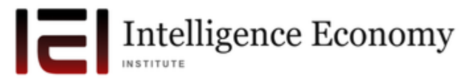



# Ontology Control Layers over LLMs

*Concrete architectures, agentic-payment illustration*

Zakaryae Boudi
Intelligence Economy Institute

June 26, 2026

## Executive summary

When a large language model (an AI system trained to generate text by predicting one word at a time, based on patterns learned from a massive training corpus) is asked to take actions on behalf of a user (book a flight, approve a payment, schedule a procedure), the system designer faces a question with surprisingly few satisfying answers: how do you constrain what the model is allowed to say, in a way that matches what the rest of the system is prepared to act on?

This document maps the space of answers. The middle layer between the language model and the executing system (the layer that converts free-form text into structured, validated, actionable commands) has many possible designs, each with different costs, different guarantees, and different audit posture. We call this middle layer the ontology control layer, where "ontology" is used in the technical sense: a structured description of the entities a system deals with (counterparties, currencies, purposes, time windows), the relationships between them, and the rules they must collectively satisfy. This document is about the design of that layer.

The contribution is fourfold. First, a four-axis design space that any control architecture sits in some configuration of. Second, four named reference architectures (Prompt-Bound, Schema-Bound, Grammar-Constrained, and Verified Mediator) that are the most coherent points in that space. Third, an honest accounting of which properties of each architecture can be formally proved, which cannot, and where the line between "engineering territory" and "research territory" actually falls. Fourth, a survey of the formal methods themselves, with particular attention to the B method (the verification methodology used to prove correctness of the Paris Metro Line 14 signaling system and a major industrial standard for safety-critical software) and its operationalization for an LLM mediator.

Formal verification runs through the document as a transverse theme rather than a single chapter, because the question "is this verifiable?" has a different answer at every layer of the stack. We name five concentric rings of verifiability, and an honest line separating the engineering side (schema soundness, validator totality, system safety invariants, all tractable today) from the research side (whether a language model truly "understands" the user, whether the system end-to-end does the right thing, open problems with no general method).

The unifying claim is simple. You do not verify the language model. You verify the system around it. The model is treated as an untrusted oracle that emits structured terms; the rest of the stack (schemas, decoders, mediators, policy gates) is what you reason about formally. Architecture D (the Verified Mediator) makes this explicit; the others get there partially. None of the four require giving up on language models for any task; they differ only in how much of the resulting system is amenable to proof, and at what cost.

#### Reading order

Section 1 maps the four-axis design space. Read this first; everything else references it. Section 2 sets up the four reference architectures side by side, with one-paragraph plain-language descriptions of each. Sections 3 through 6 are deep dives into each architecture, with worked examples and explicit accounting of what is and is not verifiable at each level. Section 7 surveys the formal-methods landscape (Coq, Lean, Dafny, F*, TLA+, Isabelle, refinement types, and the B method) with the B method treated in detail because of its uniquely strong industrial track record. Section 8 is the verification taxonomy: five rings of verifiability, with the honest line between engineering and research. Section 9 covers

hybrid composition: what real production stacks look like. Section 10 walks the same agentic-payment task through all four architectures. Sections 11 through 12 are decision frames: which architecture for which situation, and how to evolve from one to the next.

## 1 The design space

Architectures for controlling the output of a language model through an ontology are not arranged along a single quality axis. Calling one "better" than another hides the trade-off it implies. A more useful framing names four independent dimensions (we call them axes) along which any architecture sits at some position, and treats the architectures themselves as points (or small regions) in that four-dimensional space.

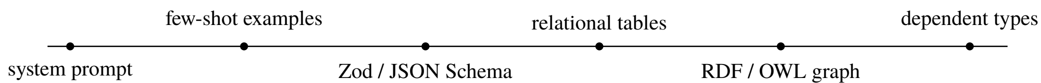
Below, each axis is explained, with examples drawn from the four architectures the rest of the document will develop. If you have read about constrained decoding, structured outputs, or proof-carrying code before, this section is a vocabulary refresher. If those terms are new, this section is the foundation everything else builds on.

**THE DESIGN SPACE: FOUR AXES**

*every ontology control architecture is a point in this space*

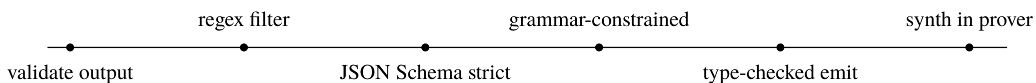
**AXIS 1 Where the ontology lives**

*Implicit (in prompt)* *Explicit (in code or DB)*



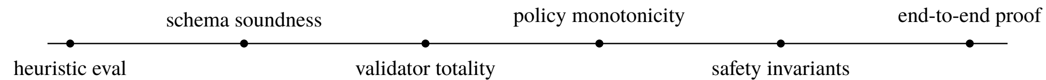

**AXIS 2 When constraints are checked**

*Post-hoc (generate then filter)* *By construction (constrained decoding)*



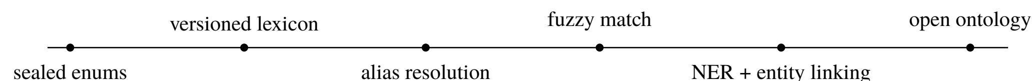

**AXIS 3 What is formally verifiable**

*Nothing* *Full functional correctness*




**AXIS 4 Openness of the world**

*Closed-world (fixed enums)* *Open-world (free text + normalization)*




*Most production systems sit in the middle of axes 1, 2, and 4. Axis 3 is where serious engineering investment shifts the answer.*

### 1.1 Axis 1: where the ontology lives

First, recall what we mean by ontology. In this document, an ontology is the structured description of what the system knows about its domain: the things that exist (counterparties, currencies, accounts, payment rails), the categories they belong to (a counterparty is one of {individual, business, agent, platform}), the relationships between them (a counterparty has zero or more payment accounts; each account is denominated in one currency and operates on one rail), and the constraints that bind them (a payment to an individual cannot have purpose "API access"). The ontology is the system's "vocabulary plus grammar": what words mean, and how they may be combined.

Axis 1 asks: where, physically, does this ontology live? At one end of the axis, it lives only inside the prompt (the natural-language instructions sent to the language model) as enumerated lists ("counterparty must be one of: Acme, OpenAI, Northwind") and demonstrative examples. Nothing in the rest of the codebase encodes the ontology directly; if a developer needs to add a new counterparty, they edit the prompt.

At the other end of the axis, the ontology is a first-class artifact in code or in a database. It might be a schema, a precise description of allowed data shapes, expressed in a tool like Zod (a TypeScript library), JSON Schema (a language-independent specification format), or Pydantic (a Python library); it might be a set of related database tables with foreign-key constraints; or, in the most demanding case, it might be a definition in a dependent type system such as Lean or Coq, where the compiler refuses to build the code if any value violates the declared structure.

The position on this axis determines what part of the system "knows" the ontology. When it lives in the prompt, only the language model knows it, and only for the duration of one call. When it lives in code, the rest of the stack can introspect it: a validator can read the schema, a decoder can derive a set of allowed tokens from it, a database can constrain inserts against it. The further to the right on this axis, the more components share a single source of truth about what the ontology says.

### 1.2 Axis 2: when constraints are checked

Constraints can be checked in two fundamentally different ways. Post-hoc (Latin for "after this") means: let the model generate whatever it wants, then run a validator over the result and reject anything that fails. By construction means: arrange things so that producing an invalid output is literally impossible. Most production systems mix the two.

Post-hoc checking is universal (you can validate anything after it exists) but it is also lossy. An invalid output costs computation to generate and a round-trip to retry. If the validator is incomplete (a regular expression that forgot to handle escaped quotes, say), invalid outputs may slip through. The cost is paid every time, in latency and in token spend.

