



# Minishell en C++

**Integrantes:**  
**Alexis Condori Rivera**  
**Yeferson Guerra Quispe**



# Indice

- Introducción
- Arquitectura
- POSIX
- Concurrencia
- Gestión de memoria
- Flujo de ejecución
- Conclusión

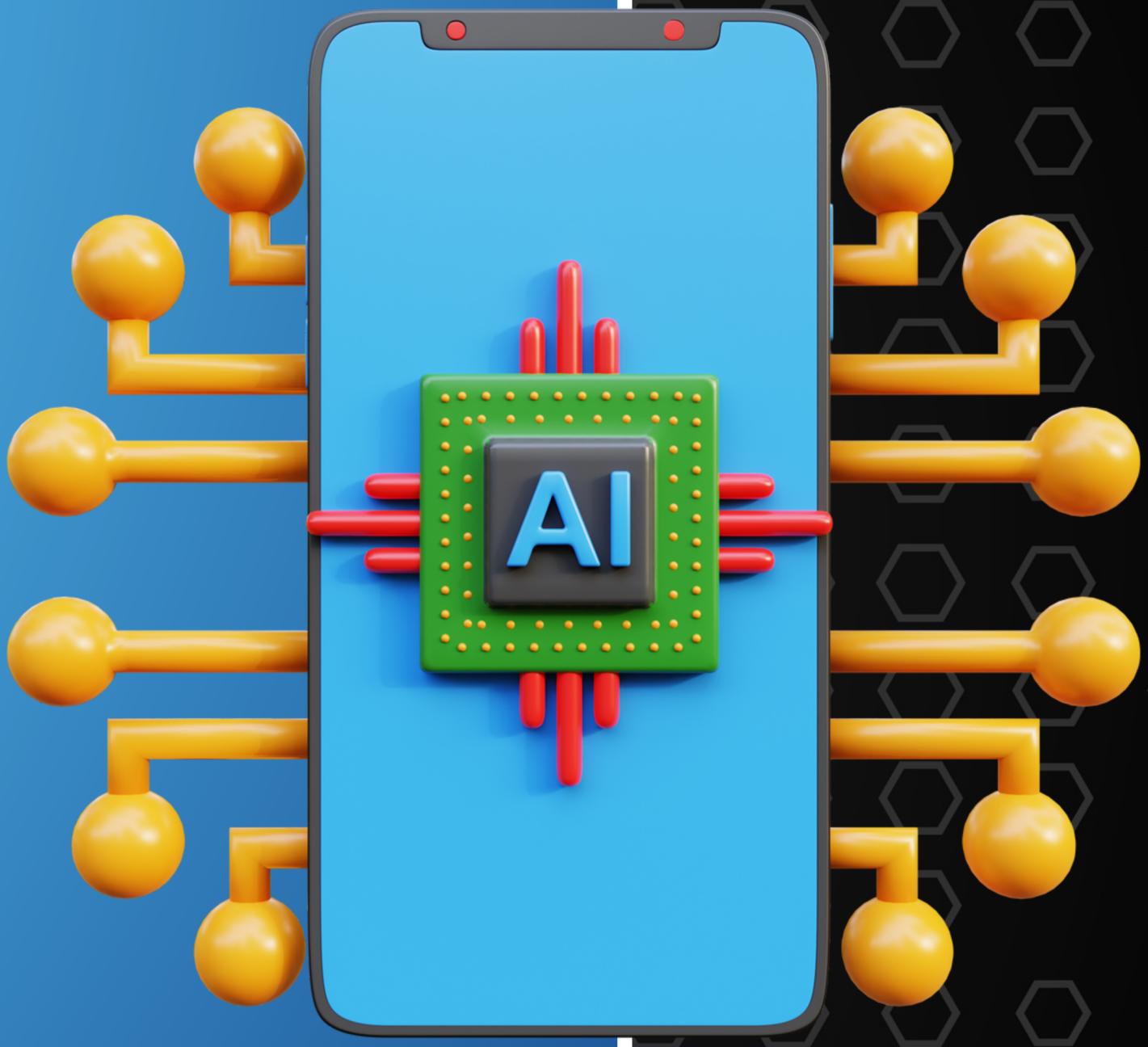


# Introducción

El proyecto Mini-Shell en C++ sobre Linux tiene como objetivo aplicar los conceptos vistos en la Unidad I.

