ENGENHARIA DE SOFTWARE & AGENTES INTELIGENTES

Sandeco Macedo

Os meus sobrinhos-netos Theo e Helena. Titio ama vocês!

Copyright © 2026


Meus Livros sobre
Inteligência Artificial
crewAI
Para iniciantes
Agentes
Inteligentes
vol. 1
Sandeco Macedo
Para iniciantes com IA
Python para
Inteligência
Artificial
Sandeco Macedo
Tudo o que você precisa saber do Python antes de criar suas IAs
Prompts em Ação
Engenharia de Prompts
para leigos
Sandeco Macedo
Resultado
das minhas
pesquisas em IA

Sumário

CAPÍTULO 1
A Morte do Vibe Coding

"Sandeco, você ficou louco? Que título é esse?" Eu sei. Parece radical. Mas antes de fechar o livro, me dá uma chance de explicar. Quando eu digo que o Vibe Coding precisa morrer, não estou dizendo que programar com IA é ruim. Pelo contrário. Estou dizendo que a forma como a maioria das pessoas programa com IA hoje é insustentável; e que, para a gente de fato evoluir no desenvolvimento de software com inteligência artificial, essa fase precisa ficar para trás. No lugar dela, precisa nascer algo com fundamento. Algo que este livro vai chamar de **AI Engineering**.

Tudo começou em fevereiro de 2025, quando Andrej Karpathy, ex-diretor de IA da Tesla e um dos fundadores da OpenAI, ou seja, alguém com currículo suficiente para ser levado a sério, publicou uma reflexão sobre seu próprio modo de programar com LLMs. Ele descreveu uma prática que chamou de **vibe coding**: você larga o controle, entra em 'flow', aceita o que a IA sugere sem questionar, e vai construindo por impulso e intuição. O nome pegou na hora. A prática já existia há meses. Karpathy só colocou palavras no que milhares de desenvolvedores já faziam; e alguns até postavam no LinkedIn com muito orgulho.

A ascensão do **vibe coding** não é um acidente nem uma conspiração. É o resultado direto da democratização das ferramentas de IA generativa. Quando o **Claude Code**, o Cursor, o GitHub Copilot e similares chegaram ao mercado acessível, qualquer pessoa com uma ideia e uma conexão com a internet passou a conseguir gerar código. Designers codando. Empreendedores construindo MVPs em um final de semana. Estudantes entregando projetos que antes levariam um semestre. A produtividade explodiu. E com ela, também explodiu a ilusão de que velocidade sozinha é suficiente; ilusão que, convenhamos, já era antiga no mundo do software, mas agora ganhou turbina.

O problema do **vibe coding** não está na IA: a IA não tem culpa de nada, ela só faz o que mandamos. O problema está na ausência de **processo**. Quando você constrói software por acumulação de prompts, sem definir o que o sistema precisa fazer, sem pensar em quem vai usá-lo, sem planejar como vai evoluir, você não está desenvolvendo software, está construindo às cegas.

Figura 1.1: O contraste entre **vibe coding** e **engenharia de software**: velocidade sem **processo** versus construção sustentável com método.

A história do software está lotada de sistemas que 'funcionavam' muito bem até o momento em que pararam de funcionar de formas memoráveis. O foguete Ariane 5 explodiu 37 segundos após o lançamento em 1996 por causa de um overflow de inteiro, código copiado do Ariane 4 sem revisar se ainda fazia sentido no novo contexto. Trezentos e setenta milhões de dólares viraram confete no céu da Guiana Francesa. O sistema HealthCare.gov, o maior portal de saúde dos EUA, colapsou no lançamento em 2013 por uma combinação de planejamento insuficiente de capacidade, erros de software, funcionalidades incompletas e práticas frágeis de validação antes da entrada em produção. Nenhum desses projetos falhou por falta de código. Falharam por falta de **processo**. O **vibe coding** em escala industrial é a versão moderna do mesmo problema, com menos foguete e mais startup.

Este livro parte de uma premissa simples: **engenharia de software** e **agente inteligente** não são opostos; são parceiros. A IA é uma ferramenta extraordinária quando usada dentro de um **processo** estruturado. Ela acelera a escrita de código, automatiza tarefas repetitivas, sugere soluções e ajuda a explorar alternativas com uma paciência que nenhum colega humano teria. Mas quem define o que construir, para quem, com quais restrições e com qual qualidade, isso continua sendo responsabilidade do engenheiro. A IA não vai perguntar se você pensou nos casos extremos. Você precisa perguntar por ela.

Este capítulo estabelece o ponto de partida: entender o que é o **vibe coding**, por que ele é insuficiente para software que precisa durar, e o que a **engenharia de software** oferece como alternativa. Nos capítulos seguintes, você vai aprender a usar agentes inteligentes com método, não como substitutos do pensamento, mas como amplificadores dele. A morte do **vibe coding** não é o fim da IA no desenvolvimento. É o começo do uso inteligente dela. E talvez o fim de algumas noites traumáticas de produção às três da manhã.

