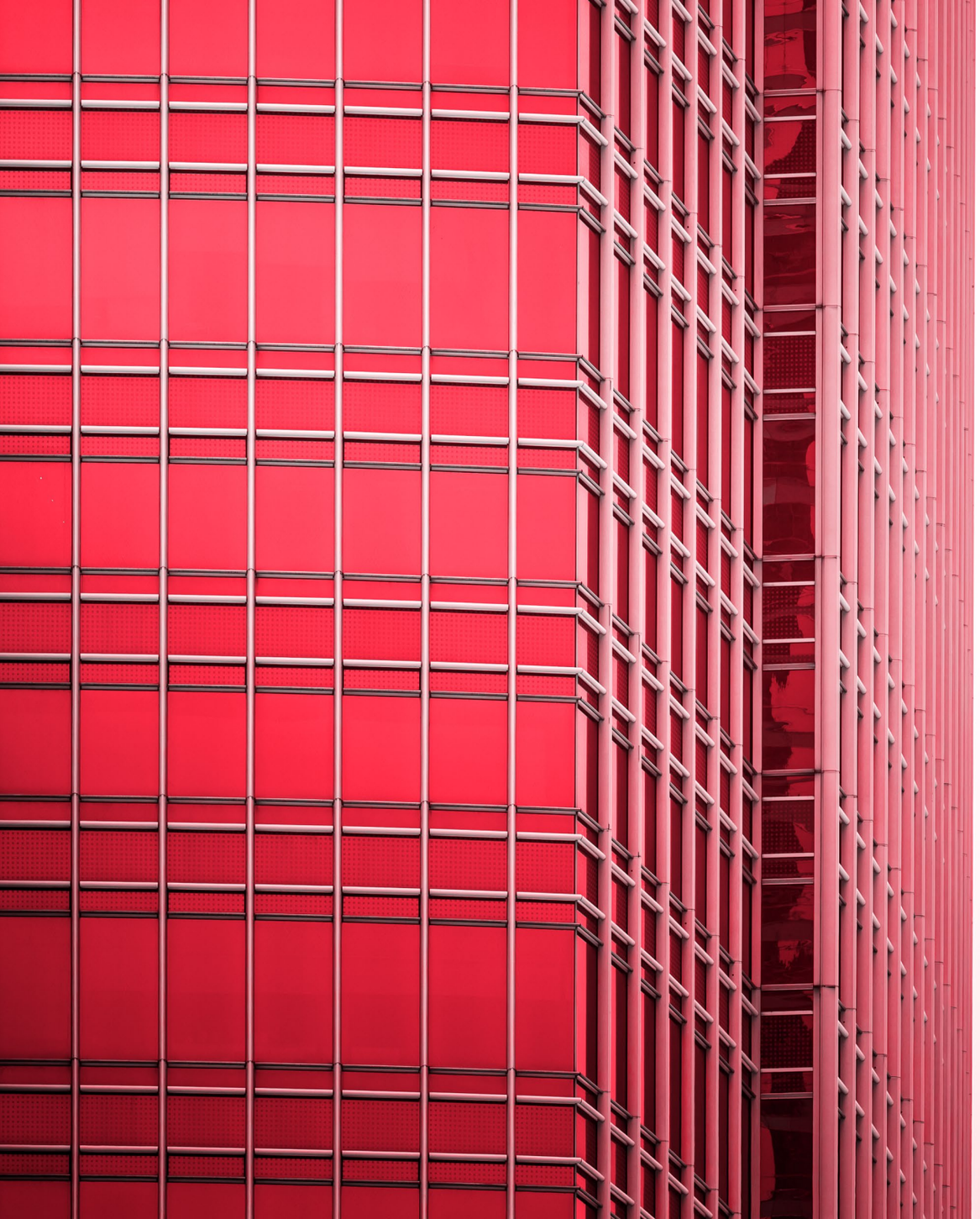
VRIZE

Incremental Legacy Modernization

Four proven methods to retire legacy systems without halting the business

Enter



Contents

Executive Summary

Core thesis

The question is not whether to modernize, but how to modernize without the project failure modes that have defined the last two decades of replacement programs. Incremental methods – Strangler Fig, Branch by Abstraction, Anti-Corruption Layer, and Parallel Run – give engineering organizations the tools to modernize continuously, ship in weeks rather than years, and de-risk at every step.

Legacy systems run the operational backbone of most enterprises. They also represent the single largest source of technical debt, security exposure, and delivery drag in the modern IT portfolio. Yet the most common response to legacy risk – a multi-year, big-bang replacement program – has a failure rate well documented and consistently disappointing.

This paper makes a different argument. Legacy modernization is most reliably achieved not by replacement, but by incremental displacement: a series of small, reversible, business-aligned moves that gradually shift functionality from the legacy estate to a modern target while the legacy system continues to run. Done well, the legacy system shrinks over time until what remains can either be retired or operated at sustainable cost.

