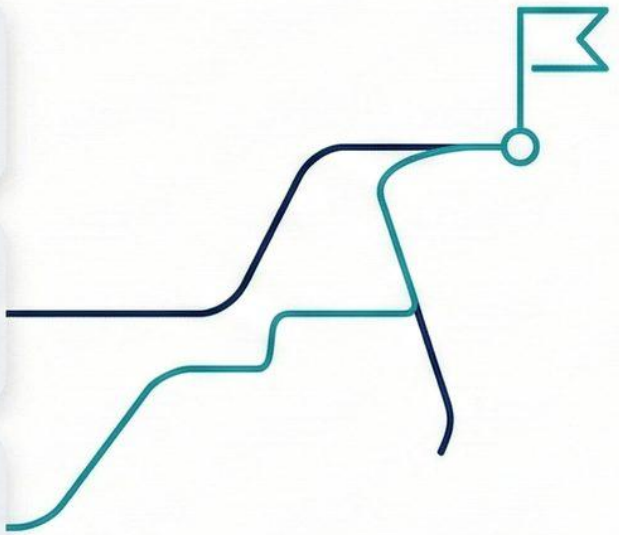
D365 Finance & Operations Development Series

Focused on MB-500 Certification Preparation

What to Expect

1. Conceptual understanding of D365 F&O development principles and architecture.
2. Structured approach covering key MB-500 exam topics and skills.
3. Insights to help you determine if this development path is the right fit for your career.



Set your expectations for a focused journey into D365 F&O development and MB-500 certification.

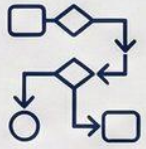
truestack.dev

Dynamics 365 F&O Developer Series: Conceptual Foundations

A concept-focused approach structured around certification preparation, explaining core topics beyond hands-on steps.



1. Architecture & Core Concepts



2. Development Patterns



3. Extensibility Models



4. Security & Best Practices

Conceptual Understanding vs. Hands-on Tutorials

This Series Focuses On:

- ✓ Explaining the “What” and “Why”
- ✓ Understanding underlying principles
- ✓ Certification preparation structure
- ✓ Building a strong conceptual foundation

Not Focused On:

- ✗ Step-by-step application clicks
- ✗ Walking through every screen
- ✗ Real-time button-pushing tutorials
- ✗ Hands-on lab exercises

These conceptual videos are designed to complement, not replace, dedicated hands-on tutorial content available separately.

truestack.dev

Important Disclaimer: Presentation Content

NOT Actual Exam Questions

Cannot share real exam questions (not allowed).

Not a source for direct exam problem examples.

Does not encourage memorizing specific questions.

Structured Preparation Notes

Based on personal preparation for the MB-500 exam (June 2025).

Organized material from extensive study and various questions.

Focuses on understanding concepts to handle variations.

Understanding the concepts properly is what helps during the exam and in real work, not memorizing specific problems.

truestack.dev

Who Is This For?



MB-500 Preparation

If you're preparing for the MB-500 certification exam, you'll find this content highly useful for your studies.



New to D365 F&O

For those who are new to Dynamics 365 Finance & Operations and want to understand the fundamental concepts.



Experienced Professionals

If you're already experienced but have gaps in certain areas, this series helps fill those specific knowledge voids.



Structured Revision

Ideal for anyone needing a structured revision of the core D365 Finance & Operations concepts.

This content is designed to cater to a wide range of learners, from beginners to experienced professionals seeking structured knowledge.

truestack.dev

What Will You Get From This Series?



INCLUDED: Conceptual Understanding

- ✓ Explanations of how D365 Finance & Operations actually works.
- ✓ Concepts broken down for easier grasping.



REQUIRED SEPARATELY: Practical Application

- Hands-on practice in actual environments.
- Practice questions from other sources.
- Real scenario work.

→ This series covers the conceptual side; the practical side, you need to do that yourself.

truestack.dev

A Note on Content Focus: Certification Preparation



The Certification Lens

- Exam-Driven Content: This material is directly inspired by Microsoft certification exam requirements.
- Emphasis on Principles: The exam heavily tests specific principles and best practices Microsoft expects you to know.



For Experienced Developers

- Seemingly Obvious Topics: You might encounter concepts that feel overly emphasized (e.g., 'extend instead of overlay').
- System vs. Theory: While the system often guides you automatically, the exam tests your theoretical understanding of 'why'.

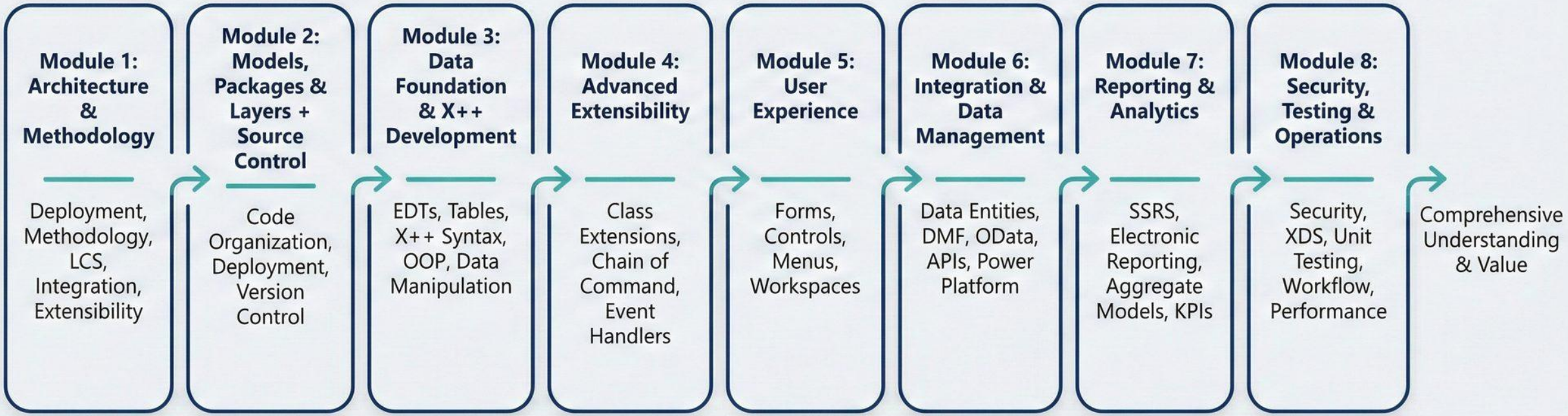


The Goal: To ensure concepts are **clear for the exam context**, not to doubt your practical experience. We stress these points for certification clarity.

truestack.dev

Series Structure & Content Organization

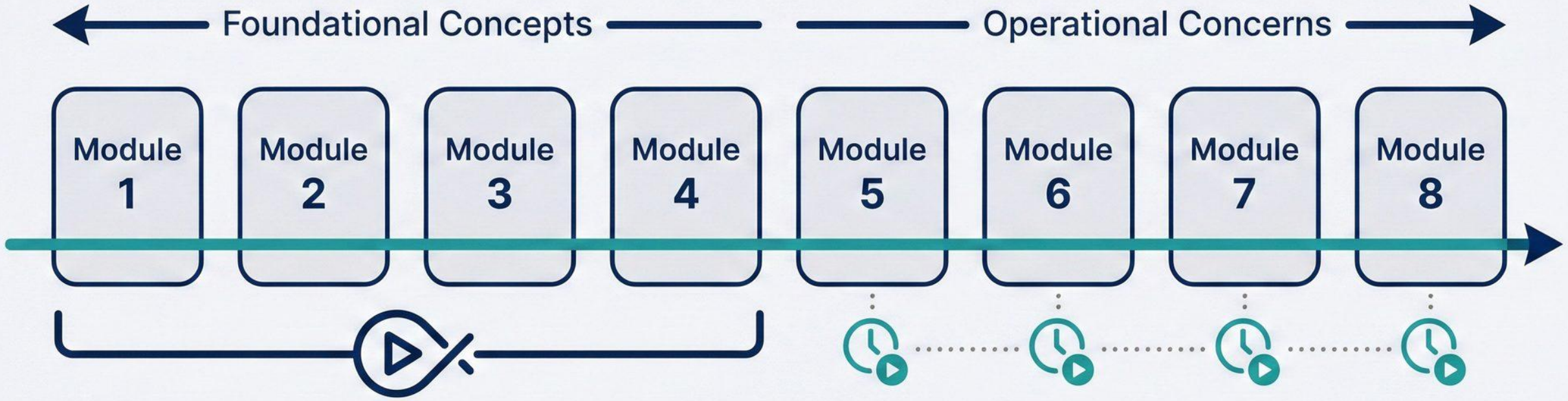
A logical progression from foundational concepts to operational expertise.



Understanding the organization helps you navigate and maximize value from each topic.

truestack.dev

Content Structure & Series Overview



Comprehensive Understanding
(Watch End-to-End)

Focused Learning
(Jump to Specific Areas)

This series is organized into 8 major modules that build logically on each other.

truestack.dev

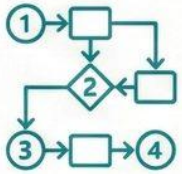
Module 1: Architecture & Methodology

Foundational Concepts



Deployment Models

Understanding how the system gets deployed, including cloud, on-premises, and hybrid options.



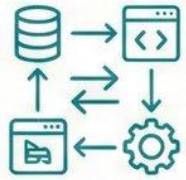
Project Methodology

Structuring and managing projects through phases like analyze, design, develop, and deploy.



LCS Workspace

Your central hub for managing environments, monitoring, and lifecycle services.



Integration Patterns

How different systems connect, exchange data, and communicate with each other effectively.



Extensibility Principles

The philosophy of customizing D365 safely using extensions without breaking core functionality.

Mastering these five foundational concepts is crucial for building scalable and maintainable Dynamics 365 Finance & Operations solutions.

truestack.dev

Module 2: Code Organization and Management

Focus on how code is organized, tracked, and deployed in the D365 Finance & Operations ecosystem.

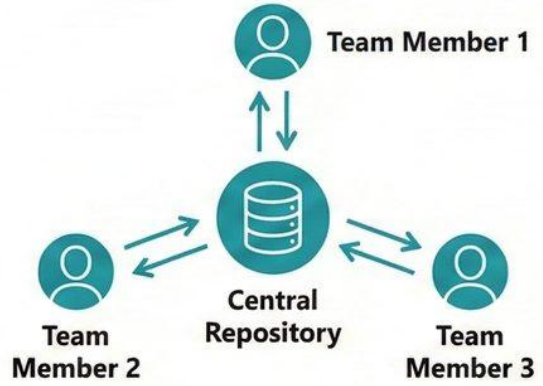
Code Structure

Models
Logical grouping of elements.

Packages
Deployment units, containing one or more models.

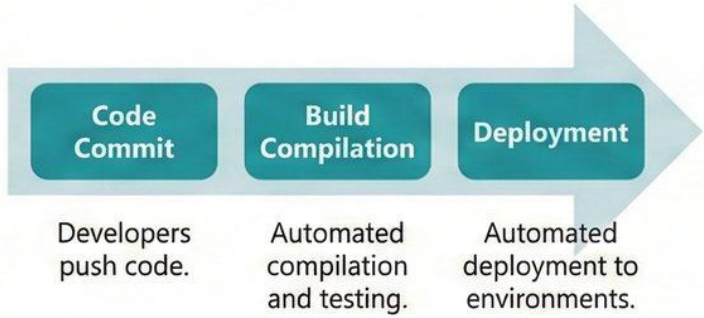
Layers
Hierarchical structure for customization and segregation (e.g., SYS, ISV, CUS).

Version Control



Enables parallel collaboration and change tracking without conflict.

Build Pipelines



Automates compilation and deployment for consistency and reliability.

Proper organization, effective tracking, and automated deployment ensure code quality and operational efficiency.

truestack.dev

Module 3: Data Foundation & X++ Development

Building solid foundations for Dynamics 365 Finance & Operations

Data Foundation

EDTs (Extended Data Types)
Define your data types.

Tables
Organize your data.

Indexes
Optimize your queries.

Relations
Connect different entities together.

X++ Development

Language Syntax
Covers the core language rules.

Object-Oriented Principles
Implements modular programming structures.

Proper Data Manipulation Techniques
Ensures efficient and correct data handling.

This module establishes the **solid foundations** for all subsequent development concepts.

truestack.dev

Module 4: Advanced Extensibility in D365 F&O



Key Philosophy: Extend Without Modifying

The modern way to customize D365 is by extending functionality, keeping the original code intact.

→ **Ensures Upgrade Safety**



Class Extensions (Chain of Command)

Add behavior to existing classes without touching them. Wraps original methods with pre and post logic.

Connects behavior, preserving original logic.



Event Handlers

Respond to specific system events to trigger custom actions. Decouples customization from core code.

Reacts to specific system triggers.

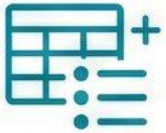


Table and Enum Extensions

Add fields and values to existing tables and enums where needed, without altering the base schema.

Expands data models and option lists.

This approach keeps your code safe when Microsoft releases platform upgrades.

Microsoft D365

Module 5: User Experience



Form Patterns

Ensure consistency and familiarity across the interface.



Controls

Provide the actual interaction points.



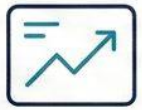
Menus and Navigation

Guide users through the system.



Workspaces

Provide role-specific dashboards tailored to what people actually need.



Tiles

Show live KPIs
Show live KPIs right where people can see them.



Personalization

Lets users customize
Lets users customize their own interface the way they like it.

Together, all of these create **professional user experiences** that people **actually want to use**.

truestack.dev

Module 6: Integration & Data Management



Data Entities

Expose your data for external systems to consume.



DMF (Data Management Framework)

Handles bulk imports and exports efficiently.



OData Services

Provides real-time REST API access when you need it.



Custom Services

Expose your business logic to other systems.



Asynchronous Patterns

Handle those high-volume scenarios that would otherwise bog things down.



Power Platform Connectivity

Integrates with the broader Microsoft ecosystem.

Connect D365 F&O with the world through efficient, scalable integration patterns.

truestack.dev

Module 7: Reporting & Analytics

Presenting data to drive better decision-making



SSRS Reports

Provide detailed analysis for deep dives.

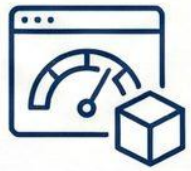
→ Complex, paginated reports



Electronic Reporting (ER)

Enables configurable documents customizable without code.

→ User-defined formats (e.g., XML, JSON, PDF)



Aggregate Models

Power live KPIs and high-level operational metrics.

→ Real-time performance indicators



Embedded Analytics

Bring insights directly into forms and dashboards where useful.

→ Contextual insights in workflows

This module helps you understand how to present data in ways that actually drive better decision-making.

truestack.dev

Module 8: Security, Testing & Operations

Operational Excellence & Production Readiness

Security & Access Control

Role-Based Security

Controls who can do what in the system.

Defines permissions and access levels based on user roles.

XDS Policies

Filter data at runtime.

Restricts data visibility dynamically based on user context.

Quality & Testing

Unit Testing with SysTest

Ensures your code quality stays high.

Automated testing framework for verifying individual code units and functions.

↳ *Promotes maintainable and reliable codebases.*

Operations & Workflow

Workflow Automation

Routes documents through approval hierarchies.

Automates business processes and approval cycles automatically.

Performance Monitoring

Keeps your systems running smoothly.

Proactively tracks system health and identifies bottlenecks in production.

This module focuses on ensuring your solution is secure, high-quality, and operationally ready for production.

truestack.dev

Consistent Topic Structure & Approach

Core Concepts

Core concepts are explained clearly—not just the ‘what,’ but the ‘why’ behind them.



Foundation & Rationale

Comparison Tables

Approach 1	Approach 2
Understand when to use different approaches and why.	
✅ Pros / ❌ Cons	✅ Pros / ❌ Cons

Best Practices

Best practices show you the professional way to do things.

- ✓ Standardized methods
- ✓ Optimized for efficiency
- ✓ Future-proof designs

Anti-Patterns

And anti-patterns highlight those common mistakes you want to avoid making.

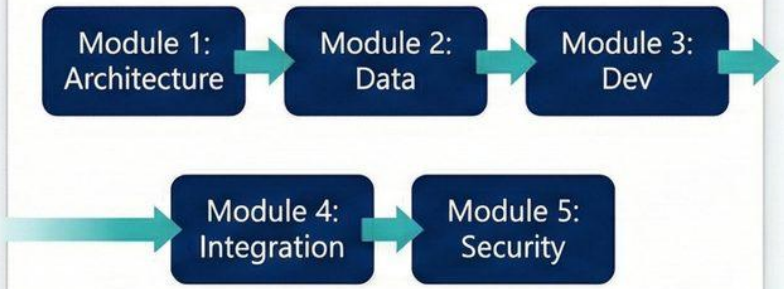
- ❌ Common mistakes
- ❌ Performance bottlenecks
- ❌ Maintenance risks

Each topic follows a structured approach to ensure clarity, practical application, and professional growth.

truestack.dev

How to Use This Tutorial Series: Flexible Learning Paths

End-to-End Comprehensive View



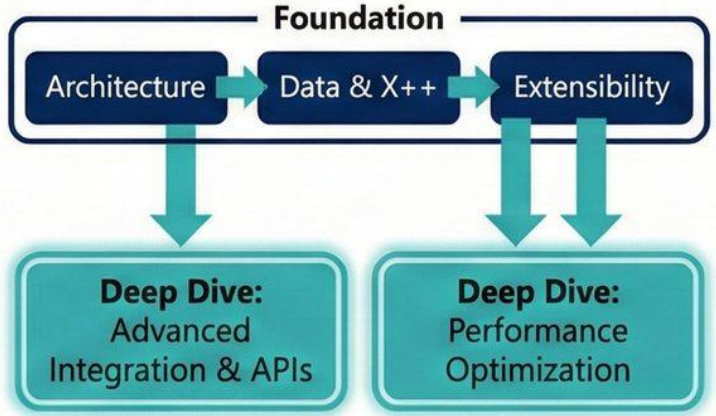
Watch the complete series sequentially to build a strong, comprehensive understanding of the entire Microsoft Dynamics 365 Finance & Operations platform.

Targeted Module Focus



Use timestamps to jump directly to specific modules or topics relevant to your immediate needs, focusing your time efficiently.

Hybrid Approach (Foundation + Deep Dive)



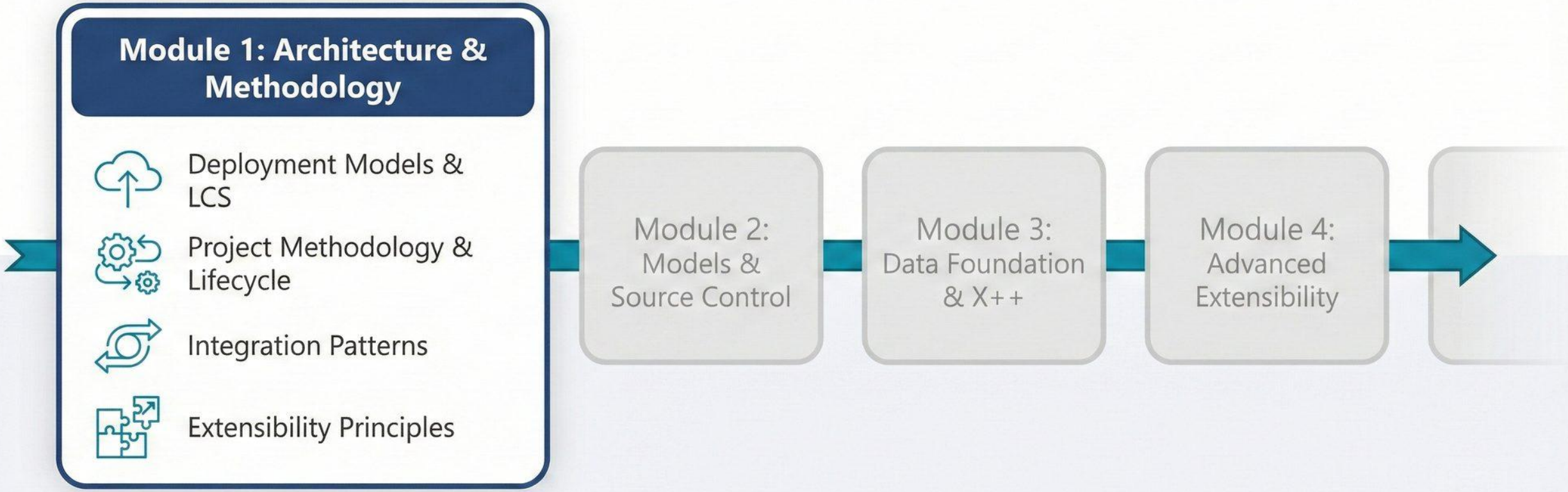
Start with the foundational modules for a solid base, then dive deep into the specific areas critical to your role or current project.

Takeaway: Adapt your learning path based on your current knowledge and specific goals for maximum effectiveness.

truestack.dev

Module 1: Architecture and Methodology

The Journey Begins: Foundation Concepts, Deployment, & Extensibility



You have the roadmap. Let’s start the journey with the foundational concepts.

truestack.dev

Foundation: Architecture & Methodology

Blueprints for Construction: Understanding Deployment, Organization, and Connectivity

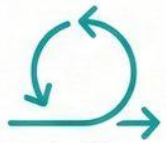
1. Deployment & Environment



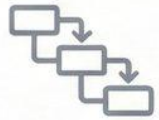
Where the system runs. Cloud vs. On-Premises options impact features, analytics, and tools.

Examples: LCS, Power BI, BYOD

2. Project Methodology



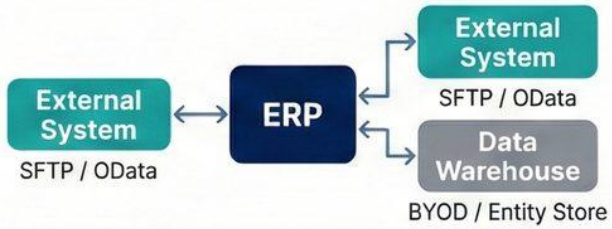
Agile (Iterative)



Waterfall (Sequential)

How we organize work. Choose based on requirement stability and feedback needs.

3. Integration Architecture



How systems connect. Patterns for batch ingestion, real-time exchange, and analytics.

D365 F&O System Foundation (Base Architecture)

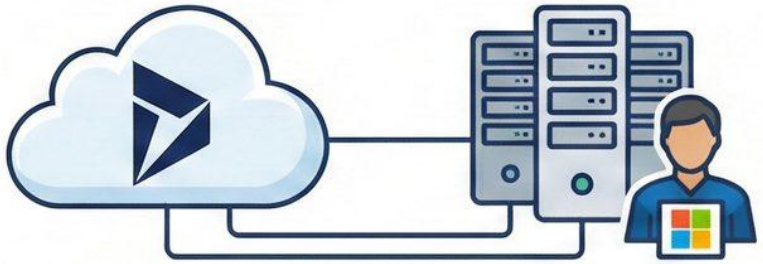
Key Takeaway: A solid understanding of deployment, methodology, and integration is the prerequisite for successful system building.

truestack.dev

D365 Deployment Architecture: Cloud vs. On-Premises

Where does your D365 system actually run? Understanding deployment-specific features is critical for design.

Cloud Deployment (Microsoft-Managed)

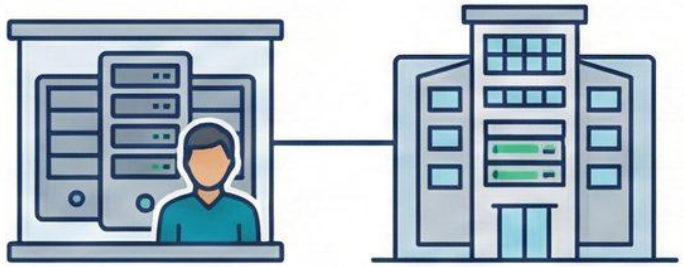


Microsoft manages the servers.

Specific Features (Cloud-Optimized):

- Task Recorder ↔ BPM library integration (LCS)
- Embedded Analytics & Power BI.com integration
- SharePoint electronic reporting storage
- Cloud-hosted Lifecycle Services (LCS) APIs

On-Premises Deployment (Self-Managed)



Runs on your own servers in your own building.

Specific Features (On-Premises-Optimized):

- Power BI Desktop against local entity store
- Local infrastructure & management
- No direct Task Recorder ↔ BPM sync
- Document management relies on local storage

Key Takeaway: While the software looks and works the same, feature differences between deployment types dictate architectural decisions.

truestack.dev

Deployment Architecture: Cloud vs. On-Premises

Cloud Deployment



- Business Process Modeler & Task Recorder (rely on cloud APIs)
- SharePoint integration for document management
- No direct access to SQL database (must use platform APIs)

On-Premises Deployment



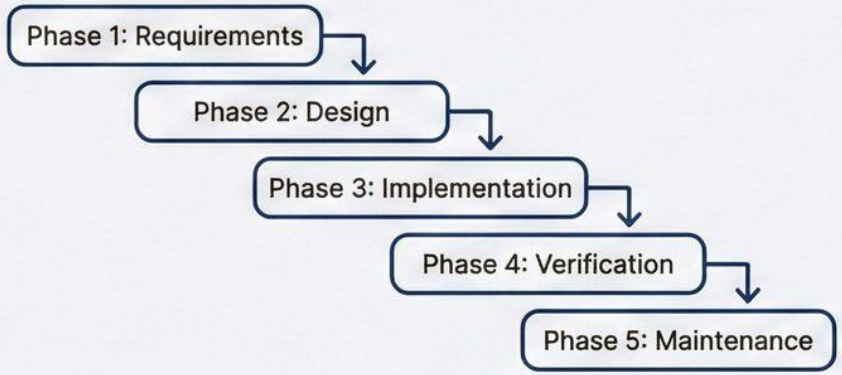
- Direct SQL access (powerful for reporting and analytics)
- No cloud-only services (e.g., BPM, Task Recorder)

Choose the deployment model based on the specific features and access required for your business situation.

truestack.dev

Organizing Project Work: Waterfall vs. Agile Methodologies

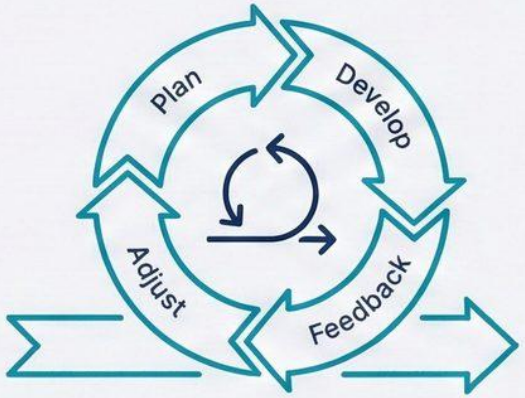
Waterfall Methodology



Traditional step-by-step approach. Finish phase one completely, then move to phase two.

- ✓ Works great if requirements never change.
- ✗ Challenges: Software requirements always change.

Agile Methodology



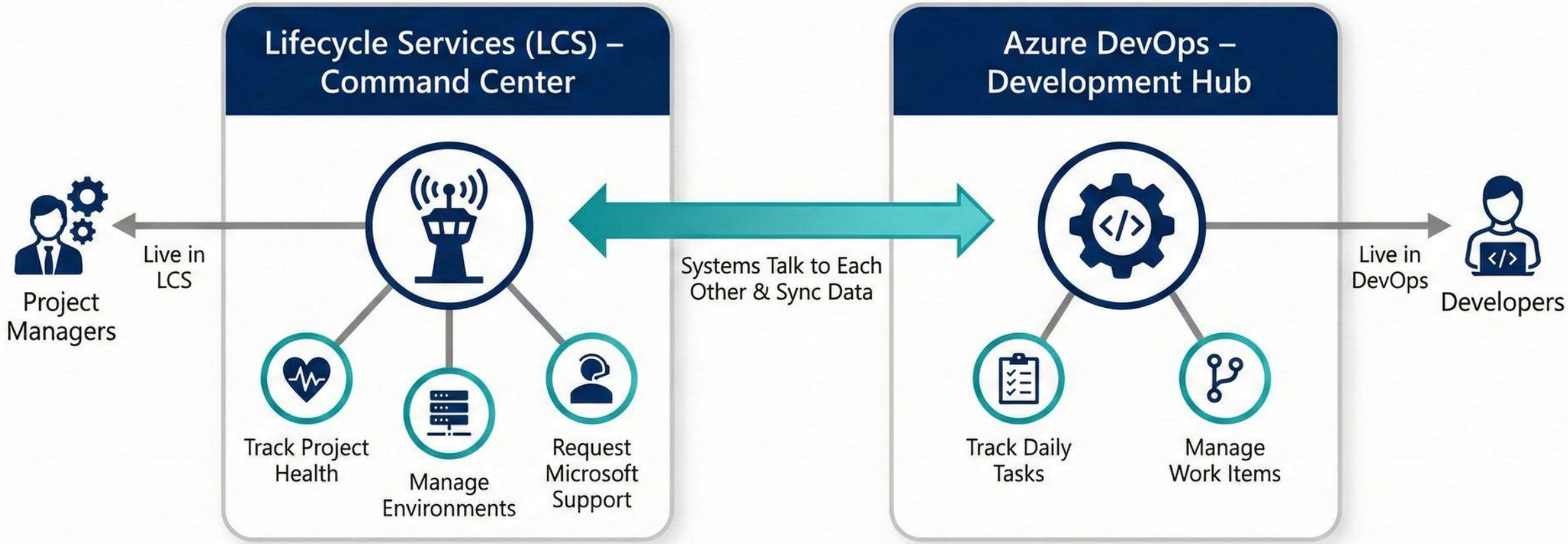
Work in short cycles, get feedback, adjust as you go.

- ✓ Adapts to shifting requirements.
- ✓ Ideal for ERP implementations where understanding evolves through use.

For ERP implementations, Agile is usually the right choice because requirements shift as the business sees the system in action.

truestack.dev

Lifecycle Services (LCS): The Project Command Center



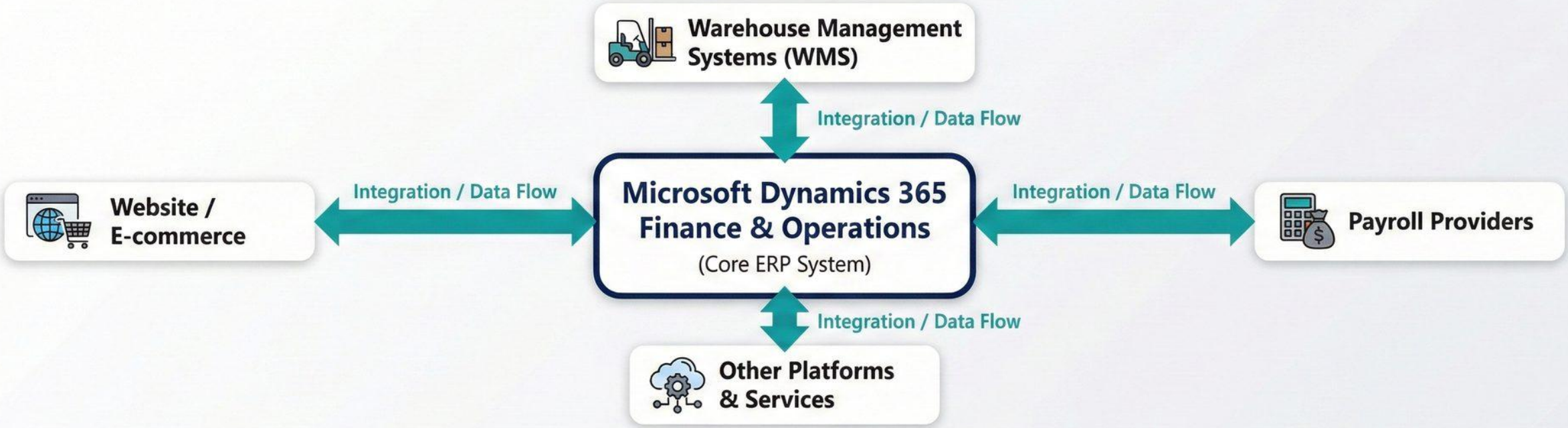
One Unified View of Project Status

Connecting project management and development for complete visibility.

truestack.dev

D365 Integration Strategy: Connecting Your Ecosystem

Understanding data flow between Dynamics 365 and external platforms.



Scenario 1: Massive Batches (Overnight)

High-volume data transfer, scheduled processing, bulk updates.

Example: End-of-day sales transactions, inventory synchronization.

Scenario 2: Real-Time Updates (Instant)

Instantaneous data exchange, event-driven, immediate feedback.

Example: Order confirmation, real-time stock availability check.

Takeaway: Successful integration requires tailoring the approach to the specific data volume and timing requirements of each connected system.

trycatch.dev

Choosing the Right Data Integration Pattern: Batch Jobs vs. OData

Think of data like cargo – match the tool to the volume.

Batch Jobs (High Volume)



Takes time to load, but moves massive volume

Scheduled/Nightly Imports

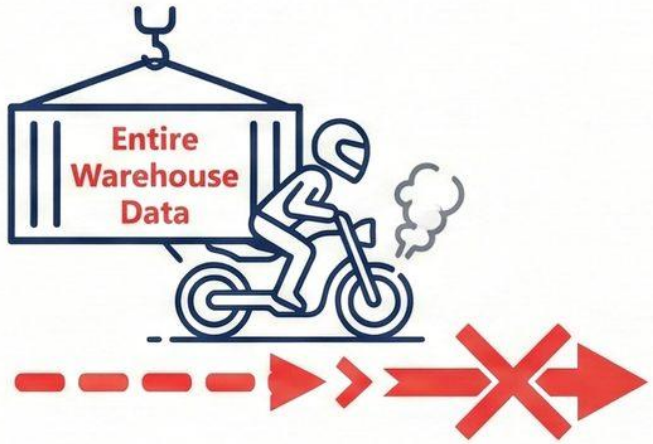
OData (Real-time, Single Item)



Fast, perfect for instant, single-item updates

Real-time API Calls

Anti-Pattern (OData for Bulk)



System Throttle & Timeout

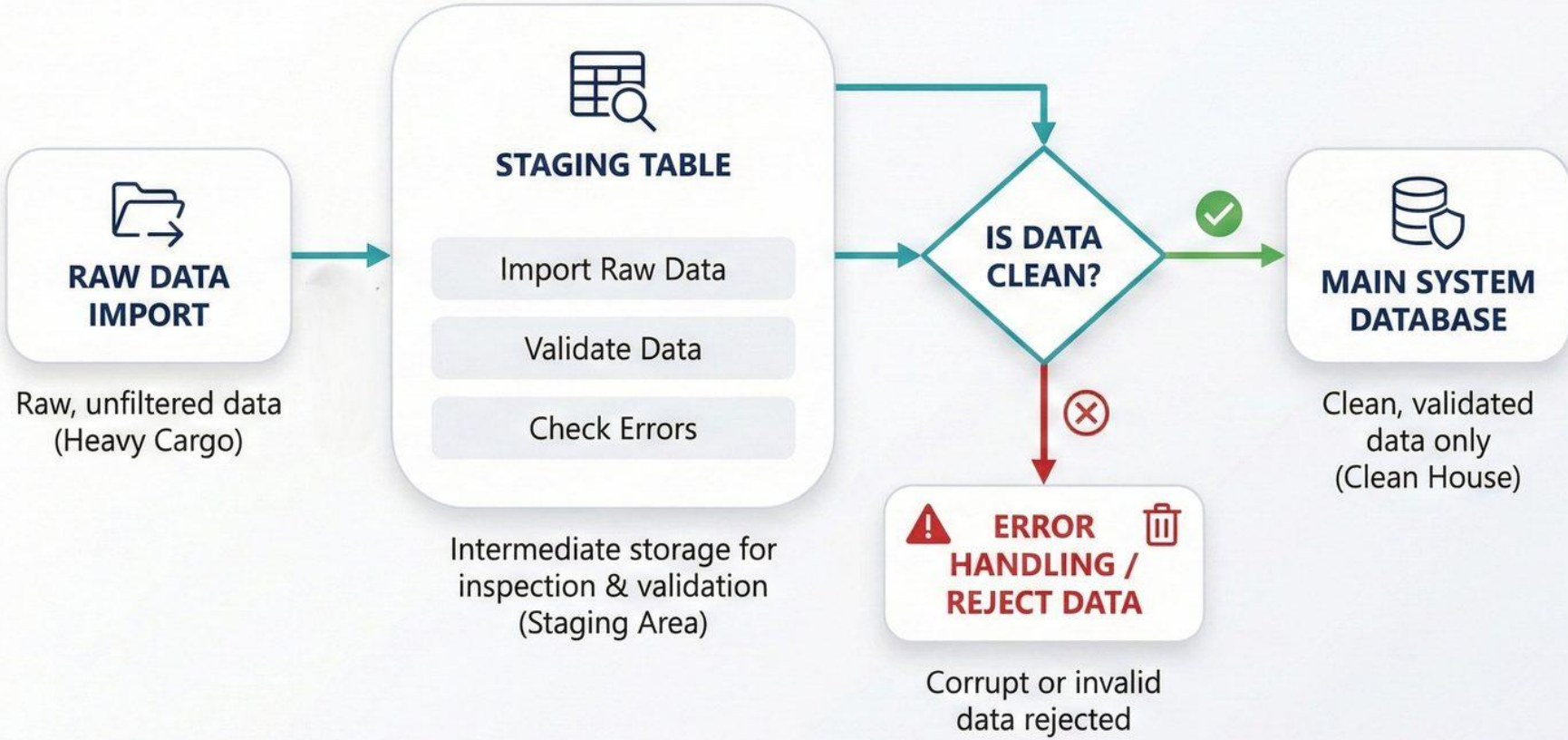
Don't push bulk data through OData

Key Takeaway: Use the right tool for the job – Batch Jobs for bulk volume, OData for real-time single records.

truestack.dev

Data Import Workflow: Using Staging Tables

Protecting database integrity with intermediate data validation



KEY BENEFITS

- Protects database from corruption
- Ensures data quality before commitment
- Fundamental for reliable batch processes

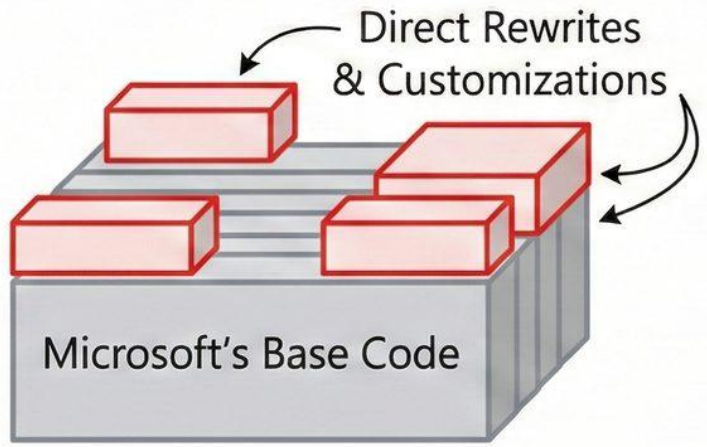
Staging tables act as a crucial buffer, allowing data inspection and cleaning before it enters the core D365 F&O system.

truestack.dev

Extensibility: Customizing D365

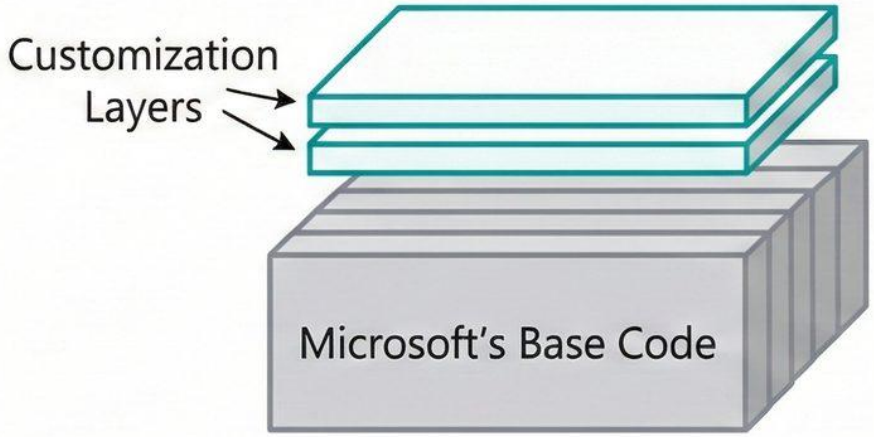
Transitioning from Overlayering to Extensions

Overlayering (Old Approach)



Conflicts during Updates.
Upgrades Break.

Extensions (Modern Approach)



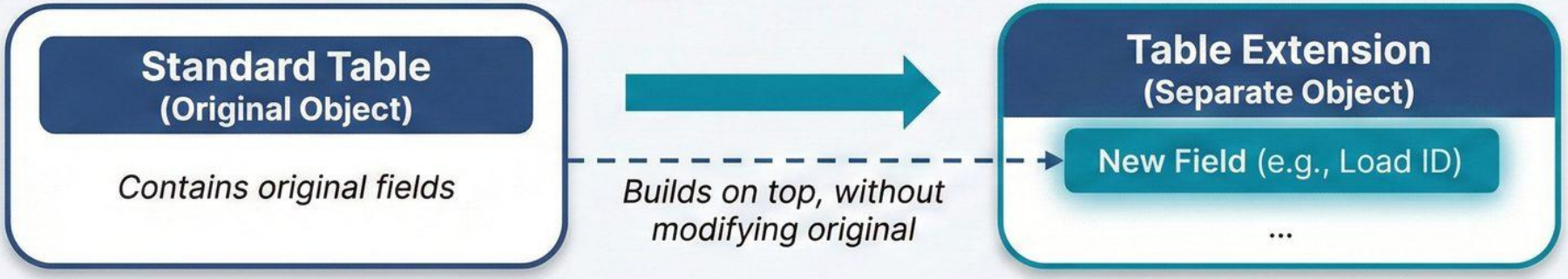
Original Stays Clean.
Customizations Separate & Safe.

Extensions allow layering customizations on top, keeping the original code clean and updateable, ensuring customizations remain separate and safe.

truestack.dev

Extending Tables & Database Synchronization

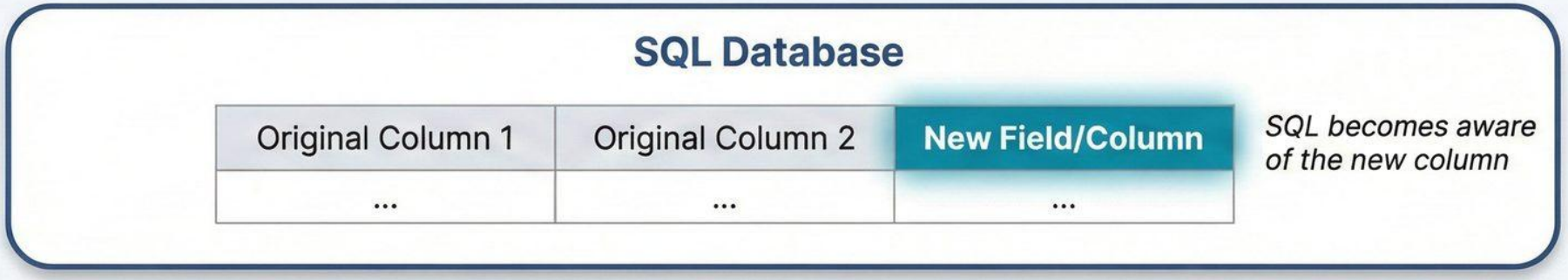
1. Create Table Extension (Code)



✅ Code Written

2. MANDATORY STEP: Database Synchronization

3. Update SQL Schema (The 'Sync')



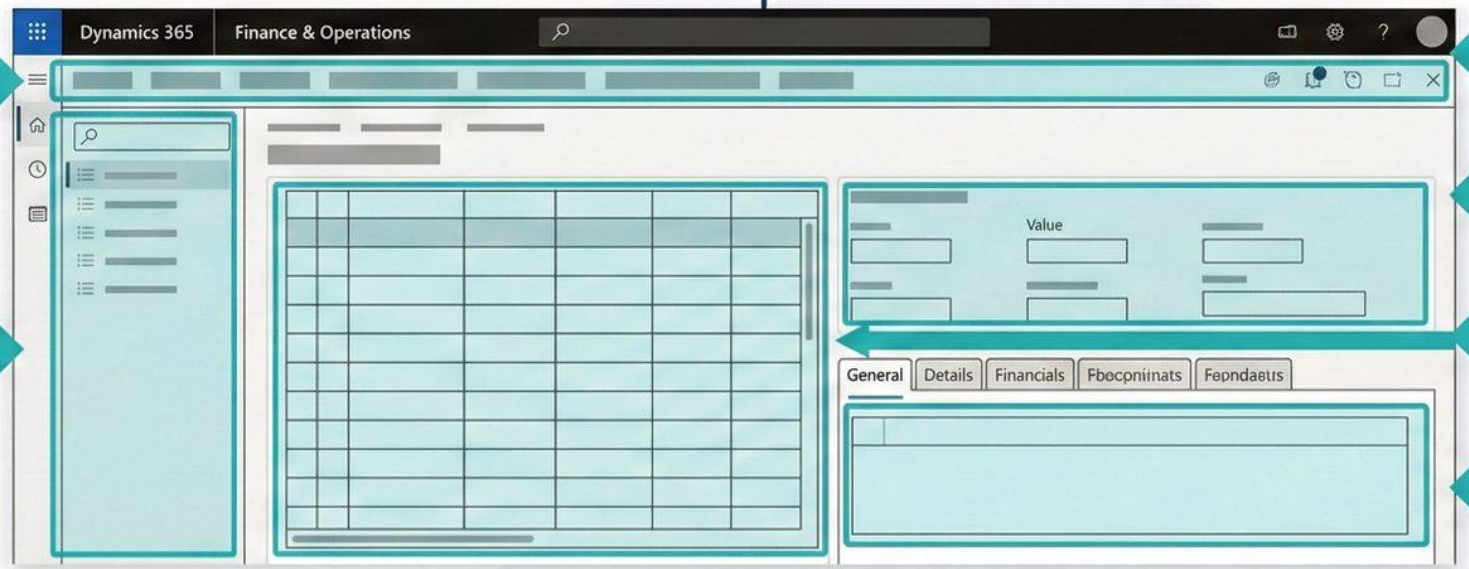
Without Sync: SQL doesn't know the new field exists → CRASH

⚠️ ❌ **If Skipped:** Code references unknown column, causing runtime errors.

truestack.dev

Extending Forms: Following Established Patterns

STANDARD FORM STRUCTURE



ACTION PANE
(Commands)

NAVIGATION LIST
(Filtering)

ACTION PANE
(Commands)

DETAILS HEADER
(Key Information)

LIST GRID
(Records)

DETAILS TABS
(Editing)

WHY THESE PATTERNS EXIST: ✓ Create consistency ✓ Make users feel at home in the interface

CONSEQUENCE OF BREAKING PATTERNS: ✗ Confuse users ✗ Fail certification requirements

Stick to the structure when adding new fields or features to maintain compliance.

truestack.dev

TOPIC 1.1 RECAP: FOUNDATIONS & STRATEGY

Connecting deployment, methodology, and integration principles



Deployment Strategy

Your choice—Cloud or On-Premises—determines tool availability.

Influences analytics, LCS features, and storage options.



Project Methodology

Agile is the right fit for software development.

Best for evolving requirements and frequent feedback cycles.



Lifecycle Services (LCS)

Your command center for the entire implementation.

Connects all pieces, tools, and project phases together.

Integration & Extensibility



Batch

High Volume Data: Use Batch Jobs.



OData

Instant Updates: Use OData.



CORE PRINCIPLE:

Always extend the platform, never modify directly.

These foundations support everything we'll cover next regarding code organization and deployment.

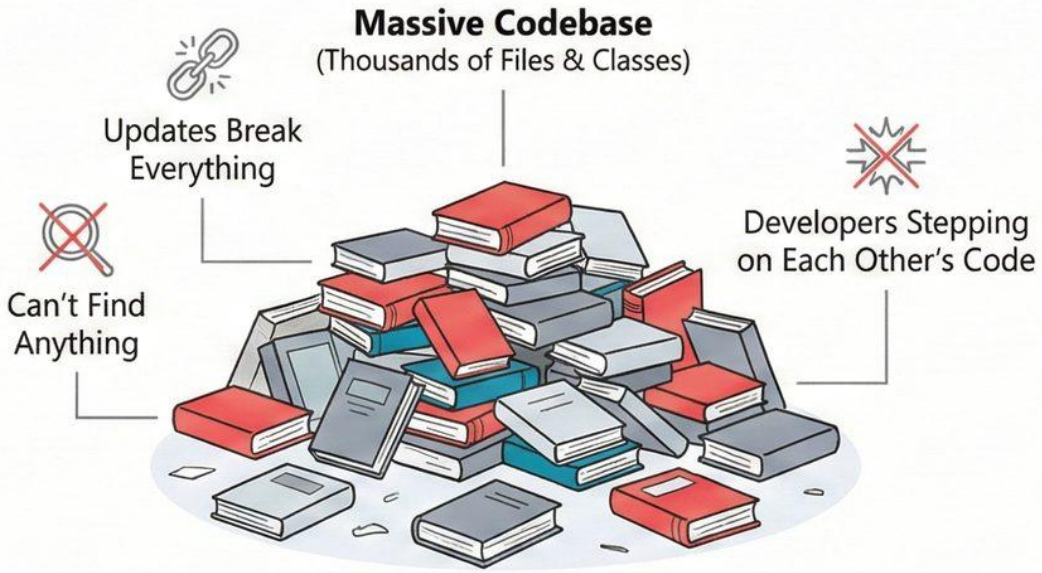
truestack.dev

Topic 1.2: Organizing the Actual Code

The Challenge of Massive Codebases

The Challenge: Unorganized Code

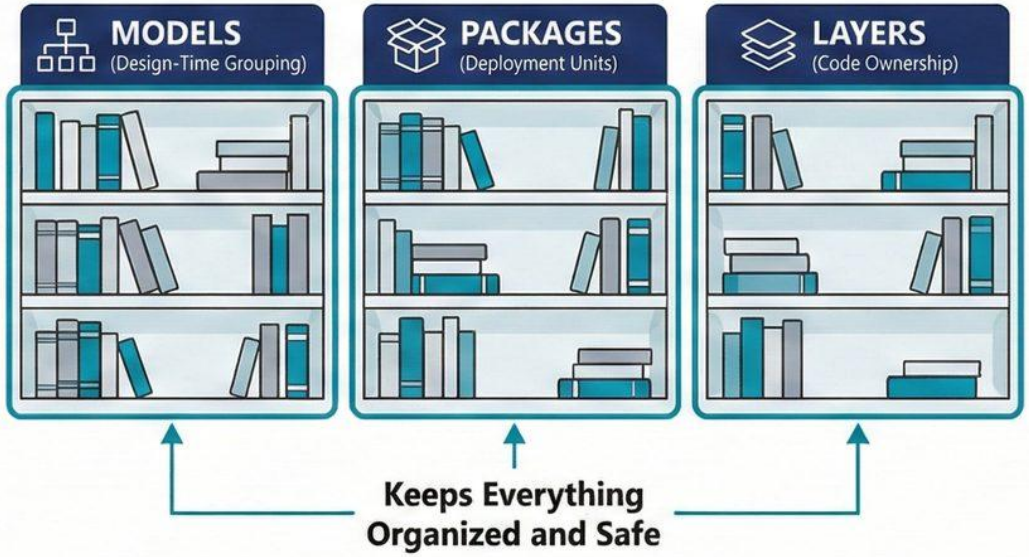
Analogy: A library with books in a giant pile



INCORRECT / CHAOS

The Solution: D365 Strict Containers

Analogy: A well-organized library with strict containers



CORRECT / ORGANIZED & SAFE

Dynamics 365 uses Models, Packages, and Layers to prevent code conflicts and ensure stability.

truestack.dev

Core Concepts: Project, Model, and Package in D365 F&O Development

PROJECT
(Development Workspace)



Your desk where you write code.

Visual Studio Workspace.

Design-time concept.

One project belongs to exactly one model

MODEL
(Feature Design)



Logical container for related code (e.g., 'Tax Calculation' extension).

Only exists during development.

One model compiles into one package

PACKAGE
(Deployment Unit)

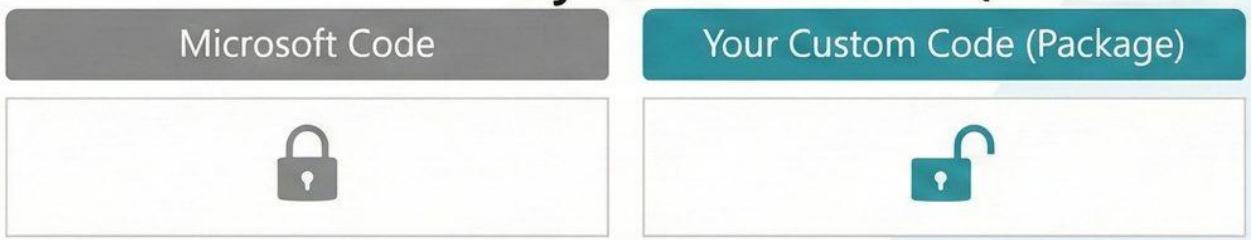


DLL file that runs on the server.

What actually gets shipped.

Deployment unit.

Why This Structure? (Benefits of Separation)

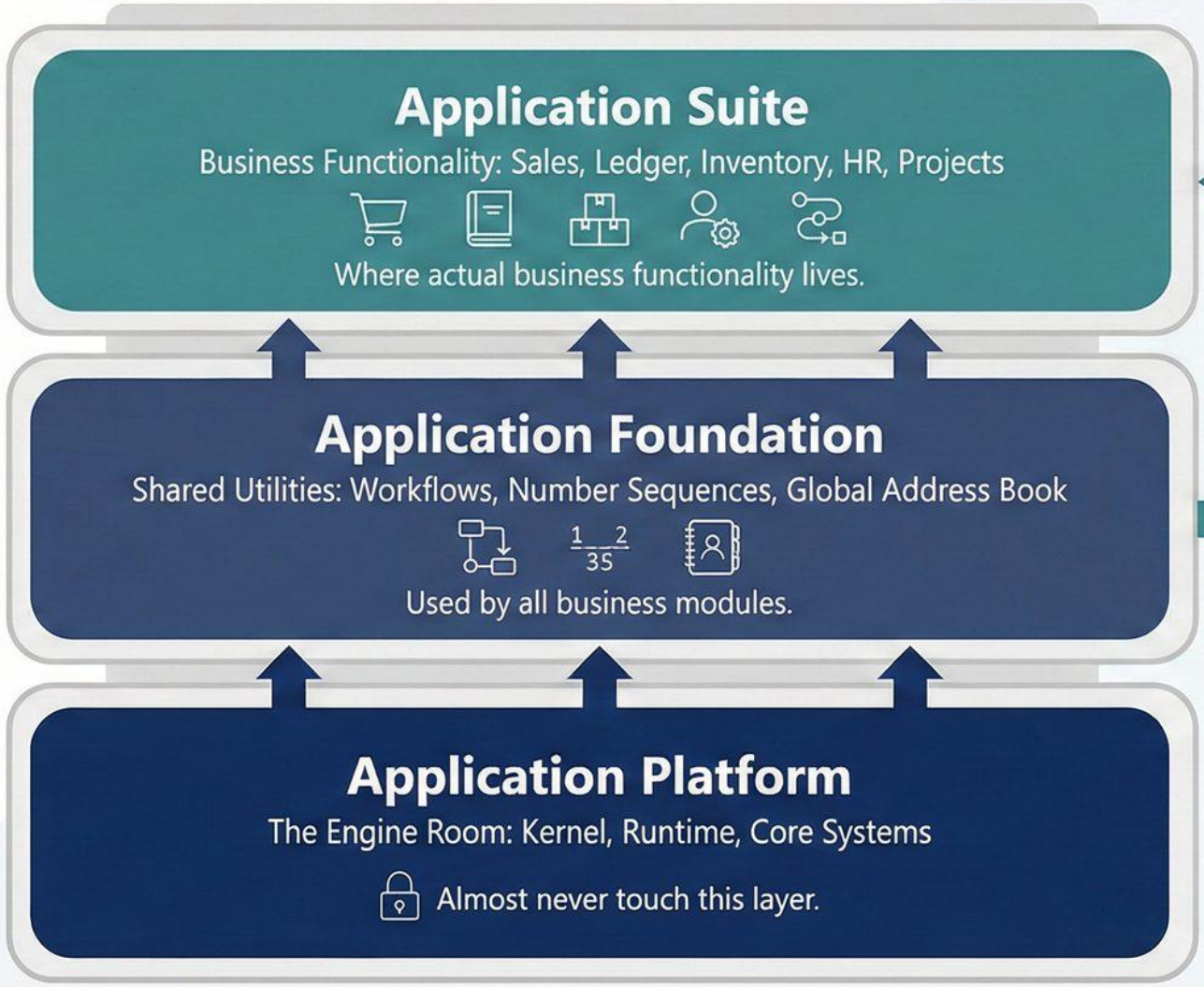


- Compiles independently
- Deploys independently
- Updates independently

This structure ensures your custom code remains completely separate from Microsoft's standard code.

truestack.dev

Microsoft's Application Packaging Structure

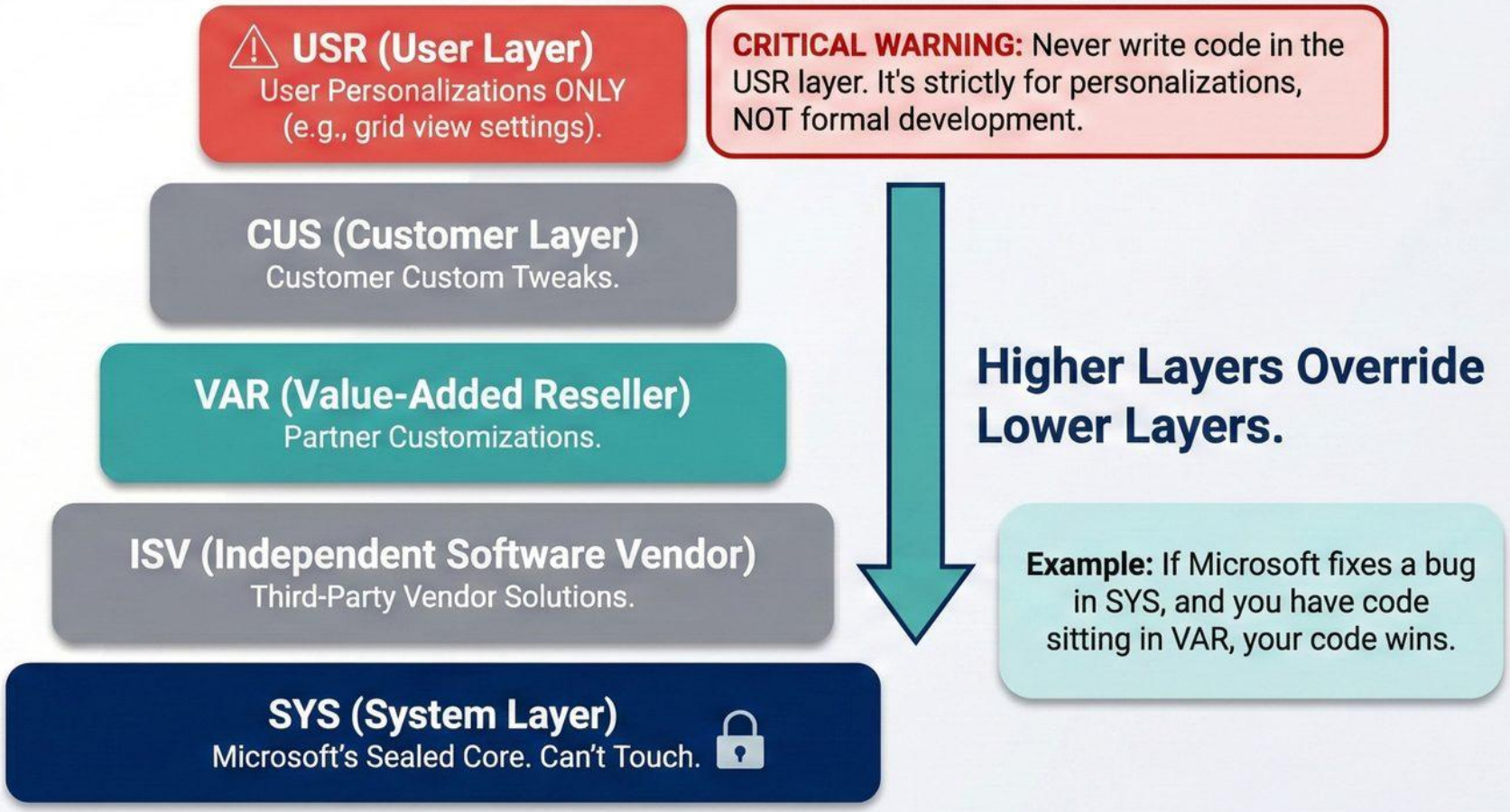


⚠ Safe development means extending at the top, not messing around at the bottom. Breaking the Platform breaks the entire system.

truestack.dev

Code Ownership and Layers

Understanding the Layer Cake Model in Dynamics 365 F&O



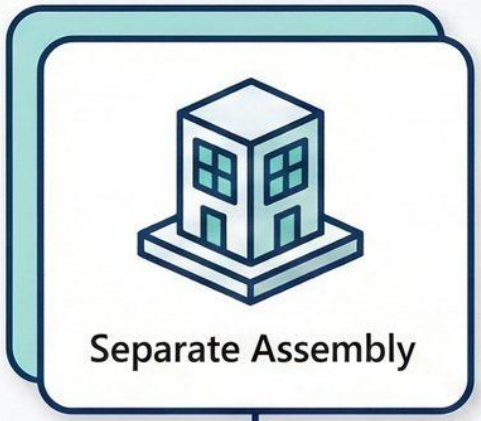
Why do these layers matter? Because this hierarchy keeps ownership crystal clear and prevents conflicts from happening.

truestack.dev

Creating a Model: Package Selection & Clean Architecture

Choose wisely for maintainable and upgradeable solutions.

CORRECT: Create New Package



- Clean Boundaries
- Your Code in its Own Assembly
- Smooth Upgrades
- Modular & Maintainable

Compiles Separately → Deploys Separately → Updates Separately

INCORRECT: Select Existing Package



- Mixed with Microsoft Code
- No Separation
- Upgrade Conflicts
- High Maintenance Risk

Code Tangled Together → Impossible to Separate → Upgrades = Nightmares

A new package ensures clean boundaries for independent compilation, deployment, and seamless updates.

Visual Studio
tidestack.dev

Understanding Model References: The Critical "Handshake"

Why your code needs to explicitly connect with standard application packages.

What Are References?



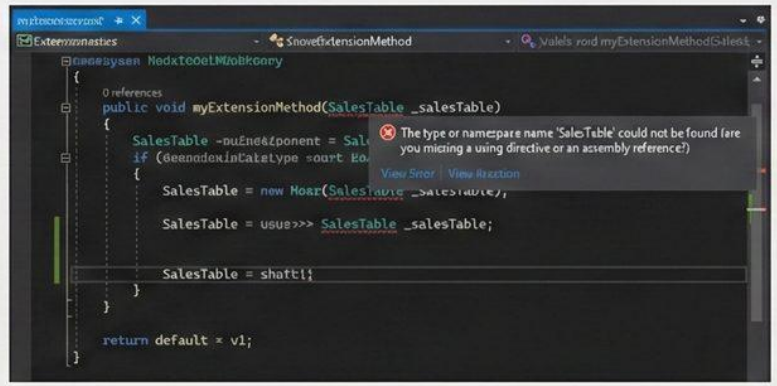
REFERENCE: The "Handshake" that allows the compiler to see Microsoft's standard code.

YOUR MODEL
(Custom Code)

STANDARD PACKAGE
(e.g., Application Suite, Application Foundation)

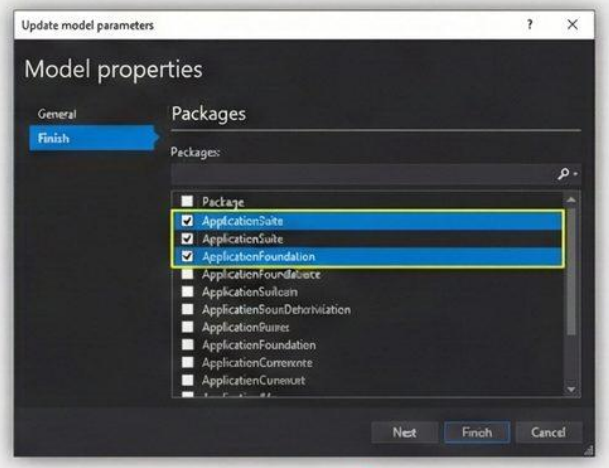
- Like looking up words in an available dictionary.
- Required to extend functionality or use standard components.

The Problem: Missing a Reference



SYMPTOMS:
IntelliSense stops working,
Red squiggly lines appear,
Build **FAILS.**

The Fix: Adding the Reference



SOLUTION: Go to model properties and add the required package reference. IntelliSense restores, Compilation **SUCCEEDS.**

TAKEAWAY: If debugging fails or IntelliSense breaks unexpectedly, always check your model references first.

truestack.dev

Common Development Mistakes & Their Impact on Upgrades

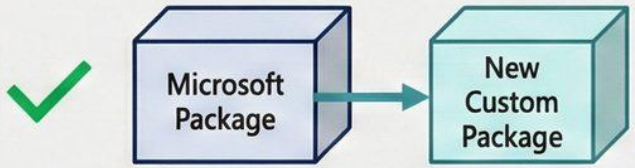
Highlighting critical errors that break isolation and impede system updates in Microsoft Dynamics 365 Finance & Operations.

Three Most Common Mistakes

Mistake 1: Adding Custom Model to Microsoft Package



Breaks isolation principle. Upgrades become impossible.



✓ **CORRECT:** Always create a new package.

Mistake 2: Writing Code in the USR Layer



USR is for user settings, NOT formal development

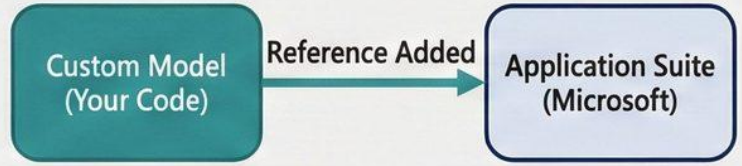


✓ **CORRECT:** Use VAR or CUS layers instead.

Mistake 3: Forgetting Package References

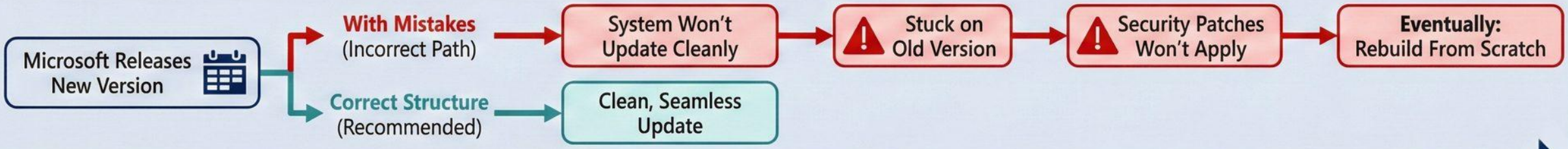


Code won't compile. IntelliSense fails. Debugging wasted.



✓ **CORRECT:** Ensure necessary references.

Why Do These Mistakes Matter? The Consequence: Broken Upgrade Paths



Get the structure right the first time—it saves significant pain and ensures future-proof systems

truestack.dev

Topic 1.3: Source Control and Branching

Recap Topic 1.2: Hierarchy, Layers, & References

Projects
Contain Code

Models
Define Features

Packages
Deploy to Servers

Layers & Ownership

Microsoft Layers	Ownership Layers
Platform	**VAR** = Partners (✓)
Foundation	**CUS** = Customers (✓)
Suite → Extend	**USR** = Actual Code (✗)

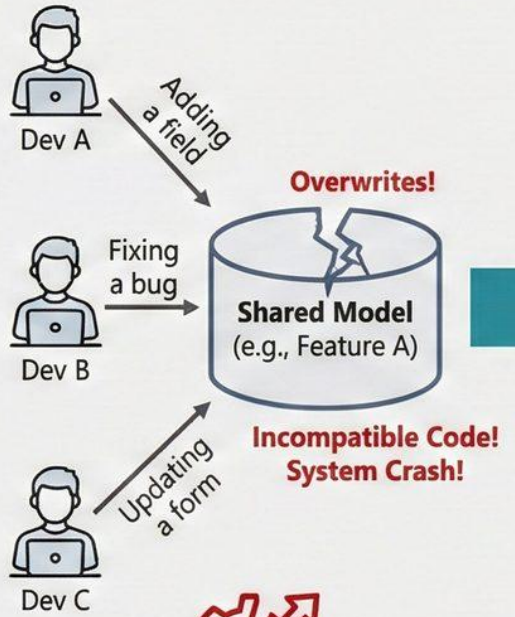


References: Handshake to see standard objects

Structure: Clean, Upgradeable, Maintainable

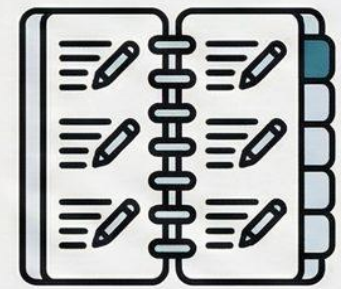
But here's the reality...

The Reality: Concurrent Development & The Solution



RESULT:
CHAOS & CONFLICT

Solution: Source Control
('Shared Notebook')



Check in Changes

Track History

Merge Work without destroying

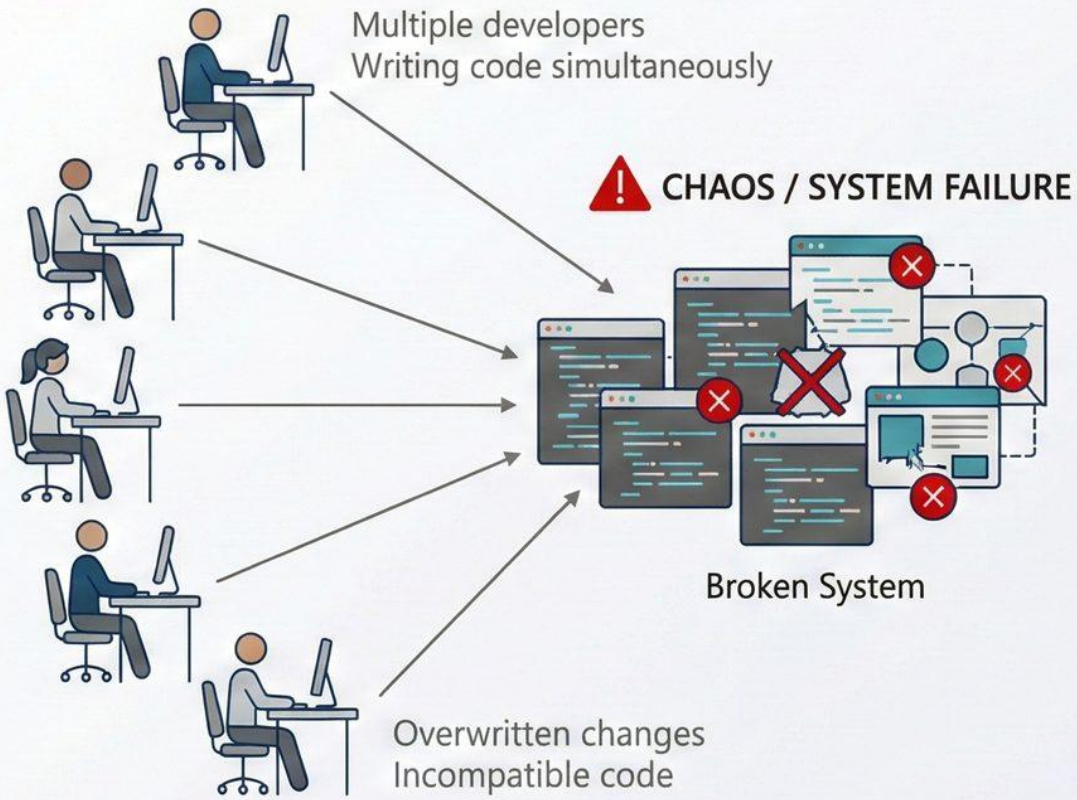
Outcome: Critical for Team Success & Collaboration

truestack.dev

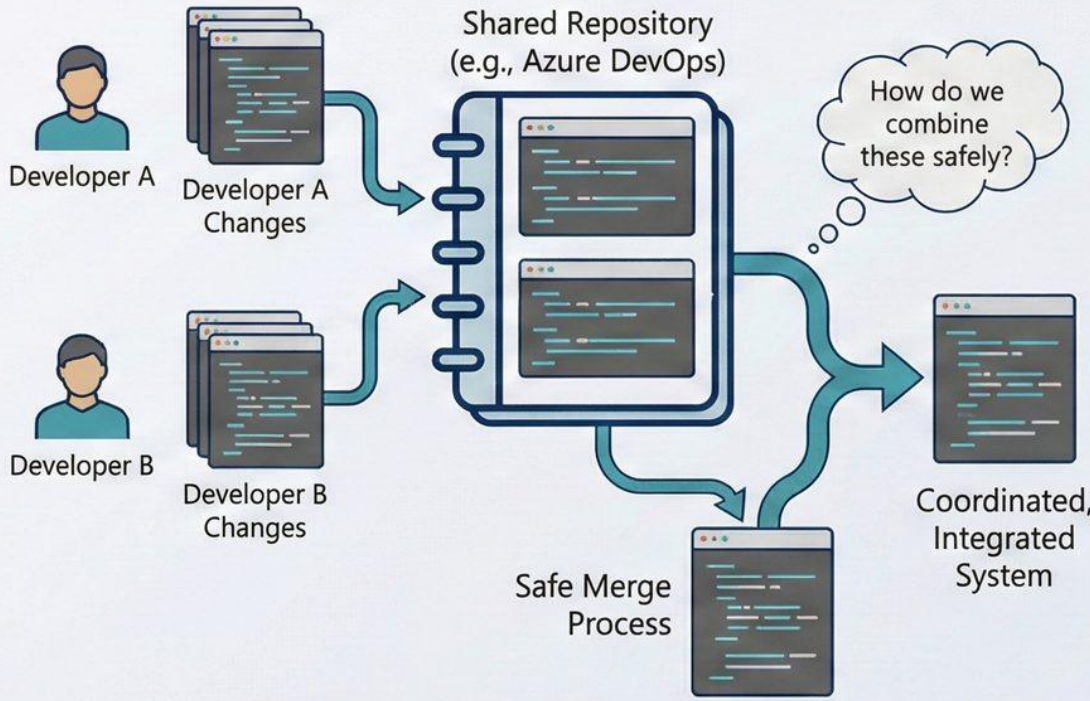
Topic 1.3: Source Control and Collaboration

Managing Simultaneous Development in Microsoft Dynamics 365 Finance & Operations

The Challenge: Simultaneous Development



The Solution: Source Control (Shared Notebook)



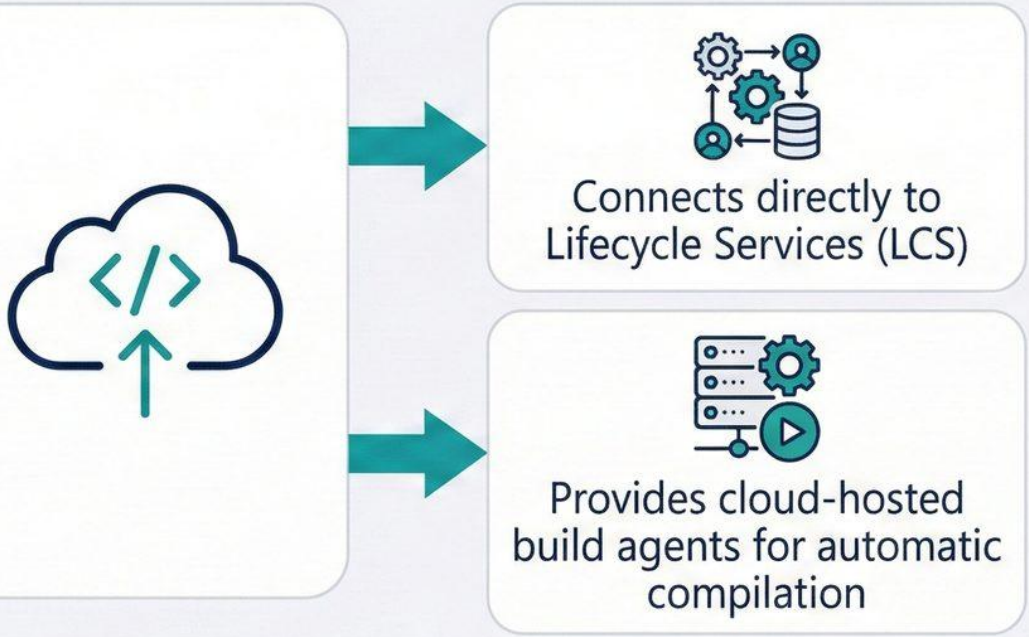
Source Control records, tracks, and merges changes from multiple developers to ensure collaboration without creating chaos.

truestack.dev

Source Control for Cloud-Hosted Dynamics 365 F&O

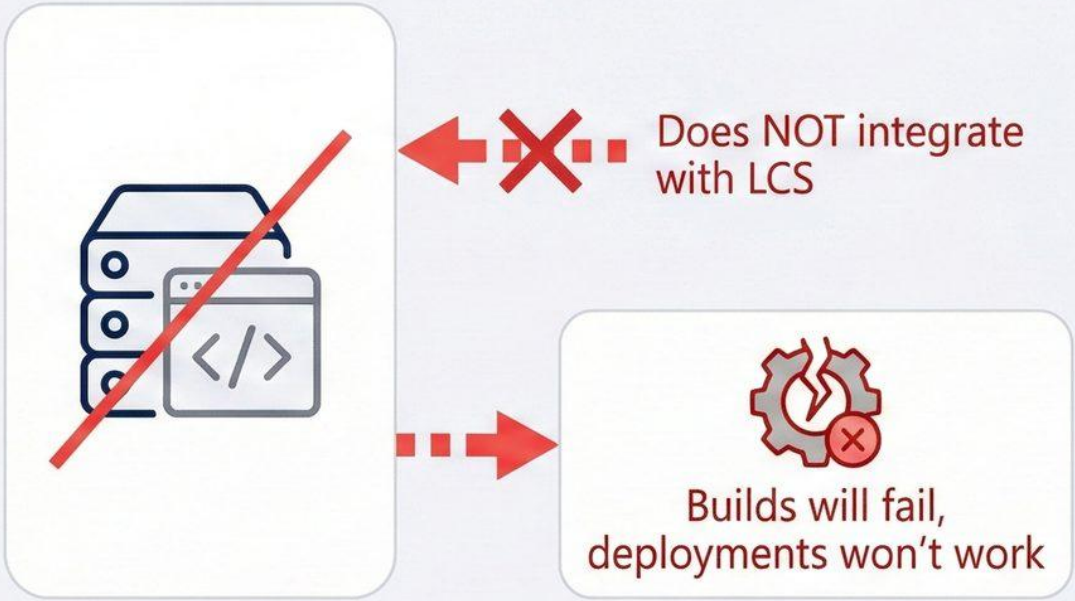
Mandatory Requirement for Cloud Environments

MANDATORY: Azure DevOps Services (Cloud)



Manages environments and deployments effectively.

UNSUPPORTED: On-Premises Tools (e.g., Local TFS, GitHub)



Setup is completely unsupported by Microsoft support.

For cloud F&O, Azure DevOps Services is mandatory for source control and supported operations.

truestack.dev

Setting up a Local Workspace



With Local Workspace (Correct Setup) ✅

- Visual Studio uses latest code from source control.
- IntelliSense works correctly.
- Compiler understands changes.
- Merges are successful.

Without Local Workspace (Incorrect Setup) ⚠️

- Visual Studio reads stale binaries on server.
- IntelliSense breaks.
- Compiler gets confused.
- Merges fail silently (unknown reasons).

Takeaway: A one-time, five-minute setup prevents hours of debugging later.

truestack.dev

Branching Strategy

Parallel Development and Controlled Code Flow

Concept: Parallel Universes

Developer A (Feature)

Independent work, no interference until merge.

Developer B (Bug Fix)

Three-Branch Strategy

Release Branch (Optional)
UAT & production staging. Very stable. Only approved code merges here.

Promote approved code

Main Branch
Integration point. Stable, tested code. Daily builds, team's combined code.

Merge when stable & tested

Dev Branch
Individual work (sandbox). Messy, daily changes. Breaking things is okay.

Code Flow: UP the chain (Dev → Main → Release)

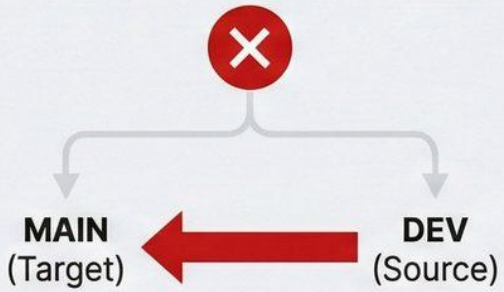
NEVER merge downward (Dev ← Main) to "keep current". Causes merge chaos!

Only pull from Main to fix specific conflicts.

truestack.dev

Code Merge Direction: Flow Towards Production

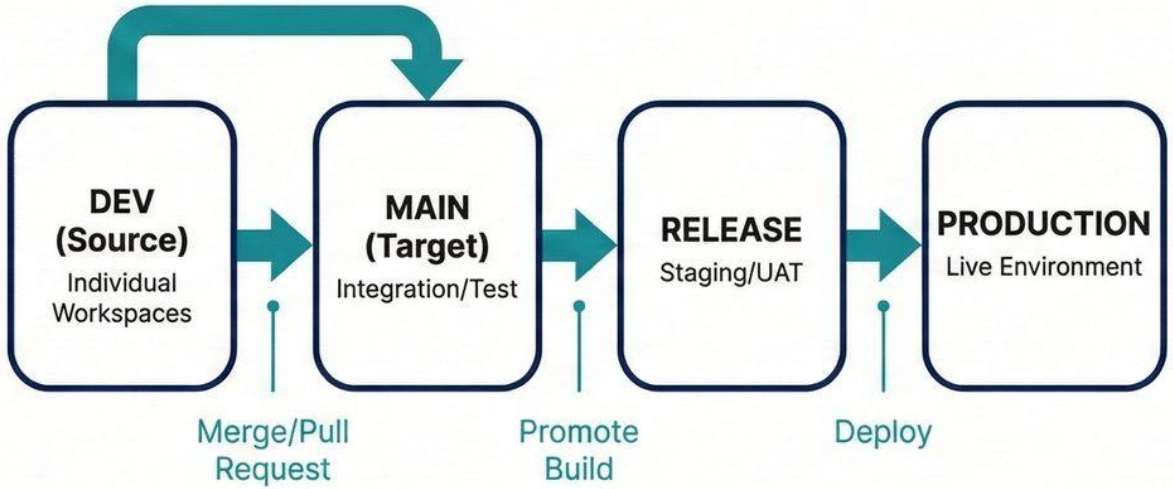
INCORRECT: Backward Merge = Catastrophe



Overwrites Main with incomplete Dev code

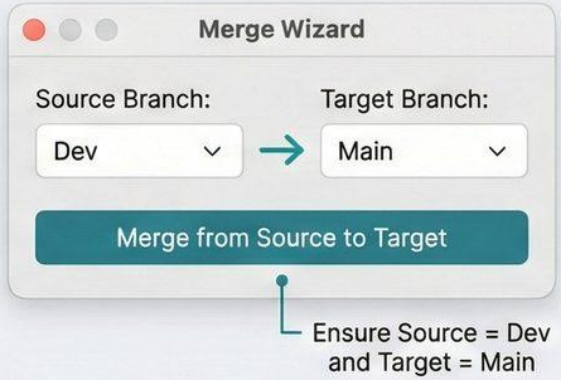
Never merge incomplete code backward into stable branches.

CORRECT: Flow Toward Production



Principle: Code always flows from Development toward Production environments. Never the reverse.

Merge Tool Terminology & Prompts



- Alternative Terms:
  - ✓ Merge into (Target)
  - ✓ Pull Request (towards Main)



Direction must always be toward production

truestack.dev

Visual Studio Merge Wizard for D365 F&O

Follow this strict sequence to merge code safely from Dev to Main branches.

1

Connect to Azure DevOps

Connect Visual Studio to your Azure DevOps project containing the D365 F&O code.

2

Map Target Branch

Map the target branch (e.g., Main) to a local folder for review.

3

Open Source Control Explorer

Access the Source Control Explorer window to see all branches.

4



Select Merge

Right-click the destination branch (Main) in Source Control Explorer and select Branching and Merging > Merge.

5

Dev → Main

Confirm Source & Target

In the wizard, confirm Source is 'Dev' and Target is 'Main'. Verify direction before proceeding.



6

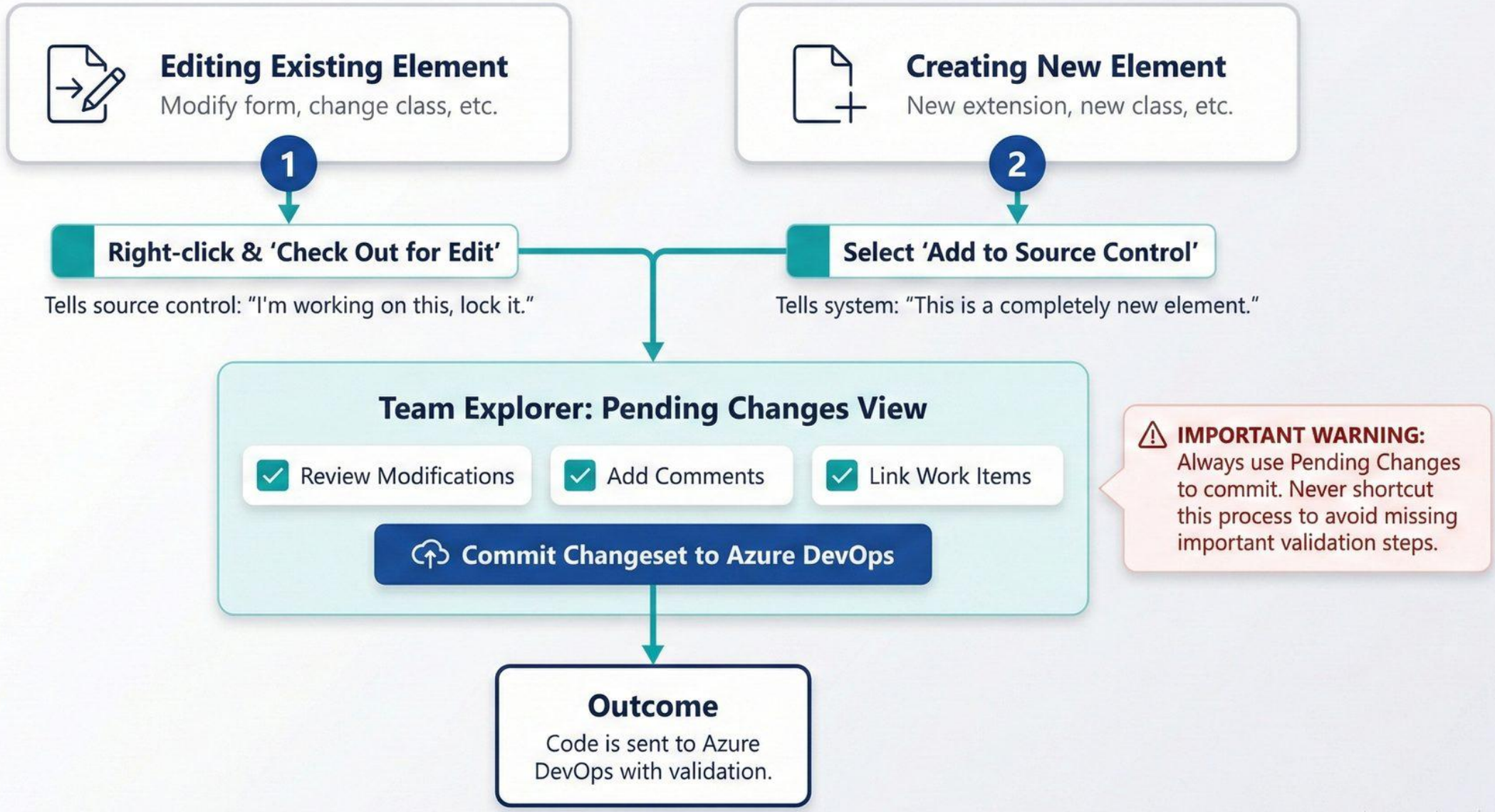
Resolve, Compile, Check In

If conflicts arise, pick versions, let compiler check, and check in the final merged result.

