

# LLMs en Programación

## De la generación de código a la solución de problemas

Clase 2: Diseño de Prompts y  
Gestión de Contexto en LLMs

Christian Luna Escudero

Grupo de investigación KDIS  
Instituto de investigación DaSCI.  
Universidad de Córdoba

19 de septiembre de 2025

KDIS Lab



DaSCI

## ¿Qué aprenderéis hoy?

- El rol de los prompts en los LLMs
- Los principios básicos del diseño de prompts
- Diversas técnicas de prompting (básicas como avanzadas)
- Como alinear los prompts para mejorar / facilitar el desarrollo de software.

## Estructura de la Clase 2

- ① Fundamentos del Prompting
- ② Técnicas básicas (Zero, One, Few-shot)
- ③ Técnicas avanzadas (CoT, Roles, Prompt Chaining)
- ④ Context Engineering: cómo diseñar el input completo
- ⑤ Actividad práctica final y discusión

## ¿Qué es un Prompt?

- Entrada textual que guía al modelo.
  - Incluye instrucción, contexto y formato.
  - Afecta directamente a la calidad de la respuesta.



# Prompt Engineering: Definición e Importancia

## ¿Qué es?

Es la práctica de diseñar y optimizar *prompts* (instrucciones) que se le dan a un modelo de lenguaje para guiar su respuesta hacia un resultado deseado.

## Aplicaciones en el desarrollo de código

- Mejora la calidad y eficiencia del código generado automáticamente.
- Documentación automática (borradores).
- Generación de fragmentos de código repetitivo.

# Prompt Engineering: Aspectos Importantes

- Objetivo: Mejorar la precisión, relevancia y utilidad de las respuestas generadas.
- Importante:
  - La calidad de la respuesta depende en gran medida de la claridad y especificidad del *prompt*.
  - Un buen *prompt* puede transformar una respuesta genérica en una solución más específica.

## Técnicas de Prompting

- Uso de ejemplos: Incorporar ejemplos específicos en el prompt para guiar al modelo.
- Iteración y refinamiento: Ajustar el prompt con base en los resultados obtenidos.
- Preguntas abiertas vs cerradas: Decidir el tipo de pregunta según el objetivo de la respuesta.
- Especificidad del contexto: Proporcionar contexto detallado para mejorar la relevancia de la respuesta.
- Tono y estilo: Definir el tono y estilo deseado para las respuestas.

# Prompt vs Instrucción clásica

- **Instrucción tradicional:** se da a un programa rígido con sintaxis cerrada.
- **Prompt:** se da a un modelo probabilístico que interpreta lenguaje natural.

## Ejemplo

```
prompt: "Genera una función que calcule la media móvil de una lista de enteros"
```

→ *Hay muchas formas correctas de responder, según el estilo, lenguaje, nivel.*

# Elementos de un Prompt

- Instrucciones
- Contexto
- Datos de entrada
- Indicador de salida

Escribe una función en Python que calcule la suma de dos números

Entrada: dos números enteros, a y b

Salida: un número entero que representa la suma de a y b

# Partes de un Prompt

- 1 Human:
- 2 Contexto de la tarea.
- 3 Contexto del tono.
- 4 Datos de fondo a procesar.
- 5 Descripción detallada de la tarea y reglas.
- 6 Ejemplos.
- 7 Datos inmediatos a procesar.
- 8 Descripción o solicitud inmediata de la tarea.
- 9 Pensar paso a paso / tomar un respiro.
- 10 Formateo de salida.
- 11 Assistant:

## Human:

Eres un asistente de desarrollo en Python. Tu tarea es generar una función que automatice el cálculo del factorial de un número. La función debe manejar entradas inválidas de manera adecuada. La función debe estar optimizada para el rendimiento e incluir manejo de errores.

Aquí tienes algunos requisitos para la función:

- La función debe llamarse `calculate_factorial`.
- Debe aceptar un único número entero como entrada.
- Si la entrada es negativa o no es un número entero, debe devolver un mensaje de error.

