RUN JEV-STYLE SYSTEM ONE MODELS LOCALLY

A Complete Guide to Ollaya and Local Decision Models

LUCAS NAVARRO

Run Jev-Style System One Models Locally

A Complete Guide to Ollaya and Local Decision Models

Steve Publications

This book is available at
https://leanpub.com/runjev-style-system-one-models-locally

This version was published on 2026-09-26

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.

Contents

A Complete Guide to Ollaya and Local Decision Models

This book teaches you how to run decision models, a distinct class of AI systems optimized for fast, structured judgments, entirely on your own hardware using the open-source Ollaya project. You will learn what Jev-style decision models are, how they differ from conventional large language models, and why that difference matters for building reliable software. From installation to production deployment, this guide walks you through installing Ollaya, pulling models, defining typed questions, calling the API from Python and JavaScript, integrating with AI agents via MCP, designing decision schemas that work in the real world, measuring performance, and operating a local decision service at scale. By the end, you will be able to take a problem that currently uses an LLM, cost money, and leaks data to the cloud, and replace it with a local model that answers in milliseconds with typed outputs your code can act on directly.

Introduction

Imagine you are building a customer support system. A ticket arrives: "I was charged twice this month and want a refund." You need to route it to the right team, assess urgency and decide whether to escalate. Today, many teams hand this to a large language model with a prompt like "Classify this ticket and give me a JSON response with intent, severity and whether to escalate." The model writes back prose or structured text that you then parse, hope is valid and run through your own validation logic. It takes hundreds of milliseconds. You pay per token. Customer data crosses network boundaries. Sometimes the JSON is malformed. Sometimes the model hallucinates a category you never defined. You add retry logic, schema validators and error handling. The whole stack feels brittle.

Now imagine a different approach. You send that ticket text and a set of typed questions to a decision model. The model answers every question in parallel, in a single forward pass, returning nothing but structured values: a chosen category from your list, a severity score from your rubric, a probability for escalation. No prose. No parsing. No malformed JSON. The answer arrives in single-digit or low double-digit milliseconds. You pay nothing per call. Your data never leaves your machine. The model cannot output anything outside the options you defined. If it is uncertain, its probabilities tell you exactly that, and you route to a human.

This is not a hypothetical future. It is what Jev-style decision models do. And thanks to open-source projects like Ollaya, you can run them locally on your own hardware, with all the control, privacy and cost advantages that brings.

This book will take you from understanding what decision models are and why they exist, to installing and running them locally, to integrating them into real production systems. It is written for software engineers, ML engineers, and technically minded developers who already know what LLMs are but have not yet explored this alternative paradigm. If you are tired of prompting, parsing and validating LLM outputs for tasks that should be simple decisions, this book is for you.

What is a decision model, exactly, and how is it different from an LLM? Why does TypeSafe call Jev a "System One" model? What does "calibrated probability" mean in practice? How does Ollaya work under the hood? How do you go from zero to a production local decision service? These are the questions we will answer, one by one, in the chapters that follow.

The structure of this book is intentional. We begin with fundamentals: the concepts behind decision models, their relationship to LLMs, and the architecture that makes them fast and reliable. Then we move to hands-on practice: installing Ollaya, running your first model, defining questions, and calling the API. After that comes integration: connecting decision models to your Python and JavaScript code, migrating from cloud Jev to local execution, and designing decision schemas that work in production. We then cover advanced topics: agent integration via MCP, hybrid architectures that combine decision models with LLMs, evaluation and benchmarking, and production deployment patterns. Throughout, every example is copy-paste-ready and based on real commands, real APIs, and real model names from the current Ollaya release.

Here is the map:

- Chapters 1 through 3 explain the decision model paradigm from first principles and compare it to LLMs across architecture, behavior, performance and reliability.
- Chapters 4 through 8 cover Ollaya itself: what it is, how to install it, how to pull and run models, how to create custom models with Modelfiles, and how to use every CLI command.
- Chapters 9 through 13 focus on integration: running the local daemon, using the API, connecting from Python and JavaScript, and migrating from cloud Jev to Ollaya.
- Chapters 14 through 18 walk through real applications: designing decision schemas, building a support triage system, creating an intent router, implementing moderation, and architecting hybrid systems.
- Chapters 19 through 24 address production concerns: MCP and agent integration, evaluation, performance tuning, observability, security and deployment.
- Chapter 25 brings it all together and points toward where this technology is going.

Let us begin.

Chapter 1: What Is a Decision Model?

Decision models are not a marketing rebranding of existing technology. They represent a fundamentally different approach to AI-assisted software, built for a different job than the language models most developers use today. To understand why they exist and why they matter, we need to start with the problem they solve.

