Topología de IA: El Stack Híbrido en DevSecOps

Estantería: DevSecOps e Infraestructura

Subtema: Gobernanza

mercedev.es — 2026-05-12 | Epic 2 - Fase 4

El Desafío (Síntoma)

Integrar Inteligencia Artificial en un flujo de trabajo a menudo resulta en un acoplamiento frágil: o bien se depende 100% de APIs de terceros (exponiendo código privado y arriesgando bloqueos por cuotas), o se depende 100% de modelos locales (saturando la RAM del equipo y limitando la automatización en servidores CI/CD de la nube). El desafío era orquestar múltiples agentes de IA para que intervengan en el momento exacto sin sacrificar el rendimiento del proyecto ni la privacidad de la autora.

La Maniobra (Lógica)

Se ha diseñado una arquitectura de **Stack Híbrido (Hybrid Stack)**, utilizando el paradigma de *Shift-Left AI* (adelantar la IA a las fases de desarrollo y compilación). La IA nunca se ejecuta en tiempo real en el navegador del usuario (cero latencia), sino que trabaja entre bambalinas.

Esquema Gráfico de Intervención

```
[ ENTORNO LOCAL ] (Privacidad Total & Cero Coste)
   |
   ├─> 1. Creación & Formato (Zero-Hallucination)
   |    └─ merci-librarian.py ---> [ Ollama / qwen2.5-coder ]
   |
   ├─> 2. QA & Análisis Estático (Sugerencias en CLI)
   |    └─ merci-audit.py -------> [ Ollama / qwen2.5-coder ]
   |
   ├─> 3. Fase de Build (Compilación de UI Inteligente)
   |    └─ merci-brain.py -------> [ Ollama / qwen2.5-coder ]
   |
   └─> 4. Gobernanza Documental (SSOT)
        └─ merci-ssot.py --------> [ Cloud: Gemini 1.5 Flash ]
(Requiere gran ventana de contexto)
```

```
             └─ Fallback --------> [ Ollama / qwen2.5-coder ]
(Degradación elegante si no hay Red)

------------------------------------------------------------------------
---------------------------

[ ENTORNO CLOUD ] (Servidores Efímeros GitHub Actions)
   │
   └─> 5. Auto-Sanación en CI/CD (Continuous Integration)
        └─ merci-auto-fix.py ----> [ Cloud: Gemini 1.5 Flash ] (No es
posible usar Ollama en runners)
```

Anatomía de la Integración

1. **Desarrollo y Redacción (`merci-librarian.py` / `merci-audit.py`):** Utilizan el motor local `qwen2.5-coder`. Intervienen en el momento en que el código o el texto está en bruto. **Por qué:** Asegura que la propiedad intelectual y los errores de código locales nunca viajen a servidores externos.
2. **Tiempo de Compilación (`merci-brain.py`):** Se ejecuta justo antes de subir a producción. El motor local escanea la web y compila sus saludos en un archivo estático (`brain_data.json`). **Por qué:** Permite que la web final muestre "respuestas inteligentes" con 0 milisegundos de latencia, sin exponer API Keys en el frontend.
3. **Gobernanza (`merci-ssot.py`):** Orquestador híbrido que lee el historial de la bitácora y actualiza el Roadmap. Interviene antes del commit final. **Por qué:** Se usa *Gemini* en la nube porque entender registros largos requiere ventanas de contexto masivas que saturan la memoria local. Si Gemini falla, implementa un *Fallback* a Ollama.
4. **Integración Continua (`merci-auto-fix.py`):** Un agente que vive en GitHub Actions e interviene si subimos código roto. **Por qué:** Los contenedores de GitHub no tienen los recursos (RAM/GPU) para ejecutar Ollama. Aquí delegamos exclusivamente a Gemini Flash mediante LiteLLM para auto-reparar el repositorio.

El Aprendizaje / Deuda Técnica

El uso de una capa de abstracción como `LiteLLM` ha sido la clave del éxito. Nos ha permitido aplicar el principio de **Agnosticismo de Modelos**: Python no necesita saber si está hablando con un servidor en China o con el puerto `11434` de nuestra propia placa base.

Esta topología demuestra que delegar tareas específicas al motor adecuado (Local para privacidad/velocidad, Nube para gran contexto/Integración y Despliegue Continuo (CI-CD)) es el único camino viable para construir un ecosistema DevSecOps verdaderamente resiliente.