Documento de Apresentação DETALHADO:

Ferramenta AI Programming Benchmark

1. Introdução

- **Objetivo Central do Projeto:**
  - Criar uma plataforma de benchmark sistemática e extensível para avaliar criticamente a capacidade de modelos de IA de ponta (Claude, ChatGPT, Gemini) em gerar código Python funcional, de qualidade e eficiente para resolver problemas de programação bem definidos.
- **Motivação Aprofundada:**
  - A ascensão da IA Generativa está a transformar o desenvolvimento de software, mas a avaliação das suas capacidades de codificação muitas vezes permanece anedótica ou limitada a exemplos simples.
  - Existe uma necessidade premente de métodos de avaliação **padronizados, reproduzíveis e quantitativos** que permitam:
    1. Comparar diferentes modelos de IA de forma justa.
    2. Identificar os pontos fortes e fracos específicos de cada modelo em diferentes tipos de problemas de programação.
    3. Rastrear a evolução do desempenho dos modelos ao longo do tempo.
    4. Fornecer dados concretos para a tomada de decisões sobre a adoção destas ferramentas.
- **Visão Geral Detalhada da Ferramenta:**
  - Esta ferramenta é um orquestrador Python que automatiza um fluxo de trabalho de benchmark complexo:
    1. **Seleção e Apresentação de Desafios:** Utiliza um conjunto pré-definido de desafios algorítmicos.
    2. **Interação com Múltiplas APIs:** Comunica com as APIs das IAs configuradas, enviando prompts específicos.
    3. **Processamento Inteligente de Respostas:** Extrai o código relevante das respostas textuais das IAs.
    4. **Avaliação Multifacetada:** Analisa as soluções geradas sob vários prismas:
       - **Qualidade Estática:** Utiliza `flake8` para aderência a boas práticas (PEP8) e deteção de erros comuns.
       - **Corretude Funcional:** Executa o código contra um conjunto de casos de teste predefinidos, verificando se produz os resultados esperados.
       - **Completude (Heurística):** Avalia se a resposta parece conter os elementos essenciais de uma solução.

        - **Eficiência (Proxy):** Mede o tempo de resposta da API e, quando possível, o tempo de execução do código nos testes.
    5. **Robustez Estatística:** Executa cada teste múltiplas vezes (`num_runs`) para mitigar a variabilidade e permitir a análise de médias e distribuições.
    6. **Relatórios Abrangentes:** Consolida todas as métricas e resultados em formatos acessíveis (JSON, CSV) e gera visualizações gráficas para facilitar a interpretação.

2. Arquitetura e Design Detalhado

- **Estrutura Orientada a Objetos (`AIBenchmark`):**
    1. A classe `AIBenchmark` serve como o contentor central, encapsulando:
        - **Estado:** Configurações (chaves API, nomes de modelos, `num_runs`, `output_dir`), definições dos desafios, resultados acumulados (`self.results`, `self.detailed_test_results`), informações do ambiente.
        - **Comportamento:** Métodos que representam cada etapa lógica do benchmark (ex: `_configure_gemini_api`, `query_claude`, `extract_code`, `evaluate_solution`, `_execute_lru_locally`, `generate_report`, `create_visualizations`).
    2. **Benefícios:** Esta abordagem melhora a legibilidade, facilita a adição de novas funcionalidades (ex: suportar outra IA, adicionar um novo tipo de avaliação) e permite instanciar e executar o benchmark de forma controlada.
- **Configuração Flexível e Hierárquica:**
    1. A ferramenta adota uma estratégia de configuração em camadas para máxima flexibilidade:
        - **Ficheiro `.env`:** Carregado no início (`load_dotenv()`), ideal para configurações base e segredos (chaves API). Define os valores padrão se não forem sobrescritos.
        - **Argumentos de Linha de Comando (`argparse`):** Analisados no ponto de entrada (`if __name__ == "__main__":`). Permitem ao utilizador especificar configurações *para uma execução específica*, **tendo prioridade sobre** os valores definidos no `.env` ou os fallbacks internos. Por exemplo, `python script.py -n 10 --openai_model gpt-4o` usará 10 runs e o modelo `gpt-4o`, mesmo que o `.env` defina `num_runs=3` e `OPENAI_MODEL_NAME=gpt-4-turbo`.
        - **Valores Padrão (Fallbacks):** Definidos diretamente no código (ex: no `__init__`), usados se uma configuração não for fornecida nem via CLI nem via `.env`.
    2. **Utilidade:** Esta hierarquia permite guardar configurações comuns no `.env` e ajustar parâmetros específicos facilmente pela linha de comando para experimentação.

