

# Programación Paralela y Concurrente

*Reporte de la práctica de paralelismo y concurrencia*

KRISTOFFER VAN STEEMBERGHE LUJÁN

2025

---

# ÍNDICE GENERAL

**1** | **Capítulo 1**  
Introducción.

**2** | **Capítulo 2**  
Diagramas del sistema.

**3** | **Capítulo 3**  
Justificación del uso de cada paradigma en cada parte.

**4** | **Capítulo 4**  
Explicación detallada del diseño.

**5** | **Capítulo 5**  
Fragmentos clave de código con explicación.

- 5.1 Diagnóstico asíncrono con IA. [10](#)
- 5.2 Asignación Secuencial de Recursos. [11](#)
- 5.3 Seguimiento y Alta en Paralelo. [11](#)

**6** | **Capítulo 6**  
Resultados de pruebas y rendimiento.

- 6.1 Pruebas para código sin modelo IA. [14](#)
- 6.2 Pruebas para código con modelo IA. [15](#)

**7**

**Capítulo 7**  
Conclusiones.

## CAPÍTULO

# 1

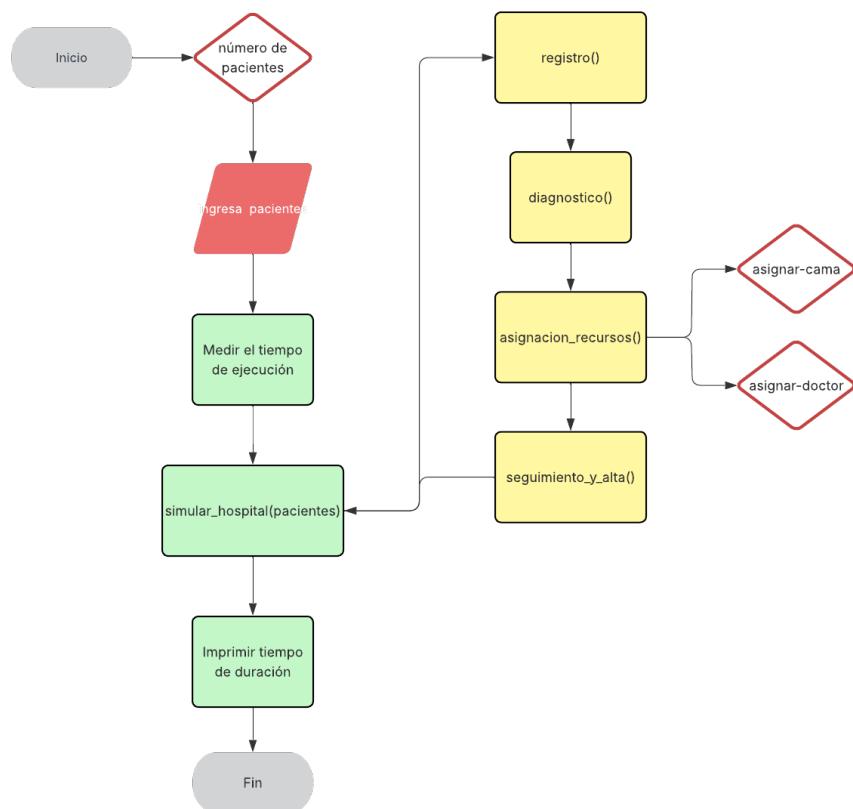
# INTRODUCCIÓN.

Este proyecto tiene como objetivo simular el flujo de pacientes en un entorno hospitalario, utilizando diferentes paradigmas de programación para optimizar el desempeño y la eficiencia. Se incorporan técnicas de programación concurrente, paralela y uso de un modelo pre-entrenado para diagnosticar imágenes médicas automáticamente, proporcionando un sistema robusto y eficiente.

