ChatGPT

Estado Actual del Pipeline de Normalización de Productos

Descripción General del Flujo de Procesamiento de Datos Post-Scraping

Después de completar el **scraping** de datos desde los sitios de retail, el sistema ejecuta un pipeline de procesamiento **post-scraping** que normaliza y categoriza los productos antes de almacenarlos en la base de datos. En términos generales, el flujo consiste en los siguientes pasos:

1. **Carga de archivos JSON de scraping:** Los scrapers guardan los resultados en archivos JSON (`busqueda_*.json` y `rotation_*.json`) que contienen los productos encontrados y metadatos de la búsqueda. Un componente monitor verifica la llegada o modificación de estos archivos y los marca para procesamiento. Solo se consideran archivos recientes (por ejemplo, últimas 2 horas) para asegurar datos frescos [1] [2].

2. **Normalización de campos:** Un proceso integrador lee cada archivo JSON y estandariza la estructura de los datos de producto. Se crea un formato uniforme con campos clave: `product_id` (identificador único por retailer, generalmente el código de producto), `name` (nombre del producto), `brand` (marca), `retailer` (tienda origen), `product_link` (URL del producto), `search_term` (término de búsqueda usado), `scraped_at` (timestamp de scrapeo), y un sub-objeto `prices` con los distintos precios capturados [3]. Este último unifica posibles variantes de precio (precio normal, oferta internet, precio con tarjeta, etc.) según el retailer [4] [5]. Además, se incorporan metadatos del scrapeo como el número de página (`page_scraped`) y el número de ciclo (`ciclo_number`) en cada producto, así como un identificador de lote o batch para trazabilidad [6].

3. **Filtrado y validación inicial:** Durante la normalización, se validan campos obligatorios básicos. Por ejemplo, si un producto no tiene `product_id` (código) **o** nombre, se descarta para evitar registros incompletos [7]. Esto previene procesar entradas vacías o mal formateadas. También se detecta el retailer origen del producto (París, Ripley, Falabella, etc.) ya sea a partir de los metadatos o del nombre de archivo, para asignarlo correctamente al campo `retailer` [8] [9].

4. **Categorización con IA:** Los productos normalizados pasan a un pipeline de **normalización y categorización** potenciado por IA. Este pipeline (basado en `AINormalizationPipelineV2`) toma cada producto e intenta clasificarlo en la **taxonomía estándar de categorías (nivel 1, 2, 3)** del sistema. Internamente funciona así:

5. **Cache y duplicados:** Antes de invocar un modelo de IA, el sistema busca si ya existe ese producto en la base de datos o en cache. Usa como clave el `product_id` (código único en el retailer) y el retailer. Si el producto ya fue procesado previamente, se recupera su categorización almacenada y se reutiliza, marcándolo como *cache hit* [10] [11]. Esto actúa como detección de duplicados a nivel de procesamiento: evita recategorizar dos veces el mismo producto y ahorra costos de cómputo. Gracias a esta verificación, si por ejemplo un producto aparece nuevamente

en otro ciclo de scraping (o incluso duplicado en el mismo JSON), no se vuelve a insertar duplicadamente; las **constraints únicas** en BD (por ejemplo, en la tabla de productos normalizados) garantizan que un mismo producto por retailer no se duplique [12].

6. **Normalización de texto:** Si no hay resultado cacheado, el pipeline realiza una limpieza del texto del nombre, marca y descripción del producto (normalización de mayúsculas, tildes, eliminación de caracteres extraños, etc.) para presentar datos consistentes al modelo de IA [13].
7. **Categorización por reglas (templates):** Como primer paso inteligente, el sistema intenta asignar categorías mediante plantillas predefinidas. Un componente denominado *CategoryNormalizer* contiene ~121 plantillas de categorías con palabras clave comunes [14]. Si el nombre o marca concuerda con una plantilla con suficiente confianza (>30%), se asigna directamente esa categoría sin llamar al modelo GPT, marcando el método usado como `"template"` y ahorrando tiempo [15] [16].
8. **Modelo GPT para clasificación:** Si no hubo coincidencia con templates (o si la confianza es baja), se construye un **prompt optimizado** y se consulta a un modelo GPT. La configuración actual utiliza un modelo rápido apodado **GPT-5 Nano** como principal, con un fallback a **GPT-4o-mini** (versión reducida de GPT-4) en caso de fallo o baja confianza [17]. El prompt incluye el contexto del producto (marca + nombre, y a veces descripción abreviada) junto con instrucciones estrictas: se proporciona una lista de categorías válidas de nivel 1 y 2 (*whitelist*) y se exige que el modelo devuelva un JSON con campos `categoria_n1`, `categoria_n2`, `categoria_n3`, `marca` detectada, `modelo` detectado, `atributos` y un score de `confianza` [18] [19]. Gracias a esta whitelist dinámica (derivada de la base de datos de categorías válidas), el modelo solo puede elegir combinaciones permitidas, reduciendo errores de clasificación.
9. **Post-procesamiento e integridad de la categorización:** Una vez obtenida la respuesta del modelo, el sistema valida la salida. Aplica un **guardia de integridad** (`DataIntegrityGuard`) que verifica que la categoría devuelta existe en la taxonomía esperada y que cumple umbrales mínimos de confianza [20] [21]. Estos umbrales pueden variar por dominio; por ejemplo, para **Smartphones** y **Notebooks** se exige $\geq$0.85 de confianza, mientras que para **Televisores** o **Perfumes** se permite desde 0.80 [22]. También verifica la consistencia de marca y modelo detectados: si el modelo de IA sugiere una marca distinta a la proporcionada originalmente, o si faltan campos esenciales, el resultado puede marcarse con `necesita_revision = True` para revisión manual. Si la salida no cumple los criterios (p. ej. confianza muy baja o categorías vacías), el pipeline puede intentar un *fallback* automáticamente (invocar el modelo alternativo GPT-4o-mini) [23]. En casos especiales, hay lógica adicional: para productos de ciertas categorías complejas se realiza una extracción técnica extra y re-categorización. Por ejemplo, para **Televisores** se extraen especificaciones como tamaño de pantalla o resolución mediante un extractor especializado y se reintenta la categorización incorporando esos datos [24]; similarmente con **Perfumes** podría ajustarse contexto (familia olfativa, género, etc.) [25]. Esta doble pasada mejora la precisión en dominios donde los atributos técnicos son clave para la categoría.

