't handle every failure case, so I wouldn't be surprised if certain interactions would cause it to crash. I did my best to be defensive where I could, but UI validation in WPF can be time-consuming and I felt there were better areas of focus. Additionally, the ability to start the Python server from the tool was put in at the last minute. It's nice to have everything in one place, but the tool doesn't do much to ensure you don't end up in a bad spot. It tries to stop it when the program ends, but won't do much if it's already running, or if the tool crashes. You can definitely end up with multiple instances of python running, though I tried to mitigate that case best I could.

Overall, I definitely ended up spending more than 4 hours on the project. The exact amount I'm unsure. The time I spent actually coding, testing, and debugging was probably around the 5-6 hour mark and I think I could have submitted a working project at that point. Where the fuzziness in time comes from is I definitely spent a non-trivial amount of time playing around in AWS, fiddling with Postman (I had never used it before), tweaking the UI, and writing documentation.