Thought Machine

Migrating to Vault Core: Migration Strategies

Vault v4.4

September 2022

Contents

About this document

Purpose

This document is part of the Vault Core Migration document suite, which in its entirety helps you understand how to migrate data from your legacy banking system into Vault Core.

This document specifically provides further detail regarding potential Vault Core migration strategies; however it may prove useful in the wider context of the migration from legacy systems across the bank's existing estate.

Scope

This document addresses the following topics:

- Key migration strategy options
- Summary migration strategies
- Posting migration approaches
- Product migration approaches

This document applies to **Vault Core release 4.4.**

Out of Scope

This document does not address the following topics:

Topic	**Where this topic is covered**	**Description**
Migration APIs	*Migrating Data to Vault Core: Migration APIs*; a PDF document made available in the Vault Core release pack.	Guide outlining the functional behaviour of the Vault Core migration APIs, including; - Migration deployment and setup - Data Loader API - Postings Migration API
Running a Migration Programme	*Migrating Data to Vault Core: Migration Programme Lifecycle*; a PDF document made available in the Vault Core release pack.	Guide outlining the key elements of the migration programme lifecycle and the delivery support model provided by Thought Machine in relation to Vault Core migration programmes.
Observability over Migration Events	*Migrating Data to Vault Core: Technical Monitoring (Self-Hosted)*; a PDF document made available in the Vault Core release pack.	Guide outlining how to monitor a Vault Core migration event using the Observability tools that Thought Machine deploys as part of a standard self-hosted deployment.

Vault Core Data Dictionary	*Vault Migration Data Dictionary*; an Excel sheet made available in the Vault Core release pack.	Spreadsheet containing field-level detail for the migration APIs necessary to support migration data mapping, including; field descriptions, formats, validation rules, and differences to equivalent BAU APIs
Data Loader API Specifications	The Thought Machine Documentation Hub.	Specifications for the Data Loader API including Requests, Streaming Topics, DLQs, REST API Calls.
Postings Migration API Specifications	The Thought Machine Documentation Hub.	Specification for the Postings Migration API including Requests, Streaming Topics, REST API Calls.

Audience

This document is intended for use by clients and partners that are involved in Vault Core migration programmes.

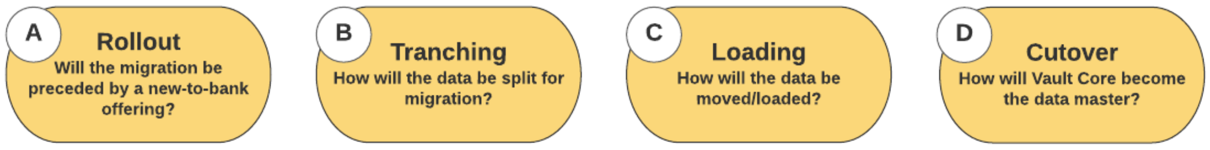
Disclaimer

Thought Machine makes no claims, promises or guarantees about the accuracy, completeness or adequacy of this document. All information, content and materials are provided 'as is' and without any representation or warranty of any kind, express or implied, including (but not limited to) the implied warranties of merchantability, fitness for a particular purpose, title, or non-infringement. To the extent permitted by applicable law, Thought Machine does not accept liability for any direct, indirect, special, consequential, exemplary, punitive, or any other losses or damages of any kind, including (but not limited to) any loss of profits, business interruption, loss of data or otherwise, even if expressly advised of the possibility of such damages.

Migration strategies

Starting migration questions

When helping you to plan your migration to Vault Core, Thought Machine recommends first considering the below 'menu' of potential migration options against a set of starting questions. By taking a view of each of these you can start to define your overall migration strategy:


A
Rollout
Will the migration be preceded by a new-to-bank offering?
B
Tranching
How will the data be split for migration?
C
Loading
How will the data be moved/loaded?
D
Cutover
How will Vault Core become the data master?

A. Rollout

Will the migration be preceded by a new-to-bank (greenfield) offering?

Greenfield followed by migration

A popular deployment option when moving to a new core is to decouple product build/launch from migration activities for all (or a subset of) products. Undertaking a greenfield product deployment for new-to-bank relationships as a precursor transition state to the migration of the back-book.

Vault's Smart Contracts make this approach simpler than traditional cores due to their flexibility and simplicity to launch into production. This can be achieved without delaying the overall migration route-to-live which will often be longer due to broader scope and complexity.

Benefits	**Challenges**
• Quicker time-to-market for new to bank customers • Allows for a period of production proving for the new product(s) • Opportunity to highlight and address any challenges on a relatively small but growing population, in advance of full migration • It may allow for an easier transition for the business where there are new or changing business processes associated with the new core	• May extend the programme timelines (and therefore costs) due to longer time to reach back-book migration and introduction of additional work in new transition state • Will likely require architectural coexistence to bring together the front and back book data for the period that it is maintained in separate cores

Migration only

The alternative to a greenfield-first migration is to migrate directly onto the newly created product (Vault Smart Contract) without a greenfield deployment as a precursor transition state.

This approach may be the only option available depending on the programme scope and context, and regardless, similar benefits to a greenfield precursor state can be achieved by undertaking friends and family/trial migrations ahead of go-live.

Benefits	**Challenges**
● Programme solely focused on single deployment event, which may result in quicker execution of back book migration	● Does not allow for the additional risk mitigation achieved through early production-proving via greenfield deployment

---

B. Tranching

How will the data be split for migration?

Bulk

Migrating all data within the scope of the programme in a single tranche is hereafter referred to as a **bulk ETL** (Extract, Transform, Load) approach. In this approach, all migrating data will be moved as part of a single migration event, often completed over a weekend, potentially supported by pre-event and/or post-event loads of data.

Benefits	**Challenges**
● Quicker route-to-live compared with tranching - shortening and simplifying the overall programme route to live ● Removes architectural coexistence complexity in moving between two cores during a period of tranched migrations	● May add time to the migration event due to the requirement to extract, transform and load all data at once. Where the event is longer there is also an increased likelihood of data change between the initial extract and cutover - this can be mitigated by more frequent delta updates and/or business process changes to restrict possible data change over the event ● Consolidates risk in that if something were to go materially wrong during the migration event (for example,. service downtime in one of your services) then all customers could be impacted.

Incremental

The alternative to a bulk ETL approach is to incrementally move the data in multiple predefined tranches. These tranches may be based on account complexity, increasing volume, or any other criteria as defined by the migration programme.

Benefits	**Challenges**
• De-risks the migration programme because not all customers or accounts can be impacted if issues are experienced over any single migration event • May enable quicker route to live for a subset of customers/accounts where tranching is based on migration complexity	• Overall programme route to live will likely be longer, extending programme costs. • Likely a requirement for architectural coexistence, for example reporting may need to be aggregated during the incremental migration period to provide a complete view of all customer assets/liabilities consistent with the current experience of mastering the accounts on a single core.

C. Loading

How will the data get moved to Vault Core?

Loading via APIs

The primary means of loading data to Vault Core is using the two dedicated migration APIs (the Data Loader API and Posting Migration API - described further in the *Migrating to Vault Core: Migration APIs* guide). Following data extraction and transformation, requests are sent to the Vault migration Kafka topics, after which the load process is automated including the streaming out of Kafka events.

Given the wider API-focused architecture that you will be using with Vault, an API-led approach should be a familiar concept by the time you start to think about migration. Taking this approach over a file-based data transfer, which is more common for migrations onto traditional cores, enables greater load control and manipulation compared to these legacy systems.

This should be considered the default migration load approach for Vault Core migrations, with the other options considered as exceptions based on the migration scope.

Benefits	**Challenges**
● Enables the benefits provided by Vault's migration APIs, in particular: ○ Migration-specific field validation/logic that better supports migration scenarios ○ Relaxation of asynchronous validations that are not relevant in a migration context ○ Performance improvements over equivalent BAU journeys ○ Other unique behaviour, for example, sequencing of loads based on configurable Dependencies so data can be sent in any order	● Fixed overhead in building and testing the ETL routine and orchestrating migration events

Onboarding/Offboarding

An alternative to automatically loading via the Data Loader API and Posting Migration API is to onboard/offboard accounts by closing on the legacy system and then reopening as new accounts in Vault Core. The opening of new accounts on Vault Core could use either the migration or BAU APIs depending on the use case. This could also be achieved using industry schemes such as the UK Current Account Switching Service or other similar schemes that exist globally.

Benefits	**Challenges**
● This approach may be preferable or even necessary where products differ considerably between source and target and customer Terms & Conditions and/or product experience is changing drastically ● Removes overhead of building and testing bespoke ETL pipeline ● Utilises existing account closure and opening processes on source and target systems, and potentially existing industry schemes	● Likely to have very high customer impact due to the standard offboarding/onboarding touchpoints (customer comms, channel re-registration, and so on) ● No historic data, in particular transaction history, will be present in Vault after onboarding ● For industry schemes there will be limited flexibility in how the switchover service operates locally (SLAs, customer notifications, supported products, and so on) and it is likely that regulator engagement would be needed to ratify such an approach

Manual

Another alternative to an API-led load is the manual keying of data via a client-provided User Interface (UI) into Vault. This is normally reserved for smaller scale complex product migrations where the fixed effort of an automated IT solution outweighs the cost/time required to migrate manually. Vault Core does not provide this UI and it would need to be built separately as part of the client migration programme.

