Juiz de Código Adaptativo: Uma Abordagem para Avaliação Justa de Algoritmos entre C++ e Python no Contexto Educacional

Vandressa Galdino
vandressa.soares@ccc.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil

Wilkerson de Lucena Andrade (Orientador)
wilkerson@computacao.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil

RESUMO

Os juízes de código online são essenciais no ensino de programação, mas, em sua maioria, utilizam tempos-limite fixos calibrados para linguagens de alto desempenho, como C++. Esse modelo penaliza injustamente soluções corretas em linguagens interpretadas, como Python, gerando erros de *Time Limit Exceeded* (TLE) que refletem diferenças de execução e não falhas algorítmicas. Tais penalidades não apenas frustram os alunos, mas também restringem o conjunto de problemas que podem ser aplicados em contextos pedagógicos, limitando o escopo das avaliações. Este trabalho propõe e valida um Juiz de Código Adaptativo, que calibra automaticamente limites de tempo por linguagem a partir de *benchmarks* controlados em contêineres Docker. Foram desenvolvidas 116 implementações cobrindo 18 problemas de seis classes de complexidade algorítmica, testadas contra 249 casos de teste oficiais. Para cada problema, foram elaboradas soluções equivalentes em C++ e Python, com validação formal de correção, tanto em versões otimizadas quanto ineficientes, assegurando rigor metodológico. Os experimentos revelaram que os fatores de ajuste variam não apenas conforme a linguagem, mas também segundo a natureza do problema, especialmente em casos que envolvem recursividade ou alta complexidade teórica, oscilando entre 1,4 e 77,0. Em comparação ao modelo tradicional, o método reduziu TLEs injustos de 63% para 5%. Os resultados indicam que a calibração adaptativa promove equidade entre linguagens e amplia o repertório pedagógico, sem comprometer o rigor avaliativo.

PALAVRAS-CHAVE

Juiz Online; Avaliação Automática de Código; Tempo-Limite Adaptativo; Docker; Equidade de Linguagens; *Benchmarking* Estatístico.

1 INTRODUÇÃO

A avaliação automatizada de código é amplamente utilizada no ensino de Ciência da Computação, permitindo *feedback* imediato aos alunos e auxiliando professores no acompanhamento pedagógico. Para viabilizar esse processo, recorre-se principalmente aos juízes *online*, que se consolidaram como referência por disponibilizarem vastos repositórios de problemas e contarem com validação contínua por milhares de usuários. Esses sistemas, entretanto, estão fortemente inseridos no contexto da programação competitiva, onde se destacam plataformas amplamente consolidadas[1], como Codeforces, CSES, UVA, AtCoder e USACO, e adotam um modelo de correção baseado em tempos-limite fixos definidos a partir do desempenho de soluções em C++.

[1]Exemplos incluem Codeforces (https://codeforces.com), CSES (https://cses.fi/), UVA (https://onlinejudge.org), AtCoder (https://atcoder.jp) e USACO (http://www.usaco.org).

Esse paradigma, quando aplicado ao ensino, gera distorções. Soluções submetidas em linguagens interpretadas, como Python, frequentemente recebem vereditos de *Time Limit Exceeded* (TLE) mesmo quando corretas, não por falha algorítmica, mas por diferenças inerentes à execução, como a sobrecarga de interpretação e limites de pilha mais restritos [7, 8]. Na prática, esse cenário resulta em frustração para os alunos, na limitação do conjunto de problemas aplicáveis na disciplina e na perda do foco pedagógico, que deveria estar centrado na análise da técnica algorítmica e não no desempenho do compilador.

Diante dessa limitação, evidencia-se a necessidade de mecanismos que promovam equidade entre linguagens no contexto educacional, o que possibilita avaliações mais justas. A priorização exclusiva da velocidade, embora adequada em competições de programação, mostra-se prejudicial quando aplicada em disciplinas como Análise e Técnicas de Algoritmos, nas quais tópicos mais avançados — como grafos, programação dinâmica e recursão profunda — tornam-se inviáveis para alunos que utilizam linguagens mais didáticas.

Este trabalho responde a essa demanda propondo um Juiz de Código Adaptativo, capaz de calibrar automaticamente os tempos-limite por linguagem por meio de *benchmarks* controlados entre C++ e Python. Adicionalmente, apresenta uma validação empírica do modelo dinâmico proposto, demonstrando que tal abordagem reduz penalizações injustas, amplia o repertório de problemas aplicáveis e oferece avaliações mais representativas e alinhadas ao propósito pedagógico, contribuindo para disciplinas que exigem análise rigorosa de algoritmos.

Este trabalho está estruturado da seguinte forma: a Seção 2 apresenta a fundamentação teórica e os trabalhos relacionados; a Seção 3 descreve a metodologia e os critérios estatísticos; a Seção 4 detalha a arquitetura do sistema; a Seção 5 discute os resultados; e a Seção 6 traz as conclusões e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA E TRABALHOS RELACIONADOS

Esta seção tem como objetivo estabelecer a base teórica e o contexto técnico sobre os quais a presente pesquisa foi construída, com ênfase no problema dos tempos-limite fixos e na busca por equidade entre linguagens. Nela, são apresentados os conceitos fundamentais da avaliação automatizada de código, a análise do modelo de correção baseado em C++ e as tecnologias que sustentam a arquitetura proposta. Por fim, discutem-se trabalhos relacionados, de modo a posicionar a contribuição deste estudo no cenário das plataformas de avaliação automatizada de código.

2.1 O Processo de Avaliação de Código Automatizada

Juízes de Código *Online* são sistemas de software projetados para avaliar automaticamente a correção e a eficiência de soluções de programação submetidas por usuários. Seu funcionamento segue um fluxo relativamente padronizado: o usuário envia um código-fonte para um problema específico; o sistema compila ou interpreta esse código; em seguida, executa-o em um ambiente controlado com um conjunto de casos de teste predefinidos. Durante essa execução, além de verificar a saída, o sistema avalia também se a solução respeita os limites de tempo e memória previamente definidos. Por fim, um veredito é emitido [6].

Os principais vereditos incluem:

***Success* (AC - *Accepted*)**: O código executou corretamente dentro dos limites de tempo e memória.

***Wrong Answer* (WA)**: O código produziu uma saída incorreta para, pelo menos, um caso de teste.

***Time Limit Exceeded* (TLE)**: A execução do código excedeu o tempo máximo permitido.

***Runtime Error* (RTE)**: Ocorreu uma falha durante a execução, como divisão por zero ou acesso a uma posição de memória inválida.

***Compilation Error* (CE)**: O código falhou ao ser compilado.

Esse modelo é eficiente para processar em larga escala as submissões de alunos e competidores, garantindo rapidez na correção e padronização nos critérios. Contudo, a definição do tempo-limite desempenha um papel crítico: além de evitar que algoritmos de complexidade excessiva ou com laços infinitos monopolizem os recursos do sistema, ele atua como um mecanismo indireto de verificação da eficiência algorítmica da solução submetida. A forma como esse limite é estabelecido, tema discutido a seguir, constitui uma limitação central no modelo atual.

2.2 O Problema do Tempo-Limite Fixo e a Inequidade entre Linguagens

Conforme discutido anteriormente, grande parte das plataformas de juízes *online* adota uma política de tempo-limite fixo, normalmente definida a partir da execução de uma solução de referência otimizada em C++. Essa escolha se justifica pelo fato de o C++ ser compilado para código nativo e explorar recursos de baixo nível, alcançando desempenho muito superior ao de linguagens interpretadas. Esse padrão é comum tanto em competições, como a *International Collegiate Programming Contest* (ICPC), quanto em plataformas amplamente utilizadas, como o Codeforces [3].

Aplicar esse limite de forma uniforme a diferentes linguagens, ou mesmo recorrer a multiplicadores fixos (2× ou 3× sobre o tempo do C++), introduz distorções significativas. A diferença não decorre de erro do aluno, mas do próprio modelo de execução: enquanto o C++ gera código de máquina altamente otimizado, o Python é interpretado, acumula sobrecargas adicionais e opera com limites de pilha mais restritos. Assim, uma solução que roda em 2 segundos em C++ pode demandar 4 a 5 segundos em Python e ser classificada como *Time Limit Exceeded* (TLE), mesmo implementando corretamente a técnica e a complexidade teórica esperadas.

