**Retail Modular Banking**

# Table of Contents

[Table of Contents 2](#_Toc193268274)

[Document History 5](#_Toc193268275)

[Introduction 6](#_Toc193268276)

[Definitions 6](#_Toc193268277)

[Domain Definition – Overall Modular Landscape 7](#_Toc193268278)

[Core Services 8](#_Toc193268279)

[Cross-Core Services 8](#_Toc193268280)

[Support Services 8](#_Toc193268281)

[Additional Support Services 8](#_Toc193268282)

[Modular Banking Design Principles 9](#_Toc193268283)

[API-First Design: A Strategic Approach to Software Development 9](#_Toc193268284)

[Event-Driven Architecture in Modular Banking 9](#_Toc193268285)

[Strategic Alignment of APIs and Events with BIAN Terminology 10](#_Toc193268286)

[User Experience in Modular Banking 11](#_Toc193268287)

[Demo User Agent 12](#_Toc193268288)

[Retail Modular Banking Deployment Prerequisites 13](#_Toc193268289)

[Technology Foundation 14](#_Toc193268290)

[Transactional Outbox 14](#_Toc193268291)

[Event Store 14](#_Toc193268292)

[Service Orchestrator 15](#_Toc193268293)

[Master Data Access Layer (MDAL) 16](#_Toc193268294)

[1. Application Architecture 17](#_Toc193268295)

[Deposits 17](#_Toc193268296)

[Products and Features Supported 17](#_Toc193268297)

[Current/Checking and Savings Accounts 17](#_Toc193268298)

[Term Deposit Accounts 18](#_Toc193268299)

[Multi-Currency Accounts 19](#_Toc193268300)

[Corporate Accounts & Deposits 19](#_Toc193268301)

[End of Day Processing (Close of Business-COB) 20](#_Toc193268302)

[Additional Features 20](#_Toc193268303)

[Lending 20](#_Toc193268304)

[Products and Features Supported 20](#_Toc193268305)

[Retail Lending 20](#_Toc193268306)

[Corporate Lending 21](#_Toc193268307)

[End of Day Processing (Close of Business-COB) 22](#_Toc193268308)

[Additional Features 22](#_Toc193268309)

[Common Business Application Components 22](#_Toc193268310)

[Holdings microservice 22](#_Toc193268311)

[Party microservice 23](#_Toc193268312)

[Product Design 23](#_Toc193268313)

[2. Deployment Architecture 24](#_Toc193268314)

[Cloud Native Deployment 24](#_Toc193268315)

[Leveraging Infrastructure Capabilities for Temenos Modular Solutions 24](#_Toc193268316)

[AWS: Temenos Modular Solutions on AWS 24](#_Toc193268317)

[Azure: Temenos Modular Solutions on Azure 25](#_Toc193268318)

[OpenShift: Temenos Modular Solutions on OpenShift 25](#_Toc193268319)

[Deployment Architecture: Deposits 25](#_Toc193268320)

[Deployment Architecture: Retail Lending 25](#_Toc193268321)

[3. Configuration and Extensions 26](#_Toc193268322)

[Product Configuration 26](#_Toc193268323)

[Extensibility Framework 26](#_Toc193268324)

[Temenos Modular Workbench 26](#_Toc193268325)

[Business Logic Extensions using Java 27](#_Toc193268326)

[Extensibility Approach for Modular Solutions 27](#_Toc193268327)

[4. Integration Architecture 28](#_Toc193268328)

[Integration Approach 28](#_Toc193268329)

[Modular Banking APIs 29](#_Toc193268330)

[APIs for Deposits 29](#_Toc193268331)

[APIs for Retail Lending 29](#_Toc193268332)

[Modular Banking Events 30](#_Toc193268333)

[Events for Deposits 30](#_Toc193268334)

[Events for Retail Lending 30](#_Toc193268335)

[Exporting Data 31](#_Toc193268336)

[Country specific variants of APIs and Events 32](#_Toc193268337)

[Internal Temenos Interfaces 33](#_Toc193268338)

[Interface between Retail Deposits and Retail Lending 33](#_Toc193268339)

[Interface between Enterprise Pricing and Deposits 33](#_Toc193268340)

[Interface between Enterprise Pricing and Retail Lending 33](#_Toc193268341)

[5. DevOps 34](#_Toc193268342)

[Packager 34](#_Toc193268343)

[Integration of Packager within Temenos Modular Workbench 34](#_Toc193268344)

[6. Application Security 35](#_Toc193268345)

[Modular Banking approach to application security 35](#_Toc193268346)

[Authentication 35](#_Toc193268347)

[Authorization 36](#_Toc193268348)

[Encryption 36](#_Toc193268349)

[7. Observability 37](#_Toc193268350)

[8. Implementation Approach 38](#_Toc193268351)

[9. Data Migration 39](#_Toc193268352)

[10. Go Live Considerations 40](#_Toc193268353)

[11. Application Support 41](#_Toc193268354)

[12. Upgradeability 42](#_Toc193268355)

[Appendix 1: Additional details on modular architecture 43](#_Toc193268356)

[Deposits 43](#_Toc193268357)

[Retail Lending 43](#_Toc193268358)

[Payments 43](#_Toc193268359)

[Limits 43](#_Toc193268360)

[Sub-Ledger 43](#_Toc193268361)

[Market and Reference Data 43](#_Toc193268362)

# Document History

|  |  |  |
| --- | --- | --- |
| **Author** | **Version** | **Date** |
| George Panagiotopoulos | 0.1 | 28 January 2025 |
|  | 0.2 | 02 February 2025 |

**Comments:**

|  |
| --- |
| This is a design document for Retail Modular Banking. It outlines all required information about how Temenos |
| applications are refactored to offer Core Banking services with granular footprint |
| V0.2 includes comments in red about discussion points, gaps and design logic. It is an internal document only |

to facilitate the planning process. All comments in red must be replaced by the actual plan once available

# Introduction

In the year 2025, Temenos stands as a leading global vendor in the banking software sector, renowned for its innovative and robust solutions. At the heart of Temenos' offerings lies its flagship product: Core Banking, a solution that has evolved significantly over the decades. Initially known as EBS (Electronic Banking System), the product underwent several transformations, including iterations such as Globus, T24, T24 Transact, and ultimately Transact. These evolutions reflect Temenos' commitment to adapting to the ever-changing demands of the banking industry.

The scope of Core Banking expanded steadily over its history, initially encompassing a broad array of features that went beyond traditional Core Banking functionalities. However, by 2018, with the launch of T24 Transact, a strategic shift began. The solution started to streamline its focus, reducing its footprint and moving towards a more modular approach. This period saw the externalization of non-core functionalities into adjacent solutions, each with distinct architectural boundaries.

The journey toward modularity intensified in 2022-23 with an initiative to refactor Core Banking into more granular services, offering specialized functionalities. The initial vision for this transformation was encapsulated in the "Composable Banking" blueprint. However, after several experimental phases and learning from past efforts, Temenos rebranded its approach as "Modular Banking" in 2025.

Modular Banking embodies a refined strategy, focusing on delivering Core Banking functionalities through discrete, reusable modules. This approach not only aligns with industry trends towards flexibility and scalability but also addresses the complexities of modern banking operations more effectively. The first phase of this initiative introduces two key services: "Retail Deposits" and "Retail Lending." These services exemplify the modular approach by providing focused functionalities tailored to retail banking needs.

This document outlines the architectural blueprint, design considerations, and implementation strategies for Temenos' Modular Banking solutions, setting the stage for future developments and deployments.

# Definitions

**Core Banking** refers to the fundamental software solution utilized by financial institutions to manage their essential operations. It encompasses a wide range of functionalities designed to streamline and support key banking activities.

**Modular Banking** addresses the limitations of legacy Core Banking systems by offering a more flexible and adaptable framework. It builds on the foundational functionalities of account management and transaction processing while introducing modular components that enhance scalability and responsiveness to modern banking challenges.

**Deposits** is a modular solution offered by Temenos, designed to streamline the handling of various deposit types and account management processes for both retail and corporate banking customers. This service encompasses features such as account opening, transaction processing, balance tracking, interest calculations, and compliance with regulatory requirements.

**Retail Lending** is a modular solution offered by Temenos, designed to manage both secured and unsecured loans for individual consumers, offering comprehensive tools for processing, managing, and monitoring retail lending activities within the banking sector.

**Enterprise Pricing** is a modular solution by Temenos designed to offer advanced pricing capabilities across core banking services. It enables institutions to streamline the pricing of loans, deposits, and financial products using sophisticated algorithms. This tool empowers risk managers, financial analysts, and product managers to set competitive and compliant prices efficiently, integrating seamlessly with existing systems while providing scalability and customization for diverse banking needs.

# Domain Definition – Overall Modular Landscape

Temenos adopts a market-demand-driven pragmatic approach in defining domains for its modular core services. This means they create solutions based on real customer needs and industry trends rather than theoretical best practices alone.

Temenos groups BIAN service domains into fewer physical services. It is essential to recognize that BIAN defines numerous service domains, which can become overwhelming when integrating them all. By grouping these domains into a smaller number of modules, Temenos simplifies the system and reduces the complexity of managing interactions between modules, particularly concerning transactions that require handling sagas or two-phase commits.

The three main reasons why this approach is optimal are as follows:

* Simplified Integration: With fewer modules, integrating different services becomes less complicated. Keeping in mind the integration complexity for N modules is O(N²) interfaces, Temenos architecture focuses on several modules, typically around 10-15 core processing services across the entire bank.
* Reduced Complexity in Transactions: By consolidating domains, the number of transactional interactions between modules decreases, making it easier to manage transactions and avoid issues like spaghetti code or unmanageable coordination efforts.
* Balanced Granularity: While reducing the number of modules helps in managing complexity, Temenos ensures that each module remains granular enough to maintain flexibility and scalability without becoming monolithic.