Aquí tienes un ejemplo de cómo debería comportarse la función:

```
\example
calculate_factorial(5) debería devolver 120
calculate_factorial(-3) debería devolver "Error: La entrada
debe ser un número entero no negativo"
```

```
\example
Aquí tienes la entrada que necesitas procesar:
\input={{NUMBER}}\input
```

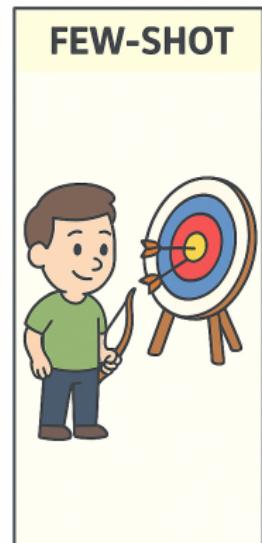
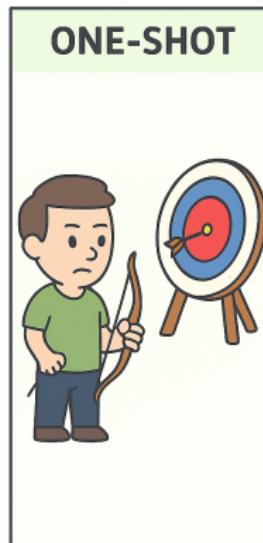
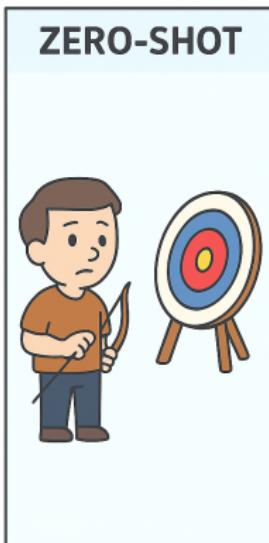
Genera el código Python para la función, considerando todos los requisitos y ejemplos proporcionados.

Piensa en tu respuesta antes de responder. Proporciona tu código dentro de las etiquetas `\code{ }\code`.

Asistente: `\code{ }\code`

## Tipos Básicos de Prompting

- **Zero-shot:** sin ejemplos, sólo una instrucción directa.
- **One-shot:** se da un ejemplo de entrada/salida.
- **Few-shot:** se dan múltiples ejemplos para guiar el comportamiento.



## Ejemplo Zero-shot

**Prompt:** “Explica el código siguiente en lenguaje natural:”

```
def diferencias_consecutivas(lista):
    return list(x - y for x, y in zip(lista[1:], lista[:-1]))
```

**Esperado:** El modelo responde sin ningún ejemplo previo.



ChatGPT 4o ▾

Compartir ...



Claro. Este código define una función llamada `diferencias_consecutivas` que toma como entrada una lista de números y devuelve una nueva lista con las diferencias entre elementos consecutivos. Aquí tienes la explicación paso a paso:

### 1. Entrada

La función recibe una lista llamada `lista`, por ejemplo:

python

Copiar Editar

lista = [10, 7, 5, 8]

### 2. Desplazamientos de la lista

Se crean dos listas internas usando *slicing*:

- `lista[1:]`: todos los elementos excepto el primero → [7, 5, 8]
- `lista[:-1]`: todos los elementos excepto el último → [10, 7, 5]

### 3. Emparejamiento con `zip`

La función `zip` empareja los elementos de ambas listas por posición:

python



Copiar Editar

Pregunta lo que quieras.

## Ejemplo Zero-shot (aplicado a backend)

### Prompt:

*“Escribe una función en Python que convierta una cadena de fecha (YYYY-MM-DD) en un objeto datetime.”*

**Comentario:** Sin ejemplo previo, el modelo asume la librería y formato más comunes.

## Ejemplo One-shot (aplicado a frontend)

### Prompt:

“Convierte este componente React a HTML plano.

`<Button>Enviar</Button>` → `<button>Enviar</button>`

`<Input type="text"/>` → ...?”

**Comentario:** El modelo imita el patrón ofrecido.

## Ejemplo Few-shot

### Prompt:

*Convierte código Python a pseudocódigo:*

*Python: for i in range(5): print(i)*

*Pseudocódigo: para i desde 0 hasta 4 hacer  
imprimir(i)*

*Python: while x < 10: x += 1*

*Pseudocódigo: ...?*

## Ejemplo Few-shot (ciencia de datos)

### Prompt:

*Python: df['A'].mean() → Calcula la media de la columna A*

*Python: df['B'].sum() → ...?*

**Comentario:** Con ejemplos previos, el modelo generaliza la estructura.

## Miniactividad en directo

**Probad estos prompts en ChatGPT/Copilot:**

- Zero-shot: “Escribe una clase Python para representar un rectángulo.”
- One-shot: añade un ejemplo previo de uso.
- Few-shot: añade dos ejemplos de clase + uso.

