Pipeline Integrity Evaluation Application

A Design Example

Aaron Viviano, Software Architect and Engineer

Presentation Overview

- About Me
- Assignment Goal and Assignment Question 1: Data Estimates and Designs
- Assignment Question 2: Security for the Designs
- Assignment Question 3: On-Premise Database Integration
- Assignment Question 4: Storage Trade Offs
- Questions and Comments

About Me

- 16 years of professional Software Engineering experience
- 10 years of professional Software Architectural experience
- 7 years working on Medical Devices and Regulated Equipment
- 3 years working on Machine Vision Systems
- I'm a team oriented self-motivated engineer who loves working with people and on challenging problems.
- My software solutions have:
  - Produced and quality checked billions of products
  - Produced 2.2 million radioactive drug doses for patients
  - Operated in realtime, asynchronous, high performance, and resource constrained environments
  - Successfully been submitted to the FDA on multiple occasions

Assignment Goal and Assignment Question 1

- The assignment goal is as follows:
    - "Design the architecture of a Pipeline Integrity Evaluation application for oil and gas pipelines. The goal is to ensure the ongoing assessment and monitoring of the structural integrity of pipelines, focusing on detecting potential threats, preventing incidents, and prioritizing maintenance activities."
- Question 1: "What cloud services do you suggest?"
- To answer this question, first its a good idea to understand how much data the system has to process. Data storage and transmission (egress) can be a large source of business costs, architectural decisions, and user experience concerns.
- For example: if the data generated for a year is in total megabytes or less, then a simple web server, with an in container database, would likely be enough. If data generated for a year is in total petabytes or more, then the system may need an on-premise component to help reduce cloud costs.

Data Estimates - 1

- Research on inline inspection technologies suggests an anomaly resolution* of 1mm^{2}**. Based on wavelength sizes, this may mean that the tools are using 300-700KHz† ultrasonic sensors.
  - Other sources†† suggest 1MHz (Lamb Waves) to 3MHz (Shear Waves) sensors are used. Possibly due to how sound waves propagate through the pipeline material? More research here would be needed to confirm.
- A 1mm^{2} resolution is 1 million data points per square meter.
- Listed pipe widths max out at 56 inches. To record this data in tenths of a millimeter, 16bits of precision is required.
  - Pipe wall thickness is significantly less than 56 inches. However, dents in a pipe may be quite severe. As such this estimate is conservative in data needs across pipe line metrics and likely 8 bits of precision isn't always enough for wall thickness or pipeline geometry.
- Thus a square meter is ~2MB of data.

* Higher resolution may be available.
** Tools in motion can affect the accuracy of this reading along the length of the pipe. Sensors are estimated at 20.48mm in width, distributed in two rows offset by 10.24mm.
† In computer vision to find a feature one needs at least 2-3 pixels along an axis, this may be the same case with ultrasonic waves as well. As such while its believed that a 343KHz can find a 1mm sized anomaly, 0.5mm resolution may be needed to be accurate at 1mm, thus a 687KHz sensor may be needed.
† † https://www.sciencedirect.com/science/article/pii/S0963869517302025#bib15

Data Estimates - 2

- Average pipeline size in the US is 26 inches*.
- For a 1km 26 inch pipe the total data generated is ~415MB.
- There are 2.4 million miles (3.8 million km)** of gas and oil pipelines in the US.
- Pipelines are inspected roughly once every 6 years†.
- Assuming a company inspects 20% of the US market a year, that is ~128,747km of pipelines per year inspected by the company's tools.
- The inspections thus generate ~53 TB of data per year for the company.
- Estimated Azure storage cost ($13,632) and networking (egress) cost ($91,176) per year for each year stored. Note this assumes no use of slower access speed storage such as glacier.

* https://blog.geocorr.com/pipeline-projects-vary-by-diameter-across-the-united-states
The distribution of pipeline sizes is unknown. More research would be needed to determine if a bell curve or logarithmic or other approach would be ideal for a more accurate estimation of pipeline surface area in the US.
** https://www.api.org/oil-and-natural-gas/wells-to-consumer/transporting-oil-natural-gas/pipeline/where-are-the-pipelines
† Diagnostics and Reliability of Pipeline Systems, Timashev and Bushinskaya, pg 133. Federal regulations 49 CFR 192.939 contains a table of required pipeline inspection rates which differs from Timashev's and Bushinskaya's noted rates. This design assumes that Timashev and Bushinskaya have industry knowledge beyond regulatory requirements.

