Back to Document Index

Data Mapping & Schema Documentation

Business Analysis Template for AI Projects

Template ID:	2.4
Category:	Data Analysis & Insights
Version:	1.0
Last Updated:	January 2026
Prerequisites:	Data Requirements Specification (1.5) Data Quality Assessment Report (2.1) Exploratory Data Analysis Report (2.2)

1. Document Purpose

This Data Mapping & Schema Documentation template provides a comprehensive framework for documenting data structures, relationships, and transformations in AI projects. Proper data mapping ensures that all stakeholders understand how data flows through systems, how it is transformed, and how different data elements relate to each other.

- Key purposes of this document include:
- Document source-to-target data mappings for all transformations
- Define data schemas for all databases, files, and APIs in the AI solution
- Establish clear data lineage from origin to consumption
- Specify data type conversions and transformation rules
- Identify relationships between data entities and attributes
- Create reference documentation for developers and analysts
- Support impact analysis when data structures change
- Enable troubleshooting of data integration issues
- Facilitate data governance and compliance activities
- Provide foundation for automated testing and validation

Back to Document Index

2. When to Use This Template

This template should be used during the design phase after data requirements are defined and before implementation begins. It bridges the gap between business requirements and technical implementation by documenting exactly how data will be structured and transformed.

Ideal Use Cases:

Data Integration Projects: When integrating data from multiple source systems into a unified data warehouse, data lake, or AI platform.

API Design and Development: When designing APIs that will serve data to AI models or consume data from external systems.

Database Design: When creating new databases or modifying existing database schemas to support AI applications.

ETL/ELT Pipeline Development: When building data pipelines that extract, transform, and load data for AI model training or inference.

Data Migration: When moving data from legacy systems to new platforms, requiring clear mapping of old structures to new structures.

Model Feature Engineering: When documenting how raw data fields are transformed into model features and what those features represent.

Data Quality Implementation: When specifying validation rules, business rules, and data quality checks that will be applied to data.

Regulatory Compliance: When demonstrating data lineage and transformations for audit, privacy, or regulatory requirements.

System Documentation: When creating technical documentation for existing systems that lack proper data documentation.

Onboarding New Team Members: When helping new developers or analysts understand the data structures and flows in complex AI systems.

3. How to Use This Template

This template guides you through systematic documentation of data structures and mappings. Follow these steps to create comprehensive data documentation for your AI initiative.

Step 1: Identify All Data Sources and Targets

Begin by cataloging every data source and target in your AI solution. Sources include databases, files, APIs, streaming data, and external services. Targets include data warehouses, data lakes, feature stores, model training datasets, and APIs. For each source and target, document the

Back to Document Index

system name, connection details, data format, update frequency, and business owner. Create a high-level data flow diagram showing how data moves between sources and targets.

Step 2: Document Source Data Schemas

For each source system, document the complete data schema. This includes table names, column names, data types, constraints, primary keys, and foreign keys. For files, document the file format, delimiter, header rows, and field structure. For APIs, document endpoints, request/response formats, and data structures. Include sample data to illustrate what actual values look like. Note any quirks or special handling requirements.

Step 3: Document Target Data Schemas

Document the schema for each target system using the same level of detail as source schemas. Specify how the target schema is optimized for its use case. For data warehouses, document dimensional models with facts and dimensions. For feature stores, document feature definitions and metadata. For APIs, document the response structure that will be returned to consumers.

Step 4: Create Field-Level Mappings

Map each source field to its corresponding target field. Specify the exact transformation applied to convert source values to target values. Document data type conversions, business rule applications, calculations, aggregations, and any logic applied. If multiple source fields combine into one target field, document the combination logic. If one source field splits into multiple target fields, document the splitting rules.

Step 5: Define Transformation Rules

Document all transformation rules that will be applied during data movement. This includes data cleansing rules, standardization rules, derivation formulas, lookup tables, and conditional logic. Specify what happens with null values, how missing data is handled, and default values. Include validation rules that check data quality. Provide examples showing input values and expected output values after transformation.

