Lesson 6.2: Core Banking and Legacy Systems

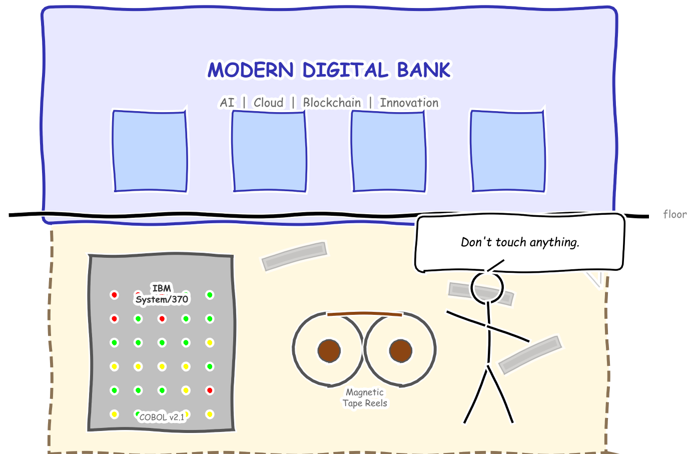
Module 6: The Infrastructure Problem

Prof. Dr. Joerg Osterrieder

May 15, 2026

Core Banking: A Candid Look

Bank Modernization: Expectations vs. Reality



"We put a chatbot on top of COBOL and called it digital transformation."

Learning Objectives

By the end of this lesson, you will be able to:

1. Describe the components of a core banking system and explain the role of the general ledger [Understand]
2. Identify the causes and consequences of technical debt in financial institutions [Understand]
3. Compare migration strategies (big bang, strangler fig, parallel run) and evaluate their risk profiles [Evaluate]
4. Explain the strangler fig pattern and why it dominates modern bank modernization [Understand]
5. Evaluate cloud migration trade-offs specific to financial services (latency, regulation, cost) [Evaluate]
6. Contrast monolithic and microservice architectures in the context of banking [Analyze]
7. Distinguish data warehouse, data lake, and data mesh approaches for financial data [Analyze]
8. Describe event-driven architecture and its advantages for real-time banking [Understand]

Bloom's levels: Understand (1,2,4,8), Analyze (6,7), Evaluate (3,5). Modernization journey: legacy to replacement.

(c) Joerg Osterrieder 2025–2026

Bridge: From Payment Rails to Bank Internals

Where we have been (Lesson 6.1):

- We traced the payment rails that connect institutions
- SWIFT, SEPA, real-time payment networks
- ISO 20022 messaging standards
- Settlement and clearing infrastructure

Where we are going:

- We see the payment rails. Now: **what is inside the banks that connect to them?**
- Core banking systems: the "engine room" of every financial institution
- Why modernizing these systems is a $300+ billion global effort

Lesson 6.1 explored the roads. Lesson 6.2 opens the hood of the vehicles that drive on them.

Every payment, every account balance, every interest calculation flows through a core banking system. Understanding these systems is essential to understanding modern banking.

What Is Core Banking? — Definition and Functions

Definition:

- The central software that processes a bank's daily transactions and posts updates to accounts and financial records
- "Core" refers to the central ledger of record — every account balance, posting, and settlement is anchored here
- Every account balance, interest calculation, and transaction flows through this system

Core functions:

- Account management (open, close, modify)
- Transaction processing (debits, credits, transfers)
- Interest and fee calculation
- Regulatory reporting and compliance

A core banking system is to a bank what an operating system is to a computer: invisible to customers, but nothing works without it.

What Is Core Banking? — Vendors and Market

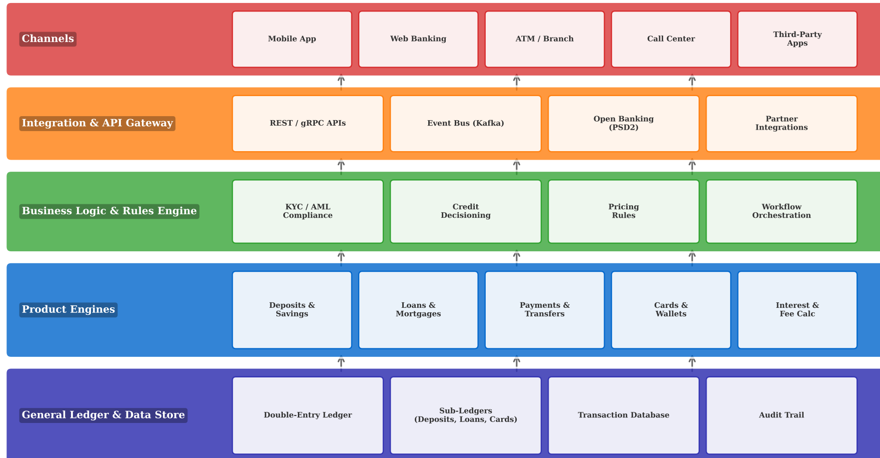
Who builds core banking systems?

