

ScaleQuality Code Maturity Model (SQCM)

Um modelo de maturidade de código baseado em evidência para a era do software gerado por IA

Versão 2.0 · Agosto de 2026

Autor: Erik Fernandes · Fundador, ScaleQuality

Contato: erik.fernandes@scalequality.io · scalequality.io

Substitui: SQCM v1.0, julho de 2026 · DOI [10.5281/zenodo.21251863](https://doi.org/10.5281/zenodo.21251863)

DOI: [10.5281/zenodo.21879222](https://doi.org/10.5281/zenodo.21879222)

Sumário executivo

A versão 1.0 deste modelo definiu **maturidade de código** como um construto mensurável no nível do repositório, avaliado exclusivamente a partir de evidência observável, sem questionários e sem autodeclaração. Ela avaliava quatro domínios: Segurança, Confiabilidade, Manutenibilidade e um quarto não representado como domínio de primeira classe nos padrões revisados na v1.0, a **durabilidade de código de IA**.

A versão 2.0 mantém esse construto inalterado e o estende por um eixo e um princípio.

O eixo é um **quinto domínio: Supply Chain**. A maturidade de um repositório não pode ser lida apenas do código que o time escreveu. Uma parcela substancial do que uma aplicação entrega foi escrita por outra pessoa e trazida por um manifesto, uma imagem base ou um template de infraestrutura. Um modelo que mede apenas código de primeira mão mede apenas parte do artefato entregue.

O princípio é o que chamamos de **fronteira de medição**: *meça cada dimensão onde o artefato dela de fato existe*. Algumas propriedades são legíveis no repositório. Outras existem apenas na imagem de container construída e, portanto, só podem ser medidas na esteira que a construiu. A v2.0 torna essa fronteira explícita em vez de fingir que uma varredura de repositório enxerga tudo, e especifica onde cada dimensão é medida e o que é reportado quando o artefato está fora de alcance.

A v2.0 também reporta, pela primeira vez, **medições primárias do próprio instrumento do modelo**: a análise se completa em cerca de cinco segundos dentro de um gate de integração contínua, e este documento publica a decomposição por estágio por trás desse número, junto com duas falhas metodológicas que encontramos ao medi-lo. Um modelo que pede às organizações que confiem em medição deve a mesma disciplina à medição de si mesmo.

1. O que a v1.0 estabeleceu, e o que deixou em aberto

A versão 1.0 argumentou que as réguas existentes respondem perguntas adjacentes àquela que os times de fato fazem. ISO/IEC 25010 e ISO/IEC 5055 medem qualidade presente [1][2]. CMMI mede processo organizacional [6]. DORA mede fluxo de entrega [7][21][22]. Nenhuma responde "*posso confiar neste código?*" sobre um repositório específico, hoje, sem um engajamento de consultoria.

A resposta da SQCM v1.0 tinha quatro compromissos estruturais, todos preservados na v2.0:

1. **Evidência, nunca declaração.** Nada autorreportado entra na nota.
2. **Confiança explícita, e estados não medidos que não pontuam.** Ausência de evidência é reportada como ausência, nunca interpolada em um número neutro.
3. **Zero configuração.** Uma avaliação de repositório parte de um ponteiro e termina em minutos, porque uma avaliação que leva semanas mede um repositório que já mudou. A evidência com escopo de artefato (seção 3) é aditiva e coletada apenas onde o artefato existe; ela não reintroduz um passo de configuração na avaliação do repositório.
4. **Reprodutibilidade por versionamento de score.** Vereditos passados são imutáveis; a evolução do modelo se aplica para a frente.

Três limitações registradas na seção 8 da v1.0 são o que a v2.0 endereça:

- O modelo lia o código do próprio time e suas práticas, mas não **o que o repositório importa e entrega**.
- A análise estática era **intraprocedural**: enxergava um padrão perigoso dentro de uma função, mas não um *caminho* perigoso atravessando arquivos.
- O veredito cobria o código da aplicação, mas não a **infraestrutura e o empacotamento** que determinam como esse código é implantado.

2. O quinto domínio: Supply Chain

2.1 Por que é um domínio e não uma categoria de achado

Na maioria das aplicações modernas, uma grande parcela dos bytes entregues não foi escrita pelo time que entrega. Eles chegam por um manifesto de pacotes, por uma dependência transitiva de uma dependência transitiva, por uma imagem base ou por um template de infraestrutura. Duas propriedades fazem disso um domínio de maturidade distinto, e não uma subcategoria de Segurança:

Ele tem um modo de falha diferente. Fraquezas de segurança de primeira mão são autorais: alguém escreveu o padrão vulnerável. A exposição de supply chain é *herdada*: o código estava correto quando foi adotado e se tornou vulnerável enquanto ninguém o tocava. Um repositório pode se degradar nesse domínio com zero commits. Nenhum outro domínio tem essa propriedade, e é precisamente por isso que a avaliação pontual é insuficiente e a medição contínua importa.

Ele tem uma remediação diferente. Uma fraqueza de primeira mão se corrige mudando o código. A exposição de supply chain normalmente se corrige mudando uma restrição de versão, que é uma

decisão diferente, tomada por pessoas diferentes, com risco diferente (uma subida de versão pode quebrar o build, e é por isso que a disciplina de atualização é ela própria um sinal de maturidade).

