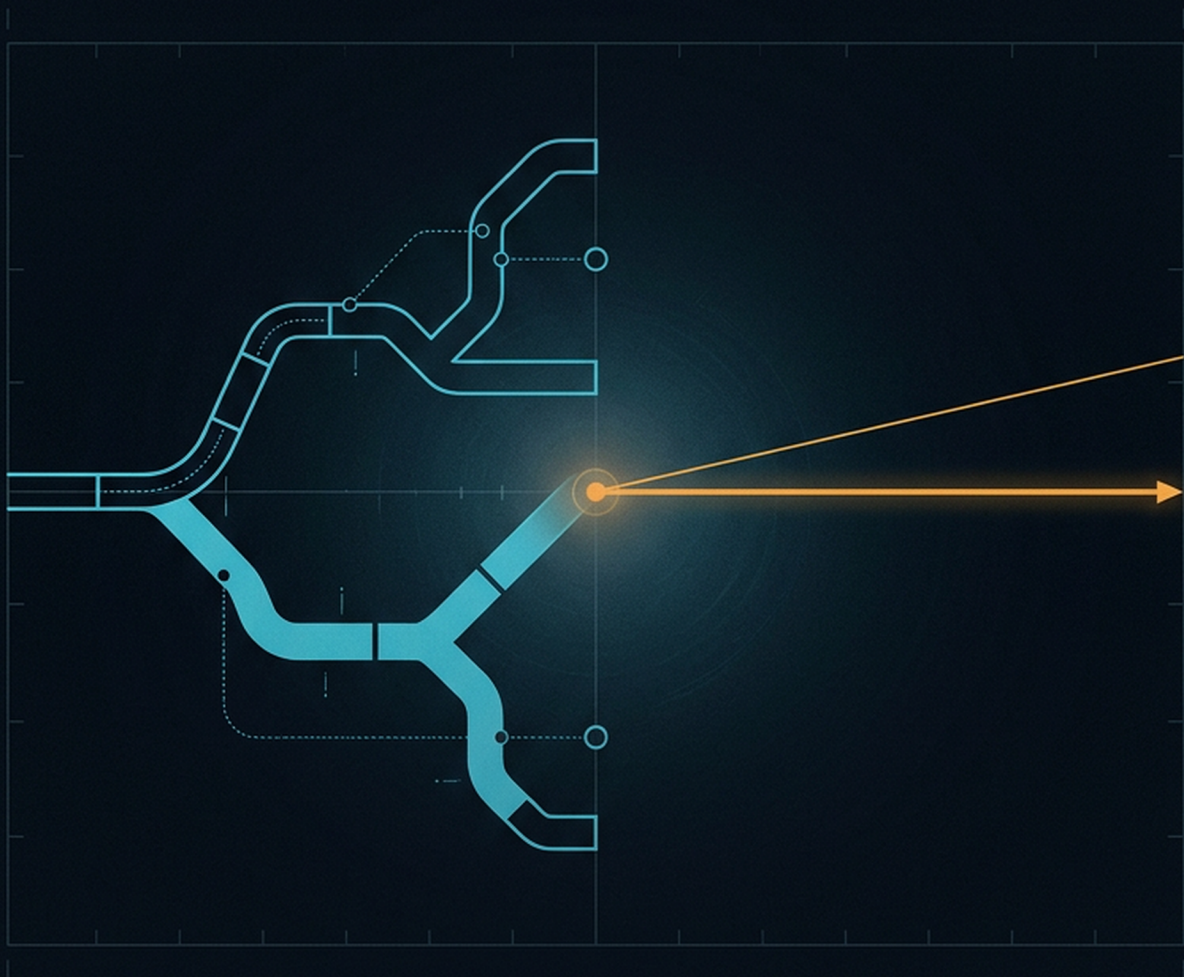


GUIDE TO

DECISION MODELS



The JEV-like family: what it is, how it is used,
and how to run, deploy and train local variants

System 1 & System 2 · Quantization & serving · LoRA / QLoRA

GROUND TRUTH BOOKS

Guide to Decision Models

What the “JEV-like” family is, how it is used, and how to run, deploy, and train local variants

Ground Truth Books

This book is available at <https://leanpub.com/guide-to-decision-models>

This version was published on 2026-10-10



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2026 Ground Truth Books

Contents

- Introduction 1**
 - Problem this book solves 1
 - Who this book is for 2
 - How to read this book 3
 - Pinned versions and reproducibility 4

- Part I: Foundations 7**
- What a Decision Model Is 8**
 - Typed answers instead of text 8
 - Single forward pass decisions 9
 - State and typed questions 10
 - Confidence and calibration 12
 - The JEV name and Clef 13
 - What decision models are not 14

- System 1 and System 2 17**
 - Dual-process theory as architecture 17
 - Reflexive versus deliberative routing 17
 - Where decision models win 17
 - Where deliberation is still required 17

- Part II: Using Decision Models 18**
- Task Taxonomy and Budgets 19**
 - Classification and intent detection 19
 - Routing and tool selection 19
 - Scoring and safety checks 19
 - Latency and throughput budgets 19
 - Cost per decision 19

CONTENTS

Decision Models in Agent Stacks	20
Decision models inside agent loops	20
Routers and cascades	20
Tool and skill selection	20
Confidence as an operating policy	20
Extraction and structured output	20
The Local Variant Catalog	21
Ollaya's listed models	21
Typed-decisions scores	21
Licenses and commercial use	21
Adjacent decision backbones	21
Multilingual checkpoints	21
Choosing a checkpoint	21
 Part III: Running and Deploying Locally	 23
Running Local Inference	24
The local loop	24
Loading a checkpoint	24
Your first choice question	25
Scoring a candidate answer	27
Reading the confidence value	27
Reproducing a quickstart	28
Debugging bad inputs	29
Quantized Formats and Exports	32
GGUF quantization tiers	32
Converting and publishing GGUFs	32
ONNX and runtime export	32
Core ML and MLX targets	32
Choosing a quant level	32
Serving with Ollaya	33
Pulling and serving models	33
Modelfiles for decision models	33
The Ollaya MCP server	33
ONNX Runtime versus llama.cpp	33
kev.serve on CUDA and MLX	33

- Serving with the TypeSafe API** 34
 - Typed questions over HTTP 34
 - The wire API shape 34
 - Using the typesafe_sdk client 34
 - Pointing clients at local endpoints 34
 - OpenAI's Decisions API 34
 - Gateways and drop-in compatibility 34

