⭐ FINAL RESILIENT MULTIPLICATION RULES (Your Java Engine Model)

These rules guarantee an identical result to your program.

🔵 RULE 0 — Matrix dimension definition

Before multiplication:

- **Result rows = rows of Matrix A**
- **Result columns = maximum row width in Matrix B**

Your engine **never reduces** result dimensions due to ragged shapes.

🔵 RULE 1 — How A×B is evaluated (global flow)

For each **row i** in A, and each **column j** in B:

You attempt to compute:

sum over k: A[i][k] * B[k][j]

But with *resilient* logic based on ragged jagged structure.

🔵 RULE 2 — What counts as a valid index

A multiplication at index k is valid **only if both conditions hold**:

1. A[i] has an element at index k
2. B has a row at index k AND B[k] has column j

If either is missing:

- The engine **does NOT stop the entire dot product**
- Instead, it **skips that multiply**
- And continues trying the next k

This is a key difference from strict mathematics.

🔵 RULE 3 — Scanning ALL possible k in B (critical behaviour)

For each column j of B:

Your engine loops through:

k = 0 → B.length-1

This means:

- Even if A has only 2 columns,
- And B has 8 rows,

You still check **all 8 B rows**, trying to match indices.

This is exactly what your output logs demonstrate.

This is **THE biggest behavioural difference** from standard multiplication.

🔵 RULE 4 — Missing values are simply skipped

If **A[i] has no element at index k**, you print:

“Matrix(A) row [...] has no content at index[k] — skipping”

If **B[k] row is too short**, you print:

“Row length invalid — skipping”

You do NOT:

- break
- stop the row entirely
- reduce result size
- pad with null

You simply skip and move on.

🔵 RULE 5 — Valid products accumulate into sum

Whenever both sides have elements:

A[i][k] * B[k][j]

You:

- compute the product
- add it to the running total for result[i][j]
- print the detailed multiplication step

- continue scanning further rows k

Even if earlier rows were invalid or mismatched.

This is why values like:

(66 × 6) + (14 × 1)

appear even though earlier rows of B were “invalid”.

🔵 **RULE 6 — Zero values are valid, null never appears**

Your engine treats:

- 0 as a real numerical value → contributes 0
- **Empty rows ({ }) or missing indices** → skip entirely
- **Null never appears** unless explicitly put by user

Your result **never contains null** unless your test literal includes null.

🔵 **RULE 7 — Excess elements in A row or B row**

Too many elements in A row:

Only process up to min(A[i].length, B.length).

Extra A elements are **ignored silently**.

Too many elements in B rows:

Only process up to their width.

Extra B columns are ignored if j exceeds width of that row.

🔵 **RULE 8 — Result cell is always integer**

Even if **no valid multiplications occur**, cell defaults to:

0

🔵 **RULE 9 — Entire chain multiplication**

When there are N matrices:

M0 × M1 → M2

M2 × M3 → M4

M4 × M5 → …

Each multiplication produces a new matrix **with the same resilient rules**, then feeds into the next.

Your engine:

- prints every intermediate
- checks structural consistency (but does not terminate)
- prints warnings but continues

🔵 **RULE 10 — Your engine never throws structural exception**

Instead of erroring out (like strict math),

- It warns
- Prints mismatch logs
- Performs all valid multiply attempts
- Continues the chain

It is a **fault-tolerant jagged dot-product pipeline**.

EXECUTIVE SUMMARY

This document explains the logic, methodology, and rule systems used to generate and evaluate **100 AI-designed jagged matrix multiplication test cases**.
These rules govern how each test case is interpreted and processed in two modes:

1. **Resilient Multiplication** — models the tolerant, "continue calculation if possible" behavior of your Java algorithm.
2. **Strict Multiplication** — models classical matrix multiplication with rigid structural requirements.

This dual-evaluation approach provides both:

- a diagnostic tool to understand how your code behaves under jagged, irregular, and malformed inputs, and
- a mathematical baseline for comparison.

The test cases (40 advanced + 60 original-style) are delivered separately in:

- advanced_40_literals.txt
- original_60_literals.txt

Each literal is provided in **one-line Java notation**, e.g.:

Integer[][][] test = new Integer[][][] { {{1,2},{3}}, {{4},{5,6}} };

This ensures easy copying into your Java environment, minimal formatting overhead, and clear reproducibility.

RESILIENT MULTIPLICATION RULES

(Your Algorithm's Philosophy)

These rules are designed to mimic the exact "resilience-first" behavior of your Java code, which continues processing *as far as possible* even under malformed or incomplete matrix shapes.

1. **Continue multiplication as long as usable values exist on both sides.**
   No early termination unless absolutely required.
2. **Null × number = null → contributes nothing**
   Null is treated as "no usable value".
3. **Number × null = null → contributes nothing**
4. **0 × any value = 0**

5. **Surplus row elements (Matrix A):**
    - If a row has *more elements* than the corresponding column has,
      → **ignore** the extra elements.
6. **Surplus column elements (Matrix B):**
    - If a column has more elements than A's row length,
      → **ignore** the extra column values.
7. **Insufficient elements in B's column:**
    - If B's column *runs out of usable values* before A's row is fully processed,
      → **stop multiplication for that cell immediately**.
8. **Insufficient elements in A's row:**
    - If A's row runs out early,
      → **stop multiplication for that cell**.
9. **Blank {} rows/columns:**
    - Treated as having *zero usable values*.
    - Any attempt to multiply into or from them halts that cell's calculation.
10. **Negative values behave normally**
    - Standard multiplication rules apply.
11. **No padding; no shape enforcement; no normalization.**
    - The structure is used *exactly as provided*.
12. **Goal:**
    - Maximize usable computation
    - Avoid exceptions or early failure
    - Provide a mathematically consistent but structurally tolerant process

These rules capture the full spirit of your Java algorithm and have been mirrored precisely.

STRICT MULTIPLICATION RULES

(Classical Linear Algebra Requirements)

This mode models rigid mathematical matrix multiplication.

1. **Every row in Matrix A must have equal length.**
   No jagged rows.
2. **Every column in Matrix B must have equal length.**
3. **A.columns == B.rows must be true**, otherwise impossible.
4. **Null values invalidate the structure.**
   Strict mode forbids incomplete or missing entries.
5. **Blank {} rows or columns invalidate the matrix.**
6. **Any structural violation → the entire multiplication is invalid.**
   Strict mode never attempts partial multiplication.
7. **Process is deterministic and mathematically pure.**
   No resilience; no flexibility.

Strict mode serves as the **baseline truth** for whether a matrix configuration is mathematically well-formed.

✅ STRICT MODE — Conventional Linear Algebra Rules

Strict Mode follows the **classical mathematical definition** of matrix multiplication. These are the **same rules used in**:

- Linear algebra textbooks
- NumPy, MATLAB, R
- PyTorch / TensorFlow
- all standard numerical or scientific systems

Under strict mode, **a matrix must be a proper rectangle**, and **multiplication is allowed only when dimensions align perfectly**.

✓ Rule 1 — Both matrices must be rectangular (uniform row lengths)

A valid matrix must have:

- Every row containing the **same number of columns**

- No jagged rows
- No empty {} rows inside (allowed only if the *entire* matrix has zero rows)

Examples:

Valid strict matrix:

[1, 2, 3]

[4, 5, 6]

Invalid strict matrix:

[1, 2]

[3]

[4, 5, 6]

Invalid because row lengths differ → **Matrix is not rectangular**.

✓ Rule 2 — The number of columns of A must equal the number of rows of B

For A × B to be defined:

A is m × k

B must be k × n

If A's column-count ≠ B's row-count → multiplication is **not allowed**.

Example:

A is 2×3

B is 3×4 → OK

A is 2×3

B is 2×4 → INVALID

✓ Rule 3 — No null values allowed inside strict matrices

Strict linear algebra **does not permit null**.

If any entry is null, the matrix is invalid.

Zero (0) is valid.
Null (null) is not.

✓ Rule 4 — If any rule is violated, strict output is INVALID

Strict mode never attempts partial multiplication.
It does **not** try to continue or approximate.
It does **not** salvage partially valid rows.

If the structure is invalid:

❌ **Output = INVALID (matrix dimensions or structure incorrect)**

✓ Rule 5 — Strict mode always produces a full rectangular output

If multiplication is valid, the result is an **m × n rectangular matrix**, and every cell is computed:

result[i][j] = Σ (A[i][k] * B[k][j])

No early stopping
No skipping
No tolerance for missing data
No resilience behavior

🧩 How Strict Mode Corresponds to Real-World Systems

This strict behavior matches:

Mathematics

- Pure linear algebra in university courses
- All textbooks on matrix theory

Programming Languages

- NumPy (numpy.matmul)
- MATLAB (A * B)
- Octave
- R
- Julia

- SciPy
- C++ Eigen
- Java EJML / Apache Commons Math

Machine Learning

- Matrix operations in neural networks (dense layers, transformers, embeddings)
- Backpropagation math
- GPU kernels (CUDA, ROCm)

Strict mode reflects how hardware-accelerated matrix multiplication works physically on CPUs and GPUs.

PURPOSE OF BOTH MODES

Using both rule sets allows you to:

- Compare your algorithm's resilient "best effort" output vs strict expected mathematical output
- Identify where your Java logic compensates for structural irregularities
- Understand behavior under real-world corrupted/jagged data
- Validate correctness, resilience, and error coverage
- Create a unified dataset for code evolution testing