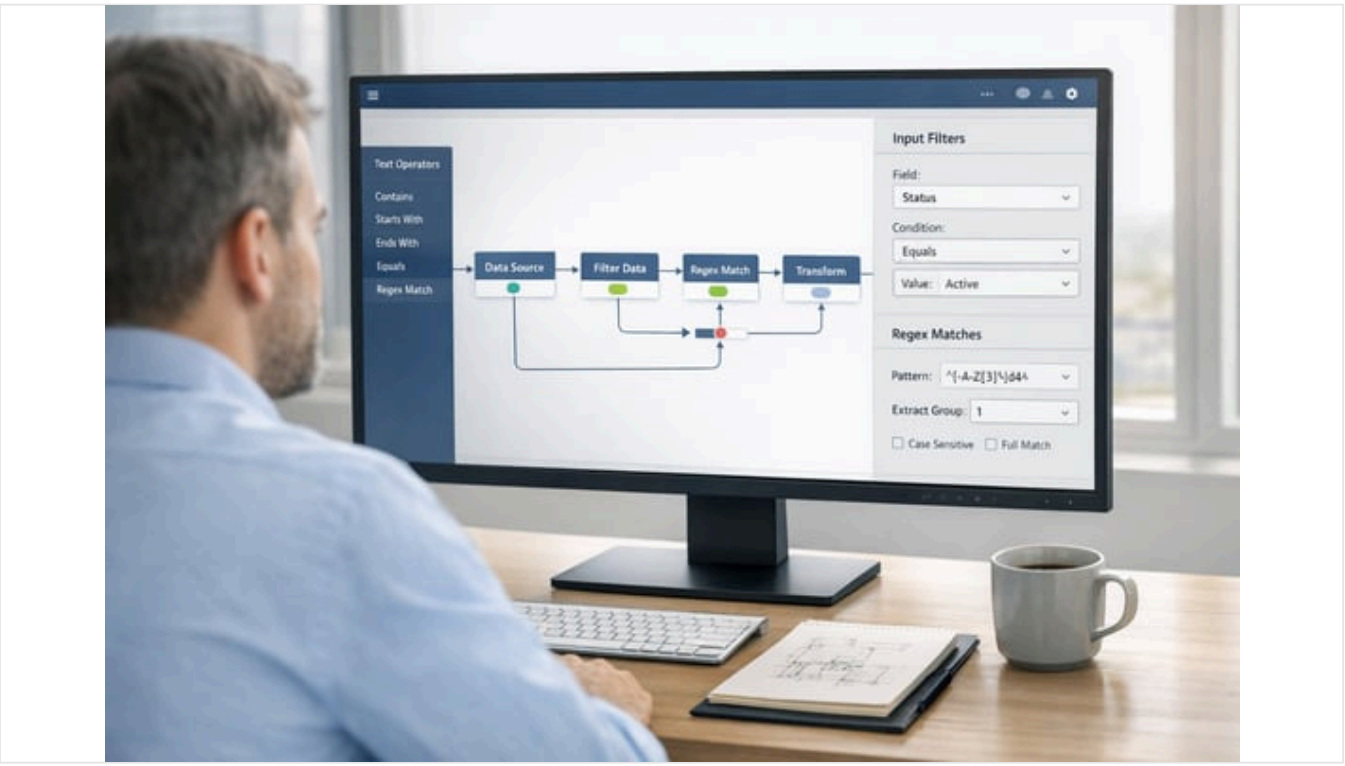
Celigo Input Filters Guide: Regex & Matches Operator

By houseblend.io Published April 11, 2026 28 min read



Executive Summary

This report provides an exhaustive examination of **Celigo's data integration platform**, focusing on *input filters*, the *"matches" operator*, and *data transformation* capabilities within Celigo's **Integrator.io** platform. Celigo is a leading integration platform-as-a-service (iPaaS) that facilitates seamless connectivity between SaaS and on-premises systems (Source: docs.celigo.com) (Source: www.celigo.com). As organizations increasingly rely on interconnected systems, effective data filtering and transformation become critical to ensure accurate, efficient integrations (Source: docs.celigo.com) (Source: www.integrate.io). This guide explains how Celigo's **input filters** selectively allow records through, how the *matches operator* and **regular expressions (regex)** are used within Celigo flows, and how Celigo's **data transformation** (specifically Transformation 2.0) can reshape data. We compare filter logic and regex techniques, detail Transformations 2.0 modes and field types, and present examples and case studies illustrating real-world usage. The analysis draws on official Celigo documentation, industry statistics, and expert perspectives to offer a comprehensive view of these capabilities, their implications for integration practice, and future trends in iPaaS and AI-driven automation (Source: docs.celigo.com) (Source: www.celigo.com).