**Preguntas para discutir:**

- ¿Cambió la calidad o el formato de salida?
- ¿Cuál fue más fácil de controlar?

## Miniactividad en directo

**Probad estos prompts en ChatGPT/Copilot:**

- Zero-shot: “Escribe una clase Python para representar un rectángulo.”
- One-shot: añade un ejemplo previo de uso.
- Few-shot: añade dos ejemplos de clase + uso.

**Preguntas para discutir:**

- ¿Cambió la calidad o el formato de salida?
- ¿Cuál fue más fácil de controlar?

# Comparativa Visual de Estrategias

| Técnica          | Entrada                                | Salida esperada                | Aplicación típica                 |
|------------------|--|--------------------------------|-----------------------------------|
| Zero-shot        | "Escribe función que ordene una lista" | Código directamente            | Tareas simples, genéricas         |
| One-shot         | +1 ejemplo de entrada/salida           | Respuesta siguiendo el patrón  | Traducir estilo o formato         |
| Few-shot         | +2 o más ejemplos                      | Respuesta precisa y formateada | Formato complejo, estilo definido |
| Chain-of-Thought | "Resuelve paso a paso..."              | Ánalisis + Respuesta           | Lógica, matemáticas, depuración   |

*Ejemplificar es enseñar: los LLMs aprenden patrones implícitos.*

# Reescribir un mal prompt

## Prompt inicial:

*“Haz que este código sea mejor”*

## Versión mejorada:

*“Refactoriza este código Python para que sea más legible, manteniendo la misma funcionalidad.”*

```
def f(x): return x * 2
```

## Mejor aún:

*“Actúa como un revisor de código senior. Sugiere una versión más clara y pythonic de esta función.”*

## Buenas Prácticas Básicas

- Sé claro y específico: evita ambigüedades.
- Usa contexto relevante si es necesario.
- Define formato de salida si importa: lista, tabla, JSON...
- Itera y ajusta: el primer prompt rara vez es el mejor.

## Ejemplo Comparativo: Prompt Ambiguo vs Claro

### Prompt Ambiguo:

*Explícame esto:*

```
def f(x): return x * 2
```

### Prompt Claro:

*Explica en español qué hace la siguiente función de Python, línea por línea.*

## Ejemplo Comparativo: Salida Estructurada

### Prompt 1 (Sin formato):

*Resume este artículo.*

### Prompt 2 (Con formato definido):

*Resume este artículo en tres viñetas clave, usando lenguaje claro y técnico.*

## Errores frecuentes en prompts básicos

- ⊗ Instrucciones vagas: “Hazlo bien”, “Optimiza esto”
- ⊗ Sin objetivo: no especifica qué cambiar o mejorar
- ⊗ Múltiples tareas sin estructura: “Refactoriza, comenta, y añade tests”
- ⊗ Falta de contexto: se asume que el modelo “adivina” el problema

## Discusión guiada para el aula

### ¿Qué habéis observado?

- ¿Qué estilo de prompting os resultó más efectivo?
- ¿Tuvisteis outputs inconsistentes con zero-shot?
- ¿Cómo cambia la salida al añadir ejemplos?
- ¿Qué errores creéis que cometéis más al escribir prompts?

# Ejercicio Guiado (Parte 1)

**Objetivo:** Redactar un prompt one-shot para una tarea sencilla de programación.

## Consigna

Crea un prompt claro que pida a un LLM generar una función en Python que ordene una lista de diccionarios por un campo llamado 'edad'.

## Pistas:

- Usa lenguaje natural claro.
- Define el formato de entrada y salida esperada.
- Puedes incluir un ejemplo mínimo.

## Checklist para un buen prompt básico:

- ¿He definido claramente la tarea?
- ¿He incluido suficiente contexto?
- ¿He especificado el formato de salida?
- ¿El modelo sabrá qué hacer con este prompt sin ambigüedad?
- ¿Puedo probarlo y ajustarlo si no funciona?

