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 Martín Terrasa Barrena

October 28, 2025

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Dedicado a...

Resumen

Este proyecto desarrolla un sistema automatizado para el análisis de precios de combustibles en las gasolineras de la provincia de Valencia. Mediante técnicas de scraping de datos públicos y procesamiento con herramientas de línea de comandos, se generan informes periódicos que identifican las estaciones de servicio con los precios más competitivos.

Palabras clave: Análisis de datos, Shell scripting, JSON, API REST, Automatización

Agradecimientos

Aquí puedes agradecer a las personas e instituciones que han contribuido al desarrollo del proyecto.

Contents

Chapter 1

OBJETIVO Y ALCANCE

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

La ejecución es diaria y no requiere intervención. El tratamiento es mínimo para poder comparar y sintetizar con criterio. La salida se publica en formato texto y en HTML. Cada ejecución queda registrada para mantener la trazabilidad.

El trabajo se ajusta a la arquitectura indicada en el enunciado: descarga, procesamiento, generación de informes, programación y registro. No se desarrolla una aplicación completa con mapas, bases de datos o interfaz gráfica. El alcance se limita a los combustibles habituales en la provincia de Valencia, con cadencia diaria. Como mejora razonable, se contempla ampliar el ámbito geográfico o parametrizar ciertos ajustes sin alterar la filosofía de simplicidad y bajo mantenimiento. El objetivo es ofrecer, cada día, un panorama fiable y útil para decidir dónde repostar.

Chapter 2

FUENTE DE DATOS Y DESCARGA

El sistema se apoya en datos públicos de precios de estaciones de servicio para la provincia de Valencia, servidos en JSON mediante un endpoint estable y accesible sin autenticación. Esto es clave por dos razones. Primero, permite ejecución desatendida: al no depender de formularios ni tokens, la petición se integra sin fricción en una tarea programada. Segundo, el formato JSON facilita la verificación estructural y la extracción precisa de campos antes de cualquier tratamiento, de modo que lo que llega al informe está respaldado por un documento coherente.

La obtención se realiza con una petición HTTP idempotente mediante `curl` en modo silencioso, adecuada para cron. Se fija la cabecera `Accept: application/json` para evitar ambigüedades en la negociación de contenido, se vuelca la respuesta a fichero y se captura el código de estado HTTP en la variable `status` con `-w '%{http_code}'`. Con esa señal decidimos si proseguir o abortar antes de tocar el JSON: solo continuamos ante un 2xx. En caso contrario, el script corta la ejecución y deja un mensaje claro en el log. Filosofía: "fallar pronto y con diagnóstico comprensible", porque no hay un operador mirando la consola.

El fichero se guarda con un identificador temporal (`RUN_ID`) en el nombre. No es un detalle estético: asegura la trazabilidad de extremo a extremo. Una ejecución concreta (por ejemplo, 24/10/2025 a las 08:00) deja, bajo ese `RUN_ID`, el JSON original, los informes en TXT y HTML y las entradas correspondientes en el log. Si más tarde surge una duda o una anomalía, es posible reconstruir qué datos se usaron y qué cálculo se hizo sin mezclar artefactos de distintos días.

La fuente incluye, además, una marca temporal propia (`Fecha`) que recoge el momento de actualización oficial del dataset. Por eso el sistema maneja dos referencias temporales complementarias: la fecha y hora de ejecución (cuando corre el cron, ver Punto 3) y la fecha de la API (cuándo se actualizaron los datos en el proveedor). En los informes se muestran ambas para que el lector entienda si está viendo resultados de hoy y de qué actualización proceden.

Un aspecto práctico del dataset: muchos campos numéricos (por ejemplo, precios) vienen con coma decimal. No afecta a la descarga, pero sí al uso. Si no se normaliza a punto decimal y no se tipa a número, cálculos como medias, mínimos u ordenaciones fallan o arrojan resultados inconsistentes. Este punto se resuelve en el Punto 4 (Proceso

y validaciones) con normalización y tipado mediante `jq`. La planificación que da la cadencia adecuada y gestiona redirecciones de salida/errores se detalla en el Punto 3 (Arquitectura y automatización).

