LEGACY MODERNIZATION FRAMEWORK · PLAYBOOK

Modernize a COBOL estate to Azure, module by module, with agents you can govern.

Brazil's core banking, tax, and social systems run on COBOL that predates the people paid to maintain it, and the workforce that understands it is leaving 15% a year. This playbook is the operating manual for modernizing a mainframe estate with agentic AI: a portfolio strategy, a governed reference architecture, a seven-phase run sheet from assess to cutover, and a factory model that turns the first shipped module into predictable throughput, all on GitHub Copilot with the model picker, AI Credits, and agent mode.

AUTHOR

Paula Silva

ROLE

AI-Native Software Engineer

CONTACT

in/paulanunes

READING TIME

~ 45 minutes, end to end

7	6	6
MODERNIZATION PHASES	CHAPTERS	HUMAN DECISION GATES

70-80%

WORK EXECUTED BY AGENTS

CHAPTERS

Chapter 00

Standing still is the highest risk

Brazil's COBOL estate, a workforce shrinking 15% a year, three regulatory forcing functions with a 24 to 36 month window, and how agentic AI inverts the understanding-heavy cost curve.

Imperative →

Chapter 01

No single strategy fits the whole estate

Gartner's 5R spectrum read as a portfolio, Microsoft's four approaches on Azure, the JOBOL trap, and scoring readiness to defend Extend over Defend.

Strategy →

Chapter 02

Five layers, routed models, governed agents

The five-layer reference architecture, model routing in the Copilot picker, MCP and copilot-instructions.md as the control plane, and six custom agents under three orchestration patterns.

Architecture →

Chapter 03

Assess, carve, translate, prove performance

The four operational phases that carry a COBOL module from unknown to production, with the numbers that separate a shipped module from a stalled one.

Run sheet →

Chapter 04

Two worlds running until the cutover gate

The strangler fig with CDC and gradual traffic shift, shadow traffic and behavioral equivalence, the five cutover conditions, and governance across all three.

Coexistence →

Chapter 05

From the first module to a factory

The six failure modes and the mitigation for each, the factory model of reusable carving rules, and a two-week start on GitHub Copilot a steering committee can fund.

Factory →

READING TIPS

> **KEYBOARD SHORTCUTS**
>
> Press `/` to search. Use `j` and `k` to move between chapters. Press `t` to toggle theme. Press `g` to go to top.

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 THE IMPERATIVE

Standing still is the highest risk.

Brazil's core banking, tax collection, and social programs run on COBOL that predates most of the people now paid to maintain it. The code is not failing. The workforce that understands it is leaving at a measurable rate, and the regulators have stopped waiting. This chapter measures the cost of standing still, and shows why agentic AI is the first thing that makes a 24 to 36 month modernization window realistic instead of aspirational.

Brazil runs on COBOL, and the people who understand it are leaving

Brazil's COBOL estate runs between 1.25 billion and 2.1 billion lines of code. It sits inside commercial banks, development banks, insurance carriers, government agencies, and retail. These systems are not broken. They clear payments, collect taxes, and run social services at national scale, and they have done so reliably for decades. That reliability is exactly what makes them dangerous to defer. A system that works is easy to leave alone, and leaving it alone is the decision that quietly compounds into the biggest exposure on the balance sheet.

The problem is not the code. It is the people. Brazil loses roughly 15% of its COBOL developers every year, and there is no replacement pipeline behind them. Universities stopped teaching the language a generation ago. The engineers who wrote the original business rules are retiring, and they are taking undocumented knowledge with them. Every year of delay shrinks the pool of people who can explain what a given program actually does. This is why I treat workforce attrition, not technical debt, as the first-order risk. You can refactor bad code. You cannot interview an engineer who left three years ago.

Three regulatory forcing functions, and a 24 to 36 month window

Three regulatory forces have converged, and each one pushes in the same direction. The first is BACEN's 2028 modernization mandate, which requires API-based open banking interfaces and cybersecurity controls that mainframe platforms do not deliver natively. The second is LGPD, whose data governance requirements (consent, lineage, and the right to erasure) were never designed into architectures built in the 1980s. The third is the compliance stack that keeps rising underneath everything: SOX, ISAE 3402, and PCI-DSS. None of these are optional, and none of them wait for a convenient budget cycle.

Set against those deadlines is the economics of understanding. Reverse-engineering business rules has historically consumed 40% to 60% of modernization budgets. That is why most projects stall in the understanding phase and never reach translation. The teams run out of money before they run out of COBOL. When I put the regulatory deadlines next to the historical cost of understanding, the usable window to act lands at 24 to 36 months. That is not a comfortable runway. It is the reason agentic AI matters here, because it attacks the exact phase that used to eat the budget.

> DEFINITION
>
> The window to act is 24 to 36 months. It is bounded on one side by BACEN's 2028 mandate and the compliance stack underneath it, and on the other by a COBOL workforce that shrinks 15% a year. Standing still does not hold the position. It spends the runway.

How agentic AI inverts the understanding-heavy cost curve

The reason legacy modernization has such a poor track record is where the money went. In the traditional distribution, understanding alone consumed about half the budget: 50% understanding, 20% translation, 15% testing, 15% cutover. Projects that spend half their money before writing a line of new code tend not to finish. Agentic

AI inverts that curve. The AI-driven distribution runs closer to 10% discovery, 20% translation, 35% behavioral equivalence testing, and 35% coexistence and governance. The budget moves out of understanding and into validation, which is the phase that actually decides whether the new system is safe to trust.

