**Global Bank Migration: Hybrid AWS Cloud Architecture**

**Overview**

A Global bank is facing critical challenges with their legacy core banking systems:

* Increased transaction loads during peak hours causing system delays
* 4-hour Disaster Recovery time, impacting business continuity
* High maintenance costs of on-premises infrastructure and data centres
* Difficulty scaling operations to new geographical regions
* Complex compliance requirements across multiple jurisdictions

**Objectives**

I will be outlining a detailed solution to migrate this core banking system to AWS Cloud. By doing this, I hope to achieve the following outcomes:

* Reduce disaster recovery time to less than 15 minutes
* Achieve 99.999% availability for critical services
* Enable real-time scalability for transaction processing
* Implement robust security controls meeting Global banking standards
* Significantly reduce operational costs
* Architectural agility to enable rapid geographical expansion

Ideally, this solution would employ a multi-account AWS architecture managed through AWS Control Tower and AWS Organisations, with separate accounts for:

* Production
* Development/Testing
* Disaster Recovery
* Security/Logging
* Shared Services

However, I can only use one AWS Free Tier account, so I will provide the most optimal solution to achieve the objectives discussed above by following the AWS Well-Architected Framework.

This **Hybrid Cloud Architecture** design emphasises robust network isolation, security, scalability, and operational excellence through a well-structured **4-VPC architecture**. Each VPC serves a specific purpose, enhancing modularity and clarity in responsibilities while addressing key pillars of the AWS Well-Architected Framework. This separation follows the principle of least privilege and ensures that security is built into the foundation of the infrastructure.

**Breaking down the VPCs**

**Production VPC (10.1.0.0/16)**

The Production VPC serves as our application hub, hosting the core banking services. Here, I deployed:

* An Amazon EKS cluster running the microservices architecture
* Three critical services: Account Management, Transaction Processing, and Authentication
* A Lambda-based fraud detection system
* Application Load Balancers for traffic distribution
* NAT Gateways enabling secure outbound internet access

The Production VPC is where the applications live and breathe, handling customer interactions while maintaining strict security protocols.

**Data VPC (10.2.0.0/16)**

The Data VPC is a fortress for sensitive banking information. Key features include:

* Strictly private subnets with no internet access
* Aurora PostgreSQL for transactional data
* ElastiCache Redis for high-speed caching
* DynamoDB endpoints for NoSQL data storage
* Direct connectivity to application services in Production VPC via Transit Gateway

This VPC embodies a "security-first" approach to data management.

**Management VPC (10.3.0.0/16)**

The Management VPC serves as the administrative control centre:

* Bastion hosts for secure administrative access
* Systems Manager integration for serverless management

This VPC provides secure, audited access for system administrators and DevOps teams.

**Security VPC (10.4.0.0/16)**

The Security VPC acts as the guardian of the infrastructure:

* AWS Network Firewall deployment
* Centralised security controls
* Traffic inspection and filtering

**Networking and Data Flow**

**Service Communication**

This architecture implements a hub-and-spoke model using **AWS Transit Gateway**. This enables secure inter-VPC communication, controlled access to data services, and centralised routing and security policies.

**External Connectivity**

The system handles external connections through multiple secure channels.

Customer Traffic is routed through **WAF** and **Shield,** and directed by **Application Load Balancers** to their appropriate microservices.

As this is a hybrid setup, the on-premises integration is achieved through **AWS Direct Connect** for dedicated, private connectivity at 10Gbps. This ensures consistent, low-latency access to cloud resources. The private connection bypasses the public internet, significantly enhancing security and reducing data transfer costs.

**Transit** **Gateway** acts as a network hub, routing traffic between the four VPCs and on-premises network. When a transaction is initiated from the legacy banking systems on-premises, it flows through Direct Connect to Transit Gateway, which then routes it to the appropriate VPC based on the request type – whether it is accessing the containerised microservices in Production VPC or querying data from the private Data VPC. This setup ensures the banking platform maintains security and performance while seamlessly integrating cloud and on-premises resources.

**Security and Encryption: Defence in Depth**

I implemented security at every layer in this architecture. The first line of defence is **AWS WAF**, configured with three critical security rules:

* **Rate Limiting**: Limits requests to 2000 per IP address, preventing DDoS attacks and brute force attempts. Automatic blocking of excessive request.
* **SQL Injection Prevention**: Deep packet inspection for SQL Injection patterns and multiple text transformations (URL\_DECODE, HTML\_ENTITY\_DECODE) to protect database integrity.
* **Geographic Restrictions**: Blocks traffic from high-risk countries to reduce attack surface.

