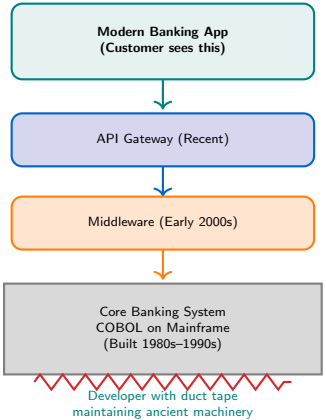
Why are the world's biggest banks running on technology from decades ago?

The tension we will explore:

- Banks depend on core systems built in the seventies and eighties, written in languages most developers no longer learn
- These systems process trillions in transactions daily with extraordinary reliability
- Replacing them risks catastrophic failure — but keeping them risks slow irrelevance

This dilemma defines core banking modernization:

- Legacy systems are too reliable to abandon carelessly
- Yet technical debt compounds until innovation budgets vanish
- Migration takes years, costs billions, and offers no guarantee of success
- The question is not whether to modernize, but how to do it without destroying the bank

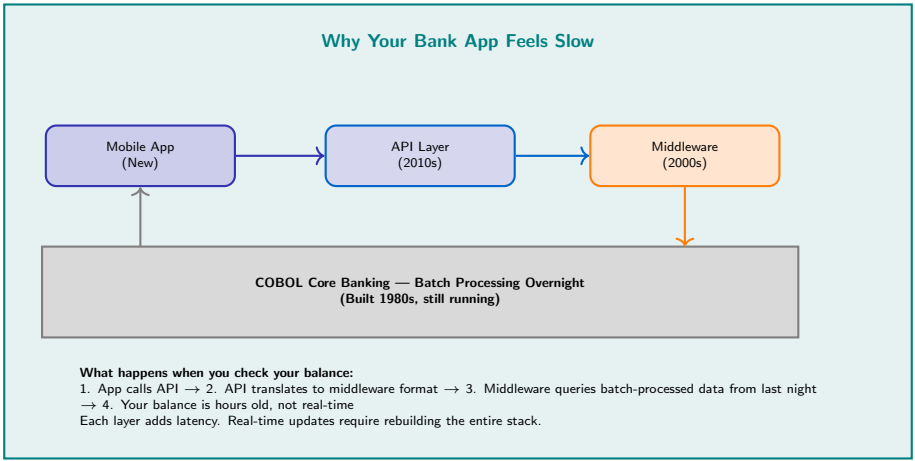


Key Insight

The shiny app is a facade — beneath it runs code older than most developers, maintained by a shrinking pool of experts approaching retirement.

Core banking systems are invisible to customers but essential to operations — and most are decades old, reliable, and nearly impossible to replace safely.

Have you ever been frustrated by a banking app that felt stuck in the past?



Takeaway

The frustration you feel is technical debt made visible — a modern interface layered over a system designed for overnight batch processing.

What are the main architectural approaches to core banking systems?

Three architectural paradigms: 1. Monolithic:

- Single codebase, shared database
- Change one module, redeploy everything
- Simple to reason about but rigid

2. Modular:

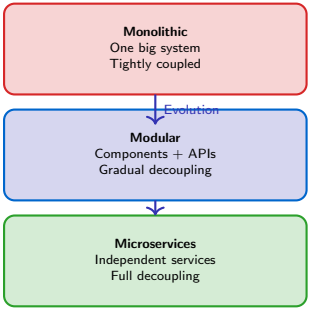
- Components communicate via APIs
- Can replace pieces independently
- Hybrid approach (wrap legacy, add new modules)

3. Microservices:

- Each service owns its data and deploys independently
- Loose coupling, high agility
- Complexity shifts to orchestration and data consistency

Choosing the right architecture depends on:

- Bank size and transaction volume
- Team maturity and operational capability
- Risk tolerance for distributed systems



Key Insight

No architecture is universally superior — monoliths are simple but rigid, microservices are agile but complex, modular approaches balance both.

How does a bank migrate from a legacy core to a modern platform without downtime?

The strangler fig migration pattern: Step 1: Identify a bounded context

- Choose a self-contained module (e.g., savings accounts)
- Define clear boundaries with the rest of the system

Step 2: Build the new service alongside the old

- Develop and test in parallel, no disruption to production
- Use a facade layer to route traffic selectively

Step 3: Route new traffic to the new service

- Start with a small percentage (canary deployment)
- Gradually increase until all new traffic uses the new system

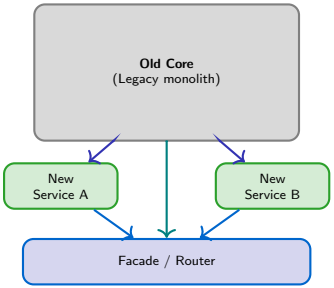
Step 4: Migrate existing data gradually

- Move accounts in batches, verify consistency
- Run both systems in parallel during migration

Step 5: Decommission the old component

- Only after all data migrated and verified
- Repeat for the next bounded context

Strangler Fig Pattern



Old shrinks as
new services grow

Key Insight

The strangler fig pattern allows continuous operation during migration — old and new coexist, traffic shifts gradually, and rollback is always possible.

Named after a fig vine that gradually replaces a host tree, this pattern is the industry standard for core banking modernization.

How do monolithic and modular core banking architectures compare?

Monolithic architecture:

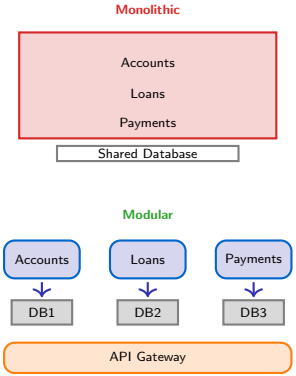
- Single deployable unit, shared database
- Change one line, redeploy everything
- Simple to debug (single process, single log)
- One bug can bring down the entire system

Modular architecture:

- Independent services with clear APIs
- Deploy and scale components separately
- Fault isolation (one service fails, others continue)
- Complexity shifts to inter-service communication and data consistency

Trade-offs:

Aspect	Monolith	Modular
Deployment	Complex	Independent
Debugging	Easier	Harder
Scaling	Vertical	Horizontal
Team size	Small–medium	Large



Key Insight

Monoliths are simpler for small teams but become bottlenecks at scale; modular architectures enable agility but demand operational maturity.

The choice between monolithic and modular is not about which is better, but which matches the bank's size, team capability, and change velocity.

What happens when a core banking migration goes wrong in the middle?

Failure modes during migration: **Data inconsistency:**

- Old and new systems show different balances for the same account
- Customers lose trust immediately
- Regulatory breach if financial records do not reconcile

Partial functionality:

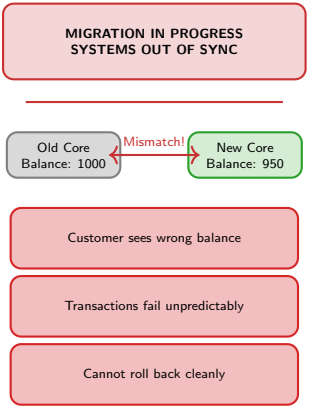
- Some transactions work, others fail unpredictably
- Customer service cannot explain failures
- Manual workarounds flood operations teams

Rollback impossible:

- If data migrated one-way, cannot revert cleanly
- Stuck in a half-migrated state
- Emergency fixes compound technical debt

Historical example:

- TSB Bank migration failure: millions locked out, wrong balances displayed, CEO resigned, hundreds of millions in costs



Key Insight

A failed migration in progress is worse than staying on the old system — customers lose trust, regulators intervene, and recovery costs spiral.

Core banking migration failures are rare but catastrophic — explaining why banks spend years planning and testing before cutover.

Where are banks at different stages of core modernization across the world?

