DIAMOND CO.

# Roblox DataStore Migration and Error Handling Kit

# Roblox DataStore Migration & Error Handling Kit: Zero-Data-Loss Solution for 2024 API Deprecations

**Version:** 1.1.0
**License:** MIT

> **Designed for:** Developers facing Roblox's 2024 API deprecations who need a battle-tested migration pipeline that prevents silent data corruption in leaderboard systems, inventory databases, and player profiles.

---

## Table of Contents

1. [Quick-Start Checklist](#quick-start-checklist)
2. [Feature Overview](#feature-overview)
3. [Full API Reference](#full-api-reference)
4. [Source Code (All Modules)](#source-code-all-modules)
5. [Performance Benchmarks & Stress-Test Logs](#performance-benchmarks--stress-test-logs)
6. [Maintenance & Community](#maintenance--community)
7. [Changelog](#changelog)
8. [License (MIT)](#license-mit)

---

## Quick-Start Checklist

| ✅ | Step | What to Do |
|---|---|---|
| 1 | **Import the kit** | Place `DataMigrationKit.lua` inside a `ServerScriptService` folder called `DataMigrationKit`. |
| 2 | **Configure retry defaults** | Call `DataMigrationKit.SetRetryConfig({Attempts = 4, BaseDelay = 1.5, Timeout = 30})`. |
| 3 | **Create a lock store** | No extra work – the kit automatically creates a `<DataStoreName>_Locks` store on first lock request. |
| 4 | **Define a transform function** | Write a Lua function that receives `oldData` and returns the new table (or raises an error). **Example:** ```lua function transform(oldData) if not oldData then return {coins = 0, inventory = {}} end local newData = table.clone(oldData) newData.coins = newData.coins * 2 return newData end``` |
| 5 | **Enqueue migrations** | `local qId = DataMigrationKit.EnqueueMigration("PlayerData", "Player_123", transformFn)` |
| 6 | **Process the queue** | Call `DataMigrationKit.ProcessQueue(20)` each server tick or from a background loop. **Warning:** The queue persists across server restarts but has a maximum capacity of 10,000 items. |
| 7 | **Monitor** | Use `DataMigrationKit.GetQueueStatus()` and `DataMigrationKit.GetRetryConfig()` to watch health. |
| 8 | **Deploy** | Publish the game – the kit handles throttling, locks, and rollbacks automatically. |

---

## Feature Overview

| Feature | What It Does | Why It Matters |
|---|---|---|
| **Automatic Retry Logic** | Retries `SetAsync/GetAsync` calls on throttling (HTTP 502/503) using exponential back-off. | Guarantees eventual consistency even under heavy load. |
| **Atomic Migration Framework** | Executes `transformFunc` inside a lock, writes the new value, and rolls back on any error. **Rollback Example:** ```lua function transform(oldData) if oldData.version < 2 then error("Incompatible data version") end return {version = 3, ...} end``` | Prevents partial writes and data loss. |
| **Session Locking** | Per-key lock stored in a separate DataStore (`<DataStore>_Locks`). Supports timeout and unique lock IDs. | Guarantees only one server migrates a key at a time. |
| **Queue Management** | In-memory FIFO queue with persistent backup (optional) and batch processing. **Conflict Resolution:** If a queued migration fails mid-execution, it will be retried with exponential backoff. | Lets you schedule thousands of migrations without exceeding rate limits. |
| **Configurable Parameters** | Retry attempts, base delay, max delay, queue batch size, lock timeout – all mutable at runtime. | Fine-tune for your game's traffic pattern. |
| **Full Error Reporting** | Every public call returns a tuple `(success, payload)` where `payload` is either the result or a **structured error table** (`{Code = "...", Message = "...", Details = {...}}`). **Example:** ```lua local ok, result = DataMigrationKit.Migrate(...) if not ok and result.Code == "TransformFailed" then warn("Migration failed: "..result.Message) end``` | Makes debugging deterministic and IDE-friendly. |
| **Performance Transparency** | Benchmarks, stress-test logs, and per-frame impact metrics are shipped with the kit. | Lets you verify the kit meets your SLA before launch. |

---

## Full API Reference

All functions live under the global table `DataMigrationKit`.

> **Note:** Every public function returns **two values** – `success` (`boolean`) and `payload` (`any`). When `success` is `false`, `payload` is an **error table** with the shape `{Code:string, Message:string, Details:table?}`.

| Function | Signature | Parameters (type, default) | Returns | Possible Error Codes |
|---|---|---|---|---|
| **SetRetryConfig** | `SetRetryConfig(config: table) -> (bool, nil)` | `config` – `{Attempts?: number=3, BaseDelay?: number=2, MaxDelay?: number=15, BackoffFactor?: number=2, Timeout?: number=30}` | `true` on success | `"InvalidConfig"` – missing or out-of-range values |
| **GetRetryConfig** | `GetRetryConfig() -> (bool, table)` | – | `{Attempts, BaseDelay, MaxDelay, BackoffFactor, Timeout}` | – |
| **SaveWithRetry** | `SaveWithRetry(dataStoreName: string, key: string, value: any) ->` | | | |



-- Example of handling a failed migration

local ok, result = DataMigrationKit.Migrate("PlayerData", "Player_123", transform)

if not ok then

if result.Code == "CorruptedData" then

-- Handle corrupted data case

warn("Data corruption detected for Player_123")

elseif result.Code == "MigrationTimeout" then

-- Handle timeout case

warn("Migration timed out for Player_123")

```
else
warn(string.format(
"[MigrationKit] %s – %s (details: %s)",
result.Code,
result.Message,
result.Details and table.concat(result.D
```