Bug in Prod

Comunidade de qualidade e resiliência • Whitepaper técnico • 2026

Da automação à vantagem competitiva

Um modelo de maturidade para implementar IA em operações de software de missão crítica

Resumo. Este documento sintetiza a primeira Talk do Bug in Prod Circle, um círculo fechado de profissionais de alta senioridade e lideranças que operam software de missão crítica e debatem a fundo o que define qualidade e resiliência na nova era de software. Como aplicar IA de forma efetiva no contexto de qualidade e resiliência, sem queimar orçamento, é um dos temas que os próprios membros mais elegem como ponto de debate, e é o ponto de partida deste material. A adoção de inteligência artificial em engenharia de software já é praticamente universal, e ainda assim a insatisfação com o retorno desse investimento cresce na mesma velocidade. O problema raramente está nos modelos: está na forma como as organizações medem o sucesso e na sequência em que implementam a tecnologia. Este documento propõe uma definição operacional de implementar IA com efetividade (gerar vantagem competitiva, e não acumular automações isoladas ou métricas de atividade), um modelo de maturidade em oito níveis para o uso de IA em engenharia, e uma arquitetura de adoção em três camadas (grafo de contexto, harness e capabilities) aplicada ao domínio de qualidade e resiliência de sistemas de missão crítica.

1. O paradoxo do investimento em IA

Vivemos no pico do ciclo de expectativa em torno da IA. A adoção entre desenvolvedores chegou a um patamar que o relatório DORA 2025 descreve como praticamente universal, próximo de 90%. Em paralelo, multiplicam-se os relatos de companhias insatisfeitas com o retorno do que investiram.

A causa desse paradoxo está, em boa parte, nas perguntas que a liderança técnica usa para avaliar o próprio progresso. Três delas aparecem com frequência:

- **"Como aumentamos a adoção de IA nos times?"** Trata o uso quantitativo da ferramenta como se fosse, por si só, retorno para o negócio.
- **"Qual é a melhor IDE ou ferramenta de IA?"** Supõe que o resultado depende da ferramenta, quando depende sobretudo do contexto em que ela opera.
- **"Quantos pull requests cada desenvolvedor gera com IA?"** Algumas organizações chegam a medir o investimento pela quantidade de requisições feitas com assistência de IA.

Essas métricas medem atividade, não valor. Elas crescem mesmo quando o impacto no negócio é nulo ou negativo. O critério que importa é outro, e é o tema da próxima seção.

A consequência prática de medir a coisa errada aparece em produção. É comum equipes relatarem uma sensação de produtividade no curto prazo (mais features, mais scripts gerados) que não se sustenta: o código para de funcionar em produção, a causa não é óbvia, e o ganho aparente não se reproduz entre pessoas diferentes do time. Em sistemas de missão crítica, onde mais da metade dos problemas chega ao usuário antes de qualquer alerta interno (NETSCOUT, 2025) e um defeito em produção custa cerca de 80 vezes mais do que em desenvolvimento (DORA, 2024), essa falsa produtividade é especialmente cara.

2. Implementar IA com efetividade é gerar vantagem competitiva

Implementar IA com efetividade significa mover o ponteiro de negócio: tornar o produto ou serviço mensuravelmente melhor diante da concorrência. Esse é o critério que separa investimento de despesa.

Vantagem competitiva nem sempre se traduz em redução de custo ou substituição de pessoas. Com frequência ela aparece como velocidade de resposta, redução de risco em fluxos que geram receita, ou capacidade de validar continuamente o que antes só era verificado de forma pontual. Automatizar tarefas, criar workflows e montar automações de menor escala são insumos legítimos desse objetivo, desde que subordinados a ele.

A definição tem uma implicação direta sobre priorização: antes de perguntar o que se pode automatizar com IA, a organização precisa identificar qual job, se transformado, gera diferencial competitivo. A resposta orienta onde vale investir maturidade, e a próxima seção dá o vocabulário para medir essa maturidade.

3. Os oito níveis de maturidade agêntica

O uso de IA em engenharia não é binário. Existe uma escada de maturidade, e cada degrau representa um salto cognitivo na forma como a organização trabalha. A Tabela 1 sintetiza oito níveis, com o valor esperado em cada um.

