Module 3: Dataverse Deep Dive Part 2

Advanced auditing, integration patterns, and performance optimization strategies for enterprise Dataverse implementations

Session Objectives

01

Master Auditing Frameworks

Implement field-level auditing and compliance tracking for enterprise data governance

02

External Integration Patterns

Leverage dataflows and virtual tables for seamless external system connectivity

03

Custom Development Strategies

Build scalable plugins and process actions for complex business logic

04

Performance Optimization

Design efficient indexing strategies and handle platform throttling constraints

Auditing & Change Tracking

Building comprehensive audit trails for enterprise compliance

Field-Level Auditing Architecture

Dataverse provides granular auditing capabilities that track changes at the field level, enabling comprehensive compliance reporting and data governance. The audit system captures who made changes, when they occurred, and both old and new values.

Audit logs are stored in separate tables optimized for retention and query performance. The system supports configurable audit policies at the organization, entity, and field levels, allowing precise control over what gets tracked.

Understanding audit data structure is crucial for building effective compliance dashboards and automated reporting solutions.

Audit Configuration Strategies

Organization Level

Global audit settings control overall system behavior

- Enable/disable audit logging
- Set retention policies
- Configure audit log cleanup

Entity Level

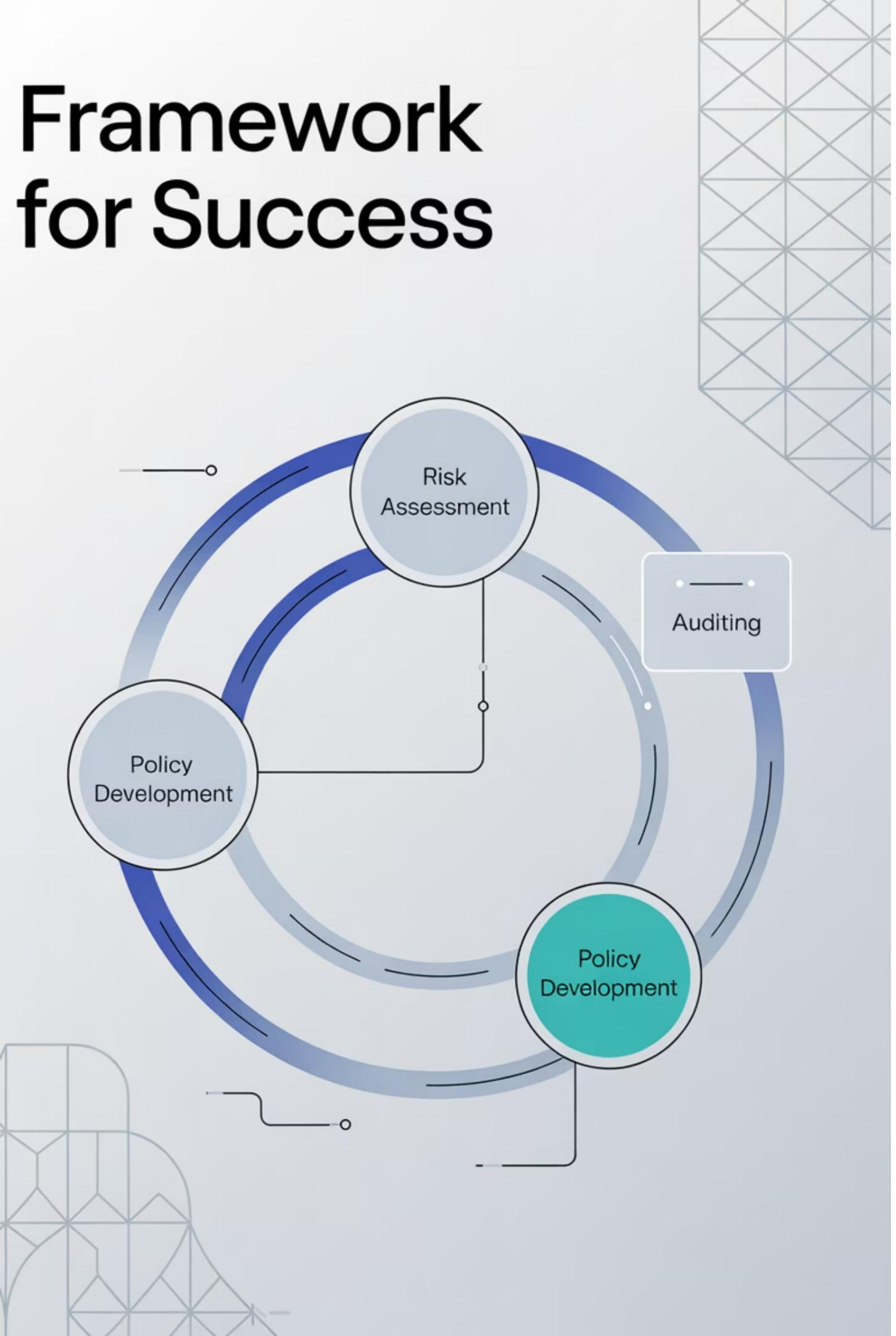
Table-specific audit configuration for targeted monitoring