- **Legacy vendors:** FIS (Systematics), Fiserv (Signature), Temenos (T24), Finastra
- **Cloud-native challengers:** Thought Machine (Vault), Mambu, 10x Banking, Finxact
- **In-house:** JP Morgan, Goldman Sachs build proprietary systems

Scale:

- FIS serves 50+ of the world's top 100 banks
- Temenos: 3,000+ institutions, 1.2 billion end users
- Global core banking market: $18B (2024), growing 10%+ annually

The vendor choice is one of the most consequential technology decisions a bank makes. Legacy vendors offer stability; cloud-native challengers offer agility.



(c) Joerg Osterrieder 2025–2026

The General Ledger: Foundation of All Banking

What is a general ledger?

- The authoritative record of every financial transaction
- Double-entry bookkeeping: every debit has a matching credit
- **Assets = Liabilities + Equity** must hold at all times
- If the general ledger is wrong, the bank is wrong

Why it matters for technology:

- Sub-ledgers (deposits, loans, trading) feed into the GL
- End-of-day vs. real-time posting is a key architectural choice
- Reconciliation between sub-ledgers and GL is a daily operational risk

Double-entry example (customer deposit):

Account	Debit	Credit
Cash (Asset)	$1,000	
Customer Deposit (Liability)		$1,000

Modern GL challenges:

- Real-time balances vs. batch posting
- Multi-currency, multi-entity consolidation
- Regulatory reporting (International Financial Reporting Standard 9 (IFRS 9), Basel III)
- Audit trail and immutability requirements

The general ledger is the single source of truth for a bank's financial position. Every modernization project must answer: "How do we migrate the GL without losing a single cent?"

Why Do Banks Still Run 60-Year-Old Code?

The COBOL reality:

- COBOL (1959) remains the dominant language for back-end mainframe transaction processing at most top-50 global banks — ≈43% of banking systems (*Reuters*, 2017)
- 220+ billion lines of COBOL in production today
- ATM terminals themselves usually run Windows or Linux; the COBOL runs downstream on the authorization host (IBM z/OS)
- Average age of a bank's core system: 25–35 years

Why banks still run COBOL:

- Extremely reliable: mainframes achieve 99.999% uptime
- Highly optimized for batch transaction processing
- Regulatory-compliant and audited for decades
- "If it ain't broke, don't fix it" mentality

COBOL is not dead—it processes $3 trillion in daily commerce. The language itself is not the problem; its reliability is precisely why banks kept it running.

COBOL and Mainframes — The Looming Crisis

The demographic time bomb:

- ∼47% of COBOL developers are 45+ and 15% are over 55 (*Micro Focus COBOL Market Survey*, 2022) — skills concentration is a recognized risk
- Universities stopped teaching COBOL decades ago, though bootcamps and reskilling programs (IBM, Micro Focus) are rebuilding the pipeline
- Knowledge locked in undocumented code and retiring experts
- During COVID-19 (2020), US states scrambled to find COBOL programmers to fix unemployment systems

Mainframe economics:

- MIPS (Million Instructions Per Second) pricing
- License costs: $1–3M per year for large banks
- Hardware refresh cycles: 5–7 years
- Vendor lock-in: switching costs enormous

The problem is not the language; it is the shrinking pool of people who understand it. When knowledge walks out the door at retirement, it does not come back.

(c) Joerg Osterrieder 2025–2026

Why Does Maintenance Eat 80% of IT Budgets?

Definition:

- The implied cost of rework caused by choosing a quick, easy solution now instead of a better approach that would take longer
- Like financial debt: you pay "interest" in the form of slower development, more bugs, and higher maintenance costs

Sources in banking:

- Decades of patches on top of patches
- Mergers: bolting together incompatible systems
- Regulatory changes requiring quick fixes
- Point-to-point integrations (spaghetti architecture)
- Undocumented business logic in code

Ward Cunningham coined "technical debt" in 1992. In banking, the compound interest on this debt has reached crisis levels—some institutions spend $1B+ annually just to keep legacy systems running.