In conclusion, Temenos’ strategic grouping of BIAN domains into fewer, market-driven modules not only simplifies integration but also makes transaction management more straightforward. This balanced approach maintains scalability and aligns closely with customer needs, making it an optimal strategy for their modular banking architecture.

The following diagram summarizes the physical breakdown of Temenos modules.

A screenshot of a diagram

AI-generated content may be incorrect.

The Temenos modular architecture is structured around three primary categories of services: core, cross-core, and supporting. Each category plays a distinct role in the overall system, contributing to efficient transaction processing, bank-wide integration, and technical support.

## Core Services

These are foundational services focused on essential banking operations:

* Deposits: Manages checking and savings accounts for individual customers, as well as term deposits.
* Lending: Handles loans for retail customers.

## Cross-Core Services

These services operate across the entire bank and integrate with core services:

* Pricing: Utilizes advanced algorithms to set prices for financial products like loans and deposits.
* Payments: Manages payment processing across the enterprise.
* Limits: Provides Limits functionality across the enterprise.
* Cash Management (Planned): Will handle cash flows, liquidity, and integration with other financial systems.

## Support Services

These services complement the core and enterprise services:

* Holdings: A query service optimized for fast retrieval of large volumes of information.
* Product Manager: Enables the creation and maintenance of all the financial products of the bank.
* Party (Customer Proxy): Acts as a proxy for the customer data required by other modular components.
* Market and Reference Data (Proxy): Provides market and reference data integration. It is usually embedded within one of the Temenos modular services and referenced by the other modules.
* Common Subledger (Planned): Will unify financial reporting across different lending services for easier management and auditing.

## Additional Support Services

These services are generic Temenos technological capabilities used across Temenos software and may also be used for Modular services.

* Generic Configuration: Ensures consistent configuration settings across Temenos solutions.
* Adapter Service: Facilitates data transformations and user-friendly interfaces, likely involving both formatting and translating between systems.
* Virtual Table: Configures external entities within the system.
* Event Store: Serves as a single source of truth for all events in the architecture.

The business capabilities should share the supporting services in a larger deployment. For example, if both Retail and Corporate Lending are deployed, they should share capabilities like Holdings, Event Store etc. They should also share the configuration & extensibility user agents as described in chapter Configuration and Extensions.

# Modular Banking Design Principles

## API-First Design: A Strategic Approach to Software Development

In the realm of software development, particularly within the intricate landscape of banking systems like Temenos' Core Banking, the API-first approach emerges as a pivotal strategy. This method revolutionizes traditional practices by prioritizing the design of Application Programming Interfaces (APIs) at the outset of the application development process.

**Key Principles of API-First Design:**

1. Focus on Interactions: API-first design emphasizes the creation of APIs that dictate how an application will interact with external systems. This approach ensures that communication channels are meticulously planned and stabilized from the beginning, fostering compatibility and reliability for future integrations.
2. Decoupling API and Implementation: By separating the API design from the underlying implementation, changes in internal workings of an application do not affect its interaction with other systems. This decoupling enhances adaptability, allowing for updates and modifications without disrupting existing external interfaces.
3. Modular Responsibility: Each application owner is responsible for wiring their APIs to the back-end and managing data transformations. This modular approach ensures that each component operates independently, facilitating easier updates and reducing the risk of system-wide disruptions.
4. Enhanced Scalability and Flexibility: The API-first approach promotes scalability by enabling independent updates to various modules. This modularity allows for the addition of new features or enhancements without requiring a complete overhaul of the system, thus minimizing downtime and risks associated with traditional monolithic architectures.

**Integration with Modular Banking:**

In the context of Modular Banking, where operations are broken down into specialized components, API-first design plays a crucial role in enhancing interchangeability and manageability. Each modular component's well-defined API ensures straightforward integration, allowing external systems to interact seamlessly without delving into internal workings. This separation of concerns significantly improves maintainability and scalability.

**Real-World Application:**

Consider Temenos aiming to integrate mobile banking features into their Core Banking system. Using an API-first approach, they would first define how external systems, such as mobile apps, should interact with these new features via APIs. Subsequent development focuses on back-end services fulfilling these API contracts, allowing frontend developers to concentrate on external interfaces while backend teams manage internal implementations.

**Challenges and Considerations:**

While the API-first approach offers substantial benefits, it also presents challenges. The need for extensive planning and documentation can initially slow down development. Additionally, improper API design may lead to integration issues or maintenance complexities in the future.

**Conclusion:**

The API-first approach stands as a strategic method for developing robust and adaptable software systems, particularly in banking where seamless integration with diverse third-party services is crucial. By prioritizing API design, this approach ensures that external interactions are meticulously planned, fostering reliability and scalability, thereby aligning perfectly with the modular evolution of banking systems like Temenos'.

## Event-Driven Architecture in Modular Banking

In the context of Modular Banking, Temenos employs an Event-Driven architecture to manage state changes efficiently. This architecture facilitates communication between independent modules, ensuring loose coupling and scalability.

**Types of Events in Temenos Architecture**

1. Command Events: These events represent instructions that initiate actions leading to state changes. For example, initiating a transfer or opening an account.
2. Business Events: These denote completed business operations, such as the successful completion of a transaction, reflecting the system's current state post-operation.
3. Data Events: These are triggered by database-level changes, such as updating a customer’s balance, indicating low-level data modifications.

**Modular Banking Context**

The Event-Driven architecture is integral to Modular Banking, allowing modules to publish events without tight coupling with subscribers. Each module can independently handle state changes based on relevant events, ensuring effective communication and operation.

**Example: Customer Transaction**

* Command Event: Initiate a transfer (e.g., "transfer funds").
* Business Event: Transfer completed successfully.
* Data Event: Update customer balance in the database.

**Challenges and Considerations**

* Ensuring all events are tracked and managed without system complexity.
* Proper event handling to ensure timely and appropriate module responses.
* Avoiding conflicts or delays in event processing across modules.

**Conclusion**

The Event-Driven architecture enhances the efficiency and scalability of Modular Banking by enabling effective communication between independent modules through Command, Business, and Data events. While challenges exist, such as managing event tracking and handling, the benefits of loose coupling and state management make it a robust choice for Temenos' banking solutions.

In today's interconnected financial landscape, aligning our company's APIs and events with industry-standard terminology like BIAN is crucial for seamless communication. However, we aim to do this without fully adopting BIAN APIs due to their limitations in quality and relevance.

## Strategic Alignment of APIs and Events with BIAN Terminology

In today's interconnected financial landscape, aligning our company's APIs and events with industry-standard terminology like BIAN is crucial for seamless communication. However, we aim to do this without fully adopting BIAN APIs due to their limitations in quality and relevance.

**Steps to Achieve Alignment:**

* Understanding BIAN Terminology:

Research and comprehend the key terms provided by BIAN relevant to our domain, particularly in banking or financial services.

* Identifying Essential Terms:

Select BIAN terms that are fundamental for communication within our sector. These will form the core of our vocabulary.

* Tailoring the Vocabulary:

Incorporate these selected BIAN terms into our API and event definitions, adapting them to align with our specific business requirements and processes.

* Documentation:

Develop a glossary section in our documentation that clearly defines each term, ensuring it reflects both industry standards and our tailored usage.

* Enforcement in APIs and Events:

Use the BIAN-aligned terms consistently in API method names, parameters, response fields, and event definitions to maintain clarity and consistency. Develop in due time a programmatically-enforced alignment,

* Training for Developers:

Conduct training sessions or workshops to familiarize our development team with the new vocabulary and its application in API design and event handling.

* Versioning Strategy:

Implement a clear versioning strategy for future updates to manage changes in terminology and ensure backward compatibility as needed.

* Resolving Naming Conflicts:

Address any overlaps between existing internal terms and BIAN terminology early to prevent confusion and maintain clarity in communication.

* Testing and Validation:

Establish automated checks or validation processes during API development to ensure adherence to the agreed-upon vocabulary, maintaining consistency across all communications.

Conclusion:

By following this structured approach, we will effectively align our APIs and events with BIAN terminology while tailoring it to meet our specific needs. This will enhance our communication both internally and externally, fostering a clearer and more consistent dialogue within the industry.

**CQRS: Command Query Responsibility Segregation in Temenos Architecture**

The Command Query Responsibility Segregation (CQRS) pattern is employed in the Temenos software architecture to enhance system efficiency and scalability by separating data modification tasks (commands) from data retrieval tasks (queries). This separation allows each responsibility to be optimized independently, leading to improved performance and reliability. For the context of Retail banking with high volumes, CQRS is considered a mandatory performance-enhancing principle.

**Key Components:**

* Core Services
  + Deposits: Manages transactions and updates related to Deposits.
  + Lending: Handles lending operations, including updates and transactions.
* Query Services
  + Holdings microservice: A read-only service that provides fast and consistent data access for high-volume queries related to arrangement accounts, transaction lists, and balances. It is eventually consistent, meaning it reflects the most recent data after a certain period but not immediately.

**Benefits of CQRS:**

* Scalability: Core services can focus on handling complex transactions without being bogged down by numerous read requests.
* Efficiency: Holdings optimizes query performance, ensuring quick and reliable access to data.
* Specialization: Each service can be tailored to its primary responsibility, enhancing overall system efficiency.

In summary, CQRS is a strategic architectural choice that balances scalability with acceptable latency for queries, while ensuring critical command operations are handled efficiently and reliably by core services.

## User Experience in Modular Banking

In the evolving landscape of banking technology, modular banking emerges as a transformative approach, offering significant advantages over traditional monolithic core systems. This shift not only impacts the technical architecture but also reshapes the user experience (UX) for both end-users and internal operations. Temenos' modular banking solutions exemplify this transformation, with a focus on specialized modules and dedicated interfaces.

