AI Data Tool - version 1

Apago, Inc.

Description

The AI Data Tool brings the power of Large Language Models (LLMs) directly into your Enfocus Switch flows. By using natural-language prompts, you can instruct the app to intelligently extract, restructure, or transform data from incoming jobs without writing complex scripts or regex patterns.

Whether you need to parse messy CSVs, convert legacy XML, or extract metadata from PDF invoices, this tool provides a flexible, "human-in-the-loop" logic to your automation flows.

What can it do?

The script handles four kinds of tasks, chosen automatically by the model based on what your prompt asks for:

- Extract — pull one or more values out of an input file. The values come back attached to the input job as a Switch dataset, so downstream elements can reference them via variable expressions.
- Reformat — rewrite the input file's content in a new style or layout (rename CSV columns, pretty-print JSON, restructure XML, etc.). The reformatted content replaces the original file.
- Convert — change format (CSV → JSON, XML → CSV, etc.). Same routing as reformat
- Split — break the input into multiple output files (split a JSON array into one file per record, etc.). Each output becomes a new child job; the input job is disposed.

Inputs supported

- Text files: JSON, XML, CSV, TXT, or anything readable as UTF-8
- PDFs and images (JPEG, PNG, GIF, WebP) — requires the Anthropic API

Compatibility

This app is compatible with Switch 25.11 and higher.

Connections

Inputs – at one is required.

Outputs – Data Success, Data Error (bad model output, API error, oversized PDF, etc) and Problem Jobs (catch-all for unrecoverable errors).

Flow elements properties

- **Prompt** – The natural-language instruction. Describes what you want done with the input. See "Writing effective prompts" below. If empty, the job fails immediately.

- **Provider** – Which LLM model to use. Model is downloaded once and cached locally.
  - Local – LLM model runs entirely on the Switch server. No network, no API cost. Good for text inputs and privacy-sensitive data. Cannot accept PDFs or images.
  - Anthropic API – Claude API. Faster, more accurate on complex prompts, handles PDFs and images natively, and has a much larger context window. Costs per-token.

- **Max input characters** – Hard cap on how many characters of the input file get embedded in the prompt. If the file is longer, it's truncated and the model is told it was truncated. Increase for longer documents on Anthropic; the local model's effective context is smaller, so raising this much above the default doesn't help.

- **Max new tokens** – Hard cap on how many tokens the model can generate in its response. Raise this for tasks that produce long outputs (e.g. converting a large CSV to JSON); lower it to bound runtime for simple extracts.

- **Model ID** – Displayed only if Local model is elected as Provider. Hugging Face model ID. Recommend ‘onnx-community/Qwen2.5-1.5B-Instruct’ a 1.54B-parameter instruction-tuned model specifically designed for instruction-following, conversational AI, and efficient deployment. Other ONNX-format chat models from Hugging Face will work if they share the same chat-template format.

- **Model cache dir** – Displayed only if Local model is elected as Provider. A writable directory where model files are downloaded on first use. Must persist across Switch service restarts. Typical value is something like “D:\Switch\Models” or “/Volumes/data/Switch/Models”

Outgoing connections properties

- None

Writing effective prompts

The model performs best with prompts that are specific about output format. A few rules of thumb:

- Say what you want, not how to do it. "Extract the invoice number" is better than "Use regex to find a string starting with INV-".

- For extracts, say exactly what you want returned. "Return just the number, no other text" prevents the model from wrapping output in explanation. "Return the value as a string" prevents quotes-in-quotes problems.

- For reformat/convert tasks, name the target format explicitly. "Convert this to JSON with keys name, email, signup_date" is much better than "make this nicer".
- For splits, describe how to split and what to name the files. "Split into one JSON file per record, named record_1.json, record_2.json, etc."
- Don't describe the input file's content if it's a PDF or image — the model sees the file directly. Just say what to do with it.

Avoid:

- Vague terms like "important info", "key fields" — different runs will pick different things.
- Asking for explanations or commentary in the same response — the script expects clean structured output.
- Multi-step instructions in one prompt. Break complex flows into multiple sequential script elements.

Example prompts

Extract — pull values out
These produce a Switch dataset attached to the input job. Reference them downstream with variable expressions like
`[Metadata.Text:Path="result",Dataset="AiResult",Model="JSON"]`

- Find the PaceJobTicket.ID and the first PaceJobTicket.JobPart.ID in this XML. Return them concatenated with a period between, e.g. "1234567.01". No quotes, no formatting, just the value.
- Extract the invoice number, total amount due, and due date from this invoice. Return all three values.
- Find any email addresses in this text. Return them as a JSON array of strings.
- What is the sum of the "amount" column in this CSV? Return just the number.
- Identify the document type (invoice, packing slip, purchase order, or shipping label). Return just the type as a single word.