- Part IV: Training and Evaluating** 36
 - Building Decision Datasets** 37
 - Intent corpora and benchmarks 37
 - Synthetic decision triples 37
 - Few-shot and SetFit sets 37
 - Multilingual decision data 37
 - Label quality and splits 37

 - Fine-Tuning with LoRA and QLoRA** 38
 - Parameter-efficient fine-tuning basics 38
 - LoRA and QLoRA mechanics 38
 - Training with Unsloth 38
 - Training with Axolotl and TRL 38
 - Training a decision head 38
 - Merging and exporting adapters 38

 - Evaluation and Calibration** 40
 - Decision accuracy versus schema validity 40
 - Calibration and ECE 40
 - The typed-decisions benchmark 40
 - The Bespoke Labs benchmark 40
 - Building an eval harness 40
 - Regression tests in CI 40

 - Conclusion** 42
 - What you can now do 42
 - Choosing your fast path 42
 - Where to go next 42

 - Appendix A: Command and flag reference** 43

Serving	43
The Ollaya CLI	43
Environment variables	44
Training, calibration and evaluation	44
Appendix B: Pinned versions register	45
What the register records	45
Checkpoints: Kev 1.0	45
Runtimes and serving builds	45
Wire-format pins	45
Benchmark pins	45
Dataset and gold pins	45
Training pins	46
License pins	46
Appendix C: Quantization format reference	47
Precision formats	47
GGUF and the GGML block schemes	47
What quantization costs a decision model	47
Training-time formats	47
Host formats and hardware quants	47
Format by deployment	47
Glossary	49
References	50

Introduction

Problem this book solves

A card transaction arrives, and your system has to decide whether it looks fraudulent and whether it needs a human review [1]. Calls like that are most of what AI does inside production software [2, 3], and teams make them with a chat model because that is the tool they have wired up.

First, the chat model generates. It predicts one token at a time, then uses what it has already produced to predict the next, so even a short answer such as `{"team": "billing"}` arrives over several generation steps [4]. Your code parses the string and validates the shape, and you pay for the output tokens on top of the input [5]. Second, it takes seconds to minutes where a decision model answers in 70 to 500 milliseconds [5]. Third, when the string is malformed or holds a value outside your list, you need defensive logic to contain it [6], and none of that work improves the decision.

A decision model removes the generation step. It reads a state and a set of typed questions and returns a value from your schema with a calibrated probability, computed in a single non-autoregressive pass [7]. TypeSafe named the category System One and shipped Jev as its first public model, describing it as “a frontier-intelligence function call: unstructured state in, typed probabilistic decisions out” [8]. The API has three question types: `choice` picks one option from a list, `score` places the state on a described scale, and `noul` answers yes or no with one probability [9]. Output is priced at zero, because the model does not generate a token stream; input runs at \$0.042 per million tokens [5].

Open models followed. Kev is a family you can download and run, built on Qwen backbones and sized from 0.8B to 27B [10]. Laya pairs an encoder with a decision head and answers in one forward pass [11]. Clef is Cloudflare’s open-weight model with the same request format as Jev [12]. Practitioners and papers call the family “JEV-like” [13, 14], a name that describes a shape rather than a vendor. They differ in size and backbone, and in what their licenses let you ship.

That variation is the problem this book solves. Knowing a decision model exists does not tell you which checkpoint you are allowed to ship, how to serve it offline, or how to show that a local variant is good enough to replace the generative call. The evidence sits in launch posts and model cards, and much of the performance data comes from the vendors who built the models. This book assembles the family into something you can run.

Why run one locally instead of calling hosted Jev? TypeSafe hosts a single version, with rate limits it can change without notice and no SLA, and the state you send leaves your perimeter on every call [2]. An open checkpoint runs inside that perimeter, so prompts never leave your network, and it keeps answering when a provider pauses signups, throttles traffic or has an outage, with no rate limit but your own hardware [15].

The claim is deliberately narrow. Routine decisions do not need a generative model. When you can write the acceptable answers down before you ask the question, a decision model returns one of them faster and cheaper than a chat model, and it cannot return a value outside the set you supplied [5, 3]. You can run one on your own hardware, serve it behind a TypeSafe-compatible endpoint, and train and measure it on your own labeled examples.

Who this book is for

You are an applied ML engineer or a technically fluent builder shipping models that run offline or on-premises. You work in Python from a terminal, and you have built or deployed a container before. You do not need to have used a decision model, and you do not need to have called Jev.

You do need to be comfortable reading a `curl` request and a `Modelfile`. The examples are commands you run at a terminal, and the chapters assume you will type them. Installing Kev, for instance, means cloning the repository and syncing with `uv`, then starting a server from the Python module [10]:

```
1 git clone https://github.com/jaredpalmer/kev.git && cd kev
2 uv sync --extra serve
3 uv run --extra serve python -m kev.serve --run jaredpalmer/kev-4b --port 8009
```

That block starts Kev-4B on your machine, on CUDA or ROCm if you have a GPU and on MLX if you have Apple Silicon [10]. Chapter 7 walks through what it returns and why.

You will not find a from-scratch derivation of transformer attention here. The book defines its notation where it appears. When a chapter says “a non-autoregressive forward pass”, it means one pass over the input that produces every position’s probability distribution at once, with no generation loop [16]. That one fact is enough to follow the architecture discussion.

TypeSafe has not published Jev’s architecture or weights, so this book treats it as a hosted service you compare against [17]. TypeSafe also describes its own speed and cost multiples as the high end of real-world gains, so the book reports them as claims to test on your own workload [18].

How to read this book

Fourteen chapters, four parts between this introduction and the conclusion. Part I (Chapters 2 and 3) defines the thing. Part II (Chapters 4 through 6) puts it to use and surveys the local variants. Part III (Chapters 7 through 10) runs and serves it. Part IV (Chapters 11 through 13) trains and measures it. Read Part I in order even if you already run a clone, because the type-safety guarantee is narrower than the marketing and chapter 2 says which half of it you can rely on [3].