Introduction and Background

The Role of iPaaS in Modern Integration

Modern enterprises operate dozens or hundreds of disparate systems and applications. An **integration platform** (iPaaS) serves as the middleware that connects these systems, automating data flows and business processes across the organization (Source: docs.celigo.com) (Source: www.celigo.com). Celigo is a cloud-based iPaaS designed to simplify integrations (e.g. **order-to-cash**, **quote-to-cash**, **hire-to-retire**) through hundreds of prebuilt connectors and customizable flows (Source: www.celigo.com). By following integration best practices – such as clear objectives, choosing known integration patterns, and maintaining stateless workflows – organizations can leverage platforms like Celigo to achieve reliable, scalable integrations (Source: docs.celigo.com) (Source: docs.celigo.com). For example, data from an e-commerce system can be filtered and transformed before syncing to an ERP, ensuring only relevant transactions are passed along in the correct format.

Celigo Integrator.io Overview

Celigo's **Integrator.io** is a visual flow-builder where users define data sources, apply filters and transformations, and map fields to destinations. Integrator.io flows consist of *export* (source) steps and *import/lookup* (destination) steps, with optional **middleware** steps. Data flows through these steps sequentially, and **filters** can be applied to control which records continue. **Input filters** are applied on import/lookup steps (so they skip unwanted incoming records), while **output filters** are applied on export or intermediate steps (so they pass only selected records forward) (Source: docs.celigo.com) (Source: docs.celigo.com). Filters use Boolean logic and comparison operators (equals, contains, etc.) to include or exclude records (Source: docs.celigo.com) (Source: docs.celigo.com). For scenarios requiring more complex logic than simple rules, Celigo also supports writing filters in JavaScript (Source: docs.celigo.com).

Celigo supports data transformation through its **Transformation 2.0** engine, which can modify, combine, or reshape input data before it is mapped to the destination. This includes flattening nested JSON, combining multiple source records, or converting data types (Source: docs.celigo.com) (Source: docs.celigo.com). Transformations can use JSONPath to select fields and Handlebars templates to compute values (Source: docs.celigo.com). In complex flows, filters and transformations work together: filters prune out irrelevant data early, and transformations then prepare the remaining data for import.

Given the importance of correct data handling, Celigo's platform provides multiple ways to match and transform data:

- **Matches Operator:** In filter rules, the "matches" operator tests whether a field equals a specified value (Source: docs.celigo.com).
- **Regular Expressions:** Celigo supports regex through Handlebars helper functions (`regexMatch`, `regexReplace`, `regexSearch`) for sophisticated pattern matching and manipulation of string data (Source: docs.celigo.com) (Source: docs.celigo.com).
- **Data Transformation:** The Transformation 2.0 engine (Rules 2.0) offers advanced mapping modes and field types (Standard, Lookup, Handlebars expression, etc.) for restructuring data (Source: docs.celigo.com) (Source: docs.celigo.com).

This report will delve deep into each of these areas. We will explain how to use input filters (including the matches operator), how to leverage regex helpers for pattern-based filtering and parsing, and how to configure data transformations. We also highlight practical examples and case studies, such as using an input filter to sync only "Adjustment" records from Amazon FBA (Source: docs.celigo.com), and how companies have applied Celigo to real-world problems (Source: www.celigo.com). Finally, we discuss implications – such as the role of AI and iPaaS in future integrations – and conclude with best practices.

Celigo Input Filters: Concepts and Use

Celigo's **input filters** are a key mechanism for controlling data flow. An input filter on an import (or lookup) step causes that step to process *only* the records that meet the filter criteria, while allowing other data to bypass that step (although subsequent steps with no filter will still see it) (Source: docs.celigo.com). In practical terms, input filters act like gatekeepers: they check each incoming record and either allow it into the step or skip it. This is useful for ignoring unwanted data without interrupting the entire flow.

For example, in an Amazon-to-ERP flow, one might only want to sync orders with status "Shipped". By adding an input filter on the relevant step, only orders containing "Shipped" in their status field would be processed, and all others would be ignored at that step (Source: docs.celigo.com). Input filters are configured using a visual rules editor in Celigo's Flow Builder.Table of filter operators and descriptions is shown below.

OPERATOR	DESCRIPTION
equals	Field is equal to value
not equals	Field is not equal to value
is greater-than	Field is greater than (>) the value
is greater-than or equals	Field is ≥ the value
is less-than	Field is < the value
is less-than or equals	Field is ≤ the value
between	Field is between two values (inclusive)
contains	Field contains a given substring (Source: docs.celigo.com)
does not contain	Field does *not* contain the substring (Source: docs.celigo.com)
ends with	Field ends with the specified substring (Source: docs.celigo.com)
null	Field value is empty (no value) (Source: docs.celigo.com)
is not null	Field has some value (not empty) (Source: docs.celigo.com)
matches	Field has exactly the specified value (Source: docs.celigo.com)

Table 1: Celigo filter operators (as of 2025) and their meanings (Source: docs.celigo.com) (Source: docs.celigo.com).

