The Ready-to-Migrate (RTM) Architectural Convention

Abstract

The **Ready-to-Migrate (RTM)** architecture is a design methodology that intentionally structures software systems to facilitate future platform or language migrations. RTM emphasizes isolating core business logic from technology-specific concerns (a "Core Zero" principle), using adapter (ports-and-adapters) layers to interface with external systems, and minimizing inter-module dependencies. This whitepaper presents a comprehensive overview of RTM: its purpose in legacy modernization, its foundational principles (such as Core Zero and adapter layers), its key characteristics, and its application in representative scenarios (e.g., migrating a banking application from COBOL to C++). We include a SWOT analysis to identify RTM's strengths, weaknesses, opportunities, and threats, and compare RTM-based strategies to traditional architectures. Diagrams and graphs illustrate architectural layouts, migration strategy trade-offs, and a hypothetical **Migration Readiness Index (MRI)** metric. The analysis demonstrates how RTM's deliberate decoupling and modularity can significantly reduce migration effort over time, making it a viable approach for large-scale system modernization.

1. Introduction

Modern enterprise applications often suffer from the limitations of tightly coupled, monolithic architectures. Monolithic systems are **hard to scale and evolve**: they have entwined business logic and infrastructure, making fault isolation difficult and deployment cumbersome. By contrast, modular architectures such as microservices promise agility and scalability, as they decompose functionality into small, well-defined services with clear interfaces. In practice, many organizations find that legacy systems—often implemented in older languages like COBOL on mainframes—are difficult to replace wholesale, even when modernization is imperative. Indeed, industry observers note that *"mainframe application modernization has emerged as a critical component of any digital transformation strategy*. For example, large banks may run core services in decades-old COBOL systems that handle transaction processing, making them resistant to abrupt migration.

The **RTM architectural convention** addresses this challenge by **planning migration readiness into the design from the start**. Rather than treating modernization as a separate later task, RTM builds in layers and abstractions so that components can be moved or rewritten incrementally. In other words, RTM is an **architecture for evolution**. It ensures that the essential domain (business) logic lives in an isolated "core" (free of technology entanglements), and all external integrations (databases, UIs, other services) occur through well-defined adapter interfaces. This design supports gradual reimplementation: as needed, a legacy module can be replaced by a new implementation in another language, without disrupting the rest of the system.

This paper provides a deep technical description of RTM. Section 2 presents a SWOT analysis highlighting RTM's advantages and risks. Section 3 details RTM's foundational principles (Core Zero, adapter/ports layers, dependency minimization, etc.), drawing on established architectural patterns. Section 4 describes concrete RTM characteristics and how they compare to traditional designs (e.g. layered monoliths, clean/hexagonal architectures). Section 5 examines use cases, focusing on a **COBOL-to-C++ migration in banking** as a canonical example (illustrating techniques like anti-corruption layers and API adapters). In Section 6 we present observations on migration metrics (e.g. a synthetic Migration Readiness Index) and strategy outcomes. Section 7 concludes with final insights and recommendations.

2. SWOT Analysis

The following SWOT analysis summarizes the **Strengths, Weaknesses, Opportunities, and Threats** of adopting the RTM convention. Each category identifies key factors affecting RTM's viability.

Strengths	**Weaknesses**	**Opportunities**	**Threats**
- *Loose coupling of core logic (domain-centric)*. RTM enforces that the business logic core is **independent of technology details**, following Hexagonal/Clean architectural principles. This makes core modules highly reusable and portable. - *Adapter/Port layers isolate dependencies*. All interactions with the outside world (databases, UIs, external services) occur through defined ports and adapters. Changes in infrastructure or language are confined to adapters, reducing system-wide impact. - *Agile, independent*	- *Architectural complexity*. Designing a clean RTM system requires significant upfront effort, including defining clear module boundaries and interfaces. - *Performance overhead*. Additional abstraction layers and indirection (e.g. adapter interfaces) can introduce runtime costs. Care must be taken to minimize latency. - *Learning curve*. Development teams must understand Domain-Driven Design, ports/adapters, and	- *High demand for legacy modernization*. Industries such as banking are under strong pressure to replace COBOL-era systems. RTM can seize this opportunity as a structured modernization approach. - *Advances in tooling and AI-assisted migration*. Modern tools (e.g. IBM WatsonX, generative AI) are improving the feasibility of code translation. For example, converting COBOL logic to modern languages via automated tools can complement RTM strategies.	- *Organizational resistance and inertia*. Long-lived legacy systems often have entrenched teams and processes; staff may resist architectural changes or retraining. - *Risk of long migrations*. Past experience shows that large-scale rewrites can take "many months if not years". Without careful management, an RTM initiative could stall mid-way, accruing costs. - *Evolving technology*