**AWS Shield Advanced** protects the ALB, from DDoS attacks, provides real-time attack monitoring, automatic mitigation and cost protection during DDoS events.

The **Network Firewall** is the second line of defence at the VPC level in the Security VPC, acting as a central inspection point. It inspects traffic between VPCs via Transit Gateway and internet-bound traffic.

Here are the two rule groups I configured:

* **Banking Protocol Rules** to allow specific SWIFT and Payment Gateway traffic, enforcing strict protocol compliance and granular control over financial transactions.
* **Threat Prevention Rules** to block suspicious transaction patterns, detect SQL injection attempts, monitor for brute force attacks and generate real-time alerts from the CloudWatch logs group.

At the instance level, I used multiple **security groups** as virtual firewalls, providing granular control over inbound and outbound traffic between services across all four VPCs. For example, I enforced the principle of least privilege by ensuring services like EKS Clusters can only communicate with necessary resources like Aurora Databases in the Data VPC, while the Bastion Hosts in the Management VPC have restrictive access to specific administrative ports.

**KMS** uses two customer master keys (CMKs): a **database key** managing encryption for Aurora, DynamoDB, S3 audit logs, and CloudTrail records, and an **application key** handling encryption for EKS clusters and Lambda functions.

Both keys feature automatic yearly rotation and 7-day deletion protection. Key policies follow least privilege principles, with granular service-level permissions.

This dual-key strategy maintains clear separation between application and data layers while ensuring comprehensive encryption across the banking infrastructure.

**Comprehensive Security Monitoring**

In my banking architecture, security monitoring is not just about detecting threats – it’s about creating a comprehensive understanding of the security posture at any given moment. I have implemented four core AWS services that collaborate to provide continuous security oversight, each serving a distinct but interconnected purpose.

**AWS GuardDuty** serves as an intelligent threat detection system, continuously analysing the entire AWS infrastructure for malicious or unauthorised behaviour. It monitors every API Call, network connection and data flow across the four VPCs. When a potential threat is detected – whether it’s unusual API calls from the production VPC or suspicious network traffic targeting the data VPC – GuardDuty immediately flags this activity.

For example, if an attacker attempts to access the Aurora database in the Data VPC using compromised credentials, GuardDuty’s machine learning models identify this as anomalous behaviour based on historical access patterns. This detection triggers an automated response through my Lambda functions, potentially blocking the suspicious IP and alerting the security team.

**Security Hub** acts as the nerve centre of the security operations, aggregating and prioritising security findings from multiple sources. In this banking system, it's particularly crucial as it ensures compliance with both PCI DSS for payment processing and CIS benchmarks for overall security best practices.

When GuardDuty detects a threat, or AWS Config identifies a compliance drift, these findings flow into Security Hub. Here, they're assessed, prioritised, and correlated with other security data. This centralisation is vital because it provides a single source of truth for the security status across all four VPCs and their various components.

**AWS Config** also plays a critical role in maintaining the security of the infrastructure by continuously monitoring and evaluating the configuration of AWS resources – this is particularly valuable for ensuring that security controls remain consistent and compliant across all VPCs.

For instance, AWS Config tracks every change made to security group rules, encryption settings, and access controls. If someone modifies a security group in the Production VPC to allow broader access than the bank’s compliance requirements permit, AWS Config immediately detects this change. It then triggers a notification and, through my automated remediation workflows, can revert the unauthorised change.

**Amazon Macie** adds another crucial layer to security monitoring by focusing specifically on sensitive data discovery and protection.

Macie continuously scans the S3 buckets where customer financial records, transaction logs, and compliance documents are stored. Configured to run weekly scans with findings published every fifteen minutes, Macie uses machine learning to identify and classify sensitive data patterns specific to banking – such as credit card numbers, bank account details, and personal identification information.

When Macie discovers sensitive data that's improperly stored or accessed, it alerts through Security Hub, integrating with my existing security workflow. For instance, if customer financial data is accidentally uploaded to a bucket without proper encryption, Macie immediately flags this as a critical finding, enabling quick remediation.

