LEGACY MODERNIZATION FRAMEWORK · PLAYBOOK

Modernize um parque COBOL para Azure, módulo a módulo, com agentes que você governa.

O core bancário, os tributos e os programas sociais do Brasil rodam sobre COBOL mais antigo do que as pessoas pagas para mantê-lo, e a força de trabalho que o entende sai 15% ao ano. Este playbook é o manual de operação para modernizar um parque mainframe com IA agêntica: uma estratégia de portfólio, uma arquitetura de referência governada, um roteiro de sete fases do assess ao cutover, e um modelo de fábrica que transforma o primeiro módulo entregue em vazão previsível, tudo no GitHub Copilot com o model picker, os AI Credits e o agent mode.

AUTORA

Paula Silva

CARGO

AI-Native Software Engineer

CONTATO

in/paulanunes

TEMPO DE LEITURA

~ 45 minutos, fim a fim

7

FASES DE MODERNIZAÇÃO

6

CAPÍTULOS

6

DECISÕES QUE FICAM COM HUMANOS

70-80%

TRABALHO EXECUTADO POR AGENTES

CAPÍTULOS

Chapter 00

Ficar parado é o maior risco

O parque COBOL brasileiro, uma força de trabalho que encolhe 15% ao ano, três forças regulatórias com uma janela de 24 a 36 meses, e como a IA agêntica inverte a curva de custo dominada pelo entendimento.

Imperativo →

Chapter 01

Nenhuma estratégia única serve ao parque inteiro

O espectro 5R do Gartner lido como portfólio, as quatro abordagens da Microsoft no Azure, a armadilha do JOBOL, e pontuar prontidão para defender Extend sobre Defend.

Estratégia →

Chapter 02

Cinco camadas, modelos roteados, agentes governados

A arquitetura de referência em cinco camadas, o roteamento de modelos no picker do Copilot, o MCP e o copilot-instructions.md como plano de controle, e seis agentes customizados sob três padrões de orquestração.

Arquitetura →

Chapter 03

Avaliar, fatiar, traduzir, provar desempenho

As quatro fases operacionais que levam um módulo COBOL do desconhecido à produção, com os números que separam um módulo entregue de um travado.

Roteiro →

Chapter 04

Dois mundos rodando até o gate de corte

O strangler fig com CDC e migração gradual de tráfego, shadow traffic e equivalência comportamental, as cinco condições de corte, e governança atravessando as três.

Coexistência →

Chapter 05

Do primeiro módulo a uma fábrica

Os seis modos de falha e a mitigação de cada um, o modelo de fábrica de regras de desmembramento reutilizáveis, e um começo de duas semanas no GitHub Copilot que um comitê diretivo pode financiar.

Fábrica →

DICAS DE LEITURA

> **ATALHOS DE TECLADO**
>
> Aperte `/` para buscar. Use `j` e `k` para mover entre capítulos. Aperte `t` para alternar tema. Aperte `g` para ir ao topo.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

00 O IMPERATIVO

Ficar parado é o maior risco.

O core bancário, a arrecadação tributária e os programas sociais do Brasil rodam sobre COBOL mais antigo do que a maioria das pessoas pagas hoje para mantê-lo. O código não está falhando. A força de trabalho que o entende está saindo a uma taxa mensurável, e os reguladores pararam de esperar. Este capítulo mede o custo de ficar parado e mostra por que a IA agêntica é a primeira coisa que torna uma janela de modernização de 24 a 36 meses realista, em vez de apenas desejável.

O Brasil roda em COBOL, e quem o entende está saindo

O parque COBOL brasileiro contém entre 1,25 bilhão e 2,1 bilhões de linhas de código. Ele está dentro de bancos comerciais, bancos de desenvolvimento, seguradoras, órgãos governamentais e varejo. Esses sistemas não estão quebrados. Eles processam pagamentos, arrecadam impostos e operam serviços sociais em escala nacional, e fazem isso de forma confiável há décadas. É justamente essa confiabilidade que torna perigoso adiá-los. Um sistema que funciona é fácil de deixar como está, e deixá-lo como está é a decisão que silenciosamente se acumula até virar a maior exposição do balanço.

O problema não é o código. São as pessoas. O Brasil perde cerca de 15% dos seus desenvolvedores COBOL por ano, e não há pipeline de reposição atrás deles. As universidades pararam de ensinar a linguagem há uma geração. Os engenheiros que escreveram as regras de negócio originais estão se aposentando, e levam consigo conhecimento não documentado. Cada ano de atraso reduz o grupo de pessoas capazes de explicar o que um determinado programa realmente faz. É por isso que trato a evasão da força de trabalho, e não a dívida técnica, como o risco de primeira ordem. Você pode refatorar código ruim. Você não pode entrevistar um engenheiro que saiu há três anos.

Três forças regulatórias, e uma janela de 24 a 36 meses

Três forças regulatórias convergiram, e cada uma empurra na mesma direção. A primeira é o mandato de modernização do BACEN para 2028, que exige interfaces de open banking baseadas em API e controles de cibersegurança que plataformas mainframe não entregam nativamente. A segunda é a LGPD, cujos requisitos de governança de dados (consentimento, linhagem e direito ao apagamento) nunca foram projetados em arquiteturas construídas nos anos 1980. A terceira é a pilha de compliance que não para de subir por baixo de tudo: SOX, ISAE 3402 e PCI-DSS. Nenhuma delas é opcional, e nenhuma espera por um ciclo orçamentário conveniente.

Contra esses prazos está a economia do entendimento. A engenharia reversa das regras de negócio consumiu historicamente 40% a 60% dos orçamentos de modernização. É por isso que a maioria dos projetos trava na fase de entendimento e nunca chega à tradução. As equipes ficam sem dinheiro antes de ficarem sem COBOL. Quando coloco os prazos regulatórios ao lado do custo histórico do entendimento, a janela útil de ação fica em 24 a 36 meses. Não é um horizonte confortável. É a razão pela qual a IA agêntica importa aqui, porque ela ataca exatamente a fase que costumava devorar o orçamento.

> DEFINIÇÃO
>
> A janela de ação é de 24 a 36 meses. Ela é limitada de um lado pelo mandato do BACEN para 2028 e pela pilha de compliance por baixo dele, e do outro por uma força de trabalho COBOL que encolhe 15% ao ano. Ficar parado não mantém a posição. Gasta o horizonte.

Como a IA agêntica inverte a curva de custo dominada pelo entendimento

A razão pela qual a modernização de legados tem um histórico tão ruim é para onde o dinheiro ia. Na distribuição tradicional, só o entendimento consumia cerca de metade do orçamento: 50% entendimento, 20% tradução, 15% testes, 15% corte.

Projetos que gastam metade do dinheiro antes de escrever uma linha de código novo tendem a não terminar. A IA agêntica inverte essa curva. A distribuição orientada por IA fica mais perto de 10% descoberta, 20% tradução, 35% testes de equivalência comportamental e 35% coexistência e governança. O orçamento sai do entendimento e entra na validação, que é a fase que de fato decide se o novo sistema é seguro para confiar.