Strictly following this sequence ensures your code moves safely from Dev to Main.

truestack.dev

Adding Code to Source Control



truestack.dev

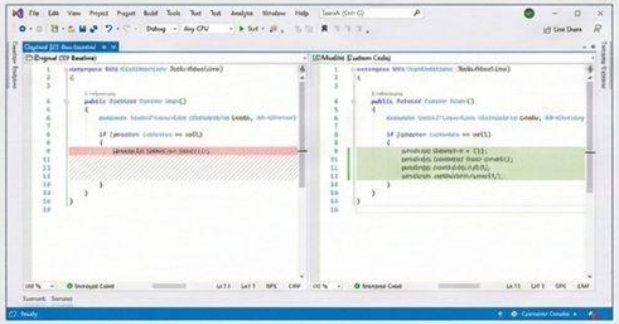
Choosing the Right Comparison Tool in Visual Studio for D365 F&O

Select the correct tool for the specific job to avoid missing important differences.

COMPARE CODE WITH BASELINE

For Comparing Custom vs. ISV X++ Code

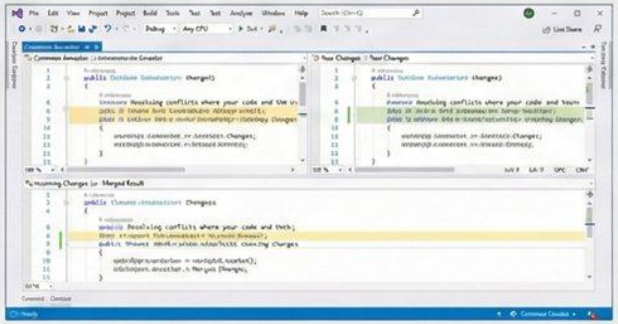
- ✓ **Scenario:** Comparing your custom X++ code against original ISV code to spot differences.
- ✓ **What It Shows:** Exactly what you changed versus the original baseline version.
- ✓ **Why It Matters:** Highlights specific code modifications without merging.



COMPARE AND MERGE CODE

For Resolving Merge Conflicts

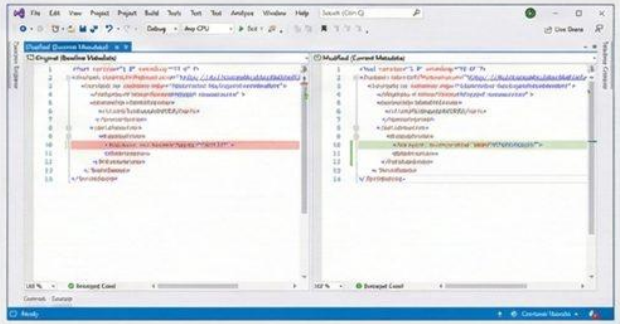
- ✓ **Scenario:** Resolving conflicts where your code and the team's code both changed the same section.
- ✓ **What It Shows:** Three versions side-by-side: common ancestor, your changes, and incoming changes.
- ✓ **Why It Matters:** Allows you to pick which version to keep.



COMPARE METADATA WITH BASELINE

For Form Metadata or UI Structure

- ✓ **Scenario:** Comparing form metadata, UI structure, or layout definitions.
- ✓ **What It Shows:** Compares the underlying XML element definitions, not the code itself.
- ✓ **Why It Matters:** Ensures correct comparison of form layout and structure.



Takeaway: Using the wrong comparison tool can lead to missed important differences and incorrect version acceptance, so always pick the right tool for the specific task.

truestack.dev

Handling Merge Conflicts in Visual Studio

Understanding the process and avoiding common mistakes during ‘Get Latest Version’

1. Action:
Get Latest Version

2. Visual Studio
Auto-Merge Attempt

No Overlap
(Silent Success)

Overlap
(Conflict Detected)

3a. Keep Working
(Seamless)

3b. Conflict:
Visual Studio Stops

INCORRECT ACTION (Anti-Pattern)

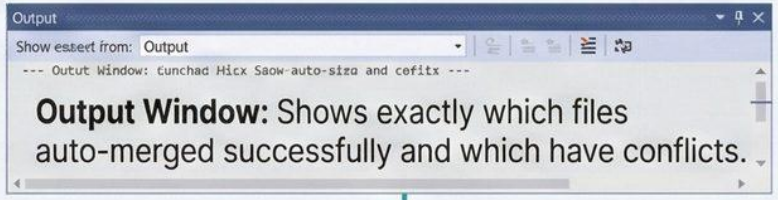
PANIC & CLICK “UNDO”

Mistake: Undo removes YOUR pending changes (uncommitted work). Auto-merged files remain merged and messed up.

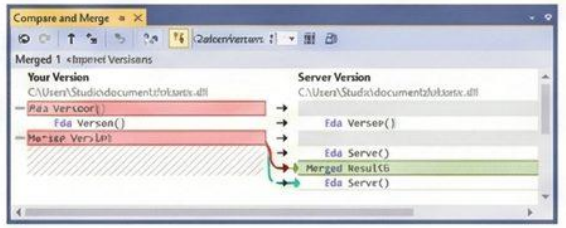
Outcome: Loss of Your Code

CORRECT ACTION (Recommended)

USE OUTPUT WINDOW TO DIAGNOSE



Open Conflicted Files in Compare and Merge Tool



Manually resolve conflicts, then try the merge again.

Outcome: Successful Merge (No Code Loss)

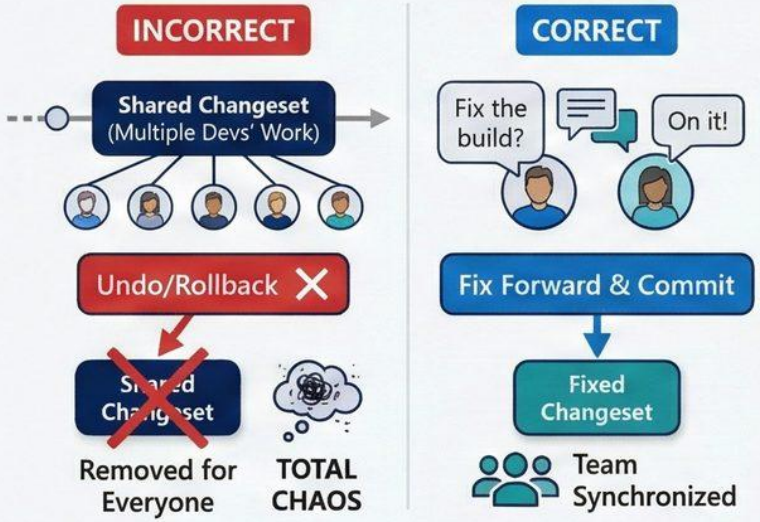
Takeaway: Instead of undoing, use the Output window and Compare tool to diagnose and manually resolve merge conflicts to prevent code loss.

truestack.dev

Three Critical Source Control Mistakes

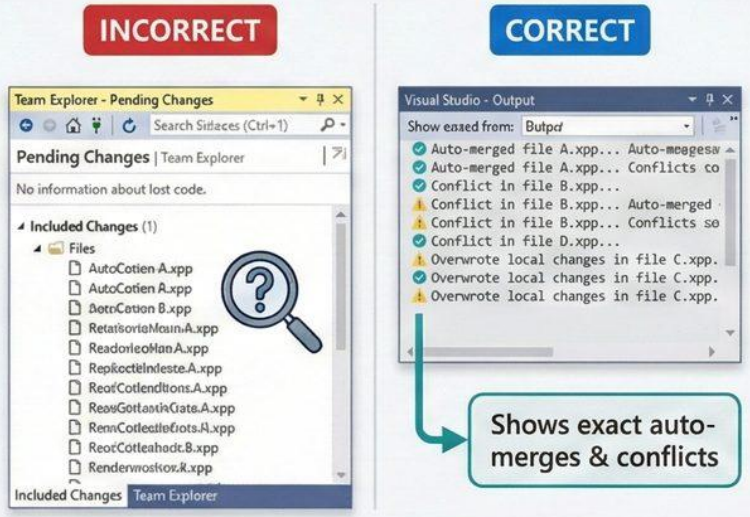
Common pitfalls developers make with source control in Microsoft Dynamics 365 Finance & Operations.

Mistake 1: Rolling Back Team Changesets



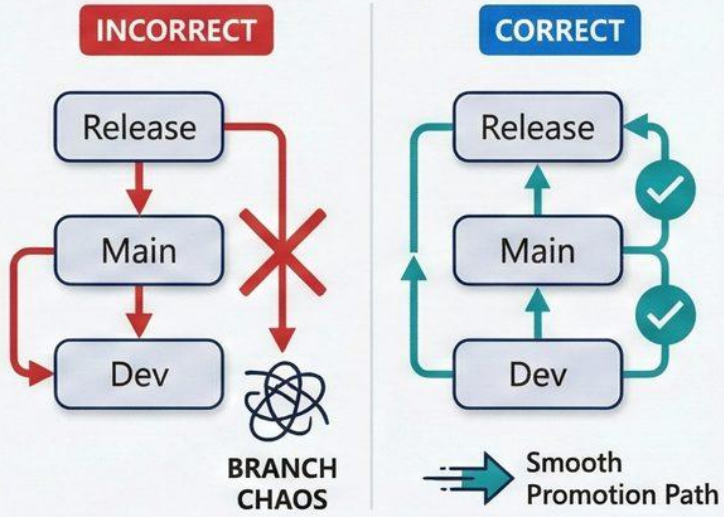
Never roll back shared work without permission; fix forward.

Mistake 2: Diagnosing Merge Loss via Pending Changes



Use the Output window, not Pending Changes, to investigate merge results.

Mistake 3: Merging Downward (Reverse Merge)



Always merge upward toward production (Dev → Main → Release).

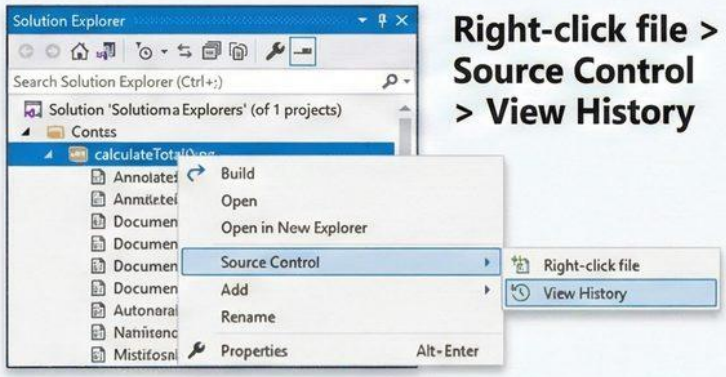
Avoid these mistakes to maintain a stable, collaborative source control environment.

truestack.dev

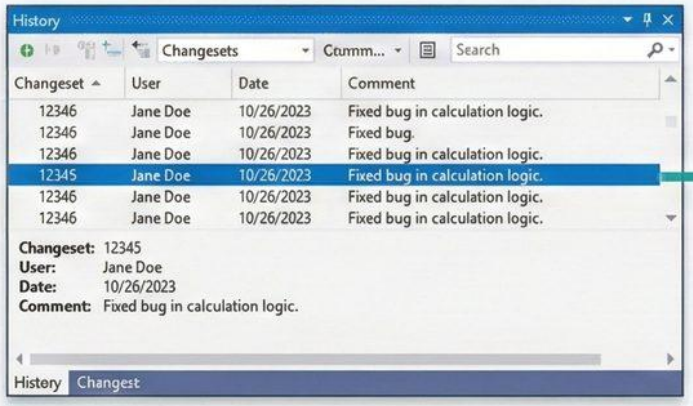
Tracking Source Control History in Dynamics 365 F&O

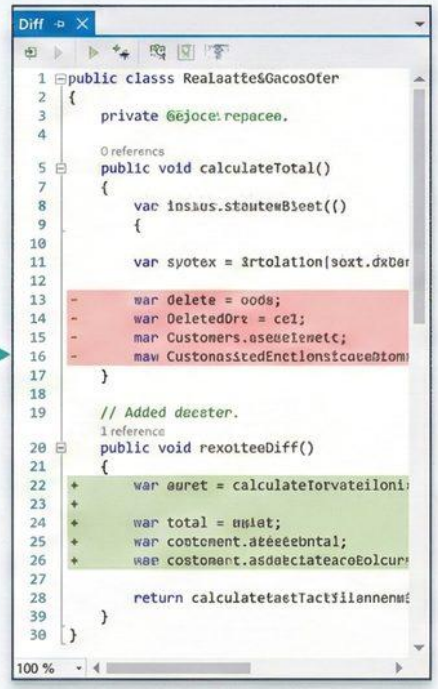
Find out who changed a specific line of code, when, and why using Solution Explorer.

Solution Explorer (CORRECT METHOD)



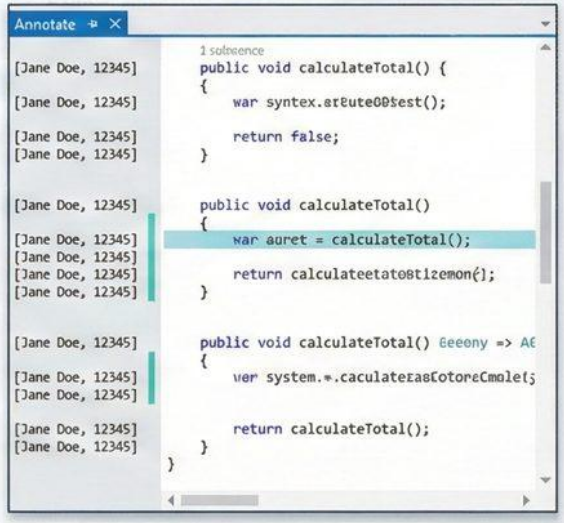
Right-click file > Source Control > View History



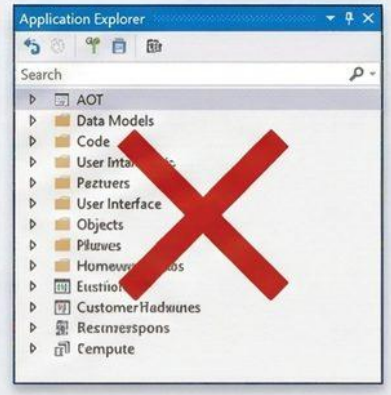


Exactly what lines changed.

Annotate View:
Shows authorship at the line level. Incredibly powerful for debugging to find who introduced a change and ask for context.



Application Explorer (COMMON MISTAKE)



Application Explorer shows **METADATA** currently deployed in the environment. It has **NO connection to source control** and **CANNOT** show history or who modified a form last.

Always use **Solution Explorer's 'View History'** and **'Annotate'** features for source control tracking; Application Explorer only shows current deployed metadata.

truestack.dev

Recap: Topic 1.3 - Azure DevOps & Version Control Setup

Azure DevOps & Workspace

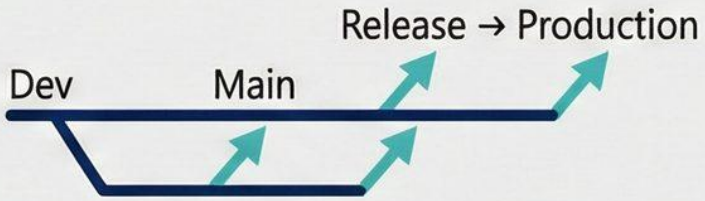


Mandatory for cloud F&O projects. Connects to LCS, provides cloud build agents.



Must set up **Local Workspace mapping: Metadata to PackagesLocalDirectory.** Ensures Visual Studio reads live source control, not stale binaries.

Branching, Merging, & Flow



Strategy: Dev (individual), Main (integration), Release (UAT). Code always flows upward.



Merge Sequence: Confirm source & target, resolve conflicts with correct tools, merge in correct direction.

Check-in & Diagnostic Tools



Check Out for Edit (existing code)
Add to Source Control (new code)
Review Pending Changes before commit.



Output window (auto-merge problems)
View History (traceability)

⚠ Application Explorer is not suitable for source control history.

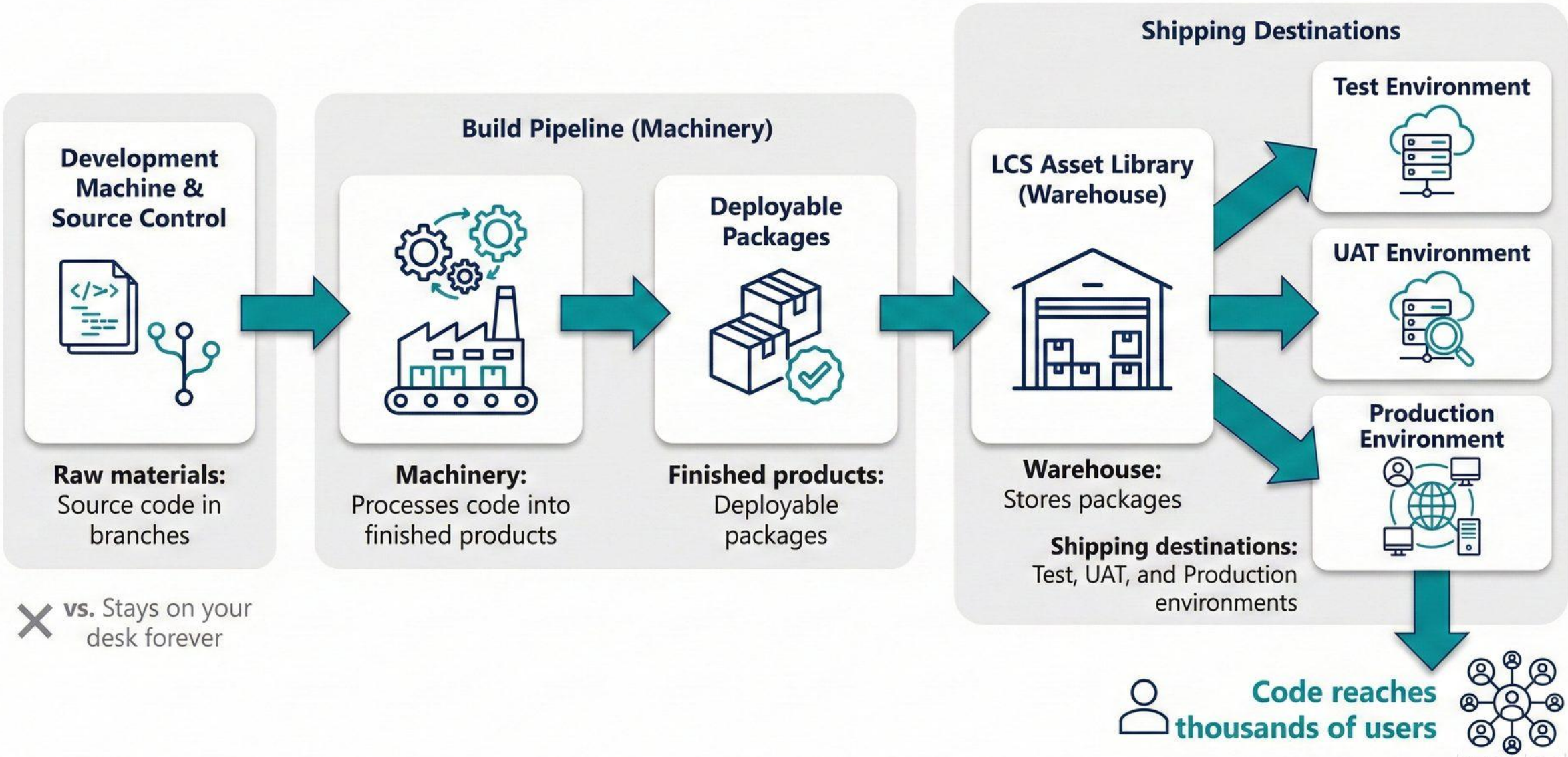


Next Step: Topic 1.4 - Build, Deployment, and LCS

Compiling code into packages and deploying to testing environments. Seeing how all these pieces actually fit together.

truestack.dev

Topic 1.4: Build, Deployment, and LCS



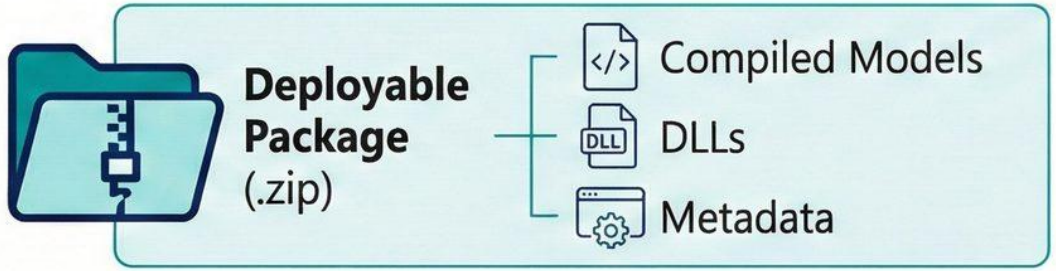
truestack.dev

Deploying Custom Functionality: The Valid Artifact

Dynamics 365 Finance & Operations consumes compiled code, not source files.

CORRECT: The Deployable Package

The only supported artifact for deployment to Sandbox and Production environments.



- Contains compiled models, DLLs, and metadata.
- Ready for installation.
- Required for non-developer environments.

INCORRECT: Invalid Artifacts

These artifacts cannot be deployed to Sandbox or Production.

- **Source Files (.axmodel, .axpp)**
  Reason: Sandbox/Production lack developer tools for compilation.
- **Managed Solutions**
  Reason: Designed for Power Platform & Dataverse, not F&O business logic.

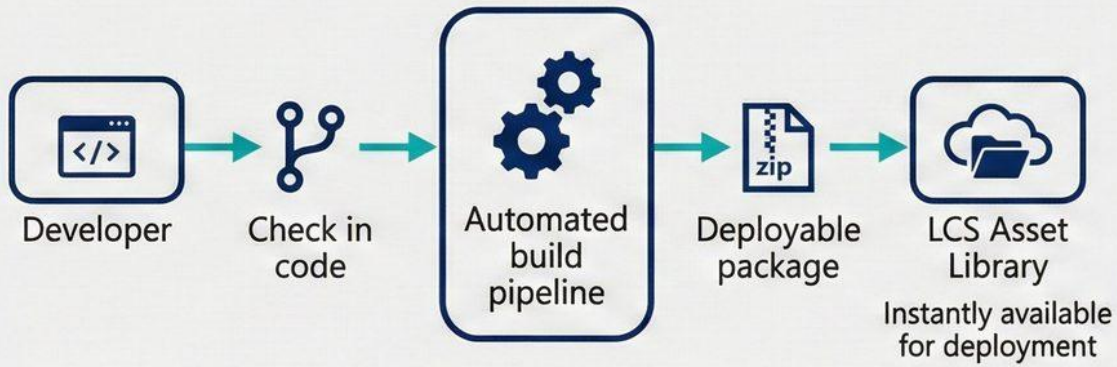
SINGLE RULE: The only valid artifact is a Deployable Package, in .zip format, compiled and ready to go.

truestack.dev

Creating a Deployable Package

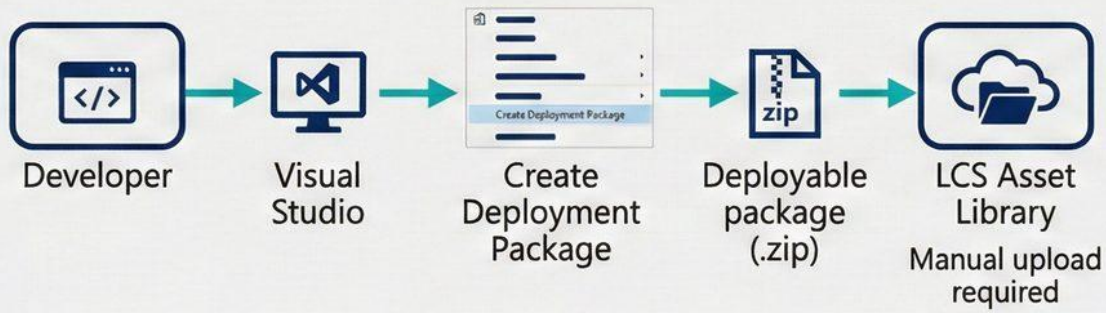
Two supported ways to create a deployable package.

1. Automated through DevOps (Best Practice)



- ✓ Repeatable
- ✓ Audited
- ✓ Removes human error

2. Manual through Visual Studio



- ✗ Error-prone in teams
- ✗ Inconsistent workspace states
- ✗ Unknown versions

Rule: Use DevOps pipelines for anything you actually care about. Only use Visual Studio packaging for isolated testing.

truestack.dev

MANUAL PACKAGE CREATION PROCESS (VISUAL STUDIO)

1

1. OPEN YOUR SOLUTION

The one containing your custom models and extensions.

2

2. NAVIGATE TO DYNAMICS 365 MENU

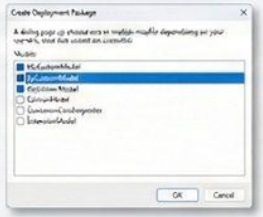
Right-click the project, select Dynamics 365, then "Create Deployment Package".



3

3. SELECT MODELS TO INCLUDE

A dialog pops up; choose one or multiple models depending on your scenario.



4

4. CLICK 'CREATE' AND WAIT

Compilation process can take a few minutes depending on code volume.

5

5. RETRIEVE GENERATED .ZIP FILE

Once complete, find the .zip in a standard folder like Downloads or Desktop on your VM.



6

6. UPLOAD TO LCS ASSET LIBRARY

Upload the retrieved .zip file to the Lifecycle Services (LCS) Asset Library.

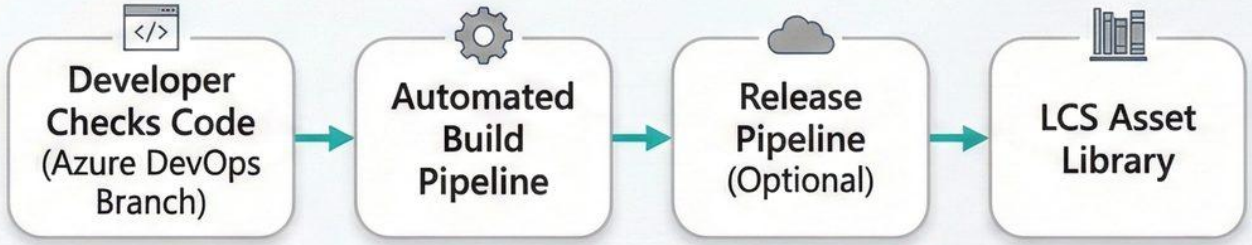
WARNING: PERSONAL TESTING ONLY

This manual process is fine for your own personal testing, but **never use it for team deployments**. Always use DevOps pipelines for production-grade ALM.

truestack.dev

Automated vs. Manual Packaging Deployment in Dynamics 365 F&O

AUTOMATED DEVOPS PIPELINE (ELEGANT APPROACH)



Automatically wakes up, compiles models, generates deployable package

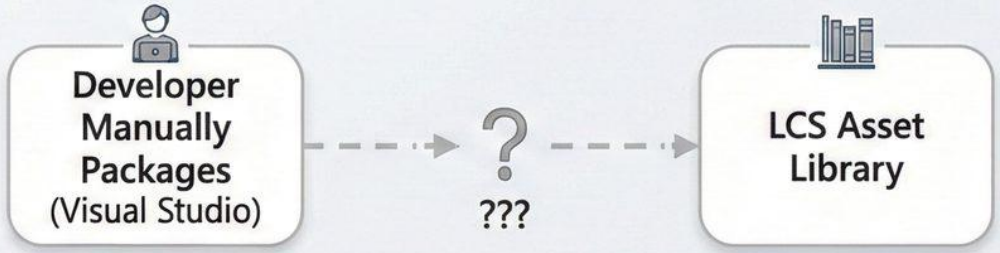
Automatically uploads package. Minutes to complete, hands-off.

Benefits

- Repeatability
- Audit Trails
- Traceability

Every deployment traceable to specific code commit & build run. "Which build was deployed?" -> Definitive Answer.

MANUAL PACKAGING (ERROR-PRONE)



Manual steps. Unknown origin (Dev or Main?), unknown version (10am or 2pm?)

Drawbacks

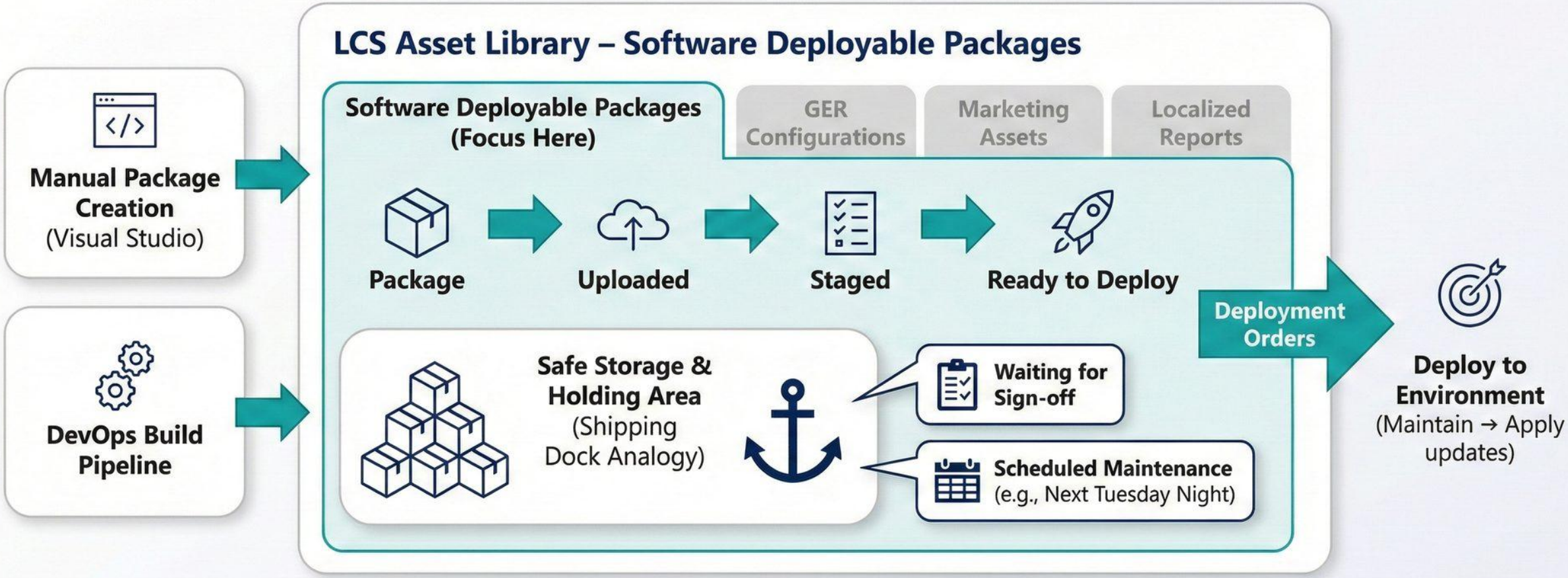
- Lost Trail Completely
- Uncertainty
- Lack of Accountability

Ambiguous: "Did you package from Dev or Main? Was it the 10am version or the 2pm version?" -> Nobody knows.

Use DevOps pipelines for anything important.

truestack.dev

LCS Asset Library: Your Deployable Package Warehouse

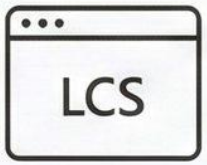


The LCS Asset Library acts as a secure warehouse where deployable packages are stored, inspected, and staged until **authorized for deployment.**

truestack.dev

Uploading a Package to LCS Asset Library

Step-by-step guide and critical distinction from deployment.



1 Open LCS & Navigate to Asset Library

Open LCS in browser and select the Asset Library.



2 Find 'Software Deployable Package' Section

Specifically choose this section, not Marketing Assets or others.



3 Click Upload & Select .zip File

Select your .zip file. Upload duration depends on file size.



4 Package Stored in Cloud Asset Library

Package is safely stored and accessible by project users.

CRITICAL POINT: Uploading vs. Deployment

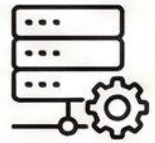


Uploading (Asset Library)

❌ NOT Deployment

- **Action:** "Put the package in the warehouse."
- **Status:** Stored safely in the cloud.
- **Outcome:** Package sits in Asset Library, inactive.

Separate Actions



Deployment (Environment)

✅ Actual Installation

- **Action:** "Take from warehouse & install in an environment."
- **Status:** Actively applied to a specific target.
- **Outcome:** New functionality is live in the environment.

Bottom Line: Uploading is preparation; deployment is the separate action of installing the package into an environment.

truestack.dev

Deployment: What's Supported and What's Absolutely Not

⚠ INCORRECT Approaches (Unsupported & Risky)

- ✗ Copying a .zip file directly to a server folder and hoping it installs itself.
- ✗ Using Remote Desktop to manually unzip and install files.
- ✗ Trying to use alternative tools or custom scripts found somewhere.
- ✗ Deploying source files like .axpp or .axmodel. (Environment lacks compilation tools).

WARNING: All these are deprecated, risky, and will absolutely break your system. Microsoft support will refuse to help.

✓ THE ONLY Supported Method (Using LCS)

1. Open the environment in LCS.
2. Navigate to **Maintain**.
3. Select **Apply Updates**.
4. Select **Apply Updates**.
4. Choose your package from the Asset Library.
5. Approve the deployment. ✓

LCS handles:

- ✓ Validation
- ✓ Installation
- ✓ Rollback capability
- ✓ Audit logging

This is the ONLY way to deploy in Microsoft-managed environments.

truestack.dev

LCS Deployment Sequence: Applying Updates to an Environment

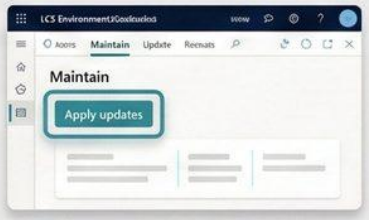
1

LCS Environment

Open target environment (e.g., Sandbox)

2

Maintain Section



Navigate to **Maintain** for updates & tasks.

3

Apply Updates



Select the actual deployment action.

4

Package Selection & Scheduling



Choose specific package from Asset Library.

Confirm & schedule (immediate or specific time).

5

Deployment Process



Duration: 30 mins - 2 hours

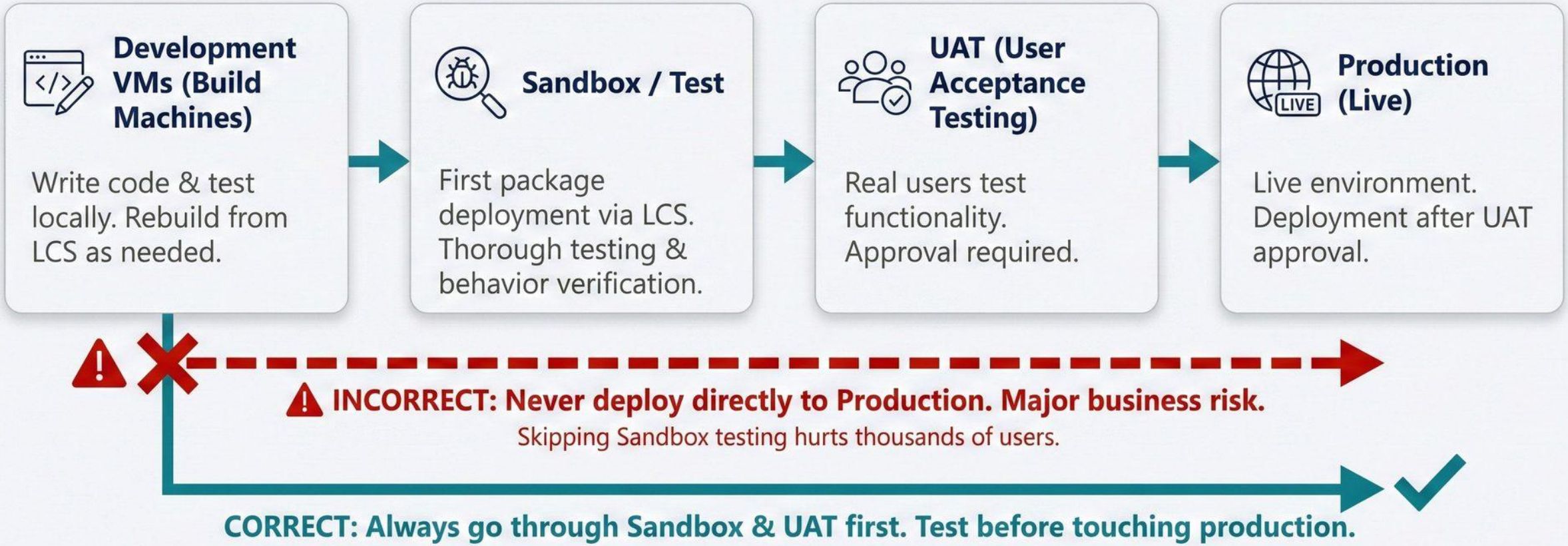
Environment unavailable; deploy during off-hours.

Custom code is LIVE in environment.

truestack.dev

Typical Environment Progression & Deployment Flow

Mandatory sequence for moving code from development to production.



The mandatory deployment sequence is: Dev development, create package, Sandbox testing, UAT user testing, then Production live.

truestack.dev

Diagnosing Post-Deployment Performance Issues in D365 F&O

Using Lifecycle Services (LCS) Monitoring Tools

Health Metrics

- CPU usage, memory consumption, request volume, overall health.
- If CPU is maxed out, indicates a clear performance problem.

SQL Insights

- Slow queries, table locks, missing indexes, blocking chains.
- Explains why a query is long-running (e.g., 5 min vs. 5 sec).
- Identifies missing indexes or N+1 query problems.

Activity Monitoring

- User sessions and request traces.
- Answers: What was a specific user doing when their screen was slow?

Troubleshooting Pattern & Environment Scope

Deployment issue? / Performance complaint? → 1. Check **Health Metrics** first. → If Okay → 2. Check **SQL Insights** next. → Still stuck? → 3. Run **System Diagnostics** on **Build VM** & upload logs to LCS.

Available only in Tier 2+ environments (Sandbox, UAT, Production). Build VM has limited diagnostics.

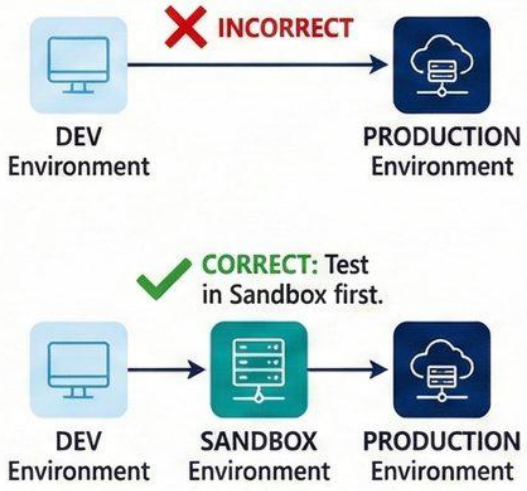
LCS tools act as a detective kit for effective troubleshooting.

truestack.dev

Four Critical Mistakes During D365 F&O Deployment

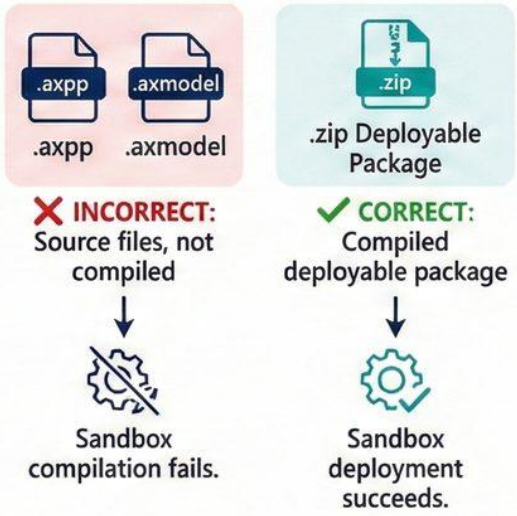
Avoid these common errors to ensure successful, stable deployments.

Mistake 1: Deploying Straight to Production



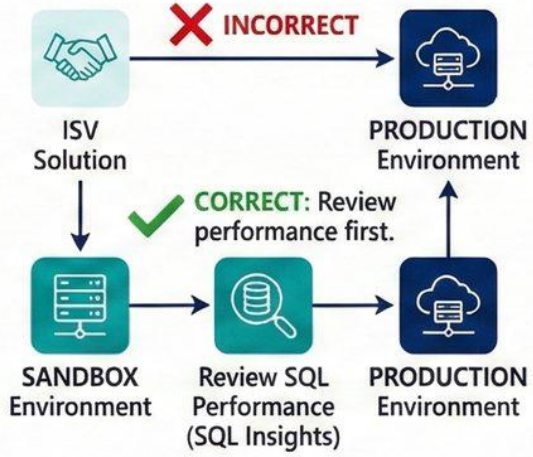
Direct deployment risks breaking fundamental features and causing widespread user downtime.

Mistake 2: Deploying .axpp or .axmodel Files



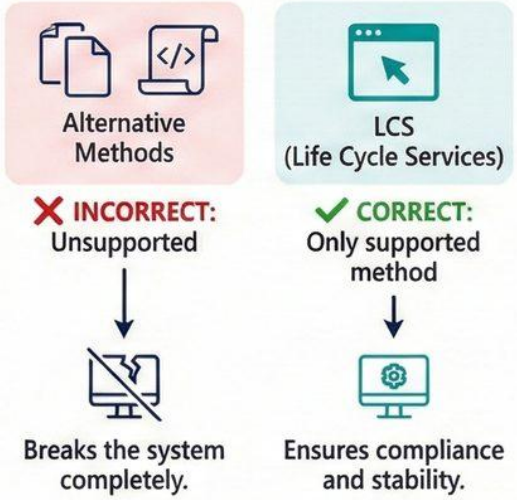
Sandbox environments lack compilation tools; always deploy compiled .zip packages.

Mistake 3: Installing ISV without SQL Review



ISV code can have inefficient queries; thoroughly review SQL performance in Sandbox.

Mistake 4: Bypassing LCS for Installation



LCS is the mandatory tool for deployments; using alternative methods will compromise the system.

Final Takeaway: Follow the supported path—test in Sandbox, use deployable packages, review performance, and always deploy via LCS.

truestack.dev

Topic 1.4 Recap: Dynamics 365 Deployment & Monitoring

Deployment Principles & Packaging

Not Source Files: Doesn't deploy .axmodel, .axpp

Deploys Compiled .zip Packages

Package Creation Methods:

- ✓ **Automated:** DevOps pipelines (Best Practice)
- ✓ **Manual:** Visual Studio (Quick Testing)

LCS Deployment Process

LCS Asset Library (Warehouse/Staging)

Deployment Sequence:

1. Open Environment →
2. Maintain →
3. Apply Updates →
4. Choose Package & Approve

Only Supported Method:
LCS "Apply Updates"
(Nothing else works properly)

Environment Progression Path

Dev → Sandbox → UAT → Production

Never skip any steps in the sequence

LCS Monitoring Tools

Health Metrics: General performance issues

SQL Insights: Database-specific problems

Activity Monitoring: User-specific issues

Now that code is deployed to Sandbox, we move to debugging and diagnostic tools in **Topic 1.5** to find and fix issues in running code.

truestack.dev

Topic 1.5: VS Tools and Debugging

The Problem: Bugs in Deployed Code

Code is deployed and running in Sandbox, but issues are reported.

- ✗ The form doesn't calculate the total correctly.
- ✗ The batch job fails silently.
- ✗ The price calculation is using the wrong value.

You can't fix what you can't see.

The Solution: Visual Studio Debugger

Visual Studio tools enable you to find and fix bugs effectively.

- ✓ Step through execution line-by-line.
- ✓ Watch variables change in real-time.
- ✓ See exactly where things are going wrong.

Navigate codebase, find bugs, inspect elements, and trace execution.

1. Navigate Codebase Efficiently



2. Find What You're Looking For



3. Inspect Forms and Extensions



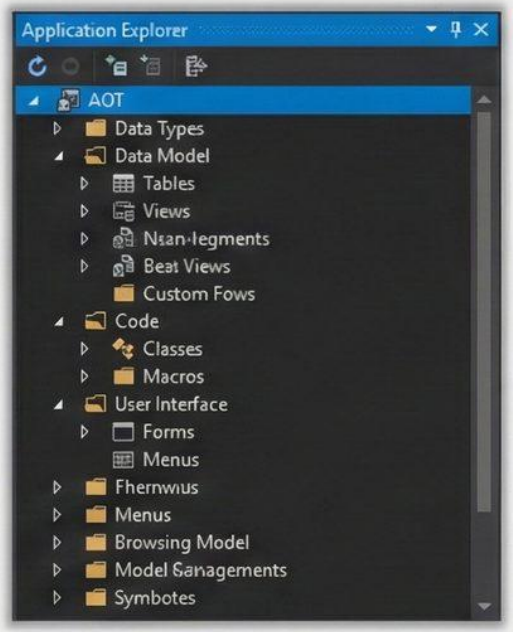
4. Use Debugger to Trace Execution



truestack.dev

Visual Studio Application Explorer: Traditional vs. Model View

Traditional View (Default)

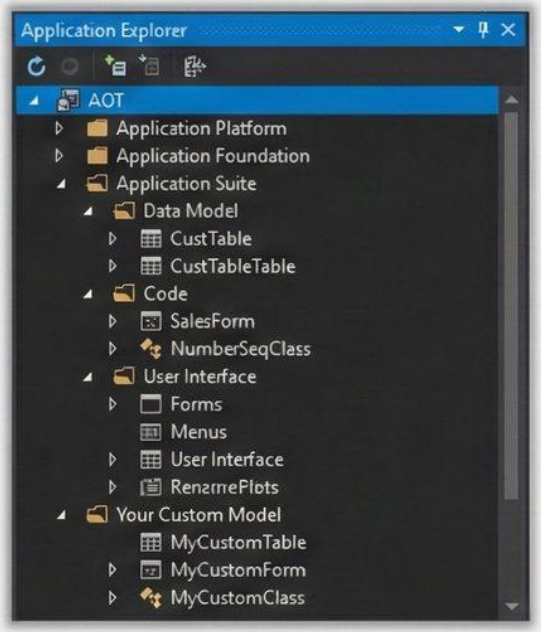


- Organizes elements by type (Tables, Forms, Classes, etc.)
- Difficult to identify model boundaries
- Confusing for architecture review and auditing

Switch View

Right-click 'AOT' node > Select 'Model View'

Model View (Recommended for Architecture)



- Organizes elements by originating model
- Clear separation of standard vs. custom code
- Instantly answers: "Which code did I customize?"
- Ideal for validating solution architecture

Switch between Traditional View and Model View depending on whether you are browsing by category or reviewing solution architecture.

truestack.dev

Visual Studio Metadata Search: Syntax & Power

Direct Metadata Access Syntax

Singular Element Type:

```
type:table
type:form
type:class
```

Sub-elements (comma separated):

```
type:table,method name:insert
```

Property & Code Filters:

```
name:SalesOrder
model:'Application Suite'
code:crosscompany
```

Combined Syntax Example:

```
type:form model:'Application Suite' name:SalesOrder
```

Critical Syntax Rules

✓ **USE SINGULAR FORMS**

```
type:table   (Correct)
```

✗ **AVOID PLURAL FORMS**

```
type:tables (Incorrect)
```

✗ **NO UNSUPPORTED OPERATORS**

```
like
contains   (Incorrect)
```

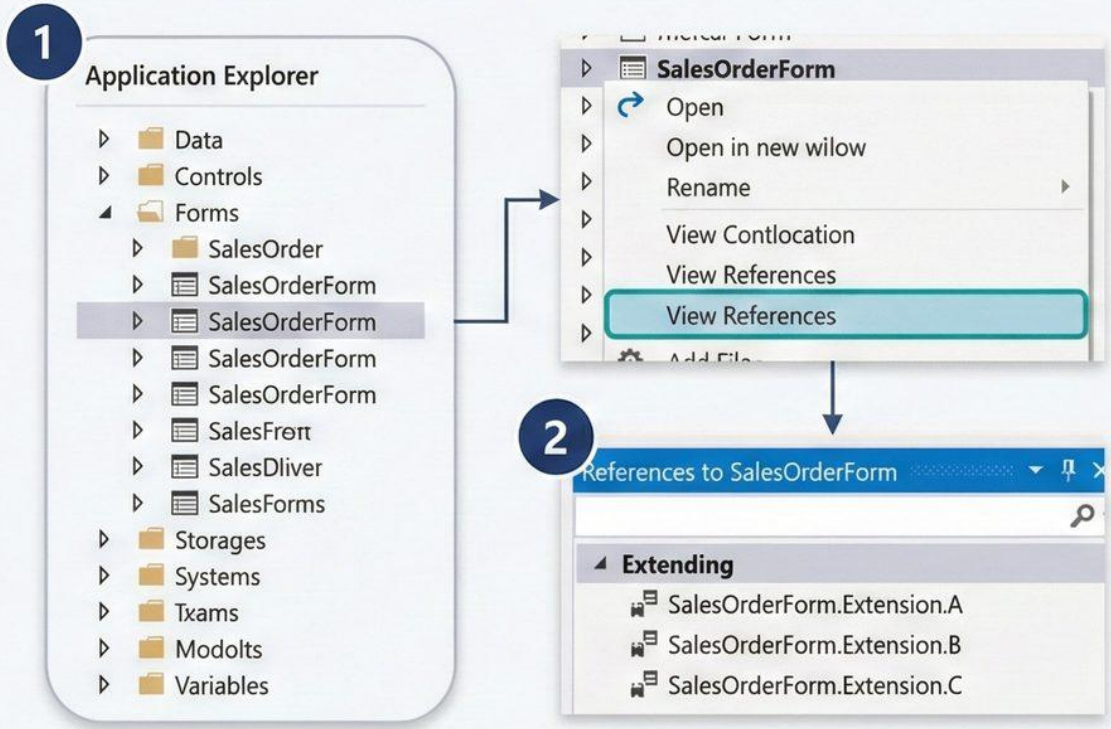
Master the supported, singular-based syntax to instantly find any metadata element.

truestack.dev

Reviewing Customized Forms and Extensions

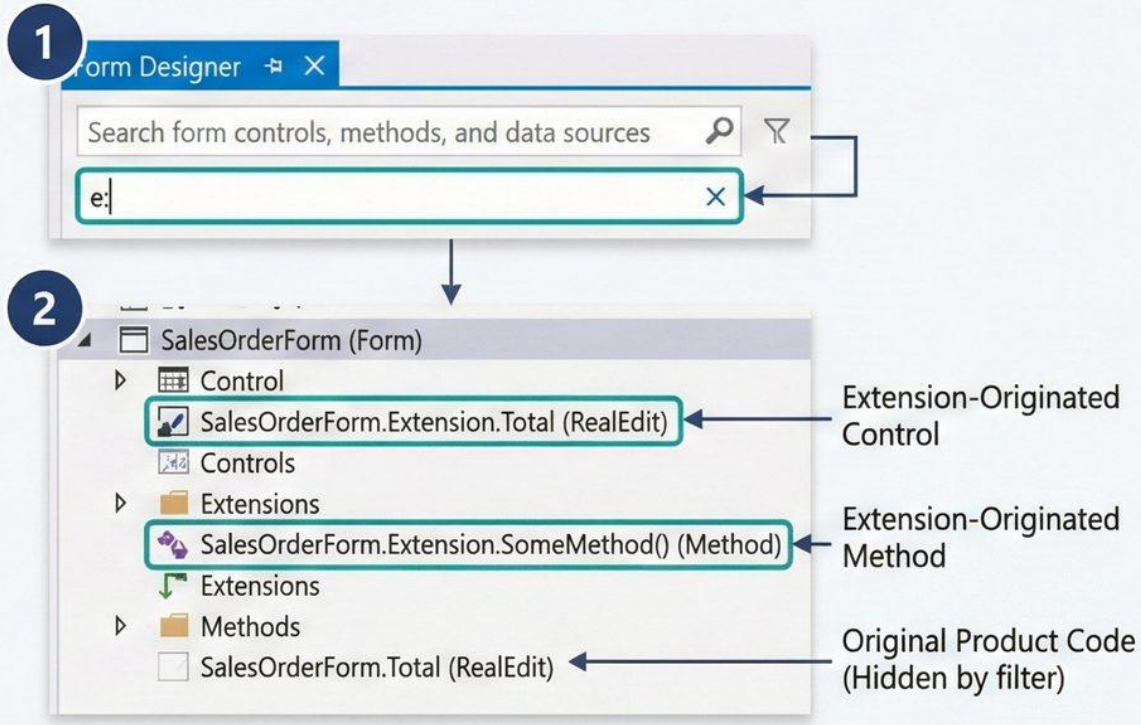
Two methods to audit extension impact in Visual Studio.

Approach 1: View References in Application Explorer



Lists all extensions affecting the form.

Approach 2: Form Designer and 'e:' Filter



Filter 'e:' displays only extension-originated elements.

Understanding these approaches is critical for auditing customization impact and distinguishing between original and extension code.

truestack.dev

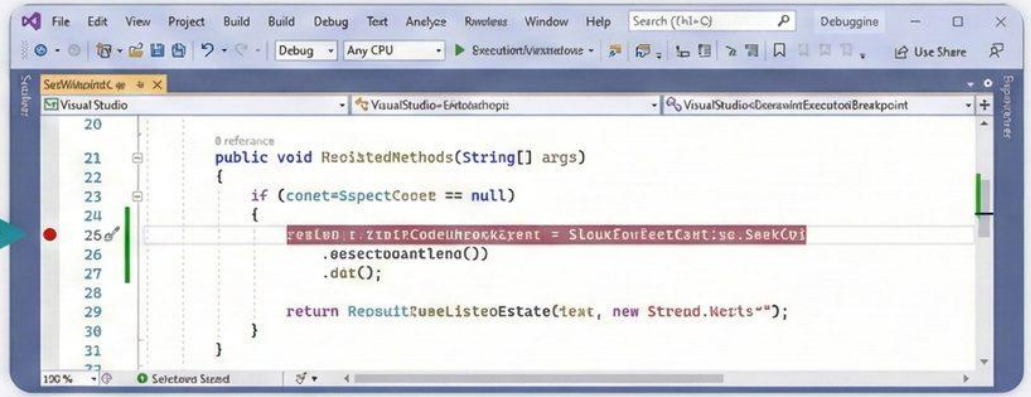
Debugging Foundation: Breakpoints

The foundation of debugging—pausing execution to inspect code state.

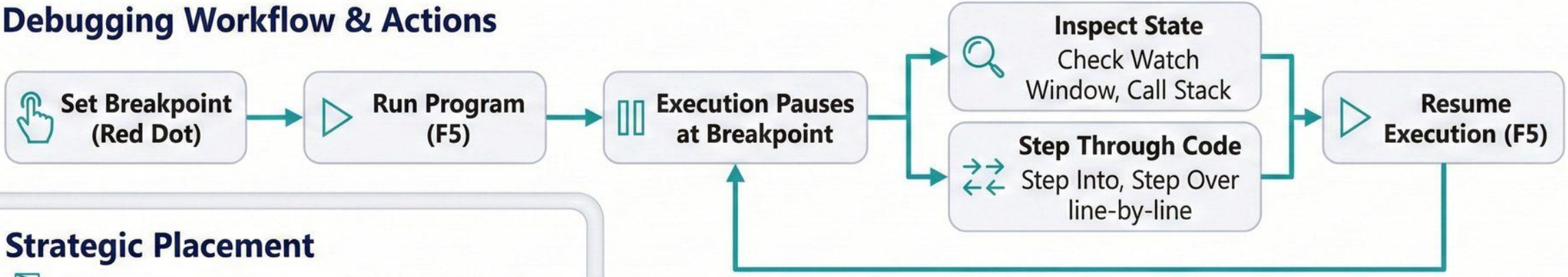
What is a Breakpoint?

- A directive to Visual Studio to stop execution at a specific line.
- Pauses execution completely—you are "frozen in time" at that moment.
- Allows inspection of variables and call stack.

Click in margin to set. Red dot indicates breakpoint.



Debugging Workflow & Actions



Strategic Placement

- Entry points of suspicious methods
- Before calculations
- After database queries execute

Trace code flow and find issues without print statements by setting multiple strategic breakpoints.

truestack.dev

Debugging with the Watch Window

Your Personal Variable Inspector for Real-Time Insights

Concept & Functionality

```
10  public void CalculateTotal(decimal unitPrice,
    int quantity)
11  {
12      decimal totolPrice = 0;
13      // Step Into: Debugger active here
14      totalPrice = unitPrice * quantity;
15  }
```

As you **Step Into** or **Step Over** code, the **Watch Window updates** in real-time, showing exactly what's in your variables at each step.

Practical Example: Price Calculation Bug

Watch 1

Name	Value	Type
unitPrice	100	decimal
quantity	5	int
totalPrice	0 *Initially empty or default value*	decimal

Initial Step (Before Calculation)

Execution Steps & Updates

Name	Value	Type
unitPrice	100	decimal
quantity	5	int
totalPrice	475	

After 'Step Into' (Bug Revealed)

UNEXPECTED VALUE:
Should be 500. Bug visible!
(Tax likely subtracted incorrectly)

Variables don't lie—the Watch Window shows you exactly what your code is doing, making bugs visible that you'd otherwise be guessing at.

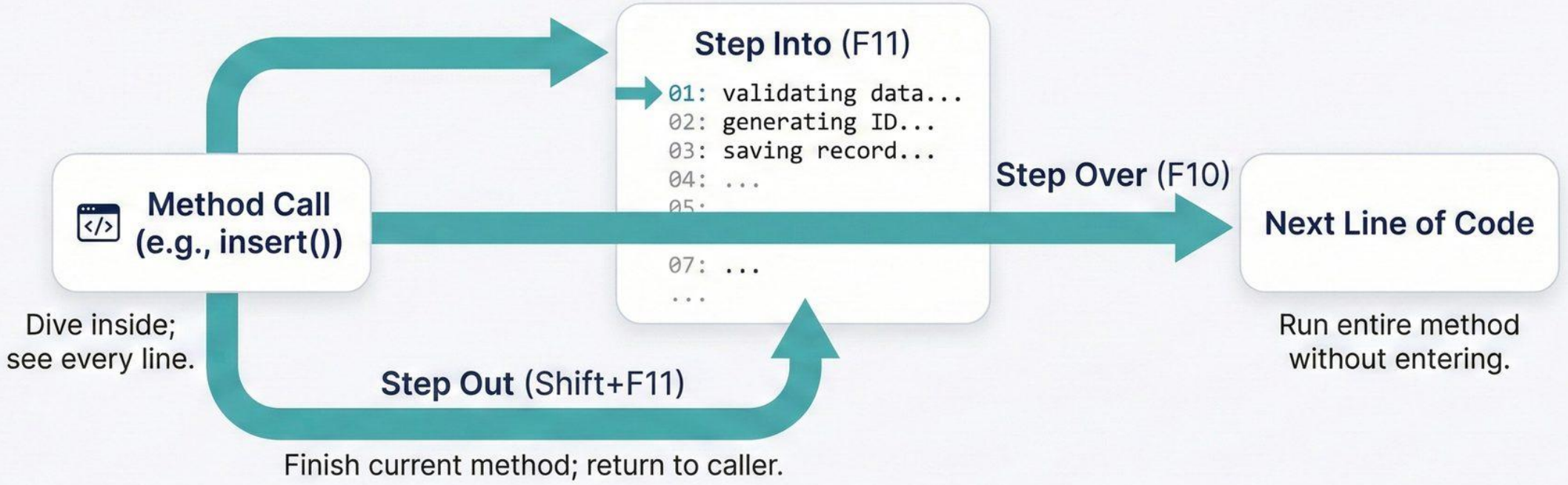
How to Add Variables
Right-click variable in code > Select "Add Watch".

How to Add Variables
Type variable name directly into the Watch Window.

truestack.dev

Visual Studio Debugging Commands: Your Code Time Machine

Navigate execution flow to isolate issues efficiently.



Remember the Pattern:

- Step Into: Investigate suspicious logic.
- Step Over: Trust the code, move past.
- Step Out: Realized you went too deep.

truestack.dev

The Complete Debugging Workflow

Practical Steps for Finding and Fixing Bugs in D365 F&O

1 Identify the Problem Area

Narrow down the issue based on user reports. E.g., 'Wrong total on sales orders' leads to SalesOrderTotal.CalculateTotal method.

2 Set Breakpoints

Open the method and place breakpoints at the entry point and return statement.

```
public class SalesOrderTotal
{
    public decimal CalculateTotal(SalesId salesId)
    {
        // Breakpoint here
        decimal total = 0;
        // ... logic ...
        // Breakpoint here
        return total;
    }
}
```

3 Start Debugging

Press F5 to run the program normally until execution hits a breakpoint and pauses.

4 Inspect Watch Window

Check all variable values at the paused state. Are they what you expected?



5 Step Through Execution

Use 'Step Into' to go inside sub-methods, 'Step Over' to skip trusted calls, and 'Step Out' to leave methods. Watch for unexpected changes.

Step Into (F11) Step Over (F10) Step Out (Shift+F11)

6 Fix and Validate

Once identified, fix the issue, remove breakpoints, rebuild, and test again.

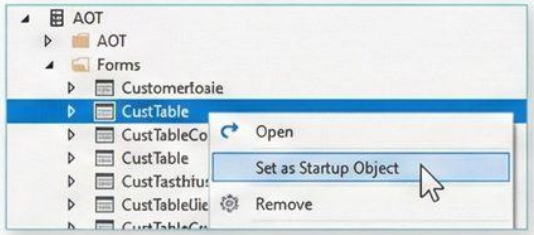
This workflow transforms uncertainty into actionable insights by allowing developers to inspect the exact state of execution.

truestack.dev

Faster Form Testing: The Startup Object Shortcut

1

Find Form & Set as Startup Object

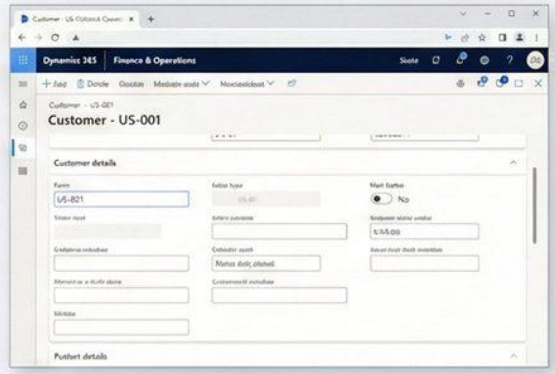


2

Press F5



3



1

Find Form & Set as Startup Object

In Application Explorer, right-click your form and select 'Set as Startup Object'.

2

Press F5

Visual Studio builds the solution, spins up a debugging session, and launches a browser.

3

Form Appears Immediately

The browser goes straight to your form. Skip all menu navigation.

Outcome: Fastest Testing Workflow

- Manually test functionality, click buttons, enter data, and see validation works.
- Add breakpoints to form code before F5 for deeper debugging if needed. No time wasted navigating menus.

truestack.dev

The Visual Studio Output Window

Information Dashboard

Shows “behind the scenes” activity: method entries, database queries, warnings, and errors when code runs.

How to Open

Go to View > Output or press Ctrl+Alt+O.

Debugging Background Processes

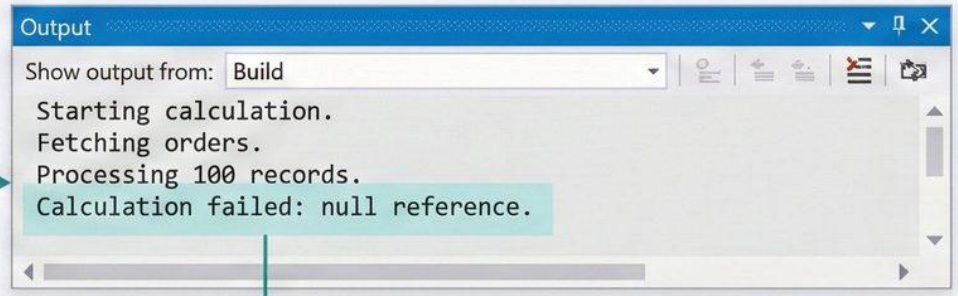
Invaluable for batch jobs, web services, and processes with no visual interface.

Add Diagnostic Output

Add diagnostic messages to your batch job code.

```
Debug.WriteLine('About to calculate total');
```

Run Batch Job



Shows execution flow and exactly where execution stopped and why.

The Output Window is your window into system execution for background processes and batch jobs where direct visibility is not possible.

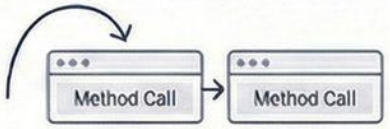
truestack.dev

Three Common Debugging Mistakes & Best Practices

1. Using Step Over vs. Step Into

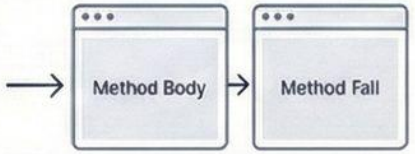
INCORRECT

Pressing F10 (Step Over) at call site, skipping method entirely. Bug remains hidden.



CORRECT

Pressing F11 (Step Into) to enter the method and inspect execution.



2. Running Without Breakpoints

INCORRECT

Pressing F5 (Run) with no breakpoints. Code executes too fast; nothing learned.



CORRECT

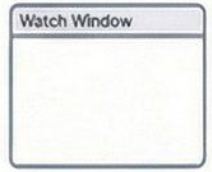
Set breakpoints at suspicious locations before running to pause execution.



3. Skipping the Watch Window

INCORRECT

Stepping through code without watching variables. No visibility into value changes.



CORRECT

Add variables to Watch Window to observe values changing at each step.



Always inspect, watch, and utilize your breakpoints for effective debugging.

truestack.dev

Debugging Real-World Scenarios

Scenario 1: Form Calculation Error

1. **Set Breakpoint:** `CalculateTotal` method entry.
2. **Run & Trigger:** Press F5, interact with form.
3. **Inspect Variables:** Watch Window (unitPrice, quantity, discount).

```
unitPrice: 100
quantity: 5
discount: 0
```

Identify Issue: Step Into, total becomes 475 (expected 500).

Fix Logic: Correct tax calculation.

Scenario 2: Silent Batch Job Failure

1. **Add Debug Logs:** Insert `Debug.WriteLine` calls.

```
Debug.WriteLine("Batch started");
Debug.WriteLine("Fetching orders");
Debug.WriteLine("Processing record 100");
Debug.WriteLine("Calculation complete");
```

2. **Run Batch Job.**
3. **Check Output:** Watch Output Window.

```
Batch started
Fetching orders
[Stops]
```

4. **Trace Execution:** Execution stopped after 'Fetching orders'.
5. **Diagnose:** Likely query error or connection issue.

Scenario 3: Incorrect Query Results

1. **Set Breakpoint:** After query execution.
2. **Run Code.**
3. **Inspect Result Set:** Watch Window.

```
resultset.Count: 100
```

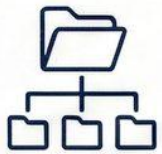
4. Analyze Data: 100 rows returned (expected 10).

Fix Query: Check and correct WHERE clause.

These scenarios cover common debugging situations encountered during development.

truestack.dev

Recap: Topic 1.5 – Essentials of Development and Debugging



Application Explorer by Model

- Reorganizes Explorer by model.
- Clarity on code location.



Metadata Search Syntax

- Find elements quickly.
- Syntax: Singular types, commas, property filters.



Finding Extensions

- Affecting a form via View References.
- Use `e:` designer filter.



Stepping Commands

- **Step Into**: Inspect logic.
- **Step Over**: Skip trusted.
- **Step Out**: Exit stack.



Debugging Tools & Workflow

- **Watch Window**: Variable states.
- **Breakpoints**: Pause execution.
- **Workflow**: Identify, set, run, inspect, step, find bug.



Launching Forms Directly

- Set as Startup Object.
- Press F5 to launch.

Next: Topic 2.1: EDTs and Enums – The Foundation of Data Types.

Summary: We've mastered navigating, searching, extending, and debugging D365 F&O code; now we build the foundation with data types.

truestack.dev

Topic 2.1: EDTs and Enums

Learning to choose, reuse, create, and control data quality with Extended Data Types and Enumerations.

Extended Data Types (EDTs)

Raw Primitives
(Naked)

- ‘string’
- ‘int’

EDT Wrapper

Metadata: Label (e.g., ‘Customer Account’)

- **Primitive Data** ‘string’ or ‘int’
- Help Text
- Format/Mask
- Relations

‘CustAccount’ (EDT)

Immediately identifiable & consistent

Enums (Enumerations)

Problem: Plain Strings

~~‘Good’~~
~~‘Medium’~~
~~‘Risky’~~
~~‘Excellent’~~ (Typo)
~~‘good’~~ (Case mismatch)

Defined Enum List

- ✓ Good
- ✓ Medium
- ✓ Risky

Prevents typos & self-documenting

Choose Right Data Type **Reuse/Create EDTs** **Enums for Data Quality**

truestack.dev

Microsoft Dynamics 365 Finance & Operations

Primitive Types vs. Extended Data Types (EDTs)

Primitive Types

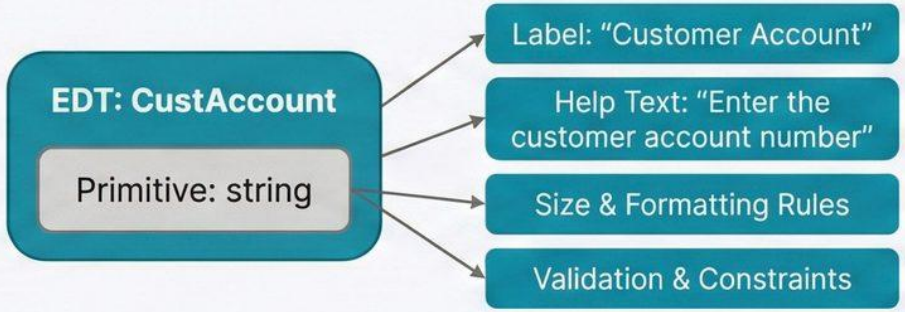
Basic, holds a value, no metadata.

- string
- int
- real
- date

- **No clarity:** A ‘string’ parameter could be anything (customer account, name, address).
- **Lack of consistency** and reusability across forms and tables.

Extended Data Types (EDTs)

Wraps primitive, adds metadata.

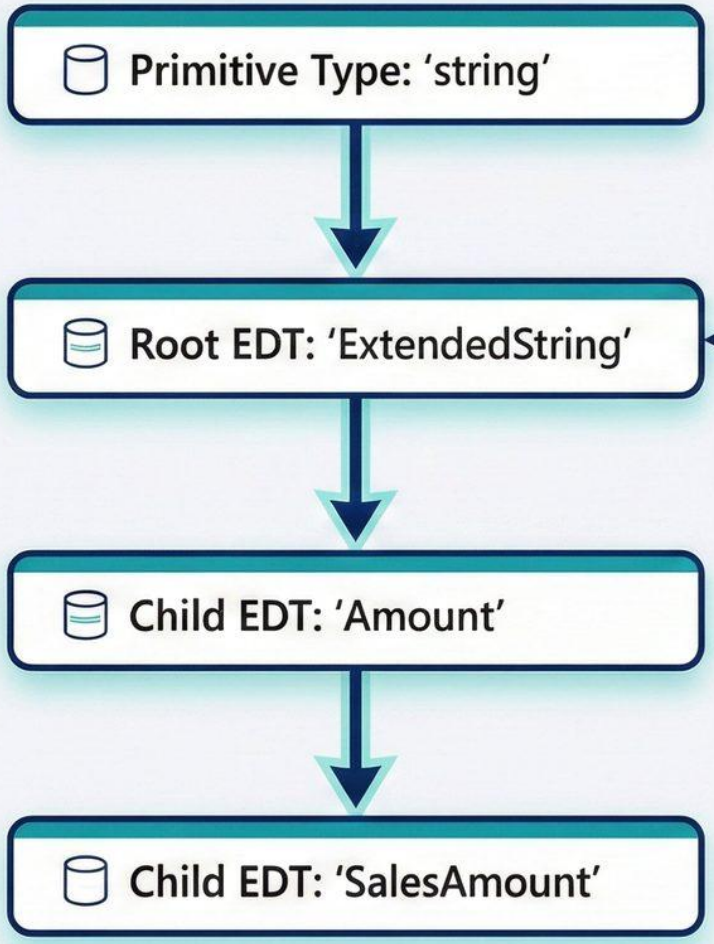


- **Immediate clarity:** ‘CustAccount total’ is self-describing.
- **Automatic** labels, help text, and consistent formatting on forms and in tables.
- **Reusability & Consistency:** Define once, use everywhere. Changes cascade.

EDTs provide reusability, consistency, and centralized metadata management, unlike basic primitive types.

truestack.dev

EDT Inheritance Hierarchy & Rules



AUTHORITY: Governs String Size (e.g., 30 characters). Is closest to primitive.

CRITICAL RULE: SIZE INHERITANCE

CORRECT: Inherited Size

- Child EDT: 'Amount'
  - String Size: 30 (Inherited)
- Child EDT: 'SalesAmount'
  - String Size: 30 (Inherited)

INCORRECT: Attempted Override

- Child EDT (e.g., SalesAmount)
  - ~~String Size: 20~~
  - REJECTED BY SYSTEM

METADATA OVERRIDE ALLOWED

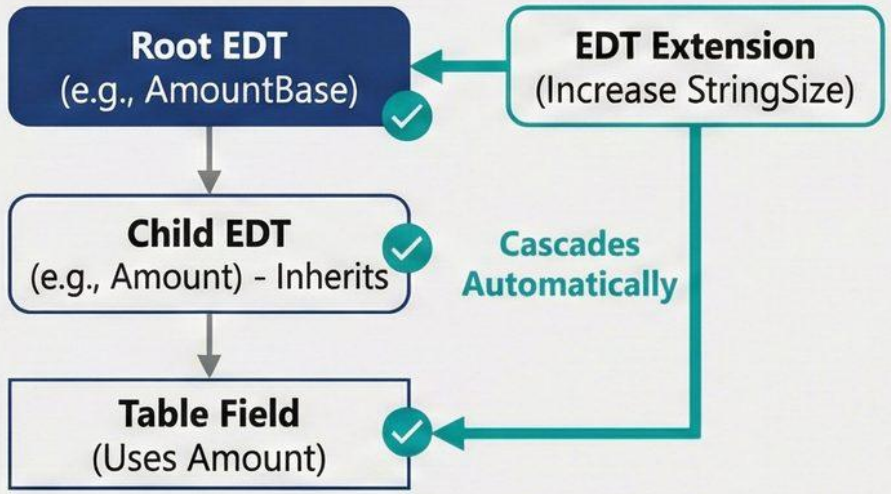
Label: 'Amount' → Specialization via Metadata → Label: 'Sales Amount'

KEY TAKEAWAY: The root EDT is the authority for string size; all children inherit it. Metadata like labels can be overridden for specialization.

truestack.dev

EDT Decision Strategy: Extend an Existing or Create a New EDT?

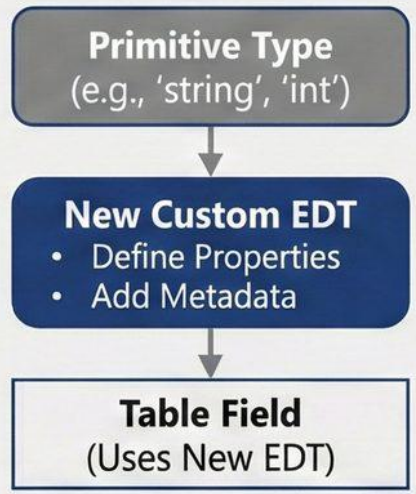
Extend Existing (Enhance & Reuse)



Action: Find and extend the root EDT. Change automatically cascades to all descendants and consuming fields.

Result: Elegant reuse.

Create New (Unique Requirements)



Action: Base on a primitive type, define properties and metadata from scratch.

Result: A new, reusable type.

The Golden Rule for Maintenance

✓ Reuse when it fits. ✓ Extend to enhance existing. ✓ Create new only for unique needs.

Maximizing reuse minimizes the maintenance burden later on.

truestack.dev

EDT Modifications: Safe vs. Unsafe Changes

✅ Safe Changes (Supported)

- ✅ Increase string size on **root EDT** (propagates to children).
- ✅ Change Labels & Help Text on **any EDT** (pure metadata).
- ✅ Create **new EDT** from a primitive type.
- ✅ Extend an EDT to modify its metadata.

❌ Unsafe Changes (Not Supported)

- ❌ Decrease string size (**Data Loss Risk**).
- ❌ Increase string size on a **child EDT** (system rejects, root governs size).
- ❌ Change alignment or fundamental properties (locked by design).
- ❌ Expect parent fields to automatically update when creating a derived EDT.

🔑 The Golden Rule

Size changes you must **never decrease**. String size can grow, but it can never shrink. This is a safety guardrail against data loss.

truestack.dev

Increasing EDT Field Size via Extension

A 5-step workflow for propagating size changes through inheritance.



Identify Root EDT

Trace hierarchy to find the base. Example: Amount → AmountBase → string.

Create EDT Extension

Extend the *root* (AmountBase), not the derived type.

Modify StringSize Property

In the extension, increase size (e.g., 20 to 30 chars).

Build Model

Compilation verifies that the extension is valid.

Synchronize Database

Schema updates. The change cascades automatically to all derived fields.

Key Takeaway: A single change at the root EDT automatically ripples through the entire inheritance tree, updating all derived fields without individual modification.

truestack.dev

Creating a New EDT from Scratch

A step-by-step guide to building reusable types from primitives.

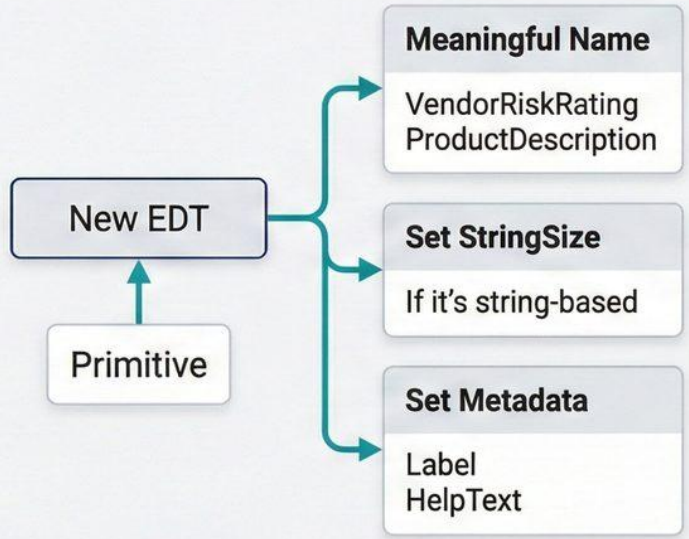
1 Step 1: Decide on a Primitive Type

- Text **String** (e.g., for text)
- **Int** or **Real** (e.g., for numbers)
- **Date** (e.g., for dates)
- **Boolean** (e.g., for true/false)

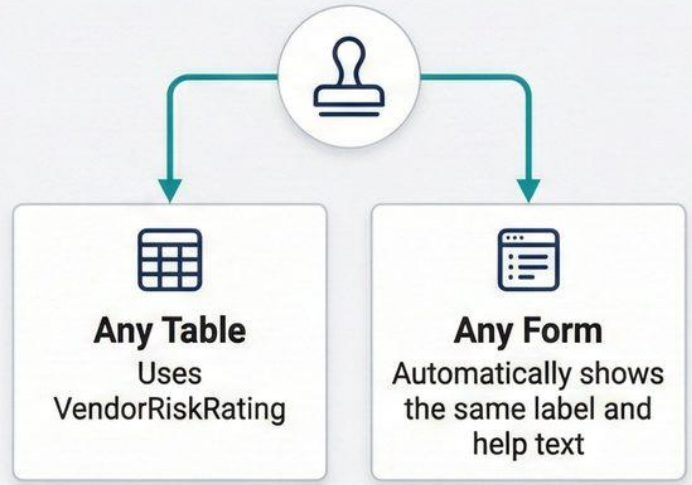
INCORRECT: Don't use derived types as the foundation

Derived Type ← Always go back to the primitive level

2 Step 2: Create and Configure EDT



3 Step 3: Reusable Type Outcome



Modify the EDT once, and all consumers update automatically.

SPECIAL CASE: Unlimited Text Fields

For large text fields (like notes or comments) → Create an EDT based on **'string'** → Set **StringSize** to **'Memo'** or **'Note'** → **Unlimited-length text field**
Useful for free-form comments

truestack.dev

Understanding Enums: Controlled Classification

Predefined, allowed values for data consistency and readability

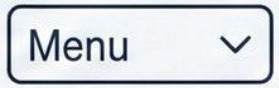
The Problem with Plain Strings



Vendor Risk Rating (String Field)
Developer types 'Excellent' ❌ ←
Developer types 'GOOD' ❌ ←
Developer types 'Med' ❌ ←

⚠ **Result:** Inconsistent Data, Typos Break Reports, Surprise Values

The Enum Solution



Vendor Risk Rating (Enum Field)

Medium

- Good
- Medium
- Risky

✅ **Result:** Defined Allowed Values, No Typos Possible, No Surprises

Why Use Enums?

- **Perfect for Classification Scenarios:** Status (Active, Inactive), Priority (Low, Medium, High), Risk (Good, Medium, Risky)
- **Self-Documenting Code:** 'VendorRisk.Good' immediately indicates meaning, unlike numeric codes (0, 1, 2)
- **Ensures** Code Readability and Data Consistency

Bottom Line: Enums make your code readable and your data consistent by defining allowed values.

truestack.dev

Creating an Enum-Based Field: A Step-by-Step Guide

Ensuring data integrity through predefined value lists.

1. Create Base Enum

In Application Explorer, create a new Enum named 'VendorRiskRating'.

VendorRiskRating

- Good (0)
- Medium (1)
- Risky (2)

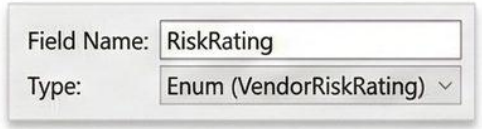
(Literal values mapped to underlying numeric values)

2. Create Table Extension

Extend the relevant table to enhance its capabilities.

3. Add New Field

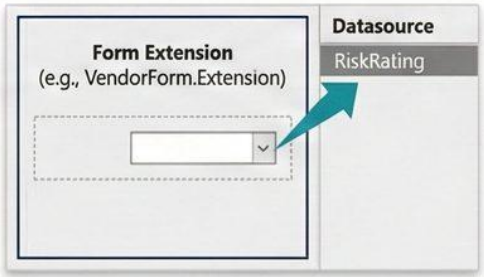
Add a new field to the extension, setting its type to the created Enum.



4. Extend Form & Add Field



4. Extend Form & Add Field



OUTCOME & BENEFITS



- ✓ Automatic dropdown display.
- ✓ Users restricted to defined values.
- ✓ Prevents typos and unexpected data.
- ✓ Self-documenting data structure.

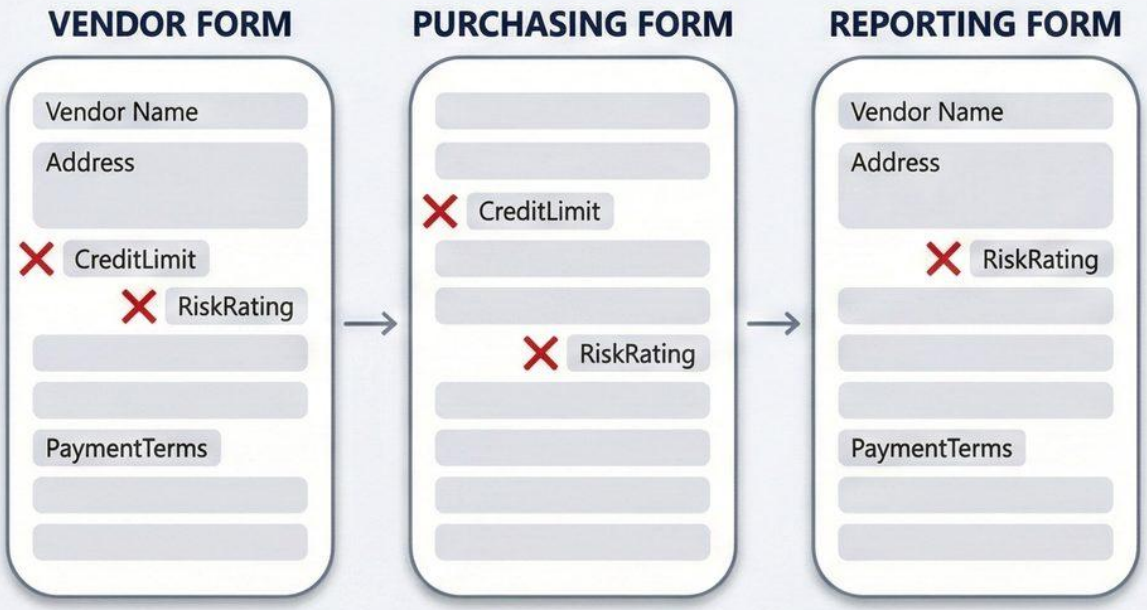
Enums ensure data integrity by limiting choices to predefined, **valid values**.

truestack.dev

FIELD GROUPS: ORGANIZATIONAL POWER & CONSISTENCY

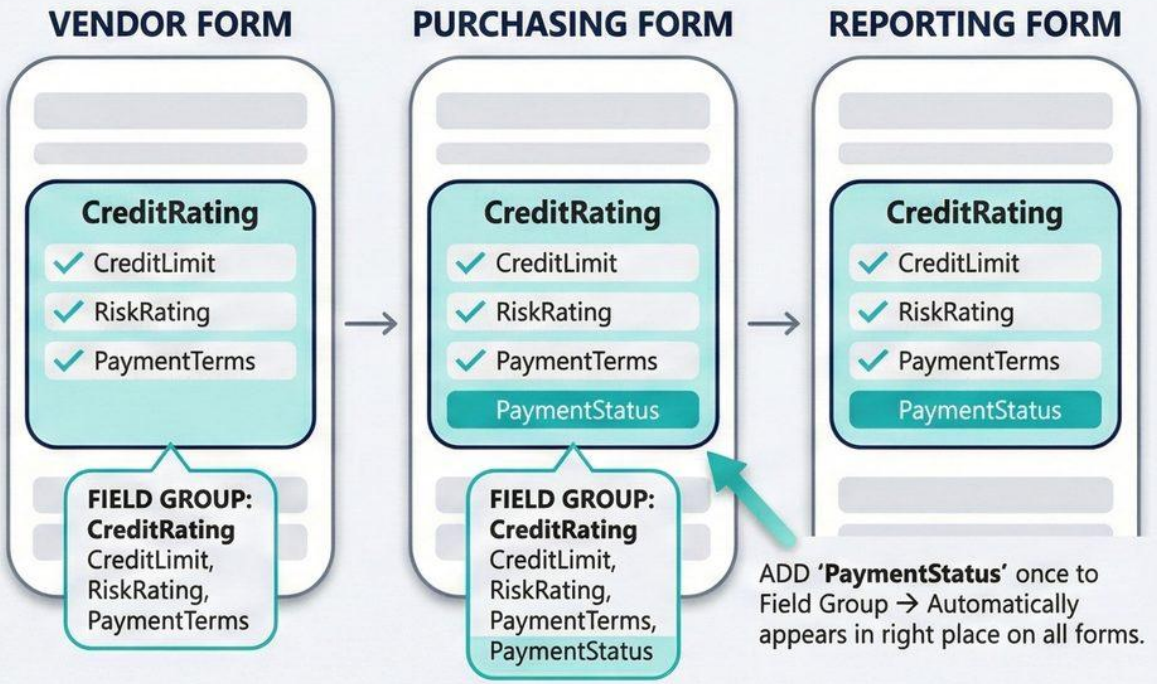
Streamlining Field Management and Ensuring Uniform Layouts

WITHOUT FIELD GROUPS: INCONSISTENT LAYOUTS



CONFUSES USERS.
DEFINES ITS OWN LAYOUT.

WITH FIELD GROUPS: GUARANTEED CONSISTENCY



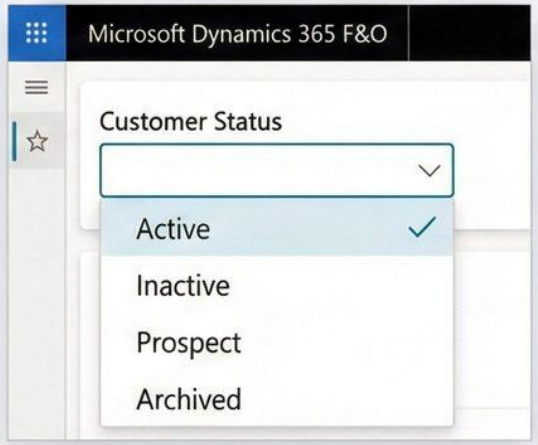
CONSISTENCY GUARANTEED.
ALWAYS LOOKS THE SAME.

