Logify: Uma Abordagem Inovadora para Desenvolvimento e Testagem de Software com Record & Replay e Processamento Natural de Linguagem

João Victor Bezerra da Silva
DART - Development, Automation, Research and Test
SIDIA Development and Research Institute
Porto Velho, Rondônia
victor.bezerra@sidia.com

Antônio Vinícius Rodrigues da Costa
DART - Development, Automation, Research and Test
SIDIA Development and Research Institute
Porto Velho, Rondônia
antonio.rodrigues@sidia.com

Paulo Daniel de Araujo
DART - Development, Automation, Research and Test
SIDIA Development and Research Institute
Porto Velho, Rondônia
paulo.araujo@sidia.com

Breno Nogueira Araujo
DART - Development, Automation, Research and Test
SIDIA Development and Research Institute
Porto Velho, Rondônia
breno.araujo@sidia.com

***Resumo*—A demanda crescente por aplicativos móveis exige ciclos de desenvolvimento ágeis e eficientes, tornando essencial a colaboração contínua entre as equipes de desenvolvedores e garantia de qualidade (*Quality Assurance* - QA). No entanto, os métodos tradicionais de desenvolvimento e teste enfrentam desafios como a manutenção dos scripts de testes, inconsistências entre versões dos aplicativos e dificuldades na adaptação de cenários de testes a cada atualização do sistema. Além disso, os desenvolvedores frequentemente enfrentam dificuldades na reprodução de bugs com base nos relatórios fornecidos pela equipe de QA. Esse problema ocorre devido à falta de clareza, precisão e objetividade nas descrições, pois há uma lacuna cognitiva e lexical entre os desenvolvedores e o time de QA. Isso leva a instruções ambíguas ou incompletas, dificultando a interpretação e a reprodução de uma forma deterministica dos cenários reportados. Para abordar essas limitações, propomos o *Log Optimization and Generation for Intelligent Functional Development & Testing* (Logify), um framework inteligente baseado em *Record and Replay* (R&R) e Processamento de Linguagem Natural (*Natural Language Processing* - NLP), que permite o versionamento eficiente de gravações para reproduções de bugs para dispositivos Android. A abordagem captura e trata interações do usuário através de logs do *Android Debug Bridge* (ADB), armazena versões das gravações em um servidor remoto e aplica NLP para buscar gravações de bugs de uma forma mais otimizada. O presente trabalho melhora a rastreabilidade dos defeitos em softwares e facilita a gravação e reprodução de cenários de testes, reduzindo o esforço manual necessário para atualização dos testes automatizados. Avaliações realizadas em um ambiente de desenvolvimento real demonstram que a solução proposta reduz os casos em que o desenvolvedor não consegue reproduzir o bug devido a instruções ambíguas ou mal comprendidas em até 31.22%, otimizando o fluxo de trabalho entre as equipes, ao mesmo tempo que fornece uma precisão ainda melhor em comparação com *scripts* manuais.**

***Index Terms*—Automated Android Testing, Logs, Natural Language Processing**

I. Introdução

A indústria de aplicações móveis está cada vez mais popular em decorrência do aumento do número de usuários que adotam dispositivos móveis[1]. Entre 2015 e 2023, observou-se um incremento quantitativo notável nos downloads das lojas de aplicativos Play Store e App Store que passaram de 15.9 bilhões para 35 bilhões[2]. Esse crescimento acentuado destaca a importância de garantir a qualidade e a confiabilidade dos aplicativos oferecidos aos usuários. Dado a esse avanço, um dos grandes desafios enfrentados pelas equipes de desenvolvimento e *Quality Assurance* (QA) é a reprodução eficiente de bugs e a manutenção de testes automatizados, especialmente em um cenário de desenvolvimento ágil e de lançamentos contínuos[1].

Sob essa ótica, a reprodutibilidade de bugs é uma questão central para equipes de QA e desenvolvedores de *software*, pois relatórios imprecisos, incompletos ou ambíguos dificultam a identificação e correção das falhas. Trabalhos como os [3] e [4] evidenciam os impactos negativos da lacuna cognitiva e lexical entre QA e desenvolvedores. Desse modo, a ausência de um mecanismo eficiente de versionamento e a inconsistência na documentação dos testes resultam em retrabalho, aumento de tempo de depuração e falhas no software não solucionadas. A manutenção manual de testes automatizados frequentemente não acompanha as mudanças no software, gerando complexidade acidental e dificultando a escalabilidade dos processos de teste [5].

De fato, vários trabalhos nos últimos anos têm proposto melhorias no processo de automatização de desenvolvimento de testes para dispositivos móveis, ferramentas como o

Recdroid[6] e AdbGpt[7] partem de relatos de bugs descritos em linguagem natural para extrair ações e comportamentos de inteface, gerando scripts de testes para serem executados utilizando comandos do *Android Debug Bridge* (ADB) ou em ferramentas para automação de interface para executar ações diretamente no dispositivo mobile. Ferramentas como o ScriptPainter[1], dependem de gravações da tela para gerar scripts de teste, e abordagens evolucionárias como o Mobolic[8], que geram scripts de testes com base em mudanças no código, não capturam com fidelidade as ações executadas pelo QA nem preservam o contexto completo da execução. Como resultado, essas soluções dificultam a reprodução precisa de falhas e não garantem rastreabilidade confiável em execuções futuras.

