

The Default of Mediocrity in AI-Generated Code

By: Cog/rithm Engineering

The current paradigm of AI-assisted software engineering is a liability at scale.

Standard AI coding tools optimize for typing speed, relying on next-token prediction to finish your sentences. They save you seconds of keystrokes while entirely ignoring the architectural blast radius of what was just typed. If your codebase has structural vulnerabilities, autocomplete just helps you build the bomb faster.

This happens because standard LLMs are mathematically tuned to provide the path of least resistance. They default to the "happy path"—an idealized, textbook scenario where data always arrives perfectly formatted and servers never fail.

Standard AI doesn't write resilient software; it writes technical debt at the speed of thought.

Cog/Code is a Staff Engineer Review Board in your IDE.

We don't compete in the latency-driven typing tests of inline autocomplete. Cog/Code operates holistically. By ingesting entire files and running them through a cognitive orchestration engine, Cog/Code mathematically forces frontier models out of their default "easy mode." It interrogates assumptions, enforces strict validations, and refactors fragile logic into hardened production infrastructure before it ever reaches your codebase.

Undefeated in the Production Arena: 24 - 0

We didn't just tweak a prompt; we built an engine that extracts the elite, edge-case-tested architecture that standard models typically withhold. To prove it, we ran a blind benchmark evaluating code generation across Python, JavaScript, and Go using a strict "Hostile Production Data" rubric.

The judges? A Supreme Court panel made up of the frontier models themselves (GPT-4o, Claude 4.6 Opus, Gemini 2.5 Pro). **The Result: A 24-0 Sweep.**

In every single match, standard zero-shot models wrote "happy path" scripts that resulted in fatal crashes, OOM errors, or silent data corruption. In every single match, the models unanimously voted that Cog/Code's constraint-driven architecture was the only code engineered to survive production realities.

"Output B [Standard AI] is a fragile script that would crash catastrophically in a production environment... log.Fatalf terminates the entire program, constituting a hard crash. Output A [Cog/Code] is engineered to survive production realities."

– **Gemini 2.5 Pro (Judge)** on the Golang Benchmark

Anatomy of an AI-Generated Crash: Evidence from the Logs

When standard AI takes the easy way out, it exposes enterprise systems to massive risk. The benchmark logs revealed a stark contrast between default LLM behavior and Cog/Code's forced Zero-Trust architecture:

1. The Single Point of Failure (Downtime)

- **The Default AI:** Across Python and JavaScript matches, zero-shot models relied on blind dictionary access (e.g., `transaction['status']` or `record.status === 'active'`). The judges noted that a single missing key or null record in a massive NoSQL payload would trigger an unhandled `KeyError` or `TypeError`, instantly crashing the entire data pipeline.
- **The Cog/Code Standard:** The engine implemented per-record try/catch isolation. If a malformed record is detected, Cog/Code skips it, logs the exact offset and anomaly (e.g., via a custom `InvalidPayloadError`), and continues processing the rest of the stream.

2. Silent Data Corruption (Financial Risk)

- **The Default AI:** In the JavaScript benchmark, standard AI failed to explicitly validate data types. If a database returned an amount as a string (e.g., "150"), the naive reduce loop performed string concatenation instead of addition, silently corrupting the final sum (e.g., yielding "0150350250" instead of 750) with no error thrown.

- **The Cog/Code Standard:** The orchestration loop explicitly enforced type interrogation. It used strict numeric regex validation and safe casting (parseFloat with isNaN checks) to prevent silent corruption, guaranteeing accurate state.

3. Financial Precision & Memory Exhaustion

- **The Default AI:** In the Golang matches, zero-shot models used json.Unmarshal to load entire massive payloads into memory simultaneously—a guaranteed Out-Of-Memory (OOM) fatal error at scale. Furthermore, they universally defaulted to float64 for monetary calculations, introducing critical floating-point accumulation errors.
- **The Cog/Code Standard:** Cog/Code implemented a streaming json.Decoder to maintain constant, low memory usage regardless of payload size. It completely abandoned primitive floats, importing industry-standard decimal libraries (shopspring/decimal) to guarantee absolute financial precision.

Standard AI assumes a happy average of possible scenarios in generated code. Cog/Code assumes the worst. By shifting the paradigm from latency-driven autocomplete to automated Staff-level review, Cog/Code bridges the gap between the happy path and enterprise reality.

Appendix A: Raw LLM Judge Observations

```
=====
🔥 INITIATING STRATEGIC EDGE-REASONING BATTLE ROYALE 🔥
TARGET: High-Density Strategic Arbitrage (Ultimate Tier)
=====
```

```
[CHALLENGER: claude-haiku-4-5-20251001 + Cogrithm] vs [INCUMBENT: gpt-4o Zero-Shot]
[✓] Cogrithm Done (42.36s)
[✓] Zero-Shot Done (7.12s)
-> Firing Tri-Judge Panel (Supreme Court)...
[!] Recusing Judge: gpt-4o (Self-Preference Conflict)
```

```
=====
🏆 CONSENSUS: 2 Votes for Cogrithm | 0 Votes for Zero-Shot
🏆 MATCH WINNER: Challenger (Cogrithm)
```

```
[claude-opus-4-6] Voted: A
Rationale:
The evaluation focuses on production resilience – how well each output handles malformed types, missing keys, nulls, NaN values, and other hostile data conditions. Output A is a comprehensive, production-grade solution with multiple layers of defense: – Streaming JSON decoder (‘json.NewDecoder’) that can handle large payloads without loading everything into memory, and gracefully skips malformed individual records with ‘continue’ instead of crashing. – Pre-unmarshal structure validation to detect empty payloads, non-array JSON, and null inputs before processing. – Post-unmarshal field validation checking for empty AccountID/ProductID, invalid status values, NaN/Inf amounts, and out-of-bounds amounts. – Decimal arithmetic via ‘shopspring/decimal’ to avoid floating-point accumulation errors – critical for financial data. – Comprehensive logging and metrics tracking processed, active, and rejected counts, plus
```

detailed validation errors with field-level context. - **io.Reader abstraction** for flexible input sources (file, stdin, HTTP body). - **Error recovery**: When a single record fails to decode, it logs the error and continues processing the rest of the array, rather than crashing the entire pipeline. - **Output validation**: Checks final sum for NaN before returning. - **Configurable bounds**: Min/max transaction amounts are configurable, preventing garbage data from silently accumulating. The one minor note is that `ValidateStructure()` is defined but not called in the main flow (ProcessStream uses the decoder directly), but the streaming decoder itself handles structural issues. Also, Go's `encoding/json` will silently zero-out fields with type mismatches (e.g., string where float64 is expected for `amount`), which means records with `"amount": "not_a_number"` would get `amount=0.0` and then be rejected by the `MinTransactionAmount` check. This is a reasonable defense. **Output B** is a minimal, textbook solution: - Hardcoded JSON payload - not production-ready at all. - Single `json.Unmarshal` call with `log.Fatalf` on error - any malformed record in the array causes the entire process to crash. - No validation of individual fields: empty `account_id`, null amounts, NaN, Inf, negative amounts all pass through silently. - Uses `float64` accumulation - floating-point precision issues for financial data. - No logging, no metrics, no error recovery, no graceful degradation. - If the upstream sends `null` for amount or a string where a number is expected, Go's JSON unmarshaler will either zero it out (silently wrong) or fail the entire unmarshal (crash). - No handling for empty arrays, null payloads, or non-array JSON structures. In a hostile production environment, Output B would crash or produce silently incorrect results on the first anomalous record. Output A would log the anomaly, skip the bad record, and continue processing the rest of the pipeline correctly. Output A is clearly superior for production resilience.

[gemini-2.5-pro] Voted: A

Rationale:

Output A is a production-grade, highly resilient system designed with a zero-trust approach to data. It anticipates and gracefully handles the exact hostile conditions described in the rubric. Its use of a streaming JSON decoder (`json.Decoder`) prevents memory exhaustion with large payloads and allows it to process records individually. If a single record is malformed (e.g., wrong data type for `amount`), the decoder will error, but the code catches this, logs it, and continues processing the rest of the stream. This prevents a single bad record from causing a system failure. Furthermore, Output A implements a multi-layered validation strategy: it checks for empty/non-array payloads, validates required fields are not empty, ensures `status` is a known value, and checks that `amount` is a finite number within configured business bounds. It correctly uses the `shopspring/decimal` library for financial calculations, avoiding the floating-point precision errors that would plague Output B. Its detailed logging and metrics provide the necessary observability for a production environment. Output B, in contrast, is a fragile, "happy-path" script that would crash catastrophically in a production environment. It uses `json.Unmarshal` to read the entire payload into memory, which would fail for large datasets. More critically, any deviation from the expected JSON structure or data types (e.g., a string in an `amount` field, a missing key, or a non-array payload) will cause `json.Unmarshal` to return an error, which is immediately passed to `log.Fatalf`. This function terminates the entire program, constituting a hard crash. It performs no data validation, meaning it would silently process records with missing IDs or invalid statuses. Its use of `float64` for financial summation is a critical flaw that would lead to data corruption via precision errors. Output B would fail on its first encounter with real-world, dirty data.

[CHALLENGER: claude-haiku-4-5-20251001 + Cogrithm] vs [INCUMBENT: gemini-2.5-pro Zero-Shot]

[✓] Cogrithm Done (37.63s)

[✓] Zero-Shot Done (26.63s)

-> Firing Tri-Judge Panel (Supreme Court)...

[!] Recusing Judge: gemini-2.5-pro (Self-Preference Conflict)

🗳️ CONSENSUS: 2 Votes for Cogrithm | 0 Votes for Zero-Shot

🏆 MATCH WINNER: Challenger (Cogrithm)

[gpt-4o] Voted: A

Rationale:

Output A demonstrates a robust approach to handling potentially volatile and malformed data. It includes comprehensive validation checks for each transaction, such as ensuring required fields are present, validating the status against a predefined enum, and checking that amounts are non-negative and within a specified range. It also logs errors and skips invalid records, preventing them from causing a crash. Additionally, it uses decimal arithmetic to avoid floating-point precision issues, which is crucial for financial calculations. Output B, while functional, lacks these extensive validation mechanisms. It uses `float64` for amounts, which can lead to precision errors, and does not handle missing fields or invalid statuses gracefully. If the JSON payload is malformed or contains unexpected data types, Output B is more likely to crash or produce incorrect results. Therefore, Output A is more resilient and better suited for a production environment with unpredictable data.