**Modular Banking Architecture**

Modular banking divides traditional banking functions into smaller, specialized modules, each handling specific tasks like deposits or loans. This approach allows banks to assemble a best-of-breed system by combining various modules from different providers. Each module is designed to be self-contained, exposing APIs for communication with other systems.

**User Interfaces in Modular Banking**

1. Product Configuration User Agent (Product Manager User Agent):

* These are tailored interfaces dedicated to configuring specific product features, such as deposits or lending.
* Product managers use these agents to set up and manage product properties efficiently, ensuring that each module is optimized for its specific function.

1. Extensibility User Agent (Workbench):

* A web-based tool designed for developers and implementation teams to extend the application's functionality.
* This interface allows for customizations and integrations without delving into the core modules, providing flexibility in system enhancements.

1. Operational User Agents (Not currently available for Temenos modular solutions):

* Modular solutions are headless for operations, meaning they lack a built-in user interface.
* Instead, these modules expose APIs, enabling them to communicate with other systems through standardized interfaces.
* The absence of a dedicated UI streamlines the integration process and avoids duplication across multiple modules.

**Centralized Operational User Agent (Bank Owned)**

To address the complexity of managing multiple modular solutions, Temenos advocates for a centralized operational user agent provided by the bank. This interface serves as the single point of access for internal users to manage all banking operations, akin to how internet or mobile banking interfaces serve external customers. Temenos currently does NOT offer such a Centralized Operations User Agent as a productized module. It is expected that such a User Agent will be developed or acquired by the bank and operate as a standalone module in the modular architecture. For Temenos past solutions which were mainly monolithic, such a user agent was available (e.g.: Browser or Transact Explorer). This component made sense when the entire core banking was operated as a single system but is not functional in modular banking.

**Advantages of a Centralized Operational User Agent:**

* Simplifies Navigation: A unified interface allows internal users to navigate across different modules seamlessly.
* Enhances Consistency: Maintains a consistent user experience, reducing the learning curve for transitioning from monolithic systems.
* Facilitates Integration: Eases integration with legacy systems by providing a robust API-driven environment.

While modular banking offers unparalleled flexibility and the potential for best-of-breed solutions, it introduces complexity that could hinder user experience. A well-designed centralized operational UI is crucial to mitigate these challenges. banks must be prepared to invest in robust infrastructure and user-centric design to ensure an intuitive and efficient system.

**Conclusion**

Temenos' modular banking approach prioritizes flexibility and specialization by making modules headless for operations and providing dedicated configuration tools for Product Configuration and Extensibility. However, the success of this model hinges on the bank's ability to develop a strong operational user agent that seamlessly integrates various modules and offers an intuitive experience for internal users. By carefully planning and investing in user-centric design and security measures, banks can harness the benefits of modular banking while minimizing complexity. Currently Temenos does not offer an operational user agent (branch banking) as a standalone product for modular solutions.

### Demo User Agent

We need a demo user to be able to sell to business audiences. In all PoCs so far we have utterly failed to engage with business due to them not understanding how APIs work. The demo user agent should resemble the Centralized operations user agent we expect the bank to have (with placeholders for their other solutions maybe in a few areas).

# Retail Modular Banking Deployment Prerequisites

In this section we need to outline the requirements the bank should have in place before considering our modular solutions. Some of these (to be expanded) are:

* A strong internal capability for integration between components, including strong capabilities with APIs and Kafka, sagas etc.
* A business operations user agent across the bank (since we are offering a headless for operations solution). It must include features such as 4-eyes-principle, audit etc.
* Container capabilities
* General Ledger
* Master CIF
* Data Warehouse, Regulatory Reporting and other downstream systems
* A capability in the DWH or elsewhere to consolidate our raw data from Events (if we don’t include something like Data Hub)
* A workflow engine
* A payments engine

We should be running these prerequisites during deal qualification. If a bank does not meet these prerequisites, then they are not ready for modular banking and should either build the foundations first or consider a monolithic core.

# Technology Foundation

This section describes a set of technology assets owned by Temenos that are a standard part of Modular Banking components. The exact usage of these assets is described for each service separately. In this section, we are describing the generic capabilities of each asset.

## Transactional Outbox

The Temenos Transactional Outbox is a critical component designed to manage the reliable and consistent processing of events within distributed systems. It leverages the outbox pattern, ensuring that all events related to a transaction are processed atomically, meaning if any part of the transaction fails, all associated events are appropriately handled or rolled back.

**Key Features:**

* Unique Identification and Cloud events Standard:

Each event is wrapped in a Cloud events envelope, which includes a unique identifier. This ensures that every event sent out is distinct, helping to manage duplicates and enforce idempotency.

* Idempotency Implementation:

Idempotency is crucial for ensuring that multiple deliveries of the same event do not lead to inconsistent states. Both internal Temenos components and external third-party consumers are expected to handle idempotency, typically by checking unique identifiers against existing records or using state management techniques.

* Event Delivery and Retries:

The outbox pattern ensures events are delivered at least once. Failed deliveries are retried until successful or a threshold is reached. This mechanism guarantees that no event is lost in transit.

* Event Ordering with Event Store:

Temenos uses an Event Store to maintain an ordered history of events, ensuring consumers process them in the correct sequence. This is vital for maintaining consistency in distributed systems where events might otherwise be processed out of order.

**Architectural Fit in Modular Banking Solutions**

Within Temenos’ modular banking solutions, each module generates events that need reliable delivery to other parts of the system or external services. The Transactional Outbox ensures these events are processed correctly, even with retries or failures. Its role is essential for maintaining data consistency and reliability in a distributed environment.

## Event Store

The Event Store is a pivotal microservice within the Temenos ecosystem, designed to support an event-driven architecture. It serves as a central hub for recording, storing, and routing events across various microservices. This section provides a detailed overview of its design, functionality, and integration within the broader system.

**Core Functionality:**

* Event Recording and Routing: The Event Store collects all events generated by microservices and forwards them to the appropriate destinations using the Outbox infrastructure.
* Immutable Storage: It stores events in an immutable manner, ensuring a reliable audit trail for compliance, debugging, and historical analysis.
* Diverse Event Types: Supports various event types such as Command Processed Events, Business Events, and Command Events, each conforming to a well-defined schema for consistency.
* Event ID Convention: Utilizes a structured naming convention (<\_MS Service Name\_>.<\_Operation Id\_>), facilitating quick identification of event origin and purpose.

**Services Provided**:

* Persistent Store: Ensures the ordering and uniqueness of events, critical for maintaining sequence integrity in distributed systems.
* History and Audit Log: Functions as a comprehensive log system, capturing all events to support auditing and compliance needs.
* Replay Mechanism: Enables microservices to replay missed events, ensuring no event is lost or ignored, which is vital for maintaining system consistency post outages.

**Architectural Fit in Modular Banking Solutions**

In the context of modular banking, the Temenos Event Store serves as the backbone for enabling seamless communication between independent microservices. It captures, stores, and routes events generated by various modules, ensuring a reliable source of truth for transactions and operations. This is crucial for maintaining coherence across services, supporting auditing, debugging, and compliance in a banking environment.

The Event Store’s ability to replay events ensures that all services stay synchronized, even after outages. It integrates with microservices through APIs, routing events based on predefined criteria. Designed for scalability, it handles high event volumes efficiently with mechanisms for error handling and retries. Security measures protect sensitive data, ensuring the integrity of financial transactions.

## Service Orchestrator

The Service Orchestrator is an integral component of Temenos’ modular banking solutions, designed to manage transactions across multiple independent microservices. It acts as a conductor, ensuring that each step of a transaction occurs in the correct order and maintains consistency across distributed systems.

**Key Features and Functionality:**

* Saga Pattern Implementation: The orchestrator employs the Saga pattern to manage distributed transactions. This approach allows for coordinating multiple services while handling failures gracefully by triggering compensating actions to undo previous steps if any part of the transaction fails.
* Sequence Management: It defines a sequence of operations, ensuring that each action is executed in a predetermined order. This sequencing is crucial for maintaining data integrity and consistency across different microservices involved in a transaction.
* Compensating Actions: Each action within a sequence has a corresponding compensating action designed to roll back changes if the transaction fails. This ensures that the system can recover to its previous state, preserving data accuracy.
* Integration with Event Store: The orchestrator works alongside the Event Store, which maintains the single source of truth for transaction states. This integration ensures that all participants in a transaction rely on one definitive record, avoiding conflicts and inconsistencies.
* Dumb Pipes and Smart Endpoints: The design emphasizes efficient data transport through “dumb pipes,” with intelligence residing at the endpoints where decisions are made based on events and commands. This architecture promotes modularity and flexibility.
* Failure Handling: The orchestrator monitors for failures during transaction execution, whether from business logic errors or system issues. It triggers failure events and compensating actions to handle rollbacks effectively, ensuring robust fault tolerance.

**Modular Banking Solutions:**

In the context of modular banking, where services like savings accounts, loans, and limits operate independently, the Service Orchestrator is vital for coordinating these components. It enables seamless participation in complex transactions while allowing each service to scale and function autonomously. This approach enhances scalability and fault tolerance, as services can operate independently yet maintain transaction integrity through orchestrated sequences.

## Master Data Access Layer (MDAL)

The Master Data Access Layer (MDAL) is a crucial component in Temenos modular architecture, designed to facilitate data retrieval and management across distributed systems. It plays a pivotal role by providing a structured approach to accessing external data sources, enabling efficient integration within larger applications.

**Key Components of MDAL**

* Data Access Components: These are modular units that encapsulate specific data access operations. Each component can be tailored for different data types or services, allowing independent deployment and management. For example, in a banking application, a Data Access Component might handle customer data retrieval from an external service.
* Request Registry: This acts as the gateway for REST APIs, mapping incoming requests to appropriate MDAL operations. It ensures that each API call is routed correctly, enhancing system efficiency and clarity.
* Master Data Access Handler: Configurable in modes (Embedded, Inline, Pre-composed), this handler manages how data is processed, catering to different system requirements and ensuring flexibility in data access strategies.