Assim, o presente estudo apresenta uma ferramenta para automatizar o processo de gravação e reprodução de testes em aplicações móveis, além de realizar o versionamento e a recuperação dessas gravações utilizando busca semântica. A solução permite capturar metadados dos aplicativos, descrições do teste realizado e interações durante a execução, armazenando-os de forma estruturada e permitindo sua posterior consulta por meio de *Natural Language Processing* (NLP). Ao integrar esses recursos, o Logify busca ampliar a rastreabilidade, facilitar a manutenção dos testes e promover a reutilização de execuções anteriores, como é o caso de cenários nos quais a automação total não é viável, como em testes explorátorios ou validações subjetivas.

II. Fundamentação Teorica

O teste de software é uma atividade essencial no ciclo de desenvolvimento, voltada à verificação e validação do comportamento esperado de um sistema. Diante disso, a automação de testes tem ganhado destaque pela sua capacidade de aumentar a cobertura, reduzir o tempo de execução e diminuir a incidência de erros humanos. No entanto, em uma análise empírica, é notório que times de QA ainda documentam manualmente os defeitos encontrados, descrevendo o passo a passo, entradas e resultados esperados para que desenvolvedores possam reproduzir o problema. Esse processo, embora comum, é suscetível a falhas de comunicação, pois relatórios incompletos, vagos ou mal estruturados estão entre os principais fatores que dificultam a reprodução de bugs e comprometem diretamente a capacidade de resolução de falhas[3].

Tal problemática é agravada por fatores como ambiguidade lexical entre equipes de QA e desenvolvimento, variações semânticas em descrições semelhantes e ausência de padronização na forma de relatar defeitos[4]. O resultado é um alto índice de falhas não reproduzidas, retrabalho entre times e degradação da confiabilidade dos processos de correção. Embora, ferramentas de rastreamento de defeitos permitam o anexo de imagens, vídeos e logs aos relatos de falhas, tais artefatos geralmente são incoporados sem nenhuma estrutura padronizada, o que limita sua reutilização em processos automatizados ou buscas subsequentes, como observado em [9], [10] e [11]. Além disso, não há, na maioria dos fluxos, mecanismos para registrar e versionar sessões manuais de testes de forma que possam ser recuperados e reaproveitadas ao longo do tempo.

Nesse contexto, os trabalhos de [12], [13] e [14] propõem o uso de métodos evolucionários ou técnicas de inteligência artificial, como algoritmos genéticos, estratégias evolutivas e programação genética para a geração automática de scripts de teste ou para a exploração autônoma de aplicações em busca de falhas inesperadas. Essas abordagens tratam o problema de teste como uma tarefa de otimização, onde o objetivo é maximizar critérios como cobertura de código, diversidade de caminhos ou detecção de exceções. Essas estratégias são estocásticas e não determinísticas por natureza, o que compromete aplicações em que o contexto exige o registro fiel do fluxo de execução e reprodutibilidade precisa.

Recdroid[6] e AdbGpt[7] apresentam ferramentas para realizar a reprodução automática das ações nos aplicativos com base em relatórios de bugs escritos em linguagem natural. Essas ferramentas utilizam técnicas de NLP ou modelos de linguagem para interpretar os relatos e gerar os *scritps* ou comandos automatizados que tentam simular o comportamento descrito. No entanto, enfrentam problemas de relatórios incompletos, mal escritos ou ambíguos que dificultam a geração de reproduções confiáveis e comprometem a reprodução do bug no ambiente do desenvolvedor.

Já Reran[15] e Valera[16] exploram abordagens baseadas na captura e reprodução precisa de eventos de entrada em dispositivos Android, buscando garantir fidelidade temporal e comportamento determinístico na reexecução de ações. Ambas operam diretamente em eventos de baixo nível capturados via comandos ADB, permitindo reproduções altamente precisas. No entanto, essas soluções apresentam limitações práticas, pois requerem *root*, ou seja, permissões elevadas do sistema operacional, e também não oferecem suporte a coleta de metadados contextuais, tampouco mecanismos de versionamento padronizado para consulta ou associação semântica com as descrições de bugs.

O Logify diferencia-se ao adotar uma abordagem que assegura a captura precisa e determinística das interações do QA permitindo o versionamento e recuperação dessas gravações através de um servidor remoto, possibilitando a reprodução fiel das sessões de teste. Além disso, esse trabalho inspira-se nos princípios do Continuos, Evolutionary, and Large-scale testing(CEL) proposto por [5]. Os autores argumentam que, devido à fragmentação de dispositivos e as plataformas em rápida evolução, é essencial adotar uma abordagem de teste que seja contínua, capaz de evoluir com o software e escalável para abranger uma ampla variedade de cenários de uso. Eles propõem uma estrutura de testes automatizados que integram esses princípios, visando melhorar a eficácia e a eficiência na detecção de falhas em aplicativos móveis.

