🔍 ANÁLISIS: Puntos críticos detectados

1. PIL.Image.resize() ANTES del Processor (SOSPECHOSO #1)

python

```
# sam3_runner.py, línea ~104-109
resized_image = image_rgb
if target_long_side and long_side != target_long_side:
    scale = target_long_side / float(long_side)
    new_width = max(1, int(round(orig_width * scale)))
    new_height = max(1, int(round(orig_height * scale)))
    resized_image = image_rgb.resize((new_width, new_height))  # ⚠
POTENCIALMENTE PELIGROSO
```

Por qué es sospechoso:

- PIL en Windows puede llamar a bibliotecas nativas (libjpeg-turbo, libpng) que tienen **bugs de threading/memory**.
- Si `Sam3Processor` **también hace resize internamente**, estás duplicando la operación y creando tensiones de memoria.
- PIL.Image.resize() **sin especificar `resample`** usa un algoritmo por defecto que puede ser lento/problemático.

Hipótesis: El resize de PIL está bloqueándose en una operación nativa (especialmente con imágenes grandes o corruptas).

---

2. Transformers dev version (5.0.0.dev0) - SOSPECHOSO #2

python

```
from transformers import Sam3Model, Sam3Processor  # ⚠ Versión dev
```

Problema crítico:

- Las versiones `dev` de Transformers **no están probadas para producción**.
- SAM-3 es relativamente nuevo, y los bindings con PyTorch pueden tener **bugs de memoria o deadlocks**.
- El método `post_process_instance_segmentation()` puede tener código no optimizado o con race conditions.

Evidencia indirecta: Si el freeze ocurre **después** de la inferencia (en el postprocesamiento), este es el culpable más probable.

---

3. Threading con PyTorch en Windows (SOSPECHOSO #3)

python

```
# manager.py, línea ~163
thread = threading.Thread(target=self._run_job, args=(job_id,), daemon=True)
thread.start()
```

Por qué es peligroso:

- PyTorch **no es completamente thread-safe** en Windows, especialmente con CUDA.
- Si `torch.set_num_threads()` no se configura explícitamente, PyTorch puede

intentar usar **todos los cores**, lo que en un i7-4790S (4 cores/8 threads) puede causar contención.
- Los **daemon threads** no tienen cleanup garantizado, y si PyTorch está en medio de una operación CUDA, pueden quedarse colgados.

Hipótesis: El thread daemon intenta acceder a CUDA mientras el GIL (Global Interpreter Lock) está bloqueado por otra operación.

---

4. Liberación de memoria incompleta

python

```python
finally:
    inputs = None
    outputs = None
    results = None
    boxes = None
    scores = None
    masks = None
    gc.collect()  # ⚠ NO garantiza liberar memoria CUDA nativa
    if torch and self.device == "cuda":
        torch.cuda.empty_cache()
```

Problema:

- `gc.collect()` **no libera memoria de tensores CUDA** inmediatamente.
- `torch.cuda.empty_cache()` solo libera la caché, **no fuerza la liberación de tensores activos**.
- Si hay referencias cíclicas en los objetos de Transformers, la memoria no se libera.

Consecuencia: Acumulación progresiva de memoria CUDA que, aunque no se vea en el monitor, puede causar un freeze cuando el driver se queda sin espacio.

---

5. Transferencia a CUDA sin validación de tamaño

python

```python
inputs = {
    key: value.to(self.device) if hasattr(value, "to") else value
    for key, value in inputs.items()
}
```

Riesgo oculto:

- Si `inputs` contiene tensores muy grandes (por ejemplo, masks preallocated), `.to(device)` puede intentar transferir **gigabytes** a VRAM de 4GB.
- La transferencia puede fallar silenciosamente o causar un deadlock en el driver de NVIDIA.

🧪 EXPERIMENTOS DE AISLAMIENTO (en orden de prioridad)

Experimento 1: Eliminar el resize manual de PIL

Comenta las líneas 104-109 y deja que Transformers haga el resize:

python

```
# COMENTAR ESTO:
# if target_long_side and long_side != target_long_side:
#     scale = target_long_side / float(long_side)
#     new_width = max(1, int(round(orig_width * scale)))
#     new_height = max(1, int(round(orig_height * scale)))
#     resized_image = image_rgb.resize((new_width, new_height))

# USAR DIRECTAMENTE:
inputs = self.processor(images=image_rgb, text=prompt_text, return_tensors="pt")
```

Si esto elimina el freeze: El culpable es PIL.Image.resize() en Windows.

Experimento 2: Forzar PyTorch single-threaded

Al inicio de `sam3_runner.py`, agrega:

python

```
import torch
torch.set_num_threads(1)  # Fuerza un solo thread
torch.set_num_interop_threads(1)
```

Objetivo: Eliminar contención de threads que podría causar deadlock.

