Educación
Secretaría de Educación Pública

TECNOLÓGICO NACIONAL DE MÉXICO

Instituto Tecnológico de Jiquilpan

INSTITUTO TECNOLÓGICO DE JIQUILPAN

“MICROSERVICIO DE INGESTIÓN PARA IA CON NESTJS”

INVESTIGACIÓN DE INFRAESTRUCTURA

INGENIERIA EN SISTEMAS COMPUTACIONALES

PRESENTA:

MORA NUÑEZ SANTIAGO ANTONIO

Erick Dalet Villanueva Mascort

JIQUILPAN, MICHOACÁN, ENERO DE 2026

2026 año de Margarita Maza

Carretera Nacional s/n Km. 202 C.P. 59510 Jiquilpan, Michoacán.
Tel. (353) 533 1126, 533 0574, 533 2348, 533 3608 y 533 3091
tecnm.mx | www.jiquilpan.tecnm.mx

Concepto Central

El microservicio de ingestión constituye el punto de entrada crítico del sistema de detección de frutas con IA. Su responsabilidad principal es recibir imágenes desde las cámaras industriales, validarlas en tiempo real y distribuirlas eficientemente hacia los componentes de procesamiento, todo esto manteniendo una latencia mínima y un throughput elevado para soportar múltiples cámaras transmitiendo simultáneamente.

Arquitectura de Recepción: Fastify como Motor de Streaming

La Decisión Arquitectónica Crítica

La arquitectura tradicional basada en Express con Multer presenta una limitación fundamental para este caso de uso: carga completa en memoria.
Cuando una cámara envía una imagen de 5MB, el servidor Express con Multer debe:

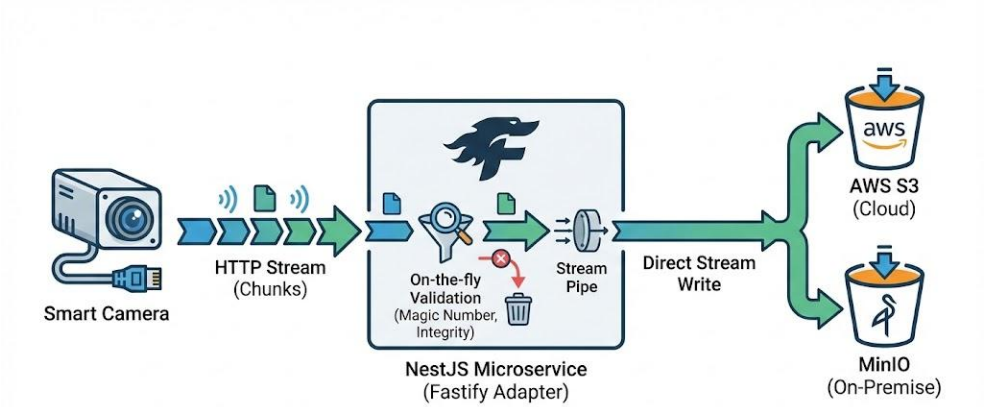
- Recibir el archivo completo en un buffer temporal
- Almacenarlo en la memoria RAM del servidor
- Validarlo después de la carga completa
- Finalmente transferirlo al almacenamiento destino

Este flujo genera dos problemas críticos en un entorno de producción industrial:
Saturación de Memoria: Con 10 cámaras enviando imágenes simultáneamente, el servidor debe mantener hasta 50MB en RAM solo para buffering, sin contar el overhead del runtime de Node.js.

Latencia Acumulativa: Cada cámara debe esperar a que su archivo se cargue completamente antes de que el servidor pueda comenzar a procesarlo, creando un cuello de botella secuencial.

Arquitectura Propuesta: Pipeline de Streaming con Fastify

NestJS abandona su configuración por defecto para implementar el adaptador Fastify con @fastify/multipart. Esta decisión arquitectónica transforma el flujo de datos de un modelo de "carga y procesa" a un modelo de pipeline continuo:



Real-time streaming architecture for low-latency image ingestion.

Diferencia clave: Fastify no espera a que el archivo se cargue completo en memoria. En su lugar, abre una "tubería" y transfiere los datos bit a bit directamente desde la solicitud HTTP hacia el almacenamiento final. Mientras los primeros megabytes ya están siendo escritos en S3, los últimos aún están siendo transmitidos por la cámara.