Embora inspirado nos princípios do CEL, a ferramenta proposta nesse trabalho não incorpora testes evolucionários, optando por uma abordagem centrada no reuso do registro de ações executadas nos aplicativos com técnicas de *Record and Replay* (R&R) e na busca semântica com base em linguagem

natural. Com isso, a ferramenta preserva os benefícios do CEL em termos de versionamento contínuo e evolução incremental e evita os desafios inerentes às tecnicas estocásticas, como a variabilidade entre as execuções e a ausência de rastreio. Essa escolha permite que o Logify mantenha um equilíbrio entre a automação e a precisão na captura das interações humanas, garantindo a fidelidade necessária para a reprodutibilidade dos testes.

A. Similaridade Semântica com modelos Pré-Treinados

O Logify utiliza o modelo *all-miniLM-L6-v2* para a implementação dos procedimentos de análise semântica. O servidor implementa o modelo para processar as descrições fornecidas pelo usuário, que incluem o nome do aplicativo, especificações de dispositivo móvel e metadados associados. Esse modelo, o qual faz parte da família *Sentence Transformers*, é projetado para mapear sentenças e parágrafos em um espaço vetorial denso de 384 dimensões, capturando informações semânticas essenciais para tarefas como busca semântica e agrupamento de textos [17]. A utilização de metadados para gerenciar e facilitar a busca e recuperação de dados é uma abordagem reconhecida na literatura, conforme discutido em [18] e [19]. Essa metodologia permite uma recuperação eficiente e precisa dos registros relevantes, aprimorando a eficácia do sistema de versionamento de testes. Em virtude dos avanços proporcionados por arquiteturas baseadas em *Transformes* [20] e [21], observou-se que técnicas de *deep self-attention distillation*, como proposta em [22], podem alcançar um desempenho comparável ao de modelos de maior dimensão, ao mesmo tempo em que apresentam custos computacionais significativamente reduzidos.

III. Logify

O paradigma de R&R foi escolhido para o desenvolvimento do framework proposto neste trabalho devido sua natureza determinística, permitindo o registro e reprodução de forma idêntica das interações realizadas nos aplicativos, garantindo a consistência dos testes [23]. Para viabilizar essa abordagem sem modificar o aplicativo sob teste, a ferramenta utiliza o ADB[24] como meio de registro das ações, explorando sua compatibilidade com dispositivos Android e sua capacidade de monitorar eventos do sistema operacional de forma não intrusiva.

A ferramenta proposta, denominada Logify, tem como objetivo a coleta e reprodução de interações via ADB, sem a necessidade de permissões avançadas do sistema operacional. Além disso, introduz um módulo de pré-processamento que converte os logs brutos em um formato estruturado, classificando automaticamente as interações do usuário em categorias como clique, deslizamento e rolagem. Conforme os resultados obtidos, esse processo aumenta a eficiência da reprodução dos testes e permite que os dados sejam armazenados de forma padronizada, permitindo seu uso na fase de reprodução.

A arquitetura do sistema proposto é estruturada em dois módulos principais: (i) o módulo de captura e reprodução de testes, baseado em R&R, e o (ii) módulo de análise semântica e versionamento, que utiliza NLP para interpretar os relatórios de bugs criados pelo QA e permitir a rastreabilidade dos testes ao longo das versões do software. A interação entre os módulos pode ser observada na Figura 1.

A. Captura e Reprodução dos Testes

A gravação e reprodução das ações executadas nos aplicativos móveis é realizada através de um cliente *desktop*. Essa ferramenta permite capturar, de maneira estruturada, as interações realizadas pelo QA ou desenvolvedor durante o uso dos aplicativos Android. Para isso, o dispositivo a ser testado deve estar conectado via cabo USB ao computador, permitindo a comunicação com a ferramenta por meio do protocolo ADB. Adicionalmente, o cliente *desktop* estabelece comunicação com o servidor remoto responsável pelo armazenamento, processamento e recuperação das sessões de teste. Essa comunicação é realizada por meio do protocolo HTTP, utilizando uma interface de programação de aplicações (API) do tipo REST, que permite tanto o envio de novos registros quanto a consulta semântica por gravações e metadados previamente armazenados.

Antes de iniciar a gravação, o sistema solicita o preenchimento de metadados descritivos, incluindo: o nome do responsável pela gravação, uma descrição detalhada do bug a ser analisado, e o nome da aplicação a ser testada. Tais informações são fundamentais para a rastreabilidade, reprodutibilidade e contextualização dos testes durante a busca semântica utilizando NLP. Adicionalmente o Logify coleta automaticamente, via comandos ADB, uma série de informações do dispositivo conectado, como modelo, versão do Android, tamanho de tela, número de série entre outras informações importantes do dispositivo Android conectado. Na tabela I, pode-se observar os principais comandos ADB utilizados para captura de metadados do dispositivo móvel.