Benefits	**Challenges**
● Removes overhead of building and testing bespoke ETL pipeline	● Only practical for very low volumes of data ● Requires controls to ensure quality of manually migrated data ● Requires development of a simple User Interface to load data into Vault using the Data Loader API

D. Cutover

How will Vault Core become the data master?

The final lens is the cutover approach; the focus of cutover is how, when and to what extent does Vault Core become the data master and operationally 'live'. The nature of the cutover activity will vary depending on scope, but could include: switching on downstream feeds (e.g. to reporting systems), routing Vault Core data to customer facing channels, switching payment/card scheme routing to Vault Core, etc.

IMPORTANT: Cutover will have an inter-dependency with your transition and target state architecture so please ensure that you include your architects in the relevant discussions.

Load and cutover together

Coupling load and cutover together is a common migration approach. This represents the final stage in the migration journey where you extract, transform and load the data and cutover as part of the same migration event.

Benefits	**Challenges**
● In many cases this will represent the simplest and most straightforward migration option ● Quicker route-to-live compared with	● Greater risk as more activity is compressed into a single event compared with a decoupled approach ● Less time to identify issues with the load

decoupled approach	through reconciliations and business testing than via the decoupled approach

Load and cutover separately

An alternative to coupling load and cutover is to separate these into distinct events. Common applications of a decoupled approach include:

- Pre-loads, where the majority of in-scope data is loaded well ahead of cutover and is maintained in the intervening period by periodic deltas; or
- Parallel Run migration strategies, whereby data is loaded and maintained via real time feeds in a 'shadow state' for a limited period of time to prove consistency of outputs with legacy.

Benefits	**Challenges**
• De-risks the overall migration by spreading load and cutover activity across separate events • Allows additional time for proving load success and enables creative overall migration strategies such as parallel run	• Extends overall route to live and associated programme costs • May introduce additional delivery or architectural complexity depending on the nature of decoupled load (for example, additional ETL for delta files, set-up of real time feed for parallel run, and so on)

Common migration patterns

Using the 'menu' of options presented in the previous section you can create an overall migration strategy that suits your requirements. Common examples of migration strategies considered by Vault Core clients are outlined in the section below.

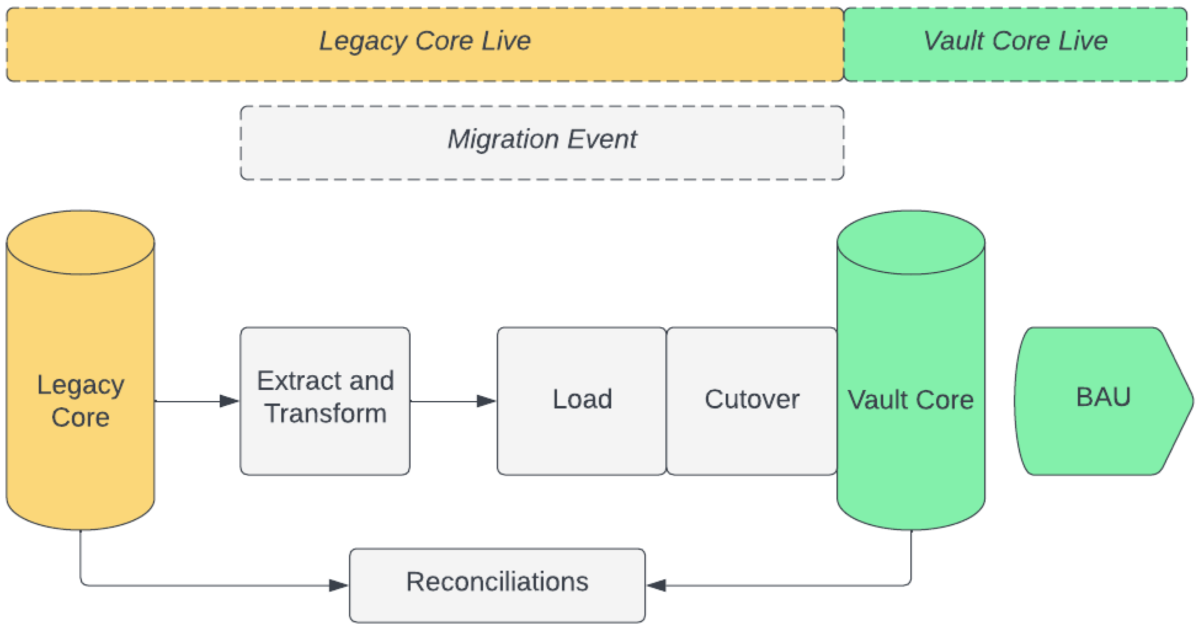
---

1. Big-Bang Migration

The big-bang migration has historically represented the most common but often also the highest risk approach to a core banking migration. Vault Core's design can support the de-risking of a big-bang migration if this is your preference, though we would always recommend considering the alternatives before committing to this.

A big-bang migration against the menu of options in the Migration options section could include:

- Rollout - Either Greenfield followed by Migration or Migration Only
- Tranching - Bulk
- Loading - Loaded via APs
- Cutover - Load and Cutover Together



Big-bang migrations follow a common pattern:

- A single or bulk ETL event practised multiple times in dry runs/dress rehearsals in the lead up to the migration event.

- An event normally completed at a low traffic period (for example, Sunday evening), which in many cases comes with associated service downtime.
- Linked to the point above, a big-bang migration approach will often have a migration programme principle or aim of completing the event as quickly as possible to reduce the impact of this service downtime.
- The re-routing of traffic from the old core to Vault Core will be done as part of the event.
- An event that represents a replacement of ‘old for new’, there is no coexistence and/or shared capabilities.

When considering a big-bang migration, Vault Core’s design brings with it a number of added benefits that should be factored into your big-bang migration event planning:

- Near real-time streaming API reconciliations and feedback - if something has not loaded as expected you are notified in real-time and can take immediate remedial action using Vault Core’s idempotent event-based design. There is no need to wait for the end of the attempted load before getting a total result or output.
- Vault’s time-series testing can be used to replicate legacy product behaviour in 'fast-forwarded' tests. This provides confidence going into a big-bang live event and acts as additional assurance testing on Transformation logic.
- The design of the Data Loader API enables greater flexibility in your load of data. There is purposefully more limited field validation compared to the Core API and you can use Dependency Groups to better manage load orchestration.

Similar to the parallel run migration approach outlined below, when considering a big-bang migration it is important to consider the benefits and challenges that come with taking such an approach:

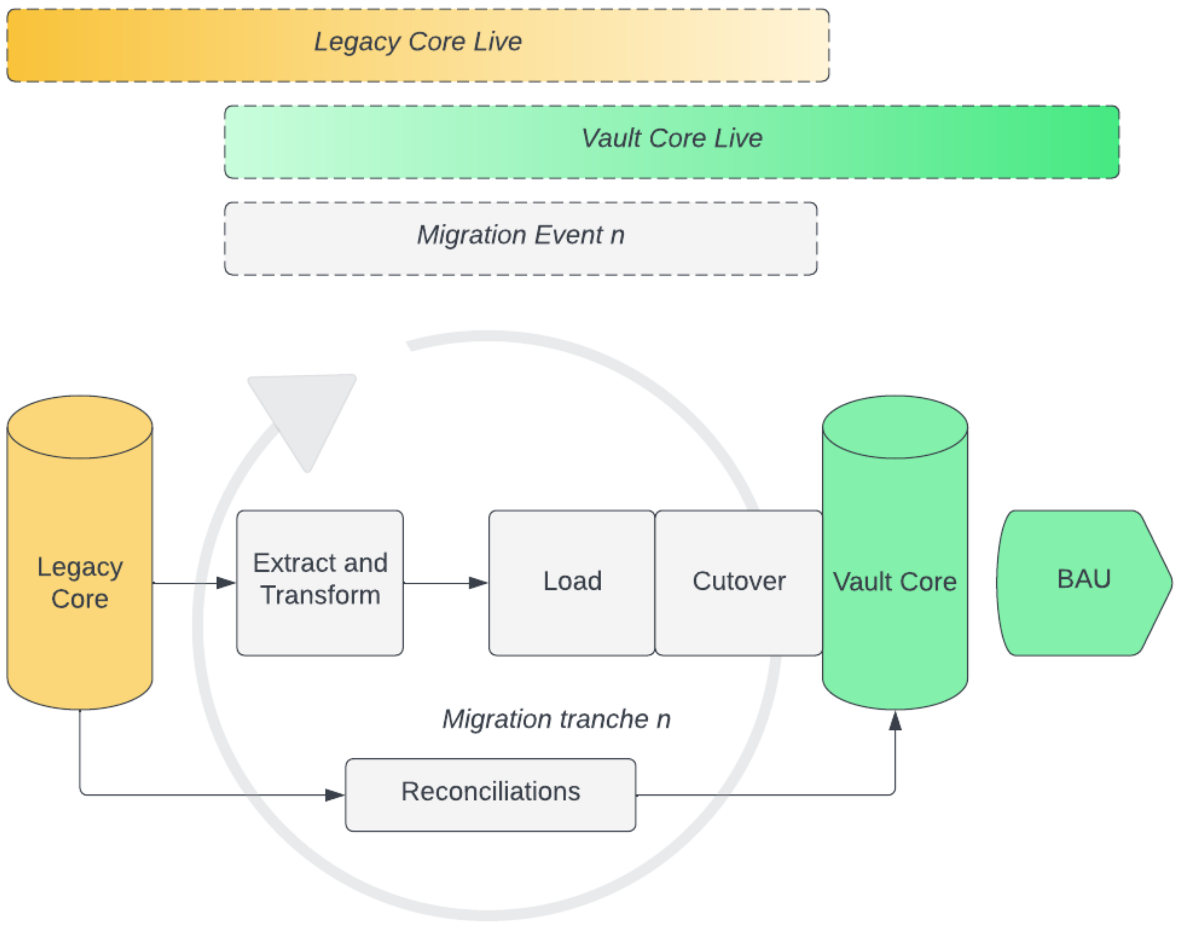
Benefits	**Challenges**
Big-bang migration represents the simplest approach that your stakeholders will be most familiar with.	It is an 'all or nothing' event, and thus carries more risk.
Less orchestration is needed compared to other approaches. For example, there is no need to generate and ‘join together’ transaction reports across two cores that you would need to do during a phased migration.	Requires more thorough testing because you are more likely to only get one opportunity at doing the live event immediately after the point of no return.
Likely to be less costly as a programme due to its comparative simplicity and quicker route to live.	The event is likely to be longer because you are moving all data in one go.