Con la fuente y la mecánica de obtención definidas, falta encajar su lugar en el proyecto y su repetición diaria. Para que cada descarga termine en el directorio correcto, conserve su `RUN_ID` y deje un rastro auditable, la ejecución no puede depender de acciones manuales ni de variables de entorno cambiantes. El siguiente apartado define la arquitectura mínima (árbol de carpetas y nombres de ficheros) y la automatización que la hace operativa (planificación, rutas absolutas y redirección de salidas). Sobre esa base estable se apoyarán las validaciones y los informes.

Chapter 3

ARQUITECTURA Y AUTOMATIZACIÓN

Con la fuente y la mecánica de obtención ya definidas, este apartado coloca esa descarga dentro de una arquitectura mínima y repetible y detalla la automatización diaria que la sostiene. La idea es simple: al ejecutarse desde cron, el proyecto hace siempre lo mismo, en el mismo sitio, y deja un rastro inequívoco de lo ocurrido, sin depender de que el usuario esté delante.

La arquitectura parte de un directorio base anclado al propio script. En ejecución programada el directorio de trabajo no es predecible; por eso se obtiene `BASE_DIR` a partir de la ruta real del script y todas las lecturas y escrituras cuelgan de ahí. Ese anclaje evita efectos de `pwd` variables y asegura que los artefactos terminen siempre bajo el proyecto, no en carpetas temporales ni en `$HOME`. A partir de `BASE_DIR` se organizan tres subcarpetas: `datos/` para el JSON crudo, `informes/` para las salidas en TXT y HTML, y `planificacion/` para el material de cron y la documentación. El árbol no es ornamental: permite revisar y entregar con orden y, sobre todo, facilita reconstruir una ejecución concreta si algo falla.

Para que esa estructura exista incluso en la primera ejecución, el script abre con una función de preparación de carpetas. No es una simple llamada a `mkdir -p`: además de crear lo que falte, deja constancia en el registro asociado a cron, de forma que pueda auditarse el estado del entorno, creado o ya existente. Este detalle resulta valioso al desplegar en máquinas nuevas o en directorios limpios, porque el log de arranque ofrece una pista inmediata sobre permisos y rutas.

La trazabilidad se apoya en una convención de nombres alrededor de un identificador temporal `RUN_ID`. Ese valor se inserta en los nombres del JSON descargado y de los informes que se generarán, de modo que cada ejecución queda encapsulada bajo su propio sello de tiempo. En paralelo se mantienen dos registros complementarios: `log.txt`, como bitácora funcional del proceso de descarga, procesamiento, informes y final; y `cron.log`, como traza del planificador con todo lo que cron envía por stdout y stderr. Separarlos acelera el diagnóstico: si cron no encuentra el script o falla el PATH, la evidencia aparece en `cron.log`; si lo que falla es una validación o un cálculo, queda documentado en `log.txt`.

La automatización se resuelve con una entrada de cron diaria. Más allá de la frecuencia concreta, hay tres decisiones que garantizan estabilidad: (i) rutas absolutas

tanto al intérprete como al script y a los ficheros de log; (ii) definición explícita de `SHELL` y `PATH` al inicio del bloque para no depender del entorno interactivo; (iii) redirección de salida estándar y de error al `cron.log` del proyecto. Con esos elementos la ejecución programada es reproducible y auditable. La línea de cron se guarda además en `planificacion/crontab.txt`, lo que permite incluirla tal cual en la memoria y verificar que la entrega corresponde con lo descrito.

El cierre del ciclo lo da un mecanismo de finalización controlada mediante `trap`. Al capturar el código de retorno global del script, se imprime una línea final inequívoca en `log.txt`: éxito si el código es 0; error con el código concreto en caso contrario. Esta técnica evita falsos positivos y simplifica la revisión, porque basta con ir al final del log para saber cómo terminó.

En términos prácticos, con estos elementos la descarga presentada en el Punto 2 encaja en un marco que no depende del contexto: el script sabe dónde escribir, cron sabe cómo lanzarlo y ambos dejan huella. El detalle de validaciones sobre estructura del JSON, tipos y normalización de decimales se desarrolla en el Punto 4, y el formato de los informes en el Punto 5. Como mejora opcional, si se teme que dos ejecuciones coincidan, puede añadirse antisolapamiento con `flock`, envolviendo la llamada de cron para asegurar exclusión mutua.