A indústria convergiu para essa separação de forma independente: os frameworks de proveniência e integridade que surgiram após os comprometimentos de alto perfil de sistemas de build e de pacotes nos últimos anos tratam a cadeia de suprimentos como sua própria camada de garantia, com controles e verificações dedicados e, no caso do framework de proveniência, níveis progressivos de garantia [28] [29]. O movimento regulatório aponta na mesma direção, com inventário de componentes e tratamento de vulnerabilidades se tornando obrigações regulatórias em jurisdições e escopos de produto definidos, e não apenas boa prática [30][31].

2.2 O que o domínio mede

Supply Chain na SQCM v2.0 agrega três famílias de evidência observável:

- **Componentes com vulnerabilidade conhecida.** Dependências declaradas e resolvidas transitivamente, confrontadas com dados públicos de advisory [32]. O sinal é a presença de componentes com advisories publicados, ponderada pela severidade do advisory e deduplicada por identificador de advisory, de modo que um advisory afetando oito pacotes seja um problema a corrigir, não oito.
- **Conteúdo da imagem entregue.** Os pacotes de sistema operacional e de runtime presentes na imagem de container que a esteira de fato constrói, incluindo se a distribuição base ainda recebe atualizações de segurança (a seção 3 explica por que isso é medido em outro lugar).
- **Obrigações de licença.** Licenças copyleft fortes e licenças não resolvíveis entre as dependências.

A terceira família é **reportada e nunca pontuada**, e a razão é um princípio de projeto que vale enunciar: *"AGPL é aceitável aqui?"* é uma questão de política da empresa, não um defeito de código. Um scanner que responde isso em nome da empresa está tomando uma decisão jurídica que não lhe cabe. O SQCM traz a obrigação à superfície e para por aí.

2.3 O que o domínio deliberadamente não afirma

A presença de um advisory não é prova de explorabilidade. Uma função vulnerável que nunca é chamada a partir de nenhum ponto de entrada alcançável é uma obrigação real (será apontada por todo auditor e todo questionário de cliente), mas não uma exposição real. A SQCM v2.0 reporta **presença**, rotulada como presença, e não afirma alcançabilidade. A seção 7 registra a análise de alcançabilidade como trabalho futuro declarado, e não como capacidade implícita.

3. A fronteira de medição

3.1 O princípio

Algumas propriedades de um sistema são legíveis em seu repositório. Outras não são legíveis em lugar algum exceto em um artefato construído que o repositório apenas descreve.

A versão 1.0 assumia, implicitamente, que o repositório era a unidade de medição. A versão 2.0 substitui essa suposição por uma regra explícita:

Meça cada dimensão onde o artefato dela existe. Onde o artefato está fora de alcance, reporte a dimensão como não medida. Nunca substitua por um proxy e apresente como se fosse a coisa.

Isso soa óbvio e é rotineiramente violado. O caso concreto que nos forçou a enunciá-lo são as vulnerabilidades de imagem de container.

3.2 O caso da imagem de container

As vulnerabilidades de uma imagem de container vivem na imagem: seus pacotes de sistema operacional, seu runtime, as camadas que um build produziu. Um repositório contém uma *receita* para essa imagem, não a imagem.

Três abordagens estão disponíveis, e duas delas são desonestas:

1. **Varrer a imagem base referenciada no Dockerfile.** Barato de implementar e errado: a base não é o que o cliente implanta. Tudo o que o build acrescenta, e cada camada que um registry privado contribui, fica invisível. Também custa um pull completo de imagem por análise, o que destrói a propriedade de latência da avaliação (seção 5).
2. **Inferir pela tag.** Reportar a imagem base como desatualizada ou fora de suporte com base em seu nome. Isso é uma heurística sobre uma string, apresentada como medição de um artefato.
3. **Medir a imagem onde ela existe.** A esteira que acabou de construir a imagem é o único lugar onde a imagem está presente, local e livre para inspeção. Varra ali e transmita apenas o resultado.

A SQCM v2.0 toma o terceiro caminho. A consequência é uma propriedade que a maioria dos instrumentos de medição não tem: **uma dimensão pode estar medida ou não dependendo de onde a medição é invocada**, e o veredito diz qual. Uma avaliação de repositório reporta o conteúdo da imagem como não medido. Uma avaliação de esteira, com a referência da imagem fornecida, reporta como medido, com a imagem identificada no próprio achado.

Duas disciplinas tornam isso seguro em vez de confuso:

- **O estado não medido nunca é um estado limpo.** Uma avaliação que não recebeu um relatório de imagem diz isso. Ela não reporta zero vulnerabilidades de imagem.
- **Sonegar evidência não pode melhorar o veredito.** Fornecer o relatório da imagem só pode baixar uma nota ou deixá-la inalterada, nunca elevá-la, de modo que uma organização não consegue manipular o veredito omitindo a imagem. A propriedade que garantimos é essa, direcional, enunciada deliberadamente em vez da afirmação mais forte e mais difícil de que toda medição adicional possível seja monotônica.

3.3 A mesma fronteira, em outros lugares

Uma vez enunciado, o princípio se aplica além de containers. Templates de infraestrutura como código *estão* no repositório, então a má configuração é medida ali. Comportamento em tempo de execução não está no repositório e não é medido de forma alguma, que é a razão pela qual a v1.0 excluiu Eficiência de Performance [2] e a v2.0 continua excluindo. A fronteira é o que distingue um escopo deliberado de um ponto cego accidental.

