Proyecto: Black Stories - IA vs IA

🎯 Objetivo

Crear una aplicación de terminal donde dos modelos de IA juegan a **Black Stories**: el Modelo 1 inventa una historia misteriosa y el Modelo 2 intenta resolverla haciendo preguntas que solo pueden responderse con "SÍ", "NO" o "NO ES RELEVANTE".

🎮 ¿Qué son las Black Stories?

Las Black Stories son acertijos narrativos donde:

1. Se presenta una **situación final** (generalmente macabra, extraña o sorprendente)
2. El jugador debe descubrir **cómo se llegó a esa situación** haciendo preguntas
3. Solo se puede responder: **SÍ**, **NO** o **NO ES RELEVANTE**
4. El jugador gana cuando reconstruye correctamente la historia completa

Ejemplo:

- **Situación:** "Un hombre yace muerto en un campo con un paquete sin abrir a su lado"
- **Solución:** Su paracaídas no se abrió

🔄 Flujo del Juego

Fase 1: Inicio

1. El **Modelo 1** recibe un prompt del sistema que le indica:
    - Debe inventar una Black Story original
    - Debe presentar solo la situación final (no revelar la solución)
    - Debe explicar las reglas del juego al Modelo 2
2. El **Modelo 1** genera y presenta:

    🎭 HISTORIA:

    [Situación final misteriosa]

    📋 REGLAS:

- Solo puedes hacer preguntas que se respondan con SÍ, NO o NO ES RELEVANTE

- Cuando creas tener la solución completa, di "RESOLVER:" seguido de tu explicación

- Tienes un máximo de 20 preguntas

¡Empieza a preguntar!

Fase 2: Interrogatorio

3. El **Modelo 2** hace preguntas (una por turno)
4. El **Modelo 1** responde únicamente: **SÍ**, **NO** o **NO ES RELEVANTE**
5. Se repite hasta que:
    - El Modelo 2 dice "RESOLVER: [explicación]"
    - Se alcanza el límite de 20 preguntas (derrota automática)

Fase 3: Resolución

6. El **Modelo 1** evalúa la explicación del Modelo 2:
    - Si es correcta → "🎉 ¡CORRECTO! Has ganado. [Explica la historia completa]"
    - Si es incorrecta → "❌ INCORRECTO. La verdadera historia es: [Explica la historia completa]"

Fase 4: Entre turnos

- Después de cada mensaje, el programa **espera que el usuario presione ENTER** antes de continuar
- Esto permite leer con calma cada respuesta

💻 Parámetros de Ejecución

Obligatorios:

-m1, --model1 Nombre del primer modelo (Story Master)

-m2, --model2 Nombre del segundo modelo (Detective)

-p1, --provider1 Proveedor del modelo 1 (gemini, ollama)

-p2, --provider2 Proveedor del modelo 2 (gemini, ollama)

Opcionales:

--save-format Formato de guardado (json|txt|md). Default: md

--max-questions Máximo de preguntas permitidas. Default: 20

--no-pause Desactiva la pausa entre mensajes

--output-dir Carpeta donde guardar conversaciones. Default: ./conversations

Ejemplo de uso:

Gemini vs Gemini

python main.py -m1 gemini-2.5-flash -m2 gemini-2.5-flash -p1 gemini -p2 gemini

Gemini vs Ollama

python main.py -m1 gemini-2.5-flash -m2 llama3 -p1 gemini -p2 ollama --max-questions 30

🎨 Visualización en Terminal

Cada mensaje debe mostrarse con:

┌──┐

│ 🎭 MODELO 1 (gemini-2.5-flash) | Story Master │

│ ⏱️ 2024-01-15 14:32:45 | ⚡ 1.2s | 🎫 156 tokens │

├──┤

│ SÍ │

└──┘

Presiona ENTER para continuar...

┌──┐

│ 🔍 MODELO 2 (gemini-2.5-flash) | Detective │

│ ⏱️ 2024-01-15 14:32:47 | ⚡ 1.8s | 🎫 203 tokens │

├──┤

```
│ ¿La muerte fue accidental?                                    │
└──────────────────────────────────────────────────────────────────────────────┘
```

Presiona ENTER para continuar...