GitHub Copilot in agent mode is what makes discovery cheap. Driven by a long-context model from the picker, it maps dependencies across thousands of files, traces execution paths, and surfaces couplings that experienced developers miss. Models in the 1-million-token class hold an entire COBOL application in context at once, copybooks, JCL, and data definitions together. Routing follows the shape of the work: Haiku 4.5 scans and triages at high volume, Sonnet 5 translates module by module, Opus 4.8 does the deep architectural analysis. All of it runs under a .github/copilot-instructions.md file that pins conventions, custom agents that own each stage of the pipeline, and MCP servers that give the agents live access to the repository, the build, and the ticketing system. Cost is metered in GitHub AI Credits, where one credit is one US cent, so a per-module credit ceiling becomes a real budget control rather than a guess.

The six decisions that stay with humans

The operating target is 70% to 80% of the work executed by agents, with 20% to 30% human oversight. That ratio only holds if the human share is spent on the decisions that carry irreversible consequences, not on reviewing syntax. Six decisions stay with people, and they do not move. Final approval of the target architecture. Validation that business rules were preserved, not just that the code compiled. Performance tuning for specific workload patterns. Production cutover sign-off. Regulatory compliance authorization. And escalation of the edge cases the agents flag but cannot resolve on their own.

Keeping those six with humans is what separates modernization from translation. Gartner is blunt about the failure mode: current GenAI tools are good at translating code and explicitly exclude architecture from their scope. Translate COBOL straight into Java without rethinking the design and you get JOBOL, which is COBOL logic wearing Java syntax, and it perpetuates every structural problem you were trying to escape. The agents do the volume. The humans own the architecture, the

compliance posture, and the moment of cutover. That division of labor is the whole method, and the rest of this playbook is how to run it.

> DEFINITION
>
> Six decisions never leave humans: final approval of the target architecture, validation that business rules were preserved, performance tuning for workload patterns, production cutover sign-off, regulatory compliance authorization, and escalation of edge cases. Agents execute the other 70% to 80%. The split is the method.

References

Primary sources for the workforce risk, the regulatory forcing functions, and the modernization approaches this chapter builds on.

- Reuters, COBOL Cowboys Aim to Rescue Aging Government Systems, on the shrinking pool of COBOL engineers and the undocumented knowledge that retires with them.
- Banco Central do Brasil (BACEN), the source of the 2028 modernization mandate and the open banking interface requirements.
- Gartner, 7 Options to Modernize Legacy Systems, the 5R spectrum and the case for incremental modernization over big-bang cutover.
- Microsoft Learn, Mainframe migration in the Cloud Adoption Framework, the reference approaches for rehost, refactor, rearchitect, and replace.
- GitHub Copilot coding agent documentation, the agent mode platform that makes automated discovery cheap enough to change the cost curve.
- Grand View Research, Mainframe Modernization Market, market context for the scale of the legacy estate now under modernization pressure.

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 PORTFOLIO STRATEGY

No single strategy fits the whole estate.

No modernization strategy survives contact with a whole mainframe estate. A COBOL portfolio holds applications that differ in business criticality, coupling, data complexity, and remaining useful life. Treating them all with one pattern, whether that pattern is rehost, refactor, or replace, guarantees waste on one end and risk on the other. This chapter treats the estate as a portfolio. It maps each application to a strategy, reads Microsoft's four approaches as Azure tooling choices, names the trap that turns translation into false progress, and shows how to score readiness and defend the Extend decision over the Defend default.

Gartner's 5R spectrum as a portfolio, not a mandate

The 5R framework from Gartner defines five levels of transformation: Retain, Rehost, Refactor, Rearchitect, and Replace. Each carries its own cost, risk, and value profile. The mistake enterprises make is picking one level and applying it to the whole estate. The portfolio approach does the opposite. It classifies each application on two axes, business criticality and technical complexity, and lands it in the pattern that fits. Gartner projects that by 2030, more than 75% of enterprises running IBM mainframes will adopt low-risk, incremental strategies over big-bang replacement.

Walk the spectrum end to end. Retain means no code change: stable systems near end of life, low effort, low value. Rehost is lift and shift to the cloud with the code kept as is, a quick infrastructure win. Refactor translates the language while keeping the architecture, and it is the sweet spot for AI acceleration. Rearchitect redesigns the boundaries first and then translates, higher effort for higher value. Replace retires the legacy outright and adopts SaaS or rebuilds. The TIME model (Tolerate, Invest, Migrate, Eliminate) gives the portfolio disposition logic that decides which level each application earns.

Where GitHub Copilot fits the classification

The classification is only tractable if you can read the estate fast. GitHub Copilot in agent mode, driving a long-context model from the picker such as Opus 4.8, can hold an entire COBOL application with its copybooks, JCL, and data definitions at once and produce a first-pass 5R score for each component. Encode the scoring rubric as a custom agent and a .github/copilot-instructions.md file so every repository is classified the same way, not by whoever happens to open it. The human architect approves the bucket. The agent does the reading and the first draft of the rationale.