The definitions come first. A decision model answers a typed question, and the JEV name and the no-hallucination claim only make sense once that shape is clear. System 1 locates the model in a dual-system architecture, which tells you which requests to route to a fast model and which to escalate to a deliberative one [17, 2]. Chapter 4 covers the task shapes that suit a bounded decision: classification, routing, scoring, safety checks [9, 2]. Chapter 5 places the decision model in an agent stack as the fast path before generation. The catalog in Chapter 6 compares local families by size and backbone, and by the license that usually decides what you can ship [10, 19]. It also reports each family’s typed-decisions score, measured on a 2,000-decision benchmark where Laya’s authors report 0.766 for their fine-tuned `laya-typed-decisions` checkpoint against Jev’s published 0.727, while the base English Laya checkpoint scores 0.362 [11].

Then you run one. Chapter 7 gets a first local decision out of a checkpoint in minutes. Chapter 8 turns quantization and export format into a hardware decision among GGUF, ONNX, Core ML, and MLX. Chapters 9 and 10 stand up the two serving surfaces: Ollaya, which gives decision models the pull, serve

and run workflow Ollama gives LLMs, with Modelfiles and an MCP surface, running ONNX models on ONNX Runtime and GGUF models on llama.cpp, so the backend follows the model’s format [20], and the TypeSafe wire API, which the same client drives whether the model is hosted, self-served, or deployed behind a compatible endpoint [21].

Part IV ends with numbers on your own workload. Chapter 11 builds state-question-answer datasets. Chapter 12 adapts a small backbone with LoRA or QLoRA. Chapter 13 measures accuracy, calibration, and regression behavior before you trust the result, because a typed answer is not automatically a correct one [3].

The examples that follow all answer the same request: a support ticket that needs three decisions. Which team should handle it? Does it need urgent human attention? How frustrated is the customer? They are one choice, one noun and one score in a single request, the request the Kev README uses in its own curl example [10].

Pinned versions and reproducibility

You cannot reproduce a decision unless the recipe names the checkpoint revision, the fitted temperature, and the serving build that produced it. Kev’s README records a Hub revision and a fitted temperature for each checkpoint, and the Kev 1.0 release carries the same `v1.0` tag on every Hub repo, so `--run jaredpalmer/kev-4b@v1.0` loads the same weights as revision `139fdd94` each time [10, 22]:

Model	Hub revision of the weights	Temperature
Kev-0.8B	9a45d25e	2.35
Kev-4B	139fdd94	2.41
Kev-9B	b5d8c18e (v2)	2.19
Kev-27B	28be62e9 (v2, full weights)	1.32

Two checkpoints with the same name and different revisions can return different probabilities for the same ticket.

Ollaya publishes small ONNX graphs, about 3 MB each, and reads the original weight files from the author’s Hugging Face repository at a pinned commit,

verified by sha256 [20]. Kev's coding-agent skill clones its repository at a pinned commit and records `kev_ref` in `result.json` for each run [23]. OpenJev's release manifest binds every artifact with a SHA-256 digest and warns that the evaluated checkpoint directory digest and the packaged inference digest are not interchangeable, so the binding is checked per inference file [24]. The artifacts differ, and each pin gives another engineer what they need to fetch the same file.

Pin the serving stack too. A deployment guide for `llama.cpp` advises against a floating `:latest` tag in production and recommends a pinned build tag such as `:b10154-cuda`, because the server HTTP API and CLI flags change between builds [25]. For training, log the exact TRL and Transformers versions, fix the random seeds for every library in the run, and record dataset versions or hashes, since an unpinned dataset can change under you [26]. Kev's evaluation harness goes further: each number in `result.json` carries the suite hash and the code hashes for the run, plus the git commit [27]. The independent Jev benchmark does the same: it pins `jev-1.13.0` for all 346,009 requests and caches every raw response in a content-addressed store, so each of its tables can be recomputed offline [28].

This book follows the same rule. Where a fact depends on a version, the version is named. Jev's measurements here come from the hosted model whose responses report the version `typesafe/jev-1.13-20260917` [14]. That string is the name a gateway such as OpenRouter returns in its response; you request the model as `jev-1.13.0` from TypeSafe's own API and as `typesafe/jev-1.13` from OpenRouter [29, 30], and TypeSafe publishes failure modes per version, so 1.13's known weaknesses are not assumed to be the next version's [2]. Kev's numbers are for the family release Kev 1.0, which the README says trains nothing new. It fixes the checkpoints, cards and serving code the next generation will be compared against [10].

A confidence cutoff tuned on one checkpoint or one hosted model does not carry to another, because the calibration and the confidence formula differ [31, 23], so measure on the workload you will actually run. A pinned dependency stops changing under you, and left unmanaged it accumulates known, unpatched vulnerabilities [32]. Decide your update path at the same time you pin. Pydantic AI's TypeSafe documentation makes the same point from the other side: `jev-latest` is an alias that moves when TypeSafe ships a release, so once you have tuned a threshold, pin the version it was tuned against, `jev-1.13.0`, and move deliberately [29].

That is the discipline the rest of the book assumes. Each recipe names its checkpoint revision and its serving build, so a decision can be reproduced and, when it cannot, the input that changed is visible. Chapter 2 starts with the typed question those pins exist to keep stable.

Part I: Foundations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

What a Decision Model Is

Typed answers instead of text

A ticket lands in your support queue, and your code has to route it to billing, technical support, sales, or an other bucket, decide whether the customer is signalling urgency, and gauge how frustrated they are. A chat model can answer all three questions, but it answers in prose. You ask for JSON, you usually get JSON, and your code still parses it, validates it, and handles the runs where the model explained itself instead of returning the object.

A decision model changes the shape of the answer. It returns a value drawn from a schema you wrote before the call, and no other value. Jev, TypeSafe’s first public decision model, exposes three question types, and every call is built from them [5, 33].

Type	The question it answers	What comes back
choice	Which of these options?	The chosen option, a probability for each option, and a confidence value
score	Where on this rubric?	A probability-weighted value across your levels, the legend, per-level probabilities, and a confidence value
noul	Is this true?	One number from 0 to 1

The table follows the API reference and TypeSafe’s own descriptions [34, 35]. A choice list holds up to 255 options [36, 33]. A score rubric is an ordered array of at least two and at most ten level descriptions, and the returned value can sit between levels rather than snapping to one [34, 33]. A noul returns the probability that the statement is true, and no more than that [34, 37].