This paper compares four incremental modernization methods that VRIZE has applied across delivery engagements in financial services, retail, and consumer-facing platforms. Each method has a specific shape of problem it solves best:

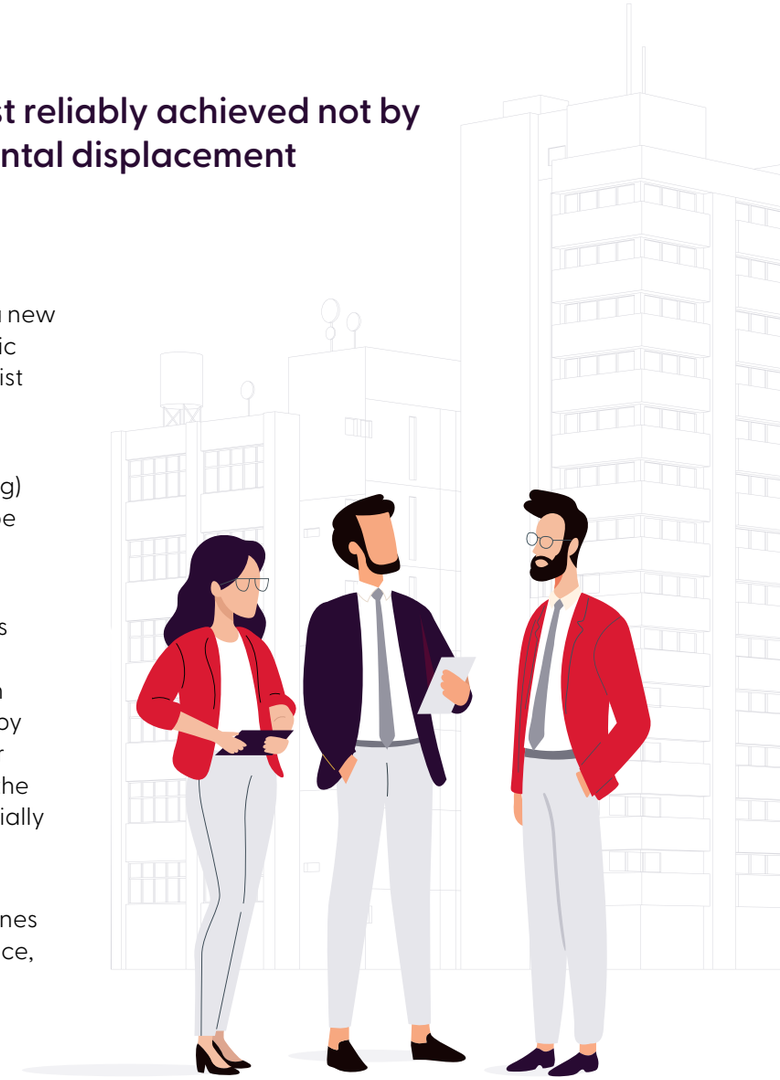
- **Strangler Fig** – for monoliths with well-defined feature boundaries that can be routed individually.
- **Branch by Abstraction** – for replacing a deeply embedded subsystem (data layer, persistence, integration) without forking the codebase.

> **Legacy modernization is most reliably achieved not by replacement, but by incremental displacement**

- **Anti-Corruption Layer** – for protecting a new system from the data model and semantic gravity of the legacy system it must coexist with.
- **Parallel Run** – for high-stakes systems (financial calculations, risk engines, billing) where confidence in equivalence must be proven, not assumed.

These methods are not mutually exclusive. The most effective modernization programs combine them deliberately: a Strangler Fig at the system perimeter, an Anti-Corruption Layer protecting the new services, Branch by Abstraction within the new services as their internals evolve, and a Parallel Run gating the cutover of any calculation-heavy or financially material flow.

The closing sections of this paper offer a selection framework, the operating disciplines that make these methods succeed in practice, and the failure modes to anticipate.



The modernization trap

The underlying problem

Big-bang modernization treats the legacy system as a fixed target to be replaced. Incremental modernization treats the legacy system as a live patient to be operated on – one tissue at a time, with the patient awake and working throughout.

Most large enterprises have attempted at least one major legacy replacement program in the last decade. The pattern is familiar: a multi-year transformation roadmap, a dedicated program team, a target architecture, a sequence of releases, and a hard cutover date. The pattern is also, with depressing regularity, a story of cost overruns, scope compromises, and a cutover that either slips by years or is quietly abandoned.