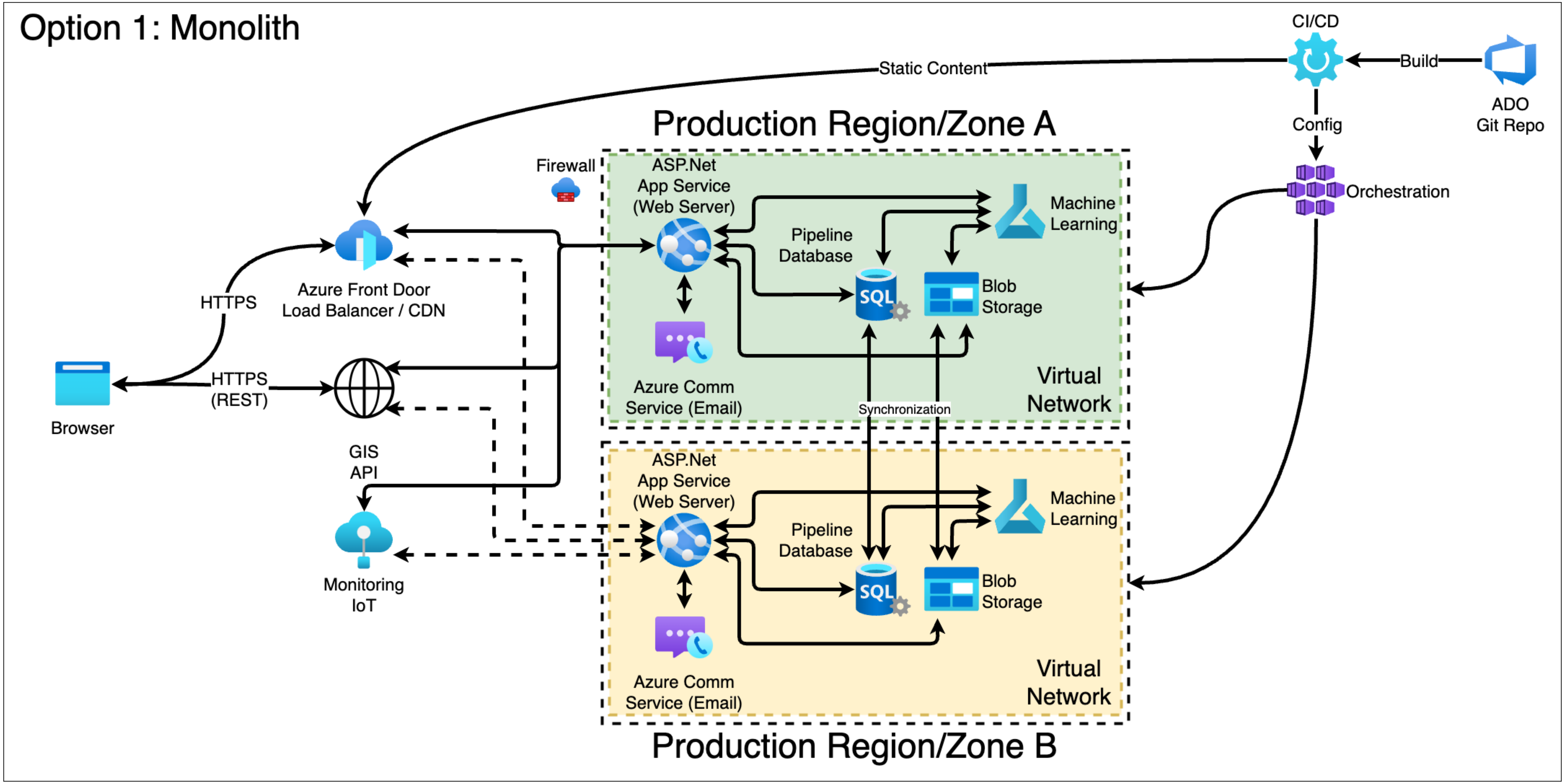
Data Estimates - 3

- The aforementioned 53TB per year of data, does not take into account:
    - All possible measured pipeline metrics.
    - Pipelines with multiple diameters.
    - Sensor type differences between EMAT, MFL, etc and their associated pipe diameter limitations.
    - Multiple sensors on a specific tool which would have a multiplicative effect on data storage needs.
    - Tool speed or environmental factors (such as liquid type, temperature, pipe material, friction, general pipeline corrosion creating uneven surfaces, etc.) affecting sensor readings.
    - 3rd party integrated sensor data, such as monitoring devices, or external non-destructive test tools, or manual ultrasonic readings.
    - Pipelines that require multiple passes due to pipeline layout or odd movement (such as back and forth motions, or sudden shifts in position, due to fluid dynamics, friction, or other sources) within a pipeline.
    - Any recommended inline inspection data acquisition best practices.
    - Individual state or local regulatory needs.
    - Actual company market share and pipeline inspection rates.
    - Pipelines outside of the US and their associated data needs, likely across multiple cloud regions and providers.
- Additional time and resources would be required to address the above complications.

Data Estimate and Design Options

- For 53TB per year, the use of the cloud is likely reasonable to reduce system cost, and hardware maintenance.
- This presentation proposes 3 design options for the system:
  - Option 1: A standard monolithic web server.
  - Option 2: A Backend for the Frontend web server with a Serverless backend to provide access to APIs and services.
  - Option 3: A Modular Monolith with optional Serverless modules.
- Note that no specific framework was chosen for the web client in all options, as the design is agnostic. I'd recommend Typescript over JS, with a combination of Web Assembly where possible. Ideally I prefer MPA approaches over SPA approaches. In this case a SPA may be the more ideal choice, given the application's similarity to desktop style applications.
- Also note that broadly speaking data regulations are not part of these designs. Specific countries may require their critical infrastructure data to be stored in their country, or a trusted partner and may have specific regulatory requirements around data access and storage.
- Finally due to lack of time the designs here don't show considerations for observability, cloud infrastructure monitoring, logging, access monitoring, permissions schemes, database table layouts, etc.

Design Option 1: Monolith - 1



Design Option 1: Monolith - 2

- Option 1 is a standard approach to application design. The Web Server is a single Monolithic executable run from an Azure App Service instance. If desired, this ASP.Net application could also run from a container.
- In front of the Web Server is an Azure Front Door Service instance. This is a load balancer and a CDN, allowing any static content to be served to the user, as well as balancing user requests between Azure Regions or Zones.
- The Web Client would be designed to request any Geospatial information, for example image tiles from a GIS service. Suggested services could be ArcGIS Online, or Azure Maps depending on user needs and pricing.
- Pipeline Monitoring is likely performed by third party IoT services or devices. The Web Server would either call into the services/devices or provide a means for the services/devices to register with the Web Server.
- Data is stored in a standard relational model, any semi-structured or unstructured data is stored in blob storage. The use of a data lake is likely unneeded given that most data will be from company designed sources and therefore standardized.
- Emails to users (such as for generating pdf reports) are sent via an Azure Communication Service instance.
- The entire application is within a virtual network. Each region / zone has its own isolated virtual network.
- The application is deployed to a specific region / zone via an Azure Kubernetes Service instance. Note that AKS operates at a region level. Therefore each region will have its own AKS instance, and the DevOps infrastructure will have to be designed to connect to each instance.
- Not pictured are the DevOps develop and QA staging environments which would be deployed to first, prior to deployment to production. Also not pictured is the approach to blue green deployment, this would likely be done in the same region / zone using separate resource groups. Option 1 likely has the least impact on DevOps given the lack of usage of Serverless Functions and limited Azure Service configurations.