1.1 O QUE É VIBE CODING E POR QUE ELE É INSUFICIENTE

Imagine a cena: são duas da manhã, um desenvolvedor abre o **Claude Code** ou o Cursor, digita 'cria um sistema de login com autenticação por e-mail e senha' e, em questão de segundos, recebe um bloco de código razoavelmente funcional. Ele cola, roda, funciona. Cola mais um pouco, roda de novo, funciona de novo. Em 48 horas, tem um 'produto'. Em 72 horas, já postou no LinkedIn: 'Construí um SaaS completo em um fim de semana com IA! #buildinpublic #solofounder #vibecoding'. Cento e oitenta curtidas. Dezessete comentários de 'incrível demais!'. Esse é o **vibe coding** em ação; e, por enquanto, tudo parece ótimo.

O termo foi cunhado por Andrej Karpathy em fevereiro de 2025 para descrever essa nova forma de programar: você esquece que existe código, entra em 'flow', aceita tudo que a IA sugere e vai em frente. Sem ler, sem revisar, sem entender. É como surfar numa onda que você não sabe de onde veio, mas enquanto está em cima, a sensação é fantástica. O problema, obviamente, aparece quando a onda quebra e você percebe que estava surfando em cima de código que valida senha com `if password == password`.

O **vibe coding** opera sobre uma premissa implícita muito conveniente: que código que funciona é código correto. Essa premissa é falsa. Sempre foi. Mas quando você tem uma IA gerando centenas de linhas por minuto, a ilusão de progresso se torna tão rápida que fica difícil perceber quando você saiu da estrada. Um sistema que 'funciona' em ambiente local, com um único usuário, sem dados reais, sem carga e com o banco de dados rodando no seu notebook não é um sistema pronto para produção; é um protótipo bem vestido que vai entrar em colapso na primeira sexta-feira à noite com dez usuários simultâneos.

O **vibe coding** não é culpa de um desenvolvedor específico; é um problema sistêmico que acontece quando ferramentas de alta produtividade chegam às mãos de pessoas sem nenhuma estrutura de **processo**. É mais ou menos o que acontece quando você dá um bisturi de última geração para alguém que nunca estudou medicina. O instrumento é excelente. O resultado, nem tanto.

O maior sintoma do **vibe coding** é a ausência de **requisitos**. Quando você não sabe o que o sistema precisa fazer de verdade, quem vai usar, em que condições, com quais restrições, em qual escala, qualquer código que 'pareça' fazer algo já satisfaz. A IA vai preencher os buracos com suposições razoáveis, você vai aceitar porque está no flow, e seis meses depois vai descobrir que essas suposições eram tudo menos razoáveis no contexto real do negócio.

Considere um cenário perfeitamente plausível: uma startup gerou, por meio de prompts sucessivos ao ChatGPT, um sistema de gestão financeira para pequenas empresas. Em três semanas, o produto estava 'pronto'. Seis meses depois, descobriram que o cálculo

de impostos aplicava uma alíquota fixa, ignorando alegremente o regime tributário de cada empresa. Nenhum teste cobria isso porque nenhum requisito tinha documentado isso. E nenhum requisito tinha documentado isso porque ninguém tinha sentado para perguntar: 'o que, de fato, esse sistema precisa calcular?' O custo de corrigir? Três vezes o custo de construir. A Receita Federal não aceita 'mas a IA gerou assim' como justificativa.

A outra face do problema é a **manutenção**. Código gerado sem estrutura é código que ninguém, nem a própria IA que o gerou, consegue manter com segurança. Quando um sistema cresce por acumulação de prompts, sem arquitetura, sem separação de responsabilidades, sem padrões, ele se transforma numa caixa preta que responde de forma imprevisível a qualquer mudança. Mexe aqui, quebra ali. Conserta ali, quebra acolá. Esse fenômeno tem nome: débito técnico. E no **vibe coding**, ele se acumula com a eficiência de uma dívida no cartão de crédito com juros de 12% ao mês.

Vale dizer, antes que alguém fique na defensiva: o **vibe coding** tem seu lugar. Para explorar uma ideia rapidamente, validar uma hipótese, construir um protótipo descartável ou aprender uma tecnologia nova, ele é genuinamente poderoso. O erro não está em usar IA para gerar código; está em confundir o protótipo com o produto, o rascunho com a entrega, o esboço com a planta baixa. Usar martelo para apertar parafuso não é culpa do martelo.

O que falta no **vibe coding**, portanto, é exatamente o que a **engenharia de software** oferece: **processo**. Um conjunto de práticas, técnicas e ferramentas que transformam a criação de software numa atividade controlada, previsível e sustentável. Não porque **processo** é burocracia corporativa inventada para torturar desenvolvedores; é porque software que importa vive por anos, é mantido por equipes inteiras e carrega responsabilidades reais sobre dados reais de pessoas reais. E para isso, vibe, infelizmente, não é suficiente.