By-construction checking is harder to set up but eliminates the failure mode. Grammar-constrained decoding (Architecture C, treated in detail in §5) reaches into the model's output generation process and forbids it from producing a token that would violate the grammar. The model literally cannot emit invalid output. Strict JSON-output mode in commercial LLM providers (OpenAI, Anthropic, Google) is a milder form of the same idea: the provider's system guarantees the response is parseable JSON conforming to a schema, by intervening during generation.

Most production systems sit somewhere in the middle of this axis. Strict JSON mode handles structural correctness by construction (the model cannot emit malformed JSON or unknown fields). Cross-field constraints (such as "if the purpose is supplier_invoice, the counterparty must be a business, not an individual") are typically checked post-hoc, because they depend on combinations the grammar cannot

express. The Verified Mediator (Architecture D) pushes the by-construction boundary further still, into the layer below the model, by making certain incoherent combinations literally inexpressible in the type system that defines what a proposal is allowed to be.

### 1.3 Axis 3: what is formally verifiable

This is the axis with the most asymmetric returns. We use formally verifiable in the strict technical sense: a property is formally verifiable if there exists a mechanical procedure that takes the program (or the specification) as input and produces a mathematical proof that the property holds. Not "we tested it and it seems to work"; not "we reviewed the code carefully"; but a machine has checked a chain of logical steps from axioms to conclusion.

At the left end of this axis, nothing about the system is formally proved. The system relies on testing, code review, monitoring, and operational feedback, all valuable, none of them proofs in the strict sense. At the right end, end-to-end functional correctness ("the system does exactly what the specification says it should do") is proved against a complete specification. In between, you get progressively stronger but more narrowly scoped guarantees: that the schema is well-formed, that the validator is total (always returns an answer, never crashes), that the policy decision function is monotone (granting more authority cannot revoke an approval), that no execution can ever violate a stated safety invariant.

The asymmetry is that engineering investment shifts the answer dramatically along this axis, but only in certain regions. Adding a schema validator gets you "schema soundness" for the cost of a few hours of work. Adding a Lean specification of the control layer gets you "no execution violates the safety invariants" for the cost of weeks or months of work. Adding a proof that the language model's output actually captures what the user meant costs… we do not know. There is no general method. The honest taxonomy in §8 separates the engineering territory from the research territory along this axis.

### 1.4 Axis 4: openness of the world

A closed-world system works with a fixed, finite, known-in-advance set of entities. The currencies it supports are an enumerated list (USD, EUR, GBP, JPY, …). The counterparties it can pay are a registered set, each with a primary key. The purposes it accepts are a sealed enum. An open-world system, by contrast, must accommodate names it has not seen and entities not yet registered: a user types "pay the cleaning company" and the system must figure out what entity that refers to, possibly registering one if no record matches.

Most real ontologies are mixed. They are closed for some categories (the currencies you support is finite and changes rarely) and open for others (the set of people or organizations you might transact with grows continuously). The right design lets each field sit at the right position on this axis.

Openness interacts with the other axes. Closed-world systems can pin the entire ontology into prompts and decoders, because the set is finite and small enough to enumerate. Open-world systems need a normalization step (the lower-cased, accent-stripped version of a name) and an entity-linking step (figuring out which database record a name refers to, possibly with disambiguation) before the closed-world part of the pipeline takes over.

**The purpose of the axes**

These four axes are diagnostic, not prescriptive. They are the questions you ask when characterizing a proposed architecture, not the answers you should reach for. The architectures presented in §2 through §6 are positions on these axes that we think are coherent, meaning

their answers to the four axes are mutually consistent, and the resulting system is internally well-formed. An architecture that pulls in different directions on different axes (an open-world vocabulary fed to a closed-world grammar decoder, with no validation step) is generally a bug, not a design. The axes help you spot such inconsistencies before you build them.

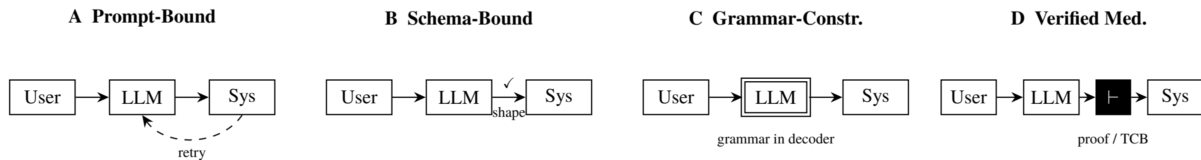
## 2 Four reference architectures

The four architectures named in this section are not exhaustive. There are many possible points in the four-dimensional design space. The four we name are the points where the trade-offs become visibly different from each other, and where most production systems aim to land, often as a hybrid of two or three.

Each architecture answers the four axes in a different way, has its own data flow, requires different tooling, and admits different formal-verification claims. The schematic below previews the four data-flow patterns; the comparison table summarizes their commitments; the sections that follow dive into each one in detail.

**Four reference architectures: data-flow patterns**

*each one is a coherent point in the design space, with its own trade-offs*




*All four treat the model as the component that proposes, never the one that executes. The side effect is always performed by deterministic code reading a validated proposal.*

| | A. Prompt-Bound | B. Schema-Bound | C. Grammar-Constr. | D. Verified Mediator |
|---|---|---|---|---|
| Ontology lives | In the LLM's prompt (English text) | Typed schema (Zod, JSON Schema, Pydantic) | Formal grammar (GBNF, context-free) | Typed graph / DSL with a proof source |
| Constraints checked | Post-hoc regex or JSON parse | Strict JSON mode at the boundary | In the decoder, by construction | Mediator proof, discharged per call |
| Verification reach | × nothing about the output | ✓ shape, types, enum closure | ✓ in-grammar; decoder sound | ✓ system invariants; side effects |
| Verdict | Lowest effort, lowest assurance | Industry default for production agents | Strong syntactic guarantees | Highest assurance, highest investment |

### 2.1 Architecture A: Prompt-Bound

The ontology is communicated to the language model entirely through the prompt (the natural-language instructions and examples included with each request). Output is validated after the fact. This is the

path of least resistance: a system prompt with a few rules ("you must return JSON with these fields…"), an example or two, and a regular expression or JSON parser checking the output.

Most prototypes start here. Many ship here. It is appropriate for low-stakes outputs that humans will review before they have any effect: drafts, summaries, classifications. It is fundamentally insufficient for high-stakes actions, because the language model is the only component that knows the rules, the rules are reasserted only as English text, and the validator catches structural mistakes but not semantic ones.

### 2.2 Architecture B: Schema-Bound

The ontology is expressed as a typed schema, a precise, machine-readable description of allowed shapes, written in a tool like Zod, JSON Schema, or Pydantic. Modern commercial LLM providers can enforce such a schema at their decoding boundary: the response is guaranteed to be JSON, to have the declared fields, to use only values from the declared enums. The receiving code parses the response through the same schema again, this time turning it into typed objects that downstream code can dispatch on.

This is the industry default for production agents that need to be reliable but do not yet need to be formally proved. The Schema-Bound architecture is enough for the great majority of agentic-payment, agentic-procurement, and agentic-scheduling systems deployed today.

### 2.3 Architecture C: Grammar-Constrained Decoding

The ontology is expressed as a formal grammar, a precise mathematical description of which sequences of characters or tokens are allowed. The decoder (the component inside the model's runtime that picks one token at a time to extend the output) enforces the grammar at every step: at each token, the set of "allowed next tokens" is computed from the grammar's current state, and the model's probability distribution is restricted to that set before sampling.