*"Un buen prompt no siempre es largo, debe ser siempre preciso."*

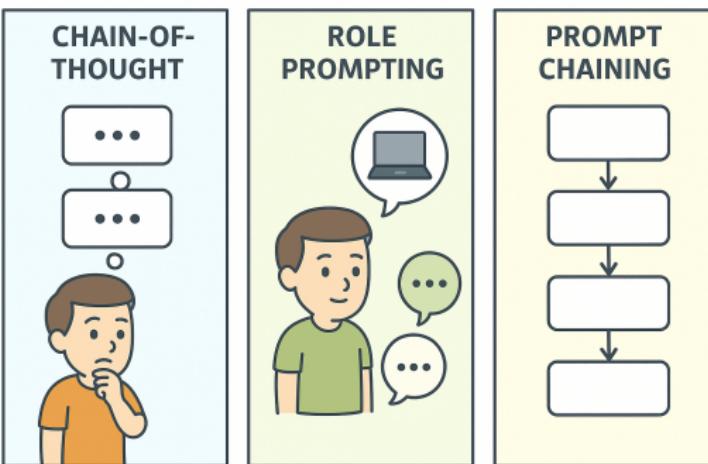
## Resumen: Fundamentos básicos del Prompting

- Conozco los estilos básicos: ZS, OS, FS
- He visto cómo afectan la salida de un LLM
- He aprendido a detectar y corregir malos prompts
- He probado escribir y ajustar mis propios prompts

*"El prompt no es solo una pregunta, es una forma de programar con lenguaje."*

## Técnicas Avanzadas de Prompting

- **Chain-of-Thought**: permite guiar al modelo para que razoné paso a paso antes de responder.
- **Role Prompting**: establece un rol explícito para el modelo (ej. “actúa como revisor de código”),
- **Prompt Chaining**: consiste en encadenar múltiples prompts, donde la salida de uno sirve de entrada para el siguiente.



# Chain-of-Thought Prompting

- Técnica para inducir razonamiento paso a paso.
- Muy útil en tareas complejas: lógica, cálculo, análisis de código.
- Puede activarse con frases como:
  - "Piensa paso a paso"
  - "Explica antes de responder"

## Ejemplo sin CoT

### Prompt:

*“¿Cuántos números pares hay entre 1 y 10?”*

**Respuesta típica:** 5

**Problema:** No hay trazabilidad ni explicación.

## Ejemplo con CoT

### Prompt:

*“¿Cuántos números pares hay entre 1 y 10? Piensa paso a paso.”*

### Respuesta esperada:

“Los números entre 1 y 10 son 1, 2, 3, ..., 10.

Los pares son: 2, 4, 6, 8, 10.

Hay 5 números pares.”

*CoT produce explicabilidad y mayor control.*

# CoT aplicado a programación

## Prompt:

*“¿Qué hace esta función? Piensa paso a paso.”*

```
def f(x): return x[::-1]
```

## Respuesta con CoT:

- La función recibe una cadena o lista.
- Usa el operador de slicing inverso.
- Devuelve la entrada invertida.

## Mini-ejercicio con CoT

### Consigna:

- Pide al modelo que diagnostique un error paso a paso: `def dividir(a, b): return a / b`
- Luego hazlo sin CoT y compara.

Discusión: ¿Cómo cambió la profundidad y claridad?

## Mini-ejercicio con CoT

### Consigna:

- Pide al modelo que diagnostique un error paso a paso: def dividir(a, b): return a / b
- Luego hazlo sin CoT y compara.

Discusión: ¿Cómo cambió la profundidad y claridad?

## Discusión: ¿Qué diferencia los resultados?

- ¿Cuál fue más útil y por qué?
- ¿La redacción influyó en la explicación?
- ¿Incluiste pasos intermedios o razonamiento?

# Role Prompting

- Técnica que asigna al modelo un “rol” explícito.
- Ayuda a:
  - Modular el tono, nivel técnico o enfoque
  - Forzar estilo experto, pedagógico, conciso, etc.
- Clave en tareas técnicas: revisión, explicación, refactorización.

## Ejemplo sin rol definido

### Prompt:

“*¿Este código está bien?*”

```
def suma(a, b): return a + b
```

**Problema:** Respuesta vaga, sin contexto o juicio técnico.

## Ejemplo con rol claro

### Prompt:

*"Actúa como revisor de código Python senior. Evalúa si esta función es correcta y sugiere mejoras."*

```
def suma(a, b): return a + b
```

### Respuesta esperada:

- “La función es correcta para enteros. Se puede mejorar añadiendo tipado y manejo de errores.”