Chapter 4

PROCESO Y VALIDACIONES

Este apartado convierte la descarga del Punto 2 en un conjunto de datos fiable y utilizable, listo para calcular indicadores y generar los informes del Punto 5. La regla es clara: no se calcula nada hasta demostrar que la respuesta tiene una estructura coherente, que los campos críticos existen y que los valores que después se tratarán como números han sido convertidos a tipo numérico. Todo queda registrado en el log, de modo que en el Punto 6 pueda evaluarse la calidad de cada ejecución y, si hace falta, auditar con precisión en qué fase apareció un problema.

El flujo comienza con una comprobación de mínimos. Primero se verifica que la variable que contiene la respuesta no esté vacía, ya que un fichero inexistente o de tamaño cero suele indicar un problema de red o permisos. A continuación se valida que el documento sea JSON bien formado mediante `jq empty`, lo que descarta descargas truncadas y errores de sintaxis. Superada esa barrera, se comprueba la presencia y el tipo de las claves críticas: `ListaEESSPrecio` debe existir y ser un array, y `Fecha` debe existir y ser una cadena. Este orden evita perder tiempo en transformaciones si la base falla y permite "fallar pronto" con un mensaje inequívoco en el log. En el mismo paso se extrae `FECHA_API`, que se mostrará en los informes junto a `FECHA_CRON` (definida en el Punto 3) para explicar con claridad cuándo se ejecutó el proceso y de qué actualización proceden los datos.

```
echo "[...] (Procesamiento) INFO Comprobando que la variable
      no este vacia" >> $NOMBRE_ARCHIVO_LOG
if [ -z "$getEstacionesValencia" ]; then
  echo "[...] (Procesamiento) ERROR la variable esta vacia"
  exit 1
fi

echo "[...] (Procesamiento) INFO Comprobando que el JSON
      es valido" >> "$NOMBRE_ARCHIVO_LOG"
echo "$getEstacionesValencia" | jq empty > /dev/null \
  && echo "[...] (Procesamiento) OK JSON valido" \
  || { echo "[...] (Procesamiento) ERROR JSON invalido";
       exit 1; }

echo "[...] (Procesamiento) INFO Verificando ListaEESSPrecio
      como array"
echo "$getEstacionesValencia" | jq -e '.ListaEESSPrecio
      | type=="array"' \
  && echo "[...] (Procesamiento) OK ListaEESSPrecio presente
      y es array" \
  || { echo "[...] (Procesamiento) ERROR Falta
      ListaEESSPrecio[]"; exit 1; }
```

Listing 4.1: Figura 4.1 — Validación inicial del JSON

A partir de ahí, el objetivo es construir un conjunto homogéneo de estaciones con tipos correctos y nombres estables. El dataset de origen publica los decimales con coma, por lo que, si se deja tal cual, precios y coordenadas se tratan como texto y cálculos posteriores como medias, mínimos u ordenaciones serán incorrectos o imposibles. Se aplica una normalización explícita: `gsub(","; ".")` para sustituir coma por punto y `tonumber?` para el cast seguro a numérico, asignando `null` cuando el valor no sea convertible. En el mismo mapeo se seleccionan y renombran los campos de trabajo (por ejemplo, `precio_g95`, `precio_diesel`, `lat`, `lon`) y se conservan atributos útiles para la lectura (`rotulo`, `municipio`, `direccion`, `horario`). El resultado es un array `estaciones` compacto y coherente: a partir de aquí, "precio" es número y no texto, y "coordenadas" son pares lat/lon válidos o nulos bien declarados.

```
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) )
    }
]')
```