Global core banking modernization landscape: Still fully legacy:

- Many regional and community banks
- COBOL on mainframes, minimal API layer
- High maintenance costs, slow product launches

Hybrid (paving over):

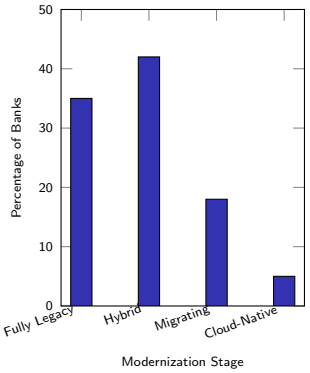
- Most large incumbent banks
- Legacy core wrapped with API gateways and middleware
- New channels modern, core unchanged

Active migration:

- Mid-size banks using strangler fig pattern
- Three to seven year programs underway
- Cloud-native vendors (Thought Machine, Mambu) gaining traction

Cloud-native from day one:

- Neobanks and digital-only challengers
- Built on modern stacks, no legacy burden
- Faster innovation but unproven at scale



Key Insight

Most banks are in the hybrid stage — legacy cores wrapped with modern interfaces, planning multi-year migrations but not yet executing.

The distribution reflects risk aversion and capital constraints — full modernization is expensive and risky, so most banks delay or incrementalize

Who wins and who loses when a bank modernizes its core – and when it does not?

Winners from successful modernization:

- Customers: faster products, better experiences
- Bank: lower maintenance costs, faster time-to-market
- Developers: modern tools, easier to hire and retain
- Regulators: better reporting, more transparency

Losers from successful modernization:

- Legacy vendors: lose recurring revenue
- Incumbent IT staff: skills become obsolete
- Short-term shareholders: years of capex before ROI

Winners from NOT modernizing:

- Short-term profit: no capex, no disruption
- Legacy vendors: lock-in continues

Losers from NOT modernizing:

- Long-term competitiveness: cannot launch products fast enough
- Customers: stuck with outdated experiences
- Developers: talent leaves for modern stacks

Stakeholder Analysis

Modernize Successfully:

Win: Customers, Bank, Devs
Lose: Legacy vendors, Old skills

Stay Legacy:

Win: Short-term profit, Vendors
Lose: Competitiveness, Talent

Migrate and Fail:

Lose: Everyone
Costs spiral, trust destroyed,
regulators intervene

Key Insight

Three questions to evaluate a core banking modernization strategy

The Migration Readiness Assessment:

Question 1: Can you run old and new in parallel with verified consistency?

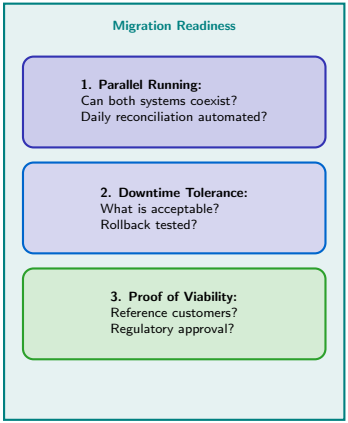
- Dual-running is mandatory for safe migration
- Every balance, every transaction must reconcile daily
- Automated comparison, not manual checks

Question 2: What is the maximum acceptable downtime?

- Zero downtime is ideal but rare
- Define what constitutes an acceptable maintenance window
- Have rollback procedures tested and ready

Question 3: Is the new system proven or experimental?

- Reference customers at comparable scale
- Regulatory approvals in your jurisdiction
- Vendor viability over ten to twenty year horizon



These three questions separate realistic migration plans from wishful thinking — if you cannot answer all three confidently, delay the cutover.

Your Challenge

Scenario: You are advising a mid-size bank with two million customers and thirty years of COBOL core banking history. The board has approved a modernization budget and asked you to evaluate three options: rebuild in-house, buy a commercial platform, or wrap the legacy core with APIs and defer full replacement.

Your task: Evaluate each option using the three questions from slide nine. For each option, assess whether it satisfies the migration readiness criteria.

Consider for each option:

- **Rebuild:** Can you run old and new in parallel? What is the timeline and risk?
- **Buy:** Is the vendor platform proven at your scale? What about data migration and customization?
- **Wrap:** Does wrapping improve agility or just delay the inevitable? How long can you defer replacement?

Deliverable: A one-page comparison table scoring each option on the three criteria (parallel running, downtime tolerance, proof of viability). Recommend one approach and justify your choice.

Reflection

This is the exact decision facing hundreds of banks worldwide — no option is perfect, and the board will demand a recommendation despite uncertainty.

Real strategic decisions in banking are made under uncertainty with incomplete information — exactly this exercise.