Universidad Politécnica de Valencia

Master de formación permanente de Big Data e Inteligencia Aritficial

Análisis de Precios de Gasolineras en Valencia

Memoria del Proyecto

Autor:
Juanfran Tomas Planells
Ignacio Galiano López

Tutor:
Andrés Terrasa Barrena

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Índice general

Capítulo 1

INTRODUCCIÓN

Repostar al mejor precio exige información actualizada y fácil de leer. Este proyecto resuelve ese punto con automatización en Bash que, cada hora, descarga precios públicos de Valencia, verifica los datos y genera un resumen.

Características principales:

- Ejecución cada hora
- Tratamiento de datos para comparar y sintetizar con diferentes criterios
- Generar reporte en formato texto y HTML

El trabajo se ajusta a la arquitectura del enunciado: descarga, procesamiento, generación de informes, programación y registro. El alcance se limita a combustibles habituales en Valencia, con cadencia horaria. Como mejora razonable, se contempla ampliar ámbito geográfico o parametrizar ajustes sin alterar filosofía de simplicidad y bajo mantenimiento.

Objetivo: Ofrecer, cada día, un panorama fiable y útil para decidir dónde repostar.

Capítulo 2

FUENTE DE DATOS

El sistema se apoya en datos públicos de precios de estaciones de servicio para Valencia, servidos en JSON mediante endpoint estable y accesible sin autenticación. Esto permite: (1) ejecución desatendida sin dependencia de formularios ni tokens, integración directa en cron; (2) formato JSON facilita verificación estructural y extracción precisa de campos antes de cualquier tratamiento.

Elemento	**Detalle**
Formato	JSON
Método	HTTP GET con curl
Autenticación	No requiere
Frecuencia	Actualización diaria
Campos clave	Precio G95/Diésel, coordenadas, dirección

Figura 2.1: Características de la fuente de datos

La obtención se realiza con petición HTTP idempotente mediante `curl` en modo silencioso, adecuada para cron. Se fija cabecera `Accept: application/json`, se vuelca respuesta a fichero y se captura código HTTP en variable `status` con `-w '%{http_code}'`. Con esa señal decidimos si proseguir o abortar: solo continuamos ante 2xx. Filosofía: "fallar pronto y con diagnóstico comprensible".

```
# Descarga con curl y captura de codigo HTTP
status=$(curl -s -w '%{http_code}' -o "$ARCHIVO_JSON" \
  -H "Accept: application/json" "$URL_API")

if [ "$status" -ge 200 ] && [ "$status" -lt 300 ]; then
  echo "[...] (Descarga) OK HTTP=$status"
else
  echo "[...] (Descarga) ERROR HTTP=$status"
  exit 1
fi
```

Figura 2.2: Descarga y validación HTTP

El fichero se guarda con identificador temporal (`RUN_ID`) en el nombre, asegurando trazabilidad de extremo a extremo. Una ejecución concreta deja, bajo ese `RUN_ID`, el JSON original, informes en TXT y HTML y entradas en el log. Si surge duda o anomalía, es posible reconstruir qué datos se usaron y qué cálculo se hizo.

La fuente incluye marca temporal propia (`Fecha`) del momento de actualización oficial. El sistema maneja dos referencias complementarias: fecha/hora de ejecución (cuando corre

cron) y fecha API (cuándo se actualizaron datos en proveedor). En informes se muestran ambas para claridad.

Aspecto práctico: campos numéricos (precios) vienen con coma decimal. No afecta descarga, pero sí uso. Si no se normaliza a punto decimal y no se tipa a número, cálculos como medias, mínimos u ordenaciones fallan. Este punto se resuelve en Punto 4 con normalización y tipado mediante `jq`. La planificación (cadencia y redirecciones) se detalla en Punto 3.

Con fuente y mecánica definidas, falta encajar su lugar en el proyecto y repetición diaria. Para que cada descarga termine en directorio correcto, conserve `RUN_ID` y deje rastro auditable, la ejecución no puede depender de acciones manuales ni variables de entorno cambiantes. El siguiente apartado define arquitectura mínima (árbol de carpetas y nombres) y automatización que la hace operativa.

Capítulo 3

ARQUITECTURA

Este apartado coloca la descarga dentro de arquitectura mínima y repetible, detallando automatización diaria. La idea: al ejecutarse desde cron, el proyecto hace siempre lo mismo, en el mismo sitio, dejando rastro inequívoco, sin depender del usuario.