Design Option 2: BFF + Serverless - 1

Option 2: BFF + Serverless Design
CI/CD
Build
ADO
Git Repo
Config
Orchestration
Static Content
Production Region/Zone A
Virtual
Network
Firewall
HTTPS
Azure Front Door
Load Balancer / CDN
Browser
HTTPS
(REST)
GIS
API
ASP.Net
App Service
(BFF)
SB
After 3
Failures
DLQ
Notifies
Service
GeoSpatial
API
Monitoring
API
Inline
Inspection
API
Non-Destructive
Test Report API
Pipeline
Metrics API
Assessment
API
Machine
Learning
PDF API
Email
API
Optional
Monitoring
IoT
IoT Upload
VM (Optional)
Pipeline
Database
Machine
Learning
Blob
Storage
Azure Comm
Service (Email)
Firewall
ASP.Net App Service
(Support)
HTTPS
Support
Browser

Design Option 2: BFF + Serverless - 2

- Option 2 consists of a thin web server, that acts as a combined API provider for various Serverless functions. These functions contain the main logic for the system. Each Serverless function is given and executes messages from an Azure Service Bus instance. If a message fails to be processed three times, the message is sent to the DLQ (Dead Letter Queue) and field service is notified. The Web Server is a single executable, running from an Azure App Service instance. If desired, this ASP.Net application could also run from a container.
    - Note that the Service Buses provide their own DLQ, so it may be possible to remove the separate DLQ from the design.
- In front of the Web Server is an Azure Front Door Service instance. This is a load balancer and a CDN, allowing any static content to be served to the user, as well as balancing user requests between Azure Regions or Zones.
- The Web Client would be designed to request any Geospatial information, for example image tiles from a GIS service. Suggested services could be ArcGIS Online, or Azure Maps depending on user needs and pricing.
- Pipeline Monitoring is likely performed by third party IoT services or devices. Querying data from Monitoring services or devices is performed by a Serverless Function. As the services or devices may have custom TCP/IP or UDP protocols with which to upload data, an optional secondary VM, running a .Net application, is provided to handle these connections. This use of another application keeps the Web Server thin and easier to maintain.
- Data is stored in a standard relational model, any semi-structured or unstructured data is stored in blob storage. The use of a data lake is likely unneeded given that most data will be from company designed sources and thus standardized.
- Emails to users (such as for generating pdf reports) are sent via an Azure Communication Service instance.
- The entire application is within a virtual network. Each region / zone has its own isolated virtual network.
- The application is deployed to a specific region / zone via an Azure Kubernetes Service instance. Note that AKS operates at a region level. Therefore each region will have its own AKS instance, and the DevOps infrastructure will have to be designed to connect to each instance.
- Not pictured are the DevOps develop and QA staging environments, which would be deployed to first, prior to deployment to production. Also not pictured is the approach to blue green deployment, this would likely be done in the same region / zone using separate resource groups. Option 2 has a higher impact on DevOps given the usage of Serverless Functions and increased Azure Service configurations. In addition more effort is needed to ensure that deployed functions are synchronized across API versions or support multiple API versions at the same time.
- Finally this design provides a separate web server for service personnel to evaluate DLQ messages.