The result is that the output is in-grammar by construction. The model literally cannot produce an out-of-grammar sequence. This is the strongest structural guarantee short of full formal verification, and it is achievable today on open-weight models (Llama, Mistral, Qwen) and on a growing number of commercial providers that expose decoder hooks.

### 2.4 Architecture D: Verified Mediator

The ontology is expressed as a typed graph; the component between the language model and the executing system (we call it the mediator) is formally proven to refine a specification. The model remains an untrusted oracle producing structured terms; the mediator is the Trusted Computing Base (TCB), the part of the system whose correctness everything else depends on. The proof obligation reads, informally: for every proposal the model emits and every authorization context, if the mediator approves an action, the action satisfies all system safety invariants.

This is the architecture for irreversible, audit-bound systems (payments, medical orders, infrastructure changes) where a regulator, auditor, or court will eventually ask "prove that this cannot happen." It is the most demanding of the four to build and the only one that gives you proven system-level guarantees about behavior in the presence of adversarial inputs.

### 2.5 What the four architectures share

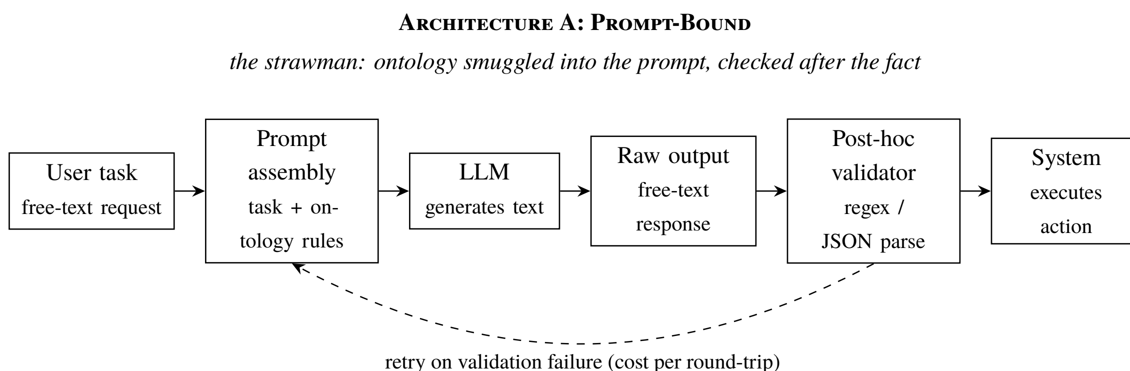
All four treat the language model as the component that proposes; none of them treat it as the component that executes. The model's output is structured (the position on Axis 1 varies) and is checked (the position on Axis 2 varies); but in every case, the side effect (the payment, the medical order, the

infrastructure change) is performed by deterministic code reading a validated proposal. This is the load-bearing invariant of the entire space. Architectures that violate it (a model that calls APIs directly without an intervening mediator) are outside the scope of this document and outside the scope of responsible deployment for high-stakes domains.

The other shared property is that the ontology is the same artifact the model consumes and the system enforces. In Prompt-Bound this fails: the model hears one version (the prompt) and the validator reads another (the regular expression). The further to the right on Axis 1 you move, the tighter the coupling: in the Verified Mediator, the schema, the projected vocabulary sent to the model, and the proof obligation are all derived from one Lean (or Coq, or Dafny, or B) source. There is no drift between what the system enforces and what the model was told to do.

## 3 Architecture A: Prompt-Bound

The simplest architecture, and the one most production systems quietly still rely on for at least some of their fields. Worth taking seriously as the baseline against which the others justify their additional complexity.




**When this works**

- Single-developer prototype, exploratory phase
- Low-stakes outputs (text summaries, classifications)
- Loose ontology that resists formalization
- Need to ship in hours, not weeks

**When this breaks**

- Any high-stakes action (payments, medical, legal)
- Adversarial environments (untrusted user input)
- Closed-world constraints (must match a known entity)
- Auditability requirements (regulator-facing trail)

### 3.1 What "prompt-bound" actually means

A prompt is the text sent to a language model to specify what the model should do. It typically combines a system prompt (instructions about the role and constraints, set by the application developer), the user's request, and sometimes example interactions ("here is a good answer; here is a bad one"). The prompt-bound architecture places the entire ontology (the rules about what entities exist, what fields are allowed, what values they may take) inside this text.

The rest of the system contains a validator, a piece of code that inspects the model's output and decides whether it is acceptable. In its simplest form, the validator is a regular expression (or "regex"), a small notation for describing patterns of characters: it can express things like "a string of digits, followed by a period, followed by more digits" but cannot, in general, validate semantic correctness. A slightly more sophisticated validator parses the output as JSON and checks that the structure matches expectations.

### 3.2 How it works in practice

The data flow has three failure modes, each invisible except in retrospective inspection of logs.

- **The model ignores or misinterprets the prompt rules.** Maybe the rules were ambiguous, maybe the model "rounded" them to a more common pattern from its training data, maybe a user instruction overrode them through prompt injection. The result is output that violates the rules.
- **The model produces output that looks valid but is semantically wrong.** The structure matches the regex; the JSON parses; the field values are plausible, but the values refer to entities that do not exist, or combinations that are incoherent for the actual mandate.
- **The validator accepts an output that should have been rejected.** The regex did not handle some edge case, or the JSON parser was too permissive. False positives are invisible until they cause a downstream failure.

### 3.3 Worked agentic-payment example

The prompt says: *You must propose a payment. The counterparty must be one of: Acme Cloud, OpenAI, Northwind. The amount must be in EUR. Return JSON with fields counterparty, amount, currency.*

The model produces output something like: "I'll pay OpenAI 1500 EUR via SEPA, here is the JSON: { ...}". The validator runs a regex to extract the JSON block from the prose. The JSON parses successfully. The downstream code reads `counterparty=OpenAI`, `amount=1500`, `currency=EUR`, and dispatches to the rail adapter (the component that talks to the payment network), which fails because no EUR account for OpenAI is registered in this workspace. The failure surfaces to the operator as a rail-level error message: "no rail-account combination for counterparty." The cause, that the language model proposed an action the workspace cannot actually execute, is not traceable from where the failure was reported.

### 3.4 What is verifiable, in the strict sense

Almost nothing about the output. You can verify (a) that the regex is well-formed (a mechanical check on the regex source), (b) that the JSON parser is total: it returns either a parsed value or a typed error and never crashes (a property of standard library implementations), and (c) that the audit log captures every reject (a property of the logging code). None of these statements tells you anything about whether the output was correct, or even structurally valid in any stronger sense than "parses successfully."

The validator's false-negative rate (the fraction of bad outputs it lets through) is bounded only by the regex's coverage, which is bounded by what the engineer thought to handle when writing it. Bad outputs that the engineer did not anticipate will not be caught.

