

Model

Designing Your SmartSuite Data Model



Executive Summary

Designing a strong SmartSuite data model is one of the most important determinants of long-term solution success. The data model is the structural backbone that turns strategy and business requirements into a working system. When modeling decisions are rushed or poorly defined, the impact is far-reaching, resulting in brittle workflows, unreliable reporting, fragile automation, and solutions that are difficult to extend or trust once deployed. Because data structures are foundational, mistakes made during this phase are among the most costly to correct later.

When approached with discipline, the Model phase delivers significant benefits across the entire solution lifecycle. A well-designed model enables predictable workflows, consistent data capture, meaningful reporting, and scalable automation. It simplifies permission design, supports integration requirements, and reduces the need for complex workarounds as the solution evolves. Most importantly, strong modeling creates confidence, for users entering data, for administrators managing the solution, and for leaders relying on the outputs.



This white paper focuses on the Model phase as the step where approved plans and requirements are translated into a clean, durable SmartSuite data structure. It provides practical guidance on defining record types, lifecycle stages, fields, and relationships, and explains how those decisions directly affect usability, reporting, automation, permissions, and integrations. The paper also highlights best-practice modeling patterns, common pitfalls observed in real implementations, validation techniques to support design sign-off, and a worked example using an Issue & Action Tracking solution.

This document is intended for services consultants, implementation partners, and client administrators responsible for designing and maintaining SmartSuite solutions. For this audience, the Model phase matters because it determines how easily a solution can be built, supported, and scaled over time. This paper assumes that business objectives, process scope, and success criteria have already been defined; readers should refer to the SmartSuite Plan white paper for guidance on establishing the inputs required to make effective modeling decisions.

Why the Model Phase Matters

The Model phase defines the structural foundation of a SmartSuite solution. During this phase, teams design the tables, fields, relationships, and record lifecycles that determine how information is captured, connected, and used across the system. While workflows, dashboards, and permissions are often the most visible components of a solution, their effectiveness is entirely dependent on the quality and consistency of the underlying data model.

When modeling best practices are neglected, the consequences tend to compound over time. Unclear record boundaries, inconsistent field definitions, or fragile relationships lead to confusing user experiences, brittle automation, unreliable reporting, and performance issues as data volume grows. These issues are especially costly because data model changes are difficult to implement once solutions are live and widely adopted, often requiring disruptive rework.

The importance of the Model phase lies in its ability to enable everything that follows. A well-designed data model supports predictable workflows, meaningful reporting, clean permission structures, and scalable automation. By prioritizing clarity and consistency in how data is structured, teams reduce complexity across design, build, and support activities while increasing confidence in the system's outputs.

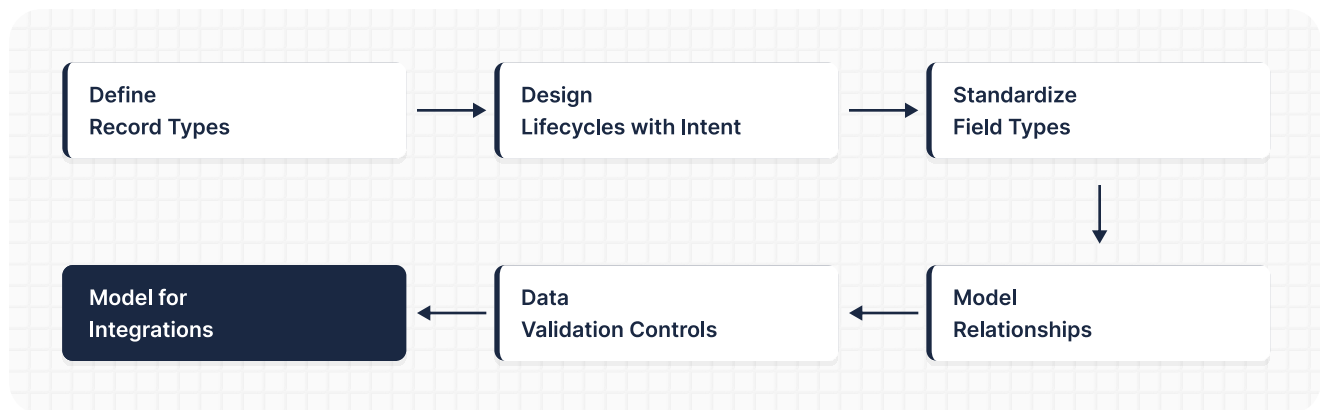
In addition, SmartSuite solutions are living systems that evolve as business needs change. Modeling decisions that emphasize extensibility and reuse rather than short-term convenience allow solutions to adapt without constant restructuring. Investing in the Model phase ensures the solution remains durable, maintainable, and trustworthy over time, providing a stable foundation for ongoing growth and improvement.