Design Option 3: Modular Monolith - 1

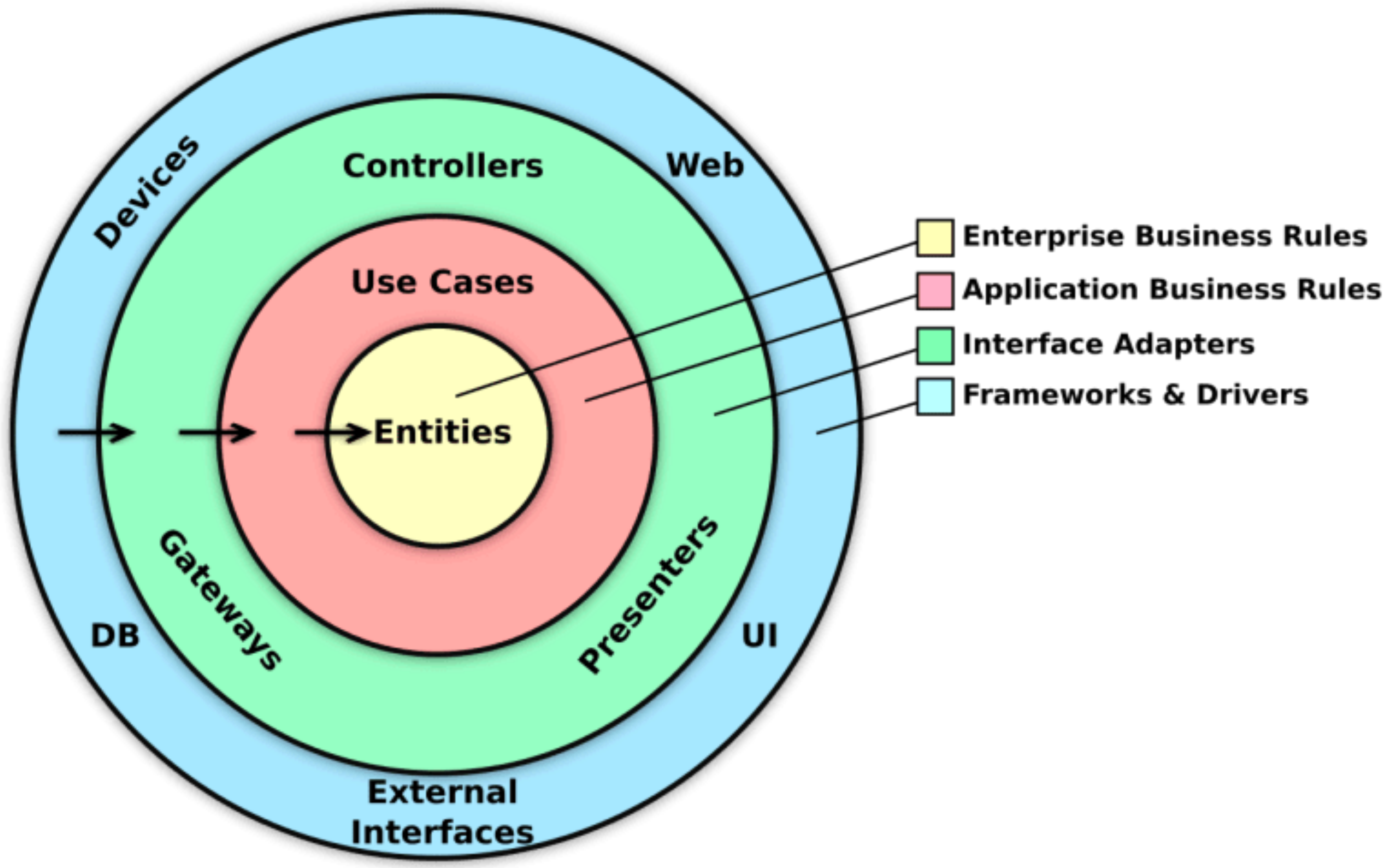
Option 3: Modular Monolith
CI/CD
Static Content
Build
ADO
Git Repo
Config
Orchestration
Production Region/Zone A
ASP.Net App Service (Modular Monolith)
Virtual Network
Sockets
Other business logic as needed.
GeoSpatial Module
Monitoring Module
Inspection Module
NDT Report Module
Pipe Metrics Module
Assement Module
ML Module
PDF Module
Email Module
Firewall
HTTPS
Azure Front Door
Load Balancer / CDN
Browser
HTTPS (REST)
GIS API
SB
After 3 Failures
DLQ
Notifies Service
GeoSpatial API
Monitoring API
Inline Inspection API
Non-Destructive Test Report API
Pipeline Metrics API
Assessment API
Machine Learning
PDF API
Email API
Pipeline Database
Machine Learning
Blob Storage
Azure Comm Service (Email)
Connects to all needed Modules
Monitoring IoT
Firewall
ASP.Net App Service (Support)
HTTPS
Support Browser

Design Option 3: Modular Monolith - 2

- Option 3 combines aspects from Option 1 and Option 2. In this case the various responsibilities of the Web Server are separated into individual modules. The design approaches for these modules are detailed as Options A, B, and C. Option 3 is unique in that each module could use one of the various options (A, B, C) as needed. As such a mixture of Options A, B, and C could be present in the Web Server at the same time. These options are detailed further into the presentation. The Modular Monolith Web Server is a single executable, running from an Azure App Service instance. If desired, this ASP.Net application could also run from a container.
- In front of the Web Server is the Azure Front Door Service instance. This is a load balancer and a CDN, allowing any static content to be served to the user, as well as balancing user requests between Azure Regions or Zones.
- The Web Client would be designed to request any Geospatial information, for example image tiles from a GIS service. Suggested services could be ArcGIS Online, or Azure Maps depending on user needs and pricing.
- Pipeline Monitoring is likely performed by third party IoT services or devices. Calling out to Monitoring services or devices is performed by the Web Server or a Serverless Function. As the service or device may have specific connections such as custom TCP/IP or UDP protocols needed to upload data from the service or device. As such this portion of the design would be handled by the thick Web Server, instead of in a separate VM as in Option 2.
- Data is stored in a standard relational model, any semi-structured or unstructured data is stored in blob storage. The use of a data lake is likely unneeded given that most data will be from company designed sources and thus standardized.
- Emails to users (such as for generating pdf reports) are sent via an Azure Communication Service instance.
- The entire application is within a virtual network. Each region / zone has its own isolated virtual network.
- The application is deployed to a specific region / zone via an Azure Kubernetes Service instance. Note that AKS operates at a region level. Therefore each region will have its own AKS instance, and the DevOps infrastructure will have to be designed to connect to each instance.
- Not pictured are the DevOps develop and QA staging environments which would be deployed to first, prior to deployment to production. Also not pictured is the approach to blue green deployment, this would likely be done in the same region / zone using separate resource groups. Option 3 has a potential to be similar to either Option 1 or 2 or somewhere in the middle in it’s impact on DevOps. This impact largely depends on the chosen approaches to Options A, B, and C for the modules in the system. Notably the more modules that use Option C, the higher the risk of a difference existing between the concrete implementation logic in the Web Server and the corresponding logic in the associated Serverless function. As such Option C requires more testing of versioning and likely impacts the amount of unit tests and integration tests needed to ensure minimal risk is achieved.

