Modernizing “Heartland Escapes” to a Cloud-Hosted Infrastructure

Aidan Polivka

CS630 – Modern Operating Systems

Colorado Technical University

November 5, 2023

Table of Contents

Project Outline

“Heartland Escapes” is a bookstore in Nebraska. They've been in Lincoln for years, opening their first store in 2010 and their second in 2019. Their recent popularity is thanks to a fresh marketing approach on social media platforms like Instagram and Tik Tok. Events like the 'school’s out reading program,' author meet-and-greets, and Halloween story readings have contributed to their success. They're planning to expand to Omaha and Grand Island.

Since their 2009 opening, 'Heartland Escapes' has made minor technology upgrades, like increasing disk space, RAM, and upgrading the processor. Their current system, hosted at their first store, includes a Windows 2008 server with an Intel Core i7-8700 processor. This processor has 6 cores and 12 threads, running at 3.2 GHz, with an L2 Cache of 256 KB per core and an L3 Cache of 12 MB. The server has 8 GB of RAM and 8 TB of Hard Disk space, with 3.5 TB in use. It hosts a point-of-sale system, an inventory system, and a public website, all written in .NET Framework 4.8 web forms. It also includes a Microsoft SQL Server for the inventory and accounting databases. A network of 8 machines between the two locations uses the Point-of-Sale web application and Inventory service. The website interacts with the inventory service so users can search for books in each store from their homes. With two new stores and a growing customer base, 'Heartland Escapes' plans to upgrade their system for scalability and ease of management by moving to the cloud.

Given their Microsoft-centric system and growth expectations, I recommend migrating their infrastructure to Azure while keeping an on-premises server as a backup. This shift to the cloud supports hardware scalability for increased traffic without requiring dedicated CPUs. Azure, a Microsoft product, aligns with their .NET applications and MS SQL Server database management system.

I suggest configuring Azure into two resource groups: one for data and infrastructure, and the other for applications. All resources should be in the us central region. In the application storage container, there will be three app services: one for the Point-of-Sale web application, one for the Inventory Service, and one for the public website. Azure App Services are like containers with a base operating system. In this case, I recommend running all services on a Windows system for simplicity. The app services would use the Premium v3 P1V3 hardware tier, which includes two vCPUs and 8 GB of RAM per instance, with the ability to scale up to 30 instances. In the data and infrastructure group, there would be an Azure SQL Server hosting both databases, and a virtual network (vNet) that manages firewall rules and security. A vNet Gateway would allow users to access the system via VPN. The public website would be the only part of the system outside of the vNet, accessible only through the gateway. The public website's IP address would also have access.

OS Processor and Core

Migration Benefits

Migrating to Azure offers several advantages. While it may seem that the new hardware is less capable on the surface, that's not the case. Each service instance has access to 2 virtual CPUs, and there are 30 available instances. This means the system now has access to 60 virtual cores, compared to the original 6 cores it shared. Additionally, Azure enables easy scaling of application resources, allowing administrators to adapt to system growth with a few clicks. The scalability of Azure App Service instances is advanced, with rules for scaling based on various metrics (Ed Baynash, 2023).

Distributing the system into individual Azure components localizes the load on individual services, reducing the impact on the system. Scaling is also a key feature, where each service can be configured to have a maximum of 10 instances, adjusting the instance number based on performance metrics like CPU usage and RAM. This configuration ensures that the system can manage a substantial amount of traffic.

Another benefit is increased system reliability. Services are now decoupled from a single machine and are managed by Microsoft on an operating system level. This eliminates single points of failure and allows for seamless transition in case of instance failures. Microsoft takes responsibility for security updates, reducing the burden on system maintainers. These updates are performed with minimal downtime. Regarding downtime, the P1V3 service tier boasts a 99.95% yearly availability, potentially surpassing the on-premises system's uptime, depending on 'Heartland Escapes' machine maintenance practices.

Distributed Systems and CPU Scaling

This upgrade is by all definitions distributed and virtual. What I mean by this is, the individual applications that were originally hosted on that single machine are now distributed into their own containers with exclusive access to a pool of processing, memory, and security resources. Azure being used as a Platform as a Service allows for this to be done very quickly and easily through their Azure Portal UI. These distributions are virtual as well. As stated in the project outline, Azure App Services are really containers, running on the base operating system defined by their App Service Plan which is a virtual host of the containers. At the heart of every cloud service is really a massive server farm with thousands of processors, even more cores, terabytes of RAM, and petabytes of disk space distributed into many virtual machines.