Reformat — rewrite the file
These produce a single output file that replaces the input.

- Rename the columns in this CSV to name, email, signup_date. Keep all the data rows unchanged.
- Pretty-print this JSON with 2-space indentation. Sort top-level keys alphabetically.
- Sort this CSV by the date column, oldest first. Keep the header row at top.
- Reformat this XML so each element is on its own line and properly indented.

Convert — change format

- *Convert this CSV to JSON. Each row becomes an object; the first row contains the field names.*
- *Convert this JSON to CSV. The array contains objects with the same keys; use those as headers.*

- *Convert this XML to JSON. Element names become object keys; attributes become properties prefixed with @.*

Split — produce multiple files

- These create one child job per output. Each child gets its own filename.
- *This file contains a JSON array of customer records. Split it into one JSON file per customer, named customer_1.json, customer_2.json, etc. Each file should contain just that one customer object.*
- *Split this CSV into one CSV per region. Name each file by the region (e.g. north.csv, south.csv) and include the original header row in each.*
- *This XML has multiple <order> elements. Output one XML file per order, named order_<orderid>.xml.*

PDF and image tasks (Provider = Anthropic only)

- The model sees the file's visual content, so prompts can refer to layout, images, or visual elements.
- *Extract the invoice number, total amount, and due date from this invoice PDF.*
- *What is the company logo at the top-left of this document? Return just the company name.*
- *Read the handwritten notes in the margin and transcribe them.*
- *Identify the type of form on this scan: tax form, application, or contract. Return just one of those three words.*

Output behavior

Extract operations

The script attaches a Switch JSON dataset named “AiResult” to the input job. The dataset's structure depends on what the model returned:

- Single value - `{"result": "<the value>"}`
- Multiple named values - `{"results": [{"filename": "po_number.txt", "content": "PO-44823"}, ...]}`

The input file passes through to Success unchanged. Reference the values downstream:

- For single-value extracts:
  `[Metadata.Text:Path="result",Dataset="AiResult",Model="JSON"]`

- For multi-value extracts
  `[Metadata.Text:Path="results[0].content",Dataset="AiResult",Model="JSON"]`

Reformat / convert operations

The output file replaces the input file. The job continues with the same name (or a new extension if format changed) to Success. No dataset is attached — the data is the file itself.

Split operations

Each output becomes a child job sent to Success. The parent job is disposed (sent to Null). Filenames are whatever the model chose — the prompt controls this.

Troubleshooting

"Job goes to Error every time" with the local model: small local models sometimes produce malformed JSON, especially on novel prompts. Try Anthropic with the same prompt to confirm the prompt itself is clear, then either tighten the prompt or stay on Anthropic for that flow.

"Model returns the right value but with extra commentary": add explicit format directions to the prompt — "Return ONLY the value, no other text" or "Return as a JSON string with no markdown fences".

"PDFs go to Problem Jobs with a 'requires Anthropic' message": that's expected. The local model is text-only. Either set Provider to Anthropic, or pre-process the PDF to text with an upstream flow element.

"First job after restart takes minutes" (local model): that's the model loading. Subsequent jobs in the same Switch service session reuse the loaded model and complete in seconds. The flow element is configured for Serialized execution so jobs queue rather than overlap.

"Anthropic returns 529 errors": Anthropic's API is overloaded. The script retries automatically with exponential backoff; if it still fails, the job goes to Error with the upstream message. Try again later, or fall back to the local model for non-vision tasks.

"The dataset on Success doesn't have the field I expected": The model may have used different field names than your downstream variable expression assumes. Either tighten the prompt to specify field names ("Return as JSON with keys 'invoice_number' and 'total'") or change the variable expression to match what the model produced.

Cost and performance notes

Local model: zero per-job cost, ~5-30 seconds per job depending on output length. First job after service start takes longer (model load).

Anthropic Sonnet 4.5: about $3 per million input tokens, $15 per million output tokens as of this writing. A typical extract prompt is under 1000 tokens in + 100 tokens out — fractions of a cent per job. PDFs are charged by page; budget about 1500 input tokens per page.

For high-volume flows, the local model is essentially free but slower; Anthropic is faster and more accurate but has running cost. Many flows mix both — local for bulk text processing, Anthropic for PDF/image steps.