Implicación técnica importante: Al usar platform-fastify, los decoradores estándar de NestJS cambian su comportamiento. El desarrollador debe acceder directamente al objeto req de Fastify para manejar el flujo de datos masivo, renunciando a algunas abstracciones de NestJS a cambio de control granular sobre el stream.

NestJS como Orquestador No Bloqueante

El rol de NestJS en esta arquitectura no es "procesar" las imágenes, sino orquestar su flujo sin convertirse en un cuello de botella. Este patrón se conoce como "Gateway asíncrono":

Principio de Delegación

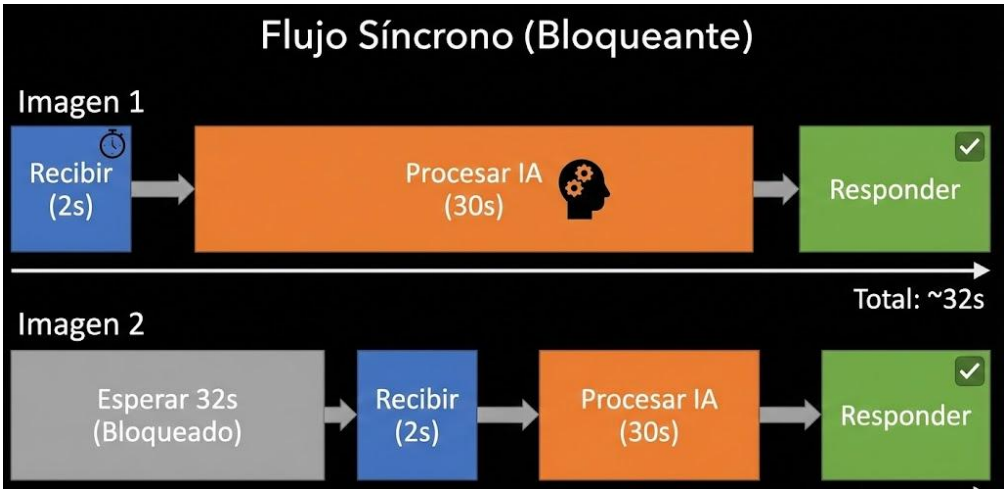
NestJS actúa como un proxy inteligente que:

- Recibe el stream de la cámara
- Valida la integridad en tiempo real (sin detener el flujo)
- Redirige el stream hacia el almacenamiento
- Notifica a los consumidores sobre la disponibilidad de datos
- Responde inmediatamente a la cámara sin esperar el procesamiento de IA

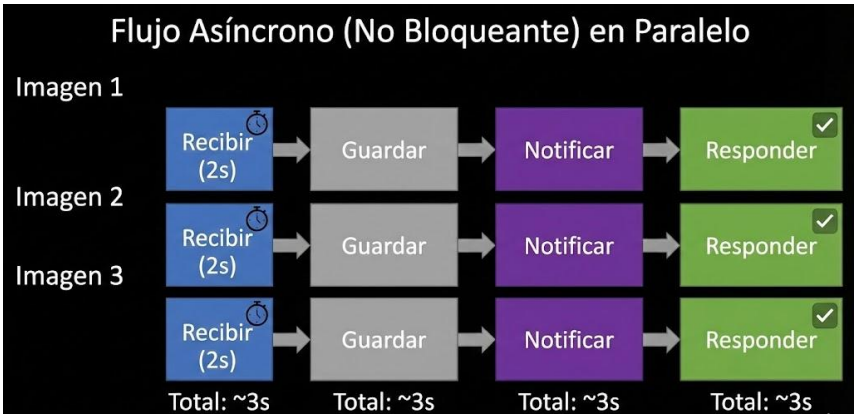
La clave está en que NestJS nunca espera a que el modelo de IA termine su inferencia. Una vez que la imagen está en S3 y el mensaje está en la cola, NestJS se desentiende completamente del proceso y queda libre para recibir la siguiente transmisión.