## Roles sugeridos según perfil

- **Frontend:** diseñador UI, experto en accesibilidad, revisor React
- **Backend:** experto en APIs REST, arquitecto de microservicios
- **ML/Data:** científico de datos, ingeniero MLOps
- **Formativo:** profesor, mentor técnico, tutor junior

*Un rol bien elegido alinea la respuesta con nuestras expectativas.*

## Mini-ejercicio: cambia el rol

### Prompt base:

*“Explica esta función.”*

```
def normaliza(x): return (x - min(x)) / (max(x) -  
min(x))
```

### Variantes:

- Como un tutor que enseña a un estudiante.
- Como un revisor que busca fallos o mejoras.

Discusión: ¿Qué cambió en el estilo de respuesta?

## Mini-ejercicio: cambia el rol

### Prompt base:

*“Explica esta función.”*

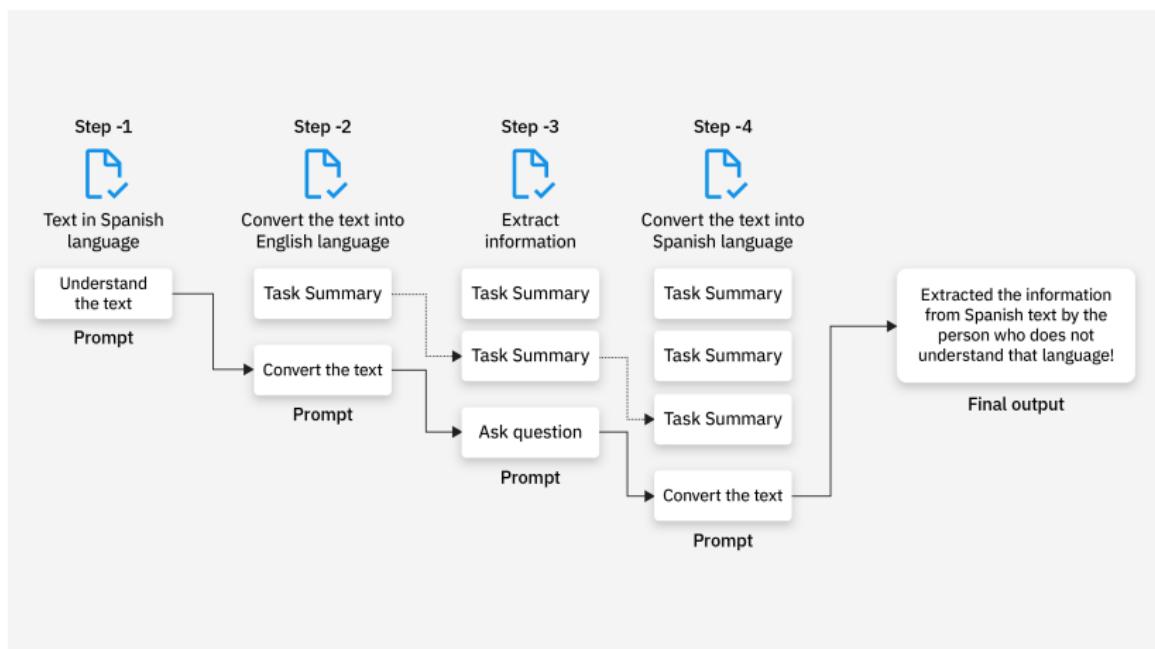
```
def normaliza(x): return (x - min(x)) / (max(x) - min(x))
```

### Variantes:

- Como un tutor que enseña a un estudiante.
- Como un revisor que busca fallos o mejoras.

**Discusión:** ¿Qué cambió en el estilo de respuesta?

# ¿No confundir el Chain-of-Thought con prompt chaining?



## Prompt Chaining

- Técnica para dividir tareas complejas en pasos conectados.
- Cada prompt usa la salida del anterior como entrada.
- Útil en flujos: análisis → generación → revisión → test.
- Se puede hacer manualmente o con herramientas (API, scripts).