Microsoft's four approaches mapped onto Azure tooling

The Microsoft framework aligns with the 5R model but names Azure as the target platform. It offers four approaches. Migrate is rehost, delivered with Azure Migrate and mainframe emulation. Modernize is refactor, delivered with the Mainframe Modernization Agent and Azure AI Foundry. Rearchitect uses Azure Functions, AKS, Service Bus, Event Grid, and Cosmos DB. Replace uses Dynamics 365, Power Platform, ISV solutions, and Microsoft Fabric. Alongside these, the Hybrid and Extend pattern wraps mainframe transactions in APIs so the legacy keeps running while it is modernized incrementally.

The tooling matters less than the routing between tools. MCP connectors give a Copilot agent live access to the sources it needs to do real work: GitHub, Azure DevOps, Backstage, the mainframe inventory, and Key Vault for secrets. The concrete work under each approach is specific. Data migration moves VSAM and IMS stores to Azure SQL. JCL batch scheduling gives way to modern orchestration. CICS transactions become REST APIs or Azure Functions. Each approach is a different target on the same platform, not a different vendor to lock into.

The JOBOL trap: why translation alone is not modernization

Gartner research (G00822567) confirms a limit that vendors tend to blur. Current GenAI tools excel at code translation and framework refactoring, but they explicitly exclude architecture modernization from their scope. Translate COBOL to Java while preserving the original code structure and you get what Gartner calls JOBOL: COBOL logic implemented in Java. It compiles. It passes a first look. And it perpetuates every structural problem of the original in a new language your team still cannot maintain.

With agents, JOBOL happens for a simple reason. A .github/copilot-instructions.md file that says only convert this COBOL program to Java gets exactly that: a line-by-line transliteration. The fix is to run a rearchitecture pass first. The AI proposes target boundaries, and humans approve them. Only then does translation target those boundaries rather than the original PERFORM THRU control flow. Gartner also documents that tools promising automatic monolith-to-microservices decomposition have largely dialed back those claims, so plan for substantial manual analysis and design regardless of tool support. Translation accuracy in typical LLM-assisted scenarios lands around 80%, and AI code assistant acceptance sits near 30% in practice, so human review is not optional.

> DEFINITION
>
> JOBOL is COBOL logic wearing Java syntax. It compiles, it passes a naive test, and it carries every structural flaw of the original into a language your new hires can read but still cannot maintain. Translation is a step inside modernization. It is not modernization.

Scoring readiness and defending Extend over Defend

Score each application on four dimensions before you assign a strategy. Code complexity covers total lines, cyclomatic complexity, and call-chain depth. Data complexity covers VSAM, IMS, DB2, Adabas, and referential integrity. Integration

complexity covers the number of interfaces, the middleware in play, and batch versus real-time patterns. Operational complexity covers batch scheduling, disaster recovery, SLAs, and compliance exposure. Function Point Analysis, following IFPUG, gives a language-independent size measure so COBOL and Natural estates compare on one scale. As a benchmark, automated COBOL-to-Java conversion runs roughly 0.5 to 2 hours per function point for straightforward code, rising to 4 to 8 hours for complex logic that needs manual review.

The scoring exists to answer one economic question. Gartner (G00823006) separates three GenAI business models: Defend, which maintains the legacy, Extend, which adds capability around it, and Upend, which replaces the model itself. The finding that should change budgets is stark. Defend strategies show a financial return of -52%, Extend strategies deliver +134%, and yet 50% of current GenAI investment still targets Defend. The reading is direct. The default instinct is to protect what already runs, and that instinct loses money.

Governing the spend with GitHub AI Credits

The Extend case only holds if you control the cost of getting there. GitHub Copilot agent usage is billed in GitHub AI Credits, where one credit equals one US cent. Set a per-module credit ceiling and an iterative validation gate before each new wave, so cost is bounded per unit of work rather than open-ended. A Defend posture keeps paying MIPS licensing and 60% to 70% of the IT budget on maintenance with no compounding return. An Extend posture spends metered credits on work that produces a modern, maintainable asset. Frame the portfolio toward Extend and Modernize, hold Defend for the applications genuinely near end of life, and let the readiness score, not instinct, decide.

References

Primary sources for the 5R spectrum, the Azure mapping, the JOBOL scope limit, and the Defend-versus-Extend economics.

- Gartner, 7 Options to Modernize Legacy Systems, the modernization spectrum this chapter reads as a portfolio rather than a mandate.
- Microsoft Learn, Mainframe migration: Cloud Adoption Framework, the four approaches and their Azure tooling targets.

- Gartner, Predicts 2026: Reducing Costs and Risks in Mainframe and Midrange Modernization, the projection that most enterprises choose incremental over big-bang modernization by 2030.
- IFPUG, Function Point Counting Practices Manual, the language-independent sizing method behind readiness scoring across COBOL and Natural.
- GitHub Copilot, the reference agent platform for classification, translation, and cost-governed execution in this chapter.

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 REFERENCE ARCHITECTURE

Five layers, routed models, governed agents.

A modernization pipeline is only as trustworthy as the architecture that runs it. This chapter describes the reference architecture I use for agentic legacy modernization: five layers from the developer interface to the perimeter, a routing rule that assigns each task to the right model in the GitHub Copilot picker, a control plane built from MCP connectors and .github/copilot-instructions.md, and six custom agents coordinated by three orchestration patterns. Every layer exists to keep autonomy inside a governed boundary.