**Integration with Microservices**

MDAL integrates seamlessly with microservices architecture by treating each Data Access Component as a potential microservice. This allows for independent deployment and scaling, aligning with broader architectural principles like SOA and microservices. Swagger’s use in defining data structures underscores API best practices, ensuring clear communication between services.

**Redis with MDAL**

Redis is a high-performance in-memory data structure store, often utilized as a caching mechanism to enhance application performance by storing frequently accessed data temporarily. Integrating Redis with MDAL can significantly improve data access efficiency, reducing latency and enhancing overall system responsiveness by providing quick access to cached information.

**Benefits of MDAL to Modular Architecture**

* Independent Development: Components can be developed and updated without disrupting the entire system.
* Ease of Maintenance: Each module’s isolation simplifies troubleshooting and updates.
* Scalability: Efficient handling of responsibilities within each component allows for system expansion as needed.

# Application Architecture

For each modular solution we need at a minimum:

* List of features that includes from business perspective with detailed functional descriptions
* List of BIAN service domains it fulfils (e.g. product types)
* List of APIs and Events available with the solution
* Complete list of components included in the service (Bill of materials)
* Complete list of interfaces with other Temenos components when available
* Expected interfaces we would require during the project (Context diagram in C4)
* Any dependencies
* A roadmap with detailed description of the current setup and then time-specific drops for subsequent features.

## Deposits

### Products and Features Supported

#### Current/Checking and Savings Accounts

Current Accounts: Current accounts are transactional banking products designed for frequent deposits and withdrawals, providing customers with easy access to their funds. They typically offer features such as debit cards, check-writing capabilities, and online banking services, catering to both individual and business needs for daily financial management.

Savings Accounts:Savings accounts are interest-bearing deposit accounts that encourage customers to save money while providing limited access to funds. They typically offer higher interest rates than current accounts and may have restrictions on withdrawals, making them ideal for accumulating savings over time.

Current and Savings Accounts are part of Deposits modular solution from R25 Annual Maintenance Release (AMR). The list below describes the supported features.

**Supported Features**

* Account Creation: Enables opening of various account types, including current and savings accounts.
* Change Product: Allows upgrading or changing an account product, such as transitioning from regular to premium savings.
* Account Maintenance: Facilitates updates to account information throughout its lifecycle via provided APIs.
* Transactions in Account: Supports financial transactions initiated from any front-end system. Deposits triggers events for accounting and reporting purposes.
* Balances and Availability of Funds: Implements trade-dated accounting, updating balances based on transaction booking dates.
* Initial Funding Control and Account Cancellation: Monitors initial funding requirements and handles account closure or penalty fees if conditions aren't met.
* Inactivity and Dormancy Management: Manages account inactivity or dormancy based on specified periods, adhering to local regulations or bank policies.
* Overdraft and Limit Balances: Offers overdraft facilities using secondary limits functionality.
* Interest and Charges Assessment: Supports various interest rates (fixed, floating, periodic) and allows charging fees at regular intervals or rule triggers.
* Tax Handling: Collects taxes on interest and charges based on customer details.
* Posting of Interest and Charges: Billed interest and charges can be capitalized within the account, settled via another TBC-maintained account, or forwarded as settlement instructions to external beneficiaries.
* Notice for Withdrawal of Funds: Enables customers to place notices for withdrawals or account closure in notice accounts, with penalties for unauthorized withdrawals.
* Statements: Configures statement conditions and delivers them via print carrier. Note: XML, CAMT, Swift formats are not supported.
* Posting Restrictions: Allows banks to restrict debits, credits, or both on an account.
* Transaction Rules: Enables configuration of restrictions on transactions such as withdrawals or deposits.
* Account Closure: Supports closure of accounts upon customer request or bank needs, with proper settlements.

**Unsupported Features**

* Eligibility: Handled by external systems (e.g., onboarding solutions), not integrated into Deposits
* Facility Restrictions: Managed via Activity Restriction. Facility Property Class does not apply to Deposits, focusing instead on UI experiences for restrictions like cheque usage.

#### Term Deposit Accounts

Term Deposits: Term deposits are fixed-term investment products where customers deposit funds for a specified period at a predetermined interest rate. These accounts typically offer higher returns than savings accounts but restrict access to funds until maturity, making them suitable for individuals seeking secure, long-term savings.

Term Deposits Accounts are part of Deposits modular solution from R25 Annual Maintenance Release (AMR). The list below describes the supported features.

**Supported Features**

* Deposit Creation: Deposits module facilitates the creation of term deposits for single or multiple owners, leveraging external systems like Party and Reference data for customer information.
* Funding Management: It allows partial or full funding of deposits. If funding isn't completed within a specified timeframe, the bank can close the deposit and return any partial funds, with automated events for accounting purposes.
* Cooling Period: Customers can redeem their deposits shortly after opening without penalties, but after this period, they may face fees or lower interest rates for early withdrawals.
* Interest Options: TBC supports fixed, floating, and periodic interest rates, tailored to different balance levels, offering flexibility in customer terms.
* Charges Management: The system can levy charges regularly or trigger them based on specific activities or rule violations, ensuring adherence to deposit terms.
* Withdrawal Flexibility: Customers can withdraw funds at any time via front-end systems, with events for accounting updates. Early withdrawals may incur penalties or reduced interest.
* Notice Deposits: Requires advance notice for redemptions; failure to provide notice may result in restrictions or penalties, aiding the bank's liquidity management.
* Deposit Simulation: Enables scenario testing throughout the deposit lifecycle without affecting live balances, useful for training and planning.
* Migration Process: Supports transitioning existing deposits from legacy systems to Temenos Deposits modular capability, involving careful coordination and data transfer.
* Statement Management: Configurable statements are available in formats like print carriers, though digital formats such as XML or CAMT aren't supported unless handled externally.
* Rollover Options: Deposits can be manually or automatically renewed upon maturity under new terms, offering flexibility for customers and banks.
* Restrictions on Activities: Banks can impose limits, such as restricting the number of withdrawals per month, to manage risks effectively.
* Dormancy Management: Inactive deposits may incur fees after a specified period, helping banks recover costs associated with maintaining them.
* Early Redemption: Allows customers to redeem deposits before maturity, potentially reducing interest or applying fees to mitigate early withdrawal impacts.
* Deposit Closure: Facilitates the closure of deposits upon customer request or bank policy, ensuring proper lifecycle management.

**Unsupported Features**

Eligibility, Delivery, Correspondence, Alerts, and Activity Messaging are managed externally, possibly within other Temenos modules or third-party systems.

#### Multi-Currency Accounts

Multi-Currency Accounts: Multi-currency accounts are banking products that allow customers to hold, manage, and transact in multiple currencies within a single account. These accounts cater to international transactions and foreign currency exchanges, providing flexibility for businesses and individuals engaged in global trade or travel.

Multi-Currency Accounts are NOT part of Deposits modular solution from R25 Annual Maintenance Release (AMR). Support for Multi-Currency Accounts will be enabled later within 2025. Please contact Temenos Product team for exact release schedule. The list below describes the supported features.

**Supported Features**

* Creation of a multi-currency account: This feature allows for the establishment of multi-currency (MCY) accounts for customers. Customer information is managed within Party, while reference and market data are maintained in the Market and Reference Data respectively.
* Sub-accounts in a multi-currency account: Users can create multiple sub-accounts under a single MCY account. However, only one sub-account is allowed per currency. Sub-accounts are created based on customer requests and are defined by the product outlined in the Sub-Arrangement Rules.
* Sub-arrangement rules: This feature specifies the necessary information regarding the customer, currency, and product associated with the sub-accounts created under the MCY account, utilizing the Sub-Arrangement Rules definition.
* Maintenance of a multi-currency account: Throughout the lifespan of an MCY account, customers or banks can initiate modifications to the account information. These changes are facilitated through APIs, and the system publishes events to notify other systems for subsequent processing.
* Transaction Processing in multi-currency accounts: Financial transactions can be initiated from external payment systems. Credit checks for transactions occur at the MCY account level, and transactions are routed to the appropriate sub-account based on currency. If a sub-account for the transaction's currency does not exist, the transaction is treated as a foreign currency (FCY) transaction and posted to the base currency sub-account. The system can also automatically transfer funds from another currency sub-account to address debit balances, and successful transactions emit events for external GL system consumption.
* Charges in multi-currency accounts: The Deposit TBC allows for the imposition of charges on MCY accounts. Charges can be scheduled or triggered by specific activities or rule violations.
* Change Product: Banks have the capability to upgrade or downgrade an MCY structure through a change product activity. This process also updates the sub-accounts to align with the new MCY product's sub-account specifications.
* Combined Statement: Statement conditions can be configured at the MCY level for combined statements, which are then transferred to the delivery system for printing.
* Multi-currency account closure: Customers or banks can request the closure of specific currency sub-accounts, multiple sub-accounts, or the entire multi-currency arrangement.
* Migration of multi-currency accounts: MCY accounts and their sub-accounts can be transferred from any legacy system to Deposits.

**Unsupported Features**

* Eligibility: The criteria for opening a multi-currency account is not supported within Deposits and must be managed by an external component, either through Temenos solutions like Pricing Engine or third-party onboarding solutions provided by the bank.
* Alerts: While alerts can be configured at both the MCY account and sub-account levels within Deposits based on business needs, the actual alert messaging is sent as an event to an external delivery system in a preconfigured format.
* Accounting: The Deposits modular solution employs soft accounting to update transaction balances within the MCY account. However, the actual accounting entries are managed externally, utilizing events generated during transactions that can be processed by any external accounting system that maintains a General Ledger (GL).

#### Corporate Accounts & Deposits

Corporate Accounts & Deposits product types will be supported from R26.

### End of Day Processing (Close of Business-COB)

The Close of Business (CoB) capability is a critical component of financial operations, designed to ensure that all daily transactions and activities are accurately finalized at the end of each business day. This process encompasses a series of batch operations that execute essential business logic, ensuring the integrity and accuracy of financial data.

During the CoB, various key activities are performed, including the calculation of accruals, which reflects earned revenues and incurred expenses, and the capitalization of interest, which updates the account balances to include interest earnings. Additionally, the CoB process addresses account dormancy by identifying inactive accounts and applying relevant policies. Charges, such as fees or penalties, are also processed during this period, ensuring that all applicable fees are accurately applied to customer accounts.

By systematically executing these batch processes, the CoB capability provides a comprehensive overview of the organization's financial status at the close of each day. This not only enhances operational efficiency but also supports compliance with regulatory requirements, ensuring that all financial activities are properly recorded and reported. Ultimately, the CoB process plays a vital role in maintaining accurate financial records and supporting informed decision-making within the organization.

### Additional Features

**Multi-Company**

The Temenos Retail modular solution includes a multi-company feature that enables the definition and management of multiple companies or entities within the platform. This functionality allows organizations operating under a multi-company or entity model to leverage a single instance of the Retail Deposits, all under one subscription, while maintaining either shared or distinct datasets for various types of information.

Additionally, the service supports intercompany accounting, facilitating transactions between accounts across different companies. This enhances operational efficiency and provides a seamless banking experience for users managing multiple entities.

**Transaction Recycler**

The Transaction Recycler feature enables the automatic retry of failed financial transactions at regular intervals. The system continues to attempt these transactions on scheduled dates until they are successfully settled, the maximum number of retry attempts has been reached, or the retry period expires. When a retry occurs, the account may be partially debited if there is available balance, ensuring that the total transaction amount is eventually processed.

## Lending

### Products and Features Supported

#### Retail Lending

Retail lending refers to the suite of financial offerings designed to meet the borrowing needs of individual consumers. This includes various loan types such as personal loans, mortgages, auto loans, and credit cards. Retail lending products are characterized by their accessibility and are typically tailored to specific consumer needs, providing funds for purposes like home purchases, vehicle financing, or personal expenses.

Retail Lending products are part of the Retail Lending modular solution from R25 Annual Maintenance Release (AMR). The list below describes the supported features.

**Supported Features**

* Loan Creation: Creation of any type of loan contract, fetching details from other systems such as Party and Reference data. Supports both call and term loan contracts for single or multiple borrowers.
* Loan Simulation: Simulation of loan contracts at every stage of the loan life cycle, including creation, prepayment, renewal, payment holiday requests, and payoff calculations.
* Loan Commitment and Top-Up: Definition and storage of terms and conditions of sanctioned loans, along with the ability to offer additional loan amounts as top-ups.
* Loan Disbursements: Manual disbursement of sanctioned amounts, initiated from external payment systems, with event generation for external accounting systems.
* Loan Limit: Management and monitoring of limits handled by an external solution, with the ability to attach sanctioned limits to loans.
* Interest on Loan: Support for multiple interest rate types (fixed, floating, periodic) with various calculation methods, including the Rule of 78 and weighted average rates for tranche disbursements.
* Loan Charges: Ability to levy charges at regular frequencies or based on specific triggers.
* Tax: Tax collection on interest and charges based on rules and customer demographics.
* Scheduling of Payments: Scheduling repayment intervals for interest, charges, and principal, with options to amend repayment terms.
* Overpayments: Accepting overpayment amounts, either as additional scheduled instalments or lump-sum payments.
* Restrictions on Loans: Configuration of restrictions on loan transactions or activities.
* Settlement of Bills: Capitalization of interest and charges to the same account or settlement requests published to external beneficiaries.
* Payment Holiday: Definition of payment holidays for loan contracts, allowing modification or skipping of scheduled payments.
* Advance Payment of Instalments: Ability for borrowers to pay upcoming instalments in advance.
* Delinquency: Automatic aging of overdue bills, calculation of fees or penalties, and production of reminders.
* Renegotiation: Capability to re-negotiate and modify existing loan terms and conditions.
* Interest Adjustments: Automatic recalculation of interest due to back-valued changes or transactions.
* Loan Rollover: Manual or scheduled renewal or rollover of loan contracts based on specific conditions.
* Change Product: Ability to change the product of a loan contract based on customer eligibility.
* Loan Prepayments: Option for customers to prepay portions of the loan principal, with potential charges or restrictions.
* Loan Pre-closure: Early full repayment of a loan with possible charges or restrictions based on configuration.
* Loan Closure: Support for loan closure based on customer requests.
* Loan Charge-off: Ability to fully or partially charge-off loans without impacting customer balances, maintaining charge-off balances and allowing for repayments allocation.
* Loan Write-off: Capability to write off loans when borrowers become insolvent.
* Migration of Lending Arrangements: Functionality to migrate loan contracts from external legacy systems to the Lending capability.
* Alerts: Configuration of alerts on loan contracts with event publishing for external systems to deliver alerts to customers or bank staff.

**Unsupported Features**

* Eligibility: The eligibility conditions and checking for a loan should be handled by an external component, either a Temenos solution (such as Temenos Enterprise Pricing) or a third-party solution (such as the Bank’s onboarding or origination solution).
* Accounting: An external accounting solution is required to handle bookkeeping and accounting entries. Lending uses soft accounting to update balances from disbursements, repayments, etc., and publishes events that should be consumed by the external system for bookkeeping.
* Limit: An external limit system is necessary for handling limit sanctioning, management, monitoring, and other related features.
* Delivery: Delivery and activity messaging are not supported by Lending and should be managed by an external system that consumes events published by Lending.
* Alerts: Delivery of alerts to customers is managed by an external system based on events consumed from Lending. Once processed, the external system sends an event back to Lending.
* Loan Securitization: Features related to loan securitization are not currently applicable in Lending.

#### Corporate Lending

Corporate Lending product types will be supported from R26.

### End of Day Processing (Close of Business-COB)

The Close of Business (CoB) capability is a critical component of financial operations, designed to ensure that all daily transactions and activities are accurately finalized at the end of each business day. This process encompasses a series of batch operations that execute essential business logic, ensuring the integrity and accuracy of financial data.

During the CoB, various key activities are performed, including the calculation of accruals, which reflects earned revenues and incurred expenses, and the capitalization of interest, which updates the account balances to include interest earnings. Additionally, the CoB process addresses account dormancy by identifying inactive accounts and applying relevant policies. Charges, such as fees or penalties, are also processed during this period, ensuring that all applicable fees are accurately applied to customer accounts.

By systematically executing these batch processes, the CoB capability provides a comprehensive overview of the organization's financial status at the close of each day. This not only enhances operational efficiency but also supports compliance with regulatory requirements, ensuring that all financial activities are properly recorded and reported. Ultimately, the CoB process plays a vital role in maintaining accurate financial records and supporting informed decision-making within the organization.

### Additional Features

**Multi-Company**

The Temenos Retail Lending modular solution includes a multi-company feature that enables the definition and management of multiple companies or entities within the platform. This functionality allows organizations operating under a multi-company or entity model to leverage a single instance of the Retail Deposits modular solution, all under one subscription, while maintaining either shared or distinct datasets for various types of information.

Additionally, the service supports intercompany accounting, facilitating transactions between accounts across different companies. This enhances operational efficiency and provides a seamless banking experience for users managing multiple entities.

**Transaction Recycler**

The Transaction Recycler feature enables the automatic retry of failed financial transactions at regular intervals. The system continues to attempt these transactions on scheduled dates until they are successfully settled, the maximum number of retry attempts has been reached, or the retry period expires. When a retry occurs, the account may be partially debited if there is available balance, ensuring that the total transaction amount is eventually processed.

## Common Business Application Components

### Holdings microservice

The Holdings microservice is an integral part of the application architecture designed to enhance data accessibility and system efficiency. Specializing in Retail Deposits, as well as Retail Lending, this read-only service adheres to the CQRS (Command Query Responsibility Segregation) pattern. By focusing on query operations, Holdings provides high availability and low latency for data retrieval, making it ideal for scenarios where real-time updates are not critical.

**Key Features**

* Read-Only Service: Primarily serves data through queries without handling write operations.
* Eventually Consistent: Ensures that the system will reflect the most recent state after a certain period, though not immediately.
* CQRS Implementation: Segregates command and query responsibilities to improve scalability and performance.

**Functionality of Holdings microservice in Modular Banking**

* Arrangement Accounts (BIAN-Aligned): Manages individual contracts or agreements between the bank and customers, such as deposit agreements or loan contracts.
* Transactions List: Provides a detailed history of transactions for each arrangement or account.
* Balances Management: Offers current and historical balance information for each contract.

**Operational Benefits**

* Higher Availability: Ensures reliable data access with minimal downtime.
* Lower Latency: Delivers fast response times for efficient query processing.

**Considerations and Challenges**

* Data Consistency: Requires mechanisms to handle delays in updating data from the source of truth.
* Error Handling: Must have strategies to detect and recover from query failures without significant user impact.
* Efficient Data Retrieval: Utilizes indexing, caching, and load balancing to optimize query performance.

**Conclusion**

The Holdings microservice is crucial for providing fast and reliable access to account-related data within a modular banking architecture. By leveraging CQRS and focusing on read operations, it supports key use cases while ensuring scalability and availability. This design contributes significantly to the overall efficiency and reliability of the banking platform.

### Party microservice

The Party microservice is an essential component designed to manage customer and business entity information within the modular banking framework. Unlike the master data management systems that handle comprehensive customer records, the Party microservice focuses on specific subsets of data relevant to individual modules such as retail lending or deposits.

**Functionality and Benefits of Party microservice in Modular Banking**

One of its key features is its ability to provide tailored APIs that allow accessing critical customer details like IDs and ages, which are essential for functionalities such as linking accounts and assessing product eligibility. The service seamlessly integrates with the Master Data Access Layer (MDAL), ensuring real-time updates through Redis caching via MDAL events. This integration enhances efficiency by maintaining accurate and up-to-date data.