Dois princípios derivam do modelo.

Os níveis não são uma hierarquia de qualidade pessoal. Estar no nível cinco não torna ninguém superior a quem está no nível dois. O degrau adequado depende do job a ser feito. Para análises simples, o nível dois pode ser suficiente e ótimo. Subir de nível tem custo, e custo só se justifica quando o job exige.

Não se salta degraus. Cada nível representa um aprendizado que viabiliza o seguinte. Uma organização cujo time ainda não usa autocomplete não chega a agentes rodando em background implementando MCP e skills. A tentativa de pular etapas, na prática de campo, falha de forma previsível: a ma-

Tabela 1. Os oito níveis de maturidade no uso de IA em engenharia.

Nível	Nome	O que caracteriza	Valor esperado
1	Tab complete	Autocomplete de código rudimentar. O modelo sugere a continuação do que está sendo digitado.	Ganho marginal de digitação.
2	IDEs agênticas	Interface de chat que navega a base de código, edita múltiplos arquivos e cria automações com aparência de autonomia.	Aceleração de tarefas pontuais de desenvolvimento.
3	Context engineering	Contexto rico é anexado de forma deliberada (documentação, integrações) para que o que o modelo gera faça sentido no domínio.	Saídas utilizáveis, menos retrabalho e descarte.
4	Compounding engineering	Primeiros feedback loops. A cada ciclo, o sistema aprende com os dados que ele próprio gera, e as execuções seguintes ficam mais precisas.	Melhoria composta ao longo do tempo.
5	MCP e skills	Super poderes acionáveis sob demanda. O agente consome ferramentas e procedimentos especializados conforme processa a solicitação.	Capacidade especializada reutilizável, menos prompt manual.
6	Harness	Orquestração complexa de LLMs que combina os níveis anteriores. Surgem as primeiras tarefas quase totalmente autônomas, com o humano como validador ou gerador de inputs.	Transformação efetiva do job. Autonomia supervisionada.
7	Autonomia ampliada	Agentes operando com supervisão mínima em escopos delimitados.	Cenário emergente, ainda restrito a certos domínios.
8	Autonomia contínua	Agentes trabalhando de forma autônoma e ininterrupta.	Hoje majoritariamente aspiracional para missão crítica.

turidade de contexto que sustentaria a autonomia ainda não existe.

4. O limiar de efetividade

A pergunta operacional não é em que nível estamos, isolada, e sim qual nível mínimo cada job exige para que a IA o transforme de fato. A esse nível mínimo chamamos limiar de efetividade, e ele varia conforme a natureza do trabalho.

Nos jobs centrais de qualidade e resiliência de missão crítica, esse limiar tende a ser alto. A razão é comum a eles: o valor está menos na execução pontual (gerar um teste, abrir um incidente) e mais em sustentar essa execução ao longo do tempo, cruzando sinais de múltiplas fontes sob um contexto unificado. Isso só se sustenta quando a orquestração de modelos opera sobre contexto estruturado. Abaixo desse patamar, a IA tende a consumir recursos sem produzir algo reutilizável.

Até os níveis intermediários é possível implementar diversas tecnologias sem obter uso efetivo de IA nesses jobs. Geração de testes end-to-end a partir de uma página, sem contexto bem construído, tende a consumir muito mais tokens do que deveria e a depender de bons prompts para que o teste funcional rode. Mesmo quando funciona, o resultado precisa ser transformado em algo reutilizável pelo restante do time, e é nesse ponto que a falta de estrutura cobra seu preço.

5. A reconfiguração dos papéis

Com IA, o volume de código cresce de forma acelerada. Como em qualquer indústria em que a produção é massificada, a qualidade tende a cair quando a produção de linhas de código vira commodity. Isso não significa entregar mais valor ao usuário final, e sim produzir mais artefato no mesmo intervalo.

> *A IA não vai substituir programadores, mas vai tornar mais fácil que programadores substituam todo o resto.*
>
> Atribuída a Naval Ravikant.