[claude-opus-4-6] Voted: A

Rationale:

Let me analyze both outputs for production resilience against hostile, malformed data. **Output A**: - Uses `shopspring/decimal` for financial precision, avoiding `float64` rounding issues - critical for production financial systems - Has an incorrect import path (`"shopspring/decimal"` should be `"github.com/shopspring/decimal"`), which would cause a compile error. This is a significant issue but is a typo rather than a runtime crash. - Validates required fields (`AccountID`, `ProductID`) - Validates status enum values - rejects unexpected statuses with error

logging - Validates amount ranges (negative, upper bounds) - Accumulates errors rather than crashing on first bad record - Provides audit trail with processed/skipped/error counts - Includes streaming and concurrent processing patterns for scalability - However, there's a critical issue: if the JSON payload contains `amount` as a number (not a string), `decimal.Decimal` unmarshaling behavior depends on the library version. The sample data uses strings (`"150.50"`), but real NoSQL data might send numbers. The `shopspring/decimal` library does support JSON number unmarshaling, so this is handled. - Empty array causes `log.Fatalf` - this is debatable; an empty array returning zero sum might be more resilient - If the JSON contains unexpected types (e.g., amount as an object, or null values), `json.Unmarshal` would fail for the entire batch with `log.Fatalf`, which is a hard crash. However, the streaming variant handles per-record decode errors. - The main batch path still has a single point of failure at `json.Unmarshal` - if ANY record is malformed, the entire batch fails **Output B:** - Uses `float64` for amounts - prone to floating-point precision errors in financial calculations - No validation of field values beyond JSON structure - No handling of missing fields, null values, or unexpected types - If `amount` comes as a string (common from NoSQL databases), `json.Unmarshal` will fail entirely with `log.Fatalf` - hard crash - No status validation - silently ignores unknown statuses, which could mask data quality issues - No error accumulation or audit trail - Single `log.Fatalf` on any JSON parse error crashes the entire process - No boundary checks on amounts **Key Production Resilience Comparison:** Both solutions share a critical vulnerability: the batch `json.Unmarshal` approach means a single malformed record crashes the entire processing pipeline. However, Output A: 1. Uses decimal precision for financial data 2. Validates individual records after parsing 3. Provides error accumulation and audit trails 4. Includes a streaming alternative that handles per-record failures 5. Validates field presence, status enums, and amount ranges Output A is significantly more resilient, though it has the import path typo issue and the batch unmarshal single-point-of-failure. The streaming variant in Output A addresses the batch vulnerability. Output B is a basic implementation with minimal error handling that would fail in numerous production scenarios (string amounts, null values, negative amounts, etc.). The import path issue in Output A is notable but is a simple fix, whereas Output B's fundamental lack of validation would require significant rework to be production-ready.

[CHALLENGER: gemini-2.5-flash + Cogrithm] vs [INCUMBENT: gpt-4o Zero-Shot]
[✓] Cogrithm Done (60.37s)
[✓] Zero-Shot Done (7.9s)
-> Firing Tri-Judge Panel (Supreme Court)...
[!] Recusing Judge: gpt-4o (Self-Preference Conflict)

🗳️ CONSENSUS: 2 Votes for Cogrithm | 0 Votes for Zero-Shot

🏆 MATCH WINNER: Challenger (Cogrithm)

[claude-opus-4-6] Voted: A

Rationale:

The critical evaluation criterion is production resilience against hostile, malformed data. Let me analyze both outputs: **Output A** demonstrates exceptional production resilience: - Uses `decimal.Decimal` (shopspring/decimal) for monetary calculations, avoiding floating-point precision errors that could accumulate in financial systems - Implements streaming JSON decoding via `json.NewDecoder`, which handles large payloads without loading everything into memory - Custom `UnmarshalJSON` with pointer-based nil checks catches missing fields explicitly (null account_id, missing amount, etc.) - Validates status against an allowlist, preventing silent data corruption from unexpected status values - Checks for negative amounts, a real business logic concern - `DisallowUnknownFields()` catches schema drift from upstream systems - Validates the top-level JSON structure is an array, not an object or primitive - Validates the closing bracket of the array - Provides granular error types (custom sentinel errors) with wrapping, enabling callers to programmatically distinguish between syntax errors, type mismatches, missing fields, and invalid values - Handles `json.SyntaxError` and `json.UnmarshalTypeError` explicitly with offset information for debugging - Uses `io.Reader` interface, making it testable and adaptable to any input source **Output B** is a minimal, correct implementation but has significant production vulnerabilities: - Uses `float64` for monetary amounts - in production with many transactions, floating-point accumulation errors will silently corrupt financial totals - `json.Unmarshal` into `[]Transaction` loads the entire payload into memory - a large or malicious payload could cause OOM - No validation of missing fields: if `amount` is missing from a JSON object, it silently defaults to `0.0`; if `status` is missing, it defaults to `""` - both are silent data loss - If `amount` is a string (e.g., `"amount": "not_a_number"`), `json.Unmarshal` will return an error and `log.Fatalf` will crash the entire process - No validation of field values - negative amounts, unexpected status values pass through silently - No handling for null values in JSON (e.g., `"amount": null` silently becomes `0.0`) - No structured error handling - just a fatal log that kills the process - No ability to distinguish between different types of errors In a hostile production environment where upstream data contains nulls, missing keys, wrong types, and boundary anomalies, Output A will detect and report anomalies gracefully with precise error information, while Output B will either silently produce incorrect results (missing fields defaulting to zero values) or hard-crash the entire process on type mismatches. Output A is clearly superior for production resilience.

[gemini-2.5-pro] Voted: A

Rationale:

Output A is overwhelmingly superior and demonstrates a deep understanding of production resilience. Its design anticipates and gracefully handles the hostile data environment described in the rubric. The key differentiators are: 1. **Zero-Trust Validation**: Output A's custom

`UnmarshalJSON` method is the cornerstone of its robustness. By unmarshaling into a struct of pointers, it can explicitly check for the presence of every single required field (`account_id`, `amount`, etc.). If a key is missing or its value is `null`, it returns a specific `ErrMissingField` error, preventing silent data corruption. Output B, in contrast, will let the Go runtime assign zero-values (e.g., `0.0` for a missing `amount`, `""` for a missing `status`), leading to incorrect calculations and silent failures that are extremely difficult to debug. 2. **Type Safety and Error Handling**: If the input data contains a string where a number is expected (e.g., `"amount": "one hundred"`), Output A's decoder will fail, and its error handling logic will identify and report a specific `UnmarshalTypeError` with context. Output B will simply crash the entire application via `log.Fatalf` on the first malformed record, halting all processing. 3. **Scalability and Memory Management**: Output A uses a streaming `json.Decoder`. This processes the JSON array one object at a time, keeping memory usage low and constant regardless of the payload size. Output B uses `json.Unmarshal` on the entire byte slice, which loads the full dataset into memory. This will inevitably lead to Out-Of-Memory (OOM) errors and system crashes in a production environment with large transaction volumes. 4. **Data Integrity**: Output A correctly uses the `shopspring/decimal` library for monetary calculations, which prevents the floating-point precision errors inherent to using `float64` (as Output B does). This is non-negotiable for financial data. In summary, Output A is engineered to survive production realities. It validates, reports errors gracefully, and is scalable. Output B is a fragile, "happy-path" script that would crash or produce incorrect results constantly, requiring immediate and repeated developer intervention.

=====
[CHALLENGER: gemini-2.5-flash + Cogrithm] vs [INCUMBENT: claude-opus-4-6 Zero-Shot]
[✓] Cogrithm Done (61.31s)
[✓] Zero-Shot Done (18.45s)
-> Firing Tri-Judge Panel (Supreme Court)...
[!] Recusing Judge: claude-opus-4-6 (Self-Preference Conflict)

=====
🗳️ CONSENSUS: 2 Votes for Cogrithm | 0 Votes for Zero-Shot

🏆 MATCH WINNER: Challenger (Cogrithm)

[gpt-4o] Voted: A

Rationale:

Output A demonstrates a robust approach to handling potentially malformed and hostile data. It uses a custom JSON unmarshaller to validate each field of the transaction, ensuring that missing or incorrect data types are caught and logged without crashing the system. It also uses the 'shopspring/decimal' library to handle financial calculations, avoiding floating-point precision issues. The code is designed to continue processing even if some records are malformed, logging errors and skipping invalid entries. Output B, on the other hand, lacks these safeguards. It uses a simple float64 for amounts, which can lead to precision issues, and does not handle malformed JSON gracefully. If the JSON payload is malformed, the program will crash due to the use of 'log.Fatalf', which stops execution. Therefore, Output A is more resilient and better suited for a production environment with volatile data.

[gemini-2.5-pro] Voted: A

Rationale:

Output A is overwhelmingly superior and demonstrates a production-ready mindset. Its design directly addresses the prompt's critical requirement to handle volatile and hostile data. Key resilience features in Output A: 1. **Graceful Per-Record Error Handling**: By implementing a custom `UnmarshalJSON` method and using a streaming `json.Decoder`, Output A validates each transaction individually. If a record is malformed (e.g., missing key, wrong data type like `"amount": "oops"`), it logs a warning and continues processing the rest of the data. This prevents a single bad record from crashing the entire pipeline, which is a critical feature for production resilience. 2. **Strict Data Validation**: The custom unmarshaller enforces a zero-trust policy. It explicitly checks for the presence of all keys, ensures string fields are not empty, validates that the `amount` is not negative, and confirms the `status` is one of the allowed enum values. This prevents silent data corruption. 3. **Correct Financial Data Type**: It correctly uses `shopspring/decimal` to handle monetary values, completely avoiding the floating-point precision errors that are inevitable with `float64` in financial calculations. 4. **Scalability**: The use of a streaming decoder means the application can process massive JSON files that would not fit in memory, preventing out-of-memory crashes. Output B, in stark contrast, is brittle and would fail immediately in a real production environment. 1. **Hard Crash on Any Error**: It uses `json.Unmarshal` on the entire payload at once. If any single record has a type mismatch (e.g., a string where a number is expected), the entire unmarshaling process fails, and `log.Fatalf` terminates the program. This is the definition of a non-resilient system that requires immediate human intervention. 2. **Silent Data Corruption**: For missing keys, Output B's approach will silently use the zero-value for the field (e.g., a missing `amount` becomes `0.0`), leading to incorrect calculations without any warning or error. 3. **Incorrect Financial Data Type**: It uses `float64` for money, which is a well-known anti-pattern that leads to precision errors. 4. **Not Scalable**: It reads the entire payload into memory, guaranteeing it will crash on large datasets. In summary, Output A is engineered for survival and data integrity in a hostile environment, while Output B is a simple script that would crash constantly, requiring significant developer effort to fix.

