DB-AICE: Simplified Data Orchestration Architecture

1. Executive Summary

This document presents a streamlined version of the DB-AICE architecture, focusing on the core data orchestration components: the Request Processor and Data Orchestrator. This simplified architecture provides a configuration-driven approach to data integration across multiple sources without requiring complex transformation logic in the initial phase.

Key Capabilities

- **Database Access**: Access database tables through configuration-driven endpoints
- **API Orchestration**: Intelligently aggregate data from multiple services and sources
- **Feature Store Integration**: Directly retrieve ML features from Feast feature store
- **Gateway Compliance**: Generate all required artifacts for API gateway integration

Business Value

- **Faster Time to Market**: Create new data endpoints with zero or minimal code
- **Simplified Integration**: Consolidate multiple data sources behind single endpoints
- **ML/Data Science Support**: Streamline access to features and model outputs
- **Enterprise Governance**: Automatic compliance with organizational API standards

2. Architectural Overview

2.1 High-Level Architecture

The streamlined architecture consists of these major components:

1. **Configuration Layer**: YAML-based configuration for endpoints and orchestration
2. **Orchestration Engine**: Core components for request processing and data orchestration
   - Request Processor
   - Data Orchestrator
3. **Data Access Layer**: Components for accessing various data sources
4. **API Gateway Integration**: Artifacts and processes for organizational compliance

2.2 Core Components Relationship

The system follows these architectural principles:

- **Separation of Concerns**: Distinct layers for configuration, orchestration, and data access
- **Declarative Configuration**: Behavior defined through configuration rather than code

- **Enterprise Integration**: Designed to integrate with organizational governance processes

3. Core Components

3.1 Configuration Engine

The Configuration Engine loads and interprets YAML-based configuration files that define:

- API endpoints and their behaviors
- Data source connections and parameters
- Integration with enterprise systems

Key aspects:

- Validates configuration for correctness and security
- Implements inheritance and composition for configuration reuse
- Supports environment-specific overrides and secrets management

3.2 Request Processor

The Request Processor handles incoming API requests and prepares them for orchestration:

Key Functions:

- Parses and validates incoming HTTP requests
- Extracts query parameters, path variables, and request body
- Validates inputs against configuration schemas
- Determines which endpoint configuration to use
- Authorizes the request based on security configuration
- Prepares the execution context for orchestration
- Handles response formatting and serialization
- Manages error responses and status codes

Example Scenario:
When a request arrives at `/customers/123?include=orders,features`, the Request Processor validates the customer ID, recognizes the `include` parameter for selective data loading, and creates an execution context that will inform the Data Orchestrator which data sources to query.

3.3 Data Orchestrator

The Data Orchestrator coordinates data retrieval from multiple sources and manages the execution flow:

Key Functions:

- Creates an execution plan based on dependencies between data sources
- Determines optimal query order and parallelization opportunities
- Manages asynchronous execution of data source queries
- Passes entity IDs and context between data source calls
- Handles timeouts and retries for external sources
- Implements fallback strategies for failed sources
- Merges results from different sources according to configuration
- Maintains request context throughout the orchestration process

Example Scenario:
For a customer profile endpoint, the Data Orchestrator might first query the customer database, then concurrently request recent orders from an order API and customer features from Feast, using the customer ID retrieved from the database as the entity key for both secondary requests.

3.4 Data Source Adapters

Data Source Adapters provide a unified interface for different data sources:

1. **Database Adapter**:
   - Connects to various SQL databases
   - Builds and executes queries based on configuration
   - Handles connection pooling and query optimization
2. **API Adapter**:
   - Manages connections to external REST APIs
   - Handles authentication, retries, and error strategies
   - Supports request templating and response mapping
3. **Feast Feature Store Adapter**:
   - Integrates with Feast for feature retrieval
   - Maps entity references between data sources
   - Handles online and offline feature serving

4. Data Flow

4.1 Request Processing Flow

1. Client submits request to API endpoint
2. API Gateway validates request and routes to DB-AICE service
3. Configuration Engine identifies endpoint configuration
4. Request Processor validates and prepares the request

5. Data Orchestrator creates execution plan
6. Data Source Adapters retrieve data from multiple sources in appropriate order
7. Data Orchestrator merges results from different sources
8. Response is formatted and returned to client

4.2 Execution Context

The execution context is a state container that holds all data, parameters, and metadata required to process a request throughout its lifecycle:

Components:

- Request information (parameters, headers, path variables)
- User authentication and authorization data
- Intermediate results from data sources
- Execution state and progress tracking
- Error information and handling flags
- Configuration references specific to the request

Example:

json Copy

```
{
  "requestInfo": {
    "endpoint": "customers/profile",
    "parameters": {"customerId": "12345", "include": "orders,features"},
    "authContext": {"userId": "operator-789", "roles": ["customer-service"]}
  },
  "primaryResults": {
    "customer": {"id": "12345", "name": "Jane Smith", "segment": "Premium"}
  },
  "intermediateResults": {
    "orders": [{"orderId": "ORD-001", "amount": 249.99}, {"orderId": "ORD-002", "amoun
    "features": {"churn_probability": 0.23, "ltv_prediction": 2450}
  },
  "executionState": {
    "completedSources": ["database", "orders_api"],
    "pendingSources": ["feast_features"],
    "errors": {}
  }
}
```

4.3 Execution Plan

The execution plan is a structured representation of all data retrieval steps required to fulfill a request, including their dependencies and execution order:

Components:

- Data source nodes (what to query)
- Dependencies between sources (sequencing requirements)
- Parallel execution groups (what can run concurrently)
- Parameter mappings (how values pass between steps)
- Fallback strategies (what to do if a step fails)