In the table above, note that the **matches** operator is effectively an equality test for string fields. Celigo's documentation defines *"matches"* simply as "has a specific value in the field" (Source: docs.celigo.com). In practice, you would use *matches* when you want to require an exact match (for example, matching a status exactly to "Active"), whereas *contains* allows partial matching. (Other operators like *equals* are generally used with numeric or date fields but could apply to strings as well.)

Celigo's filter editor allows complex rule sets. You can group conditions with AND/OR logic and even negate groups using NOT (Source: docs.celigo.com) (Source: docs.celigo.com). Once the rules are set, they determine which records pass through. For example, Celigo documentation illustrates adding an input filter to only sync FBA **InventoryAdjustments** of type "Adjustments" (Source: docs.celigo.com) (Source: docs.celigo.com). In that example, the user selects `record.[Event Type]` (string field), operator *contains*, and the value "Adjustments". Only records with that text in the Event Type would pass (Source: docs.celigo.com) (Source: docs.celigo.com).

In addition to rule-based filters, Celigo allows you to write filters in JavaScript. By toggling the filter editor from *Rules* to *JavaScript*, any logic can be scripted (Source: docs.celigo.com). The rules toggle is useful for scenarios like concatenating multiple fields or doing regex tests within the filter. The documentation notes that JavaScript filters can handle complex conditions (e.g. *"Concatenate multiple string fields, and then filter based on the results of the concatenated value"* or *"Filter records where a date field is older than a certain number of days"* (Source: docs.celigo.com). This flexibility means that if the standard operators and boolean groups are insufficient, a custom JS function can implement arbitrary logic.

Overall, input filters let you **prune unwanted data early in the flow**, improving efficiency and data quality. For instance, Celigo best practices suggest filtering out records by criteria to reduce downstream processing (Source: docs.celigo.com) (Source: docs.celigo.com). In one company case, adding an input filter to limit Amazon FBA records to only "Adjustments" event types prevented irrelevant shipments or transfers from being sent to NetSuite (Source: docs.celigo.com). This kind of targeted filtering keeps flows performing well and avoids downstream data errors or overload.

Celigo Regex and the "Matches" Operator

The “Matches” Operator in Filters

As noted above, the Celigo filter *matches* operator is used for exact matching. Unlike some platforms that let you enter regex patterns directly into filter rules, Celigo’s built-in filter rules use *matches* as a strict value check (Source: docs.celigo.com). In other words, *“matches”* in the filter context means *equals* for a string. For example, if you set a rule `field matches "ABC123"`, it will only pass records whose **field** equals “ABC123” exactly. This is akin to the “equals” operator but specifically for string fields (the “equals” operator in the UI is often phrased for numeric or date fields).

Given that *matches* behaves like an equality check, one might wonder how to perform pattern matching using regular expressions. The Celigo filter UI does not directly expose regex matching. Instead, Celigo relies on its **Handlebars helpers** for regex-intensive operations (which are used in mapping/expressions, not in the simple filter UI). To perform regex filtering, you have a few options:

- **JavaScript Filter:** Switch the filter to JavaScript mode and use JavaScript’s `RegExp` or `.match()` within the filter code to test patterns.
- **Preprocess with Transformation:** Use a Transformation step (which supports Handlebars) to add flags or fields, then filter based on that.
- **Handlebars in Mappings:** Sometimes one can do filtering logic indirectly by setting conditions in transform or mapping.

For straightforward cases (exact match rather than pattern), *matches* is appropriate. For example, to filter records where `record.Status` exactly equals “Shipped”, one would use:

```
Field: record.Status
Operator: matches
Value: Shipped
```

No partial matching or wildcards are implied. On the other hand, `contains` would also match “Shipped” if the status field contains it anywhere, and `does not contain` would exclude it. In summary, the *matches* operator enforces exact equivalence (Source: docs.celigo.com).

Regular Expression Helpers in Celigo Flows

While the primary filter interface does not use regex, Celigo provides powerful *regex helpers* for use in data transformations and mappings. These helpers operate via Handlebars expressions (double braces) and empower you to search, extract, or replace text based on patterns. The main helpers are:

- **{{regexMatch field regex ...}}** – Matches and returns the text of the regex match.
- **{{regexReplace field replacement regex}}** – Replaces text matching a regex with a given replacement.
- **{{regexSearch field regex}}** – Searches for a regex match and returns the index of the match.

