códigofacilito

Clase 1. Bootcamp AI Engineer

Bienvenida al Bootcamp

Instructor: Ramsés Alejandro Camas Nájera

6 de febrero, 2026

códigofacilito

Sobre mí

Ramsés Alejandro Camas Nájera

- **Head of AI @ MAPS Disruptivo | CEO @ RavanTech**
- MSc en Inteligencia Artificial — Universitat Politècnica de València
- Reserach: VISMAID (COMIA 2025, publicado por Springer)
- Especialidades: IA aplicada, Edge Computing, Multimodal AI, Automatización
- Director académico de este bootcamp

¿Qué esperar de este bootcamp?

16 clases

4 módulos, 1 proyecto transversal

🛠️ Práctico

Cada clase produce un artefacto de código reutilizable

🚀 Resultado

Un sistema de IA completo desplegado y funcional

⚡ Nivel

Intermedio-avanzado. Asumimos que ya sabes programar.

/01

El Panorama del AI Engineering

La revolución que estamos viviendo

2022 ChatGPT democratiza los LLMs

2023-24 Explosión de herramientas, frameworks y aplicaciones

2025-26 La industria pide ingenieros que construyan con IA, no solo que la usen

Estamos en el momento equivalente a los primeros años de la web: quien construye ahora, define el futuro.

¿Qué es un AI Engineer?

- ✕ **No es un Data Scientist** — no entrena modelos desde cero todos los días
- ✕ **No es un ML Engineer** — no optimiza pipelines de entrenamiento
- ✕ **No es un "prompt engineer"** — eso es una habilidad, no un rol

✓ **Un AI Engineer diseña, construye y despliega productos que integran modelos de IA en sistemas reales**

Piensa en: pipelines de RAG, agentes autónomos, copilotos empresariales

El espectro de roles en IA

ML Researcher	→ Investiga y publica nuevas arquitecturas
ML Engineer	→ Entrena, optimiza y sirve modelos
Data Scientist	→ Analiza datos y extrae insights
AI Engineer	→ Construye productos end-to-end con modelos existentes

El AI Engineer es el puente entre los modelos y los usuarios finales

Agenda de la clase

👋	5 min	Bienvenida y presentación
📚	50 min	Bloque 1: Panorama del AI Engineering + teoría
☕	10 min	Preguntas + Break
💻	50 min	Bloque 2: Setup práctico, cliente LLM, demo
❓	5 min	Preguntas finales

¿Cómo funcionan los LLMs? (Alto nivel)

- Un LLM es una red neuronal entrenada para predecir el siguiente token (palabra/subpalabra)
- Arquitectura base: Transformer (Vaswani et al., 2017 — "Attention is All You Need")
- Entrenados con cantidades masivas de texto (internet, libros, código)
- No "piensan" ni "entienden" — modelan distribuciones de probabilidad sobre secuencias de texto
- Pero el resultado es sorprendentemente útil para tareas del mundo real

¿Qué es un token?

- Los LLMs no procesan letras ni palabras: procesan tokens
- Un token ≈ 3-4 caracteres en inglés, puede variar en español

```
"Inteligencia artificial" → 3-5 tokens
```

¿Por qué importa?

- Pagas por tokens consumidos
- La ventana de contexto se mide en tokens
- Todo tiene un costo: input + output

💡 Optimizar tokens = optimizar costos y rendimiento

El paradigma de interacción con LLMs

No programamos los LLMs — les enviamos instrucciones en lenguaje natural

System message

Define el comportamiento del modelo

User message

La petición del usuario

Assistant message

La respuesta del modelo

Esta estructura de mensajes es la unidad fundamental con la que trabajaremos todo el bootcamp

Proveedores y modelos

OpenAI	GPT-4o, GPT-5-nano, o1, GPT-OSS
Anthropic	Claude Sonnet, Claude Opus, Claude Haiku
Google	Gemini 3 Flash, Gemini 3 Pro, Gemma
Otros	Llama, Mistral, Qwen, GLM, Nemotron, DeepSeek

Un buen AI Engineer sabe cuándo usar cuál y puede cambiar sin reescribir todo

La API como interfaz universal

Todos los proveedores convergen en un patrón similar:

1. Autenticarte con una API key
2. Enviar mensajes (system + user)
3. Recibir respuesta + metadata (tokens usados, latencia)

Esta uniformidad nos permite abstraer el proveedor detrás de un cliente genérico.
Eso es exactamente lo que construiremos hoy.