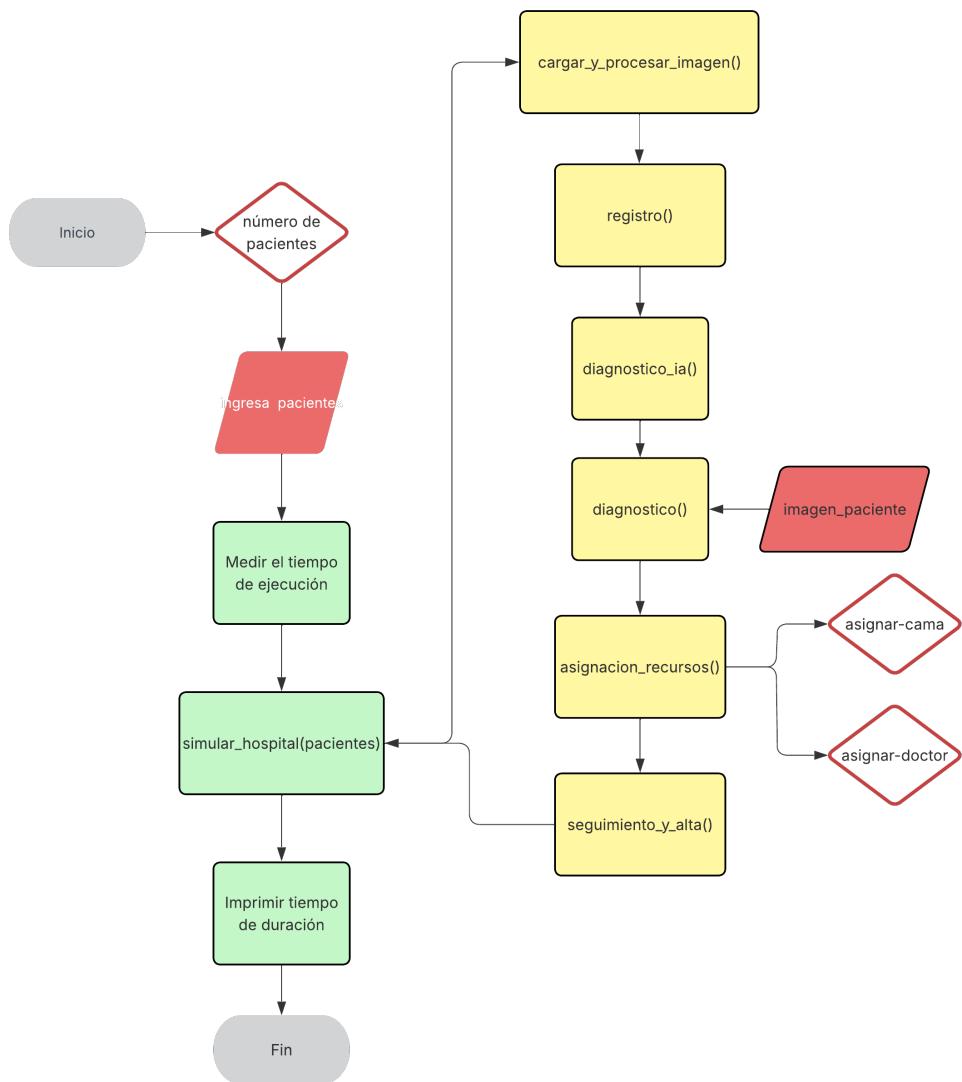
# DIAGRAMAS DEL SISTEMA.

A continuación se muestran los diagramas de flujo de cada código:

- Diagrama para `simulacion-hospitalaria.py`:



■ Diagrama para simulacion-hospital-ia.py:



# JUSTIFICACIÓN DEL USO DE CADA PARADIGMA EN CADA PARTE.

# 3

- **Programación Asíncrona (asyncio):** Se utiliza para manejar múltiples diagnósticos simultáneos, limitando el número concurrente mediante un semáforo. Esto permite controlar eficientemente el uso de recursos críticos como procesos de IA.
- **Multiprocessing:** La asignación de altas médicas se realiza mediante multiprocessing para optimizar el uso del procesador, permitiendo que esta etapa del proceso se ejecute paralelamente al flujo principal del programa.
- **Programación Secuencial:** El registro y asignación de recursos se ejecutan de manera secuencial para mantener el orden lógico de los procesos y asegurar que los recursos críticos (cama y doctor) sean asignados adecuadamente antes de cualquier procedimiento adicional.
- **IA con TensorFlow y VGG16:** La integración de IA se realiza mediante el uso del modelo preentrenado VGG16 para diagnosticar imágenes, aportando precisión y automatización en la etapa diagnóstica del proceso.

## CAPÍTULO

# EXPLICACIÓN DETALLADA DEL DISEÑO.

# 4

El diseño del proyecto se basa en la simulación realista de un hospital:

1. **Registro:** Realiza un registro inicial de cada paciente.
2. **Diagnóstico con IA:** Aplica procesamiento asíncrono con un modelo pre-entrenado (VGG16) para predecir diagnósticos basados en imágenes médicas.
3. **Asignación de Recursos:** Asigna primero una cama y posteriormente un doctor, de manera secuencial.
4. **Seguimiento y Alta:** Realizado en paralelo usando multiprocessing para acelerar la liberación de recursos y mejorar la eficiencia general.

La estructura se coordina mediante una función central que gestiona las etapas, manteniendo una ejecución ordenada y eficiente.

Se decidió usar este patrón debido a que era requisito para la entrega.

Para la implementación del modelo decidí preguntarle a *ChatGPT* de que manera podría implementarlo en la función del diagnóstico:

¿De qué manera se pueden implementar modelos de IA preentrenados en la función `diagnostico()`?

