UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA - CCET
DEPARTAMENTO DE COMPUTAÇÃO - DC

Pedro Vinícius Guandalini Vicente

NodeRock: Uma Abordagem para Seleção de Testes Suscetíveis a *Event Races* em Projetos Node.js

SÃO CARLOS - SP
2025

Pedro Vinícius Guandalini Vicente

NodeRock: Uma Abordagem para Seleção de Testes Suscetíveis a *Event Races* em Projetos Node.js

Trabalho de conclusão de curso apresentado ao Departamento de Computação da Universidade Federal de São Carlos, para obtenção do título de bacharel em Ciência da Computação.

Orientador: Prof. Dr. André Takeshi Endo

SÃO CARLOS - SP
2025

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia - CCET
Departamento de Computação

Comissão avaliadora

Membros da comissão examinadora que avaliou e aprovou a Defesa do Trabalho de Conclusão de Curso do candidato Pedro Vinícius Guandalini Vicente, realizada em 08/12/2025

Prof. Dr. André Takeshi Endo
Instituição: Universidade Federal de São Carlos

Prof. Dr. Rogério Galante Negri
Instituição: Universidade Estadual Paulista

Prof. Dr. Vinícius Humberto Serapilha Durelli
Instituição: Universidade Federal de São Carlos

Dedicatória

Dedico este trabalho aos meus pais, Valdevino e Elaine, minha avó Maria Elena, meu irmão João e todos os meus amigos e familiares que me apoiaram nessa difícil e longa caminhada.

AGRADECIMENTO

Sou eternamente grato pelo amor, sacrifício e confiança dos meus pais, Valdevino e Elaine, que sempre me incentivam e sempre estão presentes em cada passo que dou. Agradeço também à minha avó, Maria Elena, por suas orações e carinho, e ao meu irmão, João, por estar comigo em momentos tão desafiadores.

Também sou extremamente grato pela comunidade de fé e amizade que encontrei na Igreja Adventista Central de São Carlos. Obrigado por cada incentivo, cada oração e pela companhia nos Sábados. Esse suporte foi essencial.

Meu reconhecimento especial vai para meu orientador e amigo André Endo, que me guiou desde a IC até o TCC com muita paciência e dedicação. Agradeço sinceramente por todo o suporte, conselho e ajuda para desenvolver este e outros trabalhos!

Se depurar é o processo de remover bugs, então programar deve ser o processo de criá-los. - Edsger W. Dijkstra

RESUMO

A plataforma Node.js é amplamente utilizada para aplicações JavaScript, empregando um modelo de execução assíncrono que é essencial para o seu desempenho. Contudo, o não determinismo desse modelo pode levar a *event races*, que são *bugs* de concorrência sutis e difíceis de reproduzir que comprometem a confiabilidade do software. As abordagens atuais para detectar *event races* frequentemente exigem análises exaustivas em toda a suíte de testes, resultando em um alto custo computacional e na ausência de mecanismos eficazes para selecionar e priorizar testes automatizados que são mais suscetíveis a *event races*. Este trabalho propõe e avalia uma abordagem chamada `NodeRock` que usa análise dinâmica para identificar e selecionar testes em projetos Node.js suscetíveis a *event races*. `NodeRock` coleta traços de execução detalhados, extrai um conjunto de 15 *features* dinâmicas que caracterizam o comportamento assíncrono e utiliza aprendizado de máquina para selecionar os testes. Os resultados fornecem evidências de que a abordagem identifica testes suscetíveis a *event races* com acurácia de 75% e revocação superior a 84%, sugerindo que a análise de métricas dinâmicas é uma abordagem promissora para priorizar a investigação desses testes.

Palavras-chave: JavaScript, Node.js, Testes Automatizados, *Event Races*, Aprendizado de Máquina

ABSTRACT

The Node.js platform is widely used for JavaScript applications, employing an asynchronous execution model that is essential for its performance. However, the non-determinism of this model can lead to event races, which are subtle and hard-to-reproduce concurrency bugs that compromise software reliability. Current approaches to detect event races often require exhaustive analyses on the entire test suite, resulting in high computational costs and a lack of effective mechanisms to select and prioritize automated tests that are more susceptible to event races. This work proposes and evaluates an approach called `NodeRock`, which uses dynamic analysis to identify and select tests susceptible to event races in Node.js projects. `NodeRock` collects detailed execution traces, extracts a set of 15 dynamic features that characterize asynchronous behavior, and uses machine learning to select the tests. The results provide evidence that the approach identifies tests susceptible to event races with 75% accuracy and recall exceeding 84%, suggesting that the analysis of dynamic metrics is a promising strategy for prioritizing the investigation of these tests.

Keywords: JavaScript, Node.js, Automated Testing, Event Races, Machine Learning

Lista de Figuras

Lista de Tabelas

Conteúdo

1 INTRODUÇÃO

JavaScript é uma linguagem de programação dinâmica e multi-paradigma, originalmente desenvolvida para criar aplicações interativas e dinâmicas na *web*. Devido à sua flexibilidade e ampla adoção, JavaScript se consolidou como uma das linguagens de programação mais utilizadas por desenvolvedores em 2025, conforme diferentes pesquisas (TIOBE Software BV, 2025; Stack Exchange, Inc, 2025).

Com a introdução do ambiente de *runtime* Node.js[1] em 2009, o JavaScript tem ganhado espaço no desenvolvimento de aplicações em outros domínios. O Node.js apresenta uma arquitetura *single-threaded* e não-bloqueante, capaz de oferecer eficiência e escalabilidade para a construção de aplicações do lado do servidor, microsserviços, aplicações para *desktop* e ferramentas de linha de comando. Além disso, o Node.js é suportado por um amplo ecossistema de pacotes: O npm[2], que oferece mais de 2 milhões de *frameworks* e bibliotecas *open source* para milhões de desenvolvedores e diversas empresas globais, como NASA, PayPal e Uber (Chrzanowska, 2024).

Para que não ocorra o bloqueio da *thread* principal do Node.js, tarefas custosas e operações de Entrada/Saída são processadas no *background* da aplicação, através de *worker threads*. Entretanto, variações no tempo de execução dessas operações assíncronas pelos *worker threads* e manipuladores de eventos são causados por fatores não determinísticos. Com isso, ordens de execução de eventos não esperadas pelo programador podem surgir (Endo e Møller, 2020; Zhou et al., 2023; Endo e Møller, 2025). Assim, geram-se *event races*, que são condições de corrida geradas pela concorrência não determinística na ordem de execução de eventos.

A presença de *event races* pode ter consequências negativas, como *crashes* da aplicação, inconsistência de dados, testes *flaky*[3] e até vulnerabilidades de segurança. Para combater esse problema, a literatura propõe diversas abordagens de detecção, que se dividem principalmente em duas vertentes, sendo elas técnicas baseadas na modelagem de relações de *happens-before*, como NodeRacer (Endo e Møller, 2020), NRace (Chang et al., 2021) e NodeRT (Zhou et al., 2023), e métodos de injeção de *delays*, como o *Node.js Asynchronous Callback Delayer (NACD)* (Endo e Møller, 2025), para expor *event races* através de múltiplas reexecuções. Embora valiosas, essas abordagens frequentemente impõem um alto custo computacional, seja pela construção de modelos complexos ou pela necessidade de múltiplas reexecuções. Esse custo torna inviável a análise exaustiva de projetos de software modernos com centenas ou milhares de testes, evidenciando uma lacuna na literatura, a ausência de uma abordagem para selecionar e priorizar os testes suscetíveis a *event races*[4].

[1] <https://nodejs.org/>

[2] <https://www.npmjs.com/>

[3] Testes que passam e falham de forma não determinística (Luo et al., 2014).

[4] Testes classificados como suscetíveis a *event races* são aqueles que exibem padrões de comportamento

Para mitigar o alto custo associado à execução de suítes de teste[5] extensas, a engenharia de software de teste consolidou duas principais vertentes de otimização, a Priorização de Casos de Teste (*Test Case Prioritization* - TCP) e a Redução de Suíte de Teste (*Test Suite Reduction* - TSR). A TCP foca em reordenar a suíte para maximizar a taxa de detecção de falhas e acelerar o *feedback* aos desenvolvedores (Rothermel et al., 2001), enquanto a TSR busca diminuir o custo computacional do teste de software ao selecionar um subconjunto representativo de testes (Harrold, Gupta e Soffa, 1993). Contudo, apesar de serem eficazes em acelerar o *feedback* e reduzir custos de execução em um ambiente de desenvolvimento, não são adequadas para o problema de seleção de *bugs* de concorrência. A razão é que suas estratégias se baseiam em heurísticas tradicionais, tais como cobertura de código, histórico de falhas passadas ou análise de código recentemente modificado, que não tratam da natureza não determinística e intermitente dos *event races*.

Nesse contexto, a hipótese central deste trabalho é que os traços de execução assíncrona de um teste podem servir como um indicador para a presença de *event races*. Em vez de depender de análises complexas de diferentes possibilidades de ordenação de conclusão de eventos, a abordagem proposta investiga se métricas dinâmicas, como o uso de *callbacks*[6] e o ciclo de vida de *Promises*[7], podem ser usadas para identificar e priorizar um subconjunto de testes. Tal abordagem permitiria que os desenvolvedores concentrassem seus esforços de depuração apenas nos testes mais suscetíveis a *event races*, otimizando o processo de garantia de qualidade e tornando a detecção de *event races* mais prática e eficiente.

Portanto, o objetivo central deste trabalho é desenvolver e investigar a eficácia de uma abordagem que combina análise dinâmica com o uso de aprendizado de máquina para selecionar e priorizar testes em aplicações Node.js suscetíveis à ocorrência de *event races*. Com isso, busca-se oferecer uma abordagem que reduza o esforço de inspeção e complemente as técnicas de detecção existentes. Dessa forma, pretende-se verificar se a análise de métricas dinâmicas de execução assíncrona é eficaz para identificar testes automatizados suscetíveis a *event races*, permitindo direcionar investigações mais aprofundadas para tais testes.

Para concretizar esse objetivo, este trabalho propõe, desenvolve e avalia o `NodeRock`, uma abordagem que implementa essa abordagem de análise dinâmica com o uso de aprendizado de máquina. O `NodeRock` opera através da instrumentação do código-alvo para extrair um conjunto de 15 *features* que caracterizam o comportamento assíncrono, como

assíncrono e métricas dinâmicas similares às observadas em testes que conhecidamente possuem *event races*, sugerindo uma propensão à condição de corrida, mesmo na ausência de falhas imediatas.

[5]Coleção de casos de teste destinados a verificar o correto funcionamento de um sistema de software (Dominik Szahidewicz, 2025)

[6]Funções passadas como argumentos para outras funções para serem executadas após a conclusão de uma operação assíncrona ou a ocorrência de um evento específico (Ecma International, 2025; Mozilla Developer Network, 2025).

[7]Objetos que representam a eventual conclusão (ou falha) de uma operação assíncrona e seu valor resultante, permitindo o gerenciamento de fluxos assíncronos de forma mais estruturada que os *callbacks* (Ecma International, 2025; Mozilla Developer Network, 2025).

o ciclo de vida de *Promises* e a frequência de chamadas com *callbacks*.

Os resultados, aplicados em 24 projetos *open-source*, apontam para a eficácia desta abordagem. O modelo proposto foi capaz de identificar testes com *event races* conhecidos com valores de acurácia de 75% e revocação superior a 84%. Além disso, a análise sugere a eficácia das métricas coletadas, apontando que características como quantidade de funções invocadas com *callbacks* e grandes intervalos para conclusão de *callbacks* e *Promises* podem indicar a presença de *event races*. Assim, o `NodeRock` oferece uma abordagem prática para guiar a investigação dos desenvolvedores, selecionando e priorizando os testes suscetíveis a *event races*.

1.1 Estrutura do Trabalho

Este trabalho está dividido da seguinte forma: o Capítulo 2 discute a revisão bibliográfica, abordando os testes automatizados, *event races* e trabalhos relacionados; o Capítulo 3 detalha a metodologia utilizada para a condução do estudo; no Capítulo 4 são apresentados e analisados os resultados obtidos; e, finalmente, no Capítulo 5, são apresentadas as conclusões e sugestões para trabalhos futuros.

2 REVISÃO BIBLIOGRÁFICA

Neste capítulo, são discutidos os principais conceitos e abordagens discutidos ao longo deste estudo. Dessa forma, são apresentados temas fundamentais que envolvem o direcionamento deste projeto, incluindo testes automatizados e presença de *event races* em projetos JavaScript. A organização do capítulo segue a seguinte estrutura: na Seção 2.1, são explorados os conceitos fundamentais da definição e amplificação de testes automatizados; na Seção 2.2, o foco é em *event races*, apresentando um exemplo motivador e abordando suas características, causas e implicações; e a Seção 2.3 apresenta uma revisão dos principais trabalhos relacionados ao tema.

2.1 Testes Automatizados

Testes automatizados são amplamente utilizados no desenvolvimento de software, servindo como uma especificação executável que valida o comportamento de um programa de forma replicável. A ascensão das metodologias ágeis, com seu foco em desenvolvimento e integração contínua, tornou a manutenção de suítes de teste abrangentes uma prática essencial para prevenir regressões e garantir a qualidade do código (Danglot et al., 2019). Contudo, a consistência e confiabilidade desses testes são desafiadas por linguagens como o JavaScript, cuja natureza dinâmica e modelo de execução assíncrono criam um ambiente propenso a erros sutis e não determinísticos (Andreasen et al., 2017).

Para enfrentar tais desafios, a análise dinâmica se consolidou como uma abordagem eficaz, pois opera observando a execução de um programa em tempo real. Em vez de inferir o comportamento a partir do código-fonte, essa técnica instrumenta a aplicação para interceptar eventos de execução, como chamadas de função e operações assíncronas, e coletar dados detalhados. A instrumentação ocorre diretamente no ambiente de execução, com *frameworks* como o NodeProf[8], demonstrando a viabilidade de realizar essa análise de forma eficiente e com baixo *overhead*[9] de performance (Sun et al., 2018). Essa capacidade de observar o comportamento real do programa torna a análise dinâmica especialmente adequada para identificar problemas que emergem da complexa interação de eventos em plataformas como o Node.js.

Dentro deste cenário, a amplificação de testes surge como uma estratégia promissora, focada em aprimorar suítes de testes existentes para verificar propriedades não consideradas originalmente na construção do teste, como a detecção de *bugs* de concorrência. Uma forma de amplificação consiste em modificar a execução dos testes para expor o programa a cenários não convencionais, como diferentes ordenações de eventos ou falhas simuladas (Zhang e Elbaum, 2014).