Field groups ensure related fields appear together in a consistent layout across all forms, simplifying management and enhancing user experience.

truestack.dev

Displaying Enums on Forms: UI Options

Dropdown (Most Common)



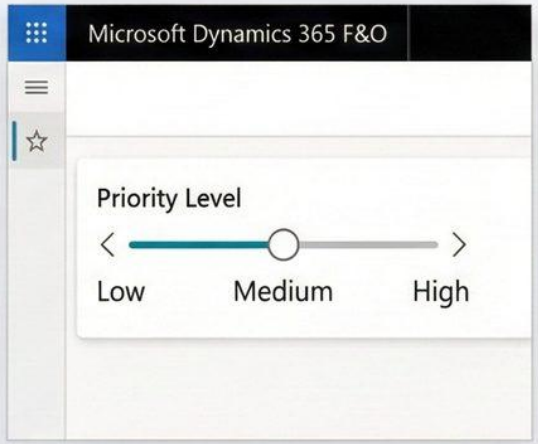
Default choice. Shows all valid values in a list. Ideal for many values, works for any size data set. Users click, see options, and select.

Option Buttons (Radio Buttons)



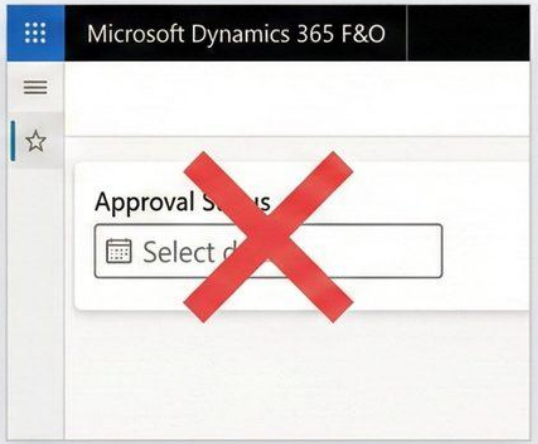
Shows all values simultaneously. Excellent for 3-5 values. Good for quick decisions, users pick one immediately.

Slider Bar



Works well for ordered, continuous values. Intuitive for ordinal data (Low → High).

What's Not Allowed: Date Field



INCORRECT. Never use a Date field for an Enum. Date fields are for date primitives, not controlled lists. System prevents this combination.

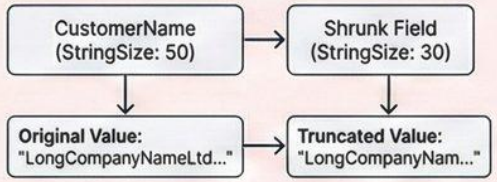
Most commonly, **a dropdown** is the right choice due to its flexibility and familiarity, but option buttons and sliders offer better UX for specific small or ordered value sets.

truestack.dev

Common Mistakes with EDTs and Enums

Mistake 1: Decreasing an EDT string size

✗ INCORRECT



Data gets truncated. Existing records lose information.

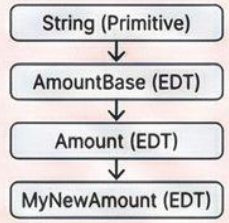
✓ CORRECT



Never shrink fields. System blocks this to prevent data loss.

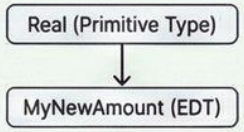
Mistake 2: Using non-primitive EDTs as the foundation for new EDTs

✗ INCORRECT



Deep inheritance chains complicate maintenance.

✓ CORRECT



Always base new EDTs on primitives: string, int, real, date.

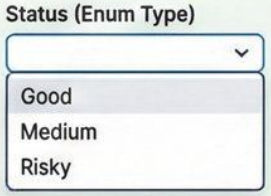
Mistake 3: Storing Enum-like values as plain text

✗ INCORRECT

Status (String)
'Good'
'Risky'
'Excellent'
'GOOD'

Typo and format risks lead to garbage data.

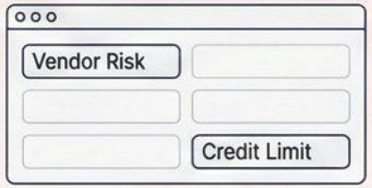
✓ CORRECT



Use an Enum type. System enforces valid values.

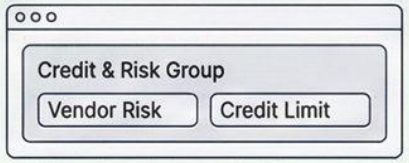
Mistake 4: Creating Enums without grouping related ones with Field Groups

✗ INCORRECT



Related fields scattered, causing confusion.

✓ CORRECT



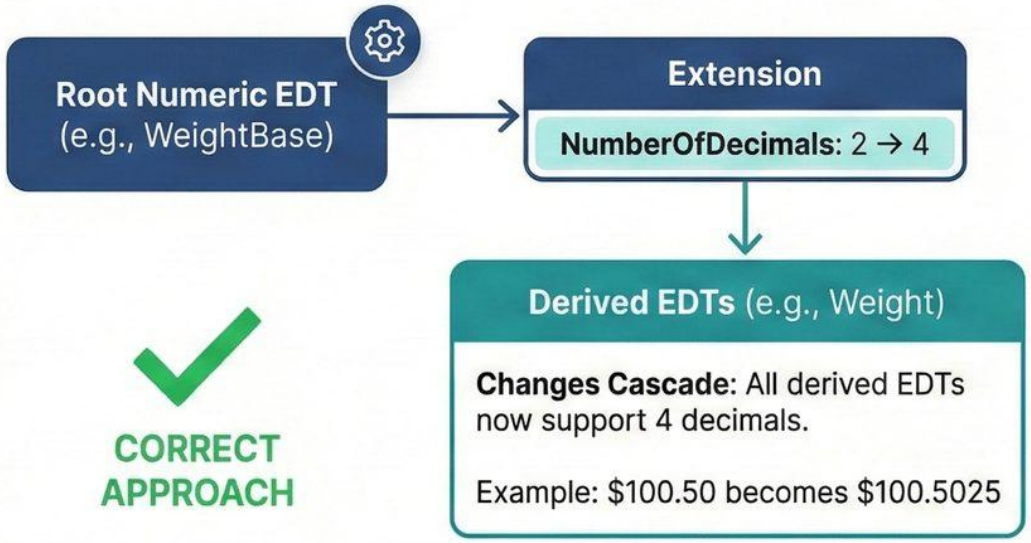
Use Field Groups to organize related fields for consistent UI.

Adhering to these best practices for EDTs and Enums ensures data integrity, maintainability, and a clear user experience.

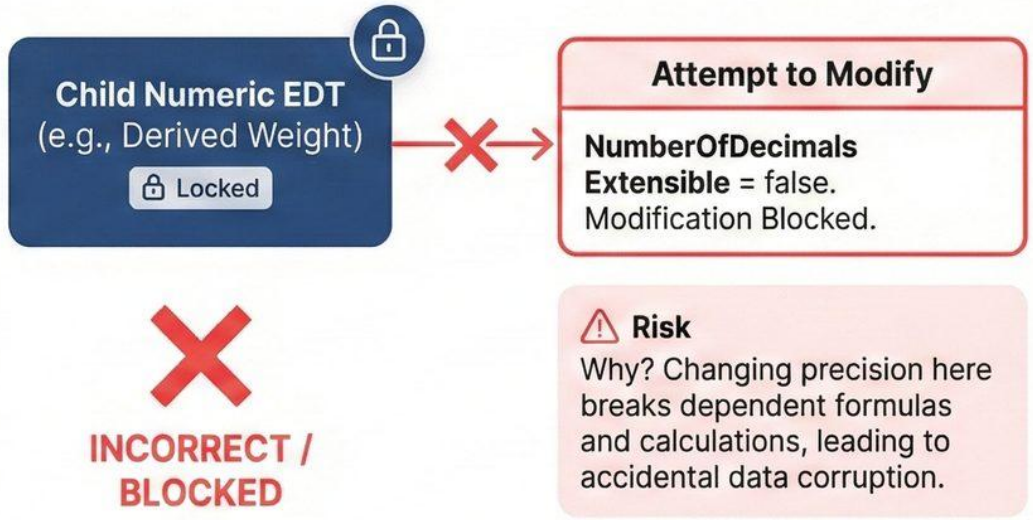
truestack.dev

Managing Numeric Precision in EDTs: Root vs. Child Rules

Modifying Precision (The Root Rule)



The Child Constraint (Why it's Locked)

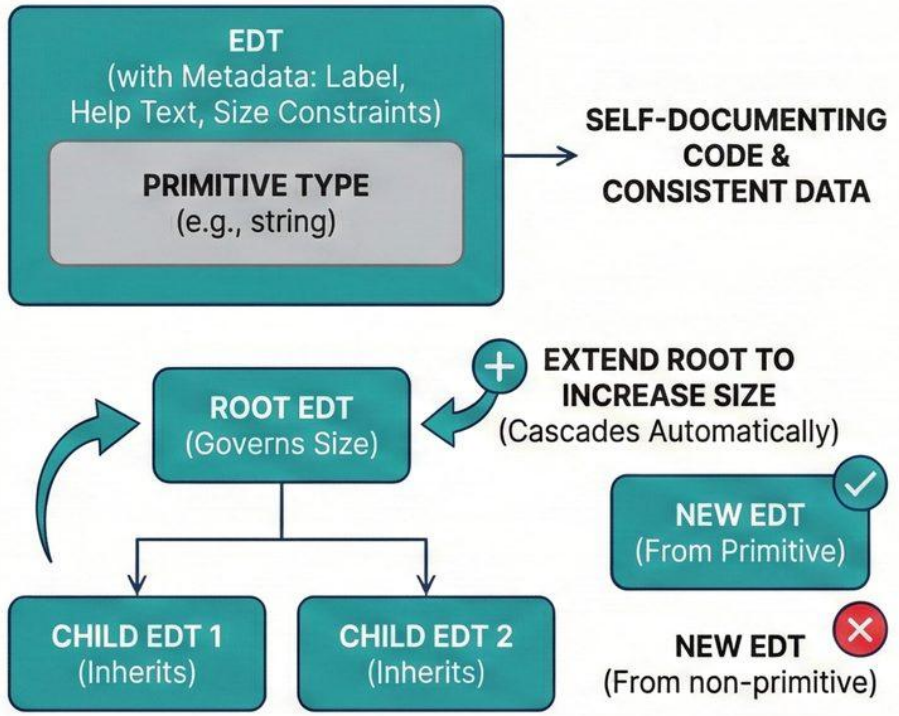


Precision management, like string size, is controlled at the root EDT level to ensure system integrity and prevent data corruption.

truestack.dev

Recap: Topic 2.1 - EDTs and Enums

Extended Data Types (EDTs)



Enums (Base Enumerations)

DEFINED LISTS (Prevents Typos, Self-Enforcing):

- Good
- Medium
- Risky



DROPDOWN | OPTION BUTTONS | SLIDER

FIELD GROUPS: Organize related fields for UI consistency.

NEXT TOPIC: Physical Storage Layer

Topic 2.2: Tables, Indexes, and Caching



TABLES | INDEXES | CACHING

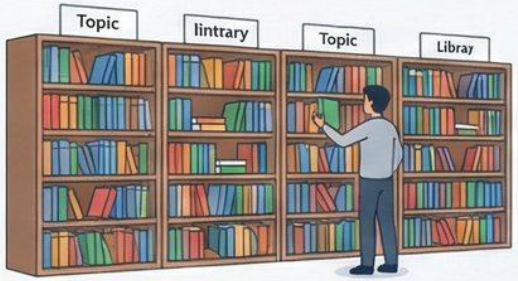
Understanding data types sets the foundation for the next phase: the physical storage layer.

truestack.dev

Topic 2.2: Tables, Indexes, and Caching

Where Data Lives: Storage, Organization, and Performance

Indexes: Organizational System for Data (Performance)



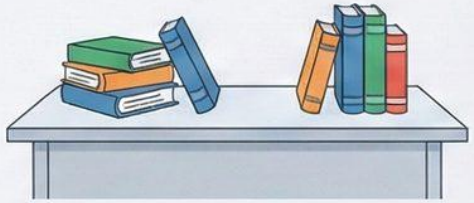
Organized (With Indexes) - Fast Access (Milliseconds)



Disorganized (Missing Indexes) - Slow Access (30+ Seconds)

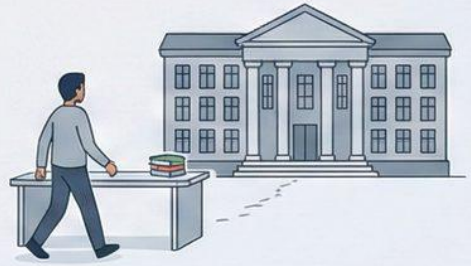
Indexes act as an organizational system for data, allowing instant lookups instead of endless searching.

Caching: Frequently-Used Data Access (Speed)



Cached Data (On Desk) - Instant Access

Non-Cached Data (Walking to Library) - Slower Access (Every Single Time)



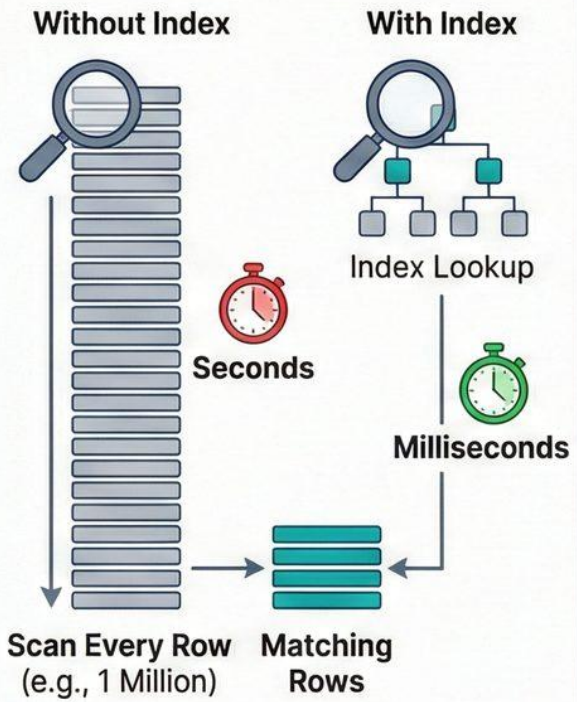
Caching keeps frequently used data readily available, avoiding repetitive fetching from the source.

Designing performant tables and implementing smart caching strategies are crucial for optimizing system speed and user experience.

truestack.dev

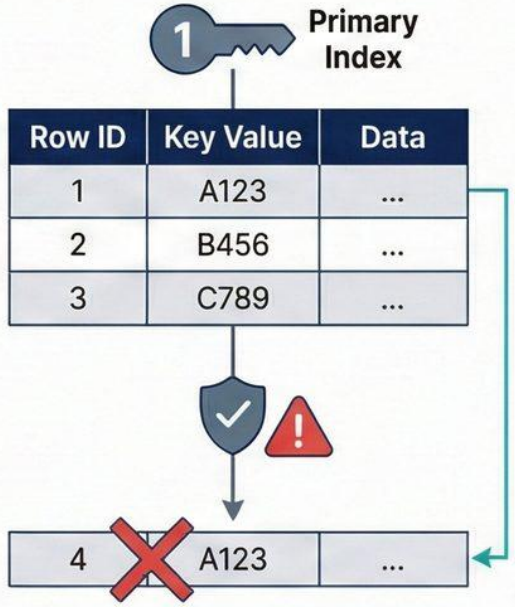
CRITICAL PURPOSES OF INDEXES

1. Performance



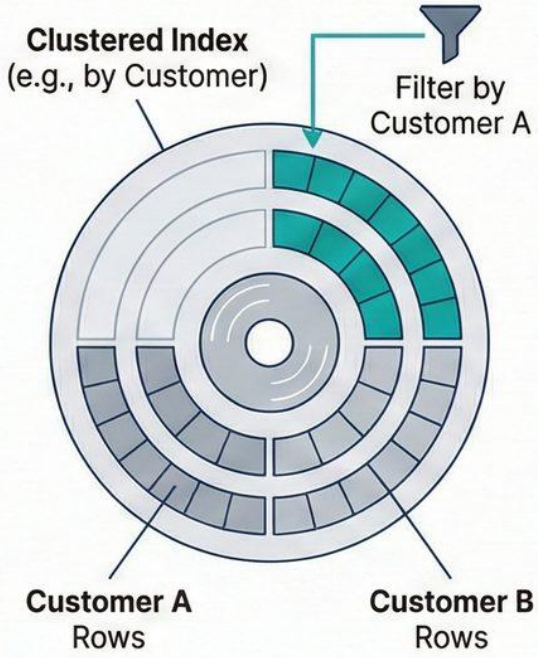
Jump directly to data; significant speed increase.

2. Uniqueness



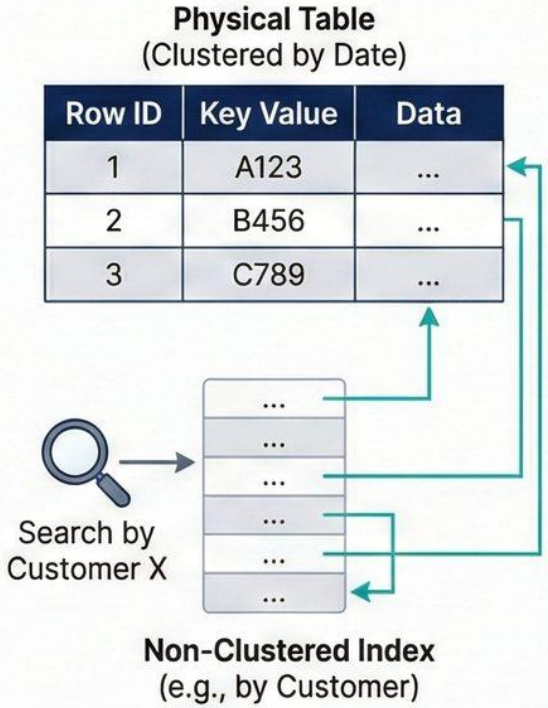
Ensures no duplicate keys; maintains data integrity.

3. Physical Ordering



Rows stored physically together; efficient for range queries.

4. Secondary Access Paths



Alternative lookup ways without restructuring physical storage.

Understanding these four purposes helps you choose the right index for the right problem.

truestack.dev

THREE MAIN INDEX TYPES

PRIMARY INDEXES

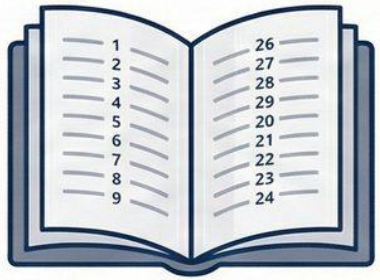
- Uniquely identify each row
- No two rows share same value (✕)
- Used for default lookups & relationship enforcement



Unique Key

CLUSTERED INDEXES

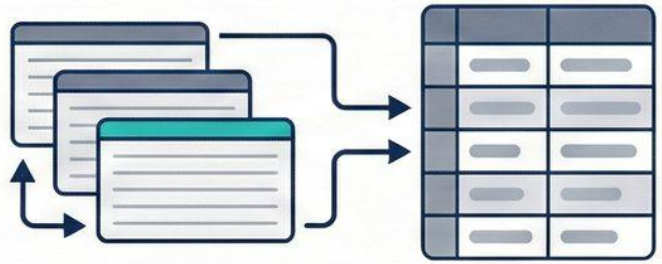
- Define physical sort order of table
- Rows stored sorted by index columns
- One table, ONE clustered index



Book Pages: Fixed Physical Order

NON-CLUSTERED INDEXES

- Provide additional logical entry points
- Does not change physical storage
- Many can exist on a single table



Index Cards: Alternative Access

CRITICAL DETAIL: COLUMN ORDER MATTERS

Index on (Customer, Date) → ✓ Filter by Customer first, **THEN** Date

Index on (Date, Customer) → ✓ Filter by Date first, **THEN** Customer

truestack.dev

How to Choose the Right Index: The Three Questions

1. Unique Identifier?

Do I need a unique identifier for each row?

→ YES →

PRIMARY INDEX

- Uniquely identifies each record.

2. Frequent Filter/Search?

Will users frequently filter or search by a specific column?

→ YES →

NON-CLUSTERED INDEX

- E.g., searching by customer name.

3. Physical Organization?

Do I need to organize data physically for efficiency?

→ YES →

CLUSTERED INDEX

- E.g., filtering by Date for range queries.

Consequences of Wrong Choices

- Missing a critical index means a query that should return in 50 milliseconds takes 30 seconds instead.
- A poorly ordered clustered index means your data isn't organized for your most common queries.

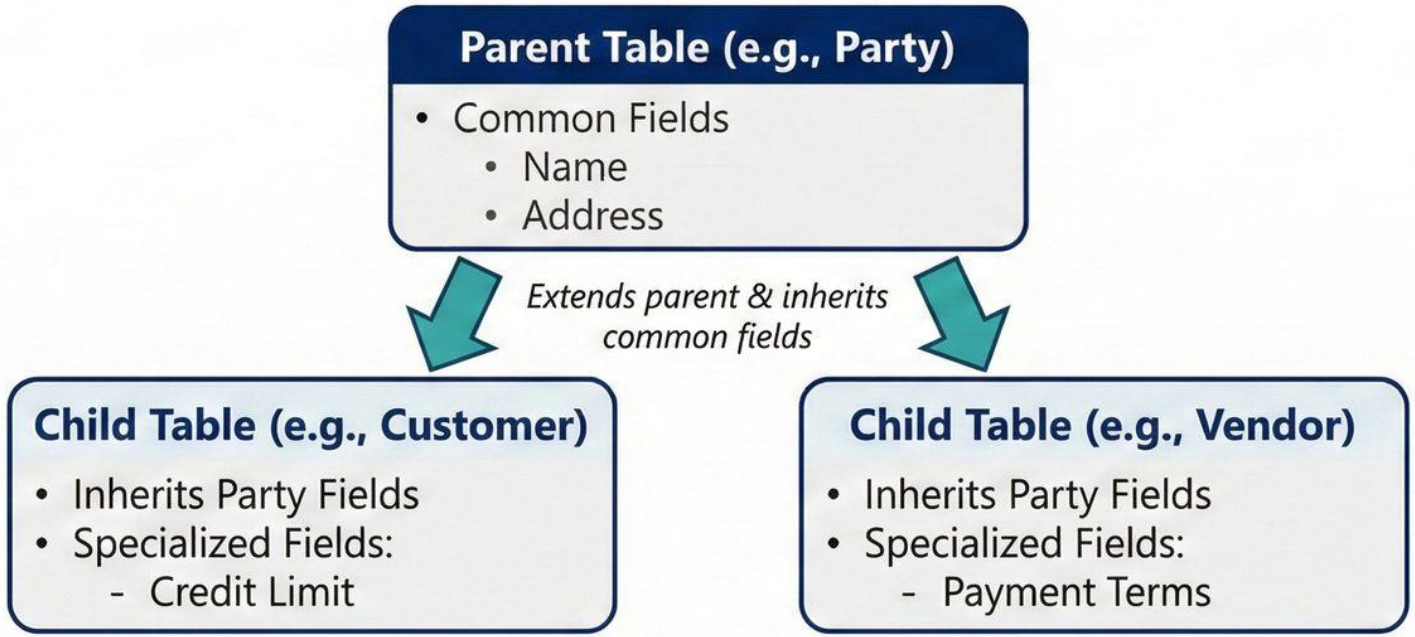
Take the time to understand your access patterns and design indexes accordingly.

truestack.dev

Table Inheritance in D365 Finance & Operations

Parent-Child Relationships for Efficient Data Modeling

Inheritance Structure & Example



Controlling Properties

- ✓ **'Extends':** Identifies the Parent Table name.
- ✓ **'SupportsInheritance':** Boolean to enable behavior.

Both properties must be configured together for inheritance to function.

Benefits of Inheritance



truestack.dev

Product Subtypes in D365 Finance & Operations

PRODUCT



- Standalone item.
- Individually identifiable.
- Uniquely defined.
- No variants, no dimensions.
- Used for commodities or simple items.

Example: Simple widget, standard bolt.

PRODUCT MASTER



- Template.
- Not sold directly.
- Defines dimensions (e.g., size, color).

Example: T-shirt master definition.

PRODUCT VARIANT



- Actual SKU created from master.
- Specific dimension values.
- Sellable item.

Example: Large Red Shirt.

ITEM



- Legal-entity-specific release.
- Defined globally, released per company.
- Separate Item record for each company.

Example: USMF company release of Large Red Shirt.

WHY DOES SUBTYPE MATTER?

Affects handling of: Variants, Purchasing, Sales, Forecasting.
Incorrect choice disrupts system behavior.
Understand product structure before creation.

truestack.dev

PURCHASING & PLANNING QUANTITY PARAMETERS

PARAMETER DEFINITIONS

Standard Order Quantity

Typical lot size. Planning uses this value to generate purchase proposals.

e.g., Standard = 100, Demand = 250 → Suggestion = 300 (three lots of 100).

Minimum Order Quantity

Smallest amount you can order. Prevents ordering too little.

e.g., Supplier won't accept less than 50 units.

Maximum Order Quantity

Ceiling or upper limit. Prevents ordering too much.

e.g., Warehouse limits you to 1000 units.

Multiple Order Quantity

Rounding constraint. Quantities must be multiples of this value.

e.g., Ships in cartons of 10; must order 100 or 110, not 105.

EXAMPLE: WORKFLOW & OUTCOME



Quantity parameters guide purchasing to ensure realistic and executable order sizes.

truestack.dev

Caching Strategies in D365FO: Optimizing Performance with CacheLookup

Balance memory usage versus database access to improve speed.



The Goal: Keep constantly used data in memory to avoid expensive database trips.

EntireTable

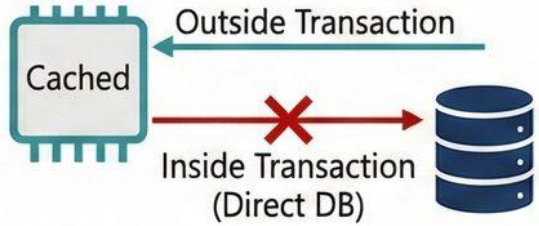


Description: Loads the whole table into memory.

Use Case: Small parameter tables (e.g., configuration, lookup values, code lists). Ideal for < 100 rows.

⚠ Avoid for large tables; eats up memory.

NotInTTS



Description: Caches records only outside transactions. Skips cache inside transactions.

Use Case: Large transactional tables.

Ensures most current data during critical transactions.

Found



Description: Caches found records and also caches unsuccessful lookups (misses).

Use Case: Stable master data (e.g., vendor lists). Subsequent queries hit cache.

FoundAndEmpty



Description: Caches found results and also caches empty (not-found) results.

Use Case: Optional, region-specific data that may not exist for all entries.

Bottom Line: Choose caching strategy based on table size, data volatility, and access patterns.

truestack.dev

Matching Caching Strategies to Table Types

Optimizing performance by aligning caching with data usage patterns

Parameter Tables

Company, CurrencyCode, UnitOfMeasure

→ **EntireTable**

Load once, serve from memory always

Small, stable, high access

Transaction Tables

Examples: SalesOrder, PurchaseOrder

→ **NotInTTS**

Latest data, performance outside transaction blocks

Huge, constantly updated, current data needed

Master Data Tables

Examples: Customer, Vendor, Product

→ **Found**

Caches found records and misses

Stable, moderate access, common lookups

Region-Specific Master Data

Example: LocalTaxRates

→ **FoundAndEmpty**

Caches both found and empty results

Exists in some regions, not others

Maximize performance and minimize stale data risk by matching caching to data usage.

truestack.dev

Table Extensions: Customizing Without Overlaying

The supported model for safe, upgradeable customizations in D365 F&O

What's Possible (Supported)

- **Adding new fields:** Create custom fields without touching the standard table
- **Adding new indexes:** Improve query performance on new fields
- **Event handlers:** Inject logic pre-insert, post-insert, pre-update, post-update
- **Extend methods (CoC):** Wrap existing methods to add behavior before/after execution
- **Validations:** Enforce business rules and ensure data quality

What's NOT Possible (Unsupported)

- **Changing structural behaviors:** Do not modify primary keys
- **Modifying data types:** Do not change existing field types
- **Removing inherited fields:** Changes break relationships and require deprecated overlaying

Key Takeaway: Extensions are the supported model going forward. They are safe, upgradeable, and avoid deprecated overlaying. Always figure out an extension-based approach.

truestack.dev

Adding New Fields to Tables: Two Supported Methods

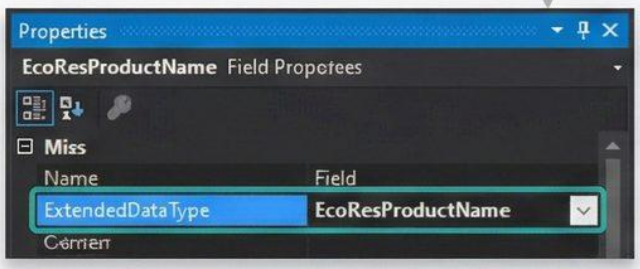
Method 1: Add Base Fields First, Then Assign EDT

1. Create Table Extension

↓

2. Add Base Field (e.g., String)
In Table Designer

↓



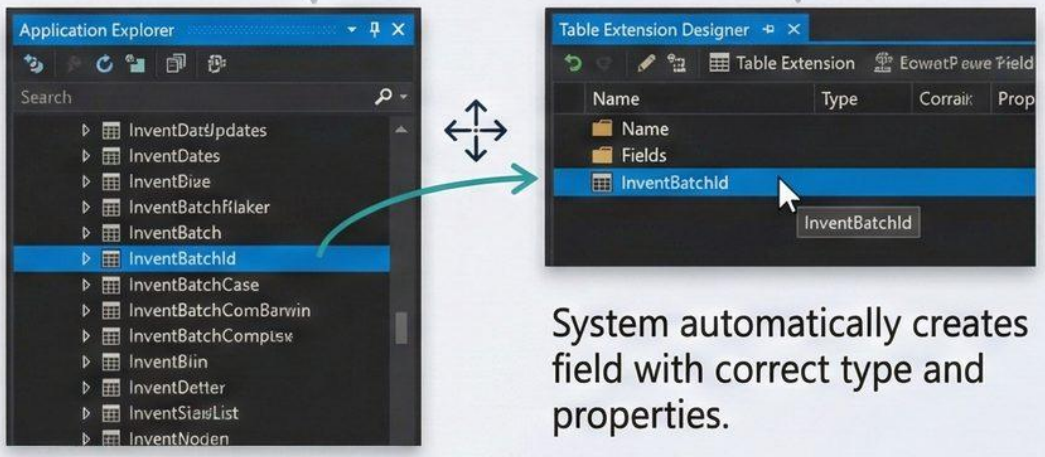
Set **ExtendedDataType** property to required EDT. Field inherits metadata.

✓ Good for incremental building.

Method 2: Drag EDTs Directly Onto Table

1. Open Application Explorer

↓



System automatically creates field with correct type and properties.

✓ Faster, immediate results.

Key Principle & Best Practices

- ✓ **Always use Table Extensions.** Never overlay standard tables. Extensions are safe and upgradeable.
- ✓ **Always assign the correct EDT.** Ensures consistent metadata (e.g., Product Name = EcoResProductName).
- ✗ **Never use Enums for EDT-typed fields** (e.g., Product Name is NOT an enum).

Choose the method that fits your workflow, but always prioritize table extensions and correct EDTs for maintainability.

truestack.dev

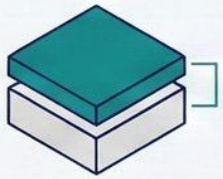
Adding Fields to Standard Forms: Rules and Best Practices

Ensure consistency, upgradeability, and compliance by following strict architectural rules.

1. Use Form Extensions (Not Overlaying)

Extensions (CORRECT)

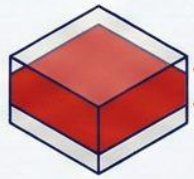
Safe & Upgradeable



Extension Layer
(Adds new functionality)

Overlaying (INCORRECT)

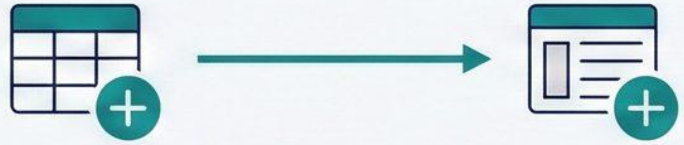
Blocks Upgrades, Not Supported



Overlay Layer
(Direct Modification)

Supported & Future-Proof → Upgrade Conflict

2. Data First, Then Display

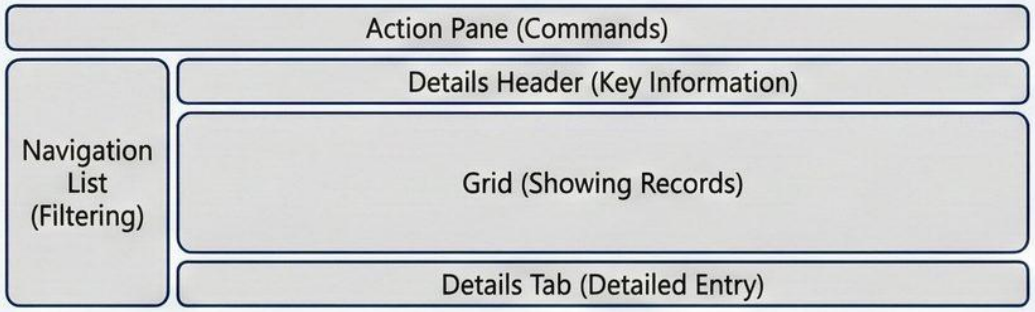


1. Create **Table Extension** & Add Field

2. Create **Form Extension** & Add Field

WARNING: Never display data that doesn't exist in the table.

3. Follow Form Patterns



Place new fields in the appropriate section based on this pattern.

4. Exposing Cross-Company Data

- **EcoResProductTranslation:** Provides Product Name (user language).
- **InventDim**: Provides Dimension Information.
- **Apply Legal Entity Filters:** Ensures data is for the correct company only.

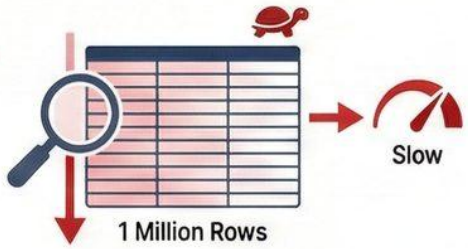
Following these rules ensures forms remain **Consistent**, **Upgradeable**, and **Compliant** with Microsoft's architecture.

truestack.dev

Four Critical Mistakes in Table Design

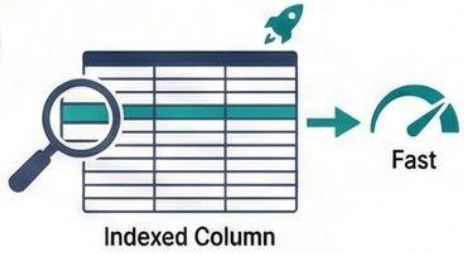
Mistake 1: Missing Indexes

INCORRECT: Full Table Scan



Glacial Performance

CORRECT: Index During Design



Efficient Queries

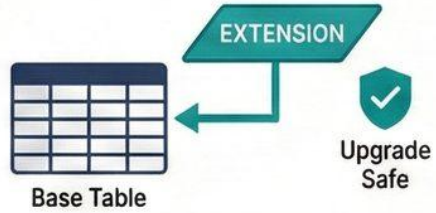
Mistake 2: Overlaying Tables

INCORRECT: Modify Standard Table Directly



Blocks Upgrades

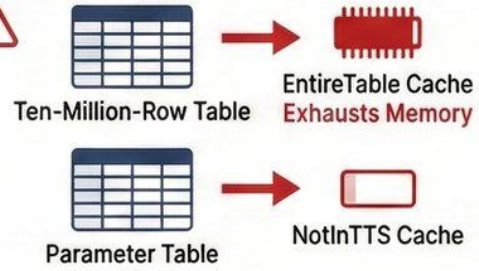
CORRECT: Use Table Extensions



Upgrade Compatibility

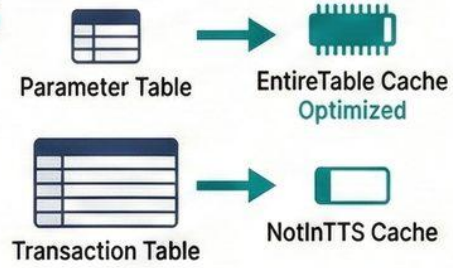
Mistake 3: Wrong Caching Strategy

INCORRECT: Inappropriate Caching



Missed Optimization

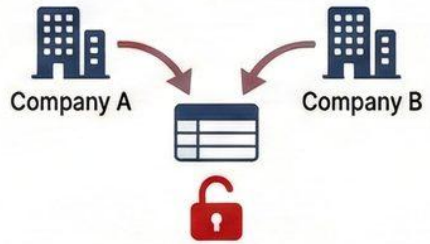
CORRECT: Select Strategy by Table Type



Performance Balance

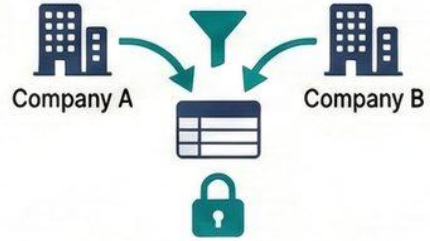
Mistake 4: No Cross-Company Handling

INCORRECT: Unfiltered Custom Fields



Data Breach Risk

CORRECT: Apply Legal Entity Filters



Ensures Data Integrity

Avoid these mistakes through disciplined table design from the start to ensure performance, upgradeability, and data security.

truestack.dev

Topic 2.2: Recap & Next Steps

Summary of key concepts covered in Topic 2.2 and introduction to Topic 2.3

Recap of Topic 2.2: Tables & Storage

Indexes

- **Primary**: Ensures uniqueness
- **Clustered**: Defines physical order
- **Non-clustered**: Alternative lookup paths
- **Column Order**: Critical for optimization

Table Inheritance

- **Properties**: Extends, SupportsInheritance
- **Benefit**: Reduces data duplication

Product Subtypes

- **Product**: Standalone items
- **Product Master**: Template
- **Product Variant**: SKUs
- **Item**: Company-specific releases

Order Quantity Parameters

- **Standard**: Typical quantity
- **Minimum**: Floor
- **Maximum**: Ceiling
- **Multiple**: Rounding

Caching Strategies

- **EntireTable**: Parameter tables
- **NotInTTS**: Transaction tables
- **Found**: Stable master data
- **FoundAndEmpty**: Optional data

Extensions (Safe Customization)

- **Table Extensions**: Add fields, indexes, logic (No overlaying)
- **Form Extensions**: Follow form patterns

Next Step: Topic 2.3

Relations and Delete Actions

Focus on how records relate to each other and how to prevent orphaned data when a parent record is deleted to maintain data integrity.



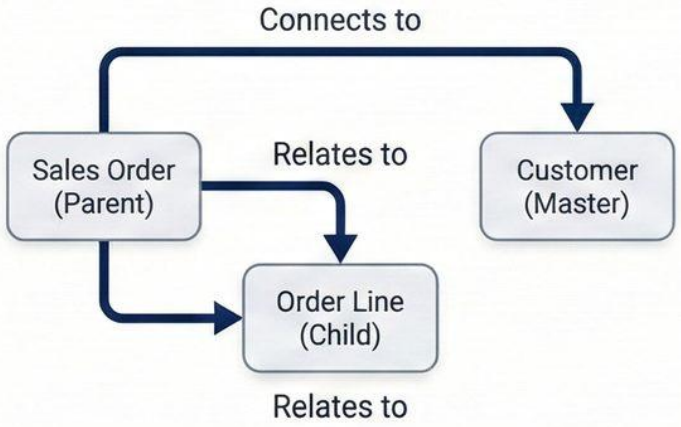
Understanding these foundational concepts is crucial for efficient D365FO development.

truestack.dev

Topic 2.3: Relations and Delete Actions

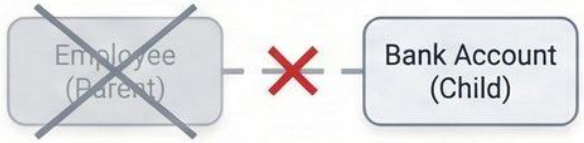
Designing safe, clean data relationships to protect integrity

The Data Model: Connective Tissue



Deletion Problems & Risks

Orphaned Data



Sitting there orphaned

Phantom Data

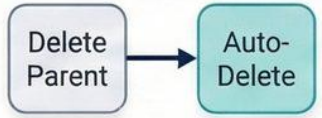


Unexplained phantom data

Delete Actions: The Solution

Cascade (Automatic)

Automatically deletes the children when parent is deleted.



Auto-Delete Children

Restricted (Preventive)

Prevents you from deleting the parent if children still exist.



Blocked if Children Exist

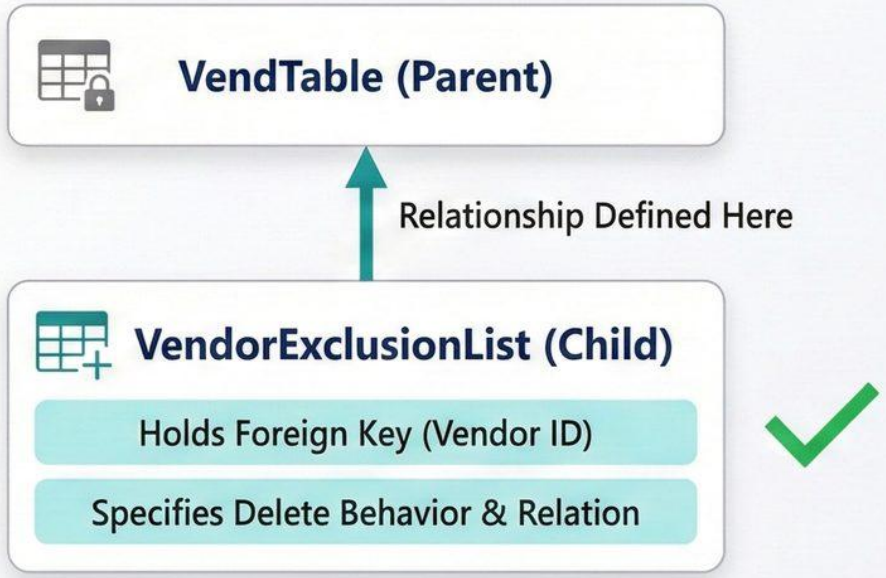
We define parent-child connections and specify exactly what happens when deletion occurs.

truestack.dev

Fundamentals of Table Relationships & Extensibility

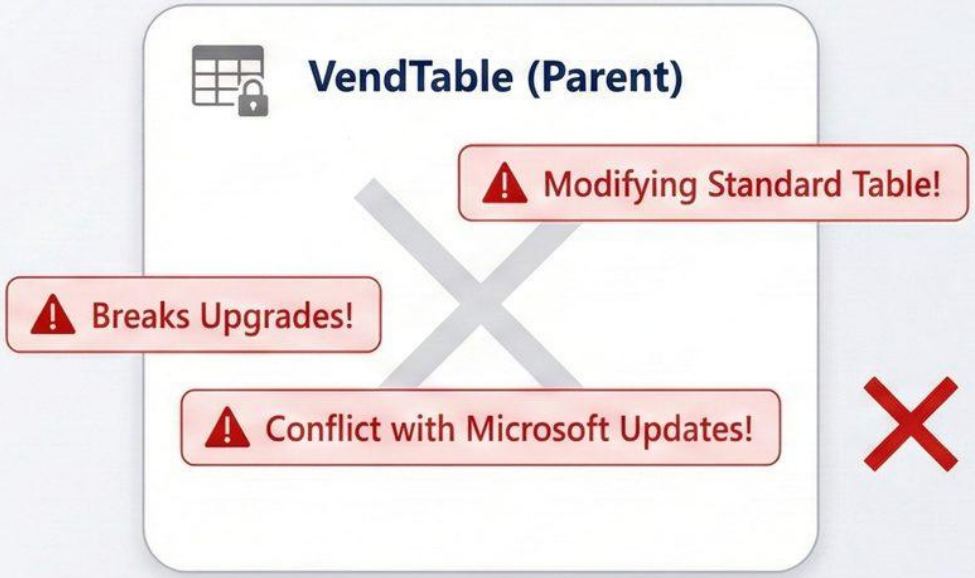
Relationships belong to the child table; avoid modifying standard parent tables for relationship logic.

Correct Approach: The Extension Philosophy



Create new child table referencing parent. Standard table stays completely untouched. Safe for upgrades.

Incorrect Approach: Anti-Pattern

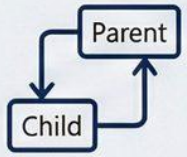


Modifying standard Microsoft tables directly leads to upgrade conflicts and system failures.

Takeaway: Always define relationships and hold foreign keys in the child table to maintain a clean, upgrade-safe extensibility model.

truestack.dev

Four Key Properties of a Relation in D365 F&O



RelationshipType

Describes the logical ownership.

- ✓ **Association**: Child is independent. Technically, parent might not be mandatory (unusual).
- ✓ **Composition**: Child depends on the parent. True parent-child dependency. Every exclusion must have a vendor.



Validate

Determines whether the system enforces that the parent exists.

Set to **Yes** when parent existence is mandatory. If the vendor doesn't exist, the system just rejects the exclusion entry outright.



ForeignKey

Your technical binding.

The foreign key field on the **child table** holds the **parent ID** – that's the actual link.



OnDelete

Most critical property. Controls what happens when the parent gets deleted.

- ➢ **None**: Let SQL handle it (might error out).
- ➢ **Cascade**: Automatically delete all the children.
- ➢ **Restricted**: Block the parent deletion if children exist.
- ➢ **Cascade + Restricted**: Delete children unless another relationship protects them.

⚠ Setting OnDelete explicitly is mandatory. Never rely on implicit SQL behavior; always explicitly state your intention in the metadata.

truestack.dev

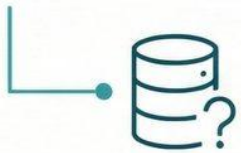
OnDelete Property: Four Options

Breakdown of deletion behaviors for child-parent relationships in Dynamics 365 F&O.

None: Minimal Configuration

Leaves deletion behavior to SQL referential integrity. SQL might allow, block, or error.

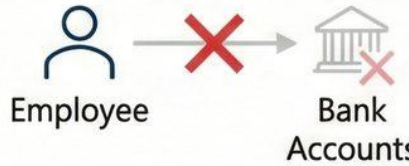
Risky; future developers won't understand intent. Avoid unless highly specific reason.



Cascade: Automatic Child Deletion

Automatically deletes all child records when parent is deleted. Ideal when children are completely dependent.

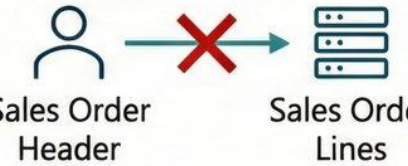
Example: Employee Bank Accounts. Deleting employee cascades to remove all associated accounts.



Restricted: Prevents Parent Deletion

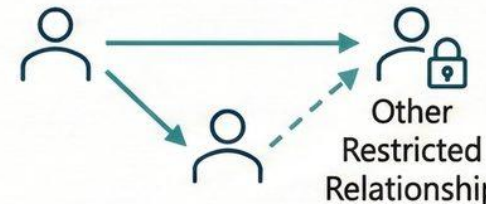
Blocks parent deletion if any child records exist. Ideal when children must never become orphaned.

Example: Sales Order Lines. Cannot delete sales order header if line items exist. Blocks deletion, prompts to remove lines.



Cascade plus Restricted: Conditional Cascade

Deletion cascades, but only if no other relationship with a Restricted rule protects the child.
Complex scenarios where a child participates in multiple relationships. Child deleted unless another relationship says "protect this record".



Choose your OnDelete based on business rules: Can children exist without parent? Use **Restricted**. Must children be removed with parent? Use **Cascade**.

truestack.dev

Practical Scenario: Vendor Exclusion List & OnDelete Behavior

1. **Create Table: VendorExclusionList**
   Add fields for the exclusion reason and the date.
2. **Add Foreign Key: VendorID**
   References the parent VendTable.
3. **Create Relation**
   ForeignKey on VendorExclusionList pointing back to VendTable.
4. **Set RelationshipType**
   Set to Composition because exclusion depends on vendor existing.

Step 5: Set Your OnDelete Property - Two Options

Option A: Cascade



Deleting a vendor automatically deletes all exclusion entries for that vendor.

Makes sense: if the vendor is gone, the exclusion becomes irrelevant.

Use Case: If exclusions are just informational, Cascade is fine.

Option B: Restricted



Deleting a vendor gets blocked if exclusion entries exist.

User must manually remove exclusions first before deleting vendor.

Use Case: Critical audit records that must be preserved (archive first).

Your business logic decides. Choose intentionally based on what your requirements actually are.

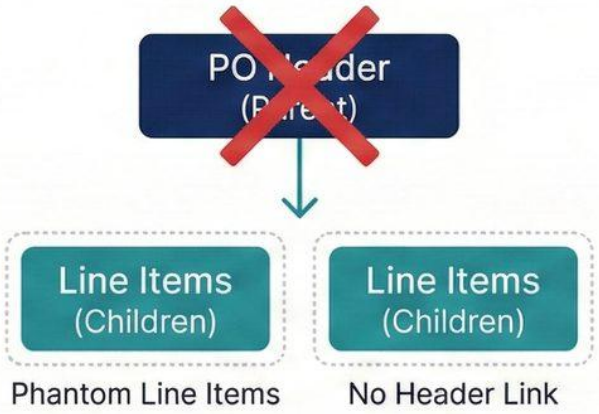
Your business logic decides—choose intentionally based on actual requirements.

truestack.dev

Restricted OnDelete: Protecting Child Data

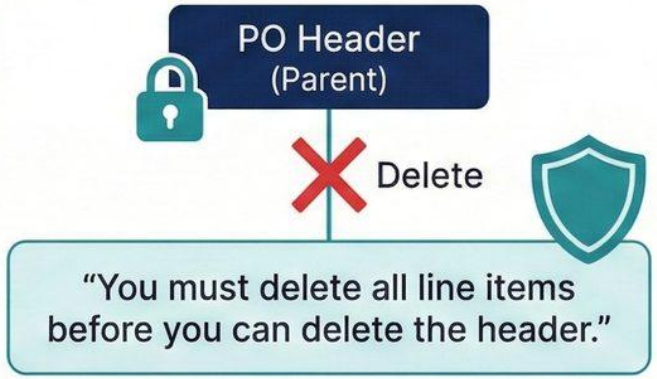
Preventing orphaned records and data corruption

Without Restricted: Orphaned Data



Reports break.
Accounting gets confused.
Data is corrupted.

With Restricted: System Guardian



Enforces business logic.
Prevents accidental data corruption.

Safe Workflow & Benefits



- Clean and safe.
- Restricted is your guardian here.

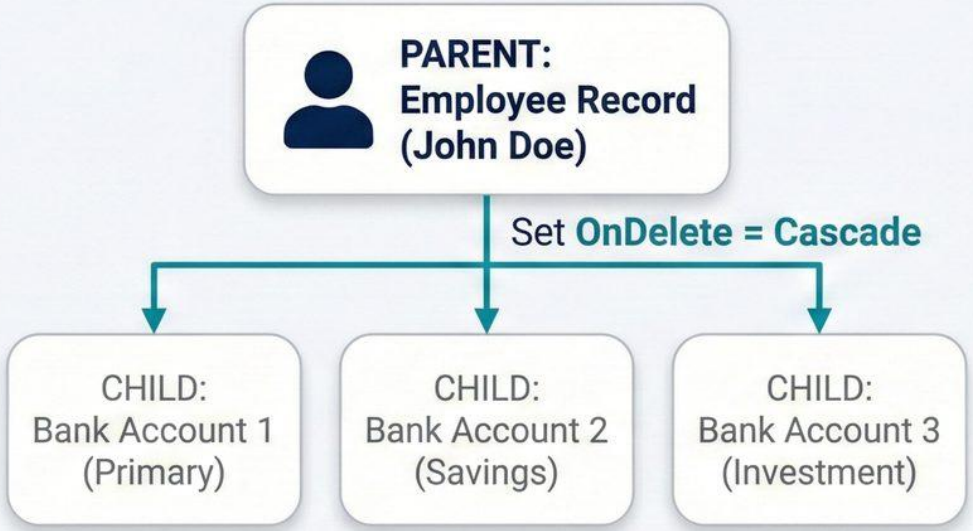
Use **Restricted** whenever children must never exist without the parent being there.

truestack.dev

Understanding OnDelete = Cascade

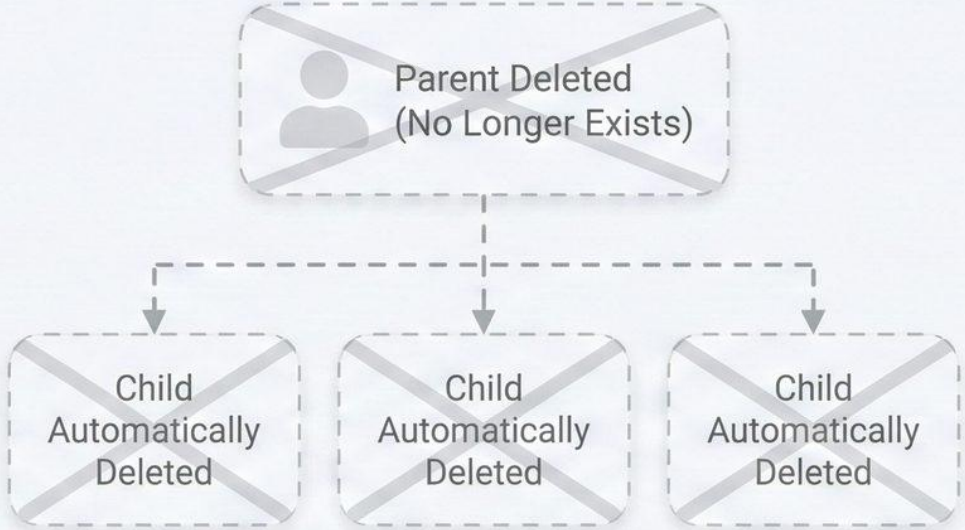
Automatic Child Record Cleanup with Parent Deletion

BEFORE: Parent & Dependent Children



ACTION:
Delete Employee Record

AFTER: Automatic Cascade Deletion



Result: Database stays clean, no orphaned records.

CASCADE CHAINING



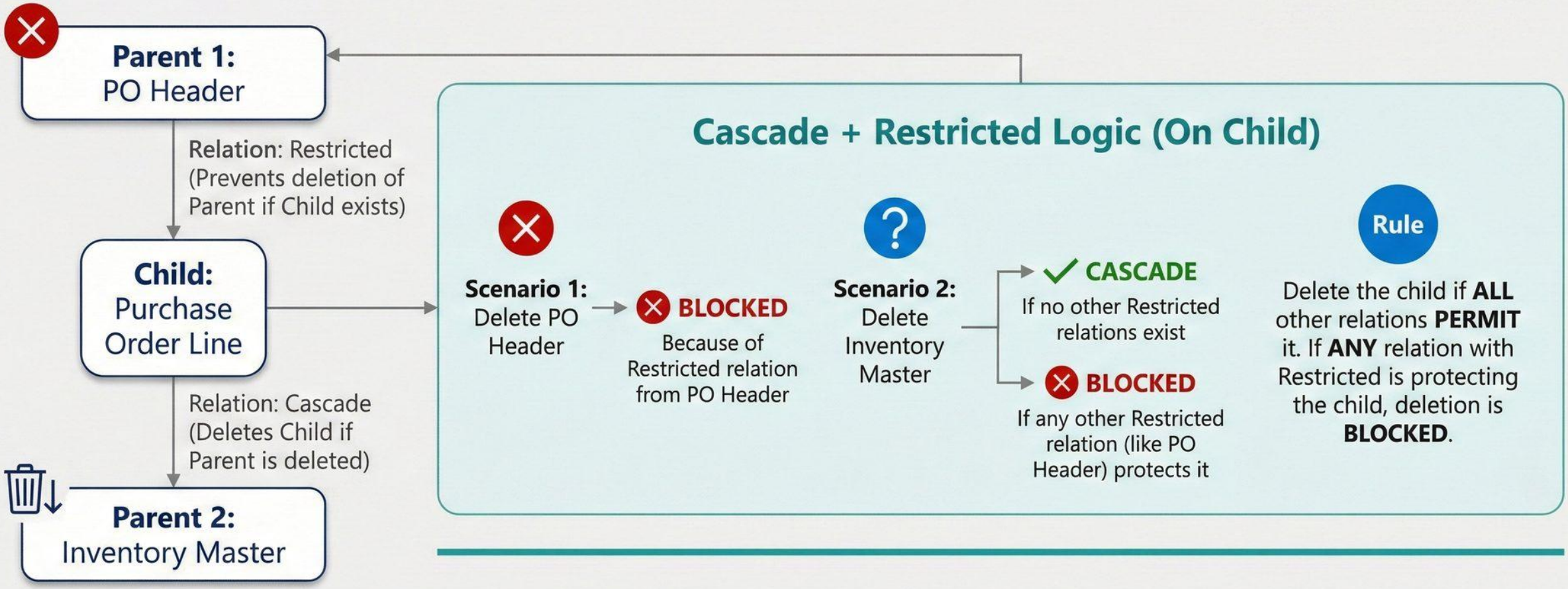
Deleting Employee cascades through entire chain:
Bank Accounts & Payment Records disappear. Complete cleanup.

TAKEAWAY: Use Cascade when children are completely dependent on the parent and should not exist without them.

truestack.dev

Cascade + Restricted Delete Behavior

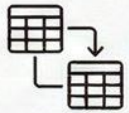
Handling complex scenarios with multiple parent relationships



Use only when business logic requires conditional cascading based on multiple parent dependencies; start with plain Cascade or Restricted.

truestack.dev

Standardized Checklist for Configuring Parent-Child Relationships

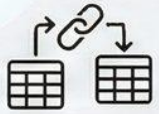
1
1. Identify Parent & Child Tables

Parent is master (e.g., Vendor, Employee). Child is dependent (e.g., Exclusion List, Bank Account).

2
2. Create/Extend Child Table

Add fields specific to that child domain on the child table.

3
3. Add Foreign Key Field

This field holds the parent ID on the child table.

4
4. Create ForeignKey Relation

Create relation on the child table pointing to the parent.

5
5. Set RelationshipType

Use *Composition* for dependent (default), or *Association* for independent children.

6
6. Set Validate

Set to *Yes* to enforce parent existence, *No* to allow orphans (rare).

7
7. Set OnDelete Behavior

Choose *Cascade* (cleanup), *Restricted* (prevent), *Cascade+Restricted* (conditional), or *None*.

8
8. Test Deletion Behavior

Create sample parent & child records, then delete the parent to verify system behavior.

Goal: Consistency. Follow this standardized checklist for every relationship to ensure consistent and predictable system behavior.

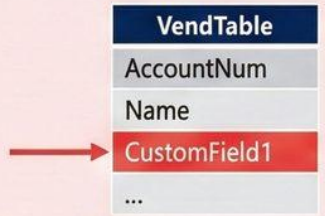
truestack.dev

Four Critical Relationship Design Mistakes to Avoid

In Microsoft Dynamics 365 Finance & Operations

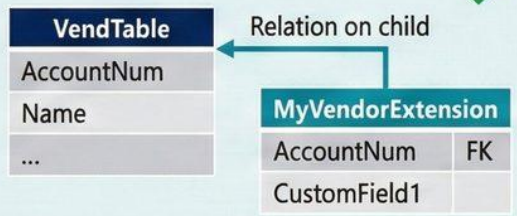
Mistake 1: Modifying master tables just to add relationships

INCORRECT



Adding custom field to standard VendTable. When Microsoft releases an update, you've got a conflict. You can't upgrade.

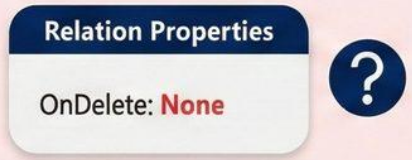
CORRECT



Create a separate child table. Relations stay on child tables. Master tables remain untouched.

Mistake 2: Relying on None for OnDelete

INCORRECT



"SQL will handle it." Behavior is implicit and unclear to future developers. Maybe it will, maybe it won't.

CORRECT



Explicitly set OnDelete to either Cascade or Restricted. State your intention clearly in the metadata.

30px

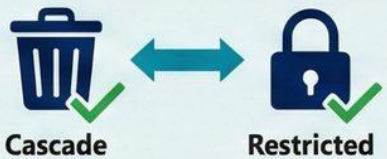
Mistake 3: Using Restricted vs. Cascade incorrectly

INCORRECT

Used Restricted
User blocks deletion, gets confused. Business logic requires automatic cleanup.

Used Cascade
Users are shocked when deletion cascades and data disappears. Business says "keep archived".

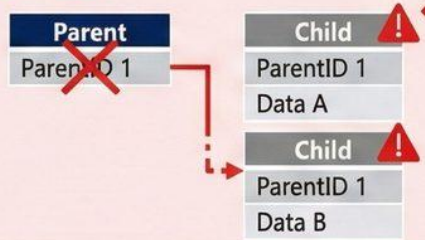
CORRECT



Understand your business rules and choose the right behavior accordingly.

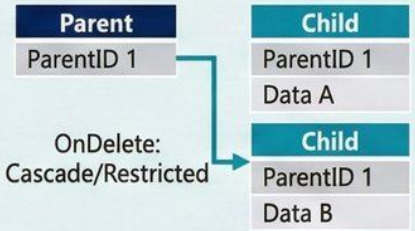
Mistake 4: Ending up with orphaned child records

INCORRECT



Parent deleted. Children are left hanging. No parent ID anymore. Reports fail. Data is corrupted.

CORRECT



Cascading or restricting prevents this. Always set OnDelete explicitly.

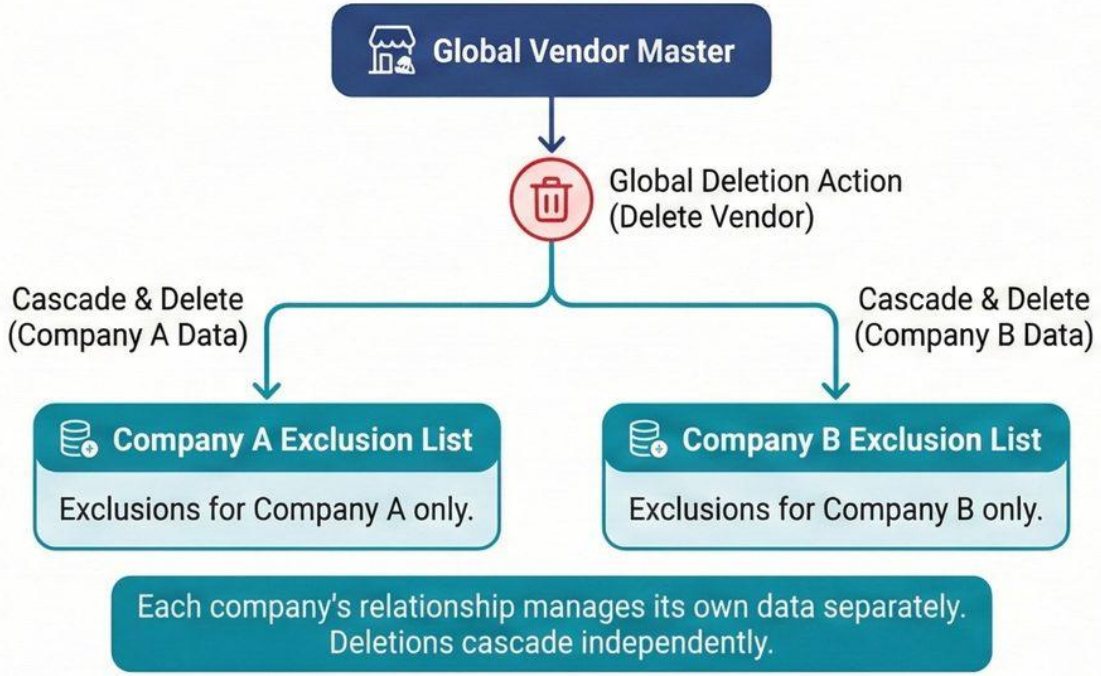
Explicitly design relationships and deletion behaviors in child tables to ensure data integrity and upgradability.

ifdestack.dev

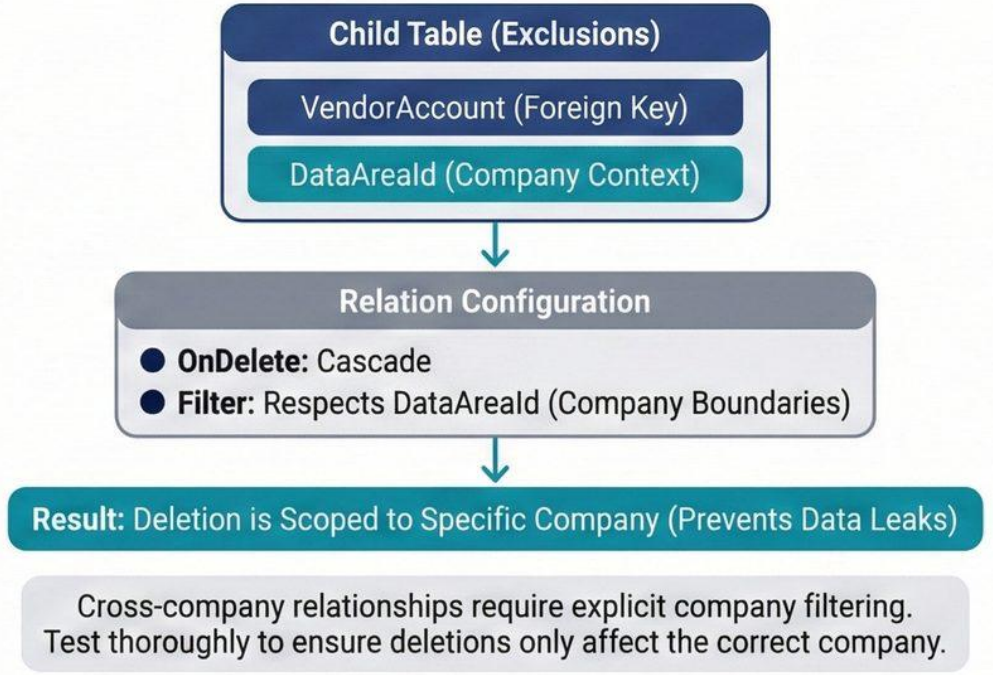
Cross-Company Vendor Deletion and Exclusion Cascading

Understanding independent cascading in multi-company systems

Conceptual Flow: Global Deletion and Company Independence



Technical Implementation: Respecting Company Boundaries



Key Takeaways & Best Practices

- Explicitly configure relations to handle company-specific cascading.
- Do not assume automatic cross-company handling by the system.
- Thorough testing is crucial to confirm correct scope and prevent unintended data loss.

truestack.dev

Testing Relationship Deletion Behavior

Test One: Parent without Children



Action: Delete parent record (no child links).

Outcome: Deletion **SUCCEEDS** (OnDelete setting irrelevant).

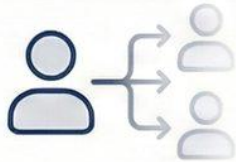
Test Two: Parent with Children (Restricted)



Action: Delete parent record (OnDelete = Restricted).

Outcome: Deletion is **BLOCKED**. System error raised (verify error message clarity).

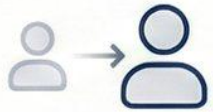
Test Three: Parent with Children (Cascade)



Action: Delete parent record (OnDelete = Cascade).

Outcome: Deletion **SUCCEEDS**. Children auto-delete (verify removal in DB).

Test Four: Direct Child Deletion



Action: Delete a child record directly.

Outcome: Deletion **SUCCEEDS**. Parent is unaffected (verify parent remains).

Verification & Best Practices



Realistic Data & Partial Cascades: Create multiple children, trigger cascades, verify all are deleted. Check that not all cascades happen instantly.



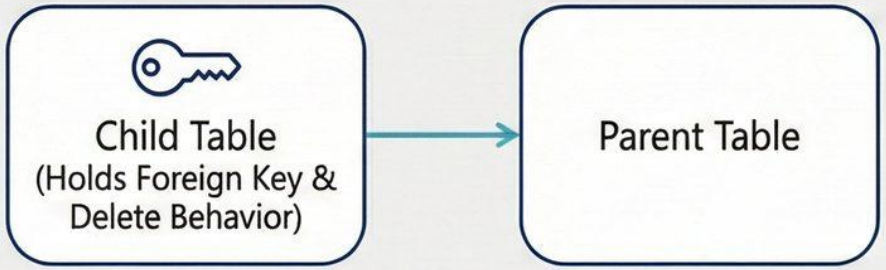
Sandbox Environment: Always run tests in a sandbox before production to prevent unexpected surprises.

truestack.dev

Recap: Table Relationships & Deletion Behavior (Topic 2.3)

Core Concepts & OnDelete Options

Child-Parent Relationship Principle



OnDelete Behavior Comparison

Option	Behavior	Typical Use Case
None (Passive)	Leaves to SQL (Not Recommended)	Minimal config, passive
Cascade	Auto-deletes children with parent	Dependent data (e.g., employee bank accounts)
Restricted	Blocks parent delete if children exist	Protecting data (e.g., sales order lines)
Cascade + Restricted	Conditional cascading	Complex scenarios

Configuration & Scenarios

Relationship Configuration Checklist

1. Identify Parent & Child
2. Add Foreign Key to Child
3. Create ForeignKey Relation
4. Set RelationshipType
5. Set Validate
6. Set OnDelete
7. Test Behavior

Practical Scenarios

- Vendor Exclusions (Stand-alone table linked to master)
- Order Management Hierarchies (Protecting header/lines)
- Cross-Company Relationships (Complex linking)

Anti-Patterns & Next Steps

Anti-Patterns to Avoid

- ✕ Modifying master tables unnecessarily
- ✕ Relying on 'None'
- ✕ Choosing wrong delete behavior
- ✕ Creating orphaned records

Testing Requirement

Always test deletion with realistic data before deployment

Next Step: Programming Fundamentals

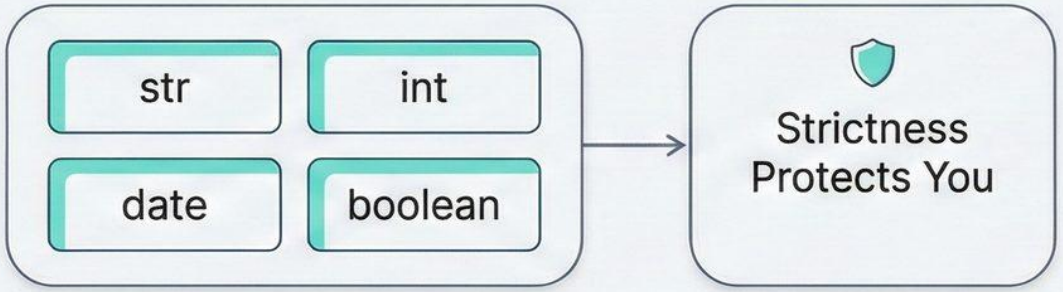
Topic 3.1: X++ Syntax, Loops, and Variables (Writing code, statements, etc.)

truestack.dev

Topic 3.1: X++ Syntax, Loops, and Variables

Introduction to Dynamics 365 Programming

X++ is Strongly Typed



Explicit Type Conversion Required



Learning Path

1. How to work with types
2. How to convert between types
3. How to write conditional logic
4. How to execute code using runnable classes

These fundamentals are essential for everyday X++ development.

truestack.dev

X++ Strict Type System & Common Types

Common X++ Types

	Type Keyword	Stores	Examples
	`str`	Text	“Dynamics 365”, “Item-A123”
	`int`	Whole Numbers	42, 100, -5, 0
	`real`	Decimals	199.99, 3.14159, -0.5
	`date`	Calendar Dates	2023-11-23
	`boolean`	True/False	true, false

Strict Type Enforcement

Weakly-Typed Language (Runtime Error)

```
// Weakly-typed example
var myName = “Alice”;
myName = 12345; // No immediate error
console.log(“Hello, ” + myName);
```



⚠ **RESULT:** "Hello, 12345"
(Unintended, no error shown)

System tries to interpret, leading to potential bugs.

X++ (Compile-Time Protection)

```
// X++ example
str customerName;
customerName = 12345; // Compiler rejects this
```

⊗ **COMPILE ERROR:** Type mismatch.
Cannot implicitly convert 'int' to 'str'.

Compiler prevents the bug *before* it happens. Must convert first.

Takeaway: X++ strict typing prevents entire classes of bugs by ensuring variables only store their declared types and enforcing conversions.

truestack.dev

X++ Conversion Functions & Type Casting

Explicitly convert between data types for type safety in X++

1. int2str (Integer to String)

Converts an integer value to its string representation.

```
int number = 42;
str text = int2str(number);
// '42'
```

2. any2str (Catch-all to String)

A catch-all function that converts almost any data type to a string.

```
anytype value = 'mixed';
str result = any2str(value);
// Useful when type is uncertain
```

3. num2str (Real to String)

Converts real (decimal) numbers to string format.

```
real amount = 123.45;
str display = num2str(amount, 0, 2, 1,
// Formats real number to string
```

4. strfmt (Formatted String Output)

Powerful function for creating complex, formatted output using placeholders.

```
str first = 'Value', second = 'Data';
str output = strfmt('%1: %2 and %3', 'Info', first, second);
// 'Info: Value and Data'
```

5. str2date (String to Date)

Converts a text string to a date value based on a sequence parameter.

```
str dateText = '15/1/2024';
date myDate = str2date(dateText, 123);
// Sequence 123 is Day-Month-Year
```

Takeaway: Always use explicit conversion functions when data types do not match; never assume implicit conversion will work.

truestack.dev

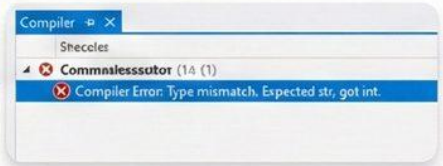
Type Conversion in X++: A Practical Example

Understanding why explicit type conversion is crucial for framework APIs like the Info log.

Incorrect Approach ❌

```
// Incorrect
info(42);
```

⚠ *Compiler Error: Type mismatch. Expected str, got int.* →



Correct Approach ✅

```
// Correct
info(int2str(42));
```

✅ *Successful Execution: int2str converts integer 42 to string '42', then passes it to info.* →



✅ **Best Practice: Always Convert Explicitly**

Never rely on implicit conversion happening magically. Explicit conversion makes your intention crystal clear to future developers who are reading your code.

truestack.dev

Boolean Expressions & Ternary Operators

Boolean Expressions

Evaluate to either true or false. Essential for controlling program flow in conditionals.



The Ternary Operator: Compact if-else

```
condition ? value-if-true : value-if-false
```

condition (evaluates to Boolean) **value-if-true** (result if condition is true) **value-if-false** (result if condition is false)

```
str category = (age > 18) ? "Adult" : "Minor";
```

condition: age > 18 value-if-true: 'Adult' value-if-false: 'Minor'

If age is 25 (true), category becomes 'Adult'. If age is 15 (false), category becomes 'Minor'.

Critical Rule: Type Consistency

Both branches of the ternary expression **must produce the same data type** as the destination variable.

✅ Correct

```
str category = (age > 18) ? "Adult" : "Minor";
```

Both branches return string, matching the destination 'str' type.

❌ Incorrect

```
str category = (age > 18) ? "Adult" : 100;
```

⚠ Type mismatch error! Branches mix string ('Adult') and integer (100).

Strict type requirements in ternary operators prevent bugs by ensuring consistent return values.

truestack.dev

Type Safety in Comparisons

Valid Comparison (Integer)

```
int quantity = 48;

// Comparing integer to integer
if (quantity == 48)
{
    // Valid comparison
}
```

Valid.
Both are integers.

Valid Comparison (String)

```
str status = 'Active';

// Comparing string to string
if (status == 'Active')
{
    // Valid comparison
}
```

Valid.
Both are strings.

Invalid Comparison (Type Mismatch)

```
int quantity = 48;
str status = '48';

// Comparing integer to string
if (quantity == status)
{
    // Compiler Error!
}
```

Invalid.
Compiler rejects it outright. Never mix types.

The '100' vs. 100 Dilemma

As a String ('100')



Text.
Sequence of characters.

Fundamentally different things.
X++ forces explicit conversion.



As a Number (100)



Quantity.
Numeric value.

X++ strictness prevents subtle bugs by requiring explicit type usage in comparisons. Always convert types to match before comparing.

truestack.dev

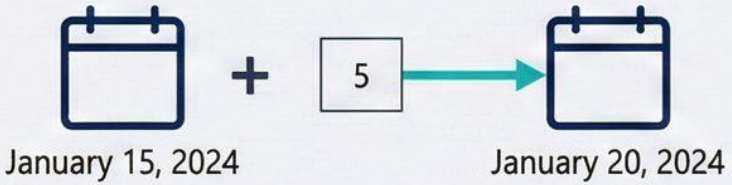
X++ Date Handling: Functions and Arithmetic

Specialized functions for robust date manipulation

Date-to-Number & Arithmetic

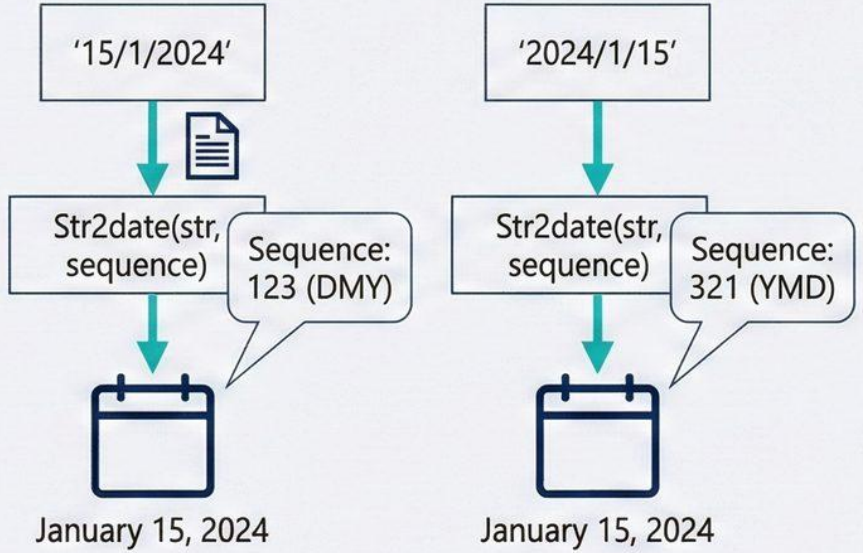


Dates are internally stored as integers.



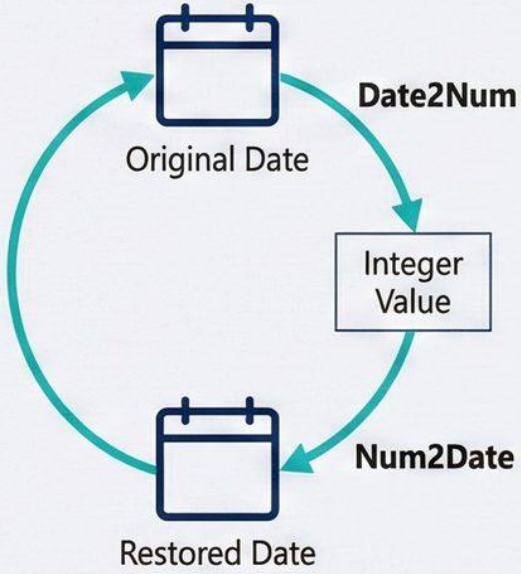
Add integers to move dates forward or backward.

String-to-Date Conversion (Str2date)



Sequence parameter interprets text format.

Reversible & Safe Conversion



Predictable handling: Convert, then revert to original date.

Consistent type handling and correct function usage ensure reliable date logic in X++.

truestack.dev

X++ Conditional Structures

If-Else: Most Flexible

Use for:

- Complex conditions
- Ranges
- Combinations of logic

```
if (quantity < 10)
{
    /* do this */
}
else if (quantity < 50)
{   /* do that */
}
} else {
    /* do this other thing */
}
```

Can handle any condition you can express

Switch: Discrete Values

Use for:

- Discrete values
- Limited set of options

```
switch (status)
{
    case Active:
        break;
    case Inactive:
        break;
    case Pending:
        break;
}
```

Cleaner, more readable for fixed values

Ternary: One-liner

Use for:

- Assign one of two values
- Simple decisions

```
str label = active ? "Yes" : "No";
```

Compact for single assignments

Choose the structure based on complexity to make your code readable for future developers.

truestack.dev

Switch Statements: The Importance of 'break'

Understanding Fall-Through Behavior and How to Fix It

The Problem (Fall-Through) ❌

```
// INCORRECT: Missing break statements
str color = 'Red';
real price;

switch (color)
{
    case 'Red':
        price = 10;
    case 'Blue':
        price = 15;
    case 'Green':
        price = 20;
}

// Final Price: 20 (Incorrect)
```

Execution continues, break statements

Execution stops to start here.

Execution continues, overwriting the price multiple times. Final result is 20, not 10.

⚠ Execution falls through to the next case automatically if 'break' is omitted.

The Solution (Using break) ✅

```
// CORRECT: With break statements
str color = 'Red';
real price;

switch (color)
{
    case 'Red':
        price = 10;
        break; // Exit switch
    case 'Blue':
        price = 15;
        break;
    case 'Green':
        price = 20;
        break;
}
// Final Price: 10 (Correct)
```

Execution stops here.

Each case executes independently and stops cleanly. Final result is correctly 10.

✓ The 'break' statement ensures execution exits the switch block after the matching case.

Always include a 'break' statement after each case to prevent unintended fall-through execution and logic errors.

truestack.dev

X++ Loop Structures: For, While, and Do-While

For Loop

Use when the number of iterations is known.

```
for (int i = 0; i < 10; i++)
{
   // Operation to repeat
}
```

Runs exactly 10 times (i from 0 to 9)

- Ideal for precise repetition.

While Loop

Keeps going as long as a condition stays true.

```
while (!queue.isEmpty())
{
   processNextItem();
}
```

Continues until the condition becomes false.

- Use when iterations are uncertain.

Do-While Loop

Guarantees at least one execution.

```
do
{
   // Execute at least once
} while (condition);
```

Executes the block once before checking the condition.

- Useful for mandatory initial action.

Choose the loop structure that best fits your specific iteration requirements.

truestack.dev

Practical Example: Loops in Action

Automating Repetition and Eliminating Duplication

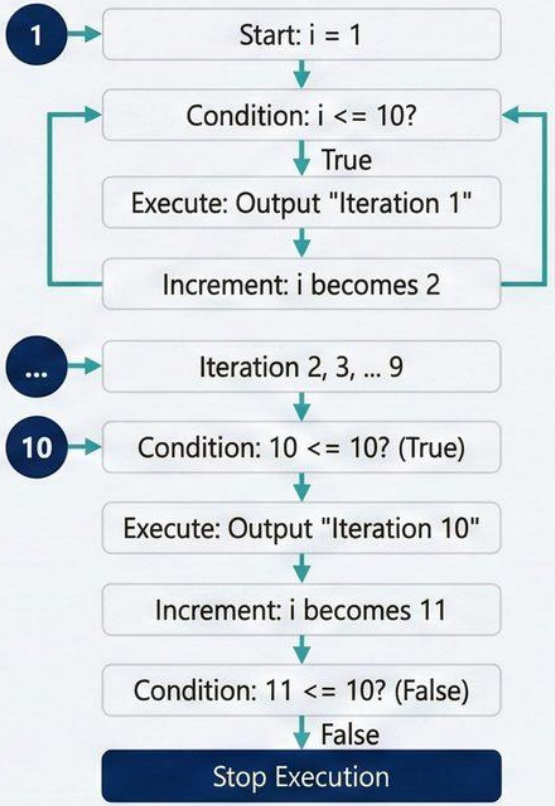
Code Example: The `for` Loop

Increments “i” by 1 after each iteration

```
// Loop from 1 to 10
for (int i = 1; i <= 10
for (int i = 1; i <= 10; i++)
{
    // Output current iteration number
    info(strFmt("Iteration %1", i));
}
```

Increments “i” by 1 after each iteration

Execution Flow & Output



Output sequence: ‘Iteration 1’, ‘Iteration 2’, ..., ‘Iteration 10’

Key Benefits of Loops

- **Eliminates Code Duplication:** Write logic once, run multiple times.
- **Automates Repetition:** Handles collections, database updates, and repetitive operations efficiently.
- **Essential Programming Tool:** Fundamental for concise, maintainable, and powerful code.

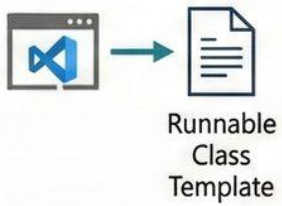
Loops are indispensable for writing efficient code by automating repetitive tasks and preventing redundancy.

truestack.dev

X++ Runnable Classes in Visual Studio

Streamlined Execution for Testing, Updates, and Isolated Operations

1. Create Runnable Class



Use built-in Runnable Class template in Visual Studio.

2. Add main Method

```
public static void main(Args _args)
{
    // Entry point
}
```

Implement the required public static entry point signature.

3. Write Business Logic



Place code for updating records, selecting data, or calculations.

4. Build Project



Compile to ensure syntactic validity.

5. Set as Startup Object



Designate the runnable class as the entry point.

6. Run (F5)



Executes within the D365 runtime with full system access.

Runnable classes enable direct execution and debugging without navigating the full application UI.

truestack.dev

Runnable Class Example: MyDataUpdate

CODE EXAMPLE: MyDataUpdate.xpp

```
// Runnable Class Example
class MyDataUpdate
{
    // Main method as entry point
    public static void main(Args _args)
    {
        SalesTable salesTable;

        // Start a transaction
        ttsbegin;

        // Select for update: Find sales order for Cust001
        select firstOnly forUpdate salesTable
            where salesTable.CustAccount == 'Cust001';

        if (salesTable)
        {
            // Update CustAccount field
            salesTable.CustAccount = 'Cust002';

            // Save changes to the database
            salesTable.update();
        }

        // Commit the transaction
        ttscommit;

        // Log completion message
        Info('Update complete.');
    }
}
```

Entry Point

Find Record & Lock for Update

Modify Data

Save Changes

Log Message

PURPOSE & BENEFITS

- ✓ **Direct Execution**: Code runs directly within the D365 runtime.
- ✓ **Full Database Access**: Interact with data without UI navigation.
- ✓ **Isolated Testing**: Validate logic or perform data corrections securely.
- ✓ **Quick Deployment**: Ideal for one-time migration scripts or data fixes.

Key Takeaway: Runnable classes provide a streamlined, UI-free way to execute X++ logic and interact with the database for testing or maintenance tasks.

truestack.dev

Critical X++ Syntax Mistakes to Avoid

1. Forgetting Type Conversion

Always convert non-string values before passing them to methods that expect a 'str'.

❌ Incorrect

```
// INCORRECT
int myValue = 123;
info(myValue); // Expects string
```

✓ Correct

```
// CORRECT
int myValue = 123;
info(int2str(myValue));
```

2. Missing 'break' in 'switch'

Always include 'break' to prevent unintended execution of subsequent cases.

❌ Incorrect — Fall-through behavior

```
// INCORRECT
switch(status)
{
    case 1:
        result = "Pending";
    case 2:
        result = "Approved";
}
```

✓ Correct

```
// CORRECT
switch(status)
{
    case 1:
        result = "Pending";
        break;
    case 2:
        result = "Approved";
        break;
}
```

3. Incompatible Ternary Branches

Both branches of a ternary expression must produce the same type as the target.

❌ Incorrect — Types must match

```
// INCORRECT
str result = isActive ? "Active" : 0; // 'Active' is str, 0 is int
```

✓ Correct

```
// CORRECT
str result = isActive ? "Active" : "Inactive"; // Both are str
```

4. Runnable Class without 'main'

Runnable classes require a 'main' method to define where execution begins.

❌ Incorrect — No entry point

```
// INCORRECT
class MyJob
{
    // Missing main() method
}
```

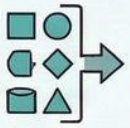
✓ Correct

```
// CORRECT
class MyJob
{
    public static void main(Args _args)
    {
        // Execution starts here
    }
}
```

Avoiding these common syntax mistakes ensures cleaner, more reliable X++ code execution.

truestack.dev

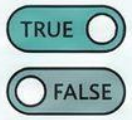
Recap of **Topic 3.1: X++ Fundamentals**



Strong Typing & Type Conversion

X++ is **strongly typed** (string, integer, real, date, boolean). Compiler enforces types strictly.

Conversion Functions:
int2str, **any2str**, **strfmt** (to strings)
str2date (string to date)



Boolean Logic & Operators

Boolean expressions evaluate to **true** or **false**.

Ternary Operators (? :)
Compact inline **if-else**. **Critical**: Both branches must return the same type.*



Conditional Structures

if-else: Complex logic.
switch: Discrete value comparison.
Ternary: Simple one-liners.