### 3.5 When this architecture is the right choice

Prompt-bound is genuinely the right choice in three situations:

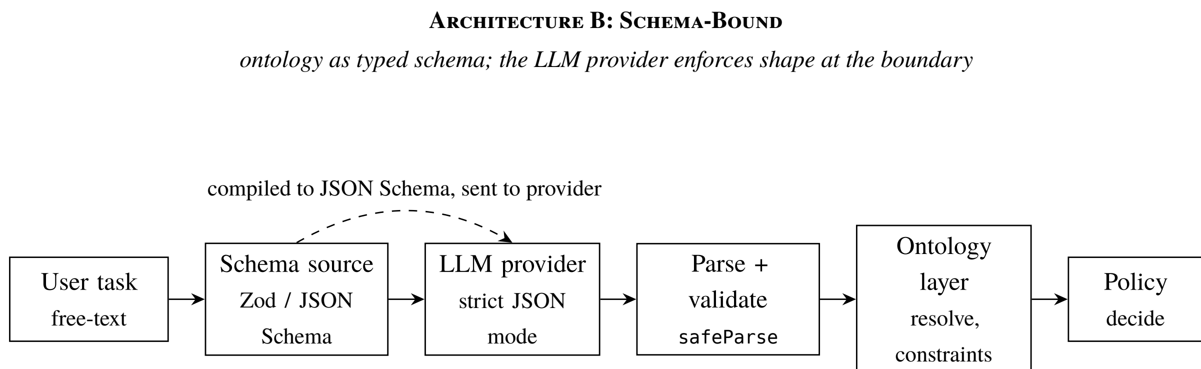
- The system is an exploratory prototype where shipping the working path matters more than the failure modes. The team is learning what the system needs to do; getting to "it can do something" is the priority.
- The outputs are low-stakes and consumed by humans who will reject obvious errors. Drafts of marketing copy, candidate translations, classifications a person reviews: bad outputs are caught downstream by human judgment.

- The ontology genuinely resists formalization. A system that advises on legal questions or interprets ambiguous medical histories may have rules that depend on context the schema cannot capture, and trying to formalize them too early would over-constrain.

For any high-stakes action (payments, medical orders, legal commitments, infrastructure changes), prompt-bound is insufficient on its own. It can survive as a residue inside a larger architecture (the system prompt still exists in B, C, and D), but it cannot bear the load alone.

## 4 Architecture B: Schema-Bound

The industry default for production language-model agents. The ontology is expressed as a typed schema; the LLM provider enforces the shape of responses at its decoding boundary; the application parses and validates the structural type before any downstream consumer reads the proposal.




**Architecture B: Schema-Bound**

*ontology as typed schema; the LLM provider enforces shape at the boundary*

*The same schema artifact is compiled to the provider, parsed on return, and consumed as a type downstream. One source of truth, no drift.*

### 4.1 What a schema is, and why this matters

A schema is a precise, machine-readable description of what shape data must take. It says: a payment proposal has these specific fields (counterparty, amount, currency); each field has this specific type (a string drawn from a known list, a positive integer, a three-letter currency code); and certain combinations are required or forbidden (if type is "Refund", then `originalIntentId` must be present).

Several tools express schemas in different ecosystems. Zod is a TypeScript library: you write the schema in TypeScript code, and Zod gives you both runtime validation and compile-time types from the same source. JSON Schema is a language-independent specification format, expressed as JSON itself. Pydantic is the Python equivalent of Zod. All three solve the same problem: they let you write the schema once and use it for validation, for type-checking, and for documentation.

The schema in this architecture serves three roles simultaneously. It is the source from which the schema document sent to the LLM provider is generated (so the provider knows what shape to enforce). It is the parser that validates the response when it comes back. And it is the type the downstream code consumes (so the rest of the application gets compile-time guarantees about what fields exist). When all three roles are filled by a single artifact, you eliminate an entire class of drift bugs where the validator and the prompt instructions slowly fall out of sync.

### 4.2 Discriminated unions and typed refusal

The idiom that makes schema-bound architectures expressive is the discriminated union, sometimes called a “tagged union” or “sum type” in the literature. A discriminated union is a type that can be any one of several alternatives, distinguished by a special “tag” field. In a payment system, the tag might be called `@type`, borrowed from JSON-LD (a widely-used convention), and the alternatives might be Payment, Refund, Subscription, and NotProposable.

Each alternative carries exactly the fields it needs. A Payment has counterparty, amount, currency, purpose, rail. A Refund has `originalId`, `amountMinor`, reason, but no counterparty (that is recovered from the original). A NotProposable has reason (an enum saying why) and an optional free-text detail. Downstream code, when it sees the `@type` value, knows exactly which fields it can read and which it cannot.

The NotProposable branch is the key innovation over Architecture A. When the language model is unable to satisfy the task (perhaps the user asked for something the mandate forbids, or the necessary evidence is not available) Architecture A leaves the model with two bad options: hallucinate a plausible-looking but wrong action, or produce free-text apologies that the validator cannot parse. In a discriminated union, refusal is a typed alternative: the model can produce `{"@type": "NotProposable", "reason": "missing_counterparty"}` and the system handles that just as cleanly as it handles a successful payment proposal.

```
// Zod, but the same idea fits Pydantic, io-ts, valibot, etc.
const PaymentIntent = z.discriminatedUnion('@type', [
  z.object({
    '@type':      z.literal('Payment'),
    counterparty: z.enum(allowedCounterparties),
    amountMinor:  z.number().int().positive(),
    currency:     z.enum(allowedCurrencies),
    purpose:      z.enum(allowedPurposes),
    rail:         z.enum(allowedRails),
  }),
  z.object({
    '@type':      z.literal('Refund'),
    originalId:   z.string().uuid(),
    amountMinor:  z.number().int().positive(),
    reason:       z.string().min(1),
  }),
  z.object({
    '@type':      z.literal('NotProposable'),
    reason:       z.enum([
      'missing_counterparty',
      'missing_evidence',
      'out_of_scope',
    ]),
  }),
]);
```

### 4.3 What the provider enforces, what the application enforces

Modern commercial LLM providers (OpenAI, Anthropic, Google) and most open-weight runtimes (llama.cpp, vLLM) expose a feature called strict JSON mode (the name varies). When you call the

model with a schema attached, the provider's runtime intervenes during generation: it constrains the output so that the result is guaranteed to be valid JSON, to have the fields declared in the schema, to use only values from the declared enumerations, and to satisfy types ("must be an integer," "must be a string"). The model cannot emit malformed JSON, cannot make up new fields, cannot return a string where you asked for a number.

This is genuine by-construction enforcement at the structural level. It is not "the model tries to be careful and we check afterward"; it is "the generation process itself is constrained, token by token, to stay within the schema." The exact strength of the enforcement varies by provider (grammar-level enforcement, Architecture C in §5, is stricter) but for most production needs, strict JSON mode is already very strong.

An important nuance: strict JSON mode enforces the schema, not the semantics. The model can emit a well-typed proposal whose meaning is wrong: it can choose a real counterparty, a valid currency, a positive amount, and still produce an action that the rest of the system cannot execute (because the counterparty has no account in that currency, for example). The Schema-Bound architecture stops at the structural boundary; semantic checks belong to the layer below the parser, in the ontology resolver.

### 4.4 What is provable at this level

Schema-bound architectures admit a clean set of verifiable properties, all of them about structural correctness, none of them about meaning:

- **Schema soundness.** The schema definition is well-formed under the schema language's rules. This is type-checked by TypeScript / Python / your language of choice; the type-checker either accepts the schema or rejects it.
- **Validator totality.** The functions that parse and validate (`safeParse` in Zod, `model_validate` in Pydantic, and equivalents) are total: they always return either a parsed value or a typed error, and never crash. This property is provable from the library's source code; the major libraries publish formal arguments for it.
- **Enumeration closure.** Under strict JSON-mode enforcement, the model cannot emit a value outside the declared enumeration. The provider documents this property; for open-weight models with grammar-constrained decoding, it follows by construction from the decoder algorithm.
- **Discriminated dispatch exhaustivity.** Modern type-checkers (TypeScript with strict mode, F#, Rust) verify that every branch of a discriminated union is handled; if you add a new branch and forget to update the dispatch, the compiler refuses to build.