[8] <https://www.dag.inf.usi.ch/software/nodeprof>

[9] Custo computacional adicional introduzido pelo processo de instrumentação e coleta de métricas dinâmicas.

2.2 *Event Races*

Aplicações orientadas a eventos, especialmente aquelas desenvolvidas em JavaScript, estão sujeitas a um tipo específico de condição de corrida conhecido como *event race* (Adamsen, Møller e Tip, 2017). Apesar de o JavaScript adotar um modelo de execução em *thread* única, sua natureza assíncrona introduz uma nova categoria de desafios. Um *event race* ocorre quando eventos assíncronos, cujo tempo de processamento é não determinístico, são executados em uma ordem diferente daquela esperada pelo desenvolvedor, resultando em comportamentos inesperados e potenciais falhas no sistema (Petrov et al., 2012; Wang et al., 2017; Endo e Møller, 2020).

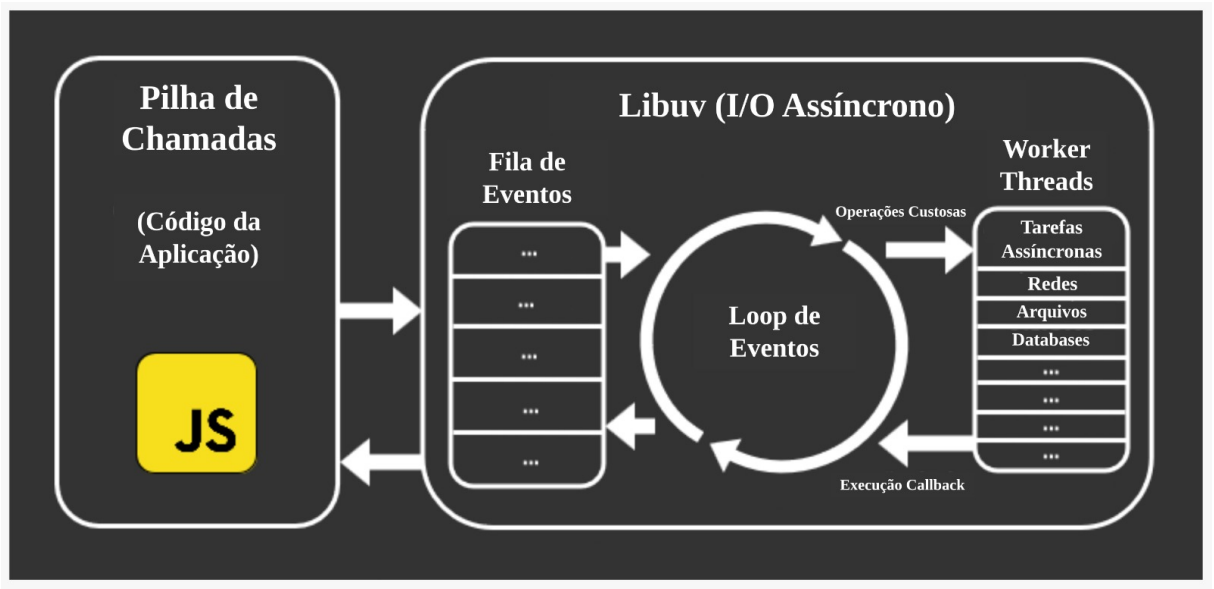
A origem desse problema está na forma como o ambiente de execução Node.js, gerencia operações custosas para não bloquear a *thread* principal. Tarefas como acesso a bancos de dados, manipulação de arquivos e comunicação via rede são delegadas a *worker threads* que operam em segundo plano *(background)* (OpenJS Foundation, 2025). No entanto, o tempo de conclusão dessas operações assíncronas pode variar devido a fatores não determinísticos, como latência de rede ou carga do sistema. Consequentemente, as funções de *callback*, que são executadas após a finalização dessas tarefas, podem ser enfileiradas e processadas em uma ordem não prevista, gerando *interleavings*[10] inesperados de eventos (Endo e Møller, 2020).

O mecanismo central que orquestra essa assincronicidade é o Loop de Eventos *(Event Loop)*, ilustrado na Figura 1. O diagrama detalha o ciclo de vida da execução de operações em JavaScript. O fluxo se inicia na Pilha de Chamadas *(Call Stack)*, onde o código da aplicação é executado. Ao encontrar operações bloqueantes ou custosas, o sistema delega o processamento para a biblioteca *libuv*[11]. Esta, por sua vez, utiliza *worker threads* para processar tais tarefas de E/S assíncrona em segundo plano, sem travar a execução principal. Assim que uma tarefa é concluída, sua função de retorno correspondente (*callback*) é inserida na Fila de Eventos *(Event Queue)*. O papel do Loop de Eventos é atuar como um monitor contínuo entre essas estruturas, pois ele aguarda até que a Pilha de Chamadas esteja totalmente vazia para, somente então, transferir o próximo evento da fila para a pilha, permitindo sua execução (OpenJS Foundation, 2025; Tuzcuoğlu, 2024). É nesse agendamento não determinístico, dependente de qual *worker thread* termina primeiro e da disponibilidade da pilha, que surgem as diferentes sequências de execução que caracterizam os *event races*.

[10] Diferentes sequências em que as operações assíncronas (como *callbacks* e resoluções de *promises*) são agendadas e executadas pelo *Event Loop*.

[11] <https://libuv.org/>

Figura 1: Etapas do funcionamento do *Event Loop* do Node.js.



Adaptado de (Tuzcuoğlu, 2024).

As consequências de um *event race* podem ser severas e difíceis de diagnosticar, pois os erros podem se manifestar de forma intermitente. A presença dessas condições de corrida pode prejudicar gravemente a funcionalidade de uma aplicação, levando a problemas como falhas abruptas (*crashes*), valores incorretos em memória, estados inconsistentes de banco de dados (Davis, Thekumparampil e Lee, 2017), testes *flaky* e até mesmo vulnerabilidades de segurança (Endo e Møller, 2020). Por isso, é fundamental que esses *interleavings* de eventos sejam explorados e tratados ainda em tempo de desenvolvimento, a fim de evitar que falhas críticas, difíceis de reproduzir e depurar, ocorram em ambiente de produção (Wang et al., 2017; Endo e Møller, 2020).

Para demonstrar na prática como os *event races* podem levar a falhas intermitentes e de difícil depuração em aplicações Node.js, será analisado um caso real detectado pela ferramenta NACD (Endo e Møller, 2025). O *event race* foi encontrado através de um teste presente na biblioteca `fs-extra`[12], que utiliza uma função da biblioteca `node-read-dir-files`[13] para realizar a cópia de conteúdos de arquivos.

O problema central se origina na função `readDirFiles` de `node-read-dir-files`, cuja implementação simplificada é apresentada na Figura 2. Essa função foi projetada para montar a estrutura e ler o conteúdo de múltiplos arquivos em um diretório de forma recursiva e assíncrona. A função recebe como parâmetros o `dirPath` com o *path* para o diretório que se deseja navegar e a função de *callback* que será executada ao final sobre a estrutura de arquivos encontrada. Em seguida, com os arquivos de dentro desse diretório (obtidos na Linha 3), a função itera sobre estes utilizando `async.forEach` (Linha 4) e,

[12] <https://github.com/jprichardson/node-fs-extra>
[13] <https://github.com/mmalecki/node-read-dir-files>

para cada entrada, dispara a operação assíncrona `fs.stat` (Linha 5) para verificar seu tipo. Se for um diretório, uma chamada recursiva a `readDirFiles` é realizada (Linha 7) e, se for um arquivo, a função `fs.readFile` é invocada para ler seu conteúdo (Linha 12). Em ambos os casos, o objeto compartilhado `result` é populado com a estrutura ou conteúdo obtidos (linha 8 e 13). E, ao final do `forEach`, a função passada como `callback` para o `readDirFiles` é executada sobre a estrutura de arquivos encontrada (Linha 18).

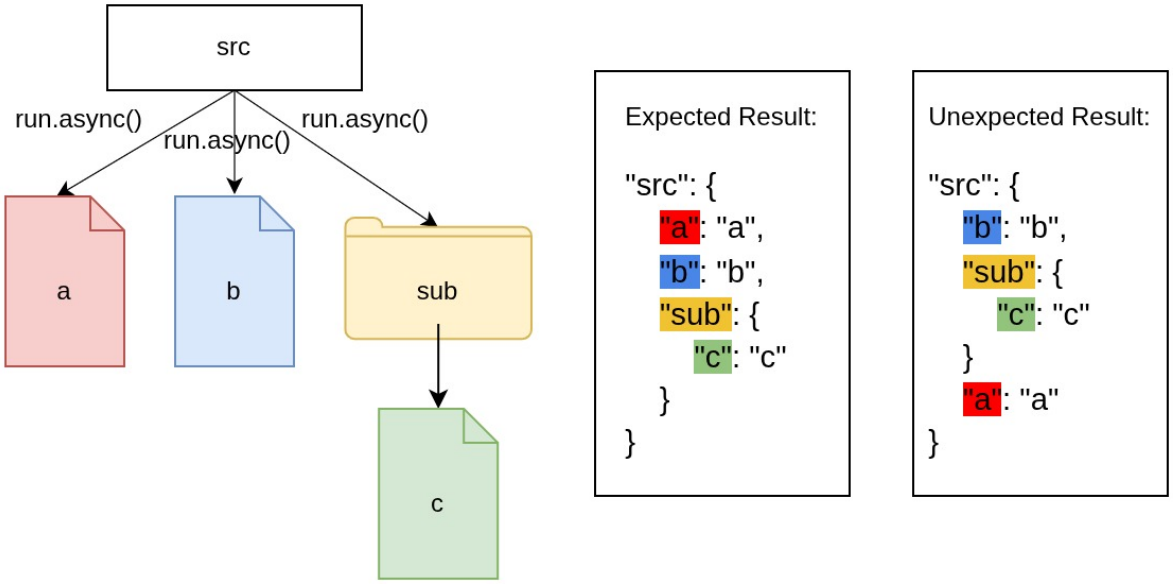
Figura 2: Versão simplificada da função `readDirFiles`, demonstrando o preenchimento assíncrono do objeto `result` que leva a um *event race*.

```
1  function readDirFiles(dirPath, callback) {
2    var result = {};
3    fs.readdir(dirPath, (files) => {
4      async.forEach(files, (file, next) => {
5        fs.stat(file, (stat) => {
6          if (stat.isDirectory()) {
7            readDirFiles(file, (subFiles) => {
8              result[file] = subFiles;
9              next();
10           });
11         } else {
12           fs.readFile(file, (content) => {
13             result[file] = content;
14             next();
15           });
16         }
17       });
18     }, () => callback(result));
19   }
20 }
```

Adaptado de:
<https://github.com/mmalecki/node-read-dir-files/blob/master/lib/read-dir-files.js>.

Como o `async.forEach` processa as entradas concorrentemente e a conclusão dessas operações é afetada por fatores externos (como o tempo de acesso ao disco e ocupação das *worker threads*), o fluxo de execução não segue uma ordem determinística. Assim, a ordem em que arquivos ou subdiretórios são adicionados ao objeto `result` pode variar entre as execuções. Por exemplo, em uma execução, um subdiretório "sub" pode ser processado e adicionado antes de um arquivo "a", enquanto em outra, o oposto pode ocorrer. Essa sutil variação na ordem dos dados, causada pelo comportamento não determinístico das chamadas assíncronas, caracteriza um *event race*. Esse comportamento é ilustrado na Figura 3 que demonstra a instabilidade analisada.

Figura 3: Ilustração de um *event race* causado pelo processamento assíncrono na função `readDirFiles`. O tratamento assíncrono pode levar a uma ordem de inserção não determinística no objeto `result`. A ordem esperada ("a", "b", "sub", "c") pode variar entre as execuções, como mostrado no resultado inesperado onde "sub" aparece antes de "a".



Fonte: Autoria própria.

O não determinismo na ordem das entradas do objeto retornado por `readDirFiles` torna-se problemática quando utilizada por um código que, inadvertidamente, depende dessa ordem. Um exemplo é o teste da biblioteca `fs-extra`, ilustrado na Figura 4. O teste aplica uma função `filter` (invocado na Linha 13) ao objeto `srcFiles` (obtido como `result` de `readDirFiles`) para remover arquivos cujos nomes terminam com "a". No entanto, a implementação original de `filter` continha uma falha lógica, nas Linhas 7-8, ao encontrar um subdiretório (`curFile instanceof Object`), a função realizava uma chamada recursiva para explorar os arquivos dentro desse subdiretório, mas retornava imediatamente (`return filter(curFile)`), fazendo com que os arquivos restantes no nível atual do `curFile` não sejam processados.

Figura 4: Versão simplificada do teste da função `readDirFiles`, responsável pelo comportamento *flaky*. O `return` prematuro faz com que arquivos sejam ignorados dependendo da ordem induzida pelo *event race*.

```
it('filters out files ending with "a"', () => {
  const src = path.join(__dirname, 'analyzedDirectory');
  readDirFiles(src, (srcFiles) => {
    function filter(files) {
      for (const fileName in files) {
        const curFile = files[fileName];
        if (curFile instanceof Object) // curFile is a subdirectory
          return filter(curFile); // DFS
        if (fileName.endsWith("a"))
          delete files[fileName];
      }
    }
    filter(srcFiles);

    expectedSrcFiles = {
      "src": {
        "b": "b",
        "sub": {
          "c": "c"
        }
      }
    }

    assert.deepStrictEqual(srcFiles, expectedSrcFiles);
  });
});
```

Adaptado de: <https://github.com/jprichardson/node-fs-extra/pull/737/files>.

A combinação do *event race* em `readDirFiles` com a falha na função `filter` resultou em um teste *flaky*. Se, devido ao *event race*, a função `readDirFiles` construísse o objeto `srcFiles` de tal forma que um subdiretório ("sub") aparecesse antes de um arquivo "a" no mesmo nível, a função `filter`, ao processar "sub", retornaria prematuramente, nunca avaliando e, portanto, nunca removendo o arquivo "a" subsequente. Isso levaria a uma falha na asserção `assert.deepStrictEqual(srcFiles, expectedSrcFiles)`, uma vez que `srcFiles` ainda conteria o arquivo "a". Por outro lado, se o arquivo "a" fosse processado antes do subdiretório, este seria corretamente removido e o teste passaria. Essa inconsistência nos resultados, dependente da ordem dos eventos assíncronos, é uma característica de um teste *flaky* induzido por um *event race* (Davis, Thekumparampil e Lee, 2017; Endo e Møller, 2020; Endo e Møller, 2025).