Para implementar modelos de IA preentrenados en la función `diagnostico()` dentro de tu simulación hospitalaria, puedes utilizar bibliotecas populares de Machine Learning y Deep Learning en Python, como TensorFlow o PyTorch. A continuación, te explico cómo puedes integrar un modelo preentrenado de IA en esa función.

**Pasos para implementar modelos de IA preentrenados:**

- 1. Seleccionar un modelo preentrenado:**
  - Puedes utilizar modelos preentrenados de bibliotecas como **TensorFlow Hub**, **Hugging Face Transformers** o **torchvision** (si trabajas con imágenes) para tareas como clasificación, detección de anomalías, etc.
  - Por ejemplo, para diagnóstico médico basado en imágenes, puedes utilizar un modelo de clasificación preentrenado para identificar condiciones específicas (como enfermedades en radiografías o imágenes médicas).
  - Si deseas trabajar con texto (como diagnóstico basado en el historial de un paciente), puedes utilizar un modelo preentrenado de NLP (Natural Language Processing), como **BERT** o **GPT**.
- 2. Integrar el modelo en la función `diagnostico()`:**
  - Lo que harás es cargar el modelo preentrenado, pasar los datos de entrada (como imágenes o texto del paciente) al modelo y obtener el diagnóstico.

Después decidí buscar videos en *YouTube* sobre modelos de TensorFlow y encontre este video [Clasificación de imágenes usando transferlearning y los modelos Resnet50, VGG19, MobileNet y VGG16](#), el cual tomé su ejemplo y traté de implementarlo en mi código, lo cual me ocasionó problemas y errores de sintaxis, pero con la ayuda de *Github Copilot* pude solucionarlos y adaptar el modelo correctamente a mis necesidades.

# FRAGMENTOS CLAVE DE CÓDIGO CON EXPLICACIÓN.

# 5

## 5.1

## Diagnóstico asíncrono con IA.

```
● ● ●
1 # Función de diagnóstico asíncrono
2 async def diagnostico(paciente_id, semáforo):
3     async with semáforo:
4         print(f"paciente {paciente_id}: Diagnóstico automatizado iniciado.")
5         # imagen_paciente = f"paciente_{paciente_id}_radiografía.jpg" # Si tuvieramos varias imágenes por paciente
6         imagen_paciente = f"image.jpg" # Ejemplo de imagen
7         predicciones = diagnostico_ia(imagen_paciente) # Llamar al modelo de IA
8
9         # Simulamos el tiempo de diagnóstico con IA
10        await asyncio.sleep(random.uniform(2, 5))
11
12        print(f"paciente {paciente_id}: Diagnóstico completado. Predicción: {predicciones[0][1]} con probabilidad de {predicciones[0][2]:.2f}")
```

Esta función utiliza un semáforo para limitar la concurrencia y evitar saturación del sistema. El modelo VGG16 proporciona un diagnóstico basado en la imagen suministrada.

Además podemos ver que hay una línea comentada:

```
# imagen_paciente = f"paciente_{paciente_id}_radiografía.jpg"
imagen_paciente = f"image.jpg" # Ejemplo de imagen
```

Podemos observar que existe una línea es si tenemos una imagen para cada paciente (paciente\_{paciente\_id}\_radiografía.jpg) y la otra línea es si queremos usar la misma imagen para todos los pacientes (image.jpg).

## 5.2

## Asignación Secuencial de Recursos.

```
● ● ●  
1 # Función de asignación de recursos para cumplir con el orden de asignar primero la cama y después el doctor  
2 def asignacion_recursos(paciente_id):  
3     # Primero asignar cama  
4     cama = f"Cama {random.randint(1, 10)}"  
5     print(f"paciente {paciente_id}: Cama asignada: {cama}.")  
6  
7     # Luego asignar doctor  
8     doctor = f"Doctor {random.randint(1, 5)}"  
9     print(f"paciente {paciente_id}: Doctor asignado: {doctor}.")  
10  
11    time.sleep(random.uniform(1, 2)) # Simulando asignación  
12    print(f"paciente {paciente_id}: Asignación de recursos completada.")
```

Esta función secuencial garantiza que los recursos críticos (cama y doctor) se asignen ordenadamente y sin conflictos.

## 5.3

## Seguimiento y Alta en Paralelo.

```
● ● ●  
1 # Función de seguimiento y alta para ser procesada en multiprocessing  
2 def seguimiento_y_alta(paciente_id):  
3     print(f"paciente {paciente_id}: Seguimiento y alta iniciado.")  
4     time.sleep(random.uniform(1, 3)) # Simulando alta  
5     print(f"paciente {paciente_id}: Alta completada.")
```

```
● ● ●  
1 # La etapa de seguimiento y alta se realiza en paralelo usando multiprocessing  
2 p = multiprocessing.Process(target=seguimiento_y_alta, args=(paciente_id,))  
3 p.start()  
4 p.join() # Espera a que el proceso termine antes de continuar
```

El uso de `multiprocessing` permite que el seguimiento y alta se manejen de manera simultánea con otros procesos, optimizando significativamente el desempeño del programa.