Because of this, there is really a lack of control around the specific processor used for your deployed Azure services. What is available to a user of Azure is the ability to upgrade the number of virtual CPUs. There isn’t much information online about how many cores are available in an Azure vCPU, they use a different unit of measurement for performance since multiple vCPUs can span across the same physical core. You can increase the processing power depending on the tier of your App Service

Plan, but I haven’t been able to find a direct comparison of the Azure Compute Unit (ACU) to existing processors, or how that ACU is calculated (Micah McKittrick, 2022). So, at my level of understanding, to “upgrade a CPU” you really need to upgrade the App Service Plan that hosts your App Services. Because of how obfuscated the user of the cloud service is from the details of the individual vCPU, the decision of whether to upgrade the App Service Plan tier (effectively upgrading the vCPU) is dependent upon cost of the upgrade, how much additional performance is expected from the upgrade, whether that performance upgrade is crucial to the design of the system, and bureaucratic approval. There isn’t much additional need for migration or security concerns since that’s all baked into Azure as a service. Since Azure has advance logging and performance reporting, testing the upgraded plan would mainly consist of monitoring.

Scheduling Algorithms

The “Heartland Escapes” modern environment is a distributed virtual environment hosted by Azure. The migration plan is to use a containerized approach by taking advantage of the platform as a service capability of Azure App Services. Because the base environments of App Service containers aren’t configurable, the proposed migration plan won’t allow for control over the scheduling algorithms. With that in mind, this section will discuss hypothetically which scheduling algorithms would be best for this environment if it were a configurable option. Because App Services are all hosted under an “App Service Plan” (ASP) and the ASP dictates the operating environment, this section will assume that the best scheduling algorithms should be optimal for all services deployed under the ASP.

Multilevel Queue

The primary algorithm that should be used in this environment is the multilevel queue scheduling algorithm. In this algorithm each process is assigned a priority level, typically decided by process type (interactive user process or background batch process), execution time, resource utilization, completion deadline constraints, and many other variables. This scheduling algorithm consists of many queues, one for each priority level. After the priority of a process is determined, it’s added to its respective queue. Each queue has its own scheduling algorithm that’s configured based upon what priority of processes it’s managing. Ideally this algorithm would also take process age into account when assigning priority and reprioritize the process at preemption to mitigate the possibility of starvation (Multilevel Queue (MLQ) CPU Scheduling).

The biggest advantage of this algorithm is how efficiently it manages processes with different priorities and properties. Critical tasks are always processed with high authority, and low priority tasks are put behind the high critical tasks. This is also a highly responsive scheduler. If you consider that interactive user processes are typically high priority tasks, the responsiveness of the system from the user’s perspective would be very high. The preemption of low priority processes in favor of high-priority processes is also a contributing factor to the responsiveness of the system (Multilevel Queue (MLQ) CPU Scheduling). This responsiveness would be important for processes like inventory requests and financial transactions, and the aging factor of the low priority tasks should ensure that batch processes like scheduled reporting aren’t starved of resources or importance. Additionally, this algorithm should be great at providing equitable resource distribution to each virtual container.

A disadvantage of this algorithm is the possible starvation of low-priority tasks. Most multilevel queue algorithms resolve this problem by reassigning priority to processes after quantum completion with aging being factored into the priority calculation. So, if a low priority task has sat for an extended period in the low priority queue, it can elevate itself in priority based upon much it has aged in the system. Another disadvantage would be the complexity of configuration. There is a lot to manage when considering assigning priority to processes (especially when considering aging policies), configuring sub-queue scheduling algorithms, and preemption policies. There is also a lot of context-switching in this scheduling algorithm between the different priority queues and preemption of low priority tasks, which can lead to excessive overhead. All these complexities can be worsened by the virtual environment (Multilevel Queue (MLQ) CPU Scheduling).

Round Robin

The secondary scheduling algorithm that should be used within the multilevel queue algorithm is the “Round-Robin” scheduling algorithm. This algorithm would be used to manage the processes in each of the multilevel priority queues. The Round-Robin algorithm has two major components, a queue and a “Time Quantum” (also known as a “Time Slice” or “Time Slot”). The time quantum is a unit of time typically in the size of milliseconds, and it’s the allotted time that the process can be acted upon before it gets sent back to the end of the process queue. This scheduling algorithm gets its name from the cyclical nature in which it manages processes. If the process can’t be completed during the time quantum, then it must wait until the other processes have had a turn to be operated upon. In the context of the multilevel queue algorithm, the high priority Round-Robin queue would have a short quantum so that each high priority task would be operated upon quickly. The low priority Round-Robin queue would have a long quantum to reduce operational overhead and state memory management (Stallings, 2018, p.409).

This scheduling algorithm is well liked because it’s fair. Every process gets an equivalent amount of CPU attention before it’s preempted. This fairness also typically results in a quick response time for a process to be acted upon since long running processes are limited by the time quantum. A short quantum would result in tasks moving quickly through the queue, which typically means an even faster response time (Stallings, 2018, p.409). This fairness is amplified by the priority assignment provided by the multilevel queue algorithm, making high priority tasks more responsive while also ensuring low priority tasks aren’t neglected. It also heightens the multilevel queue algorithm’s ability to effectively distribute resources to virtual entities.