Comando ADB	Descrição
adb shell wm size	obter tamanho da tela do dispositivo móvel
adb shell getprop ro.build.version.release	obter versão do android
adb shell getprop.ro ro.build.version.sdk	obter api level
adb shell getprop ro.product.model	obter nome do dispositivo móvel
adb shell getprop ro.product.device	obter tipo de dispositivo móvel
adb shell getprop ro.serialno	obter serial number do dispositivo móvel

Tabela I
COMANDOS ADB EMPREGADOS NA EXTRAÇÃO DE METADADOS DO DISPOSITIVO MÓVEL

Após a coleta dos metadados, o Logify executa um processo de inicialização da aplicação informada, que envolve a limpeza do estado da aplicação por meio de comandos ADB e, em seguida, sua execução. A partir desse ponto, o sistema passa a escutar os eventos do dispositivo por meio do comando `adb shell getevent -lt`, armazenando informações como coordenadas dos toques(ABS_MT_POSITION_X e ABS_MT_POSITION_Y), eventos de toques(BTN_TOUCH)

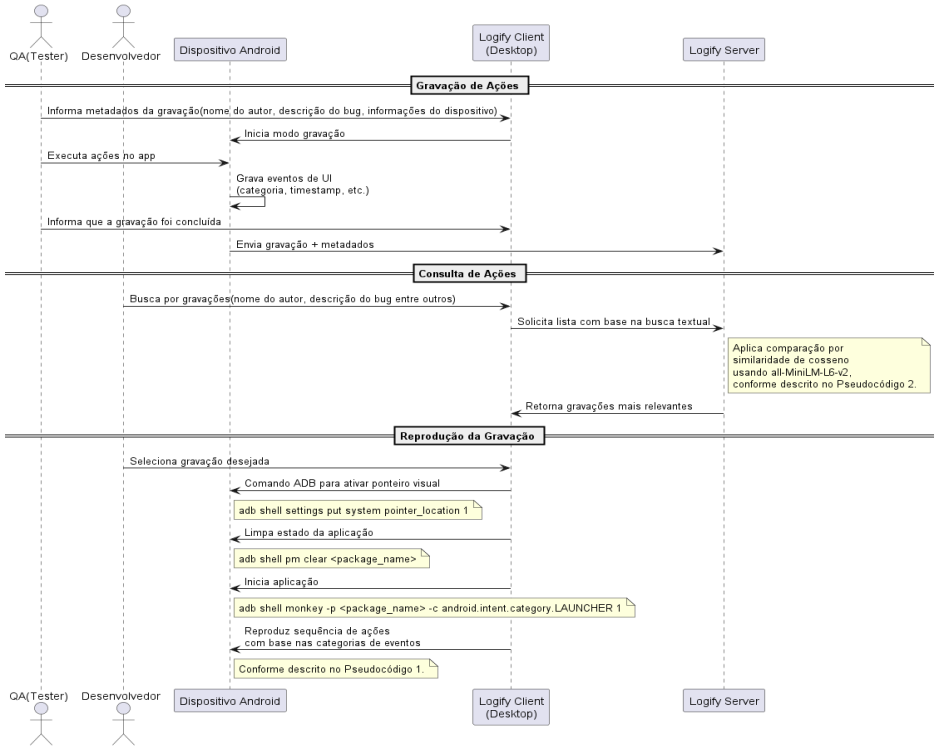

Figura 1. Diagrama de sequência ilustrando das ações de gravação, reprodução e consulta do Logify

e marcações temporais(*timestamps*). Esses eventos são oriundos do subsistema de entrada do *kernel* Linux, base do sistema operacional Android, que expõe eventos de entrada no dispositivo móvel. Cada linha capturada pelo comando `adb shell getevent -lt` representa uma estrutura `input_event`, conforme definido em [25]. Essa padronização permite interpretar de forma precisa interações complexas como toques simultâneos e gestos multi-toques.

Ao finalizar a gravação das ações o arquivo de logs brutos serão enviados ao servidor remoto junto aos metadados associados, onde será tratado e salvo em uma estrutura Javascript Object Notation(JSON), de modo que agrupe os metadados com as ações gravadas durante a execução do teste em uma estrutura padronizada que agrupa categoria da ação (clique, deslizamento ou rolagem), tempo de deslizamento ou rolagem e tempo da ação como um todo. Após o processamento desse arquivo ele é salvo no servidor para posterior consulta via NLP.

As gravações das execuções dos testes, juntamente com seus respectivos metadados, são persistidas no servidor remoto, permitindo que desenvolvedores as consultem posteriormente com base em critérios como descrição do bug, nome da aplicação e data de ocorrência. Durante a consulta, os parâmetros fornecidos pelo usuário são comparados semanticamente com os metadados armazenados, utilizando vetores de representação textual gerados pela biblioteca *all-MiniLM-v6*[17]. A similaridade entre as descrições é quantificada por meio da métrica de cosseno, possibilitando a ordenação das gravações mais pariformes de acordo com a proximidade semântica como descrito no Pseudocódigo 1.