## Ejemplo de Encadenamiento Manual

**Objetivo:** Crear una API para convertir grados Celsius a Fahrenheit.

- 1 Paso 1 (análisis):** “¿Qué endpoints necesito?”
- 2 Paso 2 (generación):** “Genera código en Flask para esos endpoints.”
- 3 Paso 3 (test):** “Escribe pruebas unitarias para el código anterior.”
- 4 Paso 4 (documentación):** “Escribe un README básico para esa API.”

## Prompt encadenado aplicado a refactor

### Paso 1:

*“Refactoriza esta función.”*

```
def calc(x): return x + x * 0.21
```

### Paso 2:

*“Añade un docstring a la versión refactorizada.” (con respuesta anterior)*

### Paso 3:

*“Genera una prueba unitaria.” (basado en la función mejorada)*

## Errores comunes al encadenar prompts

- ✗ No mantener coherencia entre etapas.
- ✗ Repetir el contexto completo en cada paso (mejor resumir).
- ✗ Olvidar que la salida anterior puede tener errores → se propagan.
- ✗ No validar interactivamente cada paso.

**Consejo:** Supervisar cada salto como si fuera una función dependiente.

## Del Prompt al Contexto

- ¿Dónde empieza el prompt y termina el contexto?
- ¿Qué ve realmente el modelo antes de responder?
- ¿Cómo aprovechar la ventana de contexto?

*Vamos más allá del diseño de un buen prompt...*

## ¿Qué es el Contexto para un LLM?

- El contexto es todo lo que el modelo ve en el "input": instrucciones, datos, ejemplos, historial.
- En modelos chat: incluye mensajes anteriores (prompt window).
- En API: tú defines el contexto cada vez (ej: prompt largo con instrucciones).
- La longitud y calidad del contexto afecta directamente el resultado.

*"Un modelo sin contexto es como un asistente con amnesia."*

# Tipos de Contexto en Interacción con LLMs

- **Explícito:** Instrucciones, ejemplos, datos en el prompt.
- **Implícito:** Historial de conversación (en chat), instrucciones de sistema.
- **Persistente:** Guardado en memoria (ej. agentes, RAGs con historial).
- **Temporal:** Sólo vive en la sesión actual.

## ¿Para qué sirve?

- Ajustar tono, precisión, formato.
- Aumentar coherencia entre turnos.
- Evitar repetir datos en cada mensaje.

# Tipos de Contexto en Interacción con LLMs

- **Explícito:** Instrucciones, ejemplos, datos en el prompt.
- **Implícito:** Historial de conversación (en chat), instrucciones de sistema.
- **Persistente:** Guardado en memoria (ej. agentes, RAGs con historial).
- **Temporal:** Sólo vive en la sesión actual.

## ¿Para qué sirve?

- Ajustar tono, precisión, formato.
- Aumentar coherencia entre turnos.
- Evitar repetir datos en cada mensaje.

# Prompt vs Context vs Chaining vs RAG

| Técnica             | Descripción   |
|---------------------|---|
| Prompt Engineering  | Instrucciones explícitas para una sola tarea.                   |
| Context Engineering | Diseño completo del input incluyendo historial, ejemplos, tono. |
| Prompt Chaining     | Secuencia de prompts interdependientes.                         |
| RAG                 | Inserta contexto externo desde una base documental.             |

→ *Diseñar el contexto es clave para controlar la conversación.*

## Buenas Prácticas en el Diseño del Contexto

- Instrucciones de sistema claras (ej: “eres un tutor experto en Python”).
- Usar ejemplos sólo si aportan valor (evitar redundancia).
- Controlar el tamaño: prompts demasiado largos → truncamiento.
- Encadenar contexto solo cuando se necesita continuidad.
- En RAG o API: separar contexto estático (instrucción) del dinámico (consulta).

*“El contexto es memoria de corto plazo. Si no lo defines tú, lo inventa el modelo.”*

# ¿Dónde termina el prompt y empieza el contexto?

- **Prompt Engineering:** defines instrucciones explícitas.
- **Context Engineering:** diseñas el input completo para cada turno o turno acumulado.
- **Prompt Chaining:** conectas múltiples etapas con outputs previos.
- **RAG:** el contexto viene de documentos externos (automático).