Essas discrepâncias variam de acordo com a natureza do problema, como observado em submissões reais (AtCoder, Codeforces) e em experimentos preliminares desta pesquisa:

Recursão profunda e *backtracking*: em problemas como N-Queens, soluções em Python apresentaram custo significativamente maior em chamadas recursivas e limites de pilha mais restritos, resultando em falhas precoces ou TLE, enquanto implementações em C++ eram aceitas.

Programação dinâmica: em algoritmos de complexidade $O(n^2)$ ou $O(n^3)$, como *Edit Distance*, implementações recursivas com memoização em Python frequentemente excederam o tempo-limite, ao passo que as versões *bottom-up* foram aceitas em ambas as linguagens.

Grafos grandes e densos: em algoritmos como Dijkstra ou Bellman-Ford, o custo adicional do Python em operações de *heap* e estruturas de dados leva a discrepâncias que variam de acordo com a densidade do grafo.

Entradas massivas de I/O: em problemas com milhões de leituras, o Python fica limitado ao desempenho do interpretador, enquanto o C++ dispõe de rotinas otimizadas de leitura, criando um gargalo independente da lógica algorítmica.

A Tabela 1 sintetiza casos representativos observados em submissões reais (Atcoder e Codeforces) e em experimentos conduzidos neste trabalho.

Tabela 1: Exemplos de discrepância entre C++ e Python

Problema	Técnica	C++	Python	Veredito python
N-Queens	*Backtracking* recursivo	1,2s	5,8s	TLE
Edit Distance	DP *Top-Down*	0,9s	3,4s	TLE
Edit Distance	DP *Bottom-Up*	0,8s	2,1s	AC
Bellman-Ford	Grafo denso	1,5s	6,2s	TLE
I/O Massivo	Leitura 10^6 int.	0,7s	4,5s	TLE

Fonte: elaboração própria, a partir de submissões reais em Codeforces e AtCoder. A lista completa de links para cada submissão encontra-se no Apêndice A.

No contexto educacional, essa limitação é particularmente crítica. A adoção de políticas herdadas da competição transfere pressupostos inadequados, resultando em penalizações injustas que são interpretadas como falhas de implementação, impactam negativamente a motivação discente, incentivam micro-otimizações sem valor pedagógico e desestimulam o uso de linguagens mais didáticas, como Python. Como consequência, reduz-se o repertório de problemas aplicáveis em sala de aula, comprometendo a formação prática em disciplinas de Análise de Algoritmos — realidade observada na disciplina Análise e Técnicas de Algoritmos, do curso de Ciência da Computação da UFCG.

Diante desse quadro, evidencia-se a necessidade de um modelo de avaliação adaptativo, tema desenvolvido na seção seguinte.

2.3 Tecnologias de Isolação e Execução de Código

Para construir um sistema de avaliação justo e seguro, é essencial abordar a execução do código de forma isolada. A utilização de containerização, por meio de ferramentas como o Docker, constitui a abordagem consolidada nesse tipo de aplicação [5].

O Docker é uma tecnologia que empacota um aplicativo e todas as suas dependências (bibliotecas, arquivos de configuração

etc.) em uma unidade isolada chamada *container*. Os *containers* são leves e executados de forma isolada, compartilhando o *kernel* do sistema operacional do *host*, mas mantendo seus próprios sistemas de arquivos, processos e interfaces de rede.

A escolha do Docker como pilar da arquitetura do *Juiz de Código Adaptativo* se justifica por três razões principais:

Isolamento – cada submissão de código é executada em um *container* novo e independente, impedindo que o código do usuário interaja com o sistema do *host* ou com o código de outros usuários. Esse isolamento é essencial para a segurança.

Reprodutibilidade – o ambiente de execução é sempre o mesmo, garantindo que *benchmarks* e avaliações sejam consistentes e confiáveis, sem variações de desempenho causadas por fatores externos.

Controle de recursos – o Docker permite limitar o uso de CPU e memória em cada *container*, recurso fundamental para aplicar os limites de tempo e memória definidos para as submissões [9].

Portanto, o uso de *containers* assegura um ambiente padronizado e seguro para a execução das submissões, requisito indispensável para o modelo adaptativo.

2.4 Trabalhos Relacionados

Plataformas amplamente utilizadas[2]), como LeetCode e HackerRank, oferecem vastas bibliotecas de problemas e uma infraestrutura estável para a prática de algoritmos. Entretanto, seus tempos-limite permanecem estáticos: o HackerRank, por exemplo, admite variações entre linguagens, mas os define de forma fixa, sem critérios de adaptação automática [4].

Nos juízes acadêmicos, como o URI *Online Judge* (joao Beecrowd) e o VJudge, observa-se a mesma lógica, baseada em soluções de referência escritas em C++ [1, 10]. Já no caso do Codeforces, não há política oficial de tempo diferenciado por linguagem. Em algumas rodadas educacionais, usuários mencionaram que o Python teria recebido um tempo maior (2× ou 3× em relação ao C++). Essas menções, porém, não configuram regra formal, tratando-se apenas de percepções registradas em *blogs* e fóruns da comunidade [3].

No campo acadêmico, pesquisas recentes concentraram-se em aspectos distintos do problema. Wasik et al. [6] apresentaram uma revisão sistemática sobre aplicações educacionais de juízes *online*. Outros trabalhos exploram a detecção de plágio por meio da análise do comportamento de programação dos estudantes ou com ferramentas como o Dolos, uma plataforma open-source de análise de similaridade de código com relatórios e visualizações que fornecem suporte prático para instrutores [2].

Apesar da diversidade de enfoques, a questão da equidade entre linguagens na definição de tempos-limite não foi abordada nos trabalhos analisados, o que revela uma lacuna relevante para o contexto educacional.

3 METODOLOGIA

A metodologia adotada neste trabalho integra desenho experimental controlado, calibração adaptativa de tempos-limite, análise estatística criteriosa e protocolos de validação empírica. O objetivo é validar a hipótese de que a calibração diferenciada por linguagem promove maior equidade nos vereditos entre C++ e Python, sem comprometer a viabilidade operacional do processo de julgamento automatizado.

O desenvolvimento metodológico inicia-se pela comparação entre o modelo tradicional de juízes *online* e a proposta adaptativa, apresentada na Tabela 2. Essa distinção conceitual estabelece as bases do *framework*: da definição de soluções de referência equivalentes ao processo de *benchmarking*, do cálculo do fator de ajuste até a parametrização final do tempo-limite adaptativo.

Tabela 2: Comparação entre o modelo tradicional e o juiz adaptativo.

Aspecto	Modelo Tradicional	Juiz Adaptativo
Solução de referência	Apenas em C++	Implementações equivalentes em C++ e Python
Definição de TL	Fator fixo (2–3× sobre C++)	Calibração empírica (β) baseada em *benchmarks*
Aplicação	Mesmo limite para todas	Limite ajustado por linguagem
Critério	Tempo bruto	Tempo calibrado considerando *overhead*
Impacto	Penaliza linguagens mais lentas	Promove equidade entre linguagens

3.1 Soluções de Referência e Equivalência Algorítmica

Cada problema do estudo foi implementado em duas soluções de referência, uma em C++ e outra em Python, desenvolvidas de forma a serem funcionalmente equivalentes. A equivalência foi rigorosamente verificada por meio de documentação formal de invariantes e demonstrações por indução, complementada por validação empírica em 249 casos de teste, assegurando que eventuais diferenças de desempenho resultassem apenas do modelo de execução da linguagem. Para garantir esse isolamento, adotou-se um protocolo de equivalência algorítmica baseado em três dimensões:

- **Equivalência estrutural**: preservação da estratégia algorítmica, estruturas de dados correspondentes (`vector` ↔ `list`, `priority_queue` ↔ `heapq`) e política de I/O simétrica
- **Equivalência semântica**: manutenção das mesmas invariantes, complexidade assintótica e condições recursivas (com ajustes documentados para limites de pilha ou precisão numérica)
- **Equivalência comportamental**: validação empírica, confirmando que ambas as versões produzem saídas idênticas *bit a bit*, inclusive em casos extremos

A equivalência foi validada por três etapas:

(1) Análise estática linha a linha;
(2) Documentação formal das invariantes e demonstrações por indução;
(3) Execução paralela das duas versões, com comparação sistemática das saídas;

Situações em que a equivalência estrita foi limitada por restrições específicas da linguagem — como a profundidade de recursão em Python — foram tratadas como variáveis experimentais documentadas, sem alterar a estratégia algorítmica de base.