Algorithm 1 Pseudocódigo 1

```
c ← consulta
s ← scripts_servidor
l_r ← lista_registros
l_d ← lista_descricao
for all s_1 ∈ s do
    if s_1.extensao = JSON then
        d ← json.loads(s_1)
        r ← RecordModel.to_record_model(d)
        l_d.adicionar(r.descricao)
        l_r.adicionar(r)
    end if
end for
m ← model_MiniLM_L6_v2()
c_b ← m.calcularEmbedding(c)
r_b ← m.calcularEmbedding(l_d)
scores ← calcularSimilaridade(c_b, r_b)
return listaReproduçãoOrdenada(l_r, scores)
```

O cliente *desktop* exibe ao desenvolvedor uma lista de gravações ordenadas por critérios de relevância, permitindo a seleção da gravação desejada com base na descrição, autor, data, versão do Android e características do dispositivo. Após a escolha, o Logify ativa um ponteiro visual do sistema utilizando o comando `adb shell settings put system pointer_location 1`, o que possibilita o acompanhamento detalhado da reprodução do erro. A seguir,

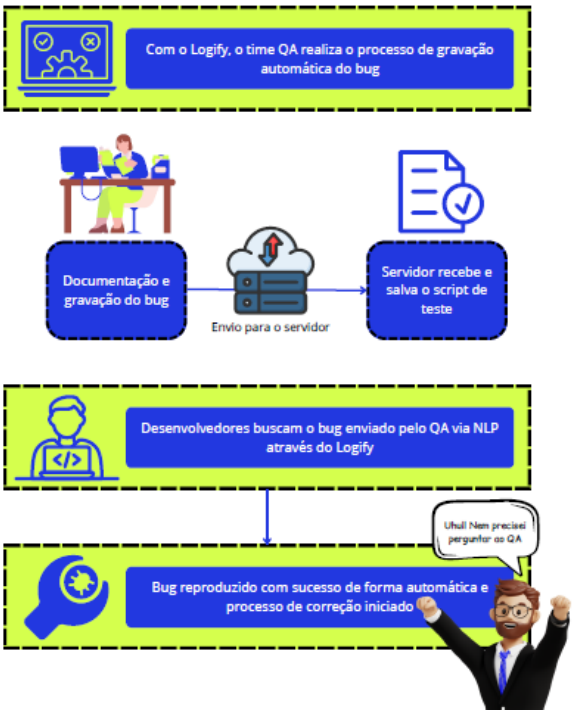

Figura 2. Fluxo de documentação do defeito até a reprodução automática com o uso do Logify

o framework limpa o estado e abre a aplicação a ser testada, tal procedimento visa garantir que o ambiente de execução seja equivalente ao contexto original da gravação, mitigando possíveis efeitos colaterais decorrentes de dados residuais ou inconsistentes. Posteriormente à abertura da aplicação, o Logify executa as ações previamente registradas, respeitando as categorias dos eventos, conforme descrito no Pseudocódigo 2.

Algorithm 2 Pseudocódigo 2

```
habilitarPonteiroVisual()
for all e ∈ tc.eventos do
    x1 ← e.x1
    y1 ← e.y1
    x2 ← e.x2
    y2 ← e.y2
    tempo_swipe ← e.tempo_swipe
    tempo_total ← e.tempo_total
    if e.tipo = tap then
        executar_tap(x1, x2)
    else if e.tipo = swipe then
        executar_swipe(x1, y1, x2, y2, tempo_swipe)
    end if
    aguardar(tempo_total)
end for
desabilitarPonteiroVisual()
```

Dessa forma, como ilustrado na Figura 2, é possível utilizar o framework proposto para gravar ações em aplicativos Android, além de associar seus respectivos metadados como nome do usuário responsável pela criação do teste, descrições dos *bugs*, *links* para *issues* em versionadores de código, tamanho da tela, versão do Android, número de série entre outras informações. Após serem gravadas, essas ações serão disponibilizadas em um servidor remoto, para serem consultadas de forma simplificada através de um cliente *desktop* utilizando linguagem natural e executadas no dispositivo móvel sem a necessidade de permissões elevadas de sistema operacional, assim mantendo um fluxo simplificado, permitindo a colaboração contínua entre desenvolvedores e o time de QA.

IV. Análise dos Resultados

De modo a ilustrar a eficiência do framework proposto nesse trabalho, foi conduzido um experimento com o objetivo de avaliar o impacto da ferramenta Logify no tempo de compreensão de reprodução de bugs em aplicativos Android. Para garantir o equilíbrio na dificuldade das tarefas e controlar variáveis individuais, foi adoltado um delineamento cruzado (*cross-over-design*).

O experimento proposto foi composto por um grupo de 6 desenvolvedores, de modo que cada participante avaliou o entendimento dos bugs em dois subconjutos(denominados A e B), compostos por cinco bugs cada, totalizando 10 bugs por participante. A condição de uso da ferramenta Logify foi alternada entre os participantes. Por exemplo, três participantes resolveram inicialmente o subconjunto A com o Logify e o subconjunto B sem a ferramenta, enquanto os outros três seguiram a ordem inversa: iniciaram o subconjunto A sem o Logify e o subconjunto B com a ferramenta. Essa alternância garantiu que todos os bugs fossem avaliados tanto com quanto sem o framework proposto, controlando a ordem de exposição às tarefas. Os resultados obtidos com o experimento estão organizados na Figura 3 que apresenta o tempo, em segundos, que cada desenvolvedor levou para compreender os passos para reprodução dos bugs com e sem a ferramenta.