O GitHub Copilot em agent mode é o que torna a descoberta barata. Acionado por um modelo de long context do picker, ele mapeia dependências em milhares de arquivos, rastreia caminhos de execução e expõe acoplamentos que desenvolvedores experientes não percebem. Modelos na classe de 1 milhão de tokens seguram uma aplicação COBOL inteira em contexto de uma vez, copybooks, JCL e definições de dados juntos. O roteamento segue a forma do trabalho: o Haiku 4.5 varre e faz triagem em alto volume, o Sonnet 5 traduz módulo a módulo, o Opus 4.8 faz a análise arquitetural profunda. Tudo isso roda sob um arquivo .github/copilot-instructions.md que fixa as convenções, custom agents que assumem cada estágio do pipeline, e servidores MCP que dão aos agentes acesso vivo ao repositório, ao build e ao sistema de tickets. O custo é medido em GitHub AI Credits, onde um crédito equivale a um centavo de dólar, então um teto de créditos por módulo vira um controle orçamentário real, e não um chute.

As seis decisões que permanecem com humanos

A meta operacional é 70% a 80% do trabalho executado por agentes, com 20% a 30% de supervisão humana. Essa proporção só se sustenta se a parcela humana for gasta nas decisões com consequências irreversíveis, e não na revisão de sintaxe. Seis decisões permanecem com as pessoas, e não saem de lá. Aprovação final da arquitetura-alvo. Validação de que as regras de negócio foram preservadas, não apenas de que o código compilou. Ajuste de performance para padrões de carga específicos. Sign-off do corte em produção. Autorização de compliance regulatório. E escalação dos casos de borda que os agentes sinalizam, mas não conseguem resolver sozinhos.

Manter essas seis com humanos é o que separa modernização de tradução. O Gartner é direto sobre o modo de falha: as ferramentas atuais de GenAI são boas em

traduzir código e excluem explicitamente a arquitetura do seu escopo. Traduza COBOL direto para Java sem repensar o design e você obtém JOBOL, que é lógica COBOL vestida com sintaxe Java, e isso perpetua todos os problemas estruturais dos quais você tentava escapar. Os agentes fazem o volume. Os humanos são donos da arquitetura, da postura de compliance e do momento do corte. Essa divisão de trabalho é o método inteiro, e o resto deste playbook é como executá-lo.

> DEFINIÇÃO
>
> Seis decisões nunca saem dos humanos: aprovação final da arquitetura-alvo, validação de que as regras de negócio foram preservadas, ajuste de performance para padrões de carga, sign-off do corte em produção, autorização de compliance regulatório e escalação de casos de borda. Os agentes executam os outros 70% a 80%. A divisão é o método.

Referências

Fontes primárias para o risco de força de trabalho, as forças regulatórias e as abordagens de modernização sobre as quais este capítulo se apoia.

- Reuters, COBOL Cowboys Aim to Rescue Aging Government Systems, sobre o grupo cada vez menor de engenheiros COBOL e o conhecimento não documentado que se aposenta com eles.
- Banco Central do Brasil (BACEN), origem do mandato de modernização para 2028 e dos requisitos de interfaces de open banking.
- Gartner, 7 Options to Modernize Legacy Systems, o espectro 5R e o argumento pela modernização incremental em vez do corte big-bang.
- Microsoft Learn, Mainframe migration in the Cloud Adoption Framework, as abordagens de referência para rehost, refactor, rearchitect e replace.
- GitHub Copilot coding agent documentation, a plataforma de agent mode que torna a descoberta automatizada barata o suficiente para mudar a curva de custo.
- Grand View Research, Mainframe Modernization Market, contexto de mercado para a escala do parque legado agora sob pressão de modernização.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

01 ESTRATÉGIA DE PORTFÓLIO

Nenhuma estratégia única serve ao parque inteiro.

Nenhuma estratégia de modernização sobrevive ao contato com um parque mainframe inteiro. Um portfólio COBOL reúne aplicações que diferem em criticidade de negócio, acoplamento, complexidade de dados e vida útil restante. Tratar todas com um único padrão, seja ele rehost, refactor ou replace, garante desperdício de um lado e risco do outro. Este capítulo trata o parque como um portfólio. Ele mapeia cada aplicação para uma estratégia, lê as quatro abordagens da Microsoft como escolhas de ferramental no Azure, nomeia a armadilha que transforma tradução em falso progresso, e mostra como pontuar prontidão e defender a decisão de Extend sobre o padrão de Defend.

Os 5R do Gartner como portfólio, não como mandato

O framework 5R do Gartner define cinco níveis de transformação: Retain, Rehost, Refactor, Rearchitect e Replace. Cada um carrega seu próprio perfil de custo, risco e valor. O erro que as empresas cometem é escolher um nível e aplicá-lo ao parque inteiro. A abordagem de portfólio faz o oposto. Ela classifica cada aplicação em dois eixos, criticidade de negócio e complexidade técnica, e a posiciona no padrão que serve. O Gartner projeta que, até 2030, mais de 75% das empresas com mainframes IBM adotarão estratégias incrementais de baixo risco em vez de substituição big-bang.

Percorra o espectro de ponta a ponta. Retain significa nenhuma alteração de código: sistemas estáveis próximos ao fim de vida, baixo esforço, baixo valor. Rehost é lift and shift para a nuvem com o código mantido como está, um ganho rápido de infraestrutura. Refactor traduz a linguagem mantendo a arquitetura, e é o ponto ideal para aceleração com IA. Rearchitect redesenha os limites primeiro e depois traduz, mais esforço por mais valor. Replace aposenta o legado de vez e adota SaaS ou

reconstrói. O modelo TIME (Tolerate, Invest, Migrate, Eliminate) dá a lógica de disposição do portfólio que decide qual nível cada aplicação merece.

Onde o GitHub Copilot entra na classificação

A classificação só é viável se você conseguir ler o parque rápido. O GitHub Copilot em agent mode, acionando um modelo de long context do picker como o Opus 4.8, consegue conter uma aplicação COBOL inteira com seus copybooks, JCL e definições de dados de uma vez e produzir uma primeira pontuação 5R para cada componente. Codifique a régua de pontuação como um custom agent e um arquivo .github/copilot-instructions.md, para que todo repositório seja classificado da mesma forma, não conforme quem por acaso o abre. O arquiteto humano aprova o bucket. O agente faz a leitura e o primeiro rascunho da justificativa.