Why big-bang modernization fails

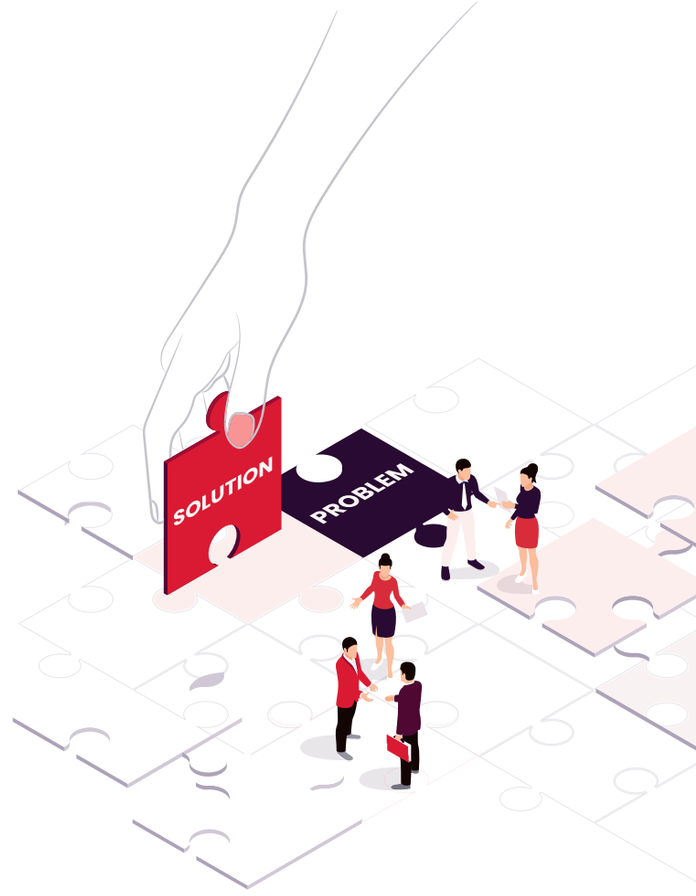
The failure modes are structural, not accidental. Four recur across industries:

- **Knowledge decay outpaces the project.** Legacy systems encode decades of business rules, exceptions, and undocumented assumptions. The people who built them have often retired. A two-year replacement program spends its first six months rediscovering what the system actually does, and discovers more every quarter after that.
- **The business does not stand still.** While the replacement is under construction, the legacy system must continue to absorb regulatory changes, product launches, and operational fixes. Every change to the legacy invalidates a portion of the replacement design, creating a moving target the new system can never quite catch.
- **Risk accumulates to a single point.** Big-bang cutovers concentrate all delivery risk into a single weekend. The decision to go live is binary, the rollback path is often impractical, and the political cost of slipping the date encourages teams to ship before they are ready.
- **Value is deferred to the end.** Stakeholders fund the program for years before seeing any business outcome. When the program inevitably encounters trouble, the absence of incremental value makes it difficult to defend continued investment.

What changed: the incremental alternative

Three shifts have made incremental modernization not just viable but preferable. First, cloud and container platforms make it trivially cheap to run a new service alongside a legacy one. Second, modern API gateways, service meshes, and feature flag platforms make it cheap to route a single user, a single transaction, or a single percent of traffic to a new implementation. Third, observability tooling makes it possible to compare the behavior of two systems in production at scale, in real time.

Together, these capabilities collapse the unit of modernization from “replace the system” to “replace one capability, one route, one calculation.” That collapse is what makes the methods in this paper work.



Principles of incremental modernization

Before examining specific methods, it is worth stating the principles they share. A modernization program that violates these principles will struggle regardless of which method it picks; a program that honors them has options.

Principle 1: **Every move is reversible**

Until a piece of functionality is fully migrated and the legacy implementation has been removed, both implementations should coexist and traffic should be routable to either one. Reversibility is what turns a high-stakes cutover into a low-stakes experiment.

Principle 2: **The legacy system is the source of truth until it is not**

During migration, the legacy system continues to be the authoritative system of record for the functionality being migrated. The new system either reads through to it, runs in parallel with it, or operates behind a strangling proxy that still routes some traffic to it. Premature declaration of cutover is the single most common cause of incident.

Principle 3: **Slice by capability, not by layer**

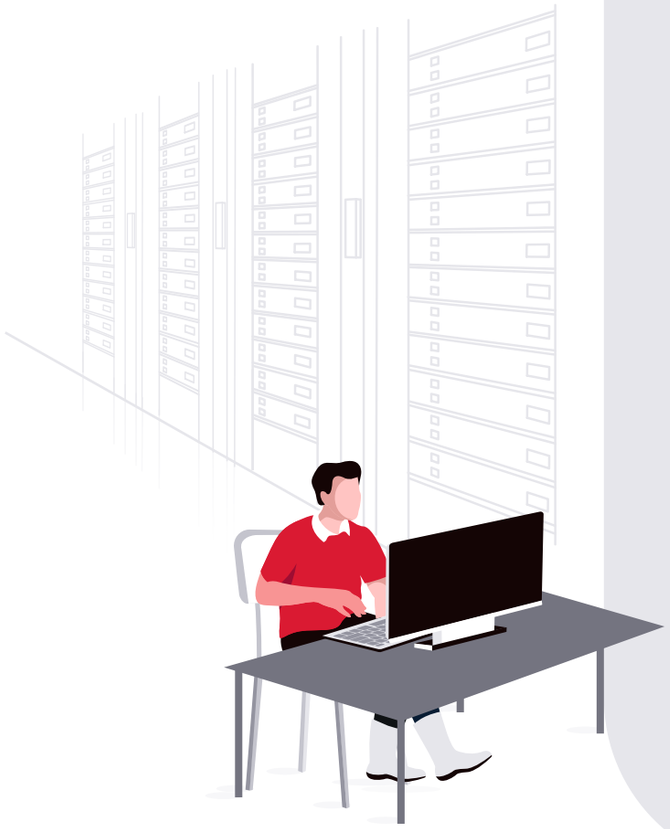
Tempting as it is to modernize "the database first, then the services, then the UI," layered modernization rarely delivers business value until the last layer ships. Vertical slices – a single user-facing capability migrated end-to-end – deliver value with each release and surface integration issues early.