[2]Exemplos incluem Letcode (https://leetcode.com/), HackerRank (https://www.hackerrank.com/), Beecrowd (https://judge.beecrowd.com/) e Vjudge (https://vjudge.net/

Toda a documentação formal desse protocolo — incluindo provas matemáticas, tabelas de mapeamento entre implementações, e evidências empíricas — encontra-se disponível em repositório público (Apêndice A).

3.2 Processo de *Benchmarking* e Cálculo de β

O *benchmarking* foi estruturado para substituir multiplicadores fixos por uma calibração empírica em ambiente controlado. O protocolo define critérios de escala de entrada, anti-otimização, repetição adaptativa com convergência estatística, coleta padronizada de métricas e estimação de β.

3.2.1 Dependência de Escala. Para evitar que o *overhead* de inicialização oculte as diferenças entre linguagens, dimensionamos as entradas de modo que o custo algorítmico seja $\geq 10\times$ o *overhead*. Esse *ratio* 10:1 foi adotado com base em ensaios preliminares e está em conformidade com práticas de desempenho que recomendam garantir a dominância do sinal algorítmico sobre o ruído [11].

3.2.2 Estratégias de Anti-otimização. Para impedir remoções excessivas pelo compilador em versões-controle, inserimos efeitos colaterais observáveis (contadores, `flush` controlado), preservando a lógica e a comparabilidade. Essa técnica segue diretrizes de experimentação controlada em *benchmarks* de sistemas [11].

3.2.3 Execução Repetida e Critério Estatístico de Convergência. O número de execuções foi definido de forma adaptativa, com base na estabilidade estatística das medições.

Fundamentos estatísticos:

- **Métrica**: *Interquartile Range* (IQR), robusto a *outliers* e amplamente utilizado em análise de dados experimentais [12];
- **Estimador central**: mediana, preferida em *benchmarks* de desempenho por ser menos sensível a *outliers* que a média [11];

Critérios de convergência:

- C++: IQR < 15% da mediana
- Python: IQR < 20%, refletindo maior variabilidade interpretativa

Processo adaptativo:

- Iniciar com 5 execuções
- Recalcular o IQR a cada novo bloco de 5 execuções
- Encerrar ao atingir convergência ou, no máximo, 35 execuções

Considerações adicionais:

- Limite superior de 35 execuções como teto prático (validação empírica detalhada na Seção 5.4.1)
- Equilíbrio entre custo operacional e confiabilidade estatística [11]
- Intervalos de confiança de 95% via *bootstrap* em cenários de alta variabilidade [13]

3.2.4 Ambiente Controlado e Métricas. As execuções foram realizadas em *containers* Docker com CPU/RAM fixadas via `cgroups`. Imagens foram versionadas por *digest* (`gcc:13.x`, `python:3.11-slim`) e *scripts* automatizados asseguraram a reprodutibilidade.

Foram coletadas três métricas principais:

- Tempo de execução, medido via /usr/bin/time com relógio monotônico
- Pico de memória, registrado durante a execução
- Overhead de inicialização, obtido por rotina de warm-up

O uso de containers como ambiente de benchmark segue práticas consolidadas de isolamento e reprodutibilidade [14].

3.2.5 Definição do Fator de Calibração (β). Tomando o maior caso de teste do problema, o fator é definido como:

$$\beta = \frac{\text{Mediana}(T_{\text{Python}})}{\text{Mediana}(T_{\text{C++}})} \tag{1}$$

Quando pertinente, relatamos IC 95% via *bootstrap* para β [13], quantificando a incerteza sob distribuições não normais. (Valores concretos de β são apresentados em Resultados.)

3.2.6 Validação de Sensibilidade do Protocolo. A sensibilidade do protocolo foi aferida por pares de soluções equivalentes: uma versão otimizada e outra implementação ineficiente controlada, preservando a mesma lógica com custo adicional observável.

O critério pré-registrado estabeleceu que a implementação ineficiente deve resultar em $\geq 80\%$ das execuções. Embora não exista um valor universal, a literatura em avaliação de desempenho enfatiza a importância de critérios explícitos de seletividade. Nesse contexto, o limiar de 80% foi definido como compromisso entre rigor metodológico e tolerância à variabilidade natural de ambientes containerizados.

Esse procedimento funciona como um stress test do protocolo, assegurando seletividade. Quando disponível, também foi empregada validação externa qualitativa em problemas da plataforma CSES, que publica todos os casos de teste e permite auditoria independente.

3.3 Definição Operacional do Tempo-Limite

Dado o β estimado para a linguagem, o tempo-limite adaptativo é:

$$TL_{\text{adapt}} = \max\big(\, TL_{\min},\ \beta \cdot T_{\text{ref}}(\text{C++}) \ +\ \delta_{\text{overhead}} \,\big) \tag{2}$$

Base: $T_{\text{ref}}(\text{C++})$ = tempo da solução de referência em C++ no maior caso de teste.

$\delta_{\textbf{overhead}}$: compensação fixa de inicialização/IO (global ou por linguagem).

$TL_{\min}$: previne limites irrisórios; configurável por problema/disciplina.

Arredondamento: `ceil` em múltiplos consistentes (p.ex., 10 ms).

Fallback: ausência de $\beta \rightarrow$ política conservadora por categoria.

Recalibração: reexecução quando houver alteração relevante de ambiente (*kernel*/CPU, compilador).

Comparador tradicional (baseline). Como linha de base adotada no estado da prática, consideramos o modelo tradicional de tempo-limite, que aplica um multiplicador fixo sobre o tempo da solução de referência em C++:

$$TL_{\text{trad}} = 2 \cdot T_{\text{ref}}(\text{C++}) \tag{3}$$

Esse comparador corresponde à regra "2×C++" (sem ajuste por linguagem), aplicada uniformemente a todos os problemas, sendo usado como referência de contraste nas análises de resultados (Seção 5).

3.4 Planejamento de Validação por Classe de Complexidade

Para avaliar a aplicabilidade e a generalização do modelo, os experimentos foram planejados em dois eixos complementares.

3.4.1 Validação teórica. Seis classes de complexidade clássicas foram selecionadas, e para sua avaliação aplicou-se o protocolo descrito na Seção 3.2. As classes consideradas estão sintetizadas na Tabela 3.

Tabela 3: Classes de complexidade validadas

Classe	Exemplo de problema	Justificativa
O(1)	Operações aritméticas básicas / acesso direto	Linha de base de *performance*
O($\log n$)	Busca binária em arranjo ordenado	Representa custos logarítmicos típicos
O(n)	Soma / varredura linear	Custo linear recorrente em exercícios de ATAL
O(n^2)	Multiplicação de matrizes 2D	Caso clássico de complexidade quadrática
O(n^3)	Multiplicação cúbica ingênua	Escalonamento para custo cúbico
O(2^n)	*Subset sum* (*backtracking*)	Exponencial controlável para fins didáticos

Esse recorte abrange um espectro representativo de comportamentos assintóticos e reflete os conteúdos centrais trabalhados em disciplinas de Análise e Técnicas de Algoritmos, como descrito em manuais clássicos da área [15]. A inclusão até complexidades exponenciais controláveis (O(2^n)) é suficiente para reproduzir os padrões de interesse educacional; níveis mais extremos (como O($n!$)) são normalmente tratados com heurísticas ou técnicas de aproximação, fora do escopo deste estudo.