As quatro abordagens da Microsoft mapeadas no ferramental Azure

O framework da Microsoft se alinha ao modelo 5R, mas nomeia o Azure como plataforma-alvo. Ele oferece quatro abordagens. Migrate é rehost, entregue com Azure Migrate e emulação de mainframe. Modernize é refactor, entregue com o Mainframe Modernization Agent e o Azure AI Foundry. Rearchitect usa Azure Functions, AKS, Service Bus, Event Grid e Cosmos DB. Replace usa Dynamics 365, Power Platform, soluções ISV e Microsoft Fabric. Ao lado dessas, o padrão Hybrid e Extend embrulha as transações do mainframe em APIs para que o legado siga rodando enquanto é modernizado de forma incremental.

O ferramental importa menos do que o roteamento entre ferramentas. Conectores MCP dão a um agente do Copilot acesso vivo às fontes de que ele precisa para fazer trabalho real: GitHub, Azure DevOps, Backstage, o inventário do mainframe e o Key Vault para segredos. O trabalho concreto sob cada abordagem é específico. A migração de dados move os repositórios VSAM e IMS para o Azure SQL. O agendamento de batch JCL cede lugar à orquestração moderna. As transações CICS viram REST APIs ou Azure Functions. Cada abordagem é um alvo diferente na mesma plataforma, não um fornecedor diferente para gerar lock-in.

A armadilha do JOBOL: por que traduzir não é modernizar

A pesquisa do Gartner (G00822567) confirma um limite que os fornecedores tendem a embaçar. As ferramentas atuais de GenAI são excelentes em tradução de código e refactoring de framework, mas excluem explicitamente a modernização arquitetural do seu escopo. Traduza COBOL para Java preservando a estrutura original do código e você obtém o que o Gartner chama de JOBOL: lógica COBOL implementada em Java. Ele compila. Ele passa numa primeira olhada. E perpetua cada problema estrutural do original em uma nova linguagem que a sua equipe ainda não consegue manter.

Com agentes, o JOBOL acontece por um motivo simples. Um arquivo .github/copilot-instructions.md que diz apenas converta este programa COBOL para Java entrega exatamente isso: uma transliteração linha a linha. A correção é rodar uma passagem de rearquitetura antes. A IA propõe os limites-alvo, e os humanos os aprovam. Só então a tradução mira esses limites, e não o fluxo de controle PERFORM THRU original. O Gartner também documenta que as ferramentas que prometiam decomposição automática de monólito em microsserviços recuaram bastante nessas promessas, então planeje análise e design manuais substanciais, independentemente do apoio da ferramenta. A acurácia de tradução em cenários típicos assistidos por LLM fica em torno de 80%, e a aceitação de sugestões de IA fica perto de 30% na prática, então a revisão humana não é opcional.

> DEFINIÇÃO
>
> JOBOL é lógica COBOL vestindo sintaxe Java. Ele compila, passa em um teste ingênuo e carrega cada defeito estrutural do original para uma linguagem que seus novos contratados leem, mas ainda não conseguem manter. Traduzir é um passo dentro da modernização. Não é a modernização.

Pontuar prontidão e defender Extend sobre Defend

Pontue cada aplicação em quatro dimensões antes de atribuir uma estratégia. A complexidade de código cobre total de linhas, complexidade ciclomática e profundidade da cadeia de chamadas. A complexidade de dados cobre VSAM, IMS, DB2, Adabas e integridade referencial. A complexidade de integração cobre o número de interfaces, o middleware em uso e padrões batch versus tempo real. A complexidade operacional cobre agendamento de batch, disaster recovery, SLAs e exposição a compliance. A Análise de Pontos de Função, seguindo o IFPUG, dá uma medida de tamanho independente de linguagem, para que parques COBOL e Natural sejam comparados em uma única escala. Como referência, a conversão automática de COBOL para Java roda cerca de 0,5 a 2 horas por ponto de função em código direto, subindo para 4 a 8 horas em lógica complexa que exige revisão manual.

A pontuação existe para responder a uma única pergunta econômica. O Gartner (G00823006) separa três modelos de negócio de GenAI: Defend, que mantém o legado, Extend, que agrega capacidade ao redor dele, e Upend, que troca o próprio modelo. O dado que deveria mudar orçamentos é duro. Estratégias de Defend mostram retorno financeiro de -52%, estratégias de Extend entregam +134%, e ainda assim 50% do investimento atual em GenAI segue mirando o Defend. A leitura é direta. O instinto padrão é proteger o que já roda, e esse instinto perde dinheiro.

Governar o gasto com GitHub AI Credits

O argumento de Extend só se sustenta se você controlar o custo de chegar lá. O uso de agentes do GitHub Copilot é cobrado em GitHub AI Credits, onde um crédito equivale a um centavo de dólar. Defina um teto de créditos por módulo e um gate de validação iterativo antes de cada nova wave, para que o custo seja limitado por unidade de trabalho em vez de aberto. Uma postura de Defend continua pagando licenciamento por MIPS e de 60% a 70% do orçamento de TI em manutenção, sem retorno que se acumule. Uma postura de Extend gasta créditos medidos em trabalho que produz um ativo moderno e sustentável. Oriente o portfólio para Extend e Modernize, reserve Defend para as aplicações genuinamente próximas do fim de vida, e deixe a pontuação de prontidão, não o instinto, decidir.

Referências

Fontes primárias para o espectro 5R, o mapeamento no Azure, o limite de escopo do JOBOL e a economia de Defend versus Extend.

- Gartner, 7 Options to Modernize Legacy Systems, o espectro de modernização que este capítulo lê como portfólio, e não como mandato.
- Microsoft Learn, Mainframe migration: Cloud Adoption Framework, as quatro abordagens e seus alvos de ferramental no Azure.
- Gartner, Predicts 2026: Reducing Costs and Risks in Mainframe and Midrange Modernization, a projeção de que a maioria das empresas escolhe modernização incremental em vez de big-bang até 2030.
- IFPUG, Function Point Counting Practices Manual, o método de medição independente de linguagem por trás da pontuação de prontidão em COBOL e Natural.
- GitHub Copilot, a plataforma de agentes de referência para classificação, tradução e execução com custo governado neste capítulo.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

02 ARQUITETURA DE REFERÊNCIA

Cinco camadas, modelos roteados, agentes governados.

Um pipeline de modernização vale tanto quanto a arquitetura que o executa. Este capítulo descreve a arquitetura de referência que uso para modernização agêntica de sistemas legados: cinco camadas, da interface do desenvolvedor ao perímetro, uma regra de roteamento que atribui cada tarefa ao modelo certo no picker do GitHub Copilot, um plano de controle formado por conectores MCP e pelo .github/copilot-instructions.md, e seis agentes customizados coordenados por três padrões de orquestração. Cada camada existe para manter a autonomia dentro de uma fronteira governada.

As cinco camadas, da interface do desenvolvedor ao perímetro

A arquitetura de referência tem cinco camadas, e cada uma se apoia na de baixo. A interface do desenvolvedor é onde os humanos conduzem o trabalho. Os engenheiros abrem o GitHub Copilot em agent mode dentro do VS Code ou no GitHub.com, e o repositório carrega um arquivo .github/copilot-instructions.md que informa a cada agente as convenções do projeto, a linguagem-alvo e as regras de modernização. Esse arquivo é a primeira coisa que um agente lê, então é ali que o controle começa.

