Model-Use Abstraction for Financial Crime Detection and Prevention Analytic Controls

David Coppin, CAMS | Principal Consultant
Financial & Transactional Analytic Solutions Australia
david.coppin@fintas.com.au
linkedin.com/in/david-coppin

April 2026

Abstract

Financial crime detection and prevention capabilities in many reporting entities have evolved as collections of domain-bound systems rather than as coherent, reusable analytical capabilities. Transaction monitoring, customer name and payment screening, customer risk assessment, and adjacent controls frequently replicate the same data pipelines, analytical logic, and governance structures, embedding core modelling primitives inside workflow-specific implementations. This fragmentation increases cost, reduces adaptability, and makes controlled change difficult as regulatory expectations shift toward effectiveness, explainability, and timely intervention.

This paper proposes model-use abstraction as an alternative architectural organising principle. Under this approach, analytical models and feature-producing logic are treated as reusable, governable capability, while domain use cases are implemented as orchestrations of those potentially shared components. Detection and prevention are framed not as separate systems, but as distinct control postures applied to the same underlying execution capability.

The paper outlines a generalised model framework, examines implications for orchestration, responsiveness, challenger testing, and governance, and describes a practical architecture and adoption path for organisations seeking to reduce duplication while improving control over change, execution reproducibility, and governance clarity. The aim is not abstraction for its own sake, but a more adaptable, explainable, and evidence-driven analytical foundation for financial crime control.

Table of Contents

Model-Use Abstraction for Financial Crime Detection and Prevention Analytic Controls 4
The Structural Problem 5
The Architectural Proposition 6
Detection and Prevention as Control Postures 6
A Generalised Model Framework 7
Domain Use Cases as Orchestrations....10
Model Sets, Challenger Testing, and Controlled Change 11
Orchestration and Responsiveness 13
What orchestration must guarantee 14
Governance Implications 15
A Practical Architecture for Shared Modelling Capability....16
Core high-level requirements....16
What common tooling can look like....17
A practical adoption path 18
Closing Thoughts....20
References....22

Model-Use Abstraction for Financial Crime Detection and Prevention Analytic Controls

Financial Crime (FC) detection and prevention analytic controls[1] in many Reporting Entities (REs) are not designed as a coherent capability but instead tend to accumulate over time. Capability often emerges through platform implementations, control obligations, review recommendations, vendor roadmaps, and organisational structure until operating domains such as Transaction Monitoring (TM), Customer and Payments Screening (CAPS), Customer Risk Assessment (CRA) scoring, Enhanced Customer Due Diligence (ECDD), Fraud and Scams each become their own domain-bound stack, with their own data plumbing, logic, governance, and operating rhythms. Those domain-bound stacks are understandable as an operating model, but they are a weak organising principle for architecture.

When analytic controls are organised as a set of domain-bound stacks, the same capabilities and data are often rebuilt in multiple places. Core data, segmentation, profiling and aggregation, text and entity resolution, scoring, thresholds, and routing logic risk being implemented repeatedly, often with different assumptions and different standards of reuse. Over time, logic hardens inside workflow engines, vendor-built scenarios, and bespoke pipelines, so change becomes brittle and expensive because the underlying dependencies are poorly exposed and understood.

This becomes important as expectations shift from nominal coverage to effectiveness, and from control existence to more defensible and explainable operation, with regulators and industry bodies increasingly framing FC analytic controls in outcomes-focused and effectiveness-oriented terms[2,3,4]. At the same time, pressure for earlier intervention is increasing, with greater emphasis on timely identification, risk-based transaction monitoring, and taking steps to protect customers and the business rather than relying solely on retrospective review[5]. This is reflected in Australia's proposed scams prevention framework [5], which emphasises proactive prevention, earlier intervention, and coordinated responsibility across industry participants to disrupt scam activity before loss occurs. Many current architectures, however, make those postures difficult to support without rebuilding logic in new wrappers.

---

[1] In this context, detection refers to the identification and surfacing of activity (typically post transactional event), relationships, or conditions that warrant further review, investigation, or other downstream action. Prevention refers to the use of similar analytical logic to support earlier intervention, such as friction, step-up, restriction, hold, or blocking actions. Collectively, these postures – detective and preventive – will be referred to as analytic controls within this paper.

[2] AUSTRAC's risk-based approach guidance emphasises outcomes, supporting evidence, and ongoing monitoring of effectiveness, rather than a checklist approach [1].

[3] The FCA describes its financial crime guidance as intended to support a more effective, risk-based, and outcomes-focused approach to financial crime controls [3].

[4] The Wolfsberg Group's statement on effective monitoring [4] highlights the need to move beyond traditional automated transaction monitoring toward approaches that produce more effective outcomes, and it places particular weight on explainability and transparency in demonstrating coverage and effectiveness. This supports the argument that financial crime controls are increasingly expected to be explainable as well as operationally effective.

[5] AUSTRAC's transaction monitoring guidance [2] frames monitoring as a risk-based process that should help businesses identify suspicious behaviour and take steps to protect both the business and its customers. This supports the view that transaction monitoring is not only retrospective surfacing but increasingly linked to more timely and responsive control action where risk warrants it.

This paper proposes a reframing: analytic controls should be designed as a reusable modelling capability, not as a collection of domain-bound point solutions tied to control use cases or vendor products. Operating domains – TM, CAPS, CRA, ECDD, Fraud, Scams controls – are better treated as orchestrations of a shared modelling capability. That is the guiding proposition; the rest of the paper considers its architectural, governance, and operational implications.

The Structural Problem

What sits underneath this discussion is not a question of tool choice, vendor choice, or local implementation quality, but a more basic architectural pattern: when analytic controls are organised as a set of domain-bound stacks rather than as a shared capability, the same weaknesses tend to recur across REs even where products, platforms, and operating models differ.

Fragmentation is typical in the data and plumbing[6] layers, where the same sources are integrated repeatedly into analytic engines, often with use-case-specific assumptions and duplicated patterns, while quality controls drift because there is no single governed source-to-feature path. Modelling capability then diverges along similar lines. To illustrate: one team builds segmentation or entity resolution well, another reimplements a weaker version, and a third embeds comparable logic inside a point solution where it cannot be reused at all.