2. Phased Migration

A Phased migration is a migration that is broken into multiple parts or tranches - normally to reduce risk compared to a single event.

A Phased migration against the menu of options in the Migration options section could include:

- Rollout - Either Greenfield followed by Migration or Migration Only;
- Tranching - Incremental;
- Loading - Loaded via APs;
- Cutover - Load and Cutover Together



Phased migrations follow a common pattern:

- The migration is broken into multiple parts or 'tranches' based on a defined 'unit' of migration (explained further below).
- The trigger for loading a tranche could be a pre-agreed route-to-live plan with tranches aligned to specific dates, or ad-hoc migrations triggered at a more granular

level (including account level). For example, bank relationship managers initiating following discussions with a customer - this should trigger a workflow that results in a set of Vault Core migration API calls.
- The ETL process can otherwise be quite similar to big-bang, with the load and cutover activities occurring together for each tranche.
- Data can also be loaded in a single event for all accounts, be maintained in a dormant state via regular deltas, and cutover on a tranche by tranche basis. This can be useful so that ETL routines do not have to be run as many times and can also support when extract routines/capabilities are more limited on legacy (for example, all or nothing data pull). Vault Core supports a concept of Tranching Accounts, which is a data attribute that can be tagged against Accounts to then control the execution of Schedules against these Accounts at the tranche level. In the context of a Phased migration this can be useful in pre-loading multiple tranches in a dormant state and then selectively cutting-over one tranche at a time.
- Phased migrations introduce additional complexity over big-bangs in that a level of 'architectural plumbing' is often needed to bring together outputs from the legacy and new core during the transition period; for example, for financial reporting.

The 'unit' of migration for these tranches will vary depending on the programme context, and could also vary from tranche to tranche. There are a variety of 'units' of migration that can be used within an Incremental ETL (customer, account, product, attribute), which are outlined further in Appendix A.

A key consequence of a phased migration (at product or customer level) is the requirement for core coexistence. Coexistence refers to the state where 2 or more cores exist at the same time or in the same place:

- It represents Vault Core alongside one or multiple legacy cores as part of a bank's architecture landscape
- It is a transitional or target state for a bank
- It may form part of a wider transformation journey
- It might operate for an extended period of time

During the coexistence period, both cores are responsible for mastering a proportion of the overall migration programme scope. This results in a requirement to feed inputs to and bring together outputs from both cores during the transition period. The exact set-up will depend on your architectural intent, budget, legacy capabilities and the end to end length of the phased migration. Modern architecture enables simpler integration and many banks will already have the primary setup requirements due to Open Banking or Payment Services Directive 2 type initiatives.

Coexistence can be a complex topic requiring a deeper level of architectural design. However, in the context of migration it may represent an opportunity to avoid a big-bang migration event. If you would like to discuss the topic of coexistence in more detail please reach out to your Thought Machine contact for client architecture support.

Benefits	Challenges
Lower risk than a big-bang migration with a proportion of customers or accounts migrated at each point to spread risk.	Additional cost and complexity of running multiple events.
De-risks a migration event with at least some core capabilities remaining on legacy core.	Likely require more extensive technical architecture planning and execution compared to other migration approaches.
Provides an opportunity to more quickly achieve Vault Core benefits while leaving more complex and long-running pieces until later in the transformation.	Linked to the above, the programme cost of running in a coexistence state for an extended period will need to be factored in when comparing it to other options.
If tied to product events (for example, end of interest accrual period), it can simplify in-flight management.	Complexity in providing a complete product or business picture during the migration period.

If you are considering entering a state of coexistence as part of your wider transformation programme, some key lessons learnt based on past experiences include:

Planning

- Establish a dedicated coexistence central team (made up of PMO and SMEs) that have sufficient senior sponsorship (steerco) and are empowered to make the difficult decisions, while aligning business strategy with program execution. Recognising that this could be a multi-year journey, tots of teams across a programme/bank are impacted by coexistence and there is often no perfect answer and so quickly and appropriately managing conflicts is key.
- Involve the impacted business teams as early as possible so the change is done with them rather than to them. Make RACI (responsible, accountable, consulted, and informed) clear to all impacted parties to ensure buy-in and accountability.
- Drive the intended coexistence vision and outcomes by having measurable KPIs e.g. customer satisfaction metrics/NPS and/or reporting on the amount of double keying needed and the proportion of effort (FTE) this is consuming.
- Given the complexity and nuances involved, general governing principles will be more effective than rigid, “one-size-fits-all” rules.
- Overinvest in a rigorous data mapping process: ensure rights to update data are 100% mapped to avoid costly reconciliation issues.
- Alignment on data definition is critical: current data practices (e.g. data definition nomenclature) may be inconsistent across the organisation, validate this first and make it consistent.

- Technology capacity planning is critical – do not neglect the legacy system, and ensure the new system's unique requirements are well understood.
- Identify the right partner(s) that can assist in this coexistence journey.

Process & Implementation

- Define the overall process for managing coexistence; that is, how will instances of a split customer experience be designed and federated out to impacted teams.
- With modern cloud architectures and loosely coupled technology stacks, do not assume a manual process to support coexistence is necessary. Codify workarounds where possible and appropriate.
- Operationalise and embedded processes as early as possible ahead of entering the formal coexistence state.
- Documentation is important but not the goal with a transitory or temporary coexistence state.
- Coexistence will likely place extra demand on Operations teams; therefore, build in extra capacity in teams to account for learning new coexistence workflows, troubleshooting and systemic inefficiencies. Embed automation as the first class citizen for daily operations and release process.
- Structure and train teams to ensure you will have no single points of failure along the operations chain.
- Conduct several test runs on copies of real bank data to identify edge cases.
- Define a proper change management process to provide visibility on the program and avoid ad-hoc surprises from affected teams or systems.
- Establish a data migration and integration squad to look specifically into data migration and governance. Be mindful of the impacted CIF and master data, accounts, products and reports.
- Place strong emphasis on testing automation and data reconciliation during the coexistence phase. Establish a QA team to govern functional and non-functional KPIs.

---

3. Parallel Run Migration

A Parallel Run migration against the menu of options in Migration options could include:

- Rollout - Either Greenfield followed by Migration or Migration Only
- Tranching - Either Bulk or Incremental
- Loading - Loaded via APs
- Cutover - Load and Cutover Separately

Within a Parallel run migration both the legacy core and Vault Core are operating together or in parallel for a limited time period within a production proving exercise. The legacy core will remain the data master and the Vault Core will operate in a 'simulated state', receiving the same inputs and producing the same outputs as the legacy core, but is not operationally live to customers. During the parallel run period, reconciliations between the cores prove whether the Vault Core is operating as expected. When proven, upstream and downstream routing is switched to make Vault Core the operational data master - this could be via an Access Control system to identify which core is the data master for each Account ID.

The key point to remember with a parallel run is that you are decoupling or separating the Vault Core load process from the cutover, with load taking place a predefined period ahead of cutover with the parallel run occurring between the two.

Parallel Run Definition

While parallel run is an optimal approach where deep proving of product behaviour is required, it is important to recognise the additional complexity that comes from running two cores, even if only one is a data master at any one time. Therefore, before deciding on a parallel run approach there is a need to consider the degree to which you will complete a parallel run and ensure this is agreed upfront with stakeholders.

For a parallel run, most clients consider matching all or almost all activities or events across legacy and Vault Core. While this may appear optimal, it introduces many additional challenges: consider the following three approaches that could be used to support parallel run (representative of approaches rather than distinct options). This does not mean there are only three distinct options; in almost all cases a client's particular adoption of a parallel run will fall at a point in between each of these levels described.

Parallel Run 'Lite'

The slimmest form of a parallel run where the focus is on proving:

- Key product lifecycle (scheduled) events - for example, interest accrual or charging of fees to the account.
- Completion of 'core' financial movements - Do external payment inflows and outflows match? For example, if account X receives two authorisations from a hotel in a single day, are these both of the right values and categorised as the correct posting type and rebalancing within Vault Core completed as expected?