O ponto relevante para qualidade e resiliência é o seguinte: não haverá substituição de pessoas puramente por IA, e sim de profissionais por outros profissionais que dominem os níveis mais altos de maturidade. Quem interage com IDEs (desenvolvedores, QA, SRE) está muito à frente de quem não interage na corrida por implementar IA, inclusive para outras disciplinas. Dentro de qualidade e resiliência, dois papéis se redesenham.

O AI Quality Engineer deixa de operar roteiros manuais e artesanais e passa a sustentar uma cobertura viva, fundamentada em dados que mudam o tempo todo. Em vez de montar massa de dados à mão, opera workflows integrados ao banco que produzem massa sob demanda.

O AI Production Engineer substitui a figura que observa dashboards de telemetria e aciona o rollback manualmente. No lugar, opera um fluxo automatizado de análise de causa raiz com geração de evidência, comunicação aos times responsáveis e decisão fundamentada no contexto.

Esse redesenho tem urgência. A proporção de QA para desenvolvedores no mercado brasileiro costuma variar de 1:10 a 1:15, e cerca de 42% dos profissionais de QA não têm conforto com scripting (PractiTest, 2024). A IA bem implementada eleva esse papel em vez de pressioná-lo, desde que a organização atravesse os níveis de maturidade na ordem certa.

6. A convergência de qualidade e resiliência via contexto

À medida que o contexto do mundo real se torna disponível para os modelos, mundos que antes se conversavam pouco passam a se integrar. Tudo o que vive no domínio de qualidade de software passa a poder ser acessado por quem opera pós-produção, e vice-versa.

Um diagnóstico simples expõe a lacuna. Duas perguntas, feitas a operações reais:

- Para o time de qualidade: *você sabe quais incidentes críticos ocorreram em produção nos últimos três meses?*
- Para o time de confiabilidade: *você sabe quais deploys causaram incidentes nesse período?*

Na média, nenhum dos dois times tem a resposta. E os dados por trás dessas perguntas (deploys, incidentes, mudanças em telemetria, logs de API, sessões de usuário, quebras de teste) poderiam alimentar o trabalho um do outro. O contexto operacional, quando estruturado, transforma esses dois mundos em um só. Vale lembrar que 78% das organizações relatam incidentes que ocorrem sem qualquer alerta prévio: a informação existe, mas não está conectada.

A consequência é um profissional mais empoderado, capaz de navegar qualidade e resiliência como um único domínio. A própria expressão "qualidade e resiliência", usada de forma deliberada ao longo deste documento, reflete essa convergência.

7. A arquitetura de adoção: contexto, harness, capabilities

Saber o nível de maturidade só é útil quando existe a base que o sustenta. A ordem de implementação importa, e invertê-la é o erro mais comum observado em campo. A arquitetura tem três camadas, nesta sequência.

7.1. Grafo de contexto

Um grafo de contexto é um mapa de conhecimento sobre o ecossistema, materializado em dados estruturados que representam tudo o que importa para qualidade e resiliência: fontes de telemetria, logs de API, acesso a sessões de usuário em produção, acesso às bases de código, a pull requests e diffs, à gestão de incidentes e ao board.

A partir dessas fontes, o grafo conecta dados mais estruturados (produtos, ambientes, sessões, ferramentas) a estruturas de alto nível como regras de negócio. As regras de negócio são um caso especial: elas costumam ser conhecimento tácito, residente na cabeça das pessoas. Enquanto não ficam acessíveis ao modelo, é muito difícil gerar valor.

Um esclarecimento importante: grafo de contexto não é conector. Integrar uma ferramenta de versionamento não cria, por si só, um grafo de contexto. O grafo é o projeto deliberado de como a organização trata o próprio conhecimento. Sem ele, qualquer diagnóstico de maturidade fica abaixo do nível um.

7.2. Harness

O harness é a solução que orquestra como os modelos são utilizados e como consomem o grafo de contexto. As ferramentas mais conhecidas da categoria incluem assistentes de codificação agênticos e orquestradores de agentes. A maioria é multimodelo.

O anti-padrão dominante no mercado é contratar harness antes de ter um grafo de contexto estruturado. O resultado é previsível: a operação queima orçamento com provedores de modelo e passa a depender de sorte ou de poucas pessoas excepcionais capazes de produzir agentes hipercontextualizados apesar da baixa maturidade de contexto. Sem o grafo, o uso do harness degenera em repetir o mesmo prompt indefinidamente.