A solução para esse comportamento instável, proposta no *Pull Request*[14] que corrigiu o teste *flaky*, foi simples, a remoção da instrução `return` na chamada recursiva da função `filter`. Essa alteração sutil, modifica a lógica da função, deixando de ser uma busca

[14] <https://github.com/jprichardson/node-fs-extra/pull/737/files>

em profundidade que explora o primeiro subdiretório encontrado e termina a iteração no nível atual para permitir que o laço `for` continue a processar os demais arquivos e subdiretórios no mesmo nível, mesmo após a conclusão da chamada recursiva. Este caso exemplifica a natureza intermitente dos *event races*. O erro lógico na função `filter` só se manifesta como uma falha intermitente devido à ordem não determinística de processamento induzida pela função `readDirFiles`. Analisando cada função isoladamente, o defeito não é óbvio. É a interação entre a lógica falha e a concorrência assíncrona que cria uma condição de erro difícil de diagnosticar e reproduzir, reforçando a necessidade de ferramentas e abordagens especializadas para identificar tais vulnerabilidades que, a uma primeira vista, podem passar despercebidas.

2.3 Trabalhos Relacionados

O trabalho estabelecido por Lu et al. (2008) foi fundamental na classificação de *bugs* de concorrência, fornecendo uma análise empírica em larga escala de *bugs* reais em sistemas C/C++ com múltiplas *threads* e criou uma taxonomia seminal de violações de atomicidade e ordem. Baseando-se neste estudo, Wang et al. (2017) aplicaram uma metodologia semelhante para o Node.js, confirmando que esta taxonomia permanece como um ponto de referência para ambientes orientados a eventos. Outro trabalho inicial de Davis, Thekumparampil e Lee (2017) também estudou problemas de concorrência no Node.js para informar o design de sua ferramenta de *fuzzing*, técnica que, neste contexto, aplica aleatoriedade ao agendamento de eventos para forçar *event races*, destacando o impacto prático dessas *race conditions*.

Após os estudos empíricos iniciais, diversos estudos se concentraram na criação de ferramentas para detectar automaticamente *event races* no Node.js, se dividindo principalmente em duas vertentes. A primeira utiliza relações de *happens-before* para modelar a causalidade entre eventos. Ferramentas como NodeRacer (Endo e Møller, 2020), NRace (Chang et al., 2021) e NodeRT (Zhou et al., 2023) exemplificam essa abordagem. O NodeRacer explora ativamente novos *interleavings* de eventos guiado por um grafo de relações *happens-before* para guiar o adiamento seletivo de *callbacks*, enquanto o NRace foca em construir grafos de relações *happens-before* mais precisos, que levam em consideração as filas de eventos de múltiplas prioridades do Node.js. Essa maior precisão permite a detecção de *event races* sem a necessidade de reexecuções, analisando um único traço de execução. O NodeRT, por sua vez, otimiza essa técnica utilizando uma representação hierárquica denominada *asynchronous call tree* (ACTree), visando reduzir o *overhead* computacional que limita a escalabilidade de seus predecessores. A principal desvantagem dessas técnicas reside no custo de construir e analisar modelos de execução, ou na dependência de uma fase de observação inicial.

A segunda vertente, de caráter exploratório, utiliza técnicas de injeção de *delays* para

expor condições de corrida. A ferramenta NACD (Endo e Møller, 2025) injeta atrasos aleatórios diretamente nas operações assíncronas do *runtime* do Node.js, dispensando a modelagem do grafo de relações *happens-before*. Sua vantagem reside na simplicidade e eficácia em expor *event races* que dependem de atrasos específicos. Contudo, sua natureza aleatória não garante a descoberta de *event races* e pode exigir um número elevado de execuções para acionar uma falha. Em comum, todas essas ferramentas de detecção focam em encontrar o *event race* ou provocar falhas devido a sua presença, impondo custos que podem ser proibitivos em suítes de testes de larga escala.

Testes automatizados, essenciais para a detecção de *event races*, podem representar um desafio devido aos elevados custos de execução, especialmente em suítes de teste extensas (Endo e Møller, 2025). Para mitigar esse problema, a pesquisa em teste de software desenvolveu duas vertentes principais, a Priorização de Casos de Teste e a Redução de Suíte de Teste. A TCP tem como objetivo reordenar a suíte de testes para maximizar uma meta de desempenho, como a taxa de detecção de falhas. A ideia central, explorada em trabalhos seminais como os de Rothermel et al. (2001) e Elbaum, Malishevsky e Rothermel (2002), é executar primeiro os testes com maior probabilidade de encontrar defeitos, proporcionando um *feedback* mais rápido aos desenvolvedores. As técnicas de TCP geralmente se baseiam em informações coletadas de execuções anteriores, como cobertura de código (instruções, *branches*) ou métricas de propensão a falhas. A grande vantagem da TCP é que nenhum teste é descartado, preservando-se integralmente a capacidade de detecção de falhas da suíte original. A desvantagem, no entanto, é que o custo total de execução da suíte completa permanece o mesmo.

Por outro lado, a TSR busca diminuir o custo geral da regressão ao selecionar um subconjunto representativo que elimina testes redundantes ou obsoletos. O objetivo, como proposto por Harrold, Gupta e Soffa (1993), é criar uma suíte menor que, idealmente, preserve a capacidade original de detecção de falhas. As abordagens de TSR frequentemente utilizam heurísticas para selecionar um conjunto mínimo de testes que satisfaçam um determinado critério de cobertura (e.g., cobrir todas as instruções executadas pela suíte original). A principal vantagem da TSR é a economia direta no custo de execução, pois menos testes são executados. No entanto, essa abordagem carrega um risco inerente, ao remover testes considerados redundantes com base em um critério específico (como cobertura de código), pode-se inadvertidamente eliminar o único teste capaz de revelar um defeito específico, comprometendo a eficácia da suíte. Trabalhos mais recentes, como o de Cruciani et al. (2019), exploram técnicas de agrupamento com base em *keywords* dos testes para realizar a redução em larga escala, buscando um equilíbrio entre eficiência e a perda de capacidade de detecção.

A aplicação de técnicas de Aprendizado de Máquina (*Machine Learning* - ML) em teste de software evoluiu por meio de diversas abordagens para otimizar a seleção e priorização de testes. Em uma frente, o aprendizado não supervisionado foi utilizado para agrupar

testes com base em seus perfis de execução. Chen et al. (2011), por exemplo, aplicaram uma variante semissupervisionada do algoritmo K-Means para incorporar resultados de execuções anteriores de testes para "supervisionar" o agrupamento de testes com comportamentos semelhantes com base em um vetor binário de chamada de funções para indicar quais função foram executadas por cada teste. O aprendizado supervisionado também foi utilizado na literatura para automatizar a priorização, utilizando modelos treinados para ordenar a suíte de testes com base na probabilidade de falha. Busjaeger e Xie (2016) demonstraram a aplicabilidade dessa abordagem ao integrar métricas como cobertura de código, contagem de *keywords* e histórico de falhas em um modelo de Máquina de vetores de suporte (*Support-vector machine* - SVM).

Apesar de seu sucesso, as abordagens supervisionadas e não supervisionadas mais exploradas no contexto de testes de software tradicionalmente operam em um regime de aprendizado em lote *(batch learning)*. Conforme discutido por Bagherzadeh, Kahani e Briand (2022), isso significa que o modelo é treinado sobre um conjunto de dados estático e precisa ser reconstruído do zero para incorporar novos dados, o que pode ser problemático para alguns modelos em ambientes de Integração Contínua, onde o software e suas suítes de teste são atualizados continuamente. Como uma das maneiras para superar essa limitação, pesquisas recentes voltaram-se para o Aprendizado por Reforço. O mesmo trabalho mostra que tais modelos podem se adaptar aos resultados de cada ciclo de *build*. Utilizando um conjunto de *features* tradicionais da Engenharia de Software, como Complexidade Ciclomática, Linhas de Código (LoC) e idade do teste, essa abordagem resolve o problema da constante desatualização do modelo, alcançando um desempenho superior a modelos supervisionados.

A literatura apresenta um conjunto extenso de ferramentas para a detecção de *event races* e técnicas de otimização de testes. No entanto, as ferramentas de detecção são computacionalmente caras para uso exaustivo, enquanto as técnicas de otimização (TCP/TSR) e as abordagens de ML existentes não são especializadas para o problema de concorrência em ambientes assíncronos em JavaScript. A abordagem apresentada neste trabalho visa preencher essa lacuna ao integrar essas áreas. É empregada análise dinâmica e ML para criar uma forma especializada de seleção e priorização de testes, focada em identificar testes suscetíveis a *event races*. Portanto, tal abordagem não atua como um substituto para as ferramentas existentes, mas sim como um complemento que direciona os esforços de depuração, tornando a detecção de *bugs* de concorrência mais prática e eficiente em suítes de testes de larga escala.

2.4 Considerações Finais

A literatura apresenta um conjunto extenso de ferramentas e técnicas para a detecção de *event races*. No entanto, essas soluções frequentemente impõem um alto custo com-

putacional ou de análise manual, tornando inviável sua aplicação exaustiva em grandes suítes de teste. Identifica-se, portanto, uma lacuna, a ausência de uma abordagem focada na priorização e seleção de testes suscetíveis a *event races*. É para preencher essa lacuna que este trabalho propõe uma nova abordagem baseada em análise dinâmica e ML, materializada na abordagem `NodeRock` (introduzida na Subseção 3.1), que visa selecionar e priorizar os esforços de depuração para os testes mais suscetíveis a *event races*.

3 METODOLOGIA DO ESTUDO

Este capítulo apresenta a metodologia empregada na realização do estudo. Para guiar o desenvolvimento e a avaliação de uma abordagem baseada em ML para a seleção de testes suscetíveis a *event races*, foram formuladas três Questões de Pesquisa (QP). Elas direcionam a exploração e a validação da abordagem proposta.

- **QP1:** Qual a eficácia de um conjunto de modelos de ML (supervisionados e semis-supervisionados) para a seleção de testes suscetíveis a *event races*?

  O propósito desta QP é verificar se a abordagem de seleção proposta consegue selecionar os testes que possuem *event races* com valores aceitáveis em métricas como acurácia, precisão e revocação.

- **QP2:** Qual a capacidade individual das diferentes características dinâmicas em distinguir testes com e sem *event races*?

  A partir desta QP busca-se avaliar a eficácia individual das métricas dinâmicas coletadas. O objetivo é quantificar a relevância de cada característica na distinção entre testes com e sem *event races* através de uma análise de Ganho de Informação.

- **QP3:** Qual o desempenho (em tempo de execução) da abordagem?

  Esta QP visa avaliar a viabilidade prática da abordagem em um cenário de detecção de *bugs* em larga escala. A investigação mede o *trade-off* entre o *overhead* introduzido pela análise dinâmica e a economia de tempo obtida ao executar uma ferramenta de detecção NACD em apenas um subconjunto priorizado de testes.

3.1 Implementação e Funcionamento da Abordagem

Para investigar as QPs mencionadas anteriormente, foi desenvolvida a abordagem de análise dinâmica denominada `NodeRock`. O `NodeRock` implementa a proposta de seleção e priorização de testes baseada em ML, permitindo uma avaliação empírica de sua eficácia e viabilidade.

O `NodeRock` foi projetado para auxiliar desenvolvedores a identificar testes suscetíveis a *event races* em aplicações Node.js. Sua finalidade não é a detecção determinística de *bugs*, mas sim a seleção e priorização, permitindo que as equipes de desenvolvimento otimizem seus esforços de depuração ao focar em testes mais suscetíveis a *event races*. Para isso, o `NodeRock` opera através de uma sequência de passos que instrumenta a execução de testes, extrai valores de métricas dinâmicas e, por fim, utiliza ML para classificar os testes mais suspeitos.

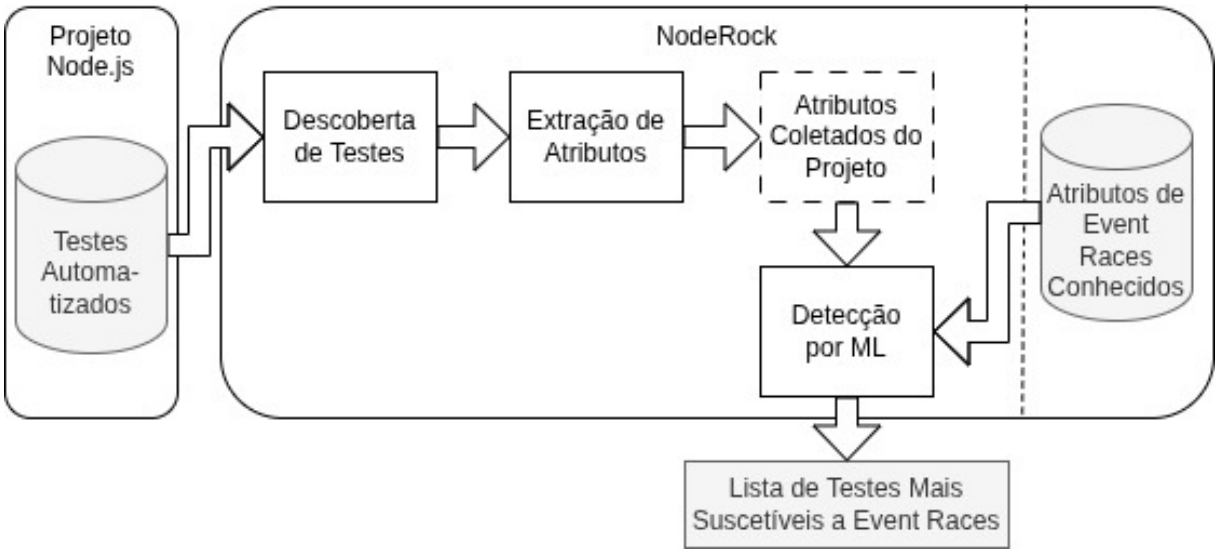
A implementação do `NodeRock` se apoia em *frameworks* consolidados para trazer estabilidade para as análises. A capacidade central de instrumentação é provida pelo NodeRT

(Zhou et al., 2023), que, por sua vez, depende do NodeProf (Sun et al., 2018) para realizar a análise dinâmica de código JavaScript sobre a máquina virtual GraalVM. Essa arquitetura permite que o `NodeRock` intercepte e registre eventos de execução sem a necessidade de modificar manualmente o código-fonte da aplicação ou os componentes internos do Node.js, sendo compatível com a versão 14 do *runtime*.