# RESULTADOS DE PRUEBAS Y RENDIMIENTO.

6

Para las pruebas utilizaremos solamente 3 terminales:

- Una para ver las estadísticas de rendimiento de la computadora con el comando `htop`.
- Otra para mostrar la información del hardware y sistema de la computadora con el comando `neofetch`.
- Finalmente, en la última ventana ejecutaremos nuestros programas con el entorno virtual activado. Cada simulación se realizará con 10 pacientes.

Las estadísticas de rendimiento sin ejecutar ningún código son las siguientes:

The screenshot shows a terminal window with several tabs open. The current tab displays system performance metrics. At the top, there's a decorative ASCII art banner followed by the command `neofetch`. The output of `neofetch` provides detailed information about the system, including the OS (Ubuntu 24.04.2 LTS x86\_64), kernel version (5.11.0-25-generic), uptime (4 mins), packages (2217), and various software environments like Wayland, DE (GNOME 46.0), WM (Mutter), and icons (Yaru-prussiangreen [GTK3/3]). Below this, the `htop` command is run, showing a real-time process viewer. The `htop` interface includes a CPU usage bar at the top, a list of processes in the center, and a footer with memory and swap usage, load average, and uptime.

ID	PRI	NICE	VIRT	RES	SHR	S	CPU%HWM	TIMER	Command	
3855	kris	20	0	4572M	288M	136M	5	25.6	2.5	0:19:44 /usr/bin/gnome-shell
3903	kris	20	0	29788	5852	3676	R	5.9	0.0	0:01:09 http
7232	kris	20	0	4572M	288M	136M	S	1.6	0.5	0:02:00 /usr/bin/gnome-shell
4761	kris	20	0	4572M	288M	136M	S	1.1	0.5	0:02:00 /usr/bin/warp-terminal
1065	systemd-oo	20	0	17556	7436	6668	S	0.7	0.1	0:00:37 /usr/lib/systemd/systemd-oomd
1222	messagebus	20	0	12380	6768	4328	S	0.7	0.1	0:01:51 /dbus-daemon -system --address=systemd: --nofork --pidfile=/run/dbus/systemd/dbus.pid --rootless --noredirect --accessx --config-file=/etc/dbus-1/system.conf
4834	kris	20	0	59776	331M	157M	S	0.7	2.8	0:00:11 /usr/bin/warp-terminal
4835	kris	20	0	59776	331M	157M	S	0.7	2.8	0:00:11 /usr/bin/warp-terminal
4840	kris	20	0	59776	331M	157M	S	0.7	2.8	0:00:15 /usr/bin/warp-terminal
5422	kris	20	0	59776	331M	157M	S	0.7	2.8	0:00:18 /usr/bin/warp-terminal
6926	kris	20	0	59776	331M	157M	S	0.7	2.8	0:00:02 /usr/bin/warp-terminal
1	root	20	0	23696	14308	9316	S	0.0	0.1	0:03:93 /sbin/init splash
478	root	10	-1	6756	1762	1064	S	0.0	0.0	0:00:00 /usr/lib/systemd/systemd-journalctl
1066	systemd-re	20	0	21580	1712	10664	S	0.0	0.1	0:00:00 /lib/systemd/systemd-reload
1066	systemd-re	20	0	21580	1712	10664	S	0.0	0.0	0:00:36 /lib/systemd/systemd-resolved

6.1

# Pruebas para código sin modelo IA.

El tiempo de duración fue de 59.34 segundos, además cada núcleo presenta una ligera carga de trabajo, el uso de memoria incrementa un poco pero no sobrepasa el 50 %.

6.2

# Pruebas para código con modelo IA.

El tiempo de duración fue de 44.98 segundos, un 24 % más rápido, además la carga en los núcleos es un poco mayor y es distribuida de forma más balanceada, además el uso de memoria incrementa en comparación con el código anterior.

En conclusión, el código que implementa el uso de un modelo pre-entrenado distribuye la carga de trabajo de mejor forma y usa más memoria con el propósito de tener un tiempo de ejecución de menor tiempo.

# CONCLUSIONES.

El proyecto demuestra efectivamente cómo diferentes paradigmas de programación pueden coexistir para crear una aplicación eficiente y realista. La combinación de programación asíncrona, multiprocessing y el uso avanzado de modelo de datos garantiza una operación fluida, rápida y precisa, apta para entornos de alta demanda como hospitales modernos.