IT & Computer Science

Instructor Name: Zubaria Noureen

Name: Abuzar Khan **Reg #:** B22F1053SE23

Section: SE (GREEN) **Date:** 11/12/2025

Software Re-Engineering

Assignment 04

Modernizing the ShopEase Inventory Management System (SIMS)

Introduction

The **ShopEase** Inventory Management System (SIMS) is a 15-year-old legacy application built using Visual Basic 6, Crystal Reports 9 and SQL Server 2005. It runs only on Windows XP–compatible hardware and lacks an API layer, mobile access and real-time stock updates. Over the years, the system has grown critical for ShopEase's retail operations but has become costly and difficult to maintain. Modernizing SIMS will improve maintainability, scalability, performance and security while reducing operational costs.

This report analyses SIMS's limitations and risks (Task 1) and proposes a modernization strategy and target architecture (Task 2) that meet management's requirement for a **cloud-based, scalable inventory system with real-time updates, REST APIs and automated deployments**.

Task 1 Legacy System Assessment

1. Technical limitations

Unsupported language/platform Visual Basic 6 was released in 1998 and Microsoft ended mainstream support in 2008. Without support, VB6 no longer receives bug fixes or security patches, leaving applications vulnerable to exploits. VB6 code targets 32-bit Windows and uses COM/ActiveX components that do not work reliably on 64-bit systems. It also lacks built-in support for web or mobile clients and cannot integrate easily with modern frameworks or APIs.

Outdated database SQL Server 2005 reached its extended support end date on **12 April 2016**. Once a product reaches end of support, Microsoft no longer provides security patches or hotfixes. Running an unsupported database exposes the system to unpatched vulnerabilities and can violate regulatory compliance (e.g., PCI-DSS, GDPR). Older versions also lack features such as automated indexing, advanced query optimizations and native support for cloud-scale availability.

Tight coupling and monolithic design VB6 encourage a single-executable monolithic architecture. Business logic is often buried in event handlers, global variables and COM objects, making it difficult to understand and modify. The lack of modularity creates large *technical debt* and hinders scaling or integrating new channels (e.g., mobile apps).

Limited performance and scalability – VB6 applications cannot leverage multi-core CPUs effectively and are prone to memory leaks and crashes during high load. Manual backup processes and Windows-XP-only deployment further limit availability. Without an API layer, SIMS cannot integrate with other systems (e-commerce site, warehouse automation) or provide real-time stock data.

2. Business limitations

- **Productivity and user experience:** The Windows-only desktop interface restricts access to office PCs and offers no mobile or web interface. Employees must re-enter data into other systems, increasing errors and slowing operations. Crystal Reports 9 requires manual report generation; there is no self-service analytics.
- **Downtime and performance issues:** Frequent downtime during sales seasons and slow response under load hinder customer service and can cause lost sales. Lack of real-time updates means inventory counts may be inaccurate.
- **High maintenance costs:** Hiring developers familiar with VB6 is difficult and expensive. Maintaining Windows XP machines, outdated database servers and manual backups requires dedicated IT staff. Poor documentation increases onboarding time.

3. Risks of keeping the system unchanged

- **Security and compliance risks:** Unsupported VB6 and SQL Server 2005 do not receive security updates. New vulnerabilities remain unpatched, increasing the risk of data breaches. Regulatory frameworks often require supported software; non-compliance can lead to fines.
- **Business continuity risk:** Hardware failure or OS upgrades could break SIMS because it depends on Windows XP and legacy drivers. Backup and restore are manual processes, increasing the risk of data loss.
- **Limited agility:** Lack of API integration prevents ShopEase from adopting e-commerce channels, mobile apps or warehouse automation. Competitors with modern systems can outpace ShopEase.