Principle 4: **Observability precedes migration**

Before routing any traffic to a new implementation, the team must be able to observe both implementations side by side: latency, error rates, output equivalence, and business outcomes. If the team cannot detect a regression in production within minutes, the migration is not safe to begin.

Principle 5: **The migration is not done until the legacy is gone**

Programs that declare success when the new system is live, while the legacy continues to run "just in case," tend to find the legacy still running five years later. Retirement of the legacy code, infrastructure, and operational responsibility is part of the definition of done.



A note on "lift and shift"

Migrating a legacy workload to the cloud without changing its architecture is sometimes useful – typically to exit a data center on a hard deadline. It is not modernization. It moves the problem; it does not solve it. The methods in this paper assume the goal is to change the system, not merely its hosting.

Four methods of incremental modernization

The four methods below are the workhorses of incremental modernization. They are not novel – most have been documented in the engineering literature for over a decade – but they are persistently underused. Each section below covers what the method is, when to use it, how to apply it, and the failure modes to watch for.

3.1 The Strangler Fig Pattern

What it is

Named for the strangler fig tree, which grows around a host tree and eventually replaces it, the Strangler Fig pattern introduces a routing layer (typically an API gateway, reverse proxy, or facade service) in front of the legacy system. Initially, the facade routes all traffic to the legacy implementation. Over time, individual routes are redirected to new services that replace the corresponding legacy functionality. The legacy system shrinks until what remains can be retired.

When to use it

- **Monoliths with identifiable feature or domain boundaries** – particularly those exposing functionality through HTTP, message queues, or batch interfaces that can be intercepted.
- **Systems where the team has more confidence in the perimeter than the internals** – the facade lets the team modernize without first understanding every internal coupling.
- **Programs that need to demonstrate visible progress** – each route migrated is a deliverable, and the legacy estate visibly shrinks release by release.

