\newpage

Sobre este material

Este PDF descreve o método que aplicamos na **Aastera Consulting** para diagnosticar problemas técnicos em sistemas — desde um endpoint que retorna `500` esporadicamente até uma migração que não termina, passando por incidentes silenciosos que só geram sintomas semanas depois.

Não é um material acadêmico. É o processo que dev experiente aplica — muitas vezes sem perceber — quando precisa entender um bug que **importa de verdade**: aquele que está custando dinheiro, derrubando a confiança do cliente ou impedindo o time de evoluir.

Compartilhamos publicamente porque acreditamos que a maior parte do que separa um time técnico bom de um time técnico mediano não está no domínio de uma ferramenta nova, mas na **disciplina com que se aborda o desconhecido**.

Boa leitura.

— Equipe Aastera Consulting

\newpage

1. Por que a maioria dos times debuga errado

Quando algo quebra em produção, a primeira reação da maioria dos times técnicos é **mexer**. Trocar a configuração. Reiniciar o serviço. Aplicar a primeira sugestão do ChatGPT. Comentar a linha que "parece suspeita". Subir e ver no que dá.

Às vezes funciona. E é aí que mora o problema: **quando funciona por acidente, ninguém aprendeu nada**.

O bug volta na próxima semana, com outra cara. O time gasta horas tentando de novo. O cliente liga insatisfeito. O dev que "consertou" não consegue explicar o que fez — e isso vai se acumulando como dívida técnica invisível, até que ninguém mais entende como o sistema funciona por dentro.

Esse padrão tem nome: **tentativa e erro**. E ele não é uma falha de capacidade individual. É uma falha de método.

O que devs experientes fazem diferente

Devs com cicatrizes — gente que já operou produção crítica, já foi acordada às 2h da manhã pra apagar incêndio, já viu a mesma classe de bug aparecer em três sistemas diferentes — desenvolvem um processo. Quase sempre o mesmo processo. E quase sempre sem conseguir colocar em palavras.

Quando você pede pra explicar, ouve coisas como "feeling", "experiência", "intuição". Mas se observar por tempo suficiente, percebe que **é sempre o mesmo loop**, repetido com diferentes níveis de profundidade conforme o caso exige.

Esse loop é, em essência, o **método científico aplicado a sistemas técnicos**. O mesmo que você aprendeu no ensino fundamental: observação, pergunta, hipótese, teste, análise, conclusão. Adaptado pra debug, vira o que descrevemos a seguir.

O custo de não ter método

Antes de entrar nas etapas, vale fixar a régua. O custo de debug sem método costuma aparecer em três lugares:

- **Tempo desperdiçado**: o que era pra levar 40 minutos vira tarde inteira. Multiplicado por todos os incidentes do mês, é um time invisivelmente menos produtivo.
- **Recidiva**: como ninguém entendeu o que causou o bug, ele volta. Geralmente em outra forma. O time apaga o mesmo incêndio em ciclos.
- **Erosão de confiança**: cliente percebe quando o sistema "está sempre com algum problema". A relação se desgasta sem que se possa apontar uma única falha grande.

Cada uma dessas três coisas é evitável com método. E o método, ao contrário do que parece, não exige nenhuma ferramenta nova. Exige disciplina.

\newpage

2. As 7 etapas, em vista de cima

O método se divide em sete etapas. Em incidentes triviais, você passa por todas em minutos. Em incidentes complexos, pode levar dias — mas o caminho é o mesmo.

```
┌─ 1. Observação      → Coletar evidência antes de pensar em solução
│
│  2. Pergunta         → Formular o problema com precisão
│
│  3. Hipótese         → Propor uma causa verificável
│
↓  4. Teste isolado    → Mudar uma variável só
│
│  5. Análise          → A hipótese sobreviveu ou caiu?
│
└─ Se caiu → volta pra 3 com hipótese nova

   6. Conclusão        → Causa-raiz identificada
   7. Generalização    → Evitar essa classe de bug no futuro
```

A etapa 5 (Análise) é o coração do loop. Se a hipótese cai, o método não te diz que falhou — te diz que **você eliminou uma possibilidade** e precisa propor outra. Esse é o ganho silencioso do método: até quando o teste dá negativo, você está progredindo.

