Heartland Escapes: Architecture Strategy

CS644 - Computer Systems Architecture

Aidan Polivka

August 18, 2024

Revisions

Number	Date	Description
1	07/28/2024	Initial Creation
2	08/04/2024	Resolve instructor feedback: Reference hanging indents
3	08/04/2024	Development Processes installation
4	08/11/2024	Resolve instructor feedback: In text citations
5	08/11/2024	Architecture Strategy
6	08/18/2024	Engineering Requirements

Table of Contents

1. Project Outline

Heartland Escapes is a bookstore that's been steadily gaining popularity in Nebraska. Their store has roots in Lincoln Nebraska and has expanded to three additional locations (two total locations in Lincoln, one in Omaha, one in Grand Island). A major reason for Heartland Escapes' popularity as a bookstore is the events that they host in store. Advertising for these events on social media has resulted in a semi-viral response, which has funded additional technological development for the company. Heartland Escapes is a relatively small company, but they have a competitive culture and a forward-thinking leadership crew that continues to look for technological solutions to compete with bookstore giants like Barnes & Noble.

Heartland Escapes has a small IT department now that they've developed so many in-house software solutions. They've decided that it would be best to retain three software engineers and the CIO and use contractors to support larger scale development efforts. This allows for constant technical support and maintenance for the company, and native first-hand knowledge of how the Heartland Escapes systems work without needing to retain a large development force.

Heartland Escapes has undergone a substantial transformation over the past few years. They migrated their point-of-sale system from on premises to Google Cloud, then conducted a large development effort to integrate company administrative processes into the point-of-sale system. So now that system is a hub for all internal processes to Heartland Escapes, including employee time clock, inventory reporting and re-ordering, accounting reporting, and inventory and product management.

Now that those projects are complete, Heartland Escapes is ready to move forward with development of their very own e-commerce platform. This platform will require integration into their pre-existing inventory and accounting systems. This architecture strategy is intended to support this development effort, resulting in a fully qualified architectural design for both the software and infrastructure of this e-commerce website.

2. Software Architecture and Evaluation

2.1 Existing Software Architecture

The pre-existing software solution at Heartland Escapes is their point-of-sale system. This system has been adapted to support additional administrative activities within Heartland Escapes. The primary components to this system include a .NET MVC application that services the client application and two supporting REST APIs: the Inventory API and the Accounting API. These APIs are both written in .NET Core C# and have their own SQL Server databases, an accounting_db and an inventory_db. The .NET MVC application also has a database that hosts data like employee timecards and store event schedules.

The infrastructure for the point-of-sale system is relatively simple. The APIs and MVC app are all deployed in GCP's Cloud Run environment. Cloud Run is a fully managed google cloud service, meaning that the infrastructure for these applications does not need to be maintained. Similarly, the SQL Server databases are hosted in Google's Cloud SQL components which are almost fully managed.

2.2 Architecture Strategy Objectives

It's important for Heartland Escapes to be able to integrate this new e-commerce solution into their pre-existing APIs for tracking and management of inventory and sales. Additionally, the maintenance burden on their development team would be greatly lessened by continued use of familiar technologies. Finally, since Heartland Escapes is a small company, we'll want to find the most cost-effective solution that requires the least amount of infrastructural maintenance.

It's also important to bring in the perspective of the domain when choosing an architectural solution. The domain of e-commerce is typically event based. Consider what needs to occur after a user checks out their cart:

1. The customers card must be processed
2. If their payment is successful, available inventory for the items purchased must be decreased
3. The customer must receive an email confirmation of their order with an invoice
4. If the items purchased reach the restock threshold, a new order must be sent to that product's supplier
5. The customers order must go to order fulfillment, where it's packaged and shipped to their home

E-commerce is also usually easily componentized. Consider all the different sub-processes within e-commerce: Inventory management, payment processing, order fulfillment, shipment tracking,

shopping cart management, product display, filtering, and recommendations, etc. Additionally, e-commerce applications typically require complex user interfaces to support these pieces of functionality.