Option 3 and Clean Architecture

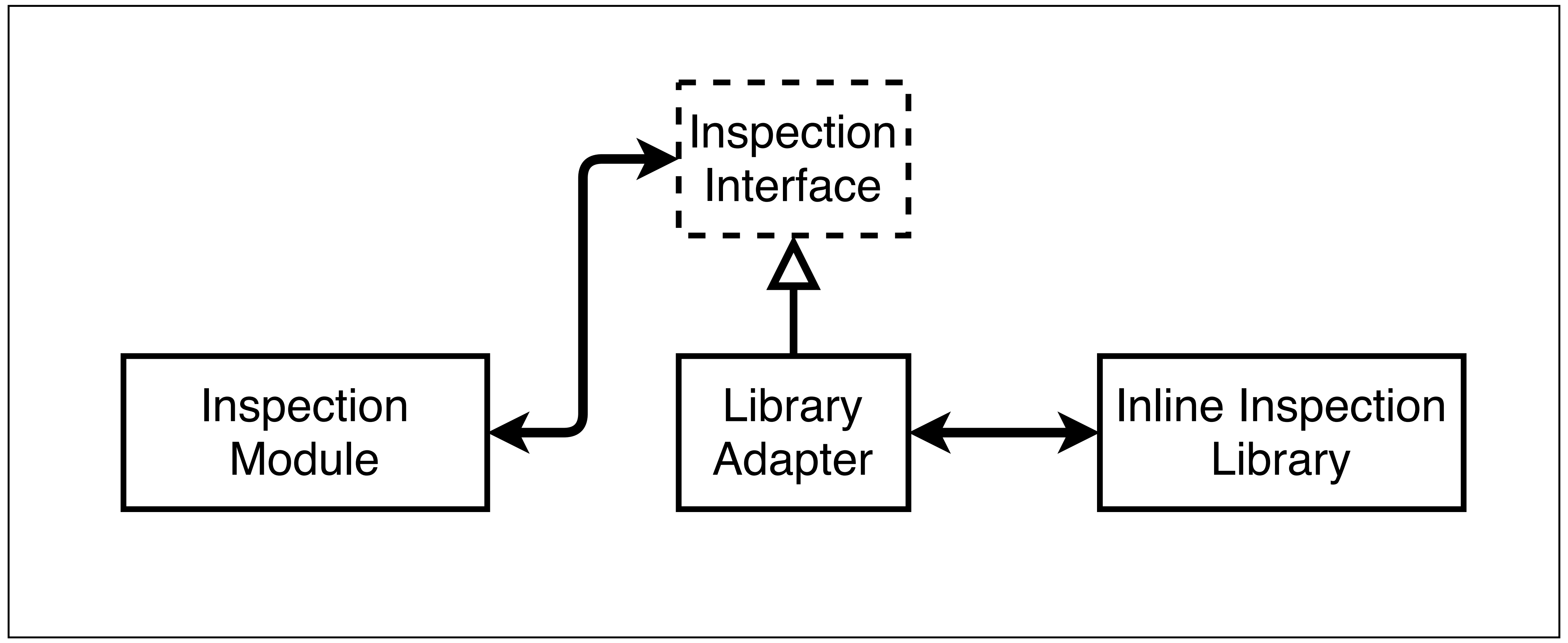
- Option 3 is based around an architectural design approach called "Clean Architecture" created by Robert Martin.
- Clean Architecture is based around the SOLID design principle of Dependency Inversion.
- The Dependency Inversion principle states the following:
  - "High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g., interfaces)."**
  - "Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions."**
- Effectively the Dependency Inversion principle passes dependencies into a module, where the dependencies inherit from an interface that the module understands.
- This approach decouples the implementation details and allows one to easily exchange implementations in a system with minimal impact. I've seen this used successfully in the past and it has an added benefit of making unit tests easier to design and implement.

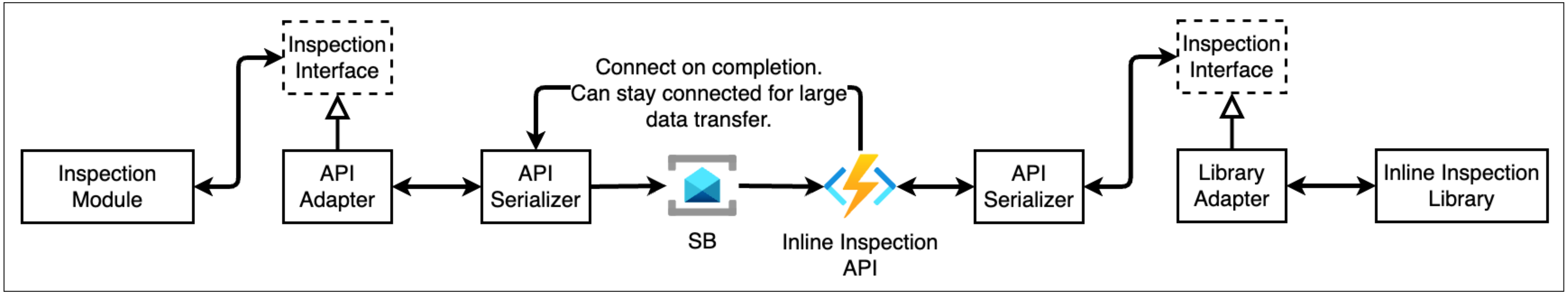


*Clean Architecture design image, copyright unknown. Original derived image, copyright Robert Martin.
** https://en.wikipedia.org/wiki/Dependency_inversion_principle

Option 3: Dependency Inversion with Options A & B

- In Option 3A (to the right), the design uses DI to allow the Inspection Module (as an example) to reference an abstract interface, while being passed in the concrete library. This is a compile time approach.
- In Option 3B* (below), the design uses DI to allow the Inspection Module to call out to an Azure function that contains the concrete implementation. This is a compile time approach.