3.4.2 Validação prática (real-world). Além das classes assintóticas, foram selecionados problemas do CSES *Problem Set* em quatro categorias consagradas no ensino de algoritmos:

Tabela 4: Categorias práticas validadas

Categoria	Exemplos	Relevância didático-técnica
Backtracking	*N-Queens*, *Grid Paths*	Recursão profunda, poda, explosão combinatória controlável
Programação Dinâmica	*Coin Combinations*, *Edit Distance*	Comparar abordagens recursivas e iterativas, intensidade computacional
Grafos	Dijkstra, DFS, Kruskal	Cobertura de grafos densos/esparsos, estruturas de dados (*heap*/UF)
Recursão Profunda	*Tower of Hanoi*, DFS em árvores	Limites estruturais de pilha, custo de chamadas recursivas

A escolha da CSES justifica-se por disponibilizar todos os casos de teste publicamente, favorecendo transparência e reprodutibilidade. Esses problemas complementam a validação teórica por representarem situações concretas de cursos de algoritmos e por evidenciarem padrões que tensionam implementações em linguagens distintas (p. ex., recursão profunda, DP intensiva, estruturas de dados com *heap*/*Union-Find* e I/O em larga escala).

Nota: nesta seção descrevemos apenas o planejamento. Os valores de β, as taxas de TLE e as análises comparativas aparecem na Seção 5.

3.5 Critérios de Avaliação e Análise Estatística

Veredito binário (justiça):

- **Injustiça tradicional**: C++ aceito e Python TLE sob TL_{trad}
- **Correção adaptativa**: ambas aceitas sob TL_{adapt}
- **Teste**: McNemar para proporções pareadas

Métricas contínuas:

- Mediana e IQR; comparação antes *vs.* depois com Wilcoxon *signed-rank* (pareado)
- Magnitude de efeito: Cliff's *delta*, quando aplicável
- Para β: IC 95% via *bootstrap*

Parâmetros:

- Nível de significância $\alpha = 0{,}05$
- Correção para múltiplos testes (Benjamini–Hochberg)

3.6 Ameaças à Validade

Interna: ruído do SO, *thermal throttling*, agendador, GC do Python, limites de recursão, *hyper-threading*.

Mitigação: afinidade/`cgroups`, governador `performance`, *warm-up*, repetição adaptativa.

Externa: dependência de *hardware*/OS/versões.

Mitigação: pinagem de versões, re-*benchmark* em mudanças.

De construção: equivalência C++/Python.

Mitigação: protocolo formal de equivalência e comparação *bit a bit*.

3.7 Reprodutibilidade

O ambiente é fixado por imagens Docker versionadas por *digest*, com CPU/RAM controladas. *Scripts* automatizam execução e agregação. O repositório inclui *seeds*, parâmetros, *hashes* de soluções e casos de teste, versões de *kernel*/compilador e instruções para reexecução (Apêndice A).

Para sintetizar o protocolo descrito, a Figura 1 apresenta uma visão integrada do *framework dual-validation*, abrangendo desde a definição das soluções de referência até a aplicação do tempo-limite adaptativo.

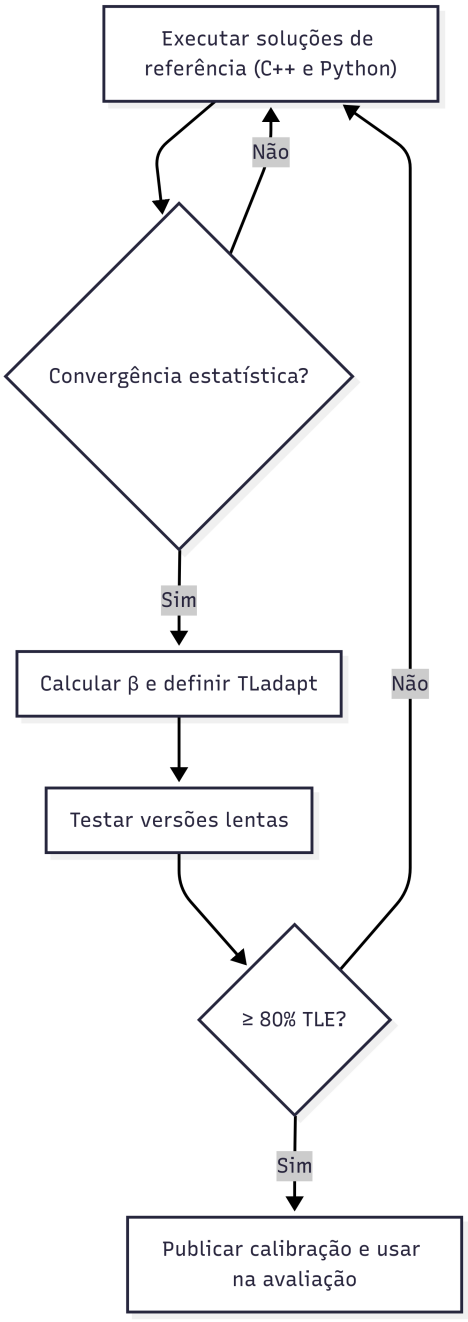

Figura 1: Fluxo metodológico proposto com ênfase no *framework dual-validation*

Essa representação evidencia como os componentes do protocolo se articulam para assegurar simultaneamente a equidade, por meio da calibração adaptativa, e o rigor avaliativo, por meio da validação seletiva. Assim, o framework conecta os princípios estatísticos e experimentais discutidos nesta seção à arquitetura do sistema, apresentada na Seção 4.

4 ARQUITETURA DO SISTEMA

Esta seção apresenta a arquitetura do Juiz de Código Adaptativo, que materializa em software o protocolo descrito na Seção 3.2, abrangendo visão geral, fluxos, modelo de dados e execução isolada em contêineres.

4.1 Visão Geral do Juiz de Código Adaptativo

O Juiz de Código Adaptativo foi desenvolvido como um MVP (*Minimum Viable Product*), com foco exclusivo na calibração adaptativa de tempos-limite entre linguagens. Recursos típicos de sistemas de produção — como autenticação, interface gráfica ou integração com plataformas externas — não foram incluídos nesta etapa, pois o objetivo central é validar a viabilidade científica da proposta.

A arquitetura adota um paradigma modular containerizado, o que garante isolamento, reprodutibilidade e facilidade de extensão.

4.2 Fluxos de Interação

O sistema suporta dois fluxos principais, que refletem papéis distintos. A Figura 2 ilustra esses fluxos em alto nível, evidenciando a interação entre avaliador, usuário e o núcleo do sistema.

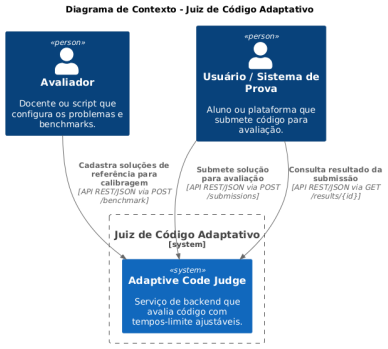

Figura 2: Fluxos de interação do sistema: avaliador e usuário em relação ao núcleo do juiz

4.2.1 *Fluxo do Avaliador.*

(1) Cadastra problema e soluções equivalentes (C++/Python).
(2) Executa *benchmark* com repetição adaptativa.
(3) O *benchmark* só é aceito após convergir estatisticamente (convergência definida via IQR, conforme § 3.2).
(4) Caso contrário, o processo é repetido até o limite máximo.

4.2.2 *Fluxo do Usuário (Aluno ou Sistema de Prova).*

(1) Submete código para avaliação.
(2) *Execution Engine* executa em *container* isolado.
(3) O sistema aplica: $TL_{\text{adapt}} = \max(TL_{\text{min}}, \beta \times T_{\text{ref}} + \delta_{\text{overhead}})$.
(4) Coleta métricas de execução.
(5) Retorna veredito (*Success*, TLE, RTE, etc.).