Step 6: Document Data Relationships

Create entity relationship diagrams showing how data entities relate to each other. Document primary key and foreign key relationships. Specify cardinality (one-to-one, one-to-many, many-to-many). Identify referential integrity constraints. Document any hierarchies or parent-child relationships in the data. This provides context for understanding how data entities interact.

Step 7: Establish Data Lineage

Trace each data element from its origin through all transformations to its final destination. Create data lineage diagrams showing the complete flow. Document every system, process, and transformation point in the journey. This is critical for impact analysis, troubleshooting, and compliance. Include information about data retention at each stage and archival policies.

Back to Document Index

Step 8: Specify Data Quality Rules

Define data quality checks that will be applied at each stage of data movement. Specify validation rules for required fields, acceptable value ranges, valid code values, format requirements, and business rule compliance. Document what happens when data fails quality checks. Define error handling procedures and notification processes. This ensures data meets quality standards before being used by AI models.

Step 9: Document Metadata and Business Definitions

For each data element, provide business metadata including plain-language definition, business purpose, business owner, sensitivity classification, and usage guidelines. Document any business rules or constraints that apply. Include examples of valid values and their meaning. This bridges technical and business perspectives, ensuring all stakeholders understand what data represents.

Step 10: Create Implementation Specifications

Translate the data mapping documentation into specifications that developers can implement. Provide SQL scripts for database transformations. Document API specifications with request/response examples. Create ETL logic specifications with pseudocode or flowcharts. Include performance considerations such as indexing strategies, partitioning approaches, and optimization techniques. Specify error handling and logging requirements.

4. Instructions for Each Section

The following sections provide detailed guidance for completing each part of the data mapping and schema documentation. Use these instructions to ensure comprehensive coverage of all data structures and transformations.

4.1 Data Source Inventory

Create a comprehensive inventory of all data sources that feed your AI solution. For each source, document the system name, description, connection information, data format, update frequency, and business owner. Include source systems such as transactional databases, data warehouses, SaaS applications, file systems, streaming platforms, and external APIs. Specify authentication requirements and access restrictions. Note any service level agreements or uptime guarantees. Document the business criticality of each source. Identify any dependencies between sources. This inventory provides the foundation for understanding your data landscape.

Back to Document Index

4.2 Data Target Inventory

Document all target systems where data will be stored or delivered. Targets include feature stores, training datasets, inference databases, data warehouses, data lakes, APIs, and reporting systems. For each target, specify its purpose in the AI solution. Document performance requirements such as query response time or throughput. Specify storage capacity and retention policies. Identify who will consume data from each target. Document backup and disaster recovery procedures. Understanding your targets ensures data is structured appropriately for its intended use.

4.3 Source Schema Documentation

Document the complete schema for each source system. For relational databases, create data dictionaries listing all tables and columns. Specify data types, lengths, precision, and scale. Document constraints including primary keys, foreign keys, unique constraints, and check constraints. Include indexes and their purpose. For files, document field positions, delimiters, text qualifiers, and header information. For JSON APIs, document the object structure with nested properties. For XML, provide the schema definition. Include data samples showing actual values. Document any data quality issues or anomalies observed in source systems. Note the volume of data in each source. This detailed documentation enables accurate mapping and transformation design.

4.4 Target Schema Documentation

Document target schemas with the same rigor as source schemas. Explain how the schema design supports the target system purpose. For dimensional data warehouses, document fact tables with measures and dimension tables with attributes. Specify slowly changing dimension (SCD) handling strategies. For feature stores, document feature definitions, data types, and update logic. For normalized databases, explain the normalization level and reasons. Document denormalization decisions and their justification. Specify partitioning strategies for large tables. Include indexing strategies optimized for query patterns. Document any special data types or structures used. Provide examples of how data will look in the target.