Para garantir sua aplicabilidade em projetos reais, o `NodeRock` foi projetado para se integrar com as principais infraestruturas de teste do ecossistema, sendo capaz de descobrir e executar automaticamente suítes de teste desenvolvidas com os *frameworks* de teste Mocha e Jest. Enquanto a instrumentação opera no ambiente JavaScript, as etapas subsequentes de processamento de dados e ML foram implementadas em Python 3, utilizando bibliotecas como `pandas` (McKinney, 2010) para manipulação de dados e `Scikit-learn` (Pedregosa et al., 2011) para normalização e aplicação de modelos.

A sequência de passos da execução do `NodeRock`, que implementa a abordagem proposta, é composto por três fases principais, conforme ilustrado na Figura 5: *(i)* Descoberta de Testes, *(ii)* Extração de Atributos e *(iii)* Detecção por ML. A execução inicia com a descoberta dos testes do projeto-alvo, prossegue para a execução instrumentada para extrair valores de métricas dinâmicas que caracterizam seu comportamento assíncrono e, por fim, utiliza um detector baseado em ML para sinalizar testes suscetíveis a *event races*.

Figura 5: Representação das fases de execução do `NodeRock` para a realização de uma análise.



Fonte: Autoria Própria.

A primeira fase, intitulada de "Descoberta de Testes", é responsável por identificar e preparar a suíte de testes automatizados do projeto Node.js alvo. O `NodeRock` se integra com *frameworks* de teste populares, como Mocha e Jest, para coletar automaticamente os nomes e endereços de arquivo de todos os casos de teste que passam com sucesso. Este

componente garante que apenas testes válidos e executáveis, que podem conter condições de corrida apesar de passarem em determinadas execuções, sejam considerados para as próximas fases de análise dinâmica.

A "Extração de Atributos" é onde ocorre a coleta dinâmica de dados e engenharia de *features* do `NodeRock`. Para cada teste identificado nesta etapa, é gerado um conjunto de 15 variáveis que quantificam o comportamento do teste executado e fornecem informações relevantes para identificar a possível presença de *event races*. O processo se inicia com uma primeira execução do teste em um ambiente instrumentado. Essa execução captura um traço geral, incluindo chamadas de função, tempos de execução e informações de contexto assíncrono do módulo *async hooks*[15].

Em seguida, realiza-se uma segunda execução separada do mesmo teste. Desta vez, o foco é coletar informações detalhadas sobre o comportamento de *Promises*, o que é alcançado através da aplicação de *monkey patching* no objeto *Promise* global. Essa abordagem direcionada permite interceptar e registrar informações do ciclo de vida de cada *Promise*, como o tempo gasto para sua resolução ou rejeição e seu estado final, que não estão disponíveis apenas no traço geral.

Com os dados brutos de ambas as execuções coletados, a abordagem combina os traços coletados e os registros das *Promises*, formando o conjunto de 15 *features* quantitativas (detalhadas na Subseção 3.3). Para garantir uma comparação justa entre projetos de diferentes escalas, o vetor de *features* resultante é normalizado por projeto. O resultado final desta fase é um conjunto de *features* padronizadas que representam numericamente o perfil assíncrono de cada teste.

A fase final utiliza ML para classificar quais testes são suscetíveis a *event races* ao compará-los com os atributos de um conjunto de testes automatizados que já conhecidamente possuem *event races*, conforme detalhado na Subseção 3.2. Para esta tarefa, foi escolhido o modelo que obteve os melhores resultados de previsão entre os dados conhecidos, conforme apresentado pelos resultados da QP1, na Subseção 4.2. Com isso, o algoritmo de *Positive and Unlabeled Learning (PU Learning)*[16] foi escolhido, um algoritmo de classificação para situações em que existam dados conhecidamente positivos e dados desconhecidos (Elkan e Noto, 2008).

Dessa forma, outro fator relevante que justifica a decisão de empregar o modelo de *PU Learning* para a tarefa de classificação de *event races* reside na natureza do conjunto de dados. Em vez de um problema de classificação binária tradicional (possui *event race* ou não possui *event race*), enfrenta-se um cenário onde apenas a classe positiva (testes com *event races* confirmados) é conhecida com certeza. A classe negativa não garante a ausência de *event races*, mas sim uma amostra em que um possível *event race* não foi encontrado e, portanto, *unlabeled*, podendo conter tanto negativos verdadeiros quanto

[15] <https://nodejs.org/api/async_hooks.html>
[16] <https://pulearn.github.io/pulearn/doc/pulearn/elkanoto.html>

positivos ainda não descobertos. O *PU Learning* é uma metodologia apropriada para essa configuração, pois permite treinar um classificador mesmo com a ausência de rótulos negativos explícitos (Elkan e Noto, 2008).

Assim, o modelo de *PU Learning* é treinado utilizando o *dataset* conhecido. O conjunto de testes com *event races* conhecidos serve como a classe positiva, enquanto todos os demais testes do *dataset* formam o conjunto não rotulado. O algoritmo de *PU Learning* utilizado gerencia a incerteza dos dados não rotulados (tratando-os como uma mistura de negativos e positivos), enquanto emprega o *Random Forest* como classificador base para efetivamente traçar as fronteiras de decisão e identificar padrões. Uma vez treinado, o classificador é aplicado a cada teste não rotulado do projeto analisado cujas *features* foram extraídas, atribuindo-lhe uma pontuação que representa a probabilidade de pertencer à classe positiva.

Consequentemente, os testes do projeto-alvo que são classificados como positivos pelo modelo (acima do limiar de confiança) são sinalizados como suscetíveis a *event races*, criando uma lista priorizada para investigações mais detalhadas. Em vez de analisar a suíte de testes inteira, os desenvolvedores podem focar seus esforços nesse subconjunto de testes suscetíveis a *event races*. Esta abordagem de seleção direcionada filtra e prioriza a análise, otimizando o processo de depuração e aumentando a probabilidade de encontrar *bugs* de concorrência de forma mais rápida, conforme verificado pela Subseção 4.4.

3.2 Seleção de Projetos

Para a construção de um *dataset* representativo, utilizado tanto no treinamento dos modelos quanto nas validações internas do NodeRock, foram agregados projetos de código aberto e disponibilizados por diferentes estudos na área de detecção de *event races* em aplicações Node.js. Especificamente, a coleta baseou-se nos *benchmarks* e projetos analisados em estudos anteriores (Endo e Møller, 2020; Zhou et al., 2023; Endo e Møller, 2025), garantindo assim um conjunto de dados diversificado e representativo de projetos encontrados na literatura.

O processo de seleção envolveu uma etapa de curadoria para buscar a consistência e a aplicabilidade dos dados. Inicialmente, foram removidos projetos duplicados que apareciam em mais de uma fonte. Em seguida, foram descartados os projetos que não puderam ser *buildados* com sucesso no ambiente de testes ou que apresentaram incompatibilidade com o *framework* de coleta de métricas dinâmicas do `NodeRock`, o que impediria a extração das *features* necessárias.

Após a fase de filtragem, o *dataset* final foi consolidado, totalizando 24 projetos únicos, que compreendem 1720 testes automatizados e 32 *event races* conhecidos e reportados. Uma visão detalhada dos projetos selecionados, juntamente com o número de testes e *event races* correspondentes, é apresentada na Tabela 1.

Tabela 1: Projetos do *dataset*.

Projeto	Total de Testes no Projeto	Testes que Conhecidamente Possuem *Event Races*
Mongo-express	33	4
Nedb	330	2
Node-archiver	35	1
Objection.js	48	1
Agent Keep Alive	24	1
fiware-pep-steelskin	210	1
WhiteboxGhost	-	1
node-mkdirp	-	1
node-logger-file-1	10	1
socket.io-1862	26	1
del	10	1
linter-stylint	-	1
Node-simpleCrawler	81	1
xlsx-extract	14	1
bluebird-2	-	1
nodesamples (express-user)	-	1
get-port	-	1
live-server-potential-race	-	1
socket.io-client	59	1
json-file-store	27	1
json-fs-store	7	1
ncp	15	4
write	12	1
Fs-extra	719	2
Total	**1720**	**32**

Fonte: Autoria própria.

É importante notar que alguns projetos listados na tabela podem apresentar um número de testes inferior ao que possuíam na versão original quando o *event race* foi reportado. Isso ocorre pois, em determinados casos, o artigo de origem disponibilizou apenas a suíte de testes relevante para a reprodução do *bug*, e não o conjunto completo. Adicionalmente, os projetos nos quais o valor na coluna "Testes que Conhecidamente Possuem *Event Races*" está marcada com um hífen (-) indicam que a fonte original forneceu um *script* de teste isolado para demonstrar o *event race*, em vez da suíte de testes completa do projeto.

3.3 Processo de Extração de Métricas dos Testes

Para caracterizar quantitativamente o comportamento de cada teste, foi implementado um processo de extração de métricas dinâmicas, resultando em um conjunto de 15

features únicas para cada execução. Além de métricas de execução frequentemente exploradas, como a duração total do teste ou a quantidade de funções invocadas, o `NodeRock` também foca na coleta de indicadores específicos do comportamento assíncrono. Essa abordagem permite uma análise mais aprofundada, quantificando métricas referentes ao uso de funções assíncronas, tempos de atraso (*delays*) de *callbacks* e o ciclo de vida e estado de *Promises*.

O processo de coleta mencionado na etapa "Extração de Métricas" é executado em duas fases distintas e sequenciais para garantir a abrangência dos dados. A primeira fase "Análise de Traço de Execução (via *Hooks*)" utiliza a instrumentação provida pelo NodeProf para gerar um traço de execução detalhado. A partir dos *logs* gerados por seus *hooks*, são extraídas métricas que descrevem o comportamento geral do teste, como a contagem de invocações de função e o rastreamento de operações assíncronas via *async hooks*. No entanto, essa abordagem não captura detalhes internos do ciclo de vida das *Promises*, como seus tempos de resolução ou rejeição.

Para suprir essa lacuna, a segunda fase da coleta "Análise de Promises (via *Monkey Patching*)" emprega a técnica de *monkey patching*[17] para modificar o objeto global *Promise* em tempo de execução. Essa abordagem permite interceptar a criação e a finalização de cada *Promise*, registrando informações que não são diretamente acessíveis pela instrumentação do NodeProf. Outra vantagem da separação em duas etapas de coleta em execuções diferentes é que como o *monkey patching* altera o comportamento nativo do código, executá-lo de forma isolada evita que a introdução de artefatos que interferem nas métricas da primeira fase. Dessa forma, minimiza-se a interferência entre as técnicas, permitindo que cada conjunto de *features* seja obtido em um ambiente mais controlado, o que favorece a integridade dos dados coletados.

O conjunto completo das 15 métricas base, separadas com base na etapa de coleta, juntamente com seus respectivos nomes no *dataset*, descrições e origem do *hook* que inspirou a métrica, está apresentado na Tabela 2.

[17] Técnica utilizada para estender ou modificar o comportamento de componentes de software em tempo de execução, sem a necessidade de alterar o código original (BrowserStack, 2025).

Tabela 2: Métricas extraídas.

Nome da Métrica	Descrição da Métrica	Origem da Métrica
Fase: Análise de Traço de Execução (via Hooks)		
`InvokeFunPre_Count`	Número de funções invocadas durante a execução.	(Sun et al., 2018).
`Invokes_with_callback`	Número de invocações de função que utilizam *callbacks*.	(Sun et al., 2018).
`Cb_Delays_Greater_Than_100_ms`	Número de *callbacks* cujo tempo de espera (*delay*) para iniciar a execução excedeu 100 milissegundos.	(Sun et al., 2018).
`Cbs_Total_delay_ms`	Soma total, em milissegundos, dos tempos de espera (*delay*) de todos os *callbacks*.	(Sun et al., 2018).
`InvokesInterval_Greater_Than_100_ms`	Número de invocações de função cuja duração de execução excedeu 100 milissegundos.	(Sun et al., 2018).
`Async_Function_Count`	Número de funções declaradas com a palavra-chave *async*.	(Zhou et al., 2023).
`Await_Count`	Número de expressões *await* utilizadas no código.	(Zhou et al., 2023).
`await_Intervals`	Intervalo de tempo, em milissegundos, entre a chamada de uma expressão *await* e o momento em que a execução é retomada.	(Zhou et al., 2023).
`Unique_Asynchook_ids`	Número de identificadores únicos de *Async Hooks* usados para rastrear operações assíncronas.	(Zhou et al., 2023).
`Total_duration_s`	Duração total da execução do teste, expressa em segundos.	(Zhou et al., 2023).
Fase: Análise de *Promises* (via *Monkey Patching*)		
`Total_Settled_Promises`	Número de *promises* que foram finalizadas (*settled*), seja com sucesso (*resolved*) ou com erro (*rejected*).	Autoria própria.
`avg_Resolved`	Tempo médio de resolução, em milissegundos, para *promises* que foram resolvidas (*resolved*).	Autoria própria.
`avg_Rejected`	Tempo médio de rejeição, em milissegundos, para *promises* que falharam (*rejected*).	Autoria própria.
`longest_Resolved`	Tempo, em milissegundos, da *promise* mais lenta a ser resolvida com sucesso.	Autoria própria.
`resolved_Percentage`	Percentual de *promises* resolvidas com sucesso em relação ao total de *promises* finalizadas.	Autoria própria.

Fonte: Autoria própria.

Um aspecto fundamental deste processo é que, para cada uma das 15 métricas, são geradas duas versões: um valor bruto ("Raw") e um valor normalizado ("Normalized"). Isso resulta em um vetor final de 30 *features* por teste. A normalização *MinMax*[18] é realizada por projeto, ou seja, os valores são escalados com base apenas nos dados dos testes pertencentes àquele mesmo projeto. Essa abordagem garante que as características de testes pertencentes a projetos menores, cujos componentes naturalmente geram valores de métricas mais baixos, não sejam ofuscadas pela magnitude de dados de projetos de diferentes escopos e cujos componentes testados são maiores. Dessa forma, a análise se torna mais sensível às variações relativas de comportamento dentro do contexto de cada aplicação, em vez de ser dominada por diferenças absolutas de escala entre projetos.

Outro aspecto a se considerar é que o valor de 100 ms utilizado nas métricas `Cb_Delays_Greater_Than _100_ms` e `InvokesInterval_Greater _Than_100_ms` foi adotado como um valor heurístico de referência. A escolha visa filtrar operações triviais e focar nas janelas temporais mais significativas, que representam intervalos ou atrasos que aumentam a exposição do fluxo de execução a variações no agendamento de eventos concorrentes, ampliando a probabilidade de ocorrência de *event races*.Embora a otimização fina deste parâmetro não tenha sido o escopo deste estudo, o valor mostrou-se suficiente para capturar variações temporais relevantes no comportamento assíncrono, conforme apresentado na Subseção 4.3.