This layered security approach contributes to an automated defence strategy that protects each component of the banking system without manual intervention, and is particularly crucial for maintaining compliance with banking regulations like GDPR and PSD2.

**Understanding the Application Layer**

**API Gateway: API Management and Security**

API Gateway serves as a sophisticated interface for the core banking services, operating as a regional endpoint, specifically designed to handle banking transactions with stringent controls.

This setup includes comprehensive usage plans that enforce strict rate limiting – allowing 100 million requests per day with a burst limit of 5000 request per second and a standard rate limit of 1000 requests per second. These limits help protect the backend services from overwhelming traffic while ensuring fair resource allocation among consumers of a global banking application. The production stage of this API includes a 118GB cache to improve response times to high transaction volume, frequent requests, multiple regions/currencies/customer profile caching and exchange rate cachingand X-Ray tracing to provide deep visibility into request flows. Logging is also enabled to capture essential request metadata, including IP addresses, caller identities, and latency metrics – crucial for security auditing and performance monitoring.

API Gateway can also invoke a Lambda function for fraud detection, where Lambda queries DynamoDB for historical transaction data and publishes alerts to SNS if suspicious activity is detected. The final purpose of API Gateway is to route internet traffic to the EKS microservices via ALB.

**Application Load Balancer: Intelligent Traffic Distribution**

Operating at Layer 7 (HTTP/HTTPS), the ALB serves as the primary entry point for customer requests, sitting in front of the EKS Cluster. The ALB routes traffic to the appropriate microservice based on URL paths – for example, when a customer initiates a transaction, the request hits /transactions/\* and routes to the Transaction Processing Service. This path-based routing enables clean service separation while providing features like SSL termination and health checks. The ALB also integrates with WAF and Shield Advanced, ensuring each request is screened for security threats before reaching the microservices.

**Amazon EKS: Heart of the Microservices**

At the core of the application layer sits Amazon Elastic Kubernetes Service (EKS), orchestrating the containerised microservices. This EKS cluster hosts three critical services: Account Management, Transaction Processing, and Authentication.

Each service runs in isolated containers, allowing independent scaling and deployment. For instance, during peak banking hours, the Transaction Processing service can automatically scale to handle increased load, while the Account Management service maintains its baseline capacity. This flexibility ensures optimal resource utilisation while maintaining consistent performance.

Using AWS's official EKS module for my first Kubernetes project was a prudent decision. This architecture required high reliability and security, and the AWS-maintained module provided battle-tested configurations and regular security updates. By leveraging this standard module, I benefited from AWS's experience in running EKS at scale while avoiding risks of custom configurations. It accelerated the development timeline by handling complex aspects like IAM integration and networking setup out of the box. For this migration project where compliance was paramount, using well-established tooling rather than reinventing the wheel was the best choice.

In terms of my specific Kubernetes configurations, I will be discussing those in a future article and how I optimised them for a global bank’s purpose.

**AWS Lambda: Event-Driven Processing**

**Security Group Guardian:**

This Lambda function is triggered by **AWS Config** change notifications, forming a critical part of the infrastructure-as-code governance. When Config detects modifications to any security group, it sends a detailed configuration change event to Lambda, which then evaluates the changes against our bank's security baseline.

The function parses the Config configurationItem to understand exactly what changed, comparing new ingress rules against our approved port list (443, 80, 22). For a highly regulated bank environment, this tight integration with Config provides a complete audit trail of security group changes and automated remediation actions. If a DevOps engineer deploys a change that doesn't align with the bank’s security policies, Lambda immediately reverts it and creates a detailed SNS notification, including which VPC was affected and what specific rules were unauthorised. This automated Config-Lambda pipeline ensures the network security policies are enforced consistently across all global environments, from development to production.

**Sensitive Data Guardian**:

Operating as part of the data security mesh, this Lambda function integrates with **Amazon Macie** to provide automated remediation for sensitive data exposures. In a banking environment handling millions of sensitive customer records, this function ensures that any unencrypted data discovered by Macie is immediately secured. It automatically enables encryption, logs the incident in **Security Hub** for compliance tracking, and alerts the security team. This automated response capability is essential for maintaining regulatory compliance across global operations and protecting customer data across all regions.