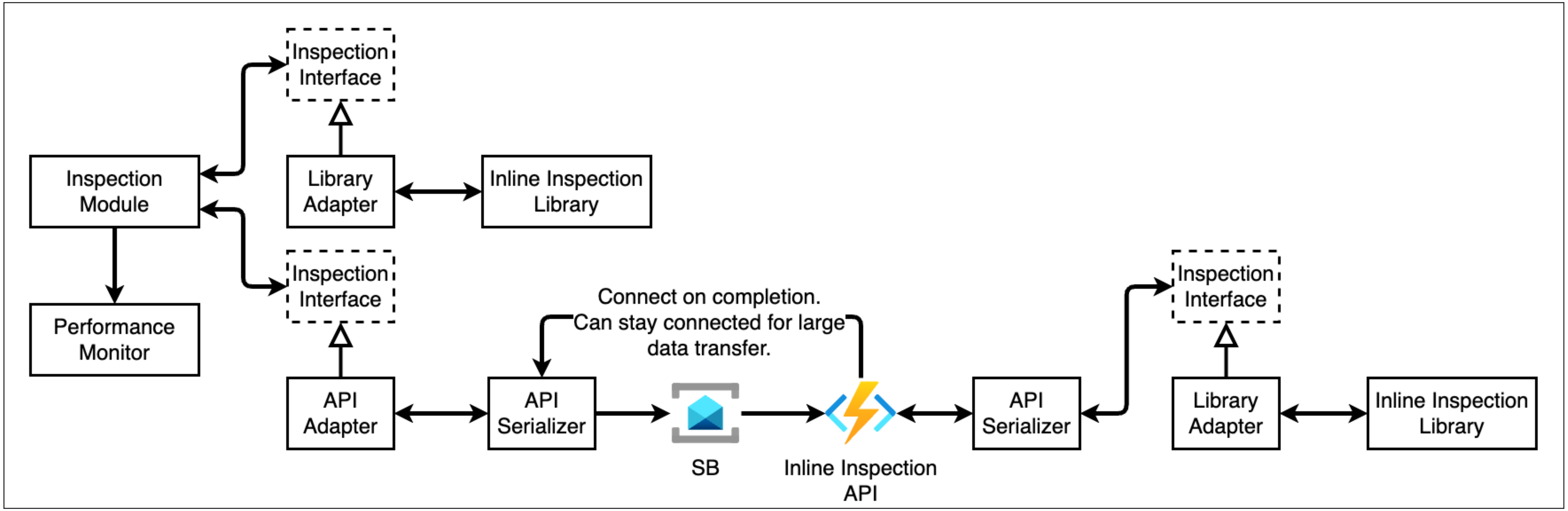




* Some delay in application logic will exists due to network calls to Serverless functions, which may affect application performance if such functions are called in low quantities due to function cold starts. As such lower overall performance may occur. The design of a module will need to take into account failures, delays, or other issues that could "freeze" any calls into the module. As such its recommended a module work asynchronously from the rest of the code base.

Option 3: Dependency Inversion with Option C

- In Option C, the design uses DI to allow the Inspection Module to dynamically either perform the work locally in the Web Server or to call a remote Serverless function. This is based on information determined in the Web Server's performance monitor. It could also be done via measuring the number of requests to a module over a given time frame.
- This approach allows for great flexibility in scaling, and allows one to create a Hybrid Serverless application. Over the long term it may be determined that specific functionality is always performed via one of the code branches. In this case, removal of the dynamic capabilities is suggested to lessen code maintenance.



Assignment Question 2

- Question: “What security concerns can you identify?
- As previously mentioned, I’m not a security expert. Here are my recommendations based on my past experience and research performed for this assignment.
- Security concerns are nominally based around the various technology choices and their usage in the design.
- In each design one should determine which security responsibilities the team is taking onboard with the chosen design and attempt to know what risks are present to the system.
  - For example if the team runs the website out of a container instead of using an Azure App Service instance, then the team has chosen to take up the responsibility of Operating System security.
  - Where as when one uses a Serverless function or similar Platform as a Service technology, Operating System security would be the responsibility of the Platform provider.
- When evaluating these goals its important to design the system following the principles of Least Privilege, Defense in Depth, and Zero Trust.

Assignment Question 2: General Recommendations - 1

- In general here are the areas of concern across all three options.
- Data Access:
  - All three designs do not rely on Azure ID or other ID platforms for credentials. Instead the credentials are handled by the Web Server. It could be desirable to use a third party to handle credentials and transfer the risk to another provider, but this will increase system costs and complexity.
- Application Security:
  - Broadly speaking much of application security revolves around programming failures. Use of best practices from the OWASP* top ten (injection, cross-scripting, etc.) and CVE** top 25 (out of bounds write, use after free, etc.) can substantially reduce security risks.
  - Additionally, choosing memory safe languages (such as those using a garbage collector or a borrow checker) can reduce programming failures, increasing security.
  - Serialization approaches are also an important consideration. In general limiting object serialization to approved types, is highly recommended, as this prevents an attacker from creating instances of objects that could provide the attacker system access or perform an attack when deserialized.
  - Finally minimizing dependancies can reduce the risk of including infected libraries. When libraries are used, as they often need to be, ensure that library binaries are stored locally in the Git repo or at least any project settings are hard-coded to a specific version, instead of wild-staring to get the latest copy from online sources. This approach can also protect one against new programming errors in libraries, leading to a more stable service and a better user experience.

* https://owasp.org/www-project-top-ten/

** https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html

Assignment Question 2: General Recommendations - 2

- Middleware:
  - For Middleware it's proposed that configuration of the software applications or platforms is critical to their secure use. Examples of configuration failures are the various leaks of personal and corporate data via S3 or similar services due to incorrectly configured storage buckets. It is recommended for the team to throughly review security settings and configurations for third party services. Finally, following the principle of Defense in Depth, it is recommended to encrypt (salt + hash), where possible, data stored in Middleware services, so that even if the data leaks, it is not usable. This is an absolute must for any user credentials, such as user names or passwords.
- Network Security (Virtual Network, API End Points)
  - Network security is still a topic for me to research in detail. Generally my understanding is to limit firewall port access, and ensure that the network is segmented to limit access even if a system is compromised. Communication should be performed using some form of encryption, likely TSL 1.3 or newer.