Ventaja de Throughput

En la arquitectura tradicional bloqueante:



En la arquitectura propuesta no bloqueante:



Mientras tanto, un Worker independiente procesa las 3 imágenes en paralelo

Seguridad y Validación: Protección del Pipeline de IA

Los sensores industriales pueden fallar, las conexiones pueden interrumpirse y los archivos pueden corromperse. NestJS implementa una validación en vuelo utilizando sus Pipes y Guards para proteger el pipeline de datos antes de que lleguen al almacenamiento:

Validación de Tipo mediante Magic Numbers

En lugar de confiar en la extensión del archivo (.jpg), que puede ser falsificada, NestJS inspecciona los primeros bytes del stream en hexadecimal para verificar la firma real del archivo:

JPG: FF D8 FF

PNG: 89 50 4E 47

Esta validación ocurre durante el streaming, leyendo únicamente los primeros bytes sin necesidad de cargar el archivo completo. Si la firma no coincide, NestJS aborta el stream inmediatamente, evitando desperdiciar ancho de banda y almacenamiento.

Validación de Integridad

NestJS monitorea continuamente el estado del stream. Si:

- La conexión HTTP se corta abruptamente
- El tamaño declarado en headers no coincide con los bytes recibidos
- El stream termina antes de completarse

NestJS lanza una excepción 422 Unprocessable Entity, cierra la conexión con el almacenamiento y libera recursos inmediatamente. Esta validación es crítica ya que garantiza que el modelo de IA nunca reciba imágenes corruptas que podrían generar resultados erróneos en el conteo de frutas.

Configuración del Entorno.

La infraestructura del sistema debe funcionar tanto en la nube AWS como en servidores on-premise dentro de la planta de procesamiento. NestJS utiliza su ConfigModule para lograr esta portabilidad:

```
Import to Postman
// El código fuente permanece idéntico, solo cambian las variables de entorno
// Desarrollo local:
S3_ENDPOINT=http://localhost:9000  // MinIO
QUEUE_HOST=localhost
// Producción en nube:
S3_ENDPOINT=https://s3.amazonaws.com
QUEUE_HOST=rabbitmq.production.com

```

Esta separación permite al equipo DevOps mover el microservicio entre ambientes sin modificar una sola línea de código, reduciendo errores humanos y facilitando la replicación de la arquitectura en múltiples plantas.

Estrategias de Almacenamiento: Data Lake para Datos Crudos

Object Storage como Fundamento

Una vez validado el stream, NestJS debe persistirlo eficientemente. La tecnología elegida es Object Storage, con dos implementaciones según el entorno:

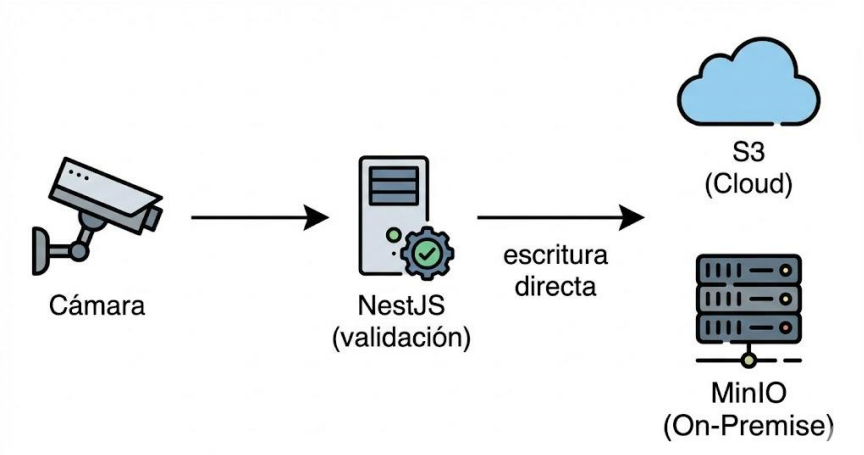
- AWS S3: Para despliegue en la nube (escalabilidad global)
- MinIO: Para despliegue on-premise (control total de datos)

Ambos exponen la misma API S3-compatible, permitiendo a NestJS usar un único código fuente para ambos escenarios.

NestJS como Proxy de Streaming

El rol de NestJS aquí es fundamental ya que actúa como un proxy transparente que redirige el flujo de datos directamente al bucket, sin almacenar nada en su propio sistema de archivos.

El stream fluye así:



¿Por qué Evitamos CDNs?