1.2 O QUE É ENGENHARIA DE SOFTWARE

Em 1968, a OTAN reuniu um grupo de cientistas em Garmisch, na Alemanha, para discutir um problema que já tinha nome: a 'crise do software'. Sistemas atrasando, estourando orçamento, entregando menos do que prometiam ou falhando em produção de formas criativas. O mais divertido é que isso foi em 1968, sem IA, sem internet, sem GitHub. Os desenvolvedores da época conseguiam criar caos de altíssimo nível mesmo sem nenhuma ferramenta moderna para ajudar. Talento puro. Daquela conferência nasceu o termo **engenharia de software** ; e com ele, a ideia radical de que construir software é uma atividade de engenharia, não de inspiração divina.

A definição clássica do IEEE diz que **engenharia de software** é 'a aplicação de uma abordagem sistemática, disciplinada e quantificável ao desenvolvimento, operação e **ma-**

Figura 1.2: Evolução do **custo de mudança** ao longo do tempo. O **vibe coding** acumula **débito técnico** de forma exponencial: cada nova feature fica mais cara do que a anterior. O desenvolvimento com **processo** estruturado mantém o custo previsível e linear. Após o ponto de inversão, cada mudança no projeto sem processo custa mais do que custaria num projeto com engenharia desde o início.

nutenção de software'. Cada palavra importa, então preste atenção: sistemática significa que existe um método. Disciplinada significa que o método é seguido, não só quando dá vontade. Quantificável significa que é possível medir, acompanhar e melhorar. E o trecho 'desenvolvimento, operação e **manutenção**' deixa claro que o ciclo não termina na entrega; ele continua enquanto o software estiver vivo, que costuma ser muito mais tempo do que o time original esperava.

Perceba que **engenharia de software** não é sinônimo de 'escrever código bonito'. Um

excelente programador que nunca levantou **requisitos**, nunca documentou uma decisão de arquitetura e nunca escreveu um teste automatizado está praticando artesanato de software, que pode ser admirável, mas não é engenharia. O código pode ser elegante como um poema. Mas se não foi construído dentro de um **processo**, ninguém além do próprio autor consegue evoluí-lo com segurança. E o autor, seis meses depois e três projetos mais tarde, também não lembra mais o que estava pensando.

O corpo de conhecimento da área está organizado no SWEBOK, Software Engineering Body of Knowledge, um documento mantido pelo IEEE que mapeia as competências essenciais do engenheiro de software. Em versões recentes, ele organiza a disciplina em múltiplas áreas de conhecimento, como **requisitos**, design, construção, testes, **manutenção**, gestão de configuração, gestão de projetos, **processo**, modelos e métodos, qualidade e práticas profissionais. Repare que 'escrever código' aparece na área de construção, que é apenas uma delas. O código é importante, mas representa uma fatia menor do que a maioria dos iniciantes imagina quando pensa em fazer software de verdade.

Os modelos de **processo** são as formas concretas de organizar essas atividades no tempo. O modelo cascata organiza tudo em fases sequenciais: **requisitos**, design, construção, testes, entrega: simples, previsível e com a flexibilidade de um bloco de concreto. O modelo espiral adiciona iterações e gestão de risco. O RAD prioriza protótipos rápidos para validação. Os métodos ágeis, Scrum, XP, Kanban, organizam o trabalho em ciclos curtos com entrega contínua de valor e reuniões diárias que muita gente ama odiar. Cada modelo tem seu contexto ideal. Nenhum é bala de prata. Mas todos compartilham uma característica essencial: existe um **processo**, e ele é seguido de forma consciente.

A **manutenção** merece atenção especial porque é sistematicamente subestimada por quem está começando. Estudos clássicos de manutenção de software frequentemente estimam que a maior parte do custo do ciclo de vida ocorre após a entrega inicial, em correções, adaptações e evoluções. Traduzindo: a maior parte do trabalho de um engenheiro de software não é construir sistemas novos, é manter sistemas existentes que outra pessoa, ou você mesmo, em outro estado de espírito, construiu. Quando esse sistema foi construído sem método, a **manutenção** se torna cara, lenta, perigosa e emocionalmente desgastante. Quando foi construído com **processo**, ela é apenas trabalho normal.

O que a **engenharia de software** oferece, no fundo, é previsibilidade; e previsibilidade tem valor imenso em contextos onde falhar custa caro. A capacidade de dizer com alguma confiança: 'esse sistema faz o que precisa fazer, pode ser evoluído por uma equipe, resiste a falhas dentro de limites conhecidos e pode ser mantido por anos sem ninguém entrar em desespero'. Não é glamouroso. Não rende post viral no LinkedIn às duas da manhã. Mas é exatamente o que diferencia software que dura de software que vira uma história de terror contada em retrospectivas de sprint.