*“El prompt es una orden. El contexto es su mundo.”*

# Ventanas de contexto en modelos modernos (2025)

- Modelos actuales soportan contextos muy largos:
  - **GPT-5\***: desde **32k tokens** hasta **128k tokens**.
  - **Claude 3.5**: hasta **200k tokens**.
  - **Gemini 2.5**: **1M tokens** (modo experimental).
- **Reto actual**: no es solo cuánta información cabe, sino **cómo organizarla** para que el modelo priorice lo relevante.
- Fenómeno **Recency Bias**: los modelos tienden a dar más peso a lo último que leen.
- Estrategias para mejorar:
  - Repetir instrucciones clave al final del contexto.
  - Usar encabezados claros y estructura jerárquica.

# Cómo estructurar el contexto para maximizar relevancia

- **Chunking semántico:**
  - Dividir documentos largos en fragmentos temáticos coherentes.
  - Mantener cada bloque autocontenido.
- **Prioridad de contexto:**
  - Ordenar de más a menos relevante.
  - Instrucciones clave al principio y al final.
- **Metadatos:**
  - Etiquetas, fechas, fuentes → ayudan al modelo a interpretar.
- **Formato consistente:**
  - Listas, tablas, JSON → facilitan el parsing interno.

# Inyección de contexto dinámico con RAG

## RAG (Retrieval Augmented Generation)

- ① Buscar fragmentos relevantes en una base documental.
- ② Inyectarlos como contexto en el prompt antes de la consulta.

### Ejemplo:

**Instrucción:** “Eres un asistente Python experto.”

**Contexto recuperado:**

[Doc 1]: “Función para imputar valores nulos usando media.”

[Doc 2]: “Código para detección de outliers en Pandas.”

**Consulta del usuario:**

“Genera código para imputar nulos en columna edad.”

### Ventajas:

- Información precisa y actualizada.
- Menos dependencia del conocimiento preentrenado.

### Riesgo:

- Inyectar demasiado → ruido y truncamiento.

## Actividad: construir un flujo de prompting

**Reto:** Diseñar un flujo de prompts para una tarea real:

*“Dado un DataFrame de Pandas, crea un flujo que:*

- (1) detecte valores faltantes,*
- (2) proponga cómo imputarlos,*
- (3) genere código para hacerlo.”*

Opciones:

- Manual en ChatGPT
- Simulado por pares (uno genera, otro continúa)

## Actividad: construir un flujo de prompting

**Reto:** Diseñar un flujo de prompts para una tarea real:

*“Dado un DataFrame de Pandas, crea un flujo que:*

- (1) detecte valores faltantes,*
- (2) proponga cómo imputarlos,*
- (3) genere código para hacerlo.”*

**Opciones:**

- Manual en ChatGPT
- Simulado por pares (uno genera, otro continúa)

## Resumen: Técnicas Avanzadas

### ¿He usado técnicas avanzadas de forma efectiva?

- ¿He usado **Chain-of-Thought** para forzar razonamiento paso a paso?
- ¿He definido un **rol claro** para el modelo (ej. experto, revisor, tutor)?
- ¿He dividido tareas complejas con **prompt chaining**?
- ¿Soy consciente de las **limitaciones del modelo** (alucinaciones, pérdida de contexto...)?
- ¿He iterado y refinado mis prompts según la salida?

*“Las técnicas avanzadas no garantizan calidad, pero sí aumentan el control.”*

# Resumen: Técnicas de Prompt Engineering

## Estilos de prompting:

- **Zero-shot:** sin ejemplos, directo.
- **One-shot:** con un ejemplo guía.
- **Few-shot:** con varios ejemplos, más preciso.

## Técnicas avanzadas:

- **Chain-of-Thought:** razonamiento paso a paso.
- **Role Prompting:** define la voz del modelo.
- **Prompt Chaining:** divide tareas largas en pasos conectados.

## Discusión abierta

### Preguntas para el grupo:

- ¿Qué técnica os pareció más potente?
- ¿Habéis usado prompts con rol o CoT en vuestro trabajo?
- ¿Qué errores cometéis más al redactar prompts?
- ¿En qué tareas reales usaríais prompting encadenado?

# ¡Hora de practicar!

**Objetivo:** Aplicar lo aprendido con ChatGPT y/o Copilot

## Propuesta libre:

Elige una de estas tareas y experimenta con al menos 2 estilos de prompting:

- Refactoriza una función con y sin rol.
- Pide código y tests usando encadenamiento manual.
- Usa zero-shot y luego few-shot para la misma tarea.

**Tip:** Guarda prompts efectivos para construir tu “caja de herramientas”.