

T202 TRANSFORM & OPTIMIZE · PRACTITIONER

Modernizing Legacy Systems with AI

Recovering knowledge from legacy code, choosing the right strategy, and demonstrating equivalence before cutover.

FORMAT

Knowledge pack · 13 slides

FOR

Architects and engineering leaders

LEVEL

Practitioner



THE PREMISE

Modernization depends on recovering undocumented knowledge

Legacy systems encode decades of rules, exceptions and workarounds, many never written down. AI lowers the cost of **recovering that knowledge**. It does not remove the need to verify it.

Recover

Business rules and dependencies, from code and runtime data.

Decide

The right strategy for each application.

Verify

Equivalence before every cutover.

STRATEGY

Seven strategies, chosen application by application

The seven Rs in AWS migration guidance. For large migrations, AWS recommends migrating first and modernizing afterward.

STRATEGY	WHAT CHANGES	CHOOSE IT WHEN
Retire	The application is switched off	The capability is unused or duplicated
Retain	Nothing, for now	Change is not yet justified
Rehost	Infrastructure only	Speed matters and the code is sound
Relocate	Hosting location only; same platform	Bulk moves of virtualized environments
Repurchase	Replaced by a SaaS or packaged product	The capability does not differentiate
Replatform	Selected components, such as the database	A managed service cuts operating effort
Refactor or re-architect	Code and architecture	The capability differentiates and must evolve

AI IN MODERNIZATION

Where AI helps today, and where it needs rigorous verification

HELPS TODAY

- ◆ Explaining unfamiliar code in plain language
- ◆ Extracting business rules into reviewable specifications
- ◆ Mapping call graphs, data access and dependencies
- ◆ Generating characterization tests from observed behavior
- ◆ Translating well-bounded, well-tested modules

VERIFY RIGOROUSLY

- ◆ EBCDIC collating order in comparisons, key ranges and sorts
- ◆ Decimal precision: packed-decimal fields, truncation unless ROUNDED
- ◆ Transaction, locking and concurrency semantics
- ◆ Behavior defined outside the code, in JCL, job schedulers and CICS
- ◆ Batch checkpoint, restart and error-handling logic

VERIFICATION

Verify each AI output before it reaches a parallel run

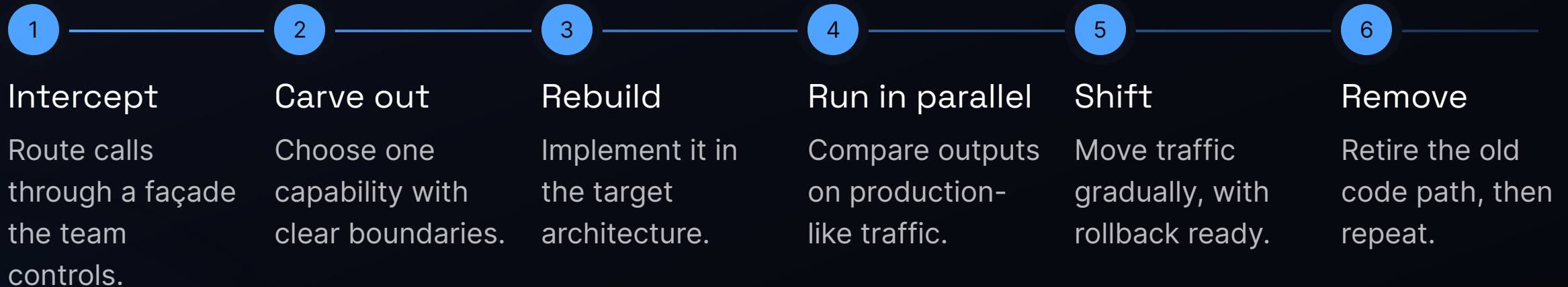
Researchers testing IBM’s COBOL-to-Java assistant write that LLM-translated code “cannot be trusted to correctly translate the original code.”

AI OUTPUT	VERIFIED BY	ACCEPTED WHEN
Extracted rules	Expert review of each rule against the code it cites	A named owner has accepted or rejected every rule
Dependency maps	Comparison with scheduler definitions and runtime records such as SMF	Every unmatched dependency is explained
Generated tests	Inputs and expected outputs captured from the legacy system, not predicted by the model	An agreed branch-coverage target is met; gaps are listed as risks
Translated code	Unit tests derived from the legacy code, static analysis and expert review	Tests pass and review findings are closed

PATTERN

Migrate incrementally with the strangler fig

Described by Martin Fowler in 2004 and revised in 2024: grow the new system around the old one until the old one can be removed.



EQUIVALENCE

Demonstrate equivalence before every cutover

- ✓ **Characterization tests**
Capture current behavior, defects included, before changing anything.
- ✓ **Reconciliation**
Compare batch and financial outputs run by run, to the last digit.
- ✓ **Agreed tolerances**
Business owners decide which differences are acceptable.
- ✓ **Parallel runs**
Run old and new on the same inputs and compare the outputs.
- ✓ **Shadow traffic**
Mirror live requests to the new system; discard its responses and block side effects.
- ✓ **Rollback**
A tested path back that also returns data written after cutover.

