Modernizing “Heartland Escapes” to a Cloud-Hosted Infrastructure

Aidan Polivka

CS630 – Modern Operating Systems

Colorado Technical University

October 24, 2023

Table of Contents

Project Outline

“Heartland Escapes” is a bookstore located in Nebraska. They have been in Lincoln for quite a few years, opening their first store in 2010 and their second store in 2019. The store has recently seen a lot of success through a new marketing scheme using popular social media platforms like Instagram and Tik Tok. Much of their popularity can be attributed to the regular events they host, like the “school’s out reading program”, author meet-and-greets, and Halloween scary story readings. Because of their recent spike in consumer interest, they plan on expanding their business to neighboring cities, one location in Omaha and another in Grand Island.

Since their store first opened in 2009, “Heartland Escapes” hasn’t had any major upgrades to their technological hardware or software besides minor increases in disk space, ram, and a processor upgrade. Their current system is hosted on the premises of their first store location and consists of a windows 2008 server machine with an Intel Core i7-8700 processor. The Intel Core i7-8700 64-bit processor operates with 6 cores and 12 threads (Intel® CoreTM I7-8700 Processor (12M Cache, up to 4.60 GHz) Product Specifications, n.d.). Its Clock rate is 3.2 GHz, its L2 Cache is 256 KB for each core, and its L3 Cache is 12 MB. The server works with 8 gigabytes of RAM and has 8 TB of Hard Disk space, 3.5 TB being occupied. This server hosts a home-grown point-of-sale system, a home-grown inventory system, and a public website all written in .NET Framework 4.8 web forms, along with a Microsoft SQL Server containing an inventory database and an accounting database. A network of 8 machines between the two locations consume the Point-of-Sale web application and Inventory service, and the website interacts with the inventory service so that users can search for books in each store location from the comfort and convenience of their own homes. With the addition of two new stores and the added complexity of an ever-expanding consumer base, “Heartland Escapes” is looking to upgrade their system to take advantage of the scalability, monitoring, security, and management simplicity of a cloud hosted system.

Seeing as their current system is very Microsoft heavy and they expect continuous growth, my proposal to “Heartland Escapes” is to migrate their infrastructure to Azure and to keep their on-premises server to store a physical backup of their cloud system. I think migrating to the cloud would be highly beneficial for their expectation of seeing continuous growth, and cloud services offer a lot of opportunities for scaling up hardware for periods of increased traffic rather than needing to always have CPUs available. This is would easily resolve the domain need created by “Heartland Escapes” periodic promotional events. Azure is the cloud platform that I’m most familiar with, and it’s also a Microsoft product which would be brand consistent with their .NET applications and MS SQL Server database management system.

I think the configuration in Azure should be separated into two resource groups, one for data and infrastructure and the other for applications, with all resources being 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 containers with a configurable base operating system, and in this case, I think all services would run on a windows system for simplicity since deploying to a different operating system could potentially lead to unforeseen issues. The app services would live under an app service plan using the Premium v3 P1V3 hardware tier. This tier has two vCPUs and 8 GB of RAM per instance and can scale up to 30 instances. This would be a starting point to scale up or down from. In the data and infrastructure group, there would be an Azure SQL Server hosting both databases, and there would be a virtual network (vNet) that manages firewall rules and security concerns between the secure parts of the system, and a vNet Gateway would exist so that users could access the system via VPN. The public website would be the only aspect of the system outside of the vNet. The vNet would only be accessible via gateway, and the website’s public IP address would have access as well.

OS Processor and Core

Migration Benefits

There are many benefits to migrating to Azure. On the surface it looks like the new hardware is less capable than the existing hardware, but that isn’t true at all. Each service instance has access to 2 virtual CPUs, and there are 30 available instances, meaning the system now has access to 60 virtual cores where the whole system originally shared 6 cores! The applications should not need that much processing power either, and Azure allows administrators to easily scale the application resources up and down. So as the demand for this system grows, the hardware can grow with it naturally at the click of a few buttons. The scalability of Azure App Service instances is so advanced that you can create rules for scaling that dictate whether another instance of the app service should be created based on date, CPU percentage thresholds, memory thresholds, and minimum and maximum threshold limits (Ed Baynash, 2023).

Because the system is now distributed into individual azure components, the load on the system is localized to individual services rather than the entire system itself. So, if the database is performing a large batch operation that’s taking up resources, the deployed services aren’t going to feel a slow down (unless they’re accessing the database). Not only is the load better distributed in this way, but again the scaling aspect applies here. Each service could be configured to have a maximum of 10 instances, and to increase or decrease the instance number by one based on average CPU Percentage metrics or RAM metrics. With that configuration it would take a lot of traffic to overwhelm any of the three services.

An additional benefit is the increased reliability of the system. These services are now decoupled from a single machine and are running on an operating system managed and upgraded by Microsoft. This means that there is no longer the single point of failure from being hosted by an on premises machine, and if there is a failure on an instance of an app service then another instance can pick up the slack. This also means that any security updates for the underlying operating system are not the responsibility of the individual maintaining the system, but the responsibility of Microsoft. This deferred responsibility is a positive in this case, because Microsoft is going to ensure that their PaaS services are as secure as possible so that they aren’t liable for any breaches. Microsoft can make these upgrades with very limited down time. Speaking of down time, the P1V3 service tier boasts a 99.95% yearly availability, which could arguably be more available than the on-premises system depending on what “Heartland Escapes” machine maintenance looked like.

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 capabilities 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 determined 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 priority queue. Each priority 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 reassign process priority at preemption to mitigate the possibility of process 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 priority 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 exacerbated 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 similar to 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

Future Considerations Using Emerging Technology

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/