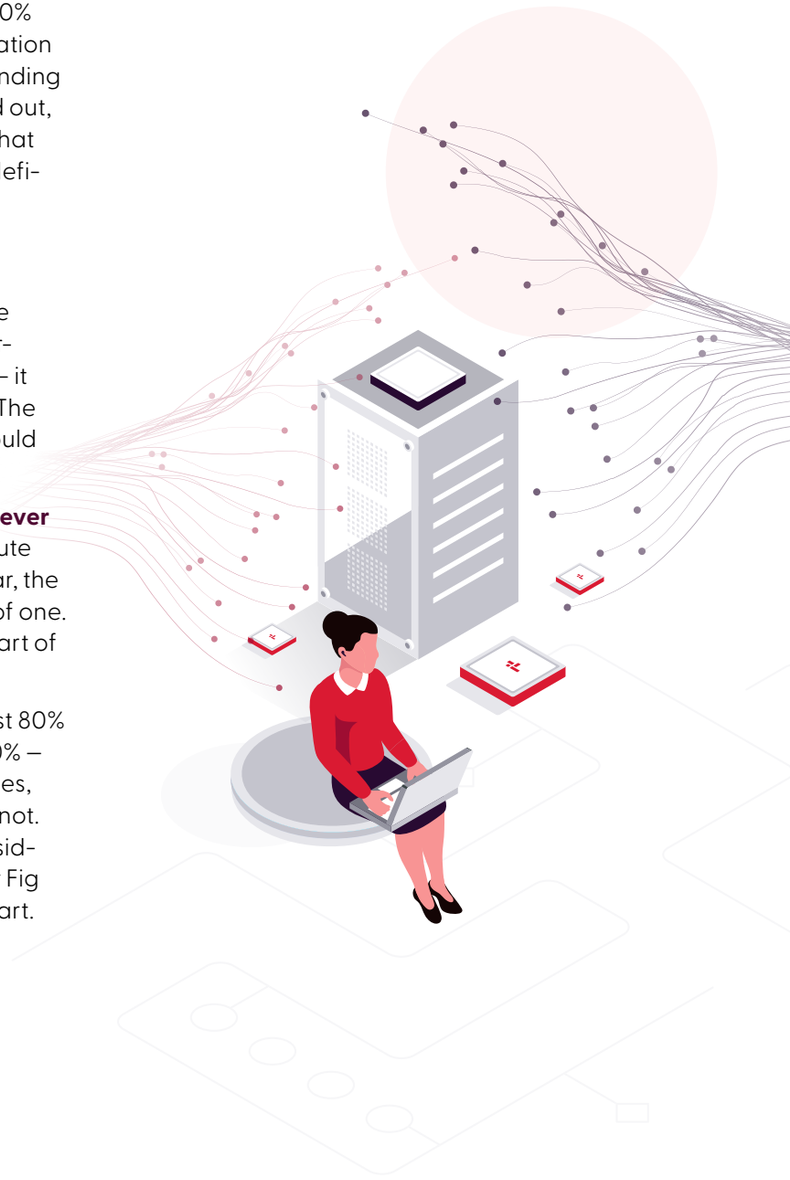
How to apply it

A Strangler Fig program typically proceeds through five stages:

- Stand up the facade in front of the legacy system. At this stage the facade is transparent – every request passes through to the legacy implementation. The facade is, however, instrumented: every route, payload, and response is observable.
- Identify the first capability to migrate. The best first candidates are read-heavy, low-risk, and bounded – a customer profile read, a catalog query, a static reference lookup. Avoid starting with capabilities that perform writes, calculations, or workflow orchestration.
- Build the replacement service for that capability. Build it in the target architecture, with the target observability, security, and deployment posture. Do not optimize for parity with the legacy; optimize for the architecture you want.
- Cut traffic over gradually. Route 1% of traffic to the new service, then 10%, then 50%, then 100%. At each stage, compare error rates, latency, and (where applicable) response equivalence to the legacy implementation.
- Remove the legacy code path. Once 100% of traffic has run on the new implementation for a defined soak period, the corresponding legacy code is deleted. Not commented out, not flagged off – deleted. This step is what distinguishes a Strangler Fig from an indefinite parallel deployment.

Failure modes to watch for

- **The facade becomes the monolith.** If the facade accumulates business logic – orchestration, validation, transformation – it becomes a second system to maintain. The facade should route and observe; it should not decide.
- **Routes are migrated but the legacy is never removed.** Teams move on to the next route before retiring the last one. Within a year, the organization runs two systems instead of one. Retirement of the legacy code path is part of the migration of each route.
- **Strangler stalls at the hard 20%.** The first 80% of routes are easy to migrate; the last 20% – typically batch jobs, scheduled processes, and tightly coupled internal logic – are not. Teams declare victory and leave the residual legacy running. A credible Strangler Fig program plans the hard 20% from the start.



3.2 Branch by Abstraction

What it is

Branch by Abstraction is a technique for replacing a deeply embedded subsystem – a persistence layer, a third-party integration, a calculation engine – without forking the codebase or stopping ongoing development. An abstraction layer (interface, port, or service contract) is introduced between the rest of the system and the subsystem being replaced. The legacy implementation sits behind that abstraction. A new implementation is built behind the same abstraction. Traffic is gradually migrated from old to new. When the new implementation is fully validated, the legacy implementation is removed.

When to use it

- **Replacing a subsystem that is called from many places in the codebase** – a database access layer, an authentication module, a payment integration, a search engine – where a forked branch would be unmaintainable.
- **Continuous delivery is non-negotiable** – the team needs to keep shipping features while the migration is in progress. Branch by Abstraction enables main-line development throughout.
- **The replacement is internal** – when the boundary being replaced is inside the system rather than at its perimeter, a Strangler Fig facade does not help. Branch by Abstraction operates at the code seam.

How to apply it

- Introduce the abstraction. Define an interface or contract that captures how the rest of the system interacts with the subsystem being replaced. The first version of this abstraction should be a direct extraction of how the subsystem is already used – do not redesign the contract yet.
- Route all callers through the abstraction. Refactor every caller in the codebase to depend on the abstraction rather than the concrete legacy implementation. This step is purely structural; behavior does not change.
- Build the new implementation behind the abstraction. The new implementation conforms to the same contract but uses the target technology, design, and architecture. Both implementations now coexist in the codebase.
- Switch implementations gradually, using feature flags. A configuration switch determines which implementation is used. Switching is reversible without code change.
- Remove the legacy implementation and, if appropriate, refine the abstraction. Once the new implementation is fully in production and the legacy is no longer needed, delete the legacy code. If the abstraction was over-fitted to the legacy's quirks, this is the time to refine it.

Failure modes to watch for

- **The abstraction leaks legacy semantics.** If the interface mirrors the legacy implementation's data shapes and error model exactly, the new implementation will be forced to emulate the legacy's quirks. The abstraction should reflect the domain, not the legacy.
- **The abstraction is over-designed before the new implementation exists.** Speculatively designing for a future implementation that has not been built often produces an abstraction that fits neither implementation well. Extract the abstraction from what exists; refine it once the second implementation is real.
- **Permanent dual maintenance.** Like Strangler Fig, the legacy implementation must eventually be removed. Teams that ship the new implementation behind a flag and never flip the flag end up paying for both systems indefinitely.