4.5 Field-Level Mapping Specifications

Create detailed field-level mappings connecting each source field to target fields. Use a standardized format showing source system, source table, source field, target system, target table, and target field. Specify the transformation applied. For direct mappings with no

Back to Document Index

transformation, indicate this clearly. For calculated fields, provide the formula. For fields requiring lookups, specify the lookup table and logic. For conditional mappings, document all conditions and outcomes. Include data type conversions with any truncation or rounding rules. Document how null values are handled. Specify default values for missing data. For fields concatenated from multiple sources, show the concatenation logic. For fields split into multiple targets, explain the splitting rules. Provide examples with sample input and output values for complex transformations.

4.6 Transformation Rule Documentation

Document all business rules and transformation logic applied to data. Start with data cleansing rules such as trimming whitespace, removing special characters, and standardizing formats. Document standardization rules for addresses, phone numbers, names, and codes. Specify how dates and times are converted between formats and time zones. Document any unit conversions such as currency, measurements, or temperatures. Provide lookup tables for code translations. Document derivation logic for calculated fields with mathematical formulas or business rules. Specify aggregation logic including grouping dimensions and aggregate functions. Document conditional logic with all branches and outcomes. For complex rules, provide decision trees or flowcharts. Include examples showing before and after values. Document any special case handling or exceptions to rules.

4.7 Entity Relationship Documentation

Create entity relationship diagrams showing how data entities relate to each other. Use standard notation such as crow's foot or UML. Identify all entities with their primary keys. Show relationships with cardinality (one-to-one, one-to-many, many-to-many). Document foreign key relationships and referential integrity constraints. Specify whether relationships are identifying or non-identifying. Document any recursive relationships. Show inheritance hierarchies if applicable. Include associative entities for many-to-many relationships. Document the business meaning of each relationship. Specify whether relationships are mandatory or optional. This provides essential context for understanding data structure and dependencies.

4.8 Data Lineage Documentation

Trace each data element through its complete lifecycle. Start with the original data source. Document every transformation point as data moves through systems. Show temporary staging areas or intermediate storage. Document data quality checkpoints. Show where data is

Back to Document Index

aggregated, filtered, or enriched. Trace to final destinations including reports, models, and APIs. Create visual lineage diagrams showing the flow. Include process names responsible for each transformation. Document timing and frequency of data movement. Specify data retention at each stage. Include information about data archival and purging. This lineage is critical for impact analysis when changes occur, troubleshooting data issues, and demonstrating compliance with data regulations.

5. Best Practices for Data Mapping and Schema Documentation

Start with Business Requirements

Always begin data mapping work by understanding business requirements, not just technical specifications. Meet with business stakeholders to understand what data means in business terms. Ask what decisions will be made with the data. Understand business rules that govern data values. Document data quality expectations from a business perspective. This business context guides technical mapping decisions. For example, knowing that customer revenue drives executive compensation affects how carefully that field must be calculated. Starting with business context prevents creating technically correct mappings that miss business intent. Involve business users in reviewing mappings to validate they match business understanding. Document business definitions alongside technical specifications.

Use Standard Naming Conventions

Establish and follow consistent naming conventions for all data elements. Use descriptive names that clearly indicate what data represents. Avoid abbreviations unless they are well-known industry standards. Use consistent case formatting such as snake_case or camelCase throughout. Specify prefixes or suffixes that indicate data type or purpose. For example, use "dt_" for date fields or "_id" for identifiers. Document your naming standards in a style guide. Apply these standards consistently across all systems. Consistent naming makes schemas self-documenting and reduces misunderstandings. It speeds development when developers can predict field names. It simplifies searching and filtering in large schemas.

Document the "Why" Not Just the "What"

Go beyond documenting what transformations are applied to explain why they are necessary. Document the business reason for each mapping decision. Explain why certain fields are excluded or included. Describe why specific transformation logic is used. Capture the history of decisions when requirements changed. This context is invaluable when someone needs to modify mappings later. Without the why, future developers may be afraid to change mappings that seem arbitrary. Documenting rationale prevents repeating past mistakes. It helps new team members understand the system faster. Include references to requirements documents, business rule specifications, or stakeholder discussions that drove decisions.