A segunda camada é a orquestração multi-modelo, a inteligência do pipeline. Um modelo orquestrador decompõe a tarefa, modelos workers traduzem e testam, modelos scanners fazem triagem, e o Microsoft Semantic Kernel coordena os handoffs. A terceira camada são os conectores MCP, as ferramentas empresariais que os agentes alcançam pelo Model Context Protocol: GitHub, Azure DevOps, Backstage, banco de dados, observabilidade, Key Vault e o rastreador de work items. A quarta camada é a plataforma, onde a governança reside: o Azure AI Foundry hospeda os modelos, o Microsoft Entra ID cuida da identidade, o Key Vault

guarda os segredos e o Azure Monitor registra cada ação. A quinta camada é o perímetro, as interfaces externas: o mainframe do cliente, que ainda mantém a fonte da verdade, os serviços Azure de destino e as APIs externas. O princípio de design é que toda camada se conecta de volta à camada de plataforma, de modo que nenhuma ação de agente escapa da identidade, do log ou da política.

Roteamento de modelos no picker do GitHub Copilot

Nenhum modelo único lida bem com toda tarefa, e nenhum modelo único compensa seu custo em toda tarefa. O picker do GitHub Copilot permite escolher o modelo por tarefa, e o pipeline usa uma regra de roteamento simples: Haiku 4.5 varre, Sonnet 5 traduz, Opus 4.8 analisa. O Haiku 4.5 executa o trabalho de alto volume e baixa complexidade: validação de sintaxe, correspondência de padrões, classificação de triagem. O Sonnet 5 faz o trabalho de código equilibrado: tradução de COBOL para Java, geração de testes, documentação e criação de pull requests. O Opus 4.8 assume o raciocínio que precisa do quadro inteiro: análise de aplicação completa, extração de regras de negócio e recomendações arquiteturais, usando seu contexto de 1 milhão de tokens para conter uma aplicação COBOL inteira com copybooks, JCL e definições de dados de uma só vez.

Roteamento é uma decisão de custo tanto quanto de qualidade. Desde 1 de junho de 2026, o GitHub Copilot cobra o trabalho dos agentes em GitHub AI Credits, em que um crédito vale um centavo de dólar, e os modelos mais pesados consomem mais créditos por tarefa. Mandar uma verificação de sintaxe para o Opus 4.8 desperdiça créditos; mandar uma decisão de arquitetura para o Haiku 4.5 desperdiça a análise. O picker torna a escolha explícita e auditável. A mesma disciplina pode ser automatizada: o Model Router do Azure AI Foundry seleciona um modelo por requisição a partir das características da requisição, e o material de origem relata que ele reduz o gasto de tokens em até 40% ao enviar requisições simples para modelos menores sem intervenção do desenvolvedor. A regra permanece a mesma, quer um humano escolha o modelo, quer o router escolha: casar o modelo com a tarefa.

Conectores MCP e o copilot-instructions.md como plano de controle

Os agentes não tocam os sistemas empresariais diretamente. Eles os alcançam pelo Model Context Protocol, um padrão aberto que funciona como um conector universal para agentes de IA. Cada servidor MCP expõe um conjunto fixo de ferramentas, e o agente chama essas ferramentas em vez de escrever seu próprio código de integração. O servidor do GitHub lê arquivos-fonte e abre pull requests. O servidor de banco de dados inspeciona esquemas e amostra dados em modo somente leitura. O servidor do Key Vault recupera credenciais para que os segredos nunca cheguem ao contexto ou aos logs do agente. O servidor de observabilidade compara a performance entre o sistema legado e o moderno. Porque o MCP é um padrão aberto, servidores de fornecedores diferentes funcionam com qualquer agente compatível, o que mantém a arquitetura livre de lock-in.

A outra metade do plano de controle é o arquivo .github/copilot-instructions.md. É o conjunto persistente de instruções que todo agente lê antes de agir: os padrões de código, a arquitetura-alvo, a regra que proíbe injetar qualquer SDK proprietário e a restrição de que o código traduzido seja código-alvo padrão e idiomático, em vez de lógica COBOL reescrita em Java. Entre o MCP e o arquivo de instruções, a arquitetura separa o que um agente consegue fazer do que ele tem permissão para fazer. O MCP define as ações alcançáveis; o arquivo de instruções e os níveis de permissão da plataforma definem as permitidas. Essa separação é o que torna seguro conceder autonomia ao agente.

Seis agentes customizados e três padrões de orquestração

Seis agentes customizados operam o pipeline, cada um com escopo estreito e um contrato claro de entrada e saída. O agente de Discovery varre o codebase, começando pelo Haiku 4.5 e escalando para o Opus 4.8 na análise profunda, e produz um grafo de dependências e um catálogo de regras de negócio. O agente de Analysis, no Opus 4.8, pontua cada componente e recomenda uma estratégia de modernização. O agente de Translation, no Sonnet 5, converte um módulo por vez e abre um pull request com documentação de mapeamento. O agente de Test generation, também no Sonnet 5, escreve testes de equivalência comportamental e

mira ao menos 85% de cobertura, com 100% nos caminhos críticos de regras de negócio. O agente de Review confere o código traduzido contra os padrões. O agente de Migration orchestration, no Opus 4.8, coordena os demais e mantém o plano.

Três padrões de orquestração cobrem as formas de tarefa que surgem, e o Microsoft Semantic Kernel implementa os três nativamente. Handoffs sequenciais movem o trabalho estágio a estágio, um agente terminando antes de o próximo começar. O padrão de handoff deixa um worker escalar de agente para agente quando encontra uma decisão que não consegue resolver. O group chat coloca vários agentes em raciocínio arquitetural colaborativo, consolidado pelo orquestrador. O perigo contra o qual esses padrões protegem é a anarquia de agentes: agentes mal coordenados que colidem e entram em loop. A defesa é um orquestrador único, um gate de validação entre cada handoff e uma ação registrada para cada passo. A autonomia é concedida por nível e revogada no primeiro gate que um agente não consegue passar.

> DEFINIÇÃO
>
> A arquitetura em uma linha: cinco camadas, da interface do desenvolvedor ao perímetro, uma regra de roteamento (Haiku 4.5 varre, Sonnet 5 traduz, Opus 4.8 analisa), conectores MCP e .github/copilot-instructions.md como plano de controle, e seis agentes customizados sob três padrões de orquestração. Toda camada se conecta de volta à plataforma, então a autonomia sempre roda dentro de uma fronteira governada.

Referências

Fontes primárias para a arquitetura de referência, a regra de roteamento de modelos, o plano de controle MCP e os padrões de orquestração de agentes.