Task 2 Modernization Strategy Proposal

1. Choosing a modernization approach

Numerous migration strategies exist; they are often described as the “7 Rs”. The following summarises the relevant approaches and their trade-offs:

Strategy	Summary (with source)	Pros	Cons
Rehost (Lift-and-shift)	Move the application as-is to new infrastructure (e.g., cloud VMs) without modifying code.	Fast migration; minimal code changes; can quickly test workloads in the cloud.	Does not modernize the application; retains technical debt; may still provide poor user experience.
Replatform	Make minimal changes so the application runs on a new platform (e.g., port VB6 code to run on supported OS).	Cheaper than full rewrite; allows gradual migration of workloads and testing.	Still tied to monolithic design; limited ability to leverage cloud scalability or modern development practices.
Refactor	Reorganize and optimize existing code to reduce technical debt and improve non-functional attributes (performance, security).	Improves performance and allows scaling; reduces technical debt and can adapt to changing requirements.	Complex and resource-intensive; needs automation and skilled developers; risk of errors and cost overruns.
Rearchitect	Redesign the application's architecture, often by breaking the monolith into microservices.	Provides flexibility, scalability and control; enables adoption of cloud-native patterns.	Requires significant coding and investment; longer timeline and higher cost.
Rewrite/Rebuild	Completely rewrite the application, retaining needed functionality but using modern languages and frameworks.	Tailored solution: can improve user experience and eliminate technical debt.	Very time-consuming and expensive.
Replace (repurchase)	Replace the system with a new off-the-shelf or SaaS solution.	Vendor handles maintenance and updates; quick adoption of new features.	Loss of control over future features; change-management challenges as users adapt to a new system.

Recommended approach: Given SIMS's outdated technology, lack of modularity and the need for real-time updates and API-based integration, a **rearchitect (and partial rewrite) strategy** is recommended. Rehosting or replatforming would still leave ShopEase with unsupported VB6 and technical debt, while a full rewrite would be costly and risk losing domain knowledge.

A phased rearchitecting approach allows the system to be **decomposed into microservices** while gradually migrating functionality and data. Critical services (e.g., inventory, orders, products) can be rebuilt using modern languages (.NET Core or Java/Spring Boot) and exposed through RESTful APIs.

Non-critical modules can initially be rehosted on cloud VMs to reduce risk. This hybrid approach balances risk, cost and agility, and positions ShopEase for continuous improvement.

2. Proposed target architecture

Microservices architecture

Microservices break a complex application into smaller, independent services that can be developed, deployed and scaled separately. For inventory management, microservices deliver improved scalability, adaptability and resilience. Services can use different technologies best suited for their tasks. Failure in one service does not crash the entire system. Faster deployment cycles support rapid feature delivery.

Technology stack

- **Cloud provider: Microsoft Azure** is a logical choice because of native support for SQL Server workloads and PaaS databases. Azure offers **Azure SQL Database**, a fully managed relational database that provides built-in high availability, scalability and performance. Using a PaaS database eliminates the need to patch or manage the underlying database engine.
- **Programming framework:** Adopt **.NET 7/8 (ASP.NET Core)** or **Java (Spring Boot)** for backend services. Both frameworks support RESTful APIs, dependency injection and containerization. Services should be packaged as **Docker** containers and orchestrated with **Kubernetes** (Azure Kubernetes Service). This enables horizontal scaling, rolling updates and fault isolation.
- **API Gateway:** Implement an API gateway (e.g., Azure API Management or Nginx) to expose REST endpoints securely. The gateway handles routing, rate limiting, authentication and versioning. It enables internal microservices to evolve independently while presenting a consistent external API.
- **Front-end:** Develop responsive web and mobile applications (e.g., React JS, Flutter) that consume the APIs. Modern user interfaces improve productivity and allow real-time stock visibility.

Data migration strategy

1. **Assessment & planning:** Use tools such as Microsoft's Data Migration Assistant to assess schema compatibility and identify feature issues when migrating from SQL Server 2005. Cleanse and standardize data to ensure quality.
2. **Provision target:** Create **Azure SQL Database** instances with appropriate service tiers and network configurations. Use **Elastic Pools** if multiple databases share resources.
3. **Schema migration:** Generate scripts to create target schemas. Where necessary, normalize or redesign tables to support microservices (each service owns its schema). For example, the Inventory Service manages product quantities, while the Orders Service owns order tables. A **data warehouse** (e.g., Azure Synapse) can consolidate data for reporting.
4. **Data migration:** For initial loads, perform an **offline migration** outside business hours using **backup/restore** or **Azure Database Migration Service**. For continuous operation, use **transactional replication** or **Azure SQL Data Sync** to replicate changes until cut-over. Validate data integrity post-migration.
5. **Decommissioning:** After successful verification, decommission the on-premises SQL Server 2005 databases and redirect clients to the new endpoints. Archive legacy data for compliance.