1.3 A PROPOSTA DO LIVRO: IA + PROCESSO = SOFTWARE DE VERDADE

Se você chegou até aqui, o diagnóstico está claro. O **vibe coding** é sedutor, rápido e insuficiente. A **engenharia de software** é robusta, comprovada e, sejamos honestos, às vezes percebida como burocrática demais para o ritmo que o mercado exige hoje. É como se alguém te oferecesse dois carros: um Fórmula 1 sem freio e um tanque de guerra sem acelerador. A pergunta que este livro responde é: e se não precisarmos escolher? E se for possível ter velocidade e controle ao mesmo tempo?

A resposta é sim; e a chave está nos **agente inteligente**s. Um **agente inteligente** no contexto de desenvolvimento de software não é apenas um autocomplete glorificado nem um estagiário digital que nunca dorme. É um componente autônomo que percebe o ambiente, toma decisões e executa ações para atingir objetivos. Quando esse agente opera dentro de um **processo** de engenharia bem definido, ele amplifica cada etapa: levanta **requisitos** com mais rapidez, gera código com padrões consistentes, escreve testes, sugere **refatoração** e documenta decisões. Ele não substitui o engenheiro; ele multiplica a capacidade do engenheiro. A diferença é importante.

Ferramentas recentes de desenvolvimento agentic, como Claude Code e Google Antigravity, apontam para uma nova geração de ambientes de desenvolvimento com IA integrada de verdade. Não são apenas editores com sugestão de código; são ambientes onde agentes podem executar tarefas complexas, operar em **worktree**s isoladas, gerenciar **branch**es, rodar testes e interagir com o sistema de arquivos de forma estruturada. Frameworks como o BMAD e o Speckit organizam o trabalho desses agentes em fluxos de desenvolvimento que respeitam as etapas clássicas da engenharia, dos **requisitos** ao **deploy**, sem abrir mão da agilidade que as ferramentas modernas permitem.

O conceito de **Agent Harness** é central nessa proposta. Um harness é um conjunto de configurações, **skill**s e **hook**s que define como os agentes operam num projeto específico. É a camada que transforma um agente genérico num colaborador especializado no seu domínio, nas suas convenções e no seu **processo**. Pense como a diferença entre contratar um freelancer que nunca ouviu falar da sua empresa e ter um desenvolvedor sênior que conhece cada detalhe do sistema. Ao longo deste livro, você vai construir harnesses para diferentes contextos e aprender a orquestrar múltiplos agentes trabalhando em paralelo sem perder o controle, nem o juízo.

Este livro está organizado em torno das grandes etapas da **engenharia de software**, com agentes inteligentes como protagonistas de cada uma delas. Começamos aqui, no capítulo 1, com o diagnóstico e a proposta. Nos capítulos seguintes, percorremos o ciclo completo: modelagem de **requisitos** com agentes, design com **design pattern**s aplicados em Python,

construção orientada por **TDD** e **SDD**, **refatoração** assistida por IA, e estratégias de **deploy** em ambientes modernos de nuvem. Cada capítulo combina teoria sólida com prática real: código que você pode executar, adaptar e colocar em produção sem precisar rezar antes de cada **deploy**.

A proposta deste livro não é que você pare de usar IA; isso seria absurdo e contraproducente. É que você pare de usar IA sem método. É que você faça a transição do **vibe coding** para o **AI Engineering**: o desenvolvimento de software onde agentes inteligentes operam dentro de um **processo** de engenharia estruturado, com **requisitos** claros, arquitetura definida, testes automatizados e entrega controlada. Não é menos IA; é mais engenharia. O desenvolvedor que domina essa combinação não está competindo com quem faz **vibe coding**; está numa categoria diferente. Enquanto o vibe coder está reconstruindo pela terceira vez um sistema que nunca foi especificado direito e respondendo tickets de bug às onze da noite, o AI Engineer já entregou, testou, documentou e está evoluindo a próxima versão. Essa é a diferença entre velocidade e sustentabilidade. E esse é o profissional que este livro vai ajudar você a se tornar.

CAPÍTULO 2

O que é software afinal?

Imagine que você pediu uma pizza. Não chegou pizza, chegou farinha, molho de tomate, queijo e pepperoni numa caixa separada. Tecnicamente, são todos os ingredientes certos. Mas não é pizza. Agora pense no código-fonte de um sistema: são linhas de instrução, variáveis, funções, classes. Tecnicamente, são os ingredientes certos. Mas também não é software, ainda. Software é o conjunto completo: o código que roda, a lógica que resolve um problema real, a interface que alguém consegue usar, os dados que persistem, as falhas que são tratadas, a documentação que permite a qualquer pessoa entender e manter o sistema e o **processo** de evolução que garante que tudo isso ainda funcione daqui a três anos. Só a massa não faz pizza. Só o código não faz software.