The scope of the 'lite' parallel run will depend on the critical success factors defined by the programme. The simplification of this type of parallel run adds to increased risk, as two complex architectures are involved; end-to-end low level financial outputs or reconciliations are not proven during the period, e.g. repayment hierarchy outputs and the proving of interest being repaid first followed by fees, to generate identical outputs or consequences across all accounts on both Vault Core and the legacy system.

Parallel Run 'Mid-Way'

As the name suggests this is the mid-way point for a parallel run. This approach focuses on proving key product lifecycle events (e.g. interest accrual) as well as ensuring consistent financial reconciliations, ultimately leading to no customer impacts (beyond any that are pre-agreed).

When taking this approach you are readily accepting and planning for a level of discrepancies due to:

- The different behaviour between the two cores - the aim of the migration may be to exactly replicate the product behaviour and outcomes between the two cores. However, this may not always be possible: for example, Vault Core does not accept zero value postings but the legacy core may do so. An exact replication would leave the accounts in the same state but, depending on the reconciliations approach, the number of transactions or PIBs on both cores for that account may differ.
- Potential differences of behaviour introduced by the parallel run - the parallel run itself may generate differences in the outputs of the two cores. A simple example is the difference in the timestamp for a posting processed by the legacy core before it is passed through various services to being processed by Vault Core. For end-of-day reconciliations, this could result in a different view of postings processed in a single day across both cores.

These discrepancies need not be a problem if the approach is agreed up front. The goal is not to align low level legacy core methods or behaviour, but rather prove that the outputs are eventually consistent and/or discrepancies are known and accounted for.

Parallel Run 'Full'

The final high level category is a full parallel run. This is deployed where the bank seeks to exactly replicate outputs and behaviour between legacy and Vault Core. All operational and financial reconciliations must exactly match, even when the ultimate result is not detrimental to customers.

Adopting this approach increases complexity as Vault Core is moving away from simply proving product behaviour to exactly replicating technical behaviour of legacy core. To reach this 'pure' parallel state is difficult to achieve, and the programme needs to carefully balance between exactly matching outputs and behaviour against the valuable risk mitigation it provides to migration. From the outset, do consider that there may be some technical behaviour that cannot be replicated due to the inherent differences in the two cores.

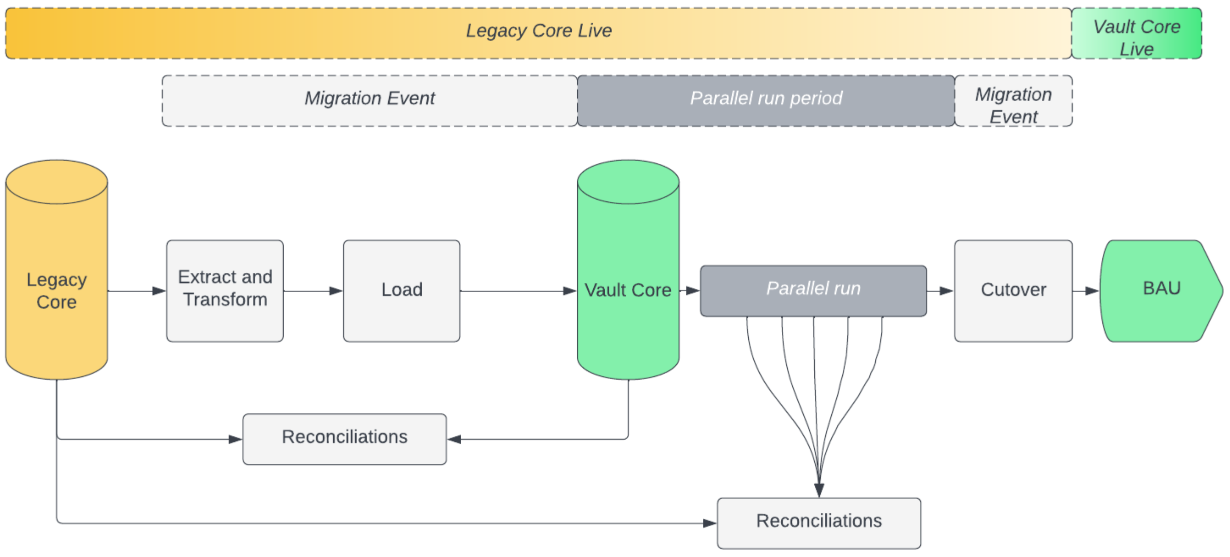
Using the example above, Vault Core cannot easily be adapted to accept zero value postings like the legacy core and, in this approach, the bank will need to either adapt behaviour of the legacy core simply for the parallel run or submit low value postings to Vault Core that are then offset and balanced to zero. This would need to then be accounted for in reconciliations.

Summary

When considering the parallel run approach to take, keep in mind what you are trying to prove. When chosen at the beginning of a migration programme, parallel run is often not an exercise to prove that Vault Core can replicate the exact technical behaviour of the legacy core. Instead, it is undertaken to replicate the product behaviour across both cores. When defining an approach, consider carefully what are the key proving points and success criteria that need to be achieved; balance costs and benefits appropriately.

Parallel Run Migration Event Execution Approach

Parallel run migration diagram



A parallel migration to Vault Core could take place as follows:

1. Legacy data is extracted, transformed and loaded to Vault Core following the Vault Core migration methodology covered in the *Migrating to Vault Core: Migration APIs* guide, but is not made operationally live via a cutover event. The legacy core remains the live data master, servicing customer requests as needed.
2. The data moved in step 1 is reconciled. In reality this may happen in parallel to the ETL process as part of the load event.
3. A data pipeline then synchronises any activity taking place on the legacy core **after** the initial data load (for example, a transaction) and sends the same data in parallel to the Vault Core in real time.
   - This could be either by intercepting the messages upstream of the legacy core and routing these to Vault Core in addition to the legacy core, or alternatively taking outputs from the legacy core and sending these to Vault Core
   - The nature of these API calls will need to be defined in advance, and may include only a subset of calls that are deemed valuable for production parallel testing (for

example, not including customer details update requests).

4. Automated (or partially automated) reconciliations run to compare outputs across both legacy and Vault Core on a predefined frequency. This is to ensure that Vault Core is producing the outputs (for example, statement data inputs) that are expected when compared to the legacy core system.
5. Finally, when it is deemed appropriate the routing will be switched (cutover) so that all traffic is **only** sent to Vault Core from this point onwards. At this point Vault Core has become the system of record (SoR) and data master for that product.

The above steps will vary depending on your risk appetite and available technical tooling. They represent only one way of performing a parallel run migration and the exact setup will vary depending on the exact situation.

Key questions when considering a parallel run migration approach

Some key questions that must be asked early on and influence a parallel run approach are:

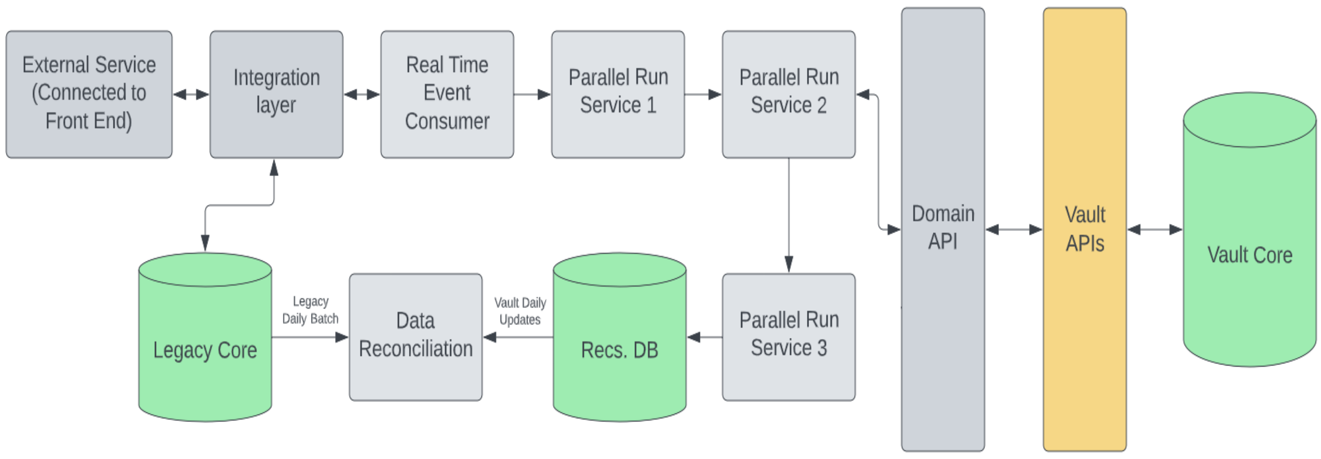
Question	**Comments**
Will the real-time feed be taken from upstream or downstream of the current core?	Where taken upstream (i.e. intercepting the messages received by the current core and sending to both in parallel) you can test Vault Core's ability to both accept and reject Postings, and where taken downstream (i.e. an output of the current core) you will be testing only acceptance and not the rejection logic.
Do historic Postings need to be migrated or can they be 'hydrated' during the parallel run period?	Where only a subset of relatively recent historic Postings are in scope for migration to Vault Core (e.g. one month) it is possible to orchestrate the parallel run period, such that it lasts for as many days of history as required. A starting balance position is all that is required at the point of data load, and the Posting history hydrates through a real-time feed during the parallel run period, and is 'complete' by the point of cutover.
Will the real-time feed replicate all possible changes on source or only a subset?	Based on the intended outcomes, you may choose to only send a subset of updates to Vault Core during the parallel run period. Consider whether updates to Payment Devices are a valuable addition, or could these be migrated as a delta update just ahead of cutover, for example.
Which APIs will be used during the parallel run period (BAU or migration)?	The migration APIs should be used for the initial data load (so as to benefit from the improvements of performance and lack of Smart Contract execution). The BAU APIs should be used for parallel run period updates (as Smart Contract execution must be executed).
How will ordering of messages be guaranteed?	It is important to ensure messages are processed by Vault Core in order where the Smart Contract logic is being executed (e.g. pre-posting hooks that will check available balance positions).