Estructura de directorios: La arquitectura parte de directorio base anclado al script. En ejecución programada el directorio de trabajo no es predecible; por eso se obtiene `BASE_DIR` a partir de ruta real del script y todas las lecturas/escrituras cuelgan de ahí. Esto evita efectos de `pwd` variables, la estructura del proyecto es la siguiente:

```
shell-proyect/
|-- analisis_json.sh      (script principal)
|-- log.txt               (bitacora funcional)
|-- cron.log              (salidas de cron)
|-- datos/                (JSON descargados)
|   +-- estaciones_*.json
|-- informes/             (TXT y HTML generados)
|   |-- informe_*.txt
|   +-- informe_*.html
+-- planificacion/        (configuracion cron)
    +-- crontab.txt
```

Figura 3.1: Estructura de directorios del proyecto

A partir de `BASE_DIR` se organizan tres subcarpetas: `datos/` para JSON crudo, `informes/` para salidas en TXT y HTML, y `planificacion/` para material de cron. El árbol facilita revisión y entrega con orden, y permite reconstruir la ejecución concreta si algo falla.

Para que la estructura exista incluso en primera ejecución, el script se abre con la función de preparación de carpetas. Además de crear lo que falte con `mkdir -p`, deja constancia en registro, permitiendo auditar el estado del entorno.

Trazabilidad y registro: Los nombres se basan utilizando el sufijo `RUN_ID`. Ese valor se inserta en los nombres del JSON descargado y de los informes generados, encapsulando cada ejecución bajo su propio sello temporal. Se mantienen dos registros complementarios:

- `log.txt`: Bitácora funcional del proceso (descarga, procesamiento, informes)
- `cron.log`: Traza del planificador (stdout/stderr de cron)

Separarlos acelera el diagnóstico: si el cron no encuentra script o falla, deja evidencia en `cron.log`; si falla la validación o el cálculo lo documenta en `log.txt`.

Automatización con cron: La implementación se despliega en servidor de Google Cloud para garantizar disponibilidad continua. El proceso de configuración sigue estos pasos sistemáticos:

1. **Preparación del entorno:** Clonado del repositorio desde GitHub (https://github.com/juanfrantomas/shell-proyect) en directorio home del servidor
2. **Verificación de permisos:** Confirmación de que el script es ejecutable mediante `chmod +x`
3. **Configuración de crontab:** Edición con `crontab -e` para añadir tarea horaria
4. **Validación del servicio:** Comprobación de estado activo de `cron.service` y monitoreo de logs del sistema

La línea de cron implementa ejecución cada hora con redirección completa de salidas. La redirección `cron.log 2>&1` captura errores críticos de permisos, sintaxis o inicialización que podrían impedir la creación del log funcional interno.

```
# Configuracion en crontab -e
0 * * * * /usr/bin/env bash /home/juanfrandeveloper/master/shell-proyect
   /analisis_json.sh >> /home/juanfrandeveloper/master/shell-proyect/
   cron.log 2>&1
```

Figura 3.2: Configuración de cron para ejecución cada hora

El servicio `cron` mantiene estado activo permanente, garantizando la automatización sin intervención manual.

Finalización controlada: El cierre lo da mecanismo de finalización mediante `trap`. Al capturar código de retorno global, se imprime línea final inequívoca en `log.txt`: éxito si código es 0; error con código concreto en caso contrario. Esto evita falsos positivos y simplifica revisión.

En términos prácticos, con estos elementos la descarga encaja en marco que no depende del contexto: script sabe dónde escribir, cron sabe cómo lanzarlo y ambos dejan huella. Como mejora opcional, si se teme que dos ejecuciones coincidan, puede añadirse antisolapamiento con `flock`, si el job anterior sigue corriendo, flock detecta el lock y no inicia otro.

Capítulo 4

VALIDACIÓN DE DATOS

Este apartado convierte la descarga del Punto 2 en un conjunto de datos fiable y utilizable. La regla es clara: no se calcula nada hasta demostrar que la respuesta tiene estructura coherente, campos críticos existentes y valores numéricos correctamente tipados. Todo queda registrado en el log para una auditoría posterior.

El flujo comienza con comprobaciones de mínimos: variables no vacías,si JSON esta bien formado con `jq empty`, con presencia del array `ListaEESSPrecio` y el campo `Fecha`. Este orden permite "fallar pronto" con mensaje inequívoco. Se extrae `FECHA_API` para mostrarla en informes junto a `FECHA_CRON`.