Technical Debt — Impact and the "Paving Over" Pattern

Measurable impact:

- Banks spend 70–80% of IT budgets on maintenance (Business As Usual (BAU), "run the bank") vs. innovation ("change the bank")
- Average time to launch a new product: 12–18 months (legacy) vs. 2–4 weeks (cloud-native)
- Integration costs: $50–200M for a typical merger

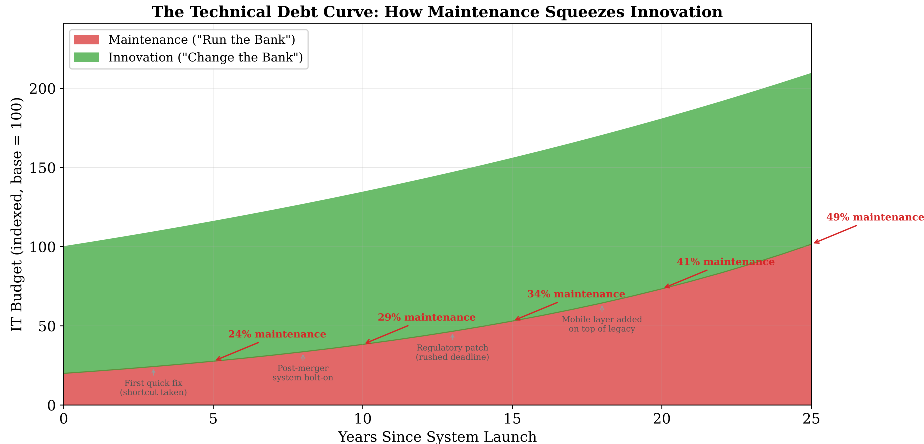
The "paving over" pattern:

- Layer 1: COBOL batch system (1980s)
- Layer 2: Java middleware (2000s)
- Layer 3: REST APIs over middleware (2010s)
- Layer 4: Mobile app calling APIs (2020s)
- Result: 4 layers of abstraction, one fragile stack

The "paving over" pattern is the default outcome when organizations avoid rewriting systems. Each layer adds latency, complexity, and fragility.

(c) Joerg Osterrieder 2025–2026

The Technical Debt Curve



- **What you see:** Stacked area chart over 25 years showing maintenance cost (red) growing exponentially while innovation budget (green) shrinks. By year 20, 85%+ of budget goes to maintenance
- **Key pattern:** Each quick fix, post-merger bolt-on, and regulatory patch compounds technical debt — maintenance rises from 30%

Core Banking Migration: Three Approaches

1. Big Bang

- Replace entire system at once
- Single cutover weekend
- All-or-nothing

Pro: Clean break, no dual maintenance
Con: Catastrophic failure risk, no rollback

Example: TSB Bank (2018)—botched migration left 1.9M customers locked out for weeks

2. Strangler Fig

- Gradually replace components
- Old and new coexist
- Route traffic incrementally

Pro: Low risk, reversible, incremental value
Con: Dual-system complexity, slower

Example: Commonwealth Bank of Australia—5-year program, 15M accounts migrated

3. Parallel Run

- Both systems process simultaneously
- Compare results daily
- Switch when confidence is high

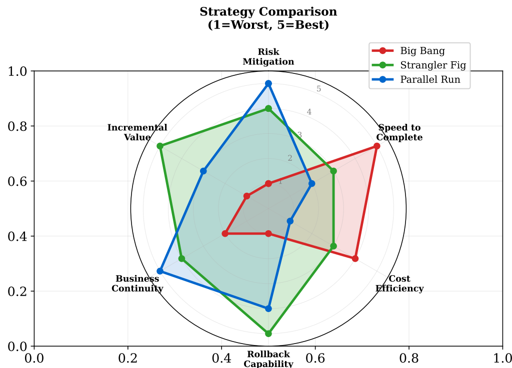
Pro: Maximum safety, full verification
Con: Double the cost, data sync challenges

Example: Common in regulatory-critical systems (trading, settlement)

There is no risk-free migration strategy. The choice depends on the bank's risk appetite, budget, and regulatory constraints.

Migration Strategy Comparison

Core Banking Migration: Strategy Comparison



Strategy Summary

Dimension	Big Bang	Strangler Fig	Parallel Run
Best for	Small banks, simple systems	Large banks, complex systems	Regulated, mission-critical
Duration	3-6 months	3-7 years	2-4 years
Risk Level	Very High	Low-Medium	Low
Cost	Lowest upfront	Medium	Highest (2x ops)
Industry Preference	Rare (neobanks)	Most common	Trading/settlement

- **What you see:** Radar chart (left) scores three strategies across 6 dimensions. Strangler Fig (green) scores highest on rollback capability (5/5) and incremental value (5/5). Table (right) shows durations and best-use cases
- **Key pattern:** Big Bang is fastest but riskiest; Parallel Run is safest but most expensive; Strangler Fig balances risk and value delivery
- **Takeaway:** Most large banks choose Strangler Fig — 3–7 years duration, low-medium risk, incremental value — despite higher

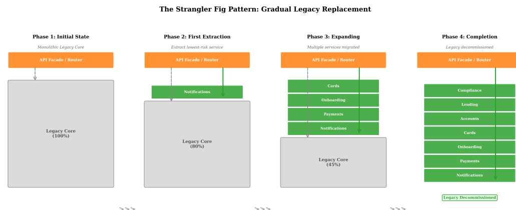
How Do You Replace a System Without Stopping the Bank?

Named after the strangler fig tree:

- A fig vine grows around a host tree
- Gradually replaces the host's structure
- Eventually the host dies, leaving only the fig
- Martin Fowler popularized this for software (2004)

How it works in banking:

1. Identify a bounded context (e.g., savings accounts)
2. Build the new service alongside the old system
3. Route new traffic to the new service (facade pattern)
4. Migrate existing data gradually
5. Decommission the old component when fully replaced
6. Repeat for the next bounded context



The strangler fig pattern is the de facto standard for bank modernization. It allows continuous delivery of value while managing risk incrementally.

Case Study: TSB Bank Migration Failure — What Went Wrong

What happened:

- TSB migrated from Lloyds Banking Group's legacy platform to Sabadell's Proteo4UK
- "Big bang" cutover over a single weekend (April 2018)
- 1.9 million customers locked out of online banking
- Some customers could see other people's accounts
- Fraud losses spiked as criminals exploited the chaos

Root causes:

- Insufficient testing under production load
- Data migration errors (account mapping)
- No phased rollout or canary deployment
- Compressed timeline driven by commercial pressure

TSB attempted to migrate 5.4 million customer accounts in a single weekend — one of the largest "big bang" cutovers ever attempted by a UK bank.

(c) Joerg Osterrieder 2025–2026

Case Study: TSB Bank Migration Failure — Consequences and Lessons

Consequences:

- CEO resigned
- £330M in costs (remediation, compensation, fines)
- 80,000 customers left the bank
- FCA regulatory investigation
- IBM report cited 586 failures across 117 categories

Lessons learned:

- Big bang migrations are high risk for retail banking
- Test at production scale, not just functional correctness
- Phased rollout with rollback capability is essential
- Board and executive ownership of IT risk
- Independent assurance before go-live

TSB is the cautionary tale every bank CTO references. The total cost exceeded 10 times the original migration budget. "Move fast and break things" does not apply to banking.

(c) Joerg Osterrieder 2025–2026

Why Are Banks Moving to the Cloud?

Why banks are moving to cloud:

- **Elasticity:** Scale compute for end-of-month, year-end, Black Friday
- **Speed:** Provision new environments in minutes, not months
- **Cost:** Operating Expenditure (OpEx) vs. Capital Expenditure (CapEx); pay-per-use instead of owning hardware
- **Innovation:** Access to AI/ML, analytics, managed databases
- **Resilience:** Multi-region redundancy out of the box

Cloud models:

- **IaaS:** Lift-and-shift VMs (least value, easiest)
- **PaaS:** Managed databases, containers
- **SaaS:** Core banking as a service (Thought Machine, Mambu)

Cloud is not a destination; it is an operating model. The question is not "should we go to cloud?" but "which workloads benefit from cloud, and under what controls?"

(c) Joerg Osterrieder 2025–2026

Cloud Migration — Banking Challenges and Industry Reality

Banking-specific cloud challenges:

- **Data sovereignty:** Customer data must stay in-jurisdiction (EU GDPR, Swiss FINMA)
- **Concentration risk:** Regulators worry about dependence on AWS/Azure/GCP
- **Latency:** Trading systems need sub-millisecond latency
- **Auditability:** Full audit trail of who accessed what data
- **Exit strategy:** What if you need to leave the cloud provider?

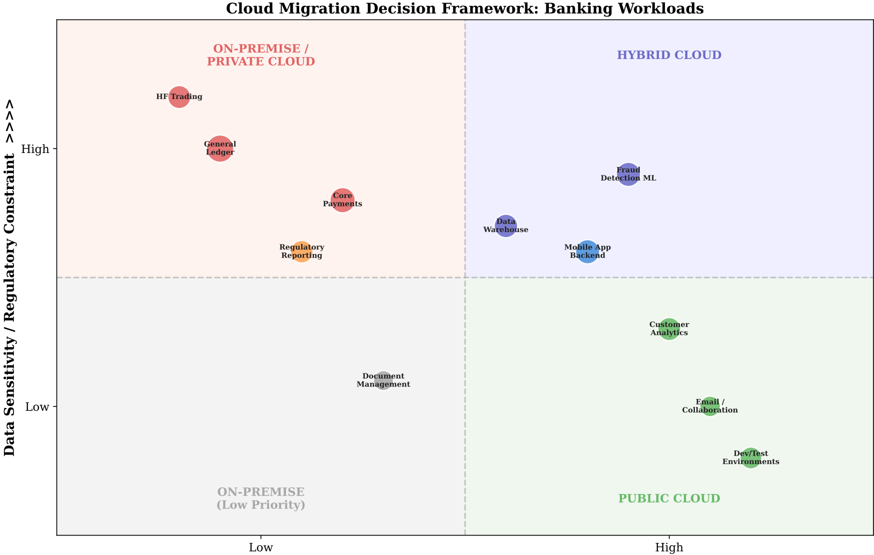
Industry reality (2024):

- Only ∼15% of banking workloads are in public cloud
- Hybrid cloud dominates: sensitive data on-premise, analytics in cloud
- Regulators now accept cloud if controls are demonstrated

Hybrid cloud dominates because regulators and risk teams demand control over sensitive data. Full public-cloud banking remains rare outside neobanks.

(c) Joerg Osterrieder 2025–2026

Cloud Migration Decision Framework



Monolithic Architecture — How Legacy Banks Are Built

Monolithic architecture:

- Single deployable unit containing all functionality
- Shared database, shared codebase
- Change one module $\rightarrow$ redeploy everything
- Tight coupling: one bug can bring down the entire system

Advantages:

- Simpler to develop initially
- Easier debugging (single process)
- No network latency between components
- Well-understood by existing teams

Most legacy bank systems are monoliths. They are simple to reason about but become increasingly difficult to change as they grow in size and age.

Microservice Architecture — How Modern Banks Are Built

Microservice architecture:

- Independent services communicating via APIs
- Each service owns its own data store
- Deploy, scale, and update independently
- Team autonomy: each team owns a service

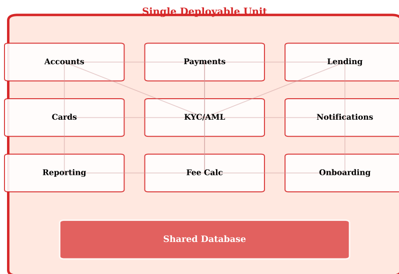
Advantages:

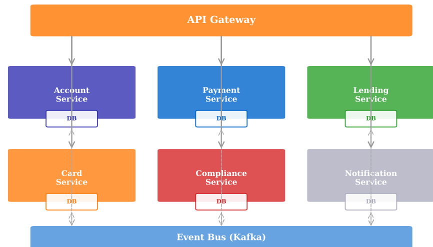
- Independent deployment and scaling
- Technology diversity (polyglot)
- Fault isolation (one service fails, others continue)
- Better alignment with agile/DevOps teams

Microservices are not universally better. They trade monolithic complexity for distributed-system complexity. The right choice depends on team size, system maturity, and operational capability.

Architecture Comparison

Monolithic vs. Microservice: Architecture Comparison





- **What you see:** Monolith (left) shows 9 tightly-coupled modules in a single box sharing one database. Microservices (right) shows 6 independent services, each with its own database, coordinated via an API gateway and event bus

API-First Architecture — Design Principles

What is API-first?

- Design the API contract *before* building the implementation
- APIs are the product, not an afterthought
- OpenAPI/Swagger specifications define the contract
- Enables parallel development: frontend and backend teams work independently

API gateway pattern:

- Single entry point for all external requests
- Handles authentication, rate limiting, routing
- Decouples internal service topology from external consumers
- Examples: Kong, Apigee, AWS API Gateway

API-first design means the API specification is written and agreed upon before any code is written. This prevents integration surprises later.

(c) Joerg Osterrieder 2025–2026

API-First Architecture — Banking Applications and Governance

Why API-first matters for banking:

- Open Banking (PSD2) mandates APIs for third-party access
- Partners, fintechs, and regulators need structured data access
- Internal microservices communicate via APIs
- API versioning prevents breaking changes

API governance in banking:

- **Authentication:** OAuth 2.0, mutual TLS
- **Authorization:** Scope-based access (read-only, transactional)
- **Rate limiting:** Protect core systems from overload
- **Audit logging:** Every API call recorded
- **Versioning:** Semantic versioning (v1, v2) with deprecation policy

---

API-first is not just a technology pattern—it is a business strategy. Banks that master APIs can become platforms, offering Banking-as-a-Service (BaaS) to third parties.

Event-Driven Architecture for Real-Time Banking

What is event-driven architecture (EDA)?

- Systems communicate by producing and consuming events
- An event = "something happened" (e.g., "account credited $500")
- Producers do not know who consumes their events
- Loose coupling: add new consumers without changing producers

Key technologies:

- **Apache Kafka:** Distributed event streaming platform
- **Event sourcing:** Store events as the primary data model (not current state)
- **CQRS:** Command Query Responsibility Segregation—separate read and write models

Banking use cases:

- Real-time fraud detection (process events as they occur)
- Instant balance updates across channels
- Regulatory event reporting (MiFID II transaction reporting)
- Customer notifications (push alerts on transactions)

EDA vs. batch processing:

	Batch	**Event**
Latency	Hours	Milliseconds
Coupling	Tight	Loose
Scaling	Vertical	Horizontal
Replay	Difficult	Built-in

Event-driven architecture is the foundation of real-time banking. Kafka processes trillions of events daily across the financial industry, enabling instant payments, fraud detection, and compliance.

(c) Joerg Osterrieder 2025–2026

Data Architecture Evolution: Warehouse, Lake, Mesh

Data Warehouse

- Structured, schema-on-write
- Star/snowflake schemas
- SQL-based querying
- **Use:** Regulatory reporting, BI dashboards

Pro: Clean, governed, performant
Con: Rigid schema, slow to adapt

Data Lake

- Raw data, schema-on-read
- Stores structured + unstructured
- Cheap storage (S3, ADLS)
- **Use:** ML training, exploration, archiving

Pro: Flexible, cost-effective
Con: Can become "data swamp" without governance

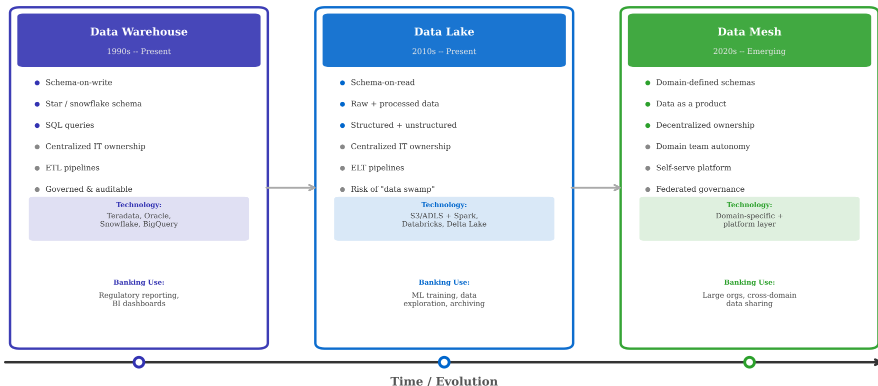
Data Mesh

- Decentralized ownership
- Domain teams own their data products
- Self-serve data platform
- **Use:** Large, complex organizations

Pro: Scalable governance, domain expertise
Con: Organizational change required, coordination overhead

Most modern banks use a hybrid: a governed data warehouse for regulatory reporting, a data lake for analytics and ML, and increasingly data mesh principles for cross-domain data sharing.

Data Architecture Evolution



- **What you see:** Three eras shown as cards on a timeline — Data Warehouse (1990s, purple), Data Lake (2010s, blue), Data Mesh (2020s, green). Each lists characteristics, technology, and banking use
- **Key pattern:** Warehouse = schema-on-write, governed; Lake = schema-on-read, flexible but risky; Mesh = domain ownership

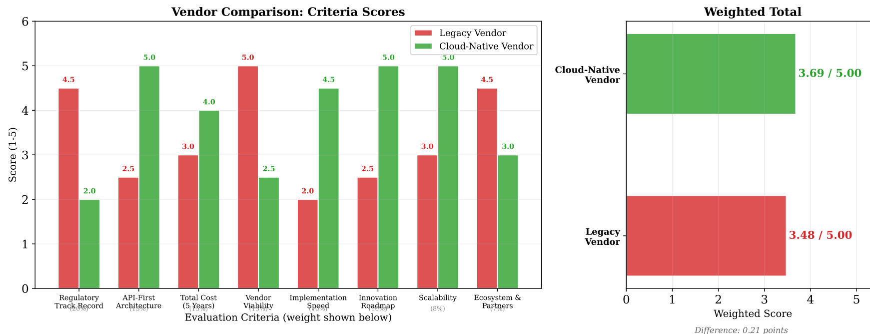
Core Banking Vendor Landscape

Vendor	Product	Architecture	Cloud	Notable Client
Thought Machine	Vault	Cloud-native, microservices	Yes (GCP)	Lloyds, SEB, Standard Chartered
Temenos	Transact	Modular, API-first	Hybrid	3,000+ banks globally
FIS	Modern Banking Platform	Modular	Hybrid	50+ top-100 banks
Mambu	SaaS Core	Cloud-native, composable	Yes (AWS)	N26, ABN AMRO
10x Banking	SuperCore	Cloud-native	Yes (AWS/GCP)	JPMorgan Chase (UK)
Finxact	Core-as-a-Service	API-first	Yes (AWS)	Acquired by Fiserv

The vendor choice is one of the most consequential technology decisions a bank makes. It determines architecture, operational model, and innovation capacity for 10–20 years.