4. Famílias de sinal acrescentadas na v2.0

Além do quinto domínio, a v2.0 acrescenta três famílias de sinal a domínios existentes. Cada uma é enunciada com o que mede e com o que deliberadamente se recusa a medir.

Má configuração de infraestrutura e container (para Segurança). Definições de container, arquivos compose, templates de infraestrutura, manifestos de orquestração e templates de cloud formation são avaliados contra um catálogo publicado de más configurações: containers rodando como root, armazenamento sem criptografia, regras de rede mais amplas do que precisam ser, transporte sem TLS.

Duas decisões de calibração aqui são, acreditamos, a diferença entre sinal e ruído. Primeiro, verificações de severidade informativa são descartadas: em um repositório real de infraestrutura elas eram a maioria de todos os achados e nenhuma delas era uma decisão que alguém tomaria. Segundo, um pequeno número de verificações que descrevem **intenção arquitetural em vez de defeito** é deliberadamente rebaixado. Um produto público *deve* ter um balanceador de carga voltado para a internet. Um veredito cujo risco mais bem classificado é "sua aplicação pública tem um balanceador público" é um veredito que treina seu leitor a parar de ler, e um modelo que ensina as pessoas a ignorá-lo falhou independentemente de sua cobertura.

Relacionado, e enunciado como princípio: má configuração de infraestrutura **não dispara o teto de exposição crítica** que uma credencial vazada ou um advisory crítico entregue dispara. Uma má configuração é um juízo de postura contra uma linha de base; uma credencial versionada é prova de exposição. Tratá-las como equivalentes tornaria o sinal mais forte do modelo indistinguível do mais fraco.

Duplicação de código (para Manutenibilidade). Duplicação é o defeito que nenhum linter reporta: o código está correto, apenas foi escrito quatro vezes, então toda correção precisa ser encontrada quatro vezes. É também, segundo a evidência de indústria reunida na seção 3.3 da v1.0, uma assinatura documentada de desenvolvimento assistido por IA [17][18][19], o que faz dela uma leitura direta da tese de durabilidade e não uma métrica emprestada.

O limiar de clone do modelo está alinhado aos limiares que detectores de clone estabelecidos usam (na ordem de dez linhas sucessivas e cem tokens; os limiares exatos variam por ferramenta e linguagem), de modo que o percentual reportado seja comparável a um número que praticantes já conhecem, em vez de idêntico ao de uma ferramenta específica. Duas decisões de calibração, de novo, importam mais que a detecção:

- **Abaixo de um limiar, duplicação não move a nota.** Alguma repetição é o custo honesto da clareza, e um instrumento que pune toda ela empurra times para abstração prematura, que é um desfecho pior do que a duplicação era.
- **Apenas código-fonte é medido.** Isso não é óbvio e nos custou um resultado falso antes de percebermos. Rodado sem restrição, o detector reportou **8,5% de duplicação** em um de nossos próprios repositórios. Restrito a formatos de código-fonte, o mesmo repositório mediu **1,7%**. A diferença eram catálogos de tradução, bundles gerados e lockfiles, arquivos que *devem* ser repetitivos. O primeiro número não era medição de coisa alguma.

Vulnerabilidades de imagem de container (para Supply Chain), conforme a seção 3.

5. Velocidade como propriedade do modelo

5.1 Por que latência pertence a um documento de metodologia

A versão 1.0 argumentou que zero configuração é um requisito epistemológico, não uma conveniência: uma avaliação que exige semanas mede um repositório que já mudou. Latência é o mesmo argumento em uma escala de tempo mais curta.

Se a maturidade deve funcionar como um contrato operacional, uma linha na esteira de entrega abaixo da qual o build falha, então a avaliação precisa caber dentro da esteira. Um instrumento que acrescenta minutos a cada pull request será desligado, e um instrumento desligado não mede nada. A velocidade da medição é, portanto, uma propriedade do modelo, não um detalhe de sua implementação.

5.2 Medido

Em um repositório de porte médio, a avaliação completa dentro de um gate de integração contínua mediu **5,25 s**, **5,29 s** e **7,17 s** em três corridas consecutivas, ponta a ponta do upload ao veredito. Condições de medição, enunciadas para que o número possa ser atacado: um container de dois vCPU e oito GB no cluster de staging, o motor rápido com todos os estágios presentes, um intervalo de polling adaptativo (ver 5.3) e três corridas consecutivas sobre o mesmo commit. Três corridas é uma amostra pequena e é reportada como tal; uma campanha maior, com mediana e latência de cauda ao longo de uma faixa de tamanhos de repositório, é trabalho declarado na seção 7, não afirmado aqui. Tempos por estágio, todos executando concorrentemente:

ESTÁGIO	MEDIDO
Análise estática (segurança, confiabilidade, manutenibilidade)	5,12 a 5,21 s
Leitura de cobertura de testes	2,03 a 2,50 s
Advisories de dependências	1,08 a 1,33 s
Duplicação	1,05 a 1,16 s
Detecção de segredos	0,12 a 0,15 s
Verificação de presença de infraestrutura	25 a 54 ms (<i>scanner pulado, sem infraestrutura presente; o scanner em si custa ~1,2 s quente, ver 5.3</i>)

Como os estágios rodam concorrentemente, a avaliação custa aproximadamente o estágio mais longo, e não a soma deles.