As próximas seções tratam cada etapa em detalhe.

\newpage

3. Etapa 1 — Observação

O que é: coletar todos os dados que existem sobre o problema, antes de pensar em qualquer solução.

Por que importa: a maior parte dos diagnósticos errados começa porque o dev pulou direto pra "deve ser X" sem olhar o que o sistema está dizendo. O sistema quase sempre está te dando informação. O problema é que ninguém parou pra ler.

O que coletar

Em ordem de prioridade:

1. **Mensagem exata do erro**. Não a memória do erro. Não "deu erro de undefined". O texto literal, completo, com aspas e tudo.
2. **Stack trace completo**. Não só a primeira linha. Quem chamou quem chamou quem.
3. **Contexto**: quando aconteceu, com qual usuário, em qual ambiente, depois de qual ação.
4. **Frequência**: aconteceu uma vez? sempre? em qual padrão (toda quinta-feira? sempre depois de X minutos?)?
5. **Diferenças**: o que mudou recentemente? Deploy novo? Banco atualizado? Cliente importou massa de dados grande?
6. **Logs adjacentes**: o que mais o sistema estava fazendo no momento? Job rodando? Pico de tráfego?

Como não fazer

> "*Tá dando aquele erro de banco lá, sabe?*"

Esse é o ponto de partida típico — e é também o sinal claro de que ninguém observou de verdade. Boa observação produz uma descrição que **outro dev consegue reproduzir mentalmente** sem precisar perguntar nada de volta.

Sinal de que você fez direito

Se você consegue escrever um parágrafo de 4-5 frases descrevendo o problema, com fatos concretos e referências exatas (arquivo, linha, momento), você terminou a etapa 1. Se não consegue, **você ainda está nela**.

> *Curiosidade: 30 a 50% dos bugs que chegam à Aastera Consulting são resolvidos só com observação rigorosa. O time já tinha tudo que precisava — só não tinha olhado direito.*

\newpage

4. Etapa 2 — Pergunta

O que é: transformar a observação numa pergunta específica e respondível.

Por que importa: perguntas vagas geram diagnósticos vagos. "Por que o sistema está lento?" é uma pergunta inútil — você não tem como responder ela diretamente. "Por que o endpoint `GET /api/orders` está respondendo em mais de 2s desde o deploy de quarta?" é uma pergunta com a qual se pode trabalhar.

A regra: a pergunta precisa ser falseável

Você precisa conseguir responder **sim ou não** com um teste. Se a pergunta admite "talvez", "depende" ou "às vezes", refine.

Pergunta ruim	Pergunta boa
Por que o app está bugado?	Por que a tela de checkout não fecha o pedido quando o usuário tem mais de 10 itens no carrinho?
Por que o sistema é lento?	Por que `GET /relatorios/vendas` leva mais de 5s quando o filtro de data é maior que 30 dias?

Por que o usuário não consegue logar?	Por que o login falha com 401 especificamente pra usuários criados antes de 2024?

Como decompor

Quando a pergunta inicial é grande demais, quebre em sub-perguntas que possam ser respondidas isoladamente. Exemplo:

- Pergunta grande: "*Por que o checkout não está finalizando?*"
- Sub-perguntas:
  - O front está enviando o pedido pra API?
  - A API está recebendo?
  - A API está respondendo sucesso ou erro?
  - Se erro, qual?
  - Se sucesso, o pedido está sendo persistido?
  - Se persistido, está com status correto?
  - Se status correto, por que o front mostra "falhou"?

Cada uma dessas perguntas tem resposta binária. Você responde uma de cada vez e elimina possibilidades.

Sinal de que você fez direito

Você consegue dizer **exatamente qual teste responderia a pergunta** que você acabou de fazer. Se não consegue, sua pergunta ainda não está boa o suficiente.

\newpage

5. Etapa 3 — Hipótese

O que é: propor uma explicação **verificável** para o problema observado.