⚠ **switch** requires 'break' after each case to prevent fall-through overwriting values.



Loops

for loop: Fixed number of iterations.
while loop: Condition-based repetition.
do-while loop: Execute first, then check condition.



Runnable Classes (Jobs)

1. Create runnable class.
2. Add main() method.
3. Write business logic.
4. Build project.
5. Set as Startup Object.
6. Run.

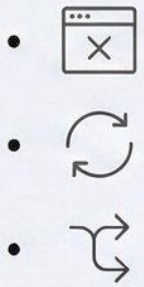
Foundational skills for everyday X++ programming. Next: Topic 3.2: Object-Oriented Programming - Classes and Methods.

truestack.dev

Topic 3.2: OOP in X++

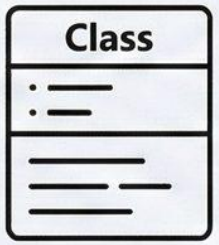
From Procedural to OOP

Previous Topics



Variables, Loops, Conditionals

New Paradigm: Object-Oriented Programming



Professional code organization. Reusable, testable, and maintainable.

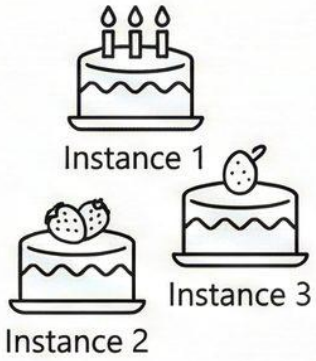
The Blueprint Analogy

Class (Blueprint)



Defines data and methods (ingredients & instructions).

Instances (Objects)



Individual entities created from the same blueprint, each with its own state.

In This Topic

1. Designing classes properly.
2. Controlling access (who can access what).
3. Defining methods (actions).
4. Using Inheritance and Interfaces.
5. Building sophisticated, reusable code.

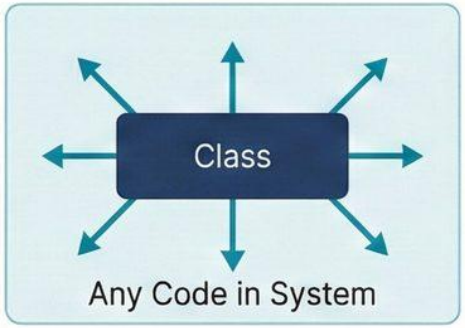
truestack.dev

Access Modifiers & Encapsulation

Controlling visibility for safer, maintainable code.

Public

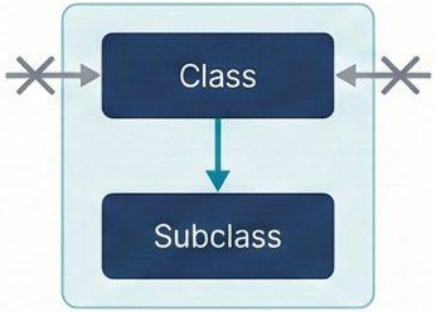
Can be called from anywhere the class is visible. Acts as the class's API or public interface.



Broadest access.

Protected

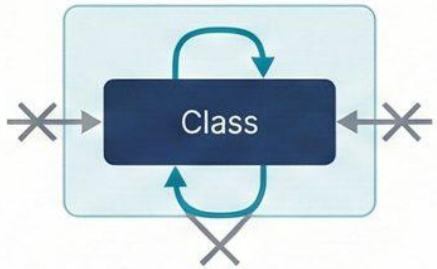
Callable only from within the class and any subclasses that inherit from it.



Restricted to inheritance hierarchy.

Private

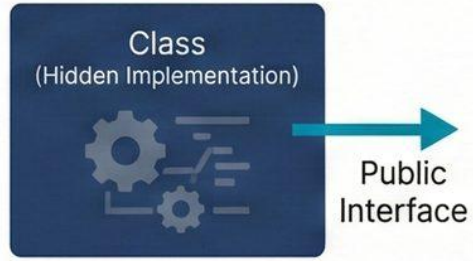
Callable only from within the same class. Used for internal helper functions.



Most restricted access.

Encapsulation

Hides implementation details and exposes only necessary parts. Ensures control, safety, and easier maintenance.



Stable public interface, hidden internals.

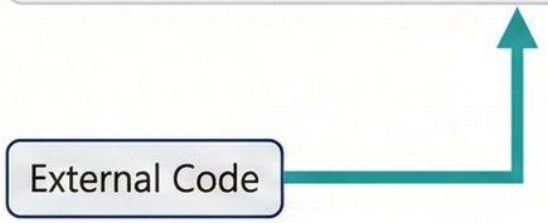
By restricting access, you protect your code from breaking changes and make it easier to maintain over time.

truestack.dev

Access Modifiers in X++: A Practical Example

Public

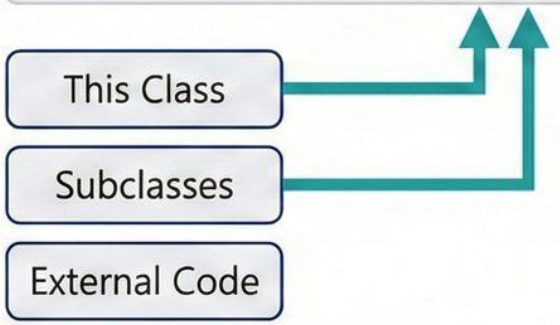
```
public void searchVendor() {
    ...
}
```



Callable from any code anywhere.
Main interface for external consumers.

Protected

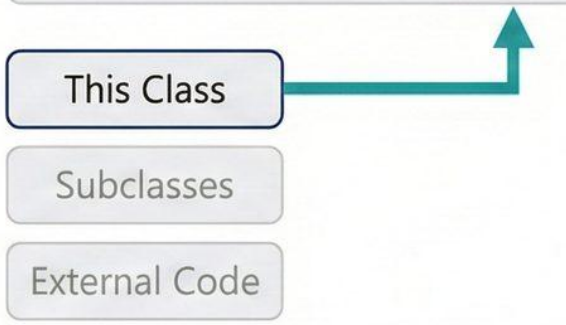
```
protected boolean validateAmount() {
    ...
}
```



Callable only from this class and its subclasses. Shared validation logic.

Private

```
private Cache getVendorCache() {
    ...
}
```



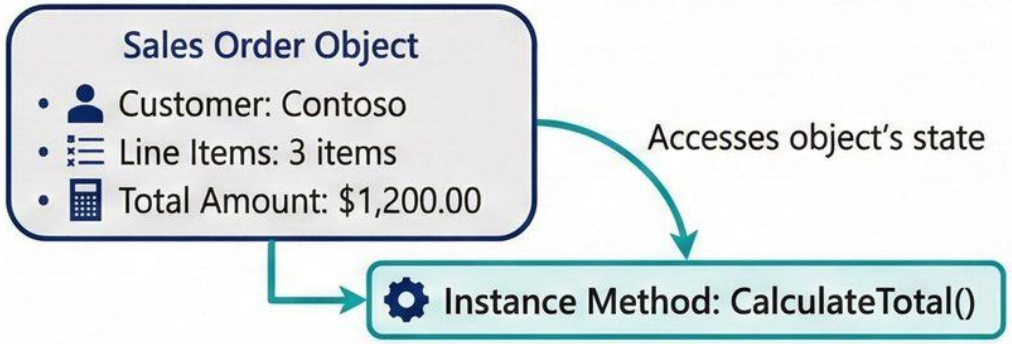
Callable only from this class itself.
Internal helper, invisible to others.

Key Principle: The Principle of Least Privilege
Expose only what's absolutely necessary. Grant minimum access needed to protect code and simplify future changes.

Note: 'internal' is rare, mostly for C# interoperability.

Instance vs. Static Methods

Instance Methods



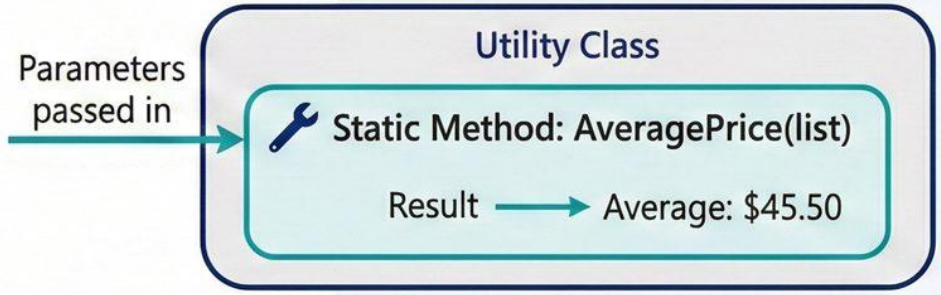
- Operate on a specific object's state.
- Can access all data of that instance (e.g., customer, items).
- Each object has its own unique state.
- Must be called on an created object instance.

```
class SalesOrder {
    public str Customer;
    public List<LineItem> LineItems;
    // ... other state fields

    public real CalculateTotal() {
        // Accesses instance state (this.LineItems)
        real total = 0;
        // logic to sum line items...
        return total;
    }
}
```

Accesses instance state

Static Methods



- Belong to the class itself, not any instance.
- Cannot access instance variables directly.
- Work purely with passed parameters.
- Called directly on the class name, no object needed.
- Typically for utility functions.

```
class MathUtils {
    // No instance state

    public static real AveragePrice(List<real> prices) {
        // Works purely with parameters
        real sum = 0;
        // logic to calculate average from prices list...
        return sum / prices.Count;
    }
}
```

No instance state

Works purely with parameters

Conclusion: Use instance methods for object-specific data and state; use static methods for utility behavior independent of any specific object.

truestack.dev

Instance vs. Static Methods: A Practical Comparison

Instance Method (e.g., Order.getTotal)

Requires an object instance to operate. Accesses specific data within that object.

```
// Create an Order object
Order ord = new Order();
// Call the instance method
ord.getTotal();
```

Accesses object-specific data/state

Static Method (e.g., MathHelper.average)

Called directly on the class. No object instance needed. Operates on passed data.

```
// Call the static method directly
// Pass data as arguments
MathHelper.average(valuesList);
```

General-purpose utility, no instance state

Use instance methods for object-specific state; use static methods for general-purpose utilities to clarify intent.

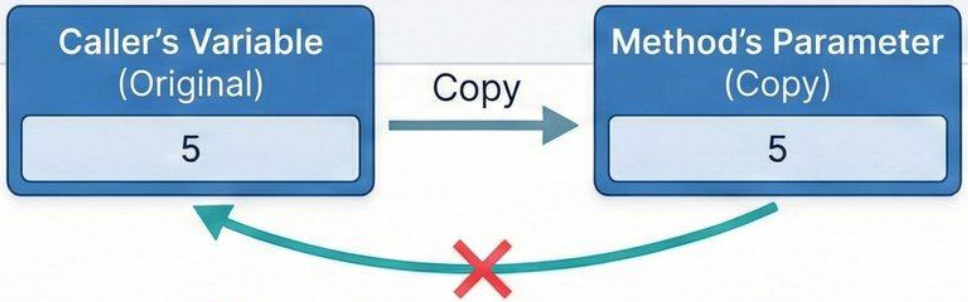
truestack.dev

Parameter Passing: Value vs. Reference

Pass by Value

Value Types

integers, strings, reals, dates



✓ No effect on original variable

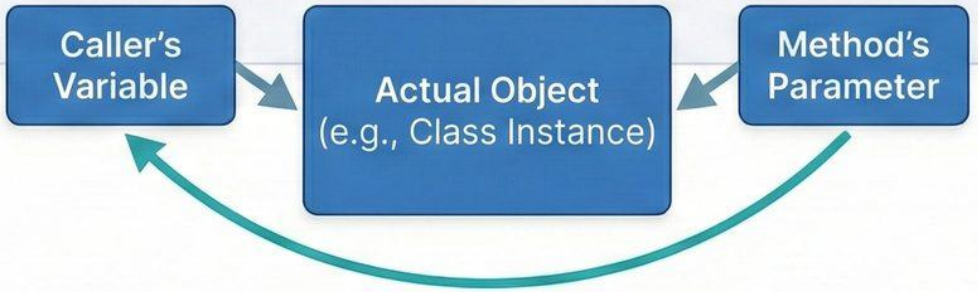
✓ Performance

Small & simple. Cheap to copy.

Pass by Reference

Reference Types

tables, classes, lists, maps



✓ Changes visible to caller

✓ Performance

Complex & large. Expensive to copy, passing reference is efficient.

Understanding this distinction is critical; pass reference types or return modified values if you expect method changes to affect the caller.

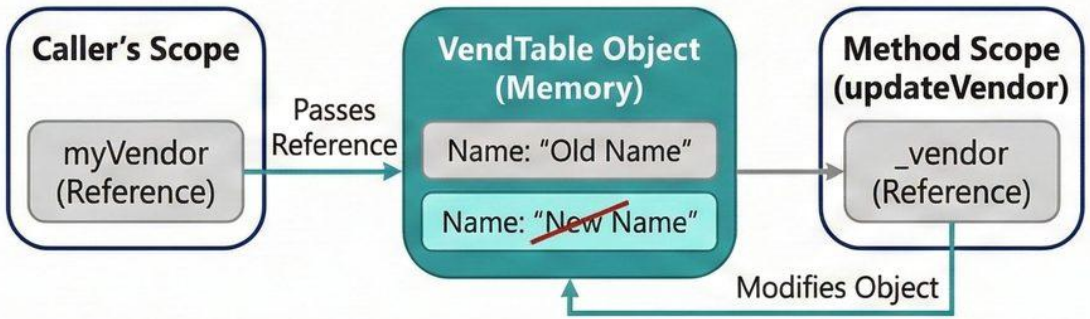
truestack.dev

Reference Semantics: Concrete Examples

Example 1: Reference Type (VendTable)

```
// Caller code
VendTable myVendor;
myVendor.Name = "Old Name";
updateVendor(myVendor);
// After call: myVendor.Name is "New Name"

// Method definition
void updateVendor(VendTable _vendor)
{
    _vendor.Name = "New Name";
}
```

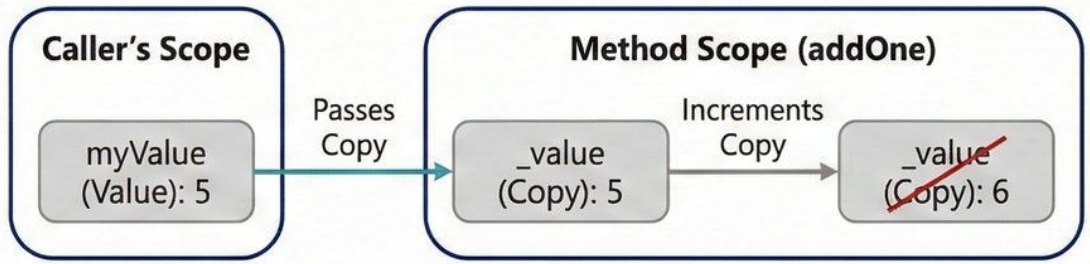


✓ **Change persisted**: Caller sees the new name.

Example 2: Value Type (Integer)

```
// Caller code
int myValue = 5;
addOne(myValue);
// After call: myValue is still 5

// Method definition
void addOne(int _value)
{
    _value = _value + 1; // _value becomes 6
}
```

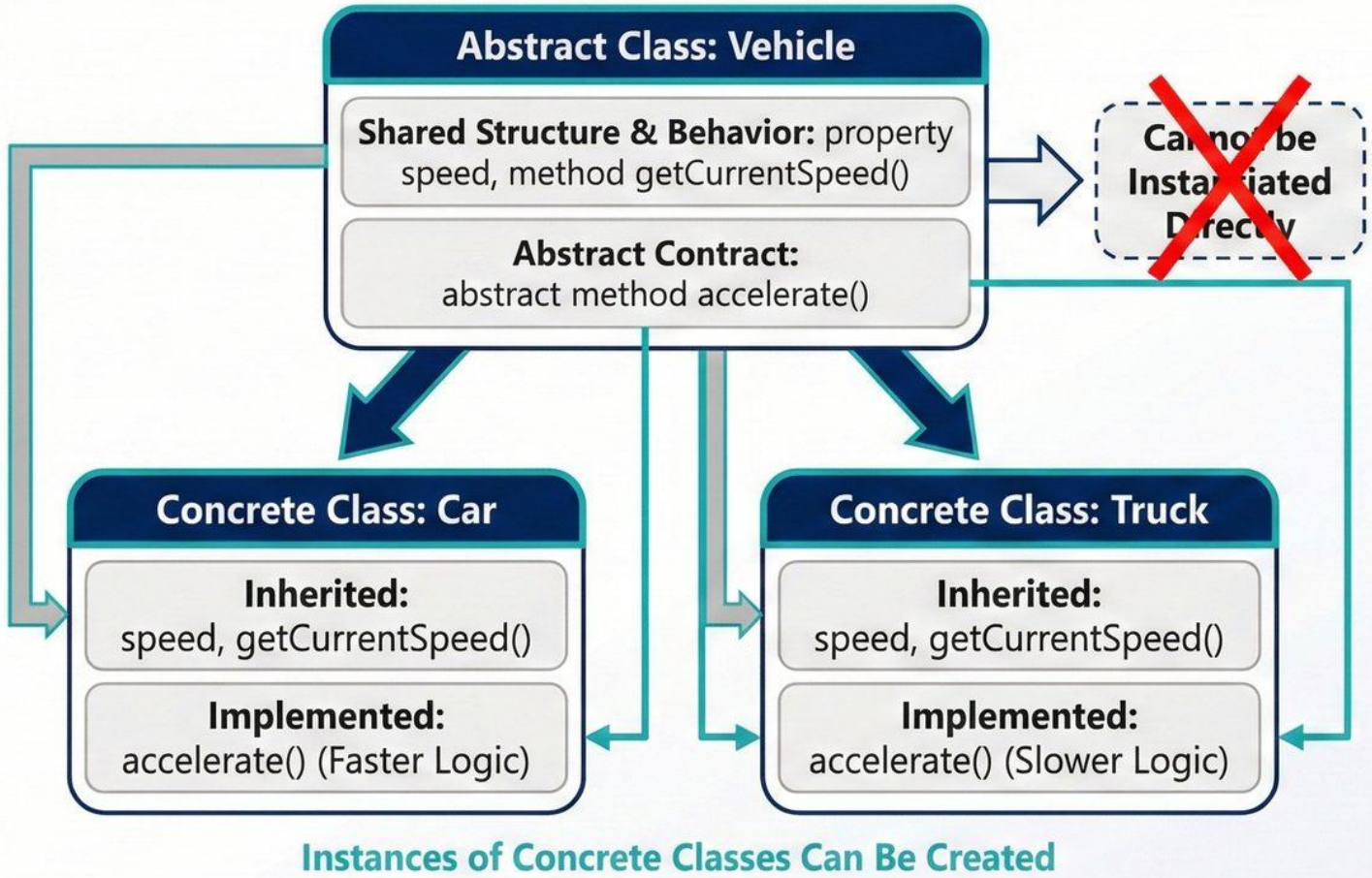


✗ **Change not persisted**: Caller sees the original value.

Distinction creates predictability: Table records passed to methods reflect changes back to the caller, while primitive types like integers do not, making code easier to reason about.

truestack.dev

Abstract Classes



Why Use Abstract Classes?

Enforce Consistency: Every child class must implement the same abstract methods.

Provide Shared Code: Child classes inherit common methods automatically, avoiding duplication (e.g., getCurrentSpeed).

Abstract classes define a common structure and contract for related classes while preventing direct instantiation and enabling code reuse.

truestack.dev

Abstract Classes vs. Interfaces

Abstract Classes

State & Methods

- Can have state (fields to store data) ✓
- Can have private and protected methods ✓
- Can have fully implemented methods (shared code) ✓
- Children inherit directly without overriding ✓

Inheritance

- → Single inheritance only (Child extends one abstract class) ✗

Primary Use Case

- Use to share actual implementation code (e.g., Base Vehicle class with common methods for Car and Truck).

Interfaces

State & Methods

- No state at all (no fields) ✗
- All methods are purely public signatures ✗
- No implementation body (pure contracts) ✗

Inheritance

- Multiple inheritance capability (Class implements multiple interfaces) ✓

Primary Use Case

- Use to enforce a contract without sharing implementation (e.g., Drawable interface for anything capable of being drawn).

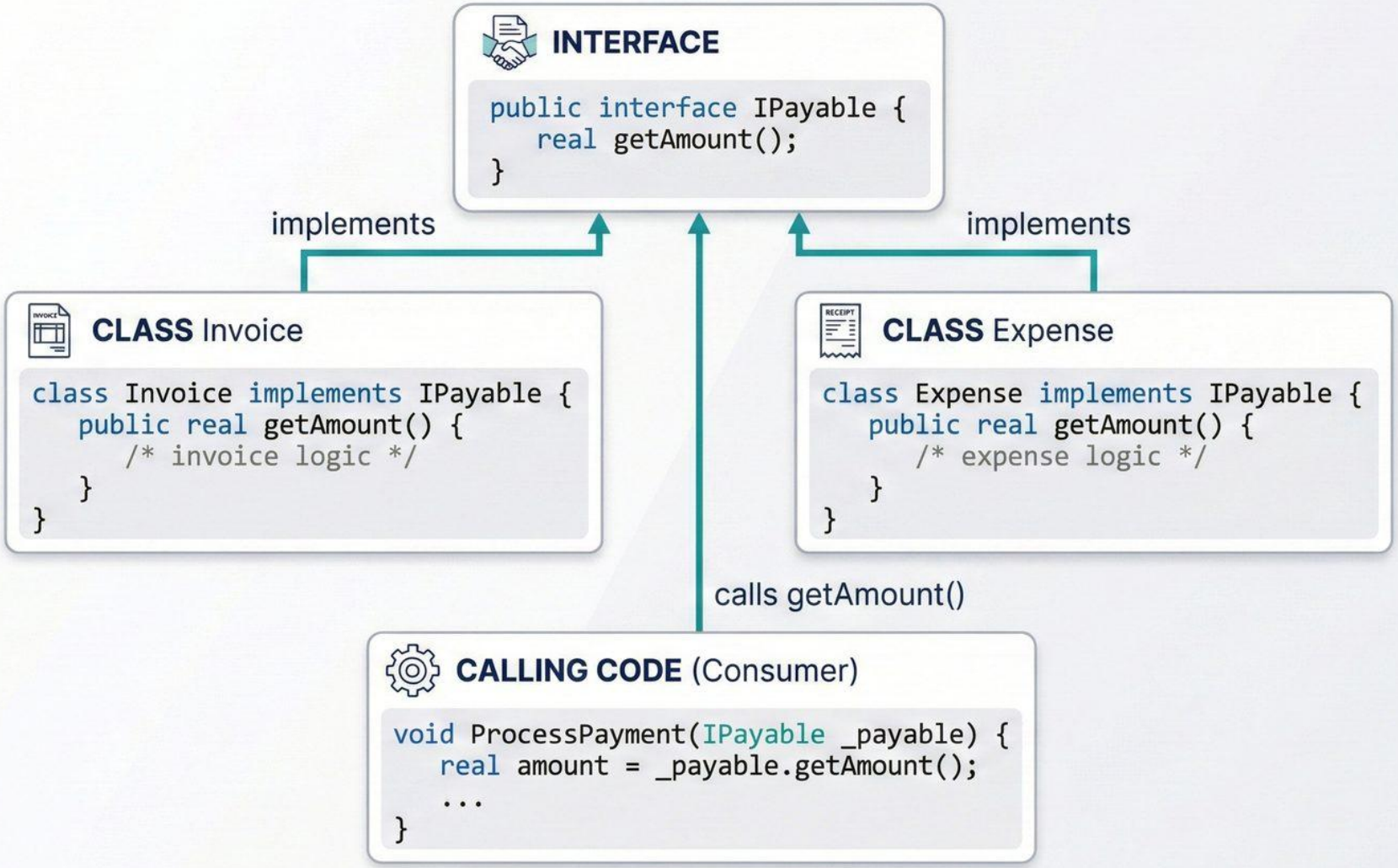
Example Scenario: Flexibility through Combination



Interfaces offer flexibility with multiple inheritance for contract enforcement, while abstract classes enable code sharing through single inheritance.

truestack.dev

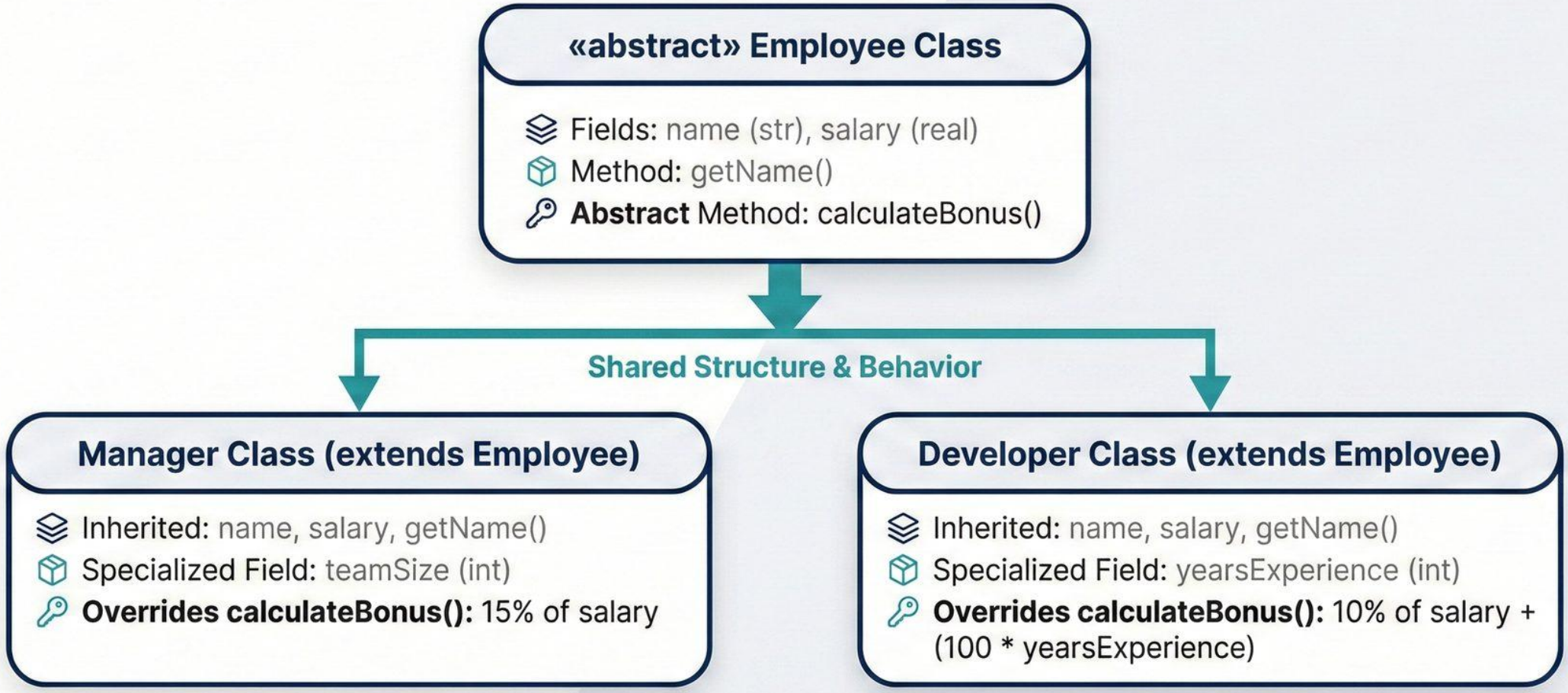
Interface Implementation Example



Consumer code interacts with the IPayable interface, oblivious to the specific concrete type (Invoice or Expense).

truestack.dev

Inheritance in Action: Employee Example



Result: Shared base class ensures implementation, but each child class provides specialized behavior for bonus calculation based on its specific state.

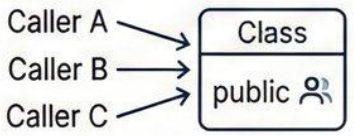
truestack.dev

4 Critical OOP Mistakes in Microsoft Dynamics 365 F&O

1. Exposing All Methods as Public

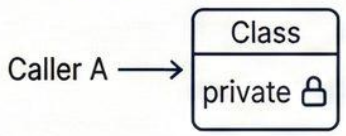
Making all methods public creates high dependencies. If you change a method, dependent code breaks.

❌ INCORRECT



Every Caller Depends on Every Method

✅ CORRECT



Keep Methods Private Unless Necessary

2. Static Methods Accessing Instance State

Static methods cannot access instance variables. The compiler rejects this. Use instance methods instead.

❌ INCORRECT

```
static void myMethod() {
   this.instanceField = 5;
   // Compiler Error
}
```

↳ Compiler Error: Cannot access instance field

✅ CORRECT

```
void myMethod() {
  this.instanceField = 5;
}
```

Instance methods can access state

3. Partial Interface Implementation

You must implement every method defined by an interface. Missing methods cause compilation errors.

❌ INCORRECT

```
interface IMyInterface {
   void method1();
   void method2();
}
```

Implemented method1, forgot method2

→ Compilation Error: Missing implementation

✅ CORRECT

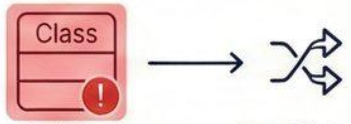
```
class MyClass implements IMyInterface {
   void method1() { ... }
   void method2() { ... }
}
```

All methods implemented

4. Overlaying Standard Classes

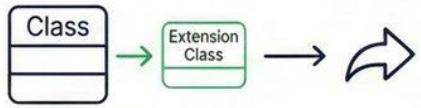
Directly modifying standard classes leads to update conflicts. Use extension classes instead.

❌ INCORRECT



Modified Standard Class → Conflict on Update

✅ CORRECT



Standard Class → Smooth Upgrade

These common mistakes are preventable by understanding OOP principles.

truestack.dev

Core OOP Design Principles for Maintainable Code

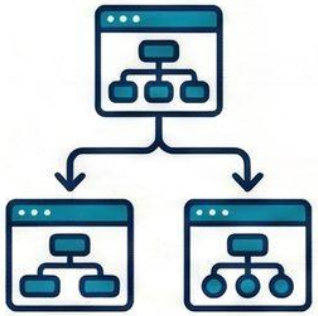
1. Encapsulation



Hide implementation details inside the class. Expose only necessary functionalities through a clean **public interface**.

→ Makes code **robust** and easy to change internally.

2. Inheritance



Share common structure and behavior through **abstract classes**. Specialized classes **extend** the base and add specific behavior.

→ Dramatically **reduces code duplication**.

3. Polymorphism



Write code to work with **interfaces** and **abstract classes** rather than concrete types.

→ Makes code **flexible** and **adaptable** to new future types.

4. Extensibility



Use **class extensions** to add behavior to standard classes. Never modify standard classes directly.

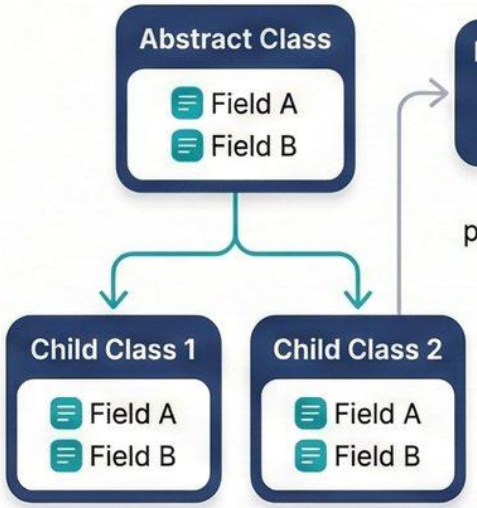
→ Extensions are **safe** and remain compatible with upgrades.

These principles work together to create code that is **maintainable**, **robust**, and **scalable**.

truestack.dev

Practical Process for Designing OOP Structures

1. Identify Required State

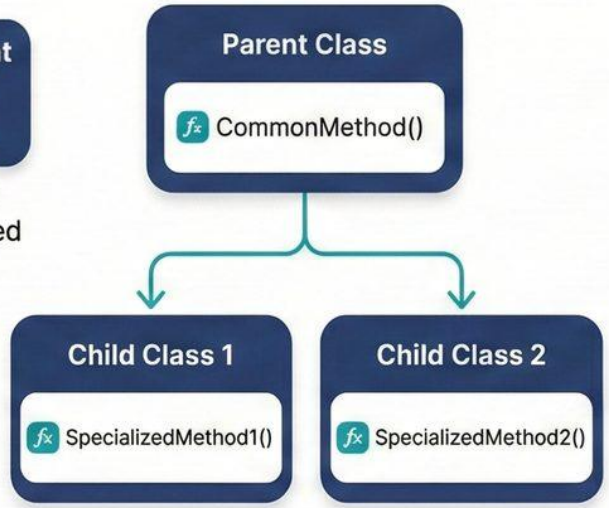




No shared parent needed

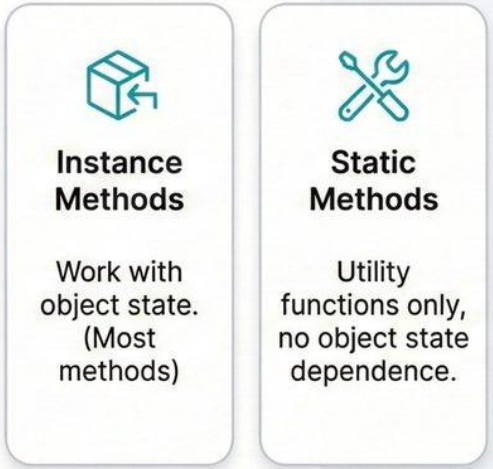
Share fields via abstract class; otherwise remain independent.

2. Identify Required Behavior

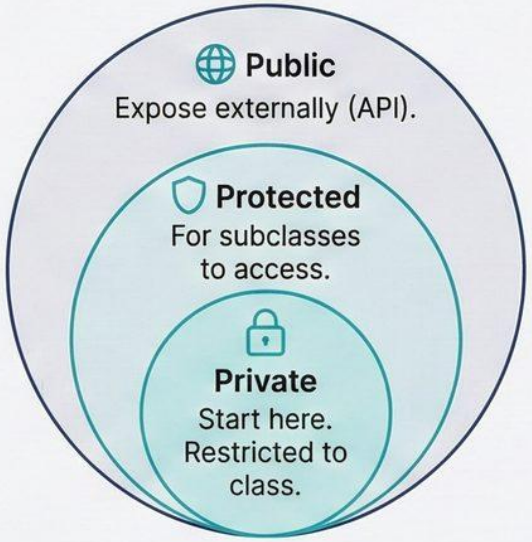


Common behavior in parent; specialized behavior in children.

3. Decide Instance vs. Static



4. Define Access Levels Carefully



TAKEAWAY: This disciplined approach creates clean, maintainable class hierarchies that are easy to understand and work with.

truestack.dev

Topic 3.2: Object-Oriented Programming Recap

Summary of Core Concepts, Design Principles, and Best Practices

1. Core Concepts & Methods

Access Modifiers (Encapsulation)

- Public: External API
- Protected: Subclass Behavior
- Private: Internal Logic

Method Types

- Instance: Operate on Object State
- Static: General Utilities (No State)

2. Data & Structure

Parameter Passing

- Value Types: Passed by Copy
- Reference Types: Passed by Reference

Abstract vs. Interfaces

- Abstract Classes: Shared Structure & Behavior (Inherit/Extend)
- Interfaces: Contracts for Unrelated Classes (Polymorphic Code)

3. Design Principles & Process

Key Principles

- Encapsulation
- Inheritance
- Polymorphism
- Extensibility

Best Practices & Design Process

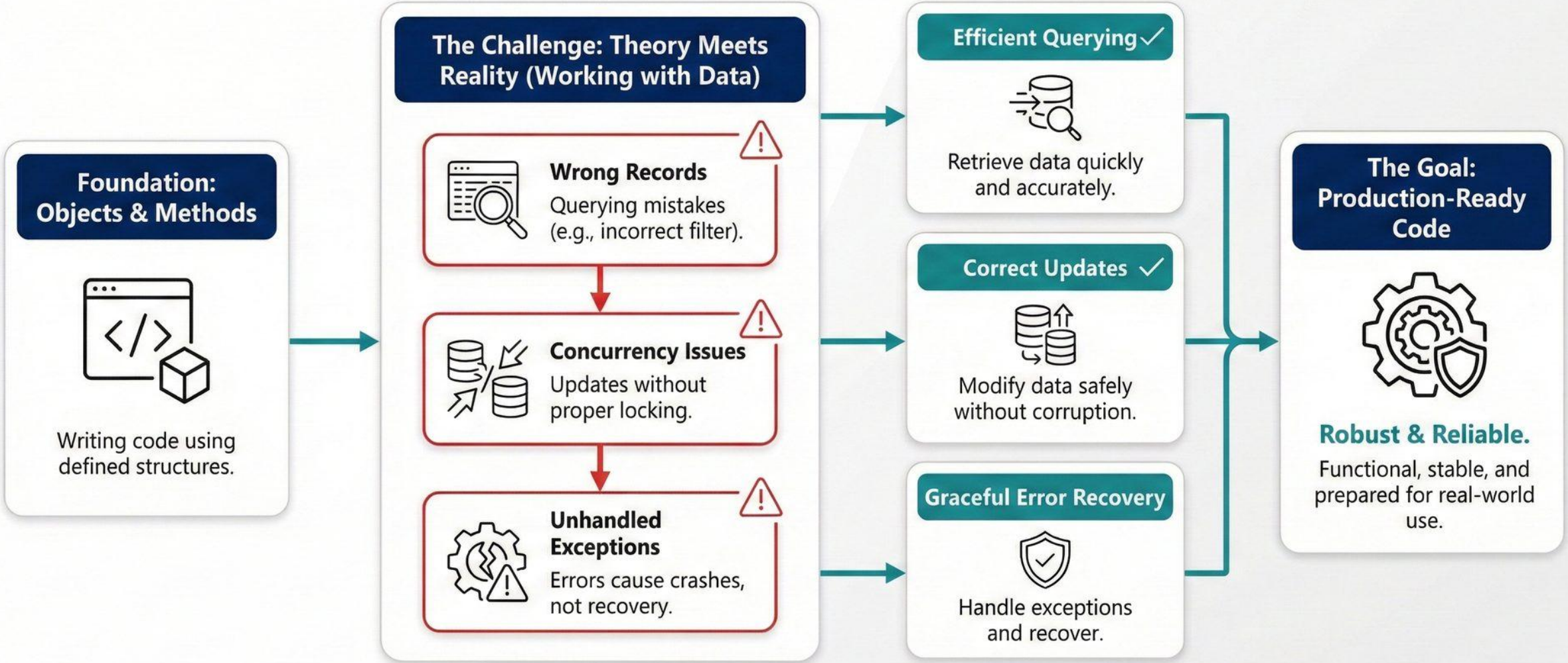
- ✓ Use Class Extensions (Not Direct Modification)
  - ➢ 1. Identify State
  - ➢ 2. Identify Behavior
  - ➢ 3. Choose Instance vs. Static
  - ➢ 4. Define Access Levels

Next Up: Topic 3.3

Data Manipulation and Exceptions: Querying, Updating, and Handling Errors Safely

truestack.dev

Topic 3.3: Data Manipulation and Exceptions



truestack.dev

The Fundamental X++ Select Statement

Basic Syntax & Filtering

Retrieves data

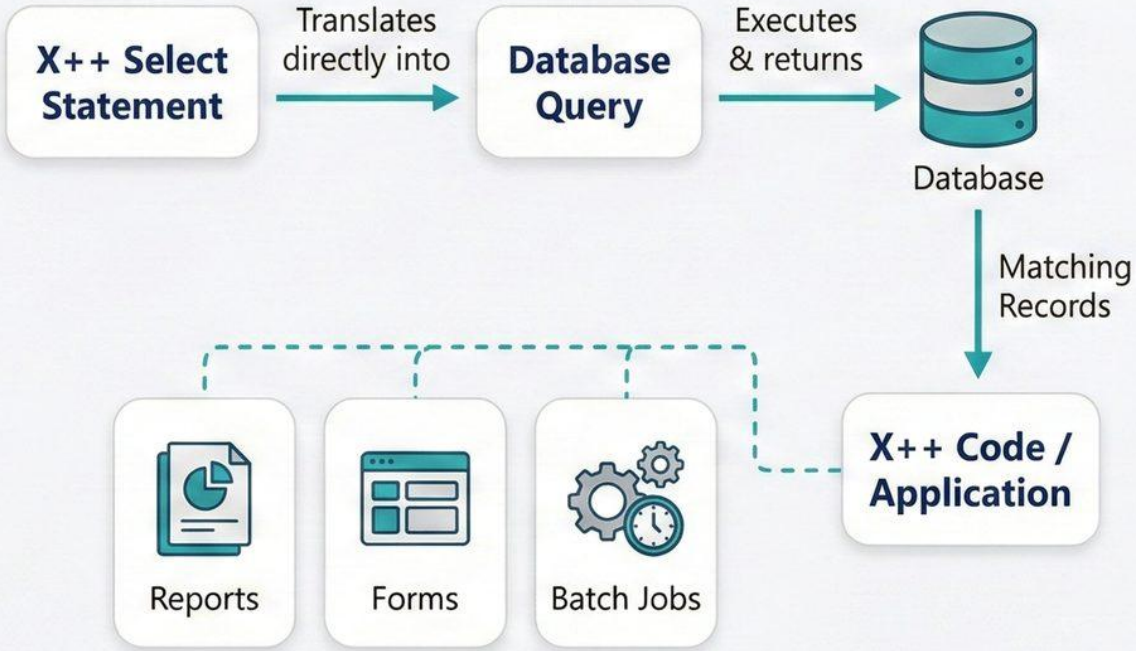
Filter

```
// Retrieve data in X++
select generateonly custTable
    where custTable.AccountNum == 'US-001' // Filter results
    order by custTable.Name asc; // Sort results
```

Sort

Selects all fields from a table like CustomerTable, filters for specific criteria, and sorts the output.

Execution Flow & Usage



Used in every application feature.

truestack.dev

Ordering Query Results in X++

Two approaches: Explicit Field Ordering vs. Index Ordering

Field Ordering (Clear & Explicit)

```
// Example: Sorts by VehicleId in ascending order
select vehicle order by VehicleId;
```

Uses 'order by' clause

Specifies actual field name

- ✓ Explicit and clear intent for anyone reading the code.
- ✓ You know exactly what field you're sorting by.



Index Ordering (Efficient & Predefined)

```
// Example: Uses the predefined VehicleIdIdx index
select vehicle index VehicleIdIdx;
```

Uses 'index' keyword

References predefined index name

- ✓ Efficient if the index already exists.
- ✓ Leverages index performance when sorting perfectly matches.



⚠ Compile Error Warning

INCORRECT: Using an index name in 'order by'

```
// This will cause a compile error
select vehicle order by VehicleIdIdx; // ERROR!
```

✗ Cannot use index name here. 'order by' only accepts field names.

CORRECT

- ✓ Use the 'index' keyword separately for index-based sorting.

Choose field ordering for clarity, and index ordering for performance when the index matches the sorting requirement.

trustack.dev

Data Filtering and Joins in Dynamics 365 F&O

Where Clauses: Filtering Records

Table: CustTable

AccountNum	Name	Region
1234	Customer A	West
1234	Customer B	West

where AccountNum == 1234

Result: Specific Customer

Account	Name	Region
1234	Customer A	West

Joins: Combining Data from Multiple Tables

Inner Join: Customers with Sales Orders



Shows only customers with at least one order

Outer Join: All Customers, Includes Orders



Shows all customers; null orders if none exist

Exists Join: Customers with Sales Orders



Shows customers that have at least one order

NotExists Join: Absence of Relationships



Customers with NO sales orders

→ **Why NotExists is so important?**
Answers questions like:

- → Show me customers with no sales orders
- → Show me inventory items that have never been purchased
- → Show me vendors with no open orders

NotExists is your tool for finding the **absence of relationships, crucial for analysis and reporting.**

truestack.dev

Real Use Case for NotExists Join

Scenario: Find Customers with No Sales Orders

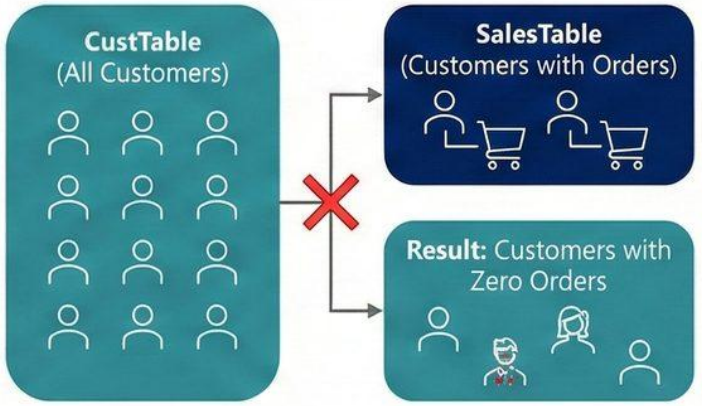


Identify potential untapped opportunities by finding customers without any associated sales orders.

X++ Query Implementation

```
select all
from CustTable,
notexists join SalesTable
where SalesTable.CustAccount
   == CustTable.AccountNum;
```

How It Works & Applications



- Perfect for targeting marketing campaigns.
- Investigate inactive customer reasons.
- Answer various business questions related to missing relationships.

NotExists is incredibly useful for finding entities with no associated records, offering valuable business insights when mastered.

truestack.dev

Understanding ForUpdate in D365 F&O

The ForUpdate Pattern



Select for Update
Locks record for exclusive access

Modify Fields
Make changes to required data

Call update()
Save changes, release lock

Why ForUpdate is Essential: Pessimistic Concurrency

Without ForUpdate (Risk)



Concurrent Modification Problem: Two users change same record. One user's changes get lost completely.

INCORRECT

With ForUpdate (Safe)



Pessimistic Concurrency: Record locked upfront. Second user waits until lock is released, preventing conflicts.

CORRECT

ForUpdate ensures exclusive record access to prevent data loss during concurrent modifications.

truestack.dev

Data Update Approaches: Set-Based vs. Row-Based



Set-Based Updates

```
update_recordset(update_records, SQL...
```

- ✓ Changes thousands of records in a single SQL statement.
- ✓ Extremely fast and efficient.
- ✓ Perfect for bulk operations (e.g., 'mark all invoices over 30 days old as due').
- ✗ No validation logic runs, no events fire - just raw data update at the database level.



Row-Based Updates

```
update() => row update...
```

- ✓ Processes one record at a time.
- ✓ Safer because everything gets properly validated.
- ✓ Each record runs through validation, events fire, business logic executes.
- ✗ Slower than set-based updates.

⚠ Anti-Pattern Warning

```
doUpdate() => update ...
```

Never use this. Bypasses everything (validation, events, logic). Considered an anti-pattern.

Choose Based on Your Needs

Speed ← → **Safety**

Set-Based — Row-Based

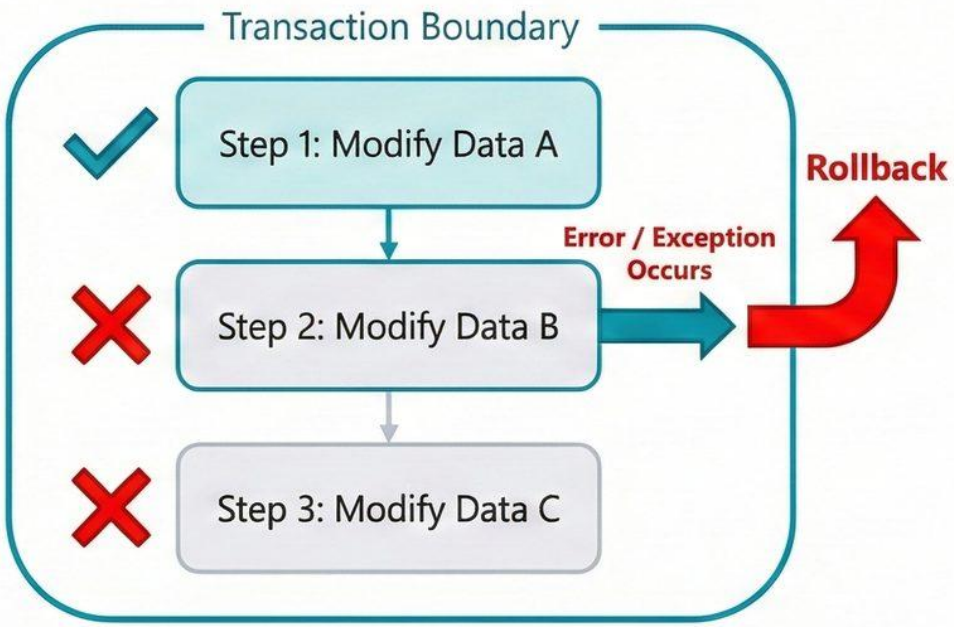
Use **set-based** for bulk updates where you don't need validation, and **row-based** for updates that absolutely must run through business logic and validation.

truestack.dev

Understanding Transactions in X++

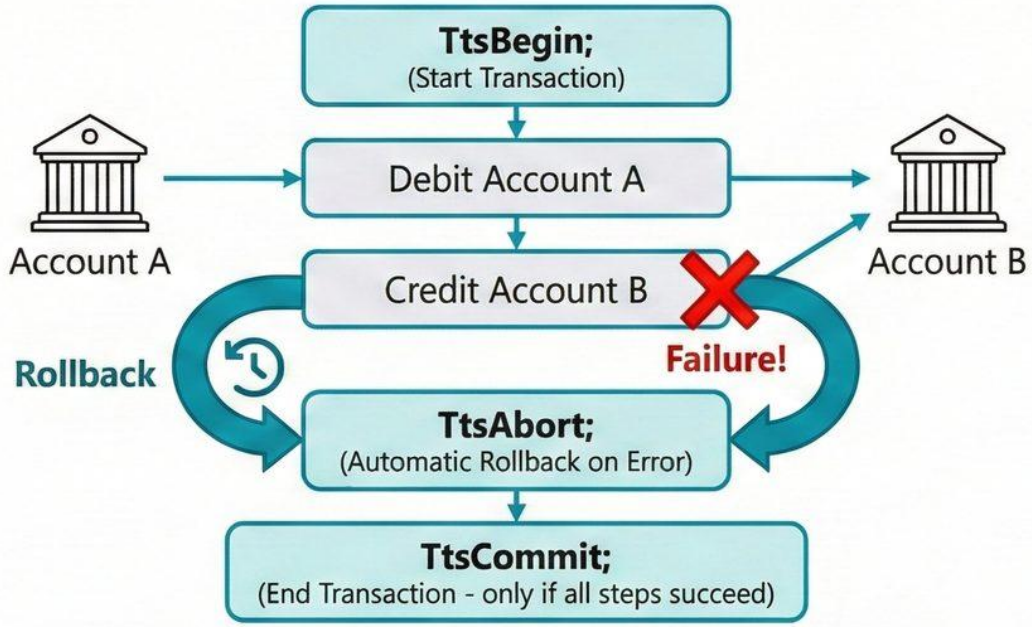
Ensuring Data Integrity with All-or-Nothing Semantics

The "All-or-Nothing" Concept



Outcome: Entire unit undoes. Database returns to original state. No partial changes.

Practical Example: Money Transfer

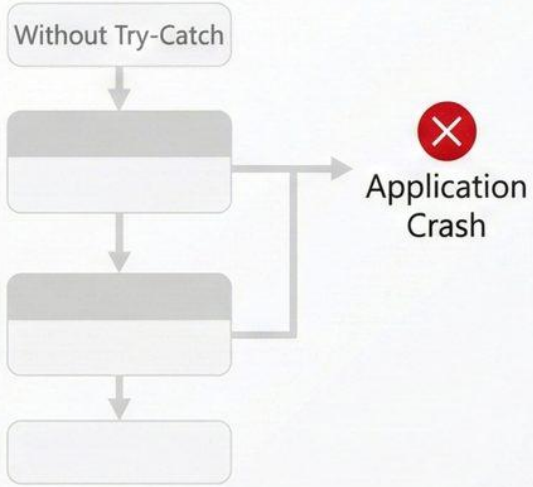


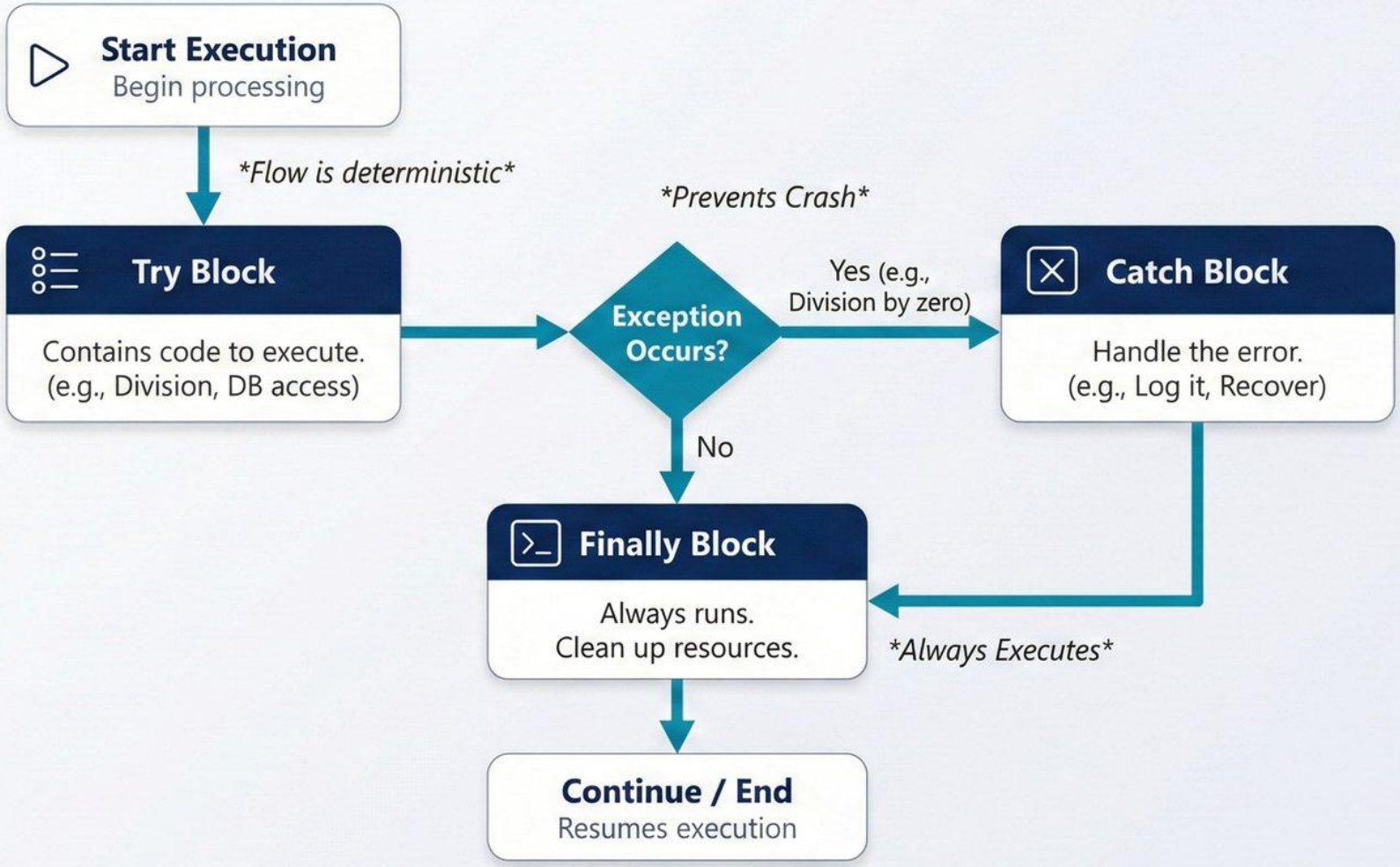
Outcome: If any step fails, no money is moved. Data integrity preserved.

Takeaway: Always wrap data modifications in **ttsBegin** and **ttsCommit** to ensure all changes are applied as a single, atomic unit or none at all.

truestack.dev

Graceful Error Handling: The Try-Catch-Finally Pattern



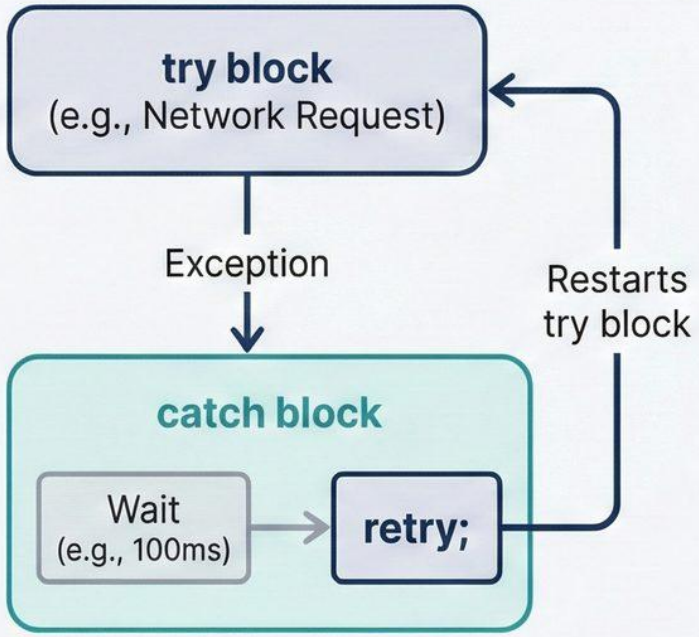


Takeaway: This structure ensures errors are handled gracefully, preventing crashes and ensuring resource cleanup, regardless of outcome.

truestack.dev

The retry Statement: Purpose and Best Practices

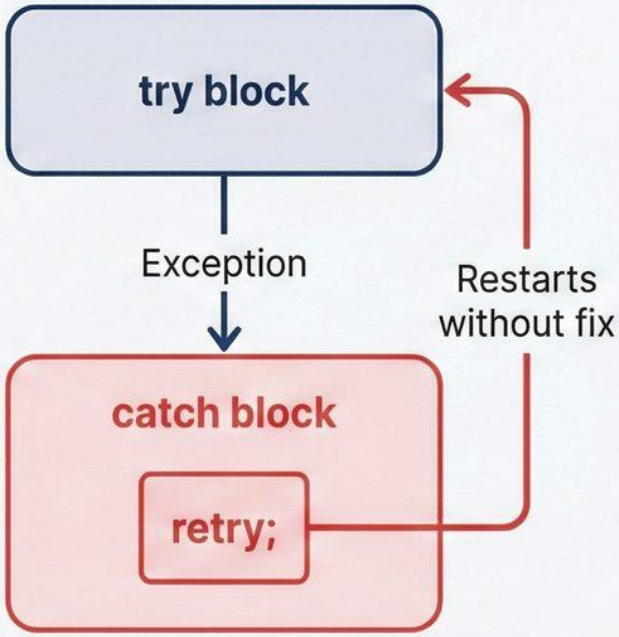
Purpose & Valid Use Case (Transient Errors)



Powerful for transient errors: network glitches, temporary locks, brief service unavailability.

Catch block waits and retries.

Critical Warning: Avoid Infinite Loops



NEVER retry endlessly without reason.

If catch block doesn't fix the problem, it creates an **infinite loop** and hangs the application.

Always ensure catch fixes something or include a retry counter.

Bottom Line: Use retry only when the catch block resolves the underlying issue or with a counter to prevent infinite loops.

truestack.dev

Exception Handling Strategies: Matching Action to Error Type

Different types of exceptions need different handling strategies for robust error handling.

Recoverable Error: Database Connection Failure

Scenario: Connection fails temporarily.

- Catch exception.
- Wait a moment (backoff).
- Retry operation.

Rationale: Connection might recover; retry logic is appropriate.

Permanent Error: Record Not Found

Scenario: Requested data does not exist.

- Catch exception.
- Log the issue thoroughly.
- Return null or default value.
- Do NOT retry.

Rationale: Retrying won't make the record appear; permanent failure.

Permanent Error: Validation Failure

Scenario: Input or business rule validation fails.

- Catch exception.
- Notify the user clearly.
- Do NOT retry.

Rationale: The same validation will fail again; requires user correction.

⚠ Unexpected Errors & Key Takeaway

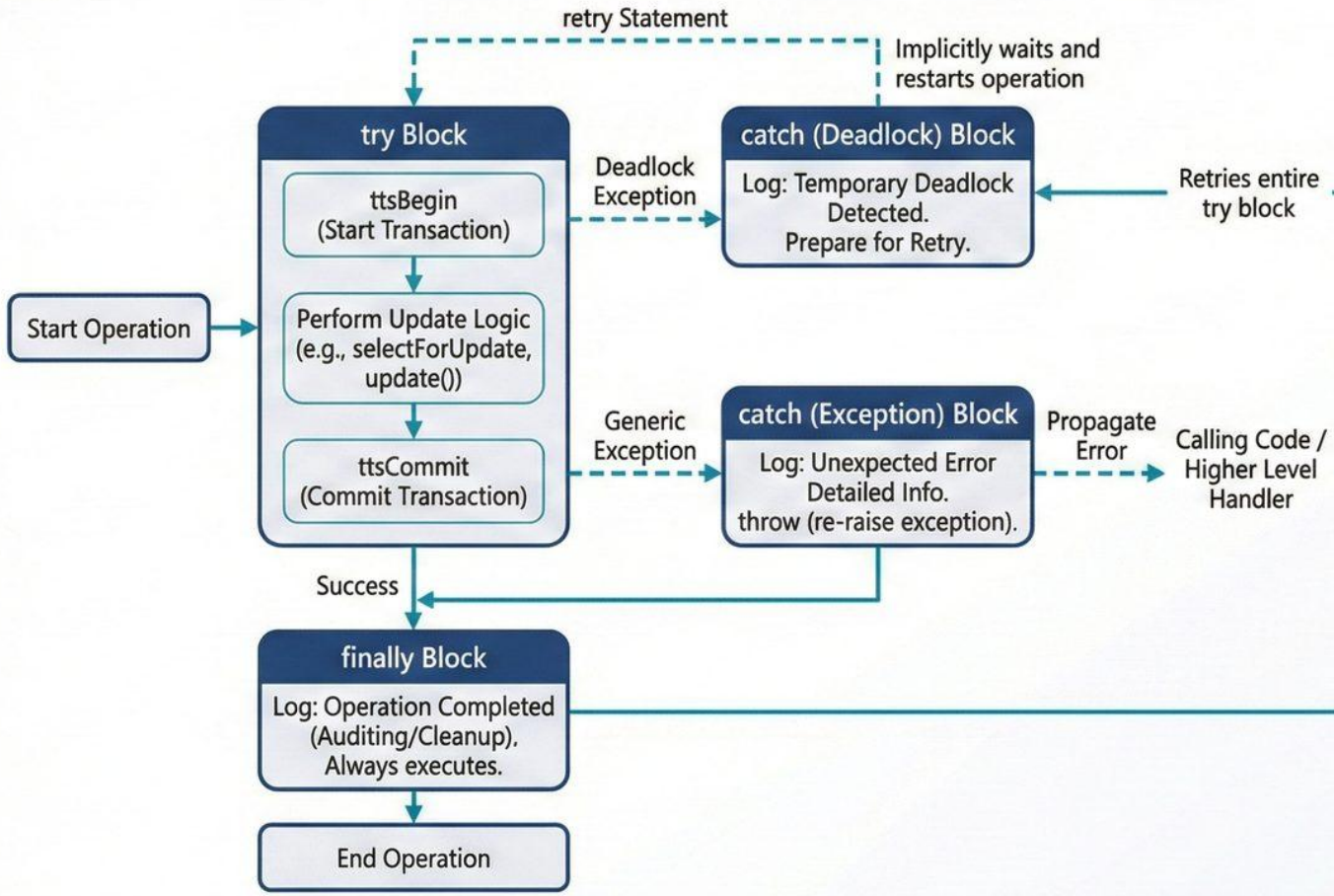
For unexpected errors not specifically anticipated: Catch them, log thoroughly for debugging, and throw the exception upward to the caller.

Bottom Takeaway: Understanding which errors are recoverable (retry) and which are permanent (log & propagate) is key to good exception handling.

truestack.dev

Production-Ready Exception Handling Pattern in X++

Intelligent deadlock recovery, robust error propagation, and consistent auditing.




```x++
try
{
    ttsBegin;
    // Perform update logic within transaction
    myTable.selectForUpdate(true);
    myTable.Field1 = "New Value";
    myTable.update();
    ttsCommit;
}
catch (Exception::Deadlock)
{
    // Log temporary deadlock and retry
    info("Deadlock detected. Retrying operation...");
    retry; // Implicitly waits and restarts try block
}
catch (Exception::Error)
{
    // Log unexpected error thoroughly and propagate
    error(strFmt("Unexpected error occurred: %1",
    throw Exception::Error; // Throw upward to calling code
}
finally
{
    // Always runs for auditing
    info("Operation flow completed.");
}
```

This pattern ensures robust handling of recoverable deadlocks, proper propagation of unexpected errors, and consistent auditing for enterprise-grade X++ applications.

truestack.dev

SysQueryRangeUtil for Dynamic Filtering

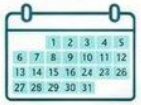
Simplify date and text filtering with built-in helper functions.

Dynamic Date Filters

CurrentDate()
Returns today's date.
Useful for daily transaction reports.

TODAY → 2023-10-27

MonthRange()
Returns the first and last day of the current month.
Ideal for 'current period' filtering.



→ 2023-10-01 .. 2023-10-31

DateRange()
Lets you specify a custom date range.
Flexible for specific reporting intervals.

START → END → 2023-01-01 .. 2023-06-30

Wildcard Filtering

Wildcard (*)
Uses asterisks, similar to SQL percent signs (%), for pattern matching.
Finds records with the specified text anywhere in the field.

Wholesale

1. ABC Wholesale Co.
2. Global Wholesale Ltd.
3. Wholesale Distributors Inc.

Benefits

- Saves significant development time.
- Ensures consistent filtering logic across the system.
- Makes filtering straightforward to learn and apply.

SysQueryRangeUtil functions streamline filtering in reports and forms, enhancing efficiency and consistency.

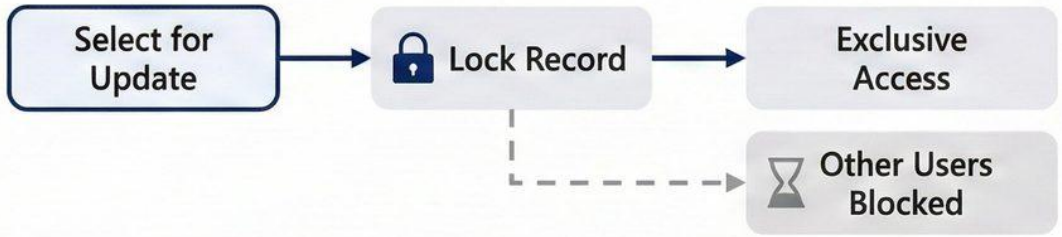
truestack.dev

Understanding Concurrency Models in D365 F&O

Comparing Approaches to Handling Data Conflicts and Throughput

Pessimistic Concurrency

Approach: Upfront Locking



- Locks records immediately upon selection for update.
- Ensures exclusive access; others wait until released.
- Simple and safe conflict prevention.

Outcome & Use Case

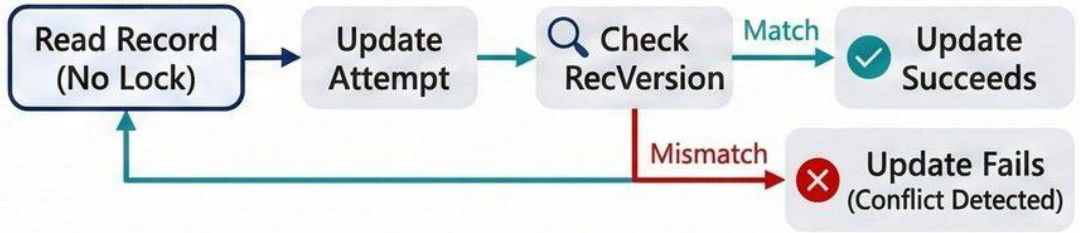
Lower throughput in high-concurrency scenarios.

Less complexity; conflict is prevented upfront.

Best for: Moderate-concurrency, simplicity valued.

Optimistic Concurrency

Approach: Check at Update



- Does not lock at read time; checks RecVersion on update.
- Update succeeds if RecVersion matches original.
- Fails if record changed; allows retry logic.

Outcome & Use Case

Higher throughput; no upfront locks blocking users.

More complexity; requires conflict handling logic.

Best for: High-throughput, multi-user scalability.

D365 Default for Scalability

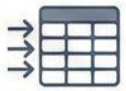
truestack.dev

Four Critical Data Manipulation Mistakes in Dynamics 365 F&O

Avoid these patterns to prevent production disasters and data corruption.

MISTAKE 1: MISSING WHERE CLAUSE

Update statement affects EVERY row instead of target rows. Leads to disaster.



```
// INCORRECT
update_recordset sampleTable
  setting Field1 = 'NewValue';
```

CORRECT PATTERN

```
// CORRECT
ttsBegin;
update_recordset sampleTable
  setting Field1 = 'NewValue'
  where sampleTable.id == 'TargetId';
ttsCommit;
```

Always include a where clause unless updating all records.

MISTAKE 2: USING DOUPDATE DIRECTLY

Bypasses all business logic, validations, and events. Causes corruption.



```
// INCORRECT
sampleTable.Field1 = 'NewValue';
sampleTable.doUpdate();
```

CORRECT PATTERN

```
// CORRECT
ttsBegin;
sampleTable.selectForUpdate(true);
sampleTable.Field1 = 'NewValue';
sampleTable.update();
ttsCommit;
```

Never use doUpdate; allow business logic to execute.

MISTAKE 3: NO TRANSACTION BOUNDARIES

Data changes are not atomic. Failures leave data in inconsistent states.



```
// INCORRECT
sampleTable1.update();
sampleTable2.update(); // If this
fails, sampleTable1 remains updated.
```

CORRECT PATTERN

```
// CORRECT
ttsBegin;
try {
    sampleTable1.update();
    sampleTable2.update();
    ttsCommit;
}
catch {
    ttsAbort;
    // Handle error
}
```

Always wrap related modifications in ttsBegin and ttsCommit.

MISTAKE 4: CATCHING EXCEPTIONS SILENTLY

Empty catch blocks hide errors completely. Application runs, but data gets corrupted unknown.



```
// INCORRECT
try {
    // Data modification logic
}
catch {
    // Empty block, no action
}
```

CORRECT PATTERN

```
// CORRECT
try {
    // Data modification logic
}
catch (Exception::Error) {
    info('An error occurred during update.');
    throw Exception::Error; // Rethrow or
   handle specifically
}
```

Always handle exceptions specifically or rethrow them; never silence errors.

Avoid these mistakes by following proper patterns to prevent expensive production disasters.

truestack.dev

Recap Topic 3.3: Data Manipulation & Exceptions

Querying & Locking

- `select` statements for querying data.
- `where` clauses to filter records.
- `order by` to sort records.
- `joins` to combine multiple tables.
- `NotExists` for finding entities without related records.
- `ForUpdate` locks records for exclusive access to modify.

Updates & Transactions

- Set-based updates: `update_recordset` for many records, fast, no validation.
- Row-based updates: `update()` method, one at a time, full business logic and validation.
- Transactions ensure all-or-nothing atomicity.

Exceptions & Concurrency

- `try-catch-finally` for graceful exception handling.
- `retry` restarts try block for recoverable errors.
- Pessimistic concurrency: locks records upfront.
- Optimistic concurrency: detects conflicts at update time.

⚠ CRITICAL ANTI-PATTERNS (Avoid!)

✕ Missing `where` clauses. ✕ Using `doUpdate` directly. ✕ Forgetting transaction boundaries. ✕ Catching exceptions silently.

Coming Up: Topic 4.1

Class Extensions and Chain of Command: How to extend standard D365 code safely without overlaying. →

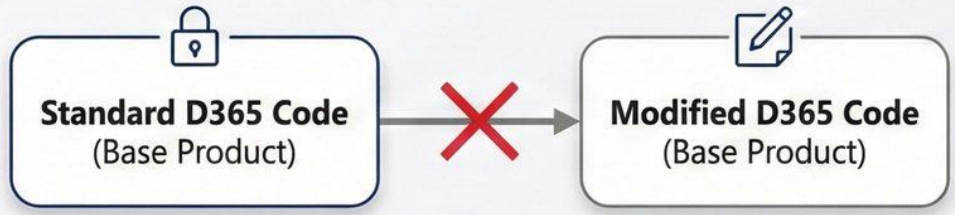
truestack.dev

Topic 4.1: Class Extensions and Chain of Command

Extending standard D365 code without breaking it.

The Challenge: Standard Code Modification

Never directly modify standard D365 classes, tables, or forms. This breaks upgrades and maintainability.



The Solution: Composition-over-Inheritance

Create extension classes to augment functionality. The platform stitches your code into the base at compilation.



Benefits

- Safe, maintainable code.
- Survives Microsoft upgrades without breaking.
- Builds on the foundation of composition-over-inheritance pattern.

Mastering extensions and Chain of Command is critical for building robust and upgrade-safe D365 solutions.

truestack.dev

Class Extensions: Adding Functionality Safely



1. Create Extension Class

New class with _Extension suffix, marked final & public.



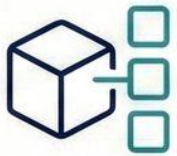
2. Declare Extension

Use [ExtensionOf] attribute for target base class.



3. Compilation Process

Compiler "stitches" extension into base class during compilation.



4. Resulting Base Class

Base class now has new methods available.



How it Works

Callers use the base class normally and automatically get access to your new functionality.



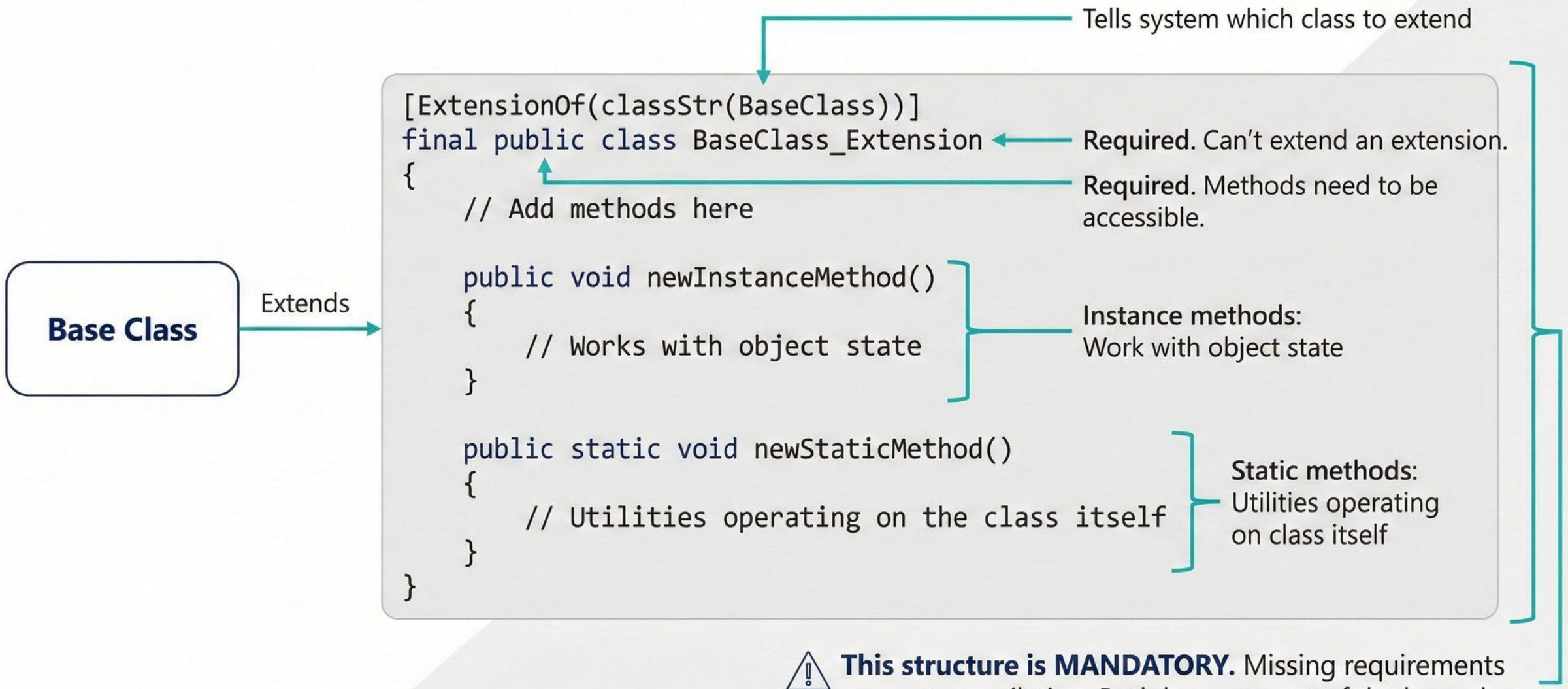
The Power of Extensions

Incredibly safe: Never touch standard code. Updates cause no conflict; extension works alongside updated base class.

The real power of extensions is **safe, conflict-free customization.**

truestack.dev

Complete Class Extension Structure

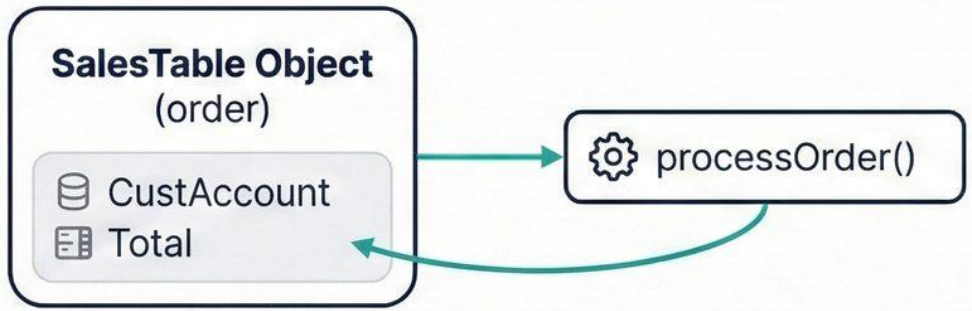


truestack.dev

Adding Methods: Instance vs. Static

Instance Methods

Gets access to the object's fields and properties.



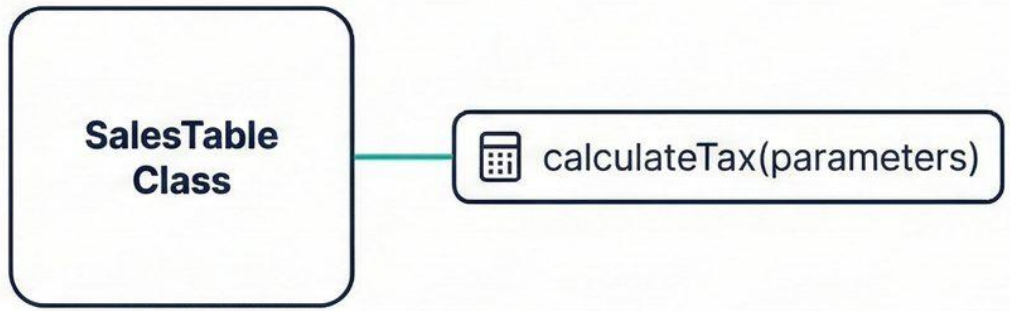

```
// Instance Method Example
SalesTable order;

// Call the instance method
order.processOrder();
// Accesses order.CustAccount, order.Total
```

Method called on object instance

Static Methods

Doesn't need an object at all. It's just a utility on the class.




```
// Static Method Example

// Call the static method directly on the class
SalesTable::calculateTax(parameters);

// Utility function, no object state needed
```

Method called on class directly

Rule of Thumb

✓ Choose instance when you need the object's data.

✓ Choose static when you just need a utility function.

truestack.dev

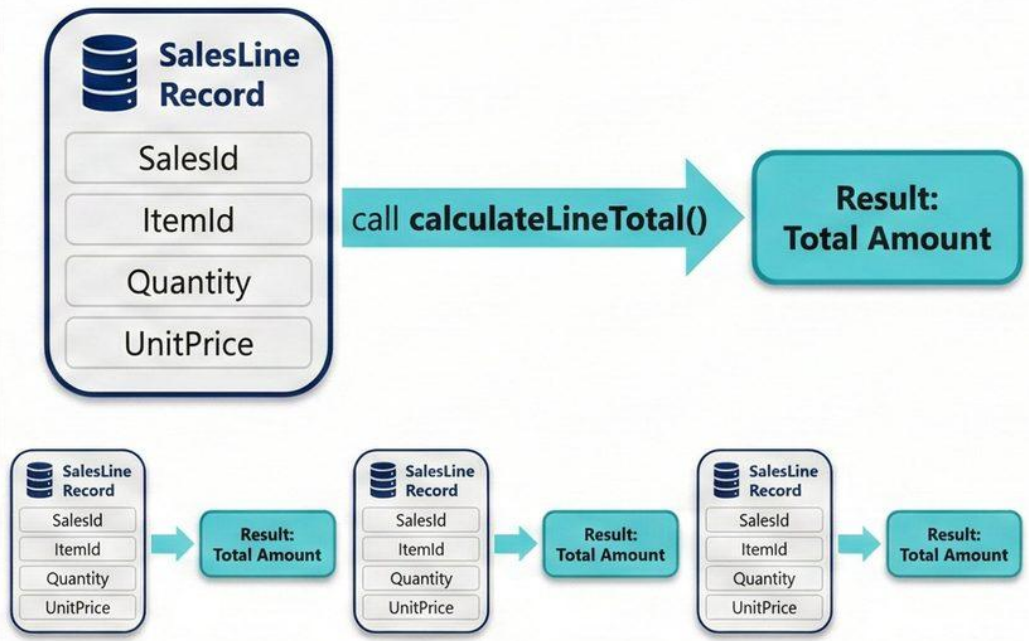
Practical Example: Extending SalesLine with calculateLineTotal

Code Extension: Adding the Method

```
[ExtensionOf(tableStr(SalesLine))]
final class SalesLine_Extension
{
    /// <summary>
    /// Calculates the total for the sales line.
    /// </summary>
    public AmountCur calculateLineTotal()
    {
        // 'this' refers to the SalesLine table record
        return this.Quantity * this.UnitPrice;
    }
}
```

Accessing fields directly using 'this' keyword.

Conceptual Usage: Reusable Logic



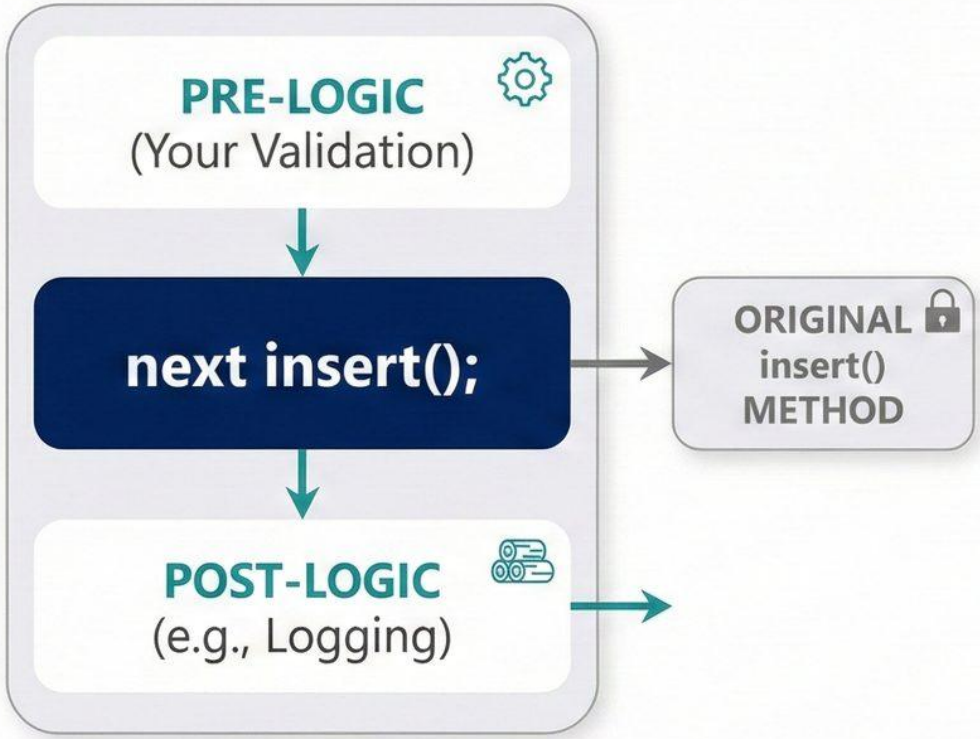
Method is available on EVERY SalesLine record. Clean and reusable.

Extending tables with new methods promotes clean, reusable logic and avoids duplication across the codebase.

truestack.dev

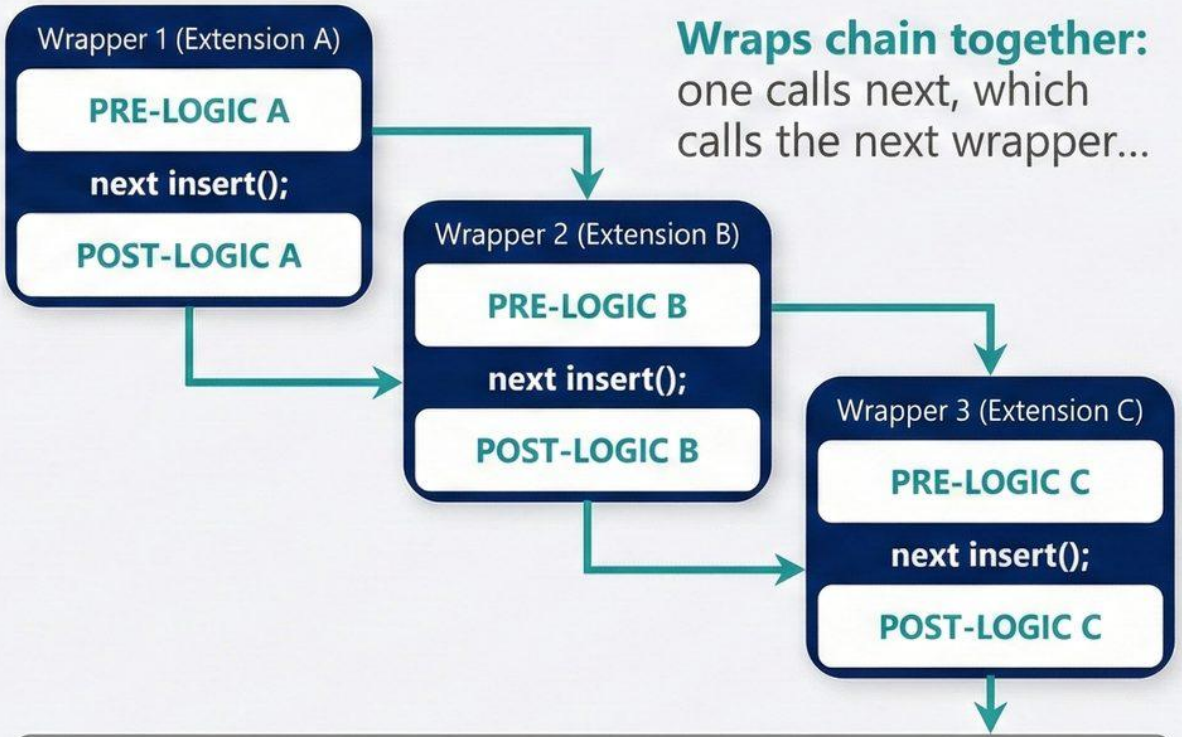
Chain of Command (CoC): Method Wrapping & Composition

Method Wrapping Concept



Wraps existing method, adds behavior before/after 'next' call.

