Serverless Migration Strategy Guide

Breaking Down Monoliths Using Set Piece Methodology

Table of Contents

---

1. Introduction

Modern organizations face a critical challenge when adopting serverless architectures: how to break down complex legacy monoliths or design new ambitious initiatives in a structured, manageable way. This document outlines a comprehensive migration strategy based on **set piece methodology**, combining Domain-Driven Design (DDD), Event-Driven Architecture (EDA), and serverless-first principles.

Key Philosophy

> **"Think big; act small; fail fast; learn rapidly."**
> — Mary and Tom Poppendieck, *Lean Software Development*

This strategy emphasizes:

- **Granular thinking** at the service level
- **Incremental development** and deployment
- **Deep operational visibility** and control
- **Separation of concerns** and isolation for resilience

---

2. The Challenge of Legacy Systems

Common Problems Faced

Organizations encounter various challenges when dealing with existing systems:

Legacy Monoliths

- **Organic growth**: Simple applications that evolved with bolted-on features
- **Technical debt**: Mix of technologies, customizations, and workarounds
- **Complexity**: Difficult to understand, modify, or maintain
- **Scalability issues**: Cannot scale individual components independently

New Product Development

- **Breadth and depth**: Modern applications span from frontend to complex backend logic
- **Global availability**: Must serve users worldwide
- **Data-intensive**: Process massive volumes of data for fine-grained insights
- **Speed to market**: Pressure to deliver quickly while maintaining quality

The Core Question

Where and how do you start? Once started, **how do you progress in the right direction?**

---

3. Vision and Focus Framework

Understanding Vision

Vision represents the complete picture—what you want to achieve:

- The entire application or system
- The business domain as a whole
- The end goal or desired outcome
- The "forest" view of the problem

Understanding Focus

Focus is the instrument to achieve your vision:

- Concentrating on smaller portions or parts
- The "trees" within the forest
- Actionable, manageable pieces
- Progressive achievement of incremental goals

The Vision-Focus Cycle

```
┌─────────────────────────────────────────────┐
│              OVERALL VISION                  │
│         (The Complete System)                │
└─────────────────────────────────────────────┘

           ↓

    ┌──────────────┐
    │   Focus on   │
    │   Part 1     │ → Complete → Move to Part 2
    └──────────────┘

           ↓

    ┌──────────────┐
    │   Focus on   │
    │   Part 2     │ → Complete → Move to Part 3
    └──────────────┘

           ↓
         [Continue...]
```

The Cosmos Analogy

Think of a complex problem like viewing the night sky:

1. **Initial View**: You see a vast canvas of bright dots (the vision)
2. **Zoom In**: Focus on one bright dot—it becomes a galaxy (sub-vision)
3. **Deeper Focus**: Within the galaxy, focus on a star system
4. **Continue**: Each star has planets, each planet has features
5. **Result**: Break impossibilities into possibilities through iterative focus

Key Lesson: Carefully analyze the task at hand and break it into manageable pieces.

---

4. Set Piece Methodology

What is a Set Piece?

The term "set piece" comes from:

- **Film production**: Individual scenes filmed in any order, then edited together
- **Theater**: Realistic scenery built to stand independently
- **Music**: Individual parts of a composition written, rehearsed, recorded separately
- **Sports**: Pre-planned plays practiced and executed

Characteristics of a Set Piece

1. **Part of a whole**: Each piece contributes to the overall vision
2. **Focused work**: Teams concentrate on one piece at a time
3. **Adequate planning**: Each piece requires design and architecture
4. **Testing essential**: Rehearsal and validation before integration
5. **Parallel development**: Different teams work on different pieces
6. **Integration**: All pieces are brought together to form the complete system

Applying Set Piece Thinking to Serverless

When migrating to or building with serverless:

Benefits

- **Clarity**: Clear understanding of different application parts
- **Incremental delivery**: Plan and develop solutions iteratively
- **Deep visibility**: Operational control at granular levels
- **Isolation**: Separation of concerns for resilience and availability
- **Team autonomy**: Engineers can own specific bounded contexts

Key Principles

- Engineers own part of the domain within bounded context boundaries
- Set piece mindset becomes easier within ownership boundaries
- Each piece can be developed, tested, and deployed independently
- Integration happens through well-defined contracts

---

5. Case Study: Customer Rewards System

Problem Statement

> **Business Requirement**: Your business needs to offer digital and physical rewards to online retail customers. Stakeholders create reward details in a CMS, which propagates changes to consumers. The rewards backend must track usage and apply business logic for issuing and redemption. A third-party CRM acts as a rewards ledger, receiving all updates.