Permite ejecutar comandos del sistema, realizar redirecciones y tuberías, y ejecutar tareas en paralelo mediante hilos POSIX.

- Se implementó utilizando llamadas al sistema POSIX como fork, execvp, pipe y pthread\_create, demostrando el funcionamiento interno de una shell real en un entorno Linux.



# Arquitectura

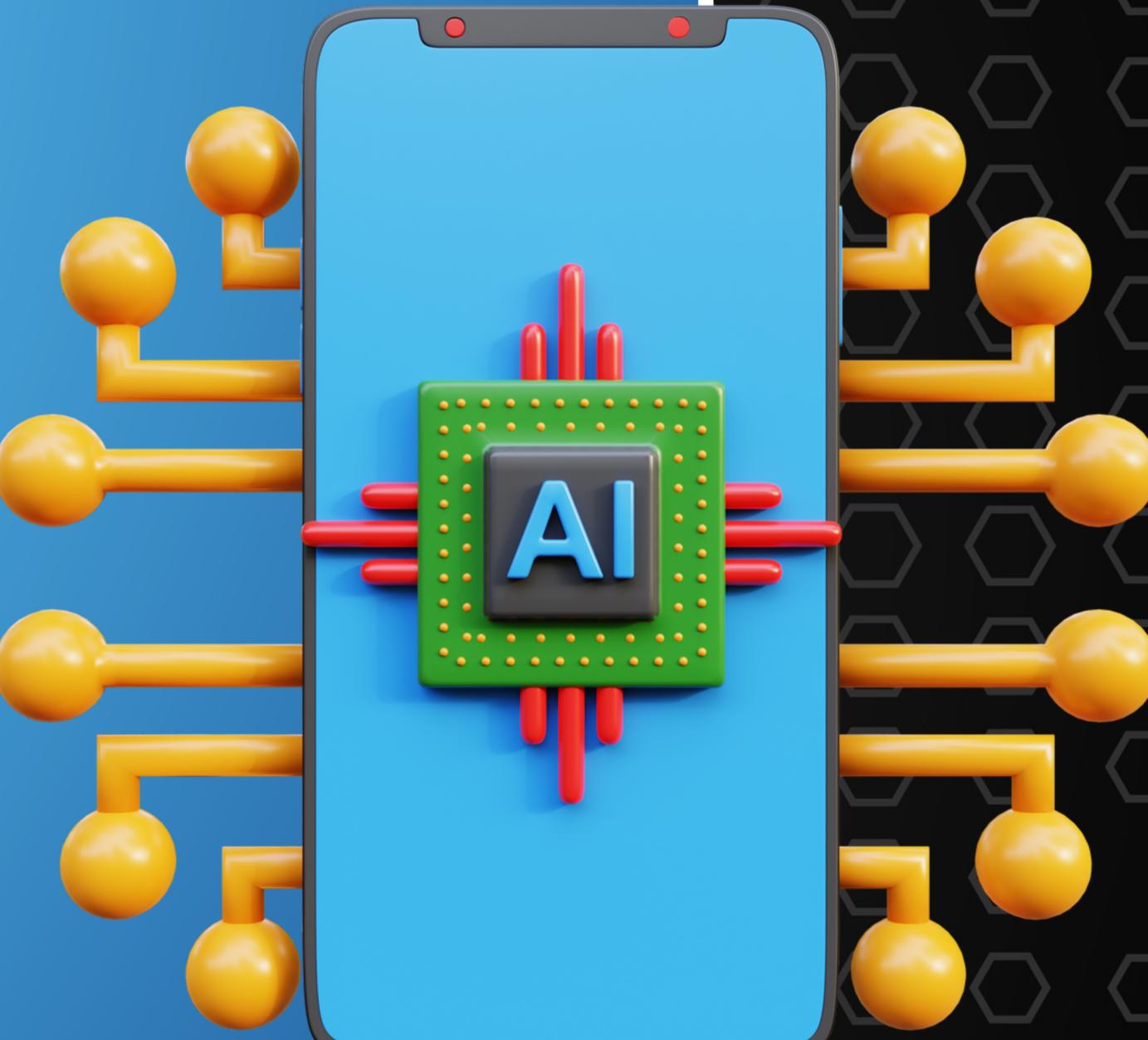
## 1. Shell

Es el núcleo del programa (Shell.cpp). Se encarga de:

- Mostrar el prompt
- Gestionar el historial, el alias y el bucle principal
- Llamar al Parser para que analice la entrada del usuario
- Decidir si el comando es interno (como cd o exit) o si debe pasárselo al Executor.

## 2. Parser

- Su única misión es "traducir". Toma la línea de texto del usuario y la convierte en una estructura de Command que el Executor pueda entender.
- Identifica si hay operaciones especiales como pipes, redirecciones o ejecuciones en segundo plano