Validación inicial: El código verifica estructura y campos críticos (ver script completo en el repositorio):

- Comprobación de variable no vacía
- Validación de JSON con `jq empty`
- Verificación de `ListaEESSPrecio` como array
- Extracción de `FECHA_API`

Normalización y tipado: El dataset publica decimales con coma. Se aplica normalización explícita con `gsub(","; ".")` y cast seguro `tonumber?`, asignando `null` si no es convertible. Se seleccionan y renombran los campos (`precio_g95`, `precio_diesel`, `lat`, `lon`) conservando atributos útiles (`rotulo`, `localidad`, `provincia`, `direccion`). El resultado: un array `estaciones` con precios numéricos reales y coordenadas válidas o nulas bien declaradas.

```
estaciones=$(echo "$getEstacionesValencia" | jq '[
  .ListaEESSPrecio[]
  | {
      id: (.IDEESS | tonumber?),
      name: .["Rotulo"],
      lat: ( (.Latitud // "") | gsub(",";".")
             | (if . == "" then null else tonumber end) ),
      lon: ( (.["Longitud (WGS84)"] // "") | gsub(",";".")
             | (if . == "" then null else tonumber end) ),
      addr: ( [ .["Direccion"], .["Localidad"], .["Provincia"] ]
              | map(select(. != null and . != "")) | join(", ") ),
      priceDiesel: ( (.["Precio Gasoleo A"] // "") | gsub(",";".")
                     | (if . == "" then null else tonumber end) ),
      priceGasolina: ( ( .["Precio Gasolina 95 E5"] //
                         .["Precio Gasolina 95 E10"] // "" )
                       | gsub(",";".")
                       | (if . == "" then null else tonumber end) )
    }
]')
```

Figura 4.1: Normalización y tipado con jq (extracto)

Métricas de calidad: Se mide la calidad registrando cuatro magnitudes: total de estaciones, estaciones sin precio G95, sin precio Diésel y sin coordenadas. Estos contadores permiten relacionar anomalías (una media extraña) en el dataset, sin aparecer en el informe final pero sí en el log.

Filtrado por combustible: Se preparan subconjuntos: solo estaciones con precio numérico válido. Este filtrado temprano garantiza que los módulos trabajen sobre información consistente y simplifica la lectura del script.

Validación final: Si ningún combustible dispone de precios válidos tras filtrar, el proceso se detiene con error claro. Es preferible declarar "no hay datos útiles" a producir informes vacíos. El log explica el motivo y permite diagnóstico rápido.

Con estas validaciones, normalizaciones y controles, el proyecto entrega un dataset tipado, segmentado y coherente al Punto 5 para medición y presentación.

Capítulo 5

GENERACIÓN DE INFORMES

Con el dataset validado y normalizado, este apartado mide y presenta. El objetivo: calcular indicadores básicos por combustible (conteo, mínimo, media, máximo) y ofrecer resultado en TXT para la lectura rápida y HTML para consultar en pantalla, manteniendo trazabilidad mediante el `RUN_ID`.

Cálculo de métricas: Se parte de los subconjuntos preparados. Para cada combustible se obtiene conteo de estaciones con precio válido y, si es mayor que cero, se calculan mínimo, máximo y media aritmética. Esta precaución evita divisiones por cero. El trabajo opera sobre valores numéricos reales, haciendo ordenación y agregación directa y fiable.

Además de estadísticas globales, se incluye Top-5 con mejores precios. Se ordena por precio ascendente y se extraen las cinco mejores.

```
# Metricas y Top-5
G95_COUNT=$(echo "$estaciones" | jq '
  [ .[] | .priceGasolina ] | map(select(.!=null)) | length')

if (( ${G95_COUNT:-0} > 0 )); then
  G95_MIN=$(echo "$estaciones" | jq '
    [ .[] | .priceGasolina ] | map(select(.!=null))
    | min | (. * 1000 | round / 1000)')
  G95_MAX=$(echo "$estaciones" | jq '
    [ .[] | .priceGasolina ] | map(select(.!=null))
    | max | (. * 1000 | round / 1000)')
  G95_AVG=$(echo "$estaciones" | jq '
    [ .[] | .priceGasolina ] | map(select(.!=null))
    | if length>0 then ((add/length) * 1000 | round / 1000)
      else null end')

  TOP5_G95=$(echo "$estaciones" | jq '
    [ .[] | select(.priceGasolina!=null)
      | { id, name, addr, priceGasolina }]
    | sort_by(.priceGasolina) | .[0:5]')
fi
```

Figura 5.1: Cálculo de métricas y Top-5 por combustible

Formateo unificado: Internamente se usa punto decimal; para el usuario se le muestra con coma y tres decimales para mantener homogeneidad. El mismo criterio se aplica en cabeceras y filas del Top-5, evitando discrepancias entre formatos:

```
(. * 1000 | round / 1000) | tostring | gsub("
textbackslash
textbackslash."; ",")
```

Informe TXT: Prioriza lectura inmediata. Inicia con cabeceras que muestran `FECHA_CRON` y `FECHA_API`, seguidamente de dos bloques (Gasolina 95 y Diésel A) con conteo y estadísticas (mín/med/max). Debajo, muestra una lista numerada con las Top-5 indicando el nombre, precio y dirección.