In the context of modular banking, the Party microservice is highly relevant as it enables modular solutions to access necessary customer information without relying on the full master data set. This setup promotes scalability and flexibility within the banking architecture.

**Deployment optionality**

Moreover, the Party microservice offers optionality. If a bank can effectively utilize tailored APIs and manage Redis caching through MDAL events, deploying the Party microservice becomes optional. This flexibility allows banks to streamline their architecture, reducing operational complexity and overhead based on their specific needs.

## Product Design

Product design and modification activities take place via a user interface called Product and Pricing Manager. The product configuration is stored in a microservice called Products microservice. For more details on the product configuration please refer to section 3 Configuration and Extensions.

This part will be expanded in chapter 3 as Product Design is part of the SDLC.

# Deployment Architecture

## Cloud Native Deployment

Temenos modular solutions leverage a cloud-native architecture built on cutting-edge technologies, utilizing containers and microservices. Each application component is packaged into a self-contained unit through containers, ensuring efficient deployment and scalability. The microservices architecture enhances flexibility by allowing individual services to be developed, deployed, and scaled independently.

This approach capitalizes on cloud-native managed services such as AWS Elastic Kubernetes Service (EKS), Azure Kubernetes Service (AKS), and OpenShift. These platforms handle significant portions of the underlying infrastructure, reducing operational burdens and enabling seamless integration with advanced cloud features.

The benefits of a cloud-native deployment are substantial, offering superior scalability, easier updates, and enhanced fault isolation, which align with contemporary DevOps practices. This setup is ideal for institutions aiming to stay at the forefront of technological innovation, providing access to the latest features and improvements.

Temenos modular solutions utilize advanced cloud-native technologies to ensure seamless deployment and efficient management. Key features include:

* Helm Charts for Deployment: Temenos modular solutions employ Helm charts, a Kubernetes package manager, to streamline the installation and management of applications on Kubernetes clusters. This tool enhances the ease and consistency of deploying applications in a cloud-native environment.
* Infrastructure as Code (IaC) Automation: The platform supports automation through IaC, allowing banks to tailor deployments to their specific infrastructure needs using provided samples. This approach ensures consistency across environments and minimizes manual errors, thereby enhancing operational efficiency.
* Flexibility with Service Choices: While Temenos modular solutions are designed to work with native managed services from cloud providers, they also offer the flexibility of using open-source components if preferred by the bank. This adaptability caters to varying technical expertise and infrastructure requirements among different financial institutions.

## Leveraging Infrastructure Capabilities for Temenos Modular Solutions

Temenos modular solutions are designed to operate efficiently within modern cloud infrastructures, leveraging advanced technologies such as Kubernetes and various database systems to ensure high availability (HA), disaster recovery (DR), and scalability. By utilizing Kubernetes, Temenos can orchestrate containerized applications, enabling seamless deployment, scaling, and management of services. This orchestration allows for automatic load balancing and self-healing capabilities, which are crucial for maintaining uptime and performance in a dynamic environment.

Moreover, the choice of robust database solutions, including both SQL and NoSQL options, enhances the resilience and scalability of Temenos applications. With cloud providers like AWS, Azure, and OpenShift, Temenos can take advantage of managed database services that offer automated backups, replication, and scaling features. This setup not only supports rapid response to changing workloads but also ensures that data is safeguarded against failures. Consequently, organizations can confidently rely on Temenos modular solutions to deliver financial services that are not only responsive and flexible but also secure and reliable.

### AWS: Temenos Modular Solutions on AWS

Temenos modular solutions on AWS utilize a variety of databases, including Amazon RDS for SQL databases and Amazon DocumentDB for NoSQL requirements. Applications are primarily deployed on Amazon Elastic Kubernetes Service (EKS). For message queuing, AWS recommends using Apache ActiveMQ 5.18, while the preferred Pub/Sub platform is Amazon Kinesis. Microservices are designed to run on Amazon ECS (Elastic Container Service) or EKS. The SQL database for microservices can be Amazon RDS, while NoSQL options include Amazon DocumentDB or PostgreSQL.

### Azure: Temenos Modular Solutions on Azure

In Azure, Temenos modular solutions leverage several supported databases, including Azure SQL Database, Azure Database for PostgreSQL, and MongoDB 8.x. Applications are deployed on Azure Kubernetes Service (AKS). The recommended message queue service for Azure is Apache ActiveMQ 5.18, while Azure Event Hub serves as the preferred Pub/Sub platform. Microservices can be deployed on Azure Container Apps (ACA) or AKS. For microservices, the SQL database is Azure Database for PostgreSQL, and NoSQL database options include Azure Database for PostgreSQL (using JSONB) or MongoDB 8.x.

### OpenShift: Temenos Modular Solutions on OpenShift

For OpenShift, Temenos modular solutions support a variety of databases, including Oracle 19c, MS SQL Server 2022, PostgreSQL 17, and MongoDB 8.x. Applications are deployed on OpenShift Kubernetes. The recommended message queue services are Apache ActiveMQ 5.18 or Red Hat AMQ Broker 7.12. For Pub/Sub, IBM Event Streams 11.5.x and Red Hat AMQ Streams 2.x are the preferred options. Microservices in OpenShift are deployed on OpenShift Kubernetes, with PostgreSQL 17 serving as the SQL database. For NoSQL needs, microservices can utilize PostgreSQL 17 (with JSONB) or MongoDB 8.x.

## Deployment Architecture: Deposits

This section can be written only when a full BoM and other aspects of application architecture are available.

## Deployment Architecture: Retail Lending

This section can be written only when a full BoM and other aspects of application architecture are available.

# Configuration and Extensions

## Product Configuration

## Extensibility Framework

The Extensibility Framework is a crucial component of the Modular Banking Solutions, enabling developers to customize and extend Temenos solutions to meet specific business requirements. This framework allows seamless integration with various internal and external systems, facilitating faster product and service innovations

**Key Features**

1. **Data Extension:** The Extensibility Framework supports the extension of Transact data models by adding user-defined data elements (fields). These extensions are propagated to APIs and become part of the Open API Specification as name or value pairs in the 'extension' group.
2. **Data Addition:** Developers can add new tables tailored to specific business needs using JSON schemas. The Virtual Table Microservice manages these tables, defining properties like data type and lookups through Temenos Workbench. These tables are updated via generic APIs or Python scripting.
3. **Validation and Defaulting:** The framework offers robust validation mechanisms via regular expressions for simple validations and Python scripting (preprocessors) for complex business rule validations.
4. **Response Formatting:** The framework offers a capability to format JSON responses to API calls via Python scripting (postprocessors).
5. **Interfaces:** The Adapter Framework Microservice, based on Apache Camel, provides out-of-the-box templates for various interface use cases:

* Event to API & API to Event: Facilitating communication with external systems via protocol transformations
* Simple Bulking: Efficiently handling bulk data transfers.
* Flat File Debulking: Processing large datasets from flat files.
* API Transformations: Customizing API responses to meet specific needs.

## Temenos Modular Workbench

In the Temenos ecosystem, the Workbench serves as the user agent for the Extensibility Framework (EF), providing essential capabilities such as importing configurations and artifacts, extending or creating new ones, and pushing these extensions back to the Core. It also integrates with the Temenos Packager for packaging and deployment. Traditionally, each service or module required a separate Workbench instance, which could become cumbersome.

To address this inefficiency, especially in the context of Modular Banking where multiple cores exist, the Modular Workbench has been developed. This innovative solution allows a single Workbench instance to connect to multiple components through configurable profiles. These profiles enable developers to switch seamlessly between different services or modules within the same interface, eliminating the need for multiple Workbenches and enhancing productivity.

The benefits of this modular approach extend beyond a single UI. It simplifies workflow management by allowing easier handling of multiple configurations and updates. Collaboration among teams is also improved, as they can work more cohesively within a unified environment. The integration with the Temenos Packager ensures smooth packaging and deployment processes across multiple components/modules.

While potential challenges such as managing dependencies or conflicts between modules exist, the modular approach offers significant advantages in scalability, flexibility, and efficiency. It allows banks to adapt quickly to changing requirements and deploy updates smoothly while maintaining consistency across different services or modules.

In summary, the Modular Workbench represents a forward-thinking solution that streamlines operations, enhances productivity, and supports the dynamic needs of Modular Banking. By leveraging configurable profiles and integrating seamlessly with essential tools like the Temenos Packager, it offers a robust framework for managing multiple components efficiently within a single interface.

## Business Logic Extensions using Java

We need to decide whether EF is sufficient, or we have to go back to Java. Assuming that EF is what we should do is in my view completely baseless:

* EF has objectively failed to deliver anything except enforcing a good behavior around interfaces
* Tier 1 or 2 banks may not be satisfied with light touch customizations, especially in the beginning where inevitable our offering will not be very feature rich.
* If we do use java we must develop and updated SDK with restrictions (e.g. ban DELIVERY customizations, restrict external API calls to 1 per routine etc.)
* **We need to re-evaluate the entire customization/testing approach in the era of GenAI. It is likely that GenAI can solve the customization problem (building java extensions, testing, refactoring during upgrades) much better than Extensibility Framework ever did.**

## Extensibility Approach for Modular Solutions

# Integration Architecture

## Integration Approach

Temenos' modular solutions are designed to be flexible, scalable, and efficient, leveraging modern integration practices to ensure seamless communication between different components. The architecture emphasizes the use of APIs and events as primary mechanisms for interaction, ensuring that each module remains focused on its core business logic without being bogged down by integration concerns.

**APIs and Events for Communication**

Temenos modular services communicate through well-defined APIs and events, which standardize interactions and reduce friction between components. APIs provide synchronous communication, allowing modules to request data or trigger actions in a direct manner. On the other hand, events enable asynchronous communication, making it possible to handle real-time data streams and decouple systems that need not interact synchronously.