	One approach is ordering messages using Kafka hashing, with wallet_id as a key, to ensure all transactions for a single digital wallet are sent to the same Vault partition and processed in order.
How will failures or issues that occur during the parallel run be treated?	Parallel Run reconciliations may flag issues that require remediation. A treatment strategy should be agreed as well as a threshold for backing out of the parallel run period where appropriate. You could consider an automated remediation pipeline, which runs on a lag after issues have been identified and applies a pre-defined fix (re-sending missed messages, applying a corrective Posting, etc.).
What are the pre-agreed reconciliation discrepancies, if any?	During the course of Vault Core Smart Contract development, there may be agreed differences in the technical or product behaviour between legacy and Vault Core. This needs to be pre-agreed with key stakeholders so that these discrepancies can be discounted or ignored during parallel run reconciliations.
How will open authorisations (Postings) be treated over the initial data load?	Where authorisations have a defined lifecycle, for example they are guaranteed to be closed after X days (by Vault Core or by an external process) it may be easier not to migrate open auths at all and accept a known discrepancy in available balance positions in the recs for a short period after the initial load completes.
What proportion of the book will be put into a parallel run state?	It does not need to be the case that the full migrating book is migrated to Vault Core and put into a parallel run state with legacy. Instead, a proportion of the book can be migrated as part of the parallel run strategy.

Parallel Run Potential Architecture During Proving Period

While it cannot be stated exactly what the high-level core banking architecture will look like during a parallel run, one potential solution could be as follows:

Parallel run - example architecture



- Customer interaction (spending, deposit, change of details, etc.) hits the BAU integration layer and goes through the BAU request / response process via legacy core, which remains the data master during the parallel run phase.
- The Real-Time Event Consumer picks up the responses from the legacy core once they hit the integration layer, and also sends these to the Parallel Run microservices
- Parallel Run Service 1 consumes messages from the Real-Time Event Consumer and produces to Kafka
- Parallel Run Service 2 consumes events from Kafka, executes data transformation based on predefined transform rules including construction of Vault API request messages, sends a copy to Parallel Run Service 3, and calls the Domain API for each request message once processed.
- Request messages are sent via the Domain API to Vault's BAU (i.e. non-migration) API topics, so that BAU decisioning and Smart contract behaviour takes place. Only customer initiated Postings are picked up by the real-time event consumer, so Product logic that generates Postings is not duplicated
- Parallel Run Service 3 consumes events that are output from either (i) parallel run data transformation or (ii) Vault load, and sends to the Reconciliations Database.
- Reconciliations are an automated daily activity that compares a set of critical fields across source and target systems in a set of standard reports. Inputs are daily batch files produced in legacy BAU, and a curated file of daily updates from Vault Core via the Recs DB

When considering a parallel run migration, consider the benefits and challenges that come with taking such an approach:

Benefits	**Challenges**
Spreads risk by separating load from cutover and enabling an interim period of production-proving.	Parallel run can sometimes introduce delivery risk to migration programmes. The recs have many more failure points than data migration

	recs, and there is a risk that you enter parallel run and never reach the business confidence necessary to leave. You need to get strong agreement from the business about the definition of “good” in the context of the parallel run, and a pragmatic understanding of the differences between recs failures that impact customer/business outcomes and those that do not.
Can better support a slower migration, which may be useful based on what speed you can extract and transform data, because the legacy core remains active throughout.	The design and set-up of a parallel run requires more design and integration work in order to route events to both legacy and Vault Core for the parallel run period.
Can provide confidence to business as there is no cutover from legacy to Vault Core until outputs have been proven during the parallel run period.	A parallel run will likely extend the length of your migration programme due to the cutover only taking place weeks or months after the load.
Potential to simplify load by removing the requirement to handle complex in-flights over the load activity. For example, this can avoid migration of authorised postings and ‘wait out’ where these have a defined or short lifecycle, living with balance variations until these have naturally resolved themselves after load. No customer impact because Vault Core is not operationally live at this stage.	There will be a need for more automated reconciliations throughout the parallel run period.

4. Onboarding/Offboarding Migration

Onboarding/Offboarding migrations are ‘migrations’ in a looser sense of the term compared to the other strategies above.

Either no or minimal data is moved from source to target using an automated ETL pipeline, and instead products are opened on the target platform and balances transferred as a final payment from the old account to the new. This is very similar to industry schemes such as the UK Current Account Switching Service.

At their core, Onboarding/Offboarding migrations trade off a reduction in technical complexity for an increase in business (customer and colleague) impact. The key impacts of this are outlined in the Onboarding/Offboarding section. Accordingly, Onboarding/Offboarding migrations represent a small proportion of overall migrations across our client base and are typically only considered where there is already a significant customer impact for other reasons (such as Card reissue, sort code or account number changing, product changing beyond all recognition, etc.).

Thought Machine will likely have significantly less involvement in an Onboarding/Offboarding migration compared to a traditional API-led migration, as the majority of the migration activities do not directly relate to Vault Core itself. To bring this to life, the questions you should be considering in the design of an Onboarding/Offboarding migration include:

Question	**Comments**
What is the trigger point for initiating Onboarding/Offboarding for each Account?	Given the nature of the Account closure/opening process, this will likely be customer initiated. For example, this could be through submission of an online form or discussion with the bank.
What is the timeframe for executing the Onboarding/Offboarding process?	This will depend on the nature of the activity, for example whether cards are being reissued, but may be driven by the length of time it takes to open and close accounts in BAU today (target and source cores respectively). The account closure process can clearly only occur once account opening has taken place, and there should be consideration given to how and when the actual switch (i.e. payment from old to new accounts) takes place and how the customer is notified.
Is there a requirement for historic transactions to be presentable/available to customers, even though these will not be present in Vault Core*?*	How will these requirements be serviced post-migration? Lack of historic data in Vault Core is a disadvantage of Onboarding/Offboarding migrations - however it may be possible to continue to service some business processes using alternative means (e.g. secondary migration to a data warehouse, retaining visibility of old accounts for a period of time, etc.).
Where account numbers and sort codes are also changing, what happens if a customer submits a payment to their old bank details?	Similar to some industry schemes, the payment could be automatically routed to their new account so long as a reference table can link the old and new account details.
How will customers be incentivised to elect to move to the new account (i.e. trigger Onboarding/Offboarding)?	This will depend on the nature of the customer relationship, but based on experience, monetary rewards (transfer bonus or preferential rates) will often suffice.

Postings Migration Approach

In many programmes, Postings will be the highest volume resource to migrate. Therefore, it is important to carefully consider the specific migration approaches available for migrating Posting Instruction Batches into Vault Core.

For a functional overview of the Posting Migration API see the Migrating to Vault Core: Migration APIs guide.

Some of the common Posting migration approaches are detailed below, which may be combined depending on migration scope and context.

---

1. Depth of Historic Postings Approaches:

An early decision in many migration programmes is the extent of Posting history that should be migrated into the Vault Core. There is no 'right' answer to this question, and the approach taken will vary depending on the programme scope and context.

Option 1: Migrate Posting history using a 'Balancing Posting'

A common approach with Vault Core migrations is to have some or all of an account's historical postings represented as a single posting instruction batch entry in Vault Core. We refer to this as a 'balancing posting'.

For example, a client intends to only migrate the last X years of postings into Vault Core, with all prior history being stored in an external archive solution. In order for the balances to be correct at migration, a single balancing posting per account will be required to represent the sum of all postings older than X years that are not migrated to Vault Core. This may be driven by regulatory data retention requirements. Alternatively, the full Postings history could be migrated as a single balancing posting with no individual Posting history.

Balancing postings are very likely not intended to be consumed downstream or made visible to customers (in the form of transactions) as this posting does not represent an actual customer transaction, rather a means of loading the correct starting balance at migration to Vault Core. This will need to be factored into the Posting Instruction Batch load approach and how you consume and act upon Vault Core posting/balance events. Additionally, when migrating using the Postings Migration API, consider the date set as the value timestamp so that it does not disrupt product behaviour on Vault Core (e.g. as part of a schedule capturing the last 24 hours of postings if the balancing posting has just been loaded).

We recommend using the Posting Type Custom Instruction for balancing postings. This specialist instruction provides the ability to directly write postings to the Postings ledger in Vault Core. A Posting Instruction Batch event will be streamed to `Vault.api.v1.postings.posting_instruction_batch.created` and the appropriate response topic upon creation of this balancing posting, which will need to be recognised and filtered within the downstream systems in order to not present to customers (it could be based on date, instruction type, metadata on the posting, and so on).