(c) Joerg Osterrieder 2025–2026

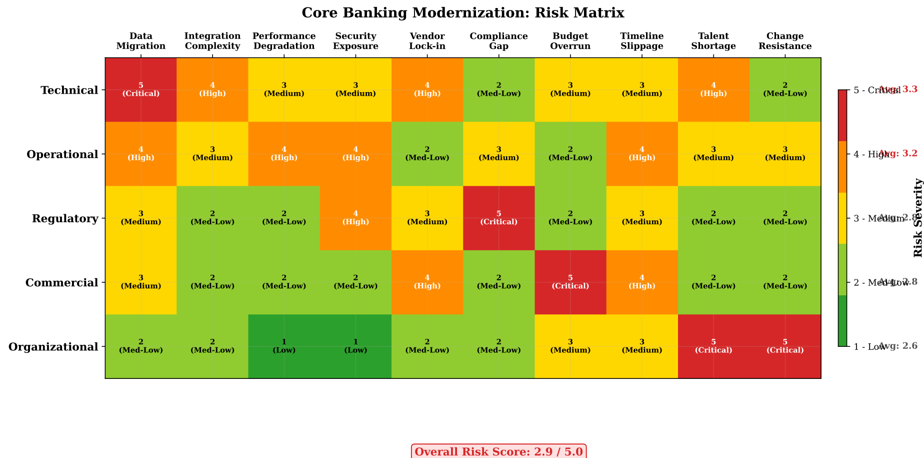
Vendor Evaluation Framework



- **What you see:** Left panel scores legacy vendor (red) vs. cloud-native vendor (green) across 8 criteria with weights. Right panel shows weighted totals: cloud-native 3.79/5, legacy 3.20/5
- **Key pattern:** Legacy wins on regulatory track record (4.5) and vendor viability (5.0); cloud-native wins on API architecture (5.0), innovation (5.0), and scalability (5.0)
- **Takeaway:** No clear winner — choice depends on bank's risk appetite and whether stability or innovation is prioritized

A rigorous evaluation framework considers technology, commercial terms, regulatory fit, vendor viability, and implementation track record. No single

Modernization Risk Matrix



- **What you see:** Heatmap showing 5 risk categories (rows) vs. 10 risk factors (columns). Red cells (5/5) indicate critical risks: data migration, compliance gaps, budget overruns, talent shortage, change resistance
- **Key pattern:** Highest-risk areas cluster around data migration (technical), compliance gap (regulatory), budget overrun

Critical Success Factors — Technology and Organization

Technology:

- API-first design from day one
- Comprehensive automated testing (10,000+ test cases)
- Data migration rehearsals (minimum 3 full runs)
- Canary deployments and feature flags
- Performance testing at 2$\times$ production load

Organization:

- Executive sponsorship at board level
- Dedicated migration team (not shared with BAU)
- Early and continuous regulator engagement
- Change management for branch and operations staff

Technology alone does not determine success. Executive sponsorship and dedicated teams are the strongest predictors of whether a migration program delivers on time.

(c) Joerg Osterrieder 2025–2026

Critical Success Factors — Process and Track Record

Process:

- Incremental delivery (strangler fig, not big bang)
- Business-outcome-driven milestones, not just technical milestones
- Independent assurance / third-party review
- Clearly defined rollback criteria and procedures

What the data shows:

- 70% of core banking migrations exceed their original timeline
- 50% exceed their original budget
- Average duration: 3–7 years for a large bank
- Success rate is higher with phased approaches
- Banks that involve regulators early face fewer delays

Core banking migration is as much an organizational transformation as a technology project. The banks that succeed treat it as a multi-year strategic program, not a one-off IT project.

(c) Joerg Osterrieder 2025–2026

Case: CrowdStrike Falcon Update (2024)

What happened

- 19 July 2024 04:09 UTC: CrowdStrike Falcon sensor configuration update pushed to Windows endpoints
- Null-pointer dereference in the channel-file parser triggered a kernel-mode crash on boot
- 8.5M Windows machines (*CrowdStrike RCA 23 July*, 2024) affected globally; Delta Airlines, banks, hospitals offline
- Insured losses: USD 5.4bn (*Parametrix 24 July estimate*, 2024)
- Manual remediation required for many endpoints (boot-loop blocked automatic fix push)

Why it matters here

- Operational-resilience case study for M6L2 core banking: third-party-software risk is on the critical path
- Direct stress test of EU DORA (Digital Operational Resilience Act) third-party-risk frameworks
- Empirical reminder that single-vendor concentration is a systemic risk even in software

Full writeup: `v4/cases/case_M6_crowdstrike.md`. **Host: M6 L2 Core Banking.**

(c) Joerg Osterrieder 2025–2026

Case: CrowdStrike Falcon Update – Lessons

- Staged rollouts: CrowdStrike subsequently moved to canary deployments for sensor channel files
- Boot-time kernel-mode hooks bypass the OS recovery path; safe-mode boot was the remediation
- Cyber insurance + business-interruption insurance: most policies covered some but not all of the cascading losses
- Pre-existing DORA testing-of-tools requirements (Art. 24) now have a textbook scenario