Essa distinção parece simples, mas tem consequências enormes. Como vimos no capítulo anterior, o **vibe coding** falha não por falta de código, mas porque produz código sem o restante do conjunto. Sem **requisitos** definidos, sem estrutura de **manutenção**, sem testes que garantam que o comportamento esperado permanece o esperado. É a diferença entre ter ingredientes e ter um prato. E da mesma forma que um chef profissional não mistura

ingredientes aleatoriamente esperando que 'saia algo bom', um engenheiro não empilha código esperando que 'funcione em produção'.

A confusão começa na própria definição. A definição técnica clássica, 'instruções que, quando executadas, produzem a função ou características desejadas', é correta, incompleta e, convenhamos, pouco útil para quem precisa tomar decisões reais de projeto. Porque ignora que software existe num contexto, atende pessoas, opera dentro de organizações, enfrenta limitações de infraestrutura e evolui ao longo do tempo. Um sistema que resolve o problema errado com perfeição técnica não é bom software, é engenharia de precisão aplicada na direção errada. Isso acontece com uma frequência que deveria assustar mais do que assusta.

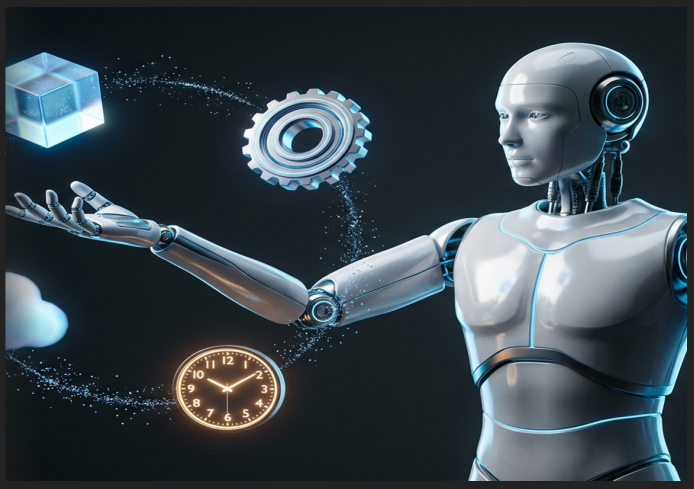


Figura 2.1: Software como artefato multidimensional: mais do que código, um sistema de software é produto, **processo**, serviço e compromisso com o tempo. Cada dimensão impõe restrições e responsabilidades distintas ao engenheiro.

O software também carrega algo que o hardware não tem: invisibilidade. Um prédio mal construído mostra as rachaduras. Um carro com defeito faz barulho. Software mal construído parece idêntico ao bem construído, até o momento em que não parece mais. Esse é um dos seus atributos mais traiçoeiros. O **débito técnico** acumulado durante meses de **vibe coding** não aparece à primeira vista no código; aparece na décima-quinta vez que o time tenta adicionar uma feature simples e descobre que o sistema inteiro resiste como se tivesse opiniões próprias sobre o que não quer fazer. E nesse momento, a pizza virou uma bagunça de ingredientes que ninguém sabe como reorganizar.

Compreender o que é software também significa entender que a crise que o acompanha desde o início não foi resolvida, apenas mudou de forma. Nos anos 70, o problema era custo e prazo. Nos anos 90, era complexidade crescente. Nos anos 2000, eram sistemas legados intratáveis herdados de gerações anteriores de decisões mal tomadas. Na década de 2020, é a ilusão de que a IA resolve o problema fundamental sem que o engenheiro precise entendê-lo. Cada geração de ferramentas resolve parte da equação e abre novos vetores de falha. Saber

onde estamos nessa linha do tempo é indispensável para saber o que fazer, e o que não fazer mais.

Figura 2.2: A crise do software nunca foi resolvida, apenas mudou de forma. Cada década trouxe novas ferramentas que resolveram parte da equação e abriram novos vetores de falha.

O desenvolvimento de software também é uma das áreas com maior concentração de crenças não verificadas tratadas como dogmas. 'Mais pessoas resolvem atrasos', 'o cliente sabe o que quer', 'documentação é perda de tempo', 'ágil significa sem planejamento'. Esses **mitos** sobrevivem geração após geração porque são convenientes, simplificam decisões difíceis e só cobram seu preço muito depois do momento em que foram adotados. Desmontá-los com dados e argumentos não é exercício acadêmico; é defesa profissional.