This approach ensures loose coupling between modules, which is crucial for maintaining scalability and flexibility as new services are added or existing ones are updated.

**Externalization of Integration Logic**

To maintain clarity and prevent the core business logic from being diluted by integration concerns, Temenos enforces a strict policy against embedding integration logic within modular services. Instead, all integration-related code and configurations must be explicitly externalized. This separation ensures that each module remains focused on its primary responsibilities, enhancing maintainability and scalability.

**Locations for Externalized Integration Logic**

There are three primary locations where integration logic can reside:

* Temenos Extensibility Framework: The extensibility framework serves as a central hub for extending and customizing Temenos solutions. It provides a set of tools and interfaces that allow developers to extend functionality without modifying the core modules.
* Industry Standard Middleware: Many banks already have established integration platforms such as iPaaS (Integration Platform as a Service), ESB (Enterprise Service Bus), or custom in-house middleware. Temenos' modular services can seamlessly integrate with these existing systems, allowing for a gradual transition to the new architecture while leveraging current investments.
* Custom Microservices (Adapters): For scenarios where no suitable middleware exists, custom microservices can act as adapters. These custom services abstract integration logic, enabling seamless communication between Temenos modules and external systems. This approach is particularly useful in highly customized environments or when integrating with legacy systems that do not support modern APIs.

**Benefits of the Integration Approach**

* Decoupling Business Logic from Integration: By externalizing integration logic, each module remains focused on its primary business responsibilities, ensuring cleaner codebases and easier maintenance.
* Support for Existing Middleware: The architecture accommodates a wide range of integration tools, allowing banks to leverage their existing investments in middleware and reducing the learning curve for transitioning to Temenos solutions.
* Customizability and Flexibility: The option to use custom microservices provides the necessary flexibility to handle unique integration requirements, ensuring that Temenos' modular solutions can adapt to diverse banking environments.

**Conclusion**

Temenos' integration architecture is designed to be robust, scalable, and adaptable, supporting a variety of integration scenarios through APIs, events, and externalized logic. By strictly separating business logic from integration concerns and providing multiple avenues for integrating with existing systems or custom solutions, Temenos ensures that its modular services remain efficient and relevant in dynamic banking environments.

This architecture not only enhances the functionality of Temenos' solutions but also positions them as a future-proof investment for financial institutions looking to stay competitive in an ever-evolving market.

## Modular Banking APIs

In the realm of modern banking technology, Modular Banking APIs represent a cutting-edge solution designed with an "API first" approach, ensuring accessibility and usability from the outset. These APIs adhere to the REST architectural style, utilizing JSON payloads for data interchange, which enhances readability and ease of parsing.

These APIs are organized around specific product definitions, such as savings accounts or loans, each having its own set tailored to manage their lifecycle processes. This includes creation, management, and other essential operations. Beyond product-specific functionalities, the APIs also encompass core financial operations like balances, transactions, and authorizations, catering to retail deposits, accounts, and lending solutions.

Deployed independently from underlying business logic components, these Modular Banking APIs operate with versatility. Their design ensures they are agnostic of the specific banking system they interface with, allowing seamless integration with both modular solutions and larger core banking systems offered by Temenos. This flexibility makes them a robust choice for diverse financial operations, ensuring compatibility across different technological landscapes.

In summary, Modular Banking APIs offer a powerful, adaptable, and efficient means of managing various financial products and services, providing banks with the tools they need to stay competitive in an ever-evolving market.

### APIs for Deposits

### APIs for Retail Lending

## Modular Banking Events

Our Business events need to match in terms of data structure to the APIs. This means we must refactor all our first class business events to align with the new APIs. This may be a significant amount of work.

Due to business events not being customizable, we must include currently Data Events capability to easily cover any gaps.

### Events for Deposits

### Events for Retail Lending

## Exporting Data

Using raw Events as the foundation of exporting data may not be sufficient. While technically possible, it will greatly add to the project duration.

Using Data Hub is not a feasible option

We likely need a “Data Product” as a new component, in line with some of the latest trends around Data Mesh. This should be tailored to each modular solution.

## Country specific variants of APIs and Events

We need a strategy for implementing country specific variants (also business logic when required)

We cannot simply copy existing CMB functionality into the modular solutions because:

* Each modular solution may have a very small country level footprint
* Existing implementations may rely on dependencies that are no longer true (e.g. there might not be sub-ledger in Lending, so existing Transact implementations that use sub-ledger features won’t work).
* The APIs for the CMB are built as extensions to IRIS

## Internal Temenos Interfaces

When we sell multiple components, they must be pre-integrated to a degree, otherwise no one will buy them. For example if we sell Retail Lending and Accounts, the expectation would be that we have a plan for the disbursement implementation or for the repayment implementation. IT is OK for these pre-integrations to have an external dependency that is going to happen during the project but we must provide at least accelerator assets and a clear plan to the bank and implementation teams rather than re-inventing the wheel.

We need to decide who owns these interfaces internally

**Composability**

* The user agent should provide the user with the ability to perform all tasks / actions allowed in all licensed / deployed business capabilities, by accessing a single user agent. That is not signing in to a deposits user agent to perform deposits task and to lending user agent to perform the lending tasks. Even if we deliver the business capabilities as headless, we will still have the “demo” user agent for demos on business audience.
* Have a Single Customer View on the user agent
* If both the business capabilities are deployed, the user should be able, in the User Agent, the define the payin / payout accounts, at the settlement instructions, in a similar way that you do in Transact.
* The transfer of funds should between accounts residing in different business capabilities should happen automatically, without requiring an external (e.g. a Payments system) solution. For example, disbursement of a loan, payment of a loan, (ad-hoc) charges collection and so on.
* If both business capabilities are licensed and deployed and one of them is “down” for maintenance. What should happen in these cases?
  + There are two approaches
    - Have a stand-in solution, which will provide fund authorizations when a system is “down” and then, synchronize the data
    - Deny the transaction
  + I would be more in favor of the first approach but, a) I don’t know in detailed how our Funds Authorization Stand-in works and if it covers the needs and, b) there will be an increase in our cost and possibly some delay in transactions.
* How our SaaS Platform will allow maintenance of multiple business capabilities, is something that should be assessed.
* How Reference & Market Data MS COBs are synchronized? What is the order?
* How Reference & Market Data are shared between the two business capabilities? And how things that are unique to each Business capability are hidden when the respective capability is not licensed / deployed?
* All the above should be considered in cases where a) The bank deploys at the same time both business capabilities, b) The bank deploys one of the business capabilities at the beginning but adds the other one in the future and, c) The bank stops the subscription to one of the two business capabilities.

Also, irrelevant to composability, but important:

* How we should be delivering the data to the banks, when they stop the subscription, especially on SaaS. We should have a “framework” capability to extract a full dump of data, of all components, and logs
* We don’t currently have an out of the box integration in the business capabilities with our exchange partners, especially on SaaS, like formpipe for advices / statements and card management / processing.

### Interface between Retail Deposits and Retail Lending

### Interface between Enterprise Pricing and Deposits

### Interface between Enterprise Pricing and Retail Lending

# DevOps

## Packager

The Temenos Packager is a critical framework designed to manage the Software Development Life Cycle (SDLC) of business configurations across Temenos Products. Its primary objective is to facilitate the consistent and reliable promotion of business configurations from development environments (DEV) through testing environments (SIT, UAT) to production (PROD). This framework is essential for business configurations that cannot be deployed as "business as usual" activities in production, such as setting up payment systems or defining complex financial rules.

The Temenos Packager consists of two key components:

* Data Packager: This component manages the storage and deployment of business configuration data. It enables the storage of business configurations in source control systems in a readable and scripted format. The Data Packager provides APIs to deploy these configurations across environments, ensuring consistency and reducing manual intervention.
* Code Packager: This component focuses on managing Java code projects for Temenos' Modular Solutions It provides templates for setting up new projects, integrating them with source control systems, and enabling their inclusion in CI/CD pipelines using Maven.

The Temenos Packager plays a pivotal role within the Modular Banking framework by efficiently managing changes across various domains, ensuring seamless integration with existing tools like the Temenos Workbench. This powerful tool caters to three primary areas of change management:

* Configuration and Extensions via Extensibility Framework: The Packager supports the extension of configurations through Temenos' extensibility framework, allowing for customized solutions tailored to specific business needs.
* Product Configuration with Product Manager: It integrates with the Product Manager, enabling the systematic configuration of product settings and features, ensuring that all changes are tracked and deployed consistently across environments.
* Code Development using Java API: The Packager handles code written in Java, leveraging APIs to facilitate seamless integration into the development lifecycle, from coding to deployment.

By encompassing these areas, the Temenos Packager offers a unified solution for managing configurations and extensions within the Modular Banking context. Thus, Modular Banking will benefit from a robust SDLC that enhances scalability, reliability, and futureproofing of its offerings.

## Integration of Packager within Temenos Modular Workbench

The Packager is seamlessly integrated with the Temenos Modular Workbench, providing a unified interface where developers can manage configurations and extensions efficiently. This integration consolidates essential tools into one accessible point, eliminating the need for switching between multiple interfaces and simplifying the development process. Enhanced by Git integration directly within the Workbench, version control becomes an integral part of the workflow, ensuring consistent and reliable code management.

This unified approach not only streamlines the Software Development Life Cycle (SDLC) but also supports efficient promotion of business configurations across environments, from development to production. By fostering a seamless experience, the combination of Packager and Workbench enhances productivity, reduces errors, and ensures scalable configuration management. This integration is particularly valuable for the evolving Modular Banking offering, as it maintains consistency and scalability in managing complex configurations and extensions.