Características:

- ✅ Colores diferentes por modelo (usar librería `rich`)
- ✅ Timestamps en cada mensaje
- ✅ Tiempo de respuesta
- ✅ Contador de tokens (si está disponible)
- ✅ Contador de preguntas realizadas
- ✅ Separadores visuales claros
- ✅ Emojis para identificar roles

💾 Guardado de Conversaciones

Ubicación:

- Carpeta por defecto: `./conversations/`
- Personalizable con `--output-dir`
- Se crea automáticamente si no existe

Nombre de archivo:

blackstory_YYYYMMDD_HHMMSS.{formato}

Ejemplo: blackstory_20240115_143245.md

Formatos soportados:

Markdown (.md) - Default

Black Story Game

Fecha: 2024-01-15 14:32:45

Modelo 1: gemini-2.5-flash (Story Master)

Modelo 2: gemini-2.5-flash (Detective)

Resultado: Victoria / Derrota

Preguntas usadas: 15/20

🎭 Modelo 1 [14:32:45] ⚡1.2s 🎫156

SÍ

🔍 Modelo 2 [14:32:47] ⚡1.8s 🎫203

¿La muerte fue accidental?

JSON (.json)


```
{
  "metadata": {
    "date": "2024-01-15T14:32:45",
    "model1": {"name": "gemini-2.5-flash", "provider": "gemini", "role": "Story Master"},
    "model2": {"name": "gemini-2.5-flash", "provider": "gemini", "role": "Detective"},
    "result": "victory",
    "questions_used": 15,
    "max_questions": 20
  },
  "messages": [
    {
      "model": "model1",
      "timestamp": "2024-01-15T14:32:45",
      "content": "SÍ",
      "response_time": 1.2,
      "tokens": 156
    }
```

```
  ]
}
```

TXT (.txt)

```
=== BLACK STORY GAME ===

Fecha: 2024-01-15 14:32:45

Modelo 1: gemini-2.5-flash (Story Master)

Modelo 2: gemini-2.5-flash (Detective)

Resultado: Victoria

Preguntas: 15/20

---

[14:32:45] MODELO 1 (1.2s, 156 tokens):

SÍ

[14:32:47] MODELO 2 (1.8s, 203 tokens):

¿La muerte fue accidental?
```

📁 Estructura del Proyecto

```
black-stories-ai/

├── main.py                  # Punto de entrada

├── pyproject.toml           # Configuración uv y dependencias

├── README.md                # Documentación completa

├── .env                     # API keys (NO COMITEAR)

├── .env.example             # Plantilla de ejemplo

├── .gitignore               # Archivos ignorados

├── .claudeignore            # Archivos ignorados por Claude
```

```
│
├── src/                    # Código fuente
│   ├── __init__.py
│   ├── game/               # Lógica del juego
│   │   ├── __init__.py
│   │   ├── orchestrator.py   # Orquestación principal del juego
│   │   ├── prompts.py        # Prompts del sistema para cada modelo
│   │   └── rules.py          # Reglas y validaciones del juego
│   │
│   ├── providers/          # Proveedores de IA
│   │   ├── __init__.py
│   │   ├── base.py         # Clase base abstracta
│   │   ├── gemini.py       # Implementación Google Gemini
│   │   └── ollama.py       # Implementación Ollama
│   │
│   ├── display/            # Visualización en terminal
│   │   ├── __init__.py
│   │   ├── formatter.py    # Formateo de mensajes con colores
│   │   └── ui.py           # Componentes UI (cajas, separadores)
│   │
│   └── storage/            # Guardado de conversaciones
│       ├── __init__.py
│       ├── saver.py        # Lógica de guardado
│       ├── formats/        # Diferentes formatos de exportación
```

```
│       │   ├── __init__.py
│       │   ├── markdown.py
│       │   ├── json.py
│       │   └── txt.py
│       └── models.py     # Modelos de datos (Message, Conversation)
│
├── conversations/        # Partidas guardadas (generado)
│
└── prompts/              # Historial de prompts del desarrollo
    ├── 001_initial_setup.md
    ├── 002_add_providers.md
    └── ...
```