- Microsoft Cloud Adoption Framework, Mainframe migration, a referência do lado de destino para mapear componentes de mainframe em serviços Azure.
- GitHub, Pick your agent: use Claude and Codex on Agent HQ, o model picker e o agent mode que dirigem a camada de interface do desenvolvedor.

- Model Context Protocol Specification, o padrão aberto por trás da camada de conectores MCP.
- Microsoft Semantic Kernel, Agent Framework, o framework que implementa os padrões sequential, handoff e group chat.
- Azure AI Foundry, Model Router, seleção automática de modelo que a fonte relata reduzir o gasto de tokens em até 40%.
- Azure AI Foundry, Claude models, a camada de plataforma que hospeda os modelos roteados.
- GitHub Copilot coding agent, o runtime de agente customizado para tarefas autônomas de modernização.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

03 FATIAR E TRADUZIR

Avaliar, fatiar, traduzir, provar desempenho: o roteiro de execução em quatro fases.

Este capítulo transforma o framework em um roteiro de execução. Quatro fases levam um módulo legado do desconhecido à produção. Avaliar mapeia o que o código realmente faz. Fatiar o divide em unidades entregáveis. Traduzir o reescreve módulo a módulo. Desempenho prova que o resultado se sustenta sob carga. Cada fase roda no GitHub Copilot em agent mode, cada uma termina em um gate de decisão humana, e cada uma produz um artefato pronto para rollback. Aqui estão as quatro fases em ordem, com os números que separam um módulo entregue de um módulo travado.

Fase 1, Avaliar: o agente de descoberta e o grafo de dependências

O agente de descoberta é a primeira coisa que você coloca de pé. Ele roda no GitHub Copilot em agent mode, definido como um custom agent por meio do .github/copilot-instructions.md, e alcança o parque mainframe através de MCP servers: um servidor GitHub para o código-fonte, um servidor de banco de dados para os schemas, um servidor de observabilidade para dados de runtime. Ele ingere programas COBOL, copybooks, JCL, programas Natural, definições CICS e schemas DB2 ou Adabas, e então produz um catálogo de programas, um dicionário de dados, um fluxo de jobs JCL, um mapa de transações e um score de complexidade de 1 a 10 para cada programa. A varredura de alto volume é roteada para o Haiku 4.5 no model picker. O raciocínio profundo de dependências é roteado para o Opus 4.8, cujo contexto de 1 milhão de tokens comporta milhares de programas de uma vez, de modo que o agente rastreia acoplamentos que atravessam todo o parque em vez de um arquivo por vez.

A saída é um grafo de dependências, e o grafo é onde a avaliação prova seu valor. Ele separa os órfãos, programas com quase nenhuma dependência e os candidatos ideais para começar, dos programas hub, chamados por muitos outros e incapazes de se mover até que seus dependentes estejam prontos. A análise estática não é uniformemente confiável, então o agente marca cada dependência com um nível de dificuldade: chamadas CALL diretas e sequências JCL resolvem com 95 a 100 por cento de acurácia, interações CICS LINK e MQ com 80 a 90 por cento, chamadas geradas e dinâmicas com 50 a 70 por cento, e comportamento que só aparece em runtime com 35 a 40 por cento. Para o nível mais difícil, o agente complementa a análise estática com registros de runtime do SMF, que capturam a CALL dinâmica e o roteamento indireto que nenhuma passagem estática enxerga. O gate de decisão que fecha esta fase é um portfólio classificado nos baldes 5R, aprovado por um arquiteto humano.

Fase 2, Fatiar: fatiamento por níveis e o menor módulo entregável

Fatiar decide o que entra em produção primeiro. Um parque COBOL bancário típico carrega décadas de copybooks compartilhados, áreas de dados comuns, chamadas CICS LINK e arquivos VSAM tocados por muitos programas. Você não consegue extrair uma peça sem saber o que ela toca. O agente de análise, rodando no Opus 4.8, lê o grafo de dependências e encontra cut-points: fronteiras onde os cruzamentos são mínimos, os fluxos de dados são bem definidos e de baixo volume, e os programas de um lado formam uma função de negócio coerente que uma API pode encapsular. Programas dentro da fronteira viram um incremento de modernização. Programas fora seguem rodando no mainframe.

A ordem segue os níveis de dificuldade, não o organograma. Módulos folha de nível 1, com baixo acoplamento, são fatiados primeiro, porque constroem a capacidade do time sobre vitórias que você consegue de fato concluir. Módulos intermediários de nível 2 vêm depois. Módulos centrais de nível 3, com alto acoplamento, ficam por último, quando o time tem a maior experiência e os padrões mais reutilizáveis. O comportamento de runtime do SMF informa a ordem: um módulo que parece isolado na análise estática, mas mostra invocação dinâmica intensa em runtime, desce na fila.

O menor módulo entregável

A unidade que importa é o menor módulo que consegue ir a produção sozinho e provar o pipeline de ponta a ponta. Para o primeiro incremento, escolha um módulo COBOL com menos de 5.000 linhas, baixa criticidade de negócio, baixo acoplamento e um cut-point limpo. Não é o módulo de maior valor. É aquele que deixa você rodar descoberta, tradução e testes comportamentais até um gate de decisão assinado, sem uma dependência dura te bloqueando. Esse primeiro módulo entregue é o que garante o orçamento da próxima onda, e transforma o plano de sequenciamento de um slide em evidência.

Fase 3, Traduzir: módulo a módulo, semântica acima da sintaxe

A tradução é a etapa que todo mundo imagina quando ouve modernização, e é a que menos falha. O Sonnet 5, o worker de tradução no model picker, reescreve COBOL em Java, .NET ou Python idiomático um módulo por vez, nunca de uma só vez. Cada módulo produz um pull request com documentação de mapeamento que amarra cada unidade traduzida de volta ao seu parágrafo de origem, e cada pull request vai para um revisor humano. Esse é o gate de decisão desta fase: nenhum módulo faz merge sem uma pessoa lendo o diff.

O objetivo é a semântica de negócio, não a estrutura sintática. O modo de falha tem nome, JOBOL: lógica COBOL reimplementada em Java, que compila, passa em um teste raso e carrega para uma nova linguagem todos os problemas estruturais do original. O Gartner é explícito ao afirmar que as ferramentas generativas atuais traduzem código bem e excluem a modernização de arquitetura do seu escopo, então a proteção contra o JOBOL é uma passagem de rearquitetura antes da tradução. O agente propõe as fronteiras-alvo, e um humano as aprova. Os testes de equivalência comportamental são gerados a partir do comportamento do COBOL original, não do código traduzido, de modo que a suíte valida o que o sistema deve fazer, e não o que a tradução por acaso produziu.

