This plan provides a structured approach to migrate the monolith to microservices on AWS with minimal downtime and a focus on data correctness. Each step can be expanded with detailed tasks, assignments, and timelines in subsequent iterations.

1. Pre-migration Assessment and Planning

- **Application Inventory**: List all endpoints, background jobs, dependencies, and third-party integrations.
- **Microservices Boundaries**:
    - Use Domain-Driven Design (DDD) to identify bounded contexts and define service boundaries, considering decomposition strategies like vertical slices (use case-focused) or functional services (step-focused).
        - **Vertical Slices**: Each microservice solves a particular use case or set of tightly-related use cases; code reuse via shared libraries.
        - **Functional Services**: Each service handles one particular step, integration, state, or thing; code reuse via service invocation.
        - Avoid combining these strategies or attempting to migrate between them.
    - Evaluate a modular monolith as an intermediate step to achieve loose coupling within a single deployment unit before full microservice extraction.
    - Assess potential performance costs (latency, duplicate calls, independent DB queries) introduced by microservices.
    - **Prerequisite**: Ensure the monolith is well-modularized with clearly defined responsibilities before attempting microservice extraction.
- **Database Analysis**: Document all tables, relationships, and data access patterns. Identify potential challenges in breaking the database.
- **Non-functional Requirements**: Define SLAs, scalability, security, and compliance requirements.
    - Rigorously assess the impact of microservices on latency and overall cost, as these factors can become significant challenges at scale.
- **Migration Goals**: Set clear milestones and success metrics (e.g., reduce latency, improve deployment frequency), focusing on solving real underlying problems rather than adopting microservices as a prestige project, and accurately estimating operational costs.
- **Risk Assessment & Rollback Strategy**: identify risks, dependencies, and rollback approach.

2. Foundational Setup

- **Tooling and Standards Definition**: Establish standards for coding, API design (e.g., RESTful maturity model, gRPC), containerization, and IaC before development begins.
- **CI/CD Blueprints**: Create reusable CI/CD pipeline templates for building, testing, and deploying microservices to ensure consistency.
- **Shared Infrastructure Provisioning**: Use Infrastructure as Code (IaC) to provision foundational AWS components like the VPC, subnets, IAM roles, and the initial API Gateway setup.
- **Observability Stack Setup**: Configure centralized logging (CloudWatch), metrics (Prometheus/Grafana), and distributed tracing (X-Ray/OpenTelemetry) before the first service is migrated.

3. Architecture Design

- **Microservices Architecture**: Each service is independently deployable, scalable, and focused on a single business capability.
- **Data Management**: Each service owns its data. Use events for inter-service communication to maintain data consistency. Address challenges like data duplication, atomicity of distributed transactions, and data consistency. Consider patterns like Saga for coordinating transactions and API composition or CQRS for data retrieval.
- **API Design**: Design APIs for each service. Use API Gateway as a single entry point.
- **Communication Patterns**: Use synchronous communication (HTTP/REST/gRPC) for immediate responses and asynchronous (message queues) for background tasks.
- **Security**: Implement a zero-trust model with OAuth2/JWT for service-to-service and user authentication. Use AWS Secrets Manager for secrets and KMS for encryption.
    - **Authentication & Authorization**: For user-facing services, use frameworks like OIDC/OAuth 2.0 (e.g., Amazon Cognito). For service-to-service, use mTLS (e.g., Amazon App Mesh) or JWTs.
    - **Increased Attack Surface**: Apply defense-in-depth principles. Place non-public service endpoints in private subnets, restrict access with NACLs and security groups, and use perimeter security tools (e.g., AWS WAF, AWS Shield) for public-facing services.
    - **Data Encryption & Protection**: Encrypt all traffic (client-to-service and inter-service) using ACM or Private CA. Enforce encryption at rest for all application data using KMS.
    - **Database Isolation**: Enforce application and database-level isolation to reduce the 'blast radius' of security events, controlling access via database users or AWS IAM roles.
- **Observability**: Centralized logging, metrics, and distributed tracing. Implement observability tools suitable for microservice architectures (e.g., Amazon CloudWatch, Logstash, OpenSearch for logs; AWS X-Ray for distributed tracing) to pinpoint errors and performance bottlenecks in distributed systems.