Domain Analysis

Identified Elements

- **Business Domain**: Online retail / Ecommerce
- **Subdomain**: Customer
- **Bounded Context**: Customer Rewards

Key Observations

1. **Asynchronous operations**: Reward content created in advance, lifetime controlled by validity period
2. **External notifications**: CMS sends notifications (webhook pattern candidate)
3. **Data transformation**: Content requires cleansing and translation (Anti-Corruption Layer - ACL)
4. **User-facing APIs**: Frontend needs synchronous request/response (microservice pattern)
5. **Third-party integration**: CRM interaction requires ACL and resilience considerations

Phase 1: Initial Vision

High-Level Components (Figure 3-28):

```
Business Stakeholder → CMS → Rewards Backend → CRM
                             ↓
                          Website
                             ↑
                         Customer
```

Phase 2: Detailed Analysis

System Characteristics (Figure 3-29):

Component	Characteristics
CMS	- Content uploads use specified file type/format - Sends reward data - Every reward has validity period - Provides notifications of content changes
Rewards Backend	- Translation of rewards content data - Triggers content changes in CMS - Issues and redeems rewards - Mapping of rewards business data
CRM	- Sends reward, issuing, and redemption details - Must consider availability and quotas
Website	- Fetches reward details - Issues and redeems rewards - Rewards API for system interaction

Phase 3: Set Piece Identification

Identified Set Pieces (Figure 3-30):

1. content-upload

- **Type**: Independent manual activity
- **Responsibility**: Content creators upload to CMS
- **Future extension**: Potential uploader service
- **Current status**: Outside immediate scope

2. Frontend

- **Type**: Web application
- **Responsibility**: Customer interaction for finding and redeeming rewards
- **Scope**: Large and complex
- **Role**: Consumer of rewards service
- **Pattern**: Web frontend technologies

3. content-updates

- **Type**: Microservice
- **Responsibilities**:
  - Implement callback webhook for CMS notifications
  - Translate rewards data between CMS and backend
  - Update CMS for rewards data changes (model synchronization)
- **Pattern**: Webhook callback microservice
- **Communication**: Synchronous request/response + asynchronous webhooks

4. rewards-service

- **Type**: Microservice
- **Responsibilities**:
  - Core business logic for rewards
  - Provide rewards service to consumers (including frontend)
  - Coordinate with other services and systems
  - Handle issuing and redemption operations
- **Pattern**: API-based microservice
- **Communication**: Synchronous and asynchronous

5. rewards-crm

- **Type**: Microservice
- **Responsibilities**:
  - Data transformation between rewards backend and CRM
  - Update CRM system with rewards data
  - Handle operational constraints (SLA, downtime, quotas)
  - Implement resilience patterns
  - Potential future: Listen for CRM updates
- **Pattern**: Integration microservice with ACL
- **Communication**: Asynchronous with resilience patterns

Phase 4: Architecture Overview

Microservices Structure (Figure 3-31):

```
content-upload → CMS → [content-updates] → [rewards-service] → [rewards-crm] → CRM
                              ↑                    ↑
                          Frontend ──────────────┘
```

Key Points:

- Lines without arrows indicate bidirectional potential
- Each hexagon represents an independent microservice
- External systems (CMS, CRM) are integrated through ACL pattern

---

6. Communication Patterns in Serverless

Three Primary Communication Methods

1. APIs (Synchronous Request/Response)

- **Use case**: Real-time operations requiring immediate response
- **Example**: Frontend fetching reward details
- **Implementation**: Amazon API Gateway + AWS Lambda
- **Characteristics**:
  - Client waits for response
  - Timeout considerations
  - Direct coupling between consumer and service

2. Events (Asynchronous Publish/Subscribe)

- **Use case**: Decoupled communication, multiple consumers
- **Example**: Reward creation event consumed by multiple services
- **Implementation**: Amazon EventBridge
- **Characteristics**:
  - Publisher doesn't know consumers
  - Multiple subscribers possible
  - Eventually consistent
  - Enables event-driven architecture

3. Messages (Asynchronous Point-to-Point)

- **Use case**: Direct communication between producer and consumer
- **Example**: Processing queue for CRM updates
- **Implementation**: Amazon SQS, Amazon SNS
- **Characteristics**:
  - More direct than events
  - Decoupled but targeted
  - Reliable delivery
  - Order control options