O número que importa para a afirmação do modelo não é a cifra absoluta, mas sua relação com o ciclo de mudança: **a avaliação cabe dentro de um pull request sem ser notada.**

5.3 Duas falhas metodológicas, publicadas deliberadamente

Um documento que publica a velocidade de um instrumento deveria publicar como a medição dessa velocidade deu errado, porque ambas as falhas são gerais e ambas teriam produzido um número em que teríamos acreditado.

Falha um: um custo que atribuímos à coisa errada, duas vezes. Acrescentar o scanner de má configuração de infraestrutura moveu a avaliação de 4,4 s para 7,1 s no mesmo repositório. A leitura óbvia era que varredura de infraestrutura é cara.

A segunda leitura era melhor mas ainda errada. O custo do scanner era *independente do tamanho do repositório* (3,47 s em um repositório de algumas dezenas de arquivos, 3,31 s em um muito maior), então concluímos que ele pagava um grande custo fixo compilando seu bundle de políticas antes de ler um único arquivo. Essa conclusão produziu uma correção certa por uma razão parcialmente incorreta: o scanner agora é invocado apenas quando existem arquivos de infraestrutura, estabelecido por uma verificação de sistema de arquivos que custa cerca de um milissegundo, e repositórios sem infraestrutura não pagam nada.

Medido adequadamente em isolamento, o scanner leva cerca de **1,2 s** quente, e cinco invocações concorrentes completam em **cerca de 2 s no total** em uma máquina com cores de sobra. A independência de tamanho era real; a magnitude não era. A maior parte dos 3,4 s era **contenção** com os outros dois estágios limitados por CPU em uma máquina de dois cores, e não trabalho que o scanner estivesse fazendo.

A lição geral é aquela de que esta seção trata: em uma máquina saturada, a duração medida de cada estágio inclui os outros estágios, e uma tabela de tempos por estágio se lê como uma atribuição quando é em parte uma divisão de escassez. A correção sobreviveu à retificação (pular um scanner que não tem o que varrer é certo de qualquer forma), mas o número que teríamos publicado como "o scanner custa 3,4 s" era uma propriedade da máquina, não do scanner.

Falha dois: um artefato do instrumento de medição reportado como propriedade do sistema. O cliente da esteira consultava o veredito em um intervalo fixo de cinco segundos. A avaliação se completa em aproximadamente cinco segundos. A consequência é que o *mesmo repositório no mesmo commit* mediu **5,26 s ou 10,28 s** dependendo de qual lado de uma fronteira de polling a análise por acaso caísse. Nenhum dos números estava errado e nenhum era uma medição do sistema: a diferença de cinco segundos era o instrumento esperando, não o sistema trabalhando. Com um intervalo adaptativo, três corridas consecutivas retornaram 5,25 s, 5,29 s e 7,17 s.

A lição geral não é sobre polling. É que **um número medido herda a resolução de qualquer coisa que o mediu**, e um instrumento que reporta em granularidade mais grossa que o fenômeno produzirá resultados estáveis, reproduzíveis e inteiramente artefatuais. Publicamos isso porque quase citamos a cifra de 10,28 s como nosso próprio benchmark.

6. Análise interprocedural

6.1 A limitação que isso endereça

A análise estática baseada em padrões casa formas dentro de um único arquivo. Ela enxerga `eval(userInput)` quando ambos aparecem juntos, e não enxerga o caso que de fato ocorre em código de produção, no qual o valor não confiável entra por uma função, é passado por duas outras e alcança a operação perigosa em um quarto arquivo. Esse caminho é onde vivem os defeitos exploráveis reais, e ele é invisível para um casador de padrões por construção.

Fechá-lo exige **análise de taint**: construir um grafo do programa e seguir valores de origens não confiáveis até sinks sensíveis, atravessando fronteiras de função e de arquivo.

6.2 Por que construímos o nosso

A análise interprocedural está disponível comercialmente, e os termos de licenciamento das principais opções foram o fator decisivo, e não a tecnologia. Uma categoria é proprietária e exigiria um acordo de OEM para ser revendida como parte de um produto. Outra categoria está disponível sem custo, mas sob termos que restringem o uso fora de contextos de código aberto e pesquisa, incluindo análise automatizada de código de terceiros como serviço hospedado [34], que é precisamente o que um produto faz. Uma capacidade sobre a qual não podemos legalmente construir um produto não é uma capacidade.

A análise profunda da SQCM v2.0 é, portanto, construída sobre uma plataforma de grafo de propriedades de código de código aberto com licença permissiva (Apache 2.0) [33], com uma especificação de taint de nossa autoria. A especificação é propriedade do modelo e não é publicada aqui, nos mesmos termos que a v1.0 estabeleceu para os pesos de pontuação.

6.3 O que foi medido durante a calibração

O resultado que vale publicar não é o conjunto de regras, mas o **comportamento de calibração**, porque ele quantifica um compromisso que todo produto de análise estática enfrenta e poucos reportam.

Contra uma aplicação de referência deliberadamente vulnerável, refinamentos sucessivos da especificação produziram **137, depois 25, depois 4, depois 0 falsos positivos**, enquanto uma suíte fixa de regressão de caminhos verdadeiros conhecidos permaneceu em **5 de 5 detectados** o tempo todo.