Listing 4.2: Figura 4.2 — Normalización y tipado con jq

Antes de pasar a cálculos, se mide la calidad del dato recibido. Si un día el proveedor publica menos campos, cambia una etiqueta o parte del territorio llega sin precios, el informe puede empobrecerse y conviene que el log lo explique. Por eso se registran cuatro magnitudes sencillas: total de estaciones, estaciones sin precio para Gasolina 95, estaciones sin precio para Diésel y estaciones sin coordenadas. Con estos contadores, cualquier anomalía posterior, como un top corto o una media anómala, se relaciona de inmediato con carencias del dataset de ese día. Estos valores no aparecen en el informe final, pero sí en el registro técnico que respalda la ejecución.

```
TOTAL_EESS=$(echo "$estaciones" | jq '. | length')

TOTAL_EESS_SIN_PRECIO_GASOLINA=$(echo "$estaciones"
  | jq '[.[] | select(.priceGasolina==null)] | length')

TOTAL_EESS_SIN_PRECIO_DIESEL=$(echo "$estaciones"
  | jq '[.[] | select(.priceDiesel==null)] | length')

TOTAL_EESS_SIN_COORDENADAS=$(echo "$estaciones"
  | jq '[.[] | select(.lat==null or .lon==null)] | length')

echo "[...] (Informe) INFO Carencia de datos:
      sin_G95=$TOTAL_EESS_SIN_PRECIO_GASOLINA
      sin_Diesel=$TOTAL_EESS_SIN_PRECIO_DIESEL
      sin_Coords=$TOTAL_EESS_SIN_COORDENADAS"
```

Listing 4.3: Figura 4.3 — Métricas de calidad del dato

Con la base ya coherente, se preparan los subconjuntos por combustible con un criterio conservador: solo entran estaciones con precio numérico válido. Este filtrado temprano aún no calcula estadísticas, pero garantiza que los módulos de medida trabajen sobre información consistente y evita lógica defensiva de última hora en los informes. Además, al separar desde ya `estaciones_g95` y `estaciones_diesel`, se simplifica la lectura del script y se reduce el riesgo de mezclar tipos en expresiones complejas.

```
# Conteo de estaciones validas por combustible
G95_COUNT=$(echo "$estaciones" | jq '
  [ .[] | .priceGasolina ]
  | map(select(.!=null))
  | length')

DIESEL_COUNT=$(echo "$estaciones" | jq '
  [ .[] | .priceDiesel ]
  | map(select(.!=null))
  | length')
```

Listing 4.4: Figura 4.4 — Filtrado por combustible

Por último, se incorpora un guardarraíl para casos límite. Si, tras filtrar, ninguno de los combustibles dispone de precios válidos, el proceso se detiene con un error claro. Es preferible declarar "hoy no hay datos útiles" a producir informes vacíos o engañosos. Este patrón, detenerse cuando falta la precondición fundamental, mantiene la integridad del sistema y simplifica el diagnóstico: basta con leer el final del log para saber por qué no se generaron resultados ese día, tal y como se detalla en el Punto 6 con la pauta de registro y el `trap` de finalización.

```
if (( ${G95_COUNT:-0} > 0 )); then
  # calcular metricas
  echo "[...] (Informe) OK Estadisticos descriptivos obtenidos
        para Gasolina 95"
else
  G95_MIN=""; G95_MAX=""; G95_AVG=""
  echo "[...] (Informe) ERROR No hay datos validos para
        Gasolina 95"
fi

if (( ${DIESEL_COUNT:-0} > 0 )); then
  # calcular metricas
  echo "[...] (Informe) OK Estadisticos descriptivos obtenidos
        para Diesel"
else
  DIESEL_MIN=""; DIESEL_MAX=""; DIESEL_AVG=""
  echo "[...] (Informe) ERROR No hay datos validos para Diesel"
fi
```

Listing 4.5: Figura 4.5 — Guardarraíl para conjuntos vacíos

Con estas validaciones, normalizaciones y controles, el proyecto entrega a los apartados siguientes un dataset tipado, segmentado y coherente. Desde aquí, el Punto 5 puede centrarse en medir y presentar, y cualquier referencia a planificación, estructura de carpetas o criterios de logging queda en los Puntos 3 y 6.

Chapter 5

INFORME E INDICADORES

Con el dataset validado y normalizado (Punto 4), este apartado se centra en medir y presentar. El objetivo es doble: calcular indicadores básicos pero informativos por combustible (cuántas estaciones tienen precio válido y cuáles son el mínimo, la media y el máximo) y ofrecer el resultado en dos formatos que cubren usos distintos: un TXT orientado a lectura rápida y un HTML más cómodo para consulta en pantalla. Todo mantiene la trazabilidad por `RUN_ID` y deja hitos en el log, sin repetir la política de registro (Punto 6).