- Enable auditing per table
- Track create/update/delete operations
- Configure access auditing

Field Level

Granular control over individual column tracking

- Select specific fields to audit
- Exclude sensitive data
- Optimize storage usage



Compliance Strategy Implementation

Effective compliance strategies require aligning audit configurations with regulatory requirements. GDPR, HIPAA, and SOX each have specific data tracking and retention mandates that must be reflected in your Dataverse audit design.

Key considerations include automated audit report generation, role-based access to audit data, and integration with external compliance management systems. The audit API enables custom reporting solutions that can aggregate and analyze change patterns across your entire data estate.

Audit Data Querying Patterns

```
// Retrieve audit records for specific entityvar auditQuery = new QueryExpression("audit"){    ColumnSet = new
ColumnSet("createdon", "userid", "operation", "oldvalues", "newvalues"),    Criteria = new FilterExpression    {
Conditions =         {            new ConditionExpression("objectid", ConditionOperator.Equal, recordId),
new ConditionExpression("createdon", ConditionOperator.GreaterThan, DateTime.Now.AddDays(-30))        }    },
Orders = { new OrderExpression("createdon", OrderType.Descending) }};
```

Efficient audit data retrieval requires understanding the audit table structure and implementing appropriate filtering strategies to manage large datasets and optimize query performance.

Performance Considerations for Auditing

Storage Impact

Audit logs consume significant storage space. Implement retention policies and archive strategies to manage long-term costs while maintaining compliance requirements.

Query Performance

Large audit tables require optimized query patterns. Use appropriate date ranges and entity filters to prevent performance degradation in reporting scenarios.

Selective Auditing

Enable auditing only for critical fields and entities. Over-auditing can impact system performance and generate excessive noise in compliance reports.