🔧 Archivos de Configuración

pyproject.toml

```
[project]
name = "black-stories-ai"
version = "0.1.0"
description = "Black Stories game with AI models competing"
requires-python = ">=3.10"
dependencies = [
    "google-generativeai>=0.3.0",
    "ollama>=0.1.0",
```

```
    "rich>=13.0.0",
    "python-dotenv>=1.0.0",
    "click>=8.1.0",
]

[project.scripts]
blackstory = "main:main"
```

.env.example

```
# Google Gemini API Key
# Obtener en: https://makersuite.google.com/app/apikey
GEMINI_API_KEY=tu_api_key_aqui

# Ollama (local, no requiere API key)
# Instalar desde: https://ollama.ai/
OLLAMA_BASE_URL=http://localhost:11434
```

.gitignore

```
# Environment
.env
.venv/
venv/

# Python
__pycache__/
*.py[cod]
*$py.class
*.so
.Python
```

Conversaciones guardadas

conversations/

IDE

.vscode/

.idea/

*.swp

*.swo

OS

.DS_Store

Thumbs.db

.claudeignore

.env

conversations/

__pycache__/

.venv/

🎯 Prompts del Sistema

Modelo 1 (Story Master)

Eres el maestro de una Black Story. Tu trabajo es:

1. CREAR una historia misteriosa original con:

   - Una situación final sorprendente/macabra

   - Una explicación lógica de cómo se llegó ahí

   - Detalles suficientes para que sea resoluble

2. PRESENTAR al jugador:

- Solo la situación final (NO reveles la solución)

- Las reglas del juego

- Límite de preguntas: {max_questions}

3. RESPONDER preguntas ÚNICAMENTE con:

- "SÍ" - si la pregunta es correcta

- "NO" - si la pregunta es incorrecta

- "NO ES RELEVANTE" - si no afecta a la solución

NUNCA des pistas adicionales ni información extra.

4. EVALUAR cuando el jugador diga "RESOLVER:":

- Si la explicación cubre los puntos clave → "🎉 ¡CORRECTO! [explica historia completa]"

- Si falta información importante → "❌ INCORRECTO. [explica historia completa]"

Mantén un tono misterioso pero justo.

Modelo 2 (Detective)

Eres un detective resolviendo una Black Story.

SITUACIÓN:

{situacion_del_modelo_1}

REGLAS:

- Solo puedes hacer preguntas que se respondan con SÍ, NO o NO ES RELEVANTE

- Tienes máximo {max_questions} preguntas

- Cuando creas tener la solución completa, di "RESOLVER:" seguido de tu explicación

ESTRATEGIA:

1. Haz preguntas amplias primero (¿Es un accidente? ¿Hay más personas involucradas?)

2. Afina según las respuestas

3. No intentes resolver hasta tener confianza

Preguntas restantes: {preguntas_restantes}

¡Empieza a investigar!

🛠️ Instalación y Uso

1. Clonar e instalar:

```
# Clonar el repositorio

git clone <repo-url>

cd black-stories-ai

# Instalar con uv

uv sync

# Configurar variables de entorno

cp .env.example .env

# Editar .env y añadir tu GEMINI_API_KEY
```

2. Ejecutar:

```
# Modo básico (Gemini vs Gemini)

uv run python main.py -m1 gemini-2.5-flash -m2 gemini-2.5-flash -p1 gemini -p2 gemini

# Con opciones personalizadas

uv run python main.py \

  -m1 gemini-2.5-flash \

  -m2 llama3 \

  -p1 gemini \

  -p2 ollama \

  --max-questions 30 \
```

--save-format json \

--output-dir ./my-games

⚠️ Manejo de Errores

El sistema debe manejar robustamente:

1. **API Key faltante:**

   ❌ Error: Falta GEMINI_API_KEY en el archivo .env

   👉 Copia .env.example a .env y añade tu API key

2. **Modelo no responde:**

   ⚠️ El modelo no respondió en 30s. Reintentando (2/3)...