### 4.5 What is not provable at this level

Equally importantly, schema-bound architectures cannot prove, and should not claim to prove:

- **Semantic correctness.** A well-typed payment to the wrong counterparty for the wrong reason still parses. The schema does not know what was intended; it only knows what was said.
- **Cross-field invariants.** A proposal with `currency=EUR` and `rail=USDC_BASE` parses (both are valid enumerations), but it is incoherent: USDC on Base is not a EUR-denominated rail. Catching this requires consulting state outside the schema.
- **System-level guarantees.** "No payment exceeds the mandate's daily cap" is a property of the system over time, not of any one proposal. The schema cannot see the daily cap; the schema cannot see the history.

- **Compositional reasoning.** Two well-typed proposals can together violate a daily-spend invariant when neither would violate it alone. Schema validation is per-call; invariants over multiple calls require state and are out of scope.

#### The schema-bound default

For most production language-model agents, those operating within a known set of intent kinds and a known set of entities, with downstream code that can dispatch on type, Architecture B is the right starting point. It is mechanically simple, the tooling is mature, and most failure modes are observable through the parser's typed error stream rather than diagnosed after the fact in production. It is also the architecture against which any move toward C or D should be justified. A Schema-Bound system with a carefully-designed ontology layer below the parser handles the great majority of agentic use cases.

## 5 Architecture C: Grammar-Constrained Decoding

The third architecture pushes by-construction enforcement deeper into the generation process. Rather than letting the model generate freely and then validating, the decoder restricts the model's output choices at every token step so that only in-grammar continuations are possible. By the time the model finishes, the output is in-grammar; no separate validation step is needed for shape.

**Architecture C: Grammar-Constrained Decoding**

*ontology as a formal grammar; the decoder forbids out-of-grammar tokens at each step*

**Token-by-token decoding (committed prefix)**

`{"@type":` | `"Payment"` | `,"cp":` | `"Acme` | `Cloud"` | `,"amount":` | `1500` | `?`

**Valid next-token mask at this position**

Acme | OpenAI | Northwind

*allowed: kept in the distribution*

~~Bitcoin~~ | ~~Wells~~ | ~~anything else~~

*forbidden: probability set to zero*

*At each step the parser state induces a set of legal next tokens. Tokens outside the set are masked to zero probability before sampling, so out-of-grammar output cannot occur.*

### 5.1 How language-model decoding works, briefly

To understand grammar-constrained decoding, it helps to understand how a language model produces text in the first place. A model's output is generated one token at a time. A token is the model's atomic unit, a short sequence of characters, like "and", "ing", or "`"@type`"; modern models have vocabularies of around 50,000 to 200,000 such tokens. At each step, the model computes a probability for every token in its vocabulary, conditioned on everything generated so far. The decoder picks one token from

this probability distribution (sometimes the most likely, sometimes a sample weighted by probability) appends it to the output, and repeats.

Grammar-constrained decoding intervenes in this loop. At each step, the system maintains a parser state tracking what the current partial output is, like a parser reading the output as it is produced. From the grammar and the parser state, the system computes the set of tokens that, if added next, would keep the output on a valid path. Tokens outside this set are masked: their probability is set to zero. The model's distribution is renormalized over the remaining valid tokens, and one is sampled. The output is in-grammar at every step; no token outside the grammar can ever appear.

### 5.2 Grammars, in plain terms

A grammar, in computer science, is a precise set of rules describing what strings of characters count as "valid" in some language. Just as English grammar says a sentence must have a subject and a verb in the right order, a formal grammar for JSON says an object must start with "{" and end with "}", with comma-separated key-value pairs in between. Grammars are written in special notations; the one most associated with grammar-constrained LLM decoding is GBNF (Grammatical BNF, used by llama.cpp), which extends the classic Backus-Naur Form notation that has been used for fifty years to describe programming languages.

A grammar for a payment proposal might look like the excerpt below. Each line defines a rule: the left-hand side is a name, the right-hand side is the set of expansions allowed for that name. The first line says: a root expands to either a payment or a not-proposable. The grammar is a kind of recipe: to generate valid output, the system "fills in" the rules left-to-right.

```
root            ::= payment | not-proposable
payment         ::= "{\"@type\":\"Payment\"," cp "," amt "," cur "}"
cp              ::= "\"counterparty\":" cp-name
cp-name         ::= "\"Acme Cloud\"" | "\"OpenAI\"" | "\"Northwind\""
amt             ::= "\"amountMinor\":" integer
cur             ::= "\"currency\":" currency-code
currency-code   ::= "\"EUR\"" | "\"USD\""
not-proposable  ::= "{\"@type\":\"NotProposable\"," "\"reason\":" reason "}"
reason          ::= "\"missing_cp\"" | "\"missing_ev\""
```

### 5.3 The grammar as ontology

In Architecture C, the grammar plays the role the schema played in Architecture B, but it is enforced earlier and more strictly. Where B says "the output must parse against this schema or we reject it," C says "the model cannot generate anything but in-grammar output, period." This eliminates entire classes of failures: truncated JSON, escape errors, hallucinated fields, accidental extra tokens at the end.

The grammar can be open in places where Architecture B would require a free-text field, and closed in places where Architecture B would have to validate after the fact. A counterparty name can be a literal alternation of registered names; a justification can be an arbitrary string up to a length bound. Every constraint that can be expressed grammatically is baked in.

### 5.4 Tooling landscape

Grammar-constrained decoding is no longer experimental. The current production-grade tools split into three categories.

- **Open-weight runtimes with native support.** llama.cpp accepts GBNF grammars directly. vLLM integrates with Outlines. The Hugging Face Transformers library supports Outlines and Guidance. These give the strongest control because you have direct decoder access.
- **Provider-side structured outputs.** OpenAI's structured outputs (driven by JSON Schema), Anthropic's tool-use enforcement, Google's controlled generation. These are usually restricted to JSON-Schema-style grammars rather than arbitrary context-free grammars, but they cover the majority of practical needs.
- **Compilation libraries.** Outlines (from dottxt), Guidance (from Microsoft), LMQL, and XGrammar compile grammars (or higher-level templates) into per-step token masks. They target whichever runtime accepts a mask, decoupling the grammar source from the runtime.

### 5.5 What is provable at this level

Grammar-constrained decoding admits a stronger and more specific set of provable properties than Architecture B:

- **Output in-grammar.** By construction. The decoder cannot emit a token outside the grammar; this is a property of the decoder, not of the output. You do not check it after the fact; you make it impossible.
- **Decoder soundness.** The token mask is correctly derived from the grammar's current parser state. This is provable about the masking algorithm itself, separately from any particular grammar.
- **Closure.** No symbol outside the grammar can ever appear in the output. Stronger than Architecture B's "output parses against schema," because it eliminates partial-output failures.
- **Decidable equivalence.** For regular grammars and a subset of context-free grammars, there is a mechanical procedure to decide whether two grammars accept the same set of strings. This lets you refactor a grammar and prove the new version accepts exactly the same outputs.