O que o arco estabelece é uma posição de projeto: a especificação foi ajustada em direção à precisão mesmo ao custo de revocação, sob o raciocínio de que **um falso positivo custa mais que um achado perdido** em um instrumento cujo valor inteiro é ser acreditado. Uma ferramenta que grita lobo é desligada, e uma ferramenta desligada tem revocação zero.

6.4 Estender a especificação, e o que a extensão custou

A especificação inicial cobria cinco classes de vulnerabilidade em uma linguagem. Ela agora cobre **dez classes em Java e sete em JavaScript/TypeScript**, cada uma carregando seu identificador CWE no próprio achado, e não em uma tabela de cobertura. Os dois catálogos são deliberadamente assimétricos: desserialização insegura e reflexão por nome são problemas de Java que mal existem em Node, e forçar uma regra em uma linguagem onde sua forma não ocorre produz ruído, não cobertura.

A extensão é reportada aqui por causa do que ela custou. Rodar as novas regras contra código real produziu **dez achados errôneos: nove falsos positivos e uma classificação incorreta**, e todos os dez compartilhavam uma única causa raiz: **um sink identificado por nome de método sem restrição sobre seu receptor**. `getInputStream` também existe em um processo, em um arquivo compactado e em um recurso de classpath; `setHeader` também é um método de construtor de JSON Web Token; `execute` também é um método de domínio comum em um objeto comum. Cada um foi corrigido exigindo que o receptor fosse o que a regra significa, e cada correção foi verificada contra o corpus em vez de presumida.

A única classificação incorreta entre esses dez merece enunciado separado, porque é um tipo de erro diferente dos outros nove. Uma regra reportou um caminho contaminado genuíno sob a classe errada: dados lidos da resposta de uma chamada HTTP de saída alcançaram uma escrita, e a regra chamou aquilo de acesso a arquivo. O fluxo era real; a classificação não. **Atribuir a classe errada não é um erro menor que um falso positivo**: é o erro que faz um time de segurança parar de ler o relatório, porque demonstra que o instrumento não entende o que está olhando.

A validação agora roda contra um corpus rotulado em vez de uma única aplicação:

SUJEITO	RESULTADO
Aplicação Java deliberadamente vulnerável, 277 arquivos-fonte	23 achados, cada um no pacote nomeado por sua própria classe de vulnerabilidade , 23 de 23
Aplicação Node deliberadamente vulnerável	5 achados, todos genuínos, incluindo um que os próprios comentários da aplicação descrevem como inseguro
Aplicação Java de referência em saúde comum, 49 arquivos	0 achados , que é o resultado correto
Código de produção da própria ScaleQuality: seis serviços e um front end web	0 falsos positivos

O corpus é mais forte que o arco de aplicação única que ele substitui, porque nos dois sujeitos vulneráveis a estrutura de diretórios codifica a classe esperada, o que dá uma verificação independente sobre a classificação e não apenas sobre a detecção. Ele permanece aquém da afirmação geral de precisão e revocação declarada como trabalho futuro na seção 7: estas são aplicações públicas de ensino, cujos defeitos foram escritos para serem encontrados.

6.5 Duas falhas do ambiente, não das regras

A seção 5.3 publicou duas maneiras pelas quais uma medição de velocidade pode ser um artefato de seu instrumento. Estender o motor profundo produziu duas da mesma espécie, e ambas são propriedades de *onde a análise roda*, e não do que ela analisa. Ambas são reportadas porque ambas são invisíveis por construção.

O motor pode deixar de rodar por completo, em silêncio. A plataforma de grafo lê seu próprio arquivo de consulta usando a codificação de caracteres padrão do host. Em um container sem locale configurado, esse padrão é ASCII, e um único byte acentuado em um comentário aborta a consulta. O estágio de parse tem sucesso; apenas a consulta morre; e uma consulta que falha é corretamente registrada como *não medida*. O resultado é uma análise que nunca aconteceu, reportada como ausência de análise em vez de erro, que é o desfecho honesto e também a razão pela qual ninguém percebeu. A condição estava latente desde que o primeiro comentário não-ASCII foi escrito.

Uma regra pode desaparecer em um ambiente e não em outro. A resolução de tipos para chamadas encadeadas depende do runtime que executa o parser. A mesma construção, a mesma versão do motor e o mesmo código-fonte resolveram para um tipo plenamente qualificado em uma máquina e permaneceram não resolvidos em outra. Como as restrições de precisão descritas em 6.4 são expressas em termos de tipos resolvidos, a regra afetada não encontrou nada onde o tipo não resolveu: dez achados em uma máquina, nove na outra, sem erro em nenhuma. As restrições de receptor agora são satisfeitas pelo tipo resolvido **ou** pelo texto da chamada, de modo que uma regra se degrada em precisão em vez de desaparecer.

A lição geral é a mesma que rege a seção 5.3, em uma camada diferente: **os resultados de um instrumento são propriedade do instrumento e de seu ambiente em conjunto**, e uma corrida de validação na máquina do autor mede um sistema que ninguém mais vai rodar. A validação agora é realizada dentro do artefato publicado, e as duas falhas acima são exatamente o que essa disciplina capturou.

6.6 Por que ela não roda em toda avaliação

A construção do grafo exige seu próprio runtime e gigabytes de memória. Medido dentro do artefato publicado contra o sujeito Java de 277 arquivos, o ciclo profundo completo custa **cerca de 33 segundos** em oito CPUs virtuais, contra **67 segundos** em quatro e **101 segundos** em duas; a escala quase linear é uma propriedade de um parser que paraleliza por arquivo. Rodar isso dentro do gate de cinco segundos não é possível, e fingir o contrário sacrificaria a propriedade da seção 5 para ganhar a da seção 6.