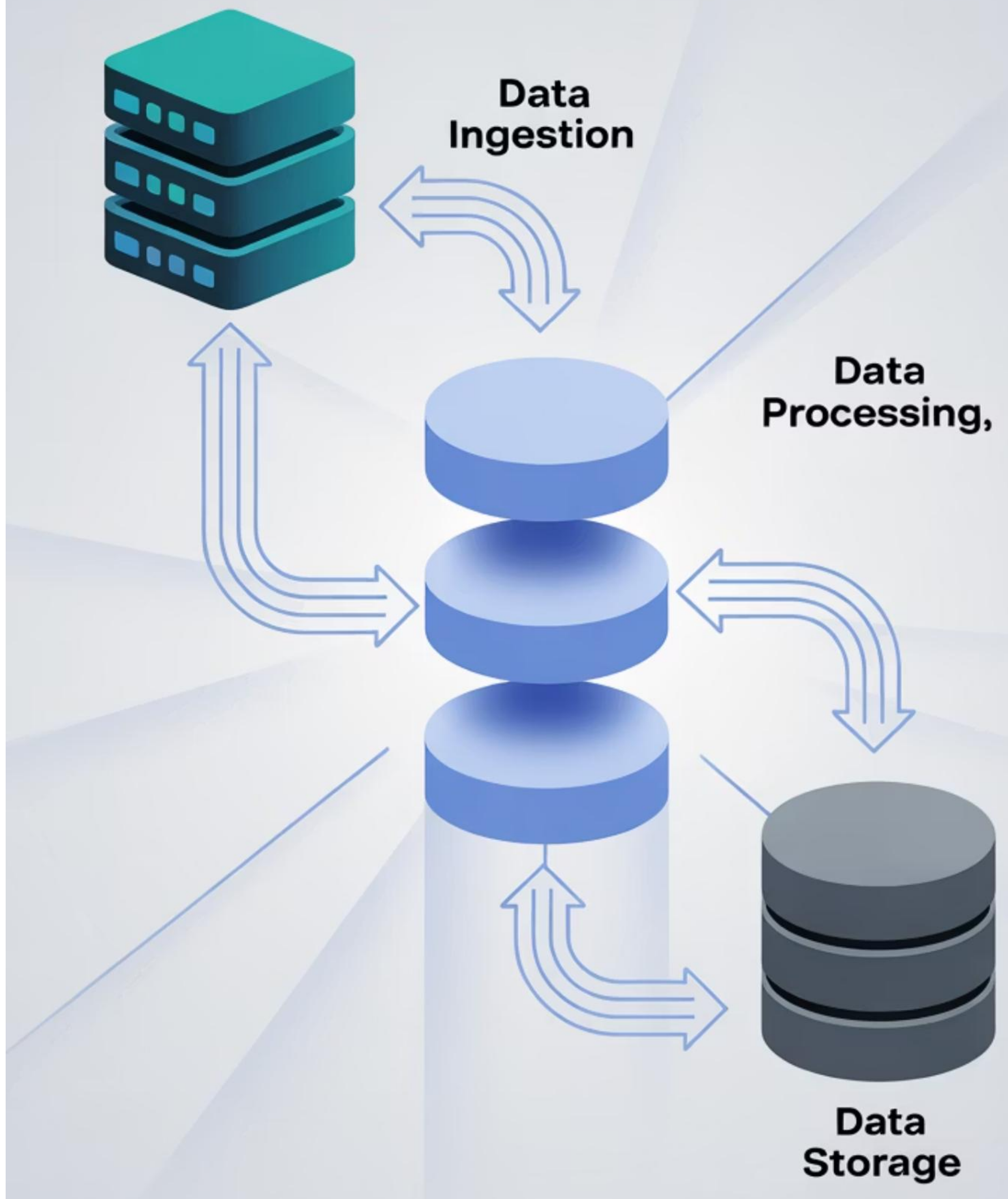
External Integration

Dataflows and virtual tables for seamless connectivity

Dataflows Architecture Overview

Dataflows provide a no-code/low-code approach for ingesting, transforming, and loading data from external sources into Dataverse. Built on Power Query technology, dataflows enable complex ETL operations with visual designers while maintaining enterprise-grade scalability and monitoring.

The architecture supports scheduled refreshes, incremental loading, and error handling, making it suitable for production data integration scenarios. Dataflows can source from hundreds of connectors including databases, APIs, files, and cloud services.



Dataflow Design Patterns

1

Data Extraction

Connect to source systems using appropriate connectors and authentication methods

2

Transformation Logic

Apply business rules, data cleansing, and schema mapping using Power Query M functions

3

Load Strategy

Configure incremental refresh and upsert operations for optimal performance

Virtual Tables Deep Dive

Virtual tables enable real-time access to external data without replication, presenting external systems as native Dataverse tables. This approach reduces data latency, eliminates synchronization complexity, and maintains a single source of truth.