Communication Architecture (Figure 3-32)

```
Frontend ←——————————— Synchronous API —————————————→ rewards-service
                                                            ↓
CMS ←—— Webhook API ——→ content-updates ←—— Events ——→ Event Bus ←—— Events ——→ rewards-crm → CRM
          Async                                                                        ↓
                                                                                  Async API
```

Integration Points:

Integration	Pattern	Communication Type
Frontend ↔ rewards-service	REST API	Synchronous request/response
CMS → content-updates	Webhook	Synchronous request/response + async callback
content-updates ↔ rewards-service	Event Bus	Decoupled asynchronous event-driven
rewards-service ↔ rewards-crm	Event Bus	Decoupled asynchronous event-driven
rewards-crm → CRM	REST API	Synchronous request/response + async webhook

Orchestration and Choreography

Choreography (Event-driven):

- Services react to events
- No central coordinator
- Loose coupling
- Example: Reward creation event triggers multiple services

Orchestration (Workflow-driven):

- Central coordinator (AWS Step Functions)
- Defined workflow
- Tight control
- Example: Complex multi-step reward redemption process

---

7. Building Microservices to Serverless Strengths

Key Mindset Shifts

1. Size Not Measured by Lambda Functions

Traditional Thinking:

- Microservice = collection of functions
- More functions = larger service
- Function count is a metric

Serverless Thinking:

- Microservice = composition of managed services
- Programming is part, not all of it
- Infrastructure as important as code
- Some microservices may have zero Lambda functions

Example Architecture:

```
API Gateway (HTTP API)
    ↓
Step Functions (Orchestration)
    ↓
DynamoDB (Direct integration)
    ↓
EventBridge (Event publication)
```

No Lambda required for:

- API Gateway → Step Functions integration
- Step Functions → DynamoDB integration (AWS SDK)
- Step Functions → EventBridge integration (native)

2. Infrastructure Definition as Code

Traditional Approach:

- Write business logic
- Deploy to existing infrastructure
- Infrastructure managed by separate team
- Clear separation between code and infrastructure

Serverless Approach:

- Infrastructure definition is part of development
- Choose tool at project start (CDK, SAM, CloudFormation, Terraform)
- Infrastructure code lives with business code
- Same team owns both

Example Stack:

- **Runtime**: Node.js with TypeScript
- **IaC Tool**: AWS CDK with TypeScript
- **Result**: Business logic and infrastructure both in TypeScript
- **Benefits**: Type safety, code reuse, unified testing

Serverless Microservice Characteristics

Composition Over Coding

```
// CDK Example: Composing infrastructure + business logic

const table = new dynamodb.Table(this, 'RewardsTable', {
  partitionKey: { name: 'rewardId', type: dynamodb.AttributeType.STRING }
});

const rewardsFunction = new lambda.Function(this, 'RewardsFunction', {
  runtime: lambda.Runtime.NODEJS_18_X,
  handler: 'index.handler',
  code: lambda.Code.fromAsset('lambda'),
  environment: {
    TABLE_NAME: table.tableName
  }
});

table.grantReadWriteData(rewardsFunction);

const api = new apigateway.RestApi(this, 'RewardsApi');
const rewards = api.root.addResource('rewards');
rewards.addMethod('GET', new apigateway.LambdaIntegration(rewardsFunction));
```

Native Service Integrations

Prefer native integrations over Lambda "glue code":

Instead of:

```
API Gateway → Lambda (passes through) → DynamoDB
```

Use:

```
API Gateway → DynamoDB (direct integration)
```

Benefits:

- Lower latency
- Reduced cost
- Fewer moving parts
- Less code to maintain

Right-Sizing Serverless Microservices

Factors to Consider

1. **Bounded Context Alignment**
   - Service boundaries match domain boundaries
   - Clear ownership and responsibility
2. **Communication Patterns**
   - Minimize synchronous cross-service calls
   - Prefer asynchronous event-driven patterns
3. **Data Ownership**
   - Each service owns its data
   - No shared databases between services
4. **Deployment Independence**
   - Can deploy without coordinating with other services
   - Backward-compatible APIs
5. **Team Ownership**
   - Small team can own entire service
   - Full-stack ownership (frontend to data)

Anti-Patterns to Avoid

✕ **Too Granular**:

- Lambda function per service
- Excessive inter-service communication
- Distributed monolith