**Real-Time Fraud Detection Engine:**

 At the heart of the transaction processing pipeline, this Lambda function acts as the first line of defence against fraudulent activities. It analyses every transaction in real-time against multiple risk vectors – including velocity checks, amount thresholds, and unusual patterns. By leveraging DynamoDB for swift access to recent transaction history, it can make instant decisions on transactions happening anywhere in the global network. The function's modular risk factor approach allows the fraud team to continuously tune and adapt detection parameters based on emerging threats, while maintaining the sub-second response times customers require and expect.

**EC2 Bastion Host:**

I deployed a highly available Bastion Host setup in the Management VPC, implemented through an Auto Scaling Group spanning multiple availability zones. Running on t3.micro instances with Amazon Linux 2, these bastion hosts serve as secure jump servers for administrative access to all banking resources across VPCs (to manage RDS databases, perform system audits or run maintenance scripts across VPCs). The configuration ensures redundancy with a minimum of one and maximum of two instances, equipped with necessary database clients and secured through strict security group rules that only permit SSH access from authorised corporate IP ranges. This setup provides a robust and secure entry point for administrative tasks while maintaining high availability and controlled access patterns.

**Multi-Region Database Layer**

In architecting the bank's database infrastructure, I implemented a multi-layer approach combining **Aurora PostgreSQL** for transactional workloads, **DynamoDB** for global scalability, and **Redis** for high-performance caching. Each component was carefully configured to meet strict banking regulations while ensuring high availability and disaster recovery capabilities.

**Aurora PostgreSQL cluster** serves as the primary transactional database, configured with two write instances and two read replicas. I chose the db.r6g.large instance class to optimise for memory-intensive banking operations. The cluster is encrypted at rest using KMS keys and enforces SSL/TLS 1.2 for all connections. The parameter group configuration reflects banking-grade requirements – it logs all SQL statements taking over 1 second, maintains detailed connection tracking, and sets reasonable timeouts (2 hours for idle transactions, 10 minutes for statements) to prevent resource exhaustion.

For global scalability and session management, I deployed **DynamoDB** **global tables** across multiple regions (eu-west-1 and eu-west-2). The table design uses a flexible schema with composite primary keys (pk/sk) and multiple Global Secondary Indexes optimised for common access patterns – session lookups, user queries, and transaction history. The global tables feature ensures sub-second replication between regions, crucial for maintaining consistent user sessions across this global infrastructure.

The caching layer utilises **Redis 7.0** in a global configuration, with a primary replication group of three nodes and a corresponding secondary group in the Disaster Recovery region. The Redis cluster primarily handles real-time session management, caches frequently accessed customer profiles and balances, and stores temporary MFA tokens - all with strict encryption and 24-hour TTL policies. I opted for cache.r6g.xlarge instances to handle high-throughput caching needs. The Redis setup includes critical security features like encryption in-transit and at-rest, plus a conservative maxmemory-policy (volatile-lru) to manage cache eviction.

Backup and disaster recovery were paramount in this design. I implemented a comprehensive **AWS Backup** plan that includes:

* Daily backups with 90-day retention
* Weekly backups retained for a year
* Cross-region backup replication
* Automated backup of all critical components including Aurora, DynamoDB, Redis, and essential S3 buckets

For high availability, databases were deployed across multiple availability zones and implemented automatic failover. Aurora's read replicas include performance insights with a 7-day retention period, allowing proactive monitoring and to optimise query performance.

Security was woven throughout the design. All database resources reside in private subnets within the Data VPC, with strictly controlled access through security groups. Aurora accepts connections only from the production VPC CIDR range, while Redis enforces similar restrictions with additional transit encryption.

This multi-layered database architecture is robust and designed to handle millions of daily transactions while maintaining sub-second response times and providing the reliability expected of a global banking infrastructure. The combination of Aurora's ACID compliance, DynamoDB's global scalability, and Redis's high-performance caching gives the perfect foundation for banking operations.

**Storage Layer: S3 for Banking Compliance, Cost Optimisation and Security**

This S3 infrastructure is meticulously designed to handle diverse banking operational needs while maintaining strict compliance with financial regulations. Each bucket serves a specific purpose with carefully crafted lifecycle policies for cost optimisation and security controls.

For audit trails, the **CloudTrail bucket** captures all API activities, while VPC Flow Logs and ALB logs provide comprehensive network visibility. These logs follow a tiered storage strategy - starting in Standard, transitioning to Glacier IR after 90 days, and moving to Deep Archive after a year. These logs are maintained for seven years to meet regulatory requirements, with versioning enabled for tamper protection.