3. **Interrupción (Ctrl+C):**

   ⚠️ Juego interrumpido por el usuario

   💾 Guardando conversación hasta este punto...

   ✅ Guardado en: conversations/blackstory_20240115_143245.md

4. **Límite de preguntas alcanzado:**

   ❌ Has alcanzado el límite de 20 preguntas

   🎭 El Modelo 1 revela la historia...

5. **Proveedor no disponible:**

   ❌ Error: No se puede conectar con Ollama

   👉 ¿Está Ollama ejecutándose? Prueba: ollama serve

📝 Logging

Usar logging básico para debugging:

import logging

```
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('blackstory.log'),
        logging.StreamHandler()
    ]
)
```

📋 Carpeta de Prompts

En cada commit significativo, guardar en `prompts/` un archivo:

```
prompts/
├── 001_setup_inicial.md          # Este prompt
├── 002_implementar_gemini.md     # Cuando se agregue Gemini
├── 003_añadir_ollama.md          # Cuando se agregue Ollama
└── ...
```

Cada archivo debe contener:

- Fecha y hora
- Descripción del cambio
- Prompt exacto usado con Claude

🚫 Restricciones Importantes

NO HACER:

❌ Ejecutar la aplicación (el usuario hace las pruebas) ❌ Instalar librerías manualmente con pip ❌ Comitear el archivo `.env` ❌ Usar print() para logs importantes (usar logging) ❌ Hardcodear API keys en el código

SÍ HACER:

✅ Usar `uv` para todas las dependencias ✅ Comentar métodos y clases claramente ✅ Manejar todos los errores posibles ✅ Crear código modular y reutilizable ✅ Seguir PEP 8 para estilo de código ✅ Usar type hints en funciones ✅ Crear tests básicos (opcional pero recomendado)

📦 Dependencias Requeridas

```
dependencies = [
    "google-generativeai>=0.3.0",  # API de Google Gemini
    "ollama>=0.1.0",               # Cliente de Ollama
    "rich>=13.0.0",                # Formateo terminal con colores
    "python-dotenv>=1.0.0",        # Carga de variables .env
    "click>=8.1.0",                # Argumentos CLI elegantes
]
```

🎉 Mensaje de Commit

```
feat: Implementar juego Black Stories con IA vs IA

- Sistema de orquestación de conversación entre dos modelos

- Soporte para providers Gemini y Ollama

- Interfaz de terminal con colores y formato rico

- Guardado automático en MD/JSON/TXT con metadata
```

- Manejo robusto de errores y Ctrl+C

- Sistema de prompts con roles específicos

- Pausa configurable entre mensajes

- Logging para debugging

✅ Checklist de Completitud

Antes de considerar el proyecto terminado:

- ☐ `main.py` funciona como punto de entrada
- ☐ Todos los proveedores implementados (Gemini, Ollama)
- ☐ Sistema de colores y formato funcional
- ☐ Guardado en los 3 formatos (MD, JSON, TXT)
- ☐ README.md completo con ejemplos
- ☐ `.env.example` creado
- ☐ `.gitignore` y `.claudeignore` configurados
- ☐ Manejo de errores completo
- ☐ Logging implementado
- ☐ Código comentado adecuadamente
- ☐ Estructura modular con responsabilidades claras
- ☐ Type hints en funciones principales
- ☐ Carpeta `prompts/` creada con primer prompt

🎓 Notas Finales

Este proyecto es un ejercicio de:

- **Orquestación de IA**: Hacer que dos modelos interactúen de forma estructurada
- **Diseño de prompts**: Crear instrucciones claras para roles específicos
- **Ingeniería de software**: Código limpio, modular y mantenible
- **UX en terminal**: Hacer que la experiencia sea clara y agradable