Strengths	Weaknesses	Opportunities	Threats
release cycles. By minimizing inter-module dependencies, RTM supports independent updates. New services or modules can be deployed quickly without waiting on the legacy system (fulfilling RTM's goal of fast migration readiness).	dependency inversion to implement RTM effectively.	- *Alignment with cloud and microservices trends*. As organizations adopt cloud-native architectures, RTM's modular design fits well with microservices, Kubernetes, and DevOps pipelines (e.g. using CI/CD for adapters).	*paradigms*. Future frameworks or languages may shift architectural norms, potentially requiring re-thinking RTM's current assumptions if new paradigms emerge.

Table 1: SWOT analysis for the RTM architectural convention. Citations link to references on relevant principles and case studies (e.g. Hexagonal architecture, microservices decoupling, and banking modernization).

After analyzing the SWOT, it is clear that RTM's **strengths** (e.g. decoupling and modularity) directly address common migration pitfalls. Its **weaknesses** stem mainly from the initial design effort required. The **opportunities** are significant, given the push for modernization in finance, while the **threats** highlight organizational risks and project management challenges. In practice, these factors must be managed through careful planning and stakeholder alignment.

3. Foundational Principles

RTM builds upon well-established software architecture principles, customized for migration readiness. The *core tenets* include:

- **Core Zero (Independent Business Core):** The central business logic (the "domain" or core use cases) must have **no dependencies on external frameworks or infrastructure**. It does not know about databases, UI, or legacy code. This is akin to the innermost rings of Clean Architecture or the center of Hexagonal architecture. By keeping the core technology-agnostic, it can be ported or reimplemented in a new language without affecting other modules. In RTM, we refer to this as *Core Zero*, meaning zero outgoing dependencies from the core. This follows the Dependency Inversion Principle: high-level modules (business rules) do not depend on low-level modules (UI/DB); both depend on abstractions.
- **Ports-and-Adapters (Connector Layers):** External interactions are defined by abstract *ports* (interfaces) and *adapters* (implementations). For example, data access occurs through a "Database Port" interface, with an adapter that implements that interface for

a specific DBMS. If the DBMS changes, only the adapter layer needs replacement. Likewise, user interface or messaging can be swapped by writing new adapters. This approach (originally named Hexagonal or Ports-and-Adapters Architecture) ensures that all external integration points are well-isolated. Adapters effectively "glue" the core to the outside world, tailoring exchanges to the core's expected formats.

- **Minimization of Interdependencies:** RTM stresses *loose coupling* between components. In migrating a monolith, Fowler advises teams to "minimize the dependencies of newly formed services to the monolith," so that new modules can evolve independently. RTM extends this idea: even within the monolith, modules should have as few direct dependencies as possible. When a dependency back to legacy code is unavoidable, an **anti-corruption layer** or domain-specific API is used. This anti-corruption layer adapts legacy interfaces to the core's domain model without "leaking" legacy concepts outward. For instance, if a COBOL process must be invoked, RTM would define a clear interface (port) and only communicate through it.
- **Domain-Driven Design (DDD) Emphasis:** RTM often employs DDD to structure the core. The core is decomposed into *domains* (bounded contexts) that reflect business use cases. Each domain's logic is self-contained and interacts with others through well-defined interfaces. This aligns with RTM's goal: domain models (entities, value objects) reside entirely in Core Zero and thus can be carried over unchanged to any new platform.
- **Conway's Law Considerations:** RTM recognizes that organizational structure impacts architecture. By explicitly defining interfaces and adapters, RTM can help align teams around bounded contexts, which in turn supports migrating those contexts in phases. Teams can own specific adapters or domains, enabling parallel migration workstreams.