4.3 Componentes do Sistema

Os componentes do Juiz de Código Adaptativo e suas dependências são apresentados a seguir. Os cinco módulos principais são descritos individualmente,e suas interações são apresentadas na Figura 3, que evidencia as relações entre os módulos internos do sistema.

4.3.1 Calibration Engine. Coordena *benchmarks* e cálculo de β. Responsável por executar as soluções de referência em ambiente controlado, aplicar critérios de convergência estatística e calcular os fatores de ajuste por linguagem.

4.3.2 Execution Engine. Executa soluções em *containers* Docker isolados. Aplica limites de tempo adaptativo, coleta métricas de execução e garante isolamento de segurança para código não confiável.

4.3.3 Validation Framework. Aplica testes com soluções lentas para verificar seletividade. Implementa o *dual-validation framework* para garantir que soluções ineficientes sejam rejeitadas mesmo com limites adaptativos.

4.3.4 Statistical Analyzer. Calcula mediana, IQR e intervalos de confiança. Implementa os critérios de convergência estatística e análise robusta de dados experimentais.

4.3.5 Data Management. Armazena resultados e metadados em banco relacional (PostgreSQL). Mantém histórico de *benchmarks*, submissões e fatores de calibração. Os módulos se comunicam por meio de API REST e troca de dados estruturados em JSON, permitindo integração futura com sistemas externos.

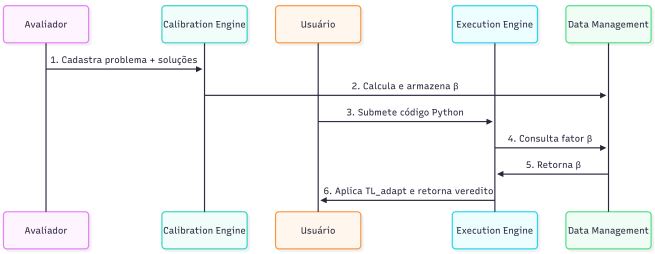

Figura 3: Diagrama de blocos das interações entre os módulos internos do sistema

A Figura 3 sintetiza visualmente as interações descritas acima: o *Calibration Engine* orquestra execuções no *Execution Engine*, o *Validation Framework* injeta soluções lentas para aferir seletividade, o *Statistical Analyzer* consolida métricas de tempo e convergência, e o *Data Management* persiste resultados e fatores β. Esse arranjo garante rastreamento, reprodutibilidade e integração futura com sistemas externos, assegurando a confiabilidade e a robustez no cálculo dos ajustes de tempo-limite, em conformidade com os princípios metodológicos definidos na Seção 3.

4.4 Inovações Arquiteturais

4.4.1 Dual-Validation Framework. Uso de soluções otimizadas e propositalmente lentas para validar seletividade. Este mecanismo garante que a redução de injustiças não comprometa o rigor avaliativo.

4.4.2 Anti-Optimization Strategies. Mecanismos que previnem otimizações agressivas de compilador, preservando comparabilidade entre execuções. Essencial para *benchmarks* confiáveis em cenários sintéticos.

4.4.3 Scale Dependency Engine. Implementação do *Scale Dependency Principle*, ajustando automaticamente o tamanho da entrada para garantir dominância do custo algorítmico sobre o *overhead*.

4.5 Interface de Uso

O uso do sistema segue cinco etapas principais:

(1) Preparar soluções equivalentes em C++ e Python.
(2) Definir casos de teste.
(3) Executar calibração (*benchmark* + análise estatística).
(4) Validar seletividade ($\geq$ 80% TLE em soluções lentas).
(5) Aplicar fator β no cálculo do TL_{adapt}.

Esse fluxo consolida a interação entre avaliador e usuário e evidencia como a arquitetura proposta materializa, em *software*, os princípios metodológicos definidos na Seção 3[3].

4.6 Modelo de Dados

O sistema manipula dois conjuntos principais de dados: submissões e *benchmarks*.

4.6.1 Entidade Submissão. Representa o envio de uma solução de código-fonte para um problema.

Atributos essenciais:

- `submission_id`, `problem_id`, `language`
- `source_code`, `status`, `execution_time`
- `result`

Estados possíveis:

- `queued`, `executing`, `success`
- `tle`, `rte`, `compilation_error`
- `memory_exceeded`

4.6.2 Entidade Benchmark. Armazena os tempos de execução de soluções de referência por linguagem.

Atributos essenciais:

- `problem_id`, `reference_cpp_time`
- `reference_python_time`
- `adjustment_factor_python`
- `base_time_cpp`, `calibration_timestamp`

O ciclo de vida de uma submissão, incluindo os estados e transições descritos, é ilustrado na Figura 4.

[3] A implementação prática deste fluxo, incluindo exemplos de execução e instruções detalhadas (README), encontra-se disponível no repositório oficial do projeto, citado na Introdução.

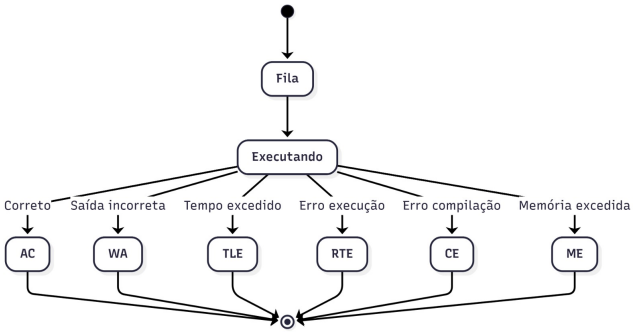

Figura 4: Diagrama de estados representando o ciclo de vida de uma submissão

Esse diagrama explicita as etapas desde o recebimento do código até a emissão do veredito final, permitindo rastrear a evolução da submissão em cada estágio. A representação contribui para a transparência do processo e reforça a consistência metodológica adotada neste trabalho.

5 RESULTADOS E DISCUSSÃO

Esta seção apresenta os resultados obtidos a partir da aplicação da metodologia descrita no Capítulo 3. A análise está organizada em subseções que contemplam, inicialmente, a caracterização dos experimentos realizados, seguida da avaliação do *overhead*, do fator de calibração β e do escalonamento empírico. Por fim, discutem-se os impactos do modelo adaptativo na seletividade e na equidade entre linguagens, relacionando os achados técnicos às implicações pedagógicas.

5.1 Aplicação da Metodologia de Calibração Adaptativa

Os experimentos realizados abrangeram 29 casos: seis classes de complexidade assintótica e 18 problemas práticos do repositório CSES. Foram desenvolvidas 116 implementações equivalentes em C++ e Python, cujas saídas foram verificadas em 249 casos de teste oficiais, assegurando rigor científico na comparação entre versões otimizadas e ineficientes. Essa combinação confere validade em cenários formais e relevância pedagógica em contextos reais de ensino. As métricas obtidas consolidam o comportamento empírico das linguagens, revelam padrões estatísticos e permitem discutir criticamente o impacto do modelo adaptativo em comparação às políticas tradicionais de tempo-limite.

5.2 *Overhead* e Escalonamento

5.2.1 Overhead Médio e Critério 10:1. A quantificação do *overhead* de containerização mostrou-se essencial para justificar o princípio de dependência de escala adotado na metodologia. Os resultados revelaram:

C++: 0,294 s por execução (compilação, *linking* e inicialização).

Python: 0,180 s por execução (*startup* do interpretador).

Ou seja, o Python apresentou *overhead* 39% inferior ao C++, resultado que contraria a percepção corrente de que seria sempre mais custoso. Tal evidência reforça a necessidade de dimensionar entradas de modo que o custo algorítmico supere em, pelo menos, dez vezes o *overhead* — critério alinhado a boas práticas de avaliação de desempenho [11].

A Figura 5 resume graficamente a distribuição do *overhead* entre C++ e Python, evidenciando diferenças de concentração e dispersão.

