Rules Engine – Reference Contract (Unique Document)

Sémantique verrouillée • Tests verrouillés • Implémentation de référence (V4-based)

Version 1.5.4 (Contract Reference)

Date: 2025-12-18

0. Statut et portée

NORMATIF pour:
• la **sémantique observable** (ce qui doit se passer),
• la **suite de tests normative** (ce qui prouve la conformité).

INFORMATIF (référence) pour:
• l'**implémentation proposée** (basée sur le moteur V4 déjà travaillé).

Une implémentation est **conforme** si et seulement si:
1) elle respecte la sémantique normative de ce document, et
2) elle passe l'ensemble des tests normatifs définis ici.

Ce document ne fige pas les choix physiques SQL (tables vs mémoire, indexation, etc.).
Il fige uniquement les **invariants** et le **comportement observable**.

1. Objectifs

Le moteur de règles a pour objectifs:
• **Robustesse**: erreurs localisées, pas d'arrêt du thread, comportement explicite.
• **Déterminisme**: mêmes entrées ⇒ mêmes sorties, indépendamment des plans SQL.
• **Performance**: mode NORMAL minimal, instrumentation uniquement en DEBUG.
• **Évolutivité**: langage fermé, extensions versionnées.

> Principe: **Le moteur orchestre; SQL Server calcule.**

2. Concepts

2.1 Thread
Un **thread** est un contexte isolé d'évaluation contenant:
• Variables initialisées (potentiellement multi-lignes / multi-valeurs),
• Règles précompilées,
• État d'exécution des règles,
• Mode d'exécution (NORMAL ou DEBUG).

Invariants:
• pas d'état partagé entre threads,
• une règle est évaluée **au plus une fois** par thread (cache),
• les erreurs n'interrompent pas le thread.

2.2 Règle
Une **règle**:
• possède une clé unique (case-insensitive),
• retourne une **valeur scalaire** (ou NULL en cas d'erreur),

• peut référencer des variables et d'autres règles via tokens `{...}`.

2.3 Variable
Une **variable** est un ensemble de valeurs associées à une clé.
Une variable peut être:
• numérique,
• booléenne,
• texte,
• NULL,
• multi-valeurs (plusieurs lignes).

3. Modes d'exécution (Performance Contract)

```
ExecutionMode ∈ { NORMAL, DEBUG }
```

3.1 NORMAL (défaut)
Objectif: performance maximale.
• Aucune journalisation détaillée (durées, SQL compilé, etc.).
• Stockage minimal dans l'état: State, Value, ErrorCategory, ErrorCode.

3.2 DEBUG
Objectif: diagnostic.
• Journalisation activée (durées, message d'erreur détaillé, SQL compilé optionnel).
• DOIT être explicitement activé (jamais implicite).

4. Collation et unicité des clés (v1.5.2)

• Les clés sont uniques **case-insensitive** selon la collation de la colonne Key.
• Collation contractuelle recommandée:
`SQL_Latin1_General_CP1_CI_AS`
• Les tables temporaires héritent de tempdb ⇒ la collation DOIT être explicitée sur `Key`.

Exemples équivalents:
• `Toto = TOTO = toto`

Le moteur **ne** fait pas de LOWER/UPPER; la comparaison est déléguée à SQL Server (collation).

5. Ordre canonique (FIRST / CONCAT / JSONIFY)

Décision normative:
> L'ordre canonique est l'ordre d'insertion dans l'état du thread (SeqId), pas l'ordre de la clé.

Conséquences:
• FIRST / FIRST_POS / FIRST_NEG retournent la première valeur selon SeqId.
• CONCAT agrège selon SeqId.
• JSONIFY produit un objet JSON dont l'ordre de sérialisation suit SeqId (utile pour debug/tests; l'objet JSON reste sémantiquement non ordonné).

6. DSL des tokens `{...}` – Sémantique verrouillée

6.1 Principe
Un token `{...}` ne fait **qu'une chose**:

1) sélectionner un sous-ensemble de clés (variables ou règles),
2) résoudre les valeurs correspondantes (lazy pour les règles),
3) appliquer un agrégateur unique,
4) retourner un scalaire.

> Aucune logique SQL n'est évaluée dans `{...}`: pas de `IIF`, pas de `COALESCE`, pas de calcul.