These helpers are documented in Celigo’s Help Center (Source: docs.celigo.com) (Source: docs.celigo.com) (Source: docs.celigo.com) and rely on the Node.js/JavaScript regex engine (Source: docs.celigo.com). In practice, they allow much more sophisticated pattern logic than the basic filter operators. Let’s examine each:

- **regexMatch:** This helper takes a text field and a regex pattern, and returns the matching portion(s) of the text. By default it returns the first match. For example, given `record.comment = "Order ID: 12345 delivered on 2025-04-20"`, using

  ```
  {{regexMatch record.comment "[0-9]{5}"}}
  ```

  will output `"12345"`, because it finds the first 5-digit sequence (Source: docs.celigo.com). You can also specify an index (0-based) if the field contains multiple matches. For instance, if the comment has “IDs: 12345 and 67890”, then `{{regexMatch record.comment "[0-9]{5}" 1}}` returns `"67890"` (Source: docs.celigo.com). Regex flags (like `"i"` for case-insensitive) can be passed as an optional fourth argument.

- **regexSearch:** This helper searches for a regex pattern and returns the position (index) of the first match in the string (with 0 = first character). It does not return the text, only the numeric index. For example, if `record.total = "$1499.95"`, then

  ```
  {{regexSearch record.total "\."}}
  ```

returns `5`, since the first "." (decimal point) is the 6th character (index 5) (Source: docs.celigo.com). A case-insensitive option can be added (`"i"` flag). If no match is found, `regexSearch` returns `-1` (Source: docs.celigo.com), which can be used in conditional logic (e.g. filter out records where `-1` indicates absence of a pattern).

- **regexReplace:** This helper performs find-and-replace using a regex. (Although not explicitly documented on a separate page, Celigo's docs mention it and the replace helper references it (Source: docs.celigo.com).) It takes a field, a replacement string, and a regex pattern (with optional flags), and returns the text with all matches replaced. For instance, to strip non-digit characters from a phone number:

  ```
  {{regexMatch (regexReplace phone "" "[^\\d]" "g") "\\d{10}$"}}
  ```

  (This example, from Celigo's docs, first uses `regexReplace` to remove anything not a digit, then `regexMatch` to capture 10 digits (Source: docs.celigo.com).) If your need is simpler (fixed substring replacement), Celigo also provides a non-regex `replace` helper (Source: docs.celigo.com).

These regex helpers rely on JavaScript's `RegExp` behavior (Source: docs.celigo.com) (Source: docs.celigo.com), meaning support for patterns like `[A-Z]`, lookahead/lookbehind, global (`g`), multiline (`m`), etc. For full pattern creation, Celigo suggests using an external regex tester like regex101 first (Source: docs.celigo.com), then embedding the validated pattern into the Handlebars.

Use Case Examples of Regex Helpers

- **Extracting substrings:** If an integration requires extracting structured data from free-form text, `regexMatch` is ideal. *Example:* Given a field `record.comment = "Order #ID12345 shipped on 2024-08-15"`, one could extract the numeric ID via

  ```
  {{{regexMatch record.comment "[0-9]+$"}}}
  ```

  which might yield `"12345"` (the `{ }}` triple-handles notation can be used to avoid Celigo's quoting).

- **Filtering by pattern:** Though filters don't directly use regex, you could use `regexSearch` within a transformation to tag records and then filter. For instance, to filter names that start with "ENG", one could create a field in a Transformation with `regexSearch(record.name, "^ENG")` and then set the input filter to `newField > -1`.

- **Complex replacements:** Suppose SKU codes in a source have varying delimiters and need standardization. One could use `regexReplace record.SKU "_" "\\W+"`, which replaces non-word characters with underscores. Combined with filters, one could then filter SKUs that now match a pattern.

Each regex helper usage should be cited to documentation and carefully tested in context. Celigo's examples (Source: docs.celigo.com) (Source: docs.celigo.com) serve as good starting points for common patterns.

Data Transformation in Celigo Integrations

When and Why to Transform Data

In any integration scenario, **data transformation** is often necessary to reconcile differences in data structure, format, or semantics between systems. Celigo's Transformation 2.0 (Rules 2.0) provides a powerful engine for this purpose (Source: docs.celigo.com). Transformations are applied to the *input data* before mapping to the output. This is especially useful in two cases:

1. **One source, multiple destinations:** When you export from one source (e.g. a complex JSON from a database) into *multiple* subsequent import steps, you may want to simplify the data first. For example, if the export includes many nested fields and arrays, a transformation can flatten or filter it so that each import only sees the relevant subset (Source: docs.celigo.com). Celigo's docs note that transformations are ideal for flattening and simplifying source records destined for multiple sinks.

2. **Multiple sources, one destination:** Conversely, if you have multiple data sources feeding a single target, you may need to consolidate them. Celigo allows transforming each source's records into a common, canonical format before sending to the destination (Source: docs.celigo.com). This lets different inputs be merged or normalized so that the destination mapping logic is simpler.

In practice, one might use Transformation 2.0 to perform tasks such as: flattening nested JSON structures, converting arrays to row-based CSV format, renaming fields, combining source records, or splitting a single record into multiple output rows (e.g. creating line items). The timing of transformations is typically immediately after export and before any imports. It lets you "shape" the data so that your final mappings are straightforward and robust.