Fase 4, Desempenho: baselines de SMF e nenhuma regressão além dos SLOs

Um módulo modernizado que está correto, mas mais lento que o mainframe, não vai a produção. Os stakeholders de negócio vão barrá-lo, e com razão. Por isso esta fase começa antes de qualquer tradução, com um baseline de desempenho construído a partir do próprio sistema legado. Os registros SMF, em particular os tipos 30, 70 a 79 e 110, entregam tempo de CPU por transação, espera de I/O e profundidade de fila. Esses dados estabelecem não só uma média, mas a distribuição completa: p50, p95 e p99. Um mainframe z16 processa mais de 25.000 transações CICS por segundo, e o baseline é o número que o alvo em Azure precisa igualar.

O baseline vira um orçamento de desempenho com um teto rígido por tipo de carga. Transações online no p95 podem rodar no máximo 10 por cento mais lentas que o baseline, no p99 no máximo 20 por cento, o tempo total de batch no máximo 15 por cento, e a vazão de batch e a latência de consulta de dados no máximo 10 por cento. Os testes de carga rodam com volume próximo ao de produção antes de qualquer tráfego ser roteado. O Application Insights com tracing distribuído compara o sistema moderno com o baseline de SMF em tempo real, e se o desempenho degradar além da barreira vermelha, o tráfego volta automaticamente para o sistema legado. Nenhuma regressão além dos SLOs definidos é aceita. O gate de decisão é a aprovação de desempenho pelo time de SRE.

DEFINIÇÃO

A régua de go/no-go de qualquer módulo, em uma linha: score de complexidade igual ou abaixo de 7, cobertura de testes igual ou acima de 85 por cento com 100 por cento nos caminhos de regras de negócio críticas, 100 por cento de equivalência comportamental contra o COBOL original, e desempenho dentro de 10 por cento do baseline de SMF. O custo permanece limitado por um teto por módulo em GitHub AI Credits, onde 1 credit equivale a um centavo de dólar. Falhe em qualquer um destes e o módulo não vai a produção.

Referências

Fontes primárias para as quatro fases: as ferramentas de descoberta e tradução, os dados de runtime do mainframe e os padrões de desempenho da Azure.

- GitHub Copilot coding agent, a referência para rodar os agentes de descoberta e tradução em agent mode.
- Model Context Protocol Specification, a interface padrão que os agentes usam para alcançar código-fonte, schemas e dados de runtime.
- IBM, System Management Facilities (SMF), os registros de runtime usados para dependências de nível difícil e para o baseline de desempenho.
- Azure Architecture, Strangler Fig Pattern, o padrão de fronteira por trás dos cut-points e da fachada de coexistência.
- Azure Architecture, Reengineer Mainframe Batch Apps, os padrões de desempenho de batch que o sistema modernizado precisa igualar.
- Azure Monitor, Application Insights, validação contínua de desempenho contra o baseline de SMF.
- Gartner, 7 Options to Modernize Legacy Systems, o espectro 5R e a ressalva de que traduzir código não é modernizar arquitetura.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

04 COEXISTIR E GOVERNAR

Dois mundos rodando até o gate de corte.

A parte difícil da modernização não é traduzir COBOL. É rodar o sistema antigo e o novo ao mesmo tempo, por meses, sem que os usuários percebam, e depois provar que o novo sistema é seguro o bastante para assumir o corte. Este capítulo cobre as três últimas fases da jornada. Coexist mantém dois mundos em sincronia. Test compara os dois sob carga real. Cutover move o tráfego de produção apenas quando cinco condições são satisfeitas. A governança atravessa as três, porque agentes que agem sem gates, tetos e logs são um passivo, não um acelerador.

Fase 5, Coexist: o strangler fig, o CDC e a migração gradual de tráfego

A Fase 5 roda o padrão strangler fig. Um roteador fica na frente dos dois sistemas e envia cada requisição para o mainframe ou para os serviços modernos no Azure. O mainframe permanece a fonte da verdade, com autoridade de leitura e escrita, até que um executivo responsável assine o gate de corte. Os serviços modernos rodam de forma idempotente ao lado dele. Três peças de infraestrutura fazem isso funcionar. O Azure API Management é a fachada por onde toda requisição passa. O Azure Front Door com roteamento por peso controla que fatia do tráfego cada sistema recebe, e os pesos mudam sem deploy. O Change Data Capture mantém os dois repositórios de dados consistentes.

A sincronização de dados é a parte que quebra projetos. A regra é simples e absoluta: nunca faça dual-write. Se os dois sistemas gravam nos seus próprios repositórios a partir da mesma requisição, os dados divergem em horas. Em vez disso, designe um sistema como fonte da verdade para cada domínio de dados e replique para o outro. O Change Data Capture lê o log de transações do mainframe e transmite as mudanças para o banco no Azure em tempo quase real, com latência

típica de 1 a 5 segundos. Confluent e tcVISION cuidam da parte específica de mainframe. Para fluxos baseados em transação, um padrão orientado a eventos no Azure Service Bus entrega semântica de exatamente uma vez. Para dados de referência que mudam raramente, uma sincronização agendada por API já basta.

A migração de tráfego é uma escada, não um salto

O tráfego avança em degraus, e cada degrau precisa merecer o próximo. O sistema moderno começa com 0% do tráfego real e roda primeiro em modo shadow, recebendo cópias espelhadas das requisições de produção sem impacto para o usuário. Depois o tráfego real começa em 1%. Ele expande para 5% somente depois que 1% ficou estável por pelo menos 48 horas, e segue por 10%, 25%, 50% e 100%. Cada degrau precisa atender aos critérios go/no-go antes que o próximo se abra. Quando o tráfego chega a 100%, o mainframe entra em standby e fica disponível por 30 a 90 dias para rollback de emergência. Cada degrau é reversível em até 15 minutos ao mudar as regras de roteamento no API Management, de modo que nenhum corte é uma porta de mão única.

Fase 6, Test: shadow traffic e equivalência comportamental

A Fase 6 é onde a confiança é conquistada. O shadow traffic é o método. Cada requisição de produção para o mainframe é copiada e enviada ao sistema moderno ao mesmo tempo. A resposta do mainframe vai para o usuário. A resposta do sistema moderno é capturada e comparada, nunca devolvida. Um motor de comparação registra três coisas para cada requisição: se as saídas coincidem, quanto mais lento ou mais rápido o sistema moderno está, e se algum dos lados produziu um erro que o outro não produziu. O shadow traffic precisa rodar por ao menos um ciclo de negócio completo, em geral um mês, para que os padrões diários, semanais e de fechamento de mês apareçam. A carga em shadow equivale à produção completa, então dimensione a infraestrutura no Azure para isso desde o início.

A equivalência comportamental é o teste de aceite. A suíte de equivalência é gerada a partir do comportamento COBOL original, não do código traduzido, porque testes escritos contra a tradução só confirmariam os próprios erros dela. A mesma entrada

roda pelos dois sistemas e as saídas são comparadas campo a campo em milhares de fixtures, incluindo casos de borda, fechamento de mês e ano bissexto. A meta é coincidência de 100%, com regras de tolerância explícitas acordadas de antemão para formatos de data e arredondamento decimal. O mutation testing verifica os próprios testes: injete um defeito e, se a suíte não pegar, a cobertura não é real. Um mutation score abaixo de 95% marca o módulo para mais geração de testes. O GitHub Copilot em agent mode conduz esse trabalho pelo model picker, com o Haiku 4.5 varrendo, o Sonnet 5 gerando testes e traduções, e o Opus 4.8 analisando as divergências que o motor de comparação expõe.