2.3 Infrastructural Architecture Options

There are three primary options for infrastructure based on the preliminary requirements outlined in section 2.2. For the sake of simplicity and cost savings, all solutions will be deployed to Cloud Run. Since Cloud Run is fully managed, pay per use and can scale to zero instances, this component is the best option for Heartland Escapes regardless of architecture solution.

2.3.1 Microservices Architecture

A microservices architecture would allow for the development of loosely coupled units of domain functionality. Each process can be broken into its own modular set of code, providing a suite of benefits. This will allow engineers to work asynchronously on different microservices without bumping elbows and enforce the separation of responsibilities between software components. This solution also provides a highly robust and independently scalable solution. If one microservice node fails, it doesn't necessarily mean the whole system is down. Also, if the product catalogue is receiving high traffic volume, product catalogue service nodes can scale independently of the payment processing service nodes. Finally, the microservices architecture can easily send requests to the inventory and accounting APIs, fulfilling that requirement for Heartland Escapes.

This would also include a separate single page application to support the user experience. The framework used will be up for debate later in this architecture strategy.

Although the microservices architecture offers many benefits, there are also complications. Because microservice communication is network based, latency may be an issue for processes that need to run through multiple services. Also, microservice architecture is a complex solution to implement that may put a strain on the maintenance teams. Without careful configuration and documentation, this solution could result in failure (IBM, n.d.).

2.3.2 Modularized Monolith Architecture

The term "monolith" tends to scare individuals because it is not as progressive a solution as microservices. When many engineers think of monoliths, they think of tangled up legacy applications that are not well modularized. However, with strong development practices and guidelines, a monolith is still a completely viable architecture with today's technology (Belcher, 2020). This solution would extend the existing point-of-sale application to support the e-commerce system. Each sub-domain listed in section 2.2 would have its own module (likely a C# project) in code. This separation of concerns will ensure that everything is written in the same language and is simple to maintain for the IT folks at

Heartland Escapes. Because this solution is enclosed within a single application, inter-process communication latency will be significantly reduced.

Also, a monolithic solution could drastically impact the feature development speed of the team in a positive way. By developing within a monolith, full vertical sliced features can be developed at a time without needing to worry about developing additional infrastructure like REST calls, CI/CD pipelines, or Pub/Sub topics and subscriptions (Ozkaya, 2023).

This solution has its pitfalls as well. Although the architecture is simple and easy to maintain, it is not as robust as the microservices architecture. If a part of the system fails, it may break the entire system. Additionally, sub-processes within the system are not independently scalable, potentially leading to increased operational costs.

2.3.3 Hybridized Solution

Because the point-of-sale MVC system is already a monolithic core, that core could be utilized for common functionality in the e-commerce system like user management and product display. This core would support the core functionality of Heartland Escapes e-commerce system including serving the client application, while microservices handle the other application functions like notifications, payment processing, order fulfillment, etc. This would provide a happy medium between the modularized monolith, and the microservices architecture.

2.4 What's the Best Architectural Option?

The best solution for Heartland Escapes is going to be 2.3.1 microservices. This is the best solution in that it provides a separation of concerns between the existing point-of-sale system and allows for independent scaling of business processes. Although this opens multiple repositories for maintenance, it would be best that the internal point of sale system stays separate from the e-commerce site for security, scaling, and separation of concerns purposes.

3. Development Processes

3.1 Current Processes

The permanent members of Heartland Escapes development team use a Kanban development system to manage their work backlog and support their maintenance efforts. Kanban works well for small scale development efforts, and particularly well with maintenance work. Kanban focuses on visual workflow management to ensure that work is not lost, work in progress limits to ensure that features are delivered, and continuous delivery to maintain a steady flow of feature delivery and issue resolution.

This project management method will work well for a small team focused on maintenance. With the small team, they can allow members of the company to submit feature requests and bug fix requests to a Kanban backlog. Those requests can be filtered by the CIO, assigned priority, and delivered to the development team. Development team members can pull items out of the backlog as they see fit but are limited to the number of items they can pull into “work in progress” at a time. Then the CIO can run metrics on the board for burn down and estimates sprint by sprint.