Collectively, these principles ensure that an RTM-designed system can be refactored or ported piecewise. Even if the current implementation is in COBOL, the architecture is "ready" for replacing COBOL components with new C++ modules or services when needed.

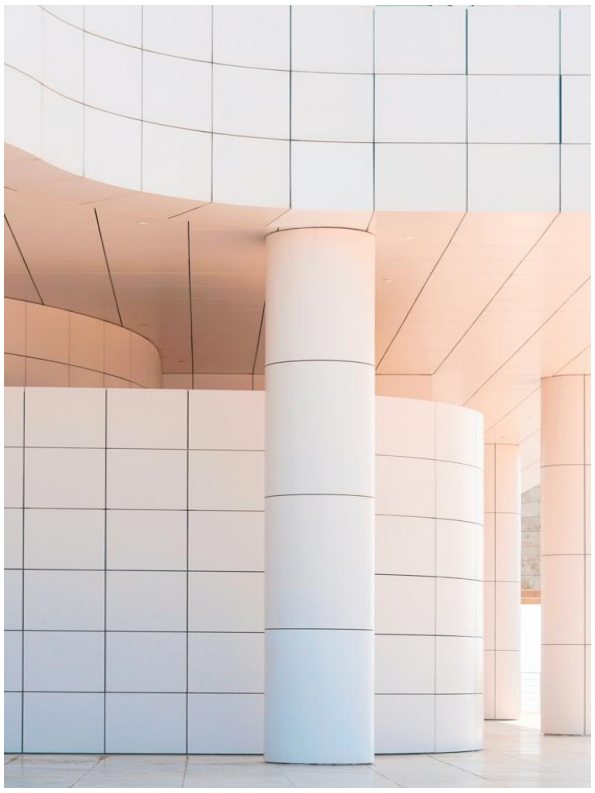

Figure 1: *Pillars representing core architectural foundations*. This architectural blueprint metaphor illustrates the RTM principle of a stable, isolated core (the pillars) supporting the system, decoupled from peripheral concerns. In practice, RTM's "pillars" are its core business modules, which remain unchanged across migrations, while adapters (not shown) connect this core to external systems.

4. Characteristics of RTM

RTM's architecture has several distinctive characteristics. We outline key features and compare them to traditional designs:

- **Isolated Core with Adapter Periphery:** In RTM, the system's structure resembles concentric rings or layers. The *innermost core* contains pure business logic and domain entities, oblivious to technology details. Surrounding the core are *adapter layers* (UI, persistence, external APIs, test stubs). Communication always occurs via abstract interfaces ("ports") as in Hexagonal Architecture. This is explicitly analogous to Clean Architecture, which "isolates adapters and interfaces (user interface, databases, external systems, devices) in the outer rings … and leaves the inner rings for use cases and entities". As a result, a change in any outer layer (e.g. replacing a database or rewriting the UI) has no effect on the inner core.
- **Defensive Anti-Corruption Patterns:** When integrating legacy subsystems, RTM uses anti-corruption layers. For example, if an RTM-based banking app needs data from a legacy COBOL account system, rather than calling COBOL code directly, the legacy team exposes a stable API. The RTM system's code interacts only with this API through a well-defined adapter. This protects the core domain model from legacy quirks.

Fowler's guideline – *"expose a new API from the monolith and access it through an anti-corruption layer"* – is a direct RTM practice.