- **Fluxo de Dados e Interação dos Componentes:**
    1. **Inicialização:** `__init__` carrega config (CLI > .env > Fallback), define desafios, prepara diretório de saída, configura logging, verifica `flake8`, coleta info do ambiente.
    2. **Loop Principal (`run_benchmark`):** Itera `num_runs` vezes. Dentro de cada run, itera sobre `self.code_challenges`.
    3. **Consulta API (por desafio/run):** Constrói o prompt -> Chama `query_claude`, `query_chatgpt`, `query_gemini` -> Recebe dicionário de resposta (`content`, `execution_time`, `last_error`).
    4. **Avaliação (`evaluate_solution` por IA/desafio/run):**
        - Recebe a resposta da API.
        - Chama `extract_code` -> Obtém string de código.
        - Chama `count_code_lines`.
        - Chama `measure_response_completeness`.
        - Chama `analyze_code_quality_flake8`.
        - Chama `_run_test_cases_fixed` (para executor externo) ou `_execute_lru_locally` (para LRU).
        - `_run_test_cases_fixed` chama `_call_executor_service` para cada caso de teste.
        - `_execute_lru_locally` cria um processo worker (`_lru_worker`) que executa o código LRU via `exec` e comunica o resultado via `multiprocessing.Queue`.
        - `evaluate_solution` agrega os resultados desta avaliação específica e adiciona um dicionário de sumário a `self.results` e entradas detalhadas a `self.detailed_test_results`.
    5. **Geração de Relatório (`generate_report` - após todas as runs):**
        - Chama `_prepare_dataframe` para criar um Pandas DataFrame agregado (média sobre `run_index`) a partir de `self.results`.
        - Chama `_calculate_aggregated_test_metrics` para calcular taxas de sucesso e tempos médios a partir de `self.detailed_test_results`.
        - Chama `_calculate_statistics` para calcular estatísticas gerais por IA (média/std dev das métricas agregadas).
        - Chama as funções `_save_*` para escrever os ficheiros TXT, JSON e CSV.
        - Chama `create_visualizations` para gerar os gráficos.
        - Chama `_display_summary_table` para imprimir o resumo final no console.

3. Funcionalidades Ultra-Detalhadas

- **Modelos de IA Suportados:** Claude (Anthropic), ChatGPT (OpenAI), Gemini (Google). A inclusão no benchmark é automática se a chave API correspondente (`CLAUDE_API_KEY`, `OPENAI_API_KEY`, `GEMINI_API_KEY`) for encontrada.

- **Desafios de Programação Detalhados:**

    1. O conjunto de desafios foi cuidadosamente selecionado para abranger diferentes conceitos fundamentais da ciência da computação e níveis de complexidade:
- | Desafio | Dificuldade | Conceito Principal Testado | Propósito | | :---------- | :---------- | :---------- | :---------- | | Fibonacci Sequence | Easy | Sequências, Iteração/Recursão simples | Avaliar compreensão de algoritmos básicos e sequências. | | Palindrome Check | Easy | Manipulação de Strings, Comparação | Avaliar processamento de texto e lógica de palíndromo. | | Two Sum | Easy | Hash Maps (Dicionários), Busca Eficiente | Avaliar uso de dicionários para busca rápida (complexidade O(n)). | | Binary Search | Medium | Algoritmos de Busca, Divisão e Conquista | Avaliar implementação de busca eficiente (O(log n)) em dados ordenados. | | Merge Sort | Medium | Algoritmos de Ordenação, Recursão | Avaliar implementação de ordenação eficiente (O(n log n)) e recursiva. | | Valid Parentheses | Medium | Pilhas (Stacks), Validação de Sequências | Testar o uso correto de pilhas para estruturas aninhadas. | | Longest Substring Without Repeating Characters | Medium | Janela Deslizante (Sliding Window), Conjuntos/Dicts | Testar a técnica de janela deslizante para problemas de substring. | | Count Islands in Grid | Hard | Travessia de Grafo/Matriz (DFS/BFS) | Testar algoritmos de exploração de matrizes 2D e conectividade. | | LRU Cache | Hard | Design de Estruturas de Dados, Dict+Lista Ligada | Avaliar capacidade de projetar e implementar estruturas de dados complexas. | | Minimum Edit Distance | Hard | Programação Dinâmica (DP) | Avaliar aplicação de DP (algoritmo de Levenshtein) em problemas de strings. |