The development team participates in Scrum ceremonies like sprint planning, sprint reviews and daily standups. Other ceremonies are held on an as-needed basis, like backlog refinement and retrospectives. The CIO acts as scrum master for the development team.

The development team currently uses CI/CD pipelines to deploy their applications automatically on merge to the devdeploy branch. Once the feature is dev tested and approved by another team member, it’s merged into the main branch and automatically deployed to the Sys environment for user acceptance testing by the CIO. Prod releases are manually deployed on a biweekly basis from the main branch (Martins, 2024).

3.2 Suggested Processes

A simple Kanban process will not fulfill all the required processes for a large-scale development effort. However, there are Agile development methods that extend Kanban and Kanban principles to better support large development efforts. One such development method is called Scaled Agile Framework, or SAFe. SAFe takes similar principles from traditional Kanban, including the visual workflow management, work in progress limits, and continuous delivery, but scales it up to fit a large development effort.

An oversized product delivery team tends to suffer from siloed knowledge. In a traditional development environment with a large crew, it can be easy for a single individual to be the only person how knows how a part of the system works. SAFe tries to resolve this issue by creating diverse teams

within the organization. For example, rather than there being a DevOps team, a Data team, and an Application Development team, there would be multiple teams each with individuals who can support DevOps, Data, and Application Development. This structure is called an Agile Release Train, or ART. With this development structure, the permanent employees at Heartland Escapes can remain exposed to the entire software stack during the development process. There would be multiple Agile Release Trains working in parallel on different epics (Agile Release Train, 2022). At this scale of a development effort, all scrum ceremonies are recommended to further refine the process. Kanban and SAFe are both proponents of continuous improvement, which is fueled by scrum ceremonies like Sprint Review and Sprint Retrospectives.

SAFe uses a Kanban board for backlog and work visualization. However, that board is populated by processes not typical to traditional Kanban. There is a role in SAFe called an Epic Owner. This individual is usually a product owner, system or enterprise architect, project manager or program director. Their job as an Epic Owner is to collaborate with the stakeholder to farm epics. In these sessions with the stakeholders, the Epic Owner is responsible for defining the epic, the lean business case, and the definition of the minimum viable product. They are then responsible for carrying that epic through the development process (Epic Owner, 2023).

The suggested deployment process is going to be the same as the current deployment process. One of the two changes that should be made is who the approval gate is for Sys and Prod. The solution architect or tech lead should approve changes to deploy in Sys, and the user acceptance testers should approve changes to deploy in Prod. The other change would be to include automated code quality gates to deployment pipelines. This includes strong integration testing coverage between services, ensuring that changes made between teams don't break service to service contracts. Additionally, unit test code coverage gates, software linting rules, vulnerability scanners and image attestation & authorization actions will be required.

3.3 Architecture

The development processes must suit the architecture Heartland Escapes is developing. To recap the suggested architecture, it is recommended that Heartland Escapes takes a microservices approach with a central Cloud Run service. The microservices should support stateless domain functionality of the e-commerce site.

With the focus on modularity and extensibility, an iterative Agile approach is key to supporting this architecture. The iterative nature of Agile's requirements gathering process requires that the system maintains its ability to be adaptive. If Heartland Escapes chooses to assign an ART per microservice, the changes made to the CI/CD pipelines for automated integration testing will be incredibly important to supporting this architecture and team structure. This method of assigning an ART per microservice also

decentralizes the decision-making process, allowing teams to be self-organizing and self-managed. This will support asynchronous development of the e-commerce site, resulting in increased velocity.

The incorporation of an Epic Owner can help facilitate Domain Driven Design within the system. Having an individual take ownership of an Epic from inception with the stakeholder to delivery with the development team will help foster domain expertise within the larger team. This will also support the collection and use of ubiquitous language between stakeholders and the project delivery team. Also, the ARTs should be working within bounded contexts, supported by the microservice architecture (Stemmler, 2019).