What follows is more than duplicated effort. Once logic is embedded in the wrong layer, the RE loses optionality: it becomes harder to apply consistent segmentation, reuse entity resolution, run controlled challenger changes without recreating large portions of the estate, or move a control posture from detective to preventative without rebuilding the same logic in a different execution framework. The visible estate may continue to grow, but the underlying capability remains less reusable and less adaptable than it appears.

The architecture also tends to develop an uneven control posture, with near-real-time prevention options remaining limited compared to post-event detective monitoring not because earlier intervention is conceptually impossible, but because the same primitives are not designed to run cleanly at different latencies or to route outputs into different forms of action. Governance risks diverging along the same fault lines, so that different standards, evidence models, and definitions of effectiveness emerge by domain even where the underlying modelling needs overlap materially.

A common counterpoint is that several vendors now offer broader, end-to-end workflows spanning multiple domain use cases, including combinations of TM, CAPS, CRA, ECDD, Fraud, Scams, etc. Although that can reduce some visible fragmentation by consolidating case management, user interfaces, configuration patterns, or a shared data model, it does not by itself establish a unified modelling capability. In many implementations, the modelling primitives remain separated even within an integrated suite: segmentation, aggregation, entity resolution, text matching, scoring, and threshold logic still sit in different modules, are configured differently, are not reusable as building

[6] Refers to the data engineering layer that moves and prepares data for analytical use. In practical terms, this includes ETL/ELT pipelines, data warehouse and data mart transformations, and the business rules embedded in those processes to clean, join, enrich, and reshape source data. It is the part of the environment that takes raw data from systems such as core banking, payments, or customer platforms and turns it into usable tables or features for modelling. While it is not the analytical logic itself, it has a material impact on how consistent, reusable, and reliable that logic can be in practice.

blocks, and remain governed by domain use case rather than by capability. The argument in this paper is directed at that deeper layer.

Seen in that light, these are better understood not as isolated concerns, but as symptoms of the same disparate design approach, where analytic controls are treated as the implementation of domain-bound controls rather than as a capability composed of reusable analytical building blocks. When that is the organising principle, fragmentation is less an accident than a predictable outcome.

The Architectural Proposition

The proposition in this paper is narrower than it may first appear. It is not that domain use cases cease to matter, or that all FC functions should collapse into one undifferentiated architecture. Workflows, latency needs, treatment options, evidence requirements, and governance expectations still differ materially across domain use cases. The proposition is that those differences should sit at the orchestration, treatment, and governance layers, while the modelling capability beneath them is designed for reuse wherever feasible and sensible. The design objective is therefore not total logical abstraction for its own sake, but controlled reuse, clearer lineage, and greater adaptability across related FC domain use cases.

From that framing, several implications follow:

- Data is a shared asset and should be governed for reuse across use cases rather than repeatedly reshaped inside domain stacks.
- Modelling primitives are more useful when treated as reusable design objects rather than hidden implementation details.
- Control posture still varies by domain use case, so the architecture needs to support detective, preventative, and mixed analytic control postures without rebuilding the same core logic.
- Responsiveness is a design choice rather than a platform accident, with batch, event-driven, on-demand, and lower-latency execution all available where appropriate.
- Use cases are better understood as orchestrations of shared capability than as monolithic analytical systems.

These implications establish the architectural basis for treating detection and prevention as different control postures applied to the same underlying capability. Before turning to the mechanics of that capability, it is useful to make that implications explicit.

Detection and Prevention as Control Postures

A key implication of this proposition is that prevention should not be treated as a separate architectural world. Most AML/CTF analytic controls remain detective, typically post-event, because it is operationally familiar: process batch data through rules, generate work items, investigate, escalate, report. Prevention, however, is not a separate universe and does not require a separate architectural doctrine. It is a different control posture applied to the same underlying analytic capability.

When events are treated as control outputs rather than workflow outcomes, they can be routed differently according to posture. An event may be sent to investigation where the objective is detective monitoring; the same event may also be routed to a service layer where the objective is to apply future friction, step-up, hold, block, or restriction. What changes is not the existence of the model and event, but the way its output is used.

Even so, prevention should not be understood as detective routing at lower latency. Different postures imply different tolerances, different service dependencies, and different operational consequences. False-positive tolerances may tighten or loosen depending on the harm model, and service integration introduces additional legal, product, and customer-experience considerations. The point is not that the same event can always be reused unchanged, but that prevention should be treated as an architectural option built on shared execution capability rather than as a separate doctrinal world or an exceptional project. Those differences also extend to governance burden, evidentiary expectations, and acceptable operational trade-offs, meaning architectural commonality does not imply identical governance treatment.

If prevention is to be treated seriously, the architecture has to support service integration as a first-class outcome of model execution. That should not be understood as an extension bolted onto a monitoring stack, but as part of the capability the architecture is intended to support. That, in turn, requires a clearer definition of the analytical objects the architecture is actually built from — what a model is, what it outputs, and how those outputs are used across different control postures.

A Generalised Model Framework

If detection and prevention are to be treated as different postures of the same analytic capability, the next question is what that capability is actually built from. The architectural issue is not whether models exist, but what is treated as the practical unit around which design and governance are organised. Many REs define FC models narrowly, if they define them explicitly at all, as something that produces an alert or contributes directly to one[7]. That is convenient in workflow terms, but weak as a unit of capability design. Alerts are workflow artefacts, often the result of roll-up, aggregation, and routing decisions; they are not a stable architectural unit.

At a high level, the contrast in this paper is between two architectural organising principles. In a domain-bound architecture, models, data pipelines, routing logic, and governance are embedded within domain stacks such as TM, CAPS, CRA, ECDD, Fraud and Scams, with reuse occurring incidentally. In a model-use abstracted architecture, those analytical components are treated as shared capability: models and feature-producing logic are governed as reusable design objects, while domain use cases become orchestrations that sequence and apply them according to control posture. The argument in this paper is that the latter provides a stronger basis for reuse, lineage, controlled change, and cross-domain adaptability.

[7] In some REs, particularly where enterprise model governance frameworks are more mature, FC analytical objects may already be brought within broader model governance standards, and the definition of “model” may therefore extend beyond alert-producing logic to include scorecards, feature-producing models, segmentation methods, or other governed analytical components. The point here is not that all REs use a narrow formal definition, but that detection capability is still often organised operationally around alert-producing logic rather than around reusable analytical units.