Fase 7, Cutover: cinco condições antes do corte

A Fase 7 vira o switch, mas apenas quando as cinco condições são satisfeitas ao mesmo tempo. Primeira, a divergência do shadow traffic fica abaixo do limiar de SLO ao longo de um ciclo de negócio completo. Segunda, a paridade de performance é confirmada contra a linha de base, com o p95 online dentro de 10% e o tempo total de batch dentro de 15%. Terceira, o procedimento de rollback foi testado, não apenas escrito. Quarta, o runbook de incidentes está em vigor e o time de plantão o ensaiou. Quinta, o executivo responsável assinou a aprovação de compliance. Quatro dessas condições são técnicas e mensuráveis. A quinta é uma pessoa responsável pelo resultado, e esse é o ponto: nenhum agente assina um corte.

A decisão go/no-go lê de um checklist fixo, não de um palpite. Score de complexidade igual ou abaixo de 7, cobertura de testes igual ou acima de 85%, equivalência comportamental em 100%, p95 online dentro de 10% da linha de base, batch dentro de 15%, zero achados de segurança críticos ou altos, e um ciclo de negócio limpo de shadow traffic. Se qualquer linha falhar, o incremento permanece em teste. O primeiro módulo entra em produção desse jeito; os próximos cem viram trabalho de fábrica, com regras de desmembramento reutilizáveis e pipelines pré-construídos. Em regime estável, um time consegue mover de 50K a 500K linhas de código por trimestre, com primeiro resultado em produção em 90 a 180 dias, contra a mediana do setor de 9 a 18 meses.

DEFINIÇÃO

As cinco condições de corte: divergência do shadow traffic abaixo do limiar de SLO, paridade de performance confirmada contra a linha de base, procedimento de rollback testado, runbook de incidentes ensaiado, e sign-off de compliance de um executivo responsável. Quatro são medidas. A quinta é um nome na aprovação.

Governança: gates de aprovação, tetos de custo em AI Credits, logging de auditoria

Governança não é uma fase. Ela atravessa todas elas. A meta do modelo operacional é 70% a 80% do trabalho executado por agentes e 20% a 30% de supervisão humana, e seis decisões permanecem com humanos por mais capazes que os agentes fiquem: aprovação final da arquitetura-alvo, validação de que as regras de negócio foram preservadas, ajuste de performance para cargas específicas, sign-off do corte em produção, autorização regulatória e escalação de casos de borda. Cada fase termina em um gate de aprovação com um artefato pronto para rollback, e um guardrail verde, amarelo, vermelho decide se um incremento pode prosseguir.

Três controles mantêm o trabalho dos agentes prestando contas. Os tetos de custo vêm primeiro. O custo de agente em pay-per-use escala rápido, então defina um teto por módulo em GitHub AI Credits, em que um crédito equivale a um centavo de dólar, meça cada execução de agente contra esse teto e pare ao atingi-lo, em vez de descobrir pela fatura. O logging de auditoria vem em segundo: cada ação de agente, versão de modelo e justificativa de decisão fica registrada, de modo que um corte possa ser explicado e, se preciso, revertido. A governança como código vem em terceiro. Um arquivo .github/copilot-instructions.md dá a toda sessão do Copilot os mesmos padrões, os custom agents carregam permissões com escopo, e os servidores MCP expõem os sistemas corporativos por uma única interface auditada. Os gates de escalação encaminham os casos difíceis para pessoas: complexidade acima de 8, qualquer violação de orçamento de performance, e achados de segurança acima da severidade média. Timeouts, circuit breakers e versionamento de agentes contêm os loops descontrolados que o Gartner chama de anarquia de agentes. Esta é a direção para onde a governança de IA já caminha, dos princípios

manuais para controles automatizados alinhados ao EU AI Act, ao NIST AI Risk Management Framework e à ISO 42001.

Referências

Fontes primárias para a arquitetura de coexistência, o método de teste em shadow e os controles de governança deste capítulo.

- Azure Architecture, Strangler Fig pattern, o padrão de referência para rodar sistemas legado e moderno lado a lado durante a coexistência.
- Confluent, Liberate Mainframe Data, Change Data Capture de bancos de mainframe para a nuvem, com uma única fonte da verdade e sem dual-write.
- Azure Front Door, Routing Methods, roteamento por peso que move o tráfego de 1% a 100% sem deploy.
- Azure API Management, Multi-Region Deployment, a fachada que roteia e espelha requisições para o shadow traffic.
- Azure Monitor, Application Insights, tracing distribuído para comparação de performance lado a lado no corte.
- GitHub Copilot coding agent, o agent mode e o model picker que conduzem descoberta, tradução e teste.
- GitHub, Automate repository tasks with agentic workflows, como gates de aprovação e verificações de qualidade são aplicados no CI/CD.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps

05 PARTE V, A FÁBRICA E O COMEÇO

Do primeiro módulo a uma fábrica: o que quebra, o que escala e por onde começar.

Modernizar um parque mainframe não falha no primeiro módulo. Falha no centésimo, quando o mesmo trabalho manual que entregou o módulo um não consegue entregar o módulo cem. Este capítulo nomeia os seis modos de falha que travam a modernização agêntica, a mitigação de cada um, o modelo de fábrica que torna a vazão previsível e um começo de duas semanas no GitHub Copilot que um comitê diretivo pode financiar.

Seis modos de falha e a mitigação de cada um

Já vi a modernização agêntica travar sempre das mesmas seis maneiras. Nenhuma delas é falha de modelo. São falhas de processo, e cada uma tem uma mitigação específica que não custa nada adotar no primeiro dia e custa muito reinstalar depois. Nomeie todas antes de o primeiro módulo entrar em produção, não depois.

A primeira é o JOBOL: lógica COBOL traduzida linha a linha para Java, de modo que o código novo carrega a estrutura antiga e nenhum dos ganhos. A mitigação é uma passagem de rearquitetura antes da tradução, em que o agente propõe os limites-alvo e um arquiteto humano os aprova. A segunda é o corte big-bang, em que todo o parque vira em um fim de semana e não há caminho de volta. A mitigação é o padrão strangler fig com change data capture e um período de coexistência de 12 a 24 meses, para que o tráfego migre gradualmente e o mainframe permaneça a fonte da verdade até que um executivo responsável assine o corte. A terceira é o test rot, em que a suíte é gerada a partir do código traduzido e, portanto, certifica a tradução contra ela mesma. A mitigação é gerar testes de equivalência comportamental a partir do comportamento COBOL original e depois fazer mutation testing na suíte para provar que ela pega defeitos injetados.