10. **Resultado de categorización:** Tras este proceso, cada producto obtiene un resultado consistente: categoría de nivel 1 y 2 (y a veces nivel 3 si aplica), una marca normalizada detectada (puede coincidir con la original o corregida si venía en el nombre), un modelo o versión si se pudo extraer, y un diccionario de **atributos técnicos** relevantes (ej. pantalla: 55", memoria: 128GB, género: "Hombre" en perfumes, etc.). Todo ello acompañado de un valor de confianza y etiquetas sobre qué modelo de IA se usó y si el resultado provino de caché o fue generado en vivo.

11. **Integración en la base de datos:** Con los productos categorizados, el sistema procede a almacenarlos o actualizar la base de datos central:

12. Se registra cada **producto normalizado** en la tabla correspondiente (usualmente `normalized_products`), guardando retailer, nombre original, marca original, categorías asignadas, marca/modelo detectados, atributos JSON, nivel de confianza y modelo IA utilizado [26] [27]. Si ese producto (definido por código + retailer) ya existía en la tabla, se actualiza su timestamp y confianza si la nueva es distinta, evitando duplicados gracias a `ON CONFLICT` en (`product_id`, `retailer`) [28].
13. Se unifica el **producto maestro** en la tabla de productos global (`products`). Si el producto es **nuevo** (no coincide con ninguno existente), se crea un nuevo registro maestro asignándole un identificador interno (`product_id`) y un `master_sku` único generado a partir de la marca, modelo y atributos clave [29] [30]. Este SKU compuesto facilita identificar el producto de forma homogénea (ej. `SAMSUNG_GALAXY_S20_128GB` para un smartphone Samsung Galaxy con 128GB). Si por el contrario el producto parece corresponder a uno existente, se hace **matching** automático: el integrador busca en la tabla de productos algún item con marca/modelo similar y misma categoría [31]. Usa una métrica de similitud que combina distancia de marca, modelo y atributos técnicos comunes [32] [33]. Si encuentra un match con alta confianza (ej. $\geq$85%), asume que es el mismo producto en distinto retailer. En ese caso, **no crea un nuevo master**, sino que reutiliza el existente.
14. Se actualizan las entradas de **precios actuales** por retailer. Cada vez que se procesa un producto, se inserta o actualiza la tabla `current_prices` con el precio (normal, internet, tarjeta) vigente de ese producto para ese retailer [34] [35]. Esta tabla de precios actuales tiene constraints únicas por (`product_id`, `retailer`) igual que la de productos, de modo que si el mismo producto-retailer ya estaba, se hace update en vez de duplicar. Además, se mantiene un campo de `last_checked_at` (última fecha de chequeo) y `last_changed_at` (última fecha en que el precio cambió). La lógica compara el precio nuevo vs el anterior; si cambió, actualiza `last_changed_at`, si es igual, lo deja [36]. También lleva un contador de revisiones (`check_count` incrementa) [37]. De esta forma, el sistema **no duplica productos ni precios** ya existentes, sino que detecta si es un producto nuevo o simplemente un nuevo registro de precio para un producto ya conocido, consolidando la información.
15. Si un producto nuevo de un retailer se emparejó a un producto maestro existente (p.ej. vimos que un televisor X de marca Y ya estaba en la base bajo otro retailer), entonces el integrador agrega una entrada de **mapeo** en `product_mappings` para vincular el código externo del retailer con nuestro ID interno maestro [38] [39]. Así, la próxima vez que se vea ese `product_code` con ese retailer, el sistema podría identificarlo directamente. *(Nota: Actualmente, la búsqueda de duplicados se apoya más en similitud de texto que en este mapping en la fase de IA; una posible mejora sería usar este mapping para saltar la categorización vía IA de forma más directa para productos ya vistos.)*
16. **Detección de cambios de precio:** Tras actualizar `current_prices`, el sistema evalúa si hubo cambios significativos en los precios respecto al valor anterior [40] [41]. Por ejemplo, si el precio bajó más de un 15% (umbral de promoción) o subió/bajó más de un 5% o $1000 CLP, se considera significativo [41]. Tales eventos se registran en la tabla de `price_changes` con detalle del antes y después [42] [43]. Esto alimenta el sistema de **alertas automáticas** para notificar promociones o alzas importantes.
17. **Snapshot histórico nocturno:** Al final del día (23:59), un proceso toma un *snapshot* de todos los precios vigentes: inserta en la tabla `price_history` los precios de `current_prices` con la fecha del día [44] [45]. De este modo se construye un histórico diario de precios por producto y retailer, manteniendo la tabla `current_prices` solo con lo más reciente. Este snapshot se programa automáticamente cada noche y asegura que al resetear el monitoreo cada día tengamos registro histórico disponible.

En resumen, tras el scraping inicial, el sistema **normaliza los datos**, **los enriquece con categorización por IA**, filtra duplicados, y finalmente **almacena la información estructurada** en varias tablas

normalizadas (productos, categorías asignadas, precios actuales e históricos, etc.), preparándola para consumo en dashboards o análisis. A continuación detallamos puntos específicos de este pipeline, incluyendo la estructura de los archivos JSON de entrada, la lógica de normalización/categorización, manejo de ciclos de scraping y consideraciones de integridad.

Estructura de Datos en Archivos de Búsqueda (`busqueda_*.json`)

Los archivos `busqueda_...json` representan la salida de scrapers de búsqueda continua en diferentes retailers. Cada archivo usualmente corresponde a la búsqueda de un término o categoría en un retailer específico durante un ciclo de scraping. Están organizados en dos secciones: un objeto `metadata` con información general de la ejecución, y una lista `products` con los productos encontrados.

- **Metadatos (`metadata`):** Incluyen campos como:
- `scraped_at`: Fecha y hora en que se realizó la extracción de datos.
- `search_term`: El término de búsqueda o categoría utilizado (por ejemplo "notebook", "perfumes").
- `search_name`: Nombre descriptivo de la búsqueda, a veces con emojis o mayúsculas para identificar la categoría (ej. "💻 Notebooks", "Perfumes 🌸").
- `search_key`: Clave interna o simplificada del término (ej. "notebook", "perfume").
- `base_url`: (En scrapers continuos) URL de la página de búsqueda que se scrapeó inicialmente 【12†】. Por ejemplo, en París se incluye la URL con query y parámetros de ubicación (`commune`).
- `total_products`: Cantidad total de productos detectados en esa búsqueda.
- `pages_scraped`: Número de páginas de resultados recorridas por el scraper.
- `ciclo_number`: Número de ciclo de scraping al que pertenece este archivo. Es un contador incremental que permite distinguir distintas ejecuciones o iteraciones de búsqueda para la misma categoría.
- `commune`: (Solo en algunos casos, e.g. París) Identifica la región/tienda usada en la búsqueda (ej. código de comuna para precios regionales).