* https://owasp.org/www-project-top-ten/
** https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html

Assignment Question 2: Option 1

- In Option 1, the design focuses on a monolithic approach minimizing the number of components in the cloud design and uses an App Service. Likely this means that the team is responsible to secure:
  - Data Access (DB / Blob Storage / User Credentials)
  - Application Security (Web Client / Web Server / Machine Learning)
  - Middleware (DB, Azure Comm Service)
  - Network Security (Virtual Network, API End Points)
- In general this design is more vulnerable than Option 2 to attack as it consists of a single application. This means that if an attacker can access part of the application, they may be able to get access to memory across the entire application. This could lead to data and credential leakage.
  - On the other hand a single application could be easier to statically analyze for security defects. It would also reduce the amount of network traffic between system components and therefore reduce the chance for man in the middle attacks or insecurely configured services, due to the lower number of services to configure.

Assignment Question 2: Option 2

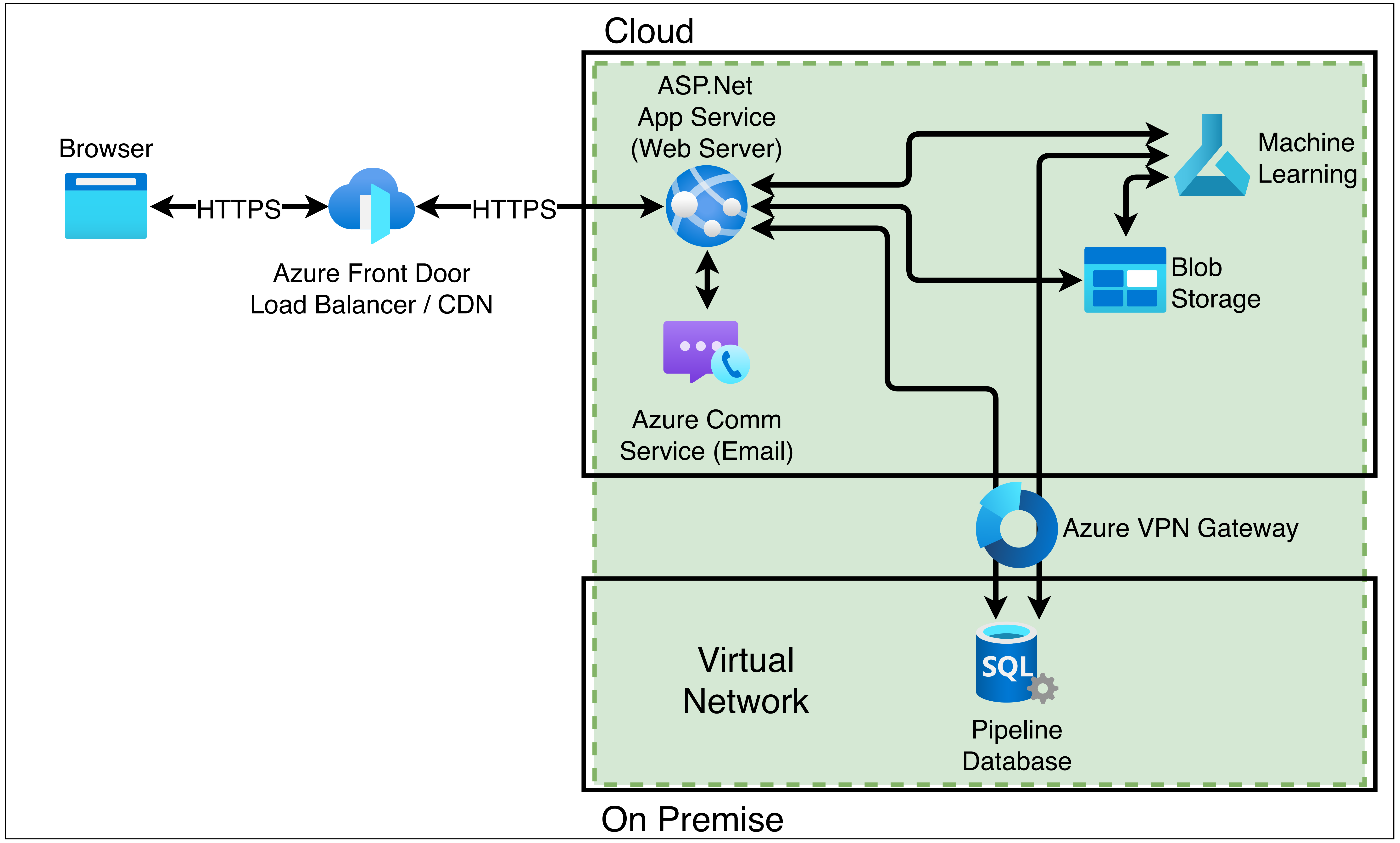
- In Option 2, the design focuses on minimizing the Web Server by constructing it as a pass through BFF and relying on application logic living in Serverless functions. Yet, this has little impact on the overall team responsibilities:
  - Data Access (DB / Blob Storage / User Credentials)
  - Application Security (Web Client / Web Server / Optional VM App / Serverless Functions / Machine Learning)
  - Middleware (DB, Azure Comm Service)
  - Network Security (Virtual Network, API End Points)
- In general this design is less vulnerable than Option 1. Its Web Server has limited functionality, which means even if an attacker gains access to the memory of the Web Server, the attacker has limited access to other data.
- Option 2 is more complex from a services, infrastructure, and network standpoint. As such there are more avenues available to exploit and attack. In addition if the Serverless functions are written in different languages, this also increases the attack surface of the application and its vulnerability to malware via libraries.

Assignment Question 2: Option 3

- In Option 3, the design ends up as a middle ground between Options 1 and 2:
  - Data Access (DB / Blob Storage / User Credentials)
  - Application Security (Web Client / Web Server / Serverless Functions / Machine Learning)
  - Middleware (DB, Azure Comm Service)
  - Network Security (Virtual Network, API End Points)
- It also has a mixture of the strengths and weaknesses of Options 1 and 2. The less the modular design uses Serverless functions the more applicable static analysis tools are to find vulnerabilities and the less infrastructure there is to create an attack surface. However, this also means if an attacker does gain access the more likely there is for a larger leak of data.
- As Option 3 moves towards the use of more Serverless functions, it's risk profile grows similar to Option 2. This lessens the risk of any specific attack, but increases the overall attack foot print of the entire system.