El cálculo parte de los subconjuntos `estaciones_g95` y `estaciones_diesel` preparados en el Punto 4. Para cada combustible se obtiene el conteo de estaciones con precio válido y, si el conteo es mayor que cero, se calculan mínimo, máximo y media aritmética. Esta precaución evita divisiones por cero y garantiza que, si el proveedor devuelve un bloque vacío, el informe lo refleje con un mensaje explícito y no con cifras espurias. El trabajo opera sobre valores numéricos reales (convertidos con `tonumber?` en el Punto 4), de modo que ordenar y agregar es directo y fiable.

```
# Metricas Gasolina 95
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')

  echo "[...] (Informe) OK Estadisticos descriptivos obtenidos
        para Gasolina 95"
else
  G95_MIN=""; G95_MAX=""; G95_AVG=""
  echo "[...] (Informe) ERROR No hay datos validos para
        Gasolina 95"
fi
```

Listing 5.1: Figura 5.1 — Cálculo de métricas por combustible

Además de las estadísticas globales, el informe incluye una selección de estaciones con mejor precio. Se ordena por el campo de precio ascendente y se extrae un Top-5 (si hay menos de cinco, se muestran las disponibles). Al ser precios numéricos, la ordenación es estable y coherente. La intención no es construir un ránking complejo, sino dar una pista diaria de dónde mirar primero. Esta selección alimenta tanto el TXT como el HTML.

```
echo "[...] (Informe) INFO Calculando Top 5 Gasolina 95
      por precio"

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

G95_TOP_N=$(echo "$TOP5_G95" | jq 'length' || echo 0)
if (( ${G95_TOP_N:-0} > 0 )); then
  echo "[...] (Informe) OK Ranking Top 5 Gasolina 95 generado
        (registros=$G95_TOP_N)"
else
  echo "[...] (Informe) ERROR Sin datos validos para Top 5
        Gasolina 95"
fi
```

Listing 5.2: Figura 5.2 — Selección del Top-5 por combustible

Para la presentación se unifica el formateo de precios. Internamente todo usa punto decimal; hacia el usuario se muestra con coma y tres decimales para mantener homogeneidad y no depender del locale. Se define una utilidad breve que transforma el número en cadena formateada; así, el mismo criterio se aplica en cabeceras y filas del Top-5, evitando discrepancias entre TXT y HTML.

```
# Dentro del procesamiento con jq para redondear a 3 decimales:
(. * 1000 | round / 1000)

# Para sustituir punto por coma se puede usar gsub en la salida:
| tostring | gsub("\\."; ",")
```

Listing 5.3: Figura 5.3 — Formateo de precios en jq

El informe TXT prioriza la lectura inmediata. Abre con una cabecera que muestra la fecha de ejecución (`FECHA_CRON`, del Punto 3) y la fecha oficial del dataset (`FECHA_API`, del Punto 4) y, a continuación, presenta dos bloques (Gasolina 95 y Diésel A) con el conteo de estaciones y las tres estadísticas (mín/med/max). Debajo, una lista numerada con el Top-5 indicando rótulo, municipio, precio y dirección. Si un combustible carece de precios válidos, se imprime un aviso claro ("No hay precios válidos..."). La estructura busca que, al abrir el fichero, sea evidente qué día y qué datos se ven y, en dos líneas, el rango de precios y por dónde empezar.

```
generar_informe_txt() {
  if (( ${TOTAL_EESS:-0} > 0 )); then
    echo "[...] (Informe) INFO Generando informe en txt"
    {
      echo "================================================"
      echo "   INFORME DE GASOLINERAS DE VALENCIA"
      echo "================================================"
      echo ""
      echo "Fecha del cron:        $FECHA_CRON"
      echo "Fechas de la API:      $FECHA_API"
      echo ""
      echo "------------------------------------------------"
      echo "GASOLINA 95"
      echo "------------------------------------------------"
      echo "Precio minimo:         $G95_MIN EUR/L"
      echo "Precio maximo:         $G95_MAX EUR/L"
      echo "Precio medio:          $G95_AVG EUR/L"
      echo ""
      echo "Top 5 mas baratas:"
      echo "$TOP5_G95"
      # ... (sigue con Diesel)
    } > "$NOMBRE_ARCHIVO_INFORME_TXT"
    echo "[...] (Informe) OK Informe generado en:
          $NOMBRE_ARCHIVO_INFORME_TXT"
  fi
}
```

Listing 5.4: Figura 5.4 — Generación del informe TXT (extracto)