OData Providers

Connect to systems exposing OData endpoints for standardized data access

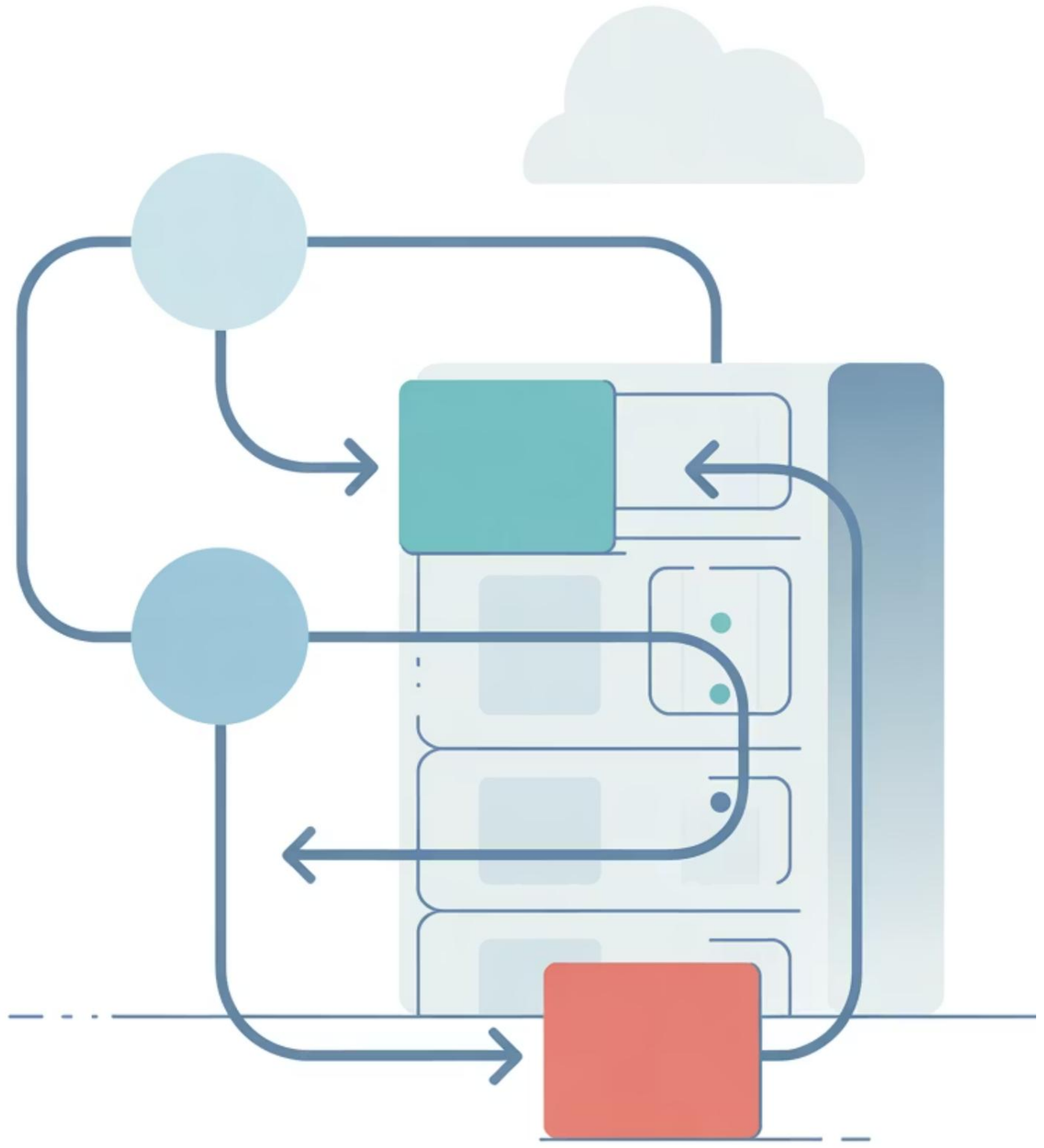
Custom Providers

Build custom data providers using the Virtual Table SDK for proprietary systems

Azure SQL Connector

Direct integration with Azure SQL databases for seamless cloud connectivity

Project Nexus



Virtual Table Implementation Considerations

Virtual tables require careful consideration of network latency, authentication patterns, and query delegation capabilities. Not all operations can be pushed down to the external system, requiring hybrid query execution strategies.

Performance optimization involves understanding which OData operations are supported by your provider and designing user interfaces that leverage efficient query patterns. Consider caching strategies for frequently accessed but relatively static data.