Security is paramount - all buckets are encrypted using **KMS keys** with bucket keys enabled for cost optimisation. Public access is completely blocked through multiple layers of controls, including bucket policies and explicit public access blocks.

For the **CI/CD pipelines**, separate buckets handle **Lambda** and **EKS deployment artifacts**, with shorter retention periods (180 and 365 days respectively) and cost-effective transitions to STANDARD\_IA storage.

The **Audit Reports Bucket** aggregates findings from CloudWatch, GuardDuty, Security Hub, and Config, providing a centralised location for security analytics while maintaining the same stringent security controls and seven-year retention policy mandated for banking records.

**Monitoring with CloudWatch and SNS**

**Security Monitoring:** I implemented extensive security monitoring through multiple layers. WAF monitoring includes alarms for both blocked and allowed requests, triggering when thresholds are exceeded (100 requests in 5 minutes), helping identify potential attacks early. Macie monitoring is configured to immediately alert on any sensitive financial data discoveries in our S3 buckets, with Event Bridge rules specifically filtering for high-severity financial data findings. The Network Firewall monitoring tracks incoming traffic patterns, with alarms set for unusual spikes exceeding 1MB per 5-minute window.

**Infrastructure Performance:** For the database layer, I maintained separate CloudWatch log groups for Aurora and DynamoDB with 90-day retention periods. Aurora CPU utilisation is closely monitored with alarms triggering at 80% utilisation, while DynamoDB is monitored for write throttling events - any throttling immediately triggers an alert as it could impact transaction processing. ALB monitoring includes alarms for 5XX errors, triggering if it exceeds 5 errors in a 10-minute period.

**API and Serverless:** Dedicated log groups track API Gateway and Lambda functions, essential for debugging and performance optimisation of the serverless architecture. These logs are retained for 90 days, providing sufficient history for incident investigation while managing costs.

**Alerting Strategy:** All alerts are centralised through an **SNS topic** for global banking alerts, ensuring the operations team receives immediate notifications of any issues. The alerting thresholds are carefully tuned to minimise alert fatigue while ensuring we catch critical issues:

* Zero-tolerance for DynamoDB throttling events
* Immediate alerts for sensitive data discoveries
* 5-minute evaluation periods for most infrastructure metrics
* Two consecutive evaluation periods required for most alarms to reduce false positives

**Cost Optimisation**: I optimised monitoring costs by standardising log retention to 90 days in CloudWatch, with longer-term logs archived to S3 with lifecycle policies. This approach balances operational needs with cost efficiency while maintaining compliance requirements.

This monitoring setup ensures the maintenance of high availability and security expected of a banking platform, while providing the visibility needed for both operations and compliance teams.

**Reflection**

This has been the toughest project I have completed so far. Implementing a multi-VPC Hybrid architecture using Transit Gateway and Direct Connect lead me down numerous rabbit holes. Assigning the correct destination cidr blocks for specific VPCs to interact, whilst also configuring each service within the VPCs and a multitude of security groups and IAM Roles was so confusing at first. However, I stuck at the process, outlined a step-by-step approach to configure the architecture in a comprehensive order. I deployed the four VPCs, Transit Gateway and Direct Connect first to AWS to ensure my foundations were sorted – once that had been completed, the rest of the design process was relatively smooth.

In this project, I focused more in depth on security, compliance, monitoring, and cost-optimisation as I aimed to fit the AWS Well-Architected Framework as best as I could. This meant attention to detail when configuring services to suit these requirements. It also gave me a better understanding of how the services integrate optimally to be delivered to a global customer-base.

I recently began organising all services into specific modules rather than making a new file for each service and storing them in the root, so I strengthened my understanding of how modules interact using outputs and variables. Not only does this make my architecture more organised, but it allows for repeatability and faster updates to code.

I will refine this architecture, making it more cost-optimised and secure, as well as constructing suitable pipelines for CI/CD, delving deeper into self-remediation with Lambda functions, and enhancing my Kubernetes configuration.

My biggest takeaway from this project was the fact that I was comfortable being uncomfortable. Although I know how to write Terraform, Python and YAML, using these skills to construct an industry-standard architecture is another story. I am still early on in my Cloud journey, however every project is an insurmountable level of knowledge gained, driving me to pursue more. I look forward to further enhancing this architecture and adding even more skills to my repertoire.