# Arquitectura

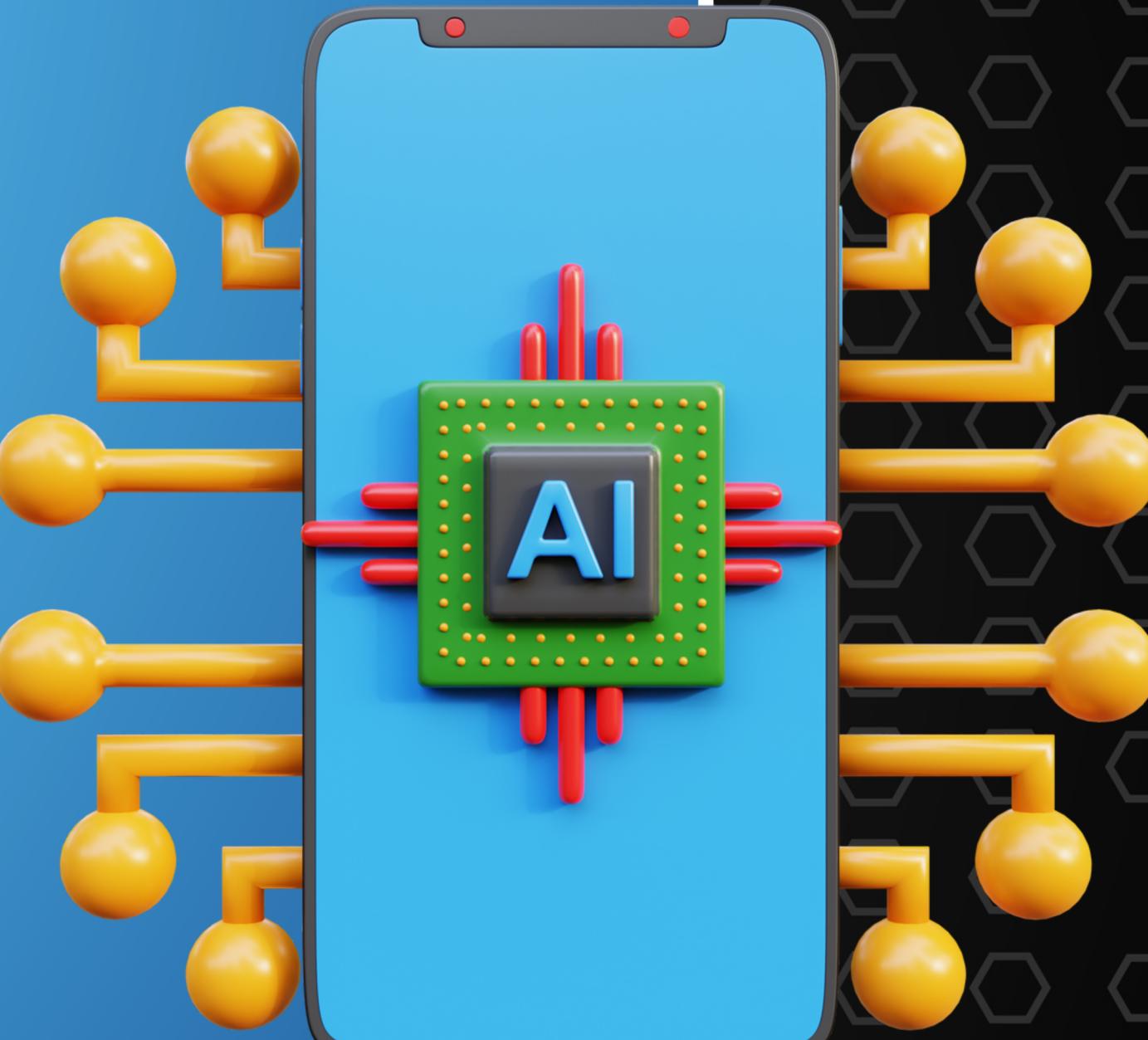
## 3. Executor

Es el "motor". Se encarga de la lógica pesada: crea procesos hijos con `fork()` y ejecuta los comandos con `execvp()`.

- Configura las pipes para conectar la salida de un comando con la entrada de otro.
- Maneja las redirecciones para que la entrada/salida vaya a archivos en lugar de la pantalla.

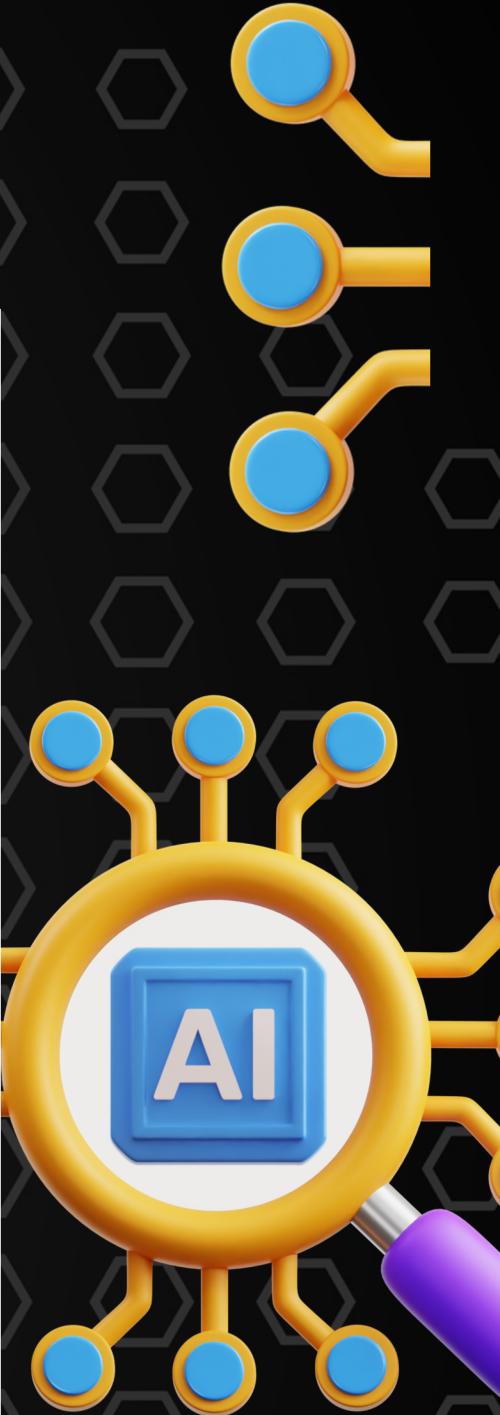
## 4. Built-in

Contiene los comandos que la shell ejecuta directamente sin crear nuevos procesos, como `cd`, `pwd` y `help`.