This algorithm also comes with its faults, some mitigated by using it in conjunction with the multilevel queue algorithm. It requires a lot of overhead for the processor to context switch since it needs to save the state of the current process and load the state of the next one (Stallings, 2018, p.409). A short quantum will exacerbate the overhead problem since processes would have a higher probability of being preempted, and this strain can be magnified by needing to manage virtual resources. On its own the Round-Robin scheduling algorithm can result in starvation of critical processes, but this problem is nonexistent when Round-Robin is utilized as a secondary algorithm to the multilevel queue algorithm. Because each process is sorted into queues based on priority, critical tasks have greater authority in the system and are processed first.

There are independent variations of the Round-Robin algorithm that also attempt to resolve the starvation issue by assigning priority to tasks. This priority assignment gives high priority tasks either a more favorable position in the queue after the quantum is completed, an increased time quantum, or both. While these standalone variations of the Round-Robin algorithm are great, if implemented in this system they would only result in unnecessary complexity.

When discussing the disadvantages of the Round-Robin algorithm some would consider the unfair treatment of I/O processes as a major drawback. Since I/O-bound processes are blocked by their device, this could result in the process being pushed back to the end of the queue because the blocking device took up the processes available quantum for that iteration through the ready queue. This is often mitigated using the Virtual Round-Robin variation, which consists of an additional I/O auxiliary queue. After the process has a first pass through the ready queue it’s sent to its devices I/O Queue. Once the device is done with the process, it’s sent to the auxiliary queue. This queue is handled in a first come first serve manner and takes priority over the Round-Robin ready queue processing. The processes in the auxiliary queue get the same time quantum to complete as the Round-Robin ready queue (Stallings, 2018, p.410-411). This shouldn’t be necessary in our system since the App Services won’t have any I/O connected devices.

Multilevel and Round-Robin “Heartland Escapes” Impact

Given the advantages and disadvantages of the Multilevel Queue and the Round-Robin scheduling algorithms, together they should be a great fit for the “Heartland Escapes” system architecture. Both algorithms are beneficial to ensuring responsiveness, Round-Robin being overall responsiveness and Multilevel Queue being critical responsiveness. Working in tandem, these two algorithms should ensure quick delivery of financial transactions for the point-of-sale system and fast service of inventory data both to the point-of-sale system and public website. Because of the Multilevel Queue’s ability to set priority of processes based on process age, this should allow batch processes like reporting jobs to complete without risk of starvation. Considering that both the Round-Robin and Multilevel Queue algorithms are strong in their ability to distribute resources to virtual entities, they should be very strong together in their ability to host this containerized environment. However, these algorithms are both weakened by overhead concerns in processes preemption, so careful configuration should be considered when determining preemptive policies and Round-Robin quantum sizes between priority queues.

OS Concurrency Mechanism

With “Heartland Escapes” being hosted on a network of web services and APIs, the most important and prevalent concurrency mechanism that will be used in this environment is message passing. Each Azure App Service is going to need to make network calls to their neighboring services. The publicly accessible website will need to communicate to the Inventory API to deliver inventory information on available books at different stores. The point-of-sale system will need to communicate to the inventory API to request and update inventory data, and it will need to make calls directly to the accounting database to log customer transactions. Considering that there will be far fewer write operations on each database than read operations, the second concurrency mechanism that I’d like to discuss will be the critical sections.

Message Passing

Message passing is an essential concurrency mechanism in most distributed computing environments, and this is especially true in a containerized microservice environment like this simple implementation for “Heartland Escapes”. The only way for our distributed services and processes to communicate with one another is over network messaging. In this system, messages are transferred between machines via HTTP or RPC protocols. Message passing requires two parties, a sender and a receiver. This sending and receiving process needs some level of synchronization, which can be achieved in a few formats (Stallings, 2018, p.237).

1. **Blocking Send, Blocking Receive**. Both the sending process and receiving process stop and wait for the messaging mechanism to complete. This is common in tightly synchronized processes; you can see this in synchronous http GET request to an API. The “Heartland Escapes” website has a page that displays a list of books, and the book data is distributed by the Inventory API. The website client will make a request to the API and wait for a response so it can build that display.
2. **Nonblocking Send, Blocking Receive**. This is also a common messaging mechanism, where the sender doesn’t care what happens after it sends the message. This is also prevalent in web development. When a book is sold the Point-Of-Sale will make a request to the Inventory API to remove a book from inventory, but it doesn’t really care about the response from the API. So, the Point-Of-Sale system will continue operating, and the API will resume a waiting state after it performs its process.
3. **Nonblocking Send, Nonblocking Receive**. In this case, neither the sender nor receiver is waiting to process other messages. You can see this in an event driven architecture, where the sender might send an event to any listening service. That listening service might be operating on a queue of messages, and it’ll operate on the message distributed by the sender when it’s that messages turn. The sender won’t wait for a response, and the receiver isn’t necessarily waiting for a message. I don’t believe that there is a good example of this in the context of “Heartland Escapes”, but the option is out there for future development.