7.3. Capabilities

Capabilities são os super poderes que a IA entrega no dia a dia: gerar testes end-to-end, correlacionar uma falha a uma causa raiz, monitorar uma jornada crítica, validar uma regra de negócio dispersa por repositórios que ninguém domina por completo.

É possível ter algumas capabilities operando sobre IA sem as camadas anteriores, e elas até geram algum valor. Mas a tendência é que, sem contexto, as entregas fiquem superficiais e os jobs consumam tokens sem produzir diferencial competitivo. Além disso, atacar capabilities cedo demais amplifica riscos: exposição de segurança e a falsa sensação de produtividade já descrita.

A regra de sequência é direta: sem contexto, não compensa ter harness; com contexto e harness, a organização pode então investir em capabilities e entrega de valor real.

8. Padrões de implementação

Os três padrões a seguir só existem quando contexto e harness estão implementados. Os exemplos foram anonimizados.

8.1. Análise de causa raiz cruzando teste e telemetria

Considere uma API de pagamentos com um plano de testes totalmente automatizado, organizado em módulos, suítes e casos escritos em linguagem natural acessível a pessoas técnicas e não técnicas. Cada caso roda sob um motor de testes end-to-end, de forma agendada ou integrada ao pipeline de CI/CD.

> **Exemplo de caso de teste, em linguagem natural:** "Enviar POST ao endpoint de pagamento com o header de simulação de falha; verificar se a API retorna o código HTTP esperado e se o corpo da resposta contém o tratamento de erro previsto." O mesmo caso serve para teste manual, validação e automação.
> **Cadência:** o caso roda a cada 5, 30 ou 60 minutos, ou a cada deploy via pipeline de CI/CD.

Um caso de resiliência (simular falha HTTP 503 via header e verificar o tratamento correto) começa a falhar de forma consistente. Como cada jornada está hipercontextualizada (hierarquia dentro do plano, tipo de teste, histórico de execuções, defeitos associados, testes relacionados na mesma suíte), uma primeira análise indica algo inconsistente: todas as execuções esperam um código e recebem outro.

O salto vem do contexto. Conectando a telemetria da aplicação ao mesmo fluxo de análise, o sistema correlaciona o instante da falha do teste com os eventos do APM e conclui a causa raiz: o header de simulação de falha nunca é interceptado pelo serviço. O teste, do lado do cliente da API, parecia

falhar; a evidência de back end mostra um bug de integração na aplicação.

> **Exemplo de fontes cruzadas na investigação:** motor de testes end-to-end (visão do cliente da API), APM de telemetria (visão de back end) e, quando necessário, aprofundamento via trace distribuído. Concluída a análise, o defeito pode ser aberto diretamente na ferramenta de gestão de issues integrada.

O valor não está em nenhuma análise isolada. Está em democratizar análises complexas para qualquer pessoa, independentemente do nível técnico, apenas por existir um contexto pré-montado.

8.2. Sessões reais de usuário convertidas em testes autônomos

O segundo padrão dispensa a geração manual de testes. Um script captura sessões reais em produção ou pré-produção: onde o usuário clica, quais requisições ocorrem, qual o payload de cada uma. Essas sessões viram contexto.

> **Exemplo de jornada capturada:** um fluxo transacional em uma aplicação web. O usuário localiza um registro por um identificador, consulta os dados retornados, parametriza uma operação (valores e opções), revisa o resultado e confirma o envio. A sessão é convertida em duas saídas: uma descrição semântica do que aconteceu e as referências de seletores que a automação web vai usar.
> **Exemplo de artefatos não estruturados transformados em contexto:** especificação técnica das regras de negócio, histórico de defeitos conhecidos e planilha de testes manuais preexistente. O modelo enriquece o contexto do fluxo de forma incremental, à medida que cada artefato é anexado.

Sobre esse contexto, uma capability de geração planeja os casos de teste priorizados (incluindo cenários de exceção, dados da própria aplicação e histórico de incidentes) e os converte em automação de forma autônoma: o sistema clona o repositório, escreve os scripts, valida que funcionam, integra ao CI/CD, revisa o código e abre um pull request.