El informe HTML ofrece la misma información organizada en tablas y con una capa visual mínima. Incluye un encabezado con las dos fechas (`FECHA_CRON` y `FECHA_API`) y, para cada combustible, un título con insignias de min/med/max formateadas. Debajo, una tabla con el Top-5 (estación, municipio, precio con coma decimal y dirección). Cada fila recibe una clase según su relación con la media del combustible (`low`, `avg`, `high`) para resaltar lo más barato sin recurrir a gráficos ni dependencias externas. El HTML es autoinclusivo (estilos embebidos). Si un combustible no tiene datos válidos, se muestran cifras vacías y una tabla sin filas, en coherencia con el TXT.

```
generar_informe_html() {
  echo "[...] (Informe) INFO Generando informe en html"

  cat > "$NOMBRE_ARCHIVO_INFORME_HTML" <<EOF
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Informe de Gasolineras de Valencia</title>
  <style>
    body { font-family: sans-serif; margin: 20px; }
    .price.low { color: green; font-weight: bold; }
    .price.high { color: red; font-weight: bold; }
    .price.avg { color: black; }
    table { border-collapse: collapse; }
    th, td { border: 1px solid #ddd; padding: 8px; }
  </style>
</head>
<body>
  <h1>Informe de Gasolineras de Valencia</h1>
  <p>Fecha del cron: $FECHA_CRON</p>
  <p>Fechas de la API: $FECHA_API</p>

  <h2>GASOLINA 95</h2>
  <p>Minimo: <span class="price low">$G95_MIN EUR/L</span></p>
  <p>Maximo: <span class="price high">$G95_MAX EUR/L</span></p>
  <p>Medio: <span class="price avg">$G95_AVG EUR/L</span></p>

  <h3>Top 5 mas baratas:</h3>
  <table>
    <tr><th>ID</th><th>Nombre</th><th>Direccion</th>
        <th>Precio</th></tr>
$(echo "$TOP5_G95" | jq --arg avg "$G95_AVG" -r '
  .[] | "<tr><td>\(.id)</td><td>\(.name)</td>
         <td>\(.addr)</td>
         <td class=\"price " +
         (if .priceGasolina < (\$avg | tonumber) then "low"
          elif .priceGasolina > (\$avg | tonumber) then "high"
          else "avg" end) + "\">
         \((.priceGasolina * 1000 | round / 1000))
         </td></tr>"')
  </table>
  <!-- ... (sigue con Diesel) -->
</body>
</html>
EOF
}
```

Listing 5.5: Figura 5.5 — Generación del informe HTML (extracto)

Ambos informes se nombran con el `RUN_ID` de la ejecución y se guardan bajo `informes/`. Así se preserva la trazabilidad de los Puntos 2 y 3: al ver un informe, se localizan el JSON crudo del mismo `RUN_ID` y la traza correspondiente en el log. El script deja constancia explícita de la generación del TXT y del HTML con mensajes INFO, lo que permite comprobar de un vistazo —en el Punto 6— si la ejecución llegó

a buen puerto y qué artefactos produjo.

```
echo "[...] (Informe) OK Informe TXT generado en:
      $NOMBRE_ARCHIVO_INFORME_TXT"

echo "[...] (Informe) OK Informe HTML generado en:
      $NOMBRE_ARCHIVO_INFORME_HTML"
```

Listing 5.6: Figura 5.6 — Confirmación de informes generados

Con esta capa de indicadores y presentación, el proyecto ofrece una salida diaria clara y utilizable sin repetir lógica de limpieza ni detalles de planificación. El Punto 6 cerrará el círculo describiendo el criterio de registro y calidad: qué se escribe en el log, cómo se distingue éxito de error y cómo localizar con rapidez una incidencia cuando algo no cuadra.

Chapter 6

REGISTRO Y CALIDAD

Este apartado define cómo se audita cada ejecución y qué señales permiten juzgar su calidad sin abrir el código ni los datos crudos. La idea es que, leyendo el log, pueda saberse qué pasó, cuándo, en qué fase, con qué resultado y con qué calidad de datos. Para ello se usan mensajes con marca temporal ISO (`[%F %T]`), una taxonomía de fases alineada con el flujo del proyecto, niveles de severidad simples (INFO, OK, ERROR) y una línea de finalización inequívoca que refleja el código de salida del proceso. Además, se separan dos canales: `log.txt` (bitácora funcional del script) y `cron.log` (todo lo que cron redirige por stdout/stderr; ver Punto 3). Con esta combinación, diagnosticar una incidencia lleva segundos: si cron no lanza, el rastro está en `cron.log`; si el script encuentra un JSON inválido o una tabla vacía, queda documentado en `log.txt`.