TECHNICAL DEBT

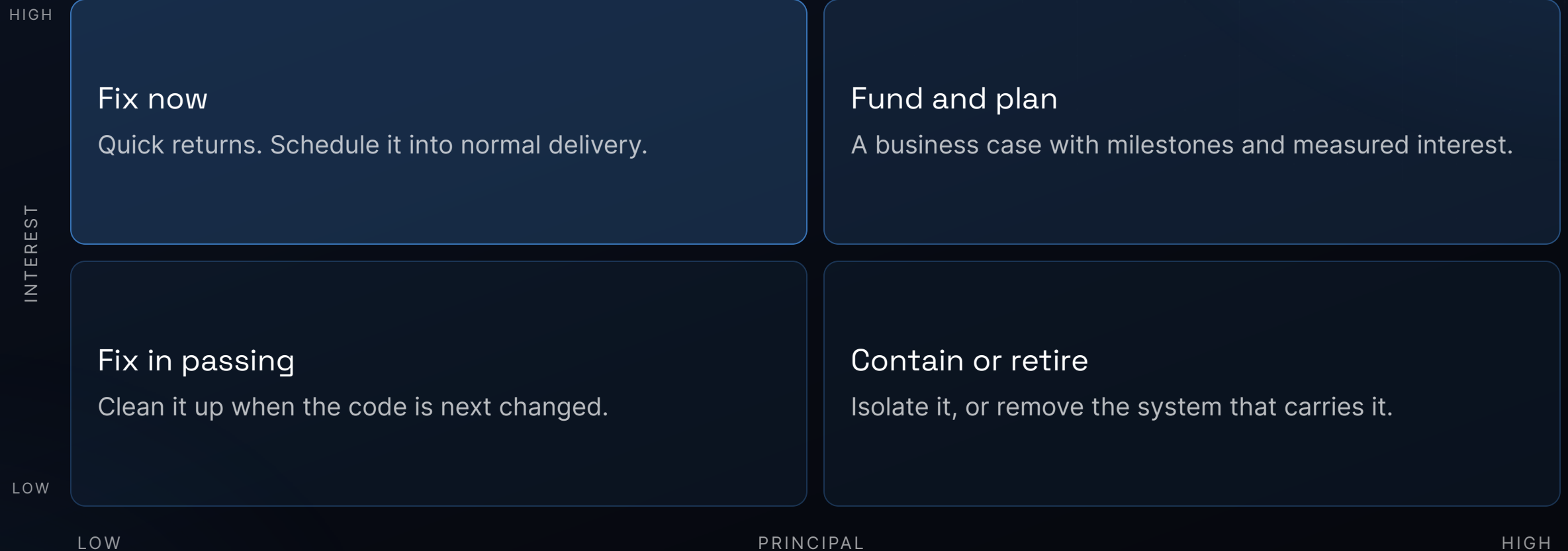
Price technical debt as principal and interest

Ward Cunningham coined the debt metaphor in 1992. In Martin Fowler’s elaboration, principal is the cost to fix and interest is the recurring cost of not fixing, such as slower change.

DEBT	SIGNAL	INTEREST PAID AS
Code	Complexity in frequently changed code, low test coverage	Slower changes, more defects
Architecture	Tight coupling, shared databases	Wider blast radius, coordination cost
Infrastructure	Unsupported operating systems and runtimes	Extended support fees, security exposure
Dependencies	Outdated libraries and frameworks	Vulnerability backlog, forced major upgrades
Knowledge	Few people understand the system	Key-person risk, slow onboarding
Data	Inconsistent definitions and copies	Reconciliation effort, reporting errors

PRIORITIES

Pay down debt where the interest is highest



DATA

Data migration is often the critical path

Profile early

Measure quality before design decisions depend on it.

Map meaning

Fields with the same name often mean different things.

Cleanse at source

Fix defects in the source system before migration.

Reconcile

Check record counts, control totals and balances after every load.

Plan the cutover

Windows, freeze periods and delta loads.

Archive history

Retain per policy and legal holds, accessible outside the new system.

PITFALLS

Why modernization programs overrun

✗ Big-bang rewrites

✓ Incremental replacement with parallel runs

✗ Line-by-line translation

✓ Redesign where the old structure no longer fits

✗ Trusting generated code without tests

✓ Characterization tests before any change

✗ Modernizing what should be retired

✓ Rationalize first, then modernize

✗ Forgetting batch jobs and reports

✓ Map scheduled jobs and downstream reports early

✗ No business owner for differences

✓ Name who accepts or rejects each deviation

GETTING STARTED

An indicative first modernization wave

WEEKS 1–4

Understand

- ◆ Choose a strategy for each application
- ◆ Recover rules and dependencies with AI, reviewed by experts
- ◆ Write characterization tests

WEEKS 5–12

Prove

- ◆ Carve out one capability behind a façade
- ◆ Run old and new in parallel
- ◆ Agree tolerances with business owners

WEEKS 13–24

Scale

- ◆ Shift traffic and retire the old path
- ◆ Repeat for the next capabilities
- ◆ Report debt and cost removed

SUMMARY

Key takeaways

- 01 Recover the knowledge in legacy code and verify it with tests.
- 02 Choose among the seven strategies per application, and retire before you modernize.
- 03 Use AI to explain, extract, test and translate, and verify every output against the legacy system.
- 04 Migrate incrementally, and demonstrate equivalence before every cutover.
- 05 Price technical debt as principal and interest, and fix where interest is highest.

NEXT COURSE · S201

AI Governance in Practice

Secure & Govern · Practitioner