A diferencia de aplicaciones web tradicionales que usan CDNs como Cloudinary para optimizar imágenes para consumo humano, este sistema deliberadamente evita servicios de optimización de imágenes porque:

1. El consumidor es un robot, no un humano: El modelo de IA (TensorFlow) requiere los píxeles exactos tal como fueron capturados.
2. Los CDNs alteran la imagen: Comprimen, redimensionan o convierten formatos para reducir ancho de banda, modificando los valores RGB que el modelo necesita para la detección precisa.
3. Preservación de calidad: La imagen cruda (RAW) mantiene toda la información visual necesaria para que el algoritmo de detección identifique correctamente las frutas, sus características y defectos.

Procesamiento Asíncrono.

NestJS maneja la logística del sistema, pero delega toda la inteligencia al componente especializado de procesamiento de IA.

Patrón de Comunicación Asíncrona

El microservicio se conecta a una cola de mensajería (RabbitMQ o Redis) implementando un patrón fire-and-forget:

- NestJS termina de subir el archivo a S3
- NestJS publica un evento en la cola:

```
{
    "evento": "nueva_fruta",
    "url": "s3://bucket-frutas/camara-01/img-20260130-143022.jpg",
    "timestamp": "2026-01-30T14:30:22Z",
    "camara_id": "camara-01"
}
```

- NestJS se olvida del proceso inmediatamente
- NestJS queda libre para recibir la siguiente foto en milisegundos

Worker de IA en Python

Un servicio completamente independiente (Python + TensorFlow) escucha continuamente la cola de mensajes:

- Recibe la notificación de nueva imagen
- Descarga el archivo desde S3 usando la URL proporcionada
- Ejecuta el modelo de detección (proceso que puede tomar 10-30 segundos)
- Guarda los resultados del conteo en la base de datos
- Confirma (ACK) el mensaje como procesado

Esta separación es crítica, mientras el Worker procesa una imagen durante 30 segundos, NestJS ya ha recibido y almacenado 50 imágenes más de otras cámaras.

Flujo de Datos Completo.

El flujo ideal de una imagen desde su captura hasta su procesamiento sigue este recorrido:

1. La cámara industrial envía un POST /api/ingestion/upload con la imagen en formato multipart/form-data.

2. NestJS recibe el stream mediante el adaptador Fastify y @fastify/multipart, sin almacenar nada en memoria.

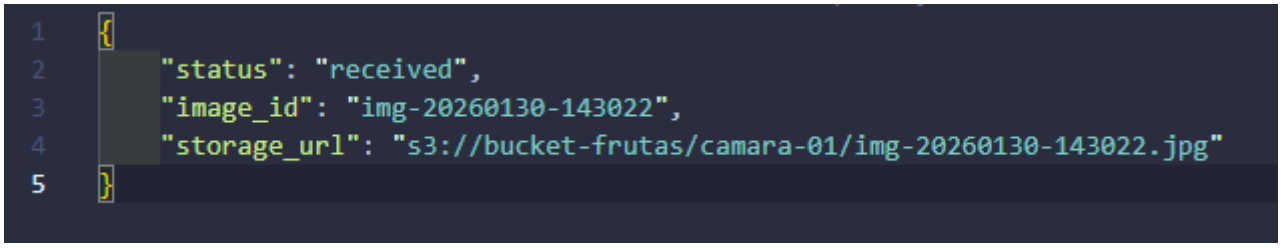
3. Mientras el stream fluye, NestJS:

- Verifica los headers HTTP (Content-Type, Content-Length)
- Inspecciona los magic numbers (primeros bytes del archivo)
- Monitorea la integridad del stream continuamente

4. NestJS hace pipe del stream directamente hacia AWS S3 o MinIO, escribiendo los bytes conforme llegan desde la cámara.

5. Una vez confirmada la escritura en S3, NestJS envía un mensaje a la cola RabbitMQ con la ubicación del archivo.

6. NestJS retorna 201 Created con la URL del recurso a la cámara inmediatamente, sin esperar el procesamiento de IA:



7. El Worker de IA, operando de forma completamente independiente:

- Consume el mensaje de la cola
- Descarga y procesa la imagen
- Guarda el resultado del conteo
- Libera recursos y espera el siguiente mensaje

Esta arquitectura garantiza que el microservicio de ingestión pueda manejar cientos de imágenes por minuto, manteniendo latencias por debajo de 200ms para la respuesta a cada cámara, mientras los Workers de IA procesan las imágenes en paralelo según su capacidad computacional disponible.