Any messages passed over a network are subject to network related synchronization complications like network latency, message delivery reliability, race conditions between machines, load balancing, and fault tolerance. If one node is making a request to the inventory API and is having networking issues, this could result in message ordering complications or the loss of the message entirely. Thankfully there are some complications that are inherently handled by Azure in the Platform as a Service architecture, like fault tolerance. Azure handles this by multi-regional redundancy.

There are infrastructure options available in Azure to increase message resiliency like Azure Arc-enabled Open Service Mesh (de Bruin, 2023) and multi-region redundancy, but this doesn’t handle all network failures. A couple common development practices in ensuring message resiliency are retries and circuit breakers. A retry policy is a configuration in your application that allows failed messages to be re-sent over a configurable duration or retry attempt limit, with a configurable wait time between retry attempts. Because we’re working with .NET applications, if we migrate our applications to .NET 7 these policies are available out of the box (Montemagno, 2023). There are also entity framework configurations that allow for message retries to our database management system (Montemagno, 2023). This message retry capability provides further advantages in fault tolerance in the event of receiving node failure. If a node was not online or incurred some other failure, the re-attempted message would be directed to an effective node.

Circuit Breakers allow for a pause between sets of retry attempts, which is useful in the scenario that the requested resource isn’t returning a response due to heavy load. This is configurable in a similar way to retry policies in .NET (Caron, 2018), but in our scenario with “Heartland Escapes” I don’t believe that circuit breaking is necessary due to the number of instances that are available for each service. Some infrastructure tools like service meshes have these retry and circuit breaking policies incorporated, but I couldn’t find any information on this for Azure Arc.

Message ordering complications, duplications, and race conditions can also be mitigated using proper concurrency conscious development practices like idempotent development and event sourcing. Idempotent development is a practice that specifically handles duplicate messages. An example of idempotent development in the context of “Heartland Escapes” is if a duplicate message is sent to purchase a book. Since the first message will log the sale of the book with the message Id, we’ll check to see if the book was already sold and if the message Id matches the current message being processed. If it's determined that this message is a duplicate, we’ll log a message to the console reflecting the scenario and cancel the process (Hyett, 2023).

Event sourcing is a data practice that consists of storing entity events and their message data, then building the entity on the fly when it’s needed to be used for some process (Kuan). Since our system isn’t necessarily event based, message ordering shouldn’t be an issue due to the nature of the system. If we wanted to turn “Heartland Escapes” into a fully fledged e-commerce system with orders, delivery systems, etc., then event sourcing would be a more warranted practice. These two concerns can also be managed with infrastructure tools like message brokers, service meshes, and event meshes.

Critical Sections

Since “Heartland Escapes” SQL Server environment is going to be hosted on a windows system, we have access to concurrency mechanisms like Critical Sections. Critical sections operate similarly to other synchronization mechanisms like mutexes and semaphores but differ in that only a single process can access a critical section at a time. Critical sections are preferred in some cases because they are a more efficient operation than a mutex or semaphore (Stallings, 2018, p.300). According to GeeksForGeeks (2010) there are three attributes to Critical sections that are integral to synchronization.

1. Mutual Exclusion. Only one process can enter a critical section at a time, which is like the functionality defined by Mutexes (which is short for Mutual Exclusion) and semaphores.
2. Progress. If the critical section is open for processing and multiple processes want to enter their critical sections, only the processes that are waiting can decide which enters the critical section. Processes in contention for a critical section can’t be deliberated forever, a process needs to be chosen in a timely manner.
3. Bounded waiting. There is a limit on how many times other processes can continue to enter a critical section ahead of a process whose requested access to a critical section, that way processes aren’t starved of access.

Critical sections offer a lot of benefits in terms of concurrency control like synchronization, prevention of race conditions, deadlock prevention, data integrity, thread safety, and mutual exclusion of shared resources. All these benefits are predicated upon the critical section being properly implemented and within a scenario that requires such a level of control. Deadlocks and starvation can all be results of poorly implemented critical sections, so they have the potential to be a double-edged sword. By nature, the concept of critical sections prevents race conditions and provides mutual exclusion in only a single process is allowed to operate on the critical section or have access to a critical resource at a time. This single process access also results in simpler synchronization. Another advantage to using critical sections is CPU utilization. Allowing processes to wait without cycling can result in a reduction of CPU utilization but increases memory overhead.

As foreshadowed in the previous segment, critical sections don’t exist without faults. Deadlocks are a concern in poor implementations of this concurrency control mechanism, and processes actively waiting on critical section access can result in high memory overhead. Poorly implemented critical sections can also result in process starvation, especially if the scope of the critical resource is large. Allowing only a single thread to access a resource at a time can severely limit parallelism as well, but the tradeoff is thread safety.