Model Phase Best Practices

The goal of the Model phase is not simply to make data fit into tables, it is to create a structure that reliably supports how work happens today while remaining flexible enough to evolve. In SmartSuite, modeling decisions directly influence automation reliability, permission design, reporting accuracy, and cross-solution reuse. The following best-practice principles reflect patterns observed in successful, scalable SmartSuite implementations.

Throughout this section, SmartSuite-specific callouts and decision checkpoints are included to help teams validate modeling choices before moving into build and configuration.



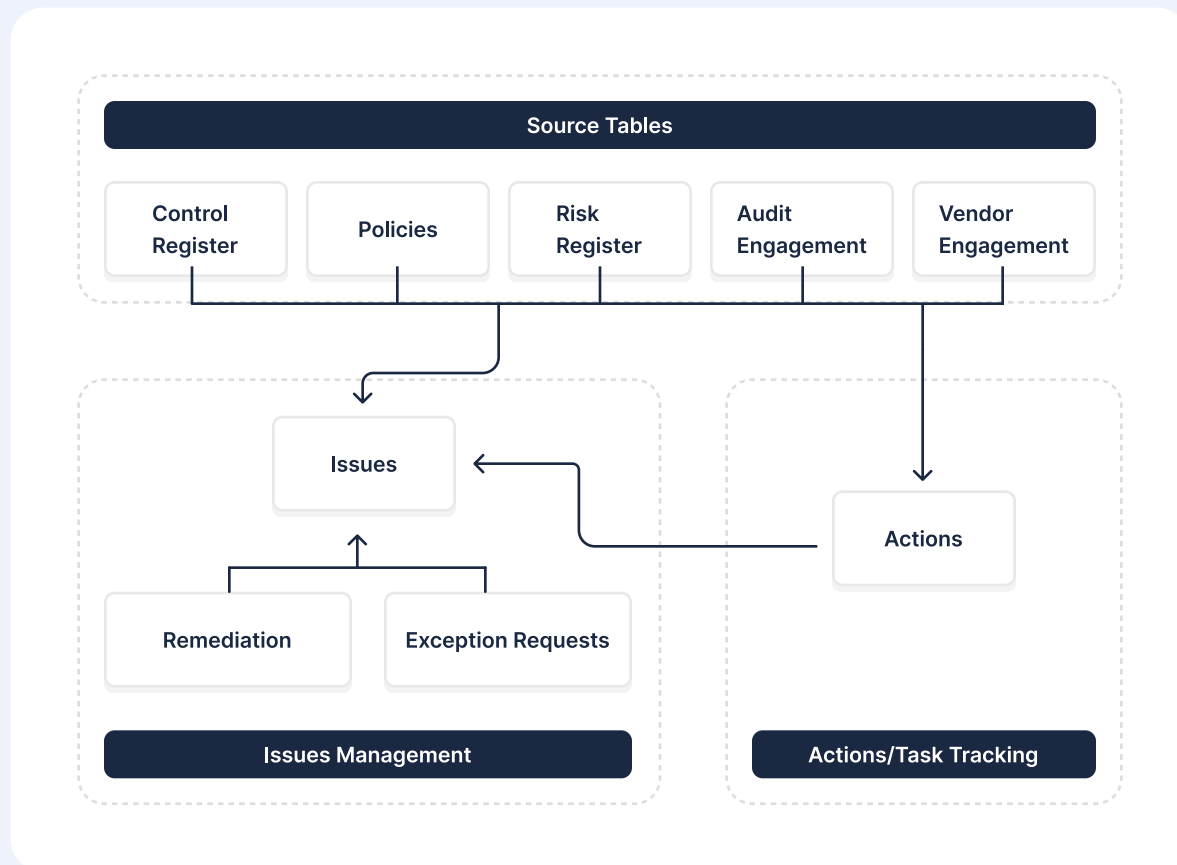
PHASE 1

Define Clear Record Types Aligned to Process Outcomes

Start by identifying the core record types required to represent the process end-to-end. Each table should represent a single, well-defined business object, not a form, view, or workflow step.