Chaining Multiple Wrappers

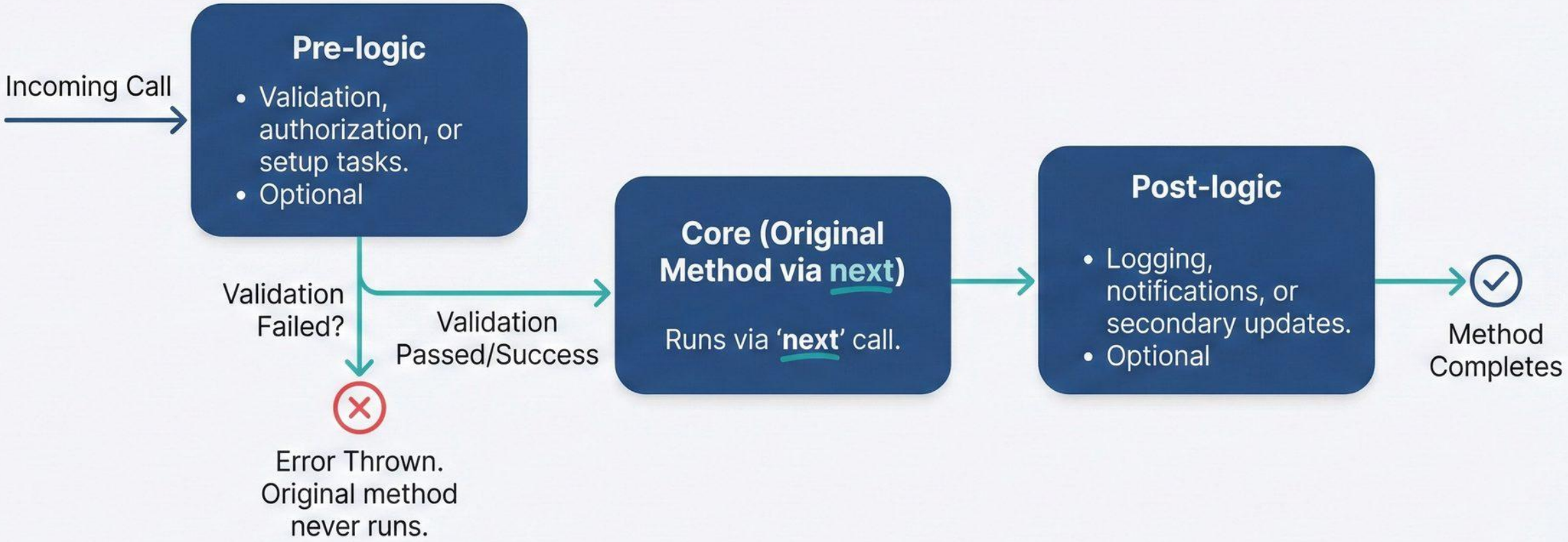


Wraps chain together: one calls next, which calls the next wrapper...

Safe & Elegant Composition

truestack.dev

Practical Chain of Command (CoC) Pattern



Flexibility: Both pre and post are optional. Powerful for **cross-cutting concerns**.

truestack.dev

Chain of Command (CoC): Signature Integrity Rule

Base Method (Original)

```
// Base method signature
public void update()
{
    // original logic
}
```

Signature must be matched exactly

CoC Wrapper (Extension)

CORRECT: Exact Match

```
[ExtensionOf(tableStr(MyTable))]
final class MyTable_Extension
{
    // Correct wrapper signature
    public void update()
    {
        next update();
        // added logic
    }
}
```

Same parameters, same return type. Intercepts call.

INCORRECT: Signature Mismatch

```
[ExtensionOf(tableStr(MyTable))]
final class MyTable_Extension
{
    // Incorrect wrapper signature
    public void update(int _param) // Added parameter
    {
        next update(); // Error
        // added logic
    }
}
```

Not a wrapper

Different method, not a wrapper. Compiler rejects it.

Maintaining signature integrity isn't negotiable here; the wrapper must match the base method exactly.

truestack.dev

Wrappin g Static Methods with Chain of Command

The CoC Pattern for Static Methods

Static methods can be wrapped using Chain of Command, following the identical pattern as instance methods: pre-logic, next invoke, post-logic, and return. Crucially, the wrapper method itself must also be declared as static to ensure correct method chaining.



Applies to static utilities like calculateTax, ensuring the chain works correctly.

```
[ExtensionOf(classStr(TaxUtility))]
final class TaxUtility_Extension
{
    // Wrapper for the static calculateTax method
    public static real calculateTax(real _amount)
    {
        // Pre-logic: Log start of calculation
        info(strFmt("Starting tax calculation for amount: %1", _amount));

        // Invoke base static method logic
        real result = next calculateTax(_amount);

        // Post-logic: Log end of calculation
        info(strFmt("Finished tax calculation. Result: %1", result));

        return result;
    }
}
```

Wrapper must be static

Pre-logic (e.g., Logging)

Invokes base static method via 'next'

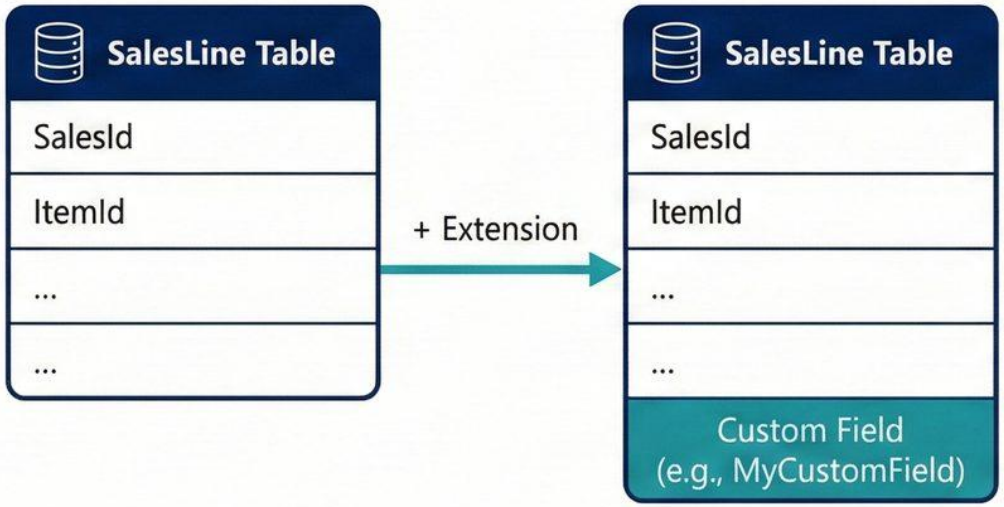
Post-logic

truestack.dev

Microsoft Dynamics 365 Finance & Operations tutorial

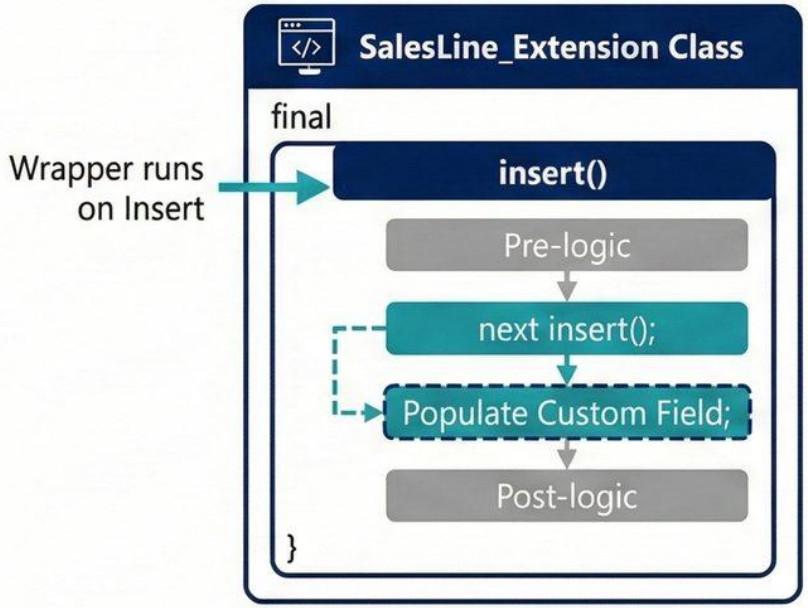
Extending Table Functionality: Fields & Logic

1. Table Extension (Adds Fields)



Creates new field on the table itself.

2. Class Extension with Chain of Command (Adds Logic)



Wraps methods to execute custom logic.

Clean Separation of Concerns: Field definitions separate from behavior.

Both extensions are required: one for fields, one for logic.

truestack.dev

D365 F&O: Correct Extension Pattern & Rules

Mandatory structure for upgradability and supportability.

CORRECT PATTERN (Mandatory) ✅

- ✓ Create an extension class
- ✓ Mark it **final**
- ✓ Mark it **public**
- ✓ Use the **ExtensionOf** attribute
- ✓ Follow the pattern exactly

Identifies the base class being extended.

```
[ExtensionOf(classStr(BaseClass))]
final public class BaseClass_Extension
{
    // Correct extension implementation
}
```

Required modifiers for extension classes.

INCORRECT PATTERN (Anti-Pattern) ❌

- ❌ Never inherit from the base class using **extends**
- ❌ Most standard classes are sealed and won't compile
- ❌ Never skip the **final** keyword
- ❌ Never skip the **ExtensionOf** attribute

❌ *Avoid inheritance; use extensions instead.*

```
// INCORRECT: Do NOT inherit!
public class MyClass extends BaseClass
{
    // This will not work for most standard classes
}
```

Missing required elements.

Following this pattern isn't optional—it's mandatory for D365 upgradability and supportability.

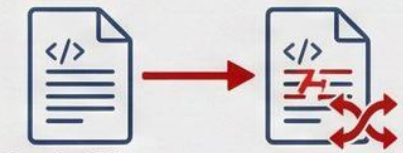
truestack.dev

Four Critical Extension Mistakes

Mistake One: Modifying Standard Code Directly

INCORRECT

Changing a Microsoft class. When Microsoft releases an update, you get a conflict and can't upgrade.



CORRECT

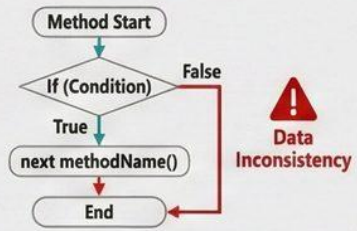
Use extensions instead.



Mistake Two: Calling next Conditionally

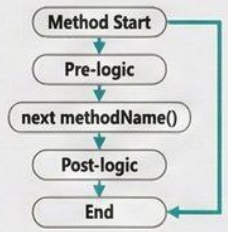
INCORRECT

Wrapping "next" inside an if-statement. If condition is false, next doesn't run, leading to data inconsistency.



CORRECT

Never wrap 'next' in conditionals – either always call next unconditionally or don't call it at all.



Mistake Three: Changing the Chain of Command Signature

INCORRECT

Trying to add a parameter or change return type. Results in signature mismatch and compilation error.

Base Method

```
public void myMethod(int param1) {
   ...
}
```

Extension Method (Incorrect)

```
public void myMethod(int param1, str newParam) {
   ...
}
```

Signature Mismatch Error

CORRECT

Signatures must be identical.

Base Method

```
public void myMethod(int param1) {
   ...
}
```

Extension Method (Correct)

```
public void myMethod(int param1) {
   ...
}
```

Mistake Four: Creating a Non-Final Extension

INCORRECT

Skipping the "final" keyword. Violates extensibility rules and causes issues down the line.

```
[ExtensionOf(classStr(MyClass))]
class MyClass_Extension {
   ...
}
```

Violates Rules

CORRECT

Always mark your extensions 'final'.

```
[ExtensionOf(classStr(MyClass))]
final class MyClass_Extension {
   ...
}
```

Compliant & Safe

Avoid these anti-patterns to ensure maintainable, upgrade-safe Dynamics 365 extensions.

truestack.dev

Workflow: Intercepting Sales Order Inserts with Logic

1. 

Sales Order Insert Intercept

A sales order insert operation is initiated.

2. 

Execute Pre-Logic

If amount > 10,000, mark as requiring approval.

3. 

Run Original Logic (next insert)

The standard insert method is called to perform the core action.

4. 

Send for Approval

After the insert completes, the approval workflow is triggered.

✓ **Outcome:** Every sales order insert is intercepted and wrapped with the approval workflow, with no modification of standard code.

truestack.dev

Form Extensions

Concept & Example

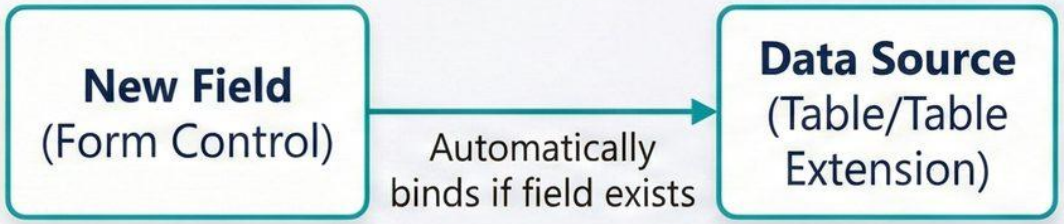
Customize standard forms without modifying the original.

Example: Create an extension for SalesOrderForm.

Process & Patterns

- Use form designer to add controls and data sources.
- Follow standard form patterns: Action Pane, Navigation List, Grid, Details Header, Details Tab.

Automatic Data Binding



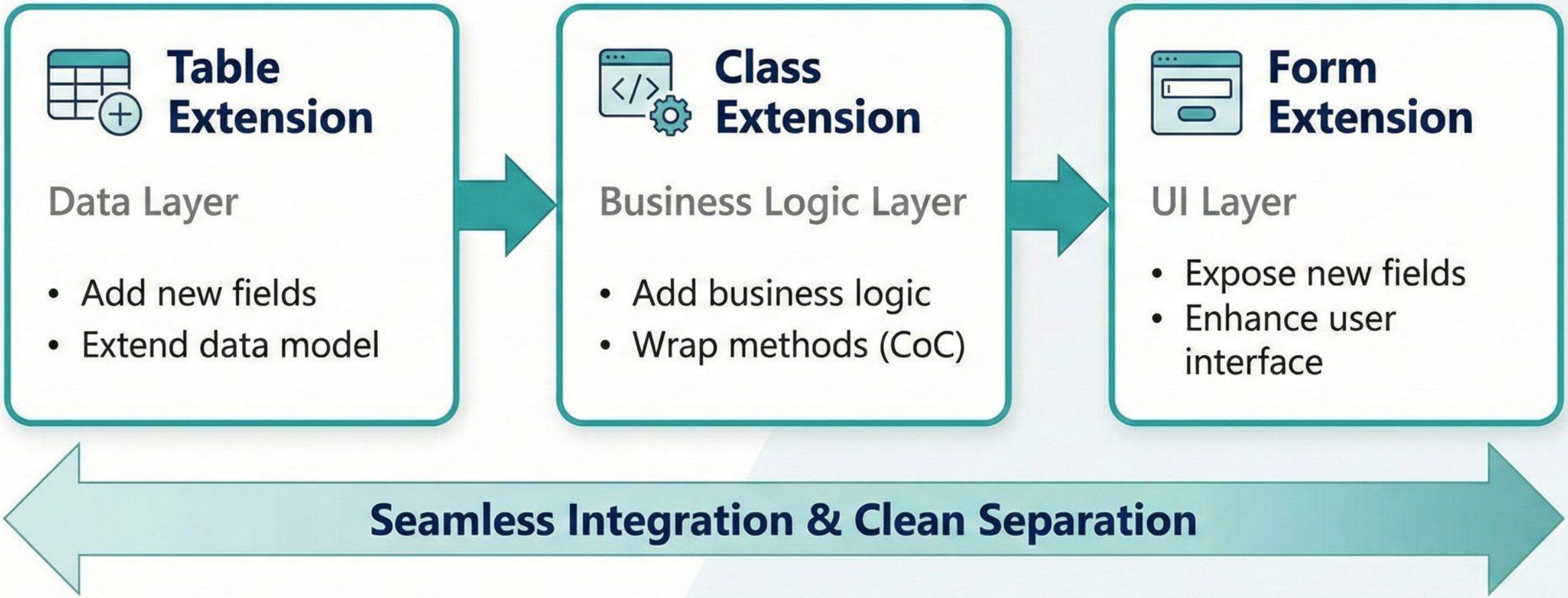
Key Benefits

- Visual and designer-friendly.
- Most customization doesn't require code.

Form extensions enable visual, designer-friendly customizations of standard forms without code modifications.

truestack.dev

Professional Customization: The Layered D365 Pattern



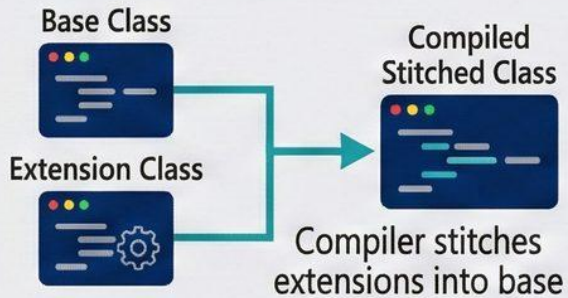
This **layered approach** is the standard D365 customization pattern, ensuring maintainable and integrated solutions.

truestack.dev

Recap: Topic 4.1 Extension Classes & Safe Practices

Extension Class Basics

- marked `final` and `public`
- with `[ExtensionOf]` attribute
- Text and `_Extension` suffix

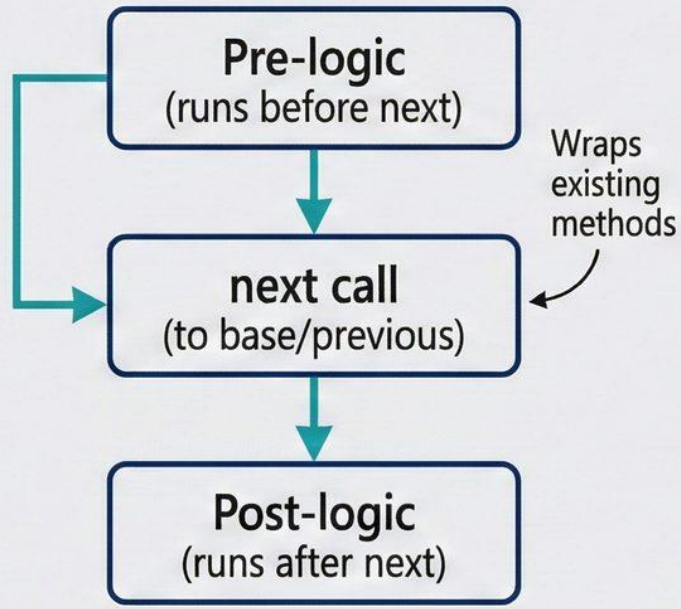


Methods

- **Instance**: work on object state
- **Static**: are utilities

Chain of Command (CoC)

Wrapper Method
(Signature must match exactly)



Extension Types & Practices

Extend Together

- **Table extensions**: add fields
- **Class extensions**: add logic
- **Form extensions**: add UI

Safe vs. Anti-Patterns

Safe Practice (✓)	Anti-Patterns (✗)
✓ Always extend, never modify	✗ Modifying standard code ✗ Conditional next calls ✗ Signature changes ✗ Non-final extensions

All of these cause problems.

Next Up: Topic 4.2 Events and Attributes

→ Discover advanced customization patterns: how to trigger code on events and mark up code with metadata.

truostack.dev

Topic 4.2: Events and Attributes

Extending Code and Triggering It at the Right Moment

Events: "Something important just happened"

Subscribing to events ensures your code runs at exactly the right time.



Code reacts intelligently to user actions & system events.

Attributes: Controlling What Can Be Extended

Mark up code to define how other developers can interact with it.

- ✓ Can be wrapped with Chain of Command?
- ✗ Is method deprecated and shouldn't be used?

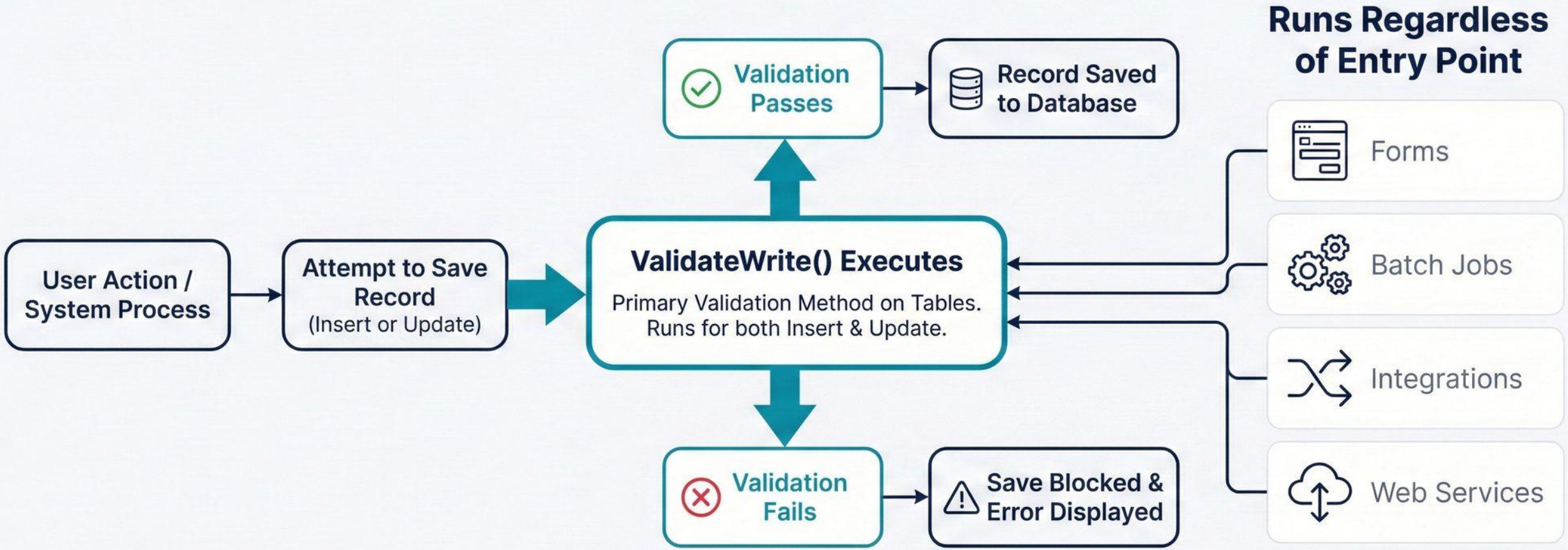
Attributes answer these questions, managing the extension lifecycle.

Master these patterns to build code that reacts intelligently to user actions and system events.

truestack.dev

Understanding ValidateWrite: Primary Table Validation

Enforcing data integrity at the **moment of persistence** during save operations.



ValidateWrite is the single point of truth for enforcing business rules and data integrity before persistence, ensuring every save path is validated.

truestack.dev

Table Methods for Validation & Persistence in Dynamics 365 F&O

ValidateWrite Method

Runs at save time for all entry points. Include record-level business rules here that must block persistence if failed.

Focus: Record-level Business Rules

ValidateField Method

Runs during form entry for individual fields. Triggered when a user tabs out of a specific field to check its value.

Focus: Field-level UI Feedback

Insert and Update Methods

Persist the data to the database. Do not place validation logic here; their purpose is data persistence only.

Focus: Data Persistence



The clear pattern is: validateWrite for record-level rules, validateField for UI feedback, and insert/update for data persistence.

truestack.dev

Creating Table Validation in Dynamics 365 F&O

1

Create Handler Class

Define a class with the `SubscribesTo` attribute for event handling.

2

Implement validateWrite Handler

Add a static method with the correct signature for the table's `validateWrite` event.

3

Add Validation Logic & CheckFailed

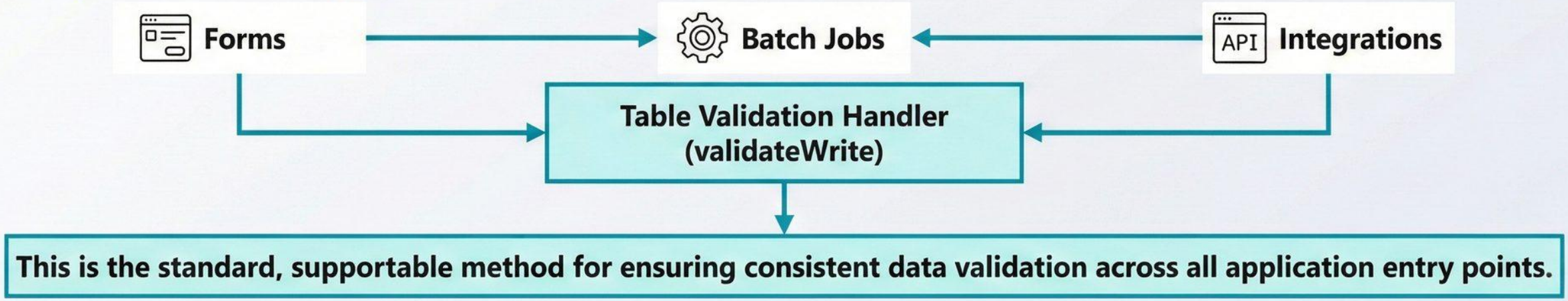
Implement business rules. Call `checkFailed` with an error message if the record is invalid.

4

Centralized Execution

This handler is triggered by all table updates (forms, batch jobs, integrations).

Scope & Standard Approach



truestack.dev

Validation Scopes in Dynamics 365 Finance & Operations

Understanding Table-Level and Form-Level Validation



Table-level validation

- Applies everywhere: forms, integrations, batch jobs.
- All entry points execute it.
- Perfect for data integrity rules that must always be enforced.

Ensures consistency across all data access paths.

☑ **Example:** A sales order must have a **customer**. (Enforced everywhere)



Form-level validation

- Applies only to that specific form.
- Only users on that form see it.
- Perfect for workflow-specific rules or rules that depend on form context.

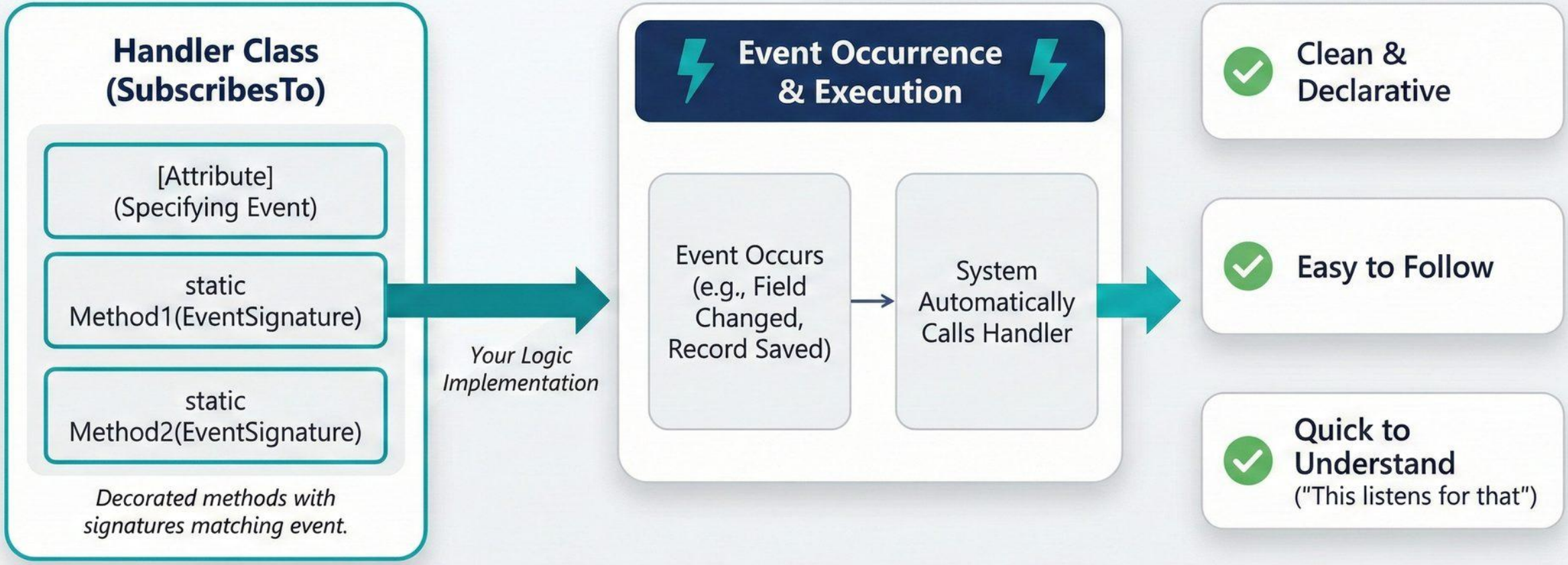
Tailored logic for specific user interfaces.

☑ **Example:** A sales order on the executive dashboard requires credit approval. (Only for that dashboard)

Choose the right scope based on where the rule applies.

truestack.dev

The Event Handler Pattern



This pattern simplifies development by clearly separating event logic from business code.

truestack.dev

Two Ways to Block a Save

Understanding validation techniques in Microsoft Dynamics 365 Finance & Operations.

1. CheckFailed (User-Facing)

- Displays an error message to the user.
- Explains why the save failed, so users understand and can correct it.
- Recommended for most validations to guide the user.

```
boolean validateWrite() {
    ...
    if (!condition) {
        checkFailed("@Label:ErrorMessage");
    }
    ...
}
```

Helps users understand

2. Return false (Internal Checks)

- Blocks the save without a visible message.
- Used for internal checks that should fail silently.
- Does not provide feedback to the user.

```
boolean validateWrite() {
    ...
    if (!internalCondition) {
        return false;
    }
    ...
}
```

Blocks silently

Takeaway: Most validation should use **checkFailed** to help users understand what's happening.

truestack.dev

Chain of Command for Business Process Validation

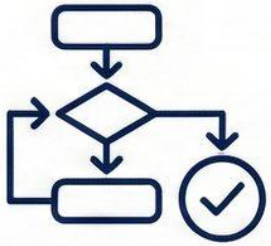
Using CoC for Process-Level Validation

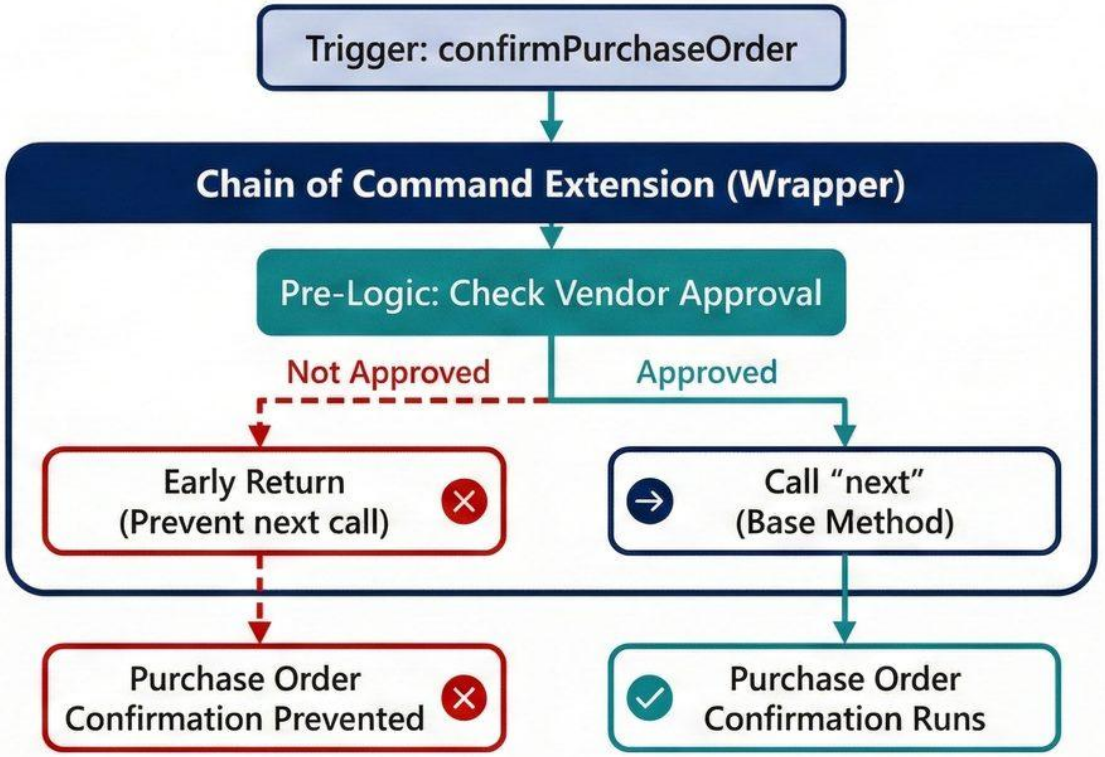
Wraps business process methods (e.g., confirmPurchaseOrder).

Adds pre-logic checks before calling the base method.

Returns early to prevent subsequent execution if checks fail.

Integrates validation directly into the process flow.





Validation happens at the right moment - before confirmation runs.

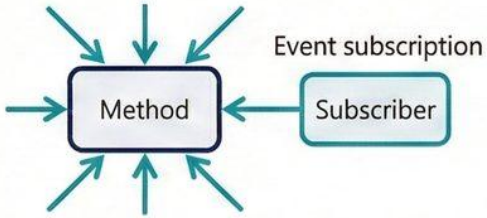
Boom takeaway: CoC enables powerful process-level validation by inserting pre-logic checks before the core business method executes.

truestack.dev

Extensibility Attributes in Dynamics 365 F&O

Controlling how your methods can be extended and defining your API surface.

[Hookable(true)]



Allows event handlers to subscribe. Enables observation points for subscribers.

✓ Mark observation-points as hookable.

[Hookable(false)]



Prevents event handler subscription. Use to protect critical methods from being hooked.

✗ Disables subscriptions.

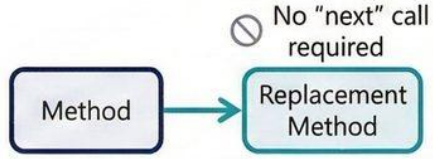
[Wrappable(false)]



Prevents Chain of Command wrapping. Ensures method logic is not wrapped by extensions.

✗ Mark critical methods as not-wrappable.

[Replaceable]



Allows Chain of Command to replace the method without requiring next. Permits full logic replacement.

✓ Enables full method replacement.

Strategic Usage

- Use these attributes to control your API surface.
- Mark critical methods as not-wrappable to protect them.
- Mark observation-points as hookable to allow subscribers.

These attributes make your intentions explicit to future developers.

truestack.dev

Intentionally Guiding Extension Points with Hookable Attribute

Hookable(true) for Developers to Observe or Extend

```
[Hookable(true)]
public void processOrder() { ... }
```

Developers log when order is processed

Mark hookable so they can subscribe and observe logic.

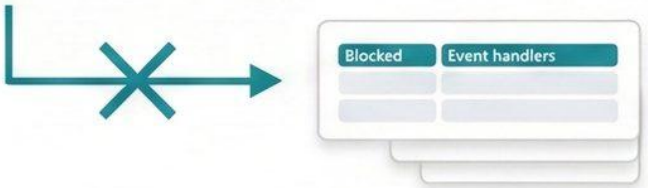


Hookable(false) for Security and Stability

```
[Hookable(false)]
public boolean ValidateSecurity() { ... }
```

Prevents event handlers from bypassing

Security-sensitive or core methods should never allow bypass. Protects security and stability.



Mark your public APIs intentionally to guide extension points and protect core logic.

truestack.dev

SysObsoleteAttribute: Enforcing Deprecation and Refactoring

Forces code updates through compile-time errors for a clean codebase.

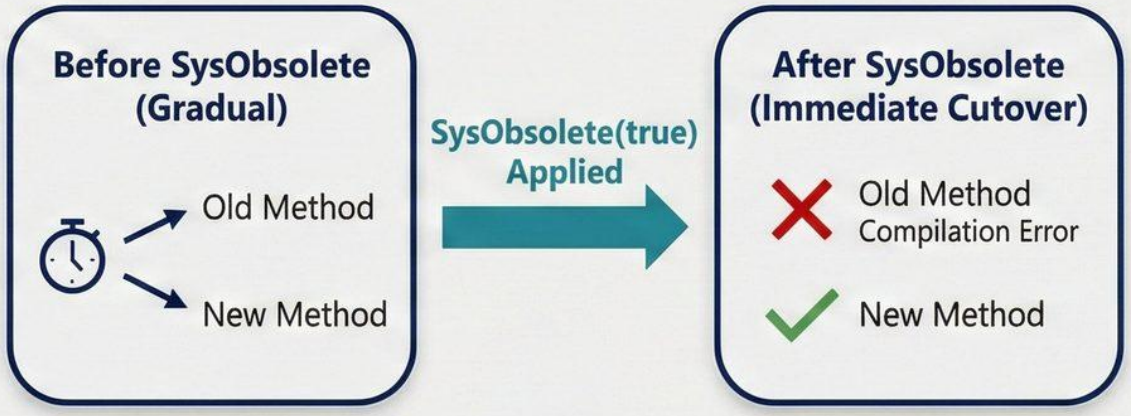
Purpose and Function

- Marks methods as deprecated to signal replacement.
- Second parameter set to 'true' triggers a compilation error.
- Prevents usage of outdated code, enforcing immediate updates.

```
[SysObsolete("Use newMethod() instead.", true)]
public void oldMethod()
{
    // Deprecated logic
}
```

Causes compilation error when called.

Impact on Development Workflow



Code fragments call both, leading to technical debt.

Developers are forced to use new method, ensuring a clean codebase.

Enforces Refactoring Across Large Codebases.

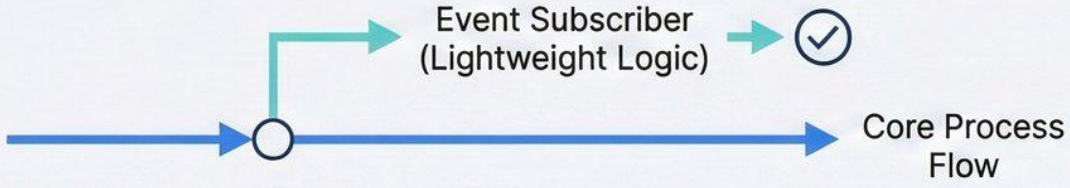
truestack.dev

Event Handlers vs. Chain of Command

Choosing the Right Tool for Customization

Event Handlers

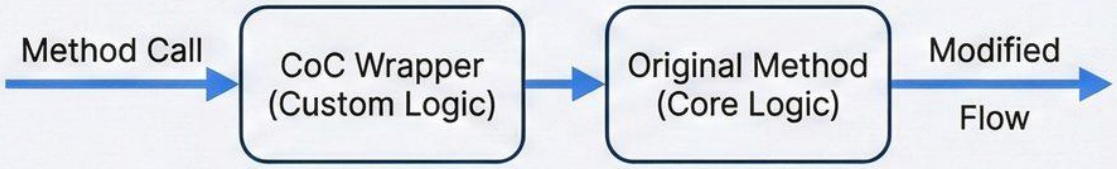
- ✓ **Purpose:** Observation & lightweight logic
- ✓ **Action:** Subscribe to events, check conditions, potentially block actions
- ✓ **Characteristic:** Lightweight and non-invasive
- ✓ **Stability:** Higher stability (non-invasive nature)



Best for: Monitoring and minor interventions without altering core flow.

Chain of Command

- ✓ **Purpose:** Wrapping or replacing methods
- ✓ **Action:** Heavy modification of logic, process changes, behavioral overrides
- ✓ **Characteristic:** Deeply integrated and impactful
- ✓ **Design:** Requires careful design (replacing core logic)



Best for: Significant behavioral changes and process overhauls.

Choose **Event Handlers** for non-invasive observation, and **Chain of Command** for deliberate behavioral overrides.

truestack.dev

Universal Data Integrity: Implementing VendTable Validation

Business Rule & Implementation (Validation Handler)

Validation Rule

Prevent saving high-risk vendors without approval.



Handler Implementation




```
function validateWrite() {
  if (this.RiskRating == 'High') {     // Check risk rating
    checkFailed('High-risk vendor cannot be saved without approval.');
    return false;                      // Block save & show message
  }
}
```

Universal Enforcement Flow (All Access Points)



Final Outcome: Professional-grade validation enforces data integrity universally across all entry points, blocking any unauthorized high-risk saves.

This approach ensures consistent validation logic, preventing data inconsistencies regardless of how the data is modified.

truestack.dev

Four Critical Dynamics 365 F&O Dev Mistakes

Common pitfalls that reduce code quality and extensibility, and how to avoid them.

Mistake One: Validation in Insert/Update

✕ Putting validation in insert or update.

These persist data, they don't validate it.



✓ Use validateWrite instead.

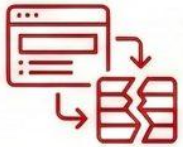
Ensures data is validated before persistence.



Mistake Two: UI-only Validation at Table Level

✕ Putting UI-only validation at table level.

Affects integrations and batch jobs without form context.



✓ Use Use form-level validation.

Keeps UI logic contained within the form context.



Mistake Three: Missing the Hookable Attribute

✕ Missing the Hookable attribute.

Your intentions are unclear to future developers.



✓ Mark it explicitly.

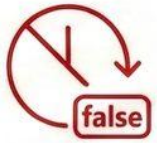
Clearly indicate extensibility intentions.



Mistake Four: SysObsolete with False

✕ SysObsolete with false.

Soft deprecation that developers will ignore.



✓ Use true to force refactoring.

Creates a compile-time error, enforcing change.



These mistakes reduce code quality and extensibility. Adopt the correct patterns for robust solutions.

truestack.dev

Microsoft Dynamics 365 Finance & Operations

Professional Event Handling Patterns

1. Domain-Based Organization



Group by business domain for findability

2. Descriptive Naming

```
[SubscribesTo(...)]
public static void Vendor_OnValidatingWrite(...)
{
    ...
}
```

✓ Clear & Specific

```
[SubscribesTo(...)]
public static void onEvent(...) {
    ...
}
```

✗ Generic & Vague

Avoid generic names; be descriptive.

3. Final for Handlers

```
[SubscribesTo(...)]
public final class VendorEventHandler {
    ...
}
```

Mark handler classes `final` to prevent accidental extension. They are subscription endpoints, not extension points.

4. Appropriate Scoping

Table-Level Scope → Global Rules (Applies everywhere)

Choose based on impact

Form-Level Scope → UI-Specific Workflows

5. System Reliability

Discipline in event handling makes the system reliable.

truestack.dev

Recap: Topic 4.2 – Validation, Events, and Extensibility

Validation Trigger Points

validateWrite: Primary table validation trigger. Runs for all save operations.

validateField: Validates individual fields during form entry.

insert & update: Persist data only. They don't validate.

Extensibility Mechanisms

Event Handlers:

- Subscribe to events without modifying code.
- Create handler class, implement static methods, decorate with event attributes.

Chain of Command (CoC):

- Wraps business process methods for conditional logic.
- Pre-logic validates before calling 'next'.

```
... {
  // Pre-logic here;
  next wrapper();
  // Post-logic here;
}
```

Controlling Extensibility & Lifecycle

Hookable Attribute:

- Public APIs: Mark as hookable (✓)
- Security methods: Mark as non-hookable (✗)

SysObsolete Attribute:

```
[SysObsolete("Use
<NewMethod>.", true)]
```

Second parameter 'true' forces compilation error, preventing use.

Event Handling Best Practices

- Organize by domain.
- Use descriptive names.
- Mark handlers final.
- Use appropriate scope.

Next Topic: 4.3 Table and Enum Extensions – Customizing data structures, extending enums, and adding fields to tables safely.

truestack.dev

Topic 4.3: Table and Enum Extensions

Mastering Safe, Upgradeable, and Maintainable Data Structure Extensions in D365 F&O

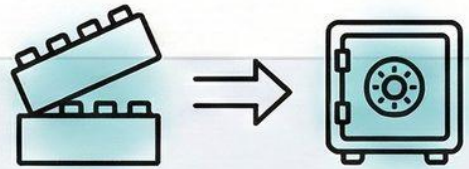
⚠ Common Mistakes (Risky & Fragile)

- ✗ Edit standard enums directly
- ✗ Edit standard enums directly
- ✗ Break upgrades
- ✗ Use integer values instead of enum literals
- ✗ Cause fragile code



✓ Correct Approach (Safe & Maintainable)

- ✓ Extend enums and tables safely
- ✓ Extend enums and tables safely without modification
- ✓ Use enum literals for robust logic
- ✓ Ensure seamless upgrades
- ✓ Master extension patterns

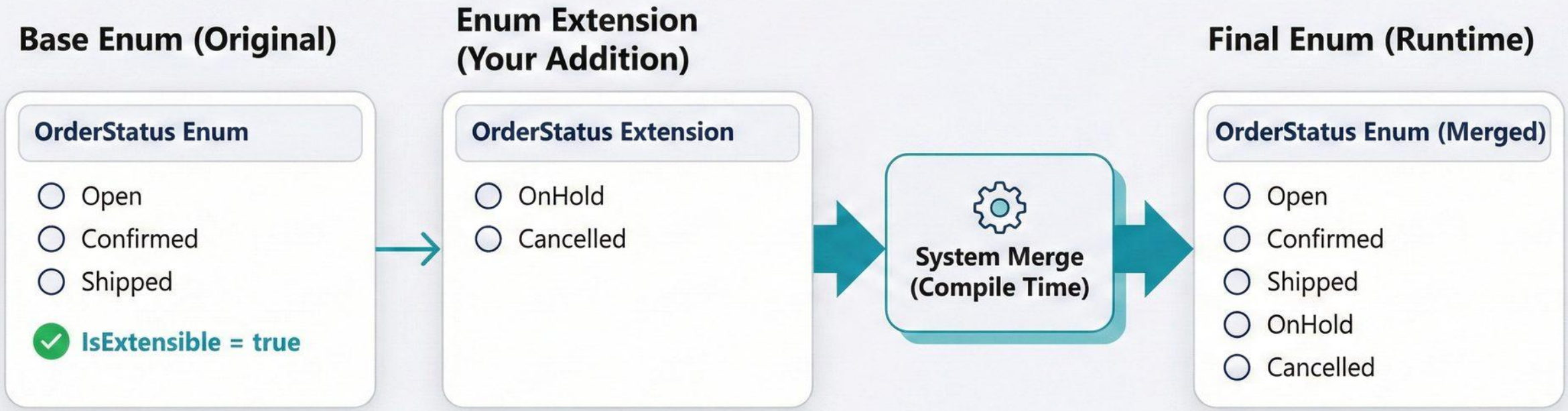


Extend data structures the right way—safe, upgradeable, and maintainable.

truestack.dev

Enum Extensions in Dynamics 365 F&O

Add values to standard enums without modifying the base object.



Key Benefits

- Base enum remains untouched
- Microsoft updates don't break the extension
- Requires base enum IsExtensible set to true

Enum extensions allow for safe, additive customization while preserving the integrity and upgradeability of the base application.

truestack.dev

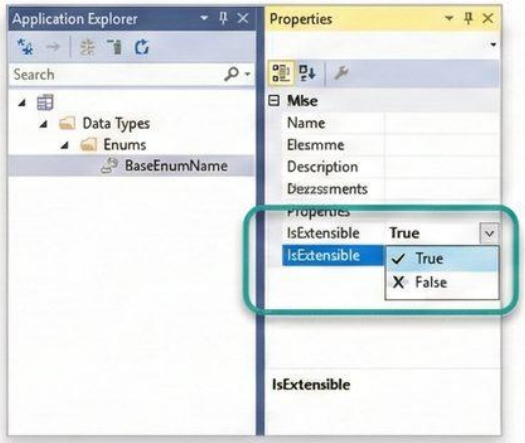
Understanding Enum Extensibility: The IsExtensible Property

The Gateway Property



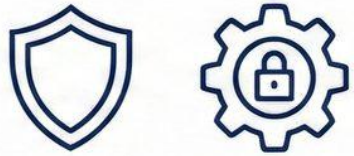
IsExtensible is the fundamental property that determines if an enum can be modified. It acts as the entry point for creating enum extensions.

Checking the Property



Open the base enum in Application Explorer and inspect the IsExtensible property value.

Why Non-Extensible?



Typically restricted for security-sensitive enums security-sensitive enums or core system enums where Microsoft requires tight control to maintain integrity.

Extensibility Request Process



If you encounter a non-extensible enum you need to extend, submit an extensibility request to Microsoft for evaluation.

Always check the **IsExtensible** property first before planning any enum extension to determine feasibility and required actions.

truestack.dev

Enum Extension Process



1 Verify Extensibility

Check that base enum's IsExtensible property is set to true.

2 Create Extension

Right-click base enum in Application Explorer, select Create Extension.

3 Add Elements

Add new elements with clear names like OnHold, Cancelled.

4 Build Project

Compile to merge extension into the base enum.

5 Use in Code

Reference new values, e.g., OrderStatus::OnHold.

Outcome: This five-step process is clean, safe, and upgradeable for adding enum values.

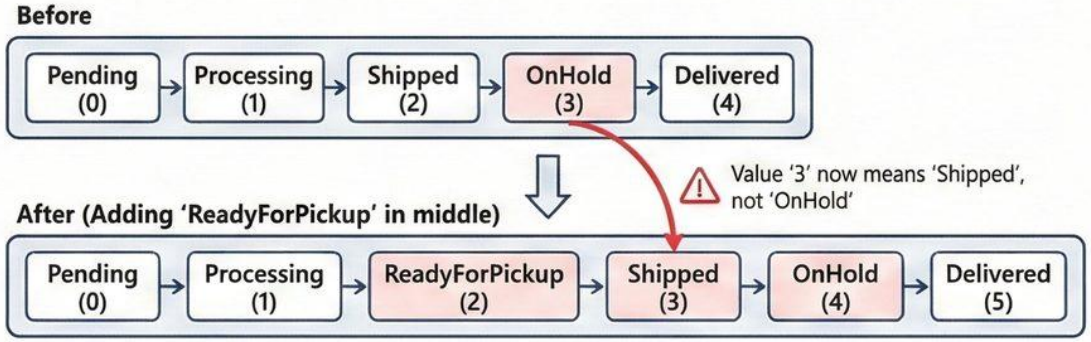
truestack.dev

Critical Rule: Avoid Integer Values for Enum Comparisons

Enums are assigned unstable integers internally; always use enum literals.

INCORRECT: Using Integer Values (Anti-Pattern)

Problem: Integer Assignment is Not Stable



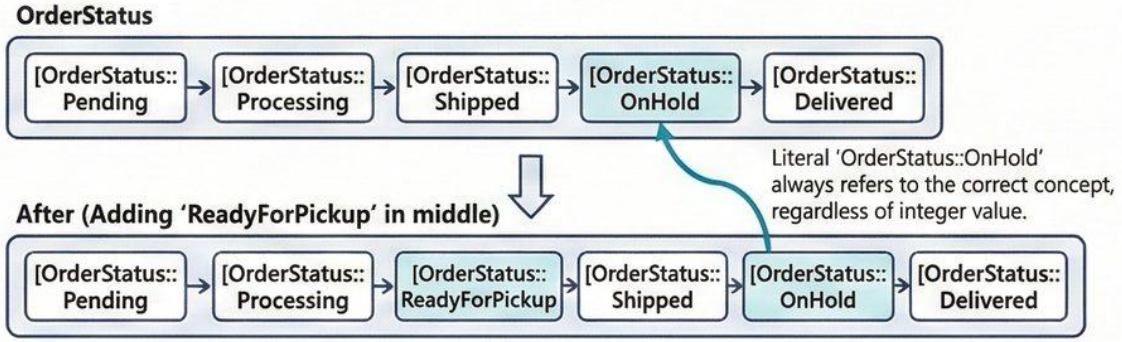

```
// INCORRECT: Relying on integer value
if (status == 3) // Checks for 'OnHold' based on old assignment
{
    // Logic for OnHold status...
}
```

Code referencing '3' now breaks or performs wrong logic.

Result: Code Breaks, Logic Errors

CORRECT: Using Enum Literals (Best Practice)

Solution: Compiler Validates Literals




```
// CORRECT: Using enum literal
if (status == OrderStatus::OnHold) // Compiler ensures this exists and is correct
{
    // Logic for OnHold status...
}
```

Compiler validates existence; code is self-documenting.

Result: Stable, Safe, Readable Code

Takeaway: Always use Enum Literals (e.g., OrderStatus::OnHold) for stability, compiler validation, and code readability. This is non-negotiable.

truestack.dev

Anti-Pattern: Range Checks for Enums

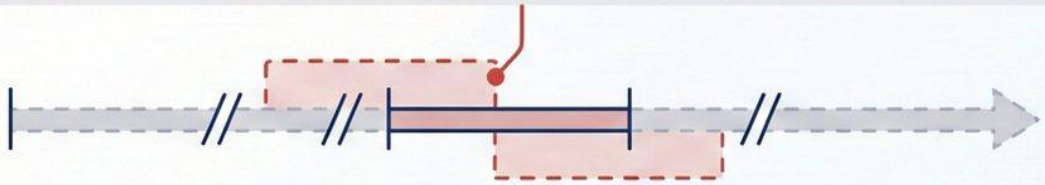
INCORRECT: Range Check Anti-Pattern

```
// Fragile and dangerous approach
if (status >= 3 && status <= 5)
{
    // Logic for OnHold, Cancelled, Pending
    processStatus(status);
}
```

Assumes consecutive integers

Fragile: New value in middle changes meaning

Meaningless if enum is reordered



Unreliable Range Definition

CORRECT: Explicit Switch Statement

```
// Crystal clear and robust approach
switch (status)
{
    case OrderStatus::OnHold:
    case OrderStatus::Cancelled:
    case OrderStatus::Pending:
        // Explicitly handle these statuses
        processStatus(status);
        break;
    // default case not shown for brevity
}
```

Explicitly lists cared-about statuses

Compiler validates every case

Crystal clear for future developers



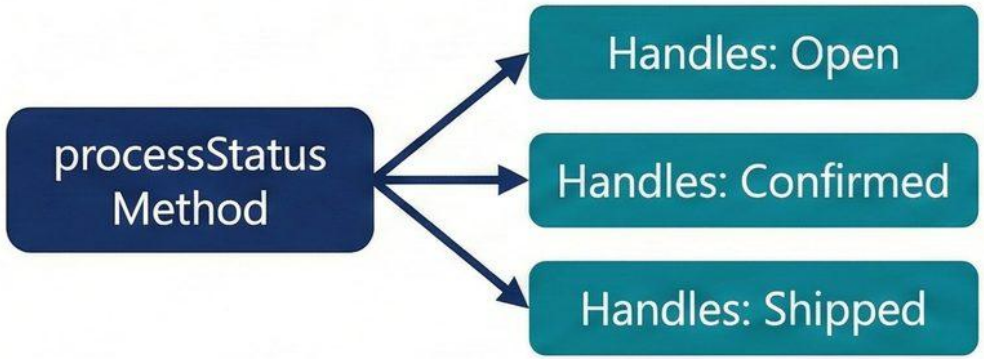
Explicit & Validated Enum Values: OnHold, Cancelled, Pending

Never use ranges; always be explicit with enum values for robust and maintainable code.

truestack.dev

Extending Enum Logic with Post Event Handlers

Base Method (Untouched)



Base logic for original values remains unmodified.

Executes After

Post Event Handler (Extension)

```
[PostHandlerFor(..., processStatus)]
public static void ...Post(...) {
    switch (status) {
        case Enum::OnHold: ... break;
        case Enum::Cancelled: ... break;
        // Base values already handled
    }
}
```



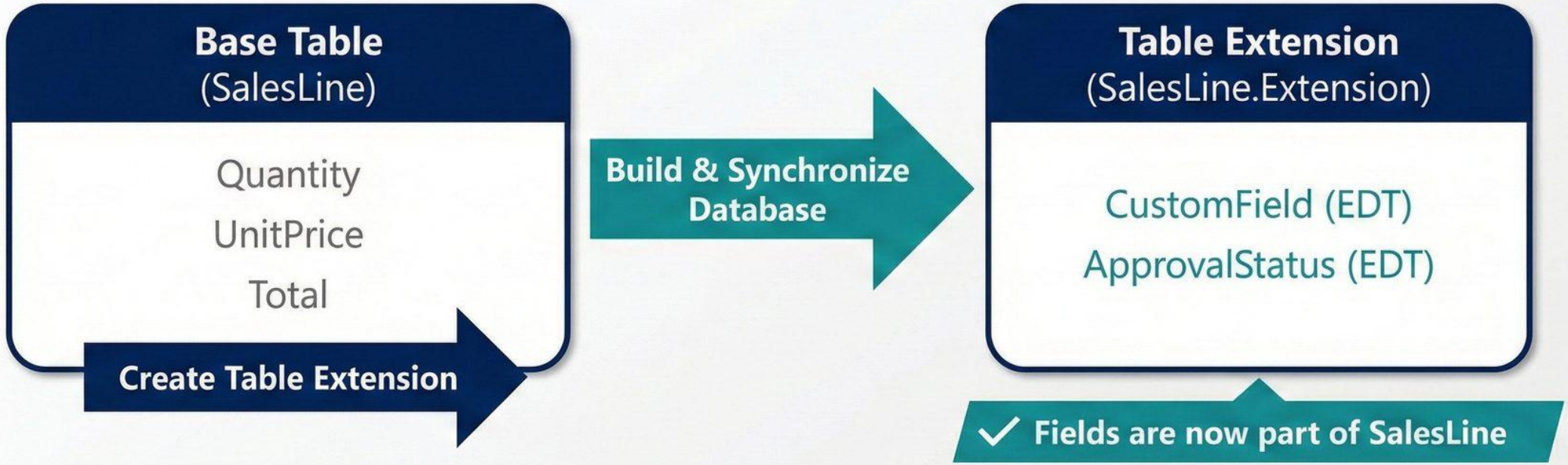
Extension logic for new values only.

✓ Clean Separation: Base method remains untouched, extension adds behavior for new values.

truestack.dev

Table Extensions in Dynamics 365 Finance & Operations

Adding fields to standard tables using the extension model



1. **1. Create Table Extension**
2. **2. Add Custom Fields & EDTs**
3. **3. Build & Synchronize**

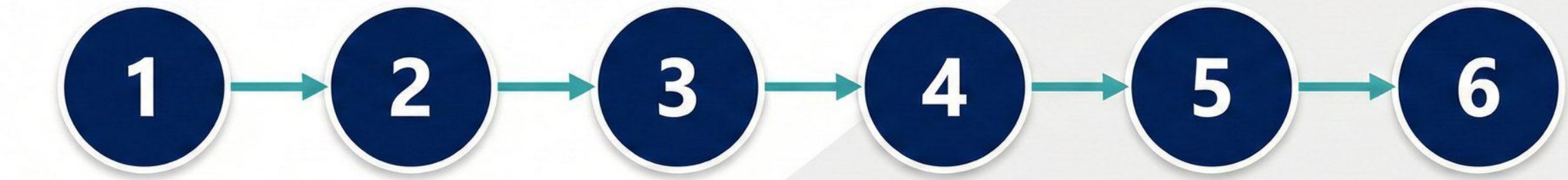
→ **Form Extensions to Display Fields**

Standard pattern for adding data:
Never modify the base table; always create a table extension.

truestack.dev

Table Extension Process

A safe and complete six-step workflow for extending tables and forms



1 Create Table Extension Object

Right-click the base table and select Create Extension to generate the extension object.

2 Add New Fields

Add fields using the correct EDT (e.g., string for CustomField, enum for ApprovalStatus).

3 Add to Field Groups

Include new fields in field groups to facilitate their display on related forms.

4 Build Project

Run compilation to validate the extension and ensure code integrity.

5 Synchronize Database

Update the SQL schema to include your new fields and metadata changes.

6 Extend Forms

Create form extensions to display the new fields, integrating with table extensions.

Outcome: A safe, validated, and complete **process for extending tables and forms in Dynamics 365 Finance & Operations.**

truestack.dev

Table Extension Property Modifications

Understanding safe vs. protected properties to ensure system integrity.

ALLOWED: Safe to Change ✅

System Tracking Fields

- CreatedBy
- FasswordBy
- ModifiedBy

UI References & Display

- FormRef
- PreviewPartRef
- TitleField1
- TitleField2

Metadata

- Tags

These properties are safe to change in extensions and do not affect the database schema.

NOT ALLOWED: Protected Properties ❌

Structural & Schema

- ⚠ Primary Index
- ⚠ Clustered Index
- ⚠ Table Group

Changing these would break the table structure and relationships.

Protected Elements

- ⚠ Delete Actions
- ⚠ Relations

Trying to change protected properties results in compilation errors. Respect the boundaries.

Understand the extension boundaries: modify only safe properties to maintain system stability and avoid errors.

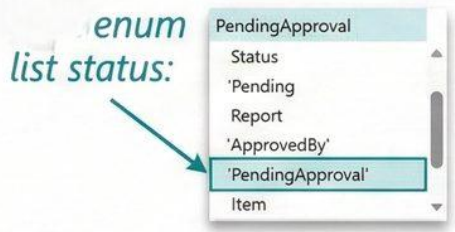
truestack.dev

Scenario: Sales Order Approval Workflow via Extensions

Implementing a business requirement without modifying standard code.

1. Extend OrderStatus Enum

Add new status: 'PendingApproval'.



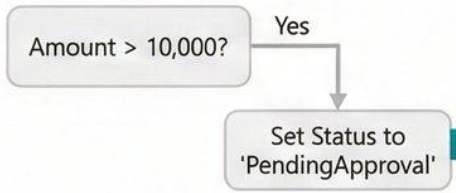
2. Extend SalesTable

Add new fields: 'ApprovalDate', 'ApprovedBy'.



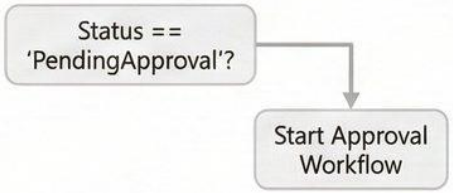
3. Create 'on saving' Event Handler

Check if order amount > 10,000.



4. Create 'on processStatus' Event Handler

Trigger approval workflow when status hits 'PendingApproval'.



Key Benefits: Combines extensions (Enum, Table, Event Handlers) in harmony. No standard code modified. Safe and completely upgradeable.

truestack.dev

Four Critical Mistakes with Dynamics 365 Extensions

Mistake 1: Integer Values for Enum Comparisons

INCORRECT

```
// Avoid: Integer values
if (myRecord.Status == 1)
{
    // Logic breaks if enum changes
}
```

Fragile, breaks the moment the enum changes.

→ ⚠ **Production Risk**

CORRECT

```
// Use: Enum literals
if (myRecord.Status ==
Status::Approved)
{
    // Stable and readable
}
```

Always use enum literals instead.

→ **Stable Solution**

Mistake 2: Doing Numeric Range Checks

INCORRECT

```
// Avoid: Range checks
if (myRecord.Status >= 1 &&
    myRecord.Status <= 3)
{
    // Assumes consecutive values
}
```

Breaks when enums get reordered or extended.

→ **Maintenance Issue**

CORRECT

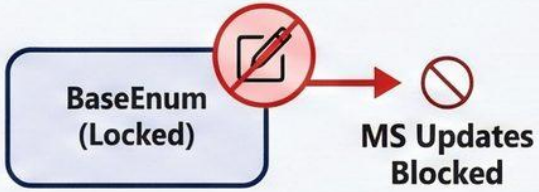
```
// Use: Explicit cases
switch (myRecord.Status)
{
    case Status::Draft:
    case Status::Review:
        // Explicitly list cases
    break;
}
```

Always list out explicit cases.

→ **Clear & Safe**

Mistake 3: Overlaying Base Enum to Add Values

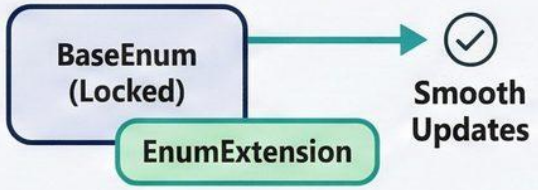
INCORRECT



Blocks Microsoft updates and breaks upgrades.

→ **Upgrade Failure**

CORRECT

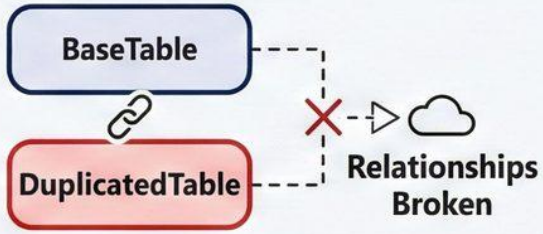


Always use enum extensions instead.

→ **Future-Proof**

Mistake 4: Duplicating Base Table to Add Fields

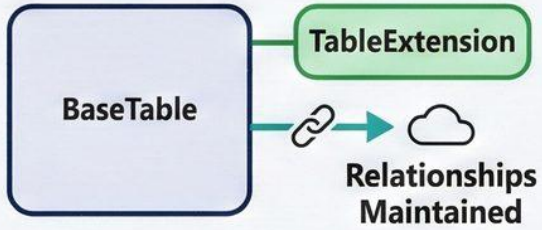
INCORRECT



Creates incompatible types and breaks relationships.

→ **Data Integrity Loss**

CORRECT



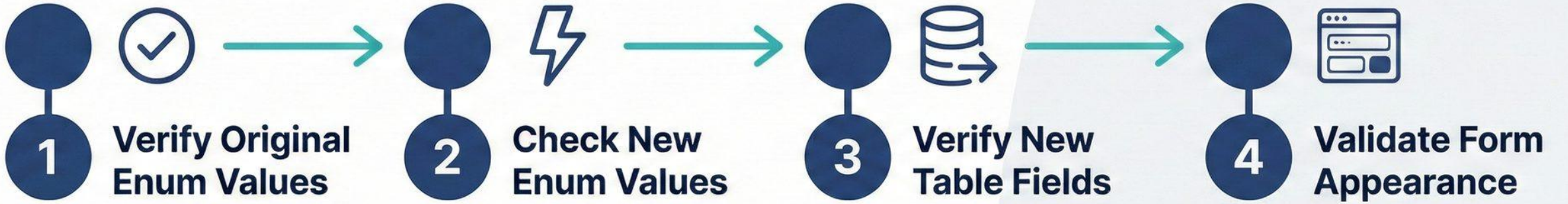
The right way? Table extensions.

→ **Consistent Types**

Avoid these anti-patterns to prevent real production problems and ensure maintainable Dynamics 365 solutions.

truestack.dev

Comprehensive Testing for Extensions



1 Verify Original Enum Values

Ensure base logic for original values (e.g., Open, Confirmed, Shipped) still functions perfectly. No regressions.

2 Check New Enum Values

Confirm event handlers execute correctly for new values (e.g., OnHold, Cancelled).

3 Verify New Table Fields

Save records with new fields populated, then query back to ensure data persists and retrieves correctly.

4 Validate Form Appearance

Open forms to verify new fields appear in correct locations with accurate labels.

Key Takeaway: Thorough testing prevents surprises in production. Remember, extensions must be additive and not break existing functionality.

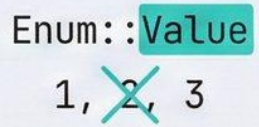
truestack.dev

Best Practices for Working with Extensions

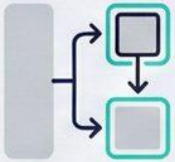
1. **Always Check IsExtensible Property**
   Don't just assume an enum can be extended - verify it first before planning any enum extensions.



2. **Use Enum Literals Everywhere**
   Never use integer values or ranges. Always reference them as OrderStatus::Value.



3. **Keep Event Handlers Minimal**
   They should only handle your extension values - don't duplicate the base logic.



4. **Document All Your Extensions**
   List out the new enum values, table fields, and handlers added for future maintainers.



5. **Always Synchronize the Database**
   After making table extension changes, sync to ensure new fields exist in SQL.



Following these practices ensures your extensions are high-quality and maintainable over time.

truestack.dev

Topic 4.3 Recap: Extending Data Structures Safely

Key Concepts & Best Practices

- ✓ **Enum Extensions**: Base enum must have `IsExtensible = true`; create extension, add values, use EnumName::**Value**.
- ✓ **Enum Literals**: ALWAYS use enum literals for comparisons (e.g., **Status::NewValue**).
- ✓ **Event Handlers**: Post handlers extend logic for new values; base methods remain unchanged.
- ✓ **Table Extensions**: Add fields with correct EDTs, build, and synchronize database.
- ✓ **Table Properties**: Limited changes allowed (CreatedBy, FormRef, TitleFields); structural properties are immutable.
- ✓ **Comprehensive Testing**: Verify base behavior, extension values, and form display.

Avoid Common Pitfalls

- ✗ **NEVER** use integers or numeric ranges for enum comparisons; they are fragile.

```
// INCORRECT: Integer comparison
if (myRecord.Status == 5) { ... }

// CORRECT: Enum literal comparison
if (myRecord.Status == Status::NewValue) { ... }
```

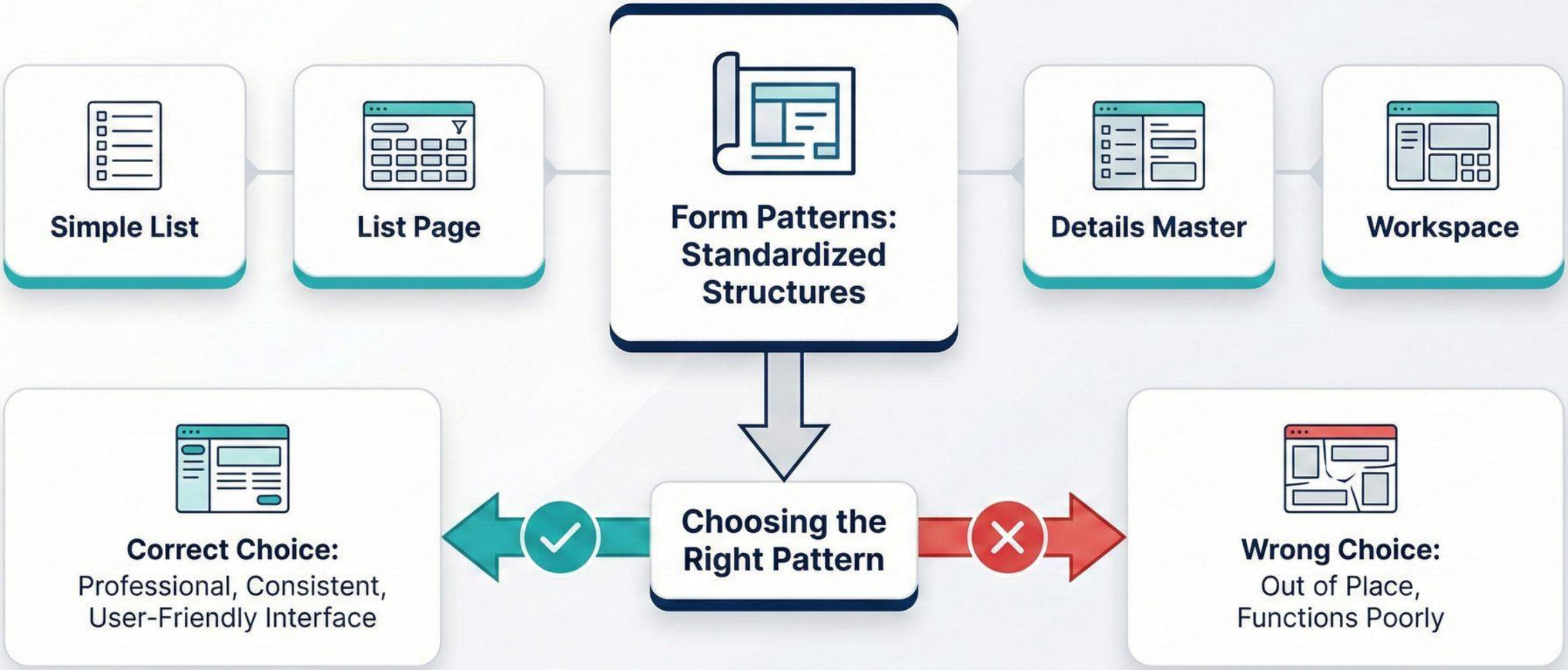
- ✗ Avoid overlayering or duplicating base objects.

Next Up: Topic 5.1: Form Patterns – Building Consistent & Professional UIs

truestack.dev

Topic 5.1: Form Patterns

Building Consistent and Intuitive User Interfaces



truestack.dev

Understanding Form Patterns in D365 F&O

Standardized combinations of UI components that define form structure, required controls, and behavior.

Core UI Components



Action Pane: Commands & Buttons

Holds commands, operations, and buttons.



Navigation List: Filters & Organizes

Filters or organizes records logically.



Grid: Displays List of Records

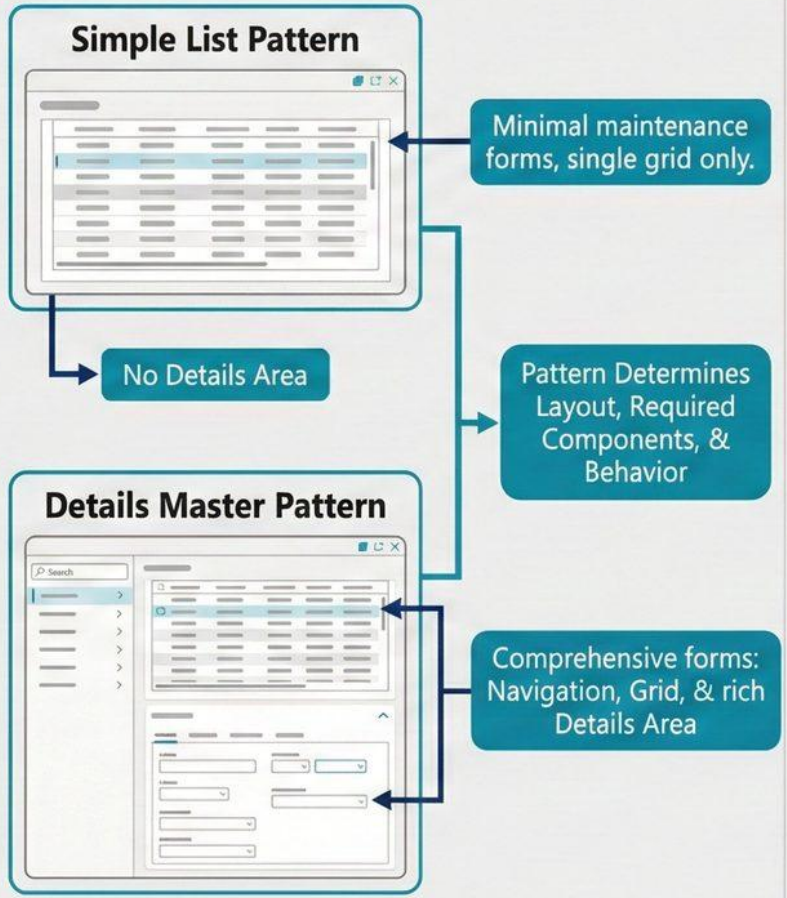
Shows rows and columns of data.



Details Area: Fields for Editing

Shows fields for data entry and detailed view.

Pattern Examples & Combinations



Pattern Purpose & Value



Serves Specific Business Purpose

- Optimized for the task (e.g., data entry, analysis, maintenance).

Ensures Consistency & Compliance

- Follows D365 standards for usability and accessibility.

Reduces Design Errors

- Pre-defined structures prevent common design mistakes.

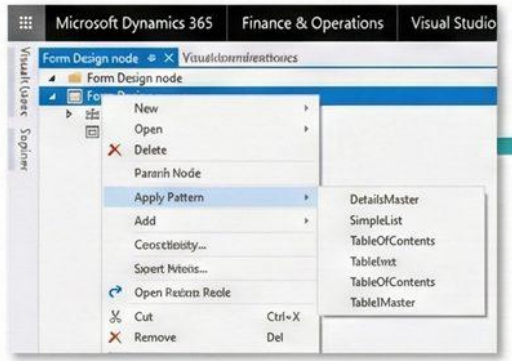
Choosing the right pattern ensures the form fits requirements and D365 standards.

truestack.de

Two Supported Methods to Apply Form Patterns

Method 1: Through the Designer

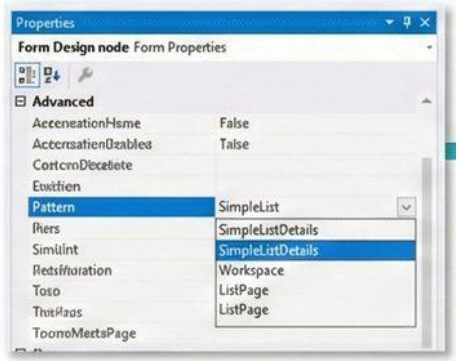
1. Right-click **Design** node
2. Select **Apply Pattern**
3. Choose **Pattern** from List
4. Results: Automatic Structure & Components



✓ Visual & Immediate Results

Method 2: Through Metadata

1. Open **Design** node properties
2. Locate **Pattern** property
3. Set **Pattern** Value
4. Results: Assigned at Metadata Level



✓ Safe & Supported

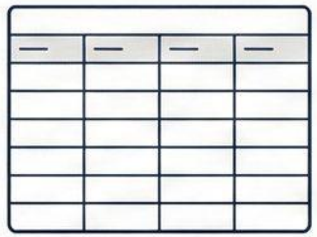
Both methods are safe and supported; use the Designer for immediate visual feedback or Metadata for direct property assignment.

truestack.dev

Microsoft Dynamics 365 F&O Form Patterns

Primary patterns optimized for specific user experiences and data complexities.

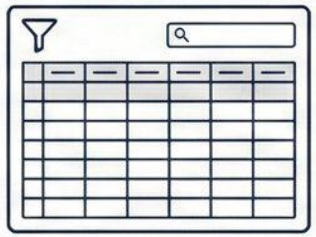
Simple List



Minimal grid layout. Perfect for basic reference data.

e.g., Units of measure.

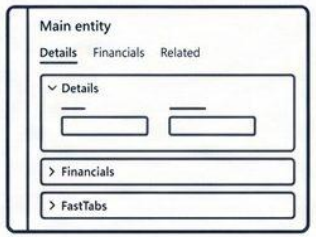
List Page



Handles large datasets with advanced filtering.

e.g., Master data like customers.

Details Master



Rich experience: entity plus related records in tabs.

e.g., Complex data like sales orders.

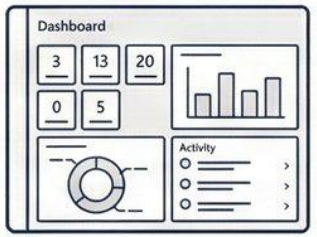
Simple List and Details



Middle ground: grid with a details panel.

e.g., Moderate complexity data.

Workspace



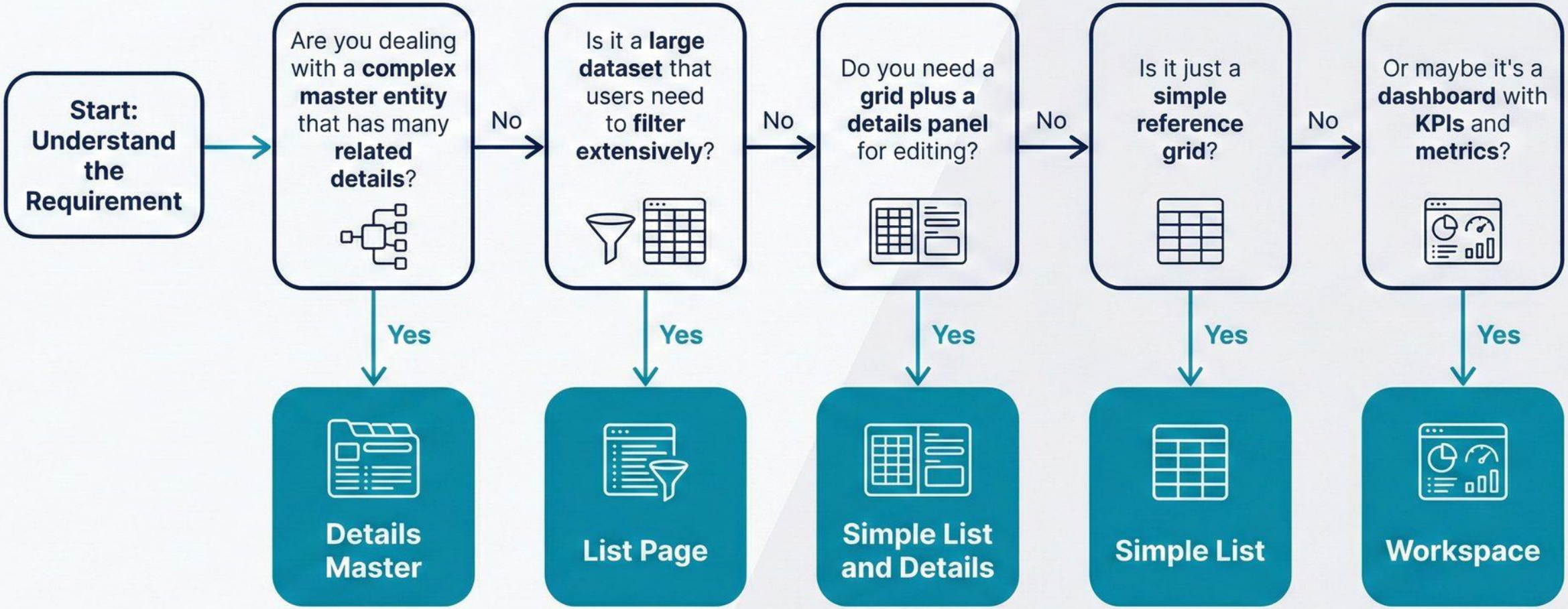
Dashboards and monitoring with KPIs and tiles.

e.g., Operational monitoring.

Choosing the right pattern ensures the form feels natural and performs well.

truestack.dev

Pattern Selection Decision Tree

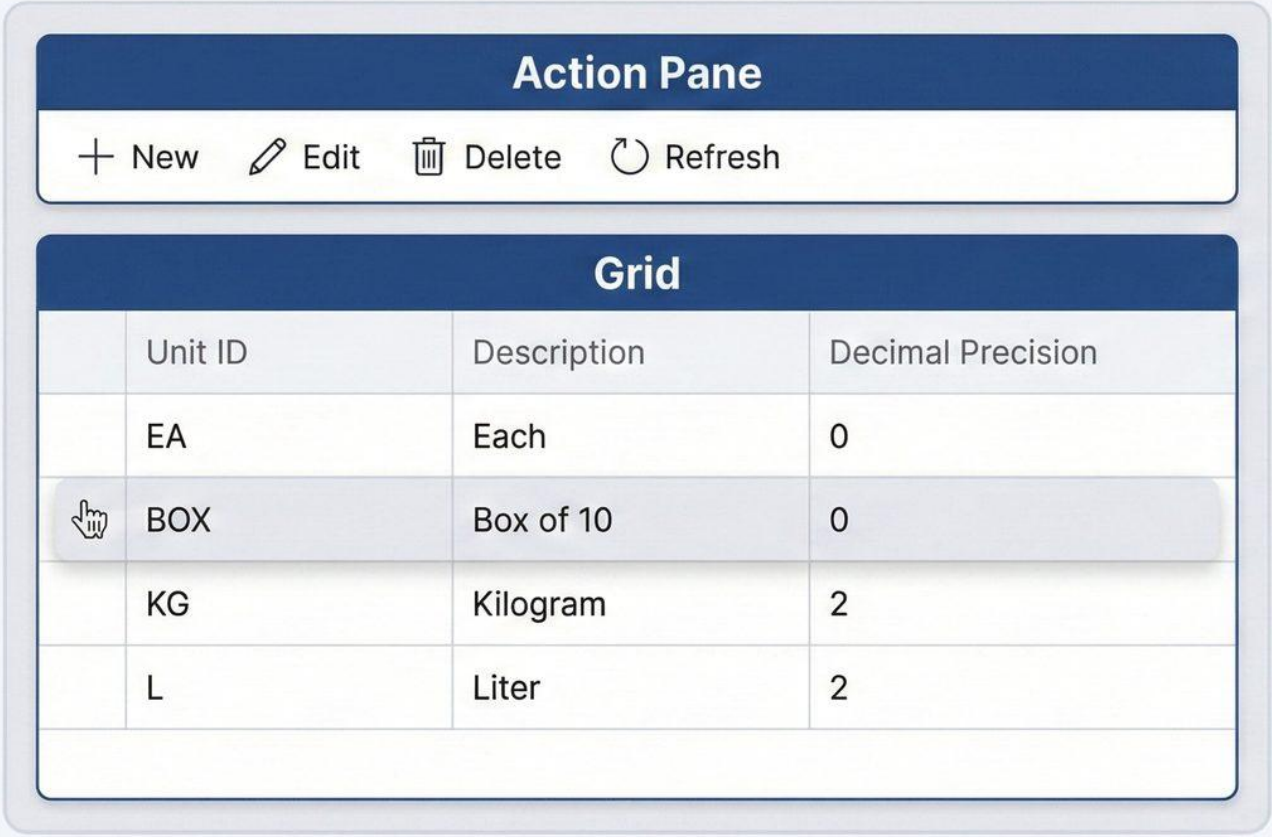


This systematic approach ensures you're choosing the pattern that actually matches the requirement.

truestack.dev

Simple List Pattern

The Most Minimal Form Pattern



Key Characteristics & Usage

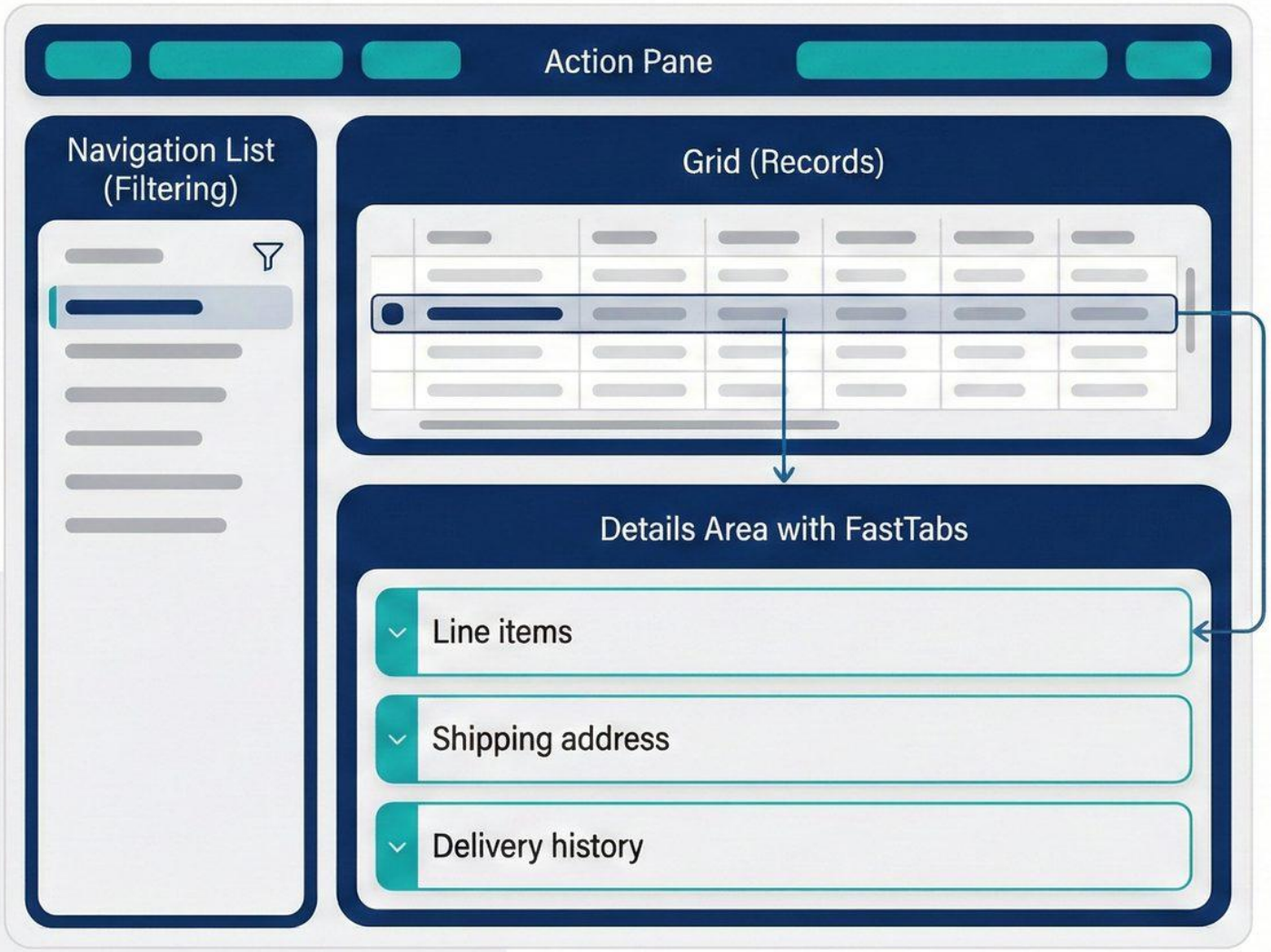
- **Minimal Structure:** Action Pane + Grid only.
- **Data Operations:** Users can Add, Edit, or Delete records directly.
- **Best for Reference Data:** Ideal for Units of Measure, Currencies, etc., with few fields.
- **Quick to Implement & Navigate.**

Details Panel	No separate details area for expanded record view.
Complex Navigation	No multi-level hierarchies or complex filtering structures.

Choose Simple List when you need a quick, straightforward interface for managing simple, flat lists of data without extra complexity.

truestack.dev

Details Master Form Pattern



Key Components & Functionality

- Action Pane
- Navigation List for filtering
- Grid for records
- Rich Details Area with FastTabs (Vertical Tabs)

Perfect For Complex Entities

Example: Sales Orders

Header Grid shows orders; clicking an order updates the Details Area with line items, shipping address, and delivery history in separate tabs.

About FastTabs

Collapsible sections that reduce visual clutter while keeping information accessible.

Details Master is powerful, but requires a complex entity structure to justify its complexity.

truestack.dev

Simple List and Details Pattern

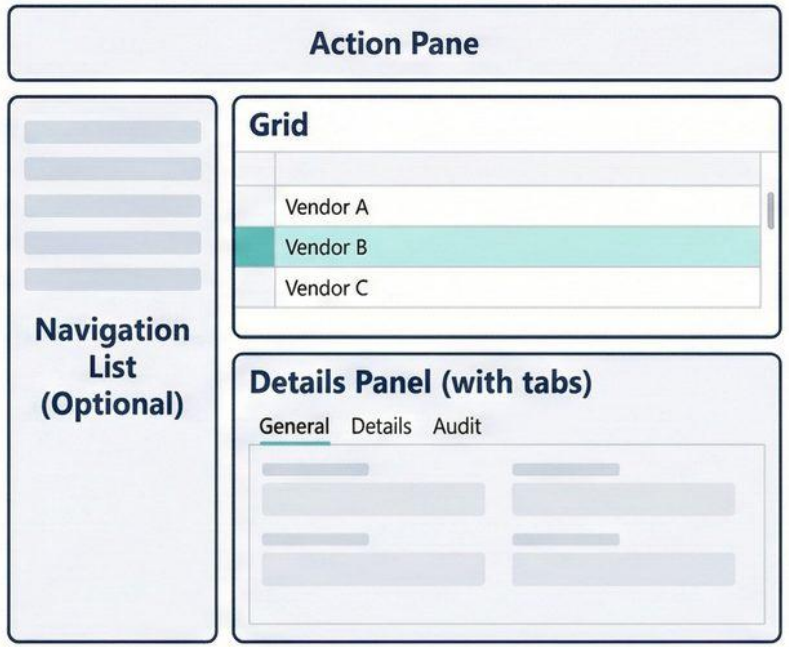
The balanced middle ground for effective data management.

PATTERN OVERVIEW & PURPOSE

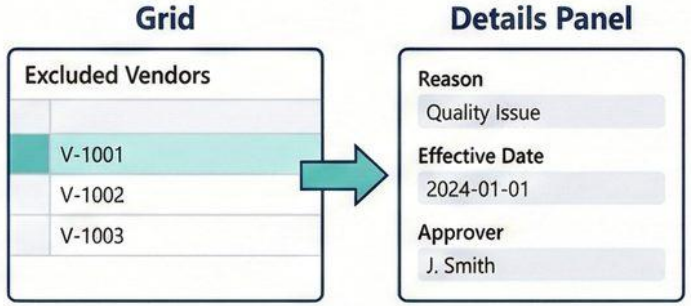


Middle ground pattern balancing minimalism and richness. Not as basic as 'Simple List', not as complex as 'Details Master'.

STRUCTURAL COMPONENTS



EXAMPLE: VENDOR EXCLUSION LIST



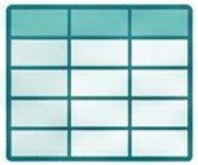
- Users can add, edit, delete exclusions easily.
- Avoids overwhelming complexity.
- Ideal for custom tables needing grid + detailed editing.

This pattern offers the perfect balance for scenarios requiring both a list view and detailed editing capabilities without unnecessary complexity.

truestack.dev

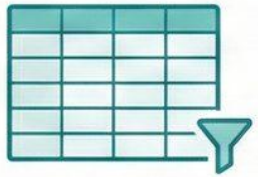
Form Pattern Comparison and Selection

SIMPLE → COMPLEX



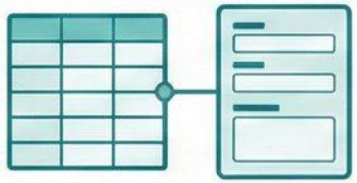
Simple List

Simplest layout. Just a grid, minimal complexity, no filtering, for basic reference data.



List Page

Large datasets with advanced filtering but no details panel.



Simple List and Details

Grid plus details panel, moderate complexity, basic filtering, for custom tracking.



Details Master

Most complex. Navigation, grid, rich details, advanced filtering, for master entities.

Think of this as a spectrum from left (simple) to right (complex).

Find where your requirement sits on that spectrum and match it to the pattern for straightforward selection.

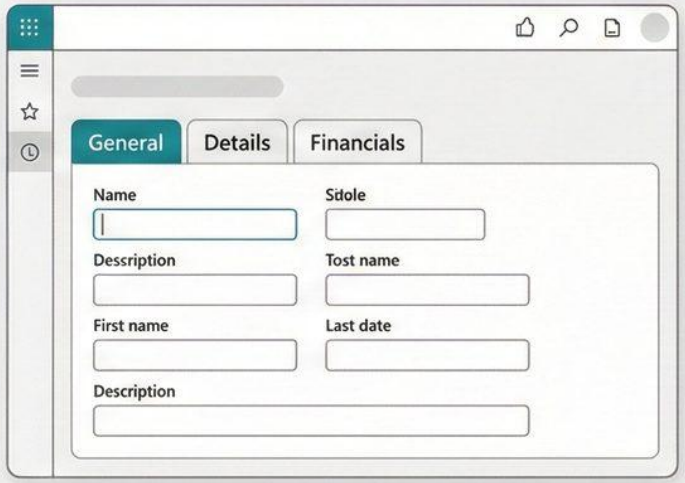
truestack.dev

Microsoft Dynamics 365 Finance & Operations

Form Tab Styles and Patterns

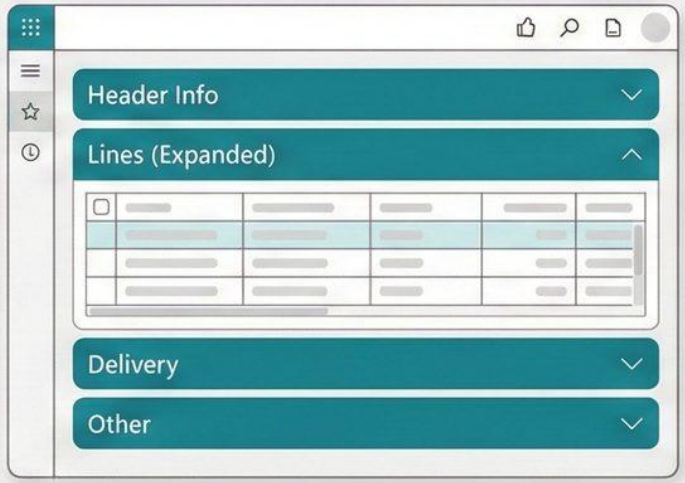
Understanding Horizontal, FastTabs, and Inline Tabs

Horizontal Tabs



Traditional style across the top. Older pattern, but still supported.

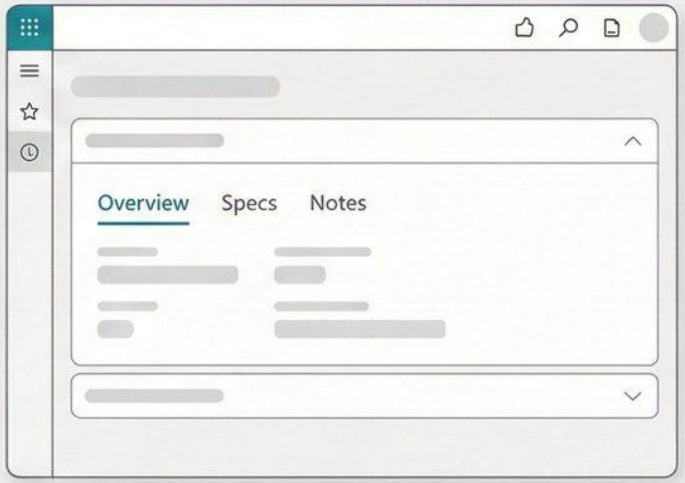
FastTabs



Vertical tabs along the left, collapsible like accordions. Modern, space-efficient, and reduce visual clutter. Used in Details Master and Simple List and Details patterns.

Modern and preferred

Inline Tabs



Minimal visual representation for specific sections within a form. Fits different design goals.

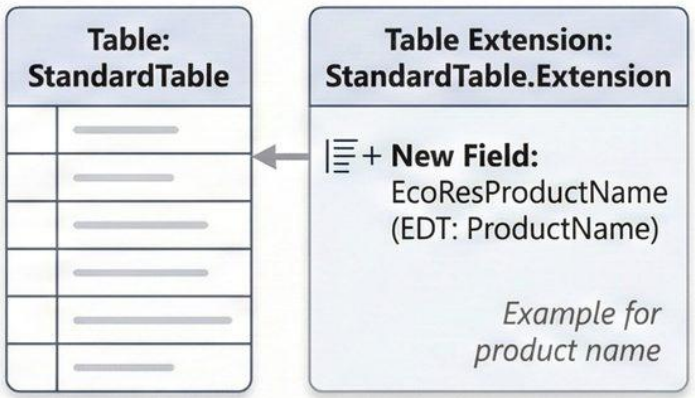
FastTabs are generally preferred for modern forms because they are space-efficient and highly intuitive.

truestack.dev

Extending Forms Safely: The Three-Layer Approach

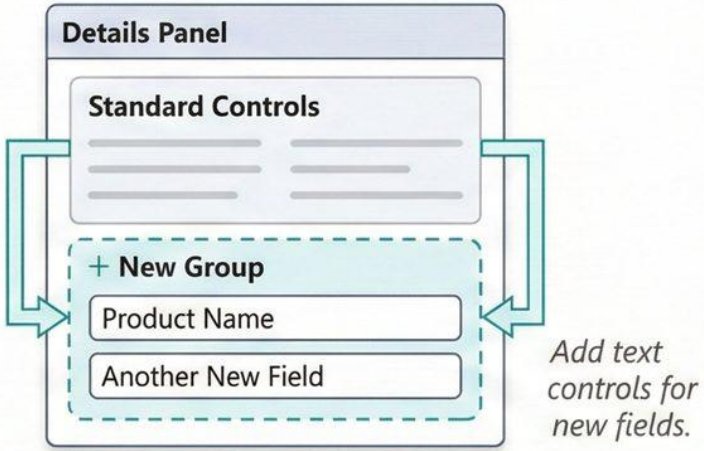
Layer 1: Table Extension

- Add new fields using correct EDTs.



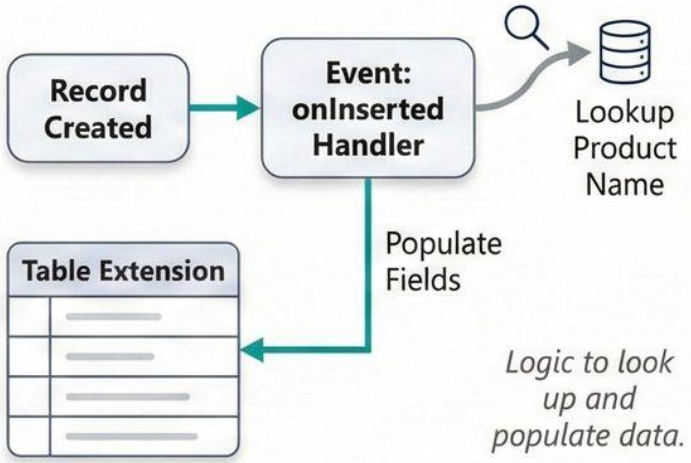
Layer 2: Form Extension

- Add UI controls to display those fields.



Layer 3: Event Handlers

- Populate the fields using onInserted handler.



All Three Layers Work Together

- ✓ No Standard Code Modification
- ✓ Safe & Upgradeable
- ✓ Keeps Code Clean & Maintainable

truestack.dev

Real-World Example: Form Extension Process



Extend Table

Add EcoResProductName EDT field and InventBatchId EDT field for batch.



1 Extend Table

Add EcoResProductIName EDT field and InventBatchId EDT field for batch.

2 Extend Form

Add a new group in the details panel for inventory details.

3 Add Controls

Add text controls that bind to these new fields.

4 Create Event Handler

Subscribe to onInserted event.

5 Receive Record

Handler receives the new record.

6 Populate Product

Look up product information and populate product name field.

7 Populate Batch

Get batch information and populate batch field.

Result: Users see the form with product name and batch automatically populated – no manual entry needed. This is professional form extension.

truestack.dev

INTEGRATION ARCHITECTURE PATHS

BATCH PATH

- Pull XML Files from SFTP (Periodic Batch Job)
- Import Raw Data to Staging Table
- Map and Transform Data using Data Management Framework (DMF)
- Write Transformed Data to Target Tables & Sales Order Headers

HIGH-VOLUME PERIODIC UPDATES

REAL-TIME PATH

- External Systems Call OData Endpoints (Immediate Data Arrival)
- Validate Schema against EDTs and Field Definitions
- Process Data & Update Inventory Immediately
- Log Changes for Audit Visibility

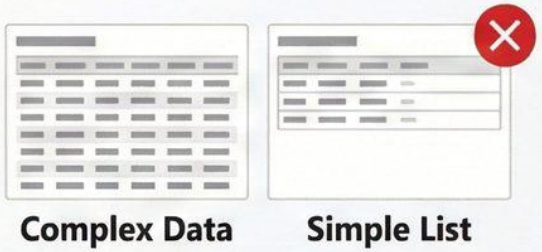
IMMEDIATE UPDATES

Both batch and real-time integration paths are valid and often used together for different data processing needs.

trustack.dev

Four Critical Form Design Mistakes

Mistake 1: Wrong Pattern



Simple List for complex data feels inadequate; user experience suffers.



Match Complexity

Always choose the pattern that matches the complexity.

Mistake 2: Modifying Standard Forms Directly



Editing standard forms causes conflicts with updates; can't upgrade.



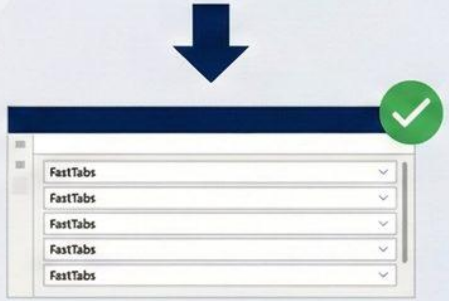
Use form extensions

Use form extensions to ensure safe upgrades.

Mistake 3: Inconsistent Tab Styles



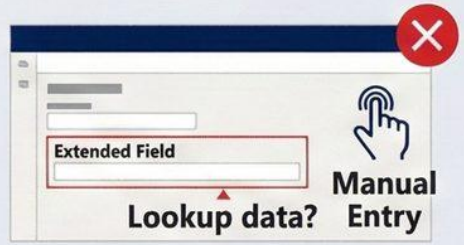
Mixing FastTabs with horizontal tabs confuses users; breaks consistency.



Single Style

Stick to one consistent tab style across forms.

Mistake 4: Extended Fields Not Populated



Adding fields without handlers requires manual data entry.



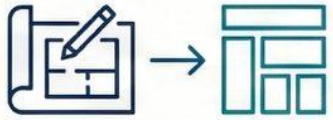
Implement Handlers

Always implement handlers to auto-populate fields.

Avoid these critical form design mistakes to ensure a consistent, maintainable, and user-friendly Dynamics 365 F&O experience.

truestack.dev

Five Best Practices for Form Design in D365 F&O



Practice 1: Choose the Correct Pattern

Really understand the requirement and match it to a pattern. Don't try to force a simple list to do complex work.

Match pattern to requirement.



Practice 2: Use Extensions Only

Never modify standard forms. Extensions keep your code safe and upgradeable.

Safe and upgradeable code.



Practice 3: Populate Your Extended Fields

When you add fields, implement handlers to populate them automatically. Don't make users manually enter data that could be looked up.

Automate with handlers.



Practice 4: Follow FastTab Conventions

If your pattern uses FastTabs, use them consistently. Don't mix tab styles.

Use consistently.



Practice 5: Document Your Pattern Choice

In comments or design documentation, explain why you chose this pattern. Future Future maintainers will understand the design decision.

Explain why.

Discipline in form design makes the system professional and maintainable.

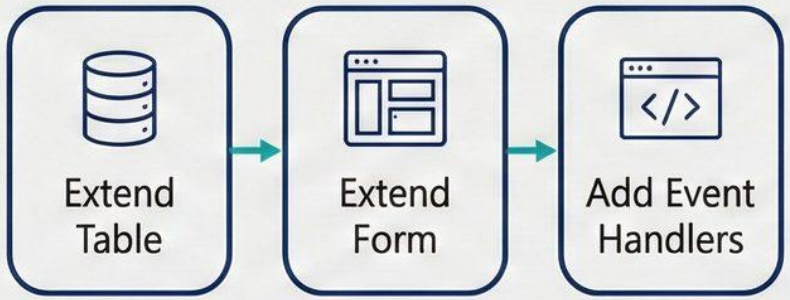
truestack.dev

Topic 5.1 Recap: Form Patterns & Extensibility Basics

Form Patterns: Consistency & Choice

- **Simple List:** Minimal grids
- **List Page:** Large datasets
- **Details Master:** Complex entities
- **Simple List & Details:** Grid + Details Panel
- **Workspace:** Dashboards

Safe Form Extension (Layers)



All three layers work together.

Application & Features

- Apply via Designer or Metadata Properties (Both valid)
- Vertical FastTabs (Collapsible sections in Details Master & Simple List & Details)

Integration Architecture

- **Batch Imports:** SFTP + Data Management
- **Real-Time Updates:** OData

Both paths serve different needs.

Best Practices (Checklist)

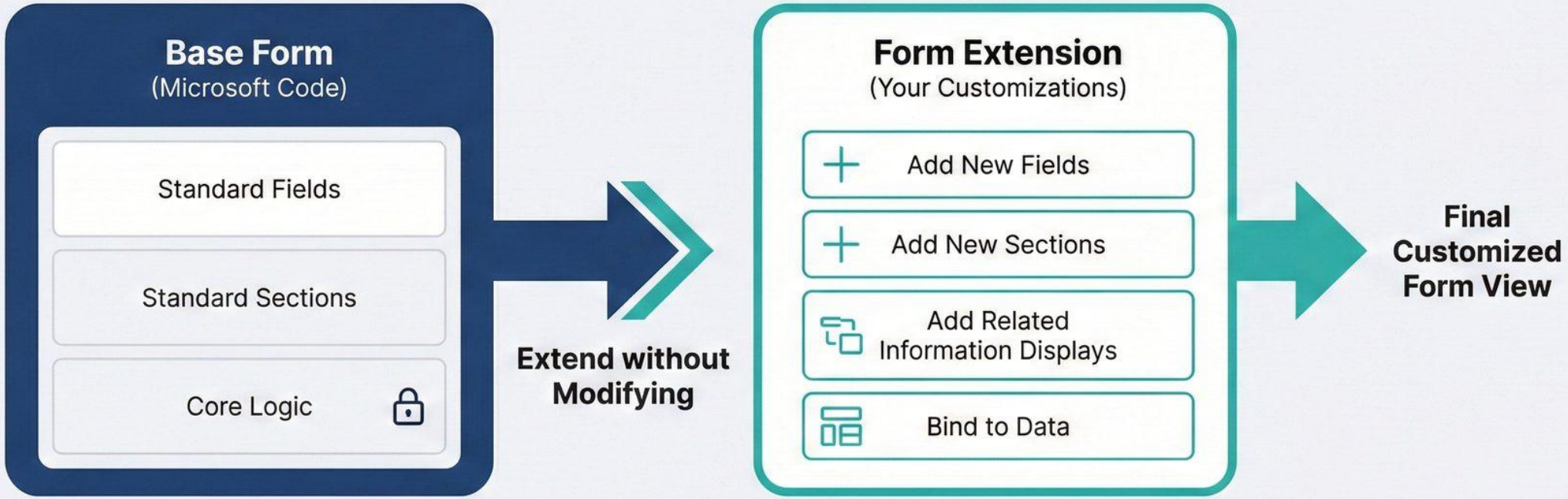
- ✓ Choose right pattern
- ✓ Use extensions only
- ✓ Populate extended fields
- ✓ Follow conventions
- ✓ Document decisions

Next (5.2): Form Extensions and Controls - Customizing layouts and adding controls.

truestack.dev

Form Extensions and Controls

TOPIC 5.2



Key Benefits of Form Extensions

- Customize without touching standard code
- Base form updates safely while extensions continue working.

Master these patterns to customize forms professionally and safely.

truestack.dev

Complete Form Extension Workflow

1. 

   Locate and Right-Click Base Form
   Find the base form in Application Explorer and access the context menu.

2. 

   Select Create Extension
   Choose the 'Create Extension' option from the menu to initiate the process.

3. 

   Extension Object Created
   A new extension object is automatically created and named (e.g., BaseFormName.ModelName.Extension).

4. 

   Add Controls in Design
   Open the design view and add your controls like text boxes, groups, or buttons.

5. 

   Bind Controls to Data
   Set the DataField property to connect controls to table fields from data sources.

6. 

   Build, Compile, and Activate
   Compile the solution. Your extension now merges automatically whenever the form is opened.

✓ **Result: User sees the base form plus customizations. Safe and upgradeable.**

truestack.dev

Extension Naming Conventions in D365 F&O

Strict Naming Rules for Consistency and Recognition

CORRECT APPROACH ✅

Pattern:
BaseFormName.ModelName.Extension

```
Example 1: SalesTable.HR.Extension
Example 2: CustTable.Finance.Extension
```

Clearly identifies the Base Object and the Model.

Consistent pattern for Forms, Tables, and Classes. Immediately recognizable.

INCORRECT APPROACHES ❌

- ❌ SalesTable.Extension (Ambiguous, missing Model name)
- ❌ SalesTableExt (Random naming, non-standard)
- ❌ Sales_Table_HR_Ext (Using underscores instead of dots)

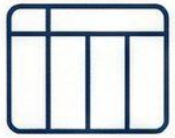
Causes confusion for future developers. Non-professional practice.

Why Naming Matters: Future developers need to recognize extensions at a glance. Consistent naming ensures professionalism.

truestack.dev

Adding New Fields: The Two-Step Extension Process

Synchronizing data and UI through table and form extensions



1. Table Extension

Add new fields using correct Extended Data Types (EDTs).

- `EcoResProductName` for product name
- `InventBatchId` for batch ID

Binds control to data



2. Form Extension

DataField property

Add a new group to the form design (e.g., in the Details section). Then, add text controls for each new field.

*Set the **DataField** property to point to your table extension field.*

This two-step process keeps your data and UI synchronized, allowing users to view and edit the new fields when the form is opened.

truestack.dev

Choosing the Right Control & Properties

Map data types to controls and configure essential properties for usability.

Data Type	Recommended Control
abc String	Text control
123 Whole Number	10 Integer control
12.34 Decimal	99.99 Real control
Date	Date control
Dropdown List	Enum control
Action	Submit Button control

Key Control Properties

Label

Display name for the control shown to users.

HelpText

Briefly explain what the field is used for.

ExtendedDataType

Enforce EDT validation, must match table field's EDT.

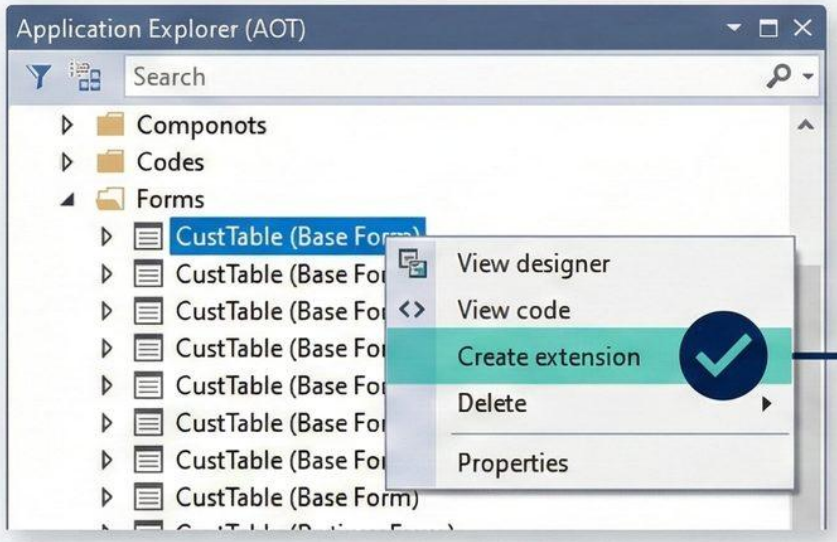
Setting these properties correctly ensures the control works as intended and provides a good user experience.

truestack.dev

Dynamics 365 Finance & Operations: Form Extensions - The Critical Rule

How to correctly extend standard forms for safety and compliance

THE ONLY CORRECT WAY: CREATE EXTENSION FROM BASE FORM



Supported, Safe Method: Creates a true form extension object, preserving upgrade path.

- Always right-click the base form in AOT & select "Create Extension".

System validates the structure of the extension automatically.

INCORRECT METHODS & RISKS

Never add base form directly to project
→ Not an extension; no effect.

Never modify base form directly
→ Overlays base; breaks upgrades.

Never create new form with matching name
→ Duplicate form; not an extension.

RISKS OF BYPASSING VALIDATION:

- Build Fails
- Overlaying (Depracated/Bad)

Takeaway: The only supported, safe method is to create an extension directly from the base form in the AOT. Avoid all other methods.

truestack.dev

Understanding FactBoxes in Dynamics 365 F&O



What are FactBoxes?

Read-only displays showing context-specific information for the currently selected record.



How They Work

Uses a FormPartControl to display a secondary form that queries and displays related data.

Refreshes automatically on new selection.



Key Benefits

Ideal for operational awareness without opening additional forms.
Reduces clicks and improves user productivity.



Powerful and Underutilized

FactBoxes are a powerful tool that is often underutilized in application design.

FactBoxes provide immediate, context-aware insights on list pages, significantly enhancing operational efficiency.

truestack.dev

D365 F&O: Filtering Mechanisms Overview

Formal & Structured Filtering (Primary Mechanisms)

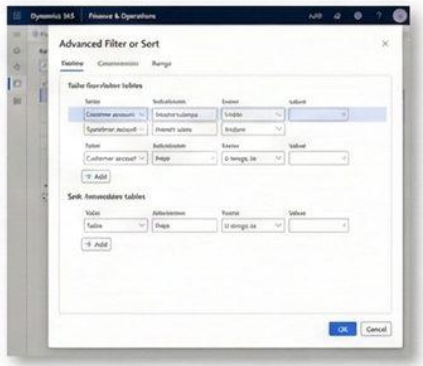
Filter Pane

Structured, multi-field filter. Users select fields, enter criteria, and the system builds a query. Supports complex logic.



Advanced Filter/Sort

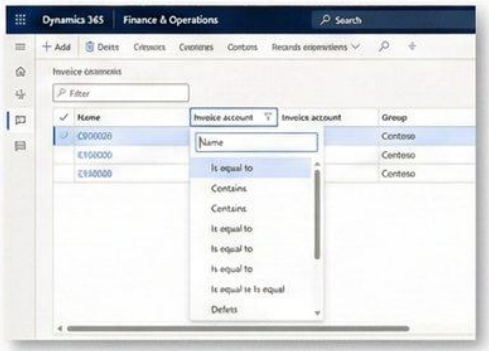
More powerful, allowing construction of complex queries including joins across tables and defining specific ranges. Supports complex filtering logic.



Convenience Filtering

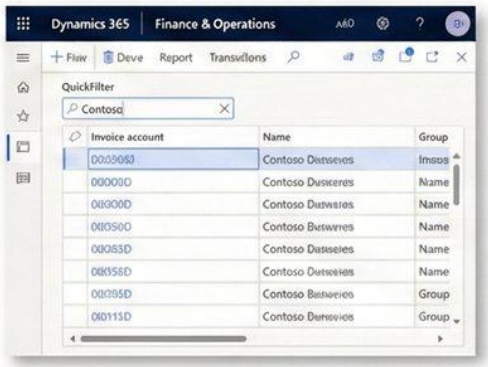
Column Filter

Simpler, in-place filtering. Users click a column header to filter based on that specific column only.



QuickFilter

Search bar located above the grid. Users type text and it performs a 'contains' search across visible fields.



Filter Pane and Advanced Filter/Sort are the primary mechanisms for complex, structured filtering, while Column Filter and QuickFilter serve as convenience tools for simpler, immediate results.

truestack.dev

Display Method Optimization Techniques

Improve performance by minimizing expensive query execution.

Recommended Optimization Techniques (Cache & Hide)

[SysClientCacheDataMethodAttribute(true)] ✓

Caches the method result per record. Prevents redundant query execution when multiple controls call the same method.

```
[SysClientCacheDataMethod(true)]
public display Amount mstAmount()
{
    // Expensive query logic here
}
```

Set Visible = false ✓

Hidden controls do not execute their display methods, eliminating wasted queries for unnecessary data.

Ineffective Approaches (Anti-patterns)

Set Enabled = false ✗

Only makes the control read-only. Does not prevent method execution.

Set Skip = true ✗

Only affects tabbing order. No performance benefit.

Cache results and hide unnecessary controls to effectively optimize display method performance.

truestack.dev

Form Extension Classes: Adding Behavior to Forms

Subscribing to Form Events & Adding Behavior



FormDataSource.init

Runs when the data source initializes. Use it to set up defaults or calculations.



FormDataSource.write

Runs when a record is saved. Use it to perform validations or updates.



FormControl.modified

Runs when a user changes a field. Use it to update related fields or enable/disable buttons.

Working Alongside Form Extensions for Complete Customization



Form Extensions

Add UI controls.

Work alongside



Form Extension Classes

Add behavior.



Result: Complete Customization

Together they provide complete customization.

This approach keeps your code separated from the UI structure.

truestack.dev

Customer Form Extension: Credit Limit & Last Order Date

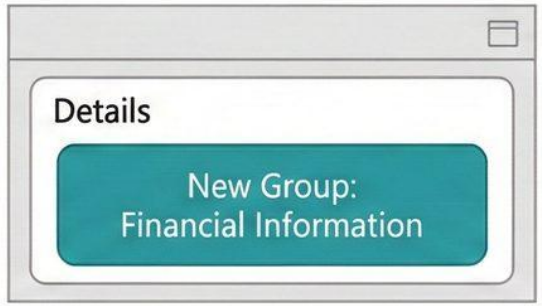
Real-world Scenario: Safe Extension for Rich Data and Validation

1 Table Extension

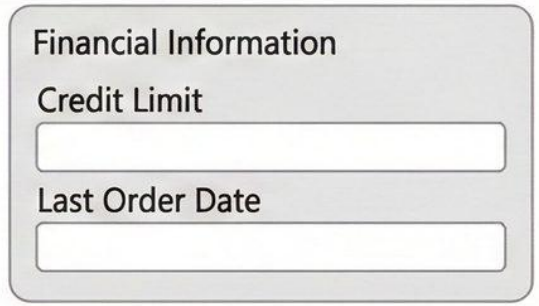
Real Type — Adds “CreditLimit” (Real)

Date Type — Adds “LastOrderDate” (Date)

2 Form Extension (Group)



3 Form Extension (Controls)



4 Form Extension Class

```
class CustomerForm_Extension
{
    [FormDataSourceEventHandler(formDataSourceStr
    (CustTable, CustTable),
    FormDataSourceEventType::Written)]
    public static void CustTable_OnWritten(...)
    {
        // Handler Logic
    }
}
```

5 Validation (onSave)

Handler validates ‘CreditLimit’ is positive

If CreditLimit <= 0, Show Error

6 Auto-population (onLoad)

Sales Tables Query → LastOrderDate

Handler queries sales tables to find and populate ‘LastOrderDate’

Result: Rich, validated form with auto-populated data, achieved safely via extensions.

truestack.dev

FOUR CRITICAL MISTAKES IN D365 F&O FORM CUSTOMIZATION

Mistake 1: Modifying base forms directly

Direct Modification
(e.g., SalesTable)

UPDATE CONFLICT & BREAKAGE
(Cannot upgrade)

Use Extensions
(e.g., SalesTable.Extension)

UPGRADE SAFE
(Preserves changes)

Always use extensions to avoid conflicts.

Mistake 2: Adding base form to project (Overlaying)

SalesTable
(Base Form)

STUCK WITH OLD CODE
(No updates)

SalesTable.Extension

RECEIVES UPDATES
(Stays current)

Create Extension from base form, don't add base.

Mistake 3: Adding controls but not binding them



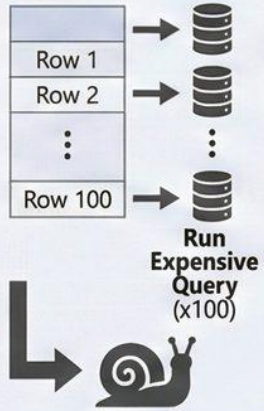
NO DATA APPEARS
(Empty box)



DATA BOUND SUCCESSFULLY

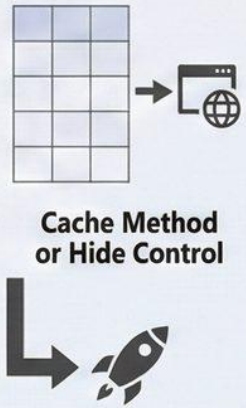
Always bind controls to data fields.

Mistake 4: Display methods executing expensive queries



Run Expensive Query
(x100)

FORM CRAWLS
(Slow performance)



Cache Method or Hide Control

OPTIMIZED PERFORMANCE

Cache expensive methods or hide the control.

Avoiding these four common mistakes ensures upgrade-safe, functional, and performant D365 Finance & Operations forms.

truestack.dev

Microsoft Dynamics 365 Finance & Operations

Five Best Practices for Form Extensions



Practice 1: Create Extension from Base Form

Always use Create Extension from the base form. This is the supported method.

✕ Never try workarounds



Practice 2: Follow Consistent Naming Conventions

Follow naming conventions consistently. BaseForm.ModelName. Extension.

It should be recognizable at a glance.



Practice 3: Bind Controls to Data Sources

Bind all controls to data sources. Set DataField and properties correctly.

✕ Don't add orphaned controls with no data.



Practice 4: Optimize Display Methods

Optimize display methods. Cache expensive methods or hide unnecessary ones.

Performance matters.



Practice 5: Document Your Extensions

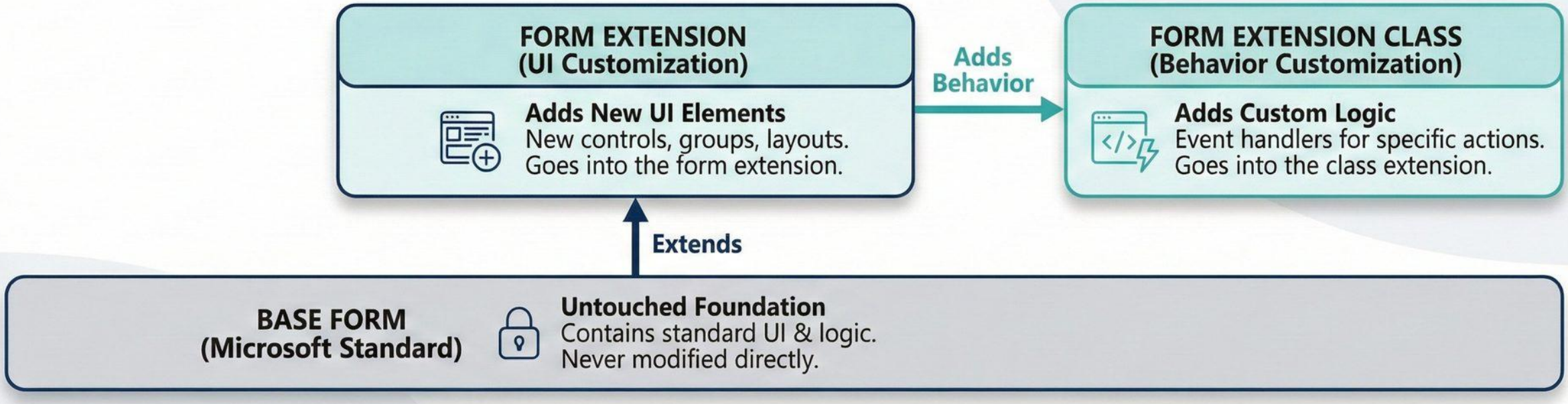
In code comments or design docs, list the controls you added and why.

Future maintainers need to understand your changes.

Discipline makes form extensions professional and maintainable.

truestack.dev

Professional Form Customization: The Three-Layer Architecture



Key Benefits & Principles

- **Clean Separation of Concerns:** Each layer has a specific responsibility.
- **Ensures Upgrade Safety:** Base form stays untouched.
- **Improves Maintainability & Clarity:** Future developers understand the separation.
- All three layers work together but are kept separate.

This layered architecture separates standard code, UI customizations, and custom behavior for a robust and maintainable system.

fridestack.dev

Topic 5.2 Recap: Safe Form Extension & Customization

Form Extension Principles & Naming

Always create extension from base form; never modify base directly.

Naming Convention:
`BaseForm.Model.Extension`

UI & Data Binding

Add controls via form designer, bind using DataField property.

Extend tables & link form controls to table extension fields for data integration.

FactBoxes & Filtering Mechanisms

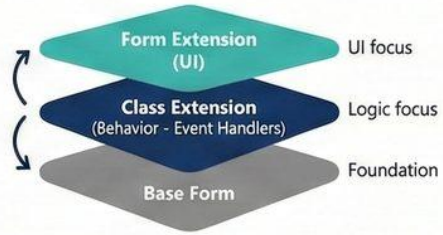
FactBoxes display related info for selected records for operational awareness.

Filtering:

- Filter Pane (Formal)
- Advanced Filter/Sort (Formal)
- Column Filter
- QuickFilter

Performance & Architecture

Optimize display methods with **caching** attribute or by hiding controls.



Three-layer architecture:
All separate, all safe, all professional.

Next Topic: Topic 5.3 - Menus, Workspaces & Tiles (Navigation Architecture)

truestack.dev

Topic 5.3: Menus, Workspaces, and Tiles

Navigation is critical to user experience. Organized menus and dashboards make systems intuitive.

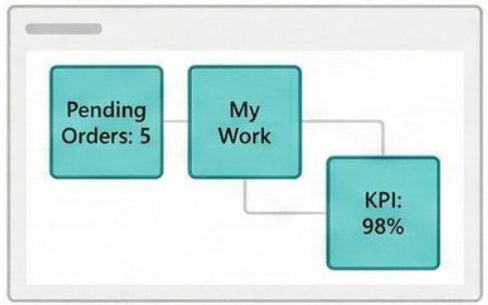
Menu Item Types

Display: For interactive **forms**

Action: For background **processes**

Output: For printable **reports**

Operational Dashboards



Build at-a-glance views using tiles and workspace elements.

Personalize Experience

Empower users to customize their layout and navigation.

Master these patterns to create professional, intuitive navigation that guides users through your system.

truestack.dev

D365 F&O Menu Item Types

Display



Opens interactive forms. Users see data, can edit, can navigate.

Specific purpose: UI interaction.

Action



Runs a runnable class. Like a batch job or processing routine. Class runs, processing happens.

Specific purpose: Processing.

Output



Runs an SSRS report. Report displays, users can print or export.

Specific purpose: Reports.

Choosing the right type is critical. Wrong type and the menu item just won't work.

truestack.dev

Simple Decision Process for Menu Items



This decision tree is clear and deterministic – no ambiguity. Each question points to exactly one answer. Use this every time you create a menu item.

truestack.dev

Display Menu Items: Opening Interactive Forms

Facilitating user interaction with application data through forms.

1. Menu Item Configuration

Name:
CustomerMaintenance

Menu Item Type:
Display

ObjectType:
Form

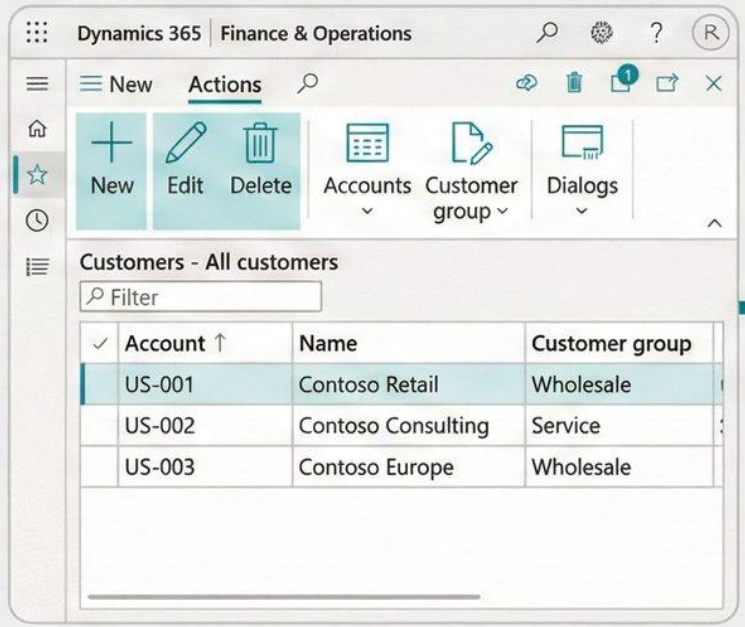
Object:
CustTable

Defines the link to a specific form (e.g., CustTable).

click

User Clicks Menu Item

2. Opens Interactive Form (e.g., CustTable)



Users can view the list, add new records, edit existing ones, and delete them.

3. Scope & Significance

- Works for List Pages
- Works for Detail Forms
- Works for Dialogs
- Any form requiring user interaction.

This is the most common menu item type in D365 F&O.

truestack.dev

Action Menu Items: Executing Batch Jobs & Automation

Triggers runnable classes for non-interactive processes like batch jobs and integrations.

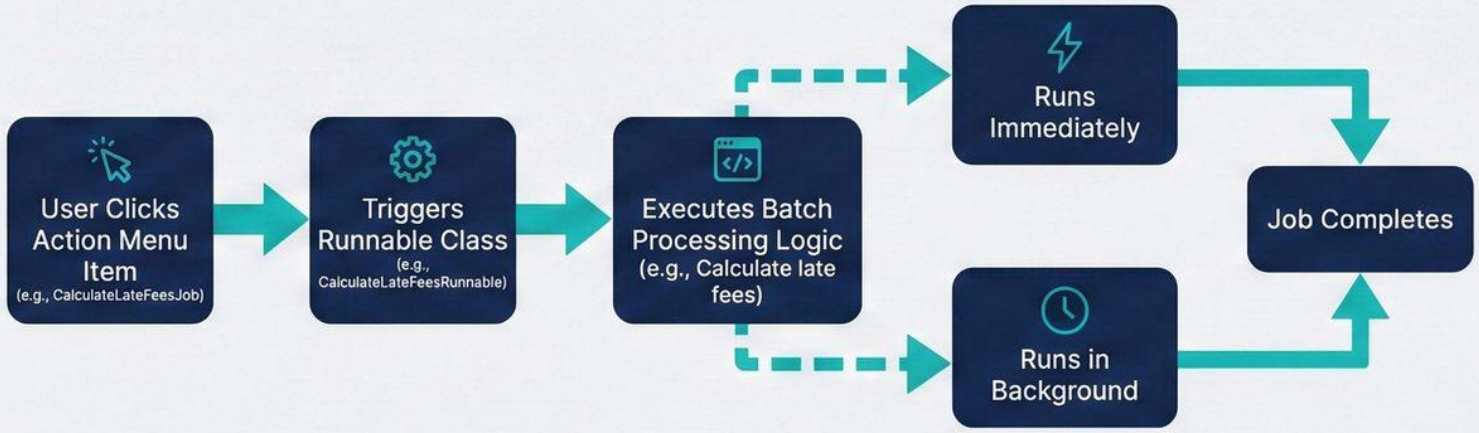
Purpose & Key Properties

- Used for batch jobs, automation, integration syncs
- No user interaction required
- Second most common menu item type

Example Configuration:

Menu Item Name:	CalculateLateFeesJob
ObjectType:	Class
Object:	CalculateLateFeesRunnable

Execution Flow



Action menu items are the mechanism for triggering non-interactive, automated processes and batch jobs directly from the menu.

truestack.dev

Output Menu Items for Reports



INCORRECT PATTERN: Do not use Display menu items to run reports. Display is for forms.

CORRECT PATTERN: Output is the required type for report execution.

Output menu items are the only correct type for executing SSRS reports.

truestack.dev

Adding a Report Button to a Form: The Complete Pattern

1



Create Output Menu Item

Define for the targeted report in AOT.

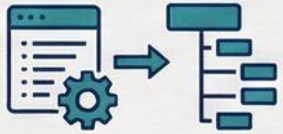
2



Add Menu Item Button Control

Crucial: Use **Menu Item Button**, NOT a command button.

3



Set MenuItemName Property

Configure control to point to your Output menu item.

4



Report Runs on Click

Button appears on form; user click executes report.

Key Takeaway

Clean & Maintainable: Menu item lives in AOT for reuse. Command buttons cannot directly reference reports.

Always use Menu Item Button when dealing with reports.

truestack.dev

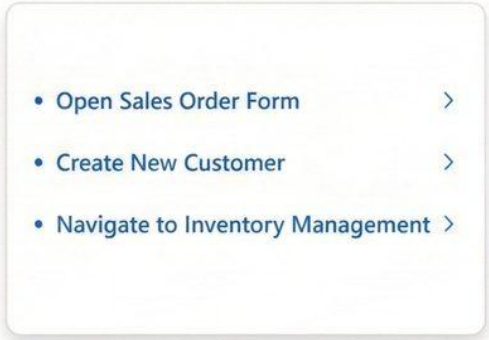
Workspace Elements: Types and Purposes

1. Tiles: Counts & KPIs



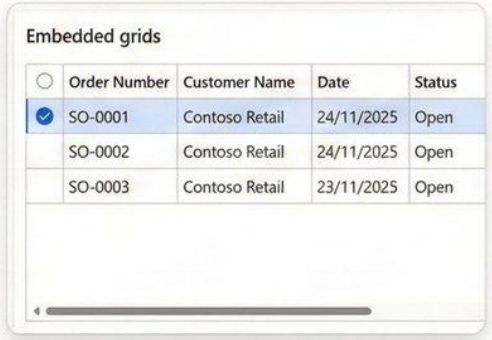
Show counts or Key Performance Indicators (KPIs). Users click to filter and view the specific records, like "Pending Confirmation Orders".

2. Links: Simple Navigation



Menu-style navigation elements. Users click to open a specific form or initiate an action.

3. Lists: Embedded Grids



Show an embedded grid inline. Users click a row to open the corresponding record's detailed form.

4. Power BI Visuals: Analytical Charts



Embeds interactive analytical charts and dashboards. Users can interact with visuals to drill down into the data for deeper insights.

Each workspace element serves a distinct purpose; choose based on the specific information and interaction you need to surface.

truestack.dev

Displaying a KPI in a Workspace

1 Create KPI measure in AOT



Define "PendingOrderCount" to calculate pending orders count.

2 Create Tile in Workspace



Add a new Tile element to the workspace design.

3 Bind KPI to Tile



Set the tile's properties to reference the "PendingOrderCount" KPI.

4 Tile Displays KPI Value



The tile dynamically shows the calculated KPI value.

5 User Clicks Tile



Clicking applies a filter and opens the underlying list.

This standard pattern uses KPI measures as the data source and tiles for display in operational dashboards.

truestack.dev

Workspace Tiles: Operational Awareness at a Glance

Visualizing Record Counts and Enabling Quick Drill-Down

1. Define the Query (Behind the Scenes)

Query Logic:
"Pending Confirmation Orders"

Status equals **pending**

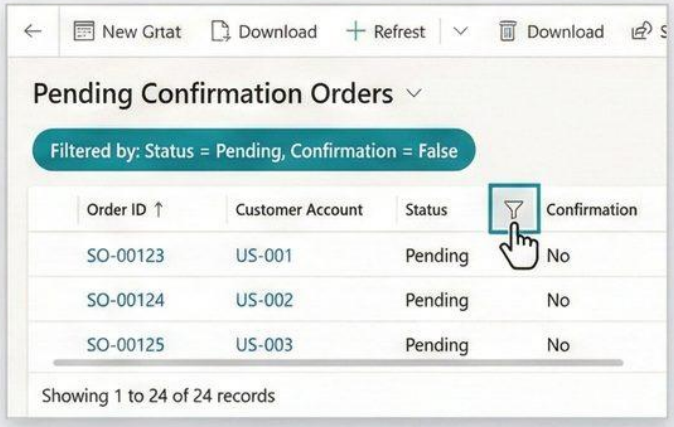
Confirmation equals **false**

2. The Tile Visualizes the Count



Calculating in real-time, tiles show exactly how many orders need attention at a glance.

3. Click to Drill Down (The Result)



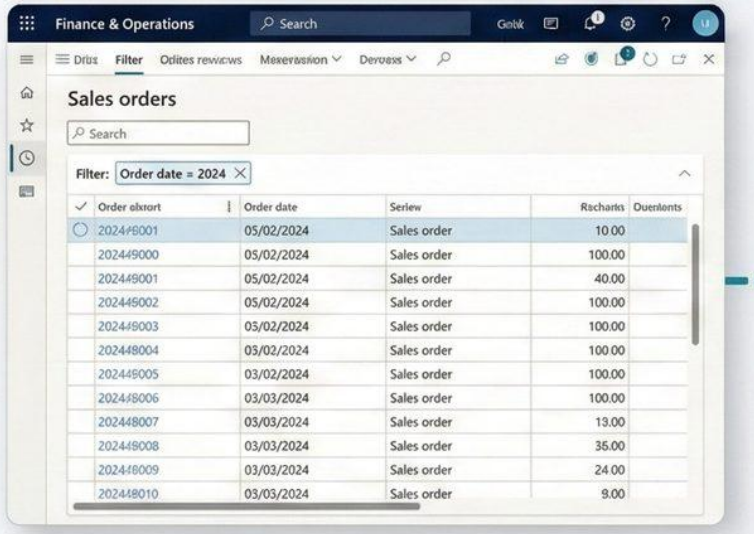
One click applies the filter, displaying the specific 24 pending orders for immediate action.

Tiles act as interactive KPIs, providing instant operational insight and streamlining workflow efficiency.

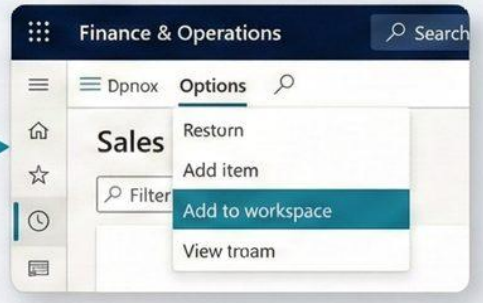
truestack.dev

User Personalization: Adding Filtered Views to Workspaces

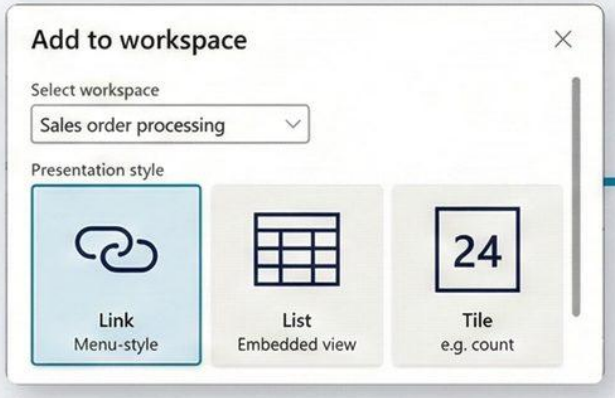
1 Filter Form (e.g., 2024 Orders)



2 Options > Add to workspace



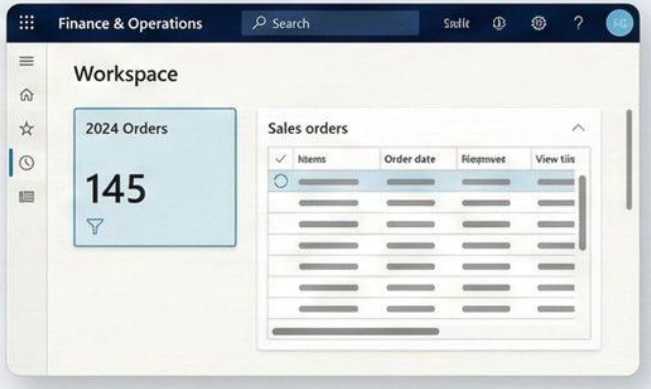
3 Choose Workspace & Style



4 Configure & Add



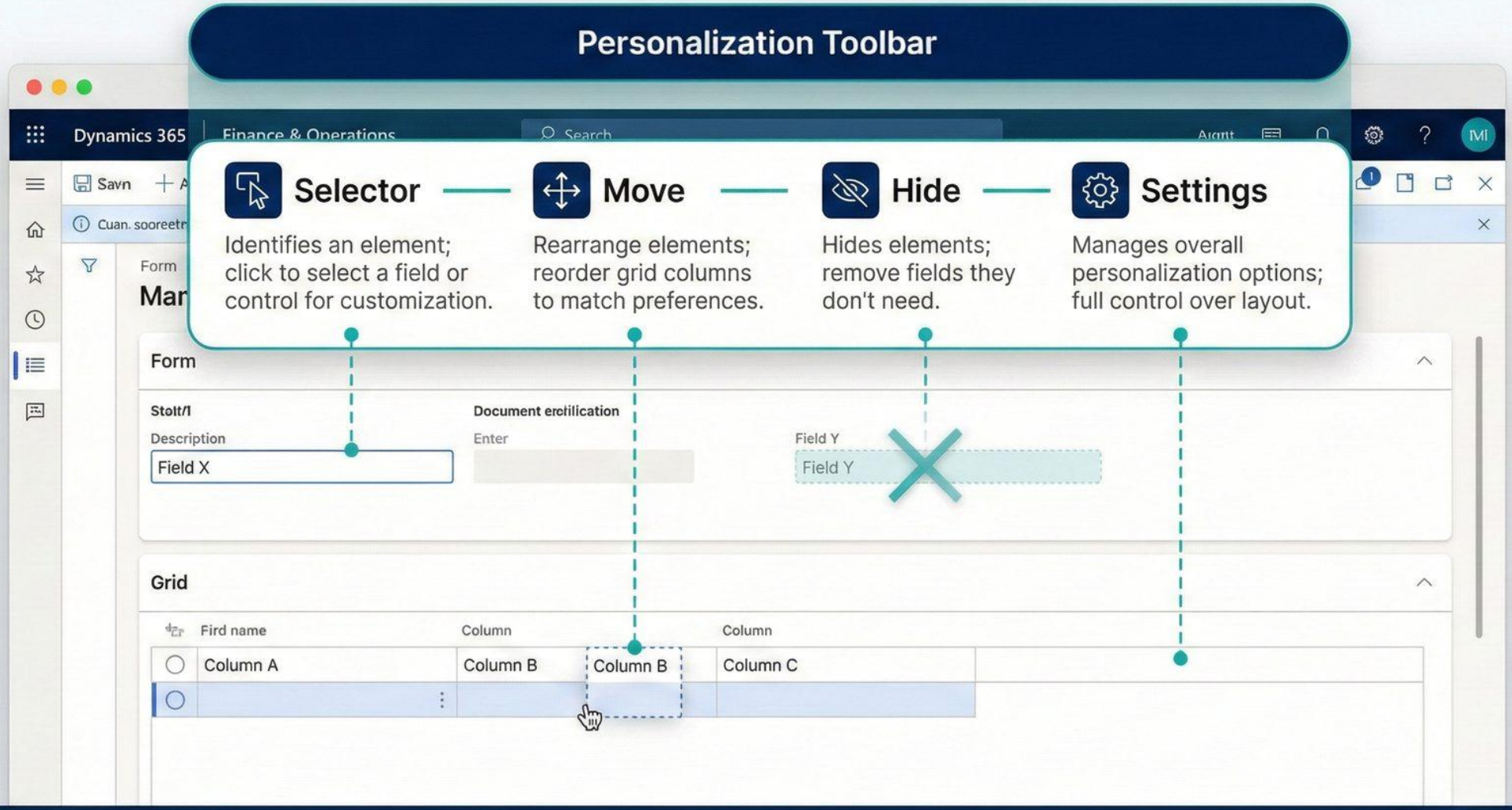
5 Personalized View Added to Workspace



User-driven personalization: tailor dashboards to specific needs without development.

truestack.dev

User Interface Personalization: Empowering the User



This feature represents professional UX design, giving users full control over their layout to customize their workspace exactly as they want.

truestack.dev

Sales Operations Workspace

Pending Confirmation Orders

24
orders pending

> Click to drill down

Overdue Shipments

8
shipments overdue

> Click to drill down

Revenue KPI (This Month)

$450,000
current month revenue

Power BI Chart - Monthly Trends



Create New

Create new orders

Operational awareness at a glance, with details just one click away – a cohesive professional dashboard design.

truestack.dev

Critical Mistakes in D365 F&O Navigation & Reporting

Mistake One: Display for Reports

INCORRECT: Using a Display menu item for a report. It won't work.

↓ Display opens forms, not reports.

CORRECT: Use an Output menu item.

Mistake Two: Action for Forms

INCORRECT: Using an Action menu item to open a form. Again, won't work.

↓ Action runs classes, not forms.

CORRECT: Use a Display menu item.

Mistake Three: Linking Reports to Command Buttons

INCORRECT: Trying to link a report directly to a command button. Not supported.

↓ Command buttons don’t have report properties.

CORRECT: Use a Menu Item Button referencing an Output menu item.

Mistake Four: Modifying Standard Menus

INCORRECT: Modifying standard menus directly.

↓ You change a Microsoft menu, and then when Microsoft updates it, you’ve got a conflict. You can’t upgrade.

CORRECT: Use menu extensions instead.

⚠ **CONSEQUENCE: These mistakes will break your navigation and prevent upgrades.**

truestack.dev

Dynamics 365 F&O Navigation: 5 Best Practices

Practice One: Display Menu Items



Use Display menu items for opening forms.

Benefit: Standard approach for UI interaction.

Practice Two: Action Menu Items



Use Action menu items for batch processing and automation.

Benefit: Handles system processing effectively.

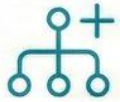
Practice Three: Output Menu Items



Use Output menu items for running reports. Never mix with forms.

Benefit: Keeps reporting execution separate.

Practice Four: Menu Extensions



Extend standard menus using menu extensions. Avoid direct modification.

Ensures code safety and upgradeability.

Practice Five: Workspace Tiles



Use tiles to make workspaces operational for key tasks.

Shows important info with drill-down.

It disciplined navigation design makes your systems professional and intuitive.

truestack.dev

Recap Topic 5.3: Menu Items & Workspaces

Menu Item Types

Display
Opens forms (most common). Use for UI interaction.

Action
Runs batch classes for processing.

Output
Runs SSRS reports.

⚠️ *Critical: Choosing the wrong type breaks the menu item.*

Workspace Elements

Tiles
Show KPIs & counts. Actionable drill-down.

Links
Menu-style navigation.

Lists
Show embedded grids.

Power BI visuals
Show analytics.

Tiles are the best option for actionable insights.

Personalization & Best Practices

Personalization (User-driven, No Dev)

Apply filters → Click Options → Add to Workspace

Best Practices

- ✓ Display for forms, Action for processing, Output for reports
- ✓ Extend menus instead of modifying
- ✓ Personalize with tiles

→ **Next Topic: 6.1 Data Entities and DMF**

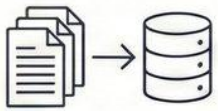
Moving bulk data & importing thousands of records efficiently (Data Management Framework).

truestack.dev

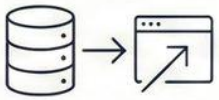
Topic 6.1: Data Entities and Data Management Framework (DMF)

Mastering Bulk Data Operations and Integrations

The Challenge: Bulk Data & Integration



Import thousands of vendor records

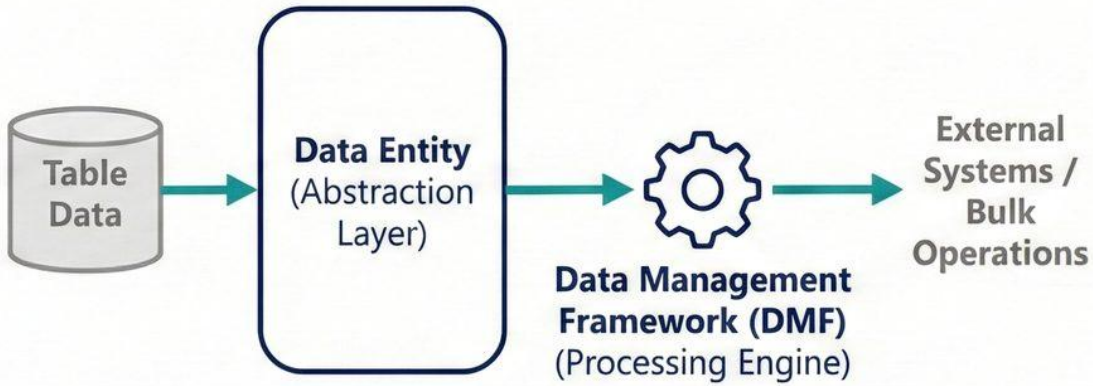


Export sales data to external systems



Integrate with other applications

The Solution: Data Entities & DMF

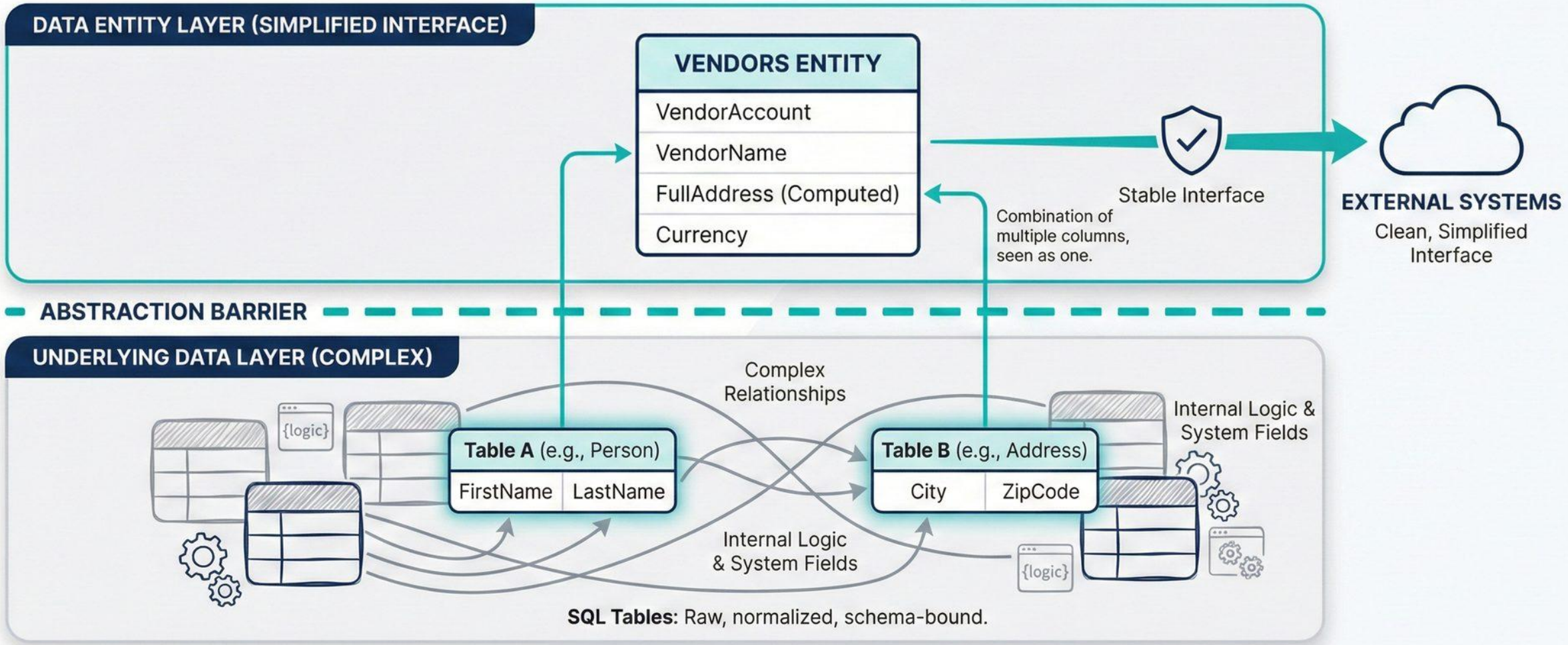


Exposes table data for integration; processes data safely and efficiently.

Master these patterns to build professional data integrations that move data safely and efficiently.

truestack.dev

Data Entities: The Abstraction Layer



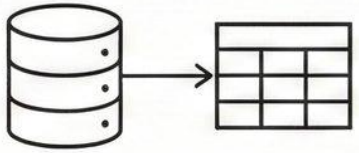
Data entities abstract underlying complexity, providing a stable interface for external systems that remains unaffected by schema changes.

truestack.dev

Data Entity Field Types

Understanding Mapped, Computed, Virtual, and Unmapped Fields

1. Mapped fields

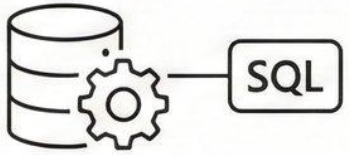


Directly from table columns (e.g., vendor account, name).

- Stored in the database
- Most common type

Straightforward & Persistent

2. Computed fields

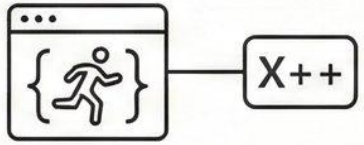


SQL expressions calculated by the database (e.g., concatenated address).

- Efficient calculation by SQL Server

High Performance Database Logic

3. Virtual fields

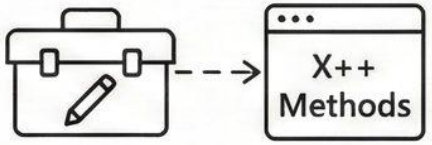


X++ code evaluated at runtime (e.g., total invoices).

- Not stored, calculated on access
- Slower but more flexible

Dynamic & On-Demand

4. Unmapped fields



Custom fields populated using X++ methods for specific logic.

- Use when custom logic doesn't fit others

Highly Customized Solutions

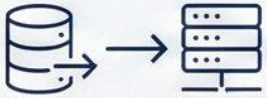
Decision Factors: Choose based on whether the data is stored, where the calculation happens, and what your performance needs are.

truestack.dev

Computed vs. Virtual Fields: Performance and Logic Distinction

Computed Fields

Mechanism



SQL Expression → SQL Server Execution

Characteristics & Best Use

- ✓ Very Fast Performance (Database-level optimization)
- ✓ Perfect for Concatenations
- ✓ Ideal for Aggregations
- ✓ Efficient for Simple Calculations

BEST FOR: Bulk Operations

Virtual Fields

Mechanism



X++ Code → AOS Execution (Runtime)

Characteristics & Best Use

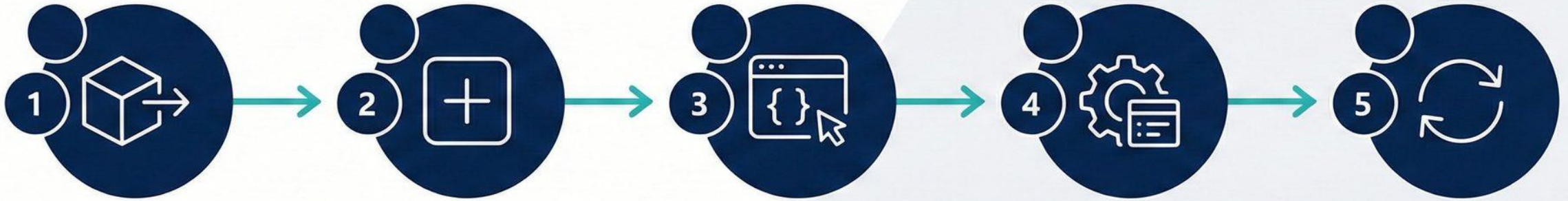
- ✓ More Flexible (Write any logic)
- ✓ Supports Complex Logic
- ✓ Slower Performance (No database optimization) ✗
- ✓ Evaluated per row at runtime ✗

BEST FOR: Complex Logic

Understanding this distinction between SQL-based Computed Fields and X++ Virtual Fields is critical for optimizing system performance.

truestack.dev

Creating Computed Output for Integration



1 Extend Data Entity

Create an extension for the standard data entity.

2 Add Unmapped Field

Add a new string field, e.g., `FullAddress`.

3 Create X++ Method

Write a method to concatenate source fields (street, city, state, zip).

4 Set Data Method

Link the unmapped field's 'Data Method' property to your X++ method.

5 Regenerate Mappings

Refresh entity mappings in Data Management Framework (DMF).

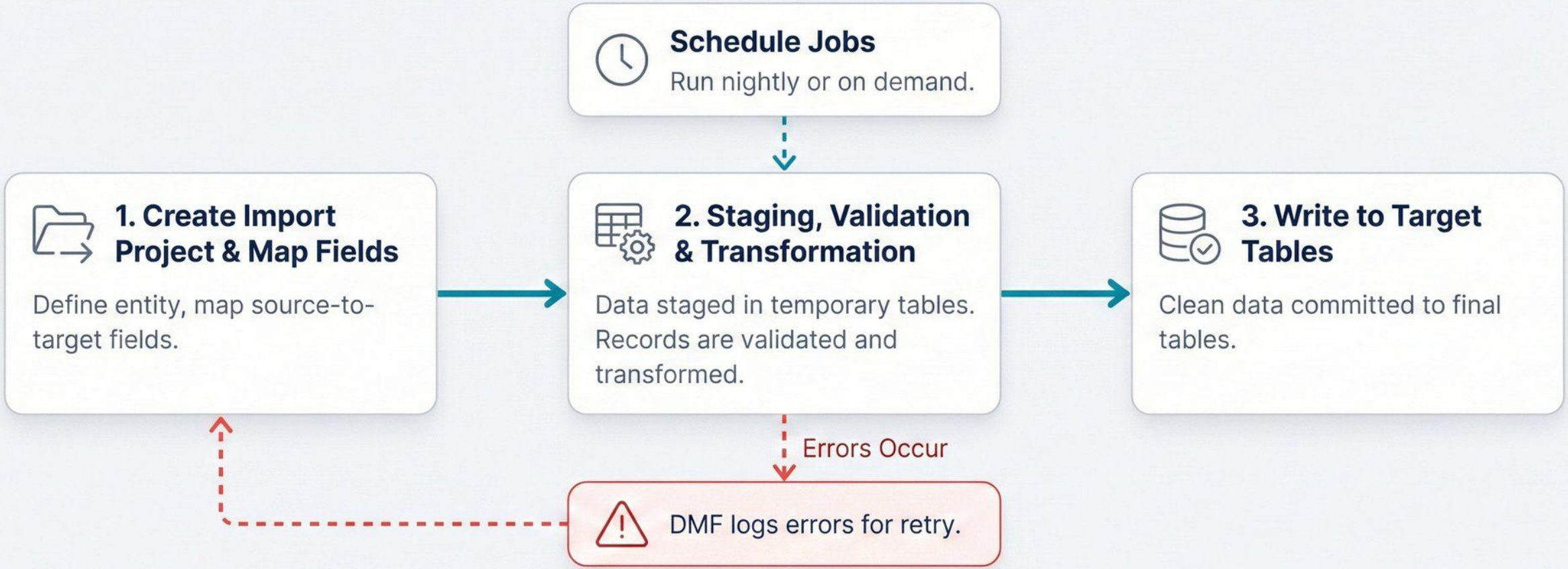
Integration Outcome

External systems can import or export the computed `FullAddress` without knowledge of underlying table structures, encapsulating complexity for clean integration.

truestack.dev

Data Management Framework (DMF) Orchestration

Bulk Data Operations Workflow



Enterprise Data Movement

Reliable, traceable, and production-ready.

truestack.dev

Complete Import Workflow in Data Management Framework (DMF)

1. 

Create Import Project

Give it a name like "VendorImport" in the DMF workspace.

2. 

Select Target Entity

Choose the entity for import, such as “VendVendorV2Entity”.

3. 

Add Source Data

Upload your source file, typically an Excel file with records.

4. 

Generate Field Mapping

DMF automatically maps source columns to entity fields.

5. 

Review and Adjust

Check mappings, rename columns, or add transformations if needed.

6. 

Click Import

DMF stages the data and writes it to the target environment.

7. 

Monitor Progress

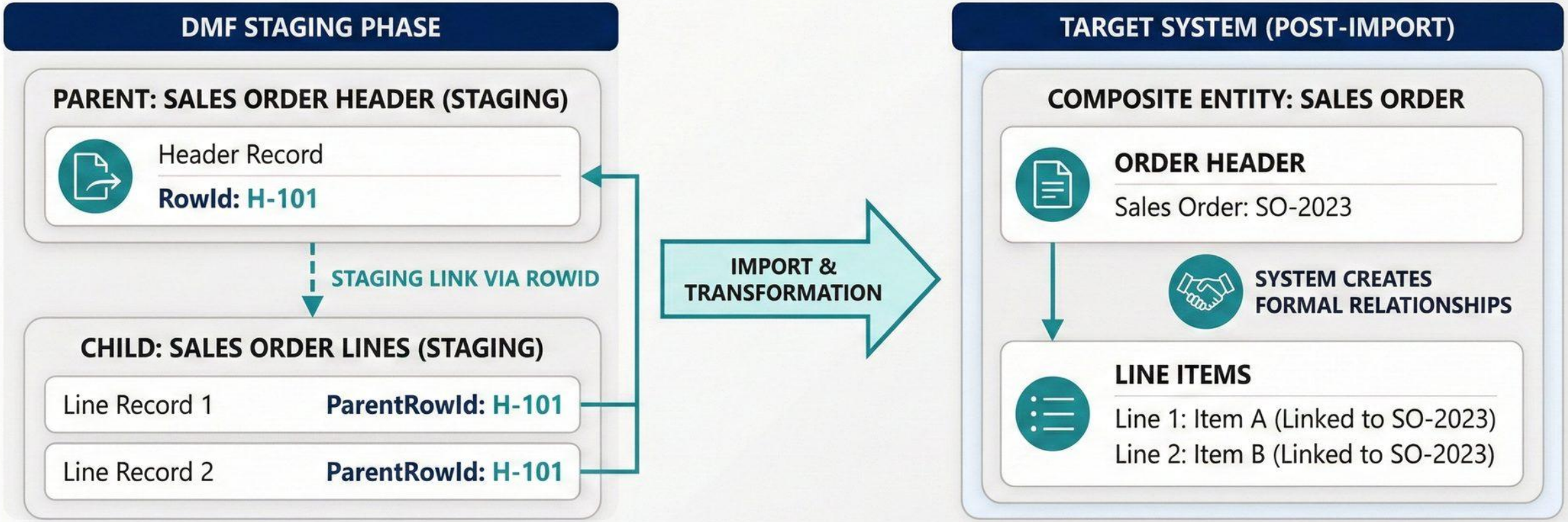
Track import status, errors, and processed record counts.

Outcome: Systematic and reliable data import into the target system.

truestack.dev

COMPOSITE ENTITIES: MANAGING PARENT-CHILD DATA IMPORTS

Example: Sales Orders (Header & Lines)



KEY BENEFITS

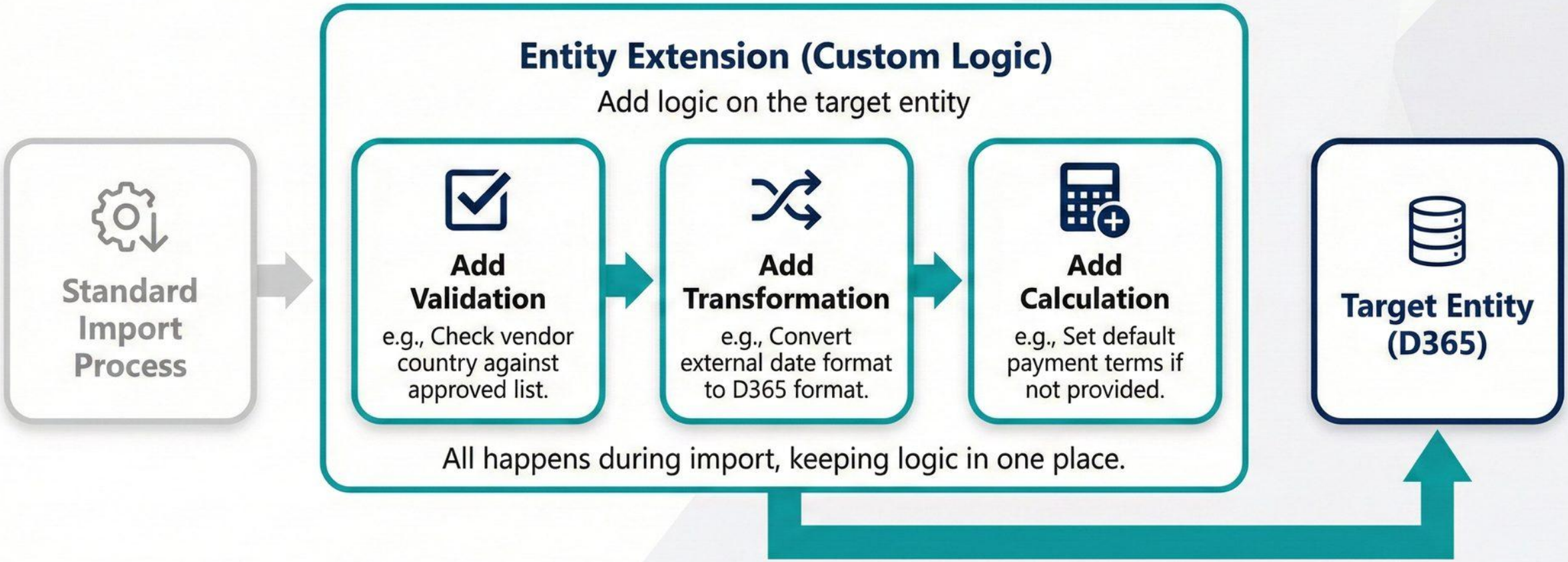
- Single Import Job Handles Both Entities
- Efficiently Manages Complex Data Structures

Composite entities streamline data migration by leveraging staging-level links (RowId/ParentRowId) to establish formal relationships in the target system during import.

truestack.dev

Customizing Imports with Entity Extensions

Modify import behavior without changing standard entities.



Key Benefit: Extensions preserve upgrade safety by avoiding modification of the standard entity.

truestack.dev

Change Tracking for Incremental Updates

Efficiently synchronizing only changed data

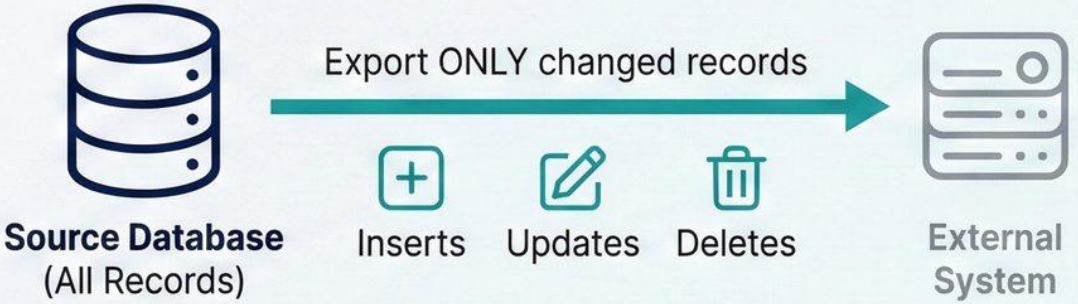
Traditional: Full Data Refresh



- ✗ Logs and exports entire dataset.
- ✗ Inefficient for large datasets.
- ✗ High bandwidth & processing time consume.

Result: Full Refreshes

With Change Tracking: Incremental Update



- ✓ Logs only Inserts, Updates, and Deletes.
- ✓ Returns records changed since last extraction.
- ✓ Super efficient for large datasets.
- ✓ Saves bandwidth & processing time.

Result: Incremental Updates

Configuration Options on Entity

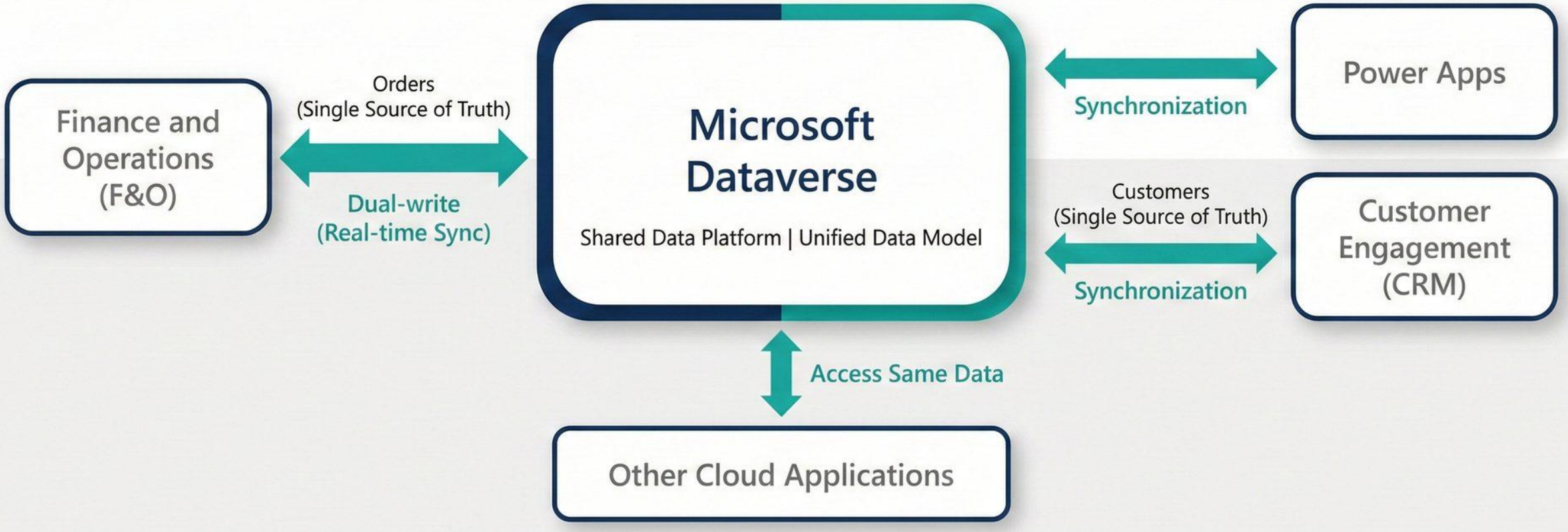
- Primary table only
- Entire entity
- Custom query

Bottom takeaway: Change tracking significantly improves efficiency by enabling systems to fetch only delta changes instead of full datasets.

truestack.dev

Dataverse: Microsoft's Shared Data Platform

Enterprise Data Integration & Unified Data Model



Dataverse creates a single source of truth across the organization, enabling enterprise data integration done right

truestack.dev

DMF Export Formats & Batch Data API

Supported Export Formats

Excel

Great for user analysis and low-volume exports.

CSV

Comma-separated integration with legacy systems.

XML

Structured integration with web services.

Package

Includes a manifest file defining the contents.

Choose the format that matches your target system.

Batch Data API



Handles file-based asynchronous integrations.

Automated Nightly Jobs

Really useful for automated nightly jobs.

truestack.dev

Database Synchronization Options After Table Extensions

Option One: Full System Sync



Go to the Dynamics 365 menu and select ‘Synchronize database’. Safest but slower method.

Option Two: Automatic Sync on Build



Set the project property ‘Synchronize database on build’ to ‘true’. Automatic sync during each build; good for development.

Option Three: Project-Only Sync



Right-click the project and select ‘Synchronize’. Faster for large solutions; synchronizes only items within the project.

Why Synchronize?

Database synchronization creates the SQL columns for your new fields. Without the sync, the fields exist in metadata but not in the actual database.

truestack.dev

Four Critical Dynamics 365 F&O Development Mistakes

Avoid these common pitfalls to ensure upgradeability, performance, and data integrity.

Mistake One: Overlaying Data Entities

INCORRECT APPROACH

Modifying standard entity directly.

Consequence: Conflicts during updates; cannot upgrade.

CORRECT APPROACH

Use entity extensions instead.

Benefit: Preserves customizations; enables smooth upgrades.

Mistake Two: Virtual Fields in Bulk Import

INCORRECT APPROACH

Using virtual fields in bulk import operations.

Consequence: X++ runs per row (e.g., 10,000 executions); severely severely impacts performance.

CORRECT APPROACH

Use computed fields for bulk operations.

Benefit: SQL-based execution; significantly faster bulk data handling.

Mistake Three: Not Generating Field Mapping

INCORRECT APPROACH

Importing data without explicit mapping.

Consequence: Columns don't map correctly; data lands in wrong fields.

CORRECT APPROACH

Generate mapping automatically in Data Management.

Benefit: Ensures accurate source-to-target data alignment.

Mistake Four: Skipping Database Synchronization

INCORRECT APPROACH

Not synchronizing database after extending tables.

Consequence: Field exists in metadata but not SQL; data cannot persist.

CORRECT APPROACH

Always synchronize with the database after schema changes.

Benefit: Updates SQL schema; ensures data persistence and integrity.

Adopt these best practices to ensure upgradeability, performance, and data integrity in your Dynamics 365 Finance & Operations implementations.

truestack.dev

Five Data Integration Best Practices

1



Data Entities for Integrations

Use data entities as the supported, primary layer for all integration scenarios.

2



Extend Entities via Extensions

Never modify standard entities directly; always use extensions for customization.

3

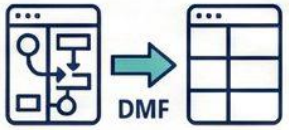


Use Computed Fields for Performance

Computed | Virtual

Leverage computed fields for bulk import efficiency. Reserve virtual fields for complex logic only.

4



Generate Mappings Automatically in DMF

Let the system handle field alignment automatically within the Data Management Framework.

5



Synchronize Database after Schema Extensions

Ensure metadata and SQL are always in sync by running database synchronization after any schema changes.

Disciplined data integration practices ensure reliable, performant, and maintainable systems.

truestack.dev

Topic 6.1 Recap: Data Entities & Bulk Operations

Key Concepts and Takeaways from the Module

Data Entities



Abstraction layer for integration. External systems see clean interface, hiding table complexity.

Four Field Types



Mapped (from columns), Computed (via SQL), Virtual (via X++), Unmapped (via custom code).

DMF (Data Management Framework)



Orchestrates bulk import/export. Create projects, map, validate, and transform.

Composite Entities



Handle parent-child relationships like sales orders with line items.

Entity Extensions



Add validation and transformation without modifying standard entities.

Change Tracking



Enables incremental extraction by identifying only changed records.

Dataverse



Shared platform synchronizing data across F&O, Power Apps, and CRM.

Database Synchronization



Mandatory after schema extensions; metadata and SQL must match.

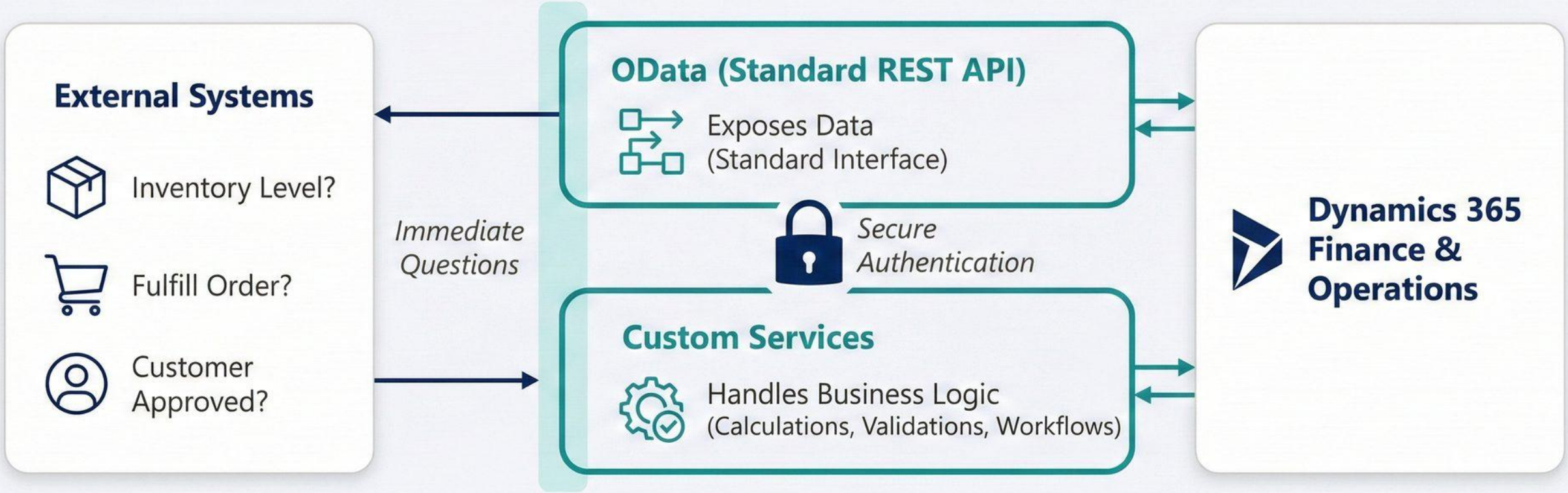
Next Up: Topic 6.2

→ **Services and APIs:** Building Real-time Integration Endpoints to expose data and logic programmatically.

truestack.dev

Topic 6.2: Services and APIs

Real-time Integration for Immediate Answers



Master these patterns to build **real-time integrations** connecting D365 to the entire ecosystem.

truestack.dev

D365 F&O Integration Technologies

Overview of Primary Patterns

OData v4

REST-based access to data entities. Real-time data querying without custom development.