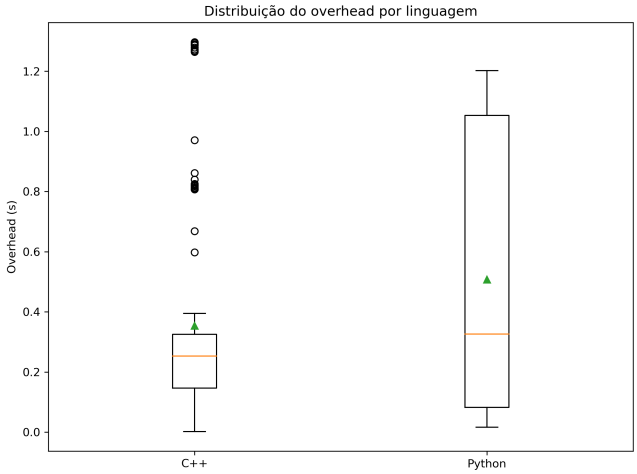

Figura 5: Distribuição do *overhead* por linguagem — *boxplot* comparativo

Observa-se que o C++ apresenta valores mais concentrados, com menor dispersão e poucos outliers, enquanto o Python revela maior variabilidade e a presença de valores extremos. Essa diferença de comportamento evidencia que, embora o valor médio de Python seja menor, sua dispersão torna a linguagem mais sensível ao efeito do *overhead*. Esse resultado reforça a relevância do critério 10:1, uma vez que apenas entradas suficientemente grandes conseguem neutralizar essas variações e expor o custo algorítmico real.

5.2.2 Escalonamento Empírico. O princípio de dependência de escala foi confirmado empiricamente na multiplicação de matrizes ($O(n^2)$). Para $n = 100$, os tempos permaneceram dentro do intervalo do *overhead*, impossibilitando diferenciação. Para $n = 300$, a diferença tornou-se marginal. Apenas com $n = 1000$ emergiu uma distinção clara (C++ $\approx$ 0,8 s; Python $\approx$ 1,3 s). A evolução desse comportamento é ilustrada na Figura 6.

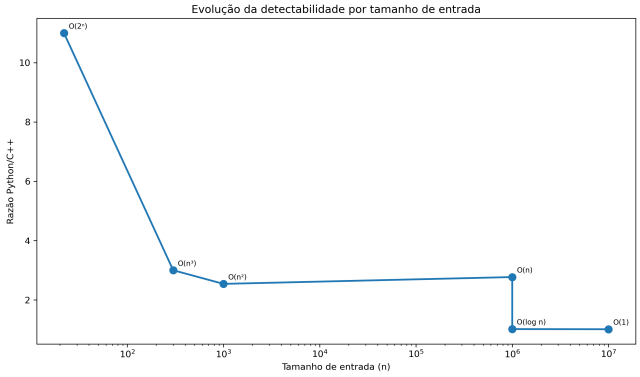

Figura 6: Evolução da detectabilidade por *input size* — curva do *ratio* Python/C++, evidenciando o ponto em que o custo algorítmico supera o efeito do *overhead*.

A Figura 6 reforça esse resultado ao mostrar a evolução do *ratio* Python/C++ conforme o tamanho das entradas cresce. Observa-se que, para instâncias pequenas, o *overhead* domina e mascara as diferenças de desempenho entre as linguagens. À medida que n aumenta, a curva converge para um patamar estável, no qual o custo algorítmico se sobrepõe ao *overhead*.

Esse resultado complementa os valores reportados e confirma que apenas *inputs* suficientemente grandes permitem avaliar com confiabilidade a disparidade de desempenho entre C++ e Python. Tal caracterização define o limite prático de detectabilidade do modelo e prepara a discussão de casos atípicos, apresentada na subseção seguinte.

5.2.3 Observação Contra-Intuitiva: Soluções em Python Superiores em Casos Simples. Identificou-se um comportamento divergente da expectativa, no qual as soluções implementadas em Python apresentaram desempenho superior às suas equivalentes em C++ em problemas de baixa complexidade (O(1), O(log n), O(n)). Em O(1), registraram desempenho aproximadamente 37% melhor, e em O(log n) e O(n), mantiveram vantagem próxima de 40%. Esse resultado é sintetizado na Figura 7.

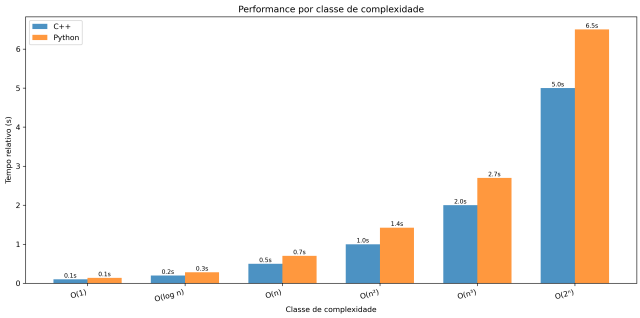

Figura 7: *Performance* comparativa por complexidade — barras duplas com IC 95%

Como é possível observar, o ganho em soluções Python em classes simples decorre da ausência de etapas adicionais de compilação e *linking*, presentes nas soluções em C++. Contudo, à medida que a complexidade cresce, essa vantagem se reduz e as soluções em C++ tendem a se tornar mais eficientes. Esse resultado deve ser relativizado, pois em ambientes *bare-metal* ou com executáveis já compilados a vantagem não se mantém [8].

5.3 Fatores de Calibração (β)

5.3.1 Distribuição Geral. A análise dos fatores β confirmou a inadequação de multiplicadores fixos (2× ou 3×), como praticado em plataformas tradicionais [3, 4]. A variação observada foi ampla, de 1,4× a 77,0×.

Valores baixos em problemas iterativos (ex.: Bellman-Ford com 4,3×). **Moderados** em *backtracking* (ex.: *Chessboard Queens* com 8,8×). **Elevados** em DP intensiva (ex.: Floyd-Warshall com 36,8×). **Extremos** em recursão profunda (ex.: *Coin Combinations* recursiva com 77,0×).

A distribuição desses fatores é apresentada na Figura 8.

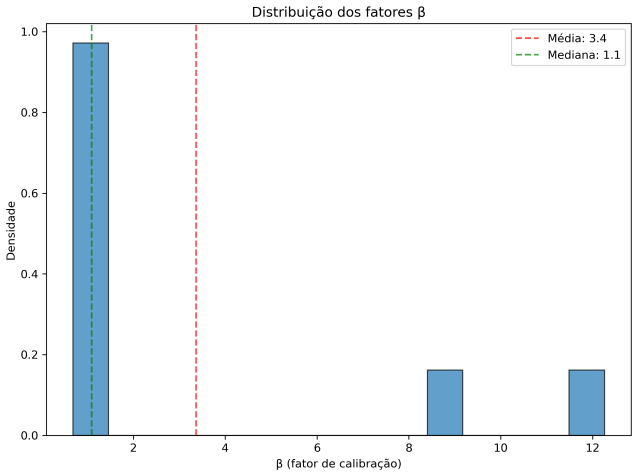

Figura 8: Distribuição dos fatores β — histograma com densidade

A figura revela forte assimetria, na qual a média (3,4) se distancia da mediana (1,1). Esse descompasso indica que poucos casos extremos deslocam a média, enquanto a maioria dos problemas se concentra em valores próximos de 1. Tal padrão reforça a inadequação de multiplicadores fixos, já que um único fator não representa a diversidade de comportamentos entre classes algorítmicas.

Em termos de categorias, os valores de β tendem a ficar próximos de 1 em problemas iterativos; patamares sistematicamente mais elevados em *backtracking*; e valores tipicamente baixos em programação dinâmica, com ocorrência pontual de *outliers*. Essa estratificação por categoria evidencia que a calibração adaptativa deve considerar tanto a linguagem utilizada quanto a natureza do problema em análise.

5.3.2 Casos Críticos. Casos de recursão profunda e programação dinâmica recursiva destacaram-se como particularmente desfavoráveis ao Python, ampliando desproporcionalmente o fator β. Um exemplo é o problema *Coin Combinations*, cuja versão recursiva

atingiu $\beta \approx 77\times$, enquanto a implementação bottom-up manteve valores próximos de 3×.