Back to Document Index

Maintain Version Control

Treat data mapping documentation as code that requires version control. Store documentation in a version control system like Git. Track changes over time with meaningful commit messages. Use branching for proposed changes before they are implemented. Tag releases that correspond to production deployments. This provides history of how schemas evolved. It enables rolling back to previous versions if needed. It shows who made changes and when. Version control enables collaboration where multiple team members can work on mappings simultaneously. It provides an audit trail for compliance purposes. For database schema changes, use migration scripts that are also version controlled.

Include Comprehensive Examples

Provide concrete examples for all mappings and transformations. Show sample source data values. Show the expected target values after transformation. Include edge cases such as null values, empty strings, maximum lengths, and special characters. Demonstrate how conditional logic branches work with examples of each condition. For complex calculations, show the formula worked out with actual numbers. Examples make abstract specifications concrete. They enable reviewers to validate logic without reading code. They serve as test cases for implementation. Developers can use examples to verify their code produces expected results. Analysts can understand data without deciphering transformation code.

Validate Mappings with Stakeholders

Do not finalize mappings without stakeholder review and approval. Walk through mappings with business users who understand source data. Review with data analysts who will consume target data. Present to data governance teams to ensure compliance. Use examples to make reviews concrete and accessible to non-technical stakeholders. Ask stakeholders to verify that transformed data will meet their needs. Identify any missing fields or incorrect transformations early. Document stakeholder approval with names and dates. This validation prevents discovering issues after implementation when changes are expensive. It builds stakeholder buy-in and shared ownership of the solution.

Plan for Schema Evolution

Design schemas and mappings with the expectation that requirements will change. Use extensible designs that can accommodate new fields without breaking existing functionality. Avoid hard-coding values that may change. Use configuration tables for lookup values instead of embedding them in code. Design APIs with versioning from the start. Document backward compatibility requirements. Consider how new fields will be added to existing tables. Plan for data type changes such as expanding field lengths. Document the change management process for schema modifications. Include impact analysis procedures. This forward-thinking approach reduces disruption when inevitable changes occur.

Standardize Data Types

Establish standard data types for common data categories and use them consistently. Define standard types for dates, timestamps, currencies, percentages, identifiers, and codes. Specify

Back to Document Index

precision and scale for numeric types. Document timezone handling for timestamps. Specify character encoding for text fields. Standardization simplifies integration when multiple systems use the same types. It reduces conversion errors. It makes data more portable between systems. Create a data type standards document that all developers reference. Include rationale for type selections such as why certain precision is required.

Document Data Quality Rules

Explicitly document all data quality rules that will be enforced. Specify required fields that cannot be null. Define acceptable value ranges for numeric fields. List valid code values for categorical fields. Specify format requirements for fields like phone numbers or email addresses. Document business rule validations such as date logical consistency. Include cross-field validations where multiple fields must be consistent. Specify what happens when data fails validation. These rules become the basis for data quality monitoring and testing. They make quality expectations explicit rather than assumed.

Use Visual Diagrams

Supplement textual documentation with visual diagrams. Create entity relationship diagrams showing data structures. Draw data flow diagrams showing movement between systems. Build data lineage diagrams tracing data from source to target. Use UML diagrams for complex object structures. Visuals communicate structure and relationships faster than text. They provide an overview that helps people navigate detailed documentation. They are more accessible to non-technical stakeholders. Use standard notation that readers will recognize. Ensure diagrams are kept up to date as documentation changes. Reference diagrams in textual documentation to connect visual and detailed views.

Automate Where Possible

Leverage tools to automate documentation creation and maintenance where possible. Use schema discovery tools to automatically extract schema information from databases. Generate data dictionaries from system metadata. Use data profiling tools to automatically document data distributions and patterns. Consider data catalog tools that maintain living documentation. However, automation cannot capture business context, transformation rationale, or quality rules. Use automation for mechanical documentation tasks. Add human insight for business meaning and decision rationale. Keep automated and manual documentation synchronized.