- `scraper`: Nombre o descripción del scraper que generó el archivo. Incluye el retailer y modalidad. Por ejemplo: `"Falabella Scraper CONTINUO con BÚSQUEDAS 🔍"` indica scraping continuo en Falabella, mientras `"París Scraper CONTINUO con BÚSQUEDAS 🔍"` indica París【11†】. Este campo ayuda a detectar el retailer (París, Ripley, Falabella) en caso que el sistema de integración lo necesite.

- **Productos (`products`):** Es una lista de objetos, cada uno representando un producto listado en la búsqueda. Los campos presentes pueden variar ligeramente según el retailer, pero en general incluyen:

- `product_code`: Identificador único del producto en el sitio (SKU, código interno, etc.). Por ejemplo, en Falabella suele ser un SKU numérico o alfanumérico, en París un SKU numérico.
- `name`: Nombre completo del producto tal como aparece en la página (incluye marca y modelo normalmente).
- `brand`: Marca del producto. En scrapers bien estructurados, este se extrae aparte cuando la página lo provee (algunas páginas muestran la marca por separado; de lo contrario, a veces se obtiene parsing del nombre).

- **Precios:** Se captura normalmente el precio de lista y cualquier oferta:
    - `normal_price` / `normal_price_text`: Precio normal (y su texto formateado con moneda).
    - `offer_price` o `internet_price` / su texto: Precio online o de oferta (si existe un descuento internet).
    - `card_price` / su texto: Precio con tarjeta de crédito de la tienda (si aplica, p. ej. "precio con CMR" en Falabella).
    - `original_price` / texto: Precio original sin descuento (en algunos casos se entrega aparte cuando hay promoción).

  Nota: En los archivos de búsqueda continua, es común tener tanto el valor numérico como la versión en texto (con símbolo peso y puntos) de cada precio [46]. Por ejemplo: `"normal_price": 59990` y `"normal_price_text": "$59.990"`. Esto facilita tanto cálculos como mostrar el precio formateado. - `product_link`: URL del producto individual (en Falabella, París, etc. suele ser un link completo a la página del producto). - `page_scraped`: Número de página de resultados donde apareció este producto (1 para la primera página, 2 para la segunda, etc.). - `search_term`, `search_name`, `search_key`: Repetidos dentro de cada producto para referencia (útiles si se procesa cada producto aisladamente más adelante). - `scraped_at`: Timestamp de extracción de ese producto (puede repetirse el del metadata si se extrajo en el mismo momento, o podría variar levemente si el scraper anota tiempo por página). - `ciclo_number`: Reiteración del ciclo, igual que en metadata, incluido en cada producto para trazabilidad. - Campos específicos de contexto: por ejemplo, `commune` en el caso de París (indicando región de precios para ese producto).

Integridad de datos en estos JSON: En general, los archivos de **búsqueda continua** presentan información completa por producto, adecuada para el pipeline: - Cada producto típicamente tiene un `product_code` no vacío y un `name` descriptivo. Esto es crucial ya que el integrador solo procesa productos con ambos campos presentes [7]. - Los precios incluyen valores numéricos, lo que permite al integrador calcular cambios y almacenarlos correctamente. El pipeline de normalización extrae estos valores numéricos al crear el dict `prices` estándar, garantizando que tengamos números para la base de datos y no solo texto [47]. - El campo `brand` suele estar poblado, lo cual ayuda al modelo de IA y al matching entre retailers. No obstante, si algún scraper no lo llenara, el sistema podría aún deducir la marca en la etapa de IA buscando la primera palabra del nombre, o apoyándose en su lista de marcas conocidas (lexicón de marcas). - Puede haber **productos duplicados** listados en un mismo JSON. Por ejemplo, en ciertos casos Falabella devolvió entradas repetidas del mismo producto en múltiples páginas o listados patrocinados, resultando en duplicados. Estos duplicados internos no se filtran en esta etapa, por lo que el archivo puede contener varios objetos con el mismo `product_code`. Sin embargo, como se explicó, el pipeline más adelante los detecta (cache/DB) y evita procesarlos dos veces o duplicarlos en la base de datos. *Un punto de mejora* sería filtrar duplicados dentro del JSON mismo antes de la categorización, para ahorrar tiempo, aunque funcionalmente el resultado final no duplica datos.

En suma, los archivos `busqueda_*.json` proveen un **snapshot estructurado** de los productos encontrados por búsqueda, rico en información (IDs, marcas, precios detallados). Sirven como insumo directo para la normalización con IA.

Estructura de Datos en Archivos de Rotación (`rotation_*.json`)

Los archivos `rotation_...json` corresponden a ejecuciones de scraping con **"rotación de proxies"** u otros métodos especiales para obtener datos donde el scraper continuo tradicional enfrenta limitaciones. En el contexto actual, se observa que estos archivos fueron generados principalmente para ciertas categorías en el retailer **Ripley**, identificados por el campo `scraper` en metadata que indica *"Proxy Rotation "*. La rotación de proxies implica que el scraper cambió de dirección IP en cada solicitud, permitiendo así extraer más páginas de resultados sin ser bloqueado.

La estructura general de estos JSON es similar (tienen `metadata` y `products`), pero presentan diferencias notables en el contenido de campos:

- **Metadatos (`metadata`):** Además de los campos comunes (`scraped_at`, `search_term`, `search_name`, `total_products`, `pages_scraped`, `ciclo_number`), incluyen indicadores de la rotación, por ejemplo:
- `proxy_rotation`: `true` para señalar que se utilizó rotación de IPs.
- `unique_ips_used`: número de IPs distintas empleadas durante el scraping (por ejemplo 10 si se usó una IP diferente por página).
- `scraper`: Describe el modo, e.g. `"Ripley Proxy Rotation "`, dejando claro que es Ripley con proxies rotativos.

- No suelen incluir `base_url` ni `commune`, ya que probablemente usaron un endpoint distinto o múltiples entradas.