The five layers, from developer interface to perimeter

The reference architecture has five layers, and each one builds on the layer beneath it. The developer interface is where humans drive the work. Engineers open GitHub Copilot in agent mode inside VS Code or on GitHub.com, and the repository carries a .github/copilot-instructions.md file that tells every agent the project conventions, the target language, and the modernization rules. That file is the first thing an agent reads, so it is where the control starts.

The second layer is multi-model orchestration, the intelligence of the pipeline. An orchestrator model decomposes the task, worker models translate and test, scanner models triage, and Microsoft Semantic Kernel coordinates the handoffs. The third layer is the MCP connectors, the enterprise tools the agents reach through the Model Context Protocol: GitHub, Azure DevOps, Backstage, the database, observability, Key Vault, and the work-item tracker. The fourth layer is the platform, where governance lives: Azure AI Foundry hosts the models, Microsoft Entra ID handles identity, Key Vault holds the secrets, and Azure Monitor logs every action. The fifth layer is the perimeter, the external interfaces: the customer mainframe that still holds the source of truth, the target Azure services, and the external APIs. The design

principle is that every layer connects back to the platform layer, so no agent action escapes identity, logging, or policy.

Model routing in the GitHub Copilot picker

No single model handles every task well, and no single model is worth its cost on every task. The GitHub Copilot picker lets you choose the model per task, and the pipeline uses a simple routing rule: Haiku 4.5 scans, Sonnet 5 translates, Opus 4.8 analyzes. Haiku 4.5 runs the high-volume, low-complexity work: syntax validation, pattern matching, triage classification. Sonnet 5 does the balanced code work: COBOL-to-Java translation, test generation, documentation, and pull request creation. Opus 4.8 takes the reasoning that needs the whole picture: full-application analysis, business rule extraction, and architectural recommendations, using its 1-million-token context to hold an entire COBOL application with its copybooks, JCL, and data definitions at once.

Routing is a cost decision as much as a quality decision. Since June 1, 2026, GitHub Copilot bills agent work in GitHub AI Credits, where one credit is worth one cent, and the heavier models consume more credits per task. Sending a syntax check to Opus 4.8 wastes credits; sending an architecture decision to Haiku 4.5 wastes the analysis. The picker makes the choice explicit and auditable. The same discipline can be automated: the Azure AI Foundry Model Router selects a model per request from the request characteristics, and the source material reports it cuts token spend by up to 40% by sending simple requests to smaller models without developer intervention. The rule stays the same whether a human picks the model or the router does: match the model to the task.

MCP connectors and copilot-instructions.md as the control plane

The agents do not touch enterprise systems directly. They reach them through the Model Context Protocol, an open standard that works like a universal connector for AI agents. Each MCP server exposes a fixed set of tools, and the agent calls those tools instead of writing its own integration code. The GitHub server reads source files and opens pull requests. The database server introspects schemas and samples

data in read-only mode. The Key Vault server retrieves credentials so secrets never land in agent context or logs. The observability server compares performance between the legacy and the modern systems. Because MCP is an open standard, servers from different vendors work with any compatible agent, which keeps the architecture free of lock-in.

The other half of the control plane is the .github/copilot-instructions.md file. It is the persistent instruction set that every agent reads before acting: the coding standards, the target architecture, the rule that forbids injecting any proprietary SDK, and the constraint that translated code must be standard, idiomatic target-language code rather than COBOL logic rewritten in Java. Between MCP and the instructions file, the architecture separates what an agent can do from what it is allowed to do. MCP defines the reachable actions; the instructions file and the platform permission tiers define the allowed ones. That separation is what makes agent autonomy safe to grant.

Six custom agents and three orchestration patterns

Six custom agents run the pipeline, each with a narrow scope and a clear input and output contract. The Discovery agent scans the codebase, starting with Haiku 4.5 and escalating to Opus 4.8 for deep analysis, and produces a dependency graph and a business rule catalog. The Analysis agent, on Opus 4.8, scores each component and recommends a modernization strategy. The Translation agent, on Sonnet 5, converts one module at a time and opens a pull request with mapping documentation. The Test generation agent, also on Sonnet 5, writes behavioral equivalence tests and targets at least 85% coverage, with 100% on critical business rule paths. The Review agent checks the translated code against the standards. The Migration orchestration agent, on Opus 4.8, coordinates the others and holds the plan.

Three orchestration patterns cover the task shapes that arise, and Microsoft Semantic Kernel implements all three natively. Sequential handoffs move work stage by stage, one agent finishing before the next begins. The handoff pattern lets a worker escalate agent to agent when it hits a decision it cannot resolve. Group chat puts several agents into collaborative architectural reasoning, consolidated by the

orchestrator. The danger these patterns guard against is agent anarchy: poorly coordinated agents that collide and loop. The defense is a single orchestrator, a validation gate between every handoff, and a logged action for every step. Autonomy is granted per tier and revoked at the first gate an agent cannot pass.

> DEFINITION
>
> The architecture in one line: five layers from developer interface to perimeter, one routing rule (Haiku 4.5 scans, Sonnet 5 translates, Opus 4.8 analyzes), MCP connectors and .github/copilot-instructions.md as the control plane, and six custom agents under three orchestration patterns. Every layer connects back to the platform, so autonomy always runs inside a governed boundary.