Este capítulo responde à pergunta do título com a seriedade que ela merece. Começa

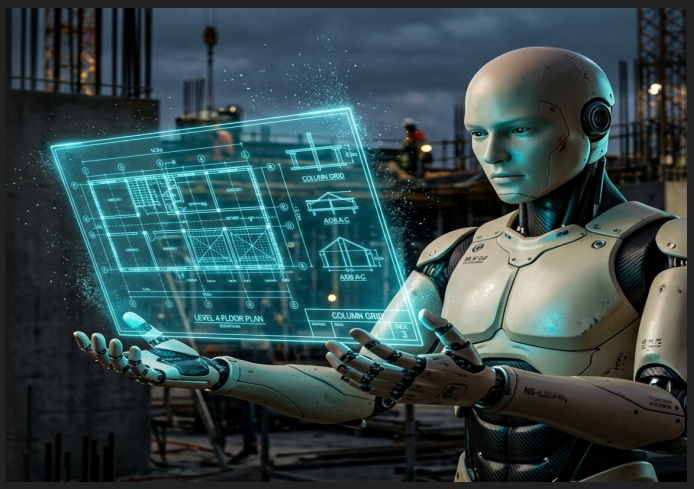
revisitando a crise do software, não como curiosidade histórica, mas como diagnóstico atual, e analisa como o **vibe coding** criou uma nova dimensão dessa crise. Em seguida, examina como software é realmente construído: os modelos de **processo** que a engenharia consolidou ao longo de décadas, a **revolução ágil** que redefiniu o ritmo do desenvolvimento, e os **mitos** que persistem apesar de tudo. Ao final, você não vai apenas saber dizer o que é software. Vai entender por que essa definição importa cada vez que você decide como, quando e com quem construir.

2.1 O SOFTWARE EM CRISE?

Quem estudou engenharia civil sabe que nenhum pedreiro sério levanta uma parede antes de ter em mãos a planta baixa, o projeto hidráulico e o projeto elétrico da edificação. Não é burocracia, é necessidade técnica: a posição de uma tubulação determina onde você pode ou não abrir um vão, a passagem de eletrodutos define o que vai dentro da laje, e a estrutura de cargas decide se aquela parede que o cliente quer remover pode ser removida ou vai derrubar o andar de cima junto. Mudar essas decisões depois que os tijolos já estão assentados não é difícil, é extraordinariamente caro, e às vezes impossível sem demolir o que foi feito. O software funciona exatamente da mesma forma. Sair escrevendo código sem ter os **requisitos** levantados, a arquitetura definida e as integrações mapeadas é o equivalente a assentar tijolos sem planta baixa: a obra avança, parece produtiva, e só revela os problemas quando já há muita coisa difícil de desfazer. A diferença é que na construção civil o erro aparece na vistoria. No software, aparece em produção, às três da manhã, na sexta-feira antes de um feriado.

Primeiro a planta
Depois o tijolo

Código sem projeto é parede sem planta baixa



Se você perguntar a qualquer gerente de projeto de software se já entregou algo fora do prazo, acima do orçamento ou com menos funcionalidades do que o prometido, a resposta vai

vir acompanhada de um sorriso maroto. Não porque é engraçado, mas porque é tão comum que virou quase um ritual de passagem. O atraso não é exceção, é o padrão. E o mais curioso é que isso não é novidade de agora: esse padrão tem nome, tem data de nascimento e resistiu a seis décadas de ferramentas supostamente capazes de resolvê-lo de uma vez por todas.

Em outubro de 1968, a OTAN reuniu em Garmisch, na Alemanha, um grupo de cientistas e engenheiros para discutir um problema que já acumulava vítimas: sistemas de software atrasando, estourando orçamento, entregando muito menos do que prometiam ou falhando de formas criativas em produção. Daquela conferência surgiu o termo **crise do software**. Era a primeira vez que a comunidade técnica reconhecia formalmente que construir software era um problema sério, sistemático e que não se resolvia apenas com programadores mais talentosos ou computadores mais rápidos. Precisava de método. Precisava de **engenharia de software**.

Figura 2.3: A conferência da OTAN em Garmisch, Alemanha, 1968: o momento em que a comunidade técnica reconheceu formalmente que construir software exigia método, não apenas talento individual.

O que caracterizava a crise de 1968 é o mesmo que caracteriza projetos problemáticos hoje: estimativas irreais aceitas sem questionamento, **requisitos** coletados de forma superficial, ausência de testes sistemáticos, e a crença de que 'vai dar certo no final'. O foguete Ariane 5 explodiu 37 segundos após o lançamento em 1996 porque um bloco de código foi reaproveitado do Ariane 4 sem verificar se ainda fazia sentido no novo contexto. O Therac-25, um equipamento de radioterapia computadorizado, administrou doses letais de radiação em pacientes entre 1985 e 1987 porque os intertravamentos de hardware do modelo anterior foram substituídos por verificações de software que nunca foram validadas adequadamente para as condições reais de operação. O portal HealthCare.gov colapsou no lançamento em 2013 diante de uma fração da carga esperada, depois de anos de desenvolvimento sem testes de integração. Em 2015, no Rio de Janeiro, o aplicativo Waze direcionou o casal Regina e Francisco Murmura para uma rua homônima dentro da favela do Caramujo, em Niterói, em vez da avenida turística que haviam solicitado. Cerca de vinte homens armados abriram fogo

contra o carro; Regina, de 70 anos, foi atingida nas costas e morreu no hospital. O algoritmo de roteamento não diferenciava contexto urbano de zona de risco; o **requisito** de segurança do usuário simplesmente não existia na especificação do sistema. Nenhum desses projetos falhou por falta de código. Falharam por falta de **processo**.