- **Productos (`products`):** Listan los productos obtenidos, con campos:

- `name`: Nombre del producto (con marca incluida en la cadena normalmente).
- `product_code`: En los casos analizados, este campo aparece **vacío** (`""`). Es posible que el método de scraping utilizado no obtuviera el SKU o identificador, o que la página de resultado de Ripley no exponga fácilmente un código. Esto es problemático porque la ausencia de código dificulta el tracking único; el integrador termina usando un identificador sintético temporal en su lugar si este falta.
- **Precios:** En los archivos de rotación, observamos que solo se incluyeron los campos de texto de los precios, por ejemplo:
  - `ripley_price_text`, `card_price_text`, `normal_price_text` con valores como `"$158.990"`. No se incluyeron los campos numéricos correspondientes (`ripley_price`, `card_price`, etc.), quedando los precios solo como cadenas formateadas.
  - Además, a veces estos tres precios de texto son iguales, lo que sugiere que el producto no tenía descuento (precio normal = precio con tarjeta en ese ejemplo). Aún así, faltan los valores numéricos.
- `product_link`: En las muestras, este campo viene vacío también. Es probable que, para reducir carga o complejidad, el scraper rotativo no almacenó el enlace al producto.
- `brand`: Este campo **no aparece explícitamente** en rotation (al menos en los primeros productos). La marca queda únicamente como parte del `name` (ej. "SET **LANCÔME** TRÉSOR..."). A diferencia de la búsqueda continua donde `brand` podía extraerse, aquí parece no haberse hecho ese parsing.
- `page_scraped`: presente, indicando la página de resultado.

- `proxy_ip` y `proxy_port`: Cada producto lleva anotada la IP y puerto proxy que se usó para extraer esa página, lo que es útil para debug de rotación pero no para la lógica de negocio en sí.
- Los demás campos (`search_term`, `search_name`, `search_key`, `ciclo_number`, `scraped_at`) se repiten igual que en busqueda.

Robustez y lógica de este enfoque de rotación: Por un lado, la rotación de proxies **logró extraer más productos** que el método continuo estándar. Por ejemplo, para la categoría Perfumes en Ripley se obtuvieron 478 productos en 10 páginas, mientras que quizá el scraper tradicional se veía limitado a menos páginas antes de bloqueo. Esto indica que la técnica fue exitosa en términos de cobertura de datos (se alcanzó posiblemente el total de productos listados). De igual forma, para Televisores ("smartv") en Ripley se obtuvieron ~474 productos con rotación.

Sin embargo, la **integridad de los datos en estos archivos rotation es inferior** comparada con busqueda: - **Campos faltantes o incompletos:** La ausencia de `product_code` y `brand` explícitos significa que al normalizar, el sistema no tendrá identificador único ni la marca separada. De hecho, el procesador de scrapers descarta entradas sin código o nombre [7], lo que implica que todos estos productos con `product_code` vacío serían inicialmente omitidos de no intervenir. (En la práctica, para integrarlos, se implementó una solución: durante el `process_scrapers_batch` se asigna un ID ficticio al producto si no tiene uno, por ejemplo `"scraper_5_Ripley"` para quinto producto de Ripley [48]. Esto permite procesarlos, aunque ese ID temporal no sirve para detectar duplicados en ejecuciones futuras). - **Precios solo en texto:** El pipeline de extracción de precios espera campos numéricos para armar el dict de precios. En rotación, como no los hay, podría resultar en `prices = {}` vacío para todos. En la integración actual, para solventar esto, se usa una función que extrae el "mejor precio" buscando cualquiera de los campos numéricos conocidos [47]. En ausencia de ellos, retorna 0 como precio [49]. Es decir, estos productos podrían entrar con precio 0 en la normalización, lo cual obviamente no es deseable. Una mejora necesaria sería parsear el texto del precio (`"$158.990" -> 158990`) antes de la carga, para rellenar al menos `normal_price`. - **Links vacíos:** No tener `product_link` impide luego crear la referencia clickeable al producto desde sistemas internos o verificar la información posteriormente. Sería conveniente extraerla. - **Marca no separada:** Esto recarga la tarea del modelo de IA, ya que GPT deberá inferir la marca del texto del nombre. Afortunadamente, como suele estar al inicio del nombre en mayúsculas, es factible que lo detecte (y lo devuelve en `marca_detectada`). Pero un enfoque más robusto sería extraerla con una lista de marcas conocidas en el scraper mismo.

En conclusión, los archivos `rotation_*.json` cumplen su objetivo de **ampliar la cobertura** de productos scrapteados en sitios difíciles, a costa de **calidad de datos**. La lógica de "prompts" aquí realmente se refiere a la estrategia de rotación (cada *request* actuando como un *prompt* al servidor con IP nueva). No involucra prompts de IA, sino tácticas de scraping. La robustez de esta técnica es buena para conseguir todos los ítems, pero los datos requerirían post-procesamiento adicional para equipararse en calidad a los de `busqueda`. Actualmente, el pipeline los procesa con ciertas adaptaciones, pero se identifican varios puntos de mejora (detallados más adelante) para que la **integración de rotación sea más consistente**, como normalizar esos campos faltantes.

Proceso de Normalización y Categorización de Productos

Esta etapa es el corazón del sistema, donde los datos crudos de los scrapers se convierten en información estructurada y enriquecida lista para base de datos. Abarca dos subprocesos: **normalización de campos** (estandarización de formato) y **categorización por IA** (asignación de

categorías y atributos). Ya describimos en la visión general cómo funciona, por lo que aquí resumiremos los aspectos clave de forma más esquemática:

- **Normalización de formato:** Al leer cada producto desde el JSON, el procesador construye un objeto unificado:
- Convierte nombres de campos propios de cada retailer al esquema estándar interno. Por ejemplo, en los JSON de Falabella, Paris, etc., se usan distintos nombres para precios o IDs; la normalización mapea `product_code` -> `product_id` [50], toma los campos de precio (`normal_price`, `card_price`, etc.) y los mete bajo un diccionario `prices` [51], entre otros.
- Añade campos calculados: el retailer (derivado del contexto), el `batch_id` o identificador de lote de procesamiento, y agrupa metadatos de origen en `scraper_metadata` (ciclo, página, search_name) [6]. De este modo, se conserva rastro de dónde vino el producto incluso después de entrar a la IA.

- Verifica requisitos mínimos: si falta nombre o ID tras el mapeo, devuelve `None` descartando el producto [7]. Si todo está bien, produce un diccionario normalizado. *Ejemplo:* un producto "Notebook Dell XPS 13" de Falabella con SKU 12345 y precios $1.000.000 (normal) y $900.000 (internet) se convertirá en:

```
{
  "product_id": "12345",
  "name": "Notebook Dell XPS 13",
  "brand": "Dell",
  "retailer": "Falabella",
  "search_term": "notebook",
  "scraped_at": "2025-09-07 11:22:55",
  "product_link": "https://...Dell-XPS13",
  "prices": {
     "normal_price": 1000000,
     "normal_price_text": "$1.000.000",
     "original_price": 1000000,
     "original_price_text": "$1.000.000",
     "card_price": 900000,
     "card_price_text": "$900.000"
  },
  "scraper_metadata": {
     "page_scraped": 1,
     "ciclo_number": 1,
     "search_name": "💻 Notebooks",
     "batch_id": "scrapers_20250907_112255"
  }
}
```

Esto permite al siguiente componente (IA) trabajar con una estructura consistente sin depender de la fuente original.