✕ **Too Coarse**:

- Multiple bounded contexts in one service
- Difficult to deploy independently
- Monolith in serverless clothing

☑ **Right Balance**:

- Aligns with bounded context
- Independent deployment
- Clear API contracts
- Manageable complexity

---

8. Techniques for Identifying Set Pieces

Domain-Driven Design Approach

1. Break Down Business Domain

```
Business Domain (Ecommerce)
    ↓
Subdomains
    ├─ Customer
    ├─ Inventory
    ├─ Order
    └─ Shipping
        ↓
Bounded Contexts
    ├─ Customer Rewards ← Our focus
    ├─ Customer Profile
    └─ Customer Support
```

2. Identify Synchronous Interactions

Questions to Ask:

- What operations require immediate response?
- Which user-facing features need real-time data?
- What are the request/response API contracts?

Example (Rewards System):

- Frontend fetching reward details → API required
- Frontend redeeming reward → API required
- CMS webhook callback → API required

3. Isolate Asynchronous Operations

Questions to Ask:

- What can be done in the background?
- What operations don't need immediate results?
- What can benefit from eventual consistency?

Example (Rewards System):

- Content updates from CMS → Asynchronous event
- CRM updates → Asynchronous event
- Reward expiration processing → Scheduled background job

4. External System Interactions

Considerations:

- **Legacy systems**: May have limited APIs, require ACL
- **Third-party platforms**: Consider SLA, quotas, downtime
- **SaaS applications**: Webhook patterns, authentication
- **Data feeds**: Corporate data lake, analytics platforms

Example (Rewards System):

- CMS integration → content-updates microservice (ACL)
- CRM integration → rewards-crm microservice (ACL + resilience)

5. Administrative Functions

Group system-specific administrative activities:

- API client creation and management
- Credential rotation
- API usage quota monitoring
- System health checks
- Configuration management

Pattern: Admin microservice or management plane

6. Notification Services

Identify push notification requirements:

- Service-to-service notifications
- Consumer notifications (webhooks)
- Event broadcasting
- Status updates

Pattern: Notification microservice or event-driven architecture

7. Shared Resources and Reference Data

Identify common data accessed by multiple services:

- Size measurements, currency conversions
- Country codes, time zones
- Product catalogs
- Configuration data

Pattern: Reference data service or cached static resources

8. Observability Requirements

Consider monitoring and analysis needs:

- Log streaming and aggregation
- Metrics collection
- Distributed tracing
- Analysis and filtering
- Alerting

Pattern: Observability layer (CloudWatch, X-Ray, third-party tools)

9. Security and Compliance

Identify cross-cutting security concerns:

- Fraud prevention
- Data inspection
- User activity monitoring

- Compliance logging
- Audit trails

Pattern: Security layer or interceptor services

Decision Matrix

Characteristic	Microservice Candidate?	Considerations
Synchronous API required	☑ Yes	API Gateway + Lambda pattern
Asynchronous processing	☑ Yes	Event-driven or queue-based
External system integration	☑ Yes	ACL pattern, resilience
Administrative functions	⚠ Maybe	Group related admin tasks
Scheduled jobs	☑ Yes	EventBridge scheduled rules
Event sourcing	☑ Yes	Dedicated event store service
Reference data	⚠ Maybe	Consider caching vs. service
Cross-cutting concerns	⚠ Maybe	Layers vs. dedicated services

---

9. Implementation Best Practices

CI/CD Pipeline Structure (Figure 3-33)

Each microservice maintains independence:

```
content-updates Microservice:
    Commit → Build → Test → Stage → Production

rewards-service Microservice:
    Commit → Build → Test → Stage → Production

rewards-crm Microservice:
    Commit → Build → Test → Stage → Production
```

Benefits:

- **Parallel development**: No pipeline conflicts
- **Independent releases**: Deploy when ready
- **Team autonomy**: Different teams, different timelines
- **Reduced risk**: Smaller, focused deployments

Event-Driven Architecture (Figure 3-34)

Central Event Bus (Amazon EventBridge):

```
                    ┌─────────────────┐
                    │  Rewards Event  │
                    │      Bus        │
                    └────────┬────────┘
                             │
        ┌────────────────────┼────────────────────┐
        │                    │                    │
        ↓                    ↓                    ↓
┌───────────────┐   ┌───────────────┐   ┌───────────────┐
│content-updates│   │rewards-service│   │  rewards-crm  │
└───────────────┘   └───────────────┘   └───────────────┘
        │                    │                    │
        └──── Publish ───────┴──── Subscribe ────┘
             Events               to Events
```

