I was having endless recording problems so I included the script because I couldn't get a great sounding take. So I wrote up a 'script' for reference.

Script:

Hi I'm robert cole and this is my implementation of my coding test.

Starting off with it running both unit tests and then postman tests locally, you can see I made some basic tests to make sure everything worked correctly. They aren't strictly unit tests but more integration tests just so I could ensure things were working and keep a tight schedule. It was also important for me that I knew I could test and debug things locally.

- How did you design the app?

Given limited time I primarily focused on testability and flexibility, I figured my lack of knowledge in finance could be compensated for by a design that's easily tested and changed. I created both services as Azure Functions with the serverless deployment in mind, I didn't really see a need to make either of them complicated enough not to be done with a serverless app, I figured: save the money. So I made sure to do things with environment variables so that it can be run in nunit, or locally with postman tests, and ideally when I deploy it would pull the values from the Azure Function Envars. I created query objects so that I could group parameters logically and the framework could bind the query string to the object for me. Then I just used a light pipeline pattern to chain processing of the requests in a modular way so I could test and swap things easily.

- How did you store and retrieve data?

I wanted to use some more of the Azure ecosystem so I opted to use the Azure CosmosDB, I didn't think the app was complicated enough to do anything fancy with SQL, it seemed very simple add/retrieve so I used the MongoDBAPI for Cosmos to keep things simple. Using the Cosmos Emulator for local testing. For NUnit I also have a very barebones store that I can Inject to test without The Cosmos emulator. I opted to just simply use the database that was created by the unit tests for the local testing as well, so I seed the database with the run of the unit tests. I figured for this example it didn't warrant a more complex seeder setup.

- What did you find challenging? Please explain.

I definitely felt the lack of understanding on how finance works in general or what kind of ways someone would use the data, important regulations to keep in mind made me a bit more timid, I was worried about cutting unused fields or eliminating things that seemed redundant like the Source and Bureau, so I opted for a design where the Source and Bureau are in the model and the query but the repository doesn't use them to retrieve the data, because that would be very easily changed. I'm more comfortable with how people use games and game engines so I think I would have done things like if i knew the AccountNumbers on the tradelines could be represented by unsigned longs instead of strings I could save some space, or make the tradeline type a boolean for secured or unsecured, or an enum if there was a few specific types of loans, or even if a secured tradeline means it's a mortgage… I didn't feel comfortable about making those assumptions so I felt like some of the things I would have done to simplify and streamline I couldn't do. For example I would consider setting up a system to convert the data coming from the source to the more friendly format so we can use it how we want.

Given the following scenarios, what design approach would you recommend?

• The data source by which Service #1 connects to for retrieving the trade lines takes approximately 5 seconds to return a response

I would figure out what I'm allowed to keep regulation wise, and store the responses from the external service into our own database as a cache. I could very easily add datetime stamps for expiry so that when the data is no longer valid I could clear it out of the database. For example at night with a scheduled event like  an Azure Function on a time trigger that just queries for any record that is expired and removes it.

• The data source only allows up to two concurrent requests

So I don't know what Azure system would let us do this easily but what I would use is something like the command pattern to bundle everything that the request needs and submit it to a queue then have two "executors" pull commands from the queue and execute them against the service, returning the results to the caller after execution. So I could isolate the ownership of a connection into two distinct executors and they could manage their connection to the external service. That way any number of clients could be spawned and if

that service decided to support more concurrent requests down the line we could just spawn more executors.

• Service #2 must be enhanced to support a single bulk request of up to 5000 o Assume there is flexibility with the consumer of Service #2. For example, the consumer could fire a single HTTP POST containing all 5000 requests and receive the results later/asynchronously. For the purposes of this exercise you can assume that as the tech lead you will advise the consumer on the best architectural approach.

I would use the Claim-Check pattern here so that we could store the bulk request in cheap storage and then retrieve it by the executor rather than cramming the whole thing on the message bus. one thing I do for something similar to this in my research project is when I get bulk requests. When the client sends the claim check the grid service returns a GUID to represent the job. I can of course use the GUID to query about the jobs status but when the job is complete I then fire a JobComplete event with the GUID on the message bus so I don't have to keep polling. The consumer could add the completed job to notifications for the user so they know it's ready, even pushing one via a websocket system like SignalR, which is what I use so that if my user is still online they get notified as soon as the job is complete. In my system I also store notifications if the user is offline so that if someone fired off some bulk requests then went home for the night they would still be able to see if the job succeeded or failed when they logged back in.

I'm hoping you enjoyed my little run-through and we can continue this process.