# Application Security

Temenos modular solutions are deployed on industry-standard Kubernetes clusters, utilizing standard database services to ensure robust and scalable infrastructure. All communication between services is secured using Transport Layer Security (TLS), a cryptographic protocol that provides privacy and data integrity over the internet.

Infrastructure security is managed through mechanisms chosen by the financial institution during the deployment process. This approach allows banks to tailor security measures according to their specific needs and compliance requirements, ensuring flexibility and customization.

Application security focuses on three key areas: authentication, authorization, and encryption.

* Authentication involves verifying the identity of users or systems accessing the application, which may include methods such as username/password, multi-factor authentication, or biometric verification.
* Authorization ensures that only authorized users have access to specific resources or operations within the application, often implemented through role-based access control (RBAC).
* Encryption secures sensitive data both at rest and in transit, encrypting data stored in databases and ensuring all communications are encrypted using TLS.

These security measures collectively protect the integrity and confidentiality of banking operations. Detailed explanations of each area will be provided in subsequent sections to ensure a comprehensive understanding of the security framework in place.

## Modular Banking approach to application security

The modular banking approach represents a significant shift from traditional monolithic core banking systems, particularly in terms of application security. This approach decouples the presentation layer from the business logic, making it headless from a business operations perspective. As a result, end-users primarily interact with a branch banking application, which acts as their main interface, rather than directly engaging with the modular core solutions.

**Key Benefits and Considerations:**

* Decoupling Presentation Layer:
  + The separation of presentation logic from business operations allows for greater flexibility in user interface design and integration with third-party applications.
  + This decoupling enhances scalability as each module can operate independently, focusing on specific banking functions without exposing core services directly to users.
* Shift in Security Focus:
  + Due to the headless nature, the branch banking application becomes the primary focus for security measures, handling authentication, authorization, and encryption effectively.
  + The modular core solutions prioritize securing APIs and ensuring data integrity since they communicate with other modules or external systems via these APIs.
* Access Control and User Interaction:
  + A small number of technical users interact directly with the modular banking application for configuration and management tasks, reducing the potential attack surface.
  + Stringent access controls are essential to ensure that only authorized personnel can manage these modules, given their critical role in system operations.

In light of the above architecture, the application security features for Temenos Modular banking solutions are designed as described in the following sections.

## Authentication

Temenos modular solutions leverage Keycloak, an open-source Identity and Access Management (IAM) tool, as an identity broker to manage authentication. Keycloak facilitates Single Sign-On (SSO), allowing users to log in once and access multiple applications seamlessly. This approach enhances security by centralizing authentication processes, reducing the administrative burden on individual applications.

Keycloak integrates with the bank's existing Identity Provider (IdP), such as Active Directory, which handles user management and password storage. This integration typically occurs via protocols like SAML or Open ID Connect, commonly used in enterprise environments for secure identity management.

By externalizing authentication through Keycloak, Temenos modules avoid handling login processes directly, ensuring consistent security policies across applications. This setup streamlines user provisioning and de-provisioning while maintaining a federated identity management solution that complements the bank's existing infrastructure.

In summary, Keycloak acts as an intermediary between Temenos modules and the bank's IdP, providing a secure and efficient authentication framework that aligns with enterprise security standards.

## Authorization

We need to decide SMS vs XACML. We also need to decide what are the expectations from authorization during the project.

## Encryption

Temenos employs a layered approach to encryption, tailored to different stages of data lifecycle:

1. **Data at Rest Encryption:**

* Mechanism: Transparent Data Encryption (TDE) supported by databases like MS SQL Server and Oracle.
* Details: TDE encrypts the entire database, including logs and backups, ensuring that data remains encrypted even when stored on disks or during backup processes.
* Implementation: This encryption is managed at the database level, often using keys stored in secure key management systems.

1. **Data in Transit Encryption:**
2. Protocol: Utilizes TLS (Transport Layer Security) versions 1.2 and 1.3 for securing data exchanged between components.
3. Configuration: Implements industry-standard TLS certificates with recommended cipher suites to ensure secure communication channels. Banks are advised to use well-known certificate authorities for issuing these certificates.
4. **API-Level Encryption:**

* Integration: Leverages third-party encryption tools available through Temenos Exchange, allowing banks to enhance security as needed.
* Tools: These pre-integrated tools provide additional layers of API-level encryption, ensuring that data exchanged via APIs remains secure and compliant with specific bank policies or regulatory requirements.

**Interactions Between Encryption Types:**

Each encryption type operates independently but works together to ensure comprehensive security. Separate keys are typically used for different encryption processes (e.g., TDE vs. TLS), though some infrastructure can be shared, such as key management systems.

**Compatibility and Security Considerations:**

Temenos ensures compatibility by standardizing encryption protocols and tools. The modular architecture allows each component to handle its specific security requirements while maintaining seamless integration.

Regular updates and patches are applied to maintain security standards, particularly for TLS configurations, ensuring ongoing protection against emerging threats.

This structured approach ensures that Temenos banking solutions provide robust security across all data states, from storage to transmission and API interactions.

# Observability

In the realm of banking technology, observability is crucial for ensuring the operational health and efficiency of deployed systems. Temenos modular banking solutions are designed to provide robust observability through a suite of integrated tools that facilitate monitoring, debugging, and performance analysis.

At the core of Temenos' observability capabilities lie three essential libraries:

1. Logging: The Log4j library is employed for generating logs within Temenos components. These logs capture detailed information about system operations, enabling effective troubleshooting and insight into component behaviour.
2. Metering: As part of OpenTelemetry, this functionality is dedicated to collecting and managing metrics. It provides quantitative data on system performance, helping administrators understand resource utilization and identify bottlenecks.
3. Tracing: As part of OpenTelemetry, tracing allows for the tracking of transactions across various services within the banking ecosystem. This is vital for understanding end-to-end processes and ensuring seamless operation.

**Integration with Bank’s Centralized Observability Infrastructure**

Temenos components are designed to seamlessly integrate with a bank's existing centralized observability infrastructure. The logs, meters, and traces generated by Temenos modules can be channelled into the bank’s data collection, storage, analysis, and visualization tools. This integration ensures that all necessary operational data is consolidated and accessible within the bank’s monitoring framework.

**Monitoring Dashboards**

A monitoring Dashboard based on the open-source stack above is being planned. The dashboard will be offered on top of Grafana as an accelerator for banks that are using the popular platform. As of R25 AMR, this dashboard will not be available.

**Importance for the Bank**

The provision of integrated logging, metering, and tracing tools by Temenos is instrumental in supporting effective system management. These capabilities enable banks to:

* Enhance Monitoring: Gain comprehensive insights into the performance and health of deployed modules.
* Streamline Debugging: Quickly identify and resolve issues through detailed logs and transaction traces.
* Ensure Compliance: Maintain regulatory standards by providing auditable records of system operations.

In conclusion, Temenos' observability solutions are integral to the efficient management and maintenance of modular banking systems. By leveraging these tools, banks can ensure optimal performance, robust security, and compliance with industry regulations.

# Implementation Approach

We need to supply a clear implementation approach to our teams that is tailored to modular banking. In the past our implementation teams quoted a higher price for modular solutions compared to Transact (!!!!!!) due to contingencies and lack of knowledge.

We might need to select a specific team (even name colleagues) that will spearhead this effort and train them properly.

# Data Migration

While we can use the standard data migration tool we would use for any project, we need to specify what is the expected scope for migrating data (in terms of specific tables). This is required for project-level estimations and to keep the implementation time low.

We must revisit our ancient guidance that we don’t migrate historical transactional data. We can copy the historical transactional data into a non-volatile copy of Holdings and easily satisfy the reporting requirement.

# Go Live Considerations

Are there any specific go live considerations considering there may be stronger dependencies?

# Application Support

Who will support these solutions? Should we trained named people to support? Since a lot of concepts fundamentally change, we cannot use standard resources that know Transact for application support without dedicated training.

# Upgradeability

What are our expectations from banks here? Are we offering annual release option? Quarterly?

# Appendix 1: Additional details on modular architecture

## Deposits

A single component will be used for all types of Deposits, both Retail and Corporate. There could be differences in the individual products, with corporate variants have more complexity, but all Deposit and Account products will be implemented in a single component.

## Retail Lending

Retail Lending will not include Limits and Collateral functionality. This is expected to be externalized.

## Payments

Currently, Payments Hub will be used for Payment Initiation & Execution if required. It is expected that banks embarking on a journey to work with modular solutions will already have a standalone Payments capability in place before starting to replace Core components. Any Payments solution can be used as long as it has real-time interfacing capabilities. Currently Payments solution is not implemented with the latest architecture guidelines for modular banking described in this document. Refactoring Payments to satisfy these guidelines is a possible future project.

## Limits

Limits will be implemented as a standalone capability and not embedded in Retail Lending or Corporate Lending. For all secured Lending projects, the existence of Limits management capability should be a prerequisite. As part of future developments, Temenos could also embed Collateral functionality within Limits domain. It is possible to use any 3rd party Limits solution the bank already owns.

## Sub-Ledger

Sub-ledger will be implemented as a standalone capability and not embedded in Deposits or Lending solutions. A single sub-ledger can manage all accounting requirements across Temenos modular solutions. It is technically feasible to use a 3rd party Sub-Ledger, however the complex configuration required may increase project effort and timelines.

## Market and Reference Data

Market and Reference Data will not be a standalone deployed capability initially. The reason for this is the current implementation has a significant footprint that does not justify the value. Market and Reference data will be deployed within the first Temenos modular solution to be part of a project. When a second modular solution is deployed, it will access (via MDAL) market and reference data from the first modular solution. The same applies to customers that already have Transact and wish to deploy a modular solution on the side. As long as Transact is on a sufficiently high release to ensure compatibility, the modular solution can retrieve (via MDAL) the market and reference data from existing Transact.

At some point in the future, Market and Reference data must be rewritten from scratch as lightweight microservices, without Close of Business and the heavy infrastructure of the main core components.