- **Processo de Benchmark Ultra-Detalhado:**

    1. **Múltiplas Execuções (`num_runs`):** Garante que cada IA tem múltiplas "tentativas" para cada problema, reduzindo o ruído e permitindo analisar a consistência das respostas. O padrão é 3, mas configurável via `-n`.

Geração do Prompt (Exemplo de Estrutura):
Implement the following Python {class_or_function}.
Description: {challenge_description}
Signature:
{challenge_signature}
Provide ONLY the raw Python code definition for the function/class and any necessary standard library imports (like `from collections import OrderedDict` if needed). Do not include any explanations, comments within the code, usage examples, or markdown formatting like ```python.

2. *Este prompt é projetado para instruir a IA a focar na geração do código puro, facilitando a extração.*
3. **Interação com APIs:**
    - **Claude/ChatGPT (requests):** Envia uma requisição POST para o endpoint da API correspondente com headers (incluindo a chave API) e um corpo JSON contendo o nome do modelo e a mensagem (prompt). Espera uma resposta JSON.
    - **Gemini (SDK + tenacity):** Usa o SDK `google-generativeai`. A função `query_gemini` envolve a chamada `model.generate_content` dentro de um decorador `@retry` da biblioteca `tenacity`.
        - `stop=stop_after_attempt(3)`: Tenta no máximo 3 vezes.
        - `wait=wait_exponential(multiplier=1, min=5, max=60)`: Espera exponencialmente entre as tentativas (5s, 10s, 20s...), até um máximo de 60s.
        - `retry=retry_if_exception_type(...)`: Tenta novamente *apenas* se o erro for `google.api_core.exceptions.ResourceExhausted` (rate limit).
        - `before_sleep`: Loga uma mensagem antes de esperar para a próxima tentativa.
    - O tempo de resposta (`api_execution_time`) é medido para cada chamada bem-sucedida ou falhada.

- **Extração e Análise do Código Gerado:**

1. **Extração (`extract_code`):**
    - Passo 1: Procura por ```` ```python\n(CÓDIGO)\n``` ````. Usa `re.search` com `re.DOTALL` para que `.` capture também newlines. O `(.*?)` é um grupo de captura não-ganancioso.
    - Passo 2: Se falhar, procura por ```` ```(?:.*\n)?(CÓDIGO)\n``` ````. Similar, mas permite qualquer linguagem (ou nenhuma) após o primeiro ```. `(?:.*\n)?` é um grupo opcional não-capturado para a linha da linguagem.
    - Passo 3: Se falhar, aplica uma heurística: percorre as linhas; se encontrar uma linha indentada ou começando com `def`/`class`, assume que o código começou e adiciona essa e as linhas seguintes indentadas. Para ao encontrar uma linha não indentada (simplista, mas evita texto explicativo posterior).
    - Passo 4 (Fallback): Se a heurística não encontrar nada, verifica se a resposta *inteira* contém keywords comuns de código (`def`, `class`, `import`, `return`, `for`, `while`, `if`, indentação). Se sim, retorna a resposta inteira; senão, retorna vazio.
2. **Análise de Qualidade (`analyze_code_quality_flake8`):**
    - Usa `flake8` (se disponível) que verifica:

