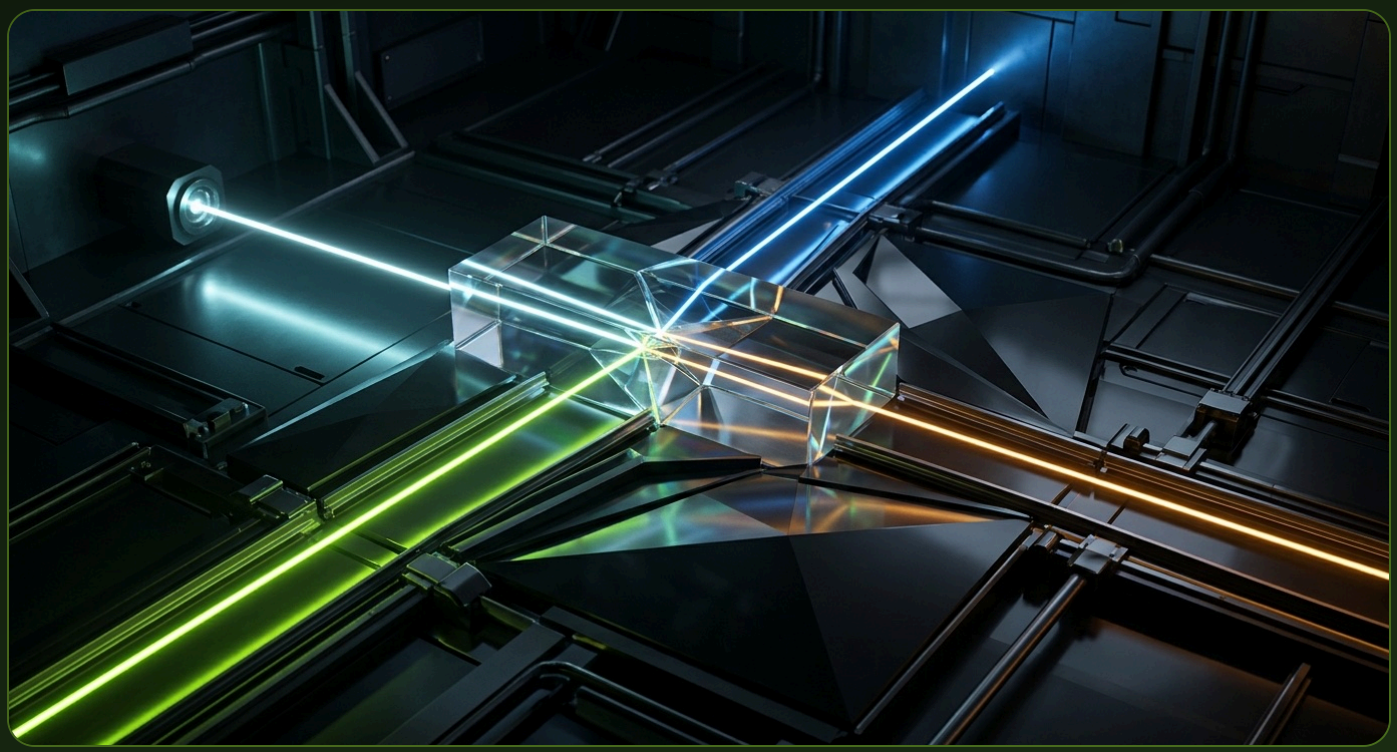
AGENT ENGINEERING SERIES • VERSION 1.0 • NOVEMBER 2026

Jev & System One Decisions

Stop asking a chat LLM for JSON and praying the parser holds. Send lean structured state and parallel typed questions, get calibrated probabilities in ~100ms, and let deterministic code make the call.

`Choice` • `Score` • `Noul` | RLCD Calibrated Probabilities | Parallel Speculative Fan-Out

Confidence-Gated Routing | `typesafe-sdk` & `@typesafe-ai/sdk` | LLM Guardrails & RAG Reranking



Judgment vs. Decision

Jev supplies calibrated probabilities; your ordinary `if` statements own every branch, tool call, and mutation.

One Call, Many Questions

Evaluate 10+ independent questions in parallel at near-constant latency—$0.042/M input tokens and $0 output tokens.

Uncertainty You Can Gate

Trained via RLCD so low-confidence edge cases escalate cleanly to humans or frontier models instead of hallucinating.

Why `JSON.parse()` Breaks & The System One Contract

Most production AI bugs live in the glue layer where you prompt a chat model for a paragraph and write brittle regex or schema parsers to extract a boolean or route key. **System One models** delete that parsing layer entirely.

Quick Navigation

1. System One Contract & Mental Model p. 2
2. Setup Paths, Gateways & Matrix p. 3
3. Annotated System One Inspector Tour p. 4
4. Three Primitives: Choice, Score, Noul p. 5
5. State Craft & Request Anatomy p. 6
6. Confidence & 4 Core Routing Patterns p. 7
7. SaaS Blueprint & One-File Policy p. 8
8. 5 Copy-Paste System One Recipes p. 9
9. Model Jaggedness & Hard Boundaries p. 10
10. System 1 + System 2 Synergy Pipeline p. 11
11. Troubleshooting & Pre-Flight Check p. 12
Core Law: Model Judges, Code Decides All

What Jev Is (And Why RLCD Beats RLHF for Software Control Flow)

1 Zero Text Generation

Jev (TypeSafe AI) never writes prose, explanations, or markdown fences. Every answer is strictly a probability distribution over options *you* defined.