API enablement

- **RESTful design:** Expose CRUD operations for products, inventory, orders and reporting through RESTful endpoints (e.g., /api/products, /api/inventory). Use standard HTTP methods (GET/POST/PUT/DELETE), JSON payloads and versioning. Document APIs with **OpenAPI/Swagger** and provide client SDKs.
- **Real-time updates:** Implement event-driven messaging (e.g., Azure Service Bus, Apache Kafka) to publish events when inventory levels change. Clients can subscribe via WebSockets or SignalR for real-time stock updates. Services should update caches (e.g., Redis) for low-latency reads.
- **Interoperability:** Provide external partners (suppliers, warehouses) with secure API access using OAuth 2.0 or client certificates. Consider GraphQL for flexible queries if reporting needs are complex.

Security enhancements

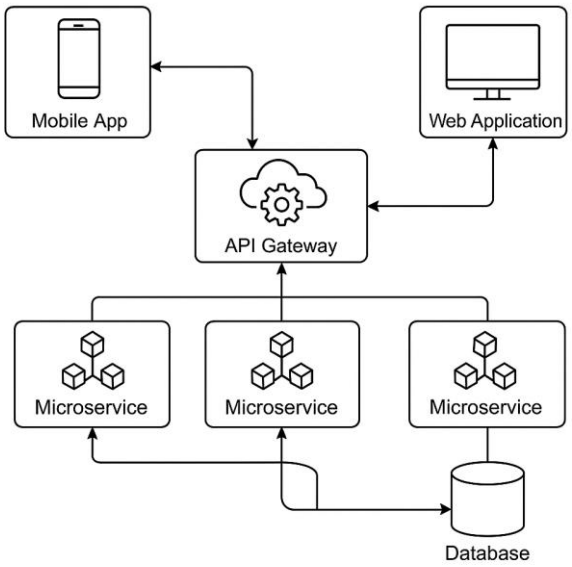
- **Patch management:** By migrating to Azure SQL Database and modern frameworks, ShopEase gains automatic patching and built-in security features. This eliminates the risk of unpatched vulnerabilities present in VB6 and SQL Server 2005.
- **Authentication & authorization:** Use federated identity (Azure AD B2C) with OAuth 2.0 and OpenID Connect for user authentication. Implement role-based access control (RBAC) within microservices. Services should verify JWT tokens and enforce least-privilege access.

- **Data protection:** Encrypt data at rest using Transparent Data Encryption (TDE) on Azure SQL Database and enable Always Encrypted for sensitive columns. Use TLS 1.2+ for data in transit. Implement secure coding practices (input validation, parameterized queries) to prevent injection attacks.
- **Monitoring & compliance:** Use Azure Monitor and Application Insights for logging and performance telemetry. Configure audit logs and anomaly detection. Adhere to compliance standards relevant to retail (e.g., PCI-DSS) and implement periodic penetration testing.

3. Reporting and documentation

For the assignment submission:

1. **Cover page and formatting:** Follow the submission guidelines: include a cover page with course details, assignment number, section and submission date. Number pages and ensure readability.
2. **Challenges and mistakes:** Document any challenges encountered during the modernization planning (e.g., understanding legacy code, data quality issues) and how you addressed them. Include screenshots of assessment tools (e.g., Data Migration Assistant reports) and diagrams of the existing and proposed architectures.
3. **Diagrams:** Provide diagrams illustrating the target architecture. Figure 1 shows an example high-level microservices architecture for SIMS (web/mobile clients interact via an API gateway with discrete services and a cloud database).



Conclusion

Modernizing ShopEase's SIMS is essential for ensuring long-term business agility and security. The legacy system suffers from unsupported technologies, poor scalability and high maintenance costs. Leaving it unchanged exposes the company to security breaches and operational risks.

A phased **rearchitect and partial rewrite** strategy adopting microservices, Azure SQL Database and modern APIs addresses these issues while controlling risk and cost. The proposed architecture enables real-time inventory updates, seamless integrations and secure, automated deployments.

By following this roadmap and documenting challenges, ShopEase can transform SIMS into a modern, resilient platform that supports future growth and innovation.

END!