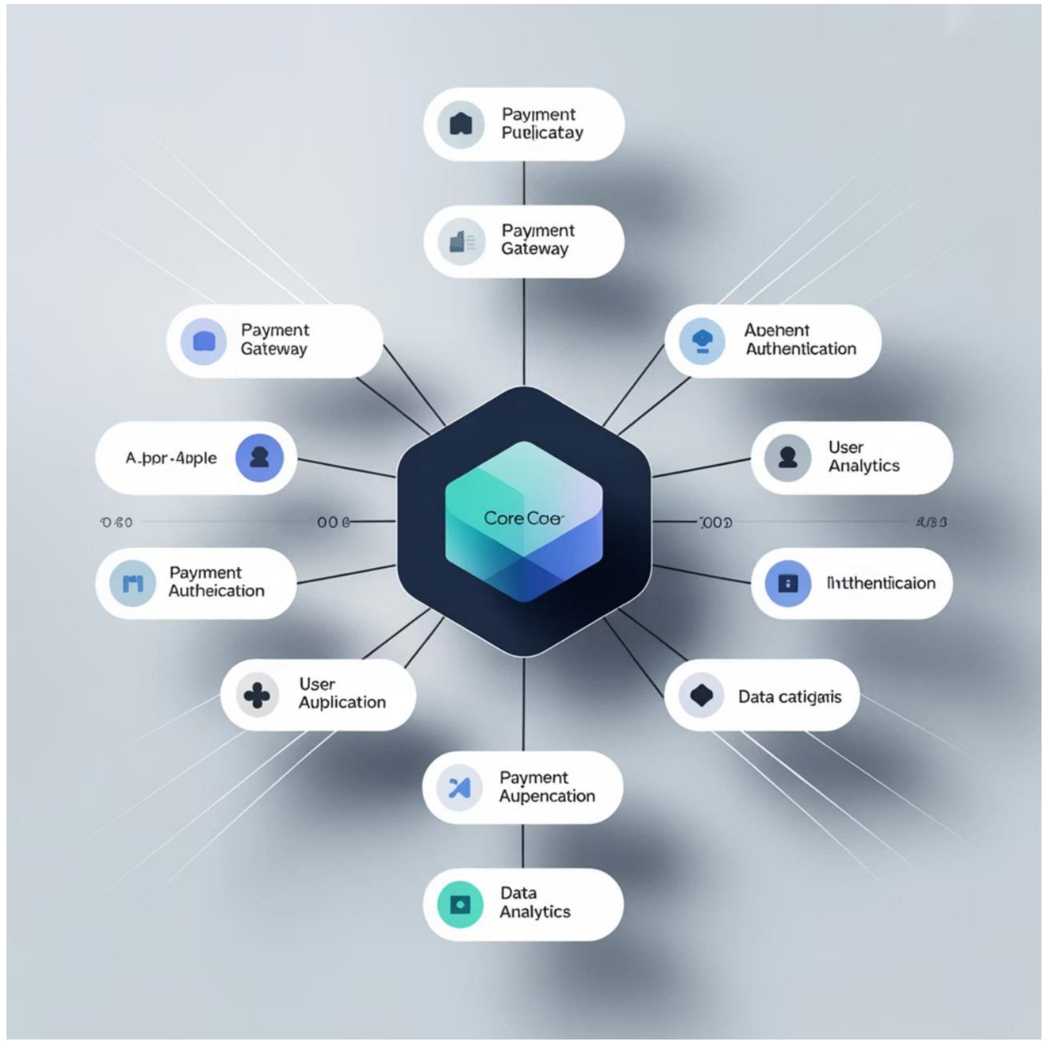
Integration Pattern Decision Matrix

Pattern	Real-time	Volume	Use Case
Dataflows	Batch	High	Historical data migration
Virtual Tables	Real-time	Medium	Live operational data
Power Automate	Near real-time	Low	Event-driven updates
Custom Connectors	Configurable	Variable	Complex business logic

Custom Development

Plugins and process actions for complex business logic

Plugin Development Architecture



Plugins provide server-side extensibility for Dataverse, executing custom business logic in response to data events. The plugin pipeline operates in multiple stages, allowing developers to implement validation, transformation, and integration logic at precise points in the data lifecycle.

Modern plugin development leverages .NET Core, dependency injection patterns, and comprehensive logging frameworks. Understanding execution context, security models, and performance implications is crucial for building reliable enterprise solutions.

Plugin Execution Pipeline

1

Pre-Validation (Stage 10)

Execute outside database transaction for initial validation and security checks

2

Pre-Operation (Stage 20)

Modify data before database operation within transaction context

3

Main Operation

Core Dataverse operation executes (Create, Update, Delete, etc.)

4

Post-Operation (Stage 40)

Execute after database operation for side effects and integrations

Plugin Development Best Practices

Stateless Design

Plugins should be stateless and thread-safe. Avoid static variables and implement proper dependency management for external service calls.

Error Handling

Implement comprehensive exception handling with meaningful error messages. Use InvalidPluginExecutionException for user-facing errors.

Performance Monitoring

Track execution time and implement timeout handling. Consider asynchronous patterns for long-running operations.