A quarta é a anarquia de agentes, em que vários agentes escrevem no mesmo repositório sem coordenação e sem trilha de auditoria. A mitigação é um orquestrador único rodando no Opus 4.8, toda ação de agente registrada e gates de aprovação em cada ponto de decisão. A quinta é o estouro de custo. No GitHub Copilot, o uso de modelos em agent mode é cobrado em GitHub AI Credits, em que um crédito equivale a um centavo de dólar, e um loop de agente sem limite pode queimar créditos sem nada a mostrar. A mitigação é um teto de créditos por módulo e gates de custo entre as waves, para que a Wave 2 só comece quando a Wave 1 tiver provado sua economia unitária. A sexta é o vendor lock-in, em que o código-alvo fica preso a um SDK proprietário. A mitigação é uma linguagem-alvo padrão e idiomática, sem injeção de SDK proprietário, e roteamento multi-modelo pelo picker do Copilot, para que nenhum modelo isolado seja indispensável.

O modelo de fábrica: regras de desmembramento reutilizáveis e custo previsível

O primeiro módulo é um trabalho artesanal. Todo módulo depois dele deve ser trabalho de fábrica. Essa virada é o objetivo central do framework, e é onde o retorno do esforço realmente aparece. Uma vez que um módulo tenha entrado em produção com testes comportamentais passando, as regras de desmembramento que o fatiaram, o template de PR que o documentou e o pipeline de CI/CD que o validou tornam-se ativos reutilizáveis. Os próximos cem módulos os herdam.

Codifique esses ativos onde o agente os lê. Regras de desmembramento e convenções de tradução pertencem ao .github/copilot-instructions.md, para que toda sessão em agent mode comece pelas mesmas regras em vez de redescobri-las. O contexto empresarial, o grafo de dependências, os dados de runtime SMF e as restrições de compliance chegam ao agente por conectores MCP que toda wave reutiliza. Custom agents encapsulam os papéis recorrentes, descoberta, tradução, geração de testes, revisão, para que a equipe os acione pelo nome, e não pelo prompt. Com isso no lugar, o custo deixa de ser surpresa. Cada módulo carrega um orçamento conhecido de GitHub AI Credits, e uma fábrica em regime estável produz de 50K a 500K linhas de código traduzido por trimestre. O primeiro resultado em produção chega em 90 a 180 dias, contra a mediana do setor de 9 a 18 meses para programas de modernização de legado que nunca se industrializam.

Um ponto de partida de duas semanas no GitHub Copilot

Você não precisa de um escritório de programa para começar. Precisa de um módulo e duas semanas. Escolha um único módulo COBOL abaixo de 5.000 linhas e com baixa criticidade de negócio, o tipo de módulo folha cuja falha não pararia o banco. Provisione o GitHub Copilot com agent mode e o Claude no Azure AI Foundry, para que os mesmos modelos do picker do Copilot, Opus 4.8 para análise, Sonnet 5 para tradução, Haiku 4.5 para varredura, estejam disponíveis ao pipeline. Rode o agente de descoberta nesse módulo e compare a saída com o modelo mental que sua equipe tem do código. As lacunas que ele encontra, e as que ele deixa passar, dizem quanto confiar nele.

Depois gere a tradução e os testes de equivalência comportamental para esse único módulo folha e escreva a linha de base de governança enquanto o trabalho é pequeno: gates de aprovação em cada ponto de decisão, um teto de GitHub AI Credits por módulo e logging de auditoria em toda ação de agente, tudo registrado no .github/copilot-instructions.md para que valha em toda sessão posterior. Ao final de duas semanas, você tem um módulo traduzido com testes comportamentais passando, um plano de governança documentado e um caso credível para a Wave 2. Esse é o artefato que conquista o próximo comprometimento orçamentário.

Pontos-chave para o comitê diretivo

Um comitê diretivo não financia uma tecnologia. Financia um plano ajustado ao risco. O plano aqui é incremental por desenho: sete fases, um gate de decisão entre cada uma e um artefato com rollback disponível em cada gate, para que o próximo comprometimento orçamentário seja sempre justificado pelo resultado anterior. A meta operacional é de 70 a 80 por cento do trabalho executado por agentes e 20 a 30 por cento de supervisão humana, com seis decisões que permanecem com humanos: aprovação final da arquitetura-alvo, validação de que as regras de negócio foram preservadas, ajuste de performance, sign-off do corte em produção, autorização regulatória e escalação de casos de borda.

Financie três salvaguardas e o resto decorre. Primeira, controle de custo: um teto de créditos por módulo e gates de custo entre as waves, porque o gasto de agentes em

GitHub AI Credits escala rápido e precisa de um teto antes de precisar de um orçamento. Segunda, prova comportamental: testes gerados a partir do comportamento COBOL original, não do código traduzido, porque uma suíte que avalia a tradução contra ela mesma não prova nada. Terceira, governança em código: gates de aprovação, logging de auditoria e controle por orquestrador único registrados no .github/copilot-instructions.md, para que as regras viajem com o repositório. Aprove essas três e o primeiro módulo vira uma fábrica. Pule-as e ele continua um experimento.

> DEFINIÇÃO
>
> A decisão do comitê diretivo em uma frase: financie um plano incremental de sete fases com um artefato com rollback disponível em cada gate, mantenha as seis decisões inegociáveis com humanos e limite o gasto de agentes com um teto de GitHub AI Credits por módulo, para que o primeiro módulo entregue vire uma fábrica repetível, e não um caso isolado.

Referências

Fontes primárias para o padrão strangler fig, o ferramental de agentes e os controles de custo em que este capítulo se apoia.

- Gartner, 7 Options to Modernize Legacy Systems, o espectro 5R e o alerta de que as ferramentas atuais de GenAI traduzem código mas excluem a arquitetura, a raiz do modo de falha JOBOL.
- Microsoft Azure Architecture Center, Strangler Fig pattern, o padrão de coexistência que mitiga o corte big-bang.
- Microsoft Cloud Adoption Framework, Mainframe migration, a orientação de referência da Microsoft para migração de mainframe em etapas.
- GitHub Copilot Documentation, About coding agent, o agent mode, a superfície de execução do começo de duas semanas.
- Microsoft Learn, Use Claude models in Azure AI Foundry, rodar os modelos do picker do Copilot via Foundry para o pipeline de modernização.

- Microsoft Learn, Cost Management and Billing, a superfície de controle de custo por trás dos tetos por módulo e dos gates de custo.
- DORA, 2025 State of AI-assisted Software Development, embasamento empírico sobre entrega assistida por IA e as salvaguardas que a mantêm estável.

Paula Silva

AI-NATIVE SOFTWARE ENGINEER

in/paulanunes

v1.0.0 · 2026-07-02

Building the future of software development with AI and Agentic DevOps