The Problem LLMs Were Never Built to Solve

Large language models are trained to predict the next token in a sequence. That objective gives them an incredible ability to generate coherent text, complete patterns, and reason about the world in ways that feel almost human. When you ask an LLM to summarize an article, write code, or answer a question, it generates a response token by token, each choice conditioned on everything that came before. This autoregressive generation is why LLMs feel so natural and flexible.

But autoregressive generation is not the right tool for every job. Consider a scenario where you need your software to make a decision: classify a support ticket, detect fraudulent activity, route a request to the correct service, or evaluate whether content violates your policy. In each case, you do not want prose. You want a typed value from a known set of options, with a probability attached so you can decide whether to act automatically or escalate to a human.

When you use an LLM for this, you typically do the following:

1. Write a prompt that includes the input data and instructions like "Respond with only valid JSON matching this schema."
2. Generate text token by token until the model produces what you hope is a valid JSON object.
3. Parse the JSON and validate that the fields match your schema.
4. Handle errors: sometimes the model adds extra text, sometimes it closes brackets incorrectly, sometimes it invents a category you never defined.
5. Optionally add retry logic, function calling, or output parsers to reduce failures.

This chain of steps is fragile. An LLM trained to predict tokens has no mathematical guarantee that it will respect your schema. The model might hallucinate an option you did not list. The output might be syntactically broken. Even when it works, you are paying for every token of generated text and waiting for each token to appear before you can proceed. For structured decisions inside software, this is often overkill in terms of cost and complexity, and underwhelming in terms of reliability.

What you actually want is not a text generation model. You want something that takes input data and a set of typed questions, and returns typed answers with calibrated probabilities. You want a "smart if statement" that understands natural language but returns values your code can use directly. That is what decision models are.

Decision Models vs. Language Models: Different Jobs

The distinction between decision models and language models is not about quality or capability. It is about specialization. A decision model is built from the ground up to answer a specific class of questions, and its architecture, training objective, and output format are all optimized for that task.

Consider a customer support ticket again. With a language model, the workflow might look like this:

- Input: raw ticket text plus a prompt asking for classification
- Process: autoregressive text generation, token by token
- Output: a prose response or JSON blob that you must parse
- Latency: hundreds of milliseconds to seconds, depending on output length
- Cost: per-token pricing for input and output
- Reliability: depends on prompt quality, model temperature, and validation code

With a decision model, the workflow is different:

- Input: ticket text plus a set of typed questions with defined options
- Process: single forward pass through the model, evaluating all questions in parallel

- Output: structured values only, namely chosen categories, scores and probabilities, matching your exact schema
- Latency: single-digit to low double-digit milliseconds
- Cost: zero per call when running locally; minimal when hosted
- Reliability: outputs are mathematically constrained to your defined options

The decision model cannot output anything outside the schema you provide. If you define a choice question with three options, namely billing, technical and account, the model will return one of those three, never anything else. This is not enforced by post-processing or schema validation. It is guaranteed by the model's architecture: the output space is literally the set of options you defined.

Language models remain essential for tasks that genuinely require generation: writing documents, composing code, translating, explaining concepts, brainstorming. Decision models are essential for tasks that require judgment: classification, routing, scoring, verification, moderation. Modern AI systems will use both. The key insight is that they are different tools for different jobs, and using a generation model for a decision problem is often the wrong choice.

Why Structure Matters in AI Systems

Structure is the backbone of reliable software. When you write code, you use typed variables, defined interfaces, and explicit contracts so that components can be composed and failures can be detected. If one function is supposed to return a boolean but returns a string instead, your type system or runtime checks will catch it. You do not leave critical decisions to unstructured text parsing.

Yet many AI integrations today violate this principle. When you call an LLM from your application, you often receive free-form text that your code then attempts to parse. This creates a mismatch between the deterministic world of software engineering and the probabilistic world of AI. You introduce brittleness into your system: edge cases where parsing fails, unexpected formats, hallucinated values that break downstream logic.

Decision models are designed to bridge this gap. They keep the intelligence and flexibility of AI while returning outputs that fit naturally into typed software. A decision model's answer is not prose that you must interpret. It is a

value of a known type with a known range, plus a probability or confidence score that tells you how reliable that value is. Your code can switch on these values, compare them against thresholds, compose them into larger decisions, and log them for audit purposes. The entire pipeline remains type-safe from model output to business logic.