Por que importa: hipótese é o ponto onde a maioria das pessoas pula etapas. "Deve ser cache" não é hipótese — é palpite. "O cache do Redis está retornando dados antigos porque o TTL está configurado pra 1h e a invalidação no save do pedido não está sendo chamada" é uma hipótese: tem mecanismo proposto e pode ser testada.

A regra: hipótese precisa ter mecanismo

Toda boa hipótese tem três partes:

1. **O que causa**: qual componente, qual condição, qual sequência de eventos
2. **Como causa**: o mecanismo pelo qual aquilo gera o sintoma observado
3. **Como verificar**: qual teste específico confirmaria ou refutaria

Se você não consegue formular as três, você ainda está chutando. Volte pra etapa 2 (pergunta) e refine.

Quantas hipóteses propor

Pelo menos **três**. Por dois motivos:

- Se você só pensa em uma, vai ter viés de confirmação e enxergar evidência só pra um lado.
- Algumas hipóteses são caras de testar e outras baratas. Listar várias permite ordenar pelo custo do teste.

Exemplo: bug = "endpoint às vezes retorna 500".

Hipótese	Mecanismo	Custo de teste
H1: query SQL ocasionalmente faz lock	Deadlock entre transações simultâneas no banco	Médio — habilitar log de slow queries por 1 dia
H2: instância do worker fica sem memória	Job grande paralelo enchendo heap	Baixo — métricas do container
H3: API externa de pagamento timeout	3rd party lento, retry insuficiente	Baixo — log de chamadas outbound

Comece pela mais barata. Se ela responder sim, você tem causa-raiz. Se responder não, você eliminou uma e passa pra próxima.

Sinal de que você fez direito

Você tem 2-4 hipóteses, cada uma com mecanismo claro e teste definido, ordenadas pelo custo de testar. Está pronto pra etapa 4.

\newpage

6. Etapa 4 — Teste isolado

O que é: mudar **uma variável só** e observar o resultado.

Por que importa: se você muda 3 coisas ao mesmo tempo e o bug some, você não sabe qual das 3 era a real. Acabou de fazer "correção" sem aprender nada — e provavelmente carregou 2 mudanças desnecessárias pra produção.

A regra: uma variável por teste

Isso significa: comparar dois cenários idênticos em tudo exceto uma coisa. Se você está testando se é o cache, **garante que tudo o mais é igual** entre a versão com cache e a sem.

Soa óbvio. Não é. Os erros mais comuns:

- **Mudar código e deploy ao mesmo tempo**: ambos podem ter mudado o comportamento. Se possível, testa o código antes do deploy.
- **Testar em ambiente diferente**: bug acontece em prod, você testa em staging. Staging tem outra versão do banco, outro volume de dados, outro tráfego. Isolamento quebrado.
- **Testar com dados diferentes**: você roda manualmente com 1 registro, mas o bug acontece com 10000. Os comportamentos podem ser categoricamente diferentes.

Estratégia: binary search no código

Quando o problema está num código grande e você suspeita de uma região, mas não sabe onde:

1. Comenta metade do código suspeito
2. Roda
3. Bug some? Está na metade comentada. Volta a metade, comenta a outra metade dela.
4. Bug continua? Está na outra metade. Comenta uma metade dessa.

5. Repete.

Em 10 iterações você isolou em ~0,1% do código original. Funciona pra reprodução técnica, não pra problemas de produção (onde comentar código não é opção).

Quando não dá pra isolar em produção

Em sistema crítico você nem sempre pode "mexer pra ver". Saídas:

- **Réplica**: criar ambiente idêntico ao prod com dados sintéticos similares
- **Shadow traffic**: duplicar tráfego real pra ambiente de teste, comparar respostas
- **Feature flag**: ligar/desligar comportamento pra subconjunto de usuários

Custa configurar. Custa menos do que adivinhar.

Sinal de que você fez direito

Você consegue afirmar: "*quando A está presente, o bug acontece; quando A não está, o bug não acontece; tudo o mais é igual.*" Se você tem essa frase, sua causa-raiz está praticamente identificada.