Critical sections used in code can be implemented using a locking mechanism. The operating thread requests an exclusive lock on a resource, and if it’s granted then it proceeds to operate. Once its operation is complete, it releases the lock for another thread to operate. In terms of “Heartland Escapes”, this would be incredibly useful in database operations. Consider a scenario where “Heartland Escapes” has a table for books in stock. This table has columns for book Id, book title, author name, genre, and quantity in stock. If an employee is selling a book while another employee is logging an increase in that book’s stock at the same time, this could lead to contention for that record in the table. However, if the operating threads can create a critical section around the individual record in the book inventory table, the risk of two processes writing to the same record at the same time is mitigated. Individual threads communicate amongst each other in the initial request for the critical section. If the request is denied, that means that another process is acting upon the critical section and the denied thread needs to wait for the resource to be open for operation. This locking mechanism can be managed by either a mutex or a semaphore in memory indicating that the resource is open or closed for use.

OS Security Risks and Mitigation Strategy

A critical step in migrating to a new operating environment is risk analysis and mitigation strategy planning. This migration effort for “Heartland Escapes” is no different, a thorough risk assessment was performed that supplied some areas of concern. This assessment evaluated networking, access control, data collection and resiliency, and malicious attack resiliency. Because “Heartland Escapes” new environment consists of containerized Azure App Services, we do not have access to the intricacies of traditional operating system risk assessments (which could be considered a risk in itself). Instead, we needed to assess the Azure Platform as a Service configuration, and how we can best secure this architecture from the outside world.

Compliance and Regulatory Risks

There are specific compliance guidelines for business domains with sensitive data, and failure to meet these requirements can result in large fines that could sink a business. There are specific regulations when considering financial data, any system that stores or transmits credit card information must adhere to the Payment Card Industry Data Security Standard (PCI DSS). If “Heartland Escapes” is storing credit card information in their accounting system, they must follow these regulations. Evaluating compliance to the PCI DSS is a fantastic way to measure the security of a system because it does not just include data security. There are 12 requirements to meet the PCI DSS (Baykara, 2022):

1. Install and maintain a firewall system to protect cardholder data.
2. Avoid vendor-supplied defaults for system passwords and other security parameters.
3. Protect stored cardholder data.
4. Encrypt transmission of cardholder data on open, public networks.
5. Protect all systems against malware, and update anti-virus software or programs.
6. Develop and maintain secure systems and applications.
7. Restrict access to cardholder data by business need to know.
8. Identify and authenticate access to system components.
9. Restrict physical access to cardholder data.
10. Track and monitor access to network resources and cardholder data.
11. Regularly test security systems and processes.
12. Maintain an information security policy which addresses information security for all personnel.

The most obvious way to mitigate risks in compliance is to simply follow regulatory policy (much easier said than done). Many of these requirements will be covered in later sections of this risk assessment and mitigation strategy, and they will be addressed as they appear.

Authentication and Authorization

Within the “Heartland Escapes” system we need to ensure that only authenticated users can access our services, databases, and the Azure cloud portal. We also need to make sure that any request to our App Services comes from an authenticated source. If we consider the Inventory API, we need to build security around all messages entering the API because there is external exposure from the public website. We’ll also want to make sure that we have role-based access control in our point-of-sale system so that only users with the proper role assignment can make sales. Role assignments will also be used for granting developers access to the Azure databases. Each service should operate under a service account so that the service isn’t reliant on the rights of the individual performing actions, but the rights of the service itself. For example, the Inventory API needs rights to perform read and write operations to the inventory database, but the individual making requests to the Inventory API does not.

The risk of not taking these actions is obvious: without authentication surrounding these services we run the risk of unqualified or malicious individuals corrupting data or performing actions they shouldn’t have access to. Thankfully, Azure supplies access to “Azure Active Directory” which allows for the creation and assignment of individual and service account roles. Individuals using the Point-of-Sale system will sign into the service using their Microsoft account, which can easily be added to the Azure Active Directory and given access by adding the account to an Azure Active Directory group. The point-of-sale system will check if the Microsoft account is within the proper Active Directory group to access the point-of-sale system at sign in. We can also create service accounts for all services and add those Active Directory accounts to permission groups for read/write access to the database. (Lipsey, 2023). The use of Active Directory groups and roles can be used to enforce requirements 7, 9, and 12 of the PCI DSS.

To secure inter-service communication, we’ll want to use authentication tokens between network calls. For a service to send a message to another service, it will need to request a token from target service with its service account. Once it receives an authentication token, it will send the request to the target service with the token in the message header.