A escala tem uma consequência de segunda ordem que vale registrar, porque inverte o raciocínio usual sobre computação: como a plataforma cobra por segundo, a configuração de oito CPUs custa cerca de três vezes mais por segundo e termina cerca de três vezes antes, então o **custo por avaliação fica**

aproximadamente inalterado enquanto a espera cai para um terço. Para uma análise fora de banda que alguém está esperando, essa é a diferença entre um resultado e uma interrupção. As duas são, portanto, **dois produtos do modelo, não duas configurações**: a avaliação rápida roda a cada mudança, e a profunda roda fora de banda, em seu próprio ritmo, com o resultado anexado ao repositório quando ficar pronto. Qualquer avaliação que solicite análise profunda onde o motor profundo esteja indisponível é servida pelo motor rápido **e registra que foi**, de modo que a proveniência nunca afirme uma profundidade que não entregou.

7. Limitações e agenda de validação

A versão 1.0 declarou seis limitações. Quatro permanecem inalteradas e não são repetidas: evidência observável tem pontos cegos; performance está fora de escopo; a atribuição de autoria de IA é heurística; correlação não é causalidade.

Dois itens da v1.0 são atualizados, e quatro são novos na v2.0.

Atualizado: os limiares de nível continuam sendo calibração de projeto. As faixas ainda derivam da tradição de cinco níveis [6] e da experiência de projeto, não de um estudo longitudinal de desfechos. A agenda é a mesma: correlacionar níveis contra desfechos observáveis em uma coorte medida continuamente, publicar distribuições de nível para calibração pública e revisar com uma versão de score incrementada que preserve o histórico.

Atualizado: a evidência de terceiros para o domínio de durabilidade mantém seus vieses conhecidos [17][18][19][26][27], com as mesmas ressalvas que a v1.0 registrou.

Corrigido durante o próprio desenvolvimento da v2.0: saturação de Supply Chain. O domínio originalmente compunha subtraindo uma dedução fixa por advisory distinto. Como as contagens de advisory em repositórios reais são grandes, ele atingia seu piso quase imediatamente: três advisories críticos, ou quatro altos, ou dez moderados, e a nota era zero. Todo sujeito real que medimos pontuou zero, o que é uma afirmação verdadeira que deixou de discriminar: ela não conseguia distinguir um repositório com um backlog gerenciável de um com backlog ingerenciável, e essa distinção é a única coisa de que o leitor precisa.

A composição agora é decaimento exponencial sobre unidades de advisory ponderadas e deduplicadas. Um advisory crítico ainda cai na faixa crítica, a severidade permanece monotônica e mais advisories nunca é melhor; o que mudou é que a faixa onde um time pode de fato agir agora se espalha pela escala em vez de colapsar sobre seu piso. A mudança foi entregue como incremento de versão de score sob a disciplina de reprodutibilidade da seção 6 da v1.0: vereditos anteriores mantêm a versão que os produziu e nunca são silenciosamente recomparados contra a nova curva.

Registramos isso como limitação e não como funcionalidade porque é uma: a calibração original foi publicada no espírito da v1.0 e estava errada de um jeito que só a medição revelou, e o mesmo risco se aplica a todo limiar deste modelo que ainda não encontrou uma população grande o suficiente. Essa é a agenda do item acima.

Novo: presença é reportada, alcançabilidade não é. Seção 2.3. A análise de alcançabilidade é um item declarado de roteiro e deliberadamente não é implicada por nenhuma saída atual.

Novo: a especificação de análise profunda é validada contra um corpus rotulado, não contra um corpus geral. Seções 6.3 e 6.4. Detecção e classificação agora são verificadas contra aplicações públicas cuja estrutura de diretórios codifica a classe de vulnerabilidade esperada, e contra código de produção para falsos positivos, em duas linguagens. O que isso estabelece é o comportamento sobre defeitos escritos para serem encontrados. Precisão e revocação gerais permanecem não estabelecidas, e a agenda é um corpus independente não escrito como material didático, além de cobertura de linguagem para além das duas.

Novo: o benchmark de velocidade é uma amostra pequena, não uma campanha. Seção 5.2. As cifras reportadas são três corridas em um repositório sob condições declaradas. Uma afirmação defensável de latência precisa de mediana e cauda ao longo de uma faixa de tamanhos e linguagens, sobre versões fixadas de instrumento e hardware; essa campanha é agenda, e os números atuais são oferecidos como propriedade de ordem de magnitude, não como resultado de benchmark.

Novo: a fronteira de medição muda o que é comparável. Como algumas dimensões são medidas apenas onde seu artefato existe (seção 3), duas avaliações do mesmo repositório podem legitimamente cobrir escopos diferentes. O modelo endereça isso com a garantia direcional da seção 3.2 (sonegar evidência não pode melhorar o veredito) e com proveniência por scanner registrada em todo veredito, de modo que quaisquer dois vereditos possam ser comparados sabendo exatamente o que cada um enxergou. Usuários que comparem notas entre esteiras instrumentadas de formas diferentes devem ler a proveniência, e o produto a expõe por essa razão.

8. Conclusão

A versão 1.0 argumentou que a era da IA precisa de uma régua para uma pergunta que as régua existentes não respondem, e definiu maturidade de código como essa régua: medida a partir de evidência observável, sem declaração, em minutos, incluindo uma dimensão de durabilidade não representada como domínio de primeira classe nos padrões revisados na v1.0.

A versão 2.0 faz duas correções nesse instrumento, e ambas vieram de medir em vez de projetar.

A primeira é que um repositório não é o artefato inteiro. Uma parte substancial do que uma aplicação entrega pode chegar de outro lugar, então um modelo de maturidade que lê apenas código de primeira mão avalia apenas parte do artefato entregue. Supply Chain entra como domínio de primeira classe por essa razão.

A segunda é que o repositório não é o único lugar onde a medição acontece. Algumas propriedades existem apenas em um artefato construído, e a resposta honesta é medi-las onde elas existem e declará-las não medidas quando estão fora de alcance, nunca substituir por um proxy e apresentá-lo como a coisa. Esse princípio, mais que qualquer scanner individual, é o que a versão 2.0 acrescenta ao modelo.

Ambas as correções apontam na mesma direção do compromisso fundador da versão 1.0. Um instrumento conquista o direito de ser acreditado sendo explícito sobre o que viu, o que não viu e quão confiante está na diferença. A versão 2.0 estende essa disciplina às medições do próprio instrumento sobre si mesmo, incluindo quatro ocasiões em que um número era um artefato de como ou onde estávamos medindo, e não uma propriedade do que medimos: duas ao cronometrar a avaliação rápida,

e duas no ambiente em que a avaliação profunda roda, onde o modo de falha não era um número errado, mas uma ausência silenciosa de número.

O modelo está em operação em scalequality.io. A metodologia permanece publicada, versionada e aberta ao escrutínio, porque é isso que separa uma régua de uma opinião.

Referências

As referências [1] a [27] vêm da SQCM v1.0. Elas são reproduzidas aqui na íntegra para que este documento resolva suas próprias citações sem exigir o registro da v1.0.

[1] ISO/IEC 25010:2023. *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model*. International Organization for Standardization, 2023. <https://www.iso.org/standard/78176.html>

[2] ISO/IEC 5055:2021. *Information technology — Software measurement — Software quality measurement — Automated source code quality measures*. International Organization for Standardization, 2021 (desenvolvida pelo CISQ/OMG). <https://www.iso.org/standard/80623.html> · <https://www.it-cisq.org/standards/code-quality-standards/>

[3] Letouzey, J.-L. "The SQALE method for evaluating Technical Debt". *Third International Workshop on Managing Technical Debt (MTD 2012)*, IEEE, 2012, pp. 31-36. DOI: 10.1109/MTD.2012.6225997. Ver também: Letouzey, J.-L.; Ilkiewicz, M. "Managing Technical Debt with the SQALE Method". *IEEE Software* 29(6), 2012, pp. 44-51. <https://ieeexplore.ieee.org/document/6225997>

[4] Oman, P.; Hagemester, J. "Construction and testing of polynomials predicting software maintainability". *Journal of Systems and Software* 24(3), 1994, pp. 251-266. DOI: 10.1016/0164-1212(94)90067-1. [https://doi.org/10.1016/0164-1212\(94\)90067-1](https://doi.org/10.1016/0164-1212(94)90067-1)

[5] Heitlager, I.; Kuipers, T.; Visser, J. "A Practical Model for Measuring Maintainability". *QUATIC 2007*, IEEE, pp. 30-39. DOI: 10.1109/QUATIC.2007.7. Operacionalização: *SIG/TÜViT Evaluation Criteria Trusted Product Maintainability*. <https://dl.acm.org/doi/10.1109/QUATIC.2007.7> · <https://www.softwareimprovementgroup.com/wp-content/uploads/SIG-TUViT-Evaluation-Criteria-Trusted-Product-Maintainability.pdf>

[6] CMMI Institute / ISACA. *Capability Maturity Model Integration (CMMI) V3.0*, 2023. <https://cmmiinstitute.com/> · <https://cmmiinstitute.com/learning/appraisals/levels>

[7] Forsgren, N.; Humble, J.; Kim, G. *Accelerate: The Science of Lean Software and DevOps*. IT Revolution Press, 2018. ISBN 978-1942788331. Programa de pesquisa: <https://dora.dev/>

[8] Inozemtseva, L.; Holmes, R. "Coverage Is Not Strongly Correlated with Test Suite Effectiveness". *ICSE 2014*, ACM. DOI: 10.1145/2568225.2568271. <https://dl.acm.org/doi/10.1145/2568225.2568271>

[9] Just, R.; Jalali, D.; Inozemtseva, L.; Ernst, M. D.; Holmes, R.; Fraser, G. "Are Mutants a Valid Substitute for Real Faults in Software Testing?". *FSE 2014*, ACM. DOI: 10.1145/2635868.2635929. <https://dl.acm.org/doi/10.1145/2635868.2635929>