Assignment Question 3 - On-Premise Database Cloud Integrations: Option 1

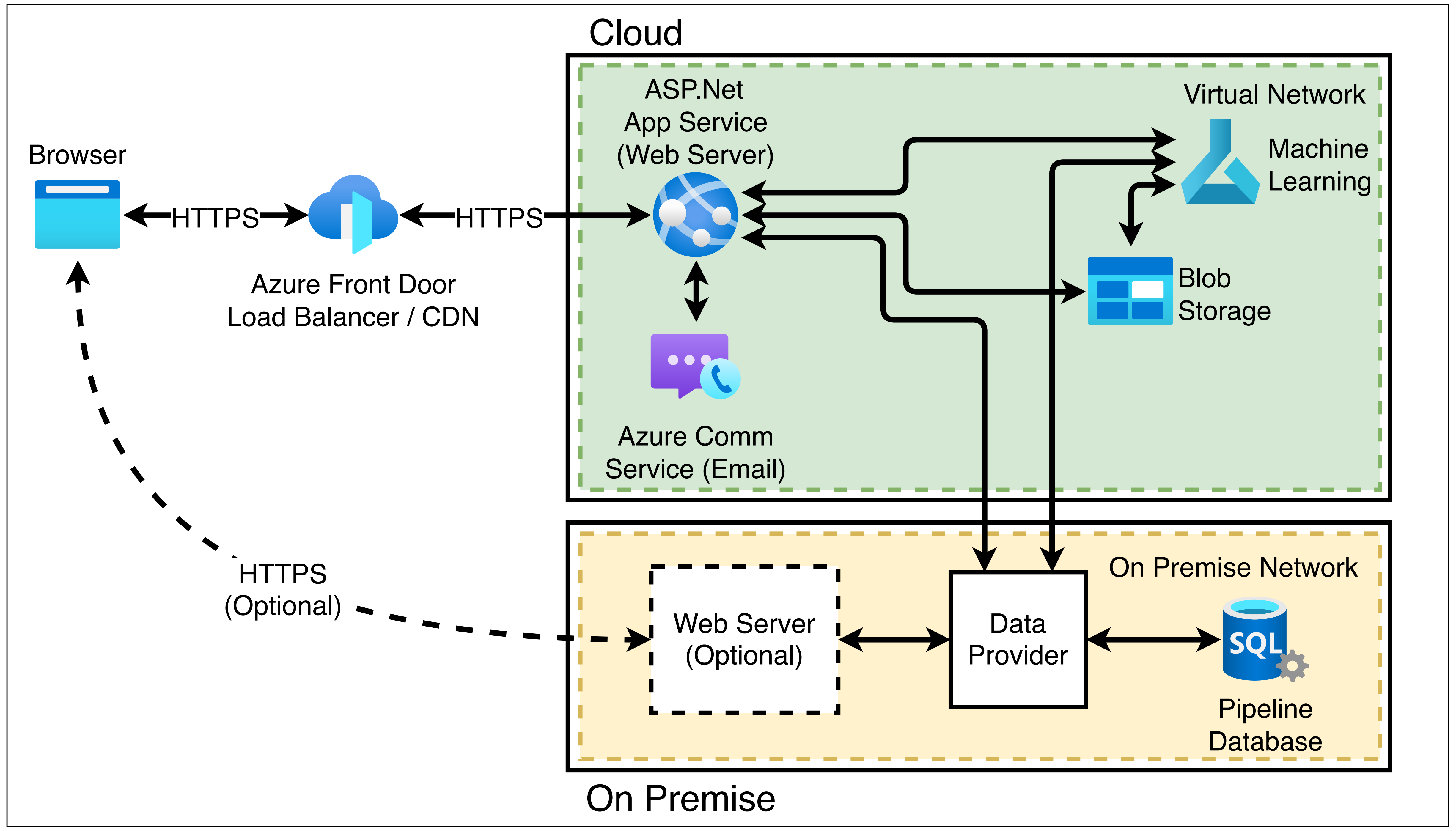
- Question: “How might you connect an on-premise database with the cloud system?”
- One option for integrating On-Premise resources is to use a shared VPN between the cloud services and the On-Premise resources.
- This can be implemented using the Azure VPN Gateway service.



On-Premise Database Cloud Integrations: Option 2

- Another option is to treat the On-Premise resource the same way as a third party API via an application that acts as Data Provider between the caller and the database. This ensures the database is not open to remote connections.

- In addition one could host a web server On-Premise to the Data Provider to allow clients to make requests to On-Premise resources. This adds challenges around scaling and integrating any credentials between the Cloud and the On-Premise resources.



Assignment Question 4

- Question: “What tradeoffs do you see between storing data in the cloud vs on-premises?”
- **For Cloud:**
  - **Advantages** include nearly unlimited* storage, easy regional and local backups, ease of data access from multiple cloud services and reduced hardware maintenance costs.
  - **Disadvantages** include increased storage costs, limited configurations, physical disk access is not allowed, high data transfer fees, increased potentially security risks, an increased difficultly in accessing data from On-Premise services, and uncontrolled inter-system latencies.
- **For On-Premise:**
  - **Advantages** include reduced storage costs, increased security, physical disk access is allowed, no data transfer fees, unlimited configurations, ease of access for On-Premise services and controllable inter-system latencies.
  - **Disadvantages** include limited storage, higher effort for regional and local backups, increased difficulty in accessing data from Cloud services, and increased hardware maintenance costs.

* Azure limits storage accounts to 5PB. This limit can be increased on request. Research suggests that AWS has unlimited storage, but has a 5TB limit per storage bucket.

www.AaronViviano.com - contact@AaronViviano.com

Thank you for your time, attention, and the opportunity.

Questions and comments?