Benefits	**Challenges**
Faster overall load time due to reduced number of postings in scope	Wider bank processes that require historic postings will need to derive from an external warehousing solution
Less data stored in operational Vault Core reducing storage costs	Balancing posting likely needs to be filtered out from customer visibility
	If Vault Core Smart Contract behaviour requires historic posting information (for example, last 1 month of historic postings) then this option will not be appropriate

Option 2: Migrate Posting history as hard settled Postings

Another common approach with Vault Core migrations is to migrate some or all of an account's historical postings as hard settled Postings in Vault Core.

Hard settled Postings have reached the end of their lifecycle and represent a final state for the Posting in question. Migrated Postings in this case would include both customer balance changes (for example, payments resulting in credits/debits) as well as any historic product activity such as application of interest accruals or fees. The sum of the migrated Postings in this option should equal the account's balance at the point of migration, so they must account for every credit or debit for that account on source.

Each migrated Posting will generate balance and posting creation events as outlined further in the Postings Migration API section of the *Migrating to Vault Core: Migration APIs* guide.

IMPORTANT: Vault Core is intended to operate as the system of record, **not** the system of truth - the latter will likely be in a cheaper data store connected to Vault Core. Therefore it is architecturally preferred to only put data into Vault Core that is needed to support the latest run position of your bank.

Benefits	**Challenges**
Posting/balance history stored within Vault Core and associated creation events are streamed, which may be valuable for downstream business processes.	Load time increases the more historic Postings that are loaded, which may impact other elements of the migration strategy (load windows, tranche sizes, and so on).
Less complex coexistence requirements where all historic Postings are stored within Vault Core.	May challenge architectural principles around Vault Core being the system of record and not the system of insight.

Option 3: Migrate Posting history and run historic schedules

Another option is to load some or all of the account's **customer initiated** posting history to Vault Core, but not **product generated** postings such as fees and interest, and then let Vault Core create and execute historic schedules immediately at the point of load and before cutover. This effectively re-executes past product behaviour, reaching the current balance position.

There may be examples where re-executing historic schedules is used to handle some in-flight scenarios, such as migrations mid-period during an interest application cycle. This is explored further in the Schedules section of this guide, including other means of handling this scenario that do not require historic schedules to be executed.

Migrating Postings history and running historic schedules is not a preferred option outside of limited use cases for in-flights due to the associated significant performance impacts over the migration event and practical challenges where instance parameters change over this historic period. Additionally, where the Vault Core Smart Contract and legacy product have differing logic this could present challenges to this approach, resulting in either customer impact or a dedicated/throwaway Smart Contract build for the sole purpose of executing the migration. We do not recommend recreating the full history of an account from a product perspective at the point of migration - this introduces significant additional scope at the point of load, and therefore knock-on risk.

Instead we recommend including historic schedule execution in the migration testing phase, specifically through Time-Series Testing.

Benefits	**Challenges**
Valuable product proving activity to ensure Vault Core smart Contract is behaving as expected (though this could also be achieved in test).	Additional effort in Smart Contract build to ensure historic schedules are created successfully, and additional effort over the migration to pause and unpause schedule creation.
	Load times increase significantly the more historic postings that are loaded which require historic schedules to execute. This may impact other elements of the migration strategy (load windows, tranche sizes, and so on).

2. Posting Load Approaches:

Separate from the approach for loading historic postings to Vault Core, there is a need to ensure that the basic Vault Core logical data model is adhered to. In the case of postings, this means postings can only be loaded when the `account_id` specified in the posting instruction batch is present in Vault Core. Attempting to load a posting **without** the associated Account being present will result in a rejection.

Account Dependency Postings Load

In order to account for this dependency when planning your load, one approach is to orchestrate your Account and postings load so that once the Account resource has been

created by the Data Loader, a listening service consumes each `AccountCreatedEvent`. Upon receiving an `AccountCreatedEvent`, the Posting Instruction Batches for that Account automatically trigger to be submitted via the Postings Migration API.
NOTE: This approach will only work if you have a single balancing posting against each account as part of the load event. If you are submitting multiple postings to each account then please follow the normal historic postings load approaches outlined above.

Benefits	**Challenges**
The account resource to posting flow ensures that all Posting Instruction Batch messages will pass the account validation Postings Migration API check.	Requires additional set-up and orchestration that will need to be designed and tested.

Sequential Load

An alternative approach to meeting the account dependency to load a posting is to sequentially load resources in the order defined in the Vault Core logical data model. This means isolating and loading customers *and then* account resources prior to attempting the posting load.

Benefits	**Challenges**
Avoids in-event load orchestration to ensure that the account has been created prior to loading a posting.	Additional event planning needed to load all upstream dependencies prior to attempting a postings load.
Provides quicker performance compared to waiting for an `AccountCreatedEvent` response	Additional delta account updates may be needed to ensure the account position or data is up to date prior to go-live. This will depend on the overall migration strategy.

Product Migration Approaches

Beyond the overall migration strategies, there are also lower-level approaches to migrating Accounts and their associated Smart Contracts.

1. Pre-Loads & Deltas

Pre-loads and deltas are a standard instrument in the migration toolbox, and refer to the loading of some data ahead of the main migration event. The target system is either not operationally live or the data is loaded in a dormant or hidden state, effectively priming the core ahead of the actual migration event whereby cutover will occur.

The pre-loading of data ahead of a migration event can take place regardless of which migration strategy has been selected (big-bang, phased, and so on), and can help in minimising the volume of data and thus length of a migration. It also de-risks the migration events by frontloading activity and reducing the scope around cutover.

Vault Core supports data pre-loads, specifically:

- Schedule creation and execution can be controlled, enabling pre-loaded data to remain static between delta updates. Schedules skipped and/or paused, controlling any product behaviour such as interest accrual or application of fees.
- Whether Vault Core data is consumed into your wider bank will depend on how you have set up to consume data from Vault Core, and may require intervention to prevent unintended behaviour from occurring. Streamed events could be ignored by your consumers for a period of time or based on particular criteria.
- Though schedules can be controlled, bank-initiated API requests and associated event streaming cannot be 'switched off' for pre-loaded accounts. Where necessary you should therefore control the API requests you are sending to Vault Core.

All Vault Core resources supported for migration are candidates for data preloads:

- **Data Loader API**: All Vault Core resources supported via the Data Loader API are good candidates for data pre-loads, particularly Customers and Accounts which are required for a Vault Core migration. These resources are likely to have relatively minimal changes (new business, closures, changes to details, and so on) in data between the pre-load and Vault Core becoming the system of record compared to the effort of the initial load itself.
- **Postings Migration API**: Historic postings, as the highest volume resource that will be migrating to Vault Core, are encouraged where possible to be loaded ahead of an event. Assuming the Customer and Account are already loaded on Vault Core, and with the correct planning, the loading of historic postings could be left to run for a long period and not constrained by normal event execution windows. As this data will not change at all (committed Postings cannot be updated once loaded to Vault Core), only an update file representing the delta of postings against transactions will be needed at the event to bring Vault Core up to date at the point of migration.

A delta update routine capturing any changes on source between extract and Vault Core cutover will need to be passed across at the point of migration. This can happen a number of times in the lead up to the event in order to keep the delta volume controlled. Delta updates could be fed via real-time feeds using Vault Core's BAU APIs similar to a Parallel Run migration, however this may not be worth the architectural complexity for a short delta window.

---

2. Migration Metadata

Some clients find value in including 'migration metadata' in API requests for migrated accounts. Migration metadata in this context refers to fields that can be used to identify that the resource in question was part of a migration event and associated information about this event.

Migration metadata could include; migration flag, migration programme name, migration tranche name, migration date.

Key Vault Core resources such as Customer, Account and Postings have metadata field objects that can store this information. For example on the Account resource the `"details"` field can store flexible key value pair strings:

```
        "details": {
          "KEY": "value1"
        }
```

---

3. Impacts of Schedules over Migrations

Vault Core Schedules kick off and orchestrate the completion of jobs at specific times; for example, interest accrual as part of End of Day processing. If you are not familiar with Schedules please consult the Documentation Hub or speak to your Thought Machine Client Delivery Manager.

Scheduled behaviour that in a BAU scenario (that is, a normal business day on Vault Core) might otherwise be correct, could within the context of a migration event result in unexpected or undesired outcomes.

How can Schedules impact migrations?

In both BAU and Migration contexts, Schedules defined in Smart Contracts are automatically created during the Account Activation step, which takes place after the associated Account resources have been successfully created in Vault Core - and assuming the status of the account is set to open. During this step the `execution-schedules-hook` creates Schedules according to the Schedule definitions in the Smart contract, and an `AccountUpdatedEvent` is streamed to signify the commencement and completion of Account Activation.

NOTE:

- It is the Account Activation Event (and not Creation) that triggers Schedules creation.
- The Account Activation Event is triggered asynchronously once an Account Creation with a Status of OPEN (including an update from PENDING to OPEN) is successful.

- Smart Contract code determines which schedules will be created and their associated start dates.
- Where Schedules have a start date in the past, they will automatically execute immediately after the Account has been created and activated.

In the context of a Vault Core migration this could present a number of challenges if not addressed:

- Where historic schedules are not intended to be executed over the migration, if the Schedule start date is tied to a date in the past, such as account opening date, this has the potential to create and execute many redundant schedules and slow the load process, and potentially change Vault Core data erroneously.
- Where historic schedules are intended to be executed during the migration, there is a risk that they execute before Postings have been loaded. This is because Schedules execute directly after Account creation, and Postings cannot be loaded until after Account creation which may be hours or days after Account load depending on the chosen Route to Live.

It is therefore very important that before you load Account resources, you understand the impact of creating Schedules as defined in your Smart Contract, and whether this is aligned to the behaviour you want, and take action accordingly.

There are a number of options available that could support your Schedule migration in ensuring the correct Schedule behaviour takes place. The choice will come down to the specific context of what your Schedule needs to achieve as well as the make-up and split of the migrating accounts.

In the section below are scenarios that you will likely encounter if a schedule needs handling during the migration. Alongside each scenario are various approach options along with pros and cons.

Controlling Schedule Creation

Smart Contract code defines when each Schedule should be created. If this is linked to a BAU field such as Account Opening Date this may present issues over migration events.

There are a variety of methods that could be used to control this date over a migration event, which includes but is not limited to:

Schedule Start Date	Description
Set to account creation date **in Vault Core**	Set in the Smart Contract and applied uniformly across all Accounts. Effectively this makes schedule creation the date of data load. **This is different from the original account creation date on the legacy system.**
Set to a specified date/time	Set in the Smart Contract and applied uniformly across all Accounts. A specific date/time that can be in the past, present, or future from which schedules will be created.
Convert to the correct Smart	An account can be created using an 'empty' Smart Contract (i.e.

Contract at cutover	no schedules are created from the activation hook running) prior to the migration event. This would enable the account to be created in a PENDING or even OPEN status while avoiding the creation and running of schedules.
Set to specified date/time using Account level Instance Parameters	Setting the date/time for Schedule creation as an Account Instance Parameter means that the date/time can vary across each Account, and be determined as part of the data transformation process during the ETL routine.
Set based on an IF statement depending on whether a particular Flag is present	The IF statement in the Contract (Schedule) determines whether or not to apply bespoke migration logic based on the presence of a Flag (or Account metadata), which differs to equivalent BAU logic. This is the recommended approach because it caters for both migration and BAU Schedule creation logic independently, and removes potential requirements for Account conversion after migration to update. The nature of the IF Statement and when schedules are created from could be any of the options outlined above.

Controlling Schedule Execution

Once the correct Schedule start date has been defined, there may still be a requirement to control the execution of schedules over an event.

This can be achieved using the Account Schedule Tag resource, which is an Account level attribute which can control how Schedules execute for a given Account or Tranche of Accounts by using the following fields:

Field	**Description**
`schedule_status_override`	Enum determining whether or not Schedules should be executed normally or not. Two options that are particularly important: `ACCOUNT_SCHEDULE_TAG_SCHEDULE_STATUS_OVERRIDE_TO_SKIPPED`: When an Account Schedule Tag is set to this status, the scheduled jobs are skipped for the jobs scheduled between `schedule_status_override_start_timestamp` and `schedule_status_override_end_timestamp`. The Scheduler jobs do still run according to the frequency defined in the Schedule, but they are immediately marked as 'done' and no Smart Contract code is executed for the jobs scheduled between the start and end timestamps. `ACCOUNT_SCHEDULE_TAG_SCHEDULE_STATUS_OVERRIDE_TO_NO_OVERRIDE`: When an Account Schedule Tag is set to this status the Schedules are not skipped and will start to immediately execute up to the current date/time.

`schedule_status_override_start_timestamp`	A timestamp indicating when to start overriding the status on account/plan schedules with the behaviour defined by the selected `schedule_status_override`.
`schedule_status_override_end_timestamp`	A timestamp indicating when to stop overriding the status on account/plan schedules with the behaviour defined by the selected `schedule_status_override`.
`test_pause_at_timestamp`	A timestamp indicating when an Account schedule with this tag will pause (stop attempting to execute and pause at the chosen date/time until unpaused); the pause will occur when `next_run_timestamp` in the underlying Schedule is equal to or greater than Account Schedule Tag `test_pause_at_timestamp`. This timestamp can be set to any date and not necessarily only the current time.

Using these four variables you can support a variety of product migration strategies. Common use cases include;

- Executing historic schedules at the point of migration by using the `test_pause_at_timestamp` to pause Schedules until Vault Core is primed with migrated data and unpausing when data is loaded.
- Skipping schedules on pre-loaded data by using a `schedule_status_override` set to SKIPPED during the period that data should not change between runs (for example, between pre-load and cutover), and then changing this to NO_OVERRIDE at the point when future Schedules should now begin to execute.

It is also possible to combine these variables to, for example, skip some historic Schedules but execute others.

Pros	**Cons**
Skipping provides control over historic schedule execution, including stopping this from taking place	Has performance considerations - the extent to which will depend on nature of Schedules, volumes, and how they are being controlled
Pause timestamp allows future schedule execution to be controlled	Impractical to implement at an individual account level
Can define at a tranche level - see tranching section of this guide	The schedules while skipped will still appear in the Operations Dashboard (Thought Machine's UI)

Worked Example: Migrating part-way through an interest application period

Schedules that have a long running accrual and application periods (for example, where interest is calculated and accrued daily but is only applied monthly) may require special handling over migration events so that the Vault Core Smart Contract reaches the same outcome it would have done if the entire period had occurred on the legacy system.

There are a variety of ways this could be managed, including but not limited to:

Phase migration by key date tranches:

Solves the issue of differing interest dates by phasing the migration by key date, migrating tranches the day after their last interest application (that is, the first day of the new cycle), across an extended period until all accounts have been migrated.

By tranching based on the key date and then migrating the day after the last period has ended (and before the EoD daily accrual schedule runs on the first day of the new period), the challenge of in-flight issues can effectively be avoided.

Pros	**Cons**
Simple implementation and does not require additional Smart Contract logic	Extends migration event and restricts potential migration strategies

Smart Contract creates and executes historic schedules:

Allow the Vault Core Smart Contract Schedule to run for the full interest accrual period, including both the portion that took place prior to the migration date and the remainder post the migration date.

You will need to establish the key date for each account and then utilise the schedule creation and execution guidance to:

- Set schedule creation to start from the beginning of the interest accrual period for each account
- Control schedule execution by then pausing schedules ahead of the load, loading Customer, Accounts, and any Postings that occurred during the interest accrual period on source, and unpausing to execute these schedules at the point of migration

Pros	**Cons**
Utilises existing Smart Contract behaviour and reduces migration event orchestration.	Requires additional implementation work, adding complexity to the overall event scope. Necessitates the migration of some historic postings.

Manage in ETL process:

'Pay-up' the portion of the period's interest that has already accrued prior to migration as part of the ETL process, sending a Posting directly into the Account's interest accrual balance address, allowing Vault Core to then add to this by accruing only from the date of migration until the end of the period.

This approach removes the requirement to manage multiple key dates across account schedule tag tranches or as instance parameters linked to smart contract code.

This option may not work if you have a drastically different methodology for calculating interest on the incumbent core - for example, accrual only takes place once at month end and there are no daily accrual figures that can be used to establish a point in time balance to migrate. If this is the case then the bank may decide to keep the existing product behaviour, loading any postings needed to arrive at the end of period position and completing the schedule on target once the migration has completed.

An alternative option is to instead send the migrated interest directly as a hard settlement at the point of migration. However, this option may impact customer terms and conditions, will result in the customer receiving a different interest amount for that month than they would have otherwise, and could require customer and colleague communications accordingly.

Pros	**Cons**
Does not require tranching of key dates because all Vault Core scheduling takes place from point of migration onward	May not work depending on incumbent core method for executing schedules/accrual
Simple to implement	Customer and colleague impacts

---

4. Tranching

What is Tranching?

A tranche is a subset of a larger dataset. In the context of Vault Core migrations, the concept of 'tranching' refers to the tranching of data into smaller chunks in order to phase a data load, or tranching accounts to enable a phased cutover. In practice, tranching is most useful as an extension of the concepts covered in the previous section on controlling the execution of schedules, whereby Schedules can be controlled at the tranche level.

Tranching as a concept in Vault Core can be applied to Account and Plan resources when loaded via the Data Loader. This section outlines how tranches can be utilised within migrations specifically, and focuses on the primary use case for this - tranching Accounts.

Defining Tranches in Account Schedule Tag requests

To define a tranche in Vault Core you will need to make use of the Account Schedule Tag resource. This resource is used to both define tranches and subsequently control the execution of schedules for these tranches. In practice:

- Account Schedule Tag tranches must be defined before Accounts (that you intended to be associated with the tranche) are loaded via the Data Loader API.
- Account Schedule Tags can be defined by either the Configuration Layer Utility (CLU) or a Core API call to The Account Schedule Tag's Create Endpoint at `POST v1/account-schedule-tags`
- Regarding message structure:
  - The `account_schedule_tags.id` field is a free-text string that will be used as the `tranche_id` field in the Data Loader request when loading Accounts. For example, to create a tranche called 'tranche_A_trial_migration' you should

set this value as the `account_schedule_tags.id` in the Account Schedule Tag creation request.
  - How you should populate the remaining fields on the Account Schedule Tag request depends on how you intend the tranche to behave once associated Accounts are created in Vault Core; specifically how Schedules should behave.
- It is important to not specify the `tranche_id` as Schedule Tags in your Smart Contract event types. Where a `tranche_id` is set in the Data Loader request, this will automatically populate in the Smart Contract Schedule upon Account creation as a `scheduler_tag_ids`. This creates a link between the Account Schedule Tag resource and the Smart Contract Schedule that allows scheduled behaviour to be controlled.

Creating Tranches in Data Loader requests

Once a tranche has been defined, it can be set as part of the Data Loader request using the `tranche_id` field. The Documentation Hub contains example messages including the `tranche_id`.

Tranches can be defined in Data Loader requests at either a Resource Batch or Resource Level (can apply to all resources uniformly in a Resource Batch or be set individually on particular Resources within a wider Resource Batch). Where tranches are defined at both levels, the Resource Level tranche will take precedence. Note:

- The `tranche_id` on the account is only set once during loading the account and cannot be updated or removed post load, so tranches must be considered and set in stone before the production data load takes place.
- Only one `tranche_id` can be associated with a single resource.

Using Tranches as part of Vault Core Migrations

Once a tranche has been defined and created, you can control the execution of Scheduled behaviour at the tranche level. This can be practically useful in the context of phased incremental migrations where data can be loaded in one go and selectively cutover.

For example:

- 100,000 total Accounts to migrate.
- Three tranches cutting over at different stages; 100 friends and family accounts in `tranche_A`, then 900 accounts in `tranche_B`, and finally 99,000 in `tranche_C`.
- Requirement for the Schedules on these Accounts to be skipped until the go-live date so that the scheduled events won't be duplicated because the legacy core system is still active at this point in time.
- Load data for all 100,000 accounts to Vault Core as part of a pre-load, and set the account schedule tag's status for all three tranches to `ACCOUNT_SCHEDULE_TAG_SCHEDULE_STATUS_OVERRIDE_TO_SKIPPED`. As a result the data is loaded in a dormant state with scheduled behaviour paused.
- To go live with only `tranche_A` accounts, edit the `tranche_A` account schedule tags status to `ACCOUNT_SCHEDULE_TAG_SCHEDULE_STATUS_OVERRIDE_NO_OVERRIDE`, which will unpause schedules and allow them to execute. This includes any historic schedules created based on the Schedule start date, as well as any future Schedules.

Effectively, this mechanism is equivalent to the switch method for go live, which will allow staggered edits of account schedules based on the tranche. Data can be loaded in what is effectively a dormant position, allowing for pre-loads ahead of a main migration event without worrying about data change.

---

5. Migration Dependencies around Customer Data

A common early concern on core banking migrations is the interplay and dependencies between customer and product data.

- Customer data, as defined here, includes fields such as customer names, addresses, contact details, marketing preferences, paperless preferences, customer level flags.
- Product data, on the other hand, can simply be understood as data that can be stored within Vault Core, acting as the product data master - account, account level flags, restrictions, payment devices.

The impact of customer data on the product (core) migration will be heavily influenced by how and where customer data for the migrating accounts is stored on source today. This is likely to be one of:

Customer Data is mastered in a Customer SoR distinct from the Product SoR

- Some banks separate customer and product data in distinct systems, maintaining a distinct Customer SoR.
- The value of a core customer system (Customer SoR) is maintenance of a single customer view across multiple product cores, which holds a view of the customer level relationships with the wider bank.
- Customer SoRs may also benefit from customer record matching and merging processes, whereby the single customer view is maintained through a set of automated rules that identify and merge duplicate customer records.
- Though some limited customer data might be stored in the Product SoR, it does not master any customer data and is not used to drive customer related business processes.
- Migrating core data in this architecture is relatively simple, and the key concerns will be around:
  - Whether there is value in maintaining a cross-reference within Vault Core to the Customer SoR, likely by storing the Customer SoR's unique customer reference/ID within Vault Core's Customer resource field `external_customer_id`.
  - Updating the existing Customer SoR to recognise Product holdings within the new Core, and potentially updating the name of the core on existing product holdings when the migration takes place.

    - Where necessary, creating pipelines between the Customer SoR and Vault Core to manage automated updates, for example changing the `external_customer_id` as a result of match and merge processes, or on customer creation/closure.

Customer Data is mastered in the Product SoR

- It is also not uncommon for product systems to master some or all customer data themselves, alongside product data.
- This makes migration into Vault Core more challenging, as Vault Core is not a core customer system and should not be used as the customer master, necessitating a separate parallel migration into a Customer SoR.
- This customer data migration will need to go through the same migration programme lifecycle as product data (including strategy, mapping, quality, and ETL build, for example), culminating in a migration event.
- The customer data migration is likely to be a prerequisite activity for migrating the product data (e.g. where the current core is expected to be decommissioned shortly after migration - it cannot continue to master customer data at this point), and broadly speaking the two options are:
    - Migrating customer data as a decoupled precursor step ahead of product migration - this de-risks the product migration by removing any dependency on customer data also migrating over the event, and can be seen as a harmonisation activity (preparing the new target infrastructure). This is preferable where possible.
    - Migrating customer data as part of the product migration event (or each event where phasing the product migration) - this elongates the overall product migration event (likely occurring first, so pushing back the timings for product migration) and adds complexity in orchestrating two migrations back to back and is not preferable where possible.

The general recommended integrations between a product core and customer core, as well as patterns and recommendations for how and where to store customer data alongside Vault Core, can be discussed with Thought Machine's client architects where required.

Appendix

Appendix A - "Units" of Migration

Unit	**Benefits**	**Challenges**
Customer led	● May reduce architectural complexity in needing to bridge the single customer view across two cores. ● Concentrates customer-level migration impact to a single event (because it does not drag out customer impact over a long period).	● If there are multiple products in scope, there is potential for long build time because the superset of Smart Contracts (products) would need to be deployed on target ahead of any migration, resulting in a loss of incremental delivery value and likely a longer overall route to live. ● Consolidates customer-level migration risk into a single event. ● Architectural complexity around core coexistence and routing data to (or reassembling data from) multiple cores, though likely less than other options.
Product led	● Significantly reduced time to production with parallel workstreams delivering products iteratively depending on chosen priorities. ● Focused priorities reduce short-term scope creep and complexity. ● Earlier realisation of benefits and evidence of programme value.	● Architectural complexity around core coexistence and routing data to (or reassembling data from) multiple cores. ● Customer impact may be felt at multiple points throughout the programme where customers have multiple products that are in scope.
Account led	● Beneficial in scenarios such as mergers and acquisitions, where attributes such as sort codes or BINs can only belong to one legal entity at a time.	● If there are multiple products in scope, there is potential for long build time because the superset of Smart Contracts (products) would need to be deployed on target ahead of any migration, resulting in a loss of incremental delivery value and likely a longer overall route to live. ● Architectural complexity around core coexistence and routing data to (or

		reassembling data from) multiple cores.
Attribute led	● Nuanced tranching selection (for example, migrating based on account value, migration based on number of business relationships, and so on) may provide specific benefits to the programme.	● If there are multiple products in scope, there is potential for long build time as the superset of Smart Contracts (products) would need to be deployed on target ahead of any migration, resulting in a loss of incremental delivery value and likely a longer overall route to live. ● Architectural complexity around core coexistence and routing data to (or reassembling data from) multiple cores.

Document history

Version	Date	Author/reviewer	Comment
0.1	13/05/2020	R Sirisomphone	Initial draft, created as the *Guidance for Postings API usage* document
02	22/05/2020	M Morosan, J McQuillan	Reviewed
0.3	26/05/2020	A O'Neill	Reviewed, with editorial and wording changes suggested.
0.4	01/06/2020	R Sirisomphone	Update after review.
1.0	02/06/2020	S Gasteratos	Approval.
1.2	27/01/2021	R Sirisomphone	Update to new template.
1.3	18/08/2021	J Measures	Created new *Migrating data into Vault* guide.
1.4	14/01/2022	J Measures	Incorporated all content from version 1.2 of the *Guidance for Postings API usage* document into the Migrating Postings chapter. Updates to Migrating Postings for Vault Core 3.0.
1.5	19/01/2022	B Harrison	Updates throughout.
1.6	19/01/2022	M Morosan, I Staden, F Morant	Review, comment, approval.
1.7	19/01/2022	J Measures	Minor updates.
2.0	19/01/2022	B Harrison	Approval
2.2	02/03/2022	F Morant, B Harrison	Added: ● Information about migration strategies ● General updates
2.3	10/03/22	F Morant	Added appendix listing core streaming api creation and update listening events.
3.0	11/03/22	B Harrison	Approved and version updated.
3.3	07/04/22	F Morant	Added detail on: 1. Posting message formats in migration contexts 2.At least once publishing guarantees.
4.1	25/04/22	F Morant	BAU Postings API guidance incorporated and wide range of smaller updates based on latest guidance and learnings

4.2	27/04/2022	B Harrison	Split content into multiple documents and updated throughout
5.0	13/05/2022	B Harrison	Approved and version updated.
6.0	17/06/22	B Harrison	Added sections: ● Migration Dependencies around Customer Data ● Planning and Process & Implementation guidance when considering a coexistence approach. Retitled sections: ● *Migration strategies* to Starting migration questions ● *Summary migration strategies* to Common migration patterns Reformatting of 4. Onboarding/Offboarding Migration and minor updates. Approved and incremented version.
6.1	13/07/2022	B Harrison	Expansion of detail on parallel runs.
7.0	15/07/2022	B Harrison	Approved and version updated.
7.1	27/09/2022	J Davey	Minor formatting changes.
8.0	28/09/2022	B Harrison	Version updated to reflect approval.

Thought
Machine