4. Infrastructure Setup on AWS

- **Compute**: Containerize all microservices using Docker. Use ECS with Fargate as the default orchestration platform for simplicity and serverless scaling. Consider EKS for workloads that need advanced Kubernetes features or multi-cloud portability. Use AWS Lambda for lightweight, event-driven connectors and background tasks. Ensure proper resource sizing and auto-scaling for ECS/EKS and Lambda functions.
    - **Containers**: Choose between Linux (smaller images, faster startup, no OS license costs) and Windows containers (for .NET Framework dependencies). Host on Amazon EC2 (faster startup with local caching, but requires OS patching) or AWS Fargate (serverless, efficient resource use, lower management overhead, but potentially longer startup).
    - **Container Orchestration**: For simpler applications or less container experience, Amazon ECS is a good choice. For large-scale applications or existing Kubernetes users, Amazon EKS offers advanced orchestration.
    - **Serverless Functions (AWS Lambda)**: Ideal for highly distributed, event-driven, or message-based applications. Resources are allocated only when needed, scaling automatically in response to events. Requires code refactoring for compatibility with the serverless invocation model and use of serverless-specific IaC tools (e.g., AWS SAM).

    - **Compute Choice Factors**: Consider business benefits vs. complexity, team skillset/training needs, future scalability, security/compliance, and automation tool compatibility.
- **Data**: Use Amazon RDS/Aurora for relational databases, matching your current SQL Server schemas if migrating. For high-throughput key-value workloads, use DynamoDB, and employ ElastiCache (Redis/Memcached) for caching frequently accessed data. Use S3 for storing artifacts, logs, and large objects, with lifecycle policies for cost optimization. Consider cross-region replication for critical data.
    - **Polyglot Persistence**: Leverage the variety of cloud database options to pick the right purpose-built database for each microservice, optimized for its use case.
    - **Common Database Choices**:
        - **Relational (RDBMS)**: For fixed data structures and ACID transactions (e.g., MySQL, PostgreSQL, Amazon Aurora).
        - **Document Databases**: For schema-less data, often read-intensive workloads (e.g., MongoDB, Amazon DocumentDB).
        - **Key-Value Databases**: High-performance storage/retrieval of unstructured data (e.g., Amazon DynamoDB for user profiles, session state).
        - **In-Memory Databases**: Sub-millisecond access for frequently accessed, ephemeral data or caching (e.g., Redis, Memcached, Amazon ElastiCache).
        - **Time Series Databases**: For high-throughput temporal data (e.g., Amazon Timestream).
        - **Graph Databases**: For data with complex connections (e.g., Neo4j, Amazon Neptune).
        - **Ledger Databases**: For cryptographically verifiable, immutable transaction history (e.g., Amazon QLDB).
    - **Database Choice Factors**: Consider business value vs. expertise/effort, future business needs, projected data size/performance, and high availability/resilience requirements.
- **Message Bus**: Use Amazon SNS/SQS for decoupled messaging between services. For event-driven architecture, leverage EventBridge, ensuring proper schema management and retry policies. Include dead-letter queues for error handling.
- **Networking/Security**: Design a VPC with public and private subnets, NAT gateways for outbound access, and security groups/NACLs for granular access control. Implement IAM roles per service, Secrets Manager/Parameter Store for credentials, and KMS for encryption at rest. Apply least-privilege principles and audit IAM policies regularly.
- **API Gateway**: Serve as a single entry point to route requests to microservices or the monolith. Implement rate limiting, throttling, and caching at the gateway level to enhance performance and reliability.
- **Infrastructure as Code (IaC)**: Define all infrastructure using Terraform or CloudFormation to ensure reproducible, version-controlled environments. Include modular templates for common patterns (VPC, RDS, ECS/EKS clusters, networking).

5. Incremental Migration Execution (Strangler Fig Pattern)

- **Lift and Shift Monolith (Optional but Recommended)**: Deploy the monolith to AWS (EC2 or ECS) to reduce network latency between it and the new microservices during the transition.

