JANUARY 23RD, 2026

Rack Vertex CML Health Check

Prepared For
John Smith Rack Vertex

Floor 18, 854 Mission St, San Francisco, CA 98452
john.smith@acmecorp.com

1.1. Executive Summary

This report presents the results of the RCA solution assessment conducted to evaluate the architectural integrity, configuration quality, performance characteristics, and maintainability of the reviewed Revenue Cloud implementation.

The objective of the audit was to assess the overall solution structure and identify areas for architectural alignment, optimization opportunities, and structural improvements that may influence long-term scalability and operational efficiency.

While the assessment considered the broader RCA context, the primary technical focus of the review was the CML configuration layer, as it represents the core logic governing product configuration behavior.

The reviewed organization recently completed an iterative Revenue Cloud implementation executed across multiple delivery phases. The solution is operational and supports active business scenarios. The audit establishes a structured architectural baseline and provides prioritized recommendations for continued solution evolution.

1.1.1.1 Findings Summary

The findings identified during the audit have been classified according to severity based on architectural impact, performance considerations, and maintainability implications.

The table below summarizes the distribution of findings identified within the scope of this audit:

Risk Score	Number of Findings
Critical	1
High	3
Medium	7
Low	2
Total	**13**

- **1 - 🟥 “Critical” Findings**
  Architectural or structural issues that pose immediate risk to performance, scalability, or commercial governance and require prioritized remediation.

  Hardcoded Pricing and Commercial Terms in CML
  The audit identified pricing and discount calculations embedded directly in the CML configuration layer. This creates a structural overlap between configuration logic and the Revenue Cloud pricing

engine, reducing transparency and increasing governance risk.
Recommended action: remove commercial calculations from CML and shift all pricing logic to native pricing mechanisms to restore proper separation of responsibilities.

- **3 - 🟧 “High” Findings**
  Significant technical concerns that may impact maintainability or efficiency if not addressed in the near term.
  - **Omitting domain definition for attributes/relations**
    Several attributes and relations lack explicit domain definitions, expanding the configuration search space unnecessarily.
    Recommended action: introduce bounded attribute ranges and controlled relation cardinalities to improve performance and predictability.
  - **Suboptimal constraint usage**
    Constraints are defined at higher hierarchy levels and reference nested entities, increasing logical coupling and runtime processing complexity.
    Recommended action: localize constraint logic closer to dependent entities and reduce cross-level references.
  - **Excessive SKU Proliferation for Preconfigured Variants**
    Multiple preset configurations are modeled as separate product records instead of using dynamic CML-driven presets. This increases catalog size and long-term maintenance overhead.
    Recommended action: consolidate preset SKUs into configurable base products using CML table constraints.
- **7 - 🟨 “Medium” Findings**
  Optimization opportunities and structural improvements that enhance long-term solution health and stability.
- **2 - 🟩 “Low” Findings**
  Minor best-practice deviations or structural refinements with limited operational impact.
- **13 - Total Number of Findings**
  Total issues identified within the audit scope across all severity levels, reflecting overall solution maturity and optimization potential.

While the solution remains operational, the presence of these high/critical issues indicates the need for targeted remediation to ensure scalability, maintainability, and alignment with Revenue Cloud best practices.

2.1.2 Estimated Remediation Effort Overview

Based on the identified findings, the estimated remediation effort has been calculated across engineering and testing activities. The estimates reflect the effort required to implement the proposed changes and validate them through unit-level testing.

Estimated Effort Summary

Scope	Total Estimated Effort
Critical & High Findings	**48h – 96h**
All Findings (Full Remediation)	**152h – 296h**

Addressing only the critical and high-severity findings would significantly reduce architectural and commercial risk exposure with a relatively contained remediation scope. This phase primarily focuses on correcting structural modeling patterns, restoring separation of concerns, and improving performance-critical constraint logic.

Full remediation would further enhance long-term maintainability, catalog governance, and performance optimization, bringing the solution into closer alignment with Revenue Cloud best practices and scalable architecture standards.

3.2. Scope and Methodology

The audit was conducted as an architectural assessment of the client's Revenue Cloud Advanced solution.

The review focused on evaluating the structural integrity, configuration logic implementation, and architectural consistency of the solution. Particular attention was given to the CML, as it forms the core mechanism for enforcing product rules, bundle behavior, and configuration validation within the quoting process.

The scope of the assessment included architectural review of the CML models and their interaction within the broader RCA configuration framework. The review did not include pricing procedures, CRM data structures, integrations, or runtime load testing.