Test Mappings Before Implementation

Do not wait until coding is complete to test if mappings are correct. Create test cases based on mapping specifications. Use sample data to verify transformations produce expected results. Test edge cases and error conditions. Validate that null handling works as specified. Check data type conversions. Verify lookup logic. Testing mappings early catches specification errors when they are easy to fix. It validates that specifications are clear and implementable. It provides test cases that developers can use during implementation. Document test results with the mappings as additional validation of correctness.

Back to Document Index

6. Common Pitfalls to Avoid

Incomplete or Outdated Documentation

The most common pitfall is creating documentation that quickly becomes outdated. Mappings are documented during design but not updated when implementation reveals issues. Schema changes are made without updating documentation. New fields are added without documenting their purpose or transformations. This happens because documentation is treated as a one-time deliverable rather than a living artifact. Teams move quickly to implementation without maintaining documentation discipline. Outdated documentation is worse than no documentation because it misleads. Establish a process where schema changes must be documented before deployment. Include documentation updates in code review checklists. Schedule regular documentation audits. Assign ownership for keeping documentation current.

Ignoring Data Quality in Source Systems

Mappings are designed assuming source data is clean and complete. Reality is discovered only after implementation when data quality issues cause failures. Source systems may have missing values, invalid codes, inconsistent formats, or violated business rules. This happens because teams do not profile source data before designing mappings. They rely on schema definitions without examining actual values. Data quality issues then require extensive transformation logic or data cleansing that was not planned. Always profile source data before creating mappings. Document data quality issues discovered. Design transformation rules to handle known problems. Include data validation checkpoints in mappings. Plan for data rejection and error handling.

Missing Business Context

Documentation focuses purely on technical specifications without capturing business meaning. Field names and data types are documented but not what data represents or how it is used. Business rules are implemented in code but not explained in documentation. This makes documentation useless to business users. It makes troubleshooting difficult because technical staff do not understand business impact. This happens when data mappings are treated as purely technical exercises. Technical staff document technical details they understand. Business context is assumed or never captured. Always include business definitions for every field. Document business rules in plain language. Explain business purpose and usage. Make documentation accessible to both technical and business audiences.

Overlooking Performance Implications

Mappings are designed for correctness without considering performance. Complex transformations are specified without regard to computational cost. Joins across large tables are designed without indexing strategy. This leads to implementations that meet functional requirements but fail performance requirements. This happens because mapping design focuses on what transformations are needed without considering how they will be executed. Performance is addressed only after implementation when problems emerge. Always consider

Back to Document Index

performance during mapping design. Document expected data volumes. Specify indexing requirements. Identify transformations that should be pre-computed. Consider partitioning strategies for large tables. Consult with database administrators during design.

Inconsistent Transformation Logic

The same business concept is transformed differently in different places. Customer name standardization follows one rule in one pipeline and a different rule in another pipeline. This creates inconsistency where the same source data produces different values in different targets. This happens when transformations are designed in isolation without coordination. Different developers implement similar logic differently. There is no central repository of standard transformations. Establish standard transformation patterns that are reused. Document canonical transformation logic for common business concepts. Create shared transformation libraries. Review mappings across pipelines for consistency. Consolidate redundant transformation logic.

Inadequate Error Handling

Mappings specify happy path transformations without addressing error scenarios. There is no documentation of what happens when source data is invalid, transformations fail, or target systems are unavailable. This leaves implementation teams to make up error handling approaches inconsistently. This happens because error scenarios are uncomfortable to think about during design. Teams focus on making things work correctly and address errors as an afterthought. Explicitly document error handling for every mapping. Specify validation rules and what happens when they fail. Define data rejection criteria. Document error logging and notification requirements. Specify retry logic for transient failures. Test error handling as thoroughly as happy paths.

Lack of Traceability

Mappings cannot be traced back to the requirements that drove them. When someone asks why a field is calculated a certain way, there is no documentation explaining the business requirement. This makes it difficult to validate correctness or evaluate proposed changes. This happens when mappings are documented separately from requirements without linking them. Requirements are captured in one tool while technical specifications live in another. Always link mappings to requirements that drove them. Reference requirement IDs in mapping documentation. Cross-reference mapping documents in requirements specifications. This traceability enables impact analysis when requirements change.