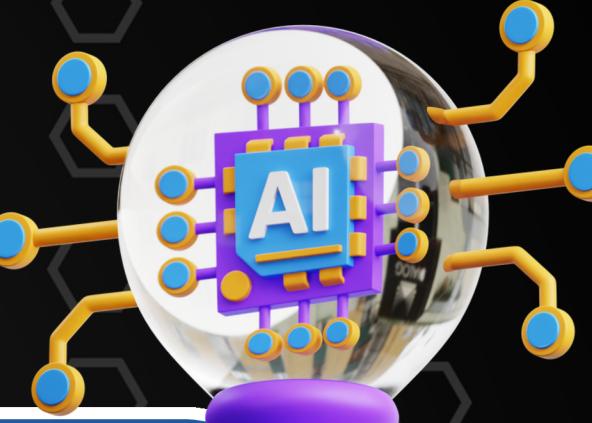


# POSIX Usadas

Categoría	Funciones	Propósito
Procesos	Fork ()	Crea un nuevo proceso hijo
	Execvp ()	Reemplaza el proceso hijo con un programa externo
	Wait () - Waitpid ()	Espera la finalización del proceso hijo
I/O	Open () - Close ()	Manejo directo de archivos
	Dup2 () - Pipe ()	Permiten comunicación entre procesos (pipes y redirección)
Concurrencia	Pthread_create Pthread_join	Creación y sincronización de hilos.
Sincronización	Sem_init Sem_wait Sem_post	Control de acceso concurrente con semáforos.
	Signal ()	Captura señales como SIGINT (Ctrl + C).



# Concurrencia y Sincronización



- Se implementó concurrencia usando hilos POSIX (pthread).
- El comando interno parallel permite ejecutar múltiples comandos simultáneamente.
- Cada hilo ejecuta un comando distinto en paralelo.
- Se emplearon mutex y semáforos para sincronizar tareas y evitar conflictos.
- El proceso principal espera con pthread\_join() hasta que todos los hilos terminen.

# Gestion de memoria

Manejo de Memoria Dinamica

Se usa una mezcla de gestión automática y manual:

- **std::vector** y **std::string** → gestionan memoria automáticamente.
- **malloc(), free(), new, delete** → para estructuras temporales

Ejemplo, **char\*\* args** de **execvp()**.

- Cada proceso libera la memoria antes de finalizar.
- Verificación con **Valgrind**:
- “No se detectaron fugas de memoria ni accesos inválidos.”

# Flujo de Ejecución de un Comando

```
minishell $ ls -la

**Flujo interno**:

1. readline() captura "ls -la"
2. Parser tokeniza: ["ls", "-la"]
3. Parser crea Command:
{
    args: ["ls", "-la"],
    inputFile: "",
    outputFile: "",
    hasPipe: false,
    background: false
}
4. Executor::executeSimple()
   └─ fork()
      └─ Hijo: execvp("/bin/ls", ["ls", "-la"])
         └─ Padre: waitpid()
5. ls ejecuta, muestra salida
6. Hijo termina con exit(0)
7. Padre continúa, muestra prompt
```

# Conclusion

El desarrollo del mini-shell permitió:

- Comprender en profundidad el funcionamiento de procesos e hilos en sistemas Linux.
- Implementar comunicación entre procesos mediante pipes y redirecciones.
- Garantizar una concurrencia segura mediante mutexes y semáforos.
- Aplicar buenas prácticas de gestión de memoria dinámica.
- Reforzar el entendimiento de llamadas POSIX y control de señales.

# Gracias