This structural guarantee changes how you think about AI in your architecture. Instead of treating AI as an unstructured black box that you must carefully guard against, you can treat a decision model like any other service in your system: it accepts inputs of a defined shape, performs computation, and returns outputs of a defined shape. The only difference is that those outputs include uncertainty estimates, and your code should respect those estimates rather than pretending every answer is certain.

Enter the Jev Paradigm

The term "Jev-style" decision model refers to the class of models popularized by TypeSafe AI's Jev, which was announced in September 2026. TypeSafe calls Jev a "System One model," a name inspired by psychologist Daniel Kahneman's distinction between fast, intuitive thinking (System 1) and slow, deliberate reasoning (System 2). System One models are designed to make the kind of snap judgments that a knowledgeable human makes in a second, given the right context: routing, classification, scoring, verification. They are fast, structured and optimized for software automation rather than conversation.

Jev's core interface is simple. You send it:

1. A state: a string, JSON object, or array of text containing the context to evaluate
2. One or more typed questions, each with a defined type and a set of options

The model evaluates every question against the state in a single parallel pass and returns structured results. Each question is answered independently, in isolation, so one question's answer does not affect another's. This prevents context rot and makes the behavior predictable.

Jev defines three question types, which TypeSafe calls primitives:

- **Choice**: select one option from a list. For example, "What is the customer's intent?" with options like billing, technical, account, other. The

model returns the chosen option, the full probability distribution across all options, and a confidence score.
- **Score**: evaluate along a rubric. For example, "How severe is this issue?" with levels like low, medium, high, critical. The model returns the selected level, probabilities for each level, and confidence.
- **Noul**: answer a yes/no question with a probability. For example, "Should this be escalated?" The model returns a probability between 0 and 1, where 1 means "yes" and 0 means "no." The probability itself serves as the confidence indicator.

Every answer is type-safe. The model cannot return a category you did not define, a score level outside your rubric, or a probability outside the 0-1 range. If the model is uncertain, its probability distribution will be spread across multiple options, and its confidence score will be low. This gives your code an honest signal about when to act automatically and when to involve a human.

Jev is proprietary and hosted by TypeSafe. However, the paradigm it represents, namely fast, typed, calibrated decision-making with a single-pass architecture, has inspired several open-source implementations. Laya, decider and others are open-weight models that follow the same design principles. Projects like Ollaya make it possible to run these models locally, giving developers control over their data, costs and infrastructure while maintaining the structural guarantees that make decision models attractive.

Jev is not the only example of this paradigm, and the concept predates TypeSafe's announcement. But Jev brought widespread attention to decision models as a distinct category and provided a concrete API specification that others can implement. When we say "Jev-style" decision model in this book, we mean any model that:

1. Accepts state and typed questions as input
2. Returns typed answers from predefined option sets
3. Provides calibrated probabilities or confidence scores
4. Evaluates questions in parallel, not through autoregressive text generation
5. Produces outputs that are structurally guaranteed to match the defined schema

Whether the model is Jev itself, Laya, decider, or another implementation, the design principles are the same. The open-source alternatives are not clones of Jev. They have different weights, different training data, and different accuracy profiles. But they share the fundamental architecture and interface that make the decision model approach valuable.

What You Will Learn From This Book

By the time you finish this book, you will be able to:

- Explain what a decision model is, how it differs from an LLM, and when to use each
- Install Ollaya on your machine, pull models, and run decisions locally
- Define typed questions with Choice, Score and Noul types for your specific use case
- Create custom model configurations using Modelfiles that embed questions and parameters
- Call the local decision model API from Python, JavaScript, and curl
- Integrate decision models into production applications like support triage, routing and moderation
- Design decision schemas that balance accuracy, granularity and maintainability
- Set confidence thresholds that trigger escalation, review, or automatic action
- Combine decision models with LLMs in hybrid architectures where each does what it does best
- Connect decision models to AI agents via the Model Context Protocol
- Evaluate model accuracy, calibration and performance with real metrics
- Deploy Ollaya in production with proper observability, security and resource management
- Migrate existing cloud-based Jev or TypeSafe integrations to local Ollaya execution

This is not a theoretical exercise. Every chapter includes working code, real commands, and practical guidance you can apply immediately. The next chapter will go deeper into the first-principles design of Jev-style decision models and explain the specific concepts, including state, questions, choices, scores, confidence, calibration and single-pass inference, that make this approach possible.

Chapter 2: Jev-Style Decision Models: From First Principles

To use decision models effectively, you need to understand the concepts that define them. This chapter explains the core primitives, the mechanics of single-pass inference, and the way decision models produce calibrated probabilities. We will focus on what is documented about the Jev paradigm and its open-source implementations, distinguishing established facts from reasonable inference where necessary.

The Jev System: Origins and Purpose

Jev is TypeSafe AI's first public System One model, announced on September 15, 2026. TypeSafe was founded by Diogo Almeida, an OpenAI co-inventor, and raised a 40 million dollar seed round led by DCVC alongside its launch. Forbes reported a valuation of 200 million dollars following the round. The model is named after William Stanley Jevons, a 19th-century economist known for his work on marginal utility and efficient resource allocation, as well as a reference to Kahneman's System 1 thinking, which means fast, intuitive judgment rather than slow, deliberate reasoning.

TypeSafe describes System One models as a class of AI models built to make fast, structured decisions that software can use directly. The key word is "structured." Unlike LLMs that generate prose, a System One model returns typed values and probabilities that your code can consume without parsing or validation. TypeSafe trained Jev using a method called Reinforcement Learning for Calibrated Decisions, abbreviated as RLCD. The specifics of RLCD have not been published in full technical detail. TypeSafe states that the training objective is to produce calibrated probabilities, which are numbers that genuinely reflect how likely an answer is to be correct, rather than just selecting the most probable option.

The core insight behind System One models is simple: most of what companies ask an LLM to do inside automation workflows is not writing, it

is deciding. Should this ticket go to billing or engineering? Is this user's behavior suspicious? Does this document match our compliance requirements? These are classification, scoring and verification tasks. They do not require prose generation, but that is exactly what LLMs do. System One models are specialized for the decision itself.

Jev is proprietary and available through TypeSafe's hosted API. However, the paradigm has inspired open-source implementations with the same interface and design principles. Ollaya runs several such models locally, including Laya, decider, NLI, and GLiClass. None of these are Jev itself. They have different weights, different training, and different accuracy characteristics. But they all implement the same basic interface: accept state and typed questions, return typed answers with probabilities.

State, Questions and Choices: Core Primitives

A decision model request consists of two parts: state and questions. Understanding these primitives is essential to using decision models effectively.

State is the context the model evaluates. It can be:

- A plain text string
- A JSON object with named fields
- An array of text strings

The state can be a support ticket, a user message, a document excerpt, a log entry, or any text-based input. TypeSafe's documentation specifies a context window of 64 thousand tokens for Jev, though the practical limit depends on the specific model and implementation. The model processes the entire state once and uses it as context for all questions in the request. This is different from an LLM conversation where each turn builds on previous turns. In a decision model, the state is the complete context, and questions are evaluated against it independently.

Consider this example state from a support system:

```
{
  "ticket": "I was charged twice for my subscription this month. My account
  ↪  shows two identical charges on the 15th and 16th. I want a refund for the
  ↪  duplicate charge.",
  "customer_tier": "premium",
  "account_age_months": 18
}
```

This state contains both the user's message and structured metadata about the customer. The decision model can use all of this information when answering questions.

Questions are the typed requests you pose about the state. Each question has:

- A unique ID
- A type (Choice, Score, or Noul)
- Instructions describing what the model should evaluate
- Criteria defining the options, rubric levels, or yes/no framing

Questions are evaluated in parallel against the same state. This means all questions see the same context, but one question's answer does not influence another's. This is a deliberate design choice. It prevents context rot, where a model's earlier answers drift or corrupt later reasoning, and makes the behavior predictable and composable. If you need to ask a follow-up question based on a previous answer, you make a second request.

Let us look at a concrete set of questions for our support ticket state:

```
{
  "intent": {
    "type": "choice",
    "instructions": "What is the primary customer intent?",
    "criteria": {
      "billing": "Issues related to charges, refunds, invoices, or payments",
      "technical": "Issues related to product functionality or bugs",
      "account": "Issues related to account management or settings",
      "other": "Anything that does not fit the above categories"
    }
  },
  "urgency": {
    "type": "score",
    "instructions": "How urgent is this request?",
    "criteria": {
```

```
      "low": "No immediate impact on the customer",
      "medium": "Customer is inconvenienced but has workarounds",
      "high": "Customer is blocked or financially impacted",
      "critical": "Requires immediate resolution within one hour"
    }
  },
  "should_escalate": {
    "type": "noul",
    "instructions": "Should this ticket be escalated to a human agent?",
    "criteria": {
      "yes": "Escalate if the issue is high or critical urgency, or involves
      ↪  financial discrepancy for a premium customer",
      "no": "Can be handled by automated processes or tier-one support"
    }
  }
}
```

Each question defines its own decision space. The intent question offers four options. The urgency question defines a four-level rubric. The escalation question is a yes/no decision. The model evaluates all three against the same state simultaneously and returns three independent answers.

Choices are the first and most common primitive type. A choice question asks the model to select one option from a list you define. The options are unordered, which means the model does not assume one is "correct" and others are "incorrect." You define the criteria for each option, and the model determines which criterion best matches the state.

Key characteristics of choice questions:

- The model returns exactly one selected option, never a combination
- The model returns the full probability distribution across all options
- The model returns a confidence score derived from the probability distribution
- The selected option is always one you defined; the model cannot invent categories

For our intent question above, the model might return:

```
{
  "type": "choice",
  "choice": "billing",
  "confidence": 0.92,
  "probabilities": {
    "billing": 0.89,
    "account": 0.07,
    "other": 0.03,
    "technical": 0.01
  }
}
```

The model selected "billing" with 89 percent probability. Confidence is 0.92, indicating the model is quite sure. Your code can act on this: route to the billing team automatically, log the decision, and proceed.

Scores are the second primitive. A score question asks the model to evaluate the state along a rubric or spectrum you define. Unlike choice, where options are unordered alternatives, score levels have an implicit ordering: low to high severity, poor to excellent quality, low to high priority.

Key characteristics of score questions:

- The model returns exactly one selected level
- The model returns the full probability distribution across all levels
- The model returns a confidence score
- The levels should form a coherent spectrum; the model uses their ordering

For our urgency question, the model might return:

```
{
  "type": "score",
  "score": 2.04,
  "confidence": 0.88,
  "legend": {
    "0": "low",
    "1": "medium",
    "2": "high",
    "3": "critical"
  },
  "probabilities": {
    "0": 0.01,
    "1": 0.09,
    "2": 0.85,
```

```
    "3": 0.05
  }
}
```

The score of 2.04 lands just above the high level (index 2), with a small tail of probability toward critical. Confidence is high, so your code can trust this assessment.

Noul is the third primitive. A noul question asks a yes/no question and returns a probability between 0 and 1, where 1 represents "yes" and 0 represents "no." The word "noul" is TypeSafe's term, likely derived from the binary nature of the decision (null as a conceptual root, though TypeSafe has not publicly explained the etymology).

Key characteristics of noul questions:

- The model returns a single probability value between 0 and 1
- This probability represents P(true), the likelihood that "yes" is correct
- There is no separate confidence score for noul; the probability serves as both the answer and the certainty indicator
- A probability of 0.7 means the model estimates a 70 percent chance the answer is yes

For our escalation question, the model might return:

```
{
  "type": "noul",
  "noul": 0.82
}
```

The model estimates an 82 percent chance this ticket should be escalated. You can set a threshold, say 0.75, and automatically escalate when the probability exceeds it. Below the threshold, you handle it normally. At a middle range, perhaps between 0.5 and 0.75, you flag it for human review.

These three primitives, namely choice, score and noul, are the building blocks of every decision model request. They are simple enough to understand intuitively but powerful enough to express complex decision logic. TypeSafe's documentation recommends keeping each question atomic: one question should represent one judgment a knowledgeable person could make in a second. Complex decisions are built by combining multiple atomic questions, not by asking one complicated question.

Typed Decisions and Probability Distributions

What makes decision models "typed" is that every answer belongs to a known type with a known range of values. This is not a soft guarantee enforced by prompt engineering or post-processing. It is a hard guarantee built into the model's output space.

For a choice question, the type is "one of the defined options." For a score question, the type is "one of the defined levels." For a noul question, the type is "a float between 0 and 1." Your code can rely on these types being correct. You do not need to validate that the answer is one of the allowed options because the model cannot produce anything else.

This typed guarantee is what allows decision models to integrate cleanly into typed software. In a Python application, you can define Pydantic models or dataclasses that exactly match the decision model's output structure. In TypeScript, you can define interfaces. When the model responds, you deserialize the JSON directly into your typed structures with no runtime validation needed. The schema is enforced by the model itself.

Every non-noul answer comes with a probability distribution. For a choice question with options A, B, C, and D, the model assigns a probability to each option. The probabilities sum to 1. The option with the highest probability is selected as the answer, but the full distribution tells you something important: how confident the model is in that selection.

If the distribution is concentrated, say option A has 0.92 probability and the others have less than 0.05 combined, the model is highly confident. If the distribution is spread evenly, with A at 0.35, B at 0.32, C at 0.23, and D at 0.10, the model is uncertain. In the latter case, your code should not act automatically. It should either escalate to a human, request more information, or use additional logic to break the tie.

Probability distributions are more informative than a simple "most likely answer." They allow your code to:

- Detect uncertainty and escalate appropriately
- Compare relative likelihoods of different options
- Aggregate probabilities across multiple questions
- Build probabilistic decision rules (for example, act automatically only if the top option exceeds 80 percent probability)

- Audit and evaluate model behavior over time

The probability distribution is not a post-hoc estimate. It is the fundamental output of the model. For encoder-based models like Laya, probabilities come from a contrastive scoring mechanism where each option is scored against the state, and the scores are converted to probabilities via a softmax function. For decoder-based models like decider, the answer is read directly from the logits of specific tokens corresponding to option labels, and those logits are converted to probabilities. In both cases, the probability distribution is intrinsic to the model's computation, not an afterthought.

Confidence, Calibration and Score Interpretation

Decision models provide two related but distinct measures of certainty: probability distributions and confidence scores. Understanding the difference is crucial for using them correctly.

The probability distribution tells you the model's assessment of each option's likelihood. For a choice question, if option A has probability 0.90, the model is saying there is a 90 percent chance A is the correct classification given the state and its training.

Confidence is a summary statistic derived from the shape of the probability distribution. TypeSafe derives confidence from the concentration of the distribution: when probability is heavily concentrated on one option, confidence is high; when it is spread across multiple options, confidence is low. Confidence is not the probability of the selected option. It is a separate measure indicating how decisive the model's judgment is.

This distinction matters because the model can be confident but wrong. If the probability distribution is heavily concentrated on the wrong answer, confidence will be high but the decision incorrect. Confidence tells you whether the model is decisive, not whether it is correct. Calibration tells you whether high confidence actually corresponds to high correctness.

Calibration is a property of the model's probabilities evaluated over many predictions, not a single prediction. A calibrated model produces probability estimates that match empirical accuracy. If a calibrated model assigns 90 percent probability to answers, approximately 90 percent of those answers

should be correct. If it assigns 60 percent probability, approximately 60 percent should be correct.

TypeSafe trained Jev using RLCD specifically to improve calibration. Laya's documentation reports an Expected Calibration Error of 0.081 after temperature fitting, compared to Jev's reported ECE of 0.246. ECE measures the average gap between predicted probabilities and actual accuracy across many predictions. A lower ECE means better calibration. Laya's lower ECE suggests its probabilities may be more trustworthy in aggregate than Jev's, though this is based on self-reported metrics and the comparison is not apples-to-apples since different datasets and methodologies may have been used.

Calibration is not perfect for any model. It is an aggregate property, not a guarantee for individual predictions. A single answer with 95 percent probability can still be wrong. Calibration tells you that over thousands of predictions, high-probability answers will be correct most of the time. This is why decision models are most valuable in batch processing and continuous operations, where aggregate reliability matters more than any single decision.

Practical implications for using probabilities and confidence:

1. Never treat a probability as absolute certainty. A 99 percent probability is not 100 percent.
2. Use confidence to detect uncertain predictions. Low confidence should trigger review or escalation.
3. Use calibration to understand aggregate reliability. If your model reports 80 percent probability on average for decisions you audit, and 80 percent are correct, it is well-calibrated for your use case.
4. Validate calibration on your own data. Published calibration metrics are useful reference points, but your actual accuracy depends on your specific domain and data.
5. For noul questions, treat the probability as the primary signal. A probability of 0.6 for "should escalate?" is different from a probability of 0.9. Set your threshold based on the cost of false positives versus false negatives.

Single-Pass Inference and the No-Generation Constraint

The term "single-pass inference" is central to understanding why decision models are fast. In a single-pass inference architecture, the model processes

the input state once and evaluates all questions simultaneously in one forward pass through the network. This is fundamentally different from autoregressive text generation, where the model generates output token by token, each token dependent on all previous tokens.

In an LLM, generating a JSON response requires the model to produce each character sequentially: opening brace, quote, field name, colon, quote, value, comma and so on. Each step depends on the previous steps. If the response has 100 tokens, the model performs at least 100 generation steps. This is why LLMs are slow for structured outputs: the sequential dependency is inherent to the architecture.

In a decision model, all questions are evaluated in parallel. The model takes the state and the question definitions as input, runs them through its network once, and produces all answers simultaneously. The latency is determined by a single forward pass through the network, not by the number of tokens in the output. Whether you ask one question or fifty questions, the latency increase is minimal because they are evaluated together.

This single-pass approach is possible because decision models do not generate text. They do not need to autoregressively produce characters or tokens. Instead, the answer to each question is computed as a score or probability from the model's internal representations. For encoder-based models like Laya, the model encodes the state into a vector representation, then compares that representation against each option using a scoring mechanism. For decoder-based models like decider, the model processes the state and questions in a single forward pass and reads the answer from specific tokens in the output logits.

The "no-generation constraint" means decision models sacrifice the flexibility of LLMs for speed and reliability. They cannot write prose, explain their reasoning, or produce creative output. They are specialized for structured decisions and nothing else. This is not a limitation in the sense of a bug. It is a deliberate trade-off. By constraining the output space and eliminating generation, decision models achieve:

- Predictable latency independent of output size
- Type-safe outputs that always match the defined schema
- Reduced hallucination risk because the model cannot output undefined values

- Lower computational cost because they avoid sequential token generation

If you need both structured decisions and prose output, you combine both types of models in your architecture. Use a decision model to classify, route, score and verify. Use an LLM to explain, draft, translate, or reason through complex scenarios. We will explore hybrid architectures in detail later in this book.

How Decision Models Avoid Hallucination by Design

Hallucination is one of the most notorious problems with LLMs. An LLM can confidently generate facts, names, numbers, or options that do not exist or were not in its input. This happens because LLMs are trained to predict plausible continuations, not to faithfully represent reality.

Decision models avoid hallucination through architectural constraints rather than through prompt engineering or post-processing. The model's output space is literally the set of options you defined. For a choice question with options A, B, and C, the model computes probabilities for exactly those three options and nothing else. There is no mechanism for the model to produce a fourth option or to generate text outside the defined schema.

This does not mean decision models are immune to error. They can still be wrong. They can select the wrong option, assign incorrect probabilities, or misclassify ambiguous inputs. But they cannot hallucinate in the same way an LLM can. They cannot invent categories, generate non-existent fields, or produce malformed JSON.

The distinction matters for reliability. With an LLM, you must validate every output. You must check that the JSON is well-formed, that all required fields are present, that values match your schema, and that options are from your approved list. With a decision model, you can trust the structure and validate only the accuracy. You can audit whether the model selected the right option, but you do not need to worry about it returning something your code cannot handle.

There is one subtle form of "hallucination" that decision models can exhibit: the model can be confidently wrong. If the model assigns 95 percent probability to an incorrect option, its answer will be structurally valid but factually

incorrect. This is not hallucination in the traditional sense, since the model did not invent an option, but it is still an error. This is why calibration matters and why you should validate model accuracy on your own data before trusting it with critical decisions.

Decision models also cannot hallucinate in the sense of generating reasoning or explanations. They return only the decision, not the rationale. If you need an explanation for why a decision was made, you must use additional tools: log the input state and decision for audit purposes, use an LLM to analyze the decision post-hoc, or rely on human review for high-stakes cases.

The structural guarantee of decision models, namely typed outputs that always match the defined schema, is their primary advantage over LLMs for structured decision tasks. It eliminates a whole class of integration failures and reduces the complexity of your error handling code. When the model's output is guaranteed to fit your type system, you can focus on the logic of your decisions rather than the mechanics of parsing and validation.

Chapter 3: Architecture Comparison: Decision Models vs. LLMs

Training Objectives and Loss Functions

Output Spaces: Tokens vs. Typed Decisions

Inference Behavior and Compute Patterns

Latency, Throughput and Memory Usage

Probability Interpretation and Calibration

Hallucination, Determinism and Reliability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Cost, Privacy and Operational Characteristics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Summary: When to Use Each

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Chapter 4: The Ollaya Project: Landscape and Positioning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

What Is Ollaya?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Ollaya, Jev and TypeSafe: Mapping the Relationships

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Open Decision Models and Compatibility

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

What “Jev-Style” and “Jev-Compatible” Actually Mean

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Project History, Community and Governance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Getting Started Prerequisites

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Chapter 5: Installing Ollaya

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Supported Operating Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Hardware Requirements and Recommendations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Installation via Install Script (Linux/macOS)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Installation via Package Managers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Docker and Containerized Installation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Verifying Your Installation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

CPU vs. GPU Execution Setup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Environment Variables and Server Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Troubleshooting Installation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Next Steps

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Chapter 6: Your First Local Decision Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Understanding the Model Registry

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Pulling Your First Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Running a Model with ollaya run

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Reading the Output: Decisions, Scores and Confidence

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Listing Installed Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Inspecting Model Details with show

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Stopping and Managing Running Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Interactive REPL Mode

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Removing Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Your First Custom Questions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Chapter 7: The Ollaya Command-Line Reference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Command Overview

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

The Pull Command: Fetching Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

The Run Command: Inference Basics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

The List Command: Inventory Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

The Show Command: Model Metadata

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

The Ps and Stop Commands: Process Control

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

The Create, Cp and Rm Commands: Lifecycle

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Environment Variables and Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Advanced Run Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Summary Table