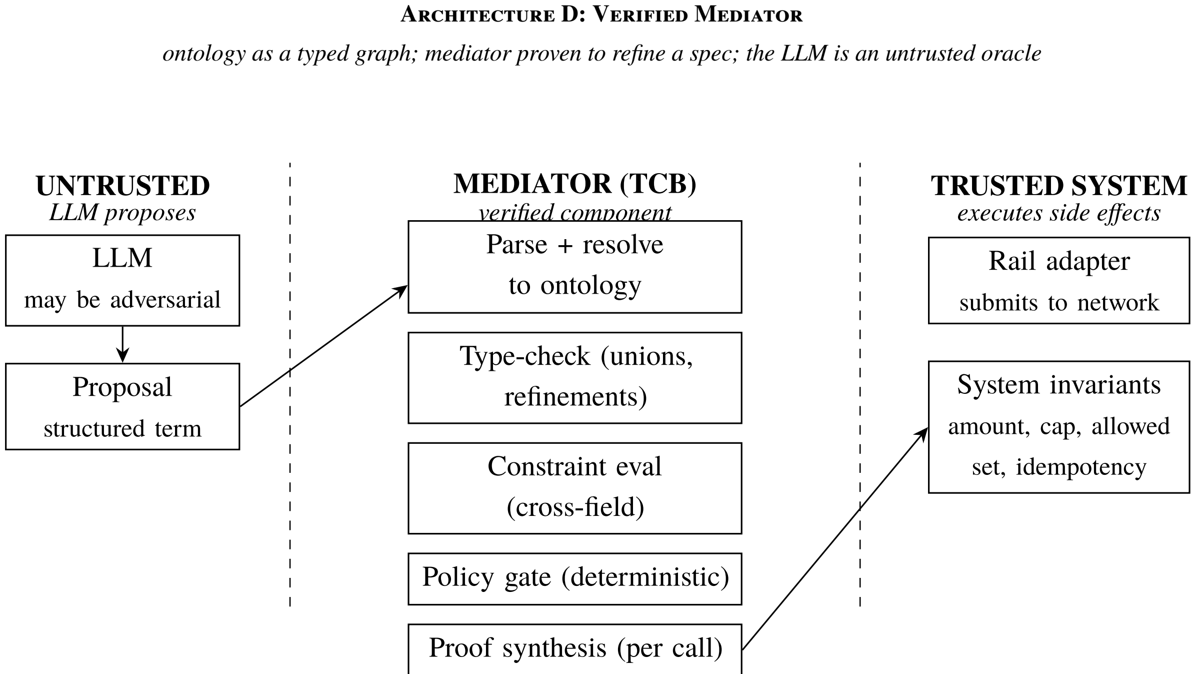
### 5.6 Trade-offs

Grammar-constrained decoding is not free. The notable costs are:

- **Provider dependency.** Not all commercial APIs expose decoder hooks. Closed-API models may not support arbitrary grammars; you may be restricted to whatever the provider has chosen to support.
- **Performance cost.** Per-token mask computation has a cost. Modern implementations (especially XGrammar) have reduced this to negligible levels for most grammars, but pathological grammars can still slow generation noticeably.
- **Distortion of generation.** When the best continuation under the unconstrained model is forbidden, the model picks a worse in-grammar continuation. This effect is real but usually small; it matters most for tasks where the model's quality depends on stylistic choices the grammar restricts.
- **Cross-field invariants still escape.** A grammar cannot express "currency must match the counterparty's account currency" without state outside the grammar. Cross-field constraints remain the ontology layer's job.

## 6 Architecture D: Verified Mediator

The most demanding of the four architectures and the only one that gives you proven system-level guarantees. The ontology is a typed graph; a mediator component sits between the language model and the executing system; the mediator is formally specified and proven to refine its specification.




*The mediator is the only component that needs proofs. Everything to its right trusts it; the LLM to its left is verified against, never trusted.*

### 6.1 The three roles, in plain terms

Architecture D introduces three roles in the system, separated by trust boundaries.

The untrusted role contains the language model. It produces proposals, structured terms in a known signature. We make no semantic warranty about these terms: the model may be prompt-injected, may hallucinate, may even be replaced by an adversary entirely. The architecture is designed to be safe regardless.

The trusted role contains the system that actually performs side effects: the rail adapters that talk to settlement networks, the database that commits the transaction, the audit log that records what happened. These components are deterministic and well-tested; we trust them to do what they are told.

Between them sits the mediator. The mediator is the Trusted Computing Base, the TCB. In security engineering, the TCB is the set of components whose correctness everything else depends on; if the TCB is wrong, no other defenses help. In this architecture, the mediator is the only component that needs proofs, because every other component either trusts the mediator (the trusted system) or is verified-against by the mediator (the LLM's untrusted output).

### 6.2 What the proof actually says

The mediator is specified in a proof assistant, a software environment specifically designed for writing mathematical proofs as if they were programs, with the computer mechanically checking each step. The major proof assistants used today include Lean 4, Coq, Dafny, F*, Isabelle/HOL, and (the focus of §7

below) the B method. The mediator function is implemented in the same language as its specification. The proof obligation is that the mediator's output, when it returns an approved action, satisfies every system safety invariant.

> **The theorem, informally**
>
> For every proposal $p$ that the language model emits, and every authorization context $M$ (the mandate): if $mediator(p, M, state)$ returns $approved(action)$, then $action$ satisfies all of $system_invariants(M, state)$. The language model is outside the TCB. The proof says: even if the model is adversarial, prompt-injected, hallucinating, deliberately attacking, no approved action violates the invariants.

### 6.3 Worked example: a payment mediator

The agentic-payment domain admits a clean specification with a small core. The proposal is a discriminated union of Payment and Refuse. The mandate is a record of limits, allowed sets, and a currency. The mediator pattern-matches on the proposal kind and either returns an ApprovedAction with an attached proof witness, or returns nothing (signaling refusal). The specification is short enough to fit on one page; the proof obligations are correspondingly tractable.

Specification (Lean 4 sketch)

```
structure Mandate where
  maxSingle : Nat
  dailyCap  : Nat
  allowed   : Finset Counterparty
  rails     : Finset Rail
  currency  : Currency

inductive Proposal where
  | payment (cp : Counterparty)
      (amt : Nat) (cur : Currency)
      (rail : Rail)
  | refuse (reason : RefuseReason)

structure ApprovedAction where
  intent : Proposal
  proof  : authorizes M state intent

def mediator :
  Proposal -> Mandate -> State ->
    Option ApprovedAction :=
  fun p M s =>
    if h : canAuthorize p M s then
      some ⟨p, ⟩h
    else none
```

Extracted mediator (TypeScript)

```
function mediator(
  p: Proposal, M: Mandate, s: State
): ApprovedAction | null {
  switch (p.kind) {
  case 'payment':
    if (p.amount > M.maxSingle)
      return null;
    if (s.dailySpend + p.amount
        > M.dailyCap) return null;
    if (!M.allowed.has(p.counterparty))
      return null;
    if (!M.rails.has(p.rail))
      return null;
    if (p.currency !== M.currency)
      return null;
    return {
      intent: p,
      proof: { /* witness, log */ }
    };
  case 'refuse':
    return null;
  }
}
```

*Lean-style specification on the left; the extracted runtime mediator on the right. The four theorems below are proved once and hold over all runs.*

Four theorems anchor the system. Each is a small, scoped statement; together they characterize the mediator's correctness.

- **MEDIATOR-SOUND.** If the mediator approves an action, the action satisfies the system invariants. This is the main correctness theorem; everything else supports it.
- **REFUSAL-EXHAUSTIVE.** The mediator approves an action only when the authorization predicate (`canAuthorize`) holds. There is no alternate path to approval, no "and also" escape hatch.
- **EXTRACTION-FAITHFUL.** The runtime mediator (compiled or extracted to TypeScript, Rust, Scala, Ada, or C) is denotationally equivalent to the specification. The translation step preserves meaning.
- **MONOTONE-IN-SPEND.** If the mediator approves an action at a higher cumulative spend, it would have approved it at a lower one. This lets auditors reason about past states without replaying every transaction.

### 6.4 What this actually buys

The proven invariants are not academic. Each one corresponds to a real claim the system can make to a regulator, an auditor, or a customer. MEDIATOR-SOUND says no approved payment ever violates the mandate, not "we have tests for this" or "we have alerts for this," but we have a mechanically-checked proof that this cannot happen. The proof holds regardless of what the model does, including being prompt-injected by hostile user input or replaced by a malicious version.