The guarantee rests on how the model answers. TypeSafe publishes neither Jev’s architecture nor its weights, so what follows is the documented behaviour

together with the open reproductions rather than a reading of Jev’s own code [38, 36]. The model scores every option in the list you sent and returns the highest, so the reachable answer space is exactly that list [5]. A choice with three options yields three probabilities summing to 1, and a fourth option has no score to win with [5]. An independent reproduction fits that shape, though it ran on open Qwen2.5 models and made no Jev API calls: scoring worked whether the model read the full option text or a one-token code for each option [36].

That is what “type-safe” means in practice. If your only choices are `billing`, `technical_support`, and `other`, the model cannot return a fourth department, and it cannot return a sentence in which a fourth department might be hiding [4]. Constrained decoding in an LLM asks for the same property, but it enforces it during generation; here the answer space is fixed before the model runs [39].

The model’s vocabulary is the list you send, so that list is the thing to write carefully. Name every value the answer may take, and describe what belongs to each one. Everything downstream of the call depends on getting those names right.

Single forward pass decisions

A chat model builds its answer one token at a time, and each token is conditioned on the tokens before it. Each generated token costs its own forward pass, made cheaper by a KV cache that computes keys and values for the new token alone [36]. The passes still run one at a time: the input to each pass is the token the pass before it produced, so the model cannot begin the next token until it has finished the last [36]. Ask the same model for a probability per option and it spends one pass per token of that answer [16]. Non-autoregressive models invert this. They produce all output positions in a single forward pass, without conditioning each position on the others, and that is where the speedup comes from [40, 41].

A decision model is non-autoregressive in exactly this sense, and the design is easier to see than the phrase. One published account of how these models work breaks a decision into four steps: pick the decision, write a prompt that lists the options and ends where the answer goes, score each option against that prompt, then normalise the scores into a probability distribution over the

options [16]. The options are known before the call, so there is no text to generate, and all of them are scored in the same pass [16].

Open implementations make the same shape concrete. Clef freezes its Qwen backbone and takes a single prefill-only pass; a joint schema head then turns the backbone's final hidden states into a score for every allowed option of every question at once [12, 42]. Its custom decision path scores the allowed answers in one forward pass rather than generating an explanation [43]. On the Jev side, the API is served by what TypeSafe calls a parallel sampler, which computes all outputs in one pass [38].

The practical consequence follows from the mechanism: with no token stream to meter, the call bills no output tokens [7, 33]. Chapter 1 gives the input rate.

Two limits on that story. The single pass is over the state you send, so a larger state still costs more input tokens even when the answer is one word. And a decision model's speed advantage is a claim you should test on your own workload: TypeSafe's headline multiples come from its own workflow evaluations, and the launch post says those sit at the high end of what users should expect [37].

State and typed questions

A call carries the same two parts. The `state` is the content to evaluate, and it accepts a plain string, a structured object, or an array of text values [34, 33]. The `questions` are a map of typed question objects, and you choose each key yourself [34].

The key detail for your code is that answers come back under the keys you supplied. The map key is not sent to the underlying model and does not affect inference, so you are free to name questions for the reviewer who will read the request in six months [34]. The SDK builds the typed object from that key, so the name you pick is the name your logs will carry.

Here is a Python request that asks three questions about one ticket. It runs against the TypeSafe SDK and defines the state and the three question types in one call [44].

```

1  from typesafe_sdk import Choice, Noul, Score, TypeSafeClient
2
3  state = {
4      "message": "I was charged twice and need the duplicate refunded today.",
5      "account_tier": "business",
6  }
7
8  with TypeSafeClient() as client:
9      response = client.system_one(
10         state=state,
11         questions={
12             "department": Choice(
13                 instructions="Which team should handle this ticket?",
14                 criteria={
15                     "billing": "Charges, invoices, and refunds.",
16                     "technical_support": "Bugs, outages, and integrations.",
17                     "sales": "Pricing, plans, and new purchases.",
18                     "other": "None of the options clearly fits.",
19                 },
20             ),
21             "is_urgent": Noul(
22                 instructions="Does `message` explicitly communicate time
23                     ↪ pressure?"
24             ),
25             "frustration": Score(
26                 instructions="How frustrated does the customer appear?",
27                 criteria=["Calm and neutral", "Concerned but civil", "Very
28                     ↪ angry"],
29             ),
30         },
31     )

```

Reading the result needs no parser, because every answer arrives under the key you chose [44]:

```

1  dept = response.answers["department"]
2  print(dept.choice, dept.probabilities, dept.confidence)
3
4  print(response.nouls["is_urgent"].noul)      # 0.98
5  print(response.scores["frustration"].score)  # 1.8

```

The department answer carries the chosen option, a probability for each option you listed, and a confidence. The is_urgent answer carries the probability that the statement is true, 0.98 in the worked example. The frustration answer carries the probability-weighted level, 1.8 on the three-level scale [44, 34].

Three rules follow from that request [44, 33]. The Choice criteria is a dictionary mapping each key to a description of what belongs to it. The Score criteria is an ordered list from lowest to highest. And the Choice includes an other option, because a list that does not cover all inputs forces the model to pick the least wrong answer.

That last point has a measured failure behind it. On 30 messages that fit none of the available categories, all 30 still drew a confidence of at least 0.99 [4]. The study records the confidence, not which category each message was forced into. A list that does not cover the input forces the model to choose its least wrong entry, which is why the design guidance is to add an “other” option whenever real inputs may fall outside your categories [17, 45]. Which answers you allow is a design decision, and it shapes the judgment the model makes [4].

The questions run in parallel and in isolation against that same state, so adding a tenth question costs its own tokens but almost no extra time [35, 33]. TypeSafe calls the resulting habit speculative fan-out. The advice is to put every question you may need into one call, even ones that only some inputs require, and let your code choose which answers to act on [17]. You only need a second request when you cannot build it until the first answer arrives, for example when the first answer decides the next question’s options [17].

Keep each question to one judgment. The guide asks for questions a knowledgeable person could judge in a few seconds [33]. If a question needs a chain of reasoning, split it into several questions and combine the answers in code [33, 17].

Confidence and calibration

Each choice and score answer carries a confidence value from 0 to 1, and it is not the probability of the selected option [37]. It is computed from the shape of the probability distribution: a distribution concentrated on one option is confident, a flat one is not [33, 17]. TypeSafe documents the choice formula as $(p_{\text{max}} - 1/n) / (1 - 1/n)$, where n is the number of options, and the reference adapter system-one-adapter 0.2.1 carries the same formula [10, 46]. For three options that reduces to $(3 * p_{\text{max}} - 1) / 2$ [33].

A worked number makes the difference concrete. Take a three-option choice with probabilities 0.80, 0.15, 0.05. TypeSafe’s documented formula returns a confidence of 0.70, while the winning probability itself is 0.80 [46].

The two are related but not equal. The remaining gaps follow from TypeSafe's formula: hold the winning probability at 0.80 and the gap between it and the confidence is 0.20 with two options, 0.10 with three, 0.067 with four, and 0.022 with ten, so a longer list narrows the gap.

`noul` questions have no separate confidence field, because the single probability is the uncertainty signal already [37, 17]. A `noul` at 0.5 means the model cannot tell. To measure a degree rather than a fact, you use a `score` [17].

Calibration is a property of those numbers over many predictions. TypeSafe trains with a method it calls Reinforcement Learning for Calibrated Decisions, which optimizes the reported probabilities against outcomes rather than against human preference [33, 18]. In a calibrated model, outcomes assigned 90 percent probability should occur about 90 percent of the time across many predictions [37]. Laya's own model card describes a similar objective: the reward is a strictly proper scoring rule, so expected reward is maximised only by reporting honest probabilities [47].

Treat calibration as a training objective and measure it on your own data. Independent validation of the method is thin, and TypeSafe's own framing of RLCD is a company-defined term [18, 37].



Do not carry a confidence threshold from one model to another. Clef's reviewed release code, at repository revision a20e258, sets the choice confidence to the winning probability itself rather than Jev's formula [46]. The same 0.80, 0.15, 0.05 distribution gives 0.70 under Jev's formula and 0.80 under the reviewed Clef helper, so an existing rule of confidence ≥ 0.75 would reject one and accept the other while the distribution is identical [46]. Preserve the returned distribution and define your acceptance statistic explicitly in your own adapter [46].

The JEV name and Clef

The name carries two references. Jev is named after William Stanley Jevons, the nineteenth-century economist behind the Jevons paradox: making steam engines more efficient increased coal consumption rather than reducing it, because cheaper power found new uses [17, 4]. The bet behind the name is that

a decision cheap enough to run by the thousand will be worth asking wherever a model was previously too expensive to call [18, 37].

TypeSafe borrows its System One label from Daniel Kahneman's *Thinking, Fast and Slow*, which splits thinking into fast, intuitive judgment and slow, deliberate reasoning [17, 35]. Chapter 3 takes up how that split maps onto decision models and reasoning models.

The sibling worth knowing is Clef. Cloudflare released it on October 1, 2026, in two sizes: Clef, a 27-billion-parameter model prioritizing accuracy, and Clef-flash, a 9-billion-parameter model built for lower latency [12, 48]. Both are post-trained from Qwen backbones, Qwen3.8-27B and Qwen3.5-9B respectively, rather than pretrained from scratch, and their custom decision path scores the allowed answers in one forward pass [43]. The weights are published on Hugging Face under the Apache 2.0 license, so you can download and run them on your own hardware [12, 49].

Clef takes the same request body as Jev. Cloudflare describes it as fully Jev-API compatible, and its launch example is the same `state` plus `questions` body with `noul`, `choice`, and `score` entries [50, 42]. The answer shapes match as well. The REST response does not. Jev returns answers at the root, while Clef's REST route wraps the Jev-shaped body under `result`, beside `success`, `errors`, and `messages`. Code written for Jev over REST therefore needs an adapter to reach answers, though a Workers AI binding returns the Jev shape directly [12, 48]. Clef shipped sixteen days after Jev, and seven of the eight models in one public benchmark accept Jev's `/v1/systemone` request format [19, 50]. A later chapter builds the adapter.

Clef differs from Jev in three places that matter for a decision, all beyond the shape of the answer [49]. It has a vision encoder and accepts images, where Jev reads text only. Its hosted context is a single 65,536-token window, and the hosted contract truncates long text state rather than splitting it into state and longest-question budgets. Jev allows 64K per request but caps the state plus longest question at 32K [49, 46]. And its weights are published, where Jev's are not, so either Clef variant can run on your own hardware while Jev cannot [12, 49].

What decision models are not

Three comparisons come up first, and each is partly right.

A decision model is not an LLM, though it is built from the same transformer machinery. It does not generate free-form text and has no chat interface; its output is restricted to typed values you defined, which makes it closer to a fast, calibrated classifier than to a chatbot [51, 52]. It cannot draft a reply, write code, or explain its reasoning [33]. The architecture that uses it well runs both: a decision model chooses the route, an LLM writes what the user reads [51, 53].

You may already run a schema-validated LLM call, and a decision model looks the same at the API boundary. The two constrain different things. Structured outputs constrain the format of generated text after the fact, while a decisions endpoint bounds the answer space before inference [39]. The difference shows under load. A constrained LLM still generates token by token, so generation can fail or stop early, and the Vercel AI SDK reports those as structured-output errors [17]. TypeSafe argues that forcing an LLM into a schema can leave some of its intelligence on the table, and trains the decision model for the structured answer from the start [33].

The comparison to a classic trained classifier is the strongest of the three. One commenter's summary, "decision model = classifier," holds up on the math: both pick from labels and return probabilities [52]. The practical difference is what happens when the labels change. A classic classifier bakes its labels into the weights and needs labeled examples and a training step per task. A decision model defines its labels in the request, so a new category costs a text edit instead of a training run [52]. Some zero-shot classifiers already accept labels at request time, so what separates the two is the training step a decision model skips [45, 52].

The guarantee is narrower than the marketing. TypeSafe plots a 0 percent hallucination figure, and its own launch notes explain where the zero comes from: the model cannot return anything outside the schema, so TypeSafe counted schema violations, which come out at zero [35]. The company is candid that the number is guaranteed by construction rather than measured, and calls it "not empirical" [3]. What you can rely on, and what a pipeline genuinely gains from, is that parsing cannot fail on a malformed value and no fabricated option can appear [3, 38].

What you cannot rely on is that the valid answer is the right one. A decision model can confidently pick the wrong option from your list, and the architecture has no guard against it [3, 37].

The published failure modes tell you where to look. TypeSafe maintains a

jaggedness page per version, and for j ev - 1 . 13 it lists literal reading (the model answers the question you wrote, not the one you meant), unreliable arithmetic and counting, dates read as text rather than ordered values, accuracy that falls as the state fills with irrelevant detail, and adversarial content that can move the answer [37, 2]. Each one is a design rule: compute numbers in code, filter the state before you send it, and write criteria explicitly [2, 37]. Because the failure modes change between versions, a threshold tuned on 1 . 13 does not carry to the next release [2].

One more failure is easy to miss because the model does not look broken when it happens. The number you get measures the model together with the wording you chose, rather than the task by itself. On one benchmark, reversing the polarity of a yes/no question with identical meaning left Jev and Clef steady, but dropped Kev 9B from an ROC AUC of 0.85 to 0.54 and pushed one model from 0.92 to 0.17 [19]. Run a new model on both wordings of your question before you trust it [19].

Before you send anything, write down the type of each question you plan to ask: a choice from a named set, a place on an ordered rubric, or a yes/no with one probability. That list is the schema the rest of your pipeline depends on. Fixing a checkpoint revision at deploy time keeps it stable, and later chapters cover how to pin one. Save the commit hash alongside the model, because a repository that tracks a branch can change under you. The Kev-4B card names its checkpoint as `main` at revision `139fdd94`, which is the form to copy [54].

System 1 and System 2

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Dual-process theory as architecture

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Reflexive versus deliberative routing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Where decision models win

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Where deliberation is still required

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Part II: Using Decision Models

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Task Taxonomy and Budgets

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Classification and intent detection

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Routing and tool selection

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Scoring and safety checks

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Latency and throughput budgets

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Cost per decision

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Decision Models in Agent Stacks

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Decision models inside agent loops

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Routers and cascades

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Tool and skill selection

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Confidence as an operating policy

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Extraction and structured output

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

The Local Variant Catalog

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Ollaya's listed models

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Typed-decisions scores

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Licenses and commercial use

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Adjacent decision backbones

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Multilingual checkpoints

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Choosing a checkpoint

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Part III: Running and Deploying Locally

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Running Local Inference

The local loop

A first local decision takes minutes. You load a checkpoint, send a state and a typed question, then read back a probability distribution and a confidence value. That confidence number will not tell you which choice to trust on its own: it can sit far from the winner's probability, two checkpoints can read the same ticket differently, and a padded state flips a confident answer. The loop here uses Kev-4B on localhost at the Kev 1.0 revision, so what you run reproduces the quickstart in the project's README [22].

Loading a checkpoint

Kev's own server is the route with the least new code: it loads the weights once and speaks the same `/v1/systemone` wire format as TypeSafe's hosted Jev, so clients you already have work against localhost [10, 63]. You need Python 3.12 or 3.13 and uv [10]. Kev-4B is the recommended starting point [94].

```
1 git clone https://github.com/jaredpalmer/kev.git && cd kev
2 uv sync --extra serve
3 uv run --extra serve python -m kev.serve --run jaredpalmer/kev-4b@v1.0 --port
  ↪ 8009
```

The first run downloads the Qwen3.5-4B base model and the Kev adapter that turns it into a decision engine [10]. A Kev checkpoint is a rank-16 LoRA adapter plus a pointer head on a frozen Qwen base. The head compares a closing decision token's hidden state with each option's closing token and returns probabilities [63]. Each checkpoint ships with a fitted temperature, the calibration knob baked in [10]. The `@v1.0` tag makes the load reproducible: every Hub repo carries it, and the Kev 1.0 release pins the checkpoint behind it, so `--run jaredpalmer/kev-4b@v1.0` loads the same weights every time [10, 22]. Kev-4B's weights sit at Hub revision 139fdd94 with temperature 2.41 [10, 54]. The server runs on CUDA or ROCm when a GPU is present, or on MLX on

Apple Silicon [10]. Kev-27B, the strongest member, wants an 80 GB GPU, and its 51 GB of full weights ship on the Hub only [22].

Verify the server from a second terminal:

```
1 curl -s localhost:8009/v1/models
```

The response lists the loaded checkpoint's details, and an independent walkthrough on a Mac read the Kev-4B entry as `Backend: MLX, Dtype: bfloat16`, and `Temperature: 2.406...` [108]. A second entry named `jev-latest` points at the same local Kev checkpoint in this setup, so treat it as an alias [108]. The server binds to `127.0.0.1` and is open by default, so set `KEV_API_KEY` to require a bearer token [10].

Laya offers the same loop without a server. If you prefer a plain Python call, run `pip install laya` and then `laya.load("convaiinnovations/laya")` [109, 11]. The router is lazy, so the first prediction downloads the selected checkpoint and reuses the Hugging Face cache [11]. The package also installs a CLI, so `laya "I was charged twice, please refund"` runs a routing decision offline with no download, and adding `--predict` downloads the checkpoint and returns full answers [11]. The English checkpoint is about 847 MB on disk [110]. This chapter stays with Kev-4B, because its server response is the one you will reproduce at the end.

Your first choice question

A request has two parts: `state`, the text you judge, and `questions`, a map of typed questions about it [34]. Each question carries a `type` and, where the type needs them, `instructions` and `criteria`. The key you choose for it is a caller-side handle and is never sent to the model [34, 111]. Send the support ticket from the Kev README:

```

1 curl -s localhost:8009/v1/systemone -H 'content-type: application/json' -d '{
2   "state": "Shoes arrived two weeks late and in the wrong size. Also I see two
   ↳ charges on my card.",
3   "model": "kev-latest",
4   "questions": {
5     "department": {"type": "choice", "instructions": "Which team should handle
   ↳ this?",
6                       "criteria": {"returns": "Exchanges, refunds, wrong or
   ↳ damaged items",
7                                     "shipping": "Delivery status, delays, lost
   ↳ packages",
8                                     "billing": "Charges, invoices, payment
   ↳ problems"}}},
9     "escalate": {"type": "noul", "instructions": "Does this need urgent
   ↳ human attention?"},
10    "frustration": {"type": "score", "instructions": "How frustrated is the
   ↳ customer?",
11                      "criteria": ["Calm", "Frustrated", "Very angry"]}
12  } }'

```

One call asks all three question types about the same state, and the questions cannot read each other's answers [10]. The criteria map carries the meaning, one description per option [10], and TypeSafe's guidance is to write criteria as you would brief a new hire, because the descriptions carry all of the meaning [111].

The response, in the form the README shows for Kev-4B running in bfl6 on an Apple M5, is a JSON fragment:

```

1 {
2   "model": "kev-latest",
3   "answers": {
4     "department": { "type": "choice", "choice": "returns", "confidence": 0.21,
5                     "probabilities": { "returns": 0.47, "shipping": 0.28,
   ↳ "billing": 0.25 } },
6     "escalate": { "type": "noul", "noul": 0.93 },
7     "frustration": { "type": "score", "score": 1.44, "confidence": 0.34,
8                      "legend": { "0": "Calm", "1": "Frustrated", "2": "Very
   ↳ angry" },
9                      "probabilities": { "0": 0.00, "1": 0.56, "2": 0.44 } }
10  },
11  "usage": { "input_tokens": 101, "output_tokens": 161 },
12  "latency_ms": 495
13 }

```

The department distribution mirrors the ticket: the three probabilities reflect the wrong size, the late delivery, and the double charge [10]. Kev picked

returns, but the margin is thin. The interface returns probabilities instead of a single label, so you can route the confident cases and send the rest to a person [10].

Choice wording moves accuracy. Asking about all candidates in one choice question beats asking one yes/no question per candidate, 83.2% versus 78.0% on MASSIVE and 78.8% versus 67.0% on BANKING77's 64 options [72]. Giving similar options a one-line "use when" description was the only change that clearly helped in that study, by +4.8 points on BANKING77 (64 options) and +4.3 on CLINC150 (all 150), while longer rules and worked examples did not help further [72]. Facts belong in the state, and a JSON state scores the same as plain text [72]. When your list might not cover every input, add an `other` or `none_of_the_above` option, because the model has to pick one of the options you supplied [17, 4].

Scoring a candidate answer

The score question places the state on an ordered rubric with the lowest level first, returning a probability-weighted level that can land between two anchors [12, 34]. Kev computes the score as the mean level index starting at 0, and returns it with a legend that maps each index back to your description [10].

The frustration answer above shows the mechanism. The three levels map to Calm, Frustrated, and Very angry, with 0.56 of the mass on Frustrated and 0.44 on Very angry, so the weighted level is 1.44, between the two and nearer Frustrated [10]. An independent local run on a different ticket produced the same shape, a severity score of 1.7751 with the High level carrying 0.8153 of the mass [108]. Other JEV-like runtimes compute the same expected level. OpenClaw's local ONNX plugin returns the expected zero-based index of the ordered criterion levels [105].

The score is a summary, so read the distribution behind it. The response prints probabilities next to the score because a 1.0 can mean all the mass on level 1 or half on level 0 and half on level 2 [17, 10]. The float alone will not tell you which of the two this ticket is.

Reading the confidence value

Confidence is a separate number from the chosen option's probability and is not interchangeable with it [37, 45]. It collapses the distribution's shape into

one value, high when the mass concentrates on one option and low when it spreads out [17, 111]. The formulas come from TypeSafe’s reference adapter, and an independent LiteRT port reproduces them from the author’s fp32 code [10, 112]. For a choice with more than one option, confidence is $(p_{\text{max}} - 1/K) / (1 - 1/K)$, with K the option count, and a single option has confidence 1. Score confidence is $\max(0, 1 - E|\text{level} - \text{mode}| / D)$, where D is the mean distance of a uniform distribution over the levels from its middle, $2/3$ for three levels [10, 112]. Neither field is a measured accuracy rate [10].

The examples check out against the formulas. Department has p_{max} 0.47 with $K = 3$, and $(0.47 - 1/3)/(1 - 1/3)$ is 0.21, the returned confidence [10]. Frustration puts 0.56 of the mass on the modal level 1 and 0.44 on level 2, so $E|\text{level} - \text{mode}|$ is 0.44 and $1 - 0.44/(2/3)$ is 0.34 [10]. The formula also explains why confidence and winner probability diverge. For probabilities 0.80, 0.15, 0.05, Jev’s documented formula returns 0.70, below the winner’s 0.80 [46]. The same divergence shows up between checkpoints. In that walkthrough, Kev-4B split the department choice 0.4155 on support against 0.4014 on billing, so with three options its choice confidence is about 0.12, while Kev-0.8B, at 0.6763, lands near 0.51 [108, 10]. Both picked the same team, yet the larger model’s winner leads its runner-up by 0.014, so its “same answer” is the ambiguous one. An example check in the article sends a case for manual review whenever the maximum probability falls below 0.8 [108]. Raw probabilities compare alternatives, so the gap between the top candidate and the runner-up flags the inputs worth a human look [109].

Calibration is an aggregate property, so if a model reports 90 percent confidence across many decisions, roughly 90 percent of them should be right [4]. Thresholds come from your own labeled data. TypeSafe’s guidance is three bands: act automatically above one threshold, flag for review in the middle, and route to a person below the other, set by labeling a few hundred examples and checking where the probabilities fall [111]. The documentation uses 0.5 as a review floor and 0.9 before a destructive action, presented as examples rather than defaults [17]. Noul answers carry no separate confidence field, so the returned probability of yes is the number you threshold [37, 34].



A confidence threshold does not transfer between servers, because implementations compute confidence differently (see Chapter 2), so reassess thresholds after any model, question, or rubric change [46, 113].

Reproducing a quickstart

Reproducing the walkthrough means pinning two things: the checkpoint bytes and the serving path. Kev 1.0 froze the checkpoints and the evaluation and serving code, so every Hub repo carries a `v1.0` tag, and the GitHub release ships the three smaller checkpoints (0.8B, 4B, 9B) as tarballs with SHA-256 checksums [10, 22]. Kev-4B's published checksums are `2e707e2e...` for the tarball and `90e81735...` for the adapter, with `head.pt` at `dd633435...` and temperature 2.4061 [22, 54]. If you serve from the release tarball instead of the Hub, run `shasum -a 256 -c SHA256SUMS.txt` against the checksum file the release ships [22].

The returned numbers depend on the serving path too, so `KEV_DTYPE=fp32` serves the exact fp32 path the evaluations use, while `bf16` is the default on GPUs [10]. `usage.output_tokens` counts the tokens in the serialized answers rather than generated tokens, because nothing is generated [10].

If you prefer the TypeSafe SDK over `curl`, the README's client works unchanged against the local server: `TypeSafeClient(api_key="local", base_url="http://127.0.0.1:8009", model="kev-latest")` [10, 114]. On the Laya path, the equivalent quickstart is the packaged CLI, `laya "Refund my card" --predict --min-confidence 0.9`, which returns the answer and marks it when the model is unsure [11]. Latency will differ from the README's 495 ms on your hardware. On Apple silicon the Laya-MLX port reports single-digit to low-teens milliseconds on an M3 Max [115].

Debugging bad inputs

A decision model can flip on material that does not change the answer. An independent study, JevOut, tested four decision systems, including Jev, on seven datasets and found that short, naturally worded context additions redirected 61.4 to 73.2 percent of each system's initially correct decisions toward a wrong option fixed in advance [116]. The redirection hit initially confident decisions, often ending in high-confidence wrong choices. It transferred across models [116]. In one recorded example, Jev assigned the correct option probability 0.97 and the fixed wrong target 0.01. After one added sentence about a related issue, with the question and its options unchanged, the target rose to 0.54 and the correct option fell to 0.44 [116].

So treat the state as a design input. A decision model’s accuracy depends heavily on the state it receives, and it degrades when that state is padded with material the question does not concern [117]. TypeSafe lists “large state full of irrelevant detail” as a documented weakness of its current build and calls the effect context rot, and the effect shows up in length alone, as Jev’s accuracy on LongBench falls from 93.5 percent at 4,000 to 8,000 tokens to 74.5 percent around 20,000 [117, 73]. Keep each question on the smallest block it concerns [117].

Budget the state to the lengths the checkpoint trained on and validated. Kev’s server accepts a state of up to 65,536 tokens plus 8,192 per question, but the small Kevs trained mostly on states of up to 384 tokens, with document and skill fine-tunes reaching 7,552 tokens, and accuracy is validated only to 8,192 tokens [10, 54]. Inputs past that length are territory the measurement does not cover, even though the server still accepts them [22]. The server validates rather than truncates, so a state over 65,536 tokens comes back 422 with the token count unless you set `KEV_TRUNCATE_STATES=1`, which reads the first 65,536 tokens and reports `truncated` plus `usage.state_tokens` in every response [10, 22]. The encoder-headed models are tighter. Laya’s English checkpoint truncates the state to 512 tokens after the question header, and options must fit a 192-token head budget [118, 107]. The model’s own guidance is to keep choice questions under about 20 options [118, 107]. At 50 options the option text needs 302 tokens against that 192-token budget, so every option is cut to its key and no description survives, and at 150 options the checkpoint refuses the question [91]. A 77-option Banking77 question leaves roughly three to four tokens per label, which is why Laya trails Jev there (see Chapter 4) [11]. Fix it by widening `head_max_len` and `max_len` for that request, or split the label set into a coarse question and a fine one [11].

First, label wording. Choice keys are rendered verbatim, and Laya checkpoints can follow labels such as `true/false` or `yes/no` instead of the option descriptions, so use semantic keys or opaque keys like `A/B` [11, 47]. Semantic labels do not make negation safe. On five cancellation examples, Laya 0.3.20 with `no_action/cancel_account` keys chose `cancel_account` for all four negated requests on the English checkpoint and two on the multilingual one, which assigned one of them probability 0.9998 [11]. Noul can follow its own labels instead of the state, so when it looks stuck, ask the same proposition as a two-option choice with neutral keys and your `yes/no` wording as the descriptions [47].

Second, option order. Where an option sits changes the answer. Laya measured five identical options whose per-slot logits span from +1.68 down to -1.91, a spread decided by position alone [11]. Kev's server exposes `POST /v1/systemone/permute`, which runs one choice question under different option orders (`n_perm` 1 to 64, default 6), so you can see how much of a decision is position [10]. Laya's `option_order` lets you average the effect out: on 62 banking intents shortlisted to 20 labels, over 49 states, rotation averaging reduced the share of answers that change with presentation order from 16.3 to 6.1 percent [11].

Third, when an answer surprises you, replay the same state with pieces removed. If a confident choice flips on an apparently irrelevant detail, the detail is load-bearing for this checkpoint, so trim it or move it into the criteria where the option descriptions can absorb it. Then re-run your labeled examples, because the decisions can change whenever the checkpoint or a question changes, even when the schema stays the same [45]. When the checkpoint answers your questions the way you expect, the next step is Chapter 12, which trains a local variant with `uv run python -m kev.train` from the same repository [114].

Quantized Formats and Exports

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

GGUF quantization tiers

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Converting and publishing GGUFs

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

ONNX and runtime export

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Core ML and MLX targets

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Choosing a quant level

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Serving with Ollaya

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Pulling and serving models

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Modelfiles for decision models

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

The Ollaya MCP server

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

ONNX Runtime versus llama.cpp

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

kevs.serve on CUDA and MLX

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Serving with the TypeSafe API

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Typed questions over HTTP

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

The wire API shape

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Using the typesafe_sdk client

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Pointing clients at local endpoints

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

OpenAI's Decisions API

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Gateways and drop-in compatibility

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Part IV: Training and Evaluating

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Building Decision Datasets

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Intent corpora and benchmarks

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Synthetic decision triples

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Few-shot and SetFit sets

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Multilingual decision data

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Label quality and splits

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Fine-Tuning with LoRA and QLoRA

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Parameter-efficient fine-tuning basics

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

LoRA and QLoRA mechanics

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Training with Unsloth

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Training with Axolotl and TRL

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Training a decision head

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Merging and exporting adapters

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Evaluation and Calibration

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Decision accuracy versus schema validity

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Calibration and ECE

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

The typed-decisions benchmark

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

The Bespoke Labs benchmark

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Building an eval harness

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Regression tests in CI

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

What you can now do

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Choosing your fast path

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Where to go next

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Appendix A: Command and flag reference

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Serving

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Kev serve

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

decider serve

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

llama-server (llama.cpp)

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

The Ollaya CLI

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Ollaya Modelfiles

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Environment variables

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

TypeSafe SDK

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Kev

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Laya

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Training, calibration and evaluation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Appendix B: Pinned versions register

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

What the register records

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Checkpoints: Kev 1.0

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Runtimes and serving builds

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Wire-format pins

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Benchmark pins

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Dataset and gold pins

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Training pins

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

License pins

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Appendix C: Quantization format reference

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Precision formats

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

GGUF and the GGML block schemes

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

What quantization costs a decision model

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Training-time formats

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Host formats and hardware quants

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Format by deployment

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

Glossary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.

References

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/guide-to-decision-models>.