        - **Estilo:** Conformidade com o guia de estilo PEP 8 (indentação, espaços, nomes de variáveis, etc.).
        - **Erros Lógicos:** Erros comuns como variáveis não utilizadas, imports não utilizados, erros de sintaxe básicos que o Python parser pode não pegar.
    - Executa `flake8 --count` num ficheiro temporário. A flag `--count` faz o flake8 imprimir o número total de erros/avisos na última linha da saída. O script parseia este número. Retorna -1 se Flake8 não está disponível, o código está vazio, ou a execução do Flake8 falha.
- **Avaliação Detalhada das Soluções:**

    1. **Heurística de Completude (`measure_response_completeness`):**
        - Pontuação base se houver código > 0.
        - Bónus se `def` ou `class` estiver presente.
        - Bónus se `return` ou `yield` estiver presente.
        - Bónus se keywords específicas do desafio estiverem presentes no código (ex: `fibonacci` ou `yield` para Fibonacci; `stack` ou `pop` para Valid Parentheses).
        - Bónus por estruturas de controle (`if`/`for`/`while`) e tratamento básico de casos limite (`if not`, `if len`, `else`).
        - Score final limitado a 100. *É uma métrica auxiliar, não um substituto para os testes de corretude.*
    2. **Teste de Corretude:**
        - **Executor Externo (`_call_executor_service`):**
            - **Payload Enviado (JSON):** `{ "code_string": "...", "function_name": "nome_da_funcao", "test_input_args": [arg1, arg2] }` (O `code_string` inclui imports comuns adicionados automaticamente).
            - **Resposta Esperada (JSON):** `{ "success": True/False, "output": valor_retornado, "stdout": "...", "stderr": "...", "error": "mensagem_erro_se_houver", "execution_time_ms": 123.45 }`
        - **Execução Local LRU (`_execute_lru_locally`, `_lru_worker`):**
            - **Motivo:** LRU Cache requer manter estado entre chamadas (`put` afeta `get` futuro), inviável com o executor externo stateless. Execução local também permite medir tempo de operação mais fino.
            - **Isolamento (`multiprocessing`):** Cria um `Process` para rodar `_lru_worker`. Isso isola a execução do código da IA do processo principal.
            - **Sandbox (`exec` com `safe_globals`):** `_lru_worker` define `safe_globals` contendo `__builtins__` restrito (apenas

funções e tipos seguros como `int`, `list`, `dict`, `len`, `print`, `Exception`, etc., **sem acesso a `open`, `os`, `subprocess`**) e `OrderedDict` (necessário para LRU). O código da IA é executado com `exec(code, safe_globals, local_namespace)`. *Atenção: Sandboxing com `exec` é complexo e potencialmente falível.*
        - **Comunicação (`Queue`):** O processo principal e o worker comunicam através de uma `multiprocessing.Queue`. O worker coloca o dicionário de resultados (`success`, `results` [lista de steps], `error`) na fila.
        - **Timeout e Controlo:** O processo principal usa `process.join(timeout=...)` para esperar pelo worker, mas com um limite de tempo. Se o tempo for excedido (`process.is_alive()` é True), o processo é terminado (`process.terminate()`, `process.kill()`) para evitar bloqueios. O `exitcode` do processo também é verificado.
    - **Comparação de Saídas:**
        - `np.array_equal(actual, expected, dtype=object)`: Compara a saída real com a esperada. É robusto para diferentes tipos (números, strings, listas, listas aninhadas) e trata `NaN`s consistentemente. Usar `dtype=object` ajuda com estruturas potencialmente heterogéneas.
        - Para "Two Sum", usa `sorted(actual) == sorted(expected)` pois a ordem dos índices não importa.
    - **Status dos Testes (em `detailed_test_results`):**
        - `Pass`: Executou sem erro e a saída corresponde à esperada.
        - `Fail`: Executou sem erro, mas a saída é diferente da esperada.
        - `Error`: Ocorreu um erro durante a execução do código no executor ou no worker local (ex: `TypeError`, `IndexError`, Timeout).
        - `API Error`: A IA não forneceu uma resposta (erro na chamada API).
        - `No Code/No Function`: Não foi possível extrair código executável da resposta da IA.
        - `Unknown Executor Status`: Resposta inesperada do serviço executor.
- **Métricas Coletadas Precisas:**