Event Types:

- **rewards.created**: New reward configured
- **rewards.updated**: Reward details changed
- **rewards.deleted**: Reward removed
- **rewards.issued**: Reward given to customer
- **rewards.redeemed**: Customer used reward

Complete Architecture (Figures 3-34)

Integrated System:

```
content-upload → CMS ←─ CMS API ─→ content-updates
                  │                      ↓
                  │                  Publishes:
                  │                  rewards.created
                  │                  rewards.updated
                  │                  rewards.deleted
                  │                      ↓
Frontend ←─ /rewards API ─→ rewards-service ←── Event Bus ──→ rewards-crm ─── CRM API ──→ CRM
                                   ↓
                             Publishes:
                             rewards.issued
                             rewards.redeemed
```

Infrastructure Components

Per Microservice

- **Compute**: AWS Lambda functions
- **API**: Amazon API Gateway (REST or HTTP API)
- **Storage**: Amazon DynamoDB tables
- **Queues**: Amazon SQS queues (for resilience)
- **Events**: EventBridge rules and subscriptions
- **Monitoring**: CloudWatch Logs, Metrics, Alarms

Shared Resources

- **Event Bus**: Amazon EventBridge (rewards-system-bus)
- **Authentication**: Amazon Cognito or IAM
- **API Management**: API Gateway custom domain
- **Observability**: CloudWatch, X-Ray, CloudTrail

Deployment Strategy

1. Infrastructure as Code

```
rewards-system/
├── content-updates/
│   ├── infrastructure/    # CDK/SAM/Terraform
│   ├── src/              # Business logic
│   └── tests/
├── rewards-service/
│   ├── infrastructure/
│   ├── src/
│   └── tests/
└── rewards-crm/
    ├── infrastructure/
    ├── src/
    └── tests/
```

2. Staged Rollouts

1. **Development**: Individual developer environments
2. **Testing**: Shared testing environment
3. **Staging**: Production-like environment
4. **Production**: Gradual rollout (canary, blue/green)

3. Monitoring and Rollback

- **CloudWatch Alarms**: Automated alerts on errors
- **X-Ray Tracing**: Distributed request tracing
- **Automatic Rollback**: On alarm threshold breach
- **Manual Rollback**: Quick rollback capability

Testing Strategy

Unit Tests

- Business logic functions
- Data transformation functions
- Validation logic

Integration Tests

- API endpoint testing
- Database operations
- External service mocks

Contract Tests

- API contract validation
- Event schema validation
- Consumer-driven contracts

End-to-End Tests

- Full workflow testing
- Cross-service scenarios
- Production-like data

---

10. Conclusion

Key Takeaways

1. Vision and Focus

- Maintain the big picture (vision) while focusing on manageable parts
- Break down complex problems iteratively
- Use the cosmos analogy: zoom in progressively

2. Set Piece Methodology

- Treat each microservice as an independent set piece
- Plan, develop, test, and deploy in isolation
- Bring pieces together through well-defined integration points

3. Domain-Driven Design

- Align services with bounded contexts
- Respect domain boundaries
- Implement Anti-Corruption Layers for external integrations

4. Communication Patterns

- **APIs**: Synchronous request/response
- **Events**: Asynchronous, decoupled publish/subscribe
- **Messages**: Point-to-point asynchronous communication

5. Serverless Strengths

- Composition over coding
- Infrastructure as code
- Native service integrations
- Granular operational control

Success Factors

☑ **Do**:

- Break down monoliths into bounded contexts
- Identify set pieces systematically
- Use event-driven architecture for decoupling
- Implement ACL for external systems
- Deploy independently
- Monitor granularly
- Test thoroughly

✕ **Don't**:

- Build distributed monoliths
- Over-decompose into too-fine-grained services
- Create shared databases between services
- Ignore operational constraints (SLA, quotas)
- Skip testing for cost reasons
- Couple services tightly

Migration Path Forward

Phase 1: Discovery

1. Analyze existing monolith or new requirements
2. Identify business domains and subdomains
3. Define bounded contexts
4. Map current state

Phase 2: Planning

1. Apply set piece identification techniques
2. Define communication patterns
3. Design event schemas
4. Plan ACL implementations
5. Define API contracts