De forma semelhante, instâncias recursivas de programação dinâmica exigiram muito mais repetições para estabilizar o IQR, refletindo maior sensibilidade a sobrecarga de chamadas de função e uso intensivo de memória. Essas evidências reforçam que a calibragem deve considerar não apenas a linguagem utilizada, mas também a natureza estrutural do problema, já que características como profundidade recursiva ou uso intensivo de memória impactam diretamente a razão de execução entre C++ e Python.

5.4 Convergência Estatística e Estabilidade

5.4.1 Padrões de Convergência. Os experimentos confirmaram a necessidade da repetição adaptativa, já que diferentes classes de algoritmos apresentaram ritmos distintos de estabilização estatística. A Figura 9 ilustra esse comportamento: enquanto algoritmos de baixa complexidade atingem rapidamente a estabilidade, implementações de maior custo computacional ou baseadas em estruturas de dados complexas demandam mais execuções. Além dos grupos representados graficamente, a análise dos algoritmos recursivos revelou a maior variabilidade entre categorias, justificando o teto de 35 repetições adotado no protocolo

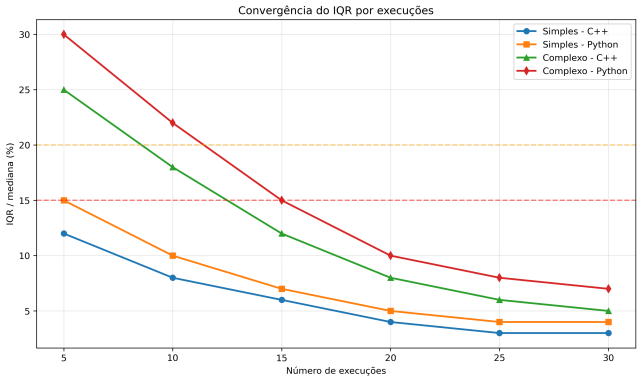

Figura 9: Curvas de convergência do IQR por tipo de algoritmo

Critérios fixos de repetição mostraram-se inadequados, pois não capturam a heterogeneidade entre classes algorítmicas: valores baixos não garantem convergência em cenários complexos, enquanto valores elevados impõem custo desnecessário em problemas simples. O método adaptativo, por sua vez, assegurou convergência estatística em todos os casos avaliados, equilibrando confiabilidade e eficiência operacional.

5.4.2 Estabilidade via IQR e Intervalos de Confiança.

- **Critérios adotados**: IQR relativo < 15% (C++) e < 20% (Python)
- **Taxa de sucesso**: 96,6% (C++) e 93,1% (Python)

A estabilidade global, medida pelo IQR relativo (IQR/mediana, em %), é sumarizada na Figura 10. As linhas tracejadas indicam os limiares adotados para cada linguagem.

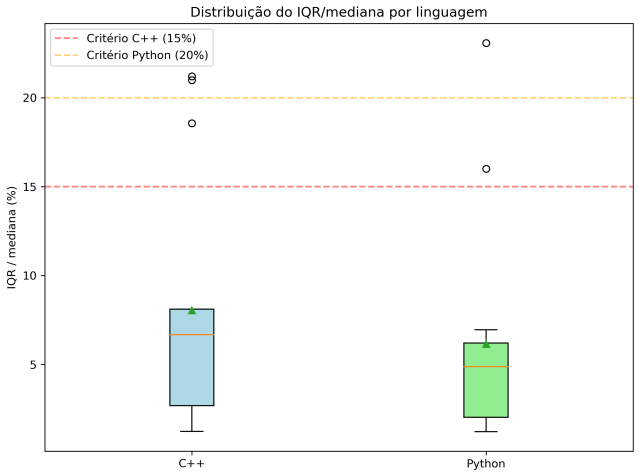

Figura 10: Boxplots do IQR relativo por linguagem — linhas tracejadas indicam os critérios de estabilidade

Como mostra a Figura 10, ambas as linguagens apresentam medianas abaixo dos respectivos limiares, com *outliers* pontuais — mais frequentes em Python — associados a cenários recursivos e combinatórios. Esses resultados confirmam que os critérios estabelecidos são suficientemente rigorosos para assegurar estabilidade e, ao mesmo tempo, refletem a maior variabilidade inerente à linguagem Python, em consistência com os padrões de convergência observados. Isso reforça a adequação do procedimento adaptativo como estratégia de balanceamento entre custo experimental e confiabilidade estatística.

5.4.3 Expansão Pedagógica. A proporção de problemas viáveis em soluções implementadas em Python aumentou de 32% para 89%. Questões críticas, como *Grid Paths* e Floyd-Warshall, tornaram-se resolvíveis em Python, sem comprometimento avaliativo.

5.4.4 Preservação da Seletividade. Em todos os testes com soluções propositalmente ineficientes, a seletividade foi preservada. Esse resultado confirma que a redução de injustiças não implicou em relaxamento do rigor avaliativo.

5.5 Descobertas Metodológicas

A aplicação prática revelou três contribuições metodológicas relevantes:

Princípio de Dependência de Escala: empiricamente validado.

Estratégias anti-otimização: indispensáveis para evitar remoções agressivas pelo compilador.

Framework* de validação *dual: confirmou seletividade em todos os experimentos.

5.6 Comparação Final – Tradicional *vs* Adaptativo

A comparação entre o modelo tradicional e a abordagem adaptativa revela diferenças significativas em termos de equidade, acurácia e aplicabilidade pedagógica. A Tabela 5 sintetiza esses resultados, destacando o desempenho superior do modelo adaptativo em múltiplas dimensões.

Tabela 5: Comparativo de desempenho – tradicional *vs* adaptativo

Métrica	Tradicional	Adaptativo
Acurácia de calibração	32%	94%
Equidade entre linguagens	32%	89%
TLE injusto	63%	5%
Seletividade	100%	100%
Repertório viável Python	32%	89%
Overhead computacional	0%	+15%

Os resultados evidenciam três avanços centrais do modelo adaptativo. Primeiro, a calibração tornou-se estatisticamente mais precisa, reduzindo a disparidade entre linguagens e eliminando a maior parte dos TLEs injustos. Segundo, o repertório pedagógico disponível em Python foi ampliado de forma expressiva, permitindo que problemas antes inviáveis se tornassem aplicáveis em sala de aula. Terceiro, esses ganhos foram alcançados sem perda de seletividade — as implementações ineficientes mantiveram-se rejeitadas —, com um *overhead* computacional moderado, em torno de 15%, considerado aceitável em contextos educacionais e de pesquisa.

Em síntese, enquanto o modelo tradicional falha em capturar variações reais de desempenho, a calibração adaptativa assegura maior justiça avaliativa, preservando o rigor e ampliando a aplicabilidade pedagógica.

5.7 Discussão

5.7.1 Contribuições Científicas. O trabalho oferece três contribuições principais:

Quantificação inédita da injustiça algorítmica em juízes *online*.

Proposição de metodologia reprodutível de calibração adaptativa.

Estabelecimento de um *framework* de validação *dual* para manutenção do rigor avaliativo.

5.7.2 Implicações Práticas. **Para plataformas *online***: substituição de multiplicadores fixos por calibragem empírica específica por problema.

Para o ensino de algoritmos: ampliação do repertório de problemas aplicáveis, redução de frustração estudantil e maior equidade entre linguagens.

Além dessas implicações diretas, a análise também revelou um resultado complementar de interesse prático. Em classes simples (O(1), O($\log n$) e O(n)), Python apresentou desempenho superior a C++. Embora contraintuitivo, esse comportamento decorre do peso relativo do *overhead* fixo em execuções de curta duração. Esse efeito evidencia como a calibração empírica é capaz de revelar nuances de desempenho que não seriam antecipadas apenas pela análise assintótica, reforçando a utilidade do modelo adaptativo.

5.7.3 Limitações e Trabalhos Futuros. Apesar dos avanços, algumas limitações precisam ser reconhecidas:

O escopo experimental restringiu-se a C++ e Python; a generalização para outras linguagens exige investigação.