Informe HTML: Ofrece misma información organizada en tablas con estilo. Incluye encabezado con fechas y, por combustible, título con insignias de min/med/max. La tabla muestra las columnas: ID, Nombre, Dirección, Latitud, Longitud, Precio Gasolina (€).

Ambos informes se nombran con `RUN_ID` y guardan en `informes/`, preservando trazabilidad. El script deja constancia con mensajes INFO de la generación del TXT y HTML, permitiendo comprobar si la ejecución terminó bien y qué documenntos produjo.

Capítulo 6

REGISTRO Y AUDITORÍA

Este apartado define cómo se audita cada ejecución y qué señales permiten juzgar su calidad. Leyendo el log puede saberse qué pasó, cuándo, en qué fase, con qué resultado y calidad de datos. Se usan mensajes con marca temporal (`[%F %T]`), taxonomía de fases alineada con el flujo, niveles simples (INFO, OK, ERROR) y línea de finalización inequívoca que refleja el código de salida. Se separan dos canales: `log.txt` (bitácora funcional del script) y `cron.log` (stdout/stderr de cron). Con esto, diagnosticar incidencias lleva segundos.

Formato y fases: Cada línea empieza por `[YYYY-MM-DD HH:MM:SS]`, sigue con fase entre paréntesis (Inicio, Descarga, Procesamiento, Indicadores, Informe, Finalizacion), nivel y mensaje claro. Las fases siguen el flujo, facilitando localizar problemas. Niveles: INFO (hitos), OK (chequeos superados), ERROR (fallos críticos con `exit 1`).

```
# Ejemplos de mensajes en distintas fases
echo "[$(date '+%F %T')] (Inicio) INFO Identificador=$RUN_ID"
     >> $NOMBRE_ARCHIVO_LOG

echo "[$(date '+%F %T')] (Descarga) OK Peticion Valida:
     HTTP=$status" >> $NOMBRE_ARCHIVO_LOG

echo "[$(date '+%F %T')] (Procesamiento) OK JSON valido"
     >> "$NOMBRE_ARCHIVO_LOG"

echo "[$(date '+%F %T')] (Indicadores) INFO Calculando Top 5"
     >> "$NOMBRE_ARCHIVO_LOG"

echo "[$(date '+%F %T')] (Informe) OK Informe generado: $TXT"
     >> "$NOMBRE_ARCHIVO_LOG"
```

Figura 6.1: Formato de mensajes y fases del log

Gestión de errores: Cuando aparece ERROR, el criterio es una salida controlada: registrar el motivo e interrumpir con `exit 1`. Esto evita falsos positivos (informes con datos corruptos) y deja pista inequívoca. Ejemplos: HTTP no 2xx o JSON inválido.

Finalización con trap: Se usa `trap` para capturar código de salida y escribir línea de cierre. Si código es 0, indica éxito; si no, informa del código. Leer el último bloque de `log.txt` basta para saber si la ejecución terminó bien o mal.

```
trap 'code=$?; if [ $code -eq 0 ]; then
  echo "[$(date "+%F %T")] (Finalizacion) INFO
       Programa ejecutado correctamente" >> "$NOMBRE_ARCHIVO_LOG"
else
  echo "[$(date "+%F %T")] (Finalizacion) ERROR
       Programa termino con codigo $code" >> "$NOMBRE_ARCHIVO_LOG"
fi' EXIT
```

Figura 6.2: Finalización inequívoca con trap

Separación de logs: `log.txt` recoge lo que hace el script (fases, validaciones, indicadores); `cron.log` captura cómo se lanzó (errores de ruta, permisos, PATH incompleto). Esto diferencia problemas antes de entrar al script vs. durante ejecución.

Señales de calidad: El log anota el total de las estaciones, cuántas carecen de precio G95/Diésel y cuántas estan sin coordenadas. Si un día la media es extraña, el log permite atribuirlo a carencia de la fuente, no error en la lógica.

Trazabilidad: Todo se ancla en `RUN_ID`. Aunque no aparece en cada línea, sí en mensajes clave (rutas de informes generados), permitiendo saltar del log al TXT/HTML y JSON crudo correspondiente para hacer la auditoría reproducible.

Con este esquema, el proyecto no solo descarga, valida, calcula e informa, también explica lo que hace de manera legible y verificable: resolución rápida de incidencias y confianza en que cada informe está respaldado por rastro claro desde el dato crudo hasta la línea final de éxito o error.