Data Loss or Corruption

“Heartland Escapes” needs to have a way to bounce back from data loss or corruption. Here are a few scenarios that could cause data loss or corruption problems for “Heartland Escapes”:

- Ransomware attack: A Ransomware Attack occurs when a malicious individual or program gains access to a system and begins encrypting all files or data. After the damage has been done to the system, the individual request money for the decryption key. In some scenarios, they will steal sensitive data and threaten to sell it for added incentive to pay out the ransom.
- Physical attack on Microsoft Server Farm: Azure touts data redundancy and other data loss mitigation practices, but if something were to happen to the server farm, we would want to ensure that we have a physical backup somewhere outside of Azure just in case.
- Malicious Insider: An individual who has access to the database may become disgruntled and decide to truncate all tables.

Without a mitigation plan for these risks, “Heartland Escapes” could lose all their inventory data and all their accounting data. This would cost the company a hefty sum of money to replace, so it’s best to take precautions. To mitigate the risk of data loss or corruption, the primary course of action is provided by Azure. Azure offers data redundancy and backup scheduling options. If something happens to “Heartland Escapes” data Azure could possibly be held liable, however it would still be a massive headache to replace. If something unfortunate happened on Azure’s part, “Heartland Escapes” could reduce the fallout by using their existing on premises server to take weekly physical backups of the cloud hosted databases and all code repositories.

Whenever a system is storing sensitive information, that data should be encrypted in network calls and encrypted in storage. This encrypted data can be further enhanced by rotating encryption keys on a regular basis and running batch methods to update encrypted columns with the new key in rotation. This can create more peace of mind in situations where a malicious actor gains access to data systems because they won’t be able to read sensitive data without a decryption key. Encryption of stored sensitive data covers objective 3 of the PCI DSS. If “Heartland Escapes” is storing credit card information from transactions, they must follow up-to-date encryption standards. The industry standard for encrypting secure data is currently (Advanced Encryption Standard) AES encryption, which would be my recommendation for encrypting secure fields in the accounting database (GeeksForGeeks, 2021).

Network Security

The “Heartland Escapes” system is for the most part closed off to the outside world, but it does use the https network protocol to send messages between services. If this network traffic is not secured a malicious individual could easily intercept messages, inject messages into the system, and farm secure data that’s sent to the accounting database. As stated in the project outline, this system will be closed to the outside world with the exception of the public website’s public IP address and VPN access through use of an Azure Virtual Network and a Network Gateway. Specific firewall rules will be used to enforce access into the vNet. The firewall working in tandem with tokenized authentication between services and service SSL certificates should tightly secure network messages. Securing network messages with TLS 1.3 provides robust encryption and data integrity, ensuring that data in transit stays confidential and unaltered. This multi-layered security approach significantly reduces the risk of unauthorized access and data breaches.

The public website having access to request data from the Inventory API creates complication in network security. This could easily lead to a Distributed Denial of Service (DDoS) attack if the system is not configured to combat it. A DDoS attack is when a bad actor attempts to overwhelm an applications’ resources, making the application unusable for individuals that want to use it for legitimate purposes. The DDoS attacker would write a script to request inventory information from the public website repeatedly to the point that the inventory API is overwhelmed. This would result in the Point-of-Sale system and public website being unusable. To mitigate this risk Azure supplies DDoS protection (Bell, 2023) which supplies monitoring and reactionary steps to a DDoS attack. We can further mitigate this by rate limiting client calls to the Inventory API from the public website. We would implement this directly in the application code; the public website would monitor how many times an IP address makes a request for Inventory Data. If this reaches a call number threshold within a certain amount of time, that client will receive a timeout. These networking precautions cover objectives 1, 4, and 6 of the PCI DSS.

Third-Party Integration Risks

“Heartland Escapes” relies on a third-party payment processor in their Point-of-Sale system. With this relationship comes a risk in availability, because if the payment processor is offline then so is the Point-of-Sale system’s functionality. Communication to the payment processor also creates risk in data security because the point-of-sale system needs to send it sensitive data. There could also be a concern that updates to the third-party services’ APIs could cause compatibility issues with “Heartland Escapes” system.

These risks can be mitigated by a few means. The most important part of choosing a third-party vendor is to thoroughly vet them beforehand. As a part of this vetting process, we’d want to evaluate the up-time availability of their services, their message encryption options, API documentation support, and customer support availability. This is the most essential part of partnering with a third-party service. We could further mitigate the availability risk by having third-party redundancies. If our primary third-party payment processing service is down, we could have another payment processing service as a backup. To comply with PCI DSS’s second objective, we’ll want to evaluate any vendor-supplied defaults for system passwords and other security parameters and replace them.

Risk Mitigation Conclusion

So far, we’ve covered all the Payment Card Industry Data Security Standard objectives except for:

5. Protect all systems against malware, and update anti-virus software or programs.
11. Regularly test security systems and processes.
12. Maintain an information security policy which addresses information security for all personnel.