3.3 Anti-Corruption Layer

What it is

Originally articulated in Eric Evans' work on Domain-Driven Design, the Anti-Corruption Layer (ACL) is a translation and isolation layer between a new system and a legacy system it must integrate with. The ACL converts between the two systems' data models, terminology, and semantics, ensuring that the new system's domain model is not contaminated by the legacy system's accumulated baggage.

Where Strangler Fig and Branch by Abstraction are migration patterns – they help move functionality from old to new – the Anti-Corruption Layer is primarily a coexistence pattern. It enables a new system to integrate with a legacy system that may persist for some time, without inheriting its model.

When to use it

- **Building a new system that must read from or write to a legacy system of record** – for example, a new customer experience platform that depends on a legacy CRM or ERP for authoritative data.
- **The legacy data model is unsuitable for the new domain** – fields have been repurposed, status codes carry hidden meaning, relationships are implicit rather than enforced.
- **Multiple legacy systems must be reconciled** – the ACL becomes the integration point that resolves conflicting representations across legacy sources.
- **The legacy system will be replaced eventually, but not immediately** – the ACL gives the new system a stable contract while the legacy is migrated behind it.

How to apply it

- Model the new domain on its own terms. Define entities, relationships, and operations as they should exist in the target system, without reference to the legacy's structure. This is the most important step; compromise here defeats the purpose.
- Build the ACL as a dedicated service or module. The ACL exposes the new domain model to the new system and consumes the legacy system's interfaces (APIs, database, message queues, files) on the other side.
- Implement translation explicitly. Every mapping between legacy fields and domain attributes is documented, tested, and version-controlled. Implicit translations – "this field usually means X" – are where corruption leaks through.
- Cache and synchronize deliberately. If the ACL caches legacy data, the cache invalidation strategy is part of the design, not an afterthought. Synchronization patterns (event-driven, pull-on-demand, scheduled refresh) are explicit choices.
- Plan the ACL's retirement. If the legacy system is eventually retired, the ACL is no longer needed and should be removed. If the legacy persists indefinitely, the ACL is operationalized as a long-lived integration service with its own lifecycle, support model, and ownership.

Failure modes to watch for

- **The ACL becomes a thin pass-through**. If the ACL exposes the legacy's data shapes with cosmetic renaming, it has failed at its core purpose. The new domain model must be genuinely independent.
- **Translation logic accumulates business rules.** Over time, teams add validation, defaulting, and derivation logic into the ACL. This logic belongs in the domain services, not in the translation layer. An ACL bloated with business logic is harder to retire than the legacy it was meant to insulate from.
- **Performance degrades silently.** Translation, caching, and fan-out queries to the legacy can produce latency that is invisible until production load. Performance budgets and observability must be designed into the ACL from day one.

3.4 Parallel Run

What it is

In a Parallel Run, the legacy and new implementations execute simultaneously on production traffic. Both produce outputs; the legacy's output is used; the new implementation's output is captured and compared. Discrepancies are investigated and resolved. Only when the new implementation has demonstrated equivalence over a sustained period – and through a representative range of conditions – is its output trusted and the legacy retired.

Parallel Run is not a replacement for the other three methods; it is a verification pattern that can be layered on top of any of them. It is most valuable when the cost of a wrong answer is high and the inputs are complex enough that test-based verification is insufficient.

When to use it

- **Financial calculation engines** – pricing, billing, risk, settlement, interest accrual, fee assessment. Any system whose output is money or a representation of money.
- **Regulated systems with audit obligations** – where a discrepancy between systems would create regulatory exposure.
- **Algorithmic decisioning** – fraud detection, credit decisioning, eligibility determination – where the new system must demonstrate consistent behavior over a long tail of cases.

- **Migrations where test coverage cannot capture the full input space** – production traffic is the only realistic test set, and equivalence must be demonstrated empirically.

How to apply it

- Capture and replay, or shadow execution. Either intercept production inputs and replay them against both systems offline, or execute both systems against live traffic in parallel. Shadow execution is more accurate but more operationally complex; capture-and-replay is simpler but may miss timing-dependent behavior.
- Compare outputs structurally, not textually. Define equivalence semantically: which fields must match exactly, which may differ within tolerance, which are expected to differ. Naive byte-level comparison produces noise that quickly erodes confidence in the process.
- Triage discrepancies systematically. Every discrepancy is one of three things: a bug in the new system, a bug in the legacy that the new system corrects, or a legitimate semantic difference. The triage process – and who has authority to classify – must be defined before the parallel run begins.
- Define exit criteria up front. How long must the parallel run continue? What discrepancy rate is acceptable before cutover? What conditions (peak load, month-end, regulatory reporting cycles) must be covered? Vague exit criteria produce parallel runs that never end.
- Cut over decisively. When exit criteria are met, the new system becomes authoritative and the legacy execution is removed. Continuing a parallel run beyond its purpose drains engineering capacity without adding confidence.