Example:

Copy

```
1. Query primary source:
   - Database: "SELECT id, name, segment FROM customers WHERE id = :customerId"
   - Required parameters: {customerId}
   - Output: customer_profile

2. Query secondary sources (in parallel):
   - Order API: "/customers/:customerId/orders"
     - Required parameters: {customerId} from step 1
     - Output: customer_orders
   - Feast: "feature_service='customer_features', entity={customer_id: :customerId}"
     - Required parameters: {customerId} from step 1
     - Output: customer_features
```

5. Configuration Model

5.1 Configuration Structure

The configuration is structured in multiple layers:

Copy

```
config/
├── config.yaml              # Global configuration
├── database.yaml            # Database connection settings
├── integrations/            # External integrations configuration
│   ├── api_sources.yaml     # External API sources
│   └── feast_config.yaml    # Feast feature store configuration
└── domains/                 # Domain-specific configurations
    ├── domain1.yaml         # Domain 1 endpoint definitions
    └── domain2.yaml         # Domain 2 endpoint definitions
```

5.2 Endpoint Configuration

Each endpoint is defined with these key sections:

- **Basic Information**: Name, description, HTTP methods
- **Database Configuration**: Table, columns, joins, filters
- **Orchestration**: Configuration for multi-source data integration
- **Parameters**: Definition of accepted query parameters
- **Response Format**: Structure of the API response

5.3 Orchestration Configuration

Orchestration is configured through a declarative model:

- **Sources**: Definition of primary and secondary data sources
- **Dependencies**: Relationships between data sources
- **Mappings**: Rules for mapping data between sources
- **Error Handling**: Strategies for handling source failures

5.4 Configuration Example (Conceptual)

yaml

Copy

```yaml
# Conceptual representation of configuration structure
name: "customers"
description: "Customer data endpoints with ML features"
endpoints:
  profile:
    methods: ["GET"]
    description: "Customer profile enriched with activity data and ML features"

    # Primary database source
    database:
      table: "customers"
      columns: ["id", "name", "email", "segment"]

    # Orchestration configuration
    orchestration:
      sources:
        # Additional sources beyond primary database
        - type: "api"
          name: "recent_activity"
          config:
            api: "activity_service"
            endpoint: "get_activity"
            parameters:
              customer_id: "{{primary.id}}"
            mapping:
              destination: "activity"

        - type: "feast"
          name: "customer_features"
          config:
            feature_service: "customer_features"
            entities:
              customer_id: "{{primary.id}}"
            mapping:
              destination: "ml_features"
```

6. Integration with Enterprise API Gateway

6.1 Gateway Integration

The system integrates with your organization's API gateway through:

1. **Repository Structure Compliance**:
   - `api-spec/` folder containing required artifacts

- OpenAPI specifications in required format
- Metadata files following organizational standards

2. **Governance Process Integration**:
   - Automated generation of required files
   - Validation against governance rules
   - Support for Pull Request based workflows

6.2 API Specification Generation

The system automatically generates OpenAPI specifications from endpoint configurations:

yaml

Copy

```yaml
# Example generated OpenAPI spec (partial)
openapi: "3.0.0"
info:
  title: "Customer API"
  version: "1.0.0"
paths:
  /customers/{customerId}:
    get:
      summary: "Customer profile enriched with activity data and ML features"
      parameters:
        - name: customerId
          in: path
          required: true
          schema:
            type: string
      responses:
        200:
          description: "Successful response"
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/CustomerProfile"
```

7. Deployment and Operations

7.1 Deployment Options

The system supports multiple deployment patterns:

1. **Containerized Microservice**:
   - Docker container deployment

- Kubernetes orchestration
- Service mesh integration

2. **Serverless Deployment**:
   - Function-as-a-Service (FaaS) deployment
   - API Gateway integration

7.2 Operational Management

Day-to-day operations are supported through:

1. **Monitoring Capabilities**:
   - Health checks
   - Performance metrics
   - Usage statistics
2. **Troubleshooting Tools**:
   - Logging framework
   - Request tracing
   - Diagnostic endpoints

8. Use Cases

8.1 Customer 360 View

Scenario: Create an API that provides a comprehensive view of customer data, enriched with ML features and recent activity.

Configuration Pattern:

- Primary source: Customer database
- Secondary sources:
  - Order history API
  - Interaction logs API
  - Feast feature store for ML features

8.2 ML-Enhanced Product Information

Scenario: Deliver product information enhanced with real-time inventory and recommendation features.

Configuration Pattern:

- Primary source: Product catalog database

- Secondary sources:
  - Inventory service API
  - Pricing service API
  - Feast feature store for product embeddings

9. Future Enhancements

While this initial architecture focuses on data orchestration, future enhancements could include:

1. **Transformation Engine**: Add capabilities for applying business logic and transformations
2. **Caching Layer**: Implement intelligent caching for improved performance
3. **Schema Validation**: Add runtime schema validation for data sources
4. **Advanced Monitoring**: Implement detailed performance and usage analytics

10. Conclusion

This simplified DB-AICE architecture provides a powerful, configuration-driven approach to building APIs that orchestrate data from multiple sources. By focusing on the Request Processor and Data Orchestrator components, it offers immediate value while laying the groundwork for future enhancements.

Key advantages include:

1. **Accelerated Development**: Create new data endpoints in hours instead of days
2. **Simplified Integration**: One API instead of many for clients to integrate
3. **Governance Compliance**: Built-in support for organizational standards
4. **ML Enablement**: Seamless incorporation of ML features

This architecture is particularly valuable for data science and ML teams who need to expose their data and features through standardized, governed APIs while maintaining flexibility and rapid iteration.