A more useful definition is execution-centric: FC models consume data and/or features[8], and they output either features or events. That definition is broad enough to cover the full range of analytical logic used in FC control design, while remaining precise enough to support governance, reuse, and controlled change. It also separates the model from the workflow that may later consume its output.

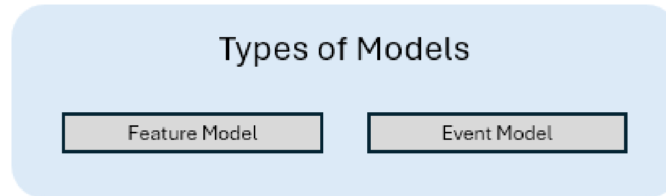

Figure 1 - Types of Models

From that framing, two categories emerge which many REs already operate implicitly, but rarely treat as first-class[9] design objects.

- **Feature models** produce intermediate outputs that other models consume. These include segmentation, normalisation, data labelling, profiling and aggregation, and text scoring or fuzzy matching outputs.
- **Event models** produce a control event. That event may become a workflow item, but it does not have to. It may route to investigation, to a preventative service, to case management, or simply be retained as evidence of control operation.

A design tension arises at this point: not every feature should be treated the same way operationally. Some features are best materialised and persisted because they are reused across multiple downstream controls or must remain available at predictable latency. Others may be recomputed on demand because they are posture-specific, low-reuse, or tightly coupled to a single execution path. The critical issue is not persistence for its own sake, but whether timeliness, latency, and reuse are being governed deliberately rather than emerging accidentally through platform behaviour. As a rule of thumb, features that materially influence multiple downstream control postures should generally be persisted with defined timeliness expectations, while posture-specific or low-reuse features may remain ephemeral where latency and control purpose allow.

With this model distinction made, noting the prior persistence concern, the next architectural move is to separate production from use. Feature models can run upstream to generate outputs that are stored, reused, and consumed repeatedly, while event models can be composed downstream to convert those features into control events under a particular orchestration.

[8] In this paper, a feature means an analytical variable or intermediate output used downstream by one or more models. Features may be derived directly from source data or produced through prior transformations, aggregations, linkage processes, scoring logic, or other upstream analytical steps. Examples might include customer segmentation outputs, rolling behavioural aggregates, scorecard values, or text-match scores.

[9] Treated as an explicit object in its own right within the architecture, rather than as a hidden implementation detail or incidental by-product of another process. A first-class design object is one that can be defined, versioned, governed, reused, and changed deliberately.

This reduces the need to embed analytical logic inside a single domain-specific implementation, because feature production and event generation can instead be organised as related but separable parts of the same execution capability.

Figure 2 illustrates the framework in simplified form rather than as a complete execution chain. Operationally, feature models may sit upstream of other feature models, and event models may consume raw data, derived features, or a combination of the two as this pattern is repeated across more complex use cases.

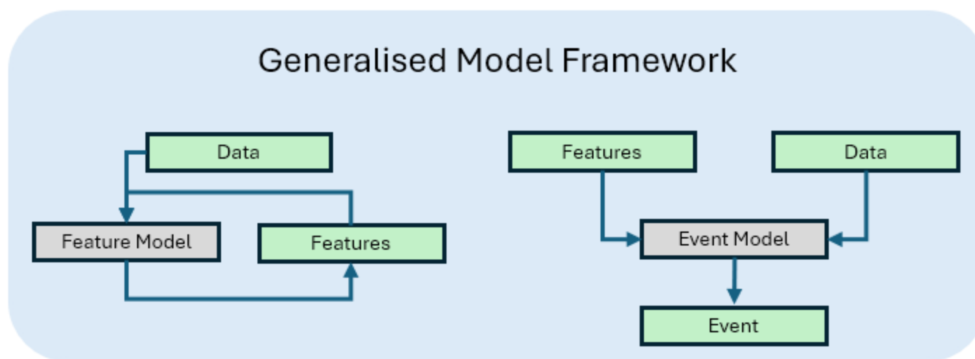

Figure 2 - Generalised Model Framework

Treating events as control outputs rather than workflow artefacts preserves optionality in how the same underlying logic is used. A model can support post-event monitoring in one orchestration and lower-latency intervention in another without being redefined each time.

This does not mean that every analytical object must be called a model in every implementation. Some REs will distinguish more sharply between models, transforms, enrichment logic, and routing logic. The design requirement is not terminological purity; it is that reusable analytical objects be treated as stable, governable units rather than hidden implementation details. Much of the real reuse value sits upstream in feature production, regardless of whether those assets are labelled feature models.

For clarity, the language in this paper is used as follows:

- **Data** refers to the inputs consumed by models and transformations. This includes both core operational data — such as customer, account, transaction, counterparty, product, channel, and workflow/system data — and auxiliary datasets that extend or contextualise the core view, such as risk assessment data, internal or external risk lists, and third-party datasets including PEP, sanctions, adverse media, or corporate registry data.
- A **model** is a logical unit of execution.
- A **feature** is an analytical output produced by a model and consumed downstream.
- An **event** is an analytical output representing a control decision boundary or signal.
- A **use case** is an orchestration of models that produces outcomes appropriate to a specific control posture.

The labels themselves are less important than the distinctions they are being used to preserve. Those distinctions shape what can be shared, what can be governed coherently, and what can be executed across domains without rebuilding the same logic in different wrappers.

Domain Use Cases as Orchestrations

If the model framework follows this paper's proposition, use cases are better understood as orchestrations of reusable models than as monolithic analytical processes. This changes the unit of design from domain-bound stacks to sequences of governed analytical components.

Most FC solutions, once stripped of workflow labels and product branding, resolve into a broadly similar analytical pattern: produce or enrich features, combine or evaluate them, emit an event, and route that event to an outcome. What distinguishes one use case from another is not the existence of those steps, but the way shared models are sequenced, parameterised, and directed in support of a particular control posture.

Cross-domain dependency also becomes explicit. One use case can condition another – for example, a customer risk score may alter TM sensitivity. In domain-bound architectures, those relationships tend to appear as bespoke integrations or operational exceptions, often mediated through separate pipelines, batch transfers, or delayed synchronisation between systems. That creates both friction and risk: one control may be acting on stale outputs from another, design changes become harder to propagate consistently, and the combined control posture may be slower or less coherent than intended.

In a model-use abstracted architecture, those dependencies are instead expressed directly through shared components and orchestration. Segmentation or behavioural aggregation can run as shared upstream feature models on periodic or event-driven cadences as customer activity develops. Those outputs can then be consumed by a customer risk scoring orchestration to produce a risk tier, which itself becomes a feature available to downstream use cases. Monitoring logic can use that feature to vary threshold sensitivity within TM. Whether those upstream features are persisted or recomputed depends on their reuse, latency requirements, and timeliness expectations, rather than on the structure of any single use case. The underlying models are not redefined for each use case; what changes is the orchestration: sequence, parameters, event conditions, cadence, and treatment of the output.

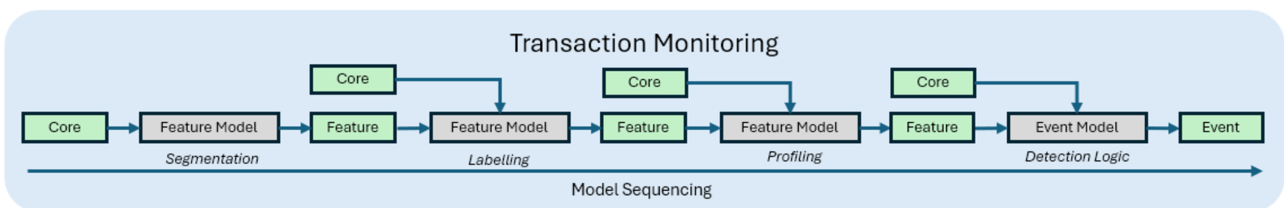

Figure 3 - Illustrative TM Model Sequence

Figure 3 illustrates a pattern in simplified form for a TM use case, showing how a monitoring flow can be understood as a sequence of feature-producing steps followed by downstream event logic. The example is intentionally simple, but it makes the architectural point clearly enough: what is often treated as a single TM implementation can instead be abstracted into potentially reusable upstream models and a downstream event model, with the sequence itself becoming part of the use-case design rather than something hidden inside a domain-specific stack.

Treating domain use cases as orchestrations preserves reuse without collapsing control intent, and allows multiple FC functions to evolve together without forcing each one to become its own stack.

Just as importantly, it creates a clearer execution object against which change can be introduced, compared, and governed. That becomes especially important once the RE wants to test alternative logic or orchestration without recreating an entire domain-bound implementation.

Model Sets, Challenger Testing, and Controlled Change

When use cases are expressed as orchestrations of shared models, the implications for change become clearer. Feature production no longer has to remain inseparable from downstream event logic, and change no longer requires large-scale rework by default. A model can be revised, re-run in isolation or within a governed set, and compared against the incumbent configuration without rebuilding the surrounding estate.

Operationally, the relevant object is not only the individual model, but also the model set: the sequence of models, dependencies, and routing logic used to produce features and events for a given use case or control posture. Individual models change because logic changes, while model sets change because component models, orchestration, dependencies, cadence, or control posture change. Both need to be governable, but for different reasons. A model is the unit of logic; a model set is the unit of execution and evidence. That distinction matters most when the RE wants to test change in a controlled way. Without a stable execution object, comparison quickly becomes entangled with the wider implementation rather than focused on the logic or orchestration actually being challenged.

Champion–challenger testing[10] often struggles - not due to a lack of statistical technique – but due to the RE does not have a stable comparison object. If the only meaningful object is an end-to-end domain stack, then testing a challenger means recreating too much of the system, with too many hidden dependencies and too much operational risk. The result is that tuning persists as local change without a coherent evidence base.

A hierarchical model structure makes that problem more manageable. At the lowest level sit individual models: segmentation logic, entity resolution, scorecards, thresholds, aggregation models, and event models. Above that sit model sets: ordered combinations of models, executed with defined dependencies, to produce features and events for a specific use case or posture.

---

[10] In this paper, champion–challenger testing refers to the controlled comparison of a current production model or model set (the champion) against one or more alternative models or model sets (the challenger/s) using production-like execution conditions and production data, or a governed subset thereof, for the express purpose of assessing comparative analytical and operational performance. This is distinct from conventional UAT, development, or release testing. The objective is not to confirm that software changes deploy correctly, but to assess whether alternative analytical logic produces better outcomes while remaining operationally usable and governable. In practice, this is best understood as a BAU analytical capability rather than as an occasional platform-change activity. Among Australian REs, this remains a persistent area of weakness: many can develop or tune logic, but fewer have mature capability to run controlled champion–challenger comparisons in a repeatable, production-aligned manner. Broadly, two implementation approaches tend to appear. The first is replicative modelling in separate analytical environments, where alternative logic is rebuilt in tools of choice and tested against production data extracts; this is flexible but creates validation challenges in proving sufficient consistency with the production champion. The second is production-like execution in vendor or platform-adjacent environments using the production codebase or closely aligned equivalents; this reduces fidelity concerns, but introduces cost, environment-management, and operational overhead. The important point is that champion–challenger testing concerns controlled comparison of live analytical logic under production-aligned conditions, not merely software testing through a standard development and release lifecycle.

Change can occur at either level. A single model may change while the set remains stable; a set may change because sequence, routing, cadence, or dependencies change even where the constituent models do not.

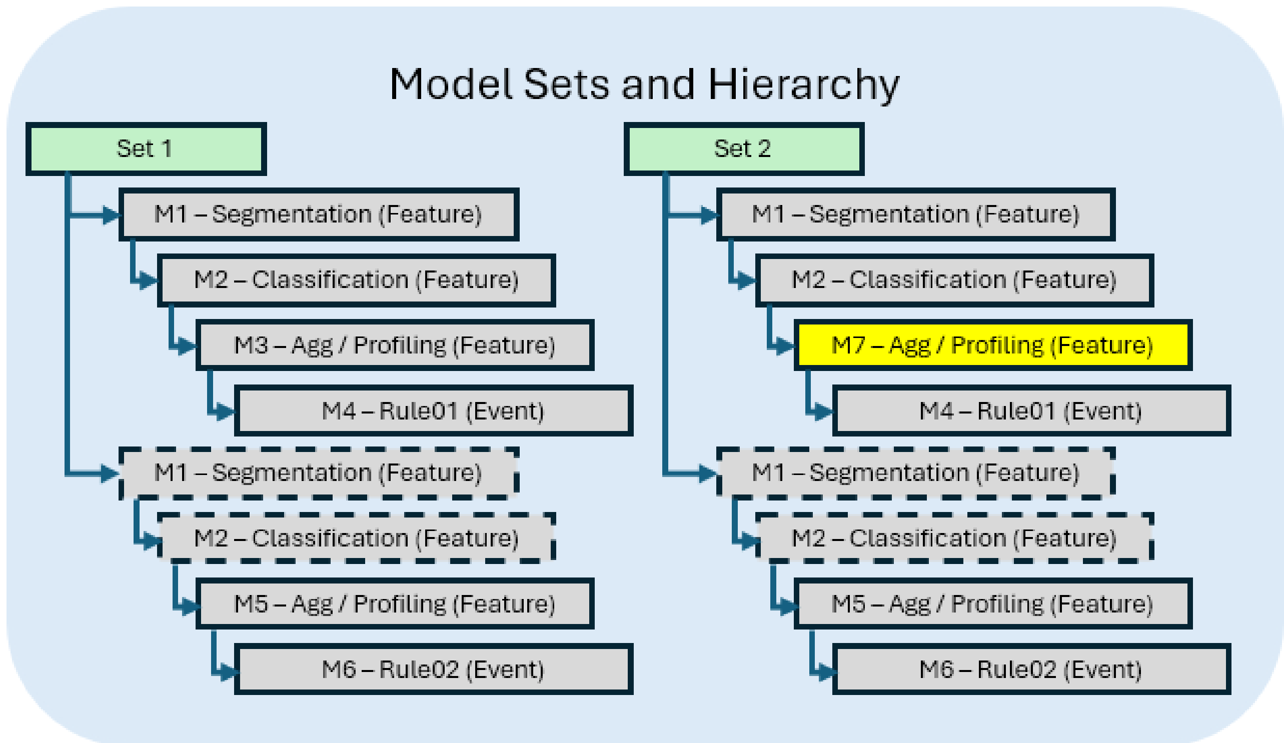

Figure 4 - Model Sets and Hierarchy – Set 1 Champion versus Set 2 Challenger

Figure 4 illustrates the hierarchy across two model sets. Two model sets are shown side by side, with Set 2 acting as a challenger to the production champion, Set 1. In the example, a shared set of upstream feature models (M1 and M2) supports multiple downstream event models (M4 and M6), each representing a distinct rule built on a common analytical basis[11].

In the challenger, most of the sequence remains unchanged, but one feature model has been replaced (M3 in Set 1 becomes M7 in Set 2, which may represent a revised version), which may be enough to alter downstream behaviour and control outputs. The diagram also illustrates that where dependencies are explicit, and feature outputs are already available; challenger testing does not require every upstream step to be rerun. That reduces both the technical and operational burden of testing, because change can be focused on the affected component, the dependent models, and the resulting outputs rather than on the entire domain-bound stack[12].

In that structure, challenger testing can be introduced either at model level, where the objective is to compare alternative logic for a single component, or at model-set level, where the objective is to compare orchestrations, dependencies, or control posture. This provides a bounded comparison anchored to a stable execution object, with visible lineage and attachable evidence. It does not

[11] This could be illustrated by two cash credit scenarios – both could utilise common segmentation and classification models.

[12] It is important to note that efficiency gains diminish as revisions occur further upstream in the model set, because a greater proportion of dependent models must be recomputed, approaching a full re-run of the sequence.

remove the need for careful measurement design, data lineage control, or operational impact analysis, but it makes those activities easier to conduct coherently.

That architectural improvement does not remove the need for organisational policy decisions. In AML especially, challenger testing may still be constrained by legal defensibility expectations, SMR/SAR consistency concerns, investigative capacity, and customer-impact considerations. Model-use abstraction enables bounded comparison against a stable execution object, but organisations still need explicit decisions about what degree of divergence in alert volumes, typological coverage, SMR/SAR outcomes, or customer treatment is acceptable while a challenger is being assessed.

This is also where the architecture begins to support stronger governance more naturally[13]. Versioning can attach to the right object. Approvals can distinguish between local model changes and orchestration-level changes. Monitoring can distinguish between upstream feature stability and downstream event effectiveness. Feature and event outputs can also be tied back to specific model or model-set versions, making comparison, lineage tracing, and impact assessment materially easier[14]. The value is not better documentation alone; it is the ability to change the estate while retaining clear control over what changed, why it changed, what outputs it produced, and what else it affected.

Orchestration and Responsiveness

When prevention is treated as a control posture on equal architectural footing with detection, responsiveness becomes a core design concern rather than a downstream implementation detail. Many FC controls default to post-event batch, not because that posture is always appropriate, but because it is the path of least resistance. The moment earlier intervention is required, the limits of a domain-bound architecture become clearer: models are not designed for low-latency execution[15], dependencies are embedded in workflows rather than managed explicitly, and integration with services is treated as exceptional rather than architectural. The issue is not whether models can run faster, but whether execution timing, dependency management, and action routing can be aligned to the control posture the RE is trying to support.

In a model-use abstracted architecture, responsiveness becomes a design parameter rather than an accidental property of the platform. Some models run periodically, others run on event, on demand, or only when a downstream decision requires them. The relevant question is not whether the estate is “real-time” or “batch”; it is whether execution cadence is aligned to control purpose. That alignment also has to be understood through dependency. Where a downstream event model is expected to support lower-latency intervention, the upstream feature models it depends on must

[13] The model-use abstraction architecture does not remove the need for organisational policy decisions. In AML especially, challenger testing may still be constrained by defensibility expectations, SMR/SAR consistency concerns, investigative capacity, and customer-impact considerations. Model-use abstraction enables bounded comparison against a stable execution object, but organisations still need explicit decisions about what degree of divergence in alert volumes, typological coverage, SMR/SAR outcomes, or customer treatment is acceptable while a challenger is being assessed.

[14] The same applies to explanation artefacts: if runs are version-bound, the documentation or evidence used to explain those runs must also remain tied to the relevant model and model-set version, otherwise comparison and audit reproduction become materially harder.

[15] In particular, those models with behavioural aggregate components.

either run at a compatible cadence or already make sufficiently current outputs available. In that sense, responsiveness posture is not only a property of the downstream control, but also of the dependency chain that supports it.

This makes the role of orchestration clearer. Orchestration should be understood not as the workflow engine, but as the execution and control layer that schedules model runs, manages dependencies, and selects model sets. Workflow then sits downstream, consuming events and routing them to action. Where those layers are collapsed into one another, reuse becomes more difficult, controlled change becomes harder to bound, and governance tends to remain use-case-specific.

This does not mean that every RE can cleanly externalise orchestration from workflow in one move. Vendor constraints, latency requirements, ownership boundaries, and service dependencies often make the separation partial rather than complete. Even so, the architectural principle still matters, because the clearer the distinction between logic, execution, and action, the greater the scope for reuse, controlled change, and coherent governance. Against that backdrop, it becomes necessary to be explicit about what orchestration must guarantee in an FC context.

What orchestration must guarantee

In this context, orchestration is more than a scheduler. It is the control surface that determines whether model execution remains coherent, auditable, and safe under real operating conditions. At minimum, orchestration should support deterministic execution against defined versions, explicit dependency handling, replayability for audit and investigation, and bounded treatment of failure.

If a model set fails part-way through execution, the architecture must make clear whether the result is partial, retried, abandoned, or rolled back, and whether downstream action is still permitted. This matters particularly in preventative settings, where partial execution, duplicate events, or inconsistent state can have direct customer consequences. The requirement is not tool-specific. It is that execution remains reproducible, version-bound, and governable even where runs are event-driven, low-latency, or distributed across chained components.

Idempotency, replay handling, partial failure isolation, and the treatment of state across executions are therefore architectural requirements for any FC environment that expects to run shared models reliably across multiple control postures.

Orchestration does not remove the underlying modelling difficulties of FC control environments. Data drift, behavioural change caused by successful controls, delayed or partial labels, and contamination of downstream outcomes remain real problems, particularly where preventative controls alter the behaviour later models observe. Model-use abstraction does not solve these issues. What it does do is make them easier to isolate and govern by exposing model boundaries, dependency chains, execution timing, and versioned outputs more clearly than domain-bound implementations typically allow.

Governance Implications

One of the practical consequences of model-use abstraction is that governance can attach to clearer and more reusable objects. When models, features, orchestration, and workflow are embedded within a domain-bound architecture, oversight risks fragmenting with them: inventories, approvals, evidence, monitoring, and change controls can all become use-case-specific even where the underlying analytical logic materially overlaps. The issue is therefore not simply one of policy, but of whether the architecture presents coherent objects to govern in the first place.

A model-use abstracted architecture improves that position by making analytical components, their dependencies, and their roles in execution more explicit. Where models, model sets, features, orchestration, and downstream action are distinguished more clearly, versioning can attach to the right level, dependencies become easier to trace, and change can be evidenced more coherently. Feature-producing components can then be governed with greater emphasis on definition, lineage, stability, and reuse, while event-producing components can be governed with greater emphasis on decision rationale, threshold logic, outcome monitoring, and control effectiveness[16].

For shared feature-producing components, this also implies explicit ownership, service expectations, and escalation paths. If a feature is consumed across multiple downstream controls, it needs to be governed more like a product than a local implementation detail: ownership of definition and quality should be clear, timeliness and stability expectations should be explicit, and downstream consumers should have a defined route for escalation where degradation or drift is detected.

Explainability should also be understood at more than one level. Individual models may need to explain their own logic, thresholds, or scoring behaviour, but model sets and orchestrations also require explanation of how components were sequenced, which versions ran, what dependencies were invoked, and how features or events contributed to the resulting control output. Explanation artefacts therefore need to attach not only to models, but also to versioned model-set executions.

The architectural objects described earlier also provide a natural structure for governance. In practice, different components carry different governance concerns, and those distinctions become clearer when they are mapped explicitly.

Architectural Object	Primary Governance Concern	Typical Evidence / Control Artefact
Data	Lineage, ownership, timeliness, and quality	Data contracts, lineage maps, SLAs, data quality controls
Feature model	Definition, reuse, stability, and drift	Feature specification, ownership, timeliness expectations, drift monitoring outputs

[16] The governance burden attached to analytical components is not uniform across FC domains. Fraud and scams teams often operate with faster preventative control-development cycles and lower regulatory evidentiary overhead than is typical in AML detective controls. Model-use abstraction does not remove those differences, but it gives a clearer basis for applying governance proportionately at the right level: shared upstream components can remain reusable and stable, while downstream event-producing controls can be governed according to their specific regulatory and operational burden.

Event model	Decision rationale, thresholds, and performance	Model documentation, threshold rationale, performance monitoring packs
Model set (orchestration)	Execution integrity, control posture, and effectiveness	Run history, versioned model-set definitions, challenger comparison evidence
Event / output	Action traceability and customer impact	Execution logs, workflow records, service action logs, audit replay artefacts

Table 1 - Mapping Architectural Objects to Governance Responsibilities

The value is not better documentation for its own sake, but a more workable basis for control over change. When the architecture makes it clearer what exists, what changed, what depended on it, and what evidence supports that change, governance becomes more proportionate and more closely aligned to the analytical capability the RE is actually operating.

A Practical Architecture for Shared Modelling Capability

If detective and preventative control postures are to operate as a shared modelling capability rather than a collection of domain-bound stacks, the first useful step is to be clear about what the environment actually needs to do. That means starting with capability requirements rather than products. The key question is not one of vendor or product, but which architectural properties must exist if reusable models, shared features, explicit orchestration, challenger execution, and controlled change are to be made practical.

Core high-level requirements

It is helpful to think of requirements as architectural capabilities rather than as product features. Different environments may realise them in different ways, but the capability itself still needs to exist if the proposition in this paper is to be operationally credible. The requirements below are not exhaustive, but they are the kinds of capabilities a model-use abstracted environment needs to satisfy.

Data and feature foundations

- Shared data access: core customer, account, transaction, counterparty, reference and auxiliary data should be made reusable across use cases rather than repeatedly rebuilt inside domain pipelines.
- Reusable feature production: the estate should support controlled production, storage, and reuse of analytical features across multiple use cases.

Execution and orchestration

- Separation of concerns: model logic, orchestration, workflow, and service action should be separable, even if the separation is only partial in early states.
- Multiple execution modes: the architecture should support batch, event-driven, on-demand, and lower-latency execution where appropriate.
- Multi-use-case consumption: the same features or model outputs should be consumable by TM, CAPS, CRA, ECDD, Fraud, Scams, etc. where relevant.

- Operational routing flexibility: events should be routable to workflow, review, or preventative service action without redefining the underlying model each time.

Change and governance

- Model and model-set versioning: both individual models and orchestrated model sets should be versioned and identifiable as stable design objects.
- Dependency visibility: upstream and downstream dependencies should be visible so that change impact can be assessed before implementation.
- Challenger support: bounded challenger execution and comparison should be possible without recreating an entire use-case stack.
- Explainability and lineage: the estate should preserve enough lineage to explain what ran, on what data, using which version, and what outputs were produced.
- Governance visibility: ownership, approval state, execution history, performance evidence, and change history should be visible for features, models, and model sets.

Monitoring

- Observability: the environment should support operational monitoring, performance monitoring, and drift monitoring across both feature and event models.

These requirements matter to make the proposition in this paper operationally testable. If an environment cannot support them, then it may still contain useful tools, but it is unlikely to function as a coherent shared modelling capability.

What common tooling can look like

These requirements do not imply a single prescribed platform or technology stack. Different REs will make different choices depending on scale, latency needs, legacy constraints, security requirements, and internal engineering maturity. Even so, once the architecture is organised around reusable modelling capability rather than domain-bound stacks, a recognisable set of technical roles tends to emerge.

In practical terms, an estate designed this way will usually require tooling[17] that can fulfil the following roles:

- object or table storage for raw and curated data — for example MinIO, Apache Iceberg, Delta Lake, PostgreSQL, or DuckDB for smaller-scale analytical workloads
- a transformation and compute layer for feature engineering and controlled transformations — for example Apache Spark, Flink, or SQL-based transformation pipelines
- a feature-serving or reusable analytical storage layer — for example Feast as an open-source feature store, or analytical tables in SQL
- a registry for models, model sets, and associated version history — for example MLflow Model Registry, with model versions, metadata, and lifecycle controls

[17] My own preference, where feasible, is for free and open-source tooling or open standards-based components, particularly for storage, transformation, orchestration, registry, observability, and model execution. That preference is driven less by ideology than by portability, transparency, cost control, and the ability to engineer capability without making vendor configuration the only place where modelling logic can live.

- an orchestration tool to run model sets and manage dependencies — for example Dagster, Apache Airflow, or Temporal, depending on whether the dominant need is data orchestration, scheduled pipelines, or durable execution across services
- model-serving components for lower-latency or API-based execution — for example BentoML or KServe
- workflow or case-management tooling for investigative handling — for example ServiceNow, or a vendor FC case-management layer, depending on whether the RE prefers BPM-style workflow, enterprise case handling, or domain tooling
- service gateways or decision APIs for preventative controls — for example internal policy / decision services, Temporal-style service orchestration, or API gateways sitting in front of product controls
- monitoring and observability tooling for execution health, drift, and performance evidence — for example Prometheus and Grafana for runtime monitoring, with lineage or observability support layered in through platform-native or open standards-based components

What matters is not whether those capabilities sit in one product or several. What matters is that the major technical roles required for reusable modelling are explicit, governable, and connected through clean interfaces. In some REs, a vendor TM or screening platform may still sit inside this proposed architecture, increasingly operating as a downstream execution or consumption environment rather than as the exclusive home of all modelling logic.

A practical adoption path

The target-state architecture can sound more ambitious than it is. It is not an instruction to rebuild the estate wholesale; it is a proposition to change the unit of design over time[18].

In many environments, especially those anchored to vendor FC platforms, this will be a constrained uplift rather than a greenfield redesign. The practical objective is not immediate ideal-state abstraction, but progressive extraction of reusable components where the architecture and operating model permit it.

A practical adoption path begins with inventory rather than platform change. Existing logic across TM, CAPS, CRA, etc. and adjacent domains should be catalogued with an explicit focus on repeated patterns. In most REs, the same primitives will appear repeatedly: segmentation, scoring, entity resolution, profiling, thresholds, and aggregation. That exercise usually reveals that the estate is less heterogeneous than it appears at workflow level and more repetitive than governance artefacts suggest.

The next step is extraction. Primitives that are repeatedly rebuilt should be lifted into shared feature models, given defined ownership, and treated as governed products rather than hidden implementation details. That is where much of the real reuse will come from, and it is also where much of the architectural debt typically sits.

[18] In many environments, especially those anchored to vendor FC platforms, this will be a constrained uplift rather than a greenfield redesign. The practical objective is not immediate ideal-state abstraction, but progressive extraction of reusable components where the architecture and operating model permit it.

A minimal registry[19] should exist early. If the organisation cannot state what models exist, who owns them, what depends on them, and where they run, it does not yet have the basis to govern change. The registry is not an administrative afterthought; it is the mechanism that makes dependencies, ownership, and lineage visible. Reuse without visibility is fragile, and change without registry, ownership, and lineage is difficult to govern.

From there, one constrained use case can be re-expressed as an orchestration of models with an explicit dependency chain. That is the point at which a challenger set becomes meaningful: not as an abstract aspiration, but as a bounded comparison against a known champion configuration. The objective is not to perfect the estate in one move, but to prove that controlled change with evidence is possible.

Integration with action should then be approached deliberately. The key question is which events route to detection and which route to prevention. Scale should come through reuse, not replication: shared primitives should exist before the approach is extended across domains, otherwise the current state is simply recreated at a larger scale.

This is better understood as a direction of travel than as a fixed end-state. The practical objective is not architectural perfection, but a detection capability that becomes progressively more reusable, governable, and adaptable as common primitives are extracted and use cases are reorganised around them.

An Illustrative Runtime Architecture for Model-Use Abstraction

The preceding sections describe the architectural principles underlying model-use abstraction and the capabilities required to support shared modelling across financial crime use cases. To ground those principles, Figure 5 presents an illustrative runtime execution view of a model-use abstracted architecture.

This view is not intended as a product-level design or a prescriptive platform blueprint. Instead, it shows how reusable analytical models, orchestration, feature persistence, and event control routing can operate together at runtime as a coherent execution capability, supporting both detective and preventative control postures without complete redundant presentation of core analytical logic.

[19] The term "registry" here is not intended to imply a specific technology platform. In early or lower-maturity environments, it may be as simple as a maintained register of models, rules, and features (for example, a spreadsheet or controlled document) capturing ownership, dependencies, execution context, and version history. The critical requirement is not tooling, but that these relationships are explicitly recorded, maintained, and governable.

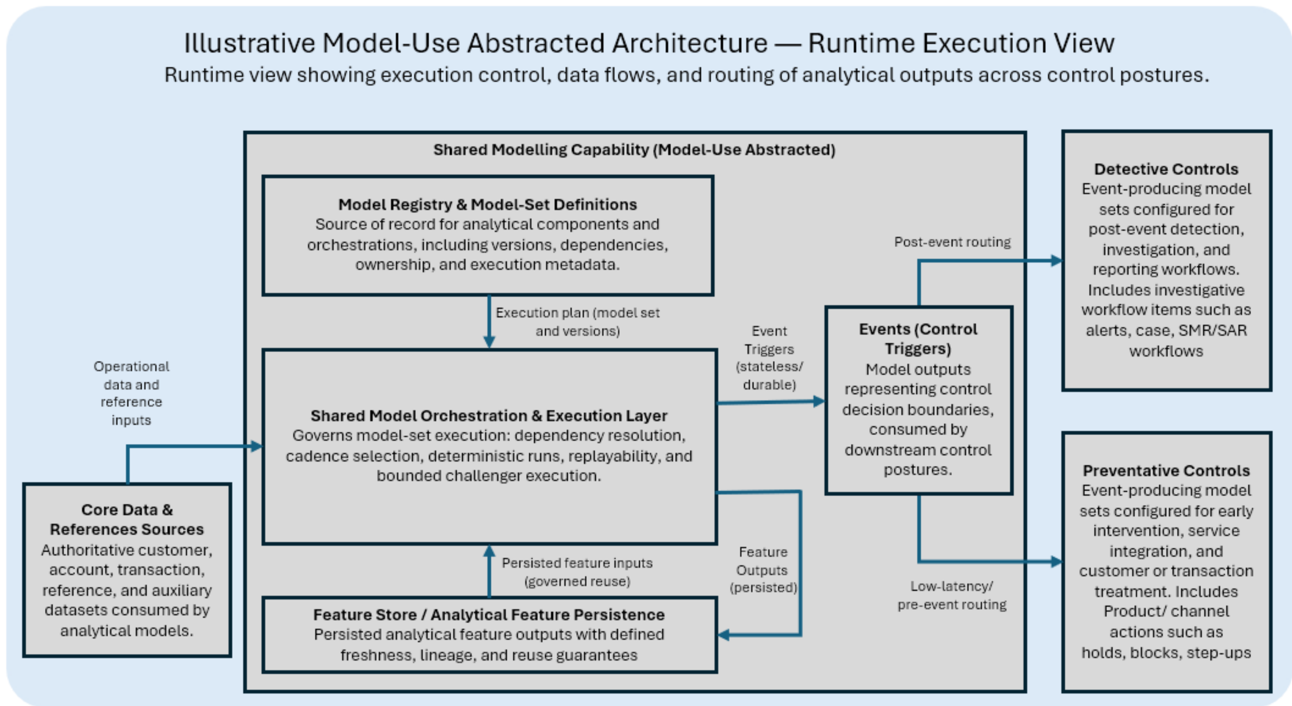

Figure 5 - Illustrative model-use abstracted architecture (runtime execution view).

Read from left to right, the figure illustrates how authoritative data is consumed by a shared execution capability operating under explicit orchestration control. Versioned model sets in the registry are instantiated deterministically, producing either persisted features for reuse or events that represent control decision boundaries. Those events are then routed according to control posture: post-event to investigative workflows for detective controls, or at lower latency to service and product integrations for preventative outcomes, or possibly both.

The key point is not the specific components shown, but the separation of concerns they reinforce. Analytical logic is defined and governed once, execution behaviour is controlled centrally through orchestration, and differences between detection and prevention are expressed through timing and routing rather than through separate stacks. This runtime perspective makes explicit how model-use abstraction translates from a conceptual organising principle into an operationally executable capability.

Closing Thoughts

REs often treat detection and prevention analytic controls as a technology problem, or as a collection of domain-bound control implementations. This paper has argued for a different view: that detection and prevention analytic controls can be better understood as a shared modelling capability, made up of reusable models, features, orchestrations, and outputs that can be applied across multiple FC use cases and multiple control postures.

That shift matters because it changes what can realistically be reused, what governance can sensibly attach to, and what can be adapted as risk and operating conditions change. When analytic controls remain organised as domain-bound stacks, REs tend to accumulate logic that is hard to reuse, hard to govern consistently, and awkward to repurpose. Treated more explicitly as a capability, the same

estate becomes easier to reuse, easier to test in bounded ways, and easier to extend across detective and preventative control postures without turning each change into a special project.

The value of the proposition is not abstraction for its own sake, but the creation of a more controlled, explainable, and adaptable execution layer for financial crime analytic controls. Models become explicit design objects and features become reusable outputs. In practical terms, the RE regains optionality: the estate becomes easier to adapt, easier to challenge in a bounded way, and easier to explain. Detection stops being something inherited from platforms, projects, and organisational boundaries, and becomes something that can be designed more deliberately.

References

[1] AUSTRAC. (2025). Preventing financial crime using a risk-based approach. Available at: https://www.austrac.gov.au/business/core-guidance/preventing-financial-crime-using-risk-based-approach [Accessed 2026-03-09].

[2] AUSTRAC. (2025). Transaction monitoring. Available at: https://www.austrac.gov.au/business/core-guidance/amlctf-programs/transaction-monitoring [Accessed 2026-03-09].

[3] Financial Conduct Authority (FCA). (2018). FG18/5: Guidance on financial crime systems and controls. Available at: https://www.fca.org.uk/publication/finalised-guidance/fg18-05.pdf [Accessed 2026-03-09].

[4] Wolfsberg Group. (2024). The Wolfsberg Group Statement on Effective Monitoring for Suspicious Activity, Part I: Moving Beyond Automated Transaction Monitoring. Available at: https://dev.wolfsberg-group.org/resources/innovation/168 [Accessed 2026-03-09].

[5] Australian Treasury. (2025). Scams Prevention Framework – Protecting Australians from scams. Available at: https://treasury.gov.au/publication/p2025-623966 [Accessed 2026-03-30].