Ignoring Data Security and Privacy

Mappings are designed without considering data sensitivity, privacy requirements, or security controls. Personal information is mapped without encryption or masking. Sensitive data is logged without protection. This violates privacy regulations and creates security vulnerabilities. This happens because security is often viewed as an infrastructure concern separate from data design. Mapping designers focus on functional requirements without considering data protection. Always classify data sensitivity in schema documentation. Identify fields containing

Back to Document Index

personal information. Document encryption requirements. Specify masking rules for sensitive fields. Define access controls. Ensure mappings comply with privacy regulations like GDPR or CCPA.

Over-Engineering Schemas

Schemas are designed with excessive complexity, attempting to anticipate every possible future need. Tables have dozens of optional fields "just in case." Multiple layers of abstraction are created for theoretical flexibility. This complexity makes schemas difficult to understand, maintain, and use. This happens when designers try to future-proof excessively. They add features that might be needed someday. Abstract designs are created before concrete use cases are understood. Follow the principle of starting simple and adding complexity only when actually needed. Design for known requirements first. Add extensibility for likely changes but do not anticipate every possibility. Refactor as new requirements emerge rather than building complexity upfront.

Poor Handling of Slowly Changing Dimensions

Dimension tables are designed without considering how attribute changes will be handled. When a customer address changes, should history be preserved or overwritten? This is not documented. This causes problems when historical analysis is needed or when requirements change. This happens because designers focus on current state without considering temporal aspects. Always specify slowly changing dimension (SCD) handling strategy. Document whether changes overwrite (Type 1), create new rows (Type 2), or add columns (Type 3). Specify effective dates and current flags. Consider the business need for historical analysis before choosing a strategy.

Insufficient Testing Specifications

Mappings are documented but test cases are not specified. Implementation teams must guess what constitutes successful transformation. Testing focuses on whether code runs without errors rather than whether results are correct. This happens because testing is viewed as separate from design. Mapping documentation is considered complete without test specifications. Always include test cases with mappings. Provide sample input and expected output for each transformation. Document edge cases that must be tested. Specify data quality validations that must pass. Make testing criteria explicit and measurable.

Not Documenting Assumptions

Mappings are based on assumptions about data characteristics, business rules, or system behavior. These assumptions are not documented. When assumptions prove incorrect, it is unclear why mappings were designed that way. This makes fixing issues difficult. This happens because assumptions seem obvious at design time and are not written down. Assumptions are implicit knowledge in people's heads. Always document all assumptions explicitly. Note assumptions about data quality, completeness, formats, and business rules. Document assumptions about system behavior and timing. Validate assumptions during implementation. Update documentation when assumptions change.

Back to Document Index

Neglecting Data Lineage

Individual mappings are documented but the complete data lineage from source to target is not traced. It is unclear which source systems ultimately feed which target fields. This makes impact analysis difficult. Changes to source systems cannot be evaluated for downstream impact. This happens because each integration point is documented in isolation. End-to-end data flow is not mapped. Create data lineage documentation showing complete paths. Use tools that automatically trace lineage where possible. Keep lineage documentation updated as systems evolve. Use lineage for impact analysis before making changes.

7. Appendices

Appendix A: Sample Data Mapping Template

Use this template format for documenting field-level mappings:

Source System: [System name]

Source Table/File: [Table or file name]

Source Field: [Field name]

Source Data Type: [Data type with length/precision]

Source Sample Values: [Examples of actual values]

Target System: [System name]

Target Table/File: [Table or file name]

Target Field: [Field name]

Target Data Type: [Data type with length/precision]

Transformation Logic: [Detailed description of transformation]

Business Rule: [Business rule being implemented]

Null Handling: [How null values are handled]

Default Value: [Default if source is null]

Validation Rules: [Data quality checks applied]

Back to Document Index

Example: Input "[example source value]" transforms to "[example target value]"

Appendix B: Common Data Types Reference

Standard data types for AI projects with recommended usage:

Text Fields: VARCHAR for variable-length text with specified maximum, CHAR for fixed-length codes, TEXT for unlimited text. Use VARCHAR(255) for short strings, VARCHAR(4000) for longer content. Always specify character encoding (UTF-8 recommended).