Esse delineamento permitiu uma comparação direta do desempenho entre as duas condições experimentais, minimizando o impacto de fatores externos como familiaridade com os bugs, ordem de execução ou diferenças individuais entre os participantes. Desse modo, o Logify reduziu o tempo médio de entendimento para reprodução de bugs em 31.22%. A redução percentual entre os tempos médios dos grupos que utilizaram o Logify (M_Logify) e os grupos que não utilizaram a ferramenta (M_Controle) foi calculado com base na seguinte relação matemática:

$$\text{Redução média (\%)} = \left(\frac{M\_Controle - M\_Logify}{M\_Controle}\right) \times 100$$

$$\text{Redução média (\%)} = \left(\frac{213{,}578 - 146{,}893}{213{,}578}\right) \times 100 \approx 31{,}22\%$$

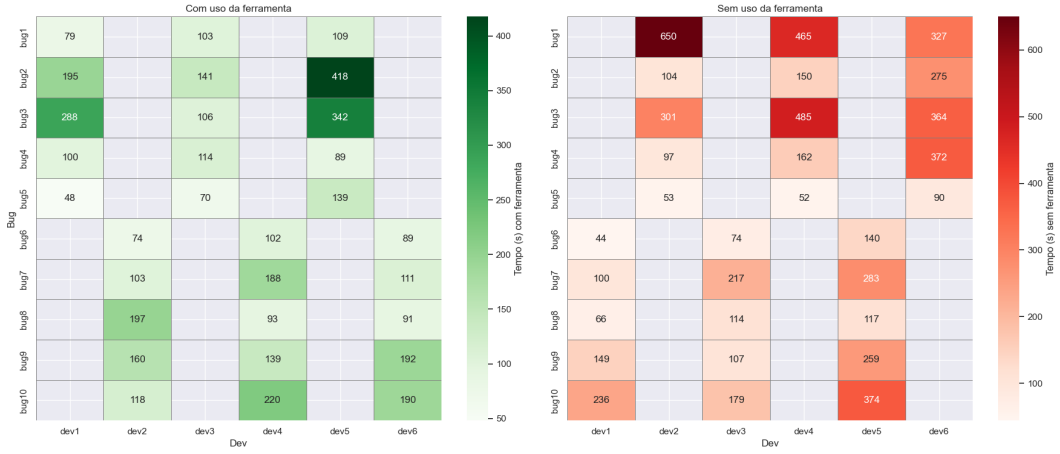

Figura 3. Resultados obtidos com o experimento com e sem o uso do Logify

A análise estatística foi conduzida para investigar a diferença no tempo de entendimento de bugs entre desenvolvedores que utilizaram a ferramenta Logify e aqueles que não utilizaram. Inicialmente, foi aplicado o teste t de Student, representado pela equação 1, assumindo homogeneidade de variâncias e rejeitando a hipótese nula. O resultado indicou uma diferença significativa entre os grupos(t = -2.43, p = 0.019, gl = 58), com intervalo de confiança de [-121,86; -11,61] segundos para 95% de nível de confiança, sugerindo que o grupo com o Logify levou menos tempo para compreender os bugs, indicando que a ferramenta proporcionou uma melhora consistente no tempo de entendimento. A estatítica do teste t de Student foi calculada por:

Desvio padrão combinado(pooled standard derivation):

$$S_p = \sqrt{\frac{(n_1 - 1) \cdot s_1^2 + (n_2 - 1) \cdot s_2^2}{n_1 + n_2 - 2}} \tag{1}$$

Erro padrão da diferença entre médias:

$$SE = S_p \cdot \sqrt{\frac{1}{n_1} + \frac{1}{n_2}} \tag{2}$$

Teste T Student para amostras independentes:

$$t = \frac{\bar{X}_1 - \bar{X}_2}{S_p \cdot \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}} \tag{3}$$

Graus de liberdade:

$$g1 = n_1 + n_2 - 2 \tag{4}$$

Considerando a diferença substancial entre os desvios padrão dos grupos (SD_Logify = 61.49; SD_Controle = 132.25), foi aplicado também o teste t de Welch, mais adequado em contextos de variâncias heterogêneas. O resultado permaneceu estatisticamente significativo (t=-2.43, p = 0.019, gl = 45.03), com um nível de confiança de 95%, a ferramenta Logify permitiu que os desenvolvedores compreendessem o bug em um intervalo de confiança de [-119,74; -13,63] segundos, é notorio que a diferença calculada pelo teste de t de Welch é consideravelmente próximo do valores encontrados no teste de t Student, o que reforça que a ferramenta proporcionou uma melhora consistente no tempo de entendimento. A estatítica do teste t de Welch foi calculada por:

Teste T de Welch:

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} \tag{5}$$

Graus de liberdade aproximados(gl de Welch):

$$gl \approx \frac{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2}{\frac{\left(\frac{s_1^2}{n_1}\right)^2}{n_1 - 1} + \frac{\left(\frac{s_2^2}{n_2}\right)^2}{n_2 - 1}} \tag{6}$$

O resultado do experimento pode ser observado na Figura 4, onde foram geradas curvas normais para os grupos com e sem uso da ferramenta. Observou-se que a curva referente ao grupo com Logify apresenta-se mais concentrada, refletindo uma menor variabilidade nos tempos de entendimento. Em contraste, a curva do grupo controle é mais dispersa, indicando maior variabilidade entre os participantes.