3.4 Ameaças à Validade

A validade dos resultados deste estudo está sujeita a algumas ameaças que merecem ser destacadas. A principal ameaça à validade externa (generalização) reside na composição do *dataset* estudado. Embora a seleção de projetos de *benchmarks* acadêmicos consolidados (Endo e Møller, 2020; Zhou et al., 2023; Endo e Møller, 2025) busque assegurar a relevância dos *event races* analisados, esses projetos podem não ser inteiramente representativos da vasta gama de aplicações Node.js encontradas na indústria. Consequentemente, o modelo treinado com base nesses dados pode apresentar um desempenho diferente quando aplicado a projetos com características ou domínios distintos. Adicionalmente, a implementação da abordagem `NodeRock` possui dependências tecnológicas específicas, como a compatibilidade com a versão 14 do Node.js e com os *frameworks* de teste Mocha e Jest, o que limita sua aplicação direta a projetos que não atendam a esses requisitos.

Outras ameaças estão relacionadas à escolha das 15 métricas dinâmicas, que embora fundamentadas na análise do comportamento assíncrono, as métricas representam um subconjunto de todas as características possíveis de um teste. É possível que outras métricas, não exploradas neste trabalho, possam ter uma eficácia superior para distinguir

[18] <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html>

testes com e sem *event races*. Além disso, o processo de instrumentação dinâmica para a extração de métricas introduz um *overhead* na execução dos testes. Embora a coleta seja separada em duas fases para mitigar a interferência, essa instrumentação pode, teoricamente, alterar sutilmente o comportamento temporal do código, influenciando as condições que levam a um *event race*. Por fim, a eficácia do classificador está intrinsecamente ligada ao número de exemplos positivos conhecidos (32 *event races*), que, embora significativo para um estudo desta natureza, é relativamente pequeno para os padrões de ML, o que pode impactar a robustez do modelo treinado.

3.5 Artefatos Gerados e Informações Adicionais

Para permitir a reprodutibilidade e análise dos resultados apresentados neste trabalho, todos os artefatos gerados estão disponibilizados publicamente. O repositório contém o código-fonte completo do `NodeRock`, juntamente com instruções de seu uso, além de *scripts* em Python utilizados para conduzir as análises das Questões de Pesquisa e o código para gerar as figuras das QPs deste documento. O material completo pode ser acessado no seguinte repositório GitHub:

<https://github.com/PedroViniciusVicente/NodeRock>

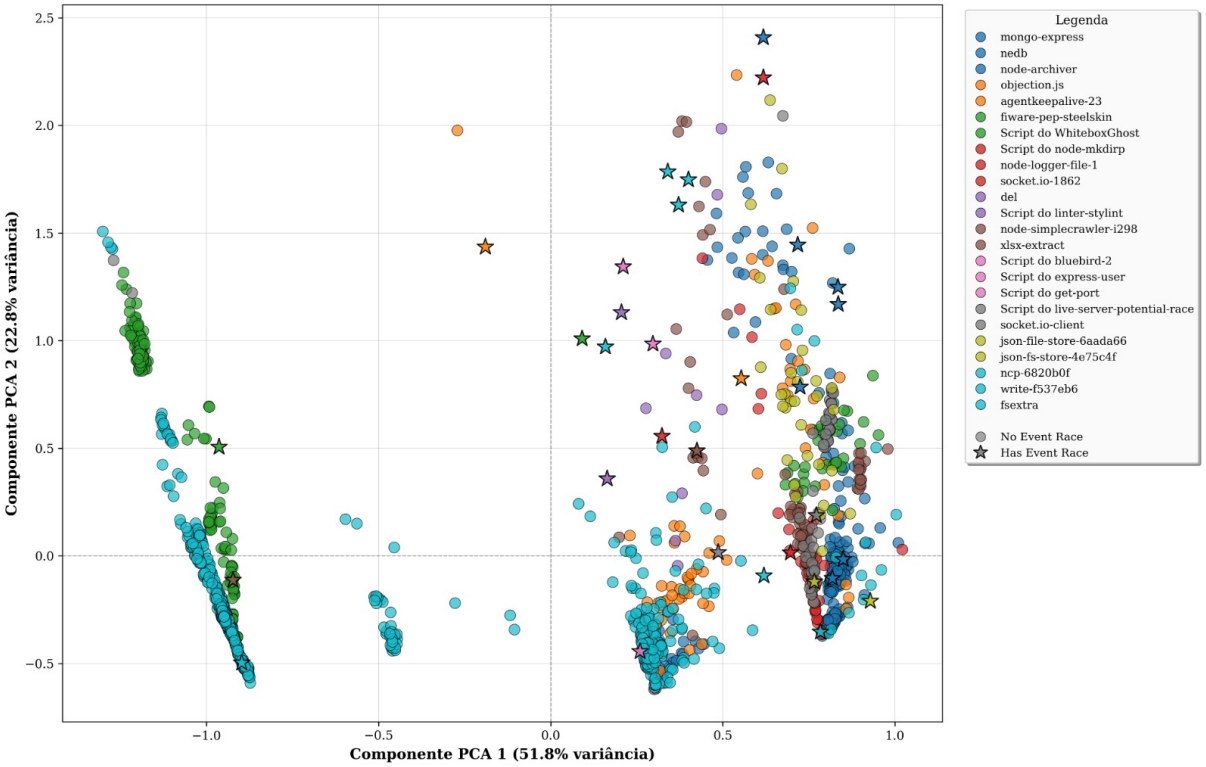
4 ANÁLISE DE RESULTADOS

Este capítulo apresenta a análise dos resultados obtidos nesta pesquisa. Serão discutidos os dados e as avaliações realizadas, fornecendo maior compreensão aplicabilidade da abordagem proposta. A organização do capítulo segue a seguinte estrutura: na Seção 4.1, é apresentada a visualização dos dados, com gráficos que ilustram a distribuição dos testes com e sem *event races*; na Seção 4.2, são discutidos os resultados relacionados à QP1, avaliando o desempenho dos modelos de ML na seleção de testes; na Seção 4.3, é abordada a QP2, analisando a capacidade de distinção das diferentes características utilizadas pelos classificadores; na Seção 4.4, é analisada a QP3, que investiga o desempenho em tempo de execução da metodologia; na Seção 4.5, são detalhadas as lições aprendidas; e, na Seção 4.6, são apresentadas as considerações finais deste estudo.

4.1 Visualização dos Dados

Para obter uma compreensão intuitiva da estrutura do *dataset* e da relação entre as métricas extraídas e a presença de *event races*, foi aplicada a técnica de Análise de Componentes Principais (*Principal Component Analysis* - PCA). A PCA é um método de redução de dimensionalidade que projeta os dados em um novo espaço de características de menor dimensão, preservando ao máximo a variância original. A Figura 6 ilustra o resultado inicial dessa análise, projetando cada caso de teste em um plano bidimensional. Cada ponto no gráfico corresponde a um único caso de teste, colorido de acordo com seu projeto de origem *(benchmark)* e marcado com um símbolo de estrela caso contenha um *event race* conhecido.

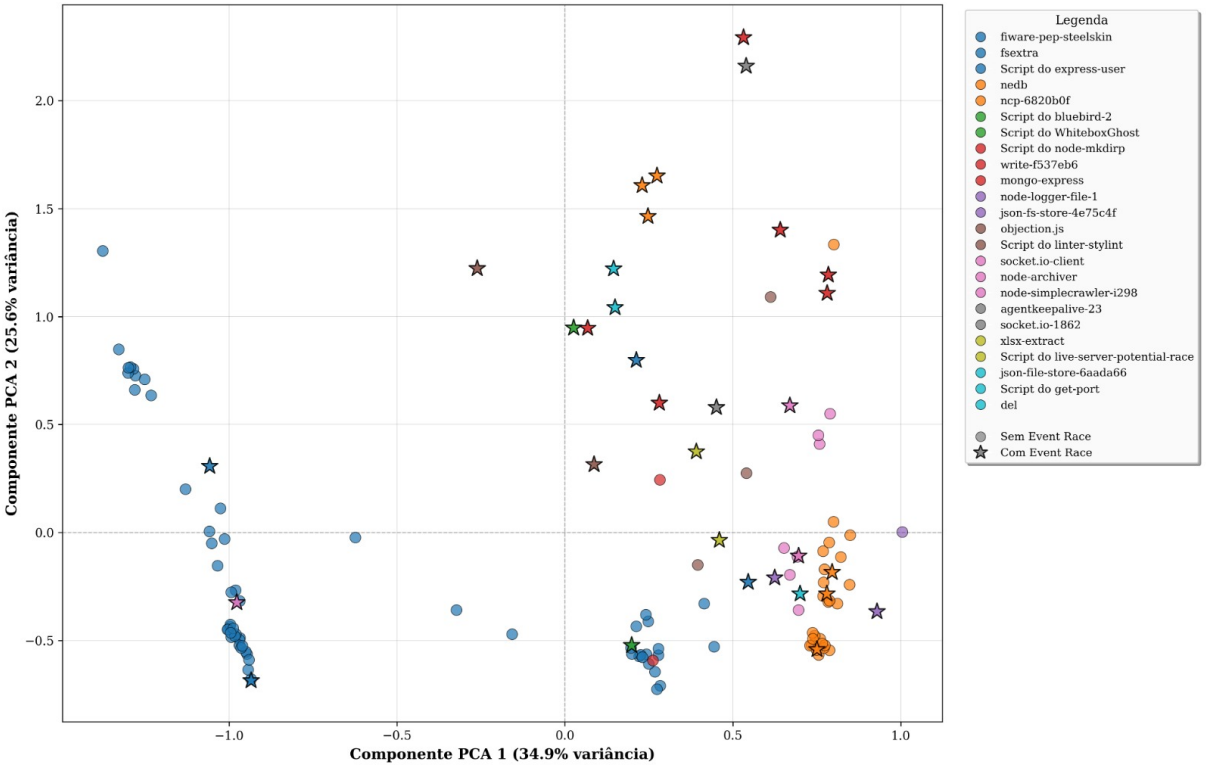
Figura 6: Distribuição de *event races* no *dataset*, colorido por projeto, visualizado através de PCA.



Fonte: Autoria própria.

A Figura 6 representa o *dataset* completo (original), que é desbalanceado, com apenas 32 instâncias positivas (com *event race*) em um total de 1720 testes (proporção de 1:53 entre as classes). Para mitigar o viés que o desbalanceamento introduz no treinamento de modelos e nas análises de ganho de informação (QP1 e QP2), foi aplicada uma técnica de subamostragem aleatória *(random undersampling)* no conjunto de testes sem *event race*. Especificamente, para cada teste rotulado como positivo, foram sorteados aleatoriamente três testes rotulados como negativos, resultando em uma proporção de 1:3 entre as classes. Essa amostragem não é utilizada na abordagem final do `NodeRock` (demonstrado através do QP3, que emprega PU *Learning* no *dataset* completo), mas serve como base para obter métricas de desempenho comparáveis e análises de Ganho de Informação mais estáveis para as QP1 e QP2. A Figura 7 ilustra a distribuição no PCA após essa amostragem. Nessa nova visualização, com menos ruído e melhor equilíbrio de classes, observa-se que os testes com *event race* (estrelas) se destacam de forma mais nítida, reforçando visualmente a ideia de que o perfil assíncrono dessas instâncias é distintivo e, portanto, pode ser classificado com modelos de ML.

Figura 7: Distribuição de *event races* no *dataset* após a amostragem, colorido por projeto, visualizado através de PCA.



Fonte: Autoria própria.

Uma primeira análise do gráfico revela uma tendência de agrupamento dos casos de teste com base em seu projeto de origem, onde pontos da mesma cor formam agrupamentos distintos e densos. Notam-se, por exemplo, concentrações de testes dos projetos `fsextra` e `nedb` em regiões específicas do gráfico. Essas concentrações são esperadas e podem ser amplamente atribuídas ao uso de métricas normalizadas (`_Normalized`). Como o processo de normalização é realizado no escopo de cada projeto individualmente, ele cria um perfil de métricas intrinsecamente distinto para cada *benchmark*, fazendo com que o projeto de origem seja uma das fontes mais fortes de variância. Isso se reflete na separação espacial observada no PCA, reforçando a hipótese de que projetos diferentes possuem, de fato, perfis de execução distintos.

Apesar da forte influência do projeto na distribuição dos dados, uma observação mais relevante para este trabalho é a disposição dos testes que contêm *event races*, indicados pelos símbolos de estrela. É notável que esses pontos não estão dispersos aleatoriamente pelo gráfico; pelo contrário, eles também exibem uma tendência de agrupamento em certas regiões ao desconsiderar o valor do rótulo do dado no PCA. De particular interesse são as áreas onde estrelas de diferentes cores (ou seja, de projetos distintos) se aproximam, como nos quadrantes inferior direito e central. Isso sugere a existência de um perfil de comportamento assíncrono comum a testes com *event races*, um padrão que transcende

as particularidades de um projeto específico.

A existência desses agrupamentos de testes com *event races* no espaço de características reduzido pelo PCA sugere que estes testes possuem um perfil de métricas dinâmicas distinguível dos testes que não apresentam *event races*. Essa separação visual indica uma oportunidade para o desenvolvimento de abordagens baseadas em ML. Se os testes com *event races* formam padrões reconhecíveis, é plausível que um classificador possa ser treinado para identificar esses padrões de comportamento assíncrono, permitindo a detecção e priorização eficaz de testes suscetíveis a *event races*, o que constitui a hipótese central investigada neste trabalho.

4.2 QP 1 - Qual a eficácia de um conjunto de modelos de ML (supervisionados e semissupervisionados) para a seleção de testes suscetíveis a *event races*?

Para responder à primeira QP1, que visa avaliar a eficácia dos modelos de ML na seleção de testes, a abordagem `NodeRock` foi aplicada ao conjunto de projetos compatíveis que contêm *event races* conhecidos, conforme detalhado na Subseção 3.2. Após a extração das métricas dinâmicas de cada teste, foi treinada e avaliada uma ampla gama de modelos de classificação, abrangendo abordagens supervisionadas e semissupervisionadas.

Para avaliar a capacidade de seleção dos testes, foram empregadas diferentes categorias de ML, cada uma com características específicas de rotulagem e classificação. Entre os modelos supervisionados, foram utilizados *Support Vector Machine (SVM)* (Cortes e Vapnik, 1995), *K-Nearest Neighbors (KNN)* (Cover e Hart, 1967), *Naive Bayes* (Duda, Hart e Stork, 2000), *Random Forest* (Breiman, 2001) e *Decision Tree* (Breiman et al., 1984). Nesses experimentos, a rotulagem seguiu de forma que os testes com *event races* conhecidos receberam o rótulo positivo, enquanto os demais foram tratados como negativos. Além dos classificadores individuais, foi avaliado um modelo de *Ensemble* por votação simples que combina as previsões anteriores utilizando uma lógica de votação inclusiva. Basta que um único modelo aponte a instância como positiva para que ela seja classificada como positiva pelo *Ensemble*.

