**Informe técnico – Inspector de anomalías (PatchCore)**

Contenido

[**Informe técnico – Inspector de anomalías (PatchCore)** 1](#_Toc210734081)

[**1) Resumen ejecutivo** 1](#_Toc210734082)

[**2) Arquitectura general** 2](#_Toc210734083)

[**3) Backend – detalles y justificación de ajustes** 2](#_Toc210734084)

[**3.1 Extracción de características (PatchCore)** 2](#_Toc210734085)

[**3.2 Mapa, score y umbral** 2](#_Toc210734086)

[**3.3 ROI (Region of Interest)** 3](#_Toc210734087)

[**3.4 Polígonos y limpieza de máscara** 3](#_Toc210734088)

[**3.5 Organización de rutas y paths** 3](#_Toc210734089)

[**4) Frontend – mejoras implementadas** 3](#_Toc210734090)

[**5) Configuración por entorno (.env)** 4](#_Toc210734091)

[**6) Endpoints (contrato)** 4](#_Toc210734092)

[**7) Estructura del repo** 5](#_Toc210734093)

[**8) Puesta en marcha (local)** 5](#_Toc210734094)

[**9) Control de versiones (hoy)** 5](#_Toc210734095)

[**10) Beneficios de los cambios clave** 6](#_Toc210734096)

[**11) Riesgos y consideraciones** 6](#_Toc210734097)

[**12) Próximos pasos (sugeridos)** 6](#_Toc210734098)

**1) Resumen ejecutivo**

Hoy dejamos **operativo y versionado** un sistema completo de inspección visual por anomalías usando **PatchCore + FastAPI** con frontend web. Se puede correr localmente (CPU/GPU), analizar imágenes sueltas y visualizar *overlays* con polígonos de la zona anómala. Además, subimos todo a GitHub (con Git LFS para binarios), dejamos el frontend separado (HTML/CSS/JS) y ordenamos la salida de resultados en static/overlays/.

**2) Arquitectura general**

* **Backend:** FastAPI + PyTorch + OpenCV + scikit-learn.
* **Modelo:** PatchCore “ligero” montado sobre **ResNet-18** (features de layer2 y layer3 + KNN).
* **Frontend:** HTML + CSS + JS vanilla (sin frameworks), servido por FastAPI.
* **Persistencia de visualizaciones:** imágenes de calor/overlay/máscara generadas por petición, guardadas en static/overlays/ y expuestas vía /static/....
* **API:**
  + GET / → sirve el frontend.
  + GET /health → devuelve configuración efectiva.
  + POST /predict → procesa imagen y retorna JSON con score, threshold, is\_anomaly, polygons[] y overlay\_url.

**3) Backend – detalles y justificación de ajustes**

**3.1 Extracción de características (PatchCore)**

* **Backbone:** resnet18 con pesos ImageNet.
* **Hooks:** capturamos **layer2** y **layer3**; después **interpolamos** layer3 al tamaño de layer2 y **concatenamos** (más contexto + detalle).
* **Normalización:** L2 por fila en el banco y en los parches de inferencia → distancias comparables y estables.
* **KNN:** NearestNeighbors con n\_neighbors = KNN\_K (en .env = 5). El promedio de las distancias a los k vecinos produce el **mapa de anomalía**.

**Motivo:** ResNet-18 equilibra velocidad/precisión; combinar layer2+layer3 replica la idea de multiescala de PatchCore; normalizar y promediar distancias reduce ruido.

**3.2 Mapa, score y umbral**

* Se calcula un **mapa de calor** (promedio de distancias por parche) y se reescala a IMG\_SIZE.
* Guardamos hmin/hmax para **normalizar** el mapa a [0,1].
* **Score** = máx del mapa **dentro de la ROI** (si hay ROI) y si no, en toda la imagen.
* **Umbral efectivo (threshold)**:
  + Base desde .env o config.json.
  + Modos de sensibilidad en mode:
    - sensitive → threshold \* 0.8
    - strict → threshold \* 1.2
  + Campo thr en la query **sobrescribe** todo (umbral manual).
* Para alinear polígonos con la decisión, convertimos el umbral a espacio normalizado:  
  thr\_norm = (threshold - hmin) / (hmax - hmin + 1e-8)  
  y **binarizamos** el heatmap normalizado con thr\_norm.

**Motivo:** Garantiza que la máscara/contornos correspondan **exactamente** al umbral que decide is\_anomaly, evitando discrepancias visuales.

**3.3 ROI (Region of Interest)**

* Dos opciones:
  1. **IGNORE\_BORDER\_PCT**: enmascara bordes (ej. 10%) para ignorar ruido en periferia.
  2. **ROI\_PATH**: PNG binaria externa (blanco=ROI).
* El **score** se calcula solo dentro de la ROI.
* En el **overlay** dibujamos el **borde de la ROI en amarillo** (Canny) como referencia.

**Motivo:** En línea con visión industrial: se ignoran zonas irrelevantes o con iluminación inestable.

**3.4 Polígonos y limpieza de máscara**

* Tras binarizar el mapa (con thr\_norm), se aplica **morph open/close** para reducir ruido.
* Se filtran contornos por **área mínima** (AREA\_MIN, p.ej. 200 px).
* Se devuelven **polígonos** solo si is\_anomaly = true (menos ruido en casos normales).