An architectural addition that may support this environment could be clean architecture. The premise of clean architecture is to separate presentation, domain, and infrastructure logic into independent layers. All database connection, third party integration, and API to API communication should be handled in the infrastructure. Domain functionality and business logic should be handled in the domain layer. API configuration and client-side code should be handled in the presentation layer. These layers should be separated within the file structure, and only available between each other using dependency injection. Having the domain logic centralized and well-defined strongly supports Domain Driven Design. Clean architecture also ensures that each layer (presentation, domain, and infrastructure) is tested independently which facilitates continuous integration and deployment from SAFe. The loose coupling of these layers supports better interoperability, a tenant of Agile development (Jacobs, 2020).

4. Architectural Design Strategy

To recap the existing architecture, Heartland Escapes has an existing .NET MVC Point-of-Sale system. This system has been adapted to support other administrative operations like inventory and sales reporting, employee timeclock management, and store event scheduling. This service acts as a monolith currently to handle all internal processes for Heartland Escapes. Additionally, there is the inventory API and the accounting API that will be utilized by the e-commerce site.

4.1 Core System Requirements

The new e-commerce site has many qualitative requirements outside of the core business requirements. Below is a list of requirements that will help drive the optimal architecture:

1. The system must be able to scale up or down based on CPU utilization.
2. The system must be useable by external users
3. The system must be well documented and maintainable
4. The system must be observable via logs, message tracing, and errors.
5. The system must support versioning.
6. The system must be able to perform actions asynchronously and in parallel.
7. System infrastructure must be easily maintained, including CI/CD pipelines.
8. CI/CD is required to quickly drop new releases.
9. The system must be testable and utilize unit testing so that regressions are covered.
10. The system must be resilient to failures and recover without human intervention if possible.
11. The system must be secure to ensure compliance with the payment card industry data security standard (PCI DSS).

By examining these requirements, a set of required system quality attributes can be developed. This list of requirements says that Heartland Escapes needs a system that is scalable, maintainable, observable, performant, reliable, available, testable, and securable.

4.2 Architecture from Requirements

4.2.1 Client Server Architecture: .NET Blazor Server

The application must be available to users of the e-commerce system. To support this availability requirement, a client-server architecture will be used to serve the application to users. The recommended framework to support this interaction between the users and the system will be ASP.NET

Blazor Server. The reasoning behind this choice is that the maintenance team has experience with .NET C# MVC, so .NET Blazor should not be too far outside of their wheelhouse. Rather than expecting the development team to support a foreign reactive framework like Angular or React, they can utilize a framework that's familiar.

The client application should be hosted within a Cloud Run service in Google Cloud Platform. This component type is fully managed by Google, meaning that there isn't any need to maintain infrastructure. Google boasts a 99.95% uptime in their Service Level Agreement (SLA) for this resource, providing yet more availability (Google, 2024).

Cloud Run services offer automated horizontal scaling using CPU, memory, or some other custom metric threshold configurations. This is a major piece in securing the requirement that the system can scale in response to load. Additionally, this cloud run resource can be configured to have multiple nodes span multiple zones or multiple regions, introducing *redundancy* and the ability to failover to another node if an error were to occur. This configuration supports the requirement for system reliability

4.2.2 Microservices Architecture

Google Cloud Platform (GCP) has many potential avenues to implement a microservices architecture. The recommendation for this project would be to use the platform as a service (PaaS) components available within GCP so that the maintenance effort for the Heartland Escapes core team is not unnecessarily increased. Additionally, only fully managed resources within GCP can scale to zero (Cloud Run, Cloud Functions, Firebase).

Utilizing the microservices architecture would support many of the required quality attributes for Heartland Escapes e-commerce system. With a microservices system, individual business processes can scale up and down independent of one another. This scaling should result in a more performant solution. Having separate services built to handle different business requirements should make the system easier to maintain if it's well documented. The separation of functionality will allow engineers to make changes to one part of the system without needing to worry as much about downstream impact (unless they're making changes to a contract between systems).

The recommendation to support the microservices architecture would be to build the system using google cloud functions with a single cloud run service. The cloud run service would support serving the client application, and the additional supporting microservices would be hosted using google cloud functions. Google cloud functions are also fully managed and offer the same uptime of 99.95% (Google, 2021).