References

Primary sources for the reference architecture, the model routing rule, the MCP control plane, and the agent orchestration patterns.

- Microsoft Cloud Adoption Framework, Mainframe migration, the target-side reference for mapping mainframe components onto Azure services.
- GitHub, Pick your agent: use Claude and Codex on Agent HQ, the model picker and agent mode that drive the developer interface layer.
- Model Context Protocol Specification, the open standard behind the MCP connector layer.
- Microsoft Semantic Kernel, Agent Framework, the framework that implements the sequential, handoff, and group chat patterns.
- Azure AI Foundry, Model Router, automated model selection that the source reports cuts token spend by up to 40%.
- Azure AI Foundry, Claude models, the platform layer that hosts the routed models.
- GitHub Copilot coding agent, the custom agent runtime for autonomous modernization tasks.

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 CARVE AND TRANSLATE

Assess, carve, translate, prove performance: the four-phase run sheet.

This chapter turns the framework into a run sheet. Four phases carry a legacy module from unknown to production. Assess maps what the code actually does. Carve slices it into shippable units. Translate rewrites it module by module. Perform proves the result holds under load. Each phase runs on GitHub Copilot in agent mode, each ends at a human decision gate, and each produces a rollback-ready artifact. Here are the four phases in order, with the numbers that separate a shipped module from a stalled one.

Phase 1, Assess: the discovery agent and the dependency graph

The discovery agent is the first thing you stand up. It runs in GitHub Copilot agent mode, defined as a custom agent through .github/copilot-instructions.md, and it reaches the mainframe estate through MCP servers: a GitHub server for source, a database server for schemas, an observability server for runtime data. It ingests COBOL programs, copybooks, JCL, Natural programs, CICS definitions, and DB2 or Adabas schemas, then produces a program catalog, a data dictionary, a JCL job flow, a transaction map, and a complexity score from 1 to 10 for every program. High-volume scanning routes to Haiku 4.5 in the model picker. Deep dependency reasoning routes to Opus 4.8, whose 1-million-token context holds thousands of programs at once, so the agent traces couplings that span the whole estate instead of one file at a time.

The output is a dependency graph, and the graph is where the assessment earns its keep. It separates orphans, programs with almost no dependencies and the ideal first candidates, from hub programs, called by many others and unable to move until their dependents are ready. Static analysis is not uniformly reliable, so the agent tags

every dependency with a difficulty tier: direct CALL and JCL sequences resolve at 95 to 100 percent accuracy, CICS LINK and MQ interactions at 80 to 90 percent, generated and dynamic calls at 50 to 70 percent, and runtime-only behavior at 35 to 40 percent. For the hardest tier, the agent supplements static analysis with SMF runtime records, which capture the dynamic CALL and indirect routing that no static pass can see. The decision gate that closes this phase is a portfolio classified into the 5R buckets, signed off by a human architect.

Phase 2, Carve: tiered slicing and the smallest shippable module

Carving decides what ships first. A typical banking COBOL estate carries decades of shared copybooks, common data areas, CICS LINK calls, and VSAM files touched by many programs. You cannot extract one piece without knowing what it touches. The analysis agent, running on Opus 4.8, reads the dependency graph and finds cut-points: boundaries where cross-links are minimal, data flows are well-defined and low-volume, and the programs on one side form a coherent business function that an API can front. Programs inside the boundary become a modernization increment. Programs outside keep running on the mainframe.

The order follows the difficulty tiers, not the org chart. Tier 1 leaf modules with low coupling carve first, because they build team capability on wins you can actually finish. Tier 2 intermediate modules follow. Tier 3 core modules with high coupling come last, when the team has the most experience and the most reusable patterns. SMF runtime behavior informs the order: a module that looks isolated in static analysis but shows heavy dynamic invocation at runtime moves later in the queue.

The smallest shippable module

The unit that matters is the smallest module that can ship on its own and prove the pipeline end to end. For the first increment, pick one COBOL module under 5,000 lines with low business criticality, low coupling, and a clean cut-point. It is not the most valuable module. It is the one that lets you run discovery, translation, and behavioral tests through to a signed decision gate without a hard dependency blocking you. That first shipped module is what earns the budget for the next wave, and it turns the sequencing plan from a slide into evidence.

Phase 3, Translate: module by module, semantics over syntax

Translation is the step everyone pictures when they hear modernization, and it is the least likely to fail. Sonnet 5, the translation worker in the model picker, rewrites COBOL into idiomatic Java, .NET, or Python one module at a time, never as a big bang. Each module produces a pull request with mapping documentation that ties every translated unit back to its source paragraph, and every pull request goes to a human reviewer. That is the decision gate for this phase: no module merges without a person reading the diff.

The goal is business semantics, not syntactic structure. The failure mode has a name, JOBOL: COBOL logic reimplemented in Java, which compiles, passes a shallow test, and carries every structural problem of the original into a new language. Gartner is explicit that current generative tools translate code well and exclude architecture modernization from their scope, so the guard against JOBOL is a rearchitect pass before translation. The agent proposes target boundaries, and a human approves them. Behavioral equivalence tests are generated from the original COBOL behavior, not from the translated code, so the suite validates what the system should do rather than what the translation happened to produce.