Strong record types in SmartSuite:

- Represent a distinct entity with a clear business meaning (for example: Issue, Risk, Vendor, Control)
- Have a defined lifecycle that can be expressed through status fields
- Support ownership, permissions, and automation independently
- Can be reused across multiple solutions or workflows



Avoid designing tables around how data will be entered or displayed. SmartSuite's flexible views, forms, and interfaces are designed to adapt to the model, not define it. For example, If a table requires large numbers of conditional fields that are only relevant in certain scenarios, this is often a signal that multiple record types are being forced into a single table.

1. Does this represent a single business object?
2. Does it have one clear owner and lifecycle?
3. Could this record exist independently of a specific workflow?
4. Would its meaning be clear if referenced by another solution?
5. How will this record connect to other tables, and will it be reusable across other solutions?

If any answer is "no", revisit the record definition.

PHASE 2

Design Record Lifecycles to Enable Critical Process Activities

Record lifecycles are central to how automation, permissions, and reporting behave. Lifecycle stages should reflect meaningful operational milestones, not cosmetic labels.

Effective lifecycle design:

- Uses a primary Status field that reflects end-to-end progression
- Avoids excessive or overlapping statuses
- Enables logic triggers (automation, conditional visibility, permissions, notifications)
- Supports reporting on aging, throughput, and bottlenecks

Statuses should be designed to be machine-actionable. If a status cannot reliably drive logic or reporting, it may not belong in the lifecycle. Status fields are frequently used as automation triggers, permission gates, and conditional visibility drivers. Poorly defined statuses result in brittle automation and unclear access rules.

For each status value, confirm:

1. Does a real-world milestone occur here?
2. Does something change in the system at this stage?
 - a. Automation triggers
 - b. Required or locked fields
 - c. Permission changes
3. Will we want to report on time spent in this stage?

If a status has no system behavior attached, consider removing or merging it.

PHASE 3

Standardize Field Types and Definitions to Drive Consistency

Field-level consistency is one of the strongest predictors of reporting and automation success in SmartSuite. Field-level inconsistency quickly undermines trust in data and limits cross-table analysis. Teams should establish a shared field dictionary before configuration begins.

Best practices include:

- Defining the purpose, data type, and allowed values for each field
- Reusing standardized fields (for example, Status, Priority, Owner) with consistent definitions
- Using controlled vocabularies for fields that drive logic, reporting, and automation
- Clearly distinguishing informational fields from decision-driving fields

Fields such as Status, Priority, Owner, and Category should never be ambiguous across tables. Multi-selects and free-text fields should be avoided for any field that drives automation, permissions, or rollups. Use single-select fields or reference tables to maintain data integrity.

For each field, validate:

1. Is this field used for reporting, logic, or permissions?
 - a. If yes, is the value set controlled and standardized?
2. Is this field reused elsewhere with the same definition?
3. Could this field eventually need to be rolled up or filtered across tables?

If consistency cannot be guaranteed, redesign the field.

PHASE 4

Model Relationships Intentionally to Support Reuse and Reporting

Relationships are foundational to SmartSuite's power, but they require intentional design. Poor relationship choices are difficult to unwind once solutions are live.

When modeling relationships, explicitly define:

- Cardinality (one-to-many vs. many-to-many)
- Ownership (which record controls creation and lifecycle)

- Reuse expectations (single-solution vs. cross-solution)
- Rollup and aggregation needs

Use parent-child relationships when one record owns another's lifecycle. Use many-to-many relationships when records must exist independently and be reused across contexts.