REFUSAL-EXHAUSTIVE rules out a class of escalation bugs that has bedeviled real systems: an "and also" path that approves an action without the authorization predicate holding, because someone wrote an extra branch in a moment of carelessness. EXTRACTION-FAITHFUL means an auditor can read the specification, then read the running code, and see that they are equivalent line-by-line, no hand-translation gap where bugs hide. MONOTONE-IN-SPEND lets you reason about historical system states without having to replay every transaction; if the mediator approved an action at the time, it would still have approved it under earlier conditions.

### 6.5 When this architecture is worth the investment

Architecture D is genuinely expensive. A team capable of writing and maintaining it productively is roughly the team capable of running a small formal-methods practice: three to five engineers with proof-assistant experience, plus the rest of the engineering organization able to read the specification language at least passively. Conservative estimate for an initial mediator from scratch: six to nine engineer-months. Ongoing cost: real, but bounded, proofs are maintained, not rewritten, as the specification evolves.

Architecture D earns its place when both of two conditions hold:

- **(a) The failure mode is genuinely irreversible.** Money moves and cannot be unmoved. Medical orders take effect and cannot be retracted. Infrastructure changes propagate before a human can intervene.
- **(b) An external party will eventually ask "prove that this cannot happen."** A regulator (financial-conduct authority, medical-device regulator, transportation safety authority); an auditor (the firm's external auditors, a customer's security review); a court (after an incident, to apportion liability).

Both conditions matter. (a) without (b) gives you a system that should be verified but where the cost is hard to justify against competing engineering priorities. (b) without (a) gives you a system where verification is gold-plating because the worst case is recoverable. Together, they make Architecture D the responsible choice and increasingly the expected one in regulated industries.

## 7 Formal methods landscape: including the B method

Architecture D refers repeatedly to “the proof assistant” and “the specification language.” There are several to choose from, and the choice has real consequences for cost, deployability, and how well the verified system will defend itself in front of a regulator. This section surveys the major options. We treat the B method in greater depth than the others, because it has a property no other proof assistant matches: it has been used to verify deployed, safety-critical infrastructure at industrial scale, and a regulator who sees B in a verification dossier has likely seen it before.

### 7.1 A tour of the major options

Eight tools cover most of the space. They differ in paradigm (the underlying logic and how proofs are structured), learning curve (the effort for a new engineer to become productive), code extraction (whether the specification can be automatically compiled to deployable code), and industrial track record (whether the tool has been used in production beyond academic settings).

**Formal methods landscape: tools for verified mediators**

*paradigm, learning curve, code extraction, and industrial track record*

| Tool | Paradigm | Learning | Code extraction | Track record / best fit |
|---|---|---|---|---|
| Coq | Dependent type theory | Steep | Extracts to OCaml, Haskell | CompCert, CertiKOS; when mathematical rigor matters most |
| Lean 4 | Dependent types + tactics | Steep, improving | Native compilation to C | mathlib, growing in CS; modern alternative to Coq, better tooling |
| Isabelle/HOL | Higher-order logic | Steep | Code gen to SML, OCaml, Haskell | seL4 verified microkernel; maximum proof automation |
| Dafny | Imperative + SMT backend | Moderate | C#, Go, Java | AWS encryption SDK, IronFleet; engineers, not researchers |
| F* | Verification via refinement types | Moderate | OCaml, F#, C | HACL*, miTLS; when code and proof should be one program |
| TLA+ | Set theory + temporal logic | Moderate (spec only) | No extraction (model only) | AWS S3, DynamoDB; verify the design, implement separately |
| B / Event-B | Refinement + set theory | Moderate, industrial | B0 to Ada or C | Paris Metro 14, RER A, ETCS; transport, medical, banking regulators |
| Refinement types | Type-system extensions | Light | Already code | Liquid Haskell, refined TypeScript; verification without leaving the host language |

*The B method earns its distinct callout: more deployed safety-critical track record than any other formal method on this page.*

### 7.2 Coq, Lean 4, and Isabelle/HOL

These three are the major academic-research-grade proof assistants. They share a common heritage in type theory and higher-order logic, and they all support proving deep mathematical theorems. Coq is the oldest of the three (1989), with the strongest track record in compiler verification: CompCert, a C compiler verified end-to-end in Coq, is famously bug-free in a way mainstream compilers are not. Lean 4 is the newest (2021), with significantly better tooling, an active community of mathematicians

(the mathlib project), and growing adoption in computer-science verification. Isabelle/HOL has been the workhorse of large-scale verification projects since the 1990s; seL4, a verified microkernel used in defense and aerospace, is its most famous deployment.

All three are powerful, expressive, and demand a real learning investment. A new engineer needs months of focused study to become productive. Once productive, they can verify properties of remarkable depth and subtlety. For an LLM-mediator project, the question is whether that depth is necessary. For a payment mediator whose logic is essentially a state machine with cross-field invariants, it may be more than is needed.

### 7.3 Dafny and F*

These two are the engineering-grade alternatives, designed for working software engineers rather than type theorists. Dafny (from Microsoft Research) lets you write code in a familiar imperative style, annotate it with pre-conditions, post-conditions, and invariants, and have an SMT solver (Satisfiability Modulo Theories, an automatic theorem prover for first-order logic with arithmetic and arrays) discharge the proof obligations automatically when possible. The result is verification that feels like rich type-checking rather than mathematical theorem-proving. AWS used Dafny to verify parts of its encryption SDK and its IronFleet distributed-systems work.

F* (also Microsoft Research) takes a different approach: it is a programming language whose type system is rich enough to express verification properties as types. You write code; the type checker either accepts it or rejects it. The HACL* project, verified implementations of cryptographic primitives used in real TLS deployments, is F*'s flagship.

For a verified mediator, either of these is a plausible choice. They lower the formal-methods barrier substantially compared to Coq or Lean, while still delivering real machine-checked guarantees. The trade-off is expressiveness: there are theorems Coq or Lean can prove that Dafny or F* cannot.

### 7.4 TLA+ and refinement types

These two are the lightweight alternatives, each making a different trade-off. TLA+ (created by Leslie Lamport) is a specification language for distributed systems. You write a model of the system, state the invariants, and a model checker explores the state space looking for violations. TLA+ does not produce executable code: you implement the system separately, and the TLA+ model serves as the specification that the implementation must follow. AWS uses TLA+ extensively for S3, DynamoDB, and several other services.

Refinement types, exemplified by Liquid Haskell and the various refined-TypeScript projects, extend an existing programming language's type system to express verification properties. Instead of "this is an integer," you can say "this is a positive integer less than the daily cap." The compiler checks the refinement at build time. The cost is low (you stay in your existing language), the gain is modest (you cannot express deep theorems), but it is real verification, and it earns its keep on the schema and validator layers of any architecture.

### 7.5 The B method: what it is and why it matters

The B method was created by Jean-Raymond Abrial, a French computer scientist who had previously co-created Z (an earlier specification language). Abrial published the foundational reference, The B-Book, in 1996. The B method has two main variants today: classical B (used with the Atelier B toolchain from ClearSy, a French software-engineering company) and Event-B (a successor designed for distributed

and reactive systems, supported by the Rodin Platform).

What distinguishes B from the academic proof assistants is its origin: B was designed from the outset for industrial software engineering, with the goal of producing deployable code that has been mechanically proved correct against a specification. The methodology, the tooling, and the certification story all reflect this orientation.

### 7.6 The refinement methodology

B's central technique is refinement. You start with an abstract specification, typically a state machine with invariants, written in set-theoretic notation that emphasizes what the system must do, not how. You then refine this specification, in one or more deliberate steps, into progressively more concrete versions. Each refinement step replaces abstract data structures with concrete ones, abstract operations with executable algorithms, and non-deterministic choices with deterministic ones. The tooling generates a set of proof obligations for each step: mathematical statements that, if proved, guarantee the refinement preserves the abstract invariants.