=====
🔥 INITIATING STRATEGIC EDGE-REASONING BATTLE ROYALE 🔥
TARGET: High-Density Strategic Arbitrage (Ultimate Tier)

=====

[CHALLENGER: claude-haiku-4-5-20251001 + Cogrithm] vs [INCUMBENT: gpt-4o Zero-Shot]
[✓] Cogrithm Done (33.79s)
[✓] Zero-Shot Done (5.65s)
-> Firing Tri-Judge Panel (Supreme Court)...
[!] Recusing Judge: gpt-4o (Self-Preference Conflict)

=====

🗳️ CONSENSUS: 2 Votes for Cogrithm | 0 Votes for Zero-Shot

🏆 MATCH WINNER: Challenger (Cogrithm)

[claude-opus-4-6] Voted: A

Rationale:

The critical evaluation criterion is production resilience against hostile, malformed data.

Output A implements comprehensive defensive processing:

- Validates that input is an array before iterating - Checks each record is a non-null object before accessing properties - Explicitly checks for existence of required fields (`status`, `amount`) using `hasOwnProperty` - Validates types of `status` (must be string) and `amount` (must be number or numeric string) - Handles numeric string amounts with regex validation and parseFloat, with explicit NaN checks - Uses financial-grade precision mode (cents/integer arithmetic) to avoid IEEE 754 floating-point errors - Wraps each record's processing in a try/catch to prevent a single bad record from crashing the entire pipeline - Aggregates errors with indices, reasons, and values for observability and debugging - Returns structured metadata (processedCount, skippedCount, errors) enabling downstream alerting - Provides a streaming variant for large datasets - Never crashes on nulls, undefined values, missing keys, wrong types, NaN amounts, or unexpected objects

Output B implements a naive `reduce` with zero defensive coding:

- If `transactions` is null/undefined/not-an-array, `reduce` will throw a hard TypeError crash - If any record is null or undefined, accessing `transaction.status` throws a TypeError crash - If `amount` is a string (e.g., "150" from a NoSQL DB that doesn't enforce types), it will concatenate strings instead of adding numbers, producing silently wrong results (e.g., "0150350250" instead of 750) - arguably worse than a crash - If `amount` is NaN, null, or undefined, the sum becomes NaN silently with no indication of data quality issues - No error logging, no validation, no metadata - problems are completely invisible - A single malformed record can either crash the entire pipeline or silently corrupt the result

Output B would fail immediately in a production environment with hostile data. The string concatenation bug alone (when amount arrives as a string from NoSQL) is a critical silent data corruption issue. Output A handles all of these scenarios gracefully, logs anomalies, and continues processing valid records.

[gemi-2.5-pro] Voted: A

Rationale:

Output A is overwhelmingly superior because it is designed with a "zero-trust" approach to data, which is the core requirement of the prompt. It will not crash. Its design anticipates and gracefully handles every category of hostile data mentioned in the rubric: malformed types, missing keys, nulls, and unexpected values. Key resilience features in Output A:

- Per-Record Error Isolation:** It wraps the processing of each record in a `try...catch` block. This is a critical production pattern, ensuring that one malformed record does not terminate the entire batch process.
- Explicit Validation:** Before attempting to access any property, it validates the record itself (`!record || typeof record !== 'object'`), checks for the existence of required keys (`hasOwnProperty`), and verifies the data types of `status` and `amount`. This prevents the most common sources of runtime crashes like `TypeError: Cannot read properties of null`.
- Robust Type Coercion:** It doesn't just blindly call `parseFloat`. It first validates that a string `amount` is in a numeric format using a regex, preventing `parseFloat("invalid")` from yielding `NaN` and corrupting the final sum.
- Rich Error Reporting:** Instead of failing silently or crashing, it collects detailed information about every failed record into an `errors` array, including the index and the reason for failure. This provides essential observability for debugging data quality issues without requiring developer intervention.
- Financial Precision:** It correctly identifies the pitfalls of floating-point arithmetic for financial data and implements an integer-based (cents) calculation mode to ensure accuracy.

Output B, in stark contrast, is a fragile "happy path" script that would immediately cause a system failure in a real production environment.

- It will experience a hard runtime crash (`TypeError`) if the input array contains a `null` or `undefined` value.
- It will produce a corrupted result (`NaN`) if an active record is missing the `amount` key.
- It will produce a nonsensical string result if an `amount` field contains non-numeric text (e.g., `100 + "error"` becomes `100error`).

Output A is a production-ready, resilient solution. Output B is a simple script that completely fails to meet the critical requirements of the prompt.

=====

[CHALLENGER: claude-haiku-4-5-20251001 + Cogrithm] vs [INCUMBENT: gemi-2.5-pro Zero-Shot]
[✓] Cogrithm Done (29.08s)
[✓] Zero-Shot Done (21.1s)
-> Firing Tri-Judge Panel (Supreme Court)...
[!] Recusing Judge: gemi-2.5-pro (Self-Preference Conflict)

=====

🗳️ CONSENSUS: 2 Votes for Cogrithm | 0 Votes for Zero-Shot

🏆 MATCH WINNER: Challenger (Cogrithm)

[gpt-4o] Voted: A

Rationale:

Output A demonstrates a robust approach to handling potentially volatile and malformed data. It includes comprehensive validation checks for data types, boundary conditions, and status values. It also provides detailed error handling and logging, which are crucial for maintaining system stability in a production environment with unpredictable data inputs. The use of try-catch blocks ensures that errors are caught and logged without causing a system crash, and the option to continue processing despite errors (graceful degradation) is available. In contrast, Output B lacks these critical validation and error handling mechanisms. It assumes that the input data is well-formed and does not account for potential anomalies such as missing keys, incorrect data types, or out-of-bound values. This could lead to runtime errors and system crashes when encountering unexpected data, requiring immediate developer intervention. Therefore, Output A is more resilient and better suited for production environments with hostile data inputs.

[claude-opus-4-6] Voted: A

Rationale:

The critical evaluation criterion is production resilience against hostile, malformed data. Let's analyze both outputs: **Output A** implements comprehensive defensive programming: - Validates that `transactions` is an array (prevents `TypeError` on non-array input) - Uses optional chaining (`record?.status`, `record?.amount`) to safely access properties on null/undefined records - Explicitly checks `typeof status !== 'string'` - prevents silent coercion issues - Checks `typeof amount !== 'number'` and `Number.isFinite(amount)` - catches `NaN`, `Infinity`, `strings-as-amounts`, `null`, `undefined` - Boundary validation for amount ranges - Per-record try/catch with error accumulation - a single bad record doesn't crash the entire pipeline - Configurable `throwOnError` flag allowing graceful degradation vs fail-fast behavior - Error logging with index and record context for debugging - Returns structured result with error details, counts, and audit trail - Provides a streaming variant for large datasets with the same resilience properties - Validates against a known set of valid statuses **Output B** has essentially zero production resilience: - No input validation - passing `null`, `undefined`, a string, or a number as `transactions` causes an immediate hard crash (`TypeError: transactions.filter is not a function`) - No per-record validation - if any record has `amount: null`, `amount: "abc"`, `amount: undefined`, or `amount: NaN`, the `reduce` silently produces `NaN` as the total with no warning or error - If a record is `null` or `undefined` in the array, `transaction.status` throws a `TypeError` - No handling for missing keys - `transaction.amount` being `undefined` silently corrupts the sum to `NaN` - `toFixed(2)` on a `NaN` result produces `"NaN"` with no error indication - No error logging, no audit trail, no way to identify which records were problematic - The alternative `for...of` loop has the same vulnerabilities In a hostile production environment with malformed types, missing keys, nulls, and unexpected values, Output B would silently produce incorrect results (`NaN` sums) or hard crash on null records. Output A would either gracefully skip bad records while logging them, or throw descriptive errors at the point of failure, making debugging straightforward. The difference in production safety is stark.

=====

[CHALLENGER: gemini-2.5-flash + Cogrithm] vs [INCUMBENT: gpt-4o Zero-Shot]

[✓] Cogrithm Done (51.22s)

[✓] Zero-Shot Done (5.23s)

-> Firing Tri-Judge Panel (Supreme Court)...

[!] Recusing Judge: gpt-4o (Self-Preference Conflict)

🗳️ CONSENSUS: 2 Votes for Cogrithm | 0 Votes for Zero-Shot

🏆 MATCH WINNER: Challenger (Cogrithm)

[claude-opus-4-6] Voted: A

Rationale:

Output A demonstrates vastly superior production resilience compared to Output B. Output A includes: (1) JSON parsing wrapped in try/catch, (2) validation that the parsed payload is an array, (3) per-record validation checking that each transaction is a non-null object with all required fields present and of the correct types (string for account_id/product_id/status, number for amount), (4) graceful skipping of malformed records with detailed logging of the index, reason, and offending record, (5) stdin error handling, and (6) proper process exit codes on failure. This means when hostile data arrives - null entries, missing keys, strings where numbers are expected, NaN amounts, undefined objects in the array - Output A will skip those records, log warnings, and continue processing valid data without crashing. Output B, by contrast, has zero validation or error handling. If the `transactions` array contains a record where `amount` is a string (e.g., "abc"), `reduce` will silently concatenate strings instead of summing numbers, producing a completely wrong result with no warning. If a record is `null` or `undefined`, `transaction.status` will throw a `TypeError` and crash the process. If `amount` is missing, it will add `undefined` to the total, resulting in `NaN` propagating silently through the sum. The code uses hardcoded sample data rather than reading from any external source, making it unsuitable for actual production use. The closing comment merely says "make sure to handle any potential errors or edge cases as needed" without actually doing so - which is exactly the kind of code that fails catastrophically in production with hostile data. Output A is clearly the winner on every dimension of production resilience: crash prevention, graceful degradation, anomaly logging, and input validation.

[gemini-2.5-pro] Voted: A

Rationale:

Output A is overwhelmingly superior because it is designed with production resilience as its core principle. It correctly anticipates a hostile data environment by implementing multi-

layered validation. First, it validates the entire payload, ensuring it's parseable JSON and an array, preventing immediate crashes. Second, and most critically, it validates each individual record for the presence and correct data type of every required field. When it encounters a malformed record (e.g., a missing 'amount' key, a string where a number is expected), it doesn't crash the entire process. Instead, it gracefully skips the invalid record, logs it with its index and the reason for failure, and continues processing the rest of the data. This ensures the pipeline remains operational and provides clear, actionable logs for developers to debug upstream data issues. Output B, in stark contrast, is a 'happy path' script that is dangerously brittle. It performs zero validation. It would suffer a hard, unhandled runtime crash (TypeError) if the input data contained a `null` value instead of a transaction object, or if the top-level payload wasn't an array. It would produce silently corrupted output (e.g., `NaN` or incorrect string concatenation) if an 'amount' field were missing or had the wrong type. This code would fail immediately in the specified production environment, requiring emergency developer intervention to diagnose the crash and add the very validation that Output A includes by default.

=====
[CHALLENGER: gemini-2.5-flash + Cogrithm] vs [INCUMBENT: claude-opus-4-6 Zero-Shot]
[✓] Cogrithm Done (40.28s)
[✓] Zero-Shot Done (17.75s)
-> Firing Tri-Judge Panel (Supreme Court)...
[!] Recusing Judge: claude-opus-4-6 (Self-Preference Conflict)

🗳️ CONSENSUS: 2 Votes for Cogrithm | 0 Votes for Zero-Shot
🏆 MATCH WINNER: Challenger (Cogrithm)

[gpt-4o] Voted: A
Rationale:

Output A demonstrates a robust approach to handling potentially malformed and hostile data. It includes comprehensive error handling for JSON parsing, structural validation, and field-level validation. It logs detailed error messages for each anomaly, ensuring that the system does not crash and provides clear feedback for debugging. This approach is crucial in a production environment where data integrity cannot be guaranteed. Output B, while functional, lacks the depth of validation and error handling present in Output A. It assumes the input is always a valid array and does not handle JSON parsing or structural anomalies, which could lead to runtime errors in a volatile environment. Therefore, Output A is more resilient and better suited for production use.

[gemini-2.5-pro] Voted: A
Rationale:

Output A is vastly superior in terms of production resilience. It is designed defensively from the ground up, anticipating the exact kind of hostile and malformed data described in the rubric. Key strengths of Output A: 1. **Handles the Full Pipeline:** It correctly accepts a raw JSON string, which is the typical starting point for a payload. Its first step is a `try/catch` block for `JSON.parse`, preventing a hard crash from malformed JSON. 2. **Multi-level Validation:** It validates the payload at every level: it checks that the parsed data is an array, then iterates and checks that each record is a non-null object. This prevents the `TypeError` that would crash Output B if the array contained `null` or a primitive value. 3. **Robust Schema Checking:** For each record, it explicitly checks for the existence (`hasOwnProperty`) and correct type of both `status` and `amount` before attempting to use them. This prevents crashes from missing keys or unexpected types (e.g., `status: 123`). 4. **Graceful Error Aggregation:** Instead of crashing or throwing an exception, it collects detailed error messages into an array and returns them alongside the successfully calculated sum. This allows the system to continue processing valid data while logging specific, actionable information about the invalid records for later debugging. This is a hallmark of a resilient production system. 5. **Flexible Data Handling:** It correctly uses `parseFloat` and an `isNaN` check, allowing the `amount` to be either a number or a numeric string (a very common real-world scenario) while safely rejecting non-numeric values like `invalid`. Output B, in contrast, is extremely brittle and would fail catastrophically in a production environment: 1. **Prone to Crashing:** Its core logic (`record.status === "active"`) will throw a fatal `TypeError` if any element in the input array is `null` or not an object, bringing the entire process to a halt. 2. **Inflexible Type Checking:** It strictly requires `typeof record.amount` to be `number`, incorrectly rejecting valid data like `amount: "150.75"` which is common in JSON payloads. 3. **Crude Error Handling:** Its main error handling mechanism is to `throw`, which is a fail-fast strategy unsuited for a data pipeline that should process valid records despite encountering some bad ones. While the example usage has a `try/catch`, the function itself is not designed to be resilient. Output A demonstrates a professional, production-ready approach by assuming data is hostile, validating it at every step, and ensuring the pipeline continues to run while providing clear diagnostics. Output B would be a constant source of production failures.

=====
🔥 INITIATING STRATEGIC EDGE-REASONING BATTLE ROYALE 🔥
TARGET: High-Density Strategic Arbitrage (Ultimate Tier)
=====

[CHALLENGER: claude-haiku-4-5-20251001 + Cogrithm] vs [INCUMBENT: gpt-4o Zero-Shot]
[✓] Cogrithm Done (33.3s)

[✓] Zero-Shot Done (6.67s)
-> Firing Tri-Judge Panel (Supreme Court)...
[!] Recusing Judge: gpt-4o (Self-Preference Conflict)

🗳️ CONSENSUS: 2 Votes for Cogrithm | 0 Votes for Zero-Shot

🏆 MATCH WINNER: Challenger (Cogrithm)

[claude-opus-4-6] Voted: A

Rationale:

Output A is dramatically more resilient in a hostile production environment. Let's analyze each against the critical rubric: **Output B** has multiple hard crash vectors: 1. **KeyError** on missing 'status': If any record lacks a 'status' key, 'transaction['status']' throws an unhandled **KeyError**, crashing the entire pipeline. 2. **KeyError** on missing 'amount': Same issue with 'transaction['amount']'. 3. **TypeError** on null/non-string status: If 'status' is 'None' or a non-string type, the '== 'active'' comparison won't crash but will silently skip records that might need attention. More critically, if 'amount' is a string like '"150.0"', the '+=' operation will crash with a **TypeError**. 4. **No input validation**: If 'json.loads()' receives malformed JSON, the script crashes with an unhandled 'json.JSONDecodeError'. If the root element is a dict instead of a list, the 'for' loop iterates over keys instead of records, producing nonsensical results or **KeyErrors**. 5. **No type checking on amount**: If amount is None, a string, or NaN, the script will crash or produce incorrect results. 6. **Not encapsulated in a function**: The code is a flat script with no reusability, no error handling, and no way to gracefully recover. **Output A** addresses all of these: 1. **Input type validation**: Checks that the input is a string before parsing. 2. **JSON parse error handling**: Catches 'json.JSONDecodeError' and wraps it in a descriptive custom exception. 3. **Root structure validation**: Ensures the parsed JSON is a list, not a dict or primitive. 4. **Per-record validation**: Checks each record is a dict, has required 'amount' and 'status' fields, validates their types. 5. **Null amount protection**: Explicitly checks for 'None' amounts. 6. **Type coercion with error handling**: Uses 'float()' with try/except to handle string amounts gracefully. 7. **Custom exception class**: Provides clear, actionable error messages with record index for debugging. 8. **Consistent return type**: Always returns 'float', eliminating downstream type ambiguity. 9. **Function encapsulation**: Reusable, testable, and composable. The only minor criticism of Output A is that it raises exceptions rather than logging and skipping bad records (which could be preferable in some pipelines to avoid halting on a single bad record). However, the fail-fast approach is a valid and often preferred production strategy, especially for financial transaction data where silent data loss would be worse than stopping. The validation errors are descriptive and pinpoint exactly which record failed and why, minimizing debugging time. Output B would require multiple rounds of developer intervention to handle the inevitable dirty data in production. Output A is production-ready as written.

[gemini-2.5-pro] Voted: A

Rationale:

Output A is vastly superior and designed for a real-world production environment, directly addressing the prompt's critical rubric. It anticipates and handles a wide range of hostile data scenarios that would cause Output B to crash instantly. Key resilience features in Output A that are absent in Output B: 1. **Input Validation**: Output A validates that the input is a string, the string is valid JSON, and the root JSON structure is a list. Output B makes blind assumptions, leading to unhandled 'TypeError' or 'json.JSONDecodeError' crashes. 2. **Schema Enforcement**: Inside the loop, Output A checks for the existence of required keys ('amount', 'status') before attempting to access them. Output B accesses 'transaction['status']' and 'transaction['amount']' directly, which will cause an unhandled 'KeyError' if a key is missing. 3. **Type Safety**: Output A explicitly checks the types of the 'status' and 'amount' fields. It gracefully handles cases where 'amount' might be a string-encoded number (e.g., "150.50") by attempting a 'float()' conversion within a 'try...except' block. This prevents the 'TypeError' that would crash Output B if it encountered a non-numeric amount. 4. **Error Handling**: Output A uses a custom exception class ('TransactionValidationError') with highly specific error messages (including the index of the faulty record), which is invaluable for debugging. Output B has no error handling whatsoever and would simply crash the entire process, requiring a developer to manually inspect logs and data to find the root cause. In summary, Output A is a production-hardened function that will survive hostile data by failing fast with clear, actionable errors. Output B is a naive, "happy-path" script that is guaranteed to cause repeated system failures in the environment described.

[CHALLENGER: claude-haiku-4-5-20251001 + Cogrithm] vs [INCUMBENT: gemini-2.5-pro Zero-Shot]

[✓] Cogrithm Done (31.17s)
[✓] Zero-Shot Done (25.57s)
-> Firing Tri-Judge Panel (Supreme Court)...
[!] Recusing Judge: gemini-2.5-pro (Self-Preference Conflict)

🗳️ CONSENSUS: 2 Votes for Cogrithm | 0 Votes for Zero-Shot

🏆 MATCH WINNER: Challenger (Cogrithm)

[gpt-4o] Voted: A

Rationale:

Output A demonstrates a robust approach to handling potentially malformed and hostile data. It includes comprehensive error handling, type validation, and uses Decimal for precise financial calculations. The code explicitly checks for the correct types and values, raising informative

errors when expectations are not met, which is crucial in a volatile production environment. Output B, while concise and Pythonic, lacks the same level of defensive programming. It assumes the presence and correctness of keys and values without validation, which could lead to runtime errors if the data is malformed. Output A's approach minimizes the need for immediate human debugging by providing clear error messages and handling edge cases gracefully, making it more resilient in production.

[claude-opus-4-6] Voted: A

Rationale:

The critical evaluation criterion is production resilience against hostile, malformed data. Let me analyze both outputs systematically. **Output A** demonstrates comprehensive defensive programming: - Handles both JSON string and pre-parsed list inputs with explicit type checking - Validates that the parsed result is actually a list - Validates each record is a dictionary before accessing keys - Uses `record.get('status')` to safely handle missing 'status' keys (returns None, no `KeyError`) - Explicitly checks for missing 'amount' field on active records and raises a clear `ValueError` - Uses `Decimal` arithmetic to avoid floating-point precision issues (critical for financial data) - Safe amount casting via `Decimal(str(amount_raw))` with `try-except` to catch non-numeric values - Provides detailed error messages with record index, actual value, and expected type for debugging - Per-record validation means one bad record doesn't silently corrupt the entire sum - Includes type hints and thorough docstrings **Output B** has significant production vulnerabilities: - The main approach (`sum(t['amount'] for t in transactions if t['status'] == 'active')`) uses direct dictionary key access (`t['amount']` and `t['status']`), which will throw a `KeyError` if any record is missing 'status' or 'amount'. This is a hard crash in production. - No validation that individual records are dictionaries - if the list contains a null, string, or integer, it will throw a `TypeError` when trying to access keys. - No validation of the 'amount' field type - if amount is a string like "N/A" or null, the sum will crash with a `TypeError`. - No `Decimal` arithmetic - floating-point errors will accumulate in financial calculations. - The alternative for-loop version is slightly better (uses `.get()` for both keys), but it silently defaults missing amounts to 0 rather than flagging the anomaly, and it still doesn't validate that records are dicts or that amount values are actually numeric. A string amount would still cause a `TypeError` on `+=`. - The JSON decode error handling is present but minimal. - No input type validation - passing a non-string, non-list input would crash. In a hostile production environment with missing keys, null values, non-numeric amounts, and malformed records, Output A would either handle the issue gracefully or raise a clear, debuggable error. Output B would crash with opaque `KeyErrors` and `TypeErrors` that require developer investigation to trace back to the root cause. Output A is clearly superior for production resilience.

=====

[CHALLENGER: gemini-2.5-flash + Cogrithm] vs [INCUMBENT: gpt-4o Zero-Shot]

[✓] Cogrithm Done (58.61s)

[✓] Zero-Shot Done (3.69s)

-> Firing Tri-Judge Panel (Supreme Court)...

[!] Recusing Judge: gpt-4o (Self-Preference Conflict)

=====

🗳️ CONSENSUS: 2 Votes for Cogrithm | 0 Votes for Zero-Shot

🏆 MATCH WINNER: Challenger (Cogrithm)

[claude-opus-4-6] Voted: A

Rationale:

The critical evaluation criterion is production resilience against hostile, malformed data. **Output B** has multiple hard crash points: 1. `json.loads()` on malformed JSON will throw an unhandled `json.JSONDecodeError`, crashing the script. 2. `transaction['status']` will throw a `KeyError` if any record is missing the 'status' key. 3. `transaction['amount']` will throw a `KeyError` if any record is missing the 'amount' key. 4. If `transaction` is not a dict (e.g., a null, string, or integer in the list), `transaction['status']` will throw a `TypeError`. 5. If `amount` is a string (e.g., "N/A") or None, the `+=` operation will throw a `TypeError`. 6. No validation that the parsed JSON is actually a list - if it's a dict, iterating will yield keys, not records, leading to `TypeError` on key access. All of these are unhandled and will cause immediate hard runtime crashes in a production environment with dirty data. Every single one would require developer intervention to debug and fix. **Output A** implements comprehensive validation: 1. Wraps `json.loads()` in a `try/except`, converting `JSONDecodeError` to a custom `InvalidPayloadError`. 2. Validates the top-level structure is a list. 3. Validates each record is a dictionary. 4. Checks for missing 'status' and 'amount' keys using `.get()` with None checks. 5. Validates 'status' is a string. 6. Wraps `float(amount_value)` in a `try/except` to handle non-numeric amount values. 7. Uses custom exception hierarchy for structured error reporting. 8. Includes comprehensive test cases demonstrating behavior with malformed records, missing keys, non-string status values, invalid amounts, empty payloads, and non-list payloads. The one nuanced consideration with Output A is that it uses a fail-fast approach (raising on the first bad record rather than skipping/logging and continuing). This means a single bad record in a batch will prevent processing of subsequent valid records. In some production scenarios, a skip-and-log approach would be preferable. However, the fail-fast approach is still a deliberate, controlled behavior - it doesn't cause an `unhandled` crash; the caller receives a well-typed exception with clear context about which record failed and why. This is vastly superior to Output B's unhandled crashes. Output A also has a minor edge case: if a record has `"amount": null` explicitly in JSON, `record.get('amount')` returns `None`, and the check `if amount_value is None` treats it as missing. For active records this is correct behavior (rejecting null amounts). For non-active records, this would raise an error even though the amount isn't needed, which is slightly over-strict but not a crash risk. Overall, Output A is

dramatically more production-resilient than Output B.

[gemini-2.5-pro] Voted: A

Rationale:

Output A is engineered for a hostile production environment, while Output B is a 'happy path' script that would crash immediately. Output A demonstrates a Zero-Trust approach to data. It meticulously validates every aspect of the input: the overall JSON structure, the type of each record, the existence of required keys ('status', 'amount'), and the data types of their values. When it encounters an anomaly, it doesn't crash with a generic Python error; it raises a specific, custom exception with a detailed message indicating the exact record and problem. This fail-fast strategy is critical in production as it prevents the processing of corrupt data which could lead to silent, incorrect calculations. A developer can immediately diagnose the issue from the exception without needing to manually inspect a large payload. Output B, in contrast, makes dangerous assumptions. It uses direct dictionary access (transaction['status']) which will cause an unhandled `KeyError` if the key is missing or a `TypeError` if the record isn't a dictionary. It performs direct addition (+= transaction['amount']) which will cause an unhandled `TypeError` if the amount is a string, null, or any non-numeric type. This code is brittle and would lead to constant, uninformative system failures, requiring significant developer intervention to debug and fix.

=====

[CHALLENGER: gemini-2.5-flash + Cogrithm] vs [INCUMBENT: claude-opus-4-6 Zero-Shot]

[✓] Cogrithm Done (52.28s)

[✓] Zero-Shot Done (15.91s)

-> Firing Tri-Judge Panel (Supreme Court)...

[!] Recusing Judge: claude-opus-4-6 (Self-Preference Conflict)

=====

🗳️ CONSENSUS: 2 Votes for Cogrithm | 0 Votes for Zero-Shot

🏆 MATCH WINNER: Challenger (Cogrithm)

[gpt-4o] Voted: A

Rationale:

Output A demonstrates a comprehensive approach to handling volatile and hostile data environments. It uses JSON schema validation to ensure data integrity, handles exceptions gracefully, and provides detailed error reporting through a structured TransactionSummary object. This approach minimizes the risk of runtime crashes and provides clear diagnostics for debugging. Output B, while simpler, lacks robust error handling and validation. It raises exceptions for missing fields or incorrect data types, which could lead to system failures in a production environment with unpredictable data. Therefore, Output A is more resilient and better suited for production use.

[gemini-2.5-pro] Voted: A

Rationale:

Output A is overwhelmingly superior in the context of a volatile production environment. Its design philosophy is built around resilience and graceful degradation. By using `jsonschema` for validation, it declaratively enforces both the presence and, critically, the *type* of each field. When it encounters a malformed record (e.g., a missing key, an amount that is a string, a status that is an integer), it catches the `ValidationError`, logs the specific record and reason for failure, and continues processing the rest of the batch. This prevents a single bad record from causing a hard crash and halting the entire pipeline. Output B, in stark contrast, is extremely brittle. Its design pattern is to raise exceptions (`KeyError`, `TypeError`) upon encountering the first piece of invalid data. This 'fail-fast' approach is inappropriate for this use case. In a real production scenario, this code would crash constantly. For example: 1. If a record has "amount": "invalid", the `sum()` function will raise an unhandled `TypeError`, crashing the script. 2. If an element in the main list is not a dictionary (e.g., `null` or a string), the `transaction.keys()` call will raise an `AttributeError`, crashing the script. 3. If a single record is missing a key, the entire batch processing stops with a `KeyError`. Output A anticipates these exact scenarios and handles them gracefully, returning a comprehensive summary of what succeeded and what failed. This makes it robust, reliable, and far easier to debug and monitor in production. Output B would cause repeated system failures and require constant developer intervention.

=====

Appendix B: AI Code Generation Evaluation Test Source

```
import os
import textwrap
import time
import json
import requests
from openai import OpenAI
from anthropic import Anthropic
from google import genai
from google.genai import types

# =====
# 1. CONFIGURATION & PROMPTS (TRANSPARENCY BLOCK)
# =====
OAI_KEY = os.environ.get("OPENAI_API_KEY")
ANT_KEY = os.environ.get("ANTHROPIC_API_KEY")
GEM_KEY = os.environ.get("GEMINI_API_KEY")
COG_KEY = os.environ.get("COG_LIVE_KEY")

oai_client = OpenAI(api_key=OAI_KEY)
ant_client = Anthropic(api_key=ANT_KEY)
gem_client = genai.Client(api_key=GEM_KEY)

# CHALLENGERS: Edge-Reasoning Models (Stripped of Micro-Models)
FAST_MODELS = [
    ("anthropic", "claude-haiku-4-5-20251001"),
    ("google", "gemini-2.5-flash")
]

# INCUMBENTS: Frontier Models
FRONTIER_MODELS = [
    ("openai", "gpt-4o"),
    ("anthropic", "claude-opus-4-6"),
    ("google", "gemini-2.5-pro")
]

# THE SUPREME COURT: Full Panel for 100% Blind Consensus
EVAL_MODELS = [
    ("openai", "gpt-4o"),
    ("anthropic", "claude-opus-4-6"),
    ("google", "gemini-2.5-pro")
]

# --- THE WORKLOAD ---
TEST_QUERY = """
Write a GoLang script to process a JSON payload of transactions from our NoSQL
database. I need you to calculate the total sum of the amount for all records where
the status is "active".
The records have this structure:
{
account_id,
product_id,
```

```

amount,
status
}
"""

TEST_DATA = ""

# --- ZERO-SHOT BASELINE PROMPT ---
SYS_ZERO_SHOT = ""

# --- THE JUDGE RUBRIC (DOMAIN AGNOSTIC) ---
SYS_JUDGE = ""
Evaluate Output A and Output B.

CRITICAL RUBRIC:
Assume the data entering this code in a real production environment is highly volatile
and hostile. It will inevitably contain malformed types, missing keys, nulls, and
unexpected boundary anomalies (e.g., strings where integers are expected, silent NaNs)
due to upstream system errors.

Indicate whether the code presented would survive these unpredictable production
realities or if it would cause a system failure.

Primary evaluations MUST focus on Production Resilience:
- Will the code experience a hard runtime crash (e.g., unhandled TypeErrors,
KeyErrors, index out of bounds) when it hits an edge case, requiring multiple prompts,
developer edits, and introspection to fix?
- Or does it enforce strict validation, handle unforeseen errors gracefully, log the
anomalies, and keep the rest of the pipeline running safely?

You MUST respond with ONLY a valid JSON object matching this exact structure:
{
"rationale": "Detailed explanation justifying the winner, explicitly addressing which
code prevents hard production crashes, handles dirty data gracefully, and minimizes
the need for immediate human debugging.",
"score_a": [int 1-100],
"score_b": [int 1-100],
"winner": "[A or B]"
}
"""

# =====
# 2. UTILITIES & SDK ROUTER
# =====
def get_provider_key(vendor: str) -> str:
    if vendor == "openai": return OAI_KEY
    if vendor == "anthropic": return ANT_KEY
    if vendor == "google": return GEM_KEY
    return ""

def truncate(text: str, max_len=150) -> str:
    if not text: return "N/A"
    text = text.replace('\n', ' ')
    return (text[:max_len] + '...') if len(text) > max_len else text

def call_local_llm(vendor: str, model: str, system: str, user: str, temp: float = 0.2)
-> str:
    max_retries = 3
    for attempt in range(max_retries):
        try:
            if vendor == "openai":
                response = oai_client.chat.completions.create(
                    model=model,

```

```

        messages=[{"role": "system", "content": system}, {"role": "user",
"content": user}],
        temperature=temp
    )
    return response.choices[0].message.content
elif vendor == "anthropic":
    if "opus" in model:
        response = ant_client.messages.create(
            model=model, system=system,
            messages=[{"role": "user", "content": user}], max_tokens=4096
        )
    else:
        response = ant_client.messages.create(
            model=model, system=system,
            messages=[{"role": "user", "content": user}],
            max_tokens=4096,
            temperature=temp
        )
    return response.content[0].text
elif vendor == "google":
    response = gem_client.models.generate_content(
        model=model, contents=user,
        config=types.GenerateContentConfig(system_instruction=system,
temperature=temp)
    )
    return response.text
except Exception as e:
    if attempt == max_retries - 1: return f"ERROR: {str(e)}"
    time.sleep(2 ** attempt)

```

```
# =====
```

```
# 3. EXECUTION PIPELINES
```

```
# =====
```

```
def run_cogrihnm_ultimate(vendor: str, model: str, query: str, data: str) -> dict:
```

```
    url = "https://api.cogrihnm.com/v1/execute/ultimate"
```

```
    headers = {
```

```
        "Authorization": f"Bearer {COG_KEY}",
```

```
        "X-Provider-Model": model,
```

```
        "X-Provider-Key": get_provider_key(vendor),
```

```
        "Content-Type": "application/json"
```

```
    }
```

```
    payload = {"query": query, "data": data}
```

```
    start_time = time.time()
```

```
    try:
```

```
        with requests.post(url, headers=headers, json=payload, stream=True) as
response:
```

```
            response.raise_for_status()
```

```
            final_result = ""
```

```
            for line in response.iter_lines():
```

```
                if line:
```

```
                    decoded_line = line.decode('utf-8')
```

```
                    try:
```

```
                        chunk_data = json.loads(decoded_line)
```

```
                        if chunk_data.get("status") == "success":
```

```
                            final_result = chunk_data.get("result", "")
```

```
                        elif chunk_data.get("status") == "error":
```

```
                            return {"output": f"Engine Error:
{chunk_data.get('detail')}", "latency": round(time.time() - start_time, 2)}
```

```
                    except json.JSONDecodeError:
```

```
                        continue
```

```
            if not final_result:
```

```

        return {"output": "Error: Stream closed without success payload.",
"latency": round(time.time() - start_time, 2)}

        return {"output": final_result, "latency": round(time.time() - start_time,
2)}

    except Exception as e:
        return {"output": f"API Error: {str(e)}", "latency": round(time.time() -
start_time, 2)}

def run_zero_shot(vendor: str, model: str, query: str, data: str) -> dict:
    start_time = time.time()
    user_prompt = f"CONTEXT:\n{data}\n\nQUERY:\n{query}"
    output = call_local_llm(vendor, model, SYS_ZERO_SHOT, user_prompt)
    return {"output": output, "latency": round(time.time() - start_time, 2)}

# =====
# 4. TRI-PANEL (SUPREME COURT) EVALUATION
# =====
def evaluate_with_panel(query: str, cog_out: str, zs_out: str, incumbent_model: str)
-> dict:
    # Calculate exact word counts natively in Python
    words_a = len(cog_out.split())
    words_b = len(zs_out.split())

    # Inject word counts as explicit metadata into the prompt
    user_eval = f"ORIGINAL PROMPT:\n{query}\n\n"
    user_eval += f"=== OUTPUT A (Word Count: {words_a}) ===\n{cog_out}\n\n"
    user_eval += f"=== OUTPUT B (Word Count: {words_b}) ===\n{zs_out}"

    panel_results = []
    cog_votes, zs_votes = 0, 0

    for eval_vendor, eval_model in EVAL_MODELS:
        if eval_model == incumbent_model:
            print(f"    [!] Recusing Judge: {eval_model} (Self-Preference Conflict)")
            continue

        raw_response = call_local_llm(eval_vendor, eval_model, SYS_JUDGE, user_eval,
temp=0.0)
        try:
            clean_json = raw_response.replace('```json', '').replace('```',
''.strip())
            data = json.loads(clean_json[clean_json.find('{'):clean_json.rfind('}') +
1])
            data['judge_name'] = eval_model

            if data.get("winner") == "A": cog_votes += 1
            elif data.get("winner") == "B": zs_votes += 1

            panel_results.append(data)
        except:
            panel_results.append({"judge_name": eval_model, "rationale": "Parsing
Error", "winner": "Error"})

    return {"panel_results": panel_results, "cog_votes": cog_votes, "zs_votes":
zs_votes}

# =====
# 5. MATRIX RUNNER
# =====
def run_whitepaper_matrix():
    print("="*80)
    print("🔥 INITIATING STRATEGIC EDGE-REASONING BATTLE ROYALE 🔥 ")

```

```

print(f"TARGET: High-Density Strategic Arbitrage (Ultimate Tier)")
print("="*80 + "\n")

for fast_vendor, fast_model in FAST_MODELS:
    for front_vendor, front_model in FRONTIER_MODELS:
        if fast_vendor == front_vendor: continue

        print(f"\n[CHALLENGER: {fast_model} + Cogrithm] vs [INCUMBENT:
{front_model} Zero-Shot]")

        # 1. Execute
        cog_result = run_cogrithm_ultimate(fast_vendor, fast_model, TEST_QUERY,
TEST_DATA)
        print(f" [✓] Cogrithm Done ({cog_result['latency']}s)")

        zs_result = run_zero_shot(front_vendor, front_model, TEST_QUERY,
TEST_DATA)
        print(f" [✓] Zero-Shot Done ({zs_result['latency']}s)")

        # 2. Evaluate
        print(f" -> Firing Tri-Judge Panel (Supreme Court)...")
        eval_data = evaluate_with_panel(TEST_QUERY, cog_result["output"],
zs_result["output"], front_model)

        # 3. Print Output
        print("-" * 80)
        print(f"🗳️ CONSENSUS: {eval_data['cog_votes']} Votes for Cogrithm |
{eval_data['zs_votes']} Votes for Zero-Shot")

        if eval_data['cog_votes'] > eval_data['zs_votes']:
            winner_str = "Challenger (Cogrithm)"
        elif eval_data['zs_votes'] > eval_data['cog_votes']:
            winner_str = "Incumbent (Zero-Shot)"
        else:
            winner_str = "Tie"

        print(f"🏆 MATCH WINNER: {winner_str}\n")

        for judge in eval_data['panel_results']:
            print(f" [{judge.get('judge_name')}] Voted: {judge.get('winner')}")

            # Force the rationale to be a string to prevent textwrap TypeErrors
            full_rationale = str(judge.get('rationale', 'N/A'))
            wrapped_rationale = textwrap.fill(full_rationale, width=100,
initial_indent="    ", subsequent_indent="    ")

            print(f" Rationale:\n{wrapped_rationale}\n")

        # 4. Conditionally Dump Full Output Texts
        if winner_str in ["Incumbent (Zero-Shot)", "Tie"]:
            print("\n" + "=" * 32 + " OUTPUT LOGS " + "=" * 32)
            print(f"\n--- [ CHALLENGER: {fast_model} + Cogrithm ] ---")
            print(cog_result["output"])
            print(f"\n--- [ INCUMBENT: {front_model} Zero-Shot ] ---")
            print(zs_result["output"])
            print("\n" + "=" * 77)

        print("-" * 80)
        time.sleep(2)

if __name__ == "__main__":
    run_whitepaper_matrix()
import os
import textwrap

```

```

import time
import json
import requests
from openai import OpenAI
from anthropic import Anthropic
from google import genai
from google.genai import types

# =====
# 1. CONFIGURATION & PROMPTS (TRANSPARENCY BLOCK)
# =====
OAI_KEY = os.environ.get("OPENAI_API_KEY")
ANT_KEY = os.environ.get("ANTHROPIC_API_KEY")
GEM_KEY = os.environ.get("GEMINI_API_KEY")
COG_KEY = os.environ.get("COG_LIVE_KEY")

oai_client = OpenAI(api_key=OAI_KEY)
ant_client = Anthropic(api_key=ANT_KEY)
gem_client = genai.Client(api_key=GEM_KEY)

# CHALLENGERS: Edge-Reasoning Models (Stripped of Micro-Models)
FAST_MODELS = [
    ("anthropic", "claude-haiku-4-5-20251001"),
    ("google", "gemini-2.5-flash")
]

# INCUMBENTS: Frontier Models
FRONTIER_MODELS = [
    ("openai", "gpt-4o"),
    ("anthropic", "claude-opus-4-6"),
    ("google", "gemini-2.5-pro")
]

# THE SUPREME COURT: Full Panel for 100% Blind Consensus
EVAL_MODELS = [
    ("openai", "gpt-4o"),
    ("anthropic", "claude-opus-4-6"),
    ("google", "gemini-2.5-pro")
]

# --- THE WORKLOAD ---
TEST_QUERY = """
Write a JavaScript(Node.js) script to process a JSON payload of transactions from our
NoSQL database. I need you to calculate the total sum of the amount for all records
where the status is "active".
The records have this structure:
{
account_id,
product_id,
amount,
status
}
"""

TEST_DATA = ""

# --- ZERO-SHOT BASELINE PROMPT ---
SYS_ZERO_SHOT = ""

# --- THE JUDGE RUBRIC (DOMAIN AGNOSTIC) ---
SYS_JUDGE = """
Evaluate Output A and Output B.

CRITICAL RUBRIC:

```

Assume the data entering this code in a real production environment is highly volatile and hostile. It will inevitably contain malformed types, missing keys, nulls, and unexpected boundary anomalies (e.g., strings where integers are expected, silent NaNs) due to upstream system errors.

Indicate whether the code presented would survive these unpredictable production realities or if it would cause a system failure.

Primary evaluations MUST focus on Production Resilience:

- Will the code experience a hard runtime crash (e.g., unhandled `TypeError`s, `KeyErrors`, index out of bounds) when it hits an edge case, requiring multiple prompts, developer edits, and introspection to fix?
- Or does it enforce strict validation, handle unforeseen errors gracefully, log the anomalies, and keep the rest of the pipeline running safely?

You MUST respond with ONLY a valid JSON object matching this exact structure:

```
{
"rationale": "Detailed explanation justifying the winner, explicitly addressing which
code prevents hard production crashes, handles dirty data gracefully, and minimizes
the need for immediate human debugging.",
"score_a": [int 1-100],
"score_b": [int 1-100],
"winner": "[A or B]"
}
"""
```

```
# =====
# 2. UTILITIES & SDK ROUTER
# =====
```

```
def get_provider_key(vendor: str) -> str:
    if vendor == "openai": return OAI_KEY
    if vendor == "anthropic": return ANT_KEY
    if vendor == "google": return GEM_KEY
    return ""
```

```
def truncate(text: str, max_len=150) -> str:
    if not text: return "N/A"
    text = text.replace('\n', ' ')
    return (text[:max_len] + '...') if len(text) > max_len else text
```

```
def call_local_llm(vendor: str, model: str, system: str, user: str, temp: float = 0.2)
-> str:
```

```
    max_retries = 3
    for attempt in range(max_retries):
        try:
            if vendor == "openai":
                response = oai_client.chat.completions.create(
                    model=model,
                    messages=[{"role": "system", "content": system}, {"role": "user",
"content": user}],
                    temperature=temp
                )
                return response.choices[0].message.content
            elif vendor == "anthropic":
                if "opus" in model:
                    response = ant_client.messages.create(
                        model=model, system=system,
                        messages=[{"role": "user", "content": user}], max_tokens=4096
                    )
                else:
                    response = ant_client.messages.create(
                        model=model, system=system,
                        messages=[{"role": "user", "content": user}],
                        max_tokens=4096,
```

```

        temperature=temp
    )
    return response.content[0].text
elif vendor == "google":
    response = gem_client.models.generate_content(
        model=model, contents=user,
        config=types.GenerateContentConfig(system_instruction=system,
temperature=temp)
    )
    return response.text
except Exception as e:
    if attempt == max_retries - 1: return f"ERROR: {str(e)}"
    time.sleep(2 ** attempt)

# =====
# 3. EXECUTION PIPELINES
# =====
def run_cogrithm_ultimate(vendor: str, model: str, query: str, data: str) -> dict:
    url = "https://api.cogrithm.com/v1/execute/ultimate"
    headers = {
        "Authorization": f"Bearer {COG_KEY}",
        "X-Provider-Model": model,
        "X-Provider-Key": get_provider_key(vendor),
        "Content-Type": "application/json"
    }
    payload = {"query": query, "data": data}
    start_time = time.time()

    try:
        with requests.post(url, headers=headers, json=payload, stream=True) as
response:
            response.raise_for_status()

            final_result = ""
            for line in response.iter_lines():
                if line:
                    decoded_line = line.decode('utf-8')
                    try:
                        chunk_data = json.loads(decoded_line)
                        if chunk_data.get("status") == "success":
                            final_result = chunk_data.get("result", "")
                        elif chunk_data.get("status") == "error":
                            return {"output": f"Engine Error:
{chunk_data.get('detail')}", "latency": round(time.time() - start_time, 2)}
                    except json.JSONDecodeError:
                        continue

            if not final_result:
                return {"output": "Error: Stream closed without success payload.",
"latency": round(time.time() - start_time, 2)}

            return {"output": final_result, "latency": round(time.time() - start_time,
2)}

    except Exception as e:
        return {"output": f"API Error: {str(e)}", "latency": round(time.time() -
start_time, 2)}

def run_zero_shot(vendor: str, model: str, query: str, data: str) -> dict:
    start_time = time.time()
    user_prompt = f"CONTEXT:\n{data}\n\nQUERY:\n{query}"
    output = call_local_llm(vendor, model, SYS_ZERO_SHOT, user_prompt)
    return {"output": output, "latency": round(time.time() - start_time, 2)}

```

```

# =====
# 4. TRI-PANEL (SUPREME COURT) EVALUATION
# =====
def evaluate_with_panel(query: str, cog_out: str, zs_out: str, incumbent_model: str)
-> dict:
    # Calculate exact word counts natively in Python
    words_a = len(cog_out.split())
    words_b = len(zs_out.split())

    # Inject word counts as explicit metadata into the prompt
    user_eval = f"ORIGINAL PROMPT:\n{query}\n\n"
    user_eval += f"=== OUTPUT A (Word Count: {words_a}) ===\n{cog_out}\n\n"
    user_eval += f"=== OUTPUT B (Word Count: {words_b}) ===\n{zs_out}"

    panel_results = []
    cog_votes, zs_votes = 0, 0

    for eval_vendor, eval_model in EVAL_MODELS:
        if eval_model == incumbent_model:
            print(f" [!] Recusing Judge: {eval_model} (Self-Preference Conflict)")
            continue

        raw_response = call_local_llm(eval_vendor, eval_model, SYS_JUDGE, user_eval,
temp=0.0)
        try:
            clean_json = raw_response.replace('```json', '').replace('```',
''.strip())
            data = json.loads(clean_json[clean_json.find('{'):clean_json.rfind('}') +
1])
            data['judge_name'] = eval_model

            if data.get("winner") == "A": cog_votes += 1
            elif data.get("winner") == "B": zs_votes += 1

            panel_results.append(data)
        except:
            panel_results.append({"judge_name": eval_model, "rationale": "Parsing
Error", "winner": "Error"})

    return {"panel_results": panel_results, "cog_votes": cog_votes, "zs_votes":
zs_votes}

# =====
# 5. MATRIX RUNNER
# =====
def run_whitepaper_matrix():
    print("="*80)
    print("🔥 INITIATING STRATEGIC EDGE-REASONING BATTLE ROYALE 🔥 ")
    print(f"TARGET: High-Density Strategic Arbitrage (Ultimate Tier)")
    print("="*80 + "\n")

    for fast_vendor, fast_model in FAST_MODELS:
        for front_vendor, front_model in FRONTIER_MODELS:
            if fast_vendor == front_vendor: continue

            print(f"\n[CHALLENGER: {fast_model} + Cogrithm] vs [INCUMBENT:
{front_model} Zero-Shot]")

            # 1. Execute
            cog_result = run_cogrithm_ultimate(fast_vendor, fast_model, TEST_QUERY,
TEST_DATA)
            print(f" [✓] Cogrithm Done ({cog_result['latency']}s)")

```

```

TEST_DATA) zs_result = run_zero_shot(front_vendor, front_model, TEST_QUERY,
print(f" [✓] Zero-Shot Done ( {zs_result['latency']}s)")

# 2. Evaluate
print(f" -> Firing Tri-Judge Panel (Supreme Court)...")
eval_data = evaluate_with_panel(TEST_QUERY, cog_result["output"],
zs_result["output"], front_model)

# 3. Print Output
print("-" * 80)
print(f"🗳️ CONSENSUS: {eval_data['cog_votes']} Votes for Cogrithm |
{eval_data['zs_votes']} Votes for Zero-Shot")

if eval_data['cog_votes'] > eval_data['zs_votes']:
    winner_str = "Challenger (Cogrithm)"
elif eval_data['zs_votes'] > eval_data['cog_votes']:
    winner_str = "Incumbent (Zero-Shot)"
else:
    winner_str = "Tie"

print(f"🏆 MATCH WINNER: {winner_str}\n")

for judge in eval_data['panel_results']:
    print(f" [{judge.get('judge_name')}] Voted: {judge.get('winner')}")

# Force the rationale to be a string to prevent textwrap TypeErrors
full_rationale = str(judge.get('rationale', 'N/A'))
wrapped_rationale = textwrap.fill(full_rationale, width=100,
initial_indent="    ", subsequent_indent="    ")

print(f" Rationale:\n{wrapped_rationale}\n")

# 4. Conditionally Dump Full Output Texts
if winner_str in ["Incumbent (Zero-Shot)", "Tie"]:
    print("\n" + "=" * 32 + " OUTPUT LOGS " + "=" * 32)
    print(f"\n--- [ CHALLENGER: {fast_model} + Cogrithm ] ---")
    print(cog_result["output"])
    print(f"\n--- [ INCUMBENT: {front_model} Zero-Shot ] ---")
    print(zs_result["output"])
    print("\n" + "=" * 77)

print("=" * 80)
time.sleep(2)

if __name__ == "__main__":
    run_whitepaper_matrix()
import os
import textwrap
import time
import json
import requests
from openai import OpenAI
from anthropic import Anthropic
from google import genai
from google.genai import types

# =====
# 1. CONFIGURATION & PROMPTS (TRANSPARENCY BLOCK)
# =====
OAI_KEY = os.environ.get("OPENAI_API_KEY")
ANT_KEY = os.environ.get("ANTHROPIC_API_KEY")
GEM_KEY = os.environ.get("GEMINI_API_KEY")
COG_KEY = os.environ.get("COG_LIVE_KEY")

```

```

oai_client = OpenAI(api_key=OAI_KEY)
ant_client = Anthropic(api_key=ANT_KEY)
gem_client = genai.Client(api_key=GEM_KEY)

# CHALLENGERS: Edge-Reasoning Models (Stripped of Micro-Models)
FAST_MODELS = [
    ("anthropic", "claude-haiku-4-5-20251001"),
    ("google", "gemini-2.5-flash")
]

# INCUMBENTS: Frontier Models
FRONTIER_MODELS = [
    ("openai", "gpt-4o"),
    ("anthropic", "claude-opus-4-6"),
    ("google", "gemini-2.5-pro")
]

# THE SUPREME COURT: Full Panel for 100% Blind Consensus
EVAL_MODELS = [
    ("openai", "gpt-4o"),
    ("anthropic", "claude-opus-4-6"),
    ("google", "gemini-2.5-pro")
]

# --- THE WORKLOAD ---
TEST_QUERY = """
Write a Python script to process a JSON payload of transactions from our NoSQL
database. I need you to calculate the total sum of the amount for all records where
the status is "active".
The records have this structure:
{
account_id,
product_id,
amount,
status
}
"""

TEST_DATA = ""

# --- ZERO-SHOT BASELINE PROMPT ---
SYS_ZERO_SHOT = ""

# --- THE JUDGE RUBRIC (DOMAIN AGNOSTIC) ---
SYS_JUDGE = """
Evaluate Output A and Output B.

CRITICAL RUBRIC:
Assume the data entering this code in a real production environment is highly volatile
and hostile. It will inevitably contain malformed types, missing keys, nulls, and
unexpected boundary anomalies (e.g., strings where integers are expected, silent NaNs)
due to upstream system errors.

Indicate whether the code presented would survive these unpredictable production
realities or if it would cause a system failure.

Primary evaluations MUST focus on Production Resilience:
- Will the code experience a hard runtime crash (e.g., unhandled TypeErrors,
KeyErrors, index out of bounds) when it hits an edge case, requiring multiple prompts,
developer edits, and introspection to fix?
- Or does it enforce strict validation, handle unforeseen errors gracefully, log the
anomalies, and keep the rest of the pipeline running safely?

```

You MUST respond with ONLY a valid JSON object matching this exact structure:

```
{
"rationale": "Detailed explanation justifying the winner, explicitly addressing which
code prevents hard production crashes, handles dirty data gracefully, and minimizes
the need for immediate human debugging.",
"score_a": [int 1-100],
"score_b": [int 1-100],
"winner": "[A or B]"
}
"""
```

```
# =====
# 2. UTILITIES & SDK ROUTER
```

```
# =====
def get_provider_key(vendor: str) -> str:
    if vendor == "openai": return OAI_KEY
    if vendor == "anthropic": return ANT_KEY
    if vendor == "google": return GEM_KEY
    return ""
```

```
def truncate(text: str, max_len=150) -> str:
    if not text: return "N/A"
    text = text.replace('\n', ' ')
    return (text[:max_len] + '...') if len(text) > max_len else text
```

```
def call_local_llm(vendor: str, model: str, system: str, user: str, temp: float = 0.2)
-> str:
    max_retries = 3
    for attempt in range(max_retries):
        try:
            if vendor == "openai":
                response = oai_client.chat.completions.create(
                    model=model,
                    messages=[{"role": "system", "content": system}, {"role": "user",
"content": user}],
                    temperature=temp
                )
                return response.choices[0].message.content
            elif vendor == "anthropic":
                if "opus" in model:
                    response = ant_client.messages.create(
                        model=model, system=system,
                        messages=[{"role": "user", "content": user}], max_tokens=4096
                    )
                else:
                    response = ant_client.messages.create(
                        model=model, system=system,
                        messages=[{"role": "user", "content": user}],
                        max_tokens=4096,
                        temperature=temp
                    )
                return response.content[0].text
            elif vendor == "google":
                response = gem_client.models.generate_content(
                    model=model, contents=user,
                    config=types.GenerateContentConfig(system_instruction=system,
temperature=temp)
                )
                return response.text
        except Exception as e:
            if attempt == max_retries - 1: return f"ERROR: {str(e)}"
            time.sleep(2 ** attempt)
```

```
# =====
```

```

# 3. EXECUTION PIPELINES
# =====
def run_cogrithm_ultimate(vendor: str, model: str, query: str, data: str) -> dict:
    url = "https://api.cogrithm.com/v1/execute/ultimate"
    headers = {
        "Authorization": f"Bearer {COG_KEY}",
        "X-Provider-Model": model,
        "X-Provider-Key": get_provider_key(vendor),
        "Content-Type": "application/json"
    }
    payload = {"query": query, "data": data}
    start_time = time.time()

    try:
        with requests.post(url, headers=headers, json=payload, stream=True) as
response:
            response.raise_for_status()

            final_result = ""
            for line in response.iter_lines():
                if line:
                    decoded_line = line.decode('utf-8')
                    try:
                        chunk_data = json.loads(decoded_line)
                        if chunk_data.get("status") == "success":
                            final_result = chunk_data.get("result", "")
                        elif chunk_data.get("status") == "error":
                            return {"output": f"Engine Error:
{chunk_data.get('detail')}", "latency": round(time.time() - start_time, 2)}
                    except json.JSONDecodeError:
                        continue

            if not final_result:
                return {"output": "Error: Stream closed without success payload.",
"latency": round(time.time() - start_time, 2)}

            return {"output": final_result, "latency": round(time.time() - start_time,
2)}

    except Exception as e:
        return {"output": f"API Error: {str(e)", "latency": round(time.time() -
start_time, 2)}

def run_zero_shot(vendor: str, model: str, query: str, data: str) -> dict:
    start_time = time.time()
    user_prompt = f"CONTEXT:\n{data}\n\nQUERY:\n{query}"
    output = call_local_llm(vendor, model, SYS_ZERO_SHOT, user_prompt)
    return {"output": output, "latency": round(time.time() - start_time, 2)}

# =====
# 4. TRI-PANEL (SUPREME COURT) EVALUATION
# =====
def evaluate_with_panel(query: str, cog_out: str, zs_out: str, incumbent_model: str)
-> dict:
    # Calculate exact word counts natively in Python
    words_a = len(cog_out.split())
    words_b = len(zs_out.split())

    # Inject word counts as explicit metadata into the prompt
    user_eval = f"ORIGINAL PROMPT:\n{query}\n\n"
    user_eval += f"=== OUTPUT A (Word Count: {words_a}) ===\n{cog_out}\n\n"
    user_eval += f"=== OUTPUT B (Word Count: {words_b}) ===\n{zs_out}"

    panel_results = []

```

```

cog_votes, zs_votes = 0, 0

for eval_vendor, eval_model in EVAL_MODELS:
    if eval_model == incumbent_model:
        print(f"    [!] Recusing Judge: {eval_model} (Self-Preference Conflict)")
        continue

    raw_response = call_local_llm(eval_vendor, eval_model, SYS_JUDGE, user_eval,
temp=0.0)
    try:
        clean_json = raw_response.replace('```json', '').replace('```',
''.strip())
        data = json.loads(clean_json[clean_json.find('{'):clean_json.rfind('}') +
1])
        data['judge_name'] = eval_model

        if data.get("winner") == "A": cog_votes += 1
        elif data.get("winner") == "B": zs_votes += 1

        panel_results.append(data)
    except:
        panel_results.append({"judge_name": eval_model, "rationale": "Parsing
Error", "winner": "Error"})

    return {"panel_results": panel_results, "cog_votes": cog_votes, "zs_votes":
zs_votes}

# =====
# 5. MATRIX RUNNER
# =====
def run_whitepaper_matrix():
    print("="*80)
    print("🔥 INITIATING STRATEGIC EDGE-REASONING BATTLE ROYALE 🔥 ")
    print(f"TARGET: High-Density Strategic Arbitrage (Ultimate Tier)")
    print("="*80 + "\n")

    for fast_vendor, fast_model in FAST_MODELS:
        for front_vendor, front_model in FRONTIER_MODELS:
            if fast_vendor == front_vendor: continue

            print(f"\n[CHALLENGER: {fast_model} + Cogrithm] vs [INCUMBENT:
{front_model} Zero-Shot]")

            # 1. Execute
            cog_result = run_cogrithm_ultimate(fast_vendor, fast_model, TEST_QUERY,
TEST_DATA)
            print(f" [✓] Cogrithm Done ({cog_result['latency']}s)")

            zs_result = run_zero_shot(front_vendor, front_model, TEST_QUERY,
TEST_DATA)
            print(f" [✓] Zero-Shot Done ({zs_result['latency']}s)")

            # 2. Evaluate
            print(f" -> Firing Tri-Judge Panel (Supreme Court)...")
            eval_data = evaluate_with_panel(TEST_QUERY, cog_result["output"],
zs_result["output"], front_model)

            # 3. Print Output
            print("-" * 80)
            print(f"🗳️ CONSENSUS: {eval_data['cog_votes']} Votes for Cogrithm |
{eval_data['zs_votes']} Votes for Zero-Shot")

            if eval_data['cog_votes'] > eval_data['zs_votes']:
                winner_str = "Challenger (Cogrithm)"

```

```

elif eval_data['zs_votes'] > eval_data['cog_votes']:
    winner_str = "Incumbent (Zero-Shot)"
else:
    winner_str = "Tie"

print(f"🏆 MATCH WINNER: {winner_str}\n")

for judge in eval_data['panel_results']:
    print(f" [{judge.get('judge_name')}] Voted: {judge.get('winner')}")

    # Force the rationale to be a string to prevent textwrap TypeErrors
    full_rationale = str(judge.get('rationale', 'N/A'))
    wrapped_rationale = textwrap.fill(full_rationale, width=100,
initial_indent="    ", subsequent_indent="    ")

    print(f" Rationale:\n{wrapped_rationale}\n")

# 4. Conditionally Dump Full Output Texts
if winner_str in ["Incumbent (Zero-Shot)", "Tie"]:
    print("\n" + "=" * 32 + " OUTPUT LOGS " + "=" * 32)
    print(f"\n--- [ CHALLENGER: {fast_model} + Cogrithm ] ---")
    print(cog_result["output"])
    print(f"\n--- [ INCUMBENT: {front_model} Zero-Shot ] ---")
    print(zs_result["output"])
    print("\n" + "=" * 77)

    print("=" * 80)
    time.sleep(2)

if __name__ == "__main__":
    run_whitepaper_matrix()

```