Avoid designing relationships to solve only a single use case. Instead, model shared master data (such as departments, risks, frameworks, or vendors) in ways that allow reuse across multiple solutions.

For each relationship, ask:

1. Can each record exist independently?
2. Should deleting or closing one record affect the other?
3. Do we need rollups in one or both directions?
4. Will this data be reused across multiple solutions?

Answers should clearly indicate parent-child or many-to-many.



PHASE 5

Build Reporting Needs into the Model, Not on Top of It

Reporting challenges often stem from modeling gaps rather than dashboard configuration issues. In SmartSuite, reporting works best when aggregation paths are designed upfront.

During modeling, identify:

- Key metrics and KPIs
- Required rollups and summaries
- Time-based analysis needs
- Cross-table reporting scenarios

Design fields and relationships so these insights can be generated without excessive formulas or workaround tables. If reporting requires manual reconciliation or repeated formulas across dashboards, the underlying model likely needs refinement.

Before sign-off, confirm:

1. Can key metrics be rolled up naturally?
2. Are time-based fields available where needed?
3. Do relationships support aggregation without duplication?
4. Does the model reflect what different user roles need to see to do their work effectively?
5. Will the available fields support meaningful gap analysis, trend identification, and decision-ready insights?

If reporting feels forced, revisit the model.

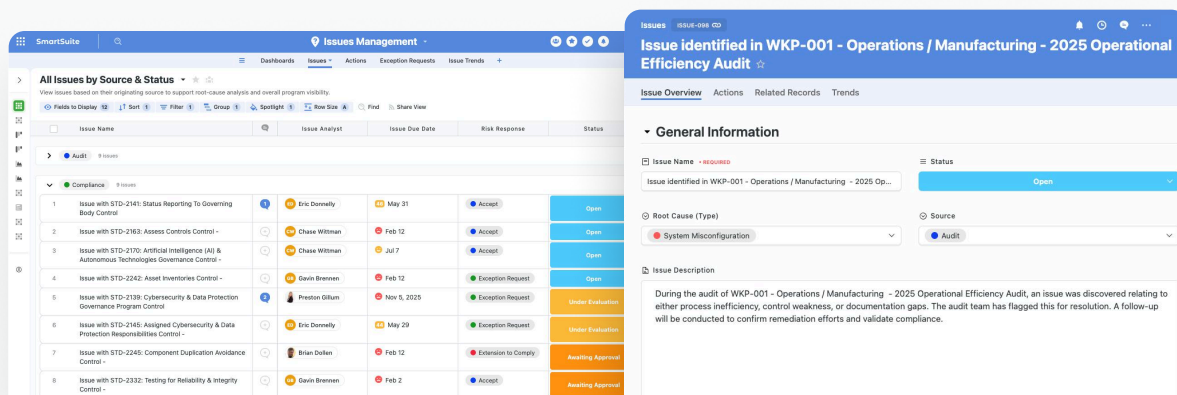
PHASE 6

Apply Data Validation Controls at the Right Time in the Lifecycle

Data quality improves when validation aligns to process and lifecycle maturity, not initial data entry. Over-validating early slows adoption; under-validating late undermines trust.

Effective validation strategies:

- Make fields required only when they become operationally necessary
- Use conditional visibility to reduce noise and focus user attention
- Use role-specific views and dashboards to surface required data at different lifecycle stages
- Apply permissions to appropriately control data visibility and edit access



Status-driven required fields and permissions dramatically improve data quality without increasing user friction.

Before finalizing validation rules, ask:

1. At what lifecycle stage does this data become critical?
2. Who is responsible for providing or maintaining this data?
3. What happens downstream if this data is missing or incorrect?
4. Should this data be visible to everyone, or only specific roles?
5. Does validation support user success, or does it create unnecessary friction?

If validation feels punitive or forces workarounds, refine the timing or controls.

PHASE 7

Design the Model for Integrations from the Start

A SmartSuite data model that supports integrations treats data as a shared asset rather than an isolated configuration. Even if integrations are not included in the initial rollout, designing with integrations in mind from the start significantly reduces refactoring, data transformation, and automation rework later.