- **Categorización por IA:** Una vez normalizados, los productos se procesan individualmente a través del pipeline de IA:

- Se crea un objeto de datos (`ProductData`) con los campos necesarios (id, name, brand, description, price, retailer, url) [52]. La descripción en este caso suele venir vacía porque en la búsqueda se tiene solo título; sin embargo, podría enriquecerse si se decidiera scrapear detalles del producto más adelante.
- **Búsqueda en caches/BD:** Como mencionado, antes de llamar al modelo GPT se hace una búsqueda en:
    1. **Tabla `normalized_products` (BD):** ¿Ya existe este `product_id` + `retailer` normalizado? Si sí, se carga la categorización previa directamente [53] [54]. Esto cubre duplicados exactos ya procesados en corridas anteriores (persistidos en BD).
    2. **Cache de IA (BD):** Si no hay entrada definitiva, se busca en una tabla de cache de categorización si existe una respuesta de IA guardada para este mismo producto (se utiliza una clave hash del contenido/taxonomía) [55] [56]. Si se halla y sigue siendo válida según las reglas de integridad, se usa esa respuesta cacheada para evitar invocar el modelo nuevamente. Esta cache de segundo nivel retiene resultados recientes o frecuentes y aplica la misma validación de integridad que tendría una respuesta nueva (ej. no aceptar categorías inválidas aunque provinieran de cache).
    3. **Cache local en memoria:** Si tampoco, se revisa una cache en memoria durante la ejecución (útil en caso de duplicados dentro del mismo lote) [57].
    4. **Si ninguna coincide, procede con IA en vivo.**
- **Consulta IA y validación:** Se prepara el prompt como detallamos (contexto y reglas) y se invoca GPT-5 nano vía API. Si falla o su resultado no pasa validaciones (por ejemplo, devuelve algo fuera de formato JSON esperado), se utiliza GPT-4o-mini como fallback [17]. La respuesta del modelo (ya sea nano o fallback) se valida mediante `DataIntegrityGuard.enforce()`, que corrige pequeños desvíos (p.ej. formatea correctamente campos de categoría, recorta strings extraños) y determina si cumple condiciones de aceptación [58] [59]. Si algo crítico falta, a veces se reintenta con más contexto (como explicado con Televisores/Perfumes). Al final, se obtiene un objeto `CategorizationResult` con las categorías limpias y campos extraídos [56] [60].
- **Incorporación de resultado:** El integrador registra la llamada IA en sus estadísticas (contando uso de nano vs fallback, hits de cache, tokens gastados, costo estimado, etc.). Luego el flujo sigue hacia el matching e inserción en BD como ya se describió.

Este proceso de normalizar + categorizar es altamente automatizado y diseñado para robustez. Combina **reglas determinísticas** (templates, whitelists, integridad) con **IA generativa**, logrando precisión con control de calidad: - El uso de *whitelist de categorías* garantiza que el sistema mantiene las categorías dentro de un conjunto predefinido (30 categorías de alto nivel y sus subdivisiones estándar) [61], evitando resultados fuera de dominio. - Las marcas detectadas se normalizan (por ej. si GPT devuelve "LG Electronics" en vez de "LG", internamente podría unificarse) y lo mismo con modelos, mediante un proceso de homogeneización en BD. - Atributos técnicos extraídos son guardados en JSON para futuras referencias (y también usados en la lógica de matching de productos para comparar similitud). - La integración con la base de datos asegura que si mañana se scrapea el mismo producto, el pipeline se da cuenta rápidamente (vía caches y constraints) y simplemente actualiza precios en vez de duplicar trabajo.

Como resultado, al finalizar la categorización, cada producto scrapeado no solo tiene sus **datos originales estructurados**, sino también **nueva información semántica**: en qué categoría general cae, qué subcategoría específica es, cuál es su marca canonical, si se pudo inferir modelo o características, etc. Esto enriquece enormemente la base de datos para usos analíticos y de comparación.

Manejo de Ciclos de Scraping y Rotaciones

En el sistema, el concepto de **ciclo** (`ciclo_number`) y de **rotación** se utilizan para coordinar y supervisar la recolección continua de datos:

- **Ciclos de scraping:** Representan iteraciones sucesivas de ejecuciones de scraping para una determinada categoría o conjunto de categorías. En la práctica actual, como se observa en los archivos, el número de ciclo incrementa cada vez que se realiza una nueva búsqueda en un retailer para la misma categoría. Por ejemplo, `busqueda_notebook_ciclo001_...json` seguido de `busqueda_notebook_ciclo002_...json` indica dos ejecuciones en días/horas diferentes (o en retailers diferentes, ver abajo) para la búsqueda de "notebook". Este número se incluye tanto en el nombre del archivo como dentro del JSON (metadata y cada producto) para identificar la fuente temporal de los datos.

  En la integración con el orquestador diario, es probable que cada ciclo corresponda a un lapso (por ejemplo, cada hora entre 6:00 y 23:00 se lanza un ciclo de scraping [62]). Sin embargo, dado que se scrapearon **varios retailers por categoría**, la enumeración de ciclos puede tener dos interpretaciones: - En algunos casos parece haberse numerado por retailer; por ejemplo, la documentación indica que en un ciclo único se podrían generar archivos `busqueda_smartphone_001` (París), `002` (Ripley), `003` (Falabella) en paralelo [63]. Es decir, el sufijo numérico distinguiría retailers en la misma ronda. - En los datos actuales, sin embargo, vimos `ciclo001` para Falabella Notebooks y `ciclo002` para Paris Notebooks (mismo término), lo que sugiere que a veces enumeraron secuencialmente a medida que cambiaban de retailer. Es posible que la implementación haya variado: inicialmente quizás se hicieron secuencial, luego en paralelo.

  En cualquier caso, el propósito del ciclo es **marcar la iteración**. Esto permite, por ejemplo, comparar los resultados del ciclo 1 vs ciclo 2 (¿hubo más productos? ¿cambios de precio?), o reiniciar contadores al iniciar un nuevo día (como indica el orquestador, a medianoche se resetea el estado y arranca ciclo fresco [62]).