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Chapter 8: Modelfiles and Custom Model Configurations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

What Is a Modelfile?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Modelfile Syntax and Structure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Defining Question Sets and Choice Spaces

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Building a Custom Model from Scratch

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Model Presets and Inheritance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Calibration Settings

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Parameter Settings

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Validation and Testing Your Custom Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Chapter 9: Running a Local Ollaya Daemon

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Starting the Ollaya Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Server Configuration and Ports

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Authentication and Access Control

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Resource Limits and Process Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Health Checks and Monitoring

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Running Ollaya as a Systemd Service (Linux)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Running Ollaya in Docker

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Security Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Next Steps

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Chapter 10: The Local Decision Model API

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

API Endpoints Overview

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

The Generate/Run Endpoint

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Request Format: Questions, State and Options

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Response Format: Decisions, Scores and Metadata

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Streaming and Batch Requests

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Error Handling and Status Codes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

API Examples with curl

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Chapter 11: Integrating with Python

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Setting Up a Python Project for Ollaya

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Making API Calls with requests and httpx

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Building a Typed Client Library

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Error Handling and Retries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Batch Processing and Concurrency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Production Python Integration Example

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Chapter 12: Integrating with JavaScript and TypeScript

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Node.js Integration Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

TypeScript Type Definitions for Decision Responses

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Browser-Based Calls to Local Ollaya

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Error Handling and Resilience

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Full Example: TypeScript Decision Service Client

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Express.js Integration Example

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Chapter 13: The TypeSafe API and Jev Integration Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

What Is the TypeSafe API?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Cloud Jev vs. Local Ollaya: API Alignment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Migrating from TypeSafe Cloud to Local Endpoints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

What Changes, What Stays the Same

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Configuration and Environment Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Type-Safe Decision Handling in Your Code

Chapter 14: Designing Good Decision Schemas

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Principles of Typed Question Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Label Space Construction and Granularity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Score Ranges and Thresholds

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Decision Hierarchies and Multi-Stage Decisions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Abstention and Human-in-the-Loop Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Schema Evolution and Versioning

Chapter 15: Real-World Application: Customer Support Triage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Problem Statement and Requirements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Decision Schema Design for Triage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Building the Ollaya Modelfile

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Implementing the Application Backend

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Integration with Messaging and Routing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Testing and Validation

Chapter 16: Real-World Application: Intent Classification and Routing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Intent Classification as a Decision Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Designing the Intent Schema

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Implementing the Router Service

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Tool Selection and Agent Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Handling Ambiguity and Edge Cases

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Chapter 17: Real-World Application: Moderation and Risk Signals

Moderation as Structured Decision-Making

Multi-Label Decision Schemas

Confidence Thresholds and Escalation

Combining Signals for Risk Assessment

Audit Trails and Explainability

Chapter 18: Hybrid Architectures: LLMs and Decision Models Together

Division of Labor: What Each Model Should Do

Decision Models as Gatekeepers for LLMs

LLMs Generating Candidates, Decision Models Scoring

Architectural Patterns and Diagrams

Cost and Latency Considerations

Chapter 19: MCP and Agent Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Understanding MCP for Decision Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Ollaya as an MCP Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Integrating Decision Models into Agentic Workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Tool Calling with Decision Outputs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Practical Integration Examples

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Chapter 20: Evaluation and Validation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Evaluation Metrics for Decision Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Building a Golden Dataset

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Accuracy, Precision, Recall and Calibration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

A/B Testing Decision Schemas

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Continuous Evaluation Pipelines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Chapter 21: Performance, Benchmarking and Optimization

Benchmarking Methodology

Latency and Throughput Measurements

CPU vs. GPU Performance

Batching and Concurrency Strategies

Memory Management and Model Loading

Caching and Optimization Techniques

Chapter 22: Observability, Logging and Debugging

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Logging Decisions and Metadata

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Structured Logs and Observability Tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Debugging Decision Outputs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Tracing Decision Paths

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Performance Monitoring Dashboards

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Chapter 23: Security and Privacy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Data Privacy Advantages of Local Execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Network Security and API Hardening

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Model Integrity and Supply Chain

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Input Validation and Abuse Prevention

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Compliance and Regulatory Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Chapter 24: Production Deployment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Deployment Architecture Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Kubernetes and Container Orchestration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Autoscaling and Load Balancing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Multi-Model Serving and Routing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Backup, Recovery and Disaster Planning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Chapter 25: Conclusion: The Future of Local Decision Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

What We Have Learned

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

The Case for Decision Models in Modern AI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Trends and Open Questions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

Next Steps for Practitioners

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/runjev-style-system-one-models-locally.