Key integration-aware modeling considerations include:

- Defining stable unique identifiers and external reference keys
- Establishing clear source-of-truth ownership for each data element
- Using controlled values that can be consistently mapped across systems
- Avoiding overloaded or ambiguous fields that require transformation
- Designing relationships that external systems can reason about

Retrofitting integrations onto an existing model often exposes unclear ownership, inconsistent field usage, and fragile relationships. Modeling for integration readiness early dramatically reduces long-term complexity.

When validating the model for integration readiness, ask:

- Which tables are likely to send or receive data from external systems?
- Do records have a stable unique identifier that external systems can reference?
- Is it clear which system is the source of truth for each key field?
- Are field values standardized and mappable across systems?
- Will relationship structures make sense outside of SmartSuite?

If integrations are not part of the initial rollout, document these assumptions now to avoid rework later.



Common Planning Pitfalls

Even well-intentioned solution designs can encounter challenges when data modeling decisions are rushed, implicit, or overly influenced by short-term needs. The following pitfalls are modeling and planning issues, not platform limitations. They are included to help teams quickly recognize patterns that can undermine scalability, reporting, and long-term usability if left unaddressed.

Each pitfall below is presented as a concise narrative: what it typically looks like in practice, why it occurs, and a clearly identified corrective modeling pattern.

Modeling Around Screens or Forms Instead of Records and Lifecycles

In practice, this looks like tables that mirror forms or views rather than business entities. Duplicate or slightly varied fields appear across tables, and conditional visibility is heavily relied upon to explain meaning. The root cause is a model designed around how data is captured or displayed instead of what the record represents and how it progresses through a lifecycle.

How to Avoid It:

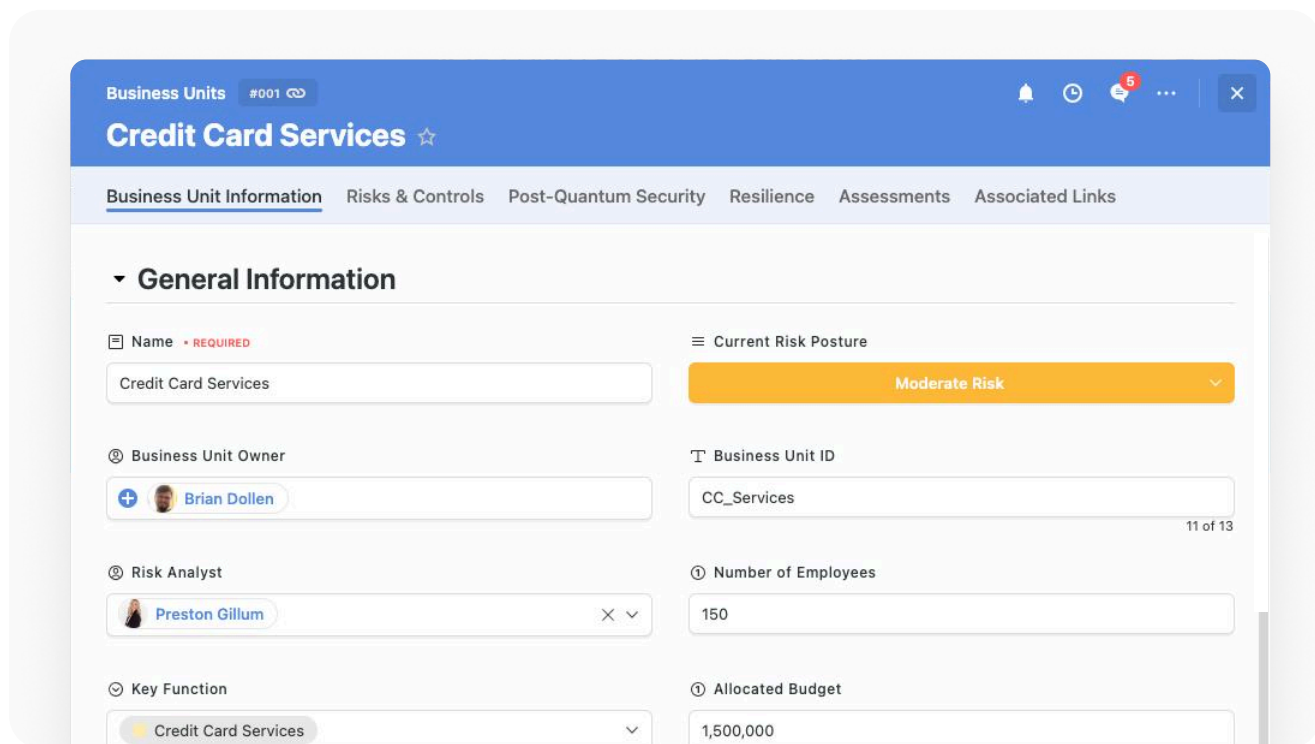
Define clear record types and explicit lifecycles first, then layer forms, views, and dashboards on top of the model.