Adicionalmente, foram adaptados algoritmos de clusterização, especificamente *HDBSCAN* (Campello, Moulavi e Sander, 2013) e *K-Means* (Lloyd, 1982), para realizar a tarefa de rotulação de agrupamento. A adaptação consistiu em agrupar os dados desconhecidos juntamente com os casos conhecidos, posteriormente, qualquer dado pertencente a um *cluster* contendo pelo menos três testes com *event races* conhecidos foi classificado como positivo.

Para complementar a análise, foram exploradas técnicas semissupervisionadas e de detecção de anomalias com configurações específicas. Os algoritmos de detecção de anomalias *Isolation Forest* (Liu, Ting e Zhou, 2008) e *One-Class SVM* (Schölkopf et al., 2001)

foram adaptados para este contexto sendo treinado exclusivamente com os dados que não contém *event races* confirmados, tratando os dados de testes com *event races* como anomalias. Na fase de classificação, instâncias consideradas "normais" (similares aos dados de treino) foram classificadas como positivas (possuem *event race*), enquanto os dados considerados *outliers* foram classificados como negativos. No caso do *PU Learning* (Elkan e Noto, 2008), os *event races* conhecidos formaram a classe positiva e os demais dados a classe não rotulada. Por fim, no *Label Spreading* (Zhou et al., 2003), aplicou-se uma heurística de propagação baseada nas métricas, na qual além dos positivos conhecidos, testes com as métricas `Invokes_with_callback_Raw` e `AsyncFunction_Count_Raw` iguais a zero foram rotulados diretamente como negativos (pois não apresentam comportamento assíncrono significativo para gerar *event races*), deixando o restante dos dados como não rotulados.

Os resultados da performance de cada modelo, avaliados com validação cruzada *Stratified 10-fold* em termos de Acurácia, Precisão, Revocação e F1-Score, são apresentados nas Figuras 8 e 9.

Figura 8: Comparação de desempenho de algoritmos supervisionados na detecção de *event races* conhecidos.

Legenda de Desempenho: ≥ 80% (Excelente) | 70% - 79% (Bom) | 50% - 69% (Regular) | < 50% (Baixo)

	Random Forest	KNN (K=5)	SVM	Naive Bayes	Decision Tree	Ensemble
Acurácia (%)	84,38	85,16	85,16	76,56	80,47	82,81
Precisão (%)	73,08	88,24	84,21	62,50	62,96	63,16
Revocação (%)	59,38	46,88	50,00	15,62	53,12	75,00
F1-Score (%)	65,52	61,22	62,75	25,00	57,63	68,57

Fonte: Autoria própria.

Figura 9: Comparação de desempenho de algoritmos semissupervisionados e de rotulação de agrupamentos na detecção de *event races* conhecidos.

Legenda de Desempenho: ≥ 80% (Excelente) | 70% - 79% (Bom) | 50% - 69% (Regular) | < 50% (Baixo)

	HDBSCAN	K-Means	Label Spreading	Isolation Forest	One-Class SVM	PU Learning
Acurácia (%)	78,12	34,38	34,38	69,53	71,88	75,00
Precisão (%)	54,35	26,36	23,47	43,64	46,55	50,00
Revocação (%)	78,12	90,62	71,88	75,00	84,38	84,38
F1-Score (%)	64,10	40,85	35,38	55,17	60,00	62,79

Fonte: Autoria própria.

Analisando a Figura 8, observa-se que os modelos supervisionados, de modo geral, apresentaram desempenhos robustos, com acurácias variando entre 76% e 85%. Modelos

como KNN (K=5) e SVM alcançaram as maiores acurácias (ambos com 85,16%) e precisão (88,24% e 84,21%, respectivamente). Contudo, esses modelos demonstraram uma capacidade limitada na métrica de maior relevância para este problema, a revocação *(recall)*. A revocação mede a habilidade do modelo de encontrar todos os testes que de fato possuem *event races*. Um baixo valor nesta métrica implica em falsos negativos, ou seja, *bugs* de concorrência que passariam despercebidos. Nesse quesito, os modelos supervisionados tiveram um desempenho modesto, com o melhor resultado *(Ensemble)* alcançando apenas 75% de revocação e os piores (*Naive Bayes*) ficando abaixo de 16%.

Em contraste, a análise da Figura 9 revela nuances sobre a análise das métricas. O algoritmo *K-Means*, com o valor mínimo de três agrupamentos, alcançou a maior taxa de revocação entre os modelos comparados (90,62%), o que a princípio pareceria ideal. No entanto, esse resultado foi obtido ao custo de uma precisão (26,36%) e acurácia (34,38%) reduzidas, indicando que o modelo tende a classificar a grande maioria dos testes como positivos, gerando um alto índice de falsos positivos. Diante disso, a abordagem de *PU Learning* se destacou como a estratégia mais equilibrada e viável. O *PU Learning* apresentou a segunda melhor revocação (84,38%), empatado com o *One-Class SVM*, mas obteve superioridade nas demais métricas em relação ao *K-Means* e *One-Class SVM*, com uma acurácia de 75,00% e precisão de 50,00%. Outros modelos, como *Isolation Forest* e *Label Spreading*, apresentaram desempenho inferior tanto na capacidade de detecção (revocação na faixa de 75%) quanto na precisão. Vale notar que o algoritmo *HDBSCAN* apresentou métricas competitivas, superando ligeiramente o *PU Learning* em acurácia (78,12%) e precisão (54,35%). Contudo, como sua taxa de revocação (78,12%) foi inferior à do *PU Learning*, optou-se por este último.

Portanto, em resposta à QP1, os modelos de ML mostram-se promissores para a seleção de testes suscetíveis a *event races*, exigindo, contudo, uma seleção criteriosa do algoritmo. Especificamente, a abordagem semissupervisionada via *PU Learning* revelou-se a mais adequada para a tarefa. Enquanto métodos como o *K-Means* maximizam a detecção às custas de tornarem a filtragem pouco eficaz (devido ao excesso de falsos positivos), o *PU Learning* oferece o melhor *trade-off*, proporcionando uma alta sensibilidade para detectar a grande maioria dos testes suscetíveis a *event races* (revocação superior a 84%), mantendo um nível de precisão suficiente para proporcionar uma redução real no esforço de inspeção e execução.

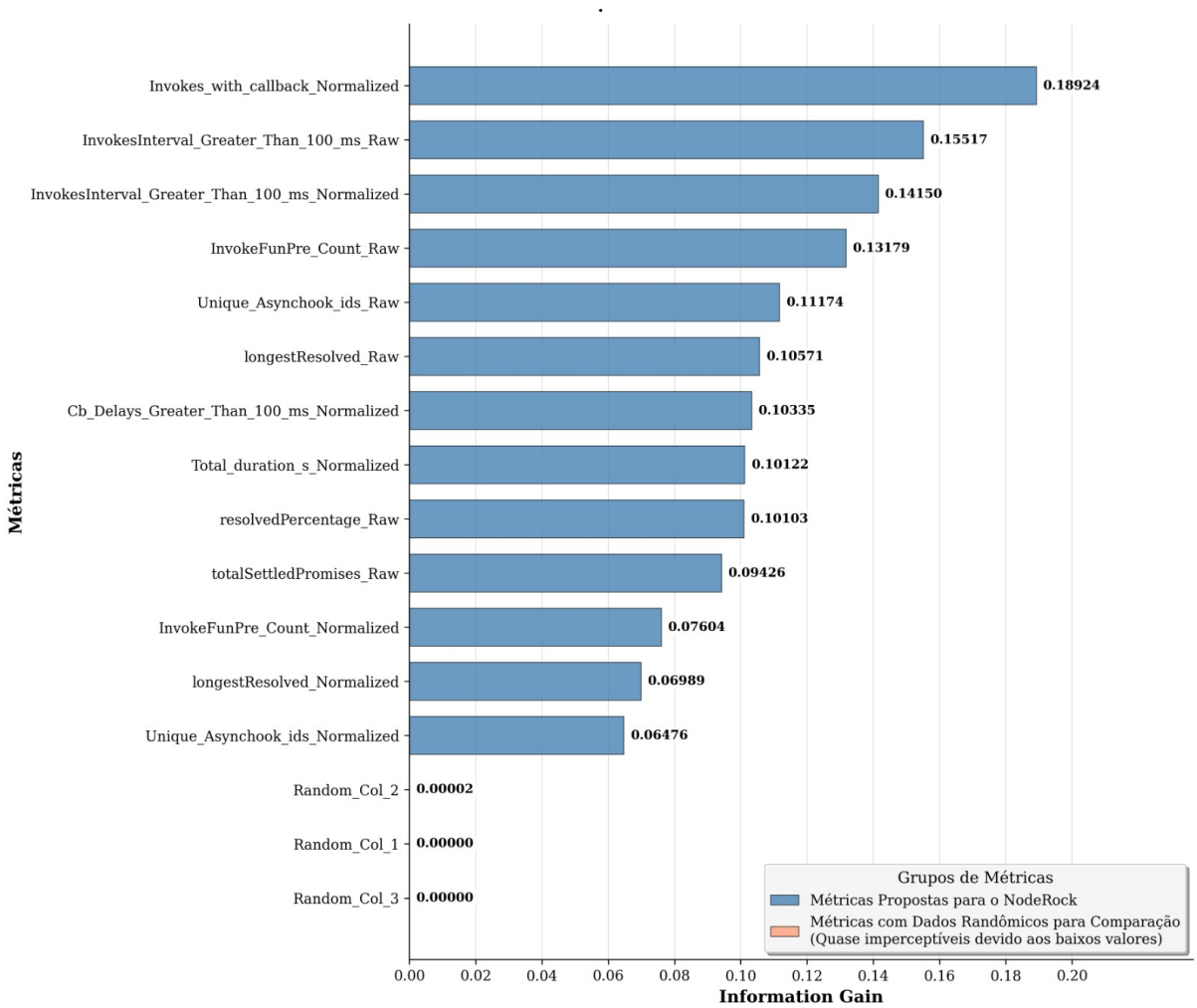
4.3 QP 2 - Qual a capacidade individual das diferentes características dinâmicas em distinguir testes com e sem *event races*?

A QP2 tem como objetivo avaliar a eficácia individual das métricas dinâmicas propostas pelo *NodeRock*, justificando sua escolha para a tarefa de identificação de testes suscetíveis a *event races*. Para isso, foi realizada uma análise de Ganho de Informação

(Information Gain - IG), utilizando o conjunto *undersampled* de projetos usados, detalhado na Subseção 3.2. O IG quantifica a relevância de cada métrica na distinção entre as classes com e sem *event race*.

Para estabelecer uma linha de base e demonstrar que as métricas propostas capturam informações significativas e não apenas ruído, foram incluídas na análise um conjunto de métricas com valores gerados de forma puramente aleatória (`Random_Col_1`, `Random_Col_2` e `Random_Col_3`). A expectativa é que as métricas do `NodeRock` apresentem um IG substancialmente superior a essas métricas de controle, validando sua importância para o modelo de classificação. Os resultados desta análise são apresentados na Figura 10, que ordena e apresenta as treze métricas com maior IG juntamente de métricas com valores puramente aleatórios.

Figura 10: Valores de Ganho de Informação entre métricas do `NodeRock` para a detecção de *event races*.



Fonte: Autoria própria.

A análise da Figura 10 fornece evidências sobre a relevância das métricas dinâmicas coletadas. A métrica que mais se destaca como informativa é a

`Invokes_with_callback_Normalized` (IG = 0.18924), que quantifica o número normalizado de invocações de função que utilizam *callbacks*. Do ponto de vista da engenharia de software, este resultado é consistente com a natureza dos *event races*: cada *callback* representa um evento adicionado ao *Event Loop*, aumentando exponencialmente o número de permutações possíveis de execução (*interleavings*). Quanto maior a quantidade de ordenações possíveis na execução de eventos, maior a probabilidade estatística de ocorrer uma ordem de execução não prevista que leve a uma falha. Logo em seguida, métricas relacionadas à duração e complexidade de operações, como `InvokesInterval_Greater_Than_100_ms_Raw` (IG = 0.15517) e `InvokeFunPre_Count_Raw` (IG = 0.13179), também apresentam alta relevância. A presença de operações de longa duração é crítica pois aumenta o intervalo de tempo entre o início de uma tarefa e sua conclusão. Durante esse intervalo, o *Event Loop* continua processando outros eventos que podem modificar o estado compartilhado da aplicação, criando as condições necessárias para uma inconsistência de dados através de um *event race*.

Nem todas as métricas propostas pelo `NodeRock` apresentam um IG significativo ao caracterizar diferentes facetas do comportamento assíncrono. Métricas como `avgRejected_Normalized`, (IG = 0.000081) e `awaitIntervals_Raw` (IG = 0.000791) são exemplos de métricas que não apresentam contribuições significativas para a detecção de *event races*, entretanto, a baixa quantidade de projetos JavaScript com estruturas modernas (*Promises* e *async/await*) no *dataset* justificam os baixos resultados de tais métricas. De maneira similiar, as métricas de controle com valores aleatórios (`Random_Col_1`, `Random_Col_2`, `Random_Col_3`) apresentaram um IG praticamente nulo (IG = 0.00000), tornando suas representações na Figura 10 quase imperceptíveis. Seus valores, próximos de zero, confirmam que estas métricas não possuem eficácia na distinção de *event races* e servem como evidência de que a capacidade de distinção das principais métricas do `NodeRock` não é fruto do acaso.