Más allá de "llamar una API"

Llamar un LLM es fácil. **Construir un sistema con LLMs es ingeniería.**

Los desafíos reales:

- ¿Cómo manejas errores y reintentos?
- ¿Cómo controlas costos cuando escalas?
- ¿Cómo sabes si la respuesta fue buena o mala?
- ¿Cómo depuras cuando algo falla en producción?
- ¿Cómo conectas el LLM con datos reales de tu empresa?

El stack del AI Engineer

Capa 6	Producción	Observabilidad, seguridad, despliegue
Capa 5	Evaluación y calidad	Métricas, testing, guardrails
Capa 4	Agentes	ReAct, multiagentes, herramientas
Capa 3	Datos y contexto	RAG, vector stores, memoria
Capa 2	Orquestación	LangChain, LangGraph, DSPy
Capa 1	Modelos	GPT, Claude, Llama, etc.

Este bootcamp recorre las 6 capas.

La diferencia entre un demo y un producto

🎮 Demo

Script de 20 líneas que llama OpenAI y muestra la respuesta

🏗️ Producto

- Maneja errores gracefully
- Registra cada interacción (logging)
- Controla costos y latencia
- Valida las respuestas del modelo
- Es reproducible y desplegable
- Tiene pruebas

Este bootcamp te lleva del demo al producto.

¿Qué vamos a construir en este bootcamp?

Proyecto transversal: **DocOps Agent**

Un copiloto empresarial que procesa documentos privados y asiste en tareas operativas

Módulo I

Setup, tokens, prompts, outputs estructurados

Módulo II

RAG serio y memoria externa

Módulo III

ReAct y multiagentes

Módulo IV

Evaluación, optimización, seguridad y despliegue

Recapitulación del Módulo I

Clase 01	Setup profesional y cliente LLM (hoy)
Clase 02	Tokens, contexto, costo y latencia
Clase 03	Prompt engineering para sistemas
Clase 04	Outputs estructurados y contratos

/02

Lo que construiremos hoy

Bloque Práctico

Objetivo de hoy

Configurar un entorno de desarrollo profesional y reproducible

Crear un cliente LLM reutilizable con logging y trazabilidad

Entregar un CLI funcional que invoque un LLM y registre cada interacción

Todo lo que construyamos hoy será el código base del resto del bootcamp

Arquitectura del proyecto

```
ai-engineer-bootcamp/
├── core/
│   ├── llm_client.py
│   ├── config.py
│   └── logger.py
├── .env
├── .env.example
├── requirements.txt
├── pyproject.toml
└── main.py
```

Estructura modular

- core/ contiene la lógica reutilizable
- Configuración centralizada en .env
- main.py como punto de entrada
- Cada módulo futuro (rag/, agents/, mcp/) se conectará con core/

"Si no es reproducible, no es ingeniería"

- En producción, el código de IA no vive solo — convive con APIs, bases de datos, servicios
- Un proyecto bien organizado desde el día 1 evita deuda técnica
- Si tu colega no puede hacer git clone + pip install + python main.py y que funcione, hay un problema

Entorno reproducible — Herramientas

`Python 3.10+`	Lenguaje central del bootcamp
`virtualenv / poetry`	Aislar dependencias por proyecto
`python-dotenv`	Manejar variables de entorno (.env)
`Git`	Versionamiento desde el minuto cero
`Docker`	Opcional: lo veremos en la clase 16

Configuración paso a paso

1. Crear repositorio y clonar
2. Crear entorno virtual (python -m venv .venv)
3. Crear .env con las API keys (nunca commitear al repo)
4. Crear .env.example como referencia (sin valores reales)
5. requirements.txt con las dependencias iniciales

El archivo .env

Almacena secretos y configuración sensible

```
OPENAI_API_KEY=sk-...
ANTHROPIC_API_KEY
=sk-ant-...
MODEL_NAME
=gpt-4o-mini
LOG_LEVEL
=INFO
```

⚠ Regla de oro

.env siempre en .gitignore. Siempre.