Since we’re using Azure’s Platform as a Service, protecting the system against malware and viruses is largely the responsibility of Azure. Since the operating system is out of our hands, there isn’t a clear way to download or upload malware to the machine hosting our app services, and we aren’t able to download our own anti-virus software to this machine either. Security system testing wasn’t covered in the other sections, but it’s a valuable way to continuously discover and mitigate vulnerabilities. There are tools available for vulnerability scanning and penetration testing that could be used to ensure our system remains secure over time. Objective 12 isn’t as much of a technological responsibility as it is a training and personnel policy responsibility. For “Heartland Escapes”, this could consist of regular training on common cyber-attacks, and training on how to physically manage customer information securely.

Future Considerations Using Emerging Technology

As of now, “Heartland Escapes” is completely focused on moving their platform to be hosted by the cloud. They have big aspirations from their exponentially growing client base, which is driving long-term conversations about having a fully featured e-commerce system. This system would allow for online book purchases, shipping, and competitive pricing matching models. There are a lot of emerging technologies that we could plan to implement to achieve this goal, but it is out of scope of the migration effort. This section is strictly a thought exercise on future work that could grow the “Heartland Escapes” enterprise and is distinct from the modernization effort.

E & M-Commerce

Leveraging an e-commerce platform that is mobile responsive could greatly increase value for “Heartland Escapes”, although it is a massive undertaking. Giving repeat customers the ability to order books online for in-store pick-up would have been incredibly useful during the early days of the Covid-19 pandemic. Though another economic shutdown isn’t foreseen, it would bring some peace of mind to the “Heartland Escapes” enterprise. This could also open the door for direct shipping so that customers don’t have to leave their homes to get new books. “Heartland Escapes” has also been considering offering monthly book subscription box where avid readers could pay $50 dollars a month to receive three books of their choice (from a list of options at a discounted rate) along with some “Heartland Escapes” merchandise. This is all assuming that “Heartland Escapes” continues their growth trajectory and can afford to operate on such a large scale. Changes like this would blow the current local giant “Barnes and Noble” out of the water. Thankfully, there are some third-party services that can make this dream a reality.

The current reigning champion of online e-commerce third party integrations is Shopify. Shopify is a platform that offers e-commerce management structures, like product management, product inventory, user account management, payment processing, and shipping. They allow the sale of products from multiple platforms including the web, mobile apps, social media, permanent stores, and popup stores. (Shopify) This “Software as a Service” platform fulfils all “Heartland Escapes” requirements of an e-commerce partner and allows for flexible pricing models depending on hardware need. They also offer a .NET software development kit for easy integration into existing applications, or templates to stand up a new e-commerce website quickly. The implementation that I’d suggest for “Heartland Escapes” is to slowly integrate Shopify into their public website, starting with the payment processing and inventory capabilities. A consideration to take when partnering with Shopify is that they don’t offer data backups, so if there is a data loss it is the responsibility of the consuming enterprise to repair it. As stated in the risk assessment, it’s important to fully vet a third party before contracting with them.

AI Driven Book Suggestions

Another feature that is desired by “Heartland Escapes” is the ability to recommend books based on purchase history. For a lot of readers, they tend to have favored authors, genres, writing style and reading complexity. In the past, to get an educated recommendation for their next book, readers would need to hear about books through word of mouth or look at the best sellers list. As the world has moved online, there should be a way for readers to achieve this by reading history. If “Heartland Escapes” partnered with major publishers to retrieve data about published books and digitally readable copies of the book, they could train an algorithm to find like-novels. Then, as users read a book, they could rate how much they enjoyed it in the “Heartland Escapes” e-commerce website and get recommendations based on the books they enjoyed reading. This would benefit both “Heartland Escapes” and partnering publishers by increasing sales and marketing exposure.

Tensor Flow is one of the largest machine learning platforms currently used in the tech industry. I personally am not experienced enough in machine learning to provide knowledge of training a machine learning model, but I can offer insight into integration with the Heartland Escapes environment. Tensor flow models are typically trained in python, and trained models can easily integrate into C#. After the data from publishers is trained into a machine learning algorithm, input parameters can be sent into the model trained by Tensor flow and the resulting book recommendations could be displayed on the public website as the first thing a signed in user sees. This “trained model” is pulled from a .pb file, so my recommendation would be to put this file on a network drive or cloud storage bucket that’s highly secured. This way if a new publisher agrees to send book data to “Heartland Escapes”, they can re-train the model and publish it to the storage location without needing to re-deploy their website. (TensorFlow - Creating C# Applications using TensorFlowSharp, 2019)

Chatbots

In the current e-commerce industry, chatbots have become a popular way for users to easily navigate a website and get answers to questions without needing to call customer support. Previously this need was fulfilled by websites having a “Frequently Asked Questions” page. A chatbot provides more accessibility to these answers by having a floating button on the screen regardless of the view the user is on. “Heartland Escapes” is concerned with customers clogging phone lines with questions that could easily be answered from the website as their enterprise grows. This is a pressure point that’s already becoming uncomfortable, so a chatbot is going to be their next endeavor after the migration effort.