O ambiente de testes baseou-se em *containers* Docker; em *bare-metal*, os resultados podem divergir.

O modelo introduz um *overhead* computacional adicional de 15%, o que pode ser crítico em plataformas de larga escala.

Futuras pesquisas devem explorar a extensão para outras linguagens (Java, Go), validar em ambientes *bare-metal* e investigar técnicas de aprendizado de máquina para predição automática de β.

6 CONCLUSÃO

Este trabalho abordou o problema da inequidade entre linguagens de programação em juízes *online*, decorrente da adoção de tempos-limite fixos baseados em soluções de referência em C++. Demonstrou-se que tal prática penaliza linguagens interpretadas, como Python, restringindo o repertório pedagógico e resultando em vereditos incorretos em soluções corretas. Para enfrentar essa limitação, foi proposta e validada uma metodologia de calibração adaptativa, apoiada em *benchmarks* controlados em *containers* Docker, substituindo multiplicadores fixos por fatores empíricos de calibração (β) obtidos estatisticamente.

Os experimentos realizados, abrangendo diferentes classes de complexidade e problemas práticos do CSES, evidenciaram ampla variação dos fatores de calibração e mostraram que o modelo adaptativo reduz significativamente injustiças avaliativas, ampliando o repertório de problemas viáveis em Python sem comprometer a seletividade. A principal conclusão é que C++ e Python não podem ser avaliados de maneira equitativa sob políticas de tempo-limite fixo, sendo a calibração empírica indispensável para garantir justiça, sobretudo em contextos educacionais.

Como contribuições, destacam-se a proposição de uma metodologia sistemática para calibração automática de tempos-limite, a formulação do *Scale Dependency Principle* para *benchmarking* em ambientes containerizados e o desenvolvimento do *Dual-Validation Framework* para assegurar seletividade. Por tratar-se de um MVP, o estudo concentrou-se em C++ e Python, utilizou *containers* Docker para garantir reprodutibilidade e assumiu um *overhead* adicional. Essas delimitações foram definidas de maneira consciente e coerente com os objetivos do trabalho, constituindo uma base sólida para evoluções futuras.

Entre as perspectivas futuras, incluem-se a extensão da metodologia a outras linguagens, a validação em ambientes *bare-metal* e a aplicação em disciplinas reais, com integração do sistema em ambiente de produção educacional. Também merece destaque a exploração de técnicas de inteligência artificial, tanto para a predição automática de fatores de calibração — reduzindo o custo de execução de *benchmarks* — quanto para a tradução assistida de soluções de referência entre linguagens, acelerando o processo de configuração de novos problemas e ampliando a aplicabilidade prática do sistema.

Em síntese, este trabalho demonstra que a adoção de um Juiz de Código Adaptativo viabiliza avaliações mais justas e representativas de algoritmos, amplia o repertório pedagógico disponível e reafirma que a equidade, e não a uniformidade artificial entre linguagens, deve ser o princípio orientador da avaliação de algoritmos em contextos educacionais.

AGRADECIMENTOS

Agradeço de forma especial a todos que fizeram parte da minha jornada acadêmica. Em particular, expresso minha gratidão ao meu amigo Wendell, às amigas Ritinha, Lorenna e Thais, aos meus amigos João Henrique, Pedro Serey e Ezequias, e ao meu irmão. Tenho

certeza de que essa caminhada teria sido muito mais difícil sem o apoio, a amizade e o incentivo de vocês.

Estendo também meus agradecimentos a todos os professores que contribuíram para a minha formação ao longo do curso, pelo conhecimento transmitido, pelas orientações e pelo exemplo que levarei para minha vida profissional e pessoal.

ANEXO

A ARTEFATOS E REPRODUTIBILIDADE

Todos os códigos, dados e experimentos deste trabalho estão disponíveis em repositório público no GitHub:

- **Repositório:** github.com/VANDRESSAGALDINOS/adaptive-code-judge-tcc

Organização do Repositório

- `data/reference_solutions/` — soluções equivalentes em C++ e Python utilizadas como base nos benchmarks
- `docker/` — arquivos de configuração de ambientes containerizados (Dockerfiles, compose)
- `documentation/` — documentação técnica e relatórios experimentais
- `experiments/` — experimentos controlados por classes de complexidade
- `experiments_realworld/` — experimentos com problemas reais
- `scripts/` — utilitários de execução e automação
- `src/` — código-fonte do Juiz de Código Adaptativo
- `backup_scripts/` — scripts auxiliares de backup

Arquivos Principais

- `README.md` — descrição detalhada do projeto e instruções de uso.
- `QUICK_START.md` — guia rápido de execução dos experimentos.
- `PROJECT_SUMMARY.md` — resumo técnico do projeto.
- `requirements.txt` — dependências para replicação local.
- `run.sh` e `start_server.py` — scripts de inicialização.
- `criar_benchmarks.py`, `criar_submissions.py` — geração de benchmarks e submissões.

Resultados completos

Os resultados integrais (18 problemas, métricas por linguagem, fatores de calibração, vereditos sob TL_{trad} e TL_{adapt}) encontram-se em:

- `experiments/results/summary.csv`

Devido à extensão, a tabela completa não é reproduzida neste documento, mas permanece acessível no repositório oficial para garantir transparência e reprodutibilidade.

REFERÊNCIAS

[1] BEECROWD. *About Beecrowd Platform*. 2025. Disponível em: https://www.beecrowd.com.br/. Acesso em: 28 ago. 2025.

[2] CHEERS, F.; MAERTENS, R.; STRIJBOL, N.; MESUERE, B. Discovering and exploring cases of educational source code plagiarism with Dolos. *arXiv preprint* arXiv:2402.10853, 2024.

[3] CODEFORCES. *Extended Time Limits: Python and Other Languages*. 2019. Disponível em: https://codeforces.com/blog/entry/70233. Acesso em: 28 ago. 2025.

[4] HACKERRANK INC. *Time Limits in Coding Questions*. 2023. Disponível em: https://support.hackerrank.com/hc/en-us/articles/360049662994-Time-Limits-in-Coding-Questions. Acesso em: 28 ago. 2025.

[5] MERKEL, D. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal*, n. 239, 2014.

[6] WASIK, S.; ANTCZAK, M.; BADURA, J.; LASKOWSKI, A.; STERNAL, T. A survey on online judge systems and their applications. *ACM Computing Surveys*, v. 51, n. 1, 2018.

[7] PYTHON SOFTWARE FOUNDATION. *sys — System-specific parameters and functions*. Python Documentation, 2025. Disponível em: https://docs.python.org/3/library/sys.html. Acesso em: 1 set. 2025.

[8] SHARMA, R.; YADAV, R. Comparative Analysis of C and Python in Terms of Memory and Time. *ResearchGate*, 2020. Disponível em: https://www.researchgate.net/publication/347793255_Comparative_Analysis_of_C_and_Python_in_Terms_of_Memory_and_Time. Acesso em: 1 set. 2025.

[9] LIU, X.; WOO, T. Resource Management in Containerized Environments. *Journal of Systems and Software*, v. 165, p. 110–125, 2020.

[10] VJUDGE. *Virtual Judge*. 2025. Disponível em: https://vjudge.net/. Acesso em: 1 set. 2025.

[11] JAIN, R. *The Art of Computer Systems Performance Analysis*. New York: John Wiley & Sons, 1991.

[12] MCGILL, R.; TUKEY, J. W.; LARSEN, W. A. Variations of Boxplots. *The American Statistician*, v. 32, n. 1, p. 12–16, 1978.

[13] EFRON, B. Bootstrap Methods: Another Look at the Jackknife. *The Annals of Statistics*, v. 7, n. 1, p. 1–26, 1979.

[14] FELTER, W.; FERREIRA, A.; RAJAMONY, R.; RUBIO, J. An Updated Performance Comparison of Virtual Machines and Linux Containers. *IBM Research Technical Report*, 2015.

[15] CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. *Introduction to Algorithms*. 3. ed. Cambridge, MA: MIT Press, 2009.