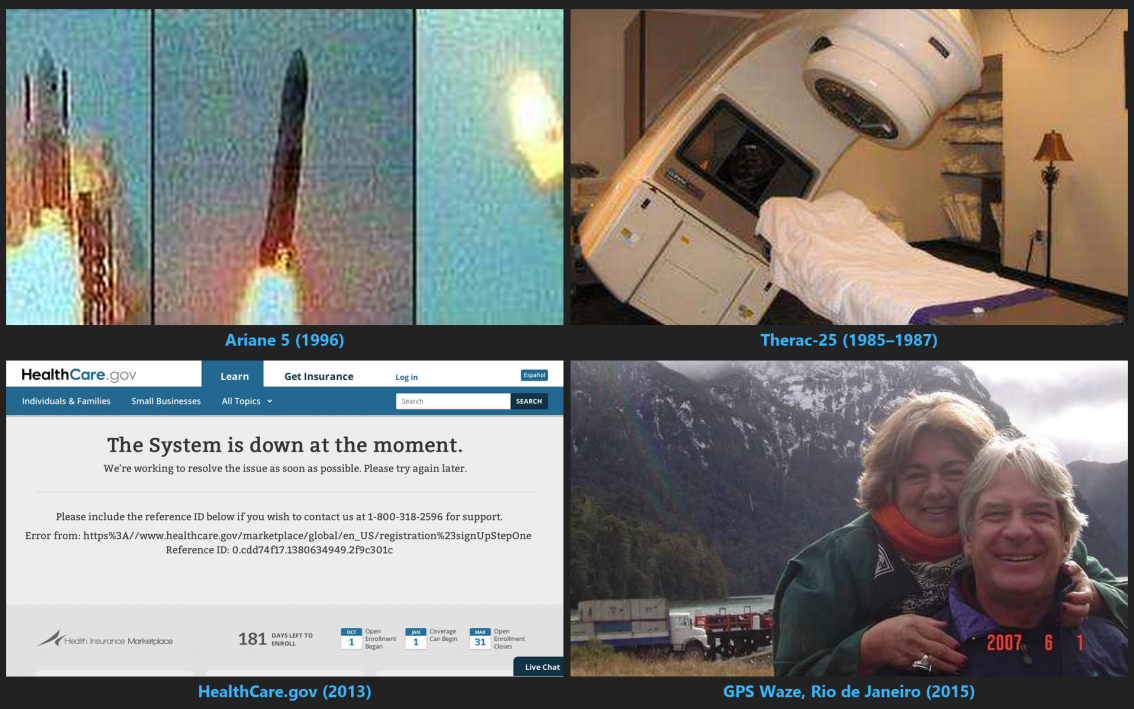


Figura 2.4: Quatro décadas de falhas causadas por ausência de **processo**: a explosão do Ariane 5 (1996), as doses letais do Therac-25 (1985-1987), o colapso do HealthCare.gov no lançamento (2013) e a morte de Regina Murmura após o Waze direcioná-la para uma favela no Rio de Janeiro (2015).

Mas falhas técnicas são apenas uma face da crise. A outra é mais sutil e mais comum: a distorção no entendimento do **requisito** entre quem pede e quem constrói.

Perceba o padrão: em todos esses casos, o software 'funcionava' em algum ambiente controlado antes de ir a público. O Therac-25 passava nos testes do fabricante. O Ariane 5 processava os dados corretamente dentro dos limites projetados para o Ariane 4. O HealthCare.gov respondia bem com poucos usuários simultâneos. O problema não era o código em si, era a ausência de um **processo** que garantisse que o código funcionaria nas condições reais de uso. Software que só funciona no ambiente do desenvolvedor é um protótipo bem disfarçado, não um produto.

A crise do software, portanto, nunca foi encerrada. Ela foi transformada. O CHAOS Report, série de estudos do Standish Group em circulação desde a década de 1990, acompanha o desempenho de projetos de software. Em uma edição de 2020, 31% dos projetos foram

classificados como bem-sucedidos[1], ou seja, entregues no prazo, dentro do orçamento e com o escopo completo. Outros 50% foram classificados como 'desafiados': entregues, mas com atrasos, custos extras ou funcionalidades reduzidas. E 19% foram simplesmente cancelados antes de chegar ao fim. Esses números são notavelmente parecidos com os de 1994. Três décadas de avanço tecnológico e a taxa de sucesso quase não se moveu.