Phase 4, Perform: SMF baselines and no regression beyond SLOs

A modernized module that is correct but slower than the mainframe does not ship. Business stakeholders will block it, and they should. So this phase starts before any translation, with a performance baseline built from the legacy system itself. SMF records, particularly types 30, 70 to 79, and 110, give transaction-level CPU time, I/O wait, and queue depth. That data establishes not just an average but the full distribution: p50, p95, and p99. A z16 mainframe can process more than 25,000 CICS transactions per second, and the baseline is the number the Azure target must match.

The baseline becomes a performance budget with a hard ceiling per workload. Online transactions at p95 may run at most 10 percent slower than baseline, p99 at most 20 percent, batch total time at most 15 percent, and batch throughput and data

query latency at most 10 percent. Load tests run at production-like volume before any traffic is routed. Application Insights with distributed tracing compares the modern system against the SMF baseline in real time, and if performance degrades past the red guardrail, traffic routes back to the legacy system automatically. No regression beyond the defined SLOs is accepted. The decision gate is a performance sign-off from the SRE team.

> DEFINITION
>
> The go/no-go bar for any module, one line: complexity score at or below 7, test coverage at or above 85 percent with 100 percent on critical business-rule paths, 100 percent behavioral equivalence against the original COBOL, and performance within 10 percent of the SMF baseline. Cost stays bounded by a per-module ceiling in GitHub AI Credits, where 1 credit equals one US cent. Miss any one of these and the module does not go to production.

References

Primary sources for the four phases: the discovery and translation tooling, the mainframe runtime data, and the Azure performance patterns.

- GitHub Copilot coding agent, the reference for running the discovery and translation agents in agent mode.
- Model Context Protocol Specification, the standard interface the agents use to reach source, schemas, and runtime data.
- IBM, System Management Facilities (SMF), the runtime records used for hard-tier dependencies and the performance baseline.
- Azure Architecture, Strangler Fig Pattern, the boundary pattern behind cut-points and the coexistence facade.
- Azure Architecture, Reengineer Mainframe Batch Apps, the batch performance patterns the modernized system must meet.
- Azure Monitor, Application Insights, continuous performance validation against the SMF baseline.

- Gartner, 7 Options to Modernize Legacy Systems, the 5R spectrum and the caveat that code translation is not architecture modernization.

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 COEXIST AND GOVERN

Two worlds running until the cutover gate.

The hard part of modernization is not translating COBOL. It is running the old system and the new one at the same time, for months, while users notice nothing, and then proving the new system is safe enough to take the switch. This chapter covers the last three phases of the journey. Coexist keeps two worlds in sync. Test compares them under real load. Cutover moves production traffic only when five conditions hold. Governance runs across all three, because agents that act without gates, ceilings, and logs are a liability, not an accelerator.

Phase 5, Coexist: the strangler fig, CDC, and gradual traffic shift

Phase 5 runs the strangler fig pattern. A router sits in front of both systems and sends each request to the mainframe or to the modern Azure services. The mainframe stays the source of truth, with read and write authority, until an accountable executive signs the cutover gate. The modern services run idempotently beside it. Three pieces of infrastructure make this work. Azure API Management is the facade that every request passes through. Azure Front Door with weighted routing controls what share of traffic each system receives, and the weights change without a deploy. Change Data Capture keeps the two data stores consistent.

Data synchronization is the part that breaks projects. The rule is simple and absolute: never dual-write. If both systems write to their own stores from the same request, the data diverges within hours. Instead, designate one system as the source of truth for each data domain and replicate to the other. Change Data Capture reads the mainframe transaction log and streams changes to the Azure database in near real time, with typical latency of 1 to 5 seconds. Confluent and tcVISION handle the mainframe-specific side. For transaction-based flows, an event-driven pattern on

Azure Service Bus gives exactly-once semantics. For reference data that rarely changes, a scheduled API sync is enough.

The traffic shift is a ladder, not a leap

Traffic moves in steps, and each step has to earn the next. The modern system starts at 0% of live traffic and runs in shadow mode first, receiving mirrored copies of production requests with no user impact. Then live traffic begins at 1%. It expands to 5% only after 1% has been stable for at least 48 hours, and continues through 10%, 25%, 50%, and 100%. Each step has to meet the go/no-go criteria before the next one opens. When traffic reaches 100%, the mainframe enters standby and stays available for 30 to 90 days for emergency rollback. Every step is reversible within 15 minutes by changing routing rules in API Management, so no cutover is a one-way door.

Phase 6, Test: shadow traffic and behavioral equivalence

Phase 6 is where confidence is earned. Shadow traffic is the method. Every production request to the mainframe is copied and sent to the modern system at the same time. The mainframe response goes to the user. The modern response is captured and compared, never returned. A comparison engine logs three things for each request: whether the outputs match, how much slower or faster the modern system is, and whether either side produced an error the other did not. Shadow traffic has to run for at least one full business cycle, usually a month, so that daily, weekly, and month-end patterns all appear. The shadow load equals full production, so size the Azure infrastructure for it from the start.