La forma del mensaje evita ambigüedades y favorece el análisis posterior con herramientas básicas (grep, less, awk). Cada línea empieza por `[YYYY-MM-DD HH:MM:SS]`, sigue con la fase entre paréntesis (Inicio, Descarga, Procesamiento, Indicadores, Informe, Finalizacion), un nivel y un mensaje claro. Las fases están ordenadas como el propio flujo, de modo que lo que aparece en (Indicadores) nunca precede a (Procesamiento) y es trivial localizar el tramo en el que ocurrió un problema. Los niveles tienen significado estable: INFO señala un hito (por ejemplo, "haciendo petición a la URL"), OK marca chequeos superados (por ejemplo, "JSON válido", "Top-5 calculado") y ERROR identifica fallos que requieren atención; si el error es crítico, el script sale con código distinto de cero en ese mismo punto.

```
# Ejemplos de mensajes de log en distintas fases:

echo "[$(date '+%F %T')] (Inicio) INFO Identificador de la
      ejecucion del CRON(FECHA)=$RUN_ID" >> $NOMBRE_ARCHIVO_LOG

echo "[$(date '+%F %T')] (Descarga) INFO Haciendo peticion
      a $urlValencia" >> $NOMBRE_ARCHIVO_LOG

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

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

echo "[$(date '+%F %T')] (Indicadores) INFO Calculando
      Top 5 Gasolina 95 por precio" >> "$NOMBRE_ARCHIVO_LOG"

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

Listing 6.1: Figura 6.1 — Formato de mensajes y fases del log

Cuando aparece un ERROR, el criterio es salida controlada: se registra el motivo y se interrumpe la ejecución con `exit 1`. Esto evita falsos positivos (informes generados a medias o con datos corruptos) y deja una pista inequívoca para quien revise los logs. Ejemplos típicos son un HTTP no 2xx en la descarga (Punto 2) o un JSON inválido al empezar el procesamiento (Punto 4). En ambos casos, el log explica el porqué y la ejecución no avanza a etapas que dependerían de un dato inexistente o incoherente.

```
# Error en descarga HTTP
if [ "$status" -ge 200 ] && [ "$status" -lt 300 ]; then
  echo "[...] (Descarga) OK Peticion Valida:
        Estado Peticion HTTP = $status"
else
  echo "[...] (Descarga) ERROR Ha fallado la peticion a
        $urlValencia. Error HTTP: $status"
  exit 1
fi

# Error en validacion JSON
echo "$getEstacionesValencia" | jq empty > /dev/null \
  && echo "[...] (Procesamiento) OK JSON valido" \
  || { echo "[...] (Procesamiento) ERROR JSON invalido";
       exit 1; }
```

Listing 6.2: Figura 6.2 — Registro de errores con salida controlada

Para que quede constancia del resultado global, al final del script se usa un `trap` que captura el código de salida y escribe una línea de cierre con la fase (Finalizacion). Si el código es 0, se indica que "el programa se ha ejecutado correctamente"; si no, se informa del código concreto. Con esto, leer el último bloque de `log.txt` basta para

saber si el día ha terminado bien o mal, y cualquier monitorización externa puede buscar esa cadena para alertar solo cuando falte o indique error. Esta línea de cierre se complementa con los INFO de generación del TXT y del HTML (Punto 5), que enlazan el estado del programa con los artefactos producidos.