Diseñando el cliente LLM

Objetivo: una abstracción que permita cambiar de proveedor sin reescribir todo

Principios de diseño:

⚙️ Configuración centralizada

No hardcodear keys ni modelos

📝 Logging automático

De cada llamada al LLM

🛡️ Manejo de errores

Robusto y graceful

📊 Métricas básicas

Tokens usados, latencia, costo estimado

core/config.py

Carga las variables de entorno con dotenv

- Define valores por defecto para cada variable
- Centraliza toda la configuración en un solo lugar

Patrón de configuración:

```
Leer de .env  →  Fallback a valor por defecto  →  Fallar explícitamente si falta algo crítico
```

core/llm_client.py — Estructura

```
class LLMClient:

  __init__(proveedor, modelo, temperatura)

  chat(messages, **kwargs)
  # Método principal: envía mensajes,

  # retorna respuesta


  _log_usage()
  # Registra tokens, latencia, costo
```

Hoy implementamos con Gemini API (vía Google AI Studio).

En la clase 8 añadiremos adaptadores locales.

Código en vivo — llm_client.py

Elementos clave:

SDK oficial	`openai.OpenAI()` o `genai.Client()`
Medición de tiempo	`time.perf_counter()`
Tokens	`usage.prompt_tokens + usage.completion_tokens`
Retorno estructurado	`respuesta + metadata`

¿Por qué logging y no solo print()?

`print()`

- Desaparece al cerrar la terminal
- Sin niveles de severidad
- Sin timestamps
- Sin contexto estructurado

`logging`

- Los logs persisten
- Niveles: DEBUG, INFO, WARNING, ERROR
- Timestamps automáticos
- Contexto configurable

En IA, la trazabilidad es crítica: si un agente toma una mala decisión, necesitas la traza completa

core/logger.py

Configuración de logging con logging estándar de Python + rich para formato

Cada llamada al LLM registra:

- → **Timestamp** Momento exacto de la llamada
- → **Modelo usado** Qué modelo respondió
- → **Tokens entrada/salida** Consumo exacto
- → **Latencia (ms)** Tiempo de respuesta
- → **Costo estimado** Lo profundizaremos en clase 2

```
Niveles configurables vía .env (LOG_LEVEL)
```

Demo — "Hello AI System"

Ejecutamos main.py:

1. Carga configuración desde .env
2. Inicializa el cliente LLM
3. Envía un mensaje de prueba
4. Muestra respuesta + métricas en consola

Este es nuestro **"Hello World" de AI Engineering**

main.py — Referencia

Diapositiva de apoyo para la demo

1. Cargar config
2. Inicializar logger
3. Crear LLMClient
4. Definir mensajes (system + user)
5. client.chat(messages)
6. Imprimir respuesta y logs

/03

Cierre

Recapitulación y próximos pasos

Buenas prácticas desde el día 1

- 🔐 Nunca hardcodear API keys en el código
- 📊 Siempre medir tokens y latencia (te salvará dinero y tiempo)
- 📦 Modularizar: cada archivo hace UNA cosa bien
- 🔄 Versionar todo en Git (incluyendo los prompts, que son código)
- 📝 Documentar decisiones de diseño

¿Qué construimos hoy?

- ✓ Estructura de proyecto profesional
- ✓ Entorno reproducible con .env y virtualenv
- ✓ core/config.py — configuración centralizada
- ✓ core/llm_client.py — cliente LLM reutilizable
- ✓ core/logger.py — sistema de logging con trazabilidad
- ✓ main.py — CLI funcional que invoca un LLM

Próxima clase — Tokens, contexto, costo y latencia

- ¿Cómo funcionan los tokens realmente? (deep dive)
- ¿Cuánto cuesta cada llamada y cómo presupuestar por funcionalidad?
- Ventana de contexto: qué pasa cuando se llena y cómo manejarlo
- **Construiremos core/tokenlab.py**

Tarea (opcional)

→ Replicar el setup completo en tu máquina

→ Modificar main.py para que acepte input del usuario por terminal (mini-chat loop)

⭐ Bonus: agregar soporte para Groq como segundo proveedor en llm_client.py

→ Subir el repositorio a GitHub

No hay fecha límite ni revisión formal — comparte tu solución en el grupo de Telegram para retroalimentación entre compañeros

Recursos recomendados

What is an AI Engineer? — Swyx	https://www.latent.space/p/ai-engineer
OpenAI API Docs	https://platform.openai.com/docs
Google AI Studio (Gemini API)	https://aistudio.google.com/
Groq Console	https://console.groq.com/
Attention Is All You Need (paper original)	https://arxiv.org/abs/1706.03762

Bootcamp AI Engineer

Clase 1 completada ✓

Nos vemos en la siguiente clase