Experimento 3: Eliminar el postprocesamiento

Comenta las líneas 122-126:

python

```
# COMENTAR:
# results = self.processor.post_process_instance_segmentation(
#     outputs,
#     threshold=0.0,
#     mask_threshold=self.mask_threshold,
#     target_sizes=target_sizes,
# )[0]

# DEVOLVER UN PLACEHOLDER:
return []  # Simular sin detecciones
```

Si el freeze desaparece: El bug está en `post_process_instance_segmentation()` de Transformers.

Experimento 4: Ejecutar sin FastAPI (script standalone)

Crea un `test_sam3.py`:

python

```python
from pathlib import Path
from PIL import Image
from sam3_runner import SAM3Runner

runner = SAM3Runner(Path("ruta/a/pesos"))
runner.load_model(safe_mode=True, device_preference="cpu")

img = Image.open("test_image.jpg")
detections = runner.run_pcs(img, "roof", target_long_side=384,
box_threshold=0.5, max_detections=5)
print(f"Detections: {len(detections)}")
```

Ejecuta:

bash

```
python test_sam3.py
```

Si NO se congela: El problema está en la interacción con FastAPI/threading.

Experimento 5: Usar versión estable de Transformers

bash

```
pip uninstall transformers
pip install transformers==4.44.0  # Última versión estable antes de 5.0
```

Si SAM-3 no está disponible: Usa otro modelo (DETR, YOLO) para verificar que el problema es específico de SAM-3.

Experimento 6: Agregar timeout a la inferencia

Envuelve la inferencia en un timeout:

python

```python
import signal

def timeout_handler(signum, frame):
    raise TimeoutError("Inference timeout")

signal.signal(signal.SIGALRM, timeout_handler)
signal.alarm(10)  # 10 segundos
try:
    with torch.inference_mode():
        outputs = self.model(**inputs)
finally:
    signal.alarm(0)  # Cancelar alarma
```

Nota: `signal.alarm()` **no funciona en Windows**. Alternativa con threading:

python

```python
import threading
```

```python
class InferenceWithTimeout:
    def __init__(self, timeout):
        self.timeout = timeout
        self.result = None
        self.exception = None

    def run(self, model, inputs):
        try:
            with torch.inference_mode():
                self.result = model(**inputs)
        except Exception as e:
            self.exception = e

    def execute(self, model, inputs):
        thread = threading.Thread(target=self.run, args=(model, inputs))
        thread.start()
        thread.join(timeout=self.timeout)
        if thread.is_alive():
            raise TimeoutError("Inference exceeded timeout")
        if self.exception:
            raise self.exception
        return self.result

# Uso:
runner = InferenceWithTimeout(timeout=10)
outputs = runner.execute(self.model, inputs)
```

⚠ ANTIPATRONES CONOCIDOS (PyTorch + Windows + SAM)

1. PIL + Multiprocessing en Windows

- PIL usa librerías C que no son thread-safe.
- Windows tiene un modelo de procesos diferente a Linux (CreateProcess vs fork).

2. CUDA + GTX 1050 Ti + Drivers antiguos

- La GTX 1050 Ti (arquitectura Pascal) puede tener drivers desactualizados que causan deadlocks con PyTorch 2.x.
- **Solución**: Actualiza drivers NVIDIA a la última versión (536+ para Win10).

3. Transformers + SAM-3 + Windows

- SAM-3 hace uso intensivo de operaciones de convolución que pueden saturar VRAM.
- El freeze puede ocurrir cuando el driver intenta paginar memoria VRAM al disco (swap).

4. FastAPI + Uvicorn + PyTorch

- Uvicorn usa `asyncio`, que no se lleva bien con operaciones sincrónicas largas (como inferencia).
- **Solución**: Usa un worker pool separado (Celery, RQ) para las inferencias.

🎯 DIAGNÓSTICO MÁS PROBABLE

Basado en el código y tu descripción, **mi hipótesis principal** es:

El freeze ocurre en `PIL.Image.resize()` debido a un deadlock en las bibliotecas nativas (libjpeg/libpng) cuando se ejecuta desde un daemon thread en Windows.

Segundo culpable más probable: **El método `post_process_instance_segmentation()` de Transformers dev tiene un bug de memoria/threading que causa un deadlock al procesar máscaras grandes.**

📋 PLAN DE ACCIÓN (orden recomendado)

1. ✅ **Ejecuta Experimento 1** (eliminar resize manual) - 2 minutos
2. ✅ **Ejecuta Experimento 4** (script standalone sin FastAPI) - 5 minutos
3. ✅ **Ejecuta Experimento 2** (forzar single-thread) - 1 minuto
4. ✅ **Actualiza drivers NVIDIA** - 10 minutos
5. ✅ **Ejecuta Experimento 5** (downgrade Transformers) - 5 minutos
6. ✅ **Ejecuta Experimento 3** (eliminar postprocesamiento) - 2 minutos

Si ninguno funciona, necesitaremos hacer **profiling a nivel de kernel** con Windows Performance Analyzer.