Reparo de software com agentes de IA sob orçamento limitado

Terminação, aceitação vinculada à árvore de código e validade do benchmark no desafio Gemma 4

Autor: Aridio Silva, pesquisador independente, Brasil
Citação preferencial: Silva, Aridio (2026) · **ORCID:** 0009-0008-2411-6995
Direitos autorais: © 2026 Aridio Silva
DOI: https://doi.org/10.34740/KAGGLE/W/125650

Resumo

Propomos um controlador de reparo com orçamento limitado e aceitação vinculada à árvore de código. O controlador abstrato exige evidência de um validador separado para a árvore exata que foi congelada antes da entrega. Isso impede que alegações alucinadas de sucesso nos testes autorizem a entrega. Sob hipóteses explícitas de supervisão, o controlador abstrato termina em até 4M + 1 transições para um orçamento de M tentativas. Uma verificação de estados finitos explorou 556 estados e 953 transições e encontrou um caminho até a aceitação quando o modelo admitiu evidência falsa. Uma auditoria do benchmark público identificou uma tarefa com requisitos de teste obrigatórios incompatíveis e três falsas aceitações em nove casos sintéticos de identidade de teste. No Kaggle, 109 execuções cobriram 86 das 87 tarefas de correção de bugs selecionadas, em sete rodadas. O verificador registrou 37 resoluções (43,0% das tarefas executadas e 42,5% das selecionadas), incluindo 20 de 54 na S54M, o lote executado sob uma configuração comum.

1 Introdução

Um agente de reparo examina um repositório e propõe um diff. O controlador gerencia o orçamento de tentativas, supervisiona a validação e vincula a autorização à árvore de código examinada pelo validador. Aplicamos princípios do programa de pesquisa SGAEIA [1] ao reparo de software com Gemma 4 [2].

Apresentamos três contribuições.

1. **Terminação e aceitação vinculada à árvore.** Provamos um limite de transições e um invariante que vincula a aceitação à árvore exata validada. Uma verificação de estados finitos produz um contraexemplo quando o controlador admite evidência falsa.
2. **Validade do benchmark.** Documentamos uma tarefa cujos testes obrigatórios não podem ser satisfeitos em conjunto sob o perfil público fixo e mostramos como um comparador de identidades pode aceitar o resultado de outro teste.
3. **Avaliação em tarefas públicas de desenvolvimento.** Relatamos 37 resoluções em 86 tarefas de correção de bugs executadas em três repositórios. Apresentamos separadamente o resultado das 54 tarefas executadas sob uma configuração comum e analisamos os resultados do verificador.

2 Trabalhos relacionados

Sistemas de reparo por geração e validação, como o GenProg, usam testes para avaliar patches candidatos [15]. Qi et al. distinguem o sucesso na suíte de testes da correção do patch em sentido mais amplo [4]. O SWE-agent fornece uma interface para interagir com repositórios e ferramentas [16]. O Agentless divide o processo em localização, reparo e validação [17], e o AutoCodeRover usa a estrutura do programa para orientar a busca no código [18]. Nosso controlador formaliza a decisão de entrega em um fluxo de reparo com orçamento limitado ao vincular a evidência do validador à árvore entregue.

Algoritmos certificadores acompanham seus resultados de evidências verificáveis [5], o SWE-Proof estuda contratos de reparo verificados por máquina [6] e o in-toto vincula artefatos a etapas da

cadeia de suprimentos de software [20]. A propriedade aqui estudada determina quando um controlador pode entregar um reparo. O SWE-bench avalia o reparo em problemas reais de repositórios [3], e o SWE-bench Verified seleciona tarefas segundo a qualidade da avaliação [19]. Para o perfil público de desenvolvimento do Gemma 4, examinamos se todos os testes obrigatórios podem passar em conjunto e se o verificador reconhece as identidades dos testes exigidos.

3 Controlador de reparo

O perfil de avaliação fixa a tarefa, o commit de base, os caminhos permitidos, as dependências e o critério de aceitação. No projeto do controlador, cada tentativa é debitada do orçamento antes de acionar o gerador. O supervisor aplica o diff em uma cópia isolada e congela a árvore resultante. Um validador separado registra seu veredito e as identidades da tarefa, do perfil, da tentativa e da árvore. No caminho baseado em testes, o critério de aceitação exige que o comando de teste termine com sucesso e que cada nó parametrizado obrigatório passe. Falhas em outros pontos da suíte do verificador também podem impedir a aceitação. No caminho formal, o validador verifica a evidência de um contrato explícito.

Para impedir que uma alegação alucinada de sucesso nos testes autorize a entrega, o supervisor obtém a evidência de aceitação do validador separado. Ele verifica se a evidência identifica a tentativa ativa e a árvore congelada e confirma a entrega dessa mesma árvore dentro do prazo autorizado. A Seção 4 prova essa propriedade para o controlador abstrato. A Seção 5 documenta a aplicação dos patches, os vereditos do verificador e as evidências de teste registradas nas execuções do Kaggle.

Uma tentativa pode terminar sem patch, após uma falha de aplicação, com evidência rejeitada ou indisponível, ou sem confirmação de entrega no prazo. Nesses casos, o supervisor descarta o candidato, revoga qualquer autorização ativa e verifica novamente o orçamento. Se ainda houver uma tentativa e reserva suficiente, o controlador debita outra tentativa e retorna à geração. Caso contrário, registra ABSTENÇÃO (ABSTENTION).

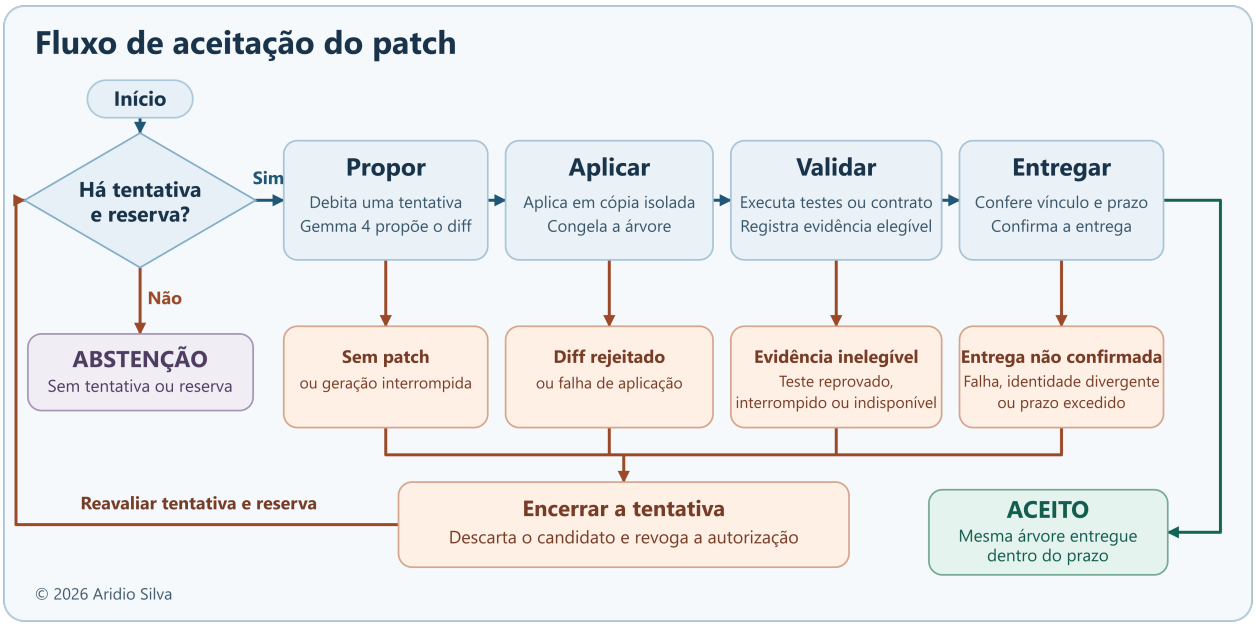


Figura 1. Fluxo do controlador abstrato de reparo. Uma tentativa encerrada sem aceitação retorna à verificação de tentativas e reserva. O estado ACEITO (ACCEPTED) exige a entrega da mesma árvore dentro do prazo. A S54M usou um orçamento operacional de uma sessão por tarefa (Seção 5).

Correspondência com o executor da S54M

Na S54M, o executor avaliou patches sobre snapshots com hashes verificados, registrou os hashes dos patches e verificou separadamente as identidades dos nós JUnit obrigatórios. O resultado medido foi o veredito do verificador. A garantia de entrega da mesma árvore, provada na Seção 4, aplica-se ao controlador abstrato.

4 Terminação e aceitação vinculada à árvore

O estado de controle é (r, b, q), em que r é o número de tentativas restantes, b é a reserva abstrata e q é a fase. As fases não terminais são pronto (ready), propor (propose), verificar (verify) e publicar (publish). O estado inicial é pronto, com r = M. Na instância verificada, o início exige b ≥ 4 para cobrir os quatro débitos de uma tentativa completa. Uma tentativa consome uma unidade de r antes de acionar o gerador. O supervisor retoma o controle quando cada operação termina ou é interrompida e executa um próximo passo permitido em tempo finito.

A função de ranking mede o orçamento de transições restante. Cada tentativa ainda não iniciada contribui com quatro transições. O peso da fase indica quantas transições podem restar na tentativa em andamento. O início consome uma transição e deixa até três outras, correspondentes aos resultados da geração do patch, da validação e da entrega. Uma falha antecipada devolve o controle à fase pronto mais cedo.

Definimos R(r, q) = 4r + w(q), com w igual a 0 em pronto, 3 em propor, 2 em verificar e 1 em publicar. Iniciar uma tentativa consome uma unidade de r e aumenta o peso da fase de 0 para 3, de modo que a variação de R é 4(−1) + 3 = −1. Avançar para a próxima fase não terminal reduz w em uma unidade. Retornar à fase pronto após uma falha zera w sem restaurar r. Assim, cada transição entre estados não terminais reduz R em pelo menos uma unidade.

R começa em 4M e é não negativo. Portanto, pode haver no máximo 4M transições entre estados não terminais e uma transição final. Com reserva inicial de pelo menos 4M, o caminho de pior caso usa todas as M tentativas. Cada tentativa leva quatro transições, falha na entrega e retorna à fase pronto. A transição final registra ABSTENÇÃO, atingindo o limite de 4M + 1. Um limite de tempo decorrido também exige limites para a duração das operações, incluindo supervisão, cancelamento e limpeza.

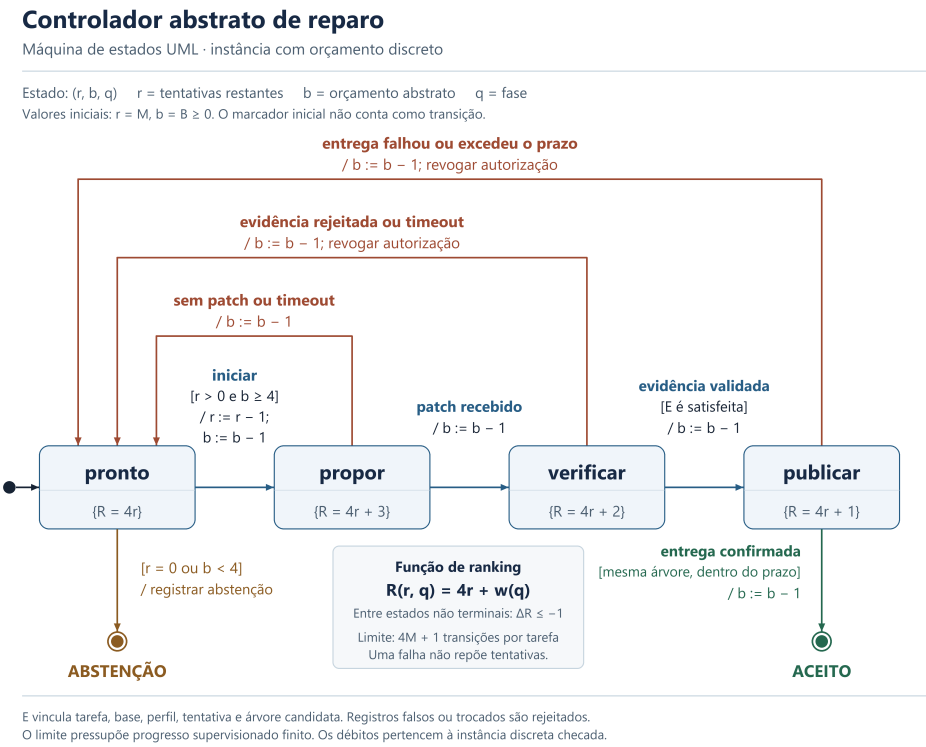


Figura 2. Modelo discreto do controlador. A fase verificar abrange a aplicação e a validação. E denota evidência elegível vinculada à árvore candidata.

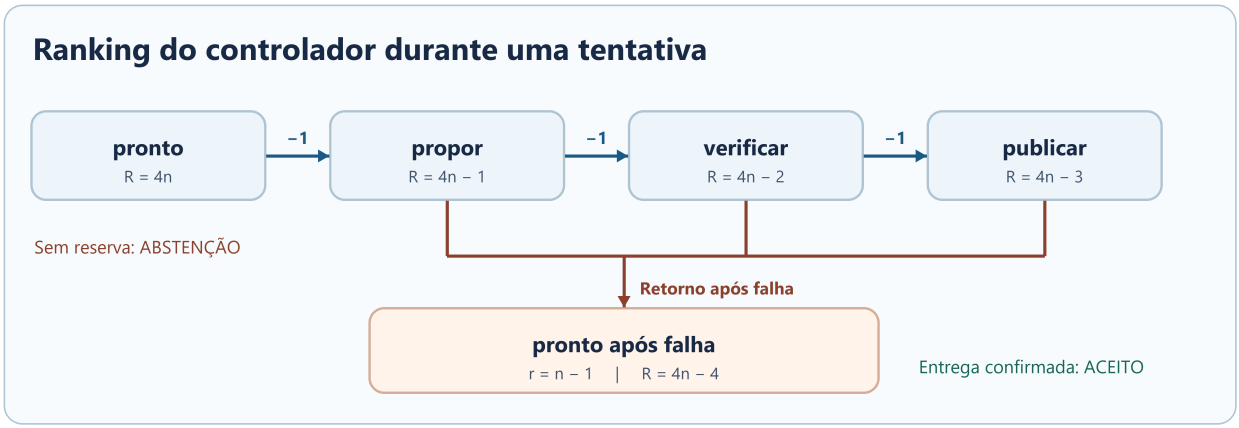


Figura 3. A função de ranking decresce quando o controlador avança pelas fases e quando retorna após uma falha. Aqui, n é o valor de r no início da tentativa.

O invariante de aceitação vinculada à árvore estabelece que toda transição para ACEITO confirma a entrega no prazo de uma árvore congelada, respaldada por evidência elegível emitida pelo validador autorizado para a tentativa ativa. A transição de verificar para publicar verifica essa evidência e fixa a identidade da árvore. A transição de publicar para ACEITO confirma a entrega da mesma árvore no prazo. Uma falha revoga a autorização antes que outra tentativa possa começar. Essas regras preservam o invariante por indução sobre as transições.

No caminho baseado em contrato, a garantia depende da solidez do verificador. No caminho baseado em testes, a aceitação atesta o resultado da suíte sob o perfil registrado. Se a evidência exigida for inalcançável nesse perfil, ABSTENÇÃO é o único estado terminal alcançável.

5 Avaliação

Verificação do modelo e piloto

Uma busca em largura em Python 3.12 partiu de 143 combinações iniciais de r entre 0 e 10 e b entre 0 e 12. Ela explorou 556 estados monitorados e 953 transições, verificando o orçamento, os estados sucessores, o decrescimento da função de ranking e a condição de evidência em ACEITO. Quando permitimos que um evento de evidência falsa levasse a publicar, a busca encontrou um caminho até ACEITO sem evidência elegível. O verificador do modelo cobre essa instância finita [13]. A prova por função de ranking cobre qualquer M finito sob as hipóteses de supervisão estabelecidas.

A política P instrui o agente a mapear requisitos para obrigações de validação, preferir um subconjunto estrito de obrigações entre planos comparáveis nos demais aspectos e invalidar as evidências afetadas por edições. No piloto sintético de paginação, o fluxo gerou e aplicou um patch que passou nos dois testes [8]. A avaliação no Kaggle usou P em todas as rodadas R1–R6 e na S54M [10].

Seleção de tarefas e desenho das rodadas

Classificamos as 129 tarefas públicas [7] segundo o trabalho predominante descrito em cada uma. O catálogo do estudo inclui 87 bugs, 21 funcionalidades, 10 tarefas de refatoração ou manutenção, cinco tarefas mistas, três de documentação, duas de testes ou automação e uma de desempenho. Selecionamos os bugs por formarem o maior grupo, com 67,4% do catálogo. A execução em lotes sequenciais permitiu investigar interrupções e ajustar a infraestrutura de execução dentro da cota disponível. As seis rodadas iniciais, R1–R6, cobriram 33 bugs catalogados, dos quais 32 foram executados. A S54M incluiu os outros 54 IDs de tarefa, sem sobreposição [10].

Nas rodadas R1–R6, reexecutamos 16 tarefas não resolvidas enquanto ajustávamos o executor. As rodadas usaram limites de 24, 30 ou 35 requisições. A configuração comum da S54M usou o checkpoint gemma-4-31b-it-qat-w4a16-ct (revisão 2 do Kaggle), vLLM 0.19.1, temperatura de 0,2 e contexto de 32.768 tokens em quatro GPUs NVIDIA L4. Ela permitia até 30 requisições em uma

sessão do agente por tarefa, sem nova tentativa automática [10]. As transições de nova tentativa são exercitadas no modelo discreto do controlador [13].

Na S54M, o tempo do processo de cada tarefa, incluindo a limpeza, teve mediana de 317,5 s e máximo de 869,9 s. A célula inteira executou por 5 h 34 min 42 s, com média de 6 min 12 s por tarefa para as 54 tarefas [10]. Esses tempos de desenvolvimento incluem a validação do patch. A trilha de predição exclui a validação do patch do orçamento de 12 horas do agente [14].

Cada tarefa foi contada uma vez, com o último resultado registrado em R1–R6 ou a única execução na S54M. As 109 execuções incluem 23 reexecuções. O executor usou o verificador do swegemma, pacote de software que executa a avaliação de patches do benchmark, com o perfil público de desenvolvimento. Para as 37 resoluções, os extratos congelados registram um patch aplicado, concordância entre os resultados registrados do verificador e execução bem-sucedida dos testes [10]. Os registros originais das execuções também documentam a confirmação das identidades JUnit obrigatórias por uma checagem separada do executor. Para a análise de falhas, comparamos os IDs obrigatórios do diff de testes com as falhas e os erros registrados pelo pytest, usando a raiz de coleta [10].

Resoluções e falhas observadas

Escopo	Selecionadas	Executadas	Resolvidas	% das executadas
Total por tarefa	**87**	**86**	**37**	**43,0%**
R1–R6, última execução	33	32	17	53,1%
S54M, configuração comum	54	54	20	37,0%

Considerando FastAPI, Rich e Requests, o verificador produziu vereditos utilizáveis para 81 das 86 tarefas executadas. As 37 resoluções representam 43,0% das tarefas executadas e 42,5% das selecionadas, incluindo a tarefa adiada neste último denominador. O intervalo nominal de Wilson de 95% para 37/86, de 33,1% a 53,6%, fornece uma referência descritiva para o conjunto observado. O FastAPI teve 19 resoluções em 36 tarefas executadas, o Rich teve 12 em 39 e o Requests teve 6 em 11 [10].

As 86 tarefas corresponderam a 109 execuções, com 55 registros para 32 tarefas em R1–R6 e 54 registros para 54 tarefas na S54M. Na ordem, R1 a R6 registraram 0/2, 0/1, 1/6, 13/26, 3/15 e 0/5 resoluções por execução. Das 17 resoluções em R1–R6, 14 ocorreram na primeira execução registrada e três em R5, quando as tarefas foram reexecutadas após mudanças no executor. Essas três estão incluídas nas 37 resoluções do estudo. A S54M produziu 20 resoluções sob uma configuração comum. A seleção de tarefas por lote, as mudanças de configuração e a reexecução seletiva de 16 tarefas limitam as comparações de desempenho entre rodadas [10].

Das 49 tarefas não resolvidas, 44 produziram patches que foram aplicados e rejeitados, 14 em R1–R6 e 30 na S54M. Um teste obrigatório falhou em 36 casos, 13 e 23, respectivamente. Em sete casos da S54M, foram identificadas falhas fora dos IDs obrigatórios. Uma rejeição das rodadas iniciais tinha informações insuficientes sobre a identidade dos testes para ser classificada. Das cinco tarefas sem veredito utilizável, uma terminou sem patch em R1–R6. Na S54M, o mecanismo de proteção de memória interrompeu dois testes e o lote prosseguiu. Em outro caso, após a aplicação do patch, um erro de sintaxe impediu a importação e o verificador retornou o código de erro de avaliação −1. No caso restante, a saída truncada deixou o patch vazio [10].

Validade do benchmark

Identidade dos testes. Em nove casos sintéticos de JUnit, o comparador do swegemma 0.2.11 aceitou indevidamente três divergências de módulo, nome de função ou parâmetro. Um comparador estrito rejeitou essas três divergências no mesmo experimento local. O registro congelado do JUnit documenta cada identidade exigida e os resultados de ambos os comparadores [12].

Satisfatibilidade do perfil. A tarefa `fastapi_14186` permaneceu no inventário como adiada. O perfil público fixo exige `test_model_description_escaped_with_formfeed`, habilitado apenas com Pydantic v1, e `test_model_field_default_required`, habilitado apenas com v2. É impossível que ambos passem no

mesmo ambiente fixo. Um diagnóstico local com o patch de referência registrou 52 casos e três testes pulados sob Pydantic 2.13.4. O pytest encerrou com código zero, e o verificador rejeitou o resultado porque um teste obrigatório foi pulado. O diagnóstico e o veredito congelados documentam as condições e os resultados dos testes [7, 9].

Robustez operacional

Em uma reprodução local, três arrays malformados excederam o limite de cinco segundos no parser do vLLM 0.19.1. A camada de compatibilidade retornou um erro explícito para cada um, e uma entrada de controle com delimitadores corretos obteve resposta nas duas versões. O registro congelado do parser documenta esses casos [11]. Na S54M, o mecanismo de proteção de memória interrompeu duas árvores de processos de teste e o lote foi concluído, conforme documentado nos registros operacionais congelados [10]. O Apêndice A vincula riscos selecionados de plataforma e avaliação às respectivas evidências congeladas.

6 Discussão e conclusão

O controlador e a auditoria do benchmark resultam em duas regras de projeto para o reparo de software com agentes. Primeiro, vincular a decisão de entrega à evidência de um validador autorizado para a árvore exata a ser entregue. Isso impede que uma alegação alucinada de sucesso nos testes autorize a entrega. Segundo, verificar se todos os testes obrigatórios podem passar sob o perfil configurado e se o verificador reconhece suas identidades completas. A tarefa adiada e o experimento com JUnit mostram como cada checagem afeta o que um veredito do benchmark permite concluir.

As execuções com Gemma 4 registraram 37 resoluções em 86 tarefas de correção de bugs executadas, incluindo 20 no lote S54M de 54 tarefas sob uma configuração comum. Os resultados de terminação e aceitação do controlador aplicam-se a fluxos de reparo capazes de impor o mesmo orçamento de tentativas, a mesma supervisão e as mesmas verificações de evidência.

Uma execução de referência (baseline) nas mesmas 54 tarefas da S54M, com o mesmo modelo, orçamento e verificador, permitiria uma estimativa pareada do efeito da política P sobre a resolução de tarefas. Estamos ampliando a avaliação para os demais tipos de tarefa no conjunto público de desenvolvimento de 129 tarefas. À medida que essas execuções forem concluídas e verificadas, relataremos os resultados por tipo de tarefa em uma revisão posterior.

Rastreabilidade. O suplemento congelado contém o inventário, 109 registros de execução, dois eventos de proteção de memória e os hashes dos sete ZIPs de origem. O script de verificação verifica os extratos e recalcula os totais de tarefas, as taxas de resolução e as contagens de falhas em testes obrigatórios. Os hashes dos arquivos compactados e de seus membros vinculam os extratos aos registros originais preservados [10].

Referências

[1] Silva, Aridio. SGAEIA, programa de pesquisa e arquitetura de referência aberta. Site do projeto.

[2] Gemma Team (2026). Gemma 4 Technical Report.

[3] Jimenez, C. E. et al. (2024). SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. ICLR.

[4] Qi, Z. et al. (2015). An Analysis of Patch Plausibility and Correctness for Generate-And-Validate Patch Generation Systems. ISSTA.

[5] McConnell, R. M. et al. (2011). Certifying Algorithms. Computer Science Review.

[6] Ma, G. et al. (2026). SWE-Proof: Can Language Models Resolve Real-World Issues with Machine-Checked Proofs?.

[7] Kaggle (2026). Dados da competição Gemma 4 Developer Agent. SHA-256 de tasks.jsonl e4b3fd60f69dbc2b9213e54eeb9636db78aefe92c1d06269d73d9f5f8f3c8ad6.

[8] Silva, Aridio (2026). Piloto sintético de paginação, extratos dos registros. Extrato congelado, 7 de novembro.

[9] Silva, Aridio (2026). fastapi_14186, perfil público e veredito. Extratos de teste e avaliação, 7 de novembro.

[10] Silva, Aridio (2026). Suplemento congelado das execuções no Kaggle até 11 de novembro. GitHub, paper-evidence-v1.2.0. SHA-256 do ZIP da S54M d9b1145ba1c78582797d29a0d1bd16466c8e9e1f0b2170d635374693d0aa3d60.

[11] Silva, Aridio (2026). Registro dos testes com arrays malformados no vLLM 0.19.1 e com a camada de compatibilidade. 4 de novembro.

[12] Silva, Aridio (2026). Registro dos testes sintéticos de identidade JUnit. 2 de novembro.

[13] Silva, Aridio (2026). Verificador de estados finitos do controlador de reparo. Versão congelada das evidências.

[14] Kaggle (2026). Regras de avaliação e de tempo da competição Gemma 4 Developer Agent.

[15] Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W. (2012). GenProg: A Generic Method for Automated Software Repair. IEEE Transactions on Software Engineering.

[16] Yang, J. et al. (2024). SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering.

[17] Xia, C. S., Deng, Y., Dunn, S., and Zhang, L. (2024). Agentless: Demystifying LLM-based Software Engineering Agents.

[18] Zhang, Y. et al. (2024). AutoCodeRover: Autonomous Program Improvement.

[19] OpenAI (2024). Introducing SWE-bench Verified.

[20] Torres-Arias, S. et al. (2019). in-toto: Providing farm-to-table guarantees for bits and bytes. USENIX Security.

[21] Silva, Aridio (2026). Catálogo R01–R72 de riscos de plataforma e ambiente no desafio Gemma 4. Edição em inglês. Versão congelada v1.2.2.

Apêndice A Riscos de plataforma e avaliação

O relatório R01–R72 [21] cataloga riscos ligados à operação da plataforma, a dependências, à avaliação e a decisões de projeto. Os casos abaixo remetem às respectivas evidências congeladas.

R65, parser sem progresso. Três arrays malformados excederam o limite de cinco segundos no vLLM 0.19.1. A camada de compatibilidade retornou erros explícitos. Registro do parser

R69, testes sensíveis à rede. Duas tarefas passaram em ensaios locais com patches de referência e acesso à rede e foram posteriormente resolvidas no Kaggle [10]. O isolamento estrito de rede não foi testado nesses ensaios. Checagens locais

R70, testes obrigatórios incompatíveis. O perfil público fixo de `fastapi_14186` exige testes habilitados por versões mutuamente excludentes do Pydantic. Diagnóstico da tarefa

Checagens relacionadas. O registro do JUnit documenta três falsas aceitações sintéticas. Os registros operacionais da S54M documentam duas interrupções pelo mecanismo de proteção de memória.

Links do projeto

Repositório público de evidências

Evidências congeladas da referência [10], com registros das sete rodadas de avaliação no Kaggle e scripts para reproduzir as contagens relatadas.

https://github.com/aridiosilva/Gemma4_Google_Deepmind_S1/blob/433e7508f713d947b417d8522bc84b96eae1da45/evidence/kaggle-bugs-2026-11-11/README.md

Licença

Este trabalho foi publicado sob a licença Atribuição 4.0 Internacional (CC BY 4.0).

https://creativecommons.org/licenses/by/4.0/

Citação

DOI: https://doi.org/10.34740/KAGGLE/W/125650

Silva, Aridio (2026). *Budget-Bounded Software Repair with AI Agents.* Kaggle.

BibTeX (registro original do Kaggle)

```
@article{aridio_silva_2026,
  title={Budget-Bounded Software Repair with AI Agents},
  url={https://www.kaggle.com/w/125650},
  DOI={10.34740/KAGGLE/W/125650},
  journal={Kaggle},
  publisher={Kaggle},
  author={Aridio Silva},
  year={2026}
}
```

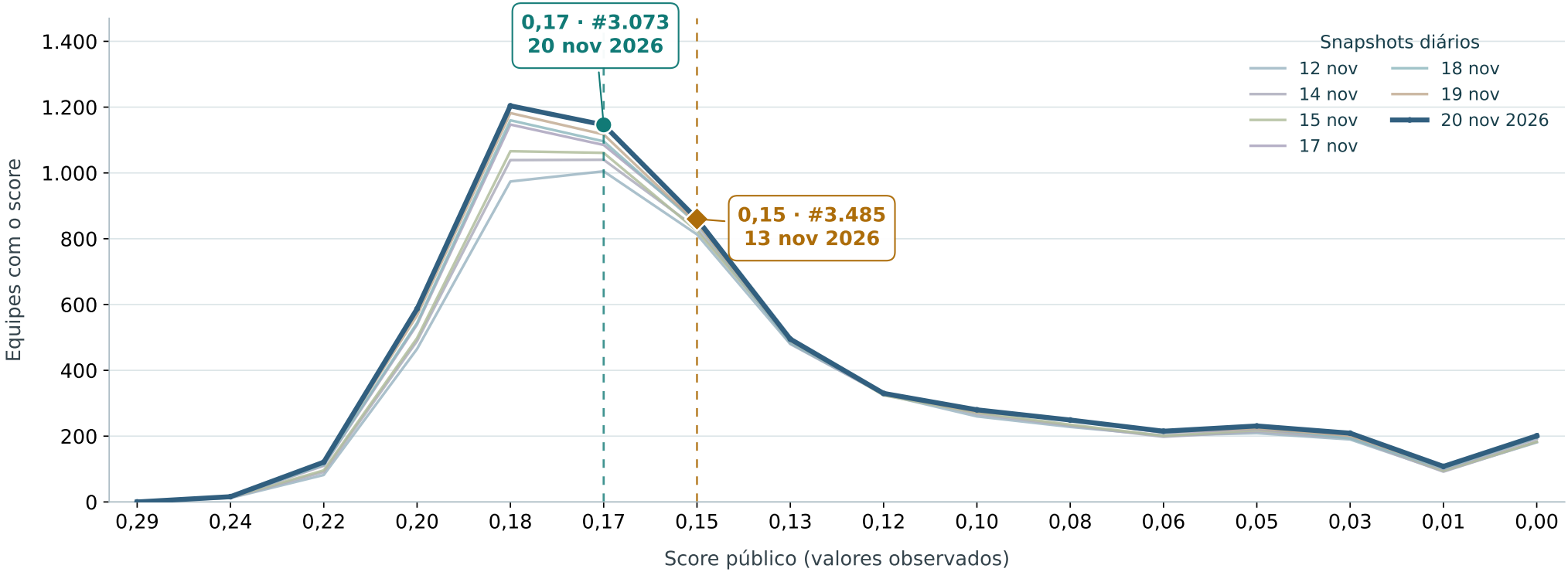
Paper publicado no Kaggle: https://www.kaggle.com/competitions/gemma-4-developer-agent-paper/writeups/budget-bounded-software-repair-with-ai-agents

Gemma 4 Developer Agent Competition 2026

Primeira participação no Kaggle: evolução no placar público

Score 0,15 → 0,17 (+13,3%) Posição 3.485 → 3.073 (412 posições)

17.814 inscritos · 7.060 participantes · 6.269 equipes — consulta em 20/11/2026, 17h09 BRT



Marco	Score	Posição	Equipes no registro
13 nov 2026	0,15	#3.485	5.523
20 nov 2026	0,17	#3.073	6.252

Distribuições: 20/11/2026, 11h49 BRT. Marcadores na curva de 20/11; ranks nas datas indicadas.
A posição é entre equipes. O snapshot contém 6.252 equipes; os totais acima foram consultados mais tarde.
Fonte: Kaggle e histórico registrado pelo autor. © 2026 Aridio Silva