Blurring Record Boundaries and Overloading a Single Table

This pitfall commonly appears as a single table attempting to represent multiple concepts, with many fields only relevant in certain scenarios. Over time, the table becomes difficult to understand and maintain. The root cause is combining conceptually distinct entities to reduce table count or accelerate initial setup.

How to Avoid It:

Separate distinct business entities into their own tables and connect them through intentional, well-defined relationships.



Inconsistent or Undefined Field Meanings Across Tables

In practice, this shows up as fields with the same name behaving differently across tables, leading to confusion and unreliable reports or rollups. The root cause is the absence of a shared field dictionary and inconsistent reuse of field definitions.

How to Avoid It:

Establish and enforce a shared field dictionary, and only reuse fields when their meaning and usage truly align.

Overreliance on Free-Text Fields Where Structure Is Required

This issue surfaces when categories, priorities, or decisions are captured as free text, resulting in inconsistent terminology and limited reporting value. The root cause is prioritizing flexibility over structure for fields that should drive logic, reporting, or automation.

How to Avoid It:

Use controlled vocabularies, single-select fields, or reference tables for any decision-driving data.

One-Off Relationship Design That Prevents Reuse

In practice, this looks like relationships built to support a single workflow, often resulting in duplicated reference data across solutions. The root cause is designing relationships reactively instead of as shared architecture.

How to Avoid It:

Model shared entities as reusable tables and design relationships with cross-solution use in mind.

Ignoring Cardinality and Directionality in Relationships

This pitfall commonly appears as rollups that double-count, unclear parent-child ownership, or difficulty aggregating data accurately. The root cause is selecting relationship types without explicitly defining one-to-many versus many-to-many behavior.

How to Avoid It:

Clearly define ownership, lifecycle control, and aggregation direction for every relationship.

Premature Optimization of Rollups and Calculated Fields

In practice, this shows up as complex formulas introduced early in the build, making it difficult to understand how values are derived and masking deeper structural issues. The root cause is attempting to solve reporting needs before validating core record structures and relationships.

How to Avoid It:

Stabilize record types and relationships first, then layer rollups and calculated fields once the model is validated.

Designing Without Realistic Sample Data

This pitfall occurs when models appear sound in theory but fail once real-world scenarios and edge cases are introduced. The root cause is validating the model without representative data volume or lifecycle complexity.

How to Avoid It:

Test the model using realistic sample data before final sign-off.

Lack of Formal Sign-Off on the Data Model

In practice, this looks like ongoing structural changes during build, resulting in instability across automation, reporting, and permissions. The root cause is proceeding without documented agreement on the data model.

How to Avoid It:

Establish a formal modeling sign-off package before moving into build and UX configuration.

Underestimating the Cost of Model Changes Post-Launch

This pitfall emerges when teams hesitate to improve the model after go-live, allowing workarounds to accumulate and confidence to erode. The root cause is underestimating how deeply model changes affect downstream logic, reporting, and integrations.

How to Avoid It:

Invest modeling discipline early to minimize long-term operational friction.

High Level Model Checklist

The checklist below supports data model validation, including table definitions, field descriptions, relationship patterns, and rollup logic. Completing this checklist produces a clear modeling sign-off package.

