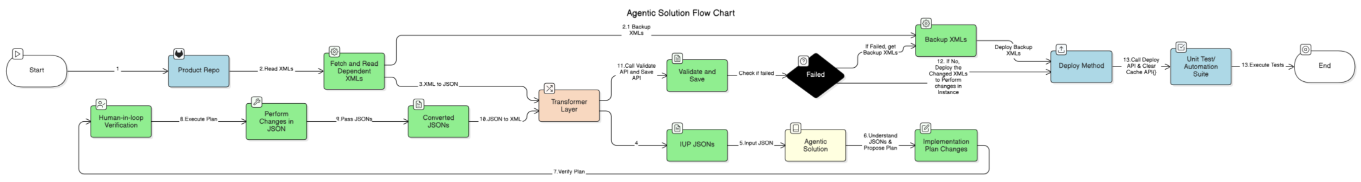
IUP Route for Metadata Changes - Overall Design

Agentic Solution can leverage IUP metadata XMLs to apply object changes reliably. It orchestrates the transformation of IUP XML files to JSON, generates a change plan, verifies it with human oversight, and then applies and validates changes via REST APIs. The process minimizes manual intervention while maintaining safety through validation checks, backups, and testing. as IUP process is the base for all changes in MetricStream and its proven method, so using this approach gives more confidence in accuracy in the process.

Architecture Flow

High-level flow of the automated IUP object change process. The system begins by retrieving IUP XML definitions from the local system and converting them to JSON format. An "Agentic Solution" engine then analyzes the JSON to propose an implementation plan for changes, which is verified by a human operator before execution. After the JSON is modified according to the plan, a validation API is called to ensure the changes are correct. If validation passes, the new XMLs are deployed (with caches cleared) and an automated test suite verifies system integrity. If validation fails, the process is halted and the original XMLs (backed up earlier) are restored to maintain stability.


Agentic Solution Flow Chart
Start
1
Product Repo
2.Read XMLs
Fetch and Read Dependent XMLs
2.1 Backup XMLs
3.XML to JSON
Transformer Layer
4
IUP JSONs
5.Input JSON
Agentic Solution
6.Understand JSONs & Propose Plan
Implementation Plan Changes
7.Verify Plan
Human-in-loop Verification
8.Execute Plan
Perform Changes in JSON
9.Pass JSONs
Converted JSONs
10.JSON to XML
11.Call Validate API and Save API
Validate and Save
Check if failed
Failed
If Failed, get Backup XMLs
Backup XMLs
Deploy Backup XMLs
12. If No, Deploy the Changed XMLs to Perform changes in Instance
Deploy Method
13.Call Deploy API & Clear Cache API()
Unit Test/ Automation Suite
13.Execute Tests
End

Workflow Steps and Components

1. **Local System (Source of IUP XMLs):** Developer does any changes on Object or want to see changes on the object. For this the objects are using IUP metadata XMLs available in local system. It serves as the source of truth for the process, providing the target "head" XML object and any referenced dependent XMLs needed for the change.

2. **Fetch & Dependency Reader:** Retrieves the target XML from the file system along with all related dependent XML files (e.g., linked personalization rules or sub-configurations). It parses references within the main XML to gather dependencies, ensuring all necessary context is loaded before proceeding.

3. **Backup Module:** Creates a secure copy of the original XMLs before any modifications. This backup preserves the current state as a recovery point in case something goes wrong. Backups are stored safely (e.g., in versioned storage) so the original state can be re-deployed for rollback if needed.

4. **Transformer Layer (XML ⇄ JSON):** Converts data between XML and JSON formats. Initially, it transforms the fetched XML files into JSON structures for easier manipulation in code. After modifications, it converts the JSON back into XML format for deployment. This allows the pipeline to work with JSON internally while delivering XML to systems that require it.

5. **Agentic Solution Engine (Plan Generation):** Analyzes the JSON representation of the IUP data to determine necessary changes for the desired update. It automatically generates an *implementation plan* – a set of proposed JSON modifications (additions, deletions, updates) – based on predefined knowledge, rules template or checklist and AI logic to achieve the target state.

6. **Human-in-the-Loop Verification:** The proposed change plan is presented for human review before execution. A developer or analyst checks the plan to ensure it is correct, comprehensive, and aligns with expectations. They can approve the plan or adjust it if something looks incorrect. This manual checkpoint helps catch errors or unintended

changes that the automated engine might miss, adding confidence in the plan.

7. **Change Executor (Apply JSON Changes):** After approval, the Change Executor applies the modifications to the JSON data. It processes each step of the plan, altering the JSON structure accordingly – updating values, removing obsolete entries, or adding new elements as specified. After this step, the JSON data in memory reflects the intended new configuration (with each change logged for traceability).
8. **Validator (REST API Validation):** Once changes are applied, the Validator calls an VALIDATE REST API to validate the updated configuration. This typically involves sending the JSON (payload needed for this request) to a service that checks schema integrity, business rule compliance, and overall consistency. Based on the response:
    - If validation **passes**: The changes are confirmed valid. The updated configuration may be saved/persisted via the API, and the pipeline proceeds to deployment.
    - If validation **fails**: The system halts the process. A recovery procedure is triggered using the Backup Module's copies to restore the original XMLs. No changes are deployed, and the failure is logged for further analysis. This decision point prevents invalid configurations from moving forward.
9. **Deployment Module (Deploy XMLs & Clear Cache):** Upon successful validation, the Deployment Module deploys the final XMLs to the live environment or application instance. This may involve invoking a deployment REST API or running a script to update the system's configuration. After deploying the new XMLs, it clears relevant caches (via API) to ensure the changes take effect immediately. *(If the deployment step itself fails, the module would initiate rollback using the backups to restore the previous state.)*
10. **Test Suite (Unit & Regression Tests):** Finally, an automated test suite runs to verify the system works correctly with the new changes. It executes unit tests for individual components and regression tests for overall personalization functionality impacted by the update. The tests confirm that requirements are met, and no regressions were introduced. Any test failures are logged and would prompt further investigation or even a manual rollback with the backups.

Non-Functional Considerations

- **Logging & Monitoring:** Each component should emit detailed logs (e.g., fetch results, backup status, transformation actions, plan details, validation outcome, deployment status). These logs provide traceability and help debug issues. Monitoring alerts on critical failures (validation or deployment errors) ensure the team is notified immediately for quick intervention.
- **Error Handling & Recovery:** The system is designed to fail safely. If any step encounters an error (e.g., network timeout, transformation exception, validation error), the pipeline catches the exception, logs the issue, and stops further execution. Thanks to the backup, a recovery mechanism can restore the last known good state: for example, if validation or deployment fails, the original XMLs from backup are re-applied. Clear error messages and status codes are provided so developers can quickly identify the root cause.
- **Extensibility & Maintainability:** The modular architecture allows updates or replacements of components without affecting the whole system. For instance, the Transformer could be extended for new data schemas, or the Agentic Solution engine's logic improved or swapped out for a more advanced module with minimal changes to other parts. New validation rules or additional steps (like extra post-deployment checks) can be added as needed. Standard interfaces (REST APIs, common data formats) between components make it easier to extend and maintain the system over time.

Pending Decision

The best route to perform changes on objects is API, which is ideal way and right way. IUP route will work but this should be plan B if API route is having lot of issues or time consuming to write new APIs to complete all the required tasks.

IUP process also has some limitations and it doesn't cover all objects (details will be mentioned below section).

The decision to go by API route or IUP route will be depends on2 things:

1. Analyze & list all APIs which are needed for Agentic Solution to call and perform all required tasks. Make sure the Input/Output for the APIs also considered. Time & resource required to write new wrapper APIs without impacting current platform, also needs to consider.
2. Analyze the limitations of IUP and how to mitigate those.

Limitations of IUP

Tips & Guidelines for Shipping Infolets, Forms, Tables, and PushInfolets

General Do’s & Don’ts

- □ **Do not change data types** in:
    - **Tables**
    - **Forms / PushInfolets**
    - **Infolets**

    Note: Though data type changes might appear to work locally by deactivating the infolet in a DEV machine, these are **not supported via IUP**.

USBC Feature Enhancements

- Blueprints and workflow emails **can now be exported and imported** as part of USBC feature implementation.

Installer Preparation Guidelines

1. □ Ensure **base module artifacts are not included** in configuration installers.
    - For example, if the base module is `AUD` and configurations are saved under `XUD`, do **not export AUD artifacts** in the `XUD` installer.
2. □ **Do not export PushInfolet** of a Form (if the Form version is 7.0 or higher).
    - PushInfolets are automatically created in the target instance during installation.
3. □ **Exclude alerts, thresholds, and run-infolets** linked to Forms from the master list.
    - These are created during installation.
4. □ **JS files** should continue to be extracted manually.
    - Form-related JS must be explicitly shipped.
5. □ **Do not extract the `_P` package** unless modified in the source instance.
    - If modified, ensure all **related data tables** are also shipped.
    - Missing tables will result in invalid `_P` packages and **Form deployment failures**.

6. □ **Avoid scripts that manipulate COMPONENT metadata**, as they may cause upgrade issues in future layers.

7. □□ **Configured Forms & auto-derived Data Objects** will not appear in the export screen unless deployed.

Current Limitations

1. □ **Form IUP does not support Workflow Transaction Security (WFTS)** — fix in progress.

2. □ **Form export will not include dependent objects** (e.g., reports, infolets) — fix in progress.

3. □ **6.x PushInfolet exports are not available via IUP** — feature deprecated; fix in progress to enable support.

4. □ **CMExtract tool does not support configured object export** — fix in progress.

5. □ **IUP does not support general data type changes**, except:

6. □ **Exception: VARCHAR to CLOB** conversion is allowed.

7. □ **Business Rules must be exported individually**; they are not included in Form IUP.

8. □ When importing a Business Rule (context) that references **non-existing fields**, the installation will still succeed.

9. □□ **For InfoCenter imports from older versions (e.g., Danube)**, you must manually run the transformation: `React_Metadata_Transformation`.

10. □□ **Exporting Forms/Objects may include internal test objects.**
    You have two options:

    - a. Remove test objects from the source instance before export.

    - b. Remove test-related sections manually from the exported IUP XML.