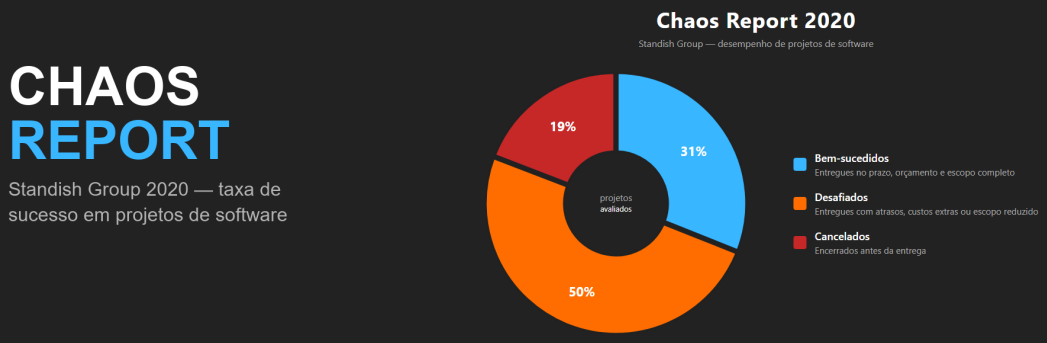


Figura 2.5: Chaos Report 2020 (Standish Group): apenas 31% dos projetos de software foram considerados bem-sucedidos, enquanto 50% foram entregues com restrições e 19% foram cancelados antes de chegar ao fim.

O que mudou foi a escala e a velocidade. Nos anos 70, um projeto de software mal conduzido atrasava meses e custava mais do que o previsto. Hoje, um sistema crítico mal construído pode vazar os dados de milhões de usuários antes do café da manhã, derrubar uma infraestrutura inteira com um deploy mal testado ou acumular **débito técnico** tão rapidamente que em seis meses ninguém mais consegue adicionar uma feature sem quebrar três outras. A crise não acabou: ela ganhou velocidade de processamento.

Entender a crise do software não é exercício de nostalgia técnica. É entender por que o **processo** existe e por que cada etapa que ignoramos tem um custo real associado, mesmo que esse custo apareça muito depois da decisão de ignorá-la. O desenvolvedor que conhece essa história não repete os mesmos erros por ingenuidade. E o engenheiro que opera **agentes inteligentes** dentro de um **processo** estruturado está, fundamentalmente, respondendo à mesma pergunta que os cientistas de Garmisch tentavam responder em 1968: como construir software que funciona de verdade, no prazo, com qualidade, e que ainda pode ser mantido depois que o entusiasmo do lançamento passar.

1 Standish Group. *CHAOS Report 2020*. O critério de sucesso considera entrega no prazo, dentro do orçamento e com escopo completo conforme definido no início do projeto.

Figura 2.6: A clássica falha de comunicação em projetos de software: cada envolvido entende o **requisito** de forma diferente, e o resultado final raramente corresponde ao que o cliente precisava de verdade.

2.2 A NOVA CRISE: SOFTWARE FEITO COM IA SEM PROCESSO

Em 1968, a crise do software tinha cara conhecida: equipes pequenas, estimativas otimistas e sistemas que cresciam mais rápido do que a capacidade de entendê-los. Era uma crise de escala e método, e a resposta foi construir método. Cinquenta e seis anos depois, chegamos a um momento em que qualquer pessoa com um navegador aberto pode pedir a um **agente inteligente** que escreva um sistema de autenticação completo em trinta segundos. E muita gente acha que isso resolve o problema. Na verdade, só troca a crise de camisa.

O **vibe coding** não é um fenômeno de programadores preguiçosos ou incompetentes. É uma resposta natural a uma ferramenta genuinamente poderosa. Quando você vê um **agente inteligente** gerar em minutos o que levaria dias para escrever, a reação instintiva é: por que não usar isso para tudo, o tempo todo, sem parar para pensar? E aí está o problema. O agente escreve código rápido. Ele não conhece o seu contexto, não entende as regras de negócio não ditas, não sabe que aquela tabela do banco de dados tem uma coluna com comportamento herdado de uma decisão tomada em 2017 que ninguém documentou. Ele escreve o que você pediu. E você, muitas vezes, nem sabe exatamente o que precisava pedir.

Pense numa situação que se tornou comum em times de desenvolvimento: um desenvolvedor junior pede ao **agente inteligente** que implemente um endpoint de upload de arquivos. O agente entrega o código em segundos, ele roda localmente, o teste manual passa, o **commit** vai para produção. Três semanas depois, alguém descobre que o endpoint aceita qualquer tipo de arquivo sem validação, que não há limite de tamanho, que os arquivos são armazenados com o nome original enviado pelo cliente e que, com a combinação certa de caracteres, é