4.2.3 Event Driven Architecture & REST

To support the level of performance required by the system in that it should be able to handle requests asynchronously and in parallel, an event driven architecture with Representational State Transfer (REST) Application Programming Interface (API) architecture would be ideal. The statelessness of the REST API architecture supports the ability to perform actions in parallel and asynchronously. Event driven architecture supports business requirements for the e-commerce system and compliments microservices architecture well.

To support the Event Driven Architecture, Google's Pub/Sub should be used for the messaging queue. This messaging system utilizes the gRPC protocol which is more performant than the traditional HTTP(S) request. Pub/Sub messages are transferred over mTLS, further securing messages in transit and supporting the requirement by PCI DSS. Also, Pub/Sub offers simple configurations that will greatly support reliability. Each of the endpoints built into cloud functions will be built using the REST API architecture ensuring stateless processing. Cloud functions spin up on demand, meaning that they scale as much as required by system load. This greatly increases the system's ability to respond to scale.

4.2.4 Exponential Backoff Pattern

If a failure does occur in the event driven system, architectural solutions must be chosen to support resilience of the message pipeline. One major decision is to utilize Exponential Backoff of message retries. If a message fails, then it must be retried after a pause. If it fails again, it should be retried after a longer pause. Up until the max threshold of retries has been reached, then the message should be stored in a bucket to be retried after the issue ailing the system is resolved. Dead letter queues and exponential backoff are both configurable in Google Pub/Sub.

4.2.5 Clean Architecture, CQRS, Domain Driven Design

Clean Architecture is a layered architecture solution that recommends splitting up the domain, infrastructure, and presentation logic into separate layers. These layers should only be able to communicate with one another via dependency injection, with domain logic as the deepest most independent layer.

Command Query Responsibility Segregation (CQRS) states that operations to an API can be lumped into one of two categories: Commands (write operations) and Queries (read operations). This development pattern separates those two operations and their relation to database connections. It also offers more asynchronous processing that matches the Event Driven Architecture.

Finally, Domain Driven Design (DDD) brings further definition to the domain layer. It states that entities and operations within the domain layer should reflect the language of the stakeholders and their

domain. This language is called “ubiquitous language”. DDD also supports a concept called “Bounded Contexts”, which refers to breaking up a large solution into areas of like domain responsibility. This principle of DDD fits in tightly with the microservices architecture and provides a way to delineate what functionality should reside in what API.

These three design patterns together strongly support maintainability and testability. Having the layered Clean Architecture and CQRS gives clear lines for unit testing between presentation, domain and infrastructure. DDD will also provide a language set to use in unit testing, and a strong separation of domain centered unit testing (Jacobs, 2021).

4.2.6 Test Driven Development

Test driven development (TDD) results in an incredibly resilient software set. TDD requires that unit tests are conceptualized and written prior to solution development. This ensures that all code is unit tested, error conditions are well thought out, and any changes to code that result in regression should be caught by pre-existing unit tests. This greatly supports maintainability an reliability of the system.

4.2.7 Canary Rollout Pattern

A robust rollout pattern is essential for keeping the system up during re-deployment. The canary rollout pattern keeps the original deployment running while creating the new deployment. The new deployment services a small subset of users while the original maintains the majority of load. Once the new deployment has proven to be bug free and not causing issues, it’ll take on all application load and the old deployment is decommissioned. Having a rollout pattern like this will ensure high availability of the system and resiliency for failed deployments.

4.2.8 Google Cloud Platform

Google Cloud Platform by default has a lot of resources available out of the gate for observability. For logging to be observable, all that’s needed is for the application to write logs to standard out and errors to standard error. Message tracing is easily configured, and network security is easily developed and maintained through GCP.

5. Engineering Requirements

1. User Authentication and Authorization: The e-commerce system must have robust authentication and authorization mechanisms for all network calls. Role-based access controls must be employed to separate authorization between customers and administrators.
    a. This requirement aligns with the architecture strategy by ensuring secure and controlled access to the system. It specifically aligns with the microservices architecture, where authentication services can be isolated and scaled independently.