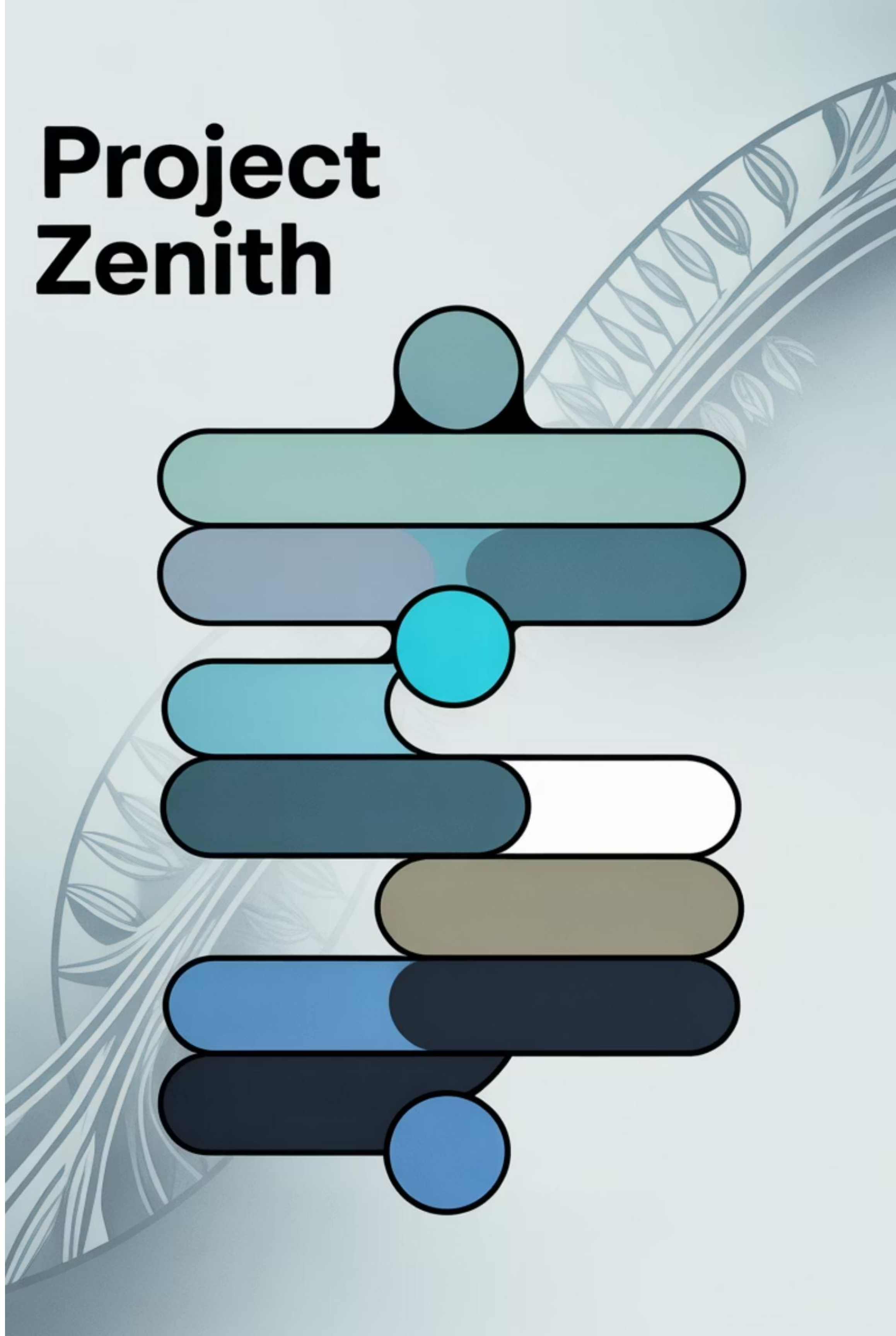
Testing Strategy

Develop comprehensive unit tests with mocked IOrganizationService. Implement integration tests for end-to-end validation.

Custom Process Actions

Custom Process Actions extend Dataverse capabilities by exposing custom business logic through declarative workflows and Power Automate flows. Unlike plugins, actions can be called directly from client applications and support both synchronous and asynchronous execution patterns.

Actions support input and output parameters, enabling reusable business logic components that can be consumed across multiple applications. They provide a higher-level abstraction than plugins while maintaining the ability to implement complex multi-step processes.



Action vs Plugin Decision Framework

Use Actions When:

- Building reusable business logic components
- Creating custom APIs for external consumption
- Implementing multi-step business processes
- Requiring declarative workflow integration
- Supporting both sync and async execution

Use Plugins When:

- Intercepting standard CRUD operations
- Implementing data validation rules
- Requiring transaction-level control
- Building high-performance operations
- Needing pipeline stage precision

Action Implementation Example

```