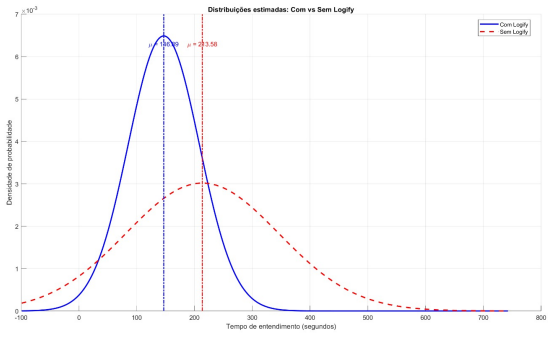

Figura 4. Comparação das curvas normais para os grupos com e sem uso da ferramenta

Na comparação entre as distribuições obtidas pelos testes t de Student e t de Welch, observou-se que ambas conduzem a mesma conclusão estatisticamente significativa, com intervalos de confiança bastante semelhantes. Essa convergência ocorre, em parte, pelo tamanho equilibrado das amostras e pela robustez do teste de Welch frente aos cenários de heterocedasticidade. Embora o teste t de Student pressuponha variâncias iguais, a aplicação do teste t de Welch garante maior segurança metodológica diante de evidente diferença entre os desvios padrão dos grupos. A Figura 5 ilustra visualmente como a variabilidade entre os grupos afeta a forma de distribuição t comparando o teste t de Student com o de Welch.

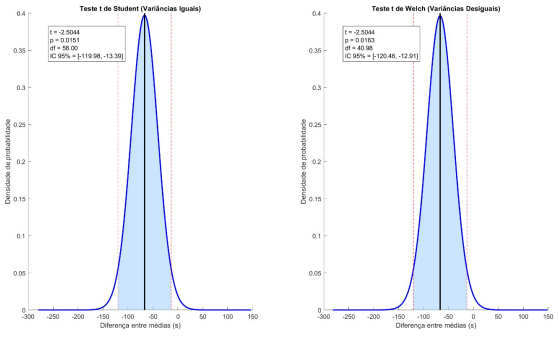

Figura 5. Gráficos em comparação ao teste t de Student e ao teste t de Welch

Além da significância estatística, foi calculado o tamanho do efeito utilizando o índice de Cohen (Cohen's d), obtendo-se um valor de $|d| = 0.68$. Esse resultado corresponde a um tamanho de efeito moderado a alto, segundo os critérios propostos por Cohen[26], indicando um impacto substancial da ferramenta sobre o desempenho dos desenvolvedores.

Durante o experimento, além das análises quantitativas, foram coletadas observações qualitativas comportamentais. Após declararem que haviam entendido o bug, os particpantes foram convidados a reproduzi-lo manualmente, como forma de validar a compreensão real da falha. Notou-se que, mesmo entre os usuários da ferramenta, alguns não conseguiam reproduzir o bug de forma imediata, evidenciando lacunas no entendimento prático.

Com base nesse comportamento, foi identificado que a inclusão de um modo guiado como melhoria futura da ferramenta, pode beneficiar ainda mais a eficácia do entendimento dos passos para reprodução da falha. Esse modo permitirá que o QA adicione anotações específicas, passos esperados ou comentários que serão exibidos ao desenvolvedor durante a reprodução do bug na tela do dispostivo móvel como uma sobreposição visual interativa, possibilitando:

- Avançar e recuar entre passos
- Visualizar dicas contextuais (”Clique aqui”, ”Acesse a aba”, etc.)
- Marcar ações como concluídas ou pendentes

V. Conclusão

Este estudo apresentou uma plataforma para realizar R&R em dipositivos Android sem modificar o aplicativo ou requisitar permissões elevadas do sistema operacional, acompanhada de uma interface interativa que visa facilitar o processo de R&R dos cenários de teste. A ferramenta permite a gravação, reprodução e análise de casos de teste, utilizando comandos ADB e metadados, visando sua posterior consulta utilizando NLP.

Os resultados do experimento mostraram que embora tenha sido conduzido com uma amostra reduzida de desenvolvedores, o delineamento experimental controlado, a natureza realista das tarefas propostas e a aplicação de testes estatísticos como o teste t de Welch, cálculo de intervalos de confiança e d de Cohen, conferem robustez às análises. O experimento foi suficiente para detectar diferenças estatisticamente significativas com 95% de confiança.

Nos experimentos conduzidos, observou-se que o tempo médio para compreender um bug sem o uso da ferramenta foi de 3.55 minutos, enquanto com o Logify esse tempo caiu para 2.44 minutos, resultando em uma redução média de 1.11 minutos por bug, correspondente a um ganho de eficiência relativa de aproximadamente 31.22%. Espera-se que em bugs de maior complexidade, esse ganho absoluto possa ser ainda mais expressivo, potencializando o impacto da ferramenta.

