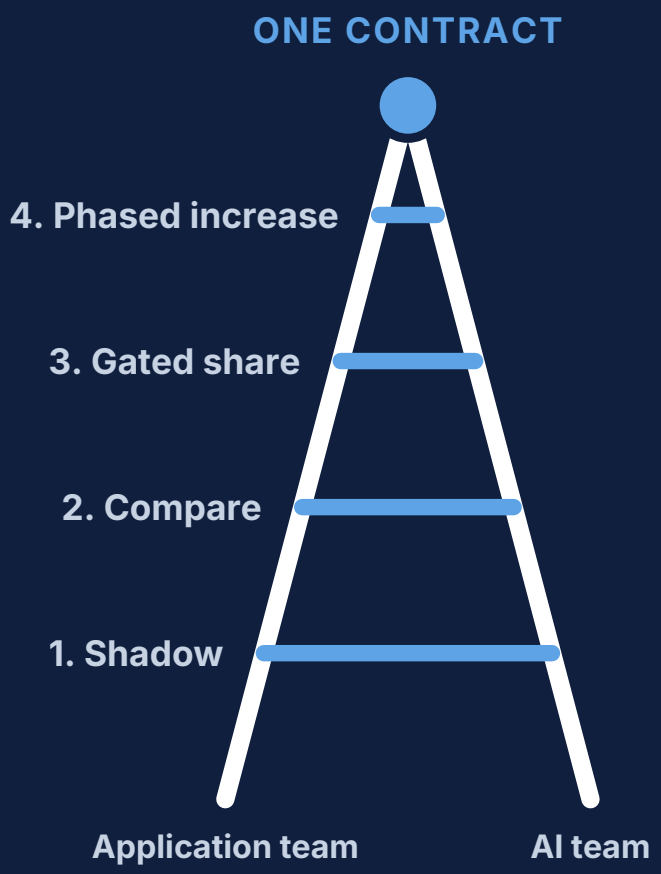
A FRAMEWORK FOR ADOPTING AI

The Stepladder Framework

Two teams, one contract, and a migration ladder from managed AI services to your own models.

Zareef Ahmed · Software architect and private AI consultant

zareef.com/stepladder-framework · September 2026



THE PROBLEM

Five tensions, and the naive approach resolves none of them

01

Speed now versus independence later

Managed services ship in weeks; owning models takes months.

02

Two roadmaps that must not block each other

If every AI change is an application change, one roadmap always waits.

03

No single provider is best at everything

The best choice moves as the market moves.

04

Quality must be measurable and reversible

A wrong extraction corrupts a record.

05

Sensitive data needs one control point

Auth, rate limits, retention, audit and residency, enforced once.

These are architectural problems, not modeling problems.

THE IDEA

One decision: treat AI as infrastructure with a contract

01

Two teams, one contract

The application team and the AI team each keep their own roadmap, joined only by a single versioned API that exposes AI as business operations: "extract the fields from this document", never "call provider X".

02

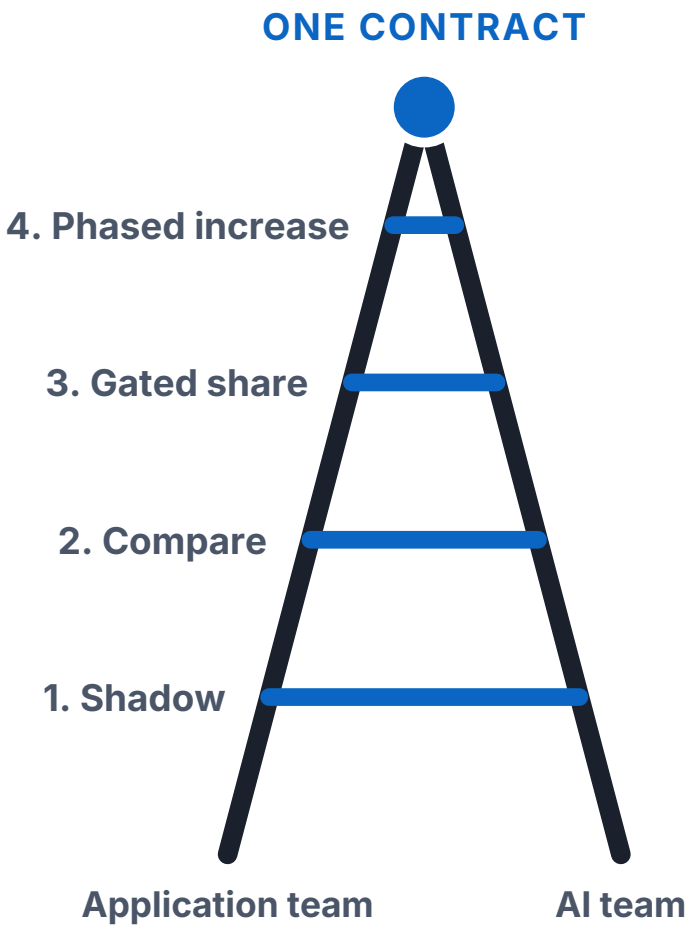
A migration ladder

Managed cloud services get you to production in weeks. Then traffic moves to models you run yourself one rung at a time: shadow, compare, gated share, phased increase.

Every rung is one configuration change from the one below it.

WHY "STEPLADDER"

The name is the picture



- **Two legs, one hinge.** The application team and the AI team meet at exactly one point: the contract.
- **Free-standing.** A stepladder leans on no wall. No provider is structurally embedded.
- **Climbed rung by rung.** Traffic moves to your own models in held, measured steps.
- **You can always step down.** Rollback is a routing change, never a deployment.

A PERSONAL NOTE

A known pattern, written down the way I use it

I do not claim the idea behind Stepladder. An abstraction layer between the applications and the AI services behind them is a known pattern, and it goes by several names: AI gateway, model router, LLM proxy, inference abstraction layer. What you are reading is my documented version of what I do with most of my clients and projects, written down once so I can point to it.