- Query vendors, customers, orders
- Broad system-to-system communication

Real-time Access

Custom Services

X++ classes exposed as web services. Executes business logic and validations.

- Calculate tax, check inventory
- Validate approvals
- Exposed via REST/SOAP endpoints

Real-time Logic

Excel Add-in

User-driven data editing in Excel with D365 sync. Convenient for ad hoc tasks.

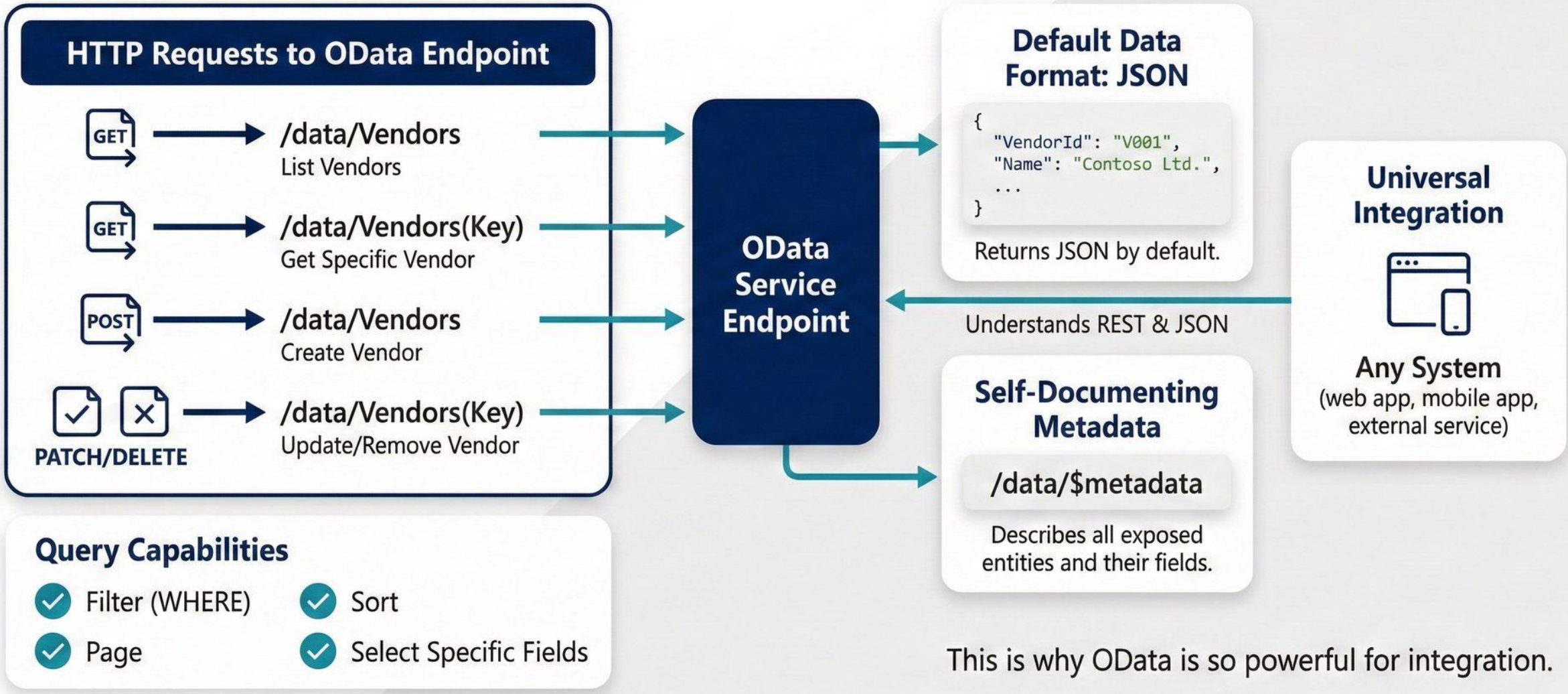
- Functional users data refresh/edit
- Not intended for programmatic automation

Not for Automation

Choose the right technology based on your need for data access, business logic, or user convenience.

truestack.dev

OData: REST-Based Data Access



This is why OData is so powerful for integration.

truestack.dev

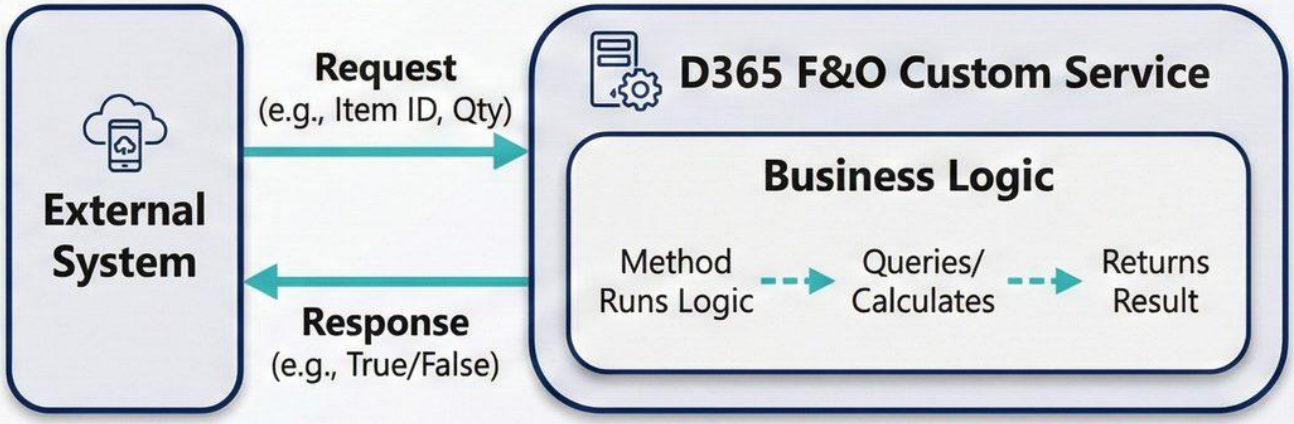
Building Custom Services in Dynamics 365 Finance & Operations

Expose complex business logic through accessible web endpoints.

1. Build the Service (X++)

1. **Create X++ Class**
   Decorated with [DataContract].
2. **Add Public Methods**
   Define methods for each specific operation.
3. **Decorate Methods**
   Use [SysEntryPointAttribute] to make callable.
4. **Register Service**
   Deploy the service group for external access.

2. Execution Flow & Example



Example: CheckInventory Operation

Input: `temID`, `Quantity` → **Process:** Queries inventory, calculates available-to-promise. → **Output:** Returns `true` or `false`.

Takeaway: Custom services are powerful for executing complex business rules that cannot be expressed through standard OData queries.

truestack.dev

OData vs. Custom Services: A Key Distinction

Choosing the the Right Tool for Data Access and Business Logic

OData (Data Access)

Straightforward CRUD operations

- Query vendors, get their records
- Update a field

Best for data access

Custom Services (Business Logic)

Complex logic, calculations, and external lookups

- Calculate tax on an order total
- Validate credit limits, reserve inventory
- Check approval rules

Best for complex logic

Key Takeaway: Most solutions actually use both for a complete solution.

truestack.dev

Consuming OData from C# .NET

A Step-by-Step Guide to Using Strongly-Typed Proxy Classes

1



Install OData v4 Code Generator

First, install the OData v4 code generator.

2



Configure MetadataDocumentUri

Second, configure it with the MetadataDocumentUri pointing to your environment.

3



Run the Generator

Third, run the generator. It reads the metadata and creates C# classes representing each data entity.

4



Use Classes in Your Code

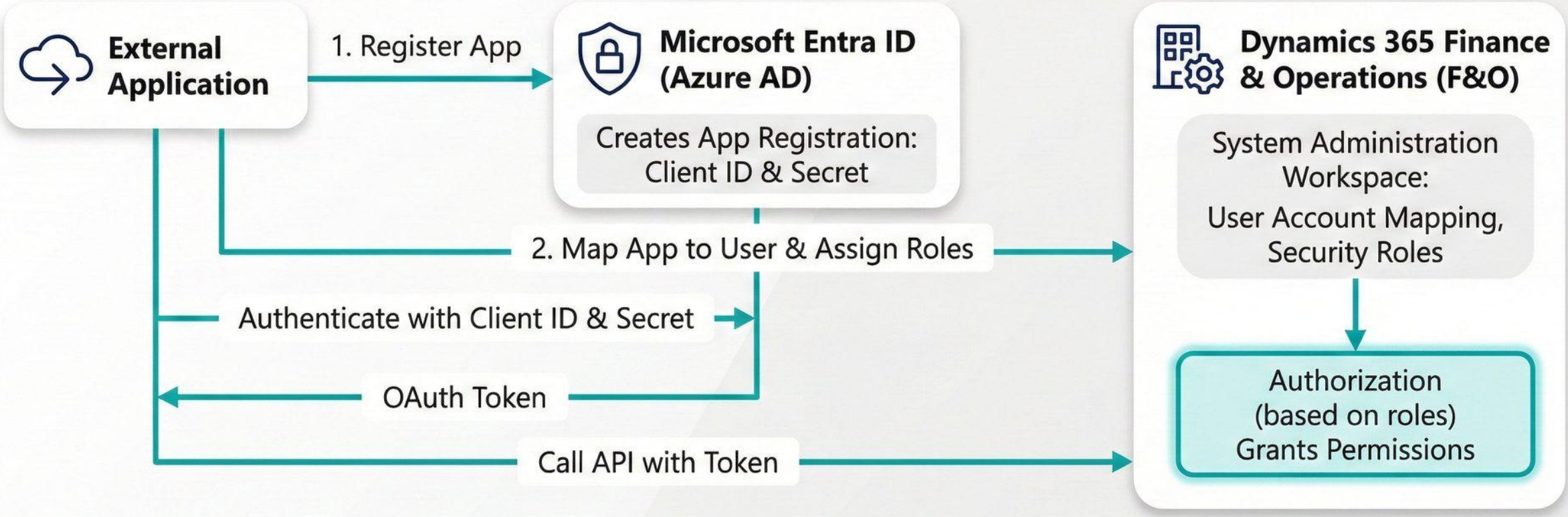
Fourth, use these classes in your code. These strongly-typed proxy classes make integration safe and IntelliSense-friendly.

Final Outcome & Benefits

Instead of dealing with string paths, you have compiled classes. Errors get caught at compile time. This is much better than manual API calls. ✓

truestack.dev

Authentication for External Applications in Dynamics 365 F&O



Entra ID handles **authentication**; F&O handles **authorization**.

Note: LCS is **not** involved in API access—it's only Entra ID and F&O.

truestack.dev

OData vs. Custom Services: When to Use Which

Choosing the Right Integration Pattern for Your Scenario



OData (Simple Data Access)

- ✓ Simpler and faster for basic data operations.
- ✓ Retrieves data (Reads and Writes).
- ✗ *Cannot* calculate results.
- ✗ *Cannot* validate complex rules.
- ✗ *Cannot* execute business logic.
- ✗ *Cannot* integrate with external APIs.



Custom Services (Complex Logic & Integration)

- ✓ Need calculated results.
- ✓ Need complex validation.
- ✓ Need business logic execution.
- ✓ Need to call external systems/APIs.
- ✓ Perfect for these complex scenarios.

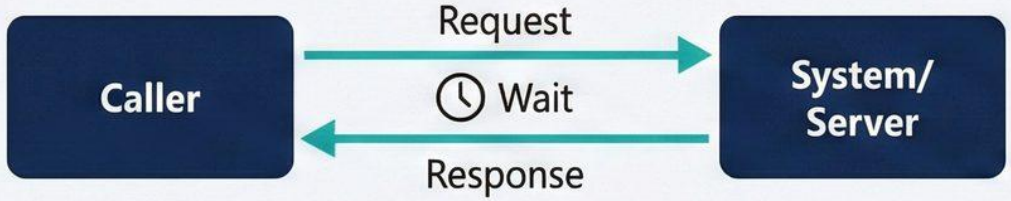
Bottom Line: Use the right tool for the job. Avoid Custom Services for simple data access.

truestack.dev

Integration Patterns: Synchronous vs. Asynchronous

Choosing the right approach based on real-time needs and processing requirements.

Synchronous Integration



Real-time. The caller makes a request and waits for a response.

Best For:

- Inventory checks
- Approval workflows
- Immediate feedback scenarios

Downside:
If network is slow or server is busy, the caller waits.

Need a real-time decision? Go synchronous.

Asynchronous Integration



Batch-oriented. The caller submits a job and continues working—results come later.

Best For:

- Bulk data sync
- Nightly imports
- Scenarios where latency is acceptable

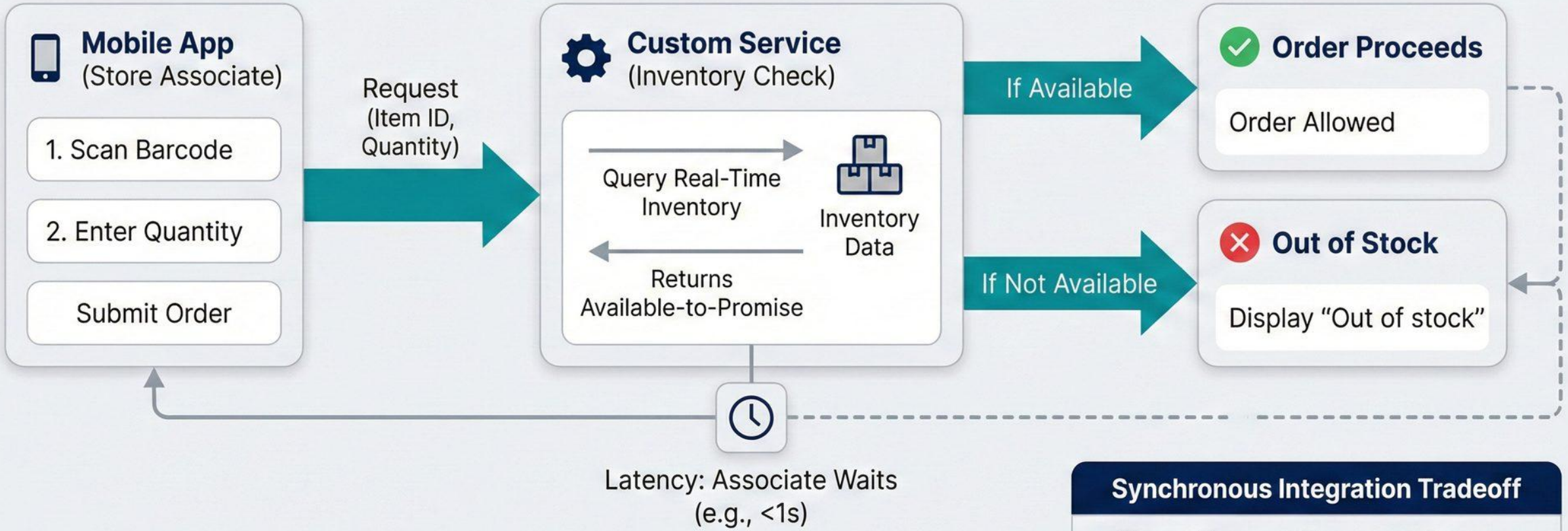
Upside:
Resilient and doesn't block the caller.

Batch processing? Go asynchronous.

Choose synchronous for immediate feedback and asynchronous for resilient, non-blocking batch operations.

truestack.dev

Real-World Synchronous Example: Inventory Check



Synchronous Integration Tradeoff

- Direct Request/Response Cycle
- Ensures Real-Time Accuracy
- Latency is Acceptable but Required

truestack.dev

Real-World Asynchronous Example: POS Sales Import

Daytime: POS Sales Collection



Morning to Evening

POS system collects sales all day. Data queues locally.

Evening Batch Job Calls DMF API

Evening: Batch Job Initiation

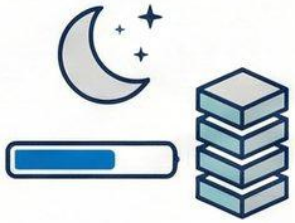


Evening

Queues Import Process

Asynchronous: POS doesn't wait

Overnight: Asynchronous Processing



Import process runs overnight.

Morning: F&O Updated



Morning

F&O has updated sales data by morning.



Latency: Hours
(Acceptable for daily reconciliation)

Efficient for high-volume, periodic synchronization with no real-time requirement.

truestack.dev

Monitoring Framework Job Status

Using the DMFExecutionSummaryStatus enum

Valid Execution Statuses (DMFExecutionSummaryStatus)

NotRun
Job is queued but hasn't started yet.

PartiallySucceeded
Completed but some records failed.

Executing
It's currently running.

Failed
The job failed with errors.

Succeeded
Completed with no errors.

Canceled
A user stopped it.

PartiallySucceeded
Completed but some records failed.

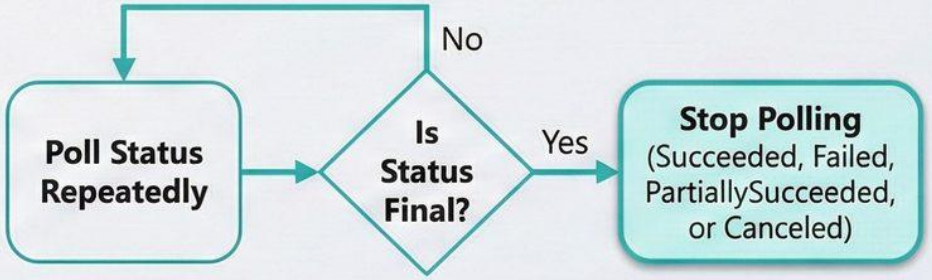
Unknown
In an indeterminate state.

Common Mistake: Invalid Status

INCORRECT

Starting is not a valid status. Do not use it.

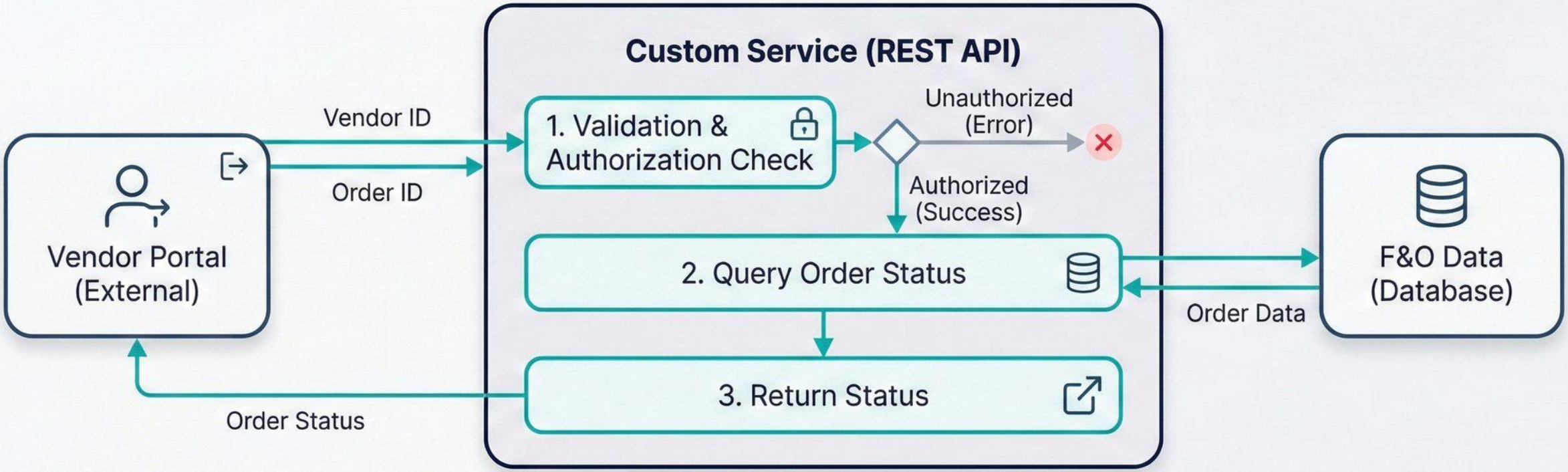
Polling Strategy



Always poll until a final status is reached; avoid using undefined values.

truestack.dev

Real-World Example: Vendor Portal Order Status via Custom Service



Key Benefits Enforced by Service:

- Controlled Access & Security (OAuth 2.0 Authentication)
- Business Logic Enforced (Authorization Validation)
- Data Isolation (Vendors see only their own orders)

truestack.dev

Four Critical Mistakes in D365 F&O Integrations

Mistake 1: Direct Table Access

❌ Accessing tables directly from external systems



Unsupported & Dangerous

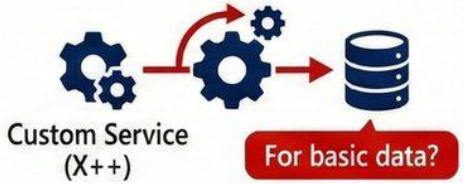
✅ Use OData or Services



Supported & Secure Pattern

Mistake 2: Custom Services for Simple Data

❌ Using custom services for simple data retrieval



Overkill & Complex

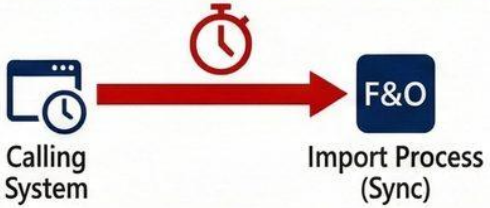
✅ Use OData



Simpler & Faster for Data

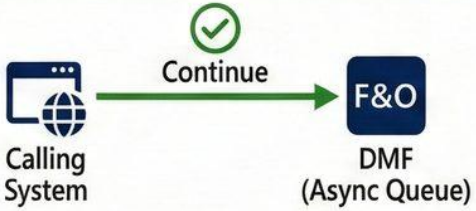
Mistake 3: Synchronous Bulk Imports

❌ Using synchronous calls for bulk imports (e.g., 10k records)



Caller Blocked, Terrible UX

✅ Use Asynchronous DMF



Non-blocking, Better UX

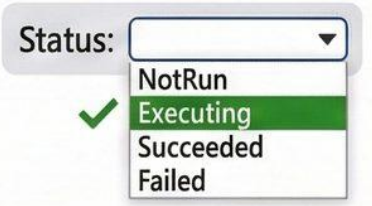
Mistake 4: Invalid Status Codes

❌ Using invalid status codes like 'Starting'



Causes Failures, Doesn't Exist

✅ Stick with Defined Enum Values



Use Valid DMFExecutionSummaryStatus

Avoid these common pitfalls to ensure secure, efficient, and reliable D365 Finance & Operations integrations.

truestack.dev

Five Best Practices for Integration Quality

1 

Use OData for Data Access Queries

Standard, documented, and requires no custom development. Best for structured data retrieval.

Highlights: Standard, Documented, Zero-Code.

2 

Reserve Custom Services for Business Logic

Ideal for calculations, validations, and complex workflows where they shine.

Highlights: Complex Logic, Validations, Workflows.

3 

Secure All APIs

Protect with Entra ID and Finance & Operations security.

Never expose unprotected endpoints.

Highlights: Entra ID, F&O Security, No Unprotected Endpoints.

4 

Use Synchronous Patterns for Real-Time Scenarios

For immediate responses where the calling system waits.

Highlights: Immediate Response, Real-Time.

5 

Use Asynchronous for Batch Operations

Suitable for bulk imports and operations tolerant of latency.

Highlights: Batch Operations, Bulk Imports.

Your integration quality depends on choosing the right pattern and following these disciplines.

truestack.dev

Topic 6.2 Recap: OData, Custom Services, and Integration Patterns



OData v4: Standard REST API

- Standard for data access.
- Query, filter, sort, page exposed entities.
- Widely supported.



Custom Services: X++ Logic Endpoints

- Expose X++ business logic as web endpoints.
- Real-time logic: tax calculation, validation, inventory checks.



Application Registration & Security

- Entra ID: Authentication.
- F&O Security Mapping: Authorization.
- LCS: No role here. (❌)



Typed C# Proxy Classes

- Generated from OData metadata.
- Strongly-typed, IntelliSense-friendly integration.



Sync vs. Async & DMF Statuses

- Synchronous: Real-time decisions.
- Asynchronous: Batch operations.
- DMF Statuses: Follow defined enum values.



Next Up: Topic 6.3

- Reacting to events asynchronously.
- Leveraging Power Platform for low-code integration.
- Focus: Event-driven integration patterns.

truestack.dev

Topic 6.3: Async and Power Platform

PREVIOUS TOPIC:
Real-time Integration

- OData
- Services

CURRENT TOPIC:
Asynchronous Patterns



- Long-running batch jobs
- Scheduled exports
- Event notifications going out to external systems

POWER PLATFORM ECOSYSTEM



Power Apps — Power Automate — Dataverse

- Central to Microsoft's ecosystem integrations
- Enables building event-driven, resilient integrations
- Scales across the entire Microsoft ecosystem

Master these patterns to build event-driven, resilient integrations that scale across the entire Microsoft ecosystem.

truestack.dev

SysOperation Execution Modes

Understanding the three execution models for background tasks in Dynamics 365 Finance & Operations



1. Synchronous

- **Execution:** Runs immediately in the user's session.
- **Session Behavior:** The form blocks until it completes.
- **User Experience:** User waits for the operation to finish.
- **Best For:**
  - Short tasks (e.g., validation).
  - Quick calculations.
- *Annotation: Fast execution but impacts user interface responsiveness.*



2. Asynchronous

- **Execution:** Runs in the background batch framework.
- **Session Behavior:** User doesn't have to wait around.
- **Scheduling:** Can be scheduled nightly or run on demand.
- **Best For:**
  - Nightly imports and exports.
  - Reconciliation tasks.
- *Annotation: Offloads processing to the server, improving user efficiency.*



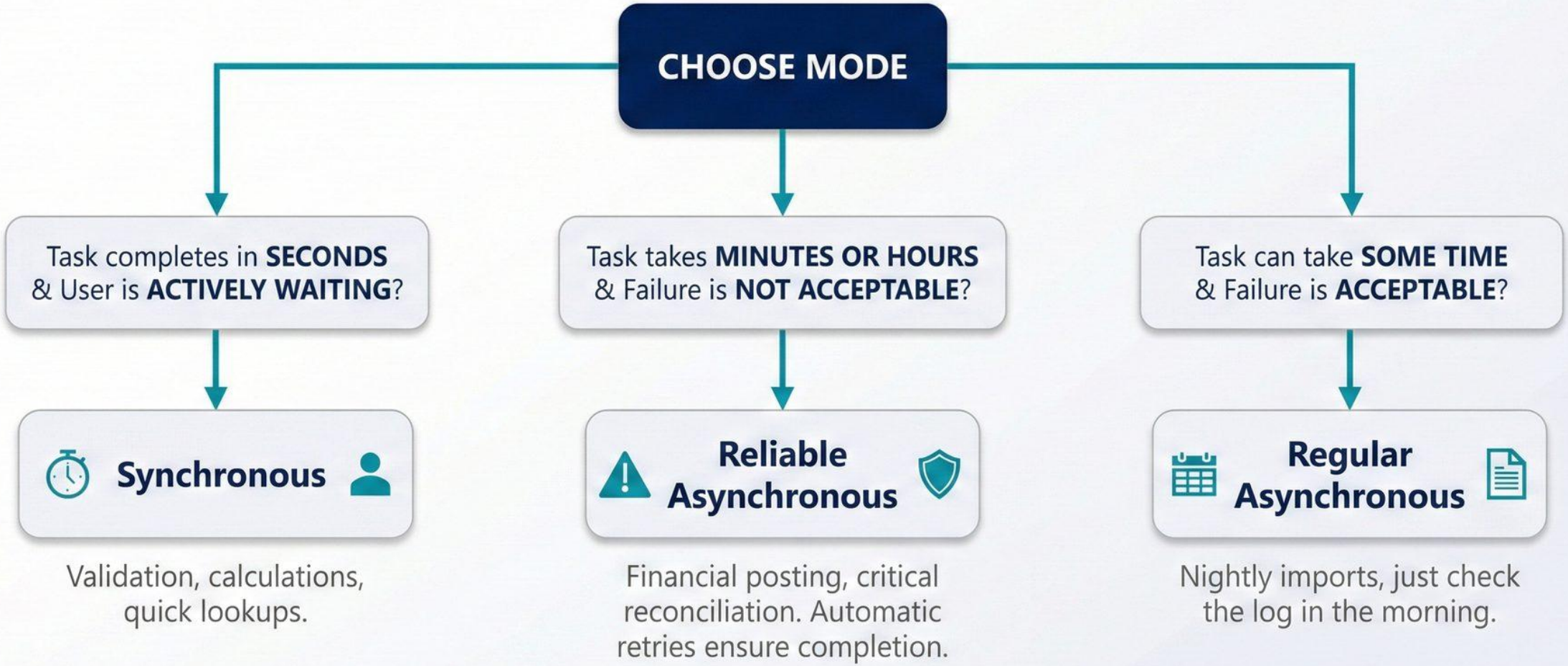
3. Reliable Asynchronous

- **Execution:** Asynchronous with enhanced reliability.
- **Session Behavior:** Runs in background, no user wait.
- **Key Feature:** Automatically retries if the batch job fails.
- **Best For:**
  - Mission-critical work.
  - Financial posting.
- *Annotation: Ensures completion for critical processes with built-in recovery.*

Choose the mode that matches your specific requirement for performance, user experience, and criticality.

truestack.dev

Decision Logic for Execution Modes



truestack.dev

Async Operations from Forms: Two Approaches

Comparing **FormRun.runAsync** and **Global.runAsync** for user interface execution.

FormRun.runAsync (Better for UI)

- ✓  User clicks button
-  Work runs in background
-  Callback fires & updates form
- ✓ Seamless user experience
- ✓ No page refresh needed

Global.runAsync (Simpler, Less Elegant)

- ✗  User clicks button
-  Work runs in background
- ✗ No callback
- ✗  Manual page refresh needed to see results

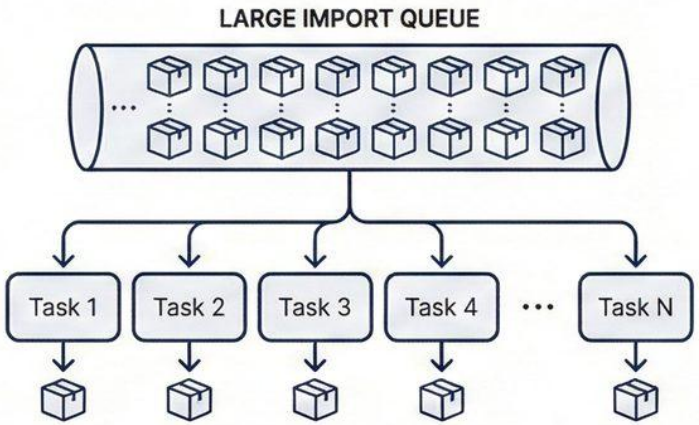
Recommendation: Use FormRun.runAsync for a professional user experience.
Use Global.runAsync when a callback isn't really needed.

truestack.dev

Batch Processing Patterns & Scalability

Comparison of Individual Task, Static Bundling, and Top Picking approaches.

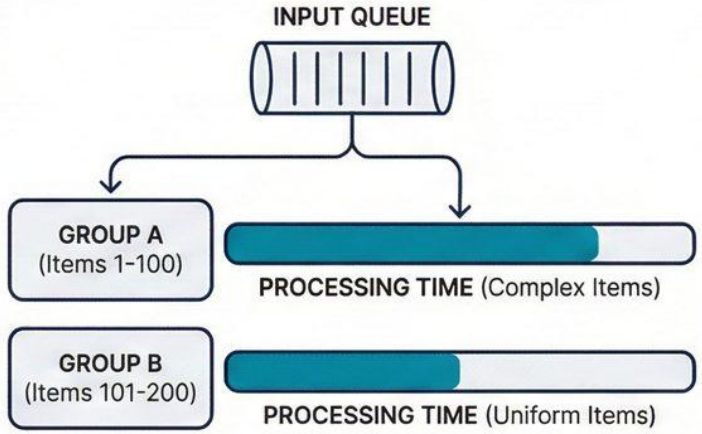
1. Individual Task Modeling



High Metadata Overhead: Thousands of tasks created for large volumes.

Simple but inefficient for high volume.

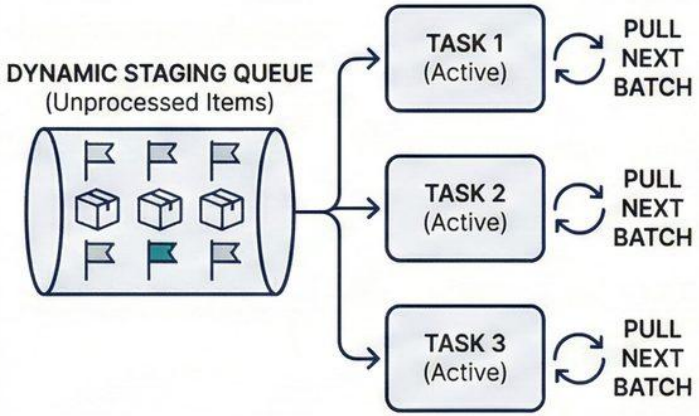
2. Static Bundling



Imbalanced Load: Complex items cause delays, underutilizing other groups.

Works only with uniform processing time.

3. Top Picking (Dynamic Queues)



Perfect Load Balancing: Tasks dynamically grab work, maximizing throughput.

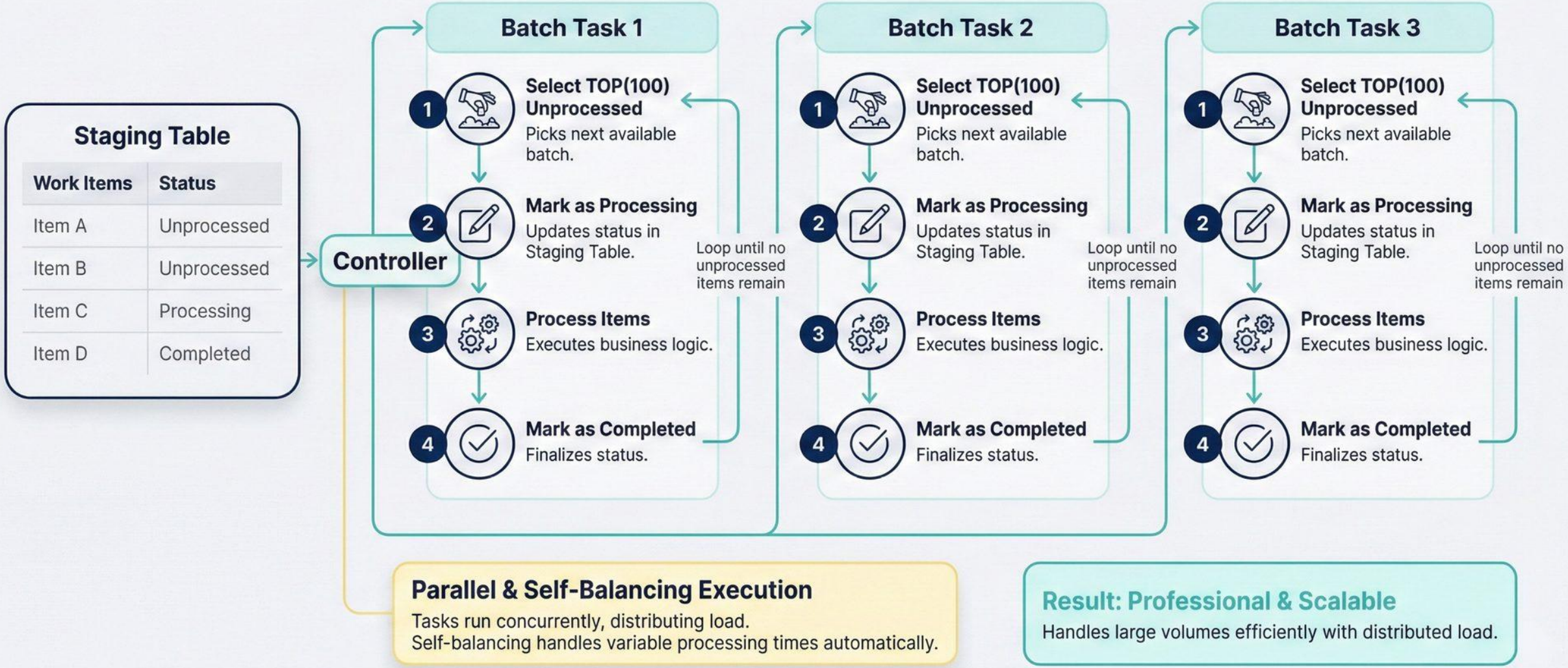
Professional pattern for scalable batch processing.

SUMMARY: Top Picking ensures optimal resource utilization and scalability by dynamically assigning work based on availability, unlike fixed or individual approaches.

truestack.dev

Top Picking Batch Workflow Pattern

Efficient, scalable, parallel processing for high-volume workloads



truestack.dev

Business Events Architecture

Overview of Core Components and Flow

Business Event Class (Raises Event)

Defines the event and raises it. Receives data, builds a contract with payload, then sends the event.

```
class SalesOrderConfirmedEvent
{
  void newFromRecord(SalesTable orderRecord)
  {
    // Build contract with order ID & customer
    this.buildContract(orderRecord.SalesId, orderRecord.CustAccount);
    // Send event
    BusinessEvents::send(this);
  }
}
```

Sends Payload

Data Contract (Holds Payload)

Holds the strongly-typed payload data fields shared by the event.

SalesOrderConfirmedContract
Order ID (String)
Customer (String)
Status (Enum)

Received by Handlers

Event Handler Class (Subscribes & Executes)

Subscribes to events, receives payload, and executes specific logic independently.

Handler 1: Send Email
Sends an email notification.

Handler 2: Post to External System
Posts data to an external API.

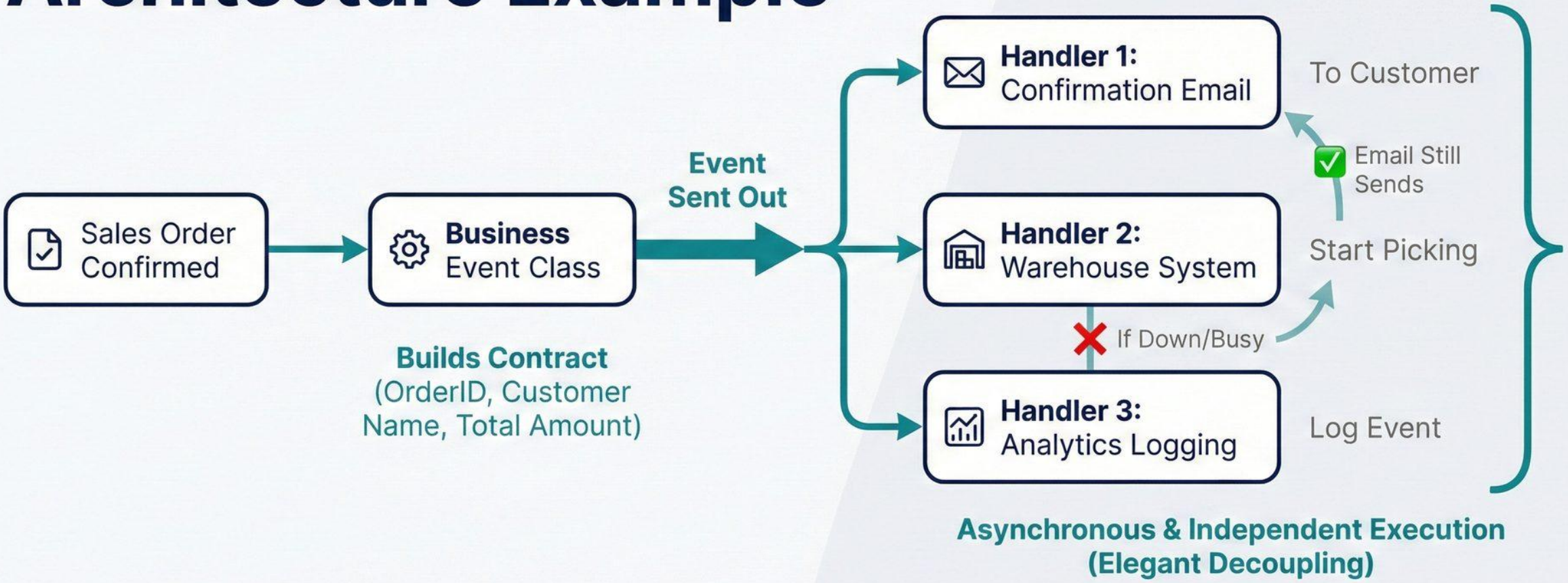
Handler 3: Update Analytics
Updates a dashboard or report.

Multiple handlers fire independently for the same event.

Key Takeaway: This architecture enables powerful, decoupled, and scalable event-driven integration scenarios.

truestack.dev

Professional Event-Driven Architecture Example



Key Benefit: Handlers run asynchronously and independently, ensuring resilience even if one system is down.

truestack.dev

Integration Tools & Their Specific Purposes



OData (Real-time Data Access)

Good for quick reads and writes, but has limited throughput. Not suited for bulk operations.

Best For: Quick, low-volume data exchange.



DMF (Data Management Framework)

Designed for bulk imports with high throughput. Ideal for importing thousands of vendor records nightly.

Best For: Large-scale, asynchronous bulk imports.



Recurring Integration (Scheduled Batch)

Basically scheduled DMF, allowing automated export data every night at a specific time.

Best For: Scheduled, recurring batch processing.



Business Events (Real-time Notifications)

Triggers immediately when a specific event occurs, like an order confirmation, allowing external systems to react instantly.

Best For: Real-time event-driven triggers.



Dual-Write (Dataverse Synchronization)

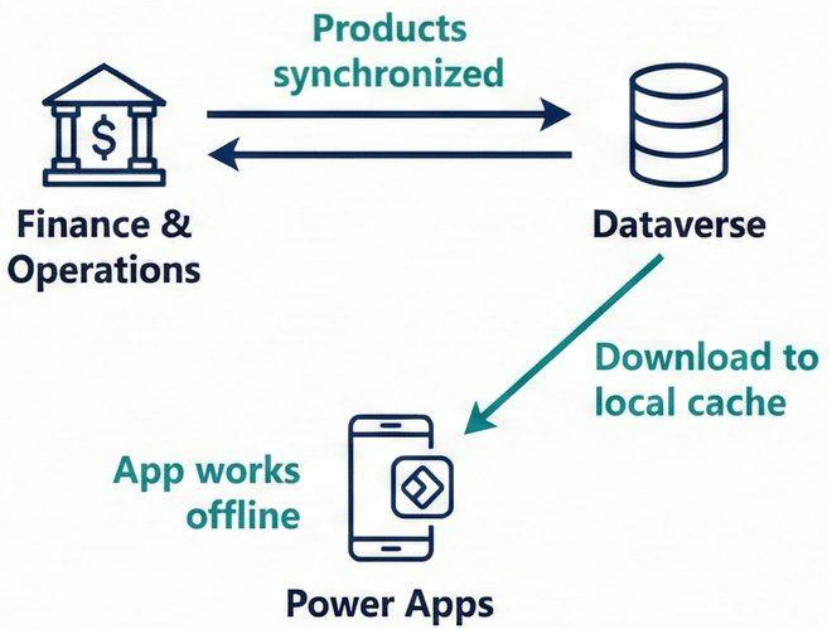
Keeps Finance & Operations and Dataverse synchronized, enabling offline capabilities for Power Apps.

Best For: Transactional sync & offline Power Apps.

truestack.dev

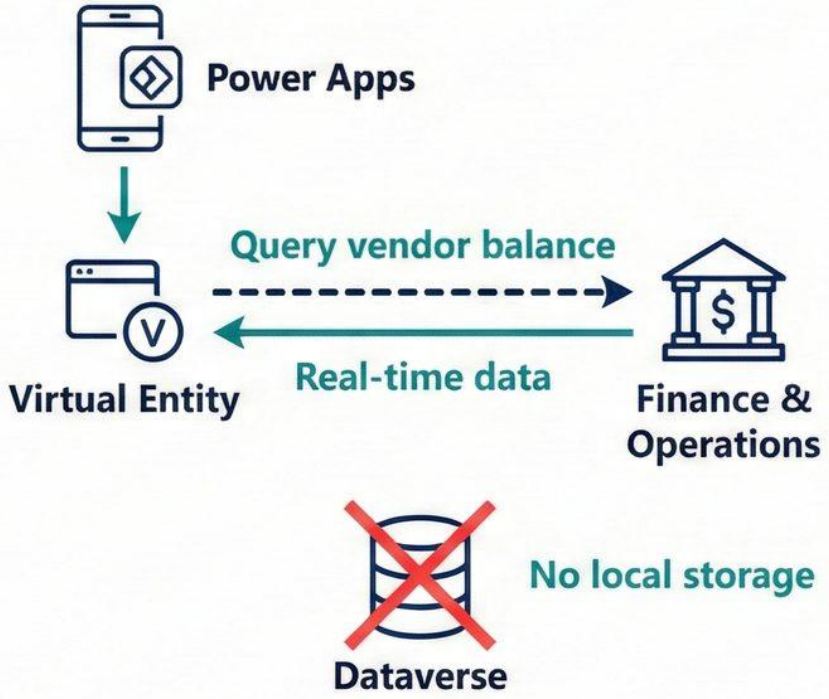
Dataverse Patterns: Dual-write vs. Virtual Entities

Dual-write



PERFECT FOR OFFLINE SCENARIOS

Virtual Entities



REAL-TIME DATA WITHOUT STORAGE

Choose dual-write when you need offline capabilities. Choose virtual entities when you want real-time data without storage.

truestack.dev

Power Apps Barcode Scanning

Mobile integration for Dynamics 365 Finance & Operations

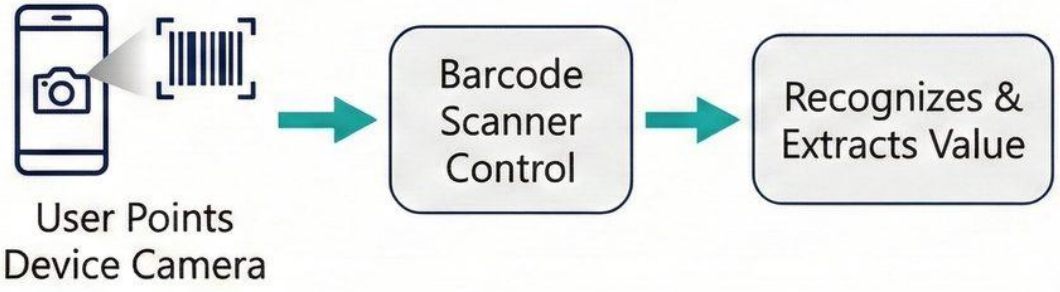
Control & Location

Found in the Media category.

Other media controls include camera, audio, and video.



How it Works



Integration & Process

App processes extracted value.



Scenarios



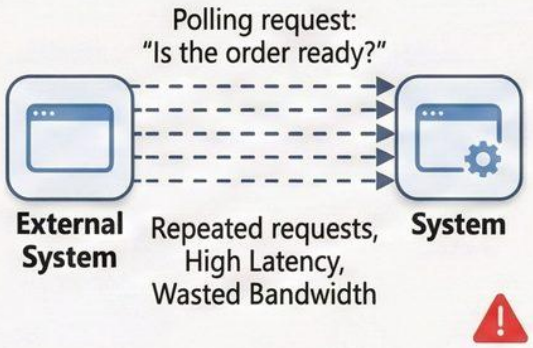
Professional mobile integration for efficient data capture and processing.

truestack.dev

Four Critical Mistakes & Corrections

Mistake 1: Polling vs. Events

INCORRECT: Polling



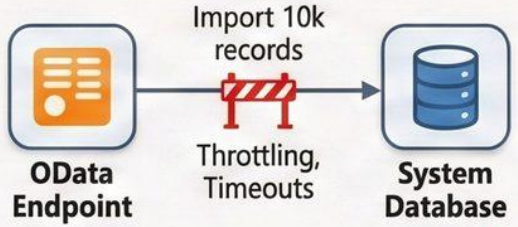
CORRECT: Use Events



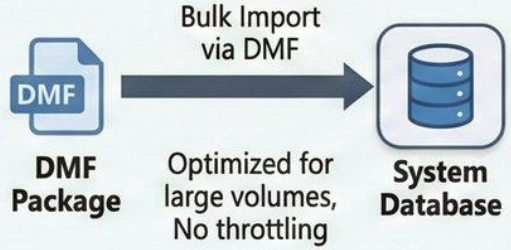
Events fire when ready, avoiding waste.

Mistake 2: OData for Bulk Imports

INCORRECT: OData for Bulk



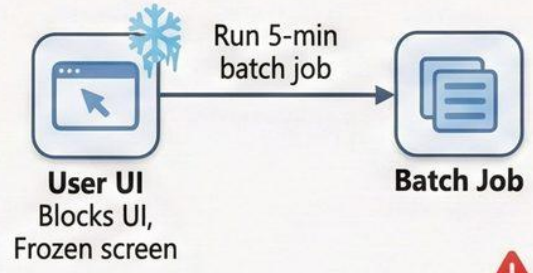
CORRECT: Use DMF



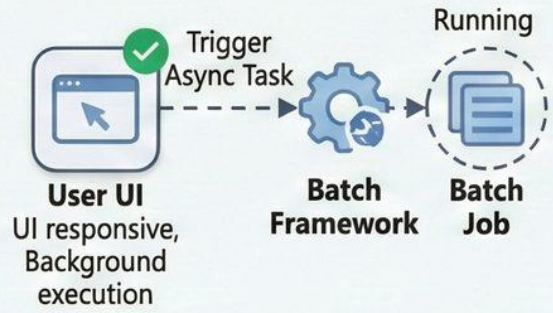
DMF handles bulk data without timeouts.

Mistake 3: Synchronous for Long Tasks

INCORRECT: Synchronous



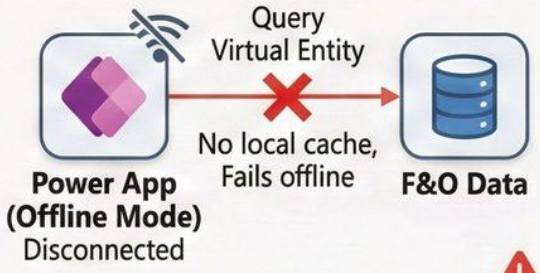
CORRECT: Use Asynchronous



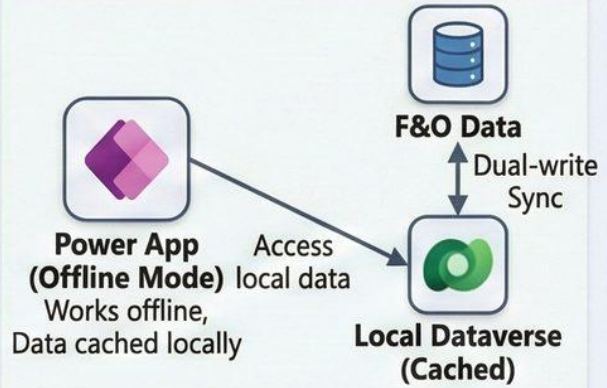
Async keeps the user interface responsive.

Mistake 4: Virtual Entities for Offline

INCORRECT: Virtual Entities Offline



CORRECT: Use Dual-write



Dual-write ensures data is available offline.

truestack.dev

Professional Design: Reuse System Fields

System Fields: Automatically Managed

Created On

Tracks when the record was created.

Modified On

Tracks when it was last edited.

Efficient Reuse Strategy

Before
Internal Schema Name

createdon

Remains unchanged internally.

RENAME DISPLAY NAME



After
User-Friendly Display Name

Entry Date

Business context label.

Schema name stays the same.
Don’t create new fields for existing functionality.

Maximize efficiency by reusing existing system fields and renaming display names for business context.

truestack.dev

Reliable Asynchronous Mode for Mission-Critical Batch Jobs

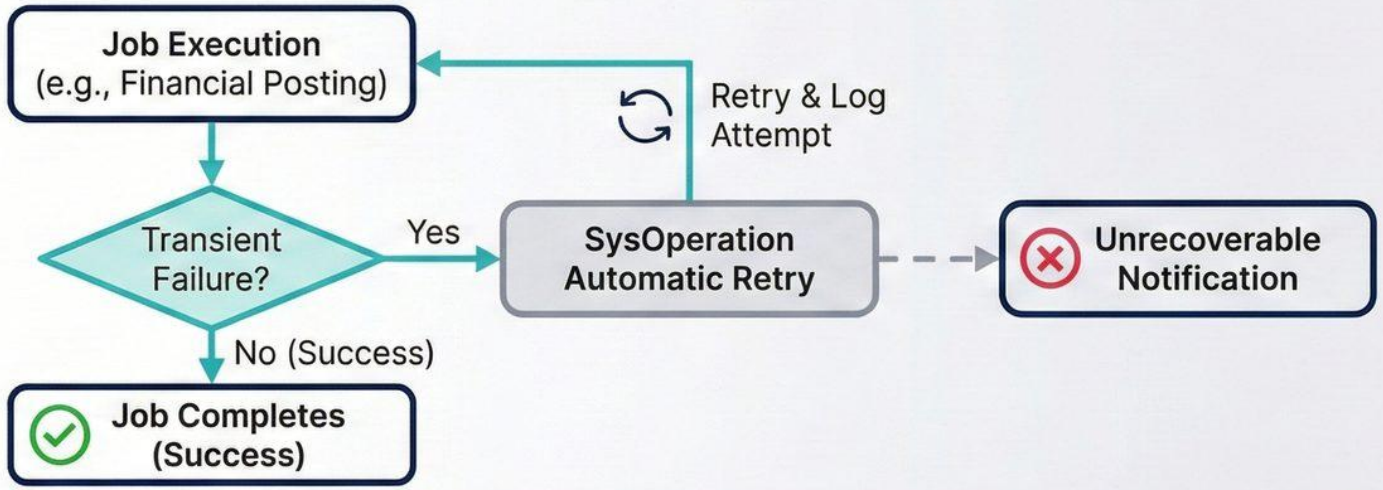
Ensuring production-grade reliability for essential processes like financial posting.

The Need for Reliability

Mission-critical jobs, such as daily general ledger entries, must not fail.

Transient failures should not delay these essential operations.

SysOperation Reliable Asynchronous Mode Flow



Production-grade reliability: *Daily financial postings are not delayed. The system automatically retries and logs attempts, ensuring completion or notification.*

Real peace of mind with robust, automated automated failure handling.

truestack.dev

Topic 6.3: Recap & Key Learnings

Async Integration, Event-Driven Patterns, and More

Async & Batch Processing

- SysOperation Execution Modes (Sync, Async, Reliable)
- FormRun.runAsync (UI callback) & Global.runAsync
- Top Picking Pattern (Staging, Dynamic acquisition)

Business Events

- Event-Driven Architecture
- Event Classes (Raise events)
- Contract Classes (Hold payloads)
- Event Handlers (Respond)

Integration Tools & Dataverse

- OData (Real-time), DMF (Bulk)
- Recurring Integration (Scheduled)
- Dual-Write (Sync/Offline)
- Virtual Entities (Real-time queries)

Power Platform & Usability

- Power Apps Barcode Scanning (Media controls)
- Reusing System Fields (Friendly display names)

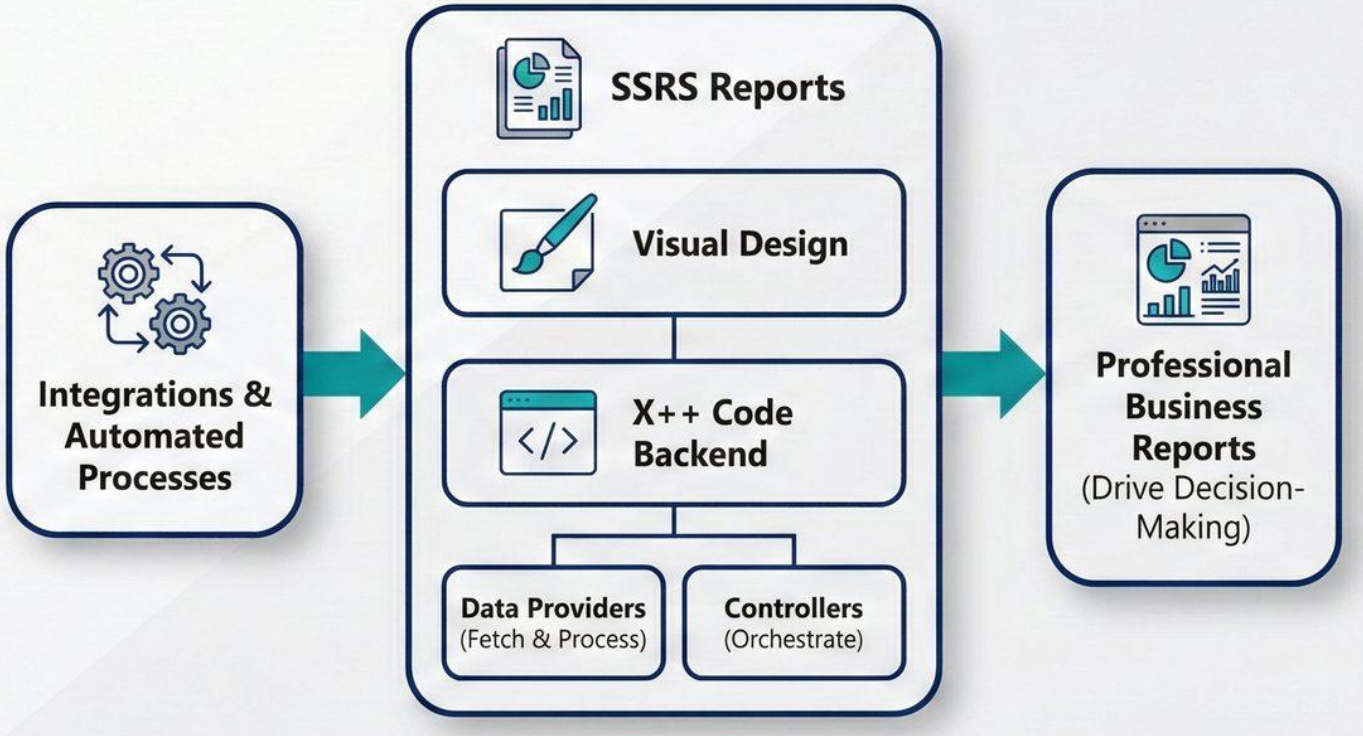
→ Next Up: Topic 7.1 on SSRS Reports – Building Professional Business Analytics

truestack.dev

Topic 7.1: SSRS Reports

Transforming Data into Business Insights with SQL Server Reporting Services

- **The Need for Reporting:** After building integrations and automating processes.
- **SSRS as the Solution:** Presenting business metrics and letting managers see trends.
- **Architecture & Design:** Reports are designed visually, but backed by X++ code.
- **Core Components:**
  - **Data Providers:** Fetch and process data.
  - **Controllers:** Orchestrate the execution.



Master these patterns, and you'll be building professional business reports that drive real decision-making.

truestack.dev

SSRS Report Architecture Components



1. Data Contract

- Holds report parameters.
- What data the user wants.
  e.g., customer, date range, product category.



2. Report Data Provider

- Queries database using parameters.
- Retrieves, processes data.
- Fills temporary tables.



3. Report Controller

- Orchestrates everything.
- Selects the right report design.
- Passes parameters to provider.
- Launches execution.



4. Report Design

- Visual layout of the report.
- Fields, tables, charts.
- All formatted nicely.



Final Report

All four work together to produce your final report output.

truestack.dev

Data Contract Class & Report Parameters

Data Contract Class Structure

Holds report parameters. Decorated with [DataContractAttribute]. Public fields represent parameters.

```
[DataContractAttribute]
public class MyReportContract
{
    [DataMemberAttribute("Parameter1", true)]
    public str parameter1;

    [DataMemberAttribute("Parameter2", false)]
    public int parameter2;
}
```

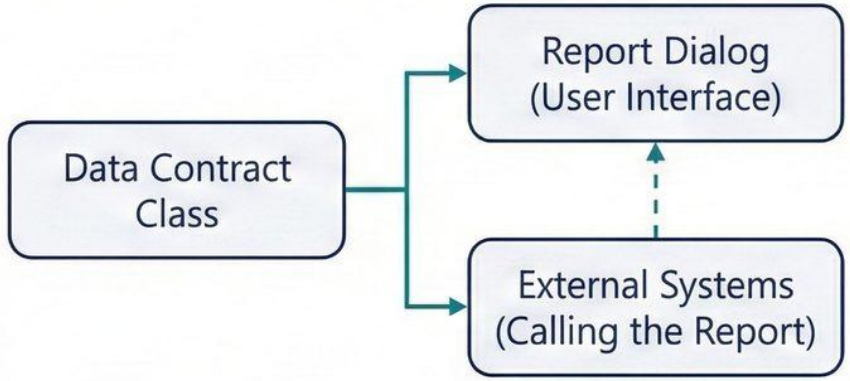
[DataContractAttribute] marks the class

Public fields define parameters

[DataMemberAttribute] makes field appear on dialog

Second parameter: True = Mandatory, False = Optional

Parameter Exposure & Contract



This class acts as the contract between the report and any caller (user or system).

Parameter Ordering

Control parameter order with [SysOperationDisplayOrderAttribute] if needed.

The Data Contract defines the interface for report execution, ensuring consistent parameter handling across all invocation methods.

truestack.dev

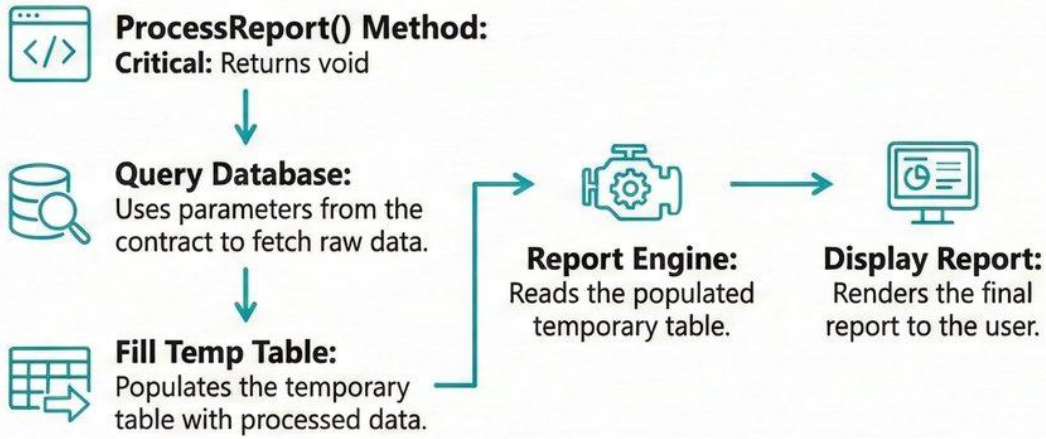
Report Data Provider (RDP)

Core Component for Data Fetching and Business Logic in SSRS Reports

Key Characteristics & Attributes

- **Fetches Data:** The central location for retrieving data.
- **[SRSReportParameterAttribute]:** Links the provider to the Data Contract.
- **Extends `SRSReportDataProviderBase`:** Inherits necessary base functionality.
- **[SRSReportDataSetAttribute]:** Marks the dataset method and specifies the temporary table holding data.
- **Business Logic Hub:** Contains all data processing and transformation logic.

Execution Flow & Methods



Dataset Method
Returns the populated temporary table buffer to the report engine.

The Report Data Provider orchestrates data retrieval, processing, and population of temporary tables, housing the report's essential business logic.

truestack.dev

Report Controller Orchestration

Extends SRSReportRunController to manage execution flow.

Orchestration & Execution



Acts as the conductor of the report execution process.

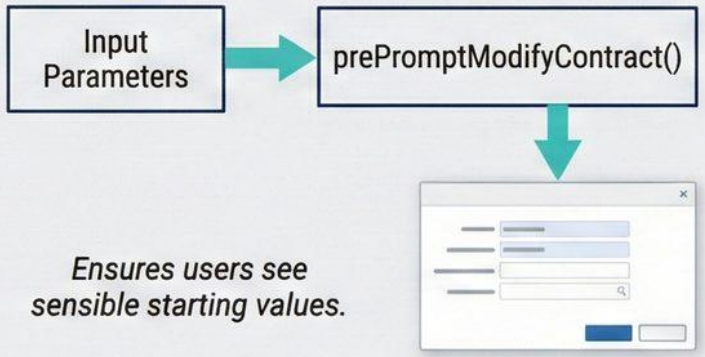


The controller is what runs when called.

Setting Default Parameters



Override **prePromptModifyContract** for reasonable defaults.



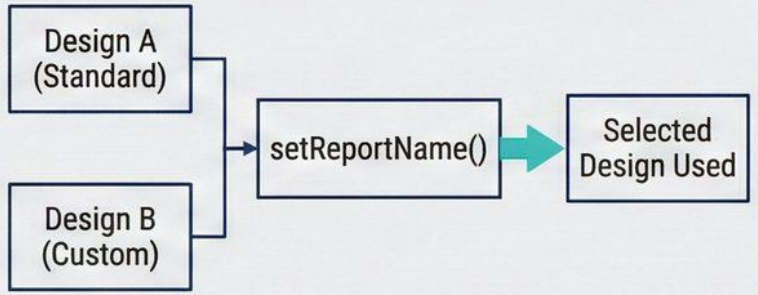
Ensures users see sensible starting values.

Dialog with Defaults Seen by User

Selecting Report Design



Override **setReportName** to choose design.



Decides which design to use when multiple exist.

Takeaway: The report controller orchestrates the entire process, from setting parameters to launching the selected report design.

Report Deployment Paths in Dynamics 365 F&O

Validating report object deployment for SSRS

VALID DEPLOYMENT PATHS



Solution Explorer

Right-click Project > Select Deploy

Deploys all report objects in that project—fast for development.



Application Explorer

Right-click Specific Report > Deploy

Deploys just that one report. Both are valid approaches.

WHAT DOESN'T WORK



Build Project

Builds your X++ code but doesn't deploy SSRS designs.



Build Models

Compiles metadata but doesn't deploy designs.



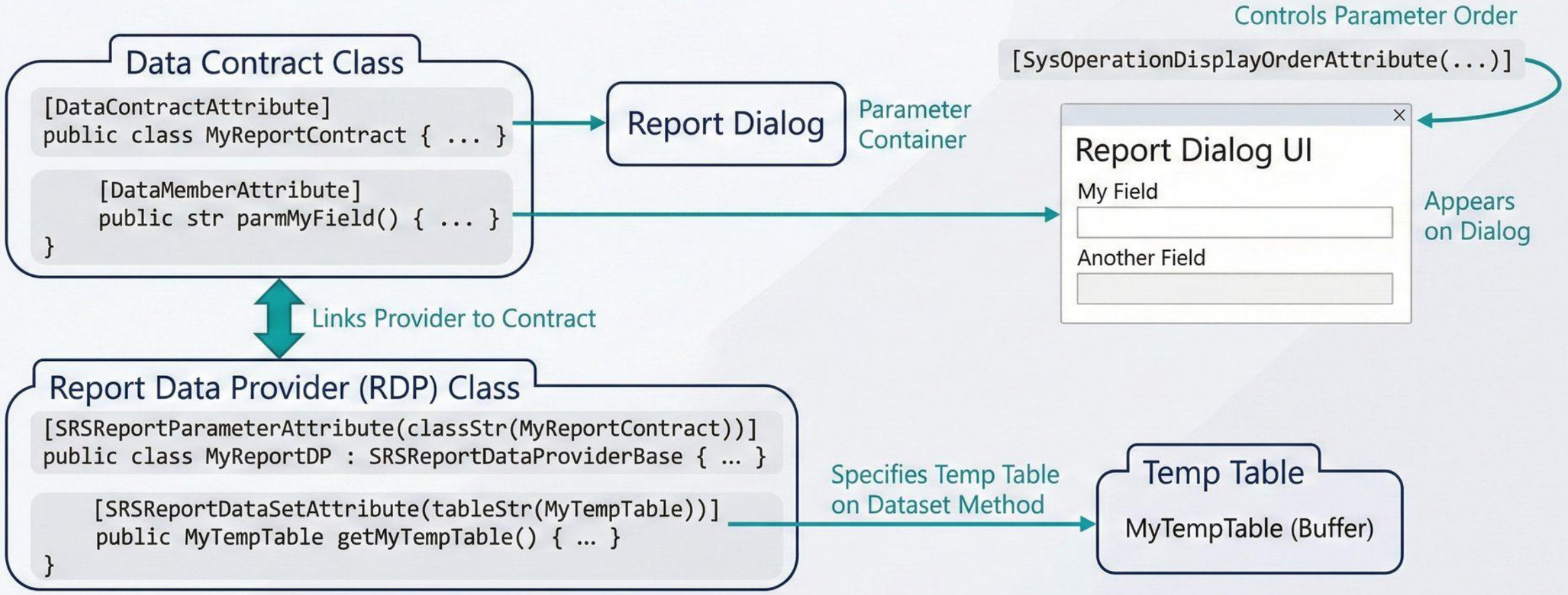
Package Deployment

Moves compiled artifacts but not designs.

Always use Solution Explorer or Application Explorer for SSRS reports.

truestack.dev

Critical Framework Attributes in D365 F&O Reporting



You've got to apply all the required attributes. If you miss any required attribute, the framework just won't recognize the class.

truestack.dev

Report Data Provider (RDP) Strict Requirements

Adherence to framework rules is non-negotiable for execution.

✅ Mandatory Requirements

Extend `SRSReportDataProviderBase`

```
[SRSReportParameterAttribute(classStr(MyReportContract))]
class MyReportDP extends SRSReportDataProviderBase
{
    ...
}
```

`ProcessReport` method must return `void`

```
public void processReport()
{
    // Logic to populate temp table
}
```

Dataset method with `SRSReportDataSetAttribute` specifying temp table

```
[SRSReportDataSetAttribute(tableStr(MyReportTmpTable))]
public MyReportTmpTable getMyReportTmpTable()
{
    return myReportTmpTable;
}
```

⚠ Common Mistakes & Consequences

Not extending the correct base class.

- Breaks the framework's expected structure.

Returning `boolean` or `int` from `ProcessReport`

```
❌ public boolean processReport() { ... return true; }
❌ public int processReport() { ... return 0; }
```

- Framework fails to execute the method correctly.

Missing `SRSReportDataSetAttribute` on dataset method.

- Report cannot identify the temporary table data source.

Strict adherence to the framework's class inheritance, method signatures, and attributes is essential for successful report execution.

truestack.dev

Process for Creating a Parameterized Report

1. 

Create Contract Class

Define the contract class containing the necessary input parameters.

2. 

Decorate with DataMemberAttribute

Apply `DataMemberAttribute` to expose fields for serialization.

3. 

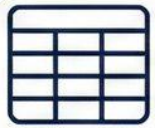
Create RDP Class

Develop the Report Data Provider (RDP) class and link it to the contract.

4. 

Implement processReport (void)

Define the `processReport` method with a `void` return type for execution.

5. 

Add Dataset Method & Attribute

Incorporate a dataset method decorated with the required attribute to define the output buffer.

6. 

Write Logic (Query & Fill Temp Table)

Write business logic to query the database and populate the temporary table.

7. 

Create Visual Design in SSRS

Design the report layout in SSRS, adding fields, tables, and charts.

8. 

Deploy from Solution Explorer

Deploy the report and its components directly from Solution Explorer.

9. 

Test via Menu Item

Verify functionality by running the deployed report from a menu item.

This nine-step process is systematic and complete, ensuring a robust parameterized report implementation.

truestack.dev

Extending Locked Standard Reports Safely

1 Extend the Provider Class



Use Chain of Command to populate the new field values in your extension.

2 Clone the Report Design



Duplicate the standard design and add the new field to your custom copy.

3 Extend the Controller Class



Override setReportName to use your cloned design instead of the standard one.

Outcome & Benefit

Now every execution uses your design with the new field, and the **standard report remains completely untouched.** It's upgrade-safe.

Bottom Takeaway: This process ensures custom designs replace standard ones without modifying base objects, maintaining upgrade compatibility.

truestack.dev

Critical Mistakes in D365 F&O Report Development

Common Pitfalls and How to Avoid Them

Mistake 1: Incorrect ProcessReport Signature

INCORRECT

```
public boolean ProcessReport() {
    // Incorrect return type
    return true;
}
```

Framework expects 'void'. Report will not execute.

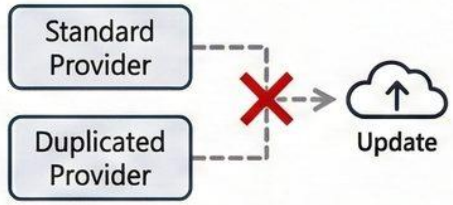
CORRECT

```
public void ProcessReport() {
    // Correct signature
}
```

Method must return 'void' for proper execution.

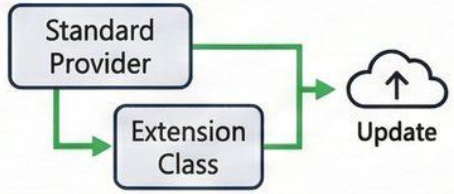
Mistake 2: Duplicating Provider Class

INCORRECT



Breaks upgradeability. Copy doesn't receive Microsoft updates.

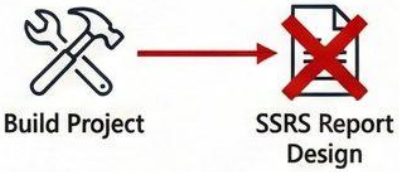
CORRECT



Use extensions. Maintains upgrade path and receives standard updates.

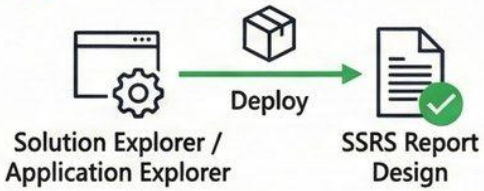
Mistake 3: Deploying via Build Project

INCORRECT



Builds code but fails to deploy SSRS designs.

CORRECT



Use Solution Explorer or Application Explorer for report design deployment.

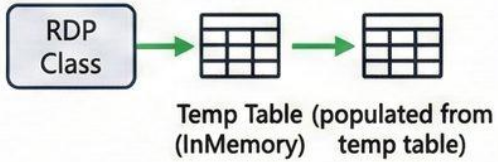
Mistake 4: Modifying Tables Directly in RDP

INCORRECT



Breaks contract with framework. Standard tables should not be modified here.

CORRECT



Always use temp tables to populate and pass data to the report.

Avoiding these mistakes ensures functional, upgradeable, and deployable reports in Dynamics 365 Finance & Operations.

truestack.dev

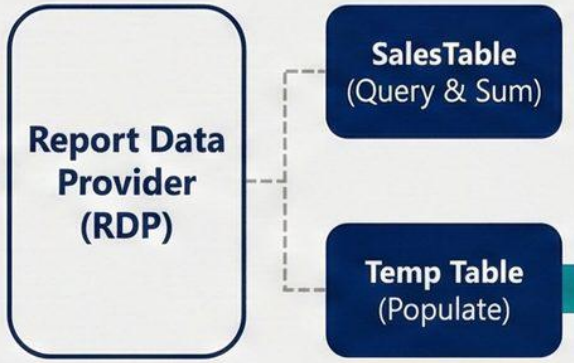
Real-World Example: Customer Sales Report Workflow

From User Input to Actionable Business Insights in Dynamics 365 Finance & Operations

1. User Parameters (Dialog)



2. Data Processing & Aggregation



3. Final Report Visualization



4. Manager Analysis (Business Value)



Identify top revenue drivers and analyze performance.

This workflow demonstrates how structured data contracts and providers transform raw data into professional business reporting.

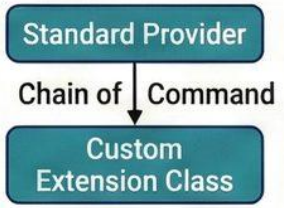
truestack.dev

FIVE BEST PRACTICES FOR SSRS REPORTS

A guide to efficient and maintainable report development.

Practice 1: Extend, Don't Duplicate

CORRECT



Use Chain of Command to customize providers—it preserves upgradeability.

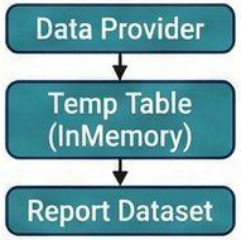
INCORRECT



Avoid duplicating provider classes.

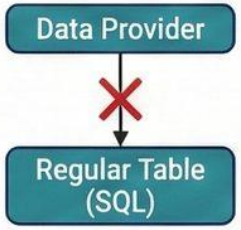
Practice 2: Use Temp Tables

CORRECT



The framework requires it.

INCORRECT



Don't query tables directly.

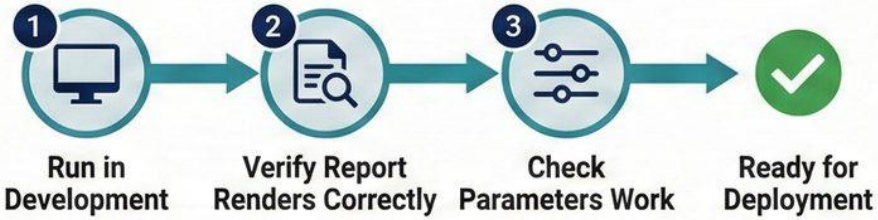
Practice 3: Apply All Required Attributes

- [DataContractAttribute] on contract class
- [DataMemberAttribute] on parm methods
- [SRSReportParameterAttribute] on RDP class

Each attribute serves a purpose.

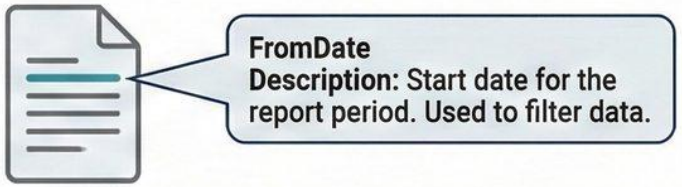
Missing one breaks the framework.

Practice 4: Test Before Deploying



Run it in development, verify the report renders correctly, check that the parameters work.

Practice 5: Document Your Parameters



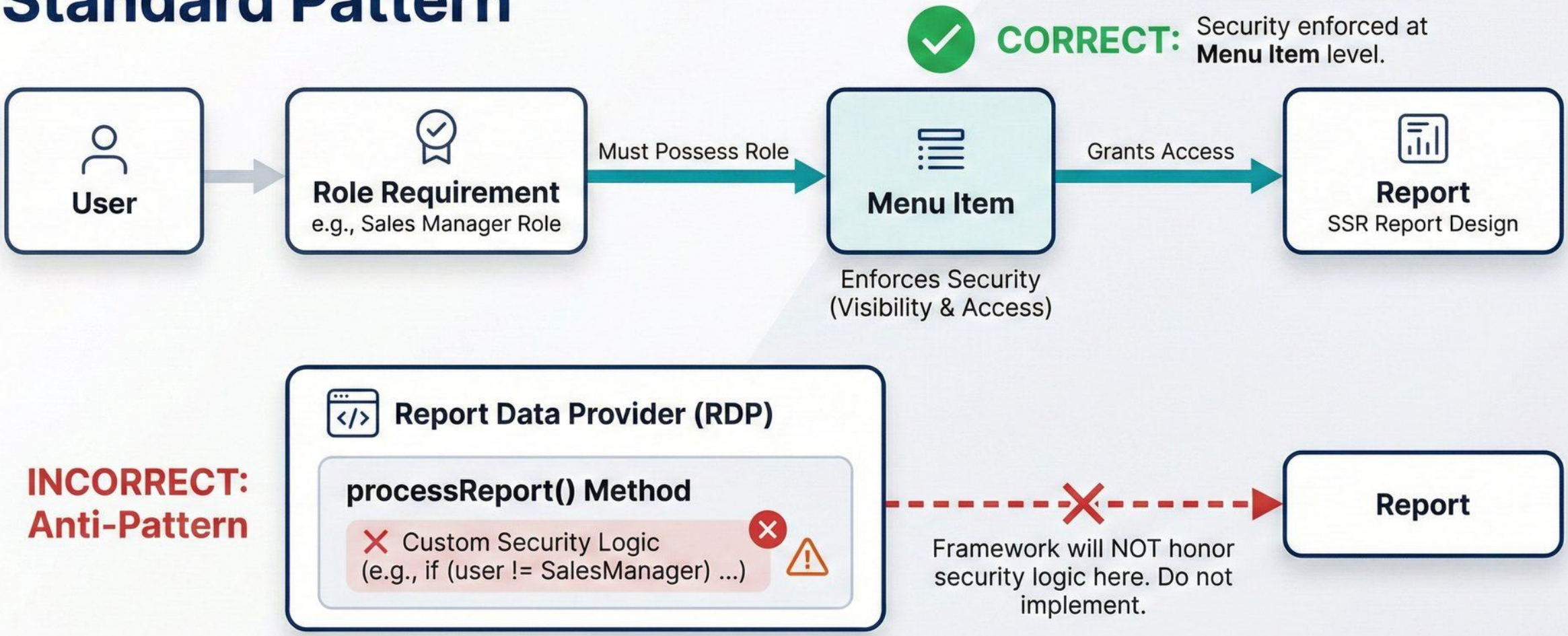
Help users understand what each parameter does.

Professional documentation really improves adoption.

Following these practices ensures maintainable, upgrade-safe, and user-friendly SSRS reports.

truestack.dev

Report Security Implementation Standard Pattern



The professional, standard way to handle report security is letting the Menu Item enforce role requirements, not within the data provider logic.

truestack.dev

Recap of SSRS Reports (Topic 7.1) and Introduction to Embedded Reporting (Topic 7.2)

Summary of key concepts covered and looking ahead to advanced analytics.

Recap: SSRS Report Architecture & Development (Topic 7.1)



SSRS report architecture: Components and their roles.

Required Framework Attributes: DataContractAttribute, DataMemberAttribute, SRSReportParameterAttribute, SRSReportDataSetAttribute.

Report Deployment: Use Solution Explorer or Application Explorer (Build Project doesn't deploy designs).

Data Provider Requirements: Extend base class, processReport returns void, dataset method marked with attribute.

Parameterized Reports: Nine-step systematic creation process.

Extending Locked Reports: Chain of Command on provider, clone design, extend controller for upgrade-safe customization.

Best Practices: Extend don't duplicate, use temp tables, apply attributes, test, document.

Report Security: Managed at the menu item level.

Next: Embedded Reporting & Advanced Analytics (Topic 7.2)

How do you build embedded reports in forms?

How do you leverage advanced analytics?

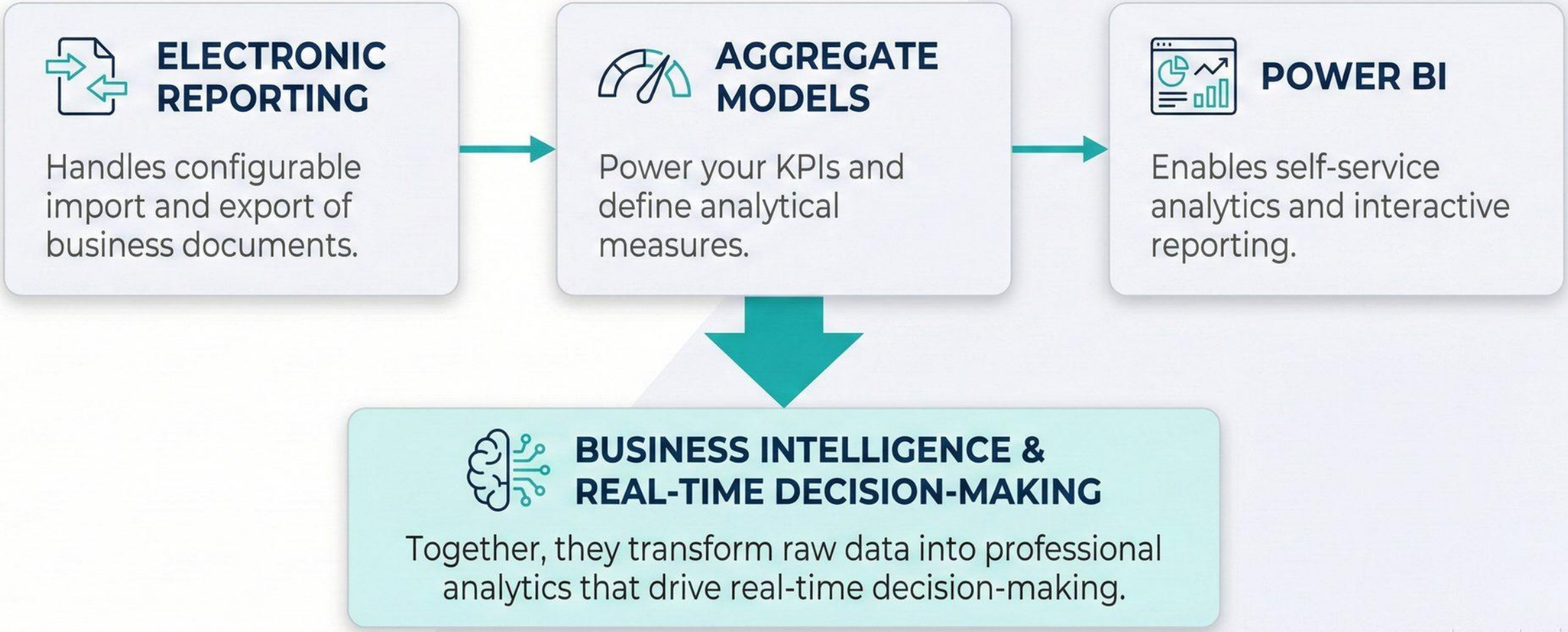
We'll learn about embedded reporting and business intelligence.

Understanding SSRS foundation sets the stage for exploring embedded and advanced analytical capabilities.

truestack.dev

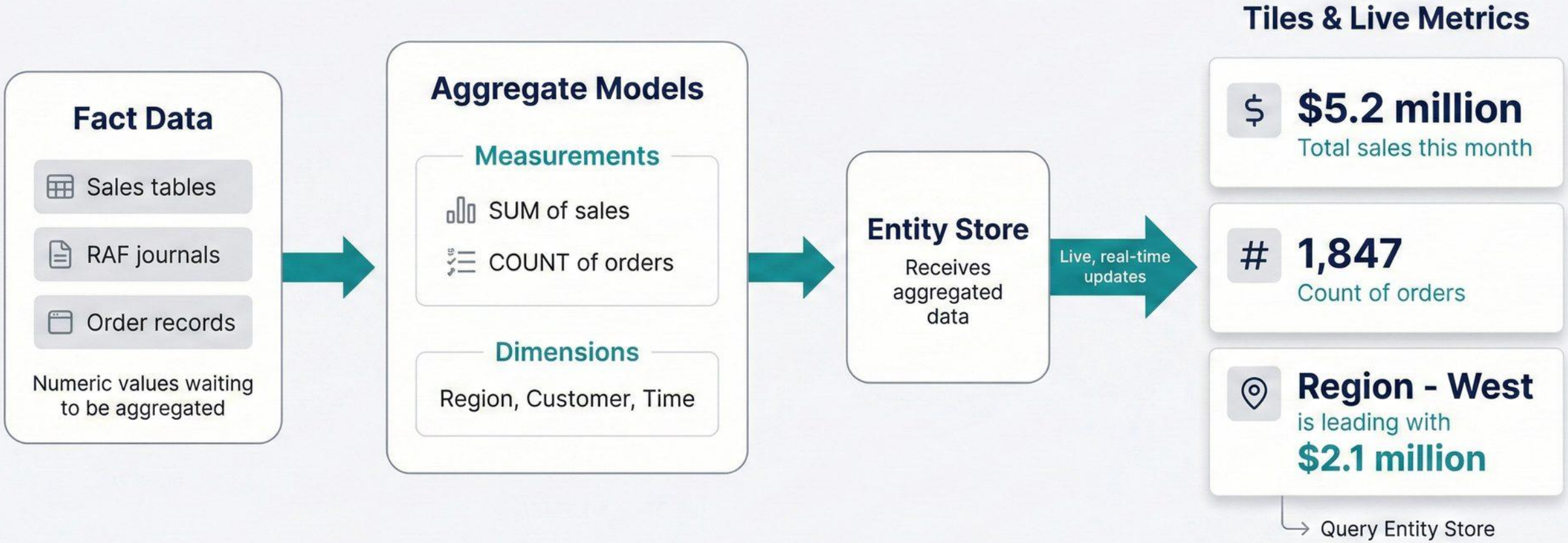
Topic 7.2: ER and Analytics – Real-time KPIs & Configurable Reporting

Moving beyond detailed SSRS reports to real-time metrics and user-configurable insights.



truestack.dev

KPI Architecture: From Fact Data to Live Metrics



KPI architecture transforms raw fact data into actionable, real-time live metrics through aggregate models and the Entity Store.

truestack.dev

Aggregate Measurements in Dynamics 365 F&O

Defining your numeric facts and key performance indicators.

Core Function

Aggregate measurements define your numeric facts, specifying calculation operations on underlying data.

- What operation to perform
- Which field to operate on
- Filters to apply

Common Operations

- **SUM**: Total sales amount
- **COUNT**: Number of orders
- **AVG**: Average order value
- **MIN**: Minimum delivery time
- **MAX**: Highest unit price

Technical Properties & Purpose

Properties

- **MeasurementType**: Specifies calculation
- **IsAggregated** flag: Pre-calculated value indicator
- **Source field**: Underlying table column

Answers Business Questions

- What is our total revenue?
- How many customers do we have?
- What's our average order size?

Aggregate measurements are the foundation for analyzing numeric data and generating business insights.

truestack.dev

Aggregate Dimensions: Your Grouping Context

Dimensions provide the context for grouping and slicing analytical data.

Region



Slice sales by geography.

↓

What are sales by region?

Customer Group



Slice by customer segment.

↓

How many orders by customer group?

Hour



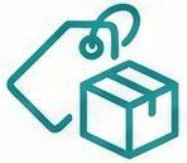
Slice by time of day.

Date



Slice by day or month.

Product Category



Slice by product line.

↓

What's revenue by product?

Requirement for Effective Analytics



Dimensions **must map** to stable fields in your tables.
Changes to dimension values shouldn't break your analytics.

Good dimensions enable meaningful analysis.

truestack.dev

Real KPI Example: Total RAFs per Hour

Fact Data & Components

RAF Journal Table

- Timestamps

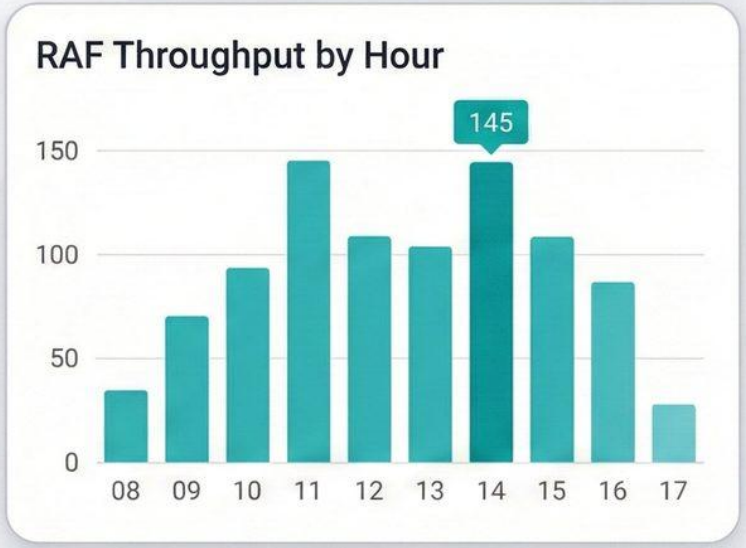
Analytics Model Definition

Measurement
COUNT of RAF lines

Dimension
Hour (extracted from timestamp)

Bind in Model → Entity Store → Deploy

Visualization & Benefits



- Managers see RAF throughput visualized by hour.
- At a glance processing volume.
- Trend analysis shows peak hours.

This process demonstrates professional analytics in action, transforming raw data into actionable insights.

truestack.dev

KPI Example: Sales by Region

Understanding Measurement, Dimension, and Business Value

Conceptual Model & Components

Fact Data
Sales table with amount and region.

↓

Measurement
SUM of sales amount.

↓

Dimension
Region.

KPI Tile: Total Sales by Region



Managers can instantly see regional performance, identify high performers and underperformers, and make resource allocation decisions.

Key Takeaway: This is what drives real business value.

truestack.dev

Four Critical KPI Modeling Mistakes to Avoid

Common pitfalls in Microsoft Dynamics 365 Finance & Operations KPI modeling and their correct solutions.

MISTAKE	WHAT IT IS	WHY IT'S WRONG	CORRECT APPROACH
✗ Using data entities for KPIs.	Used for data import and export tasks.	Not optimized for analytics or reporting performance.	✓ Use aggregate models instead.
✗ Using TempDB tables.	Session-temporary database tables.	Data disappears when the user logs off, not suitable for persistent analytics.	✓ Use fact tables for persistent data.
✗ Using numeric fields as dimensions.	Trying to group sales or other facts by amount (e.g., sales by amount).	Does not make sense; numeric fields are measurements, not categories.	✓ Amount should be a measurement, dimensions are categories.
✗ Creating manual aggregated views.	Writing custom SQL queries for views.	The framework generates aggregate views automatically from your model.	✓ Let the framework handle it automatically.

Ensure efficient and accurate KPI modeling by leveraging **aggregate models**, persistent **fact tables**, **correct dimension categorization**, and the **automatic framework** for aggregations.

truestack.dev

Creating a KPI Tile in 3 Steps

1



1. Create Aggregate Model & Deploy

Define measurement (what to display) and dimension (what to group by), then deploy to Entity Store. The framework builds the analytics database.

2



2. Create Tile & Define Filter

Create a tile in the workspace. Point it to your measurement and specify any filters, such as showing only West region sales.

3



3. Pin to Workspace & View Live

Pin the tile to a workspace to display on the dashboard showing real-time data as it comes in.

Outcome: Your managers always see current metrics updated immediately as sales come in.

truestack.dev

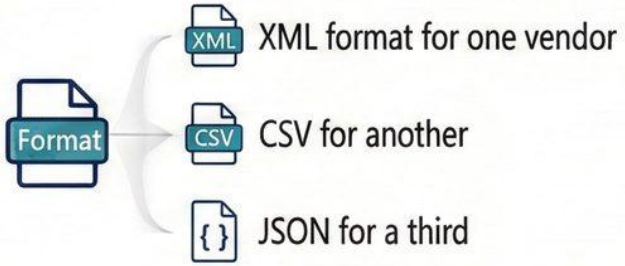
Electronic Reporting (ER) - Core Components

Configurable import/export, data transformation, and validation logic

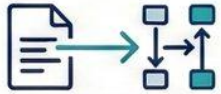
Format



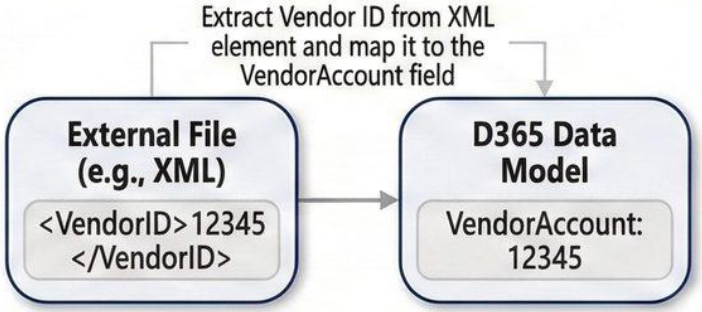
Defines your file structure. Specifies the layout and organization of external data files.



Format Mapping



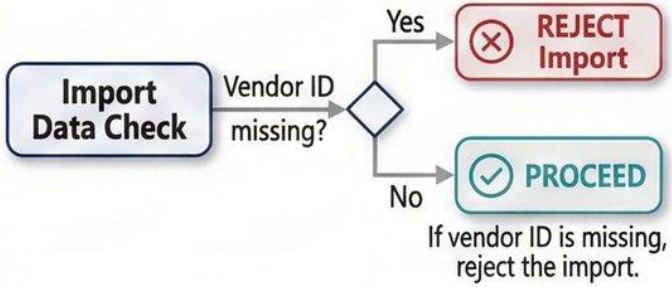
Transforms that file structure to the D365 data model. Bridges external data to internal fields.

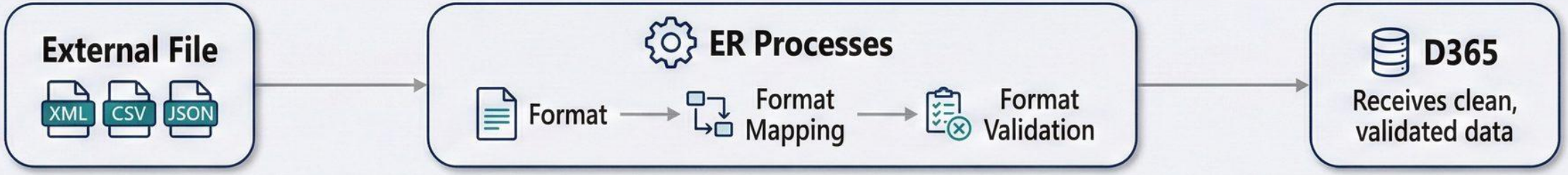


Format Validation



Enforces your business rules. Ensures data integrity during import.



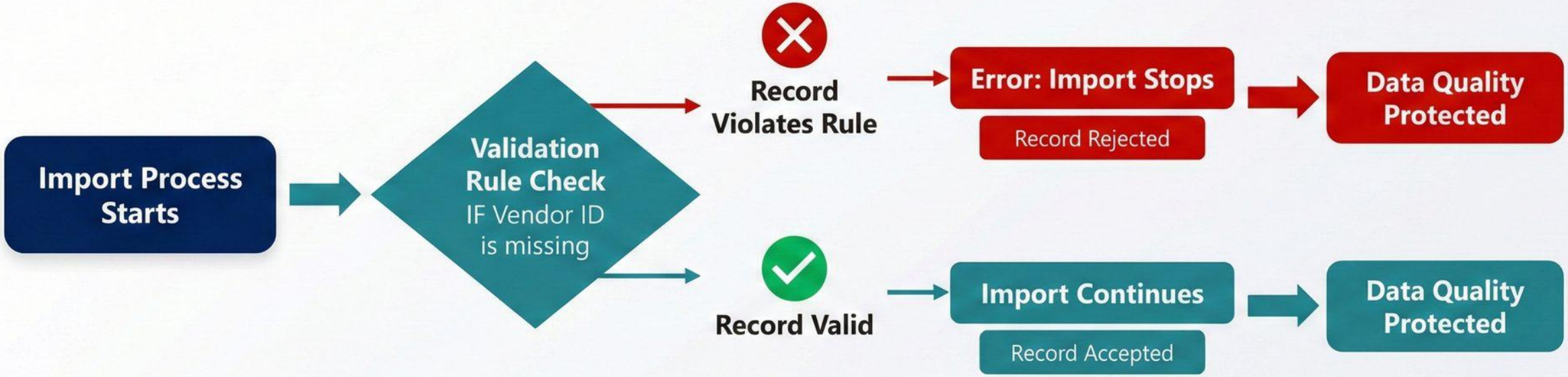


These three components work together to ensure seamless and accurate data handling in Dynamics 365.

ruestack.dev

ER Format Validation: Enforcing Data Quality

Define rules to protect your system from bad data during import.



Error (Critical)

- Stops import immediately.
- Used for critical business rules.
- Example: Vendor ID is required.

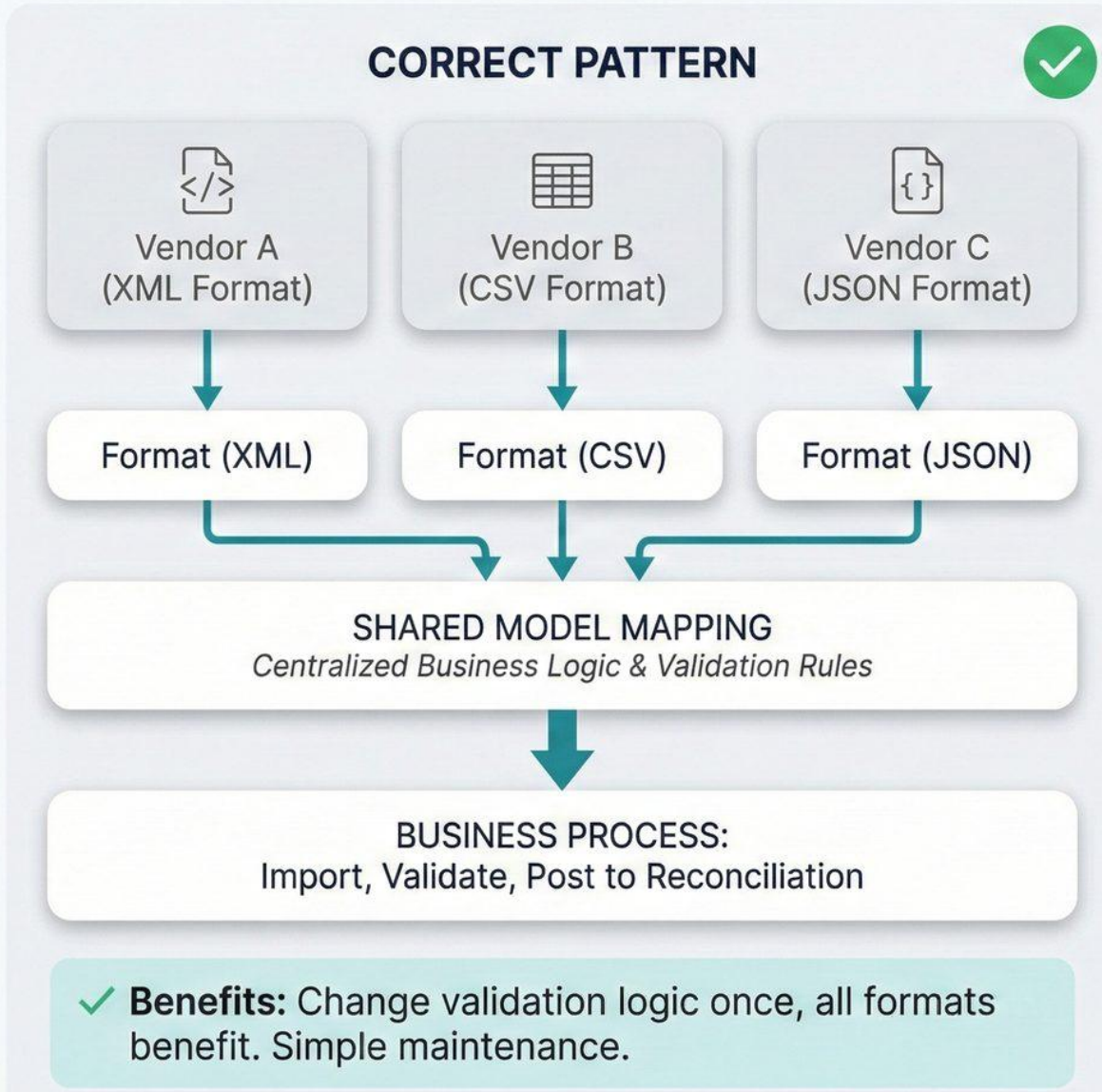
Warning (Non-Critical)

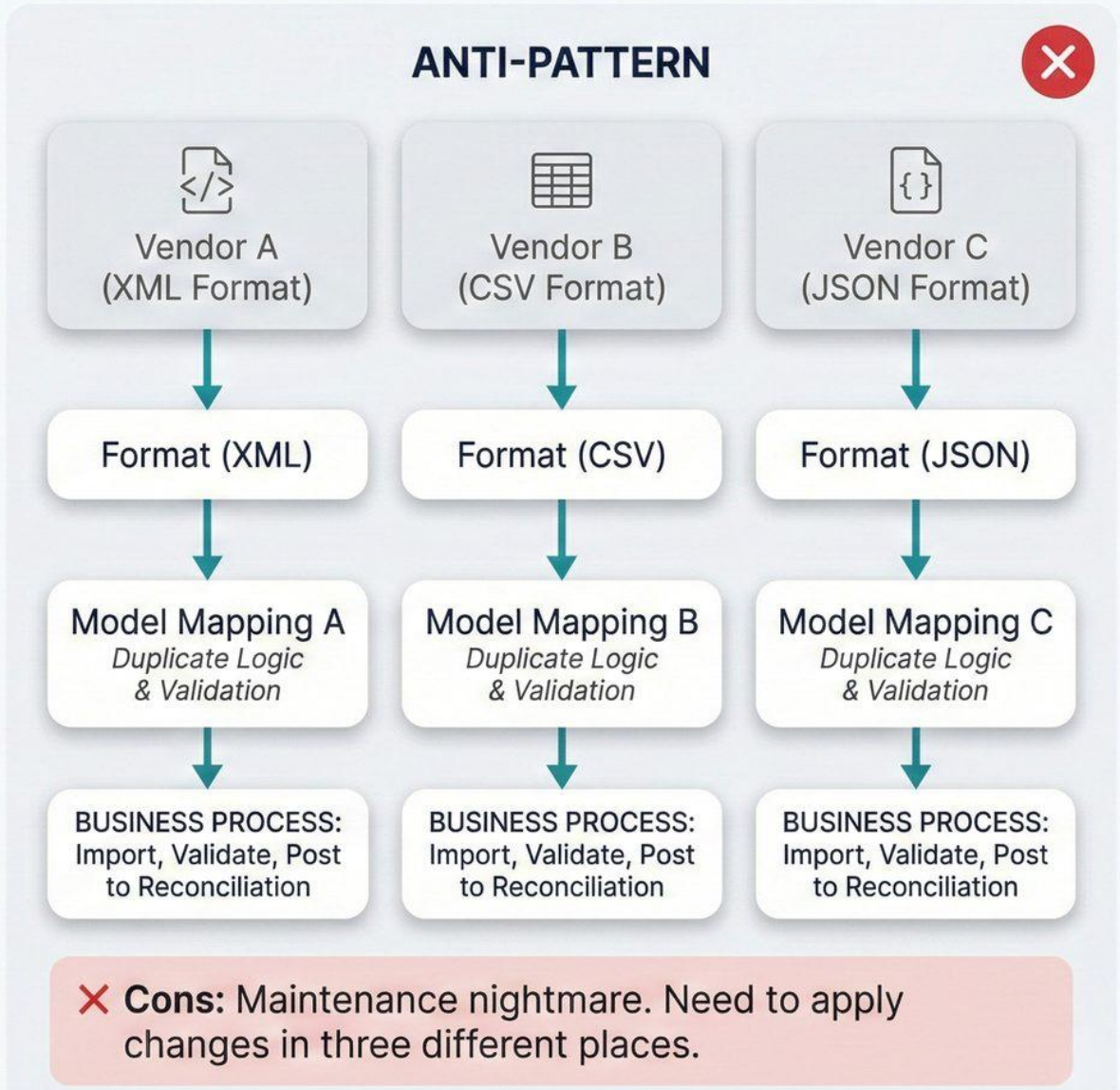
- Logs issue but continues import.
- Used for less severe issues.
- Example: Missing street address.

Validation keeps bad data out of your system.

truestack.dev

ER Reuse Pattern: Single Model Mapping





truestack.dev

Dynamics 365 F&O: BI Components

Flexible analytics options tailored to user needs and roles.

Tiles

Live KPIs pinned to workspaces – single metrics that are quick to scan.



Quick metric

Dashboards

Role-based homepages tailored to specific needs.

Sales Manager sees sales KPIs

Warehouse Manager sees inventory KPIs

Role dashboard

Self-Service BI

Users create their own analytics using Power BI without developer involvement.



Custom analysis

Choose components based on user need – quick metric, role dashboard, or custom analysis.

truestack.dev

Business Document Management & Electronic Reporting

Configurable business documents without code deployment.

1. Configure Electronic Reporting (ER) Format

Create an Electronic Reporting format.

Tag property set to 'Invoice' for identification.



2. Print Management Execution

User initiates print action.

Print Management Finds Format

Automatically selects tagged format for output.

Generates Document using 'Invoice' Tagged Format.

3. Configurable Document Templates



Completely Configurable. No Code Needed.

Just update the ER format to change the template.

Powerful customization without developer involvement.

Business Document Management uses Electronic Reporting tags to flexibly configure and generate business documents without requiring code deployments.

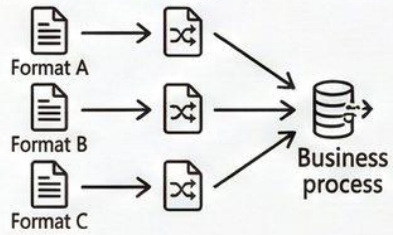
truestack.dev

Four critical mistakes to avoid

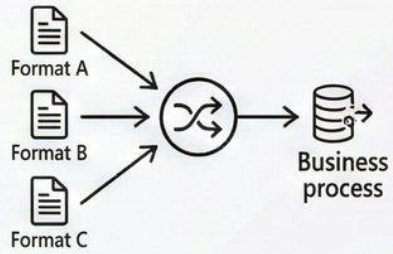
Mistake 1: Multiple model mappings

INCORRECT: Multiple model mappings for the same business process. Vendor statements in three formats, each with their own mapping.

(Maintenance nightmare)



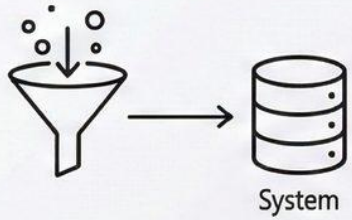
CORRECT: Use one mapping shared by all three formats.



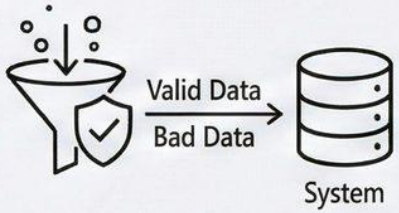
Mistake 2: No format validation

INCORRECT: Importing any data without checking it.

(Bad data pollutes your system)



CORRECT: Always validate format.



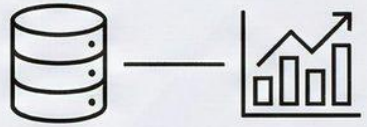
Mistake 3: TempDB in aggregate models

INCORRECT: TempDB tables in aggregate models.

(Session-temporary data can't support persistent analytics)



CORRECT: Use fact tables.

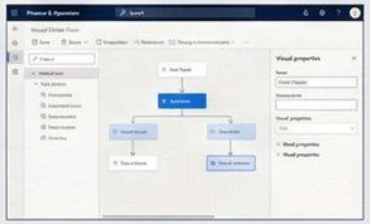


Mistake 4: Manually coding ER formats

INCORRECT: Manually coding ER formats (XML structures).



CORRECT: Use the visual and powerful ER designer.



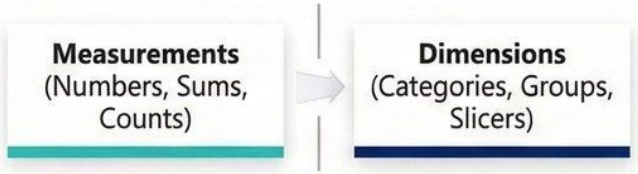
Avoid these common pitfalls to ensure maintainable, reliable, and performant Dynamics 365 Finance & Operations implementations.

truestack.dev

Five best practices for you

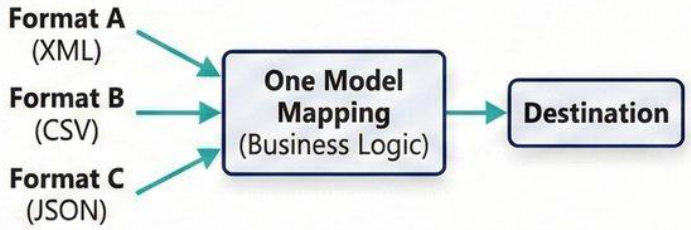
Practice one - understand measurements and dimensions

Measurements are numbers to display, dimensions are categories to group by.



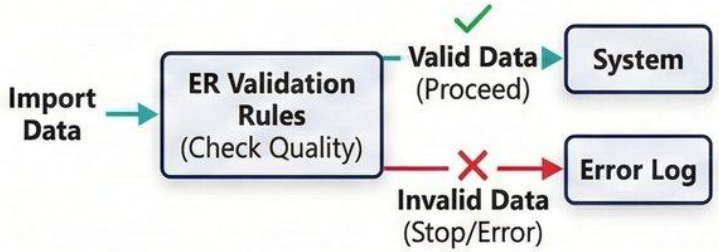
Practice two - reuse model mappings

Different external formats, same business logic. One mapping, multiple formats.



Practice three - validate all imports using ER validation

Protect your data quality.



Practice four - deploy aggregate models to Entity Store

That's where your KPIs get their data from.



Practice five - use tags for Print Management

Bind ER formats to document types.



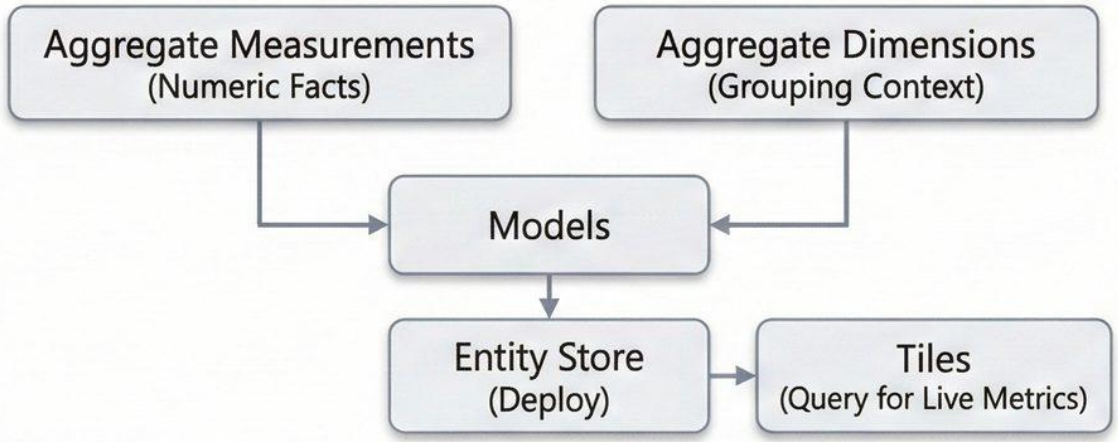
**Binds format to document type*

These professional practices ensure quality analytics.

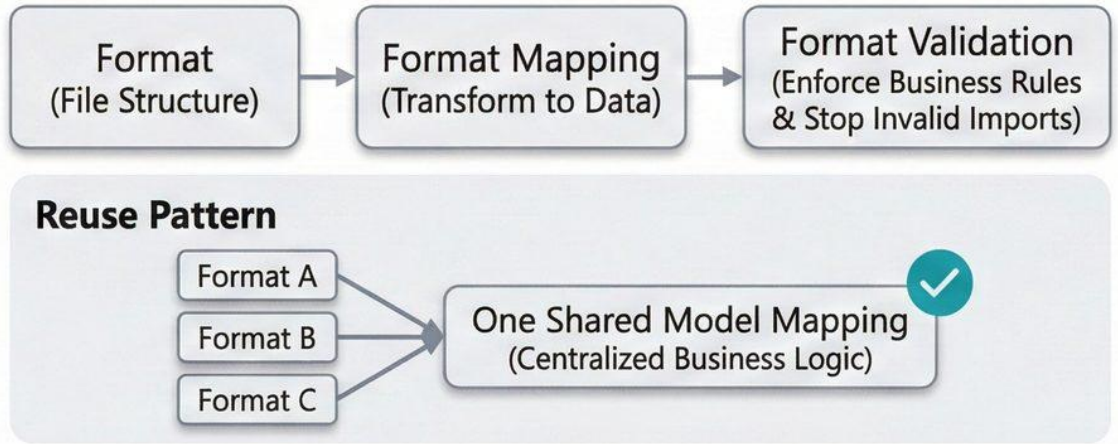
truestack.dev

Recap Topic 7.2: Analytics, Reporting, and BI Architecture

KPI Architecture



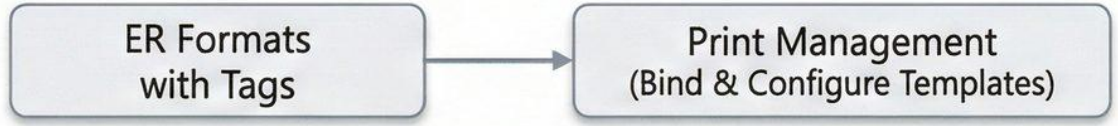
Electronic Reporting (ER)



BI Components

Tiles	Dashboards	Self-Service BI
Glanceable KPIs	Role-Based Homepages	User-Authored Analytics

Business Document Management (BDM)



Best Practices Summary

- ✓ Understand Measurements vs. Dimensions
- ✓ Reuse Model Mappings
- ✓ Validate All Imports
- ✓ Deploy to Entity Store
- ✓ Use Tags for BDM

Next Step: Topic 8.1 - Security Architecture (Roles, Permissions, Data Access Control).

truestack.dev

Topic 8.1 - Security Architecture

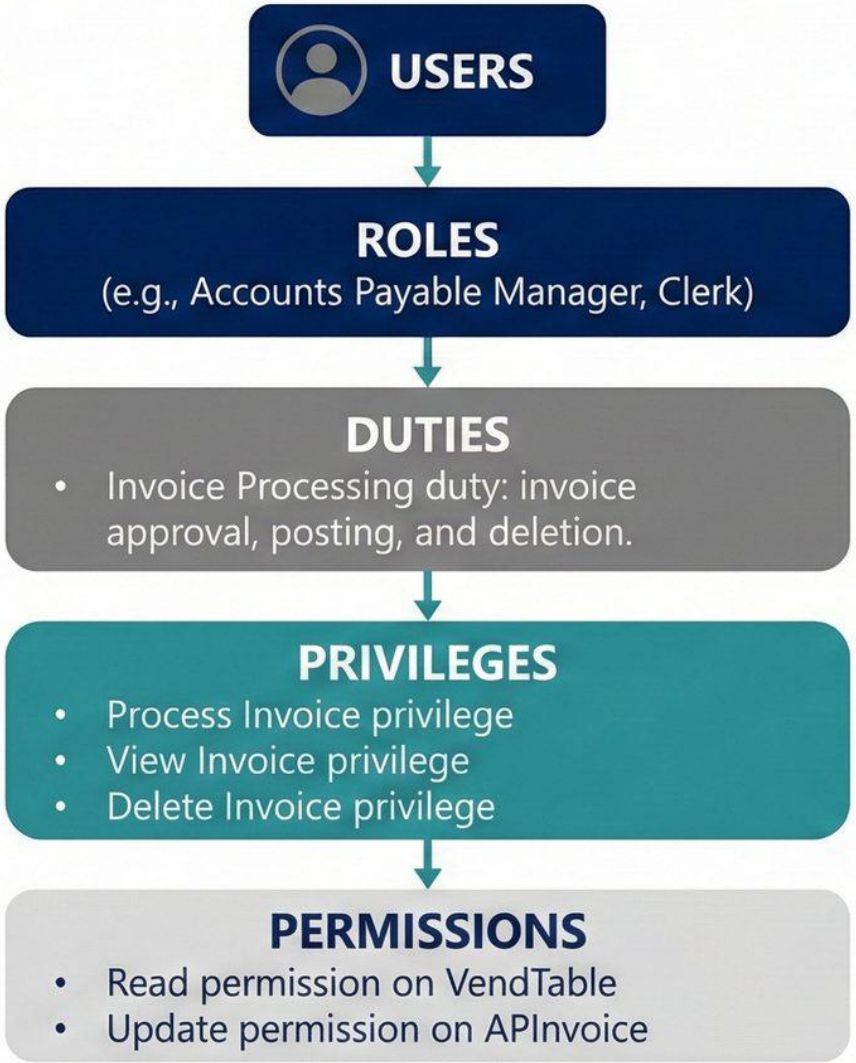
Layered security, data protection, and compliance in D365.



Master these patterns to build secure systems that protect data while enabling users to do their jobs.

truestack.dev

Dynamics 365 Finance & Operations: Security Hierarchy



HIERARCHY BENEFITS

- Ensures organization and maintainability.
- Changing a **single privilege** affects all associated users automatically.
- **Roles** are the least granular element assigned to users.
- **Duties** group related responsibilities.
- **Privileges** define specific tasks.
- **Permissions** control table and field operations.

Security follows a structured hierarchy for efficient management and granular access control.

truestack.dev

Security Components & Their Purpose

ROLE



User-facing assignment.

About user assignment.

DUTY



Groups related privileges.

About grouping related things.

PRIVILEGE



Grants access to specific operations like forms or reports.

About operations.

PERMISSION



Controls table and field CRUD operations.

About data.

Understanding this separation is critical to designing security that’s maintainable.

truestack.dev

Comprehensive Security Frameworks Working Together

Layered approach combining Role-Based, Extensible Data, and Table Permissions security.

1. Role-Based Security

Controls who can use which forms and operations.

Accounts Payable Clerk

- ✓ Open Invoice Processing form.

Finance Manager

- ✓ Run financial reports.

2. Extensible Data Security (XDS)

Filters data at runtime based on policies.

Accounts Payable Clerk

- ✓ See invoices for their vendor group only.
- ✗ See all vendors' invoices.

3. Table Permissions Framework (TPF)

Controls CRUD (Create, Read, Update, Delete) on tables and fields.

Accounts Payable Clerk

- ✓ Read invoices.
- ✗ Delete invoices.

All three layers together create comprehensive security.

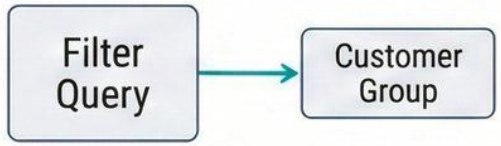
truestack.dev

Extensible Data Security (XDS) Overview

1. Define Filter Logic

Create a query defining the filter criteria.

Example: "Show only customers in my group".

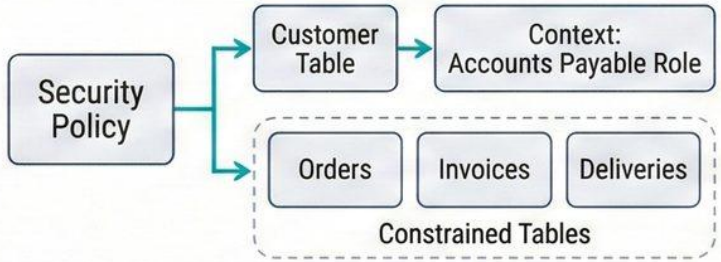


2. Configure Security Policy & Context

Create a security policy linking the query to the primary table (e.g., Customer table).

Define the context (e.g., apply to "Accounts Payable role").

Add constrained tables for related data (e.g., orders, invoices, deliveries).



3. Runtime Outcome (Transparent Filtering)

Transparent filtering occurs at runtime.

Accounts Payable users only see their customer group across all forms and reports.



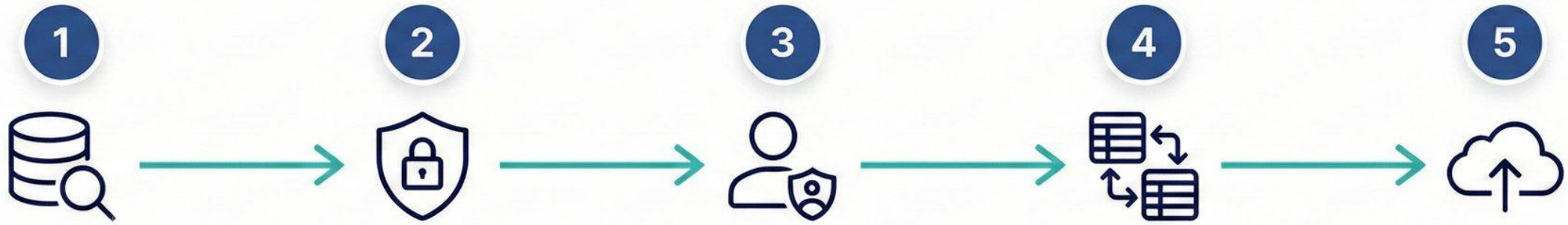
Why use XDS?

Powerful for multi-tenant scenarios or department-specific data access.

XDS provides powerful, transparent **row-level filtering** at runtime for securing data based on user context.

truestack.dev

Building an XDS Policy Step by Step



1 Create Query with Filter

CustomerTable WHERE CustomerGroup equals the user's customer group.

2 Create Security Policy

Link the policy to the CustomerTable.

3 Define Context

Apply it to Accounts Payable Clerk role.

4 Add Constrained Tables

Sales orders, invoices, and other related tables are constrained by the same filter.

5 Deploy It

The policy is now active.



Result: Users see only their customer group everywhere. Transparent, powerful filtering.

truestack.dev

Securing Forms with Differentiated Access (CashDisc Example)

MANAGER ROLE
Needs full access including delete.

CLERK ROLE
Needs view-only access.

1. **Verify Manager Access**
   Use Security Diagnostics to confirm existing configuration.
2. **Create "View" Privilege**
   New privilege with View access to the CashDisc form.
3. **Create Duty Extension**
   Contain the new privilege in a duty extension.
4. **Assign to Clerk Role**
   Assign the duty extension to the Clerk role.

FINAL OUTCOME: Managers retain delete access; **Clerks** can only view. This follows the least privilege principle.

truestack.dev

Four Critical Security Mistakes to Avoid

MISTAKE (INCORRECT)	CORRECTION (BEST PRACTICE)
Modifying standard roles Changing Microsoft roles creates conflicts during updates, preventing upgrades.	**Use role extensions** Utilize role extensions to apply changes without affecting the standard definition.
Using XDS for UI access XDS is designed for row-level data filtering, not controlling UI element access. Breaking the model.	**Use role-based security for UI** Always rely on the standard role-based security framework for managing UI access.
Adding menu items directly to roles Bypassing the privilege and duty structure causes maintenance issues and breaks hierarchy.	**Use privileges and duties** Maintain the structured hierarchy by assigning menu items to privileges, then duties, then roles.
Granting table access directly to users Directly granting table access violates the security model and breaks architecture.	**Access flows through privileges** Ensure all table access is granted via permissions within privileges, adhering to the model.

Adhering to the layered security model ensures maintainability, compliance, and seamless upgradability in Dynamics 365 Finance & Operations.

truestack.dev

Securing Reports

1.

1. Create Privilege



Contain the report's output menu item.

2.

2. Create Duty Extension



Contain the privilege.

3.

3. Assign Duty



Assign to appropriate role (e.g., Accounts Payable Manager).

4.

4. Secure Outcome



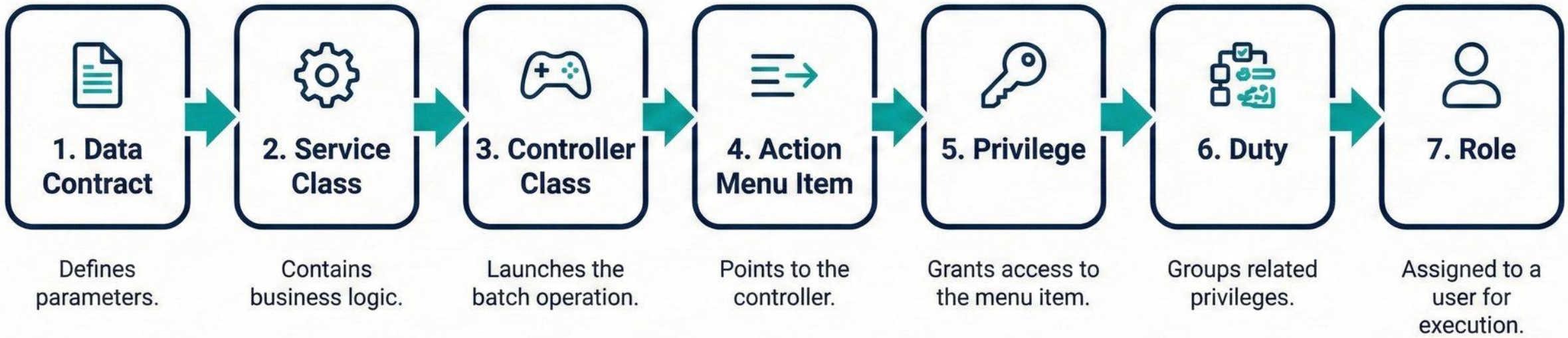
Only users with that role can run report. Non-managers access denied.

All objects stay in the required model.

truestack.dev

Batch Job Security and Execution Flow

Securing batch jobs using the role-based security hierarchy.



Key Takeaway: Batch jobs are secured like other operations, using the standard privilege and duty hierarchy.

truestack.dev

Real-World Example: Recurring Batch Jobs & Security

Job One: End-of-Month Invoice Updates



Schedule: Last Day of Month

Security: Accounts Payable Manager Privilege

Job Two: Update Purchase Lines based on Modified Discounts



Schedule: Weekly

Security: Purchasing Manager Privilege

Shared Security Architecture: Privilege → Duty → Role



Different users execute different jobs based on their roles.

This is professional recurring automation with proper security.

truestack.dev

Saved Views: Enhancing Usability and Productivity

Standard functionality for personalized, instant filtering

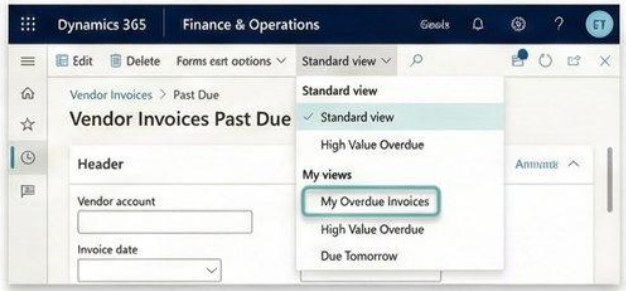
1. Apply Filters & Define

- Users apply filters to forms (e.g., Vendor, Due Date, Amount).



2. Save as Named View

- Save current filter criteria as a unique, named view.



3. Instant Application

- One click applies the saved filter instantly next time.



Key Benefits & Functionality

- Standard D365 feature – No custom code required.
- Dramatically improves productivity by eliminating manual filter recreation.

Saved Views allow users to personalize their experience for faster data access and greater efficiency.

trustack.dev

Security Diagnostics: Your Troubleshooting Tool

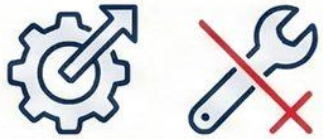
Determine User Access Levels to Specific Forms



truestack.dev

FIVE SECURITY BEST PRACTICES

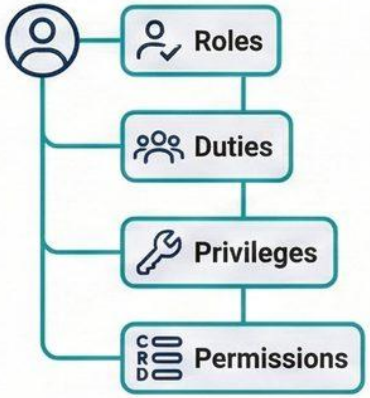
Practice 1: Extend, Don't Modify



Always extend standard roles, duties, and privileges – never modify them directly.

Why? Modifications break upgrades, but extensions are safe.

Practice 2: Use the Hierarchy Properly



Roles get assigned to users, duties group privileges, privileges grant operations, and permissions control CRUD.

Understand this flow.

Practice 3: Follow Least Privilege Principle

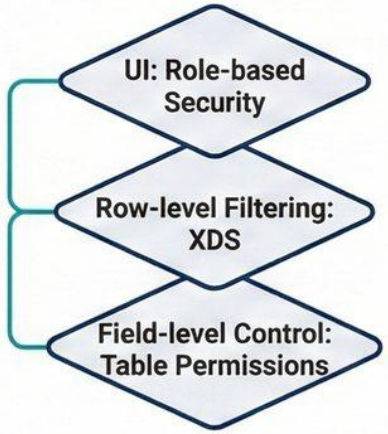


Grant only the access that's actually necessary.

Clerk Role	Clerk Role
View Only	Delete Access

A clerk doesn't need delete access; restrict to view only.

Practice 4: Layer Your Security



Use Role-based for UI, XDS for row filtering, and Table Permissions for field control.

All three working together.

Practice 5: Test Access Regularly



Use Security Diagnostics to verify your configuration.

Verify configuration is actually correct.

truestack.dev

Recap: Topic 8.1 Security Hierarchy & Frameworks

Review of security models, implementation, and best practices.

Security Hierarchy



Three Security Frameworks

Framework	Purpose	Scope	Notes
Role-Based Security	Controls UI & operations	UI and business logic	Built on roles → duties → privileges → permissions
Extensible Data Security (XDS)	Filters data at runtime	Row-level data	Uses policies, queries, contexts
Table Permissions	Controls CRUD operations	Table/field-level	Enforced at privilege level

Use all three for comprehensive security

Implementation & Best Practices

XDS Implementation Sequence

1. **Create Query** — Define filters
2. **Create Policy** — Link query
3. **Define Context** — Apply filter
4. **Add Constrained Tables** — Row-level filtering

Extending Security

✓ **Extend standard roles, duties, privileges**
Preserves upgradeability

✗ **Never modify directly**

Applying Security & Looking Ahead

Securing Forms & Reports

Managers
Have full access

Clerks
Restricted to view-only

Differentiated access via privileges & duties.
Consistent pattern for reports & batch jobs.

Best Practices Summary

- ✓ Extend don't modify
- ✓ Use hierarchy properly
- ✓ Follow least privilege
- ✓ Layer security
- ✓ Test with diagnostics

NEXT: Topic 8.2 Testing and Workflow

Ensuring quality and handling approvals

Understanding the layered security architecture is crucial before moving on to quality assurance and approval processes.

truestack.dev

Topic 8.2: Testing and Workflow

Securing Quality Assurance through Comprehensive Testing and Optimized Processes

Task Recorder



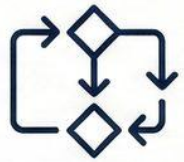
Captures business processes for both documentation and testing.

SysTest Framework



Provides unit testing capabilities.

Workflow



Routes documents through your approval logic.

Performance Tools



Help you diagnose issues.

Master these patterns to build professional systems with proper testing, clear workflows, and optimized performance.

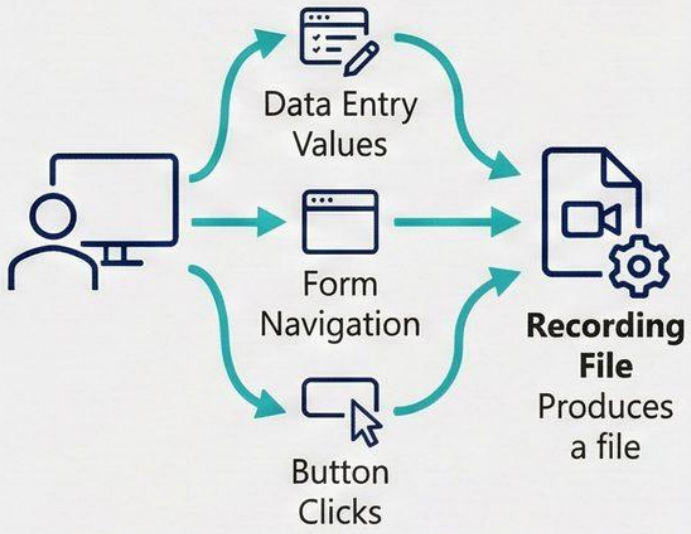
truestack.dev

Task Recorder Modes

Capturing, guiding, and automating business processes in D365 F&O

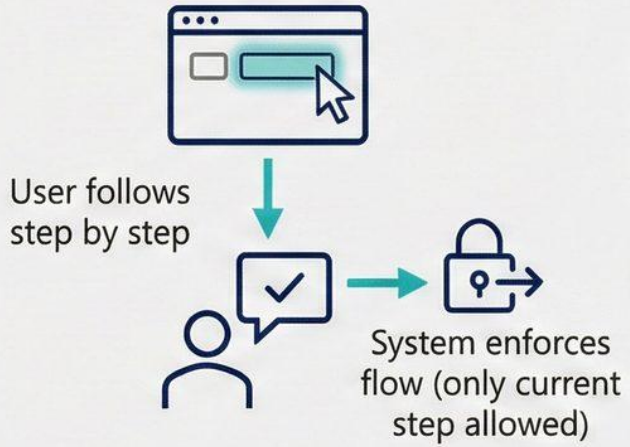
Recording Mode

Captures the business process as steps are performed.



Guide Mode

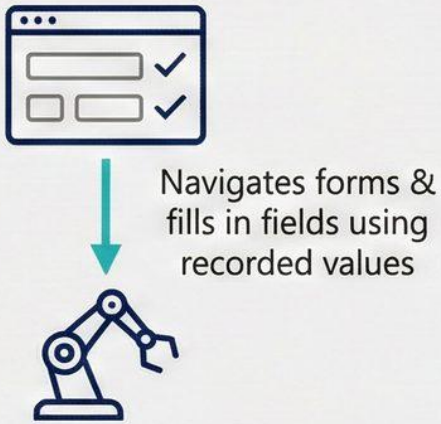
Plays back the recording interactively.



Great for training purposes.

Playback Mode

Replays all steps automatically.



Perfect for regression testing or automated demos.

Three distinct modes for documentation, training, and automation.

truestack.dev

The Task Recorder Workflow

1. Recording Process

1 REC

Start Recording

Open Task Recorder and begin capturing steps.

2

Perform Business Process

Execute actions like creating a vendor, entering details, and saving.

3

Stop Recording

Complete the process; system saves all steps.

2. Multiple Deliverables

Export to Word

Create comprehensive documentation.

→ Documentation

Export to XML

Create test automation file for RSAT.

→ Test Automation Files

Create Shared Task Guide

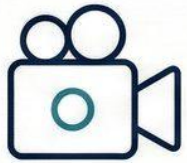
Develop interactive training materials.

→ Training Materials

One recording efficiently produces multiple valuable deliverables for documentation, testing, and training.

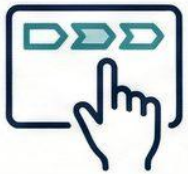
truestack.dev

Task Recorder Capabilities



Capture Business Process

Through the UI



Create Interactive Guides

Teach users step-by-step



Replay Recordings Automatically

For demonstrations or initial testing



Export to Word

For process documentation & share with non-technical users



Export as XML for RSAT

Automated regression testing tool

One tool, many uses. Powerful for documentation, training, and automation.

truestack.dev

SysTest Framework: Professional Unit Testing in Dynamics 365

Discover, isolate, and analyze unit tests with metadata-driven capabilities.

Core Capabilities

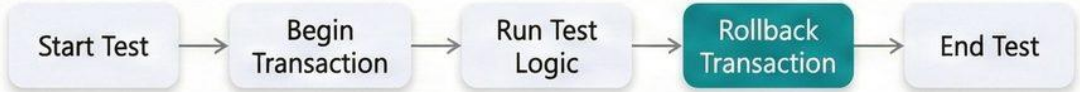
Discovery & Execution

Test classes contain test methods marked with [SysTestMethodAttribute]. The framework automatically discovers and runs these marked methods.

```
[SysTestMethod]
public void testCalculation() { … }
```

Transaction Isolation

Each test runs within a transaction that **automatically rolls back** upon completion, ensuring clean isolation with no side effects between tests.



Metadata & Reporting

Metadata & Categorization

The framework maps code coverage and categorizes tests using metadata such as Owners, Priorities, and Properties for better organization and filtering.

Attribute	Example Value
Owner	'John Doe'
Priority	'High'
Property	'Feature: Calculations'

Result Analysis

Test results clearly show which tests **passed** and which failed, providing actionable feedback for developers.

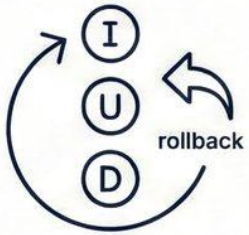


SysTest is a professional unit testing framework designed for reliability, isolation, and effective test management in Dynamics 365.

truestack.dev

Transaction Isolation Modes

AutoRollback

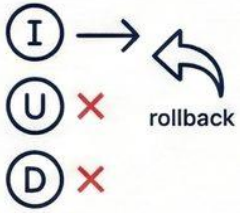


Modern & Clean Approach. Perfect Isolation.

All database changes (Inserts, Updates, Deletes) are rolled back after test completes.

Test runs, rolls back, next test starts fresh.

LegacyRollback

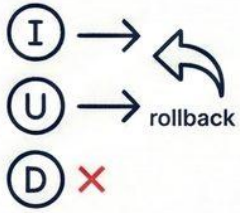


Backward Compatibility.

Only rolls back inserts.

Useful for older code.

LegacyRollbackWith UpdateTracking

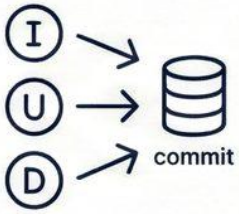


Rarely Used.

Tracks updates, but rarely used.

Legacy scenarios.

None



No Rollback.

No rollback at all. Changes persist.

Good for integration tests needing committed data.

Choose the isolation mode based on your specific testing needs and the required data persistence.

truestack.dev

Key SysTest Attributes for Test Management

Purpose and usage for discovery, categorization, and quality gates.

This one's required.

SysTestMethodAttribute

Marks a method as a test for discovery. This attribute is required.

SysTestInactiveTestAttribute

Disables a test without removing it; useful for work in progress.

SysTestOwnerAttribute

Assigns ownership to a specific individual, for example, 'Jane owns this test'.

SysTestPriorityAttribute

Sets priority so you can run high-priority tests first.

SysTestPropertyAttribute

Adds custom metadata to the test method.

SysTestTargetAttribute

Links the test to the specific artifact it is actually testing.

SysTestCheckInTestAttribute

Enforces gated check-in, acting as a quality gate; test must pass before code check-in.

Each attribute serves a specific purpose for structured unit testing within the SysTest framework.

truestack.dev

Real Unit Test Example

Professional testing for code quality

Test Structure & Attributes

- Class: VendorServiceTest
- Method: TestCreateVendor
- Attributes for Discovery, Linkage, and Gated Check-in

Execution & Isolation

- AutoRollback mode for clean isolation
- Test creates a vendor instance
- Assertions verify database existence and data accuracy

Outcome

- Passes if all assertions pass; Fails if any assertion fails

SysTestMethod: Marks for discovery

```
[SysTestMethodAttribute]
[SysTestTargetAttribute(typeof(VendorService))]
[SysTestCheckInTestAttribute]
public void TestCreateVendor()
{
    using (var transaction = new SysTestTransaction(SysTestTransactionMode.AutoRollback))
    {
        // Arrange
        var service = new VendorService();
        string expectedAccountNumber = "V-1001";

        // Act
        var vendor = service.CreateVendor(expectedAccountNumber, "Test Vendor Name");

        // Assert
        Assert.IsNotNull(vendor, "Vendor should be created.");
        Assert.IsTrue(vendor.ExistsInDatabase(), "Vendor must exist in database.");
        Assert.AreEqual(expectedAccountNumber, vendor.AccountNumber, "Account number
                does not match.");
    }
}
```

SysTestTarget: Links to VendorService class

SysTestCheckInTest: Gates check-in (must pass)

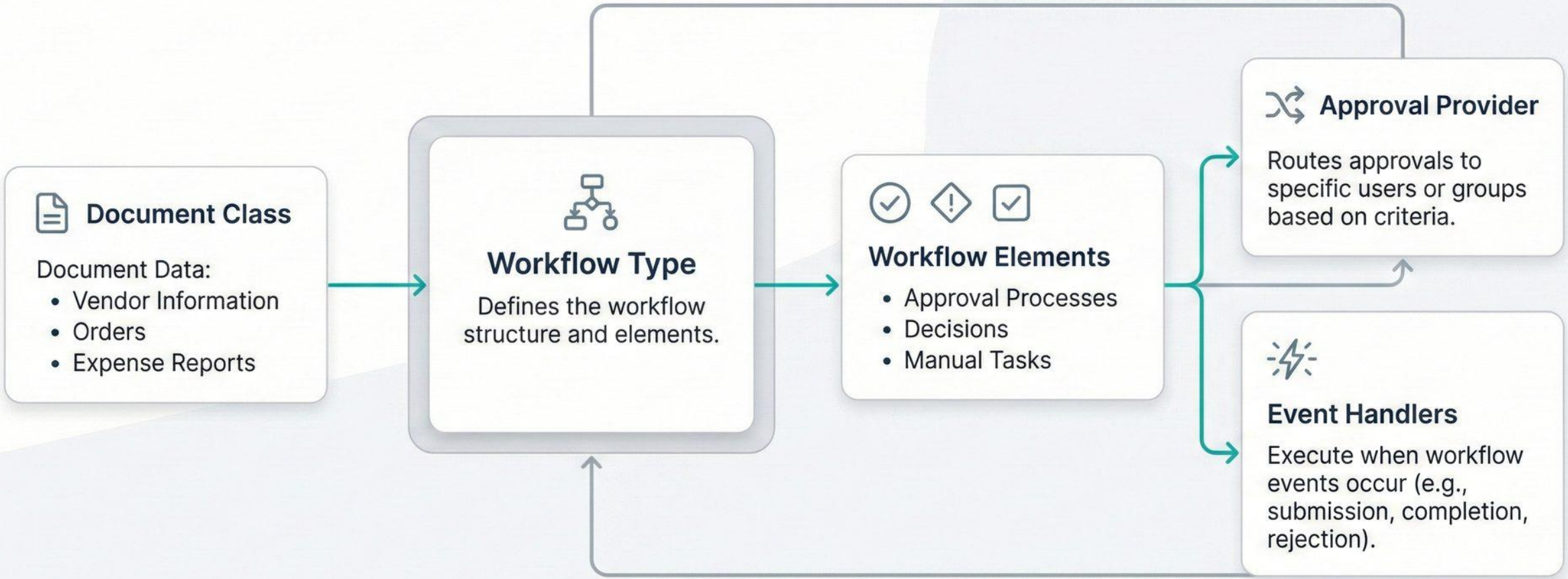
AutoRollback mode for clean isolation

If all pass -> Test Passes; If any fail -> Test Fails

This professional unit testing approach ensures your code meets quality standards through rigorous, isolated verification.

truestack.dev

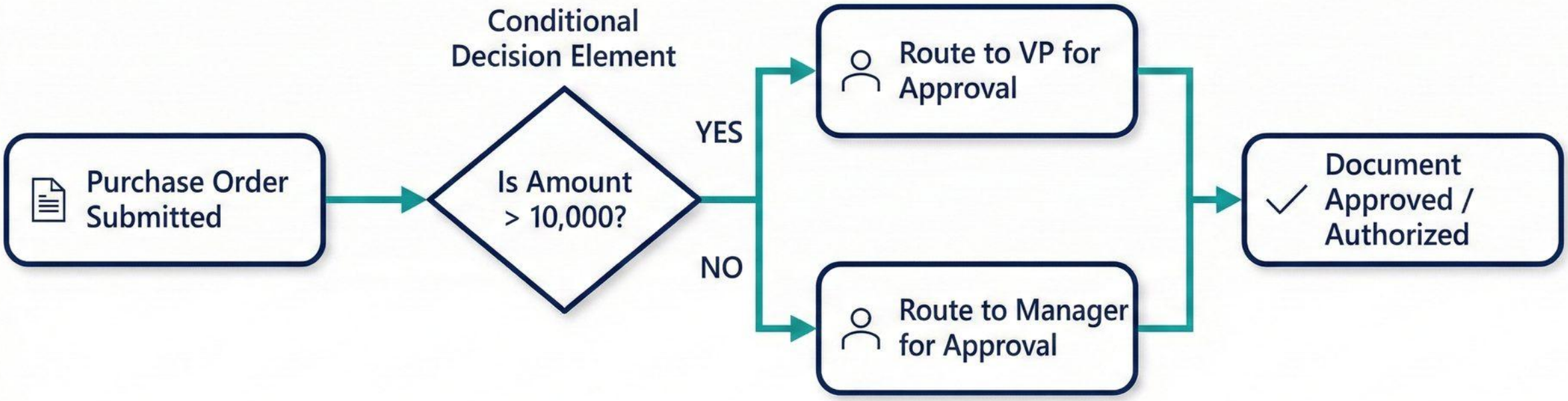
Workflow Architecture Components



Together, these components create a complete approval process with proper routing and notifications.

truestack.dev

Conditional Approval Routing



The conditional decision element automatically evaluates business rules to ensure documents flow to the correct approver without user intervention.

truestack.dev

Workflow Elements and Their Use Cases

The Five Workflow Elements

Approval process
Routes to users for approval or rejection.

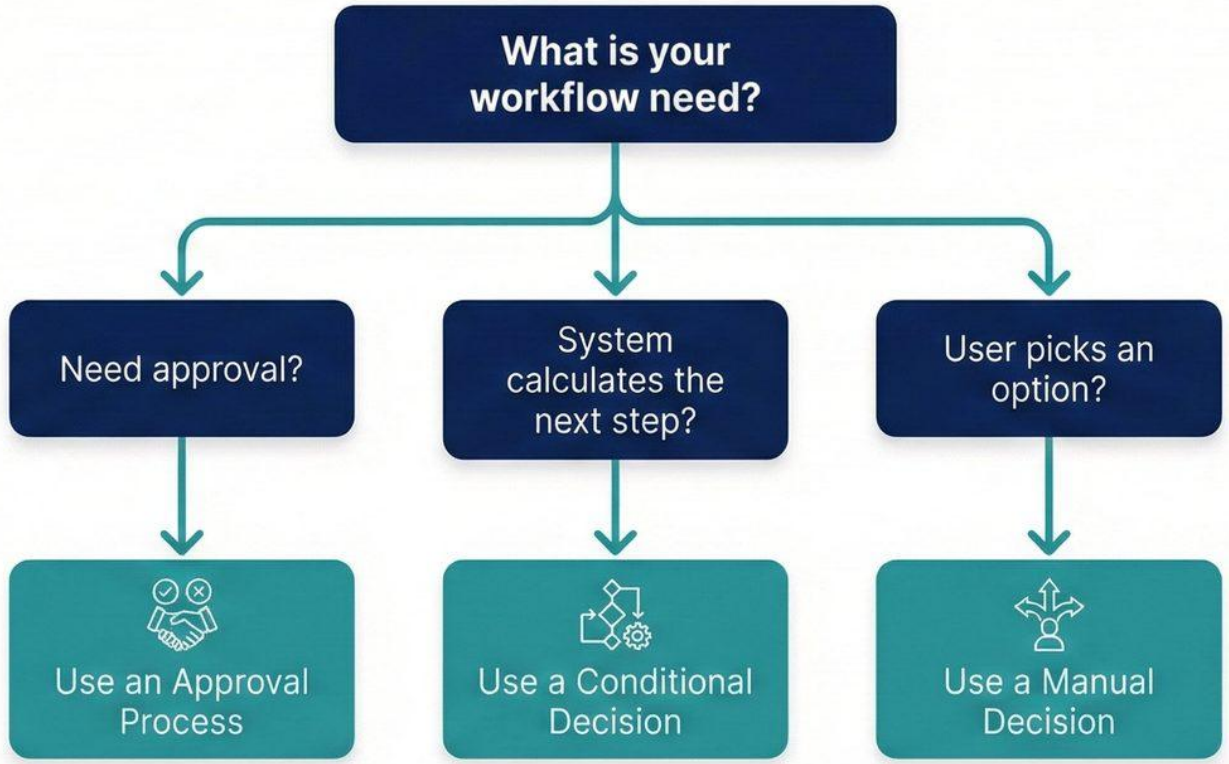
Manual decision
Lets the user pick which branch to take.

Conditional decision
Auto-evaluates business logic.

Automated task
Executes system logic.

Manual task
Assigns work to a user.

Choosing the Right Element



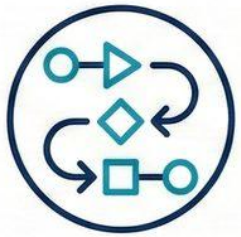
Choose workflow elements based on whether you need user approval, system calculation, or manual input.

truestack.dev

Workflow Association Types & Scope

Control the scope of workflow application based on organizational requirements.

Association Concept



Determines which legal entities a workflow definition applies to.

Company Association



Scope: Applies to **one legal entity**.

Use Case: Useful when different companies have different rules.

Best for: Single company rules.

Global Association



Scope: Applies to **all legal entities**.

Use Case: Useful for shared corporate rules.

Best for: Corporate-wide rules.

Choose your association type based on whether rules are entity-specific (Company) or corporate-wide (Global).

truestack.dev

Custom Workflow Development Sequence



1. **Develop Components**
   Document class, event handlers, and approval provider.
2. **Create Workflow Type**
   In AOT, bind your classes.
3. **Configure Workflow Steps**
   In designer - add approval processes, decisions, and tasks.
4. **Enable**
   On the source table and in the client.
5. **Deploy and Test**
   Finalize and verify functionality.

CRITICAL WARNING: If you skip a step, the designer can’t find the compiled classes and you’ll get errors. Follow this sequence exactly.

Strict adherence to this five-step sequence is essential for successful custom workflow creation.

truestack.dev

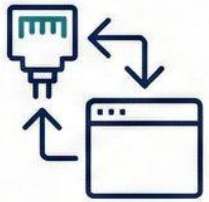
Performance Monitoring Tools

Comprehensive Diagnostics for Dynamics 365 Finance & Operations



LCS Environment Monitoring

- System-wide health view
- CPU, memory usage
- SQL performance metrics
- Batch job history



Fiddler

- Client-to-server HTTP traffic
- Form load time analysis
- Request latency measurement
- Payload sizes



Trace Parser

- Single-process details
- X++ call stack visualization
- SQL statements execution
- Duration breakdown



SQL Insights

- Database-level analysis
- Blocking & deadlocks
- Transaction locks
- Transaction waits



Activity Monitoring

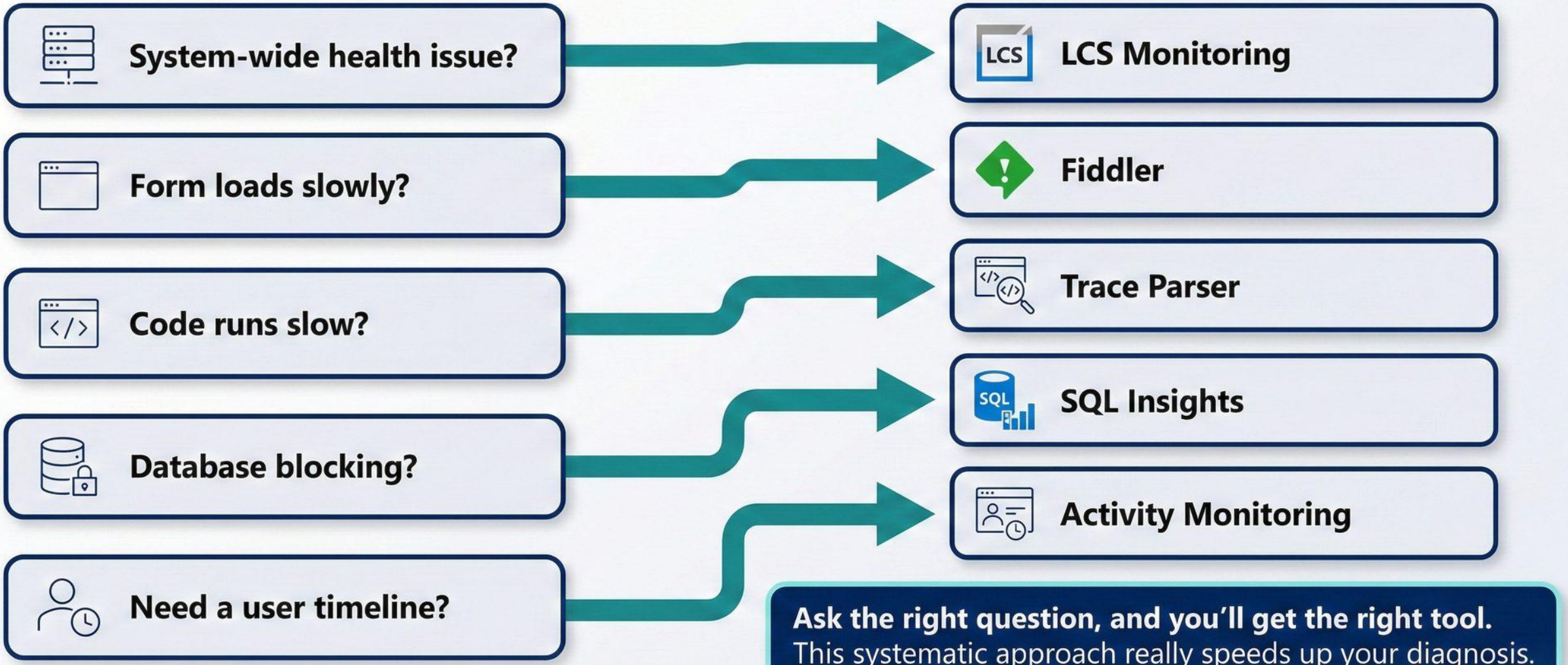
- User activity tracking
- Session counts
- User actions & timeline
- User-specific actions

Together, these tools cover full stack diagnostics.

truestack.dev

Performance Troubleshooting: A Decision Tree Approach

Match the Right Tool to the Right Problem for Faster Diagnosis



truestack.dev

Five Best Practices for D365 F&O Quality

Practice One: Use Task Recorder



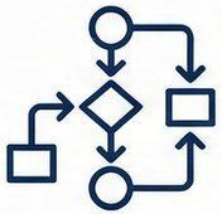
It documents processes and enables testing and training.

Practice Two: Write unit tests with SysTest



You get transaction isolation and code coverage tracking.

Practice Three: Design workflows carefully



With conditional logic and proper approval routing

Practice Four: Use the correct performance tools



And diagnose problems systematically.

Practice Five: Test performance regularly



So you catch issues before production.

Discipline ensures quality.

truestack.dev

Topic 8.2: Final Recap

Task Recorder

- Documents business processes
- Interactive training (Guide mode)
- Automated testing (Playback mode)

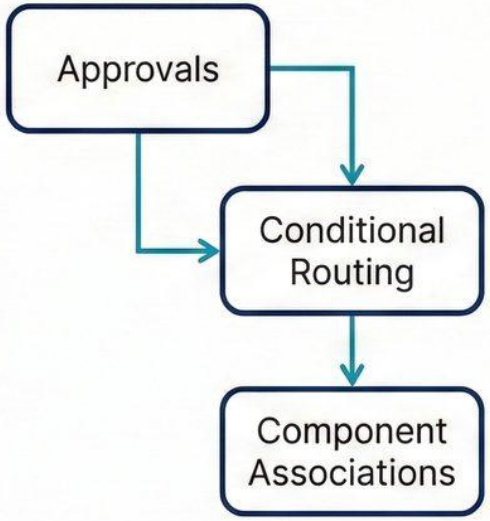
Supports guidance artifacts & automation assets.

SysTest Framework

- Enables unit testing
- Transaction isolation
- Structured attributes

Metadata-driven categorization & code coverage mapping.

Workflow Architecture



Enforces business document routing & approval logic.

Performance-Diagnostic Tools

- **LCS Monitoring**
  System-wide health metrics
- **Fiddler**
  Client-to-server HTTP traffic
- **Trace Parser**
  X++ & SQL trace analysis
- **SQL Insights**
  Query blocking & transaction locks
- **Activity Monitoring**
  User session & action analysis

Targets different layers for full-stack visibility.

Comprehensive tools and frameworks enable efficient testing, structured workflows, and full-stack performance diagnostics.

truestack.dev

Closing Summary

A comprehensive journey through Dynamics 365 Finance & Operations development

THE COMPLETE JOURNEY



Foundational Cloud Architecture

Platform, Models, & Deployment

X++ & Data Models

Syntax, EDTs, Tables, & OOP

Integrations & Patterns

OData, DMF, & Services

Security & Operations

Role-based Access & Workflows

Advanced Extensibility & Framework

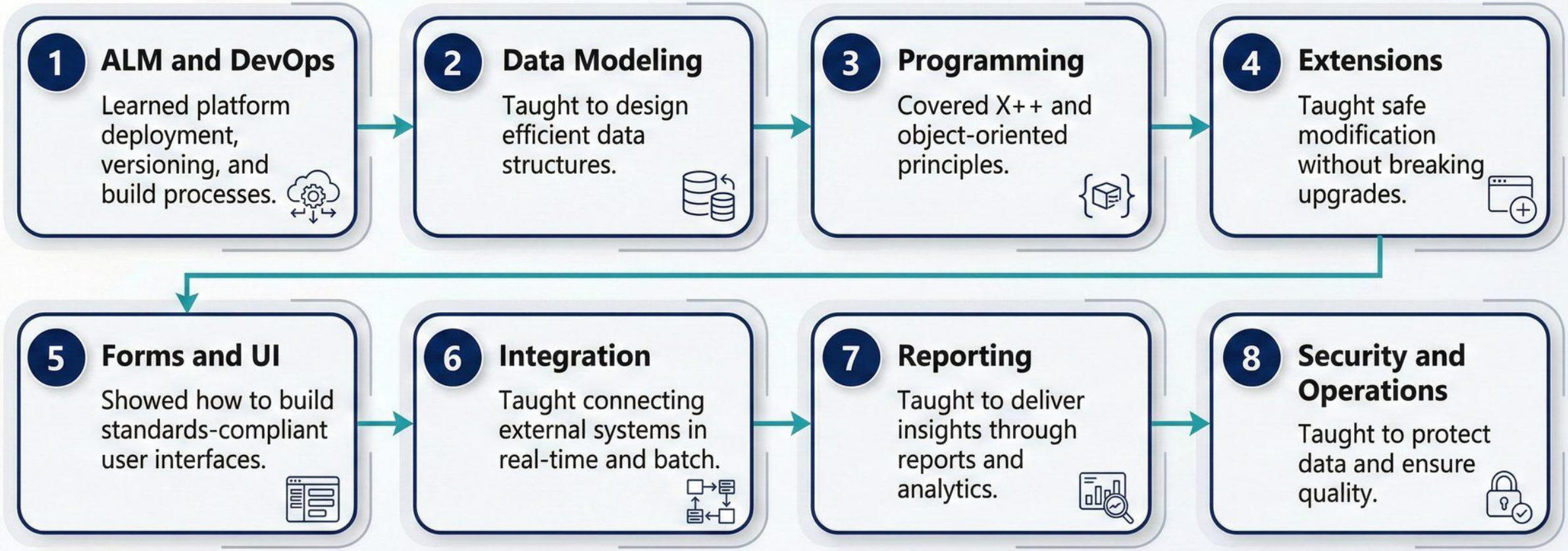
Extensions, Chain of Command, & Best Practices

This final topic brings it all together, helping you understand how these pieces form a complete professional development framework.

truestack.dev

Complete Development Curriculum: Dynamics 365 F&O

A Comprehensive Journey Across Eight Modules

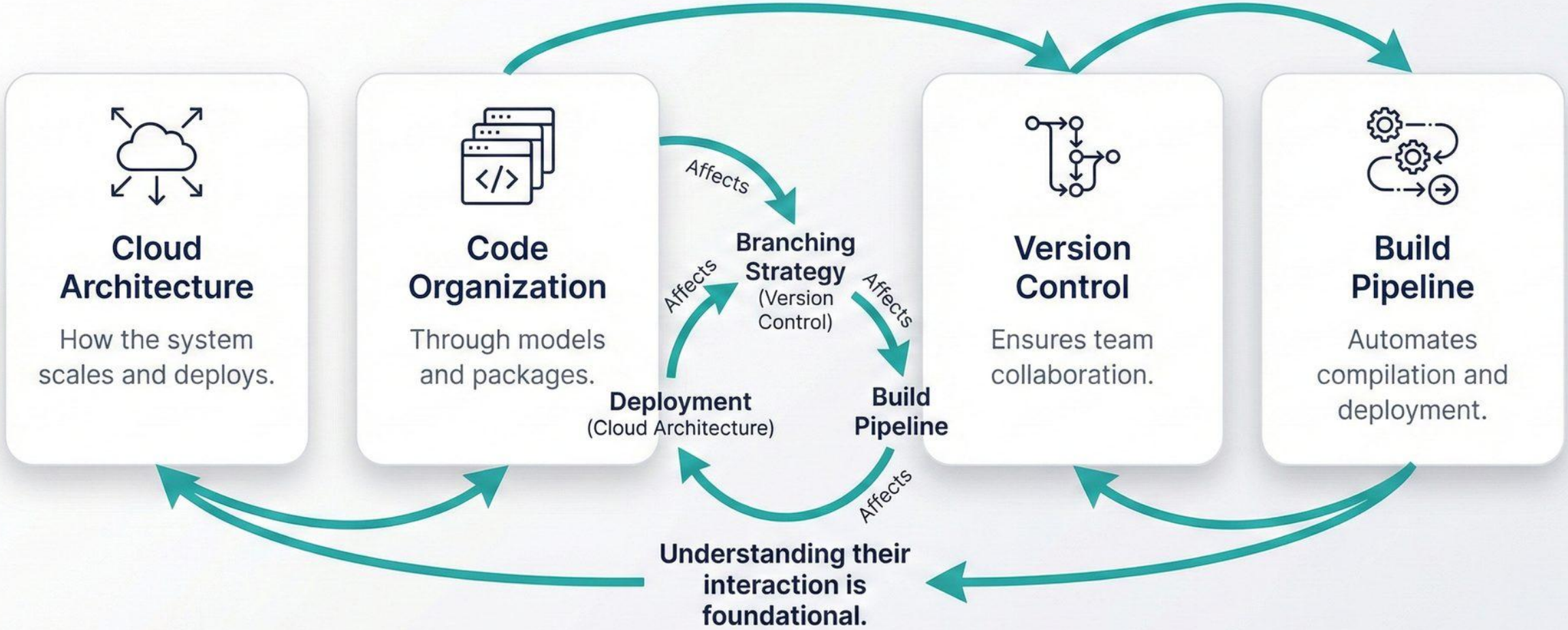


Together, this series provides a complete development curriculum for Microsoft Dynamics 365 Finance & Operations.

truestack.dev

The Platform Foundation Pillars

Interconnected elements for a scalable and collaborative development environment.



truestack.dev

The Core of Any Solution: Data and Code

Data Design



EDTs (Extended Data Types)
Enforce type safety.



Tables
Organize your data logically.



Relations
Connect entities efficiently.



Indexes
Optimize query performance.

Code Quality



Object-Oriented Principles
Create maintainable code.



Set-Based Operations
Handle data efficiently.



Exception Handling
Prevents surprises.



Naming Conventions
Make your code readable.

"When both your data and your code are excellent, the entire system performs well and scales reliably."

truestack.dev

Modern Extensibility Philosophy in Professional Development

NEVER MODIFY MICROSOFT'S STANDARD CODE (INCORRECT ✗)

Direct modification of platform code

Risks code conflicts during updates

Breaks upgrade path

ALWAYS EXTEND IT (CORRECT ✓)

Class Extensions (Chain of Command)
Adds behavior, wraps standard logic

Table Extensions
Adds new fields to existing tables

Event Handlers
Reacts to system events

THE RESULT: UPGRADE SAFETY

- Microsoft updates release without breaking your code
- Upgrade without fear
- Code is not affected by platform changes

MODERN SOFTWARE ENGINEERING BEST PRACTICES — SEPARATION OF CONCERNS — CLEAN DEPENDENCIES — UPGRADE SAFETY

The modern extensibility philosophy ensures maintainable, upgrade-safe solutions by extending platform capabilities without modifying standard code.

truestack.dev

Understanding User Experience Layers

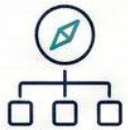
Forms



- ✓ Present data for entry and viewing
- ✓ Follow patterns: Simple List, Details Master, List Page
- ✓ Users recognize familiar structures

Intuitive data interaction.

Navigation



- ✓ Guides users to what they need
- ✓ Menu Items, Tiles, Links
- ✓ All organized logically

Effortless movement through the system.

Dashboards



- ✓ Provide executive overview
- ✓ Role-based dashboards
- ✓ Show relevant KPIs

Instant insight and performance tracking.

When all three layers are designed well, users have a professional, intuitive experience, spending less time searching and more time actually doing their job.

truestack.dev

Integration Architecture: The Four Pillars

Covering Every Integration Scenario



OData

Go-to for real-time data access. Moderate throughput for immediate immediate queries.



DMF (Data Management Framework)

Heavy lifter for bulk operations. High throughput for large volume import/export.



Custom Services

Complex business logic requiring real-time calculations.



Business Events

Asynchronous notifications for event-driven architecture.

Together, these four patterns cover literally every integration scenario you'll encounter: Real-time needs, batch processing, complex logic, and event handling.

truestack.dev

Analytics in Dynamics 365: Three Distinct Layers



3. Power BI (Self-Service Analytics)

User-created reports and dashboards. Self-service capabilities for exploring data.

Interactive, visual data storytelling.



2. Electronic Reporting & Aggregate Models

Powers Live KPIs and Configurable Documents. Users can customize templates themselves without developer help.

Real-time operational metrics & flexible output.



1. SSRS Reports (Foundation)

Detailed Business Analysis. Users can really drill down and investigate what's happening.

Historical, granular data view.

Complete Business Visibility

- ✓ Decision-makers get real-time KPIs.
- ✓ Analysts get detailed reports for deep dives.
- ✓ Power users get self-service capabilities.

It's all about making your data tell the story.

truestack.dev

D365 Security Architecture: A Layered Approach

Role-Based Security



Controls who can use which features.

- ✔ Accounts Payable Clerk: Cannot delete vendors (✘).
- ✔ Finance Manager: Can post to general ledger (✓).

Extensible Data Security (XDS)



Filters data at runtime.

Clerk only sees vendors in their specific group, not everyone.

Table Permissions



Controls CRUD operations on fields.

Some fields are read-only (✘), others you can update (✓).

DEFENSE-IN-DEPTH

Multiple layers protecting your system. Even if one layer is breached, others remain active for protection.

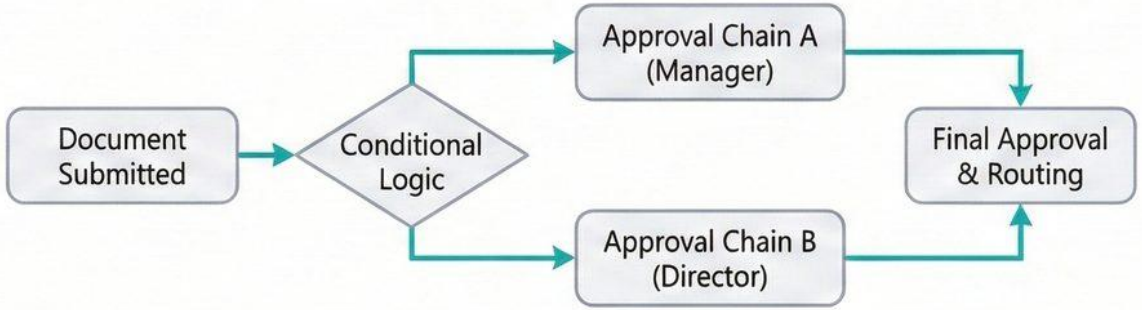
truestack.dev

Quality and Operations: Ensuring Production Excellence

Testing Strategy

- Unit Tests (Code Quality)
- Integration Tests (Feature Validation)
- Task Recorder (Documentation & Automation)

Workflow Automation



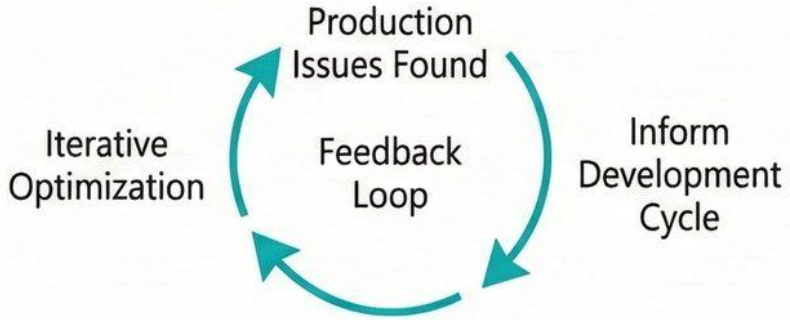
Performance Monitoring Tools

LCS
System Health & Telemetry

Fiddler
Network Traffic Analysis

SQL Insights
Database Performance & Queries

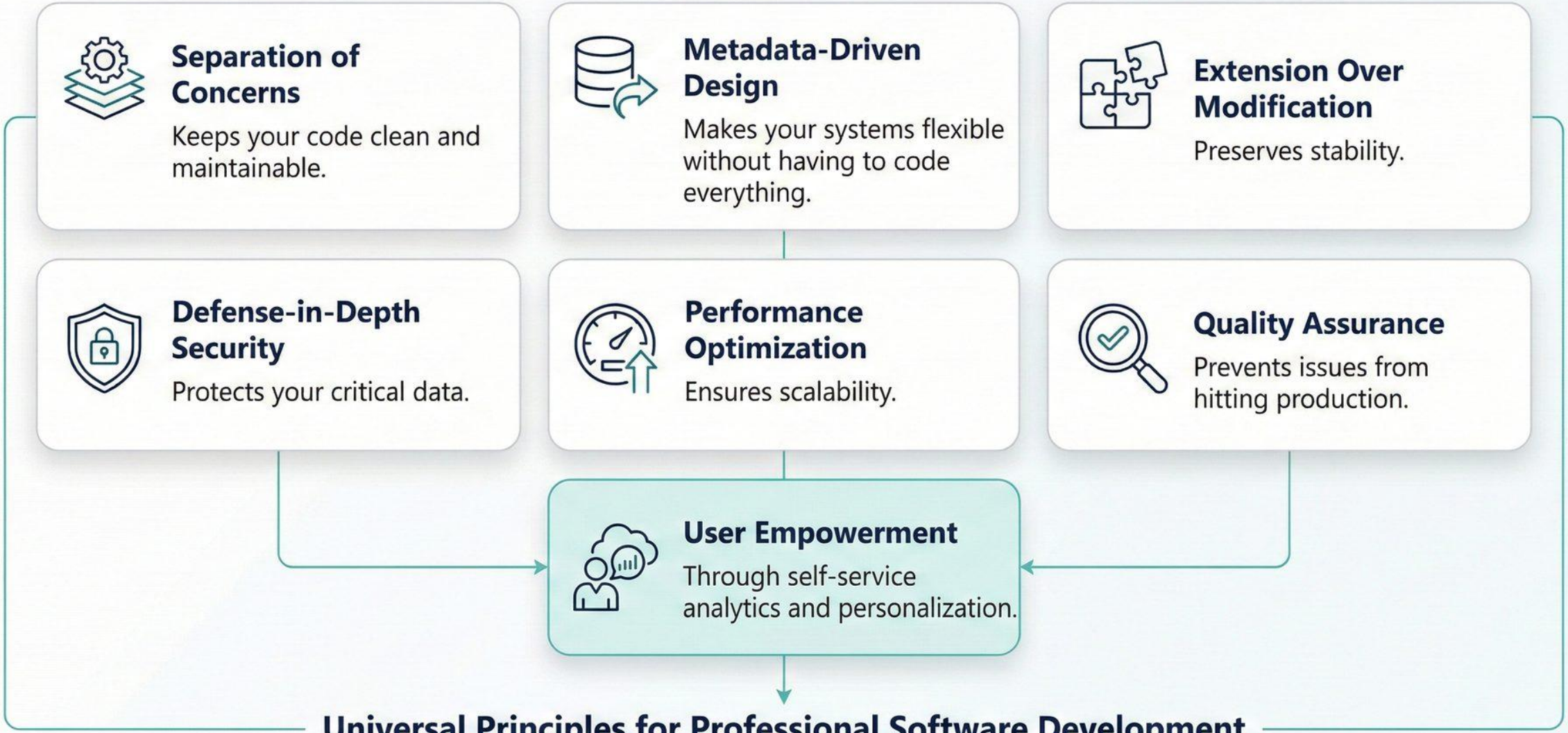
Continuous Improvement



Optimization is an iterative process, creating feedback loops where production insights inform the next development cycle.

truestack.dev

Modern Software Engineering Principles for Enterprise Applications



truestack.dev

Closing Notes: Avoiding Costly Mistakes

Common Pitfalls & Consequences	Preventive Solutions & Best Practices
❌ Modifying standard code (Breaks upgrades)	→ ✓ Extend instead of modifying
❌ Ignoring patterns (Inconsistent user experience)	→ ✓ Follow the patterns
❌ Poor performance (User frustration)	→ ✓ Optimize proactively
❌ Weak security (Breach risks)	→ ✓ Layer your security
❌ Not testing (Bugs slip into production)	→ ✓ Test continuously

This knowledge prevents costly mistakes that you might have otherwise made.

truestack.dev

Professional Development Takeaways



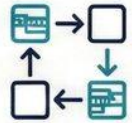
1. Extend Safely

Use proper extension mechanisms. Never modify standard code.



2. Follow Patterns

Whether for forms, navigation, or security. Patterns create consistency and predictability.



3. Think Integration

Understand when to use OData, DMF, custom services, and events.



4. Secure Thoroughly

Layer multiple security frameworks.



5. Test Rigorously

Use unit tests, integration tests, and performance tests.



6. Optimize Proactively

Monitor systems and improve continuously.

These six principles, applied consistently, create professional solutions.

truestack.dev

Closing Summary: A Comprehensive Journey

Dynamics 365 Finance & Operations Development



Complete Development Stack Knowledge

- From Cloud Architecture to Performance Diagnostics
- From Basic X++ to Complex Integrations
- From Role-Based Security to Business Event Handling

You now have the Knowledge Framework, Patterns, Best Practices, and Decision-Making Criteria to build production-quality solutions (for MB 500 certification).

truestack.dev