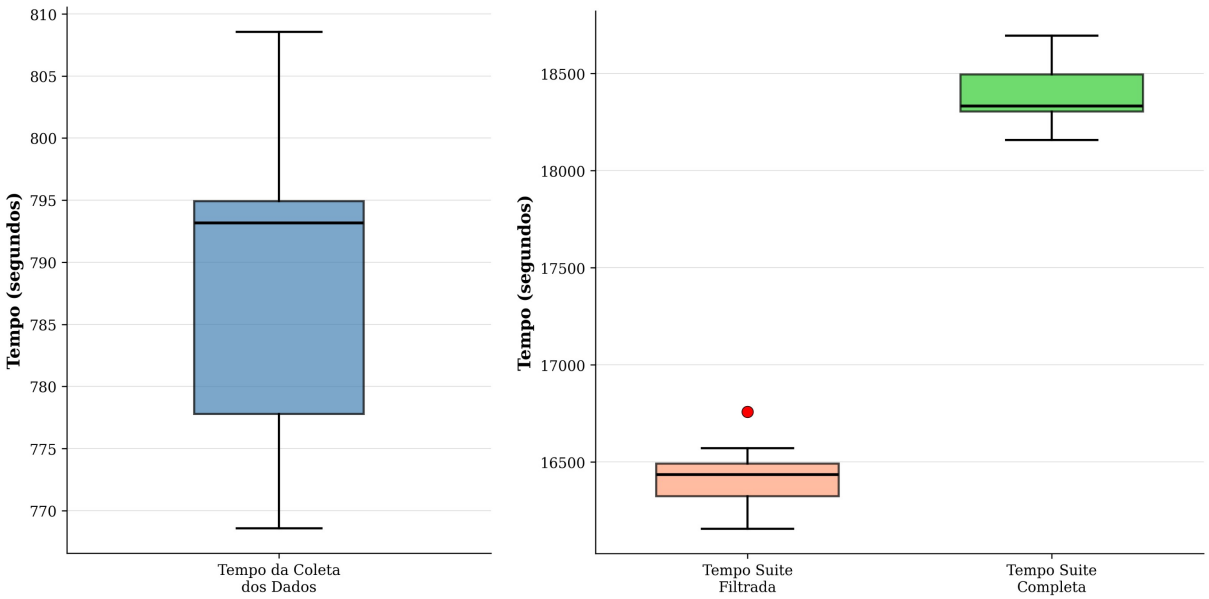
Portanto, em resposta à QP2, a análise de IG dá suporte à escolha das métricas dinâmicas implementadas no `NodeRock`. Elas não apenas superam a linha de base aleatória, mas também demonstram, através de seus valores de IG, que capturam características distintivas do comportamento assíncrono que levam à ocorrência de *event races*. Isso reforça que a abordagem de análise do `NodeRock` está fundamentada em indicadores relevantes e informativos.

4.4 QP3 - Qual o desempenho (em tempo de execução) destes modelos?

A QP3 investiga a viabilidade prática da abordagem, avaliando seu desempenho em termos de tempo de execução e o impacto de seu *overhead*. Para responder a esta questão,

foram conduzidos experimentos comparativos utilizando o projeto `node-archiver`, um dos projetos onde a presença de *event race* é conhecida, o número de testes automatizados é considerável, e possui compatibilidade com a versão do `NodeRock`. A abordagem de detecção de *event races* utilizada é não determinística, exigindo múltiplas execuções para aumentar a probabilidade de expor uma condição de corrida. Portanto, foi adotada a metodologia de realizar 100 execuções da suíte de testes com a ferramenta NACD. O objetivo é comparar o desempenho de duas abordagens sob este regime de teste exaustivo: uma análise completa da suíte e uma análise selecionada e priorizada pelo `NodeRock`.

A primeira abordagem, que serve como *baseline*, consistiu em executar o NACD sobre o Conjunto de Testes Completo do projeto `node-archiver`. A segunda abordagem integrou o `NodeRock` ao processo. Inicialmente, o `NodeRock` foi executado com o modelo de *PU Learning* para analisar e gerar um conjunto de testes filtrados, composto de apenas 24 dos 35 do total de testes do projeto que foram indicados como mais suscetíveis a *event races* de acordo com o modelo e priorizados na ordem de execução. Em seguida, a ferramenta NACD foi aplicada sobre este conjunto filtrado e reordenado. Para garantir a consistência estatística dos resultados, a execução de cada uma dessas abordagens foi repetida 15 vezes. As distribuições dos tempos de execução e da eficiência na detecção da primeira falha são apresentadas, respectivamente, nas Figuras 11 e 12.



(a) *Overhead* da Coleta de Dados.

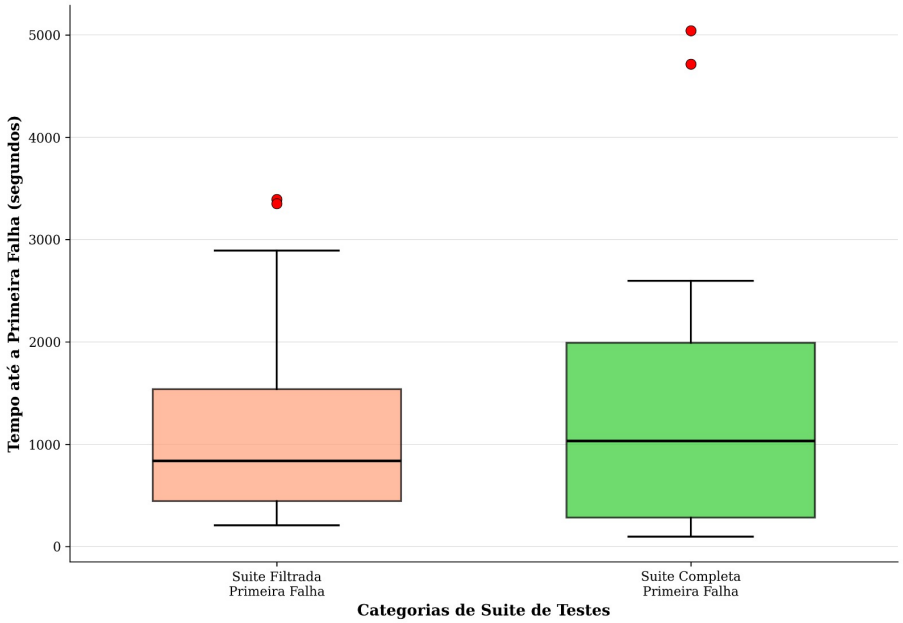
(b) Execução das Suítes de Testes.

Figura 11: Análise comparativa do Tempo de Execução de 100 execuções.

Fonte: Autoria própria.

A Figura 11 detalha o impacto do `NodeRock` no tempo total do processo. Observa-se na Figura 11(a) que o *overhead* de coleta de dados apresenta uma mediana de aproximadamente 793 segundos, com baixa variância (12s). Ao somar este custo ao tempo médio de

Figura 12: Análise comparativa do Tempo até a Primeira Falha para que o NACD exponha o *event race*.



Fonte: Autoria própria.

execução da suíte filtrada (16.416s), ilustrada através do *boxplot* laranja na Figura 11(b), obtém-se um tempo total de processo de aproximadamente 17.209 segundos. Comparado à média da suíte completa de 18.388 segundos, conforme apresentado no *boxplot* verde, a abordagem proporcionou uma economia de tempo total de cerca de 6,4%. Embora a redução no número de testes tenha sido de 31%, a economia de tempo foi mais modesta. Essa economia mais modesta no experimento de execução de 100 *runs* pode ser justificado pela hipótese de que os testes selecionados pelo modelo, por possuírem características de operações assíncronas mais suscetíveis a *event races* segundo o modelo de ML, são naturalmente mais complexos e demorados do que os testes descartados. Contudo, os *boxplots* ilustram que a suíte filtrada mantém uma distribuição consistente abaixo da suíte completa, evidenciando a redução de esforço computacional.

Outra métrica coletada é apresentada na Figura 12, que evidencia a eficiência da abordagem na detecção de falhas (*Time to First Failure*). A abordagem com a suíte filtrada apresentou uma mediana de 837 segundos para encontrar o primeiro *event race*, inferior à mediana de 1.031 segundos da suíte completa. Além disso, a dispersão dos dados mostra que a abordagem completa possui uma cauda longa superior mais acentuada, com o terceiro quartil (Q3) chegando a quase 2.000 segundos, enquanto a suíte filtrada mantém o Q3 em torno de 1.540 segundos.

Contudo, uma análise aprofundada deve considerar o tempo total de resposta, somando-se o *overhead* inicial de coleta de dados (mediana de 793s) ao tempo de detecção da suíte filtrada. Sob essa ótica, o tempo efetivo até a primeira falha na abordagem proposta

supera os valores observados na suíte completa. Esse comportamento é parcialmente justificado pela falta de eficácia do ordenamento neste caso específico, visto que o teste que continha o *event race* foi posicionado pelo modelo apenas na 21ª colocação entre os 24 testes filtrados. Uma hipótese provável para esse ranqueamento é que o teste contendo o *event race* apresenta valores de métricas de operações assíncronas inferiores aos testes classificados no topo da lista. Como o modelo atribui pontuações de probabilidade baseadas na similaridade com os padrões dos exemplos positivos do treinamento (frequentemente caracterizados por alta densidade de operações assíncronas), o modelo priorizou outros testes com maior quantidade de operações assíncronas em detrimento deste teste estruturalmente mais simples, porém com presença de *event race* confirmada.

Consequentemente, a ferramenta de detecção precisou executar a maior parte da suíte filtrada antes de expor a falha, o que, somado ao custo inicial de instrumentação, retardou o *feedback* da primeira ocorrência. Ainda assim, essa abordagem não implicou em perda na capacidade de detecção, uma vez que a média de falhas encontradas no conjunto filtrado (11,5) manteve-se estatisticamente equivalente à do conjunto completo (10,7).

Portanto, em resposta à QP3, os resultados sugerem a viabilidade técnica da abordagem de filtragem do `NodeRock`, apresentando *trade-offs* entre economia de processamento e tempo de detecção. A abordagem obteve uma redução no tempo total de execução (6,5%), indicando sua potencial capacidade de economizar recursos computacionais em cenários de reexecução intensiva. Entretanto, a análise do tempo até a primeira falha evidencia que a eficiência de tempo da abordagem é sensível ao custo fixo de extração de atributos e à precisão do ranqueamento dos testes. Neste experimento, o posicionamento tardio do teste crítico impediu um ganho de velocidade significativo na detecção inicial, sugerindo que a utilidade da abordagem é maior em contextos que demandam a redução do tempo total gasto em múltiplas execuções ao invés da velocidade de descoberta do primeiro defeito.

4.5 Lições Aprendidas

A condução deste estudo permitiu extrair lições e *insights* sobre a natureza dos *event races* e a metodologia para sua identificação. A principal lição aprendida foi a constatação de que o comportamento assíncrono de um teste, quando quantificado através de métricas dinâmicas, funciona como um indicador para a distinção de testes com e sem *event races*. Isso significa que, para suspeitar da presença de um *event race*, não é estritamente necessário compreender a semântica do *bug* ou modelar complexas relações de causalidade, como as de *happens-before*; o próprio perfil de execução, caracterizado pela intensidade de uso de *callbacks*, *Promises* e operações de longa duração, já constitui um indicador consistente.

Outra descoberta foi o desempenho superior da abordagem de *PU Learning* em compa-

ração com classificadores supervisionados tradicionais. A lição extraída é que o problema de identificar testes com *event races* não se enquadra perfeitamente em um paradigma de classificação binária padrão. A classe de testes "sem *event race*" não é uma certeza negativa, mas sim uma coleção de instâncias "não rotuladas", onde um *bug* pode existir sem ter sido descoberto. O *PU Learning* é projetado para este cenário de incerteza (Elkan e Noto, 2008), o que explica sua eficácia superior e o posiciona como uma escolha preferencial para futuras pesquisas na área de detecção de *event races*.

Por fim, a análise de desempenho em tempo de execução (QP3) apresentou o complexo (*trade-off*) entre seleção de testes e tempo de detecção. A redução quantitativa da suíte de testes não se traduz linearmente em economia de tempo, uma vez que os testes selecionados pelo `NodeRock`, por possuírem mais operações assíncronas, tendem a ser os mais custosos computacionalmente. Além disso, observou-se que o custo fixo de instrumentação para coleta de métricas é um fator crítico que pode retardar o *feedback* inicial. Apesar dessas limitações temporais, a abordagem alcançou uma otimização de recursos sem prejuízo estatístico à capacidade de detecção, mantendo o número de falhas encontradas no conjunto filtrado equivalente ao da suíte completa dentro do intervalo de confiança.

Essa relação entre *overhead* inicial e economia total sugere o cenário de uso mais adequado para a abordagem. Devido ao custo fixo de instrumentação e coleta das métricas, o `NodeRock` mostra-se menos vantajoso para validações rápidas a cada *commit* (onde a velocidade de *feedback* é prioritária). Em contrapartida, o *NodeRock* apresenta-se como uma abordagem promissora para auxiliar execuções periódicas e abrangentes de teste de código, como as compilações noturnas (*Nightly Builds*)[19] ou validações semanais de regressão. Nesses cenários, onde o volume de testes é massivo e a janela de execução permite processos de longa duração, a economia percentual no tempo de processamento traduz-se em redução de custos de infraestrutura, justificando o tempo investido na fase de coleta de métricas.

No entanto, a principal lição prática é o reconhecimento do risco associado a qualquer abordagem de seleção. Como demonstrado na análise dos modelos (QP1), a revocação, embora alta (84,38%), não é de 100%. Isso implica que existe uma pequena, mas real, probabilidade de que um teste contendo um *event race* seja incorretamente descartado pelo `NodeRock`. Portanto, a aplicação desta abordagem deve ser vista como uma estratégia de otimização de risco, pois permite que as equipes foquem a maior parte de seus recursos de depuração nas áreas mais prováveis de conter falhas, mas não elimina a necessidade de análises completas periódicas ou de outras abordagens de garantia de qualidade.

[19]Processo de construção e teste de software agendado para ocorrer automaticamente, geralmente fora do horário comercial, permitindo a execução de testes demorados que seriam inviáveis durante o desenvolvimento ativo (BrowserStack, 2025).

4.6 Considerações Finais

A análise dos resultados apresentada neste capítulo permitiu responder às três Questões de Pesquisa formuladas. Os resultados sugerem que *(i)* a aplicação de modelos de ML, com destaque para o *PU Learning*, é uma abordagem eficaz para a seleção de testes suscetíveis a *event races*; que *(ii)* as métricas dinâmicas propostas para caracterizar o comportamento assíncrono são relevantes e possuem capacidade para distinção de testes; e que *(iii)* a abordagem do `NodeRock` é viável, oferecendo benefícios de desempenho em cenários de teste realistas. Em conjunto, esses resultados fornecem evidências sobre a hipótese central deste trabalho: a seleção de testes baseada em análise de perfil de execução é uma abordagem promissora e um complemento às ferramentas de detecção de *event races* existentes.

5 CONCLUSÕES

Este trabalho abordou o desafio de detectar *event races* em aplicações Node.js, um problema cuja relevância aumenta com a complexidade do software, mas que soluções existentes frequentemente impõem um custo computacional elevado para aplicação em larga escala. Para mitigar esse problema, foi proposto e desenvolvido o `NodeRock`, uma abordagem que utiliza análise dinâmica e ML não para detectar diretamente *event races*, mas para selecionar e priorizar os testes suscetíveis a *event races*, otimizando o esforço de depuração.