The endpoint of the refinement chain is B0, a deterministic, executable subset of the B notation. B0 looks like a structured programming language. Atelier B can mechanically translate B0 code into Ada or C, with the translator itself certified as part of the toolchain. The result is deployable code that has a chain of mathematical evidence back to the abstract specification.

**B-METHOD REFINEMENT CHAIN: APPLIED TO AN LLM MEDIATOR**

*the abstract specification descends through refinements to deployable Ada / C code, each step proved*

**Level 0: Abstract machine.** *Set-theoretic model of the mediator. Invariants stated; operations described non-deterministically.*

```
MACHINE Mediator
VARIABLES allowed, daily_spend
INVARIANT allowed ⊆ COUNTERPARTIES ∧ daily_spend ∈ NAT
OPERATIONS
  approve(p) = PRE p.amount ≤ max_single ∧
    daily_spend + p.amount ≤ daily_cap THEN ...
```

↓ proof obligation discharged

**Level 1: Refinement.** *Add data structures (set becomes table). Each refinement is proved to preserve the abstract invariants.*

```
REFINEMENT Mediator_R1
REFINES Mediator
VARIABLES allowed_table, spend_register
INVARIANT allowed_table ∈ NAT → COUNTERPARTIES ∧
  ran(allowed_table) = allowed ∧ spend_register = daily_spend
OPERATIONS approve(p) = ... /* same body, refined */
```

↓ proof obligation discharged

**Level 2: Implementation (B0).** *Deterministic, executable subset of B. No more set-comprehension; only loops, conditionals, basic types.*

```
IMPLEMENTATION Mediator_I
REFINES Mediator_R1
OPERATIONS
  approve(p) = VAR found IN found := FALSE;
    /* linear scan or hash lookup */
    IF p.amount ≤ MAX_SINGLE THEN ... END
```

↓ proof obligation discharged

**Level 3: Target language.** *Atelier B translates B0 to Ada (SIL-4 certified) or C. The translation is itself a verified component.*

```
procedure Approve (P : in Proposal; OK : out Boolean) is
  Spent : Natural := Spend_Register;
begin
  if P.Amount > Max_Single then OK := False;
  elsif Spent + P.Amount > Daily_Cap then OK := False;
  else ... end if;
end Approve;
```

*Each downward step is a proof, mechanically checked by Atelier B. The final Ada or C is deployable, with a chain of evidence to the abstract specification.*

### 7.7 Industrial track record

B has been used to verify infrastructure that you have probably walked through or ridden on. Selected deployments:

- **Paris Metro Line 14 (Meteor)**, fully driverless since 1998. The automatic train operation and the platform-door safety system were specified, refined, and proved in B. The line has operated without a signaling-related safety incident for over a quarter-century.
- **RER A automatic operation**, the Paris regional express network's busiest line; the automatic operation system on the central section was upgraded using B.
- **Roissy-Charles-de-Gaulle airport shuttle (CDGVAL)**, the automated people-mover between terminals at the Paris airport; verified in B.
- **Sao Paulo Metro Line 4**, the first fully driverless metro line in the Americas; signaling verified in B.
- **ETCS railway signaling**, the European Train Control System, the continent's standard for mainline rail safety; Alstom uses B extensively for SIL-4 (the highest safety integrity level) components.
- **Areva nuclear safety systems**, components in nuclear plant control systems have been developed with B.

No other proof assistant on the page has comparable production deployment in safety-critical domains. seL4 (Isabelle/HOL) is in some defense and aerospace systems but at smaller scale; CompCert (Coq) is a verified C compiler, useful but not itself a deployed safety system; the AWS uses of Dafny and TLA+ are at large scale but for availability rather than safety regulation. B is the formal method with the deepest regulator-facing track record.

### 7.8 Operationalizing the B method for an LLM mediator

How does B apply to a Verified Mediator? The shape of the application is natural: a mediator is essentially a state machine with invariants over a transaction history, and that is precisely the kind of system B was designed to specify. The operational steps for a team adopting B for this purpose:

- **Specify the mediator as an abstract B machine.** Define the state (the mandate, the spend history, the set of allowed counterparties). Define the invariants (no payment exceeds the per-transaction cap; the cumulative spend does not exceed the daily cap; only verified counterparties may be paid).

Define the operations (propose, approve, reject) non-deterministically, capturing what may happen without saying how.

- **Refine the abstract machine into intermediate machines.** Replace mathematical sets with concrete data structures (hash maps, sorted arrays). Replace abstract operations with algorithms. Each refinement preserves the invariants and is mechanically proved to do so.
- **Refine to B0, the executable subset.** All operations are now deterministic. All data structures are concrete. The B0 code is, syntactically, close to Ada or C.
- **Translate B0 to Ada (preferred for SIL-4) or C.** Atelier B does this mechanically; the output is deployable. For a system that must be Ada (transportation safety, defense), this is the standard route. For systems where C is acceptable, B can target C directly.
- **Wrap the deployed mediator in a service interface** that receives proposals from the LLM-facing layer and returns approve/refuse decisions. The service is a thin shell around the verified core; the proof artifact follows the deployment.

### 7.9 Where the B method earns its place

The argument for choosing B over a more expressive but less industrially-deployed alternative comes down to four points.

- **Regulator familiarity.** In the European Union, transportation safety regulators (ERA, national equivalents) have direct experience reviewing B-verified systems.
- **Deployability discipline.** B forces the refinement-to-B0-to-Ada/C pipeline. There is no gap between "we proved properties of an abstract specification" and "what runs in production." Other proof assistants offer code extraction but the extraction is often less central to the methodology.
- **Methodology fit for the problem.** A mediator is a state machine with invariants. B's methodology was designed for exactly this shape. Lean and Coq can express more, dependent types, advanced category theory, but a payment mediator does not need that expressive power.
- **Tooling maturity.** Atelier B has been in industrial use since the 1990s. The toolchain is mature, the user community (concentrated in France, Brazil, the United Kingdom, and Japan) is professional, and the proof obligations the tool generates are typically discharged with a high degree of automation.

#### The honest caveats

The B method has costs. Its notation is set-theoretic and distinctive; engineers without a formal-methods background need real training (a multi-week course; a textbook study) before they are productive. The certified-engineer community is small enough that hiring is a real constraint outside France. The tooling, while mature, is a commercial product from ClearSy with the support model of a specialized vendor rather than a mass-market platform. Event-B (the modern variant, supported by the open-source Rodin Platform) addresses some of these concerns: it is more expressive for reactive and distributed systems, the tooling is open-source, and the community has broader academic engagement. For a fresh project, Event-B is often the better starting point even where classical B would historically have been chosen.

### 7.10 A decision rule for picking the formal method

If you are building a Verified Mediator and have to pick the underlying formal method, the following questions usually settle the choice within minutes:

- **Will this system face a transportation, medical, or banking safety regulator?** If yes, B (or Event-B) starts the conversation in the easiest place. The track record is the argument.
- **Is the team primarily software engineers, or does it include formal-methods specialists?** Engineers without proof-assistant background find Dafny or F* the gentlest entry; Lean 4 has the best modern tooling among the academic options.
- **Do you need to extract running code, or only verify a specification?** Code-extraction needs steer you toward Lean, F*, B/Event-B, or Dafny. Specification-only needs (where the implementation is built separately) make TLA+ the natural choice.
- **How expressive does the specification need to be?** For a state-machine-with-invariants (the typical mediator), B/Event-B, Dafny, or F* are sufficient. For deeper theorems (cryptography, compilers, mathematics), Lean 4, Coq, or Isabelle/HOL pull ahead.