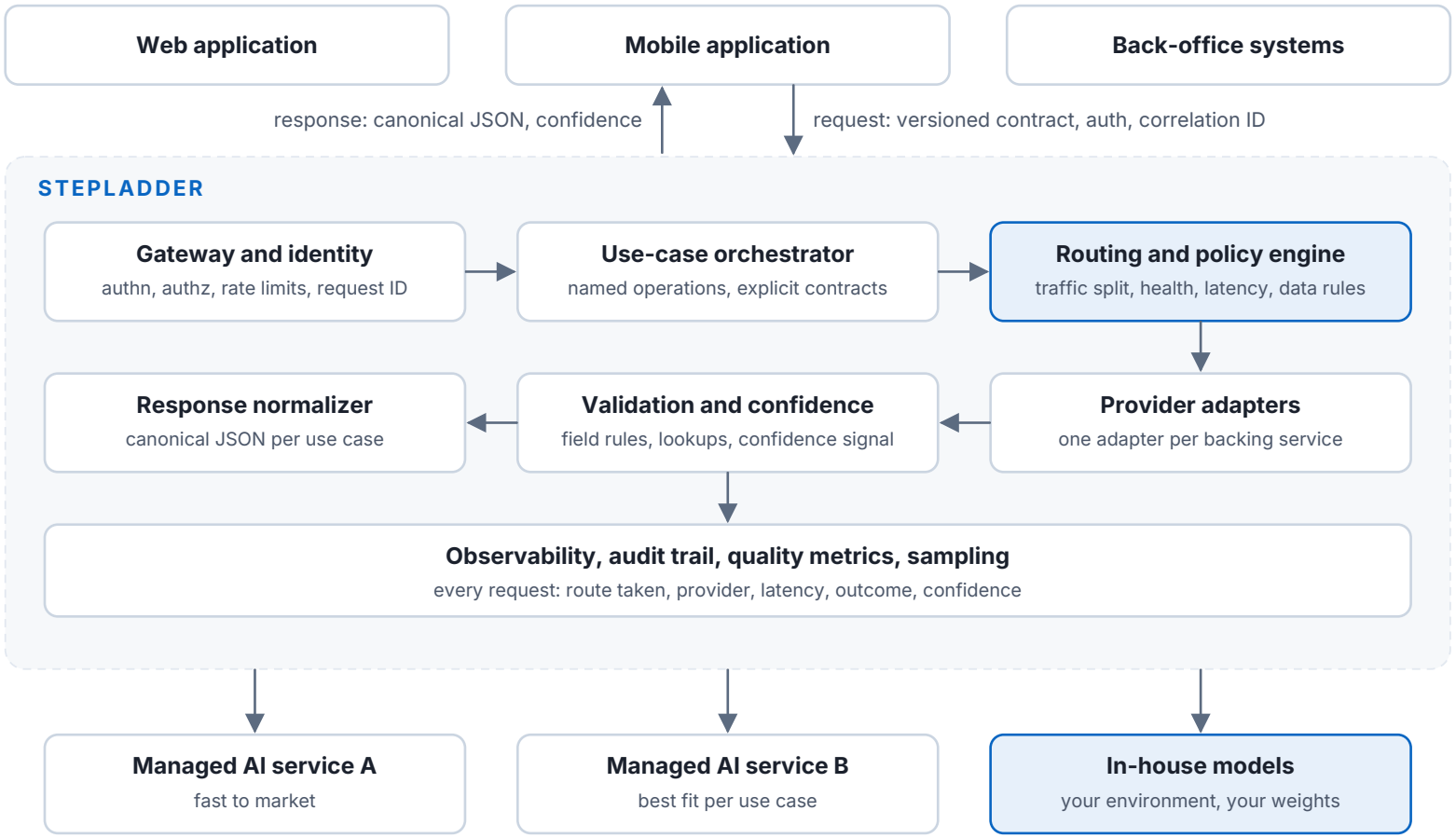
I originally called it AIBOX, and it still runs under that name at several client locations. I may keep using AIBOX with clients and customers. Stepladder is the name I chose for the documented version, because it says what the pattern does: two legs joined by one hinge, climbed one rung at a time.

For me, AIBOX was never simply a router. It is my auditor, my cache layer, my load balancer and my reinforcement learning data collector. Because every request and every answer passes through one place, each one can be recorded, served again without a second call to the provider, spread across providers and model instances by load and health, and turned into the graded examples that train the next model.

ZAREEF AHMED

ARCHITECTURE

Reference architecture



ARCHITECTURE

Six building blocks, each with one job

01

Gateway and identity

Auth, entitlements, rate limits, correlation ID. Consumers hold credentials for the layer only.

02

Use-case orchestrator

Named operations with explicit input and output contracts; one call or a pipeline, the consumer cannot tell.

03

Routing and policy engine

Configuration, not code: traffic split, provider health, latency budget, data-handling rules.

04

Provider adapters

One per backing service. A managed service, an open-weight model in your own environment, a fine-tuned specialist: all just adapters.

05

Validation, confidence and normalization

Raw output is never returned. Canonical schema, field rules, a confidence signal.

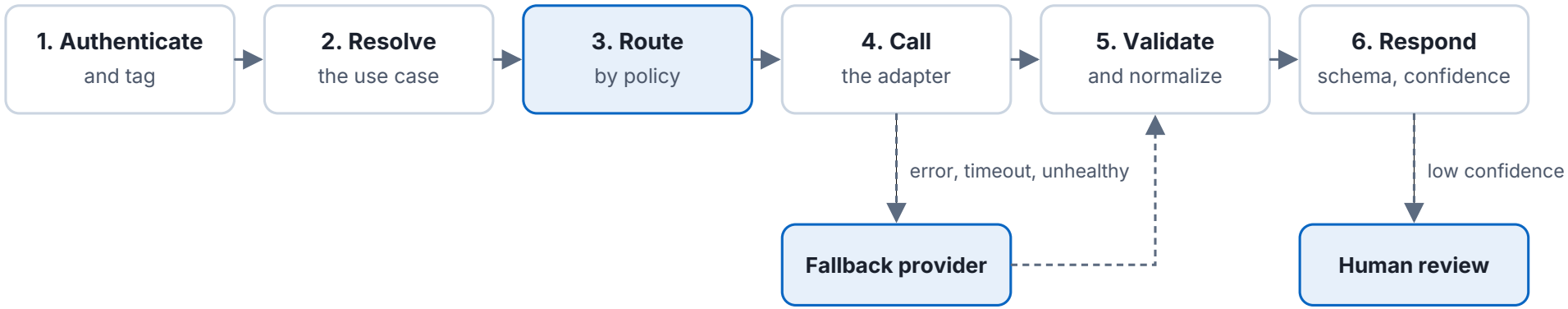
06

Observability and audit

Every request recorded with route, provider, latency, outcome and confidence. This is what makes change safe.

DATA FLOW

How a request flows



Every hop writes an audit record

request ID, use case, route taken, provider, latency, outcome, confidence, plus sampled payloads for evaluation

Fallback on error. A failed call is retried through another provider; the consumer never sees which one answered.

Human review on low confidence. Uncertain results go to a person, not to the record. Every hop writes an audit record.

The contract is the whole point

- **An input schema** per use case, versioned: a breaking change is a new version, never a surprise.
- **An output schema** in canonical JSON, identical whichever provider answered.
- **A confidence signal** with a documented meaning, so consumers can set thresholds.
- **Actionable failures:** "retake required", "needs human verification", not three vendors' error codes.
- **A correlation ID** to quote back when something needs investigating.

What the consumer is never told

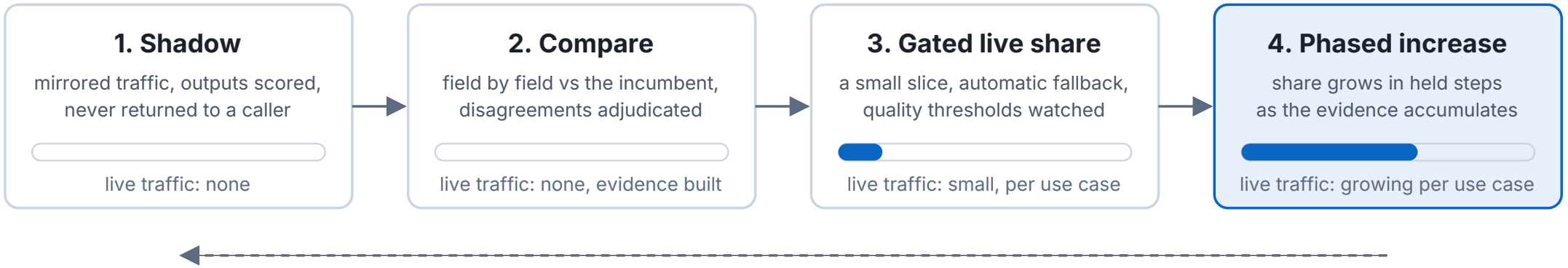
Which provider answered.
Which model.
Which prompt.
Which version of the pipeline.

If that leaks into consumer code, the abstraction is gone and every future change is a coordinated release again.

Normalize aggressively at the boundary.

MIGRATION

The migration ladder: from managed services to your own models



Rollback at any rung: revert the routing policy. One configuration change, no deployment.

> **Climb it per use case, not globally.** Every rung is one configuration change away from the previous one, and rollback is a change to the routing policy, not a deployment.

MIGRATION

Why the order matters

BACKWARDS

Most organizations build the model first

Build the in-house model, then try to integrate it, and discover the evaluation and routing problems at the end.

STEPLADDER

Build the layer first

By the time the first in-house model exists, the machinery to evaluate it, route to it and roll back from it is already running in production, with months of real traffic to test against.

Hybrid is a destination, not a waypoint. Keeping managed services routable is a resilience feature even when the goal is full in-house inference.

ORGANIZATION

Two teams, one contract

CONCERN	DIRECT INTEGRATION	WITH STEPLADDER
Changing AI provider	Code change, regression cycle, coordinated release	Routing configuration change with an audit record
Upgrading a model	Blocked on the application release train	Shipped on the AI team's own cadence
Adding a use case	New integration, credentials and error handling per consumer	New use case on an existing platform, one contract
Rolling back a bad change	Hotfix and emergency deployment	Revert the routing policy, effective immediately
Provider credentials	Distributed across every consuming application	Held only inside the layer

The contract is the hinge: the one point where the two legs meet, and what lets each leg carry its own weight.

GOVERNANCE

Governance by design

01

One control point

Auth, authorization, rate limits, retention and audit, enforced once.

02

Credential containment

Provider credentials exist only inside the layer.

03

Policy-aware routing

Payload classes that must stay inside a boundary never leave it.

04

Full audit trail

What was asked, which engine answered, with what confidence.

05

Human in the loop

Uncertain results go to people, not to the record.

Governance is a property of the layer, not a review step bolted on afterwards.

PRINCIPLES

Six design principles

01

Build the abstraction before you need it

Its value is everything after the first use case.

02

Routing belongs in configuration, not code

That is what makes reversible migration possible.

03

Normalize aggressively at the boundary

One canonical schema per use case, enforced on the way out.

04

Shadow mode is the cheapest evaluation you will ever run

Real traffic, zero production risk.

05

Design for the low-confidence case first

Knowing when you are unsure beats being slightly more accurate.

06

Hybrid is a destination

Keep every proven provider routable.

FIT

When to use it, and when not to

USE STEPLADDER WHEN

- more than one application will consume AI
- you expect to change providers or models
- you intend to run your own models eventually
- the outputs write to records that matter
- the data needs one enforcement point

Any two of these justify the layer.

DO NOT BUILD IT WHEN

A single application makes a single AI call with no plan to change. A direct integration is right there; introduce the layer when a second consumer or provider appears.

The pattern earns its keep through reuse and change. Without either, it is ceremony.

ADOPTION

How to adopt Stepladder: eight steps

01

Name the use cases as business operations

Never as "call provider X".

02

Define the contract for each

Input and output schema, confidence, failure outcomes. Version it from day one.

03

Stand up the gateway and the audit trail

Before the first adapter.

04

Write the first adapter against a managed service

Speed to first value is the point.

05

Put validation and normalization at the boundary

Return nothing raw.

06

Move routing into configuration

With an audit record and a tested rollback.

07

Add the second provider or first in-house model

As an adapter, then climb the ladder per use case.

08

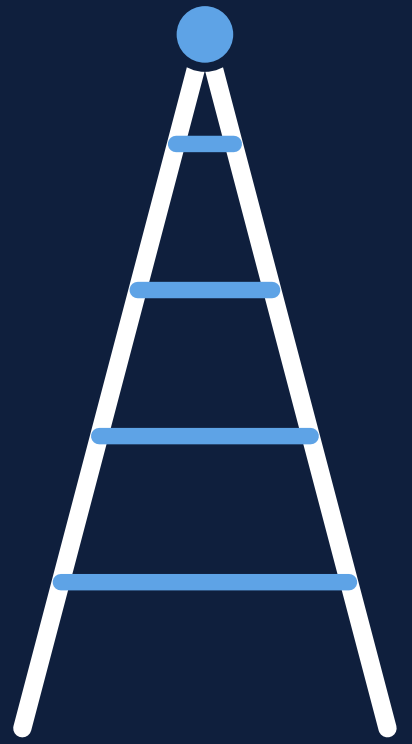
Make promotion routine

Automated comparison and quality gates, then the next use case.

WORKING WITH ME

Two teams. One contract. A ladder to your own models.

I design and build Stepladder-style layers as part of private AI deployment consulting: the contract, the routing and policy engine, the evaluation machinery, and the migration from managed services to models you run yourself.



Zareef Ahmed · zareef@zareef.com · zareef.com/stepladder-framework

Full framework: zareef.com/zareef-ahmed-stepladder-framework.pdf