Transformation 2.0: Modes

Celigo's Transformer provides three **modes** of operation (Source: docs.celigo.com), selectable via a dropdown:

- **Modify the Input:** This mode makes targeted changes *within* the original record while keeping all other fields intact. Use this for small adjustments; for example, correcting a phone number format, trimming whitespace, or adding a computed flag, while leaving the bulk of the record structure unchanged (Source: docs.celigo.com).
- **Create Output Record from Input:** This mode allows building a brand-new output record from scratch using fields (or calculations thereof) from the input. It is useful when you only want to pass a subset of fields or when merging fields into new fields. For instance, when consolidating multiple source types, you define exactly how each part of the output should derive from the input (Source: docs.celigo.com).
- **Create Output Rows from Input:** This mode is for converting one input record into *multiple* output rows (flattening array elements into separate rows). An example use-case would be transforming a single order with a list of items into multiple line-item rows for export as CSV (Source: docs.celigo.com). Each array element can become a new output record, enabling workflows that require row-based outputs.

Choosing the correct mode is key. Generally, "Modify Input" is quick and keeps things simple, "Output Record" grants full control over the final structure, and "Output Rows" is specialized for one-to-many record creation.

Transformation Field Types

Within a transformation mapping, each output field has a **field type** that determines how its value is computed (Source: docs.celigo.com) (Source: docs.celigo.com). The four types are:

FIELD TYPE	DESCRIPTION	EXAMPLE
Standard	Directly maps a field from the source to the output (default) (Source: docs.celigo.com).	Map `record.firstName` → `FirstName`.
Hard-coded	Provides a fixed static value for the output field, regardless of input.	Set `Country` = `"USA"` in all output records.
Lookup	Uses a lookup table to map one input value to another output value (Source: docs.celigo.com).	Map `"apple"` → `"banana"` for any occurrence of `"apple"`.
Handlebars Expression	Uses a custom Handlebars template or formula for the output value (Source: docs.celigo.com).	Concatenate `{{record.firstName}}` `{{record.lastName}}`.

Table 2: Celigo Transformation field types and usage (Source: docs.celigo.com) (Source: docs.celigo.com).

The **Standard** and **Hard-coded** types are straightforward: either pass through the value or set a constant. **Lookup** is useful for simple value conversions (e.g. currency code lookups or status code mapping). **Handlebars Expression** provides the most flexibility, allowing complex logic (concatenation, arithmetic, conditional expressions, encoding, etc.) via templating (Source: docs.celigo.com) (Source: docs.celigo.com). For instance, you could multiply two fields (`{{ =}}`) or apply `upper` / `lower` functions with Handlebars.

Celigo's Transformation engine supports data types like string, number, boolean, object, arrays, etc., and can automatically coerce between types where possible (Source: docs.celigo.com). If conversion is not possible, the mapping will fail (e.g. mapping a string to a number without parse). There are also settings for handling nulls and documenting field logic via comments (Source: docs.celigo.com).

Using JSONPath and Handlebars in Transformations

Celigo leverages **JSONPath** to select fields from complex input records (Source: docs.celigo.com). JSONPath is a query language for JSON (similar to XPath for XML) that lets you specify fields even if they are nested or in arrays. The transition to using JSONPath means you can write expressions like `order.items[*].sku` or `$.customer.address.city` to pick values. The Celigo documentation notes that JSONPath expressions can include array slices and recursive descent ("*" syntax) (Source: docs.celigo.com). This is critical when the data has nested structures; you can precisely navigate to the needed information for transformation.

Handlebars expressions can then compute or format these values. For example, to combine first and last name: `{{record.customer.firstName}} {{record.customer.lastName}}`, or to format dates. Handlebars also support branching (via `{{#if}}`) and many built-in helpers. Celigo provides custom tools and extensions (e.g., math operations, string functions) inside Handlebars to empower transformations (Source: docs.celigo.com). For instance, if you needed to encode a field for a URL, you could use a Handlebars helper like `{{encodeField record.text}}` (hypothetical).

Transformation Example

Suppose a flow receives an e-commerce order JSON like:

```
{
  "order": {
    "id": 123,
    "customer": {"firstName": "Alice", "lastName": "Smith"},
    "items": [ {"sku": "A1", "qty": 2}, {"sku": "B2", "qty": 1} ],
    "date": "2025-07-01T12:34:56Z"
  }
}
```

We want to send to a destination that expects records with fields: `OrderID`, `CustomerName`, `Sku`, `Quantity`, `OrderDate`. Using **Create Output Rows from Input** mode, we could define:

- `OrderID (Standard) = record.order.id`
- `CustomerName (Handlebars) = {{record.order.customer.firstName}} {{record.order.customer.lastName}}`
- Then an *Output Rows* mapping: Palce `record.order.items[*]` as the input array.
  For each item, we map:
  - `Sku (Standard) = record.sku`
  - `Quantity (Standard) = record.qty`
  - `OrderDate (Standard) = record.order.date`

Celigo would produce 2 output rows: one for SKU=A1, qty=2, one for SKU=B2, qty=1, each including the OrderID and CustomerName repeated. In this example, we'd use JSONPath like `order.items[*]` (Source: docs.celigo.com) for the rows, and Handlebars to build `CustomerName`. The result is row-based data suitable for CSV export, flattened from the nested JSON.