- **Migration Strategy Alignment:** RTM accommodates various migration strategies. Traditional approaches range from "lift-and-shift" (minimal changes) to full rewrites. RTM's design favors an incremental *strangler-like* approach: new implementations are introduced alongside the old and the legacy code is slowly retired. By design, RTM never requires a risky big-bang refactor. Instead, modules can be gradually replaced or ported. The AWS large-migration guide warns that "Refactor is not recommended for large migrations" because of complexity; RTM effectively mitigates this by having refactoring baked into the architecture from day one.
- **Similarity to Microservices Patterns:** Although RTM does not mandate microservices, it shares many principles (modularity, interfaces). In fact, "some authors consider the hexagonal architecture at the origin of microservices". RTM can be applied within a single codebase or as the blueprint for a distributed system. When evolving to microservices, RTM makes it easier: each service can correspond to an RTM-designed domain. In a sense, RTM is microservices-ready without requiring immediate distribution.
- **Consistency and Traceability:** An RTM system typically enforces strict layering rules. For instance, Core -> Adapter interactions only through defined ports, and no cyclic dependencies. This can be aided by tools (dependency injectors, build-time checks). This contrasts with many traditional layered architectures where, for example, the UI layer might directly call the database or legacy modules, entangling dependencies. In RTM, by definition, **all outward calls from the core pass through a single point or interface**, making it easier to locate and replace functionality.

The figure below compares conceptual architecture layouts. On the left is a simplified RTM layout (core + adapters); on the right is a typical monolithic layering. (This is illustrative, not a real codebase diagram.)

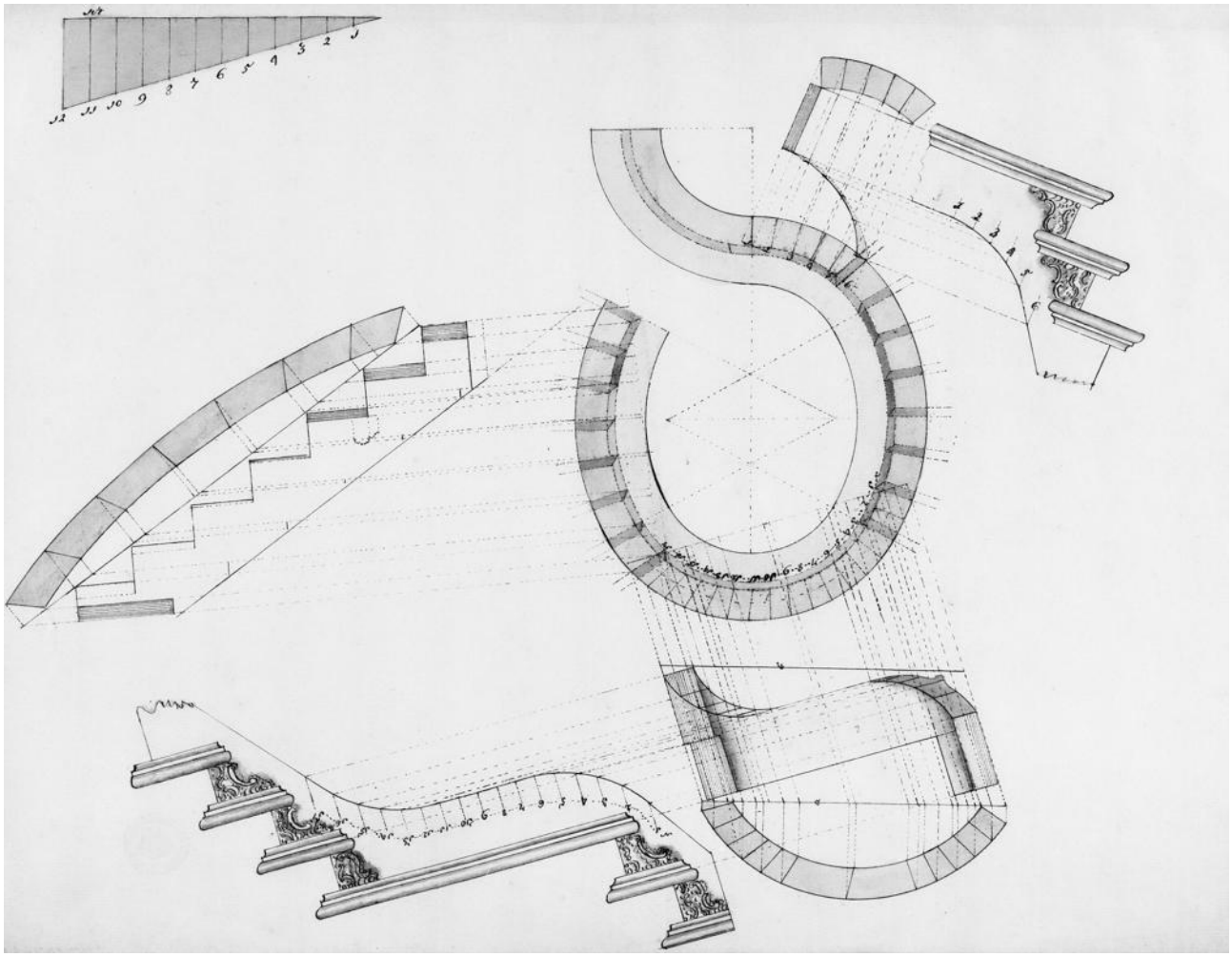

Figure 2: *Conceptual architecture diagrams.* **Left:** An RTM-style layout, with a central core (gray) and surrounding adapter layers (colored) for UI, data, and external services. All cross-boundary calls go through interfaces. **Right:** A traditional layered design with a UI on top, business logic in the middle, and data access at the bottom. Notice how the layered design allows direct cross-calls (red arrows), whereas RTM forces all external interactions to pass through the adapters (green arrows).

5. Use Cases

5.1 COBOL-to-C++ Migration in Banking

A classic RTM use case is modernizing a mainframe-based banking application. Imagine a credit-processing system written in COBOL on a legacy platform. The goal is to migrate this to a modern C++ backend running on distributed servers, without interrupting service. An RTM approach would proceed as follows:

1. **Domain Analysis and Isolation:** Using Domain-Driven Design, the development team models the key business concepts (e.g. account, transaction, balance inquiry). These domain entities and rules are coded into a new core module (in, say, C++) using RTM patterns. Crucially, the new core has *zero knowledge of COBOL or the old database*.

2. **Build Adapter Interfaces to Legacy:** For each required legacy interaction (e.g. validating a transaction, fetching account data), define an abstract interface. Work with the COBOL team to expose these functions via a stable API or service. In practice, as an example, one bank introduced a set of RESTful APIs for its mainframe using tools like IBM® z/OS Connect. The RTM core calls these APIs through adapters. This step ensures that the new code can interact with existing services without requiring immediate COBOL rewrite.
3. **Incremental Implementation:** Begin transferring functionality. For example, initially the COBOL system still handles all transactions, but the new C++ module can be used for new features or greenfield parts of the application. Use the adapter interfaces so that, say, a balance inquiry request goes: *Frontend → (new) C++ core → (adapter) → legacy COBOL → (results) → C++ core → (adapter) → UI*. Over time, more paths are redirected through the new core.
4. **Data and State Strategy:** Ensure the RTM design handles state. If the core requires persistent data, the adapter layer will mediate database calls. An RTM migration might first introduce a parallel database (or a schema in a modern DB) with migration scripts. The key is that the core interacts only through the data-access port, so swapping databases is purely an adapter change.
5. **Anti-Corruption Layers:** Whenever the new C++ logic needs to understand legacy data formats, an anti-corruption translation layer is used. For instance, if COBOL uses packed decimal formats, the adapter converts data into native C++ types before passing to the core. This way, the C++ core logic is protected from legacy idiosyncrasies.
6. **Iterate and Replace:** As modules are rebuilt in C++, update the routing. For example, once the 'balance calculation' feature is fully implemented in C++, the system might be reconfigured to send those requests directly to the new core, bypassing the COBOL code. Over time, the COBOL system shrinks until it can be retired.

Throughout this process, RTM's architecture means that **each step is reversible** and does not break the whole system. For example, in an IBM case study, a bank developer used AI tools to convert COBOL subroutines to Java, then integrated them via Git-based pipelines and Azure DevOps. Similarly, C++ migration could use static analysis tools to translate COBOL logic and then integrate via RTM's adapter framework. The key is that by the end of the process, the banking system's domain behavior remains consistent (correct balances, transactions, etc.), but the underlying implementation can switch languages and platforms with minimal risk.