\newpage

7. Etapa 5 — Análise

O que é: olhar o resultado do teste e decidir se a hipótese sobreviveu, caiu ou ficou ambígua.

Por que importa: o maior erro nesta etapa não é técnico — é cognitivo. É o **viés de confirmação**: enxergar o resultado e interpretar como prova da hipótese, mesmo quando os dados são ambíguos ou contraditórios.

Os três resultados possíveis

Todo teste produz um dos três:

1. **Confirmação clara**: a hipótese previu X, X aconteceu, nada mais explica X. Você tem causa-raiz.
2. **Refutação clara**: a hipótese previu X, aconteceu Y, e Y é incompatível com a hipótese. Hipótese eliminada. Volta pra etapa 3 com outra.
3. **Ambiguidade**: o resultado é compatível com a hipótese mas também com outras coisas. Você não sabe.

Ambiguidade é o caso mais perigoso. É onde dev experiente pisa no freio e dev iniciante acelera.

A regra: na dúvida, é ambíguo

Se você ouvir a si mesmo pensando "eu acho que é isso", "deve ser", "provavelmente" — você está no caso 3. Não declare vitória. Refine o teste pra eliminar a ambiguidade.

> *Exemplo real: time mudou o pool de conexões do banco de 10 pra 50 conexões e o bug "sumiu". Declararam vitória. Duas semanas depois o bug voltou, dessa vez pior. Causa real: era um leak de conexão em outro lugar. Aumentar o pool só mascarou.*

Quando a hipótese cai

Boa notícia: cada hipótese eliminada te aproxima da causa-raiz. Não trate como fracasso. Anote o que testou, por que caiu, e siga pra próxima hipótese da sua lista da etapa 3.

> *"Falsificação progressiva" é o nome técnico disso. É como funcionam todas as ciências aplicadas — e debug de produção é exatamente isso.*

Sinal de que você fez direito

Você consegue afirmar: "*hipótese H1 sobreviveu/caiu/ficou ambígua porque o teste mostrou X.*" Sem "achismos". Sem "imagino que".

\newpage

8. Etapa 6 — Conclusão

O que é: declarar a causa-raiz e separar do **sintoma**.

Por que importa: a confusão sintoma vs causa-raiz é o segundo maior pecado em debug (o primeiro é tentativa e erro). Você "consertou" o sintoma — ótimo. Mas a causa real continua lá, esperando aparecer de outro jeito.

Sintoma vs causa-raiz

Sintoma é o que o usuário ou o monitoramento percebe. "*Endpoint retorna 500.*"

Causa-raiz é o mecanismo por trás. "*Um job assíncrono está atualizando a tabela de usuários sem WHERE, zerando o campo* `email` *de todos. Quando o endpoint busca usuário pra autenticar, recebe* `null` *e quebra na linha 47.*"

Tampar o sintoma seria adicionar `?? ''` na linha 47 e seguir a vida. O bug volta amanhã.

A regra: "os cinco porquês"

Pra cada conclusão que você chegar, pergunte "por quê" cinco vezes. Se a quinta resposta ainda parecer profunda, você provavelmente está perto da causa-raiz. Se você desce com "porquês" e cai num "porque o código está assim", você parou cedo demais.

Exemplo:

1. Por que o endpoint retornou 500? → Porque o usuário veio null.
2. Por que o usuário veio null? → Porque o email não bateu na query.
3. Por que o email não bateu? → Porque o campo email está vazio no banco.
4. Por que o campo está vazio? → Porque um job de migração de dados atualizou sem WHERE.
5. Por que esse job atualizou sem WHERE? → Porque foi escrito sem revisão e nunca testado em ambiente com dados reais.

A causa-raiz não é "endpoint retorna 500". É "processo de revisão de scripts de migração é inexistente". O fix pontual é restaurar emails. O fix de raiz é instituir code review pra scripts de migração.

Sinal de que você fez direito

Você consegue escrever em 2 parágrafos: (a) o que aconteceu, (b) por que aconteceu, e (c) qual é o fix imediato vs qual é o fix estrutural.