Para trabalhos futuros, pretende-se expandir a plataforma com novas funcionalidades, incluindo suporte a diferentes tamanhos e densidades de tela, inspirado em trabalhos como o Mosaic[27], superando uma das limitações atuais da ferramenta, bem como a implementação de um modo guiado, que orientará de forma interativa o passo a passo para o usuário durante a execução de testes. Embora os resultados indiquem um efeito claro da ferramenta Logify na redução do tempo de entendimento dos bugs, estudos futuros com amostras maiores podem ajudar a refinar a estimativa da diferença entre grupos e validar os achados em cenários mais amplos, bem como investigar se o impacto observado se mantém em cenários mais diversos com maior variação de perfis de desenvolvedores, tipos de bugs e ambientes de trabalho.

Acknowledgment

Este artigo é resultado do projeto de Pesquisa e Desenvolvimento DART, realizado pelo Instituto Sidia de Ciência e Tecnologia, em parceria com a Samsung Eletrônica da Amazônia Ltda. Publicidade em conformidade com o disposto no artigo 39 do Decreto nº 10.521/2020.

Referências

[1] Y. Choi, A. Seo, and H.-S. Kim, “Scriptpainter: Vision-based, on-device test script generation for mobile systems,” in *2022 21st ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pp. 477–490, IEEE, 2022.

[2] Statista, “Combined global apple app store and google play app downloads from 1st quarter 2015 to 1st quarter 2023,” 2024. Acessado em: 2025-02-11.

[3] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pp. 308–318, 2008.

[4] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, "Understanding the test automation culture of app developers," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 1–10, IEEE, 2015.

[5] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk, "Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 399–410, IEEE, 2017.

[6] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, and W. G. Halfond, "Recdroid: automatically reproducing android application crashes from bug reports," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 128–139, IEEE, 2019.

[7] S. Feng and C. Chen, "Prompting is all you need: Automated android bug replay with large language models," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pp. 1–13, 2024.

[8] Y. L. Arnatovich, L. Wang, N. M. Ngo, and C. Soh, "Mobolic: An automated approach to exercising mobile application guis using symbiosis of online testing technique and customated input generation," *Software: Practice and Experience*, vol. 48, no. 5, pp. 1107–1142, 2018.

[9] D. Wang, Z. Zhang, S. Feng, W. G. Halfond, and T. Yu, "An empirical study on leveraging images in automated bug report reproduction," *arXiv preprint arXiv:2502.15099*, 2025.

[10] C. Bernal-Cárdenas, N. Cooper, K. Moran, O. Chaparro, A. Marcus, and D. Poshyvanyk, "Translating video recordings of mobile app usages into replayable scenarios," in *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, pp. 309–321, 2020.

[11] Y. Wu, C. Lin, A. Liu, L. Zhao, and X. Zhang, "Crowdsourced bug report severity prediction based on text and image understanding via heterogeneous graph convolutional networks," *Journal of Software: Evolution and Process*, vol. 36, no. 11, p. e2705, 2024.

[12] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 416–419, 2011.

[13] M. Harman, P. McMinn, J. T. De Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," in *LASER Summer School on Software Engineering*, pp. 1–59, Springer, 2008.

[14] P. McMinn, *Evolutionary search for test data in the presence of state behaviour*. PhD thesis, University of Sheffield, 2005.

[15] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing-and touch-sensitive record and replay for android," in *2013 35th international conference on software engineering (ICSE)*, pp. 72–81, IEEE, 2013.

[16] Y. Hu and I. Neamtiu, "Valera: an effective and efficient record-and-replay tool for android," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, pp. 285–286, 2016.

[17] W. et al., "all-minilm-l6-v2: A compact and efficient sentence embedding model." https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2, 2022. Acessado em: 2025-02-10.

[18] R. Devarakonda, G. Palanisamy, and J. Green, "Digitizing scientific data and data retrieval techniques," *arXiv preprint arXiv:1010.3983*, 2010.

[19] I. V. Chilingarian, P. Di Matteo, F. Combes, A.-L. Melchior, and B. Semelin, "The galmer database: galaxy mergers in the virtual observatory," *Astronomy & Astrophysics*, vol. 518, p. A61, 2010.

[20] A. Waswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *NIPS*, 2017.

[21] J. Devlin, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[22] W. Wang, F. Wei, L. Dong, H. Bao, N. Yang, and M. Zhou, "Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers," *Advances in Neural Information Processing Systems*, vol. 33, pp. 5776–5788, 2020.

[23] W. Lam, Z. Wu, D. Li, W. Wang, H. Zheng, H. Luo, P. Yan, Y. Deng, and T. Xie, "Record and replay for android: Are we there yet in industrial cases?," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pp. 854–859, 2017.

[24] "Android debug bridge (adb)." https://developer.android.com/tools/adb?hl=pt-br. Accessed: 2025-04-04.

[25] "Eventos do dispositivo android." https://source.android.com/docs/core/interaction/input/touch-devices?hl=pt-br. Accessed: 2025-04-07.

[26] J. Cohen, *Statistical power analysis for the behavioral sciences*. routledge, 2013.

[27] M. Halpern, Y. Zhu, R. Peri, and V. J. Reddi, "Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 215–224, IEEE, 2015.