Checklist Item	Description	Benefits
Clear Record Types Defined	Each table represents a single, well-defined business object.	Improves clarity and scalability.
Ownership and Lifecycles Established	Records have clear owners and meaningful lifecycle stages.	Enables governance and automation.
Field Definitions Standardized	Fields use consistent definitions and controlled values.	Improves data quality and trust.
Relationships Intentionally Designed	Cardinality and ownership are explicitly defined.	Prevents reporting and rollup issues.
Reuse and Extensibility Considered	Shared entities support cross-solution use.	Reduces duplication and redesign.
Reporting Requirements Mapped	Metrics and KPIs are supported by the model.	Enables reliable insights.
Role-Based Experience Considered	Views and visibility align to user responsibilities.	Improves adoption and usability.
Validation and Permissions Aligned	Data requirements match lifecycle maturity.	Ensures quality without friction.
Integration Readiness Assessed	Identifiers and data ownership are defined.	Simplifies future integrations.
Sample Data Validated	Model is tested with realistic data scenarios.	Surfaces issues before build.



Model Use Case Example: Issue & Action Tracking

This example shows how careful modeling sets the foundation for a scalable Issue and Action Tracking solution. By defining record types, lifecycles, fields, and relationships with reporting, validation, and permissions in mind, the model ensures clarity, accountability, and long-term usability.

Modeling Decision	Benefit
Step 1: Define Clear Record Types Rather than capturing everything in a single table, the team identified three distinct business objects and modeled them as separate tables: • Issue – Captures a discrete problem, finding, or gap requiring remediation. • Action Plan – Captures a defined set of actions intended to resolve one or more Issues. • Action Update – Captures progress updates, evidence, or status changes for a specific Action Plan.	Separating record types avoids overloading a single table, clarifies ownership, and supports independent lifecycles, enabling cleaner reporting, permissions, and automation.
Step 2: Design Meaningful Lifecycles Each record type was given its own lifecycle aligned to real operational milestones: • Issues progress from Identified → Assessed → Remediation In Progress → Closed. • Action Plans progress from Draft → Approved → In Progress → Completed. • Action Updates progress from Submitted → Reviewed → Accepted.	Purposeful lifecycles ensure statuses are machine-actionable and usable for automation, reporting on aging, and accountability tracking.
Step 3: Standardize Fields and Controlled Values Key fields were standardized using shared definitions and controlled values: • Issue Priority uses a single, consistent value set. • Status and Owner fields share common definitions across tables. • Impacted Departments reference organizational data rather than free text.	Standardized fields enable consistent terminology, reliable rollups, trustworthy dashboards, and predictable automation behavior.

Modeling Decision	Benefit
Step 4: Model Relationships Intentionally Relationships were designed based on reuse and reporting needs: • Issues and Action Plans use a many-to-many relationship to support reuse. • Action Plans and Action Updates use a parent-child relationship for clean rollups. • Issues relate to Risk, Compliance, and Organizational Hierarchy records for context.	Intentional relationship design prevents duplication, supports reuse, and ensures aggregation behaves predictably as the solution scales.
Step 5: Design for Reporting from the Start Reporting requirements were identified during modeling, including: • Issue and Action aging by priority and department. • Overdue and completed Action Plans. • Risk and compliance impact summaries.	Building reporting paths into the model allows insights to be generated naturally without complex formulas or workaround tables.
Step 6: Apply Validation and Permissions by Lifecycle Stage Validation and access controls were aligned to lifecycle maturity: • Issues require minimal data at creation but enforce full impact assessment before remediation. • Action Plans require a Primary Owner before approval. • Action Updates restrict edits once reviewed or accepted.	Lifecycle-aligned validation improves data quality while minimizing user friction and workarounds.
Step 7: Prepare for Integration and Scale Even without immediate integrations, the model assumed future expansion: • Stable unique identifiers were defined for Issues and Action Plans. • Source-of-truth ownership was clarified for key fields. • Controlled values were designed to align with external systems.	Integration-ready modeling reduces future refactoring and supports long-term scalability.

When modeling their Issue and Action Tracking solution, the team leveraged the information gathered during the planning phase to establish their data model. The following key modeling decisions were made: