**Question 1:** What is a Data Lake? Explain its benefits and how it might benefit a client.

A data lake is a highly scalable and flexible repository for storing raw unstructured or semi-structured data before any processing or analytics have been performed. Data in a data lake is often recently ingested, and the data lake often serves as a “staging area” for data before clear, well-defined use cases and processing pipelines are fleshed out and implemented. This contrasts with a “data warehouse”, which is a scalable repository for storing processed, structured data that is ready to serve immediate business needs. Data lakes and data warehouses are *complimentary* technologies, and often data from a data lake is fed through a processing pipeline where a structure or schema is applied to the data, and then ultimately stored in a data warehouse for future querying.

Data lakes are useful when it becomes impractical to process data *as it is generated.* In today’s “data-driven economy”, there are often simply not enough human resources available to process the enormous volumes of data businesses are generating in real time. Data lakes provide an immediate staging area to hold data as it is generated, so potentially valuable data is not lost. This gives businesses and their data workers a “time buffer” within which to implement processing infrastructure to utilize this data, and discover new, potentially unforeseen, use cases for the data. The alternative is to simply discard the data, which amounts to lost business value and missed business opportunity.

**Question 2:** What are the pros and cons of serverless architectures?

Serverless computing services like AWS Lambda and Azure Functions are the latest innovation in “cloud native computing”. As with any technology, serverless computing architectures come with benefits and drawbacks. The foremost benefit is the agility and flexibility it gives to software developers and the applications they build. With a serverless computing architecture, one does not need to worry about or manage the underlying infrastructure a given piece of code runs on. This burden is left to the cloud provider, allowing businesses to utilize the “practically infinite” scalability of the provider. Functions within an application become fundamentally decoupled from one another, and need not even be written in the same language, giving developers huge flexibility in how applications are written and designed. Finally, most “Function as a Service” serverless computing offerings provide broad integration with the ecosystem of other services available in typical cloud computing environments, allowing developers to focus on writing software without managing the cumbersome details of the infrastructure surrounding their application logic (databases, API routing, load balancing, etc.)

The primary benefit of serverless computing architectures (offloading management of the computing infrastructure) could also be viewed as its primary drawback. With a serverless computing architecture, you exchange *convenience* for *control.*  For certain applications, control of the underlying infrastructure may be a hard requirement, in which case serverless computing architectures simply aren’t suitable. Additionally, while on the surface serverless computing architectures seem to increase *simplicity,* they often end up increasing *complexity*. Serverless computing architectures are inherently distributed, and as the scope of a given application built with such and architecture grows the distributed nature of the application can make maintenance and debugging of an application complex and time consuming.

**Question 3:** Please provide a diagram of the ETL pipeline from Section 1.5 using serverless AWS services. Describe each component and its function within the pipeline.

Oh boy, this one could take awhile.

**Question 4:** Describe the process of deploying infrastructure as code on AWS or Azure and why this is a good method of architecture delivery for clients.

Process:

1. Choose an appropriate tool, of which there are many (Terraform, CloudFormation, Azure ARM Templates, Ansible)
2. Plan and design the desired state of your infrastructure (this part is arguably the most work)
3. Declare/describe that state in the syntax of whatever tool you choose
4. Click the magic deploy button!

Good because:

1. Repeatable/consistent
2. Version controllable
3. Self-documenting
4. Faster and less error prone
5. Portable