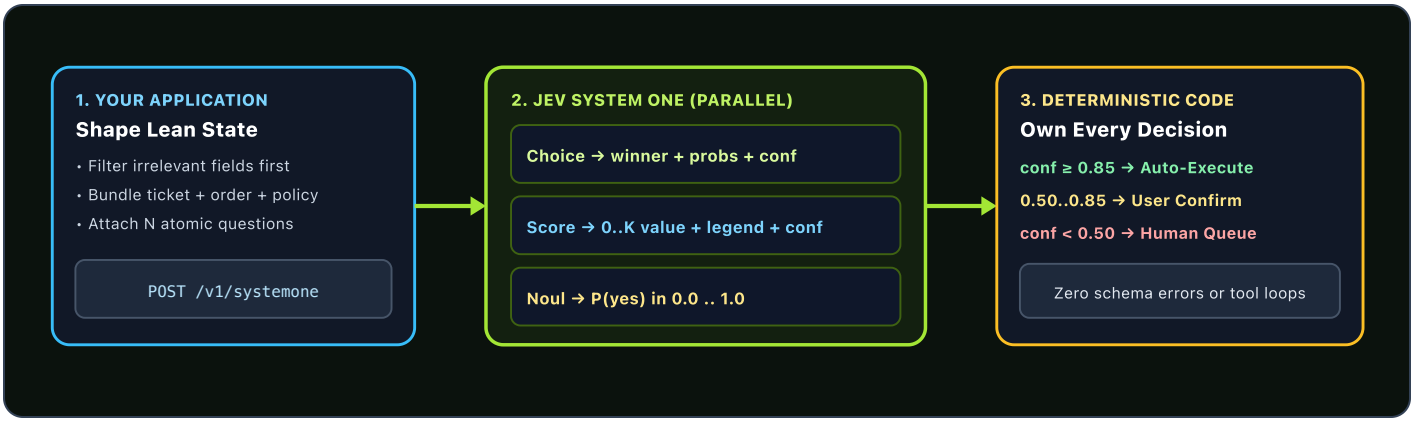
2 Trained With RLCD

Chat LLMs use RLHF (optimizing for pleasing prose). Jev uses **Reinforcement Learning for Calibrated Decisions** so `0.80` tracks an ~80% empirical hit rate.

3 Not a Coding Agent

Never set `jev-latest` as the brain of Claude Code or Cursor. Use your coding agent to write product code that calls Jev on hot request paths.

Architecture: Unstructured State In, Typed Probabilistic Judgments Out



The Core Separation of Powers

Jev supplies the judgment; your code supplies the decision. Because Jev is stateless, cannot call tools, and cannot emit unparsed text, a wrong prediction is just a bounded value you can gate on confidence—never a rogue agent loop.

Four Paths to First Decision & Architecture Matrix

Every client speaks the exact same JSON contract over `POST https://api.typesafe.ai/v1/systemone`. Start in the browser Playground in 60 seconds, then drop the Python or TypeScript SDK into your server routes.

1. Install Official SDKs, Coding-Agent Skill & Gateways

```
# 1. Server-side API key from https://console.typesafe.ai/keys (NEVER expose in browser/NEXT_PUBLIC_)
export TYPESAFE_API_KEY="ts_..."

# 2. Official Python (3.10+) & TypeScript/Node (20+) SDKs
uv add typesafe-sdk                 # or: pip install typesafe-sdk
npm install @typesafe-ai/sdk

# 3. Teach your coding agent (Claude Code, Cursor, Codex, OpenCode) Jev's exact API & patterns
claude plugin marketplace add typesafe-ai/skills && claude plugin install typesafe@typesafe-ai
npx skills add typesafe-ai/skills --skill typesafe-ai

# 4. Optional AI Gateways (confirm current model strings in gateway docs before pinning)
# Vercel AI Gateway: baseURL "https://ai-gateway.vercel.sh/typesafe", model "typesafe-ai/jev"
# OpenRouter:        base_url "https://openrouter.ai/api",            model "~typesafe/jev-latest"
```

2. Economics & Specs at a Glance (`jev-1.13.0` / `jev-latest`)

$ $42 / Billion Input Toks

Equals **$0.042 per 1M input tokens** (~$0.00002 for a 500-token ticket + 6 questions). **Output tokens are $0.00 (free).**

64k Context & Throughput

`64k` tokens per request (`32k` budget for `state` + longest question). Dynamic rate limits around 250k tok/s & 1,200 RPM.

0 Zero Training on Data

Text/JSON/arrays only (pre-OCR images first). Requests never train base weights; Enterprise plans add Zero Data Retention (ZDR).

3. Decision Matrix: Rules vs. Generative LLM vs. System One (Jev)

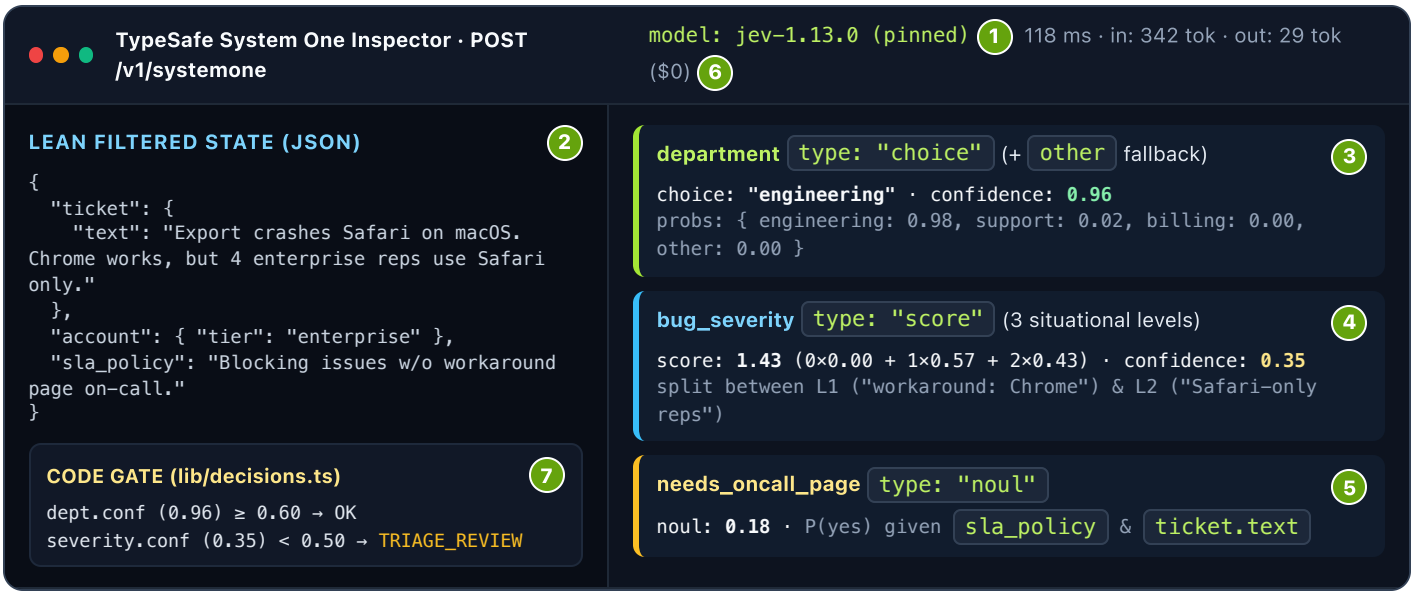
Dimension	Deterministic Code Rules	Generative LLM (JSON Mode)	Jev (System One Decision Model)
Return Contract	Exact primitive value	Generated prose/JSON you must validate	**Typed value + full distribution + `confidence`**
Sweet Spot	Math, dates, counts, exact regex/SQL	Drafting prose, multi-step synthesis, code	**High-frequency classification, rubrics & gates**
Unseen Edge Input	Silently misses or hits wrong branch	Often copes; occasionally invents fields/facts	**Spreads probability & lowers `confidence`**
Typical Latency	Microseconds (in-process)	1–8+ seconds (scales with output tokens)	**~70–500 ms** (vendor figure; verify per region)
Who Owns Flow?	Your code	Often delegated to an autonomous loop	**Strictly your code** (model never acts)

Security Alert: Verify Official TypeSafe Domains Before Sending Keys!

Several unaffiliated third-party lookalike domains (`thejevai.com` , `jevaimodel.net` , `jevmodel.net` , `bestjevai.com`) market themselves using the Jev name. Only send credentials to `https://api.typesafe.ai/v1/systemone` , `https://console.typesafe.ai` , and `https://docs.typesafe.ai` .

Annotated Tour: System One Playground & Inspector

Open `console.typesafe.ai/playground` to test state filtering and question criteria interactively before writing code. Here is how a single parallel request evaluates all three primitives at once.



Numbered Inspector Legend

1 **Pinned Version ID:** Exploring with `jev-latest` is fine, but production pins `jev-1.13.0` so tuned thresholds never drift on release day.

2 **Filtered Named State:** Passes only the exact fields questions read—keeping policy facts in `state` and out of hardcoded prompts.

3 **Choice Distribution:** Picks `"engineering"` across unordered options (up to 255) with an `other` safety valve and `0.96` peakedness.

4 **Probability-Weighted Score:** Lands between rubric levels (`1.43`) and exposes honest ambiguity (`confidence: 0.35`) when Chrome vs. Safari split.

5 **Binary Noul Probability:** Returns one number `0.0→1.0` representing `P(yes)`—no separate confidence field needed for 2 outcomes.

6 **Zero-Cost Output Billing:** All three answers evaluated in one parallel trip; you pay strictly for the `342` input tokens.

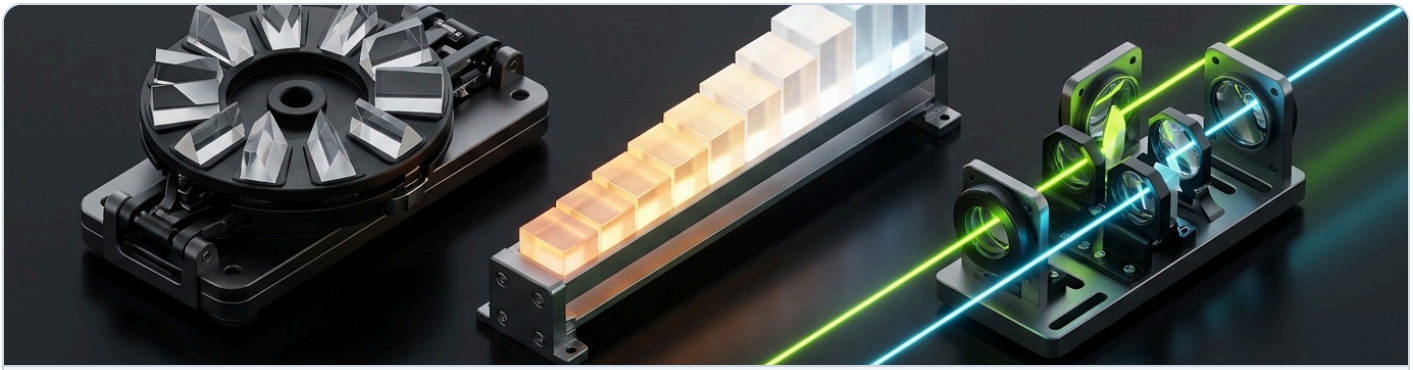
7 **Deterministic Policy Gate:** Because `bug_severity.confidence` is `0.35`, your TypeScript route escalates severity review to a human lead while routing straight to Engineering.

Notice Why Low Confidence Here Is a Feature, Not a Bug

A chat LLM forced into JSON mode picks `"severity": "P1"` or `"P2"` with fake 100% certainty. Jev shows a `57% / 43%` split because a workaround exists for the app (Chrome) yet not for Safari-locked users—flagging the exact ambiguity a human should arbitrate!

Mastering the Three Primitives: `Choice`, `Score` & `Noul`

Every judgment in System One maps to one of three shapes. Choose by the mathematical structure of the answer you need—and batch all of them into a single request.



Pick by Output Geometry: Use `Choice` for unordered sets (up to 255 keys), `Score` for ordered situational rubrics (2–10 levels), and `Noul` for atomic yes/no probabilities (`0.0–1.0`).

1 `Choice` (Unordered)

- Returns `choice`, `probabilities`, and `confidence` (up to **255 keys**).
- **Always include** `other: null` so out-of-scope inputs never force a bad match.
- Disambiguate close twins with `{what, not_for, examples}` objects.

2 `Score` (2–10 Levels)

- Returns `score` (`Σ i·p_i`), `legend`, `probabilities`, `confidence`.
- **Describe situations, never degrees:** write *"Broken, workaround exists"*, not *"Medium"*.
- Measure **one dimension** per Score; give extreme cases their own top level.

3 `Noul` (Binary `P(yes)`)

- Returns single float `noul` in `0.0..1.0`.
- **One condition per Noul:** split *"angry AND wants refund"* into two Nouls.
- Phrase positively (`1.0 = yes`). Note: `0.5` means *unsure*, never "medium skill"!

Upgrading Confused Options With Structured Criteria (Python SDK)

```
from typesafe_sdk import Choice, Noul, TypeSafeClient

with TypeSafeClient(model="jev-1.13.0") as client:  # reads TYPESAFE_API_KEY
    res = client.system_one(
        state="Has my returned pair of size-10 trail runners reached your warehouse yet?",
        questions={
            "intent": Choice(
                instructions="Which return workflow applies to this message?",
                criteria={
                    "return_policy": {"what": "Eligibility rules or starting a new return",
                                      "not_for": "Progress of a parcel already shipped back",
                                      "examples": ["Can I return worn shoes?", "How long is the return window?"]},
                    "return_status": {"what": "Tracking or refund timing of a return already sent",
                                      "not_for": "Asking whether an item is eligible to be returned",
                                      "examples": ["Did my return arrive?", "When will my refund post?"]},
                    "other": None,
                },
            ),
            "already_shipped": Noul(instructions="Does the customer state they already sent the item back?"),
        },
    )  # Cookbook result: 13 batched questions in 1 call ran 12.2x cheaper & 10.0x faster than 13 single calls!
```

State Craft: Why Input Hygiene Is 80% of Accuracy

Unlike chat sessions where teams dump entire database rows into context "just in case," Jev accuracy drops sharply when `state` contains irrelevant noise. Filter ruthlessly in code before calling the API.

Color-Coded Anatomy of a High-Signal System One Request

[1. PINNED MODEL] `"model": "jev-1.13.0"` — Locks exact weights so calibrated thresholds never shift unexpectedly.
[2. NAMED STATE OBJECT] `"state": {"ticket": {"msg": "..."}, "order": {"charges": [...]}, "refund_policy": "..."}` — Separates facts & policy from judgment logic.
[3. DOT-PATH POINTER] `"instructions": "Does refund_policy support the request in ticket.msg given order.charges?"` — Directs attention to exact fields.
[4. SITUATIONAL CRITERIA] `"criteria": {"true": "Explicit duplicate captured charge covered by policy", "false": "Single valid charge or outside policy"}`
[5. ESCAPE VALVE] `"other": null` on every `Choice` — Prevents forced high-confidence misclassifications on out-of-distribution inputs.
[6. ATOMIC DECOMPOSITION] Ask 3 simple questions (`refund_asked`, `duplicate_charge`, `policy_covers`) and combine with `&&` in code!

A Put Policy in `state`, Not Prompts

Pass your live `refund_policy` or plan entitlements inside `state`, and keep `instructions` generic (*"Does `refund_policy` cover this request?"*). When business policy changes tomorrow, zero question definitions or thresholds need rewriting.

B Structured Dict Instructions

`instructions` and `criteria` accept JSON objects as well as strings! Pass labeled candidate records inside each question to run 50-way entity deduplication or passage reranking in a single HTTP round trip.

Worked Pattern: Single-Trip Entity Deduplication Across Candidate Records

```
from typesafe_sdk import Noul, TypeSafeClient

def build_dedupe_questions(candidates: list[dict]) -> dict[str, Noul]:
    return {
        f"same_as_{c['id']}": Noul(
            instructions={
                "candidate_record": {"name": c["name"], "city": c["city"], "last_company": c["last_company"]},
                "question": "Is the incoming 'resume' in state for the exact same person as 'candidate_record'?",
            }
        )
        for c in candidates
    }

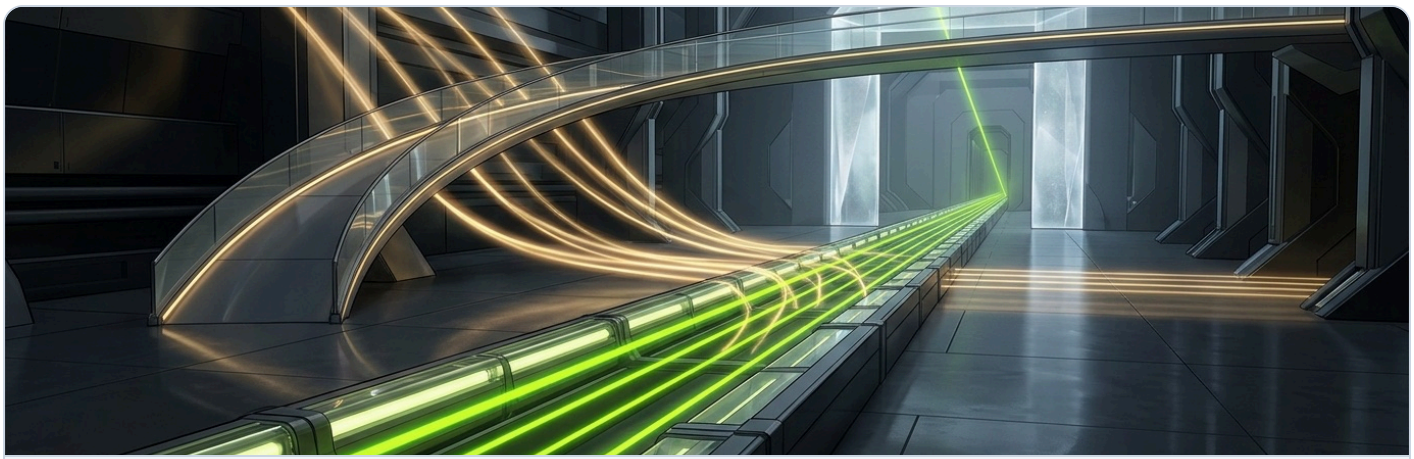
with TypeSafeClient(model="jev-1.13.0") as client:
    res = client.system_one(state={"resume": parsed_resume_text}, questions=build_dedupe_questions(top_25_rows))
matches = [cid for cid, ans in res.nouls.items() if ans.noul > 0.85]  # Zero prose, 1 round trip!
```

When to Make a Second Request Instead of Batching

Batch everything into one request *unless* Question B's option keys literally depend on Question A's winner (e.g., walking a 5,000-node taxonomy tree) or you must fetch an external database row first. Otherwise, speculative single-trip batching always wins.

Confidence Gating & The Four Core Patterns

Rank with probabilities; gate automation with confidence. Calibration applies across populations, not single calls—so risk-sensitive actions always pair answer values with explicit confidence floors.



Three-Band Risk Control: Never apply one global threshold. Showing an account balance tolerates `confidence ≥ 0.60`; moving money demands `> 0.90` plus confirmation; `< 0.50` routes straight to a human.

The Four Architectural Patterns That Cover 95% of Production Workloads

Pattern	How It Works in Code	High-Leverage Use Case
1. Speculative Fan-Out	Send all candidate questions in one call (`category`, `bug_severity`, `refund_asked`); ignore unused keys on non-matching branches.	Support & intake pipelines avoiding serial waterfall latency.
2. Confidence-Gated Routing	Branch on `(answer, confidence)` pairs: auto-act above high bar, ask user to confirm in middle band, escalate below floor.	Refunds, transfers, account mutations, autonomous tool gates.
3. Composite Scoring	Ask 3–5 atomic `Score`/`Noul` dimensions, normalize each to `0..1`, and compute weighted sums in code (`0.4*py + 0.4*arch...`).	Lead prioritization, candidate rubrics (IC vs. EM weights from 1 call), spam risk.
4. Intent Routing	Place Jev at the front door to pick between a $0 SQL handler, a cheap small LLM, a frontier reasoning model, or a human specialist.	Slashing frontier LLM bills by 60–80% without losing quality on hard queries.

Composite Scoring + Three-Band Routing Snippet

```
# Tune weights in Git PRs without touching prompts — reuse identical atomic answers across multiple policies!
spam_risk = (
    0.45 * answers["requests_credentials"].noul
    + 0.30 * answers["sender_identity_mismatch"].noul
    + 0.25 * answers["unexpected_reward"].noul
)
if spam_risk >= 0.70:
    quarantine_message(msg_id)          # High risk -> deterministic quarantine
elif spam_risk >= 0.35:
    enqueue_human_review(msg_id)        # Middle band -> human moderator queue
else:
    deliver_to_inbox(msg_id)            # Low risk -> fast delivery
```

Production Blueprint: One-File Policy & Quality Gates

Keep every question definition and numeric threshold inside **one dedicated policy file** (`lib/decisions.ts`) so product and security reviews read a single 60-line file instead of hunting across route handlers.

1. Centralize Questions & Thresholds (`lib/decisions.ts`)

```
import { choice, noul, score } from "@typesafe-ai/sdk";

export const INTAKE_QUESTIONS = {
  request_type: choice("What is the user asking for?", {
    new_report: "Generate a new competitive audit report", edit_report: "Modify an existing report",
    billing: "Invoices, refunds, or payment failures",     support: "Product bug or how-to question", other: null,
  }),
  is_abusive:       noul("Does the message contain threats, slurs, or harassment?"),
  refund_requested: noul("Is the user explicitly asking for money back or account credit?"),
  urgency:          score("How time-sensitive is this request?", [
    "Can wait indefinitely", "Needed this week", "Needed today", "Production outage or active revenue loss",
  ]),
};

export const QUALITY_QUESTIONS = {
  rubric_fit: score("How completely does 'report' satisfy the delivery rubric?", [
    "Empty or off-topic", "Partial sections; misses core ask",
    "Answers ask but thin evidence", "Complete, specific, and thoroughly evidenced",
  ]),
  claims_supported: noul("Is every factual claim in 'report' supported by a cited passage in 'sources'?"),
};

export const THRESHOLDS = {
  minIntakeConfidence: 0.60, refundToHuman: 0.70, minRubricScore: 2.20, minCitationSupport: 0.75,
} as const;
```

2. Pre-Delivery Quality Gate Around Paid Deliverables (`lib/quality-gate.ts`)

```
import { TypeSafeClient } from "@typesafe-ai/sdk";
import { QUALITY_QUESTIONS, THRESHOLDS } from "@/lib/decisions";

const client = new TypeSafeClient({ model: "jev-1.13.0" }); // Server-only TYPESAFE_API_KEY

export async function gatePaidDeliverable(report: string, sources: string[]) {
  const { answers } = await client.systemOne({ state: { report, sources }, questions: QUALITY_QUESTIONS });

  if (answers.rubric_fit.confidence < 0.50) {
    return { action: "human_review" as const, reason: "ambiguous_rubric_fit" };
  }
  if (answers.rubric_fit.score < THRESHOLDS.minRubricScore ||
      answers.claims_supported.noul < THRESHOLDS.minCitationSupport) {
    // Never bill a customer for a hallucinated or incomplete LLM draft!
    return { action: "retry_draft" as const, fit: answers.rubric_fit.score, cited: answers.claims_supported.noul };
  }
  return { action: "ship" as const };
}
```

Observability Habit: Log the Replay Tuple, Mind Debug Bodies

Persist `{model, question_version, answers, confidence, chosen_action}` to Postgres so you can replay threshold changes offline. Note that setting `TYPESAFE_LOG_LEVEL=debug` redacts auth headers but logs raw request/response bodies—keep debug off in PII environments!

5 Real-World Copy-Paste System One Recipes

Drop these five patterns into your Python or TypeScript stack to replace dedicated cross-encoders, binary moderation walls, and flaky LLM-judge parsers.

Recipe & Goal	Copy-Paste Production Pattern
1. Zero-Service RAG Reranker Filter noisy vector chunks in 1 call	Build one `Noul` per retrieved chunk: `q = {f"p{i}": Noul(instructions={"query": query, "passage": c.text, "q": "Does passage help answer query?"}) for i, c in enumerate(chunks)}`. Keep chunks with `ans.noul > 0.60` sorted descending.
2. 4-Outcome LLM Safety Battery Route `pass`, `review`, `block`, `support`	Screen inputs/outputs with parallel Nouls (`jailbreak`, `harmful_request`, `medical_dosage`, `self_harm`) + 1 harm `Score`. Route `self_harm > 0.5` to crisis resources (`support`), mild medical asks to `review`, and high-severity exploits to `block`.
3. Atomic Tool-Call Verifier Catch bad agent args before execution	Before executing a high-impact agent tool call, send `state={"user_ask": ask, "proposed_call": call}` with atomic Nouls: `right_tool`, `matches_user_scope`, `units_consistent`, `no_invented_ids`. Require `all(n > 0.85)` to run.
4. Fast CI Eval Suite Scorer Deterministic exit-code PR gate	Validate Jev on 20+ human-labeled JSONL rows first; then gate PRs in milliseconds: `passed = sum(client.system_one(c["state"], QS).choices["intent"].choice == c["expected"] for c in rows)` → `sys.exit(0 if passed / len(rows) >= 0.90 else 1)`.
5. Honest LLM A/B Adapter Benchmark Jev vs. GPT/Claude/Gemini	Run identical questions against frontier LLMs on *your* dataset before committing: `pip install 'system-one-adapter[openai,anthropic,gemini]'` → `SystemOneAdapterClient(structured_outputs=True, llm_answer_mode="probabilities").system_one(..., provider="openai", model="gpt-4o-mini")`.

Why Four Guardrail Outcomes Beat Two

Binary `allow / block` walls frustrate legitimate edge users and fail vulnerable ones. Caching one Jev probability vector lets you apply `strict` vs. `permissive` product policies in pure code!

Calibrate Before You Gate CI

Being fast and cheap never guarantees domain alignment. Always measure Jev's agreement against 20–50 human-labeled rows on your own traffic before letting it block pull requests or customers.

Know the Edges: `jev-1.13` Jaggedness & Hard Rules

TypeSafe publishes explicit failure modes ("jaggedness") for `jev-1.13`. Respecting what Jev *cannot* do is what separates rock-solid production pipelines from mysterious accuracy drops.

The 7 Documented Jaggedness Traps (And Exact Code Fixes)

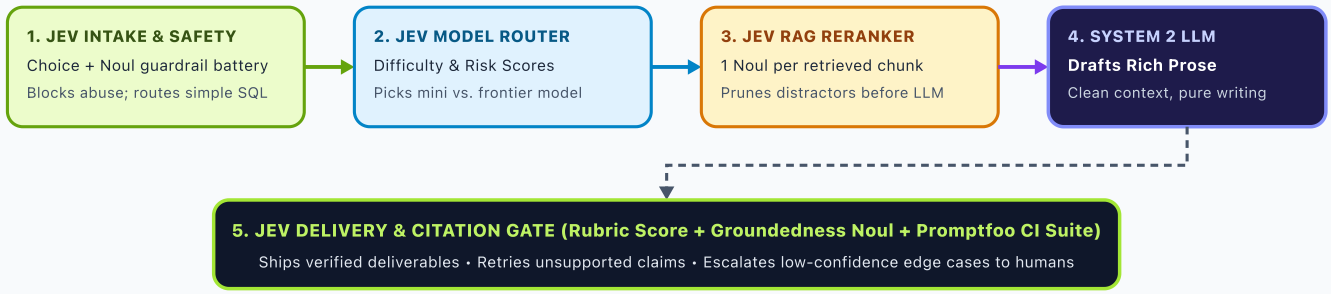
Jaggedness Trap	What Goes Wrong	The Clean Engineering Pattern
1. Counting & Arithmetic	Asking *"Are more than 5 of these 40 items defective?"* fails—Jev is not a calculator.	Fan out **40 parallel `Noul` questions** (one per item) in one call, then run `sum(p > 0.7 for p in nouls) > 5` in code!
2. Date & Time Ordering	Reads dates as text tokens rather than ordered timestamps (*"Is Date A 30 days before B?"*).	Extract date fields with `Choice` (month/day/year + `not_stated`) or regex, then compare with native `datetime`.
3. Hyper-Literal Reading	Answers the exact words written, not implied business intent.	State boundary conditions explicitly inside `criteria`; align `instructions` and `criteria` so they never contradict.
4. Double Negatives	Multi-hop indirection or *"Is it untrue that X lacked Y?"* degrades calibration.	Ask direct, single-hop positive questions pointed at named `state` paths.
5. Cross-Question Sums	`Noul(X) + Noul(not X)` is not constrained to equal `1.00` across independent questions.	Never rely on cross-question probability identities; use a single `Choice` if options must sum to `1.0`.
6. Adversarial State	Prompt injections inside untrusted user text can nudge probabilities.	Strip untrusted markup, write explicit criteria, and never let low/medium confidence trigger destructive actions alone.
7. Alias Drift (`jev-latest`)	New model releases move probabilities slightly against tuned thresholds.	Pin `model="jev-1.13.0"` in production and log `response.model` on every decision trace.

Worked Fix: Counting Defects Without Asking the Model to Count

```
# WRONG: Noul("Do at least 3 reviews mention shipping damage?") -> unreliable mental math!
# RIGHT: 1 atomic Noul per review in a single batched call + deterministic Python sum:
questions = {
    f"dmg_{i}": Noul(instructions={"review": r, "q": "Does 'review' report physical shipping damage?"})
    for i, r in enumerate(reviews)
}
resp = client.system_one(state={"product_sku": sku}, questions=questions)
damaged_count = sum(1 for ans in resp.nouls.values() if ans.noul >= 0.75)
```

Cross-Tool Synergy: The System 1 + System 2 Sandwich

Kahneman's *Thinking, Fast and Slow* becomes a concrete software architecture when you wrap fast, sub-cent System One judgments (Jev) on **both sides** of slow, creative System Two reasoning (Claude, GPT, Gemini).



How Each Tool Plays to Its Exact Strength

Pipeline Stage	Engine Assigned	Why This Division Wins on Cost, Latency & Reliability
Intake & Guardrails	Jev (`Choice` + `Noul`)	Screens jailbreaks and routes `order_status` to $0 SQL handlers in ~100ms before waking any generative LLM.
Complexity Routing	Jev (`Score` × 2)	Sends 75% of routine tasks to fast mini models, saving frontier reasoning tokens strictly for reconciled hard evidence.
Passage Filtering	Vector DB + Jev `Noul`s	Drops irrelevant vector chunks so the System 2 writer sees only crisp supporting evidence (higher faithfulness, lower input bill).
Synthesis & Code	Frontier LLM (System 2)	Freed from awkward JSON classification duties to do what LLMs do best: nuanced prose, code edits, and synthesis.
Outbound Verification	Jev + Deterministic Code	Grades rubric completeness and per-claim citation support in one parallel check before charging or showing the customer.

The Recommended 30-Day Rollout Order

Week 1: Test real tickets in Playground & ship one read-only intake classifier logged beside existing code. **Week 2:** Collect 30 human labels & tune `lib/decisions.ts` thresholds. **Week 3:** Enable confidence-gated routing + paid-deliverable quality gate.

Troubleshooting Clinic & Pre-Flight Checklist

Keep this page next to your `lib/decisions.ts` policy file. Nearly every surprising probability or API error maps to one of seven mechanical fixes below.

Top 7 System One Pitfalls & 10-Second Fixes

Symptom You Observe	Root Cause & Immediate 10-Second Fix
1. `confidence` stays stuck below `0.40`	Overlapping rubric levels, compound multi-part question, or missing facts in `state`. Split into 1 dimension per question and rewrite `Score` levels as concrete situations.
2. Model confuses two similar `Choice` keys	Short string labels blur boundary cases. Upgrade both keys to structured objects with explicit `what`, `not_for`, and `examples` arrays, plus `other: null`.
3. Easy cases fail when full JSON row is passed	Context rot from noisy distractor fields. Filter `state` in code first so only fields cited by your questions cross the wire, and reference exact dot-paths.
4. Counts, totals, or date checks are wrong	Asking Jev to do math or calendar ordering. Emit **one `Noul` per item** (or extract date parts) and run arithmetic/date comparisons in native Python/TypeScript.
5. Threshold pass rates shifted without code edits	Using floating alias `jev-latest` across a model update. Pin exact version `jev-1.13.0` in production and re-calibrate on your golden JSONL set before bumping.
6. HTTP `401`, `422`, or `429 / 529`	`401`: internal whitespace or wrong `TYPESAFE_API_KEY` (verify `api.typesafe.ai` domain!). `422`: inspect body for malformed criteria field. `429/529`: enable SDK exponential backoff.
7. Coding agent invents non-existent SDK methods	Outdated training priors or stale skill. Run `npx skills add typesafe-ai/skills --skill typesafe-ai` (or update plugin) so the agent reads current SDK signatures.

6-Point Pre-Flight Checklist: Before Shipping Any System One Decision

☐ **1. Server-Only Key & Verified Domain:** `TYPESAFE_API_KEY` lives strictly on the server talking to `api.typesafe.ai` (never browser code or lookalike URLs).

☐ **2. One-File Policy & Pinned Model:** All questions and risk-tiered `THRESHOLDS` live side-by-side in one file with `model` pinned to `jev-1.13.0`.

☐ **3. Atomic & Escape-Hatched Primitives:** Every question tests one condition, `Noul`s read positively (`1 = yes`), and every `Choice` includes `other: null`.

☐ **4. Ruthlessly Filtered State:** Code strips unrelated fields before calling `system_one()`; zero math, counting, or date comparisons inside instructions.

☐ **5. Confidence Fallbacks Wired:** Low-confidence predictions route explicitly to human review, user confirmation, or retry—never blind execution.

☐ **6. Validated on Labeled Traffic:** Calibrated against 20–50 real labeled examples (and benchmarked via `system-one-adapter`) before gating users or CI!