2. Product Catalog Management: The e-commerce system must have a robust product catalog system that allows for easy management, categorization, and updating of books and other products. This system should support features like search, filtering, and sorting based on various criteria (e.g., genre, author, price).
    a. This requirement aligns with the microservices-based architecture, allowing the catalog service to scale independently and remain highly available. This product catalog also should integrate with the existing Inventory API.

3. Shopping Cart and Checkout: Heartland Escapes e-commerce system should employ a shopping cart feature where customers can add, modify, or remove books. The checkout process should include multiple payment options, order summary, and address management. Customer carts should be preserved across session closings and refreshments.
    a. The architecture should support integration with third-party payment gateways while ensuring the system remains secure and compliant with financial regulations. The microservices architecture should support this need well and maintain interoperability should the payment processor change. Additionally, the Agile development method will allow for decisions to be made about the intricacies of the cart (for example, a decision might be made to use Firebase to store cart information once this set of work is pulled into refinement). Test driven development should limit any regressions in functionality should changes need to be made to the payment sub-system.

4. Order Management: Any e-commerce system should have an order management system that tracks the entire order lifecycle from purchase to delivery. This system should provide customers with real-time updates on their order status via email.
    a. This feature aligns well with the proposed architecture strategy. Utilizing event-driven architecture will help ensure that order status updates are handled efficiently and in real-time. Microservices architecture should help keep this functionality modular and separate from the rest of the system.

5. Search Functionality: The product management sub-system should include advanced search functionality, allowing customers to find books based on keywords, ISBN, author, or other attributes. The search should be performant and return relevant results.
    a. This requirement will be well supported by CQRs, Clean Architecture and Test-Driven Development. These aspects of the architectural strategy will ensure that the search functionality is robust and well developed, and the test-driven development will ensure that future enhancements to the search functionality will be regression proof.

6. Customer Reviews and Ratings: Many e-commerce systems allow customers to leave reviews and ratings for books they have purchased. Heartland Escapes e-commerce system should support this as well, with potential future improvements to incorporate reviews from critics. This feature should include moderation tools and the ability to filter reviews by rating.
    a. This feature is again supported by the microservices architecture in that the reviews can be hosted on an independent microservice, allowing for independent scaling. Future updates to support reviews from critics and other online sources should not impact other areas of the system because of the segmentation of the microservices architecture.

7. Personalized Recommendations: Avid book readers and consistent customers would love a strong recommendation engine that suggests books based on customer behavior, preferences, and purchase history. This engine should update recommendations in real-time as customer interactions change. Ideally, the algorithm will consider the writing style of the author and content themes using artificial intelligence.
    a. Machine learning services could be integrated as a separate microservice, ensuring scalability and flexibility in deploying models. This separation of the machine learning algorithm will also be critical with how quickly machine learning technology changes.

8. Inventory Management: Heartland Escapes e-commerce system must utilize the existing inventory management system that tracks stock levels in real-time, triggers restock alerts, and integrates with the order management system to update available quantities upon order placement. Extensions of the inventory API may need to be made to support this requirement.
    a. Integration with the existing systems is essential for this architecture strategy since the e-commerce system is going to utilize the same inventory and warehousing as the stores. This will incorporate the Inventory API into the new microservices ecosystem. Allowing this service to stand alone will ensure it remains responsive and scalable.

9. Wishlist Feature: The e-commerce system will provide a wish list feature that allows customers to save books for future purchases. The wish list should be easily accessible from the customer's account and should allow for easy addition and removal of items. The system should also send notifications when items in the wish list are on sale.
    a. This is a big benefit of the microservices architecture. The notifications from the wish list microservice and the order management and fulfillment microservice can both utilize the

same code in a correspondence microservice using the event driven architecture. As with the other microservices, the wish list service will be able to be scaled independently.