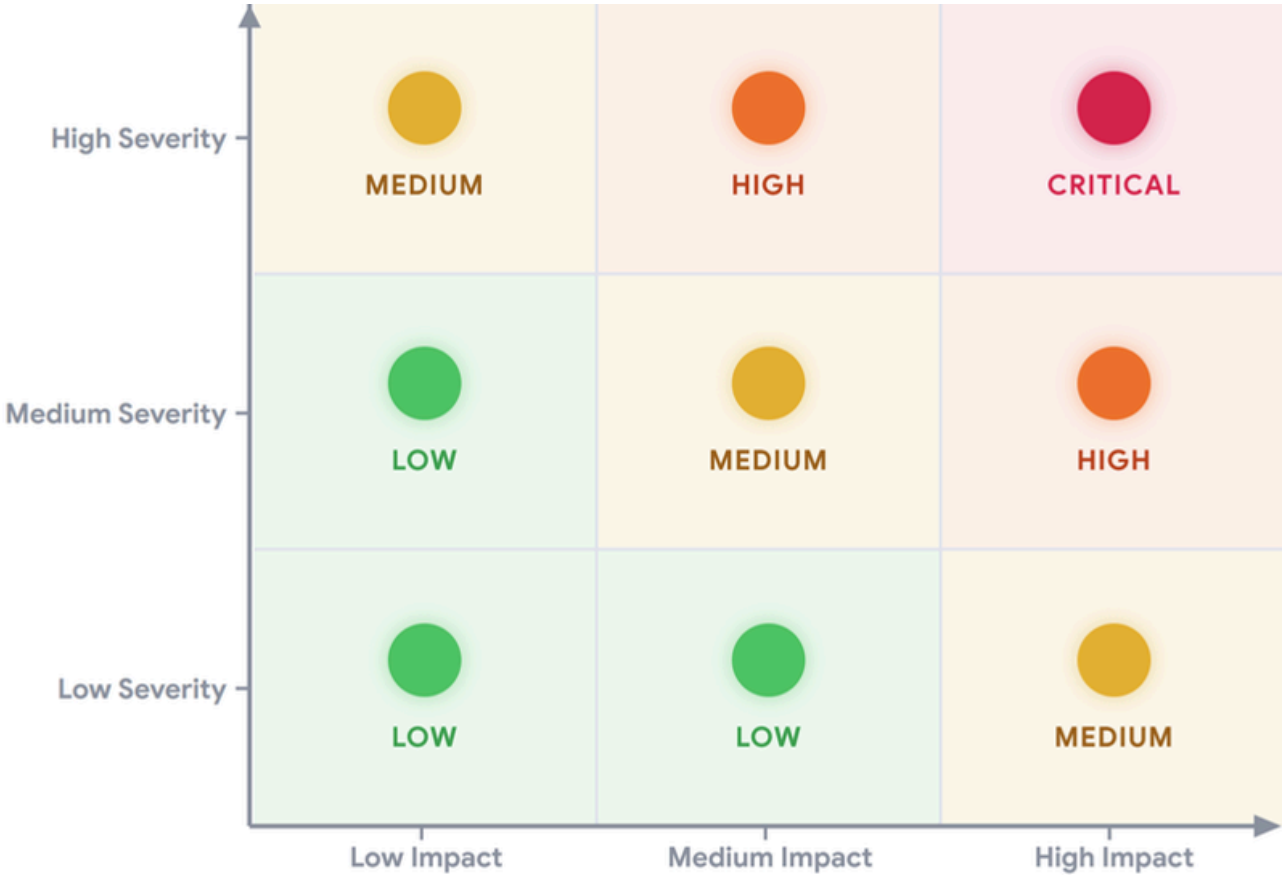
The assessment combined manual expert analysis with structured model-level evaluation techniques. Observations were documented in a standardized format and classified according to severity to support prioritization and structured remediation planning.

A detailed description of the reviewed solution and assessment approach is provided in Section 4 of this report.

3.1.2.1 Overall Risk Score Matrix

The Overall Risk Score Matrix provides a high-level visual representation of the overall health and risk exposure of the reviewed CML solution. The matrix is used to summarize the combined effect of identified findings by evaluating two key dimensions: issue severity and business/technical impact.

- The vertical axis (Severity) reflects how critical a specific issue is from a technical or operational perspective, based on factors such as system stability, configurator correctness, performance degradation, and maintainability risks.
- The horizontal axis (Impact) represents the potential business or operational consequences of the issue, including effects on configuration performance, sales operations, scalability, user experience, and delivery timelines.



The combination of these two dimensions determines the final **Risk Score**, ensuring that both technical depth and business exposure are reflected in the following classification:

- **Low**
  The reviewed solution shows only minor issues or best-practice deviations that have limited operational or business impact. No immediate remediation is required, and improvements can be addressed gradually as part of regular optimization activities.
- **Medium**
  The solution contains several issues that may affect performance, scalability, maintainability, or operational efficiency over time. While not immediately critical, remediation planning is recommended to prevent potential risks from increasing as the solution evolves.
- **High**
  Significant issues have been identified that can materially affect system performance, operational stability, or delivery timelines. Remediation actions should be prioritized and scheduled in the near term to reduce operational risks and prevent further impact.
- **Critical**
  The solution contains severe issues that present immediate risks to business operations, configuration correctness, or system stability. Immediate remediation is strongly recommended to prevent service disruption, incorrect configurations, or major operational failures.

3.2.2.2 Radar Chart

The Radar Chart provides a consolidated view of solution maturity across the core architectural and operational dimensions assessed during the audit. Each axis represents a key evaluation area:

- **Efficient Constraint Processing**
  Efficiency of configuration evaluation, constraint modeling quality, and overall runtime behavior of the configuration engine.
- **Optimized Catalog Architecture**
  Structural design of bundles and products, hierarchy clarity, reuse patterns, and SKU governance discipline.
- **Complete Pricing & Logic Decoupling**
  Separation of commercial parameters from configuration logic, alignment with Revenue Cloud pricing models, and pricing flexibility.
- **High Maintainability & Code Standards**
  Code structure, reuse patterns, naming conventions, domain definitions, and adherence to modeling best practices.
- **Operational Readiness**
  Governance model, documentation quality, ownership clarity, deployment practices, and long-term supportability.

The resulting Radar Chart highlights structural balance across these dimensions and identifies areas requiring architectural refinement.

4.3. Current State Assessment

The reviewed Revenue Cloud solution has completed core implementation activities and progressed through functional validation cycles, including user acceptance testing. The configuration model reflects the intended business scenarios and supports the defined product structure within the current testing scope.

At the time of the assessment, the solution is in a late pre-production stage, with go-live preparation underway. The audit was initiated as a structured architectural review prior to full production deployment, providing an independent evaluation of the configuration model before transition to live operations.

While the implementation demonstrates functional completeness, this assessment focuses on evaluating structural quality, maintainability, performance characteristics, and overall architectural alignment to ensure readiness for long-term operational use.

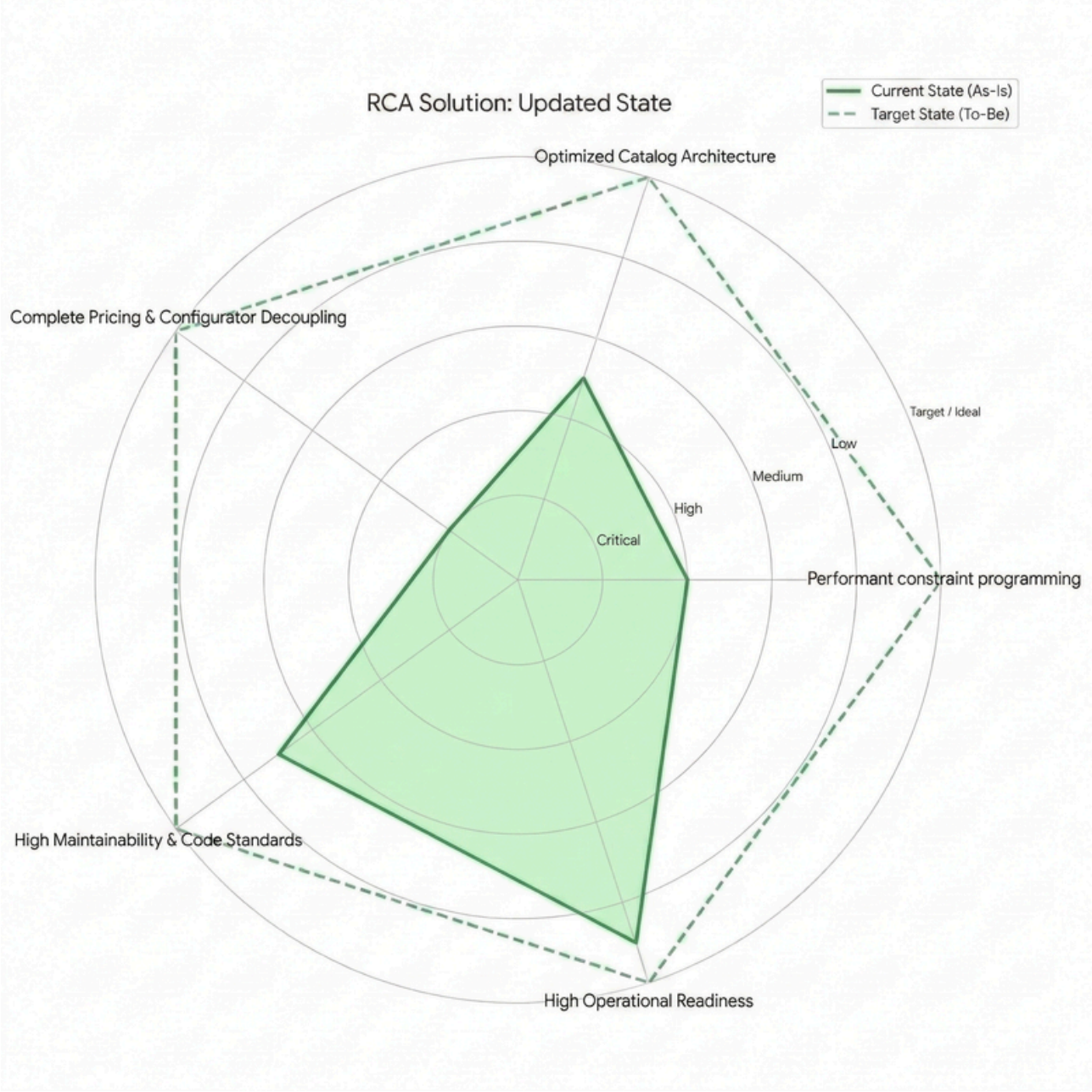
4.1.3.1 Overall Risk Level

Overall Risk Level: **High**

The solution contains several performance-impacting modeling patterns and structural maintainability issues that, while not currently blocking operations, introduce scalability risks and increase long-term operational overhead. Remediation planning is recommended in the near term.

4.2.3.2 Technical Health Overview

The Radar Chart below illustrates the current state of the architecture across five key dimensions compared to the recommended target state:



Radar Chart Axis	Findings (ID/ Topic)	Rationale & Scope

Efficient Constraint Processing	**F-001, F-003, F-004, F-006, F-007**	Includes all factors slowing down the CML engine: inefficient tree-search paths, lack of domain boundaries, suboptimal constraint propagation, and excessive decimal precision.
Optimized Catalog Architecture	**F-002, F-003, F-010**	Targets the "SKU explosion" pattern: duplicating product records for different sales contexts (standalone/bundle), billing models (one-time/sub), or static pre-configurations.
Complete Pricing & Logic Decoupling	**F-011**	Addresses the highest technical risk: embedding commercial terms (terms, price calculations, and discounting) directly within the technical CML configuration layer.
High Maintainability & Code Standards	**F-002, F-004, F-005, F-012, F-013**	Focuses on "code debt" and lifecycle costs: redundant type definitions, lack of shared constants (define), disorganized file structure, and unnecessary metadata tagging.
Operational Readiness	*N/A (Business Value)*	Represents the current functional state. It indicates that despite the technical risks listed above, the system is meeting immediate business configuration requirements.

Key Observations:

- **Operational Success vs. Technical Debt:** While **Functional Alignment** is high (the system works for the sales team today), it is achieved through a high degree of "technical debt" in other areas.
- **Critical Rigidity:** The **Logic Decoupling** score (1.0) represents the most significant risk. Hardcoding commercial terms within the CML engine prevents the business from reacting quickly to market changes without developer intervention.
- **Performance Bottlenecks:** The **CML Performance** score (1.5) indicates that as the product complexity or user load increases, the system is likely to experience significant latency due to suboptimal constraint processing.
- **Catalog Expansion Risk:** The current **Catalog Architecture** is not optimized for growth. The practice of creating new SKUs for every permutation will lead to exponential maintenance costs.

This assessment serves as a baseline for the prioritized remediation roadmap. Addressing the "Performance" and "Decoupling" axes will yield the highest immediate ROI for system stability and business agility.

5.4. Prioritized Issues & Recommendations

The following section presents the findings identified during the assessment.

5.1.“Critical” Findings:

- Rack Vertex. Finding №11

5.2.“High” Findings:

- Rack Vertex. Finding№3
- Rack Vertex. Finding№4
- Rack Vertex. Finding №10

5.3.“Medium” Findings:

- Rack Vertex. Finding№1
- Rack Vertex. Finding №2
- Rack Vertex. Finding №5
- Rack Vertex. Finding№6
- Rack Vertex. Finding№7
- Rack Vertex. Finding№8
- Rack Vertex. Finding №9

5.4.“Low” Findings:

- Rack Vertex. Finding№12
- Rack Vertex. Finding №13

6.5. Architectural Recommendations

Based on the comprehensive audit of the Revenue Cloud solution, several structural and modeling anti-patterns were identified across Catalog Management, Pricing, and CML Configuration. To ensure long-term scalability, performance, and maintainability, we recommend a strategic remediation approach focused on four core architectural pillars.

6.1.A. Product Catalog Rationalization (Single Source of Truth)

The current catalog architecture suffers from severe SKU proliferation due to a misunderstanding of CPQ capabilities.

- **Consolidate Contextual SKUs:** Eliminate duplicate product records created for different sales scenarios (e.g., bundled vs. standalone) or different billing models (one-time vs. subscription). Leverage standard Revenue Cloud features such as Product Selling Models, Subscription Pricing, and Context-Aware Pricing Rules to utilize a single physical product record across all scenarios.
- **Dynamic Configurations over Hardcoded Variants:** Stop bypassing the CML engine with hardcoded SKUs for product presets. Transition to unified, highly configurable base products that utilize CML `table` constraints to dynamically apply predefined configurations based on user selection.

6.2.B. Strict Separation of Commercial and Technical Concerns

Mixing business/pricing logic with technical configuration introduces severe inflexibility and revenue risk.

- **Decouple Pricing from CML:** Eradicate all hardcoded commercial data (e.g., subscription term lengths, pricing tiers) from the CML models. The CML engine must exclusively govern the technical validity of a product. Commercial rules must be managed via standard CPQ Pricing Engines, Price Books, and dynamic context variables passed from the Quote Line into the configuration session.

6.3.C. Constraint Engine Performance Optimization

The CML models currently force the solver into heavy computational overhead. Optimizing how constraints are evaluated will drastically improve configuration response times.

- **Maximize Arc-Consistency:** Refactor inline expression variables into explicit constraint-based assignments to prevent the solver from evaluating unnecessarily large search trees.
- **Bound the Search Space:** Enforce explicit domain ranges for numeric attributes and define strict cardinality limits for relations (e.g., `int capacity[0..1000]`). Unbounded domains force the engine to evaluate millions of unnecessary paths.
- **Optimize Propagation Mechanisms:** Replace boolean expression variables that act as constraint drivers with **Named Constraints**. This ensures the conditions fully participate in solver domain

propagation. Additionally, limit decimal precision (e.g., `decimal(2)`) where maximum system precision is not functionally required.

6.4.D. Model Maintainability and Structural Integrity

The codebase currently contains excessive duplication, largely driven by auto-generated Visual Builder structures and suboptimal logic placement.

- **Abstract and Reuse:** Transition away from bloated, flat type definitions where every product has its own identical structure. Implement generic, reusable CML `type` definitions that encapsulate shared attributes and behaviors, and reference them via relations.
- **Centralize Shared Domains:** Utilize the CML `define` statement to extract globally reused domains (e.g., standard picklist values) into single constants, preventing redundancy and simplifying future updates.
- **Localize Configuration Logic:** Eliminate cross-level constraints where parent bundles directly manipulate grandchild attributes. Push logic down closer to the dependent entity and utilize the `parent()` function to maintain loose coupling and high structural transparency.

7.6. Reference Materials

- Internal CML Modeling Best Practices
- Configuration Performance Optimization Guidelines
- Constraint Engine Modeling Recommendations
- Standard Domain Definition Patterns
- https://resources.docs.salesforce.com/rel1/doc/en-us/static/pdf/CML_User_Guide.pdf

8.Rack Vertex. Finding №1

8.1.Tree search vs Arc consistency

Finding ID	Risk Score	Category	Affected Area	Estimated Effort
F-001	Medium 🟨	Performance	CML Configuration Model / Constraint Processing	16h - 32h

8.1.1. Description

Several configuration expressions are implemented using inline expression variables that force the constraint engine to resolve large tree-search evaluation paths during runtime. This modeling approach leads to the creation of complex evaluation trees containing a large number of intermediate nodes that must be traversed before a valid solution can be derived.

Instead of decomposing calculations into constraint-based assignments, the current approach embeds expressions directly into variable definitions. This prevents the engine from efficiently applying arc-consistency propagation techniques and results in significantly higher computational overhead during configuration evaluation.

```
...

boolean isCableTopEntry = cableEntry == 'Top';

...


constraint(isCableTopEntry -> cableManagementkit[CableManagementKit] > 2);


...
```

8.1.2. Impact

The inefficient constraint modeling approach may lead to increased configuration calculation times, especially in complex bundles with multiple dependent expressions. As the configuration model grows in size and complexity, the performance impact can become more pronounced, potentially affecting configuration responsiveness, user experience, and system scalability.

8.1.3. Recommendation

Refactor expression-based variable definitions into constraint-based assignments that allow the constraint solver to apply arc-consistency propagation. Defining intermediate calculation variables separately and linking them through explicit constraints enables more efficient evaluation and reduces unnecessary search-tree expansion.

```
...

constraint(cableEntry == 'Top' -> cableManagementkit[CableManagementKit] > 2);

...
```

8.1.4.Quick Wins

Prioritize refactoring of high-frequency or performance-sensitive calculation areas by replacing inline expression variables with explicit constraint definitions. Even partial adoption of constraint-based modeling in performance-critical sections can provide immediate configuration performance improvements.

8.1.5.Estimated Effort

The remediation effort is estimated as “*Medium*”, as the change requires refactoring existing calculation definitions across affected CML components, validating resulting constraint behavior, and regression-testing configuration scenarios to ensure that functional outcomes remain unchanged while performance improves.

9.Rack Vertex. Finding №2

9.1.CML structure generated by Visual Builder

Finding ID	Risk Score	Category	Affected Area	Estimated Effort
F-002	Medium 🟨	Maintenance	CML Model Structure / Type Definitions	16h - 32h

9.1.1. Description

The current CML model contains a large number of auto-generated types and relations created by Visual Builder, where each individual product is represented by a dedicated type definition with identical attribute structures. This modeling approach results in repeated type definitions containing the same attribute sets and behavior, significantly increasing the size and structural complexity of the CML model.

Because each product is modeled as a standalone type, any future attribute updates, logic changes, or structural adjustments must be applied repeatedly across multiple types, increasing the probability of inconsistencies and making long-term maintenance more difficult. The current approach also reduces overall model readability and complicates navigation and troubleshooting for development teams.

```
...

type RackRackSystem42U : LineItem {
  relation rackframe : RackFrame;
  relation frontdoor : FrontDoor;
  relation reardoor : RearDoor;
  relation mountingrails : MountingRails;
  relation basepanel : BasePanel;
  relation coolingunit : CoolingUnit;
  relation monitoringmodule : MonitoringModule;
  relation smartlock : SmartLock;
  relation environmentalsensorkit : EnvironmentalSensorKit;
  relation fireSuspensionmodule : FireSuspensionModule;
}

...
```

9.1.2. Impact

Excessive duplication of type definitions increases maintenance effort, slows down future enhancements, and introduces a higher risk of inconsistent configuration behavior when updates are applied unevenly across duplicated structures. As the catalog grows, the complexity and size of the configuration model will continue to expand, potentially affecting long-term scalability and maintainability of the solution.

9.1.3. Recommendation

Refactor the model to use generic reusable product types that encapsulate shared attributes and behavior. Individual products should reference these generic types through relations instead of defining product-specific types with duplicated attribute structures. This approach significantly reduces model size, improves readability, and simplifies future maintenance and enhancements.

```
...

// All of the previous components will be assigned to this one time
// because they all share the same behavior and logic
type RackComponent;

type RackRackSystem42U : LineItem {
   relation rackComponents : RackComponent;
}
...
```

9.1.4.Estimated Effort

The remediation effort is estimated as “*Medium*”, as the change requires restructuring type definitions, updating relations and associations referencing existing product-specific types, and validating that configuration behavior remains functionally consistent after the refactoring.

10.Rack Vertex. Finding №3

10.1.Omitting domain definition for attributes/relations

Finding ID	Risk Score	Category	Affected Area	Estimated Effort
F-003	High 🟧	Performance	CML Attribute and Relation Definitions	8h - 16h

10.1.1. Description

Several attributes and relations within the CML model are defined without explicit domain constraints. When domains are not specified, the configuration engine assumes default value ranges that may be excessively broad (e.g., full integer ranges or unrestricted relation cardinalities). As a result, the solver must evaluate a significantly larger search space when processing configuration logic.

In the reviewed model, attributes such as numeric values and relations representing product associations are defined without bounded ranges or cardinality constraints. This modeling approach unnecessarily expands the number of possible evaluation paths the engine must consider during constraint solving.

```
...

relation rackRackSystem42U: RackRackSystem42U;

...

constraint(rackRackSystem42U[RackRackSystem42U] <= 1);
```

...

10.1.2. Impact

Excessively wide domains increase solver computation time and may negatively affect configuration performance, particularly in complex bundles or high-volume configuration scenarios. As the model grows, the cumulative performance impact may become more noticeable, affecting configuration responsiveness and scalability.

10.1.3. Recommendation

Define explicit domain ranges for numeric attributes and cardinality limits for relations wherever feasible. Restricting attribute ranges and relation cardinalities reduces the solver search space, allowing the

configuration engine to evaluate constraints more efficiently and improving overall configuration performance.

```
// This approach also allows to get rid of constraints that limit the domain
...

relation rackRackSystem42U: RackRackSystem42U[0..1];

...
```

10.1.4.Quick Wins

Identify frequently used numeric attributes and high-volume relations that currently lack domain constraints and introduce reasonable bounded ranges based on real business limits. Even partial domain restriction across performance-critical model areas can produce measurable performance improvements.

```
...

int capacity; // [-2147483648..2147483647] is bound by default
int capacityOptimized[0..1000]; // Should satisfy all of the scenarios. 4.29KK lower

...
```

10.1.5.Estimated Effort

The remediation effort is estimated as “*Low*”, as the change primarily involves adding domain definitions to existing attributes and relations, followed by validation testing to ensure that newly introduced bounds accurately reflect business requirements and do not restrict valid configuration scenarios.

11.Rack Vertex. Finding №4

11.1.Suboptimal constraint usage

Finding ID	Risk Score	Category	Affected Area	Estimated Effort
F-004	High 🟧	Maintenance, Performance	CML Configuration Logic / Constraint Modeling	8h - 16h

11.1.1. Description

The reviewed CML models include constraint definitions implemented at higher hierarchy levels that directly reference attributes of nested child or grandchild entities. Although this modeling pattern is functionally valid and produces the intended configuration behavior, it introduces indirect logical dependencies across multiple hierarchy layers. During execution, such cross-level constraints require the engine to internally generate additional virtual attributes and auxiliary constraint structures to maintain logical consistency.

This approach increases structural coupling between parent and nested entities and reduces transparency of logical ownership within the model.

```
...

type RackBundle : LineItem {
  ...

  constraint(
    isSystem42uSelected ->
    rackRackSystem42U[RackRackSystem42U].fireSuspensionmodule[FireSuspensionModule]
== 1
   );
}

...
```

11.1.2. Impact

The current implementation does not prevent correct system operation. However, cross-level constraint usage increases solver processing complexity and introduces additional internal evaluation steps. Over time, this may contribute to increased configuration latency, reduced model readability, and greater effort required for structural modifications. As the solution expands, this pattern may negatively influence scalability and long-term maintainability.

11.1.3. Recommendation

Move the logic closer to the dependent entity and use the parent() function to reference bundle-level attributes where needed. In this case, defining the Addon attribute using parent() eliminates the need for a cross-level constraint entirely. This approach simplifies the model, improves readability, and reduces runtime overhead by removing redundant constraint propagation.

```
...

type RackRackSystem42U {
  ...

  boolean isRootSystem42uSelected = parent(isSystem42uSelected);

  ...

  relation fireSuspensionmodule : FireSuspensionModule;

  ...

  constraint(isRootSystem42uSelected ->
fireSuspensionmodule[isRootSystem42uSelected] == 1);
}

...
```

11.1.4.Estimated Effort

The remediation effort is estimated as “Low”, as the change involves replacing the cross-level constraint with a localized attribute definition. The refactor is limited in scope and primarily requires regression validation to confirm that the configuration behavior remains consistent after the update.

12.Rack Vertex. Finding №5

12.1.Use `define` when possible

Finding ID	Risk Score	Category	Affected Area	Estimated Effort
F-005	Medium 🟨	Maintenance	CML Type Definitions / Shared Domains	8h - 16h

12.1.1. Description

The CML model repeats identical picklist domain values across multiple types. This duplication introduces avoidable maintenance overhead and increases the risk of inconsistencies when domain values need to be updated in the future.

Instead of defining the same domain separately in each type attribute, CML supports extracting shared domains into a single global definition using define, which can then be referenced across all relevant attributes.

```
...

string   powerTier = ['UpTo3kW', 'UpTo5kW', 'UpTo10kW', 'UpTo20kW', '20kWPlus'];

...
```

12.1.2. Impact

Duplicated domain definitions make the model harder to maintain and more error-prone over time. If picklist values need to be adjusted, the change must be implemented and verified in multiple places, increasing the likelihood of missed updates and inconsistent configuration behavior.

12.1.3. Recommendation

Extract commonly reused picklist domains into global constants using define, and reference these constants in all attributes that share the same domain. This reduces duplication, improves readability, and centralizes future changes to a single location.

```
...
define POWER_TIER ['UpTo3kW', 'UpTo5kW', 'UpTo10kW', 'UpTo20kW', '20kWPlus']

...

string   powerTier = POWER_TIER;

...
```

12.1.4.Quick Wins

Identify the most frequently repeated picklists and refactor them first into global `define` statements. This yields immediate maintainability improvements with minimal risk.

12.1.5.Estimated Effort

The remediation effort is estimated as "*Low*", as the change is primarily a structural refactor: introducing one or more define constants and updating attribute definitions to reference them. Regression testing is recommended to ensure no behavioral changes were introduced, but functional risk is typically limited since only domains are being centralized.

13.Rack Vertex. Finding №6

13.1.Named constraints

Finding ID	Risk Score	Category	Affected Area	Estimated Effort
F-006	Medium 🟨	Performance	CML Constraint Modeling / Domain Propagation	8h - 16h

13.1.1. Description

The model uses an expression variable to calculate a logical condition and then references that variable inside a constraint. While this is valid CML, it can be suboptimal for solver execution. When a condition is implemented as an expression variable, it may be removed from effective domain propagation during execution, which can force the solver to spend additional cycles resolving the constraint and the variable separately.

This pattern is most commonly introduced when teams aim to improve readability or reuse a condition in multiple places. However, in constraint-based models, encapsulating reusable logic as a named constraint is preferred because it preserves solver propagation benefits while still reducing code duplication.

```
...

boolean isCableTopEntry = cableEntry == 'Top';

...

constraint(isCableTopEntry -> cableManagementkit[CableManagementKit] > 2);
constraint(isCableTopEntry -> hasSidePanels == false);
```

...

13.1.2. Impact

This approach can cause avoidable performance overhead, particularly in bundles with higher relation cardinalities or in models where the condition is evaluated frequently. The solver may require additional iterations to reach a consistent state because the expression variable does not participate in propagation as effectively as a named constraint. Over time, widespread use of this pattern can contribute to slower configuration evaluation and reduced scalability.

13.1.3. Recommendation

Replace reusable expression variables used to drive constraints with named constraints. Defining the condition as a named constraint preserves reusability while allowing the configuration engine to apply propagation more efficiently. This improves runtime performance without sacrificing readability or structural clarity.

```
...

constraint isCableTopEntry (cableEntry == 'Top');
constraint(isCableTopEntry -> cableManagementkit[CableManagementKit] > 2);
constraint(isCableTopEntry -> hasSidePanels == false);

...
```

13.1.4.Quick Wins

Review existing boolean expression variables that primarily serve as constraint drivers and refactor them into named constraints, starting with performance-sensitive configuration areas.

13.1.5.Estimated Effort

The remediation effort is estimated as “Low”, as the change is localized and primarily structural. It involves replacing expression variables with named constraints and validating that configuration behavior remains unchanged across relevant scenarios.

14.Rack Vertex. Finding №7

14.1.Decimal number precision

Finding ID	Risk Score	Category	Affected Area	Estimated Effort
F-007	Medium 🟨	Performance	Numeric Attribute Definitions	8h - 16h

14.1.1. Description

Numeric attributes defined with the decimal type without explicitly specifying precision default to higher internal precision (typically four decimal places). While functionally correct, unnecessarily high precision increases the computational workload of the constraint engine, particularly in models containing multiple calculations or optimization steps.

In scenarios where such precision is not required for business logic or pricing accuracy, limiting decimal precision can improve model execution efficiency.

```
...

decimal depth = [800, 1000, 1200];
decimal width = [600, 800]; decimal
load = [800, 1500];

...
```

14.1.2. Impact

Higher than required decimal precision slightly increases computation time and resource usage, especially in large or calculation-heavy CML models. Although the impact is typically moderate to low for small models, the cumulative effect can become noticeable in complex configurations or high-volume transaction environments.

14.1.3. Recommendation

Explicitly define decimal precision using `decimal(n)` where appropriate, based on the actual precision required by the business use case (e.g., `decimal(2)` for pricing or measurement values that require only two decimal places).

```
...

decimal(2) depth = [800, 1000, 1200];
decimal(2) width = [600, 800];
decimal(2) load = [800, 1500];

...
```

14.1.4.Quick Wins

Review commonly used calculated numeric attributes (pricing factors, dimensional calculations, derived quantities) and standardize their precision where full default precision is not necessary.

14.1.5.Estimated Effort

The remediation effort is “*Low*”, as it primarily involves updating attribute declarations to include explicit precision definitions and performing lightweight regression testing to confirm that calculation accuracy remains aligned with business requirements.

15.Rack Vertex. Finding №8

15.1.Duplicate Product SKUs for Bundled and Standalone Contexts

Finding ID	Risk Score	Category	Affected Area	Estimated Effort
F-008	Medium 🟨	Product Catalog Architecture	Catalog Management / Bundle Configuration	16h - 32h

15.1.1. Description

The system currently utilizes separate product records for identical physical items depending on their sales context (standalone vs. bundled). For instance, there are two distinct PDU (Power Distribution Unit) products maintained in the catalog: one specifically configured to be sold within a bundle, and a duplicate SKU intended for standalone sales.

This modeling approach violates the "single source of truth" principle in master data management. There is no technical or business necessity for product duplication, as modern CPQ architectures are designed to utilize a single product record across multiple selling scenarios, adjusting behavior or pricing dynamically based on the context.

```
...

type PDU : LineItem;

...
```

15.1.2. Impact

Maintaining duplicate products significantly increases administrative overhead. Every update to product specifications, pricing, or translation requires double the effort and introduces a high risk of inconsistencies. Furthermore, it leads to SKU proliferation, which degrades catalog performance and heavily fragments sales reporting and forecasting (as revenue for the exact same physical product is split across multiple SKUs).

15.1.3. Recommendation

Consolidate duplicate products into a single, unified SKU that can be sold both as a standalone item and as an option within a bundle. Utilize context-aware pricing rules, bundle-specific price overrides, or configuration rules to manage these variations without cloning the product record.

```
...

type PDU32A : LineItem {
   ...
}

...
```

15.1.4.Quick Wins

Immediately halt the creation of new duplicate products for upcoming catalog releases. Identify a specific, easily manageable product category (e.g., the PDU accessories) to execute a pilot consolidation, which will help establish the standard operating procedure for the rest of the catalog.

15.1.5.Estimated Effort

The remediation effort is estimated as “Medium”. While the architectural concept is straightforward, the execution requires data mapping to identify all duplicates, updating bundle configurations to point to the consolidated SKUs, adjusting pricing/configuration rules to handle context (bundled vs. standalone), and safely retiring the redundant products without breaking in-flight quotes.