Data Analysis and Performance Considerations

Celigo's filtering and transformation features are built to handle enterprise volumes. For instance, Celigo reported processing **36.5 billion records** during Black Friday/Cyber Monday 2025, with no slowdowns (Source: www.celigo.com). Such scale underscores why robust filtering is essential: pruning irrelevant records early significantly reduces processing load on downstream systems. According to Celigo, they maintained 100% uptime during that peak period (Source: www.celigo.com). Superior filter design contributes to reliability by ensuring flows handle only valid data.

From an industry perspective, the need for strong integration is acute. A MuleSoft survey found organizations average ~897 applications but only ~29% integration between them (Source: www.integrate.io). This disconnection costs companies significant ROI – well-integrated firms see *10.3*× ROI from AI initiatives vs *3.7*× for poorly integrated firms (Source: www.integrate.io). Although not Celigo-specific, this highlights that clean, well-filtered

data flows (like those Celigo enables) are key to unlocking business value. Conversely, integration failures are common; one report notes *84% of system integration projects fail or partially fail* due to issues like poor requirements or data problems (Source: www.integrate.io). Proper use of filters and transformations in Celigo can help avoid such pitfalls by enforcing data quality and schema expectations before integrating systems.

Data transformation is similarly critical: 64% of organizations cite data quality as their top challenge in integration (Source: www.integrate.io). By using Celigo's transformations, integrators can enforce data formats (e.g. normalize date formats, trim whitespace) and catch anomalies. For example, a Celigo flow might use a rule **"null"** or **"is not null"** operator to filter out records with missing key IDs (Source: docs.celigo.com). This avoids sending incomplete data that would cause errors in the ERP. Even beyond filters, transformation 2.0 can validate or enrich data (e.g. lookup tables to fill missing values). In sum, rigorous filtering and data shaping help ensure that only high-quality, compliant records proceed – addressing the very concerns (data quality, silos) that plague digital transformation success (Source: www.integrate.io) (Source: www.integrate.io).

Case Studies and Real-World Examples

Case Study: Filtering Amazon FBA Adjustments

Celigo's documentation provides a concrete example of using input filters in a retail scenario (Source: docs.celigo.com) (Source: docs.celigo.com). In the *Amazon (FBA) Inventory Adjustments* flow to NetSuite, different event types (Shipments, Transfers, Receipts, etc.) are present, but the business only wanted to sync **"Adjustments"** entries. By adding an input filter on the export source step, the integrator specified:

```
Field: record.[Event Type]
Operator: contains
Value: Adjustments
```

Only records where the `Event Type` contained "Adjustments" would pass. The Celigo guide walks through this setup: selecting the field, setting operand type to Field/String, using AND, choosing *contains*, and entering the text "Adjustments" (Source: docs.celigo.com) (Source: docs.celigo.com). After saving and running the flow, the customer observed that only "Adjustments" records synced into NetSuite; other event types were ignored.

This illustrates two points: (1) Input filters can be added *on any step* of a flow (including on the source tile via the "Define input filter" dialog) (Source: docs.celigo.com). (2) Filters with basic operators (contains, equals, etc.) are often sufficient for real-world needs. In this case, regex was not needed — a simple *contains* rule solved the problem. If, however, the requirement had been to match a pattern (e.g. event type codes that all start with "ADJ-"), a JavaScript filter or regex helper could be used.

Case Study: Enterprise Fulfillment Integration

In a large-scale deployment, **Therabody** (a consumer electronics company) used Celigo to integrate orders, inventory, and fulfillment across multiple systems (Source: www.celigo.com) (Source: www.celigo.com). They replaced a legacy iPaaS and built flows for order management and 3PL integration. Using Celigo, Therabody connected NetSuite ERP with multiple third-party logistics providers (both domestic and international) in real-time. Crucially, they could sync inventory levels and automate fulfillment without manual workarounds (Source: www.celigo.com).

Although the case study focuses on high-level outcomes (e.g. reduced complexity, efficient order processes), it implies extensive use of Celigo features. For instance, connecting with 3PLs often involves filtering the feed by warehouse or SKU, and transforming data into the formats each system expects. The success at scale (handling peak season operations with reduced labor costs) indicates that Celigo's filters and transformations were reliably processing large transaction volumes. The Therabody story underscores how an enterprise can rely on Celigo for mission-critical integrations involving multiple data sources and targets (Source: www.celigo.com) (Source: www.celigo.com).