Failure modes to watch for

- **Discrepancy fatigue.** If the first weeks of parallel run produce hundreds of unexplained differences, the team's appetite for triage collapses. Mitigation: start parallel run only after the new system has passed structural and unit-level equivalence testing, so production discrepancies are the long tail, not the bulk.
- **The legacy is treated as ground truth even when wrong.** Years of accumulated production output have an aura of correctness that does not always survive scrutiny. A parallel run will surface legacy bugs. The governance process must allow the legacy to be ruled wrong, with appropriate stakeholder agreement.
- **Infrastructure cost.** Running two implementations against production traffic doubles compute, doubles data access, and adds the comparison cost. Budget and ownership of this cost must be agreed before the run starts.

Years of accumulated production output have an aura of correctness that does not always survive scrutiny. A parallel run will surface legacy bugs. The governance process must allow the legacy to be ruled wrong, with appropriate stakeholder agreement.

Selecting the right method

The four methods are complementary, not competing. Most non-trivial modernization programs use at least two of them, and large programs typically use all four in different parts of the estate. The table below summarizes how to select between methods. The first decision is rarely about the method; it is about the shape of the problem.

Method	Best fit	Primary risk	Combines well with
Strangler Fig	Monolith with intercepted perimeter; visible-progress programs; vertical-slice migrations.	Stalling at the last 20%; facade accumulating business logic.	Anti-Corruption Layer behind the facade; Parallel Run for high-stakes routes.
Branch by Abstraction	Replacing an embedded subsystem (DB, integration, auth) without forking the codebase.	Leaky abstraction; permanent flagged dual maintenance.	Parallel Run for the new implementation; can sit inside a Strangler-Fig service.
Anti-Corruption Layer	New system must coexist with a legacy system of record; bridging conflicting models.	Becoming a pass-through; absorbing business logic; silent performance degradation.	Strangler Fig (ACL behind the facade); long-term coexistence patterns.
Parallel Run	Financial, regulated, or algorithmic systems where equivalence must be proven empirically.	Discrepancy fatigue; cost overhead; indefinite duration with no decisive cutover.	Layered on top of any other method as a verification gate before cutover.

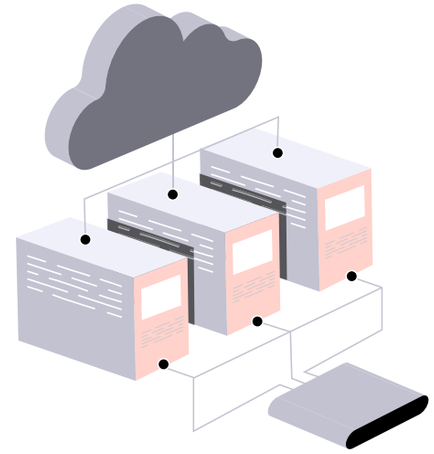
The single most predictive question

When VRIZE assesses modernization programs that are in trouble, the most predictive question is: "What will be deleted, and when?" Programs that cannot answer this question precisely tend not to retire legacy. Programs that can – and that hold themselves to the deletion dates – succeed.

A decision sequence for program leadership

VRIZE recommends program teams work through the following decision sequence when scoping a modernization initiative:

1. **Identify the unit of migration.** What is the smallest thing that can be migrated end-to-end? If the answer is "a route or a capability," Strangler Fig is the operating frame. If the answer is "a subsystem within the codebase," Branch by Abstraction is the operating frame.
2. **Identify the data and semantic coupling.** Will the new system depend on data still owned by the legacy? If yes, an Anti-Corruption Layer is required, regardless of which migration pattern frames the program.
3. **Identify the correctness threshold.** Is the functionality being migrated calculation-heavy, financially material, or regulated? If yes, a Parallel Run is required before cutover, regardless of which migration pattern frames the program.
4. **Plan the retirement explicitly.** For every legacy component touched by the program, the migration plan should specify what "retired" means, who is accountable for the retirement, and the deadline by which retirement is complete.



The VRIZE operating model for incremental modernization

Why retirement accounting matters

The economic case for modernization rests on retiring legacy cost — license, infrastructure, support, opportunity cost of engineering attention spent on legacy. Programs that migrate without retiring deliver the cost of modernization without the benefit. Retirement accounting is the mechanism that prevents this outcome.

Methods alone do not deliver modernization. What separates programs that finish from programs that stall is the operating discipline around the methods. VRIZE applies a consistent operating model across delivery engagements, built on five components.

Modernization slices and the weekly delivery rhythm

Each modernization program is decomposed into slices — vertical capability migrations sized to complete within a single sprint. A slice has a defined entry condition (legacy behavior characterized, observability in place), a defined exit condition (traffic cut over, legacy code path removed, retirement logged), and a named accountable engineer. Weekly delivery reviews track slices by status, not by story points.

Pre-migration discovery and characterization

Before any code is written, the slice is characterized: inputs, outputs, dependencies, traffic profile, edge cases captured from production logs, and the equivalence criteria against which the new implementation will be measured. VRIZE's discovery artifacts are reusable assets that compound across slices.