A orquestração é auditável. Cada execução expõe os passos em fila, os subagentes disparados, as skills acionadas e o raciocínio do orquestrador, o que dá controle sobre o que roda e por quê. No fim, há um repositório com todos os commits do agente, pronto para inspeção humana.

Em operações reais, esse fluxo gerou da ordem de 86 testes complexos em torno de 40 minutos, de forma autônoma. O papel humano se desloca da produção para a validação, com revisão de segurança e checagem de alucinações antes de confiar no resultado.

8.3. Ambientes legados de baixa maturidade

O cenário ideal de dados estruturados quase nunca existe. Um padrão eficaz para ambientes de baixa maturidade vem de uma operação de faturamento e emissão fiscal de grande porte.

O contexto: um produto de faturamento e emissão fiscal, múltiplas aplicações e repositórios, ownership difusa entre squads, e dependências em microsserviços internos legados sem documentação. Um exemplo concreto dessa dependência: o microsserviço que funciona como motor de cálculo tributário pertence a uma squad, que por sua vez depende de outro microsserviço interno legado que ninguém documenta. O risco que não pode se materializar não é um botão quebrado ou um fluxo secundário com erro, e sim uma regra de negócio violada, como uma alíquota incorreta na emissão de uma nota fiscal.

A solução não exigiu automação de testes complexos nem massa de dados sintética, frequentemente inviável nesse tipo de produto. Em vez disso, construiu-se um motor de regras de negócio orientado à base de código. O sistema ingere a base (e, quando disponível, documentos e sessões) e gera um mapa de contexto do produto: por jornada, identifica onde cada regra de negócio vive no código, mesmo quando ela está fragmentada entre banco, back end e front end em repositórios distintos.

> **Exemplo de regra de negócio mapeada:** "uma nota fiscal só pode ser emitida com a alíquota correspondente ao estado de destino e ao tipo de produto da operação." O sistema registra a decisão da regra e os arquivos onde ela reside, mesmo dispersa entre camadas.
> **Exemplos de defeitos de regra que o padrão previne:** uma operação interestadual emitida com a alíquota interna; um produto com tributação especial enquadrado na regra geral sem passar pela exceção formal exigida.

Sobre esse mapa, opera uma camada de governança: qualquer pull request que viole uma regra crítica de negócio é bloqueado, e a aprovação passa a ser de quem governa a regra de negócio, não apenas da revisão padrão de código.

> **Exemplo de bloqueio:** um pull request que altera a tabela de alíquotas na camada de front end é interceptado antes do merge e encaminhado para aprovação de negócio.

O efeito prático é interromper a entrada de defeitos críticos em produção. Em operações que despejavam dezenas de defeitos a cada centena de deploys, o padrão reduz drasticamente esse número. O custo é um processo mais burocrático, equivalente a um code review orientado a negócios, e a contrapartida é proteção direta contra a exposição fiscal.

Uma observação de contexto local: no Brasil, a complexidade e a baixa padronização dos dados tornam a geração de massa sintética ainda mais difícil, o que reforça o valor de abordagens orientadas à base de código em ambientes legados.

9. A economia da autonomia

Quanto mais autônomo o uso de IA, mais o custo e o tempo se tornam variáveis de primeira ordem. Gerar centenas de casos de teste de forma autônoma sem critério não gera valor se a conta inviabiliza a operação. Vantagem competitiva continua sendo o critério.

Três disciplinas tornam a autonomia sustentável:

- **Inteligência de custo e guard rails:** limites que impedem o mau uso e tornam a conta previsível.
- **Roteamento de modelos:** trabalhos sofisticados endereçados a modelos de fronteira, trabalhos simples a modelos de baixo custo. A escolha do modelo por tipo de tarefa é parte do projeto, não um detalhe.

- **Monitoramento em tempo real:** observar qual modelo é mais útil para cada tipo de tarefa e ajustar continuamente.

Há também a disciplina de decidir o que vira insumo. Nem todo bug deve gerar um caso de teste. Definir os critérios de abordagem (qual tipo de teste para qual tipo de aplicação e plano) é parte do trabalho de engenharia que sustenta a autonomia.