public class CalculateLoanApprovalAction : IPlugin{ public void Execute(IServiceProvider serviceProvider) { var context =
(IPluginExecutionContext)serviceProvider.GetService(typeof(IPluginExecutionContext)); var serviceFactory =
(IOrganizationServiceFactory)serviceProvider.GetService(typeof(IOrganizationServiceFactory)); var service =
serviceFactory.CreateOrganizationService(context.UserId);  // Extract input parameters var loanAmount =
(decimal)context.InputParameters["LoanAmount"]; var creditScore = (int)context.InputParameters["CreditScore"]; var annualIncome =
(decimal)context.InputParameters["AnnualIncome"];  // Business logic implementation var approvalResult = CalculateApproval(loanAmount,
creditScore, annualIncome);  // Set output parameters context.OutputParameters["IsApproved"] = approvalResult.IsApproved;
context.OutputParameters["ApprovedAmount"] = approvalResult.ApprovedAmount; context.OutputParameters["InterestRate"] =
approvalResult.InterestRate; }}
```

Performance Optimization

Indexing strategies and throttling management

Dataverse Indexing Fundamentals

Dataverse automatically creates and manages indexes based on usage patterns and query analysis. Understanding index types, creation strategies, and maintenance operations is essential for optimizing large-scale implementations with complex query requirements.