Example Workflow: Regex in Action

To illustrate **regex usage** in a practical flow, imagine a e-commerce integration where product SKUs from a marketplace include suffixes that we need to strip out. For example, incoming SKUs might look like `"ABC123-XL"` or `"XYZ999-KIDS"`, and the target system expects just `"ABC123"` or `"XYZ999"`. In Celigo, one could add a transformation field using `regexReplace` to remove the suffix after the dash:

```
{{regexReplace record.SKU "" "-.*$"}}
```

This Handlebars expression uses the regex `-.*$` to find a dash and all following characters, replacing them with an empty string. After this transformation, the SKU becomes normalized. Then a filter could use matches on the cleaned SKU, or simply map it to the destination field. This approach demonstrates combining regex replacement in a transform, followed by standard mapping — a workflow that might be needed in many B2C scenarios.

Discussion: Implications and Future Directions

Broader Perspectives

From a **technical perspective**, Celigo's combination of point-and-click interfaces with advanced options (filters, JSONPath, Handlebars) hits a sweet spot between usability and power (Source: docs.celigo.com) (Source: docs.celigo.com). Less technical users can implement rules with dropdowns and logical operators, while advanced users can drop into JavaScript or write handlebars for complex cases. This reflects a general trend in integration tools: enabling both codeless "low code" and code-driven customization (Source: www.celigo.com) (Source: www.celigo.com). The consequence is that organizations can reduce developer backlog by empowering analysts to build simple filters and transformations, yet still accommodate developers for edge cases.

From a **business perspective**, using input filters and transformations effectively improves data governance and trust. By filtering out noise (test records, non-relevant sales channels, duplicate events) and reshaping data into consistent formats, companies reduce data errors and manual reconciliation. Industry research indicates that **poor data quality costs trillions** globally (Source: www.integrate.io); thus, anything that minimizes garbage-in (via filters) is valuable. Celigo's platform thus serves not just as a connector, but as an enforcement point for data quality standards.

There is also an **operational perspective**: Celigo flows with filters can lower system loads and avoid rate-limit issues. For example, filtering out unwanted records reduces API calls to downstream systems (pay-as-you-go costs). Celigo's own metrics (Billions of records processed, zero downtime) (Source: www.celigo.com) show the importance of efficiency — even small inefficiencies at scale can translate to big issues. Designing tight filters and smart transformations is thus as much about cost-saving as correctness.

Integration with AI and Future Trends

Looking forward, Celigo is positioning itself as central to AI-driven automation in enterprises (Source: www.celigo.com) (Source: www.celigo.com). The logic of filters and transformations intersects with AI in two ways:

1. **AI-Assisted Mapping:** Celigo now offers AI tools like *AI Code Assistant* (which can generate Handlebars or JavaScript from natural language instructions) and *Knowledge Bot* for doc guidance (Source: www.celigo.com). In the future, one might instruct the system "filter orders where EventType starts with 'ADJ'" and have Celigo auto-generate the filter. Similarly, Celigo's AI features could suggest regex patterns or transformations based on sample data. This can help remove guesswork when writing complex filter or transform logic.

2. **Intelligent Filtering:** The "create or explain filter rules using Celigo AI component" suggests that Celigo is exploring using AI to recommend filter criteria (Source: docs.celigo.com). One could imagine an AI analyzing historical flow data and advising, "90% of events with type 'Ship' also have FieldX='Y', maybe filter X>Y for better performance." While speculative, it indicates a trend where AI aids in integration design. Celigo's unified platform approach aims to support both integration and AI/ML workflows on the same canvas (Source: www.celigo.com) (Source: www.celigo.com).

More generally, enterprise integration is moving toward **self-service and orchestration**. Celigo's vision (per their blog) is "one unified platform" for integrations, APIs, and event-driven or agentic AI processes (Source: www.celigo.com) (Source: www.celigo.com). In this context, filters and transformations remain fundamental primitives: controlling data flows even in advanced iPaaS scenarios. For example, extremely complex workflows (such as AI-driven customer support chatbots that fetch data from multiple systems) still require filtering (e.g. ignore non-customer emails) and transforming output into user-friendly text. The consistency of flow logic is what makes AI outputs reliable.

Limitations and Alternatives

While Celigo's approach is robust, there are some considerations:

- The filter "matches" operator is limited to exact matches. Organizations requiring more flexible pattern matching *within the filter itself* may find the interface constraining. The workaround is using JavaScript or pre-processing, which introduces complexity. Some competing integration tools allow regex in the filter UI natively. However, in Celigo this gap is mitigated by the transformation layer.
- Reliance on Handlebars for complex logic means that users must learn its syntax. Although Celigo's documentation and community are helpful, there is a learning curve to mastering expressions like `{{regexMatch}}`, loops, and conditions.
- Data transformations can introduce latency if not carefully designed. While Celigo auto-scales, extremely large nested transformations could hit performance limits. Thus, best practice is often to push data preparation as close to the source as possible (e.g. filter out unnecessary rows before pulling them).