Behavioral equivalence is the acceptance test. The equivalence suite is generated from the original COBOL behavior, not from the translated code, because tests written against the translation would only confirm its own mistakes. The same input runs through both systems and the outputs are compared field by field across thousands of fixtures, including boundary, month-end, and leap-year cases. The target is a 100% match, with explicit tolerance rules agreed in advance for timestamp formats and decimal rounding. Mutation testing checks the tests themselves: inject a defect, and if the suite does not catch it, the coverage is not real. A mutation score

below 95% flags a module for more test generation. GitHub Copilot in agent mode drives this work through the model picker, with Haiku 4.5 scanning, Sonnet 5 generating tests and translations, and Opus 4.8 analyzing the divergences the comparison engine surfaces.

Phase 7, Cutover: five conditions before the switch

Phase 7 flips the switch, but only when five conditions all hold. First, shadow traffic divergence sits below the SLO threshold across a full business cycle. Second, performance parity is confirmed against the baseline, with online p95 within 10% and batch total time within 15%. Third, the rollback procedure has been tested, not just written. Fourth, the incident runbook is in place and the on-call team has rehearsed it. Fifth, the accountable executive has signed the compliance approval. Four of these are technical and measurable. The fifth is a person accountable for the outcome, which is the point: no agent signs a cutover.

The go/no-go decision reads from a fixed checklist, not a judgment call. Complexity score at or below 7, test coverage at or above 85%, behavioral equivalence at 100%, online p95 within 10% of baseline, batch within 15%, zero critical or high security findings, and one clean business cycle of shadow traffic. If any line fails, the increment stays in testing. The first module ships this way; the next hundred become factory work with reusable carving rules and pre-built pipelines. At steady state a team can move 50K to 500K lines of code per quarter, with first production in 90 to 180 days against an industry median of 9 to 18 months.

> DEFINITION
>
> The five cutover conditions: shadow traffic divergence below the SLO threshold, performance parity confirmed against the baseline, rollback procedure tested, incident runbook rehearsed, and compliance sign-off from an accountable executive. Four are measured. The fifth is a name on the approval.

Governance: approval gates, cost ceilings in AI Credits, audit logging

Governance is not a phase. It runs across all of them. The operating model target is 70% to 80% of the work executed by agents and 20% to 30% human oversight, and six decisions stay with humans no matter how capable the agents get: final approval of the target architecture, validation that business rules were preserved, performance tuning for specific workloads, production cutover sign-off, regulatory authorization, and escalation of edge cases. Every phase ends at an approval gate with a rollback-ready artifact, and a green, yellow, red guardrail decides whether an increment can proceed.

Three controls keep agent work accountable. Cost ceilings come first. Agent cost on pay-per-use scales fast, so set a ceiling per module in GitHub AI Credits, where one credit equals one US cent, meter every agent run against it, and stop at the ceiling instead of finding out from the invoice. Audit logging comes second: every agent action, model version, and decision rationale is recorded, so a cutover can be explained and, if needed, reversed. Governance as code comes third. A .github/copilot-instructions.md file gives every Copilot session the same standards, custom agents carry scoped permissions, and MCP servers expose enterprise systems through one audited interface. Escalation gates route the hard cases to people: complexity above 8, any performance budget violation, and security findings above medium severity. Timeouts, circuit breakers, and agent versioning stop the runaway loops that Gartner calls agent anarchy. This is the direction AI governance is already moving, from manual principles toward automated controls aligned with the EU AI Act, the NIST AI Risk Management Framework, and ISO 42001.

References

Primary sources for the coexistence architecture, the shadow-testing method, and the governance controls in this chapter.

- Azure Architecture, Strangler Fig pattern, the reference pattern for running legacy and modern systems side by side during coexistence.
- Confluent, Liberate Mainframe Data, Change Data Capture from mainframe databases to the cloud, with a single source of truth and no dual-write.

- Azure Front Door, Routing Methods, weighted routing that shifts traffic from 1% to 100% without a deploy.
- Azure API Management, Multi-Region Deployment, the facade that routes and mirrors requests for shadow traffic.
- Azure Monitor, Application Insights, distributed tracing for side-by-side performance comparison at cutover.
- GitHub Copilot coding agent, agent mode and the model picker that drive discovery, translation, and testing.
- GitHub, Automate repository tasks with agentic workflows, how approval gates and quality checks are enforced in 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 PART V, FACTORY AND START

From the first module to a factory: what breaks, what scales, and where to start.

Modernizing a mainframe estate does not fail on the first module. It fails on the hundredth, when the same manual work that shipped module one cannot ship module one hundred. This chapter names the six failure modes that stall agentic modernization, the mitigation for each, the factory model that makes throughput predictable, and a two-week start on GitHub Copilot that a steering committee can fund.

Six failure modes and the mitigation for each

I have watched agentic modernization stall in the same six ways. None of them are model failures. They are process failures, and each has a specific mitigation that costs nothing to adopt on day one and a great deal to retrofit later. Name them before the first module ships, not after.

The first is JOBOL: COBOL logic translated line for line into Java, so the new code carries the old structure and none of the benefit. The mitigation is a rearchitect pass before translation, where the agent proposes target boundaries and a human architect approves them. The second is the big-bang cutover, where the whole estate flips in one weekend and there is no way back. The mitigation is the strangler fig pattern with change data capture and a coexistence period of 12 to 24 months, so traffic moves gradually and the mainframe stays the source of truth until an accountable executive signs the switch. The third is test rot, where the suite is generated from the translated code and therefore certifies the translation against itself. The mitigation is to generate behavioral equivalence tests from the original COBOL behavior, then mutation-test the suite to prove it catches injected defects.