\newpage

9. Etapa 7 — Generalização

O que é: olhar a causa-raiz e perguntar: **que outras situações têm essa mesma classe de problema, e ainda não estouraram?**

Por que importa: cada bug resolvido é uma oportunidade gratuita de eliminar uma classe inteira. Se você só conserta o caso específico e não generaliza, o mesmo tipo de problema vai aparecer em outro lugar — e você vai gastar tempo de novo.

Três perguntas a fazer

Sempre que terminar um diagnóstico, antes de fechar o ticket:

1. **Onde mais isso pode estar acontecendo?** Se foi um job sem WHERE, varre os outros jobs. Se foi falta de validação num form, audita os outros forms. Se foi race condition num lugar, procura race conditions em lugares parecidos.
2. **Que mecanismo no nosso processo permitiu isso?** Code review faltando? Teste não escrito? Documentação ausente? Falta de monitoramento? Resposta normalmente aponta pra mudança de processo, não de código.
3. **Como detectaríamos isso mais cedo da próxima vez?** Quase sempre é instrumentação: log, alerta, métrica, teste automatizado. Cada bug bem analisado deveria deixar pelo menos um "trip wire" novo pra detectar a próxima ocorrência.

A regra: deixe a base melhor que encontrou

O fix imediato é o mínimo. A generalização é o que separa um time que aprende de um time que apaga incêndios em loop.

Não precisa ser obra de arte. 30 minutos pra adicionar um teste de regressão. 1 hora pra escrever um post-mortem curto. 15 minutos pra atualizar a documentação. Compound interest técnico.

Sinal de que você fez direito

Você não fechou o ticket sem deixar pelo menos **uma** das três coisas: teste novo, monitoramento novo, ou nota de processo.

\newpage

10. IA no debug: armadilhas comuns

Você provavelmente está usando IA pra debug — ChatGPT, Claude, Cursor, Copilot. Não tem problema: bem usada, IA acelera várias etapas do método. Mal usada, IA piora o vício de tentativa e erro disfarçando de "produtividade".

As três armadilhas mais comuns

1. Pular direto pra hipótese sem observar. Cola o stack trace na IA e pergunta "como conserto isso?". A IA, que não viu seu sistema, palpita confiantemente. Você aplica. Quebra outra coisa. Pergunta de novo. Loop infinito.

Fix: faça a etapa 1 (Observação) ANTES de envolver a IA. Quando perguntar, traga contexto: ambiente, frequência, mudanças recentes.

2. Aceitar a primeira resposta como verdade. IA fala com tom de certeza mesmo quando está errando. Resposta segura ≠ resposta certa. Aplicar sem testar é a versão moderna de "deve ser X".

Fix: use a resposta da IA como **uma das hipóteses** (etapa 3), não como conclusão. Verifique antes de aplicar.

3. Pedir conserto direto em vez de pedir investigação. "Como conserto?" → IA inventa um conserto. "Que testes me ajudariam a descobrir a causa raiz disso?" → IA te dá um plano de investigação.

A segunda pergunta é infinitamente mais valiosa. Use IA como par de investigação, não como oráculo.

Quando IA brilha no método

- **Observação**: ajuda a interpretar stack trace longo, traduzir mensagem de erro obscura, identificar padrão em log gigantesco.
- **Hipótese**: bom pra brainstorming. "Que possíveis causas pode ter um endpoint que responde 500 ocasionalmente sob carga?" — IA lista 10, você filtra pelos plausíveis.
- **Teste**: pedir queries SQL pra investigar, comandos shell pra extrair métrica, snippets de código pra reproduzir. Bom uso.

Quando IA atrapalha

- **Análise** (etapa 5): IA não tem acesso ao seu sistema, não pode dizer se uma hipótese caiu ou não. Análise é trabalho seu.
- **Conclusão e generalização**: requer conhecimento contextual da sua arquitetura, do seu negócio, da sua equipe. IA chuta.