    1. **Por Run (em `self.results`):** `run_index`, `ai`, `challenge`, `difficulty`, `api_execution_time` (s), `lines_of_code` (int), `completeness` (0-100), `flake8_issues` (int ou -1), `num_execution_errors` (int, nº de testes falhados/errados *nesta* run), `first_execution_error` (str), `code` (str),

`full_response` (str), `execution_results_raw` (list/dict da resposta do executor/worker).

2. **Por Teste (em `self.detailed_test_results`):** `run_index`, `ai`, `challenge`, `difficulty`, `test_case_index`, `input_args_str`, `expected_output_str`, `actual_output_str`, `status` (Pass/Fail/Error...), `execution_time_ms` (-1 se N/A), `error_message`, `stdout`.
3. **Agregadas (calculadas em `generate_report`):**
    - Médias por IA/Desafio (sobre as runs): `api_execution_time`, `lines_of_code`, `completeness`, `flake8_issues` (média dos válidos), `avg_num_execution_errors`.
    - Métricas de Teste Gerais (calculadas sobre *todos* os testes de *todas* as runs): `avg_exec_success_rate` (% média de Pass sobre Attempted), `avg_exec_time_ms` (média do tempo dos Pass), `total_tests_passed`, `total_tests_attempted`.

- **Geração Detalhada de Relatórios e Visualizações:**

    1. **Ficheiros de Saída:**
        - `environment_info...txt`: Registo completo do setup da execução.
        - `..._all_runs.json`: Lista de dicionários, um para cada resultado de `evaluate_solution` (um por IA/Desafio/Run). Ideal para processamento posterior.
        - `..._test_details_all_runs.csv`: Tabela com uma linha para cada caso de teste executado em cada *run*, mostrando inputs, outputs, status, tempo, erro. Essencial para análise granular de falhas.
        - `..._summary_agg.csv`: Tabela com uma linha por IA/Desafio, mostrando as métricas médias calculadas sobre as `num_runs`. Útil para comparações diretas por desafio. Inclui cabeçalho comentado com info do ambiente.
        - `..._stats_agg.csv`: Tabela com uma linha por Métrica Geral (Avg API Time, Overall Success Rate, etc.) e colunas para cada IA, mostrando a estatística agregada (média, std dev, soma) sobre todos os desafios. Visão de alto nível do desempenho geral. Inclui cabeçalho comentado.
    2. **Tabela de Sumário (Console):** Usa `tabulate` para exibir o conteúdo do `benchmark_stats_agg.csv` de forma legível diretamente no terminal após a execução.
    3. **Gráficos Gerados (`matplotlib`):**
        - **Barras (por métrica/desafio):** *Insight:* Comparação direta do desempenho médio das IAs em cada desafio específico para uma dada métrica. Ajuda a ver em que desafios uma IA se destaca ou falha.

    - **Box Plot (por métrica):** *Insight:* Mostra a distribuição (mediana, quartis, outliers) do desempenho médio das IAs *entre* os diferentes desafios. Ajuda a avaliar a consistência do desempenho de uma IA e a comparar a variabilidade entre elas.
    - **Radar:** *Insight:* Visão holística e comparativa do perfil de desempenho de cada IA através de múltiplas métricas chave normalizadas. Permite identificar rapidamente quais IAs são mais equilibradas ou se destacam em eixos específicos (ex: velocidade vs. corretude).
    - **Tempo LRU:** *Insight:* Foco no desempenho específico da implementação do LRU Cache, comparando o tempo médio por operação entre as IAs.