Por fim, todo dado gerado pela IA deve permanecer acessível. Uma decisão de arquitetura recomendável é expor automaticamente, via protocolo de contexto, tudo o que o sistema produz, de modo que planos de teste, evidências e regras fiquem disponíveis para consulta posterior, inclusive como base de conhecimento para outras equipes.

10. Roadmap de adoção

A adoção começa pelo diagnóstico, não pela ferramenta.

1. **Verifique a base.** Antes de medir o nível, confirme se existe um grafo de contexto bem construído. Sem ele, a maturidade efetiva fica abaixo do nível um e o diagnóstico não se sustenta.

2. **Diagnostique o nível atual por job.** Para cada job-to-be-done relevante, identifique em que degrau a operação está hoje e qual o limiar de efetividade daquele job.

3. **Avance um degrau de cada vez.** Identificado o degrau, mire o próximo. Um exemplo de baby step: uma operação no nível três (contexto fornecido manualmente via documentos e arquivos de instrução) avança ao nível quatro implementando um feedback loop simples, em que cada teste gerado torna o próximo mais rápido. Isso costuma exigir um mecanismo de eventos e um repositório de conhecimento persistente.

4. **Produtize o conhecimento em vez de mantê-lo em texto solto.** Procedimentos trafegados apenas como arquivos de texto são difíceis de versionar e fáceis de perder o controle. Empacotá-los em capabilities acionáveis por qualquer pessoa, sem depender da manutenção manual de instruções, é o salto que leva do nível cinco em diante.

5. **Calibre a ambição ao job.** Harness é caro. Faz sentido para jobs cujo valor competitivo justifica autonomia. Para análises simples, parar em um nível mais baixo pode ser a decisão correta.

11. Conclusão

A insatisfação com o retorno da IA em engenharia raramente é um problema de modelo. É um problema de método. Organizações medem atividade quando deveriam medir vantagem competitiva, e contratam autonomia antes de construir o contexto que a sustenta.

Para qualidade e resiliência de sistemas de missão crítica, o caminho é nomeável: definir valor como diferencial competitivo, diagnosticar a maturidade por job, construir o grafo de contexto antes do harness, e só então investir em capabilities. O limiar para transformar de fato esses jobs costuma ser alto, no patamar em que harness e contexto estruturado se combinam, e os papéis de qualidade e confiabilidade se redesenham em torno de cobertura viva e análise de causa raiz fundamentada em contexto.

A tecnologia para isso já existe. O diferencial passa a ser a disciplina de implementá-la na ordem certa.

Apêndice: glossário

Grafo de contexto mapa estruturado do conhecimento sobre o ecossistema de software (fontes de dados, produtos, jornadas, regras de negócio) acessível aos modelos.

Harness camada que orquestra o uso de modelos e o consumo do grafo de contexto; inclui assistentes agênticos e orquestradores de agentes.

Capability função de alto valor entregue pela IA sobre a base de contexto e harness (geração de testes, análise de causa raiz, validação de regra de negócio).

MCP e skills mecanismos que dão ao agente acesso, sob demanda, a ferramentas e procedimentos especializados.

Feedback loop ciclo em que o sistema aprende com os dados que gera, melhorando execuções subsequentes (base do compounding engineering).

Job-to-be-done tarefa concreta que a organização precisa realizar, usada como unidade de diagnóstico de maturidade.

Limiar de efetividade nível mínimo de maturidade em que a IA transforma de fato um job, em vez de apenas consumir recursos.

AI Quality Engineer / AI Production Engineer papéis que sustentam, respectivamente, cobertura viva de qualidade e fluxos automatizados de confiabilidade em produção.

Fontes dos dados citados. Adoção de IA praticamente universal entre desenvolvedores: DORA, 2025. Mais da metade dos problemas descobertos primeiro pelo usuário: NETSCOUT, 2025. 78% dos incidentes ocorrem sem alerta prévio: estudo de confiabilidade, 2026. Defeito em produção cerca de 80 vezes mais caro que em desenvolvimento: DORA, 2024. Proporção QA:desenvolvedores no Brasil de 1:10 a 1:15: prática de mercado. 42% dos profissionais de QA sem conforto com scripting: PractiTest, 2024.

Publicado pela comunidade Bug in Prod • 2026