Os resultados obtidos através da avaliação empírica apontam para a eficácia da abordagem proposta. A investigação indicou que: *(i)* a abordagem de *PU Learning* se mostrou superior, alcançando uma revocação de 84,38%, permitindo identificar a grande maioria dos testes com *event races*; *(ii)* o perfil de execução assíncrona, capturado por métricas dinâmicas, possui uma eficácia distintiva para diferenciar testes com e sem *event races*; e *(iii)* a metodologia é viável, reduzindo o tempo total de execução em cenários de teste exaustivos.

A principal contribuição deste trabalho, portanto, é a verificação empírica da seleção de testes baseada em perfil de execução como uma abordagem eficaz e complementar na detecção de *event races*. O `NodeRock` não visa substituir ferramentas de detecção como o NACD, mas sim torná-las mais eficientes ao direcionar seu uso. Ao atuar como uma camada de triagem inteligente, a abordagem permite que equipes de desenvolvimento foquem seus recursos limitados nas áreas de maior risco do código, tornando o processo de garantia de qualidade mais ágil e assertivo.

5.1 Trabalhos Futuros

As direções futuras a serem exploradas incluem:

Expansão e Refinamento do Conjunto de Métricas: Investigações futuras podem explorar a inclusão de novas *features*, como métricas de análise estática do código-fonte (e.g., complexidade ciclomática, linhas de código ocupadas pelo teste) ou extraídas de grafos de chamadas. Além disso, novas métricas podem ser incluídas para investigar a influência de mecanismos internos do *runtime*, especificamente o *Garbage Collector* do Node.js. Verificando se as pausas para coleta de lixo do *Google Chrome V8 Engine*[20] podem alterar as trocas de contexto no *Event Loop*, e procurar por correlações com a manifestação de *event races*.

Aprimoramento do Modelo com *Datasets* Expandidos: A robustez do modelo de ML está ligada à diversidade e quantidade dos dados de treinamento. Uma direção futura importante é avaliar o desempenho do `NodeRock` em um *dataset* maior e mais diversificado,

[20] <https://nodejs.org/en/learn/getting-started/the-v8-javascript-engine>

contendo um número significativamente maior de *event races* conhecidos, provenientes de uma gama maior de projetos. Isso permite não apenas validar a generalização do modelo, mas também explorar arquiteturas de aprendizado mais complexas, como redes neurais profundas, que podem capturar padrões mais sutis e potencialmente aumentar a precisão e a revocação da classificação.

Otimização e Ajuste Fino dos Modelos de ML: Para maximizar a eficácia da classificação, trabalhos futuros podem focar na otimização de hiperparâmetros dos modelos. Paralelamente, pode-se conduzir uma análise aprofundada de seleção de atributos (*feature selection*) para identificar e remover métricas redundantes ou de baixa relevância. Isso pode mitigar riscos associados à maldição da dimensionalidade, garantindo que o modelo foque apenas nas características com maior poder discriminativo.

Integração Automatizada com Ferramentas de Detecção: Um avanço natural é a criação de um *pipeline* de teste totalmente automatizado. Nesse sistema, ocasionalmente, quando novos testes forem adicionados ao ambiente de desenvolvimento o `NodeRock` é acionado, selecionando o subconjunto de testes suspeitos e, em seguida, acionando automaticamente uma ferramenta de detecção, como o NACD, apenas nesse conjunto priorizado. Isso pode transformar o `NodeRock` de uma abordagem de recomendação em um componente ativo do ciclo de depuração.

Análise de Consumo de Recursos e Sustentabilidade: Além da eficácia na detecção, é fundamental avaliar o impacto da ferramenta no ambiente de execução. Estudos futuros podem incluir medições detalhadas de uso de CPU, ocupação de memória RAM e consumo energético, comparando cenários com e sem a instrumentação do `NodeRock`. Tais métricas são essenciais para validar a viabilidade da ferramenta em ambientes de Integração Contínua (CI) com recursos limitados.

Otimização de *Overhead* via Análise Híbrida: Dado que o custo fixo de instrumentação dinâmica representa uma parcela significativa do tempo total, uma arquitetura híbrida pode ser explorada. Uma etapa preliminar de análise estática pode descartar testes manifestamente síncronos ou triviais. Dessa forma, o `NodeRock` aplica a instrumentação dinâmica pesada apenas no subconjunto de testes onde ela é estritamente necessária, reduzindo o tempo total de coleta sem comprometer a precisão.

Generalização da Abordagem: Finalmente, é interessante investigar a generalização desta metodologia para outros contextos. A abordagem de análise de perfil de execução combinada com ML pode ser adaptada para priorizar testes que revelam outros tipos de *bugs* (e.g., *memory leaks*) ou aplicada a outros ecossistemas assíncronos, como Python com asyncio[21] ou C#.

[21] <https://docs.python.org/3/library/asyncio.html>

Referências

1 TIOBE Software BV. *TIOBE Index for December 2025*. 2025. Disponível em: <https://www.tiobe.com/tiobe-index/>. Acesso em: 9 de dezembro de 2025.

2 Stack Exchange, Inc. *2025 Developer Survey*. 2025. Disponível em: <https://survey.stackoverflow.co/2025/technology>. Acesso em: 19 de agosto de 2025.

3 CHRZANOWSKA, N. *12 Top Applications Written in Node.js - Examples from Big Companies*. 2024. Disponível em: <https://www.netguru.com/blog/top-companies-used-nodejs-production>. Acesso em: 19 de agosto de 2025.

4 ENDO, A. T.; MøLLER, A. Noderacer: Event race detection for node.js applications. In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. [S.l.: s.n.], 2020. p. 120–130.

5 ZHOU, J. et al. Nodert: Detecting races in node.js applications practically. In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2023. (ISSTA 2023), p. 1332–1344. ISBN 9798400702211. Disponível em: <https://doi.org/10.1145/3597926.3598139>.

6 ENDO, A. T.; MØLLER, A. Event Race Detection for Node.js Using Delay Injections. In: ALDRICH, J.; SILVA, A. (Ed.). *39th European Conference on Object-Oriented Programming (ECOOP 2025)*. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. (Leibniz International Proceedings in Informatics (LIPIcs), v. 333), p. 9:1–9:28. ISBN 978-3-95977-373-7. ISSN 1868-8969. Disponível em: <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2025.9>.

7 LUO, Q. et al. An empirical analysis of flaky tests. In: CHEUNG, S.; ORSO, A.; STOREY, M. D. (Ed.). *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. ACM, 2014. p. 643–653. Disponível em: <https://doi.org/10.1145/2635868.2635920>.

8 CHANG, X. et al. Race detection for event-driven node.js applications. In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.: s.n.], 2021. p. 480–491.

9 Dominik Szahidewicz. *What Is a Test Suite? A Complete Guide*. 2025. Disponível em: <https://bugbug.io/blog/software-testing/test-suite/>. Acesso em: 23 de novembro de 2025.

10 ROTHERMEL, G. et al. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, v. 27, n. 10, p. 929–948, 2001.

11 HARROLD, M. J.; GUPTA, R.; SOFFA, M. L. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, Association for Computing Machinery, New York, NY, USA, v. 2, n. 3, p. 270–285, jul. 1993. ISSN 1049-331X. Disponível em: <https://doi.org/10.1145/152388.152391>.

12 Ecma International. *ECMAScript® 2025 Language Specification*. 2025. <https://262.ecma-international.org/#sec-call>. Acesso em: 10 de dezembro de 2025.

13 Mozilla Developer Network. *Callback function*. 2025. <https://developer.mozilla.org/en-US/docs/Glossary/Callback_function>. Acesso em: 20 de novembro de 2025.

14 Ecma International. *ECMAScript® 2025 Language Specification*. 2025. <https://262.ecma-international.org/#sec-promise-objects>. Acesso em: 10 de dezembro de 2025.

15 Mozilla Developer Network. *Promise*. 2025. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise>. Acesso em: 20 de novembro de 2025.

16 DANGLOT, B. et al. A snowballing literature study on test amplification. *J. Syst. Softw.*, v. 157, 2019. Disponível em: <https://doi.org/10.1016/j.jss.2019.110398>.

17 ANDREASEN, E. et al. A survey of dynamic analysis and test generation for JavaScript. *ACM Comput. Surv.*, v. 50, n. 5, p. 66:1–66:36, 2017.

18 SUN, H. et al. Efficient dynamic analysis for node.js. In: *Proceedings of the 27th International Conference on Compiler Construction*. New York, NY, USA: ACM, 2018. (CC 2018), p. 196–206. ISBN 978-1-4503-5644-2. Disponível em: <http://doi.acm.org/10.1145/3178372.3179527>.

19 ZHANG, P.; ELBAUM, S. Amplifying Tests to Validate Exception Handling Code: An Extended Study in the Mobile Application Domain. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, v. 23, n. 4, 2014.

20 ADAMSEN, C. Q.; MØLLER, A.; TIP, F. Practical initialization race detection for JavaScript web applications. *PACMPL*, v. 1, n. OOPSLA, p. 66:1–66:22, 2017.

21 PETROV, B. et al. Race detection for web applications. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. [S.l.]: ACM, 2012. p. 251–262.

22 WANG, J. et al. A comprehensive study on real world concurrency bugs in Node.js. In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. [S.l.]: IEEE Computer Society, 2017. p. 520–531.

23 OpenJS Foundation. *The Node.js Event Loop*. 2025. Documentação oficial do Node.js. Disponível em: <https://nodejs.org/en/learn/asynchronous-work/event-loop-timers-and-nexttick>. Acesso em: 21 de agosto de 2025.

24 TUZCUOğLU, E. *Exploring Node.js Event Loop: A Complete Guide*. 2024. Blog post no Medium. Disponível em: <https://medium.com/@erimtuzcuoglu/exploring-node-js-event-loop-a-complete-guide-79d8e735818e>. Acesso em: 21 de agosto de 2025.

25 DAVIS, J.; THEKUMPARAMPIL, A.; LEE, D. Node.fz: Fuzzing the server-side event-driven architecture. In: *Proceedings of the Twelfth European Conference on Computer Systems*. New York, NY, USA: Association for Computing Machinery, 2017. (EuroSys '17), p. 145–160. ISBN 9781450349383. Disponível em: <https://doi.org/10.1145/3064176.3064188>.

26 LU, S. et al. Learning from mistakes - a comprehensive study on real world concurrency bug characteristics. In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008)*. Seattle, WA, USA: ACM, 2008. p. 329–339. Disponível em: <https://dl.acm.org/doi/10.1145/1346281.1346323>.

27 WANG, J. et al. A comprehensive study on real world concurrency bugs in node.js. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.: s.n.], 2017. p. 520–531.

28 ELBAUM, S.; MALISHEVSKY, A.; ROTHERMEL, G. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering*, v. 28, n. 2, p. 159–182, 2002.

29 CRUCIANI, E. et al. Scalable approaches for test suite reduction. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2019. p. 419–429.

30 CHEN, S. et al. Using semi-supervised clustering to improve regression test selection techniques. In: *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. [S.l.: s.n.], 2011. p. 1–10.

31 BUSJAEGER, B.; XIE, T. Learning for test prioritization: an industrial case study. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2016. (FSE 2016), p. 975–980. ISBN 9781450342186. Disponível em: <https://doi.org/10.1145/2950290.2983954>.

32 BAGHERZADEH, M.; KAHANI, N.; BRIAND, L. Reinforcement learning for test case prioritization. *IEEE Transactions on Software Engineering*, v. 48, n. 8, p. 2836–2856, 2022.

33 MCKINNEY Wes. Data Structures for Statistical Computing in Python. In: WALT Stéfan van der; MILLMAN Jarrod (Ed.). *Proceedings of the 9th Python in Science Conference*. [S.l.: s.n.], 2010. p. 56 – 61.

34 PEDREGOSA, F. et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, v. 12, n. 85, p. 2825–2830, 2011. Disponível em: <http://jmlr.org/papers/v12/pedregosa11a.html>.

35 ELKAN, C.; NOTO, K. Learning classifiers from only positive and unlabeled data. In: *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA: Association for Computing Machinery, 2008. (KDD '08), p. 213–220. ISBN 9781605581934. Disponível em: <https://doi.org/10.1145/1401890.1401920>.

36 BrowserStack. *What is Monkey Patching?* 2025. Disponível em: <https://www.browserstack.com/guide/monkey-patching>. Acesso em: 24 de novembro de 2025.

37 CORTES, C.; VAPNIK, V. Support-vector networks. *Mach. Learn.*, Kluwer Academic Publishers, USA, v. 20, n. 3, p. 273–297, set. 1995. ISSN 0885-6125. Disponível em: <https://doi.org/10.1023/A:1022627411411>.

38 COVER, T.; HART, P. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, v. 13, n. 1, p. 21–27, 1967.

39 DUDA, R. O.; HART, P. E.; STORK, D. G. *Pattern Classification (2nd Edition)*. USA: Wiley-Interscience, 2000. ISBN 0471056693.

40 BREIMAN, L. Random forests. *Machine learning*, Springer, v. 45, n. 1, p. 5–32, 2001.

41 BREIMAN, L. et al. *Classification and Regression Trees*. [S.l.]: CRC press, 1984. ISBN 9781315139470.

42 CAMPELLO, R. J. G. B.; MOULAVI, D.; SANDER, J. Density-based clustering based on hierarchical density estimates. In: PEI, J. et al. (Ed.). *Advances in Knowledge Discovery and Data Mining*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 160–172. ISBN 978-3-642-37456-2.

43 LLOYD, S. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, v. 28, n. 2, p. 129–137, 1982.

44 LIU, F. T.; TING, K. M.; ZHOU, Z.-H. Isolation forest. In: *2008 Eighth IEEE International Conference on Data Mining*. [S.l.: s.n.], 2008. p. 413–422.

45 SCHöLKOPF, B. et al. Estimating the support of a high-dimensional distribution. *Neural Computation*, v. 13, n. 7, p. 1443–1471, 2001.

46 ZHOU, D. et al. Learning with local and global consistency. In: *Proceedings of the 17th International Conference on Neural Information Processing Systems*. Cambridge, MA, USA: MIT Press, 2003. (NIPS'03), p. 321–328.

47 BrowserStack. *Test Strategies for Daily and Nightly Builds*. 2025. Disponível em: <https://www.browserstack.com/guide/test-strategies-for-daily-and-nightly-builds>. Acesso em: 25 de novembro de 2025.