**Motivo:** Se mejora legibilidad y se evitan falsos positivos pequeños.

**3.5 Organización de rutas y paths**

* Creamos helper \_abs() y BASE\_DIR para resolver variables relativas del .env (Windows friendly).
* Aseguramos STATIC\_DIR y **subcarpeta dedicada** OVERLAYS\_SUBDIR="overlays" → evita llenar la raíz de static/.

**Motivo:** Carpetas consistentes y portables; menos errores por rutas relativas en distintos equipos.

**4) Frontend – mejoras implementadas**

* **Separación real** en templates/index.html, static/css/app.css y static/js/app.js.
* UI limpia con:
  + **Preview** de imagen.
  + Selector entre **umbral del backend** vs **umbral manual**.
  + **Sensibilidad** (normal, sensitive, strict).
  + **Chips de configuración** (img\_size, knn\_k, threshold\_base) leyendo GET /health.
  + **Badges** de resultado: **NORMAL** / **ANOMALÍA**.
  + Vista del **overlay** y **JSON** de respuesta para depurar.

**Motivo:** Hicimos el flujo de prueba rápido, visualmente claro y sin dependencias externas.

**5) Configuración por entorno (.env)**

Variables clave (todas leídas en el backend):

* IMG\_SIZE (ej. 256) → debe coincidir con el memory bank.
* KNN\_K (ej. 5) → vecinos para KNN.
* THRESHOLD (ej. 0.40) → umbral base estable.
* AREA\_MIN (ej. 200) → filtro de ruido en polígonos.
* IGNORE\_BORDER\_PCT (ej. 10) / ROI\_PATH → control de ROI.
* ARTIFACTS\_DIR → carpeta del memory bank y config.
* STATIC\_DIR → carpeta pública estática (sirve /static).
* OVERLAYS\_SUBDIR → subcarpeta donde se guardan overlays.

**Nota:** Si quieren que .env se cargue automáticamente, se puede añadir:

from dotenv import load\_dotenv; load\_dotenv()

al inicio del main.py. Hoy dejamos lectura directa de variables de entorno (PowerShell setx / $env:).

**6) Endpoints (contrato)**

* GET /health  
  Devuelve:
* {
* "status":"ok", "device":"cuda|cpu", "img\_size":256,
* "knn\_k":5, "threshold":0.4,
* "ignore\_border\_pct":10.0, "roi\_path":null|".../roi.png"
* }
* POST /predict (FormData: file + opcionales thr, mode)  
  Respuesta:
* {
* "score": 0.612341,
* "threshold": 0.4,
* "is\_anomaly": true,
* "polygons": [[[x,y],...], ...],
* "overlay\_url": "/static/overlays/nombre\_overlay.png"
* }

**7) Estructura del repo**

Backend/

main.py

models/patchcore/

config.json

memory\_bank\_core.npz

static/

css/app.css

js/app.js

overlays/ # ← resultados generados

templates/

index.html

requirements.txt

Dataset/ # (incluido hoy, con Git LFS)

notebooks/

.gitattributes # configurado para LFS (png/npz/etc)

.gitignore # incluye venv, \_\_pycache\_\_, overlays

.env.example

README.md

**8) Puesta en marcha (local)**

# Clonar con LFS (importante):

git lfs install

git clone https://github.com/DelcastApe/Inspector-anomalias-patchcore.git

cd Inspector-anomalias-patchcore/Backend

# Entorno

python -m venv .venv

.venv\Scripts\activate

pip install -r requirements.txt

# Opcional: copiar .env

copy ..\.env.example .env

# Levantar

uvicorn main:app --reload --port 8000

# Abrir: http://127.0.0.1:8000

**9) Control de versiones (hoy)**

* Creamos repo, configuramos **branch main**, añadimos **Git LFS** para binarios grandes (imágenes, .npz).
* **Primer push**: backend + frontend + notebooks + dataset (para el equipo).
* Ajustamos README.md y .gitattributes/.gitignore.
* Verificación en GitHub: estructura correcta y contenidos visibles.

**10) Beneficios de los cambios clave**

* **Umbral coherente** entre decisión y polígonos (mapeo a thr\_norm): explicabilidad y consistencia.
* **ROI flexible** (borde ignorado o máscara externa) aumenta robustez y reduce falsos positivos.
* **Overlays separados** en static/overlays/: orden y limpieza del proyecto.
* **Frontend ordenado** y sin dependencias, apto para demos rápidas.
* **Paths robustos** (Windows/Linux) gracias a \_abs() y BASE\_DIR.
* **/health** permite verificar **config efectiva** al vuelo.

**11) Riesgos y consideraciones**

* **Dataset en el repo:** hoy lo incluimos para facilitar la ejecución del equipo; en proyectos reales se recomienda **no versionar** datos grandes o sensibles y usar artefactos versionados por release.
* **Torch/cuDNN:** en máquinas con GPU conviene instalar el wheel específico de CUDA; de lo contrario, se usa CPU (más lento).
* **Consistencia con memory bank:** IMG\_SIZE, PATCH\_STRIDE y el banco deben corresponder al entrenamiento.

**12) Próximos pasos (sugeridos)**

1. **Dockerización** (Dockerfile + .dockerignore + docker-compose.yml) para correr en 1 comando.
2. **Release v0.1** en GitHub con binarios/artifacts adjuntos.
3. **Tests rápidos** (al menos smoke tests para /health y /predict).
4. **Métricas opcionales** (tiempo de inferencia, tamaño de defectos).
5. **Soporte de lotes** y/o integrar webcam si aplica.

.