- **Identify and Prioritize the First Module**: Choose a low-risk, loosely coupled module with minimal dependencies to extract first as a pilot.
    - Consider starting with a Proof of Concept (POC) for a new or existing feature to clarify challenges and consequences.
    - **Component Selection Best Practices (Strangler Fig)**:
        - Select components with good test coverage and low technical debt.
        - Select components with independent scaling requirements.
        - Select components with frequent business requirement changes and frequent deployment needs.
- **Implement the First Microservice**: Develop the microservice in .NET Core, containerize it, and set up its own database and CI/CD pipeline based on the established blueprints.
    - For code separation, first refactor and test monolithic code internally, then move validated functionalities to microservice projects.
    - For .NET projects, prioritize upgrading to .NET Core/containerization as a prerequisite or early step.
- **Data Synchronization Strategy**:
    - **Dual Writes**: Write to both the old and new databases simultaneously.
    - **Change Data Capture (CDC)**: Use AWS DMS or Debezium to capture changes from the monolith's database and apply them to the new database.
    - For live systems, add timestamps to data, create new tables, use CDC (e.g., DynamoDB Streams) for future writes, copy old data (e.g., via S3 export + Glue or custom scripts) with duplicate handling, and then switch to new tables. Each microservice owns its data, with access only via its API, enabling independent scaling and capacity allocation.
    - Consider a two-step data migration: initially allow microservices to write to the monolithic database, then gradually migrate data per service, acknowledging that data is often the biggest challenge.
    - For relational databases, weigh the trade-offs of one DB cluster per microservice (cost) versus different databases/schemas in the same cluster (management, independent scaling). Consider Aurora Serverless with its cost implications.
- **Traffic Routing**: Use API Gateway to route specific requests to the new microservice. Gradually shift traffic.
- **Data Validation and Reconciliation**: Implement automated scripts to continuously compare data between the old and new databases to ensure correctness during the migration phase.
- **Decommission Monolith Module**: Once 100% of traffic is routed to the new service and data is validated, decommission the corresponding code and database tables from the monolith.
- **Iterate**: Repeat the process for the next prioritized module.
- **Assistive Refactoring Tools**: Consider using tools like AWS Microservice Extractor for .NET to simplify the process of refactoring older monolithic applications into smaller code projects, providing analysis, visualization, grouping, and automated code transformation.

6. Testing Strategy

- **Unit Tests**: For each service's internal logic.
- **Integration Tests**: Test interactions between a service and its direct dependencies (e.g., database, message queue).
- **Contract Tests**: Ensure that services adhere to their API contracts to prevent breaking changes between consumers and providers.

- **End-to-End Tests**: Test critical user journeys that span multiple services.
- **Performance Tests**: Load test each service and the system as a whole to validate non-functional requirements.
- **Resilience and Chaos Tests**: Intentionally introduce failures (e.g., service downtime, network latency) to test system resilience and fallback mechanisms.

7. Go-Live, Stabilization, and Iteration

- **Continuous Go-Live**: Recognize that "Go-Live" is not a single event but a continuous process of incrementally strangling the monolith.
- **Post-Deployment Monitoring**: After each service migration, closely monitor system health, latency, error rates, and business metrics. Utilize tools like Amazon CloudWatch, Splunk, NewRelic, Amazon OpenSearch Service for log analysis and threat detection, and Amazon Managed Service for Prometheus/Grafana for metrics.
- **Performance Tuning**: Adjust resource allocation and service configurations based on real-world load.
- **Cost Analysis and Optimization**: Regularly review AWS costs against projections and implement optimization strategies (e.g., rightsizing, Savings Plans).
- **Retire Monolith**: Once the final module is extracted, plan the final shutdown of the monolith application and its dedicated infrastructure.
- **Post-Mortem and Knowledge Sharing**: After each significant migration milestone, document lessons learned an

8. Long-Term Operations & Maintenance

- **Automated Operations**: Configure auto-scaling for each service and implement automated backup, disaster recovery drills.
- **Runbooks & On-call**: Maintain incident response playbooks and ensure proper on-call coverage.
- **Governance and Standards Evolution**: Regularly review and update architectural standards, security policies, and cost management practices.
- **Continuous Improvement**: Collect feedback, measure SLOs, and continuously refine services and processes.