```
# Trap para comprobar el exito de la ejecucion
trap 'code=$?; if [ $code -eq 0 ]; then
  echo "[$(date "+%F %T")] (Finalizacion) INFO El programa
        se ha ejecutado correctamente" >> "$NOMBRE_ARCHIVO_LOG"
else
  echo "[$(date "+%F %T")] (Finalizacion) ERROR El programa
        termino con codigo $code" >> "$NOMBRE_ARCHIVO_LOG"
fi' EXIT
```

Listing 6.3: Figura 6.3 — Finalización inequívoca con trap

La separación entre `log.txt` y `cron.log` añade otra capa de claridad. `log.txt` recoge lo que hace el script (fases, validaciones, indicadores, informes), mientras que `cron.log` captura cómo se lanzó (errores de ruta, permisos, PATH incompleto, etc.). Dado que cron ejecuta con un entorno mínimo (ver Punto 3), redirigir stdout y stderr a `cron.log` evita perder mensajes del planificador y permite diferenciar si el problema estaba antes de entrar en el script (por ejemplo, "no se encuentra el fichero") o durante su ejecución (por ejemplo, "JSON inválido").

```
# Ejemplo de linea en crontab con redireccion:
SHELL=/bin/bash
PATH=/usr/local/bin:/usr/bin:/bin

0 8 * * * /ruta/absoluta/al/proyecto/analisis_json.sh \
  >> /ruta/absoluta/al/proyecto/cron.log 2>&1
```

Listing 6.4: Figura 6.4 — Redirección de cron a cron.log

Además del estado del programa, el log deja señales sobre la calidad del dato. Como se definió en el Punto 4, por cada ejecución se anotan el número total de estaciones recibidas, cuántas carecen de precio en Gasolina 95 y Diésel y cuántas llegan sin coordenadas. Estas cifras no están dirigidas al usuario final, pero sí a quien mantiene la automatización: si un día el Top-5 es corto o la media es extraña, el log permite atribuirlo de inmediato a una carencia objetiva de la fuente en esa fecha, en lugar de culpar a la lógica del informe (Punto 5). La interpretación de resultados se apoya así en hechos verificables.

```
TOTAL_EESS=$(echo "$estaciones" | jq '. | length')

TOTAL_EESS_SIN_PRECIO_GASOLINA=$(echo "$estaciones"
  | jq '[.[] | select(.priceGasolina==null)] | length')

TOTAL_EESS_SIN_PRECIO_DIESEL=$(echo "$estaciones"
  | jq '[.[] | select(.priceDiesel==null)] | length')

TOTAL_EESS_SIN_COORDENADAS=$(echo "$estaciones"
  | jq '[.[] | select(.lat==null or .lon==null)] | length')

echo "[...] (Informe) INFO Carencia de datos:
      sin_G95=$TOTAL_EESS_SIN_PRECIO_GASOLINA
      sin_Diesel=$TOTAL_EESS_SIN_PRECIO_DIESEL
      sin_Coords=$TOTAL_EESS_SIN_COORDENADAS" >> $NOMBRE_ARCHIVO_LOG
```

Listing 6.5: Figura 6.5 — Señales de calidad de datos en el log

Toda esta traza se ancla en el `RUN_ID` de la ejecución (Puntos 2 y 3). Aunque `RUN_ID` no aparezca en cada línea, sí figura en los mensajes clave (por ejemplo, al anunciar las rutas de los informes generados), lo que permite saltar del log al TXT/HTML y al JSON crudo correspondiente. Este vínculo sostiene la auditoría reproducible: si algo no cuadra en un informe, se puede reconstruir la historia completa con los artefactos exactos de ese día y hora.

Como buenas prácticas operativas, se mantiene el timestamp ISO para facilitar ordenación y búsquedas; se evita la verbosidad inútil (mensajes concisos y accionables); y se cuida el locale al formatear precios (la función del Punto 5 fuerza notación coherente sin depender de la configuración del sistema). Para instalaciones con varias semanas de ejecución, es recomendable rotar `log.txt` y `cron.log` con una política sencilla (retener 7 días, comprimir y truncar), de forma que el tamaño de los ficheros no crezca indefinidamente sin aportar valor.

```
# Archivo /etc/logrotate.d/shell-proyect
/ruta/absoluta/al/proyecto/log.txt
/ruta/absoluta/al/proyecto/cron.log {
    daily
    rotate 7
    compress
    copytruncate
    missingok
    notifempty
}
```

Listing 6.6: Figura 6.6 — Propuesta de logrotate (opcional)

Con este esquema, el proyecto no solo hace (descarga, valida, calcula, informa), también explica lo que hace de manera legible y verificable. La consecuencia práctica es doble: resolución rápida de incidencias y confianza en que cada informe está respaldado

por un rastro claro, desde el dato crudo hasta la línea final de éxito o error.