Numeric Fields: INTEGER for whole numbers without decimals, BIGINT for large integers, DECIMAL(p,s) for precise decimal numbers where p is precision and s is scale. Use DECIMAL for currency (DECIMAL(19,4) recommended). FLOAT and DOUBLE for approximate numeric values in scientific calculations.

Date and Time: DATE for calendar dates without time, TIMESTAMP for date and time with timezone, TIME for time of day. Always store in UTC and convert to local time in presentation layer. Use TIMESTAMP(6) for microsecond precision.

Boolean: BOOLEAN for true/false values. Avoid using 0/1 or Y/N in place of proper boolean types. Use NULL for unknown rather than false.

Identifiers: Use INTEGER or BIGINT for surrogate keys. Use VARCHAR for natural keys. Consider UUID for distributed systems requiring globally unique identifiers.

Appendix C: Data Quality Rules Checklist

Common data quality rules to document for each field:

Completeness: Is field required or optional? What percentage must be populated?

Validity: What values are acceptable? Is there a list of valid codes?

Format: What format is required (date format, phone format, email format)?

Range: For numeric fields, what are minimum and maximum acceptable values?

Back to Document Index

Precision: How many decimal places are required for numeric fields?

Length: What are minimum and maximum lengths for text fields?

Uniqueness: Must values be unique? Are duplicates allowed?

Consistency: How does field relate to other fields? What cross-field rules apply?

Timeliness: How current must data be? What is maximum acceptable age?

Accuracy: How is accuracy verified? What is acceptable error rate?

Appendix D: Schema Change Process

Standard process for managing schema changes:

Step 1 - Change Request: Document proposed change with business justification, impacted systems, and estimated effort.

Step 2 - Impact Analysis: Identify all systems, reports, and processes affected by change. Assess risk and complexity.

Step 3 - Design Review: Review proposed schema change with architects and technical leads. Validate against standards.

Step 4 - Documentation Update: Update all schema documentation, mapping specifications, and data dictionaries before implementation.

Step 5 - Testing: Create test cases covering new schema elements. Test backward compatibility if required.

Step 6 - Deployment: Deploy schema changes with rollback plan. Monitor for issues after deployment.

Back to Document Index

Step 7 - Verification: Verify schema change is complete and working as expected. Confirm documentation is updated.

Appendix E: Professional Standards References

This template aligns with professional standards for data management and business analysis:

BABOK Guide (7 key areas):

- Data Modeling
- Data Dictionary
- Data Flow Diagrams
- Requirements Analysis and Design Definition
- Data Analysis
- Interface Analysis
- Solution Evaluation

DMBOK (10 key areas):

- Data Architecture
- Data Modeling and Design
- Data Integration and Interoperability
- Metadata Management
- Data Quality
- Reference and Master Data
- Data Lineage
- Data Governance
- Data Mapping
- Schema Management

PMBOK Guide (5 key areas):

- Requirements Documentation
- Define Scope
- Create WBS
- Plan Quality Management
- Verify Scope

Back to Document Index

Document Usage Rights and Disclaimer

This Business Analysis Template for AI Projects is provided as a starter document to assist business analysts in documenting data mapping and schema specifications.

Usage Rights:

✓ You may freely use, modify, and customize this template for your AI projects

✓ You may adapt the mapping framework and methods to fit your needs

✓ You may incorporate this into your data management processes

✓ You may share this template within your organization

Restrictions:

✗ You may not resell, redistribute, or commercialize this template

✗ You may not claim original authorship of this framework

✗ You may not remove these usage rights statements

Disclaimer:

This template is provided as-is without warranties. While it incorporates professional best practices for data mapping and schema documentation, users are responsible for conducting thorough analysis appropriate to their specific context, regulatory requirements, and organizational standards. The template should be customized based on your unique needs and validated with appropriate subject matter experts.