  El sistema de pipeline utiliza `ciclo_number` principalmente para logging y auditoría. Internamente, no cambia la lógica de normalización, pero queda registrado en `scraper_metadata` de cada producto procesado. Así, en la base de datos podríamos guardar la info de qué ciclo se insertó originalmente un producto, útil para análisis de frescura de datos o debugging (si un producto vino de un ciclo anómalo, etc.).

- **Rotaciones:** Como ya explicamos en la sección de archivos `rotation`, las rotaciones son una *estrategia de scraping* distinta, no un ciclo aparte en términos de orquestación (aunque en archivos se ve como ciclo 006, 010, etc., porque se ejecutaron en determinadas iteraciones). En esencia, la rotación es **parte de un ciclo** donde para cierto retailer se decide usar proxies rotativos para obtener los datos.

  La necesidad de rotación surge por restricciones anti-scraping. En retailers como Ripley, probablemente después de unas pocas páginas la IP era bloqueada o se devolvían menos resultados. La solución fue implementar un scraper que cada vez que pide una página de resultados cambia su IP (tomando de un pool de proxies). Esto se transparenta en los metadatos (`proxy_rotation: true`, `unique_ips_used: X`). Durante esa rotación, el scraper puede tardar más debido al cambio de conexión, pero logra enumerar todas las páginas (en nuestros ejemplos, 10 páginas completas). Es por ello que vemos, por ejemplo, un salto de

`ciclo_number`: para Perfumes quizás los ciclos 1-9 fueron otros retailers/modo, y en el 10 se decidió hacer rotación en Ripley, resultando en `rotation_perfume_c010...`.

Desde la perspectiva del pipeline post-scraping, **no hay diferencias fundamentales**: los archivos rotation se procesan igualmente si se les pasa en la ruta de entrada. La única diferencia es que requieren los ajustes mencionados (manejo de falta de código, etc.). El pipeline en sí no distingue si un producto vino de rotación o no para la IA; simplemente lo ve como otro JSON de entrada.

Sin embargo, a nivel de orquestación, las rotaciones podrían planificarse con menos frecuencia o en momentos específicos (quizá de noche, para no saturar proxies). Podría haber un plan como: *"Ejecutar scraping normal cada hora, pero cada cierto número de ciclos, hacer una rotación proxy para retailers complicados"*.

Vale destacar que la rotación en scraping es costosa en recursos y puede aún así ser frágil (depende de la calidad de los proxies). En los datos actuales funcionó bien (10 IPs para 10 páginas, 100% cobertura). Un monitoreo de la integridad muestra que se obtuvo ~471 productos únicos en Smart TV Ripley y ~471 en Perfumes Ripley, lo que sugiere que casi todos los productos fueron capturados. A pesar de la ausencia de algunos campos, la **integridad básica** (cantidad de resultados) fue superior usando rotación versus no usarla.

En resumen, **los ciclos** se manejan como identificadores de iteración continua (ya sea por hora o por retailer dentro de una hora, según la configuración), mientras que **las rotaciones** son un método especial aplicado dentro de ciertos ciclos para mejorar la cobertura de datos. Para el equipo es importante entender ambos conceptos: los ciclos permiten organizar la ejecución periódica y no procesar archivos antiguos (el procesador solo toma los de las últimas X horas, filtrando por timestamp [64]), y las rotaciones son una táctica necesaria para ciertos sitios, pero cuyos datos requieren atención especial en la post-normalización.

Esquema del Flujo hasta el Depósito en Base de Datos

Integrando todo lo anterior, podemos delinear el **pipeline completo** desde que se genera un archivo de scraper hasta que los datos llegan a la base de datos, excluyendo la mecánica del scraping en sí. En forma de pasos secuenciales:

1. **Generación de archivo JSON:** Un scraper (por retailer y categoría) produce un archivo `busqueda_<categoria>_cicloXXX_fecha.json` o `rotation_<categoria>_cicloXXX_fecha.json` en una carpeta monitoreada.
2. **Monitoreo y disparo de procesamiento:** El monitor de archivos detecta la creación/modificación [65] y registra la llegada del archivo (nombre, total de productos, etc.) [66] [67]. Este monitor es *no intrusivo*, solo observa y reporta stats, pero típicamente luego un proceso orquestador o script programado recoge esos archivos para procesarlos.
3. **Selección de archivos recientes:** El integrador (ya sea mediante un comando manual o dentro del orquestador diario) busca los archivos JSON recientes de scrapers en las rutas configuradas [1]. Normalmente se define una ventana (por ejemplo 2 horas) para no re-procesar archivos viejos. Los lista y ordena, luego imprime cuántos encontró [2] [68].
4. **Iteración sobre archivos:** Por cada archivo:
5. Se abre y carga el JSON completo en memoria.
6. Se obtiene la lista de productos y la metadata [69].

7. Se determina el retailer asociado a ese archivo (sea por nombre de carpeta o campo `scraper` en metadata) [70].
8. Se lleva un acumulado del total de productos encontrados para métricas.
9. *En un entorno real*, aquí podría haber lógica para paralelizar la siguiente sub-etapa por producto, pero en la integración actual incluso se limitó a 30 productos por archivo para pruebas [71] [72]. En producción se removería ese límite para procesar todos.
10. **Procesamiento de cada producto:** Para cada producto del archivo:
11. Se normaliza el formato construyendo el diccionario estándar (`product_data`). Esto incluye extraer el mejor precio disponible numérico [47], asignar un ID adecuado (o generar uno temporal si falta) [73] [74], entre otras asignaciones.
12. Se instancia un objeto `ProductData` y se pasa al pipeline de IA para **categorización** [75] [76]. Aquí, internamente ocurre todo lo de caches, GPT, etc. explicado antes. El resultado es un objeto `CategorizationResult` con las categorías y atributos.
13. Si el resultado tiene una categoría válida (categoria_n2 distinta de "Sin categoría" u output no vacío) [77] [78], se considera **procesado con éxito**. Se incrementa el contador de productos normalizados. Si por alguna razón la IA no asignó categoría (muy raro, a menos que sea un producto totalmente fuera de lo esperado), se podría marcar como no categorizado.
14. (En la integración actual, imprimía en consola el nombre truncado y la categoría asignada para verificación manual [79]).
15. **Almacenamiento en BD:** A medida que cada producto obtiene su categorización, el pipeline realiza las operaciones en la base de datos:
16. Determina si es producto nuevo o existente vía matching.
17. Inserta/actualiza en `products` (master) y obtiene un `product_id` interno si no lo tenía.
18. Inserta/actualiza en `normalized_products` la entrada de categorización por retailer [80] [81].
19. Inserta/actualiza en `current_prices` el precio actual [34] [35].
20. Inserta en `product_mappings` si es un nuevo retailer para un producto existente [82].
21. Registra cambios significativos en `price_changes` si corresponde [42].
22. Todas estas operaciones están envueltas en transacciones atómicas por producto, aprovechando `ON CONFLICT` para manejar duplicados de forma idempotente (por ejemplo, si procesamos dos duplicados seguidos, el segundo hace conflict y update, pero la lógica de nuestro integrador posiblemente evite llamarlo dos veces de todas formas).
23. **Finalización del lote:** Tras procesar todos los productos de todos los archivos, se imprime/loguea un resumen con estadísticas: cuántos archivos, cuántos productos en total se leyeron, cuántos se normalizaron/categorizados con éxito, etc. [83] [84]. También se indica, en la integración, la base de datos de destino (PostgreSQL en tal host) donde quedaron guardados [85].
24. **Verificación y limpieza:** Opcionalmente, se pueden ejecutar scripts de verificación de integridad post-proceso (por ejemplo, `analyze_data_integrity.py` mencionado en la documentación [86] [87]) que comprueban que no haya datos inválidos en BD, o revisar que la cache de especificaciones técnicas esté poblada. Asimismo, el sistema puede preparar reportes diarios consolidando lo procesado, precios cambiados, etc., como de hecho hace el orquestador en su reporte de las 06:30 [88] [89].