See also: M6 L2 Core Banking; full writeup at `v4/cases/case_M6_crowdstrike.md`**.**

(c) Joerg Osterrieder 2025–2026

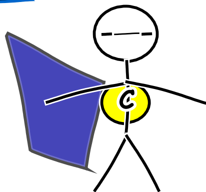
One Last Thought. . .

```
01 ACCOUNT-BALANCE PIC 9(15)V99.
05 CUSTOMER-ID PIC X(20).
PERFORM CALC-INTEREST
  UNTIL END-OF-FILE.
MOVE SPACES TO WS-AREA.
```

MAINFRAME.COBOL (1987)

???

ERROR: Nobody knows
what this code does



She can read

Our savior!

The last person who understands the code becomes the most powerful person in the building

Sometimes the best way to remember a concept is to laugh about it.

Key Takeaways

1. **Core banking systems** are the central nervous system of a bank—processing every transaction, maintaining every balance, and feeding every report
2. **The general ledger** enforces double-entry bookkeeping and is the authoritative record; migrating it is the hardest part of any modernization
3. **Technical debt** compounds like financial debt—banks spending 70–80% of IT budgets on maintenance is the symptom
4. **COBOL is not the problem**; the shrinking talent pool and undocumented business logic embedded in legacy code are
5. **The strangler fig pattern** is the industry standard for migration: gradual, reversible, incremental value delivery
6. **Cloud migration** in banking is a regulatory and architectural decision, not just a hosting decision
7. **Microservices and event-driven architecture** enable the agility banks need but introduce distributed-system complexity
8. **Data architecture** is evolving from centralized warehouses to hybrid approaches including data mesh for domain ownership

---

Modernizing core banking is the defining technology challenge of the 2020s for financial institutions. Getting it right determines competitive survival.

Summary: Lesson 6.2

Concepts Covered:

- Core banking system components and the general ledger
- COBOL, mainframes, and the legacy burden
- Technical debt: sources, measurement, and consequences
- Migration strategies: big bang, strangler fig, parallel run
- TSB Bank case study (migration failure)
- Cloud migration: drivers, models, and banking-specific challenges
- Monolithic vs. microservice architecture
- API-first design and API governance
- Event-driven architecture (Kafka, event sourcing, CQRS)
- Data warehouse, data lake, and data mesh
- Vendor landscape and evaluation framework

What comes next:

- Lesson 6.3: Identity, authentication, and cybersecurity infrastructure
- Lesson 6.4: Operational resilience and disaster recovery

Key numbers to remember:

- 95% of ATM transactions run on COBOL
- 70–80% of bank IT budgets go to maintenance
- 70% of core migrations exceed their timeline
- £330M: cost of TSB's failed migration
- $18B: global core banking software market (2024)

Core banking modernization is the foundation on which all other digital finance innovations are built. Without it, real-time payments, open banking, and embedded finance remain aspirations.

Discussion Questions

1. A mid-size European bank (2M customers, €50B assets) is running a 30-year-old COBOL core. The board asks: “Should we modernize?” What questions would you ask before recommending a strategy?
2. Why did TSB choose a big bang migration, and what organizational pressures might have overridden the technical risks? Could the same failure happen today?
3. A neobank built cloud-native from day one argues it has “zero technical debt.” Do you agree? What new forms of technical debt might cloud-native systems accumulate?
4. Compare the data architecture needs of a retail bank (millions of small transactions) vs. an investment bank (fewer but larger, more complex transactions). Would they choose the same approach?
5. If COBOL processes $3 trillion daily and achieves 99.999% uptime, is the push to modernize driven by genuine need or by technology fashion? Argue both sides.

These questions have no single right answer. They require balancing technical, commercial, regulatory, and organizational considerations—exactly like real decisions in banking.

Self-Check: Core Banking

Attempt these before turning the page.

1. [Understand] In one sentence each, contrast big bang, parallel run, and strangler fig migration strategies. Which has the lowest implementation risk?
2. [Apply] A bank runs 700 COBOL programs totalling 40M LOC. Migration estimate: 5 LOC/hour refactored, $80/hour engineer. Compute total cost. Over what horizon does a 15% reduction in ops cost (ops = $500M/yr) justify this investment?
3. [Evaluate] Monzo's ledger-service split (2019) enabled 24/7 settlement. Why do most incumbents not follow? Name two non-technical reasons.

Solutions hidden unless `\solutionstrue` **is set before compiling.**

(c) Joerg Osterrieder 2025–2026