The chatbot that I would recommend is the Microsoft Bot Framework. Because “Heartland Escapes” is migrating to Azure and their software is written on .NET, this is the most natural option due to how easily it can integrate into both of those environments. This bot can be used for website navigation, answering frequently asked questions, and integrating with e-commerce systems to provide price information and inventory status. It’s also multi-lingual, allowing users to interact with the website in their native language. The functional utility of the chatbot is dependent upon the development skills of the individuals performing the integration, so I’d recommend contracting skilled developers who have expertise in Microsoft Bot Framework. (Azure AI Bot Service | Microsoft Azure)

References

Intel® CoreTM i7-8700 Processor (12M Cache, up to 4.60 GHz) Product Specifications. (n.d.). Ark.intel.com. https://ark.intel.com/content/www/us/en/ark/products/126686/intel-core-i7-8700-processor-12m-cache-up-to-4-60-ghz.html

Ed Baynash et. al. (2023, April 10). Get started with autoscale in Azure - Azure Monitor. Learn.microsoft.com. https://learn.microsoft.com/en-us/azure/azure-monitor/autoscale/autoscale-get-started?toc=%2Fazure%2Fapp-service%2Ftoc.json

Micah McKittrick et. al. (2022, April 29). Overview of the Azure Compute Unit - Azure Virtual Machines. Learn.microsoft.com. https://learn.microsoft.com/en-us/azure/virtual-machines/acu

Multilevel Queue (MLQ) CPU Scheduling. (2017, September 26). GeeksforGeeks. https://www.geeksforgeeks.org/multilevel-queue-mlq-cpu-scheduling/

Stallings, W. (2018). Operating Systems: Internals and Design Principles. Pearson Education Inc.

de Bruin, P. (2023). Manage and deploy Kubernetes in Azure Arc - Azure Architecture Center. Learn.microsoft.com. https://learn.microsoft.com/en-us/azure/architecture/hybrid/arc-hybrid-kubernetes

Montemagno, J. (2023, October 6). Implement HTTP call retries with exponential backoff with Polly - .NET. Learn.microsoft.com. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/implement-resilient-applications/implement-http-call-retries-exponential-backoff-polly

Montemagno, J. (2023, March 1). Implement resilient Entity Framework Core SQL connections - .NET. Learn.microsoft.com. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/implement-resilient-applications/implement-resilient-entity-framework-core-sql-connections

Caron, R. (2018, June 28). Using the Retry pattern to make your cloud application more resilient | Azure Blog | Microsoft Azure. Azure Blog. https://azure.microsoft.com/en-us/blog/using-the-retry-pattern-to-make-your-cloud-application-more-resilient/

Hyett, A. (2023, September 22). Idempotency - What it is and How to Implement it. (2023, September 22). DEV Community. https://dev.to/alexhyettdev/idempotency-what-it-is-and-how-to-implement-it-4008

Kuan, M. (n.d.). Event Sourcing pattern - Azure Architecture Center. Learn.microsoft.com. https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing

GeeksForGeeks. (2010, December 25). Critical Section in Synchronization. GeeksforGeeks. https://www.geeksforgeeks.org/g-fact-70/

Lipsey, S. (2023, October 23). How to manage groups. Learn.microsoft.com. https://learn.microsoft.com/en-us/entra/fundamentals/how-to-manage-groups

Baykara, S. (2022, January 1). PCI DSS Control Objectives. PCI DSS GUIDE. https://pcidssguide.com/pci-dss-control-objectives/

GeeksForGeeks. (2021, October 15). Advanced Encryption Standard (AES). GeeksforGeeks. https://www.geeksforgeeks.org/advanced-encryption-standard-aes/

Bell, A. (2023, August 28). Azure DDoS Protection Overview. Learn.microsoft.com. https://learn.microsoft.com/en-us/azure/ddos-protection/ddos-protection-overview

API libraries. (n.d.). Shopify. Retrieved November 5, 2023, from https://shopify.dev/docs/apps/tools/api-libraries

TensorFlow - Creating C# Applications using TensorFlowSharp. (2019, August 7). CodeProject. https://www.codeproject.com/Articles/5164135/TensorFlow-Creating-Csharp-applications-using#trainingusingpython_cnnarchitecture

Azure AI Bot Service | Microsoft Azure. (n.d.). Azure.microsoft.com. Retrieved November 5, 2023, from https://azure.microsoft.com/en-us/products/ai-services/ai-bot-service/?ef_id=_k_51deb14534ac1146f14ad40d2949fc6a_k_&OCID=AIDcmm5edswduu_SEM__k_51deb14534ac1146f14ad40d2949fc6a_k_&msclkid=51deb14534ac1146f14ad40d2949fc6a#features