4. Tecnologias Utilizadas (Papel Detalhado)

- **Python 3:** Linguagem base, escolhida pela sua vasta gama de bibliotecas para ciência de dados, automação e APIs web.
- **`requests`:** Biblioteca padrão de facto para realizar chamadas HTTP de forma simples e robusta (APIs REST, serviço executor).
- **`pandas`:** Essencial para a manipulação dos dados dos resultados. Cria DataFrames que facilitam a agregação (`groupby`), cálculo de médias, e preparação dos dados para CSVs e gráficos.
- **`numpy`:** Usada para operações numéricas eficientes, cálculo de médias/desvios padrão, e crucialmente para a função `np.array_equal`, que fornece comparação robusta de estruturas de dados complexas (saídas dos testes).
- **`matplotlib`:** Biblioteca fundamental para a geração de todas as visualizações estáticas (barras, box plots, radar).
- **`google-generativeai`:** SDK oficial do Google para interagir com a API Gemini de forma conveniente.
- **`python-dotenv`:** Utilidade simples para carregar variáveis de ambiente de um ficheiro `.env`, separando configuração e código.
- **`tenacity`:** Biblioteca poderosa para adicionar lógica de retentativas (retry) a funções, usada para tornar a chamada à API Gemini mais resiliente a falhas temporárias (rate limits).
- **`argparse`:** Módulo padrão do Python para criar interfaces de linha de comando amigáveis, permitindo configurar a execução do script facilmente.
- **`multiprocessing`:** Módulo padrão para criar e gerir processos separados, essencial para a execução isolada e com timeout do código LRU Cache. Usa `Process`, `Queue` e `freeze_support` (para compatibilidade Windows).
- **`logging`:** Módulo padrão para registrar eventos de forma flexível, permitindo diferentes níveis de detalhe (INFO, DEBUG, WARNING, ERROR) e formatação.

5. Pontos Fortes Detalhados da Ferramenta

- **Comparação Abrangente e Multi-facetada:** Vai além da simples verificação "funciona/não funciona". Avalia a *qualidade* (`flake8`), a *corretude* (testes), a *eficiência* (tempos), e a *completude* (heurística), fornecendo um perfil de desempenho rico.
- **Métricas Objetivas e Quantificáveis:** Baseia-se em execuções de testes (Pass/Fail) e ferramentas padrão (`flake8`), minimizando a subjetividade e permitindo comparações numéricas diretas e reproduzíveis.
- **Robustez e Confiabilidade:**
    - **Múltiplas Runs:** Reduz o impacto de flutuações aleatórias.
    - **Tratamento de Erros:** Lida graciosamente com falhas de API, erros de rede, rate limits (`tenacity`), erros na execução do código gerado, e timeouts.
    - **Isolamento:** A execução do código (externa ou local via `multiprocessing`) previne que uma solução falhada derrube todo o benchmark.
- **Flexibilidade e Configurabilidade:** Adapta-se a diferentes necessidades através da configuração fácil de modelos, número de runs, diretório de saída, e URL do executor via CLI e/ou `.env`.
- **Relatórios Detalhados e Visuais:** Produz múltiplos artefactos (JSON, CSVs, Gráficos, Tabela Console) que atendem a diferentes níveis de análise, desde a visão geral até a depuração granular de falhas.

6. Considerações e Possíveis Evoluções Detalhadas

- **Segurança da Execução Local (`exec`):**
    - **Risco:** Apesar do esforço de sandboxing (`safe_builtins`), `exec` pode ser explorado. Um código malicioso poderia tentar usar técnicas como introspecção (`__subclasses__`, `__globals__`) para aceder a funcionalidades restritas ou executar código arbitrário se a sandbox não for perfeita.
    - **Mitigação Atual:** Limita os builtins e globals disponíveis. Usa `multiprocessing` para isolamento a nível de processo.
    - **Alternativas Futuras:** Usar containers (`docker run ...`), microVMs (`firecracker`), ou WebAssembly para um isolamento mais forte e padronizado. Avaliar se o `executor_service` externo poderia ser adaptado para lidar com o estado do LRU.
- **Expansão dos Desafios:** Adicionar desafios que testem:
    - Programação Concorrente/Assíncrona (`asyncio`, `threading`).
    - Manipulação de Ficheiros ou APIs Externas (requereria um executor mais avançado).
    - Bibliotecas Específicas (ex: `pandas`, `numpy` - se o ambiente de execução permitir).
    - Geração de testes unitários para o próprio código.
- **Outras Métricas de Análise:**

- **Complexidade Ciclomática:** Medir a complexidade do código gerado (ex: usando a biblioteca `radon`). Código mais simples é geralmente mais fácil de manter.
- **Análise de Uso de Memória:** Integrar ferramentas de profiling de memória durante a execução dos testes (complexo de implementar de forma robusta e multiplataforma).
- **Análise de Docstrings/Comentários:** Avaliar a qualidade da documentação gerada pela IA (requereria análise de linguagem natural).