6.2 Grammaire (normative)

```
Token ::= "{" Lookup "}" Lookup ::= [ Aggregator "(" ] Selector [ ")" ]
Selector ::= [ "rule:" ] IdentifierOrPattern IdentifierOrPattern ::=
Identifier Aggregator ::= FIRST | SUM | AVG | MIN | MAX | COUNT | FIRST_POS |
SUM_POS | AVG_POS | MIN_POS | MAX_POS | COUNT_POS | FIRST_NEG | SUM_NEG |
AVG_NEG | MIN_NEG | MAX_NEG | COUNT_NEG | CONCAT | JSONIFY
```

Agrégateur par défaut: **FIRST**.

6.3 Identifiants (rappel)
• espaces autorisés,
• quotes `'...'` ou `"..."` autorisées,
• échappement: `''` dans `'...'`, `""` dans `"..."`.
• caractères structurants interdits hors quotes: `{ } [ ] ( ) :`.

6.4 Sélection (SqlLike)
La sélection est réalisée via un comportement de type SQL LIKE sur `Key`.
• case-insensitive (via collation),
• peut retourner 0, 1 ou N clés.
• si 0 clé ⇒ ValueSet vide.

7. États d'exécution et récursivité (robustesse)

7.1 États (fermés)
```
NOT_EVALUATED EVALUATING EVALUATED ERROR
```

7.2 Résolution lazy d'une règle (normative)
• Si EVALUATED ⇒ retourner la valeur.
• Si ERROR ⇒ retourner NULL.
• Si NOT_EVALUATED ⇒ passer à EVALUATING, exécuter, puis EVALUATED ou ERROR.
• Si EVALUATING (ré-entrée) ⇒ récursivité détectée ⇒ passer la règle en ERROR, retourner NULL.

7.3 Propriété de continuation
Une règle en ERROR **ne bloque pas** le thread. Les règles non liées continuent à s'évaluer.

8. Gestion unifiée des erreurs (toutes les règles)

8.1 Contrat
Toute erreur lors de l'évaluation d'une règle (récursivité, division par zéro, type mismatch, SQL, etc.):
• marque la règle **ERROR**,
• associe `ErrorCategory` et `ErrorCode`,
• valeur scalaire = **NULL**,
• le thread continue.

8.2 Catégories (fermées)
```
RECURSION, NUMERIC, STRING, TYPE, SQL, SYNTAX, UNKNOWN
```

8.3 Codes minimaux recommandés
• RECURSION / RECURSIVE_DEPENDENCY
• NUMERIC / DIVIDE_BY_ZERO

• NUMERIC / OVERFLOW
• TYPE / TYPE_MISMATCH
• SYNTAX / INVALID_EXPRESSION
• SQL / SQL_ERROR
• UNKNOWN / UNEXPECTED

8.4 Interaction avec les agrégateurs
Une règle en ERROR contribue une valeur NULL.
Comportement SQL standard:
• SUM/AVG/MIN/MAX/COUNT/CONCAT: NULL ignoré
• FIRST: peut retourner NULL si première valeur (SeqId) est NULL/ERROR
• JSONIFY: clé présente avec valeur `null`

9. Compilation SQL (littéraux)

Le compilateur (hors tokens) normalise:
• `"texte"` → `'texte'` (avec échappement `'` → `''`)
• `2,5` → `2.5` (séparateur décimal)
• aucune transformation sémantique supplémentaire.

PARTIE II — TESTS NORMATIFS (VERROUILLÉS)

10. Structure d’un test (normative)

Chaque test DOIT produire:
• Category, Name
• InputExpression
• Expected
• Actual
• Pass (0/1)
• Details

11. Fixtures normatives (jeu minimal)

11.1 Variables (MONTANT)
Valeurs insérées DANS CET ORDRE (SeqId):
1) 100
2) 200
3) -50
4) 150
5) -25
6) NULL

11.2 Variables (LIBELLE)
Ordre (SeqId):
1) 'A'
2) 'B'
3) NULL
4) 'C'

11.3 Règles (exemples)
• BBB1 = 10
• BBB2 = -5
• BBB_NULL = NULL

• EXPENSIVE = incrémente un compteur lors de l’exécution puis retourne 1

12. Matrice de tests (exhaustive v1.5.4)

12.1 Parsing / tokens
• T01: aucun token ⇒ aucun remplacement
• T02: extraction multi tokens
• T03: identifiant avec espaces `{MONTANT HT}`
• T04: identifiant quoté `'...'` et échappement
• T05: identifiant quoté `"..."` et échappement
• T06: `rule:` sélection de règles
• T07: SqlLike case-insensitive

12.2 Collation / unicité
• C01: unicité CI: insertion `Toto` puis `toto` ⇒ violation attendue
• C02: résolution `{TOTO}` = `{toto}`
• C03: temp table collation explicitée (garde-fou)

12.3 Ordre canonique
• O01: FIRST(MONTANT)=100
• O02: FIRST_NEG(MONTANT)=-50 (premier négatif par SeqId)
• O03: CONCAT(LIBELLE)='A,B,C' (NULL ignoré, ordre SeqId)
• O04: JSONIFY(rule:BBB%) sérialise selon SeqId (tolérance ordre JSON)

12.4 Agrégateurs numériques
• A01: SUM(MONTANT)=375
• A02: SUM_POS(MONTANT)=450
• A03: SUM_NEG(MONTANT)=-75
• A04: AVG(MONTANT)=75
• A05: AVG_NEG(MONTANT)=-37.5
• A06: MIN(MONTANT)=-50
• A07: MAX(MONTANT)=200
• A08: COUNT(MONTANT)=5 (NULL ignoré)
• A09: COUNT_POS(MONTANT)=3
• A10: COUNT_NEG(MONTANT)=2
• A11: ensembles vides: SUM/AVG/MIN/MAX/FIRST ⇒ NULL; COUNT ⇒ 0; JSONIFY ⇒ {}

12.5 Lazy & cache
• L01: EXPENSIVE référencée deux fois ⇒ compteur=1
• L02: règle déjà évaluée ⇒ pas de réexécution
• L03: isolation thread: compteur réinitialisé sur nouveau thread

12.6 Erreurs (global)
• E01: division par zéro (NUMERIC/DIVIDE_BY_ZERO) ⇒ NULL, thread continue
• E02: overflow ⇒ ERROR (NUMERIC/OVERFLOW)
• E03: type mismatch ⇒ ERROR (TYPE/TYPE_MISMATCH)
• E04: SQL invalide ⇒ ERROR (SYNTAX/INVALID_EXPRESSION)
• E05: récursivité directe ⇒ ERROR (RECURSION/RECURSIVE_DEPENDENCY)
• E06: récursivité indirecte A→B→A ⇒ A,B ERROR, thread continue
• E07: agrégation tolérante: SUM(rule:...) ignore NULL issus d’erreurs

12.7 Performance / modes
• P01: NORMAL: aucune écriture debug
• P02: DEBUG: écritures debug présentes (durées etc.)
• P03: NORMAL plus rapide que DEBUG à charge identique (test de ratio/ordre)

PARTIE III — IMPLÉMENTATION DE RÉFÉRENCE (V4-BASED, INFORMATIF)

13. Objectif de l’implémentation de référence

Fournir une base fiable (moteur V4) qui:
• respecte la sémantique v1.5.4,
• passe les tests,
• sert de point de départ à des optimisations (sans casser les invariants).

14. Principes V4 conservés

• résolution par procédures stockées,
• substitution token→valeur,
• exécution SQL via `sp_executesql`,
• cache par thread.

15. Adaptations v1.5.4 à appliquer sur V4

1) **Ordre canonique**: introduire SeqId (IDENTITY) et l’utiliser pour FIRST/CONCAT/JSONIFY.
2) **États**: ajouter ERROR et la logique EVALUATING (récursivité).
3) **Erreurs**: envelopper toute exécution SQL de règle en TRY/CATCH ⇒ ERROR + NULL.
4) **Modes**: NORMAL sans debug; DEBUG avec table dédiée (#ThreadDebug) ou colonnes.
5) **Collation**: expliciter collation Key sur #ThreadState pour alignement tempdb.

16. Liberté d’optimisation (non normative)

Autorisé:
• changer le schéma physique,
• indexer différemment,
• remplacer SQL_VARIANT par colonnes typées,
• optimiser la détection/compilation des tokens,
• paralléliser si résultat strictement identique.

Interdit:
• modifier l’ordre (SeqId),
• introduire calcul dans `{}`,
• masquer une erreur (pas de substitution silencieuse autre que NULL),
• réévaluer une règle plusieurs fois dans le même thread.

17. Annexes: tables de référence (codes)

Les valeurs exactes des codes peuvent évoluer, mais les catégories sont fermées.
Toute implémentation DOIT au minimum stocker ErrorCategory + ErrorCode.