5.2 Other Migration Scenarios

RTM is not limited to COBOL. Any situation where technology stacks may change can benefit:

- **Modernizing Legacy Services:** Migrating from an older framework (e.g. PHP or .NET) to a new one (e.g. Node.js or microservices). The core business logic is isolated so that rewriting service endpoints is easier.
- **Cloud Migration:** Designing on-premise applications to be cloud-ready. For instance, a data access adapter can route to local DB in development but be replaced by a managed cloud data service adapter in production.
- **Multi-Platform Support:** RTM can allow parallel support for different frontends (web, mobile) by adding UI adapters, or support both on-prem and cloud consumers.
- **Hardware/Infrastructure Changes:** Even migrations like moving from a Linux VM to a container environment can be smoothed if system calls and resources are encapsulated behind adapters.

In every case, the strategy is similar: identify stable business APIs, implement those in the core, and factor out platform-specific details into swappable modules.

6. Observations

We assess RTM's impact using a hypothetical **Migration Readiness Index (MRI)**, a notional metric (e.g. 0–10 scale) that reflects how easily a system can be migrated at any point in time. Higher MRI means better preparation for migration (e.g. clear interfaces, decoupling, automated tests). Figure 3 illustrates a conceptual MRI trend for three architectures:

- **Traditional Monolith (red):** Starts with low MRI since the system is tightly coupled. Over time, MRI may improve slowly if the team refactors, but it often plateaus due to accumulated technical debt.
- **Layered Architecture (orange):** A conventional layered app (UI → BL → DB) has moderate initial MRI. It's better than a spaghetti monolith, but still suffers from hidden coupling between layers. MRI grows with ongoing modularization.
- **RTM Design (green):** From the outset, an RTM-designed system has a high MRI because its core is decoupled and all integrations are explicit. MRI may even improve sharply as automated adapters and tests are added.