7. Como Executar o Benchmark Detalhado

1. **Ambiente:** Assegurar que Python 3 está instalado e acessível no PATH.
2. **Dependências:** Criar e ativar um ambiente virtual (recomendado: `python -m venv venv && source venv/bin/activate` ou `venv\Scripts\activate` no Windows). Instalar as dependências listadas num ficheiro `requirements.txt`: `pip install -r requirements.txt`. (O ficheiro `requirements.txt` deve conter `requests`, `pandas`, `numpy`, `matplotlib`, `google-generativeai`, `python-dotenv`, `tenacity`, `argparse`, `flake8`, `tabulate`).
3. **Configuração (`.env`):** Criar um ficheiro chamado `.env` na mesma pasta do script. Adicionar as chaves API como variáveis de ambiente:
   Snippet de código
   # Exemplo .env
   - OPENAI_API_KEY="sk-..."
   - GEMINI_API_KEY="AIza..."
   - CLAUDE_API_KEY="sk-ant-..."
   - Opcional:
   - OPENAI_MODEL_NAME="gpt-4o"
   - GEMINI_MODEL_NAME="gemini-1.5-flash-latest"
   - CLAUDE_MODEL_NAME="claude-3-haiku-20240307"
   - EXECUTOR_SERVICE_URL="http://localhost:8080/execute"
   - NUM_RUNS=3
   - OUTPUT_DIR="benchmark_output"

4. **Execução (Exemplos de Linha de Comando):**

   - Execução padrão (usa `.env` e defaults, `num_runs=3`):
     python nome_do_script.py

   - Execução com 5 runs, output em `run_results`, usando modelo Gemini específico e log detalhado:

python nome_do_script.py -n 5 -o ./run_results --gemini_model gemini-1.5-flash -v

Execução usando apenas OpenAI (assumindo que as outras chaves não estão no `.env` ou são inválidas) e sobrescrevendo a URL do executor: python nome_do_script.py --executor_url "https://meu-executor.com/exec"

5. **Resultados:** Após a conclusão, verificar a pasta de saída (padrão: `benchmark_reports`) para os ficheiros JSON, CSV, TXT e PNG. Observar a tabela de resumo impressa no terminal.

8. Conclusão e Impacto

A ferramenta AI Programming Benchmark oferece uma plataforma valiosa para a avaliação sistemática e comparativa de modelos de IA na geração de código. Através de uma combinação de testes de corretude, análise de qualidade e medição de desempenho, juntamente com relatórios detalhados e visualizações, permite obter *insights* objetivos sobre as forças e fraquezas de diferentes IAs nesta tarefa crítica. A sua arquitetura modular e configurável torna-a adaptável e útil para pesquisadores, desenvolvedores e equipas que procuram escolher ou entender melhor as ferramentas de IA para programação.

- **Seleção Informada de Ferramentas:** Ajudar desenvolvedores e equipas a escolher o modelo de IA mais adequado para as suas necessidades específicas de programação.
- **Identificação de Lacunas:** Revelar os tipos de problemas onde as IAs ainda lutam, direcionando futuras pesquisas e desenvolvimento de modelos.
- **Monitorização de Progresso:** Permitir re-executar o benchmark periodicamente para avaliar a evolução dos modelos de IA ao longo do tempo.
- **Base para Pesquisa:** Fornecer um framework e dados para estudos académicos sobre as capacidades e limitações da IA na engenharia de software.

Em suma, este projeto representa um passo significativo em direção a uma compreensão mais rigorosa e útil do papel crescente da IA na programação.