10. Customer Support and Helpdesk: The e-commerce site should offer a customer support system that includes a helpdesk, live chat, and ticketing system. Customers should be able to easily contact support and track the status of their inquiries.
    a. Again the microservices architecture would be beneficial, allowing the support system to interact seamlessly with other services like order management, user accounts, and correspondence.

5.1 Architecture Strategy Evaluation

Pros:

1. Scalability: The chosen microservices and event-driven architectures allow individual components to scale independently, ensuring the system can handle large volumes of traffic, especially during peak times like holidays or Heartland Escapes events.

2. Flexibility: Decoupling services provides the flexibility to update or replace parts of the system without affecting the entire application.

3. Reliability: With distributed services, failures in one service do not bring down the entire application, increasing the overall resilience of the system. Also, test driven development will reduce regression after new feature development and elevate the robustness of the system.

4. Security: With robust authentication and authorization, distributed systems offer a security advantage by reducing the attack surface. Meaning that with the system spread out, penetration into one microservice does not necessarily mean that the bad actor will be able to get into the rest of them.

5. Reusability: The modular nature of microservices allows for code to be reused easily. The primary example being the correspondence service that supports emails to the customers.

Cons:

1. Complexity: Microservices architecture introduces complexity in terms of service communication, data consistency, and system monitoring.

a. Mitigation Strategy: Implement service orchestration tools and monitoring solutions to manage and visualize inter-service communications and ensure data consistency across the system. These tools my include a service mesh like Istio and an API Gateway

2. Latency: Distributed systems can suffer from higher latency due to the need for inter-service communication.
    a. Mitigation Strategy: Optimize API calls, use asynchronous communication where possible, and deploy services close to each other geographically to minimize latency. Additionally, caching can be used in areas where the system repeatedly makes the same requests.

3. Deployment Overhead: Managing and deploying multiple microservices requires more sophisticated CI/CD pipelines and container orchestration tools.
    a. Mitigation Strategy: Invest in automated deployment pipelines and container orchestration platforms like Kubernetes to streamline the deployment process and reduce overhead.

4. Platform Dependency: The architecture proposed is not platform agnostic, meaning that this system as architected is deeply tied to Google Cloud Platform
    a. Mitigation Strategy: Containerize all the microservices so that they remain portable. Consider using Kubernetes instead of Cloud Run since Kubernetes is platform agnostic.

6. Emerging Technologies

TBD

References

Belcher, M. (2020, September 8). *Breaking Down the Monolith*. Codurance. https://www.codurance.com/publications/2020/09/08/breaking-down-the-monolith

IBM. (n.d.). *What are Microservices? | IBM*. Www.ibm.com. https://www.ibm.com/topics/microservices

Ozkaya, M. (2023, April 16). *Microservices Killer: Modular Monolithic Architecture*. Medium. https://medium.com/design-microservices-architecture-with-patterns/microservices-killer-modular-monolithic-architecture-ac83814f6862

Agile Release Train. (2022, October 24). Scaled Agile Framework. https://scaledagileframework.com/agile-release-train/

Epic Owner. (2023, June 30). Scaled Agile Framework. https://scaledagileframework.com/epic-owner/

Jacobs, J. (2020, February 19). A Brief Intro to Clean Architecture, Clean DDD, and CQRS. Jacobsdata.com. https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs

Martins, J. (2024, January 19). What Is Kanban? A Beginner's Guide for Agile Teams [2023] • Asana. Asana. https://asana.com/resources/what-is-kanban

Stemmler, K. (2019, November 5). Comparison of Domain-Driven Design and Clean Architecture Concepts | Khalil Stemmler. Khalilstemmler.com; khalilstemmler.com. https://khalilstemmler.com/articles/software-design-architecture/domain-driven-design-vs-clean-architecture/

Google. (2021, May 12). *Cloud Functions Service Level Agreement (SLA) | Google Cloud*. Google Cloud. https://cloud.google.com/functions/sla

Google. (2024, January 24). *Cloud Run Service Level Agreement (SLA)*. Google Cloud. https://cloud.google.com/run/sla

Jacobs, J. (2021). *A Brief Intro to Clean Architecture, Clean DDD, and CQRS*. Jacobsdata.com. https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs