

How Jev works: calibrated decision models

Victor Dibia
Independent Researcher
USA
victor.dibia@gmail.com

Interactive version, with live figures: <https://victordibia.com/explainers/jev/>
(revision 2, last updated 2026-09-24).

TypeSafe AI recently released Jev, a model that promises low latency (TypeSafe reports 70 to 500 ms per call) and inference so cheap that output tokens are not metered[2, 6]. At the same time, TypeSafe says Jev “achieves similar levels of intelligence on System One tasks compared to existing LLMs”[2]. This is fascinating.

A claim like this raises some natural questions. How does it work: what is the architecture, and how does it differ from the typical autoregressive model? What are the key tradeoffs, and when does it make sense to use a model like Jev? Can I build, train or use one for my own use cases, and what does performance look like?

This article works through those questions in order: how Jev and models like it work, experiments I ran on open Qwen models to test the idea and to train a small one, and what we know about Jev itself.

1 How decision models like Jev work

Jev is a “System One” model (the name echoes Kahneman’s System 1, fast intuitive judgement) from TypeSafe AI, described as “a frontier-intelligence function call: unstructured state in, typed probabilistic decisions out”[2].

Models like Jev answer in four steps:

- (1) **Start with a decision.** Given a **state** (the text the decision is about) and a list of options, pick one. For example: a customer writes to a bank, “*I want to change my address.*”, and we need to pick their intent from a list of 10 potential intents.
- (2) **Turn it into a sentence to complete.** Write a prompt that lists the options and ends exactly where the answer goes:

Classify the customer message into exactly one of these intents.

Intents: Refund not showing up; age limit; edit personal details; ... (10 in all)

Message: I want to change my address.
Intent: ____
- (3) **Score every option.** Append each option to the prompt and ask the model how likely that exact text is as the completion (the product of the probabilities of its tokens, the word pieces a model reads and writes). The options are known in advance, so nothing is generated, and all of them are scored in parallel. We call this step **scoring**: reading the model’s probabilities for text we supply, instead of generating text.
- (4) **Normalise.** Divide these probabilities by their total so they add up to 1. The results are the **option probabilities**, and the highest is the answer.

If you have built a text classifier, this will look familiar. A few years ago I fine-tuned BERT for text classification: a classification head

on top of the model gives a probability for each class. A decision model gives the same kind of output, except the classes are defined at inference time, in the prompt, instead of being fixed when the model is trained.

TypeSafe has not disclosed how Jev is implemented. Its API and statements fit these steps, and so do the open reproductions[14, 22] and our own experiments[13], with one difference: the reproductions do step 3 by reading a one-letter code for each option instead of the whole option text, which *Can a cheaper readout do as well?* (Section 14) measures. One of them, djev, uses a diffusion language model instead, which fills a blank for every question in a single step[19]. See the section below on *What we know about Jev* (Section 21).

Jev packages this as an API with a few distinctive traits:

- **No text generation.** “System One models do not write replies, produce code, or generate explanations of their reasoning”[8].
- **Many questions, one read of the input.** The model “processes the state once and evaluates all questions against it in parallel”[6].
- **Three question types, all lists.** A *choice* picks one of up to 255 options[4]. A *score* rates the state on 2 to 10 ordered levels[7]; I call it the *rating-scale question*, because “scoring” already means step 3. A *noul* (“short for bernoulli”[24]) answers yes or no with one probability.
- **Fast and cheap.** TypeSafe reports 70 to 500 ms per call, against 3 to 329 s for frontier large language models (LLMs) on the same tasks[2], and charges \$0.042 per million input tokens with output free[6].

2 Calibrated decision models

While Jev is one implementation, we can think of this class of models as **calibrated decision models**. *Decision* means the goal is to select from a list of options, *not to generate text*. *Calibrated* refers to the model’s **confidence**, the probability it gives to the option it chose. A model is **calibrated** when its confidence matches its accuracy: if it gives a thousand answers at 90% confidence, about 900 of them are right.

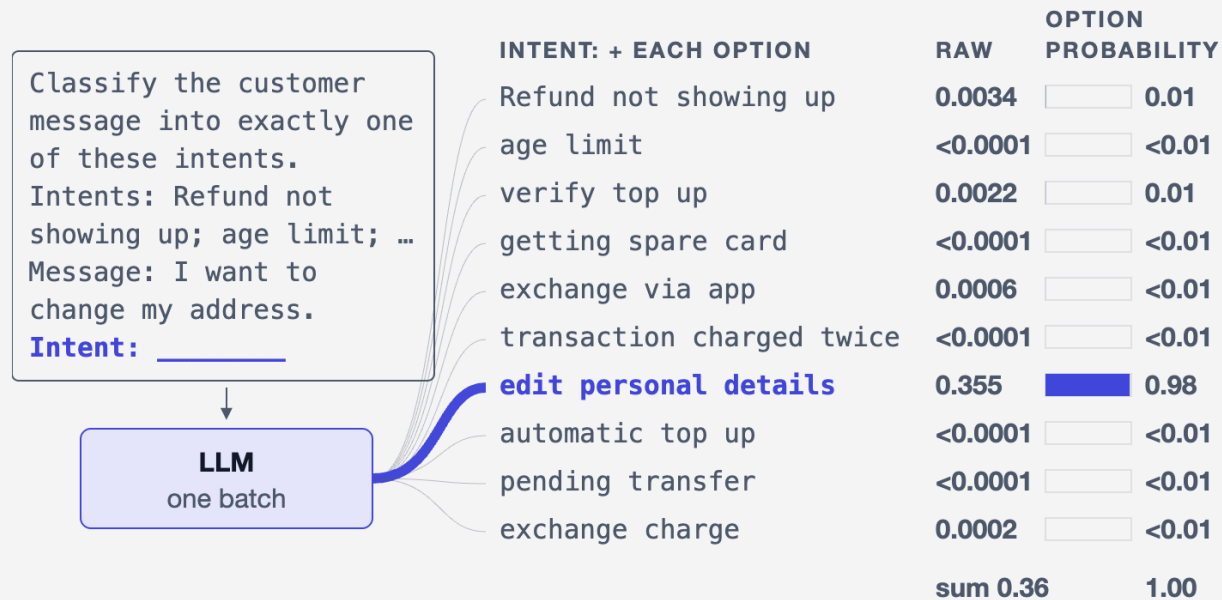
In my opinion, confidence is important here because it lets us build systems and apps that do something ML models are notoriously bad at: say “I don’t know” or “I’m not sure” when confidence is low.

The avid reader is likely getting suspicious here: calibration is a hard problem. Early evidence suggests Jev is well calibrated. One independent benchmark measured its **expected calibration error** (ECE) at 0.0588 on 662 prompt-injection messages[15], meaning its confidence and its accuracy differed by about 6 percentage points on average. A stock model came close on the same task, as *What we know about Jev* (Section 21) shows.

How a decision model picks an answer, in four steps

An open 7B model with the prompt used in the experiments. All probabilities are measured.

STEP 4 · NORMALISE



Divide each by their total (0.36) so they add up to 1.

The highest is the answer.

Figure 1: The running example goes through the four steps on an open 7B model (the setup is in the experiments section). In step 3, the raw probability of *edit personal details* is only 0.36, because the model split the first word between “Edit” and “edit”; after normalising in step 4 it is 0.98.

3 Use cases for decision models

Decision models have a few properties that make them useful in production. First, the model can only *select from a provided list of options*, which removes a whole class of hallucinations and parsing failures: there is no answer that is not one of your options. Second, when the model is well calibrated, its confidence lets you build more thoughtful experiences that *reflect certainty/confidence*. The system can act when it is sure, ask a clarifying question when it is not, or hand off to a human. Third, it is *fast and cheap* enough to call on every step of a workflow, not just occasionally.

All of this suits tasks with structured output, and many production processes have that shape. A decision model fits when:

- **The answer can be listed:** a category, an action, a yes or no, a level on a scale.
- **The decision is frequent or on the hot path,** so latency and cost matter.

- **One judgement is enough.** Everything happens in one pass, so tasks that need step-by-step reasoning are a poor fit; TypeSafe lists arithmetic and counting among Jev’s weak spots[5].
- **You can check the confidence** against labelled examples from your own task.

Computer use. In the computer-use chapter of my book, Designing Multi-Agent Systems[11], latency and cost are two challenges I highlight: complex tasks need many model calls, one per action, so delays and costs compound. One of the mitigations the chapter points to is smaller, faster models tuned for interface understanding and action prediction. A decision model fits that description: it reads the state of the page and selects the next action from the list of possible actions. Jev takes text only[6], so here the state would be the page’s text rather than a screenshot.

Table 1: Use cases for decision models

Use case	Input	Options	Why a decision model helps
Computer use and web agents	the page as text (DOM or accessibility tree)	the candidate actions: click, type into a field, scroll	every step of the agent currently waits for a generated action
Agent trajectory evaluation	an agent’s trace	pass or fail, or rubric levels	cheap fleet-wide metrics, and fast enough to run inside optimisation loops
Form filling	a document or conversation	each field’s allowed values	many fields answered from one read of the input
Routing and guardrails	an incoming message	an intent, a model or tool to route to, injection yes or no	runs on every request, before anything else
Gameplay and real-time control	the game state	the legal moves	a decision every frame

Agent evaluation. Teams running agents need both fleet-wide metrics and task-specific ones, and observability platforms need to score agents quickly. Evaluation matters even more inside optimisation loops, where an evaluator is called thousands of times, and a fast yes/no or rating-scale question is a good fit for both.

The one caveat is evaluation of the decision model itself. We need evals that tell us not just whether it is accurate, but whether it is calibrated, measured on labelled examples. The experiments below show why: accuracy can look fine while the confidence is badly off.

4 Why scoring is fast

To see why scoring works, and why it is fast, we first need to look at how a language model produces text.

An autoregressive model takes text in and produces text out. The input is split into **tokens**, pieces of text such as a word or part of a word, from a fixed vocabulary (152,064 of them for Qwen2.5-7B, used later[23]). Each token becomes a vector, and the vectors pass through a stack of identical layers. After the last layer, every position is turned into one number per vocabulary entry, and a softmax turns those numbers into probabilities: the model’s prediction of the next token. One run of the input through all the layers is a **forward pass**.

One forward pass produces a next-token probability distribution at every position of the input, all at once.

There are two ways to turn those probabilities into a decision: generate the answer, or score the options.

5 Generating a decision

To **generate** text, we run a loop around the model: run a forward pass, pick a token from the distribution at the last position, append it to the input, and repeat until the model produces a stop token. There are two ways to get a decision this way:

- (1) **Generate a label.** Ask for the answer. The model generates a few tokens, for example *edit personal details*, and you match that text to one of your options.
- (2) **Generate probabilities.** Ask for a JSON object with a probability for each option. The model generates it token by token: every option name and every number.

This loop is serial and hence *slow*: each forward pass needs the token the previous one picked, so the passes cannot run at the same

time. Modern models make each pass cheaper, mainly with a **KV cache**, which stores the work already done on earlier tokens so that each pass only processes the one new token. **But the passes still run one after another.** A short label takes only a few generation steps, while a probability for every option takes one step per token of the whole answer. Both also depend on the model behaving: a generated label can match no option, and generated JSON may not parse.

6 Scoring a decision

The model first reads the prompt: one forward pass over all its tokens, which also fills the KV cache. **Unlike generation, this pass runs in parallel, because every token of the prompt is known before we start.** Scoring relies on the same fact: the options are known in advance too. A second forward pass reads every option side by side, each one reading the cached prompt, and normalising their probabilities gives the answer. That is two passes, however long the options are, and **scoring cannot return anything off the list, because you supplied the text of every option.**

7 What scoring gives up

- **Options are judged separately.** Each option is scored on its own, and one option cannot see another. The model never compares *edit personal details* with *transaction charged twice* directly; the comparison happens only when we normalise.
- **Different wordings compete.** A score is the probability of a piece of text, not of an answer being right. In Figure 1, the model split its first word between “Edit” and “edit”, and only one of them is counted[17]. Word options the way the model would say them.
- **Cost grows with the number of options.** Scoring reads every token of every option, so its cost grows with questions $\times$ options. A generated label costs about the same however many options there are: the model compares the options inside one forward pass and generates only the winner. Giving each option a one-token code, such as a letter, and reading only that avoids the second pass[14, 22]; *Can a cheaper readout do as well?* (Section 14) measures what it costs.

Generating and scoring the same decision, pass by pass

All probabilities are real, measured on Qwen2.5-7B-Instruct for this prompt.
Curve width follows probability.

PROMPT

... 76 tokens: instructions + 10 intents I want to change my address
(assistant turn)

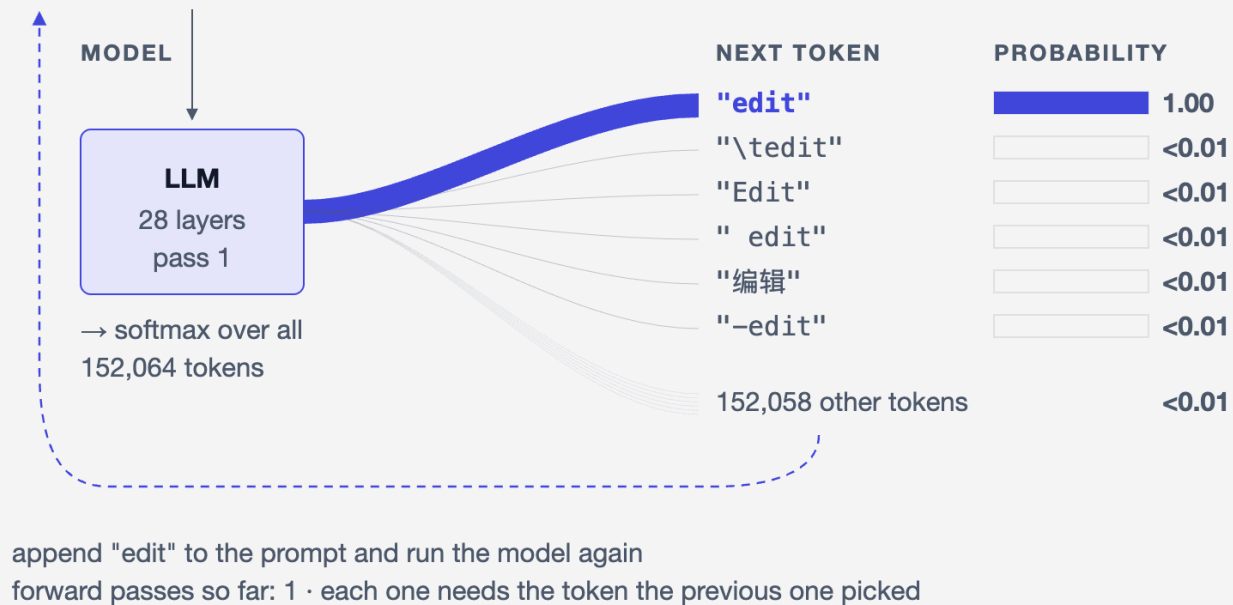


Figure 2: The intent question from Figure 1 is answered both ways with the same prompt, on the 7B instruct model. Generating takes one pass per token of the answer, each waiting for the one before; scoring takes two, whatever the answer. Shown: mode: generate the answer.

8 Experiments with Qwen2.5 1.5B and 7B

To test all of this, I ran quick experiments on Modal using open Qwen2.5 models. The goal was not the fastest possible numbers, but to compare the approaches under identical conditions and report what we see and the conditions under which we see it[13].

- **Models:** Qwen2.5 1.5B and 7B[23], each as a **base** model (trained only to predict the next token) and an **instruct** model (further trained to follow instructions and chat, the kind most APIs serve).
- **Hardware and software:** one NVIDIA A10G GPU on Modal, Hugging Face transformers 4.46.3, 16-bit weights, one request at a time. No serving engine such as vLLM, which would make generation faster but add its own variables.

- **Scoring code:** my own script, implementing the two passes described in *Scoring a decision* (Section 6). An option's **option score** is the sum of its tokens' log-probabilities (the log of their product), and normalising the scores gives the option probabilities. I checked it against a slow version that recomputes the prompt for every option, on a 0.5B model; the two agreed to within 0.00003 in 32-bit arithmetic, and in the 16-bit arithmetic used for the runs they picked the same answer.
- **Data:** for the intent question, 1,200 Banking77 test messages, sampled with a fixed seed, with 2, 10, 25 or 77 options (for fewer than 77, the correct intent plus randomly chosen wrong ones)[21]. For a yes/no question, "is this message a prompt injection?", all 662 messages of the deepset prompt-injections dataset[10].

- **Cost:** about \$4 of GPU time across every run, as billed by Modal. The code and the raw results are in the experiments repository.

9 Is scoring faster than generation?

On the 7B instruct model, reading a 339-token prompt took 0.115 s, and each generated token added 36.9 ms. A generated label (11 tokens) took 0.47 s, and generated probabilities for 10 options (125 tokens) took 4.57 s. **Scoring the same question took 0.23 s.**

In every setting where generated probabilities were run, scoring was 7× to 54× faster than generating probabilities on the 7B, and 13× to 132× on the 1.5B. With 77 options, generated probabilities would be about 890 tokens, roughly 33 s on the 7B by the fitted line (not run).

Against generating a single label, the answer depends on the number of options, the trade-off from *What scoring gives up* (Section 7). With 5 questions about the same state on the 7B, scoring was 9.8× faster at 2 options, 3.5× at 10 and 1.7× at 25, and at 77 options it was slower (0.5×). A longer state narrows the gap too: 3.5× at 256 state tokens, 2.6× at 1,024 and 1.4× at 4,096, because processing the state starts to dominate.

Part of the slowdown at many options comes from our setup, not the technique. Our code makes a copy of the prompt’s KV cache for every option before reading it; serving engines such as vLLM share a single copy, so they would move the break-even point to more options. A serving engine would also make generation several times faster, so all the speedups here are upper bounds. Either way, every token of every option still has to be read, so scoring’s cost keeps growing with the number of options. More questions about the same state, by contrast, are cheap: the state is read once, and each question only adds its own options.

10 Is scoring as accurate as generation?

Scoring was at least as accurate as generating a label.

At 10 options the two approaches picked the same answer on 98.5% of messages. Generated labels were measured on a subset of 400 messages, scoring on all 1,200. Model size matters more than the approach: at 77 options, scoring was right 39% of the time with the 1.5B instruct model and 64% with the 7B.

11 Are the models calibrated?

Base models were close to calibrated; instruct models were not. At 10 options, the base models’ ECE was 0.05 (1.5B) and 0.02 (7B), while the instruct models’ was 0.18 and 0.10. Scale helped but did not fix it: at 77 options, the 7B instruct model’s average confidence was 95%, and it was right 64% of the time. The base models go against earlier findings that neural networks and language models are poorly calibrated[16, 18]; the instruct models agree with them, and match what OpenAI reported for GPT-4: well calibrated after pretraining, and significantly less so after the post-training that turns it into a chat model[20].

Figure 4 shows where the instruct model goes wrong. We sort its answers into groups by confidence and compare each group’s average confidence with the fraction that were right, which gives a **reliability diagram**; a calibrated model sits on the diagonal. The

instruct model puts 1,152 of its 1,200 answers above 90% confidence, at an average confidence of 99.8%, and only 90.6% of those are right.

12 Calibrating with temperature scaling

Temperature scaling divides every option score by one fitted number, the temperature, before normalising. It never changes which option wins, so accuracy stays the same, but it spreads the confidence out. Fitted on 600 labelled messages and tested on the other 600, it brought the 7B instruct model’s ECE from 0.10 to 0.03, about as good as the 7B base model (0.02). Because it keeps the order, it only moves where the 90% line falls. Before scaling, about 108 of the answers above 90% were wrong; after, about 23. **The catch is that it needs labelled examples from your own task.**

One more choice affects calibration: how an option’s tokens are combined into a score. Averaging their log-probabilities instead of summing them made the 7B base model underconfident (44% average confidence, 85% right), while summing made the instruct model overconfident. Pick one way of computing the score, and calibrate that one.

13 What about generating the probabilities directly?

Asking the model to generate its probabilities as JSON was the worst approach we measured. At 10 options, 32% of the 1.5B’s outputs and 7% of the 7B’s were not valid JSON. On the outputs that did parse, the 7B was right 84.1% of the time, against 86.3% for a generated label and 88.8% for scoring, and its ECE was 0.13, against 0.10 for raw scores. It was also the slowest.

14 Can a cheaper readout do as well?

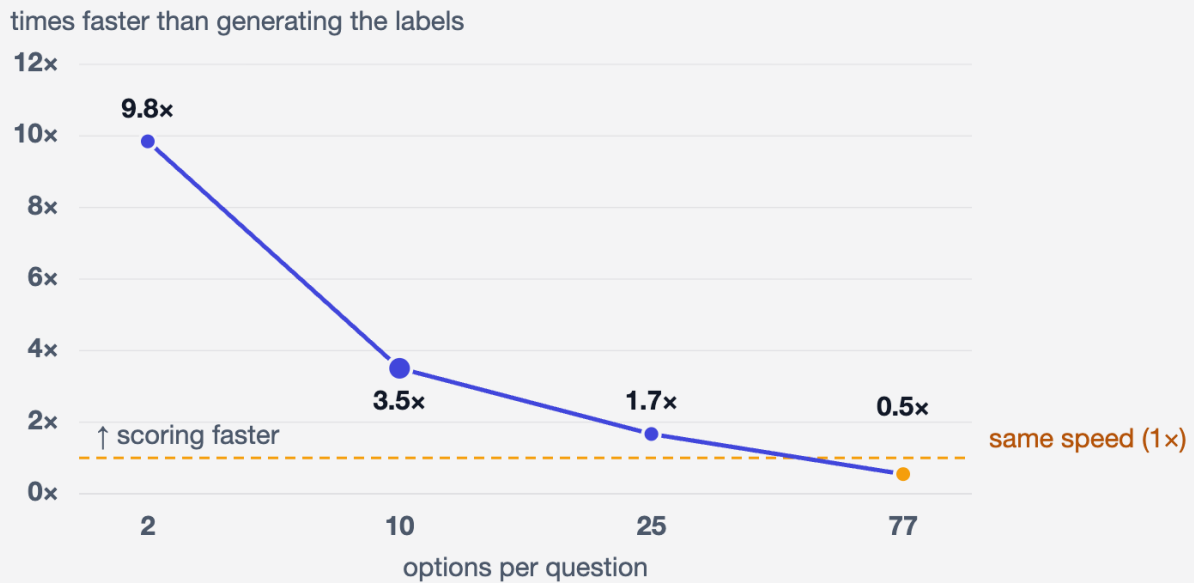
Full-option scoring pays for every token of every option (*What scoring gives up* (Section 7)), yet on the running example almost all the difference between options sits in the first token: once the model has produced “edit”, the rest of *edit personal details* is close to certain (0.999 and 1.000), because it is copying from the list in the prompt. A cheaper readout, used by the open reproductions[14, 22], lists the options with a letter each (A, B, C, and two-letter codes past 26) and reads the probability of each letter at the answer position. **Each code is one token**, so a single forward pass gives a probability for every option, however many there are. Fine-tuned letter classifiers, such as the one in Together AI’s tutorial, generate the letter instead[1]; on the 7B instruct model the generated letter matched the readout’s top letter on 99.5% of messages.

I ran the letter readout on the same messages, options and order as the full-option scorer, so every difference below is measured on the same items.

On the 7B instruct model, letters were within about two points of full-option scoring up to 25 options, and 5.4 points behind at 77. The 7B base model kept up to 25 options but was 22.0 points behind at 77, the 1.5B instruct model fell well behind above two options, and the 1.5B base model at every count. Mapping a letter back to its meaning is a task in itself, and small models do it poorly. The 1.5B base model also showed position bias: with two options it picked A 87% of the time, when A was right 52% of the time. Calibration was not the problem. After temperature scaling,

Scoring's lead over generating a label shrinks as the options grow, and reverses at 77 options

Qwen2.5-7B instruct. Each request asks 5 questions about the same 256-token state, each with the number of options on the x-axis.



At 10 options: generating the labels took 1.82 s, full-option scoring 0.52 s.

Figure 3: Each point shows how many times faster scoring was than generating a label, at 2, 10, 25 and 77 options. A request asks several questions about the same 256-token state, and the state is read once for all of them, which is why more questions favour scoring. Shown: questions: 5 questions; model: Qwen2.5 7B.

Table 2: Is scoring as accurate as generation?

7B instruct	10 options	77 options
Generate a label: accuracy	86.3%	58.3%
Generate a label: answer matches no option	1.3%	7.5%
Score the options: accuracy	88.8%	63.5%

Table 3: Accuracy: letters vs full option

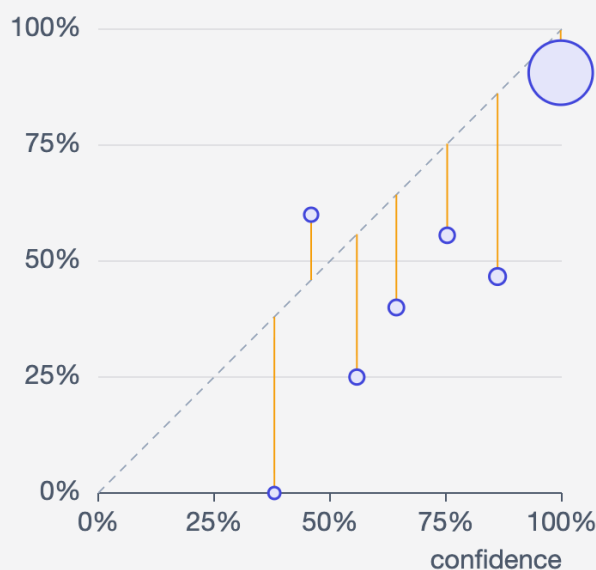
Accuracy: letters vs full option	10 options	25 options	77 options
7B instruct	88.6% vs 88.8% (−0.2)	77.8% vs 79.8% (−2.1)	58.1% vs 63.5% (−5.4)
7B base	81.8% vs 84.8% (−3.0)	71.0% vs 73.4% (−2.4)	37.6% vs 59.6% (−22.0)
1.5B instruct	63.2% vs 73.8% (−10.5)	39.8% vs 59.5% (−19.7)	9.2% vs 39.3% (−30.1)
1.5B base	54.3% vs 68.9% (−14.7)	29.3% vs 51.9% (−22.6)	4.0% vs 35.9% (−31.9)

the 7B instruct model's ECE at 10 options was 0.028 with letters and 0.028 with full options.

The gain is in cost. With the state shared across questions, reading the letters for 20 questions of 77 options took 0.54 s on the 7B,

1,152 of 1,200 answers have confidence above 90%, and 90.6% of those are right

Banking77 with 10 options, Qwen2.5-7B-instruct. Dot area shows how many answers fall in each group.



ACCURACY

88.8%

same in both views

CALIBRATION ERROR

0.099

95% interval 0.085 to 0.119

IF YOU AUTOMATE ABOVE 90%

1,152 automated

about 108 of them wrong

AVERAGE CONFIDENCE

98.6%

temperature 1.00 (none)

Figure 4: On the 7B instruct model with 10 options, 1,152 of 1,200 answers have confidence above 90% as the model gives them, and 90.6% of those are right. After temperature scaling (below), 872 do, and 97.4% of those are right. Accuracy is 88.8% in both views. Shown: model: 7B instruct; probabilities: as the model gives them.

against 13.6 s for full-option scoring. For a single question, reading the letters costs about one pass over the prompt (0.14 s at 10 options), and generating the letter instead adds one more step (0.17 s).

15 Is a calibrated model always useful?

No. Calibration says the confidence matches how often the model is right. It says nothing about how often the model is confident. **A calibrated model that knows little gives low-confidence answers most of the time (near 50% on a yes/no question), and almost none of them clear a 90% bar.**

Take the yes/no question “is this a prompt injection?”, asked of all 662 messages. The 1.5B instruct model had an ECE of 0.04 after scaling, and an accuracy of 62.8%, against 60% for always answering “not an injection”. (For yes/no questions I used Platt scaling, which fits two numbers instead of one.) The 7B instruct model reached 93.0% when the prompt said what the assistant is for.

The figure below turns that into a calculation. It assumes a model that is perfectly calibrated, varies only its accuracy, and counts how many cases clear a 0.90 bar.

So judge a decision model by how many cases clear your bar, and set the bar from your costs.

16 Can a small model be trained to decide?

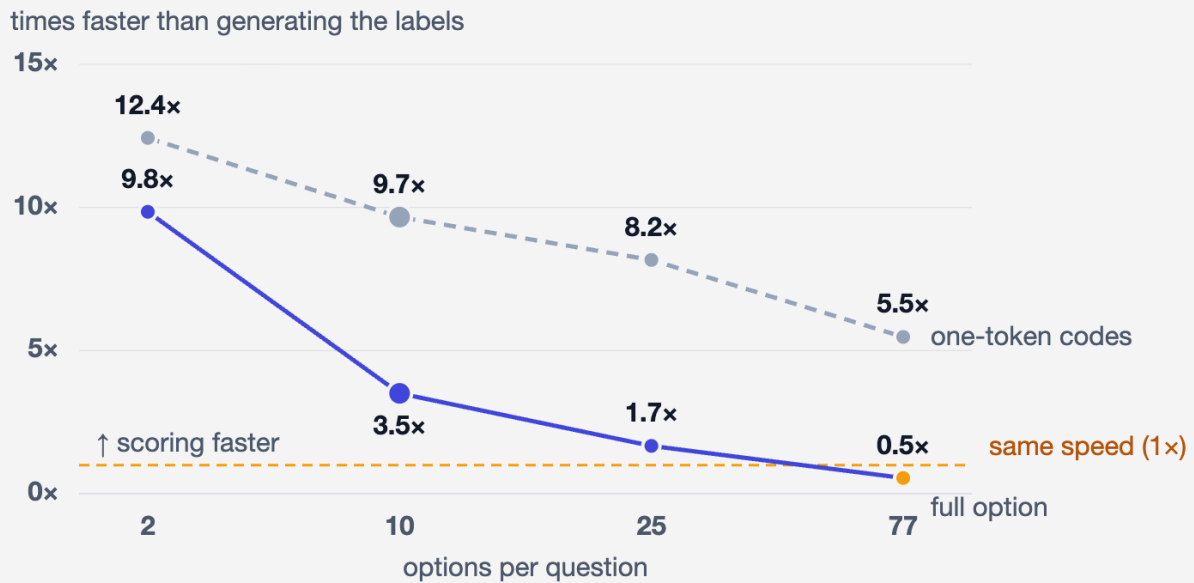
Everything above used stock models, and the smaller they were the worse they read option letters. A 0.5B model is small enough to run almost anywhere, so the practical question is whether a short training run can teach it the readout. This section trains Qwen2.5-0.5B-Instruct on three decision tasks, between one and 38 minutes each on one GPU, and measures what changed on held-out data[12].

17 What does the stock model do?

The first task is a game, because a game shows a decision every tick. Snake on a 10x10 board: the program writes the state as text (head, food, which of the four moves are safe and how far each is from the food) and lists the moves as options A to D. The model reads one letter’s probability per move. A small search that plans a path to the food and checks that the snake can still reach its own tail afterwards is the **solver**; it labels any position with its best move or moves, so labels cost nothing.

With one-token codes, scoring stays ahead of generating a label at every option count

Qwen2.5-7B instruct. Each request asks 5 questions about the same 256-token state, each with the number of options on the x-axis.



At 10 options: generating the labels took 1.82 s, full-option scoring 0.52 s, one-token codes 0.19 s.

Figure 5: This is Figure 3 with one-token codes added as the dashed line. Reading one code per option removes the cost of reading every option in full, so scoring stays ahead of generating a label at every option count. The code timings come from a separate run on the same GPU type; the accuracy cost is in the table above. Shown: questions: 5 questions; model: Qwen2.5 7B.

The stock 0.5B has a habit, and it depends on the prompt. With the moves always listed as A up, B down, C left, D right, as a game controller would list them, it picks “A” on 86% of 600 positions and drives into the top wall after about 6 ticks in every game. With the letters shuffled per position it agrees with the solver 42.2% of the time, against 25% for a random pick, and splits its picks between “D” (53%) and “A” (38%). On Banking77 messages with 10 options the same model picks the tenth letter, “J”, 40% of the time[12]. This is the option-token bias measured across many models by Zheng and colleagues[25]: a prior on the letters themselves, not on what they stand for.

18 How is the model trained?

Each task has its own source of labels, and each test set is kept apart from training, so the numbers below measure generalisation and not memorisation. Snake positions are labelled by the solver, and the test positions come from games with different seeds. Banking77

uses the dataset’s own train and test splits, and 17 of the 77 intents are held out of training entirely, so the test can show whether the model learned to read options or learned the intents. The prompt-injection messages from the earlier section are split in half.

In every training example the letters are shuffled: which letter stands for which move, intent or answer is random per example. A model trained this way cannot lower its loss by preferring a letter; it has to read the option text. The loss is **cross-entropy** on the answer letter: the negative log-probability the model gives the correct letter, at the same position the readout reads. Training uses a **low-rank adapter** (LoRA), small trainable matrices added alongside the model’s frozen weights, here 8,798,208 trainable parameters, for one **epoch**, one pass over the training data. The three runs together cost about \$0.86.

At 75% accuracy, 20% of cases clear a 0.90 bar and the rest go to review

This is a calculation in which every model is perfectly calibrated and only accuracy varies. Dashed lines show third-party accuracies for stock models (n = 24).

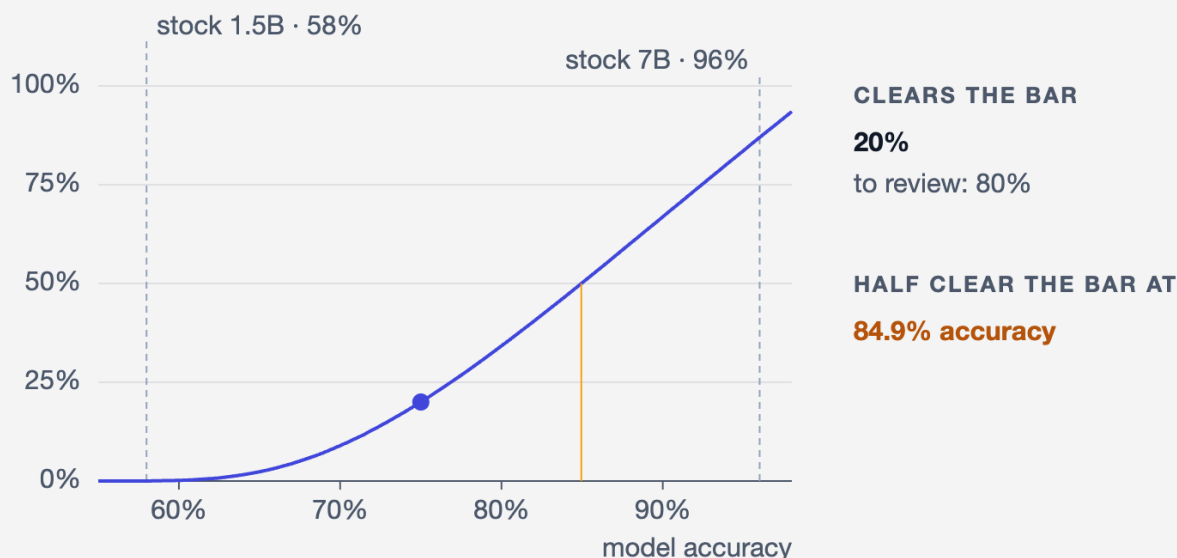


Figure 6: This figure is a calculation, not a measurement. Every model in it is perfectly calibrated and only accuracy varies. With a 0.90 bar, 60% accuracy lets 0.2% of cases through, 75% lets 20%, and 90% lets 67%. Dashed lines show accuracies a third party measured for stock 1.5B and 7B models on 24 cases[22]. Shown: model accuracy on the task: 75%; automate when probability ≥ 0.90 .

19 What changes during training, and does calibration improve with it?

On Snake the agreement with the solver goes from 42.2% to 100.0% on held-out 10x10 positions and to 100.0% on the 12x12 board (from 41.0% before). Across 3 repeat runs of the recipe the final held-out agreement is 99.3% to 100.0% (99.7% to 100.0% on the 12x12 board). ECE moves from 0.188 to 0.000, but at 100% accuracy with a mean confidence of 100% the error is zero by construction, so Snake says nothing about calibration; Banking77 does.

On intents it trained on, the trained 0.5B reads the options better than the stock 7B did in the earlier section (88.6% at 10 options, 58.1% at 77), though the 7B's option sets were drawn differently and its messages include the unseen intents, so compare loosely. The unseen intents are the closer measure of the readout itself, since at 10 and 25 options the model can also rule out familiar intents; the gap to the trained intents is what the model learned about those intents. Calibration follows the same split: the trained model is confident about intents it has seen and overconfident about intents it has not, even after temperature scaling (ECE 0.099). That is the

case the earlier sections made for labelled examples from your own task.

On prompt injection, a two-way decision where the letter mapping is trivial, accuracy goes from 48.1% to 99.3% and ECE from 0.334 to 0.009 (0.002 after temperature scaling), with the caveat about paraphrases in the settings above.

20 Does it play?

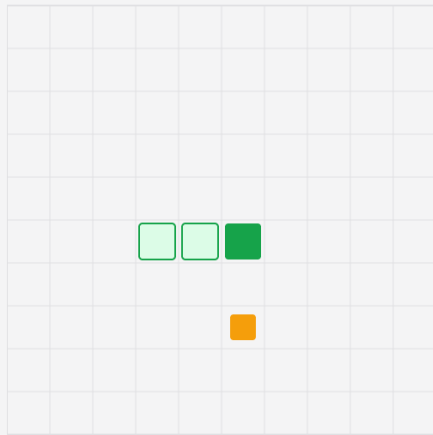
Each player played 10 games from the same seeds, 300 ticks at most. On the 10x10 board the stock model scores 0.0 and dies after 6 ticks; the trained model scores 31.5 and survives the full 300 ticks in 3 of 10 games; the solver scores 35.0. On the 12x12 board the stock model scores 0.0, the trained model 30.7 (survives 8 of 10), the solver 32.1.

That first run has a hole, and the held-out test did not show it. Its training positions came from the first 60 ticks of each game, so the longest snake it ever saw had 14 cells, and the held-out positions were harvested the same way. In play the snake grows past that, and every death of the first trained model came at a length of 25 to 41 cells: positions from a distribution it had never seen, which a

Same seed, tick 0: stock model score 0, trained model score 0

Recorded games, 10x10 board; training positions from the first 60 ticks of games. Bars: each model's move probabilities this tick.

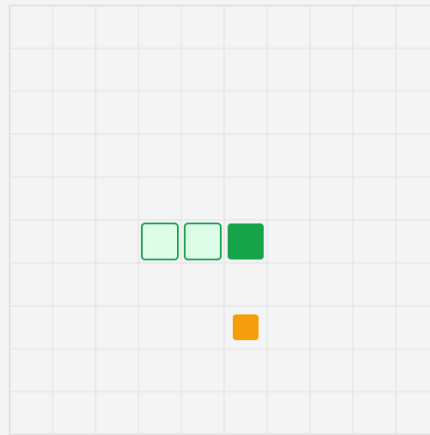
BEFORE: STOCK MODEL



A. up 53%
 B. down 6% · **best (solver)**
 C. left 9%
 D. right 32%

move taken: up · game 1 of 2 · stock over 10 games: mean score 0.0

AFTER: FINE-TUNED



A. up 0%
 B. down 100% · **best (solver)**
 C. left 0%
 D. right 0%

move taken: down · trained over 10 games: mean score 31.5, survived 3 of 10

Figure 7: The stock and trained models play the same Snake game side by side: the stock model (left) drives into the wall within a few ticks, while the trained model (right) keeps playing. On the text tasks, the bars show each model's option probabilities on the same held-out example, with the correct option tagged. Every recording was made on a server; the page runs no model. Shown: use case: Snake; board: 10x10; training positions: first 60 ticks of games; recorded game: 1; speed: normal.

Table 4: Banking77, 0.5B letter readout

Banking77, 0.5B letter readout	Stock	Trained
Trained intents: accuracy, 10 options	23.0%	96.2%
Trained intents: accuracy, 77 options	3.4%	81.0%
Unseen intents: accuracy, 10 options	15.8%	80.5%
Unseen intents: accuracy, 77 options	0.0%	49.3%
Trained intents: ECE	0.226	0.021
Unseen intents: ECE	0.266	0.160

test set drawn like the training set cannot reveal. Harvesting the training positions from full 300-tick games instead (87,986 positions, snakes up to 37 cells) and training once more with the same recipe gives a model that scores 35.2 on 10x10 and survives 10 of 10 games,

and 33.7 with 10 of 10 on the 12x12 board. With one run each, the data is the likely cause, not a proven one. With ten games per player and scores that spread over ten points, the trained model and the solver are within the same range, not ranked. To see the two trained

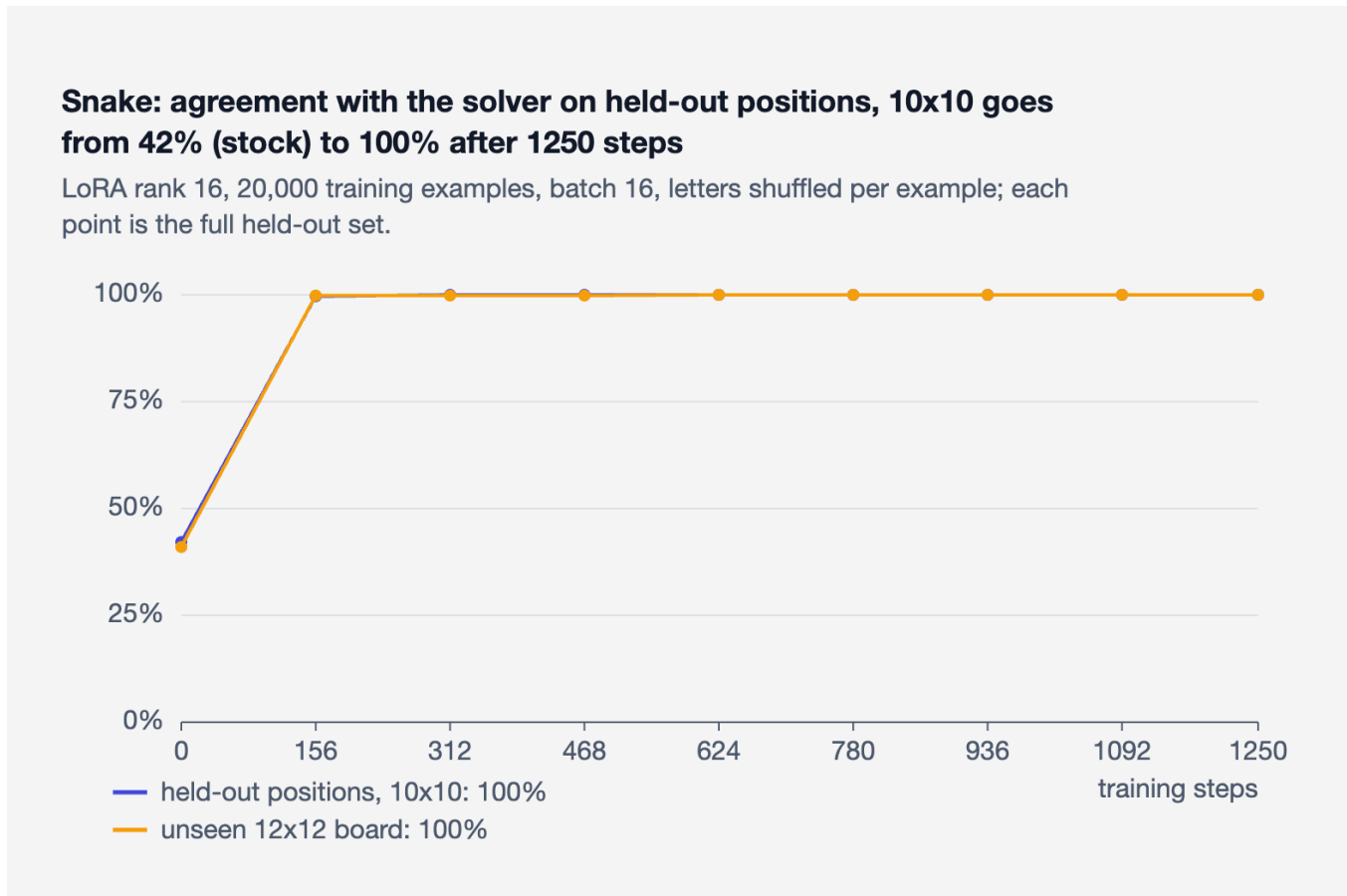


Figure 8: Each line shows a held-out metric at every training checkpoint, with step 0 as the stock model. The second line is the harder test set: the 12x12 board for Snake, and intents never seen in training for Banking77. Shown: use case: Snake; metric: accuracy.

models side by side, switch “training positions” in Figure 8 and step to the late ticks of a game.

A 0.5B model that read four shuffled options little better than chance learned to read them in one pass over solver-labelled positions, and the learning carried to a board size and to intents it had never seen. What it learned is to pick from a description that already lists which moves are safe and how far each is from the food; the description is the program’s work. The one failure in play was most likely a data failure, and a test set drawn the same way as the training set inherits its blind spots.

This is one model, one adapter rank and one epoch, with no seed or hyperparameter sweep: what a first attempt gets, not the best a 0.5B model can do.

21 What we know about Jev

TypeSafe has said little about how Jev works. When asked on Hacker News, their CEO described the architecture as “close to the chest for now”, and made only one statement about the mechanism: “strings (and all sequential data structures) are not allowed at all - this is how we make sure all outputs can be computed in parallel”[24].

That rules out generation, and it is why I think Jev is most likely a form of scoring: a model that reads probabilities for a fixed set of options instead of writing an answer.

What TypeSafe does say is that Jev is trained to be calibrated, using a method they call Reinforcement Learning for Calibrated Decisions (RLCD), aimed at “answers with epistemically honest probabilities”[2]. The method has not been published. **Scoring any model gives you speed and answers that are always on the list, but an instruct model’s confidence needs extra work before you can trust it: training for calibration, as TypeSafe says they did with RLCD, or labelled examples from your own task, as temperature scaling needed in our experiments.** It is also the part TypeSafe has published the least about. They publish no reliability diagram or ECE[9], and their one accuracy-by-confidence number is 27 of 30 answers correct above 0.9 confidence, on an older model, which leaves a wide 95% interval (74% to 97%)[3]. Their own benchmark also scores Jev against labels “generated via an average of the responses of GPT-6 Astra and Claude Fable 5.1”[9], so its accuracy numbers tell us how often Jev agrees with those two models, not how often it is right.

The best outside evidence comes from a third party who ran Jev on the same 662 prompt-injection messages we used[15]. With the task described in the prompt, they measured 96.5% accuracy and the 0.0588 ECE mentioned earlier; without it, 89.7% and 0.0928. Our stock 7B instruct model, with no special training, reached 93.0% with an ECE of 0.061, and 86.9% without the task description[13]. The prompts differ and the measurements were made by different people, so I would read this as indicative only. Still, it suggests a stock model gets within a few points of Jev’s accuracy on this task, with similar raw calibration.

One question stays open: whether Jev reads every option in full, as our scorer does, or reads all of them from one position, like the letter readout. Two details point to the second: choices are capped at 255 options, and TypeSafe’s launch post says that past the cap Jev falls back to “scoring independently then making an explicit choice”[2]. Timing requests from 2 to 255 options would be strong evidence: if latency stays flat, Jev reads one position per question.

22 Conclusion

In this article, we looked at how decision models like Jev work, in four steps: start with a decision, represent it as a sentence to complete, score the probability of each option, and normalise the results into a probability for every option. Ideally those probabilities are calibrated, so the confidence can be trusted.

We then tested the idea on open Qwen2.5 models. **Scoring was 7× to 54× faster than generating the probabilities, and about 9.8× faster than generating a label with 2 options (7B model, 5 questions).** Its lead shrinks as the options grow, and on the 7B at 77 options it was slower, unless each option gets a one-token code, which kept scoring ahead at every option count with little loss of accuracy on the 7B model.

Calibration turned out to matter as much as speed. Base models were close to calibrated, but instruct models were overconfident: at 77 options, the 7B instruct model’s average confidence was 95%, while it was right 64% of the time. Temperature scaling fixed most of that, but it needs labelled data, and this is also where training for calibration, as TypeSafe says it does for Jev, could help.

Training a small model also worked. After a short training run with shuffled letters, a 0.5B model went from 7.1% to 66.5% accuracy on Banking77 intents it had never seen, and from 42.2% to 100.0% agreement with the solver on held-out Snake positions.

If you want to build on this, the sequence I would follow is:

- (1) **Build an eval set** from your own task, in the same sentence-completion format, with labelled answers.
- (2) **Measure accuracy and calibration**, not just accuracy: a reliability diagram and ECE on that set.
- (3) **Fix what the measurements show**: temperature scaling if the confidence is off and you have labels, fine-tuning if accuracy is too low, and one-token codes if you have many options and need speed.
- (4) **Set the confidence threshold from your costs**, and send everything below it to a person or a follow-up question.

No Jev API calls were made for this article. If you have tried Jev or built something similar, tell me what breaks.

References

- [1] Together AI. 2026. How to train your own Jev: fine-tunes Qwen3.5-4B with LoRA (rank 8, one epoch, about \$17) on 37,840 examples (the total in its data table and repository README; its prose says 38,340) to answer with one option letter; the letter is generated under a regex constraint and evaluation reports accuracy only (code: github.com/togethercomputer/tev1) (accessed 2026-09-24). <https://www.together.ai/blog/how-to-train-your-own-jev>
- [2] TypeSafe AI. 2026. Introducing System One Models & Jev (accessed 2026-09-19). <https://typesafe.ai/blog/introducing-system-one-models-and-jev>
- [3] TypeSafe AI. 2026. TypeSafe cookbook: Classification using confidence (60 SEC filings, 75 groups, jev-1.12) (accessed 2026-09-19). https://docs.typesafe.ai/cookbooks/classification_using_confidence
- [4] TypeSafe AI. 2026. TypeSafe docs: Choice (accessed 2026-09-19). <https://docs.typesafe.ai/primitives/choice>
- [5] TypeSafe AI. 2026. TypeSafe docs: Known limitations of jev-1.13 (accessed 2026-09-19). <https://docs.typesafe.ai/model-jaggedness/jev-1.13>
- [6] TypeSafe AI. 2026. TypeSafe docs: Models (jev-1.13.0 price, limits, text-only input) (accessed 2026-09-19). <https://docs.typesafe.ai/models>
- [7] TypeSafe AI. 2026. TypeSafe docs: Score (accessed 2026-09-19). <https://docs.typesafe.ai/primitives/score>
- [8] TypeSafe AI. 2026. TypeSafe docs: System One (accessed 2026-09-19). <https://docs.typesafe.ai/concepts/system-one>
- [9] TypeSafe AI. 2026. TypeSafe workflow evals (accessed 2026-09-19). <https://evals.typesafe.ai/>
- [10] deepset. 2023. deepset/prompt-injections: 662 messages, 263 labelled as injections (accessed 2026-09-20). <https://huggingface.co/datasets/deepset/prompt-injections>
- [11] Victor Dibia. 2026. Designing Multi-Agent Systems, Chapter 5: Building Computer Use Agents (action generation, interface representation, action execution). <https://multiagentbook.com>
- [12] Victor Dibia. 2026. Fine-tuning made for this page, 2026-09-24/25: our own low-rank adapter (rank 16, all seven projection matrices) on Qwen2.5-0.5B-Instruct for three decision tasks (Snake with solver labels, Banking77 intents with 17 intents held out of training, deepset prompt injections), letters shuffled per example, one A10-class GPU (A10 or A10G) on Modal; checkpoint evaluations on held-out data, recorded games before and after. Code (train.py, baseline.py, snake.py), the split report, the run files and the commands to reproduce: github.com/victordibia/explainer-experiments, folder jev/finetune. <https://github.com/victordibia/explainer-experiments/tree/main/jev/finetune>
- [13] Victor Dibia. 2026. Measurements made for this page, 2026-09-20: our own option scorer on Qwen2.5 1.5B and 7B (base and instruct), one NVIDIA A10G on Modal, 16-bit weights, Hugging Face transformers 4.46.3, one request at a time, median of 3-10 repeats after 2 warmups. Code and raw JSON: github.com/victordibia/explainer-experiments, folder jev/scoring. <https://github.com/victordibia/explainer-experiments/tree/main/jev/scoring>
- [14] fcoury. 2026. openjev at commit b4782a6: option scoring in one forward pass with shared-input KV reuse, Qwen3.5-4B on an RTX 3090 (third-party, MIT) (accessed 2026-09-19). <https://github.com/fcoury/openjev>
- [15] Gaurav-Gosain. 2026. jev-sec-bench at commit fdb16b9: Jev on the 662 deepset prompt-injection messages, with calibration error (third-party measurement) (accessed 2026-09-19). <https://github.com/Gaurav-Gosain/jev-sec-bench>
- [16] Pleiss Guo, Sun, and Weinberger. 2017. On Calibration of Modern Neural Networks (‘modern neural networks, unlike those from a decade ago, are poorly calibrated’; temperature scaling ‘is surprisingly effective’). <https://arxiv.org/abs/1706.04599>
- [17] Shwartz Holtzman, West, Choi, and Zettlemoyer. 2021. Surface Form Competition: Why the Highest Probability Answer Isn’t Always Right (‘different surface forms compete for probability mass, even if they represent the same underlying concept’). <https://arxiv.org/abs/2104.08315>
- [18] Araki Jiang, Ding, and Neubig. 2021. How Can We Know When Language Models Know? On the Calibration of Language Models for Question Answering. TACL 9 (‘We examine three strong generative models – T5, BART, and GPT-2 – and study whether their probabilities on QA tasks are well calibrated, finding the answer is a relatively emphatic no.’) (abstract read 2026-09-23). <https://arxiv.org/abs/2012.00955>
- [19] mmastrac. 2026. djev: the Jev decision API on DiffusionGemma-26B, a discrete diffusion language model (‘one denoise step gives a distribution over each slot’; ‘each label must be a single token’); deployment wrapper at github.com/taeold/djev-run (third-party, not run by us) (accessed 2026-09-23). <https://github.com/mastrac/djev>
- [20] OpenAI. 2023. GPT-4 Technical Report, Figure 8 and the text above it: ‘the pre-trained model is highly calibrated ... after the post-training process, the calibration is reduced’; ‘The post-training hurts calibration significantly’. <https://arxiv.org/abs/2303.08774>
- [21] PolyAI. 2020. Banking77: customer messages to a bank, each labelled by people with one of 77 intents (test split read from the mteb/banking77 mirror, 3,076 messages; 1,200 sampled with a fixed seed) (accessed 2026-09-20). <https://>

- huggingface.co/datasets/PolyAI/banking77
- [22] rorshopping. 2026. jev-on-a-laptop at commit 5821d91: typed decisions on stock 1.5B-8B models, 16 GB M5 MacBook Air (third-party measurement) (accessed 2026-09-19). <https://github.com/rorshopping/jev-on-a-laptop>
- [23] Qwen team and Alibaba Cloud. 2024. Qwen2.5-7B-Instruct model card and config.json: 28 layers, hidden size 3,584, 28 attention heads, 4 key-value heads, vocabulary 152,064, 32,768-token context (accessed 2026-09-20). <https://huggingface.co/Qwen/Qwen2.5-7B-Instruct>
- [24] via Hacker News TypeSafe AI CEO. 2026. Hacker News thread 'Introducing System One Models and Jev', comments by user CompleteSkeptic ('CEO here'): ids 49719122 (no sequential outputs), 49718824 (architecture), 49718407 (noul = bernoulli) (accessed 2026-09-19). <https://news.ycombinator.com/item?id=49717558>
- [25] Meng Zheng, Zhou, Zhou, and Huang. 2024. Large Language Models Are Not Robust Multiple Choice Selectors (selection bias comes largely from token bias toward specific option ids, e.g. 'A'; PriDe debiasing). <https://arxiv.org/abs/2309.03882>