Finalmente, los datos quedan persistidos en la base **listas para ser consultadas** por las demás partes del sistema (por ejemplo, un dashboard de comparación de precios entre retailers, o un módulo de alertas que envié emails si un producto bajó de precio). Las tablas clave actualizadas en el proceso son: - `products` (catálogo maestro de productos normalizados únicos), - `normalized_products` (histórico o log de categorización por cada aparición de producto por retailer) [90], - `current_prices` (estado actual de precio y stock por producto-retailer) [90], - `price_changes` (registro de cambios significativos detectados), - `price_history` (históricos diarios de precios, llenada en snapshots), - `ai_categorization_cache` y `ai_cache` (tablas de cache de respuestas IA), - `specs_cache`

(cache de especificaciones técnicas extraídas, para evitar reconsultar datos técnicos de un modelo conocido) [91], - `product_mappings` (mapeo códigos externos a internos para duplicados cross-retailer).

Cabe resaltar que durante todo este flujo, se mantienen **restricciones de unicidad e integridad referencial** en BD para garantizar la consistencia. Por ejemplo, `product_id` en `products` es clave primaria y se referencia en las demás tablas; combinaciones únicas evitan duplicar registros ya existentes. La documentación indica que tras la integración se preservaron todos los constraints anti-duplicados existentes, confirmando la consistencia del diseño [92].

En suma, el pipeline garantiza que los datos pasen de **HTML crudo** a **información normalizada en BD** en cuestión de minutos, con intervención mínima manual, aprovechando IA para entender el significado de cada producto y una sólida lógica para insertar o actualizar registros adecuadamente.

Áreas de Mejora Identificadas

A pesar de que el sistema está funcionando de manera integral y automática, el análisis detallado de los archivos JSON y la lógica actual revela varias oportunidades de mejora en la estructura y robustez:

- **1. Consistencia de campos entre scrapers:** Asegurar que todos los scrapers (incluyendo los de rotación) produzcan los campos esenciales. Vimos que en `rotation_*.json` faltan `product_code`, `brand`, y precios numéricos. **Mejora propuesta:** modificar el scraper de rotación para extraer el SKU (quizá parseando la URL de cada producto o un atributo oculto) y la marca (podría usar una lista de marcas conocidas para identificarla en el nombre). Asimismo, extraer los valores numéricos de los precios mostrados. De este modo, los archivos de rotación tendrían el mismo formato rico que los de búsqueda continua, evitando tratamiento especial. Esto también permitiría utilizar plenamente la detección de duplicados por código y reduciría el riesgo de insertar precios como 0.

- **2. Pre-procesamiento de duplicados antes de IA:** Actualmente, si un JSON contiene el mismo producto repetido (lo cual ocurrió, e.g. Falabella notebooks listó 1200 productos pero solo ~59 únicos, indicando repeticiones múltiples), el pipeline invoca la IA para el primero y para el resto detecta duplicado vía cache/BD. Funciona, pero es redundante cargar esos duplicados hasta esa etapa. **Mejora propuesta:** filtrar productos duplicados dentro de cada archivo JSON antes de la normalización/categorización, utilizando como llave el `product_code` + retailer (y quizá considerar mismo ciclo, por seguridad). Se podría hacer en el `ScrapersDataProcessor.extract_products_from_file` o justo después de reunir `all_products`. Esto ahorraría tiempo de procesamiento y llamadas a cache/BD innecesarias. Ya hay mecanismos en BD para no duplicar, pero evitarlo antes mejorará rendimiento (especialmente relevante si en una búsqueda se repiten decenas de veces como vimos).

- **3. Uso de `product_mappings` para skip IA:** Cuando un producto de un retailer ya fue emparejado a un master en el pasado (lo que implica que tenemos en `product_mappings` su código externo), en teoría no haría falta volver a pasar por la IA para categorizarlo: sabemos que corresponde al mismo master SKU, por ende ya conocemos su categoría. El pipeline actual primero categoriza y luego hace matching; podríamos invertir para casos conocidos. **Mejora propuesta:** al iniciar `process_new_product`, consultar `product_mappings` por (`external_retailer_id`, `retailer`). Si existe un mapping apuntando a un `product_id` ya categorizado, saltar directo a _add_price_to_existing_product y _insert_normalized_product usando la categoría ya conocida de ese product_id, omitiendo

`ai_pipeline.categorize_product`. Esto aceleraría el ingreso de productos ya vistos en otro retailer. Habría que manejar con cuidado por si la categoría evolucionó, pero generalmente debería ser estable.

- **4. Refinamiento del *Data Integrity Guard* y flujos de fallback:** Aunque la integridad de categoría ya se controla, podemos ampliar algunas reglas:

- Exigir presencia de marca detectada para categorías donde siempre debe haber (ej. Electrónica: sin marca confiable, el resultado quizás no es útil, podría marcarse `necesita_revision`).
- Implementar una política para productos que salen con categoría "General" o "Sin categoría" incluso tras fallback: quizá meterlos a una cola manual. Por ahora se imprimía como "Sin categoría..." en la integración de prueba [93], pero en producción conviene registrarlos en BD marcados para análisis humano.