The fourth is agent anarchy, where several agents write to the same repository with no coordination and no audit trail. The mitigation is a single orchestrator running on Opus 4.8, every agent action logged, and approval gates at each decision point. The

fifth is cost runaway. On GitHub Copilot, agent mode model usage is billed in GitHub AI Credits, where one credit is one US cent, and an unbounded agent loop can burn credits with nothing to show for it. The mitigation is a per-module credit ceiling and cost gates between waves, so Wave 2 only starts once Wave 1 has proven its unit economics. The sixth is vendor lock-in, where the target code is tied to one proprietary SDK. The mitigation is a standard, idiomatic target language, no proprietary SDK injection, and multi-model routing across the Copilot picker so no single model is load-bearing.

The factory model: reusable carving rules and predictable cost

The first module is a craft project. Every module after it should be factory work. That shift is the entire point of the framework, and it is where the return on the effort actually lands. Once one module has shipped to production with passing behavioral tests, the carving rules that sliced it, the PR template that documented it, and the CI/CD pipeline that validated it all become reusable assets. The next hundred modules inherit them.

Encode those assets where the agent reads them. Carving rules and translation conventions belong in .github/copilot-instructions.md, so every agent mode session starts from the same rules instead of rediscovering them. Enterprise context, the dependency graph, the SMF runtime data, the compliance constraints, reaches the agent through MCP connectors that every wave reuses. Custom agents wrap the recurring roles, discovery, translation, test generation, review, so the team drives them by name rather than by prompt. With those in place, cost stops being a surprise. Each module carries a known credit budget in GitHub AI Credits, and a steady-state factory produces 50K to 500K lines of translated code per quarter. First production lands in 90 to 180 days, against an industry median of 9 to 18 months for legacy modernization programs that never industrialize.

A two-week starting point on GitHub Copilot

You do not need a program office to start. You need one module and two weeks. Pick a single COBOL module under 5,000 lines with low business criticality, the kind

of leaf module whose failure would not stop the bank. Stand up GitHub Copilot with agent mode and Claude on Azure AI Foundry, so the same models in the Copilot picker, Opus 4.8 for analysis, Sonnet 5 for translation, Haiku 4.5 for scanning, are available to the pipeline. Run the discovery agent on that module and compare its output against the mental model your team holds of the code. The gaps it finds, and the ones it misses, tell you how much to trust it.

Then generate the translation and the behavioral equivalence tests for that one leaf module, and write the governance baseline while the work is small: approval gates at each decision point, a per-module credit ceiling in GitHub AI Credits, and audit logging on every agent action, all captured in .github/copilot-instructions.md so they bind every later session. At the end of two weeks you have a translated module with passing behavioral tests, a documented governance plan, and a credible case for Wave 2. That is the artifact that earns the next budget commitment.

Key takeaways for the steering committee

A steering committee does not fund a technology. It funds a risk-adjusted plan. The plan here is incremental by design: seven phases, a decision gate between each one, and a rollback-ready artifact at every gate, so the next budget commitment is always justified by the last result. The operating target is 70 to 80 percent of the work executed by agents and 20 to 30 percent human oversight, with six decisions that stay with humans: final approval of the target architecture, validation that business rules are preserved, performance tuning, production cutover sign-off, regulatory authorization, and escalation of edge cases.

Fund three guardrails and the rest follows. First, cost control: a per-module credit ceiling and cost gates between waves, because agent spend on GitHub AI Credits scales fast and needs a ceiling before it needs a budget. Second, behavioral proof: tests generated from the original COBOL behavior, not the translated code, because a suite that grades the translation against itself proves nothing. Third, governance in code: approval gates, audit logging, and single-orchestrator control captured in .github/copilot-instructions.md, so the rules travel with the repository. Approve those three and the first module becomes a factory. Skip them and it stays a science project.

DEFINITION

The steering committee decision in one sentence: fund an incremental seven-phase plan with a rollback-ready artifact at every gate, keep the six non-negotiable decisions with humans, and cap agent spend with a per-module GitHub AI Credits ceiling, so the first shipped module becomes a repeatable factory rather than a one-off.

References

Primary sources for the strangler fig pattern, the agent tooling, and the cost controls this chapter relies on.

- Gartner, 7 Options to Modernize Legacy Systems, the 5R spectrum and the warning that current GenAI tools translate code but exclude architecture, the root of the JOBOL failure mode.
- Microsoft Azure Architecture Center, Strangler Fig pattern, the coexistence pattern that mitigates the big-bang cutover.
- Microsoft Cloud Adoption Framework, Mainframe migration, Microsoft's reference guidance for staged mainframe migration.
- GitHub Copilot Documentation, About coding agent, agent mode, the execution surface for the two-week start.
- Microsoft Learn, Use Claude models in Azure AI Foundry, running the Copilot picker models through Foundry for the modernization pipeline.
- Microsoft Learn, Cost Management and Billing, the cost-control surface behind per-module ceilings and cost gates.
- DORA, 2025 State of AI-assisted Software Development, empirical grounding on AI-assisted delivery and the guardrails that keep it stable.

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