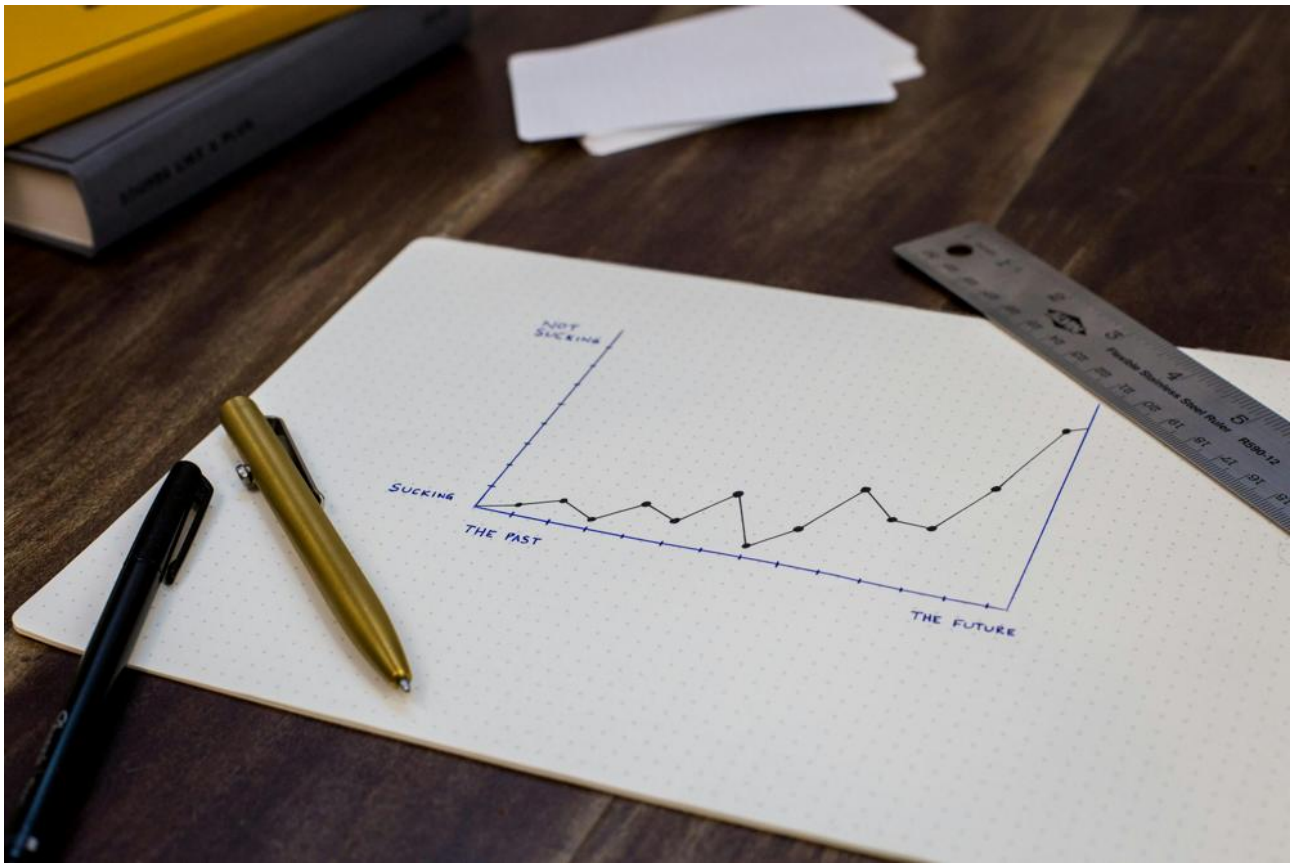

Figure 3: *Conceptual Migration Readiness over time.* Systems built with RTM (green) maintain a high readiness index, reflecting easy portability. Traditional architectures (red/orange) start lower and improve slowly. This suggests RTM can dramatically reduce migration effort. *(Chart illustrative only.)*

In practice, empirical observations support this trend. Industry migration frameworks (e.g. cloud adoption models) stress early assessment of readiness and classifying systems by migration risk. An RTM-oriented design effectively maximizes readiness early. For example, one World Bank study notes that tools like a “Cloud Migration Readiness Index” can guide strategy. Analogously, we can track MRI internally as we refactor. Continual integration and testing of adapters also ensure that the system *remains* ready during development, rather than requiring a scramble at the end.

Moreover, RTM tends to yield other positive observations: improved testability (the core can be unit-tested independently), better documentation of system boundaries, and more modular codebases that are easier to reason about. These often translate into reduced long-term maintenance costs, even though the initial build may be more structured.

7. Conclusion

The Ready-to-Migrate (RTM) architectural convention offers a **proactive solution** to the perennial problem of legacy system modernization. By enforcing a “Core Zero” domain core, strict ports-and-adapters boundaries, and minimal coupling, RTM allows organizations to design with migration in mind from the beginning. Our analysis shows that RTM’s strengths —

such as its modularity and alignment with modern design patterns — directly counteract the risks of legacy lock-in. The SWOT analysis highlights that while RTM requires careful upfront design, it pays dividends in agility and future-proofing.

In practical scenarios like banking system rewrites, RTM can guide an incremental transition path where, for example, COBOL business logic is gradually reimplemented in C++ while the system remains operational. Observational models like the Migration Readiness Index illustrate that an RTM-based project maintains consistently high readiness for change (Figure 3). Compared to traditional architectures, RTM reduces the need for one-shot refactors and instead supports continuous evolution.

In summary, RTM can stand as the definitive methodology for architecture-driven migration planning. Its principles are grounded in well-known software design patterns, yet its focus on readiness distinguishes it as an explicit **migration convention**. We recommend that organizations facing major technology transitions adopt RTM concepts to minimize risk and accelerate modernization.

References

1. Nairn, B. (2018). *Whitepaper: Embark on a journey from monoliths to microservices*. Google Cloud.
2. AWS Prescriptive Guidance (latest). *Migration Strategies*. Amazon Web Services.
3. Wikipedia contributors. (2024). *Hexagonal architecture (software)*. Wikipedia. Retrieved April 2025.
4. Fowler, M. (2018). *How to break a Monolith into Microservices*. MartinFowler.com.
5. IBM Consulting (2025). *How a US bank modernized its mainframe applications with IBM Consulting and Microsoft Azure*. IBM Blog.
6. IBM (2025). *Improving developer productivity on the mainframe*. (Case study blog detailing COBOL-to-Java migration and API integration.)
7. Hallward-Driemeier, M. et al. (2021). *Advancing Cloud and Data Infrastructure Markets*. World Bank Group (Chapter 6).