- Evaluar la utilidad de un modelo de validación (`o1-mini` mencionado en config) para doble-check. Vi que hay una preparación para un modelo de validación O1-mini [94], posiblemente para verificar JSONs generados. Si no está activo, podría ser algo a explorar para garantizar que el JSON de salida cumpla esquemas (aunque GPT5-nano ya se le da formato fijo).

- **5. Ampliación de datos enriquecidos:** Actualmente, la descripción del producto no se utiliza en categorización (pues en búsquedas no tenemos descripciones). Podría considerarse en futuras iteraciones scrapear la descripción breve o ficha técnica del producto en la página individual y pasarla al pipeline de IA. Esto le daría al modelo más contexto para clasificar correctamente, especialmente útiles para productos ambiguos. Claro está, implicaría más scraping (más carga) por producto, quizás no factible para cientos a la hora; pero se podría hacer para productos nuevos solamente o mediante un pipeline secundario bajo demanda.

- **6. Manejo optimizado de ciclos y concurrencia:** Para acelerar el flujo, se pueden ejecutar scrapers en paralelo (como la doc indica) y potencialmente procesar archivos en paralelo también. La implementación actual parece procesar archivos secuencialmente y dentro limita productos a 30 por prueba. **Mejora propuesta:** remover la limitación de 30, y permitir procesamiento batch concurrente (quizá usando asyncio gather en `process_products_batch`). También, clarificar el esquema de numeración de ciclos: si se mantiene un esquema global, está bien, pero si no, al menos documentar internamente que 001 no siempre es el mismo retailer. Podría incluso añadirse en metadata un campo `retailer` explícito (además de en cada producto), para no depender de naming.

- **7. Logging y monitorización:** Actualmente hay logging con emojis muy útil en consola. Se podría robustecer con:

- Un **dashboard de monitoreo** en tiempo real que lea las stats (archivos procesados, productos nuevos, errores) que el orquestador y el monitor ya calculan [95] [96]. Esto ayudaría al equipo a identificar cuántos productos se están normalizando por día, cuántos se rechazan, etc.
- En caso de error en una llamada de IA (excepción de API, timeout), el pipeline hace fallback o guarda raw. Sin embargo, sería bueno capturar métricas de error para saber si el modelo falla con ciertos productos frecuentemente y por qué, permitiendo mejorar prompts.

- **Costos de IA:** incluir monitoreo del costo acumulado diario (el orquestador tiene un campo stats.total_cost_usd [97] [98]). Asegurarse de respetar el límite diario configurado (en config aparece max_daily_cost $10 [99]). Podríamos implementar que al acercarse al límite, se reduzca la frecuencia o se apague la categorización automática temporalmente (feature futura quizás).

- **8. Calidad de datos de entrada:** Aunque escapa un poco a "post-scraping", es relevante mencionar: mejorar los scrapers para evitar datos incoherentes mejora todo el pipeline. Por ejemplo, Falabella devolviendo 1200 resultados con duplicados sugiere que tal vez la página listó combinaciones de filtros. Podría ajustarse el scraper para reducir eso (ej. scrapear con filtros refinados o eliminar patrocinados duplicados). Cada mejora en scraping se traduce en menos carga para la normalización.

- **9. Escalabilidad del sistema de categorías:** Actualmente se apoya en GPT con una whitelist de 30 categorías. Si el catálogo de categorías creciera o cambiara (nuevas categorías de productos), habría que actualizar la whitelist y posiblemente las plantillas. Sería útil tener esto parametrizado (quizás cargando las categorías de la BD en vez de codificadas). De hecho, el CategoryWhitelist ya obtiene de DB o local la taxonomía [100], lo cual es bueno. Asegurarse de sincronizarlo con cualquier cambio en la taxonomía de negocio.

- **10. Documentación y mantenimiento:** Mantener documentos actualizados (como este) en el repositorio para que nuevos desarrolladores entiendan el flujo. La presencia de archivos en `docs/` y `archive_2025-09-07` es buena señal. Sería conveniente consolidar esa documentación en un solo lugar vivo (evitar confusión entre doc archivada y actual).

Para concluir, el proyecto de **normalización ML** post-scraping ya integra múltiples componentes de forma exitosa: scraping multi-sitio, procesamiento unificado, inteligencia artificial para categorización, y almacenamiento con control de duplicados y precios históricos [101] [102]. Las mejoras sugeridas apuntan a hacerlo **más robusto y eficiente**, reduciendo la necesidad de limpiezas manuales y mejorando la calidad de los datos que alimentan al sistema. Implementar estas mejoras incrementará la confiabilidad del sistema en producción y facilitará su escalabilidad a más categorías o retailers en el futuro. Cada miembro del equipo podrá enfocarse en expandir funcionalidades sabiendo que la base de datos resultante es consistente, íntegra y rica en información gracias a este pipeline bien estructurado. [92] [103]

---

[1] [2] [10] [11] [13] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [47] [48] [49] [53] [54] [55] [56] [57] [58] [59] [60] [64] [68] [69] [70] [71] [72] [73] [74] [75] [76] [77] [78] [79] [83] [84] [85] [93] [94] [100] ai_normalization_pipeline_v2.py
https://github.com/Pachonchox/sistema-normalizacion-ml/blob/def7de5187655477c0149896968701124673 1bf9/ai_normalization_pipeline_v2.py

[3] [4] [5] [6] [7] [8] [9] [46] [50] [51] scrapers_data_processor.py
https://github.com/Pachonchox/sistema-normalizacion-ml/blob/def7de5187655477c0149896968701124673 1bf9/scrapers_data_processor.py

[12] [14] [61] [63] [86] [87] [90] [91] [92] [101] [102] [103] INTEGRATION_COMPLETE.md
https://github.com/Pachonchox/sistema-normalizacion-ml/blob/def7de5187655477c0149896968701124673 1bf9/archive_2025-09-07/INTEGRATION_COMPLETE.md

[26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [52] [80] [81] [82] master_product_integrator.py
https://github.com/Pachonchox/sistema-normalizacion-ml/blob/def7de5187655477c0149896968701124673 1bf9/archive_2025-09-07/master_product_integrator.py

[62] [88] [89] [96] [97] [98] [99] daily_orchestrator.py
https://github.com/Pachonchox/sistema-normalizacion-ml/blob/def7de5187655477c0149896968701124673 1bf9/daily_orchestrator.py

65 66 67 95 scrapers_file_monitor.py

https://github.com/Pachonchox/sistema-normalizacion-ml/blob/def7de5187655477c0149896968701124 6731bf9/scrapers_file_monitor.py