Desenho do Experimento

Introdução:
Este experimento tem por objetivo avaliar se há diferenças significativas de desempenho (tempo de resposta e tamanho do payload) entre chamadas feitas via API REST e chamadas feitas via API GraphQL em repositórios GitHub. A seguir, descrevem-se cada item do desenho experimental.

A. Hipóteses Nula e Alternativa

- H_0 (Hipótese Nula): Não há diferença significativa entre REST e GraphQL em termos de tempo de resposta e tamanho do payload.
- H_1 (Hipótese Alternativa): GraphQL é mais eficiente do que REST, gerando menor tempo de resposta e/ou payload de tamanho menor.

B. Variáveis

1. Variáveis Dependentes (o que será medido)
   • Tempo de resposta: intervalo decorrido (em segundos) entre o envio da requisição e o recebimento da resposta completa.
   • Tamanho do payload: tamanho (em bytes) da resposta retornada pela API.
2. Variável Independente (o que será manipulado)
   • Tipo de API: duas condições distintas
   – Condição A: chamada via API REST
   – Condição B: chamada via API GraphQL

C. Tratamentos e Endpoints

Cada "tratamento" corresponde a uma modalidade de consulta a ser aplicada aos objetos experimentais (repositórios GitHub).

1. Tratamento 1 – REST API
   • Endpoint:
   GET https://api.github.com/repos/{owner}/{repo}
   • Descrição: essa chamada retorna os metadados do repositório (incluindo número de issues). O experimento irá medir o tempo de resposta e o tamanho do payload dessa resposta JSON.
2. Tratamento 2 – GraphQL API
   • Endpoint:
   POST https://api.github.com/graphql
   Headers: Content-Type: application/json
   Body:

```
{
  "query": "
    query($owner:String!, $repo:String!) {
      repository(owner:$owner, name:$repo) {
```

```
            issues(first:100) { totalCount }
          }
        }
      ",
      "variables": { "owner": "{owner}", "repo": "{repo}" }
    }
```

• Descrição: esta consulta GraphQL solicita exatamente o mesmo dado (número total de issues), mas em um único POST. Também serão coletados tempo de resposta e tamanho do payload retornado.

D. Objetos Experimentais

• Repositórios GitHub selecionados
– Conjunto de pares (owner, repo) extraídos de uma lista pré-definida (por exemplo, 10 repositórios populares de diferentes domínios).
– Cada par será utilizado como objeto experimental em ambos os tratamentos (REST e GraphQL), garantindo experimento pareado.

E. Tipo de Projeto Experimental

• Desenho Pareado (Within-Subjects / Repeated Measures)
1. Cada repositório escolhido será avaliado sob as duas condições (REST e GraphQL), em rodadas separadas.
2. Como cada objeto (repositório) vive em ambas as condições, eventuais diferenças intrínsecas do repositório (tamanho, número de issues, localização geográfica do servidor) ficam controladas.
3. Serão realizadas medidas repetidas: a mesma sequência de repositórios será consultada várias vezes, alternando a ordem (REST primeiro ou GraphQL primeiro) para evitar viés de cache ou de ordem.

F. Quantidade de Medições

• Número de replicações (n)
– Para cada repositório e cada tratamento, execute n = 30 medições independentes de tempo de resposta e tamanho do payload.
– Total de medições = (número de repositórios) × 2 tratamentos × 30 repetições.
– A escolha de 30 repetições baseia-se em boas práticas estatísticas para obter distribuição aproximada à normalidade dos tempos.

G. Ameaças à Validade

1. **Validade Interna**
   • Variação de rede: podem ocorrer flutuações na conexão Internet entre a máquina de teste e os servidores GitHub, afetando o tempo de resposta.
   – Mitigação: intercalar as chamadas REST/GraphQL de forma aleatória; garantir rede estável; descartar outliers (por exemplo, latências acima de 3× desvio-padrão).
   • Caching no GitHub ou no cliente HTTP: respostas podem ser armazenadas em cache, reduzindo artificialmente o tempo de resposta.

– Mitigação: incluir cabeçalhos HTTP que desabilitem cache (por exemplo, "Cache-Control: no-cache"); aguardar intervalo mínimo de 1 segundo entre requisições.

2. **Validade Externa**
   • Generalização para outros repositórios ou situações: usar apenas repositórios populares pode não refletir comportamento em repositórios pequenos ou privados.
   – Mitigação: selecionar repositórios heterogêneos (variedade de linguagens, portes e números de issues);, se possível, incluir repositórios privados/menos populares num segundo lote de testes.
3. **Validade de Conclusão (Statistical Conclusion Validity)**
   • Erro de medida: imprecisão na captura de timestamps pode enviesar o cálculo de tempo.
   – Mitigação: obter timestamps em milissegundos usando o cliente HTTP (por exemplo, Python com time.time() antes e após a requisição); armazenar dados brutos para auditoria.
   • Tamanho do payload: diferenças de codificação/compactação podem causar variação.
   – Mitigação: medir o tamanho exato do corpo da resposta JSON em bytes (sem contar headers), garantindo consistência.
4. **Validade de Construct (Construto)**
   • Definição de "eficiência": se considerarmos apenas tempo ou apenas tamanho, pode haver discrepância.
   – Mitigação: analisar separadamente os dois indicadores; opcionalmente, calcular um índice combinado de eficiência (por exemplo, efficiency_index = tempo × tamanho).
5. **Validade Social/Ecológica**
   • Comportamento em produção vs. teste: condições artificiais de medição podem não refletir uso real por múltiplos clientes simultâneos.
   – Mitigação: reconhecer que o experimento avalia apenas tempo de resposta single-thread; indicar essa limitação no relatório e sugerir estudos futuros com carga concorrente.

Procedimento Detalhado de Coleta e Análise

1. **Preparação do Ambiente**
   a) Definir lista de repositórios em "repos_list.csv", contendo colunas owner, repo.
   b) Exportar variável de ambiente GITHUB_TOKEN com token válido (escopo de leitura de repositórios).
   c) Definir parâmetros em experiment.py:
   - GITHUB_TOKEN = os.getenv("GITHUB_TOKEN")
   - REPOS_CSV = "repos_list.csv"
   - TRIALS = 30
   - OUTPUT_RESULTS = "experiment_results.csv"

2. **Execução do Script de Experimento (experiment.py)**
   Para cada repositório (cada linha em repos_list.csv):
   a) Alternar aleatoriamente entre ordem de chamada (REST primeiro ou GraphQL primeiro).
   b) Tratamento REST:

• start_time = time.time()
• response = requests.get(f"https://api.github.com/repos/{owner}/{repo}", headers={…})
• elapsed = time.time() – start_time
• payload_size = len(response.content)
• Gravar em experiment_results.csv:
owner, repo, treatment, time_seconds, payload_bytes
(ex.: octocat, Hello-World, REST, 0.234, 2048)

c) Aguardar 1 segundo (reduzir efeitos de cache).

d) Tratamento GraphQL:
• query = """
query($owner:String!, $repo:String!) {
repository(owner:$owner, name:$repo) {
issues(first:100) { totalCount }
}
}
"""
• variables = {"owner": owner, "repo": repo}
• start_time = time.time()
• response = requests.post("https://api.github.com/graphql",
headers={"Authorization": f"bearer {GITHUB_TOKEN}", "Content-Type": "application/json"},
json={"query": query, "variables": variables})
• elapsed = time.time() – start_time
• payload_size = len(response.content)
• Gravar nova linha em experiment_results.csv com treatment = GraphQL.

e) Repetir passos b, c e d 30 vezes (ou conforme TRIALS).

3. **Análise Estatística (analysis.py)**
   a) Carregar experiment_results.csv em um DataFrame pandas.
   b) Agrupar por (owner, repo, treatment) e calcular:
   • média e desvio-padrão de time_seconds.
   • média e desvio-padrão de payload_bytes.
   c) Salvar resumo agregado em experiment_summary.csv com colunas:
   owner, repo, treatment, mean_time, std_time, mean_payload, std_payload.

4. **Geração de Dashboards e Histogramas (dashboard.py)**
   a) Ler experiment_summary.csv.
   b) Plotar histogramas comparativos:
   • Distribuição de tempos (REST × GraphQL) para todos os repositórios.
   • Distribuição de tamanhos de payload (REST × GraphQL).
   c) Salvar figuras em arquivos .png e/ou gerar dashboard interativo (por exemplo, HTML ou Jupyter Notebook).

Figura Descritiva