Phase 3: Implementation

1. Start with least risky set piece
2. Implement in isolation
3. Test thoroughly
4. Deploy independently
5. Monitor and learn

Phase 4: Integration

1. Connect set pieces via events
2. Implement choreography/orchestration
3. Test integrated workflows
4. Validate end-to-end scenarios

Phase 5: Optimization

1. Monitor performance and costs
2. Refine boundaries as needed
3. Optimize communication patterns
4. Enhance observability

Final Thoughts

The journey to serverless is not about technology alone—it's about:

- **Mindset**: Think in terms of managed services and composition
- **Discipline**: Apply structured methodologies like DDD and set pieces
- **Iteration**: Progress incrementally, learn continuously
- **Team Culture**: Embrace ownership and autonomy

By breaking down complex problems into manageable set pieces, focusing on one piece at a time, and bringing them together through well-designed integration patterns, organizations can successfully migrate to serverless architectures that are scalable, resilient, and maintainable.

Appendix A: Rewards System - Complete Specification

Set Piece: content-updates

Purpose: Handle content synchronization between CMS and rewards backend

Responsibilities:

- Implement webhook endpoint for CMS notifications
- Translate CMS data model to rewards model
- Update CMS when rewards change internally
- Implement Anti-Corruption Layer

APIs:

- `POST /rewards/content`: Webhook for CMS notifications

Events Published:

- `rewards.created`
- `rewards.updated`
- `rewards.deleted`

Events Subscribed:

- `rewards.*.internal.*`: Internal reward changes

Infrastructure:

- API Gateway (webhook endpoint)
- Lambda functions (transformation logic)
- DynamoDB (staging table)
- EventBridge (event publication)
- SQS (dead letter queue)

Set Piece: rewards-service

Purpose: Core rewards business logic

Responsibilities:

- Manage reward lifecycle
- Issue rewards to customers
- Process reward redemptions
- Coordinate with other services
- Serve frontend APIs

APIs:

- `GET /rewards`: List available rewards
- `GET /rewards/{id}`: Get reward details
- `POST /rewards/{id}/issue`: Issue reward to customer
- `POST /rewards/{id}/redeem`: Redeem reward

Events Published:

- `rewards.issued`
- `rewards.redeemed`
- `rewards.expired`

Events Subscribed:

- `rewards.created`
- `rewards.updated`
- `rewards.deleted`

Infrastructure:

- API Gateway (REST API)
- Lambda functions (business logic)
- DynamoDB (rewards table, customer-rewards table)
- Step Functions (complex redemption workflows)
- EventBridge (event pub/sub)

Set Piece: rewards-crm

Purpose: Integration with external CRM system

Responsibilities:

- Transform rewards data for CRM
- Update CRM with reward activities
- Handle CRM availability issues
- Implement retry and resilience patterns
- Respect CRM quotas and rate limits

Events Subscribed:

- `rewards.issued`
- `rewards.redeemed`
- `rewards.expired`

Infrastructure:

- Lambda functions (CRM integration)
- SQS (buffering and retry queue)
- DynamoDB (state tracking)
- EventBridge (event subscription)
- CloudWatch (monitoring and alarms)

Appendix B: Further Reading

Books

- **Domain-Driven Design** by Eric Evans
- **Lean Software Development** by Mary and Tom Poppendieck
- **Building Event-Driven Microservices** by Adam Bellemare
- **AWS Lambda in Action** by Danilo Poccia

AWS Services Referenced

- **AWS Lambda**: Serverless compute
- **Amazon API Gateway**: API management
- **Amazon EventBridge**: Event bus
- **Amazon DynamoDB**: NoSQL database
- **Amazon SQS**: Message queuing
- **AWS Step Functions**: Workflow orchestration
- **Amazon CloudWatch**: Monitoring and logging
- **AWS X-Ray**: Distributed tracing
- **AWS CDK**: Infrastructure as code
- **AWS SAM**: Serverless application model

Patterns and Practices

- Anti-Corruption Layer (ACL)
- Event-Driven Architecture (EDA)
- Domain-Driven Design (DDD)
- Bounded Contexts
- EventStorming
- Choreography and Orchestration
- Circuit Breaker Pattern
- Saga Pattern

Document Version: 1.0
Last Updated: October 2, 2025
Authors: Based on serverless migration best practices and enterprise case studies

---

This document provides a comprehensive guide to migrating legacy monoliths to serverless architectures using the set piece methodology. Organizations should adapt these strategies to their specific contexts, business domains, and technical constraints.