The platform supports clustered indexes on primary keys, non-clustered indexes on frequently queried columns, and composite indexes for multi-column query patterns. Custom index requests can be submitted for specific performance scenarios through Microsoft support channels.

Index Strategy Design

1

Query Analysis

Identify frequently executed query patterns and join operations

2

Index Planning

Design composite indexes for multi-column filters and sort operations

3

Performance Monitoring

Monitor query execution plans and index utilization statistics

4

Optimization

Request custom indexes for specific high-volume query scenarios

Query Optimization Techniques

Filter Optimization

- Place most selective filters first
- Use indexed columns in WHERE clauses
- Avoid functions on filtered columns
- Implement proper date range filtering

Join Strategies

- Minimize the number of joins
- Use appropriate join types
- Consider denormalization for read-heavy scenarios
- Implement join hint strategies

Column Selection

- Request only required columns
- Avoid SELECT * operations
- Use ColumnSet effectively
- Consider pagination for large result sets

Understanding Dataverse Throttling

Dataverse implements sophisticated throttling mechanisms to ensure platform stability and fair resource utilization across tenants. Throttling occurs at multiple levels including API request rates, concurrent connections, and resource consumption patterns.

The platform uses adaptive throttling algorithms that consider historical usage patterns, current system load, and tenant-specific quotas. Understanding these mechanisms is crucial for designing resilient applications that handle throttling gracefully.

Throttling Limits and Boundaries

6000	52560000	20	2
API Requests per Minute	Requests per Day	Concurrent Connections	Execution Time Minutes
Standard per-user limit for Dataverse API calls	Daily allocation for high-volume scenarios	Maximum simultaneous connections per user	Maximum duration for long-running operations

Throttling Mitigation Strategies

Exponential Backoff

Implement progressive retry delays with jitter to avoid thundering herd problems

Request Batching

Combine multiple operations into batch requests to reduce API call overhead

Caching Strategies

Implement client-side caching for frequently accessed, slowly changing data

Asynchronous Processing

Use message queues and background processing for non-time-critical operations

Resilient Application Design Patterns

```
// Retry logic with exponential backoffpublic async Task ExecuteWithRetryAsync(Func> operation, int maxRetries = 3){    int attempt = 0;    TimeSpan delay =
TimeSpan.FromMilliseconds(500);        while (attempt < maxRetries)    {        try        {            return await operation();        }        catch
(ServiceException ex) when (ex.StatusCode == 429)        {            if (attempt == maxRetries - 1) throw;                        await Task.Delay(delay);
delay = TimeSpan.FromMilliseconds(delay.TotalMilliseconds * 2);            attempt++;        }    }        throw new InvalidOperationException("Max retry attempts
exceeded");}
```

Implementing proper retry logic with exponential backoff ensures your applications handle throttling scenarios gracefully while respecting platform limits.

Module Summary and Next Steps

Audit Mastery

You now understand field-level auditing implementation and compliance strategy design for enterprise data governance requirements.

Integration Excellence

Master dataflows and virtual tables for seamless external system connectivity with optimal performance characteristics.

Custom Development

Build sophisticated plugins and process actions that extend Dataverse capabilities while maintaining enterprise reliability standards.

Performance Optimization

Design efficient indexing strategies and implement throttling-aware applications for scalable enterprise deployments.