Alternative approaches could include doing some filtering upstream (in the source query) or downstream (post-import) instead of in Celigo. For example, if the source is a database, adding a `WHERE` clause might reduce the need for a Celigo filter. But Celigo's advantage is abstracting these details: the UI makes it easier to change without redeploying queries or code.

Conclusion

Celigo's integrator platform offers a flexible blend of visual configuration and advanced logic for managing data flows. **Input filters** allow users to precisely control which records flow through each step, conserving resources and maintaining data relevance. The filter operators (equals, contains, matches, etc.) cover most common needs (Source: docs.celigo.com) (Source: docs.celigo.com), and complex conditions can be handled via grouping or by switching to JavaScript for custom logic (Source: docs.celigo.com).

For pattern-based processing, Celigo provides **regular expression helpers** (`regexMatch`, `regexSearch`, `regexReplace`) within its Handlebars/Transformation framework. These live up to the power of JavaScript's regex engine and enable slicing, dicing, and cleaning of textual data (Source: docs.celigo.com) (Source: docs.celigo.com). While the built-in filter UI does not natively use regex, Celigo's architecture ensures that regex capabilities are readily accessible when needed, either in transformations or scripted filters.

Data transformation in Celigo is comprehensive. Transformation 2.0 introduces multiple modes and field types to reshape data as needed (Source: docs.celigo.com) (Source: docs.celigo.com). It supports flattening of nested documents, splitting records, and generating derived values using JSONPath and Handlebars (Source: docs.celigo.com) (Source: docs.celigo.com). We have illustrated how transformations can turn complex source JSON into flat output formats suitable for downstream applications. Together with input filters, transformations allow a broad spectrum of integration patterns – for example, Celigo customers have implemented end-to-end order fulfillment and 3PL syncing flows using exactly these features (Source: www.celigo.com).

Looking ahead, Celigo is enhancing these capabilities with AI-assisted tools, making it easier to create and document filters and transformations. The role of iPaaS is expanding from simple connectivity to being the **"automation foundation"** for AI-driven processes (Source: www.celigo.com). In that context, Celigo's filter and transform logic will remain central – even as platforms evolve, organizations will need to ensure data is accurate and properly shaped at every stage (Source: www.celigo.com) (Source: www.celigo.com).

In summary, mastering Celigo's input filters, regex helpers, and transformations is crucial for building robust integrations. Together they provide a complete toolkit: filters enforce which records proceed, regex helpers allow fine-grained parsing of text, and transformations remodel data into target schemas. Applying these tools thoughtfully – as in the Amazon Adjustments example (Source: docs.celigo.com) or enterprise flows like Therabody's (Source: www.celigo.com) – yields efficient, reliable data pipelines. For any integration project, rigorous use of these features, backed by testing and monitoring (as Celigo best practices advise (Source: docs.celigo.com), will ensure that data flows are correct, performant, and future-proof.

References

- Celigo Help Center – *Apply filters* (overview of input/output filters, operators) (Source: docs.celigo.com) (Source: docs.celigo.com).
- Celigo Help Center – *Regular expressions (regex)* (regexMatch, regexReplace, regexSearch usage) (Source: docs.celigo.com) (Source: docs.celigo.com).
- Celigo Help Center – *regexMatch helper* (examples of extracting regex matches) (Source: docs.celigo.com).
- Celigo Help Center – *regexSearch helper* (examples of regex search returning index) (Source: docs.celigo.com).
- Celigo Help Center – *replace helper* (substring replace and note on regexReplace) (Source: docs.celigo.com).
- Celigo Help Center – *Transformation 2.0* (modes, field types) (Source: docs.celigo.com) (Source: docs.celigo.com); *JSON path and handlebars* (Source: docs.celigo.com).

- Celigo Help Center – *Overview of integration best practices* (Source: docs.celigo.com) (Source: docs.celigo.com).
- Celigo Blog – *Enterprise iPaaS: Why Celigo leads* (architecture, performance stats) (Source: www.celigo.com) (Source: www.celigo.com).
- Celigo Customer Story – *Therabody* (enterprise integration example) (Source: www.celigo.com).
- Celigo Help Center – *Add input filter for FBA Inventory Adjustments* (step-by-step filter example) (Source: docs.celigo.com) (Source: docs.celigo.com).
- Integrate.io AI & Integration Stats – *Data Transformation Challenge Statistics 2026* (industry data on integration and failure rates) (Source: www.integrate.io) (Source: www.integrate.io).

Tags: celigo, input filters, matches operator, data transformation, regular expressions, integrator.io, ipaas, handlebars helpers

---