Observability-first delivery

Observability is not added to the new implementation after release; it is the first thing built. Every new service ships with structured logging, distributed tracing, request/response capture for comparison, business-level metrics, and dashboards that compare old and new side by side. If observability is not in place, the slice cannot proceed to traffic migration.

Engineering quality gates

VRIZE enforces engineering quality standards as non-negotiable gates: automated test coverage, contract tests at integration boundaries, security scanning, performance benchmarks, and documented rollback procedures. These gates apply to modernization slices identically to greenfield work. GenAI-assisted engineering is mandated as part of the toolchain to maintain delivery velocity at the required quality bar.

Retirement accounting

Every legacy component touched by the program is tracked on a retirement ledger: its current state (live, dual-running, traffic-migrated, code-removed, infrastructure-decommissioned), its retirement deadline, and the named owner. The ledger is reviewed at the same cadence as the delivery rhythm. A legacy component that misses two retirement deadlines becomes a program-level escalation.

VRIZE enforces engineering quality standards as non-negotiable gates: automated test coverage, contract tests at integration boundaries, security scanning, performance benchmarks, and documented rollback procedures.

Common pitfalls and how to avoid them

Across the modernization programs VRIZE has assessed and delivered, the same handful of failure patterns recur. Naming them is the first step to avoiding them.

Pitfall 1:

Treating modernization as an IT project rather than a business program

Modernization programs that are scoped, funded, and governed entirely within IT tend to deliver technical outcomes without business outcomes. Slice selection should be driven by business value – which customer journeys, which operational processes, which regulatory exposures matter most – not by technical convenience. A business sponsor with decision authority over slice prioritization is non-negotiable.

Pitfall 2:

Underinvesting in the legacy understanding

Teams routinely underestimate how much undocumented behavior the legacy system carries. The countermeasure is dedicated discovery investment per slice – at least 15% of slice effort, and often more for slices touching calculation-heavy or workflow-heavy functionality. Discovery findings are durable assets and should be captured in a form that outlasts the program.

Pitfall 3:

Allowing the new system to drift from its target architecture

Under delivery pressure, teams take shortcuts: a quick database integration to the legacy schema, a calculation lifted verbatim, an authorization model inherited rather than redesigned. Each shortcut creates new technical debt in the system that was supposed to retire technical debt. Architectural review at the slice level – not just at the program level – is the countermeasure.

Pitfall 4:

No clear cutover authority

Decisions about when to migrate the next percentage of traffic, when to remove the legacy code path, and when to declare a slice retired must rest with named individuals empowered to make them. Diffuse decision rights produce indefinite parallel running. VRIZE recommends a single Migration Decision Owner per slice and a Program Cutover Authority for cross-slice decisions.

Pitfall 5:

Treating modernization as a one-time event

The same forces that produced the current legacy will, if unchecked, produce the next legacy on top of the modernized system. Modernization programs should leave behind durable engineering practices – observability, automated testing, retirement accounting – that make the next decade's modernization continuous rather than episodic.

> **Successful modernization requires more than technology upgrades. Organizations must align business and IT priorities, understand legacy systems, maintain architectural discipline, establish clear decision ownership, and adopt lasting engineering practices.**

Conclusion

VRIZE perspective

Modernization is not a project. It is an operating capability. The methods in this paper, applied with discipline, turn legacy modernization from a recurring crisis into a continuous engineering practice.

Incremental modernization is not a technique; it is a stance. It accepts that the legacy system has value, that the business cannot stop while modernization happens, that the team will discover things during migration that no amount of upfront analysis would have surfaced, and that the only credible mitigation for these realities is to keep the unit of change small and the path of reversibility short.

The four methods covered in this paper – Strangler Fig, Branch by Abstraction, Anti-Corruption Layer, and Parallel Run – are the building blocks. Used together, with the operating discipline of slice-based delivery, observability-first engineering, and retirement accounting, they offer a path to modernization that does not depend on heroic execution of a multi-year program.

The organizations that modernize successfully over the next five years will not be the ones with the most ambitious target architectures. They will be the ones that ship the smallest credible slice this month, retire the corresponding legacy component, and repeat – sprint after sprint, until what remains of the legacy is small enough to retire entirely.

> **Incremental modernization succeeds through small, manageable changes that reduce risk and maintain business continuity. By combining proven modernization methods with disciplined execution, organizations can deliver steady progress, retire legacy components incrementally, and achieve lasting transformation without relying on large-scale, high-risk programs.**

Author

Naharaajan J
Chief Delivery Officer

Incremental Legacy Modernization

Founded in 2020, VRIZE unites a team of 450+ industry professionals, all geared towards crafting frictionless digital experiences. With specializations in experiential commerce and data science, our global reputation is anchored by innovation and strategic acumen. Driven by the core tenets of customer centricity, ownership, agility, integrity, and respect, VRIZE stands as a benchmark in industry excellence. Explore more on **LinkedIn**.

www.vrize.com