- [10] Rahman, F.; Devanbu, P. "How, and Why, Process Metrics Are Better". *ICSE 2013*, IEEE. <https://ieeexplore.ieee.org/document/6606589/>
- [11] Nagappan, N.; Ball, T. "Use of Relative Code Churn Measures to Predict System Defect Density". *ICSE 2005*, ACM, pp. 284-292. DOI: 10.1145/1062455.1062514. <https://dl.acm.org/doi/10.1145/1062455.1062514>
- [12] Bird, C.; Nagappan, N.; Murphy, B.; Gall, H.; Devanbu, P. "Don't Touch My Code! Examining the Effects of Ownership on Software Quality". *ESEC/FSE 2011*, ACM, pp. 4-14. DOI: 10.1145/2025113.2025119. <https://dl.acm.org/doi/10.1145/2025113.2025119>
- [13] Avelino, G.; Passos, L.; Hora, A.; Valente, M. T. "A Novel Approach for Estimating Truck Factors". *ICPC 2016*, IEEE. <https://arxiv.org/abs/1604.06766>
- [14] Krasner, H. / CISQ. *The Cost of Poor Software Quality in the US: A 2022 Report*. Consortium for Information & Software Quality, dezembro de 2022. <https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2022-report/>
- [15] Pearce, H.; Ahmad, B.; Tan, B.; Dolan-Gavitt, B.; Karri, R. "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions". *IEEE Symposium on Security and Privacy (S&P) 2022*. <https://arxiv.org/abs/2108.09293>
- [16] Perry, N.; Srivastava, M.; Kumar, D.; Boneh, D. "Do Users Write More Insecure Code with AI Assistants?". *ACM CCS 2023*. DOI: 10.1145/3576915.3623157. <https://dl.acm.org/doi/abs/10.1145/3576915.3623157>
- [17] GitClear. *Coding on Copilot: 2023 Data Suggests Downward Pressure on Code Quality*. 2024. https://www.gitclear.com/coding_on_copilot_data_shows_ais_downward_pressure_on_code_quality
- [18] GitClear. *AI Copilot Code Quality: 2025 Data Suggests 4x Growth in Code Clones*. 2025. https://www.gitclear.com/ai_assistant_code_quality_2025_research
- [19] GitClear. *The Maintainability Gap: AI Code Quality in 2026*. Janeiro de 2026. https://www.gitclear.com/the_ai_code_quality_maintainability_gap
- [20] Peng, S.; Kalliamvakou, E.; Cihon, P.; Demirer, M. "The Impact of AI on Developer Productivity: Evidence from GitHub Copilot". arXiv:2302.06590, 2023 (preprint). <https://arxiv.org/abs/2302.06590>
- [21] DORA / Google Cloud. *Accelerate State of DevOps Report 2024*. <https://dora.dev/research/2024/dora-report/>
- [22] DORA / Google Cloud. *State of AI-assisted Software Development 2025*. Setembro de 2025. <https://dora.dev/dora-report-2025/>
- [23] Fortune. "Google's code is now more than 25% AI-generated, CEO Sundar Pichai says". Outubro de 2024. <https://fortune.com/2024/10/30/googles-code-ai-sundar-pichai/>
- [24] CNBC. "Satya Nadella says as much as 30% of Microsoft code is written by AI". Abril de 2025. <https://www.cnbc.com/2025/04/29/satya-nadella-says-as-much-as-30percent-of-microsoft-code-is-written-by-ai.html>

- [25] Semafor. "Google CEO says 75% of company's new code is AI-generated". Abril de 2026. <https://www.semafor.com/article/04/24/2026/google-ceo-says-75-of-companys-new-code-is-ai-generated>
- [26] Liu, Y.; Widyasari, R.; Zhao, Y.; Irsan, I. C.; Chen, J.; Lo, D. "Debt Behind the AI Boom: A Large-Scale Empirical Study of AI-Generated Code in the Wild". arXiv:2603.28592, março de 2026 (preprint). <https://arxiv.org/abs/2603.28592>
- [27] "Quality Assurance of LLM-generated Code: Addressing Non-Functional Quality Characteristics". arXiv:2511.10271, novembro de 2025 (preprint). <https://arxiv.org/abs/2511.10271>
- Novas na v2.0:*
- [28] OpenSSF. *SLSA — Supply-chain Levels for Software Artifacts*. Open Source Security Foundation. <https://slsa.dev/>
- [29] Open Source Security Foundation. *OpenSSF Scorecard*. <https://securityscorecards.dev/>
- [30] The White House. *Executive Order 14028: Improving the Nation's Cybersecurity*. Maio de 2021 (estabeleceu expectativas de bill of materials de software para o fornecimento de software federal). <https://www.federalregister.gov/documents/2021/05/17/2021-10460/improving-the-nations-cybersecurity>
- [31] União Europeia. *Regulamento (UE) 2024/2847, de 23 de outubro de 2024 — Cyber Resilience Act*, sobre requisitos horizontais de cibersegurança para produtos com elementos digitais. <https://eur-lex.europa.eu/eli/reg/2024/2847/oj>
- [32] Google. *OSV — Open Source Vulnerabilities*. Base de dados distribuída de vulnerabilidades e esquema. <https://osv.dev/> · <https://ossf.github.io/osv-schema/>
- [33] Joern — Plataforma de análise de código open-source baseada em Code Property Graphs. Licença Apache 2.0. <https://joern.io/> · <https://github.com/joernio/joern>
- [34] GitHub. *CodeQL — Terms and Conditions*. Uso restrito a contextos de código aberto e acadêmicos/pesquisa; análise automatizada de código de terceiros e uso hospedado/comercial excluídos sem acordo separado. <https://github.com/github/codeql-cli-binaries> · <https://docs.github.com/en/code-security/codeql-cli>