> *Princípio simples: IA é boa pra **gerar opções**. É ruim pra **decidir**. Use ela na divergência (gerar hipóteses, opções de teste), não na convergência (escolher causa-raiz, definir fix).*

\newpage

11. Anti-padrões mais comuns

Sinais de que o método foi pulado. Se você vê alguns desses no seu time, ainda há ganho fácil a capturar.

Anti-padrão	O que está acontecendo	Etapa pulada
"Reiniciar resolveu"	Sintoma foi removido sem entender a causa. Volta.	1, 3, 6
Stack trace lido só na primeira linha	Falta de observação completa.	1
"Deve ser X" sem teste	Hipótese tratada como conclusão.	3, 4
Várias mudanças num commit "pra ver no que dá"	Sem isolamento de variáveis.	4
"Funcionou no meu"	Ambiente diferente do que reproduz o bug.	4

"Provavelmente é isso"	Análise sem certeza, declaração de vitória prematura.	5
Ticket fechado sem post-mortem	Sem generalização.	7
Mesma classe de bug reaparece a cada 2 meses	Causa-raiz nunca identificada de verdade.	6, 7

\newpage

12. Checklist rápido — para usar durante um incidente

Imprima ou mantenha aberto:

1. Observação

- [ ] Tenho a mensagem exata do erro?
- [ ] Tenho o stack trace completo?
- [ ] Sei quando, com quem, em qual ambiente?
- [ ] Sei a frequência e padrão?
- [ ] Sei o que mudou recentemente?

2. Pergunta

- [ ] Minha pergunta é específica o suficiente pra ter resposta binária (sim/não)?
- [ ] Sei exatamente qual teste a responderia?

3. Hipótese

- [ ] Listei pelo menos 3?
- [ ] Cada uma tem mecanismo (não só "deve ser X")?
- [ ] Estão ordenadas pelo custo de testar?

4. Teste isolado

- [ ] Estou mudando uma variável por vez?
- [ ] Os dois cenários comparados são idênticos exceto pela variável?
- [ ] Ambiente, dados e versão de código são os mesmos?

5. Análise

- [ ] Resultado é claro (confirmou, refutou) ou ambíguo?
- [ ] Se ambíguo, refinei o teste antes de declarar conclusão?
- [ ] Estou usando "porque o teste mostrou X" ou "porque eu acho"?

6. Conclusão

- [ ] Sei diferenciar sintoma de causa-raiz?
- [ ] Passei pelos 5 porquês?
- [ ] Sei qual o fix imediato e qual o fix estrutural?

7. Generalização

- [ ] Onde mais isso pode estar acontecendo?
- [ ] Que processo nosso permitiu isso?
- [ ] Vou deixar pelo menos um teste/monitor/nota nova?

\newpage

Sobre a Aastera Consulting

Esse método é o que aplicamos diariamente com clientes recorrentes da **Aastera Consulting**. Quando uma empresa nos contrata, não fazemos só "consertar bugs" — instalamos o método no time, deixamos infraestrutura pra detectar problemas mais cedo, e direcionamos decisões técnicas pra que sistemas envelheçam bem.

Atendemos empresas que querem:

- Reduzir incidentes em produção sem precisar contratar um diretor de tecnologia
- Direcionar o time técnico (1, 2, 3 desenvolvedores internos) com experiência sênior
- Decidir bem sobre tecnologia, arquitetura e processo
- Diagnosticar problemas pontuais que estão custando tempo e dinheiro

Trabalhamos em três modelos:

- **Recorrente** — acompanhamento contínuo, a partir de R$ 490/mês, ajustado por caso
- **Pontual** — diagnóstico técnico de um problema específico, R$ 1.500 fixo, entrega em 7 dias úteis
- **Projeto** — migrações, refatorações, implementações com escopo definido

O ponto de partida é sempre uma conversa gratuita de 30 minutos pra entender seu cenário.

Agende em: cal.com/aastera/diagnostico-30min

Saiba mais: tech.aastera.com/consultoria

Contato direto: tech@aastera.com · WhatsApp +55 54 9 9919-9990

Aastera Consulting é uma linha de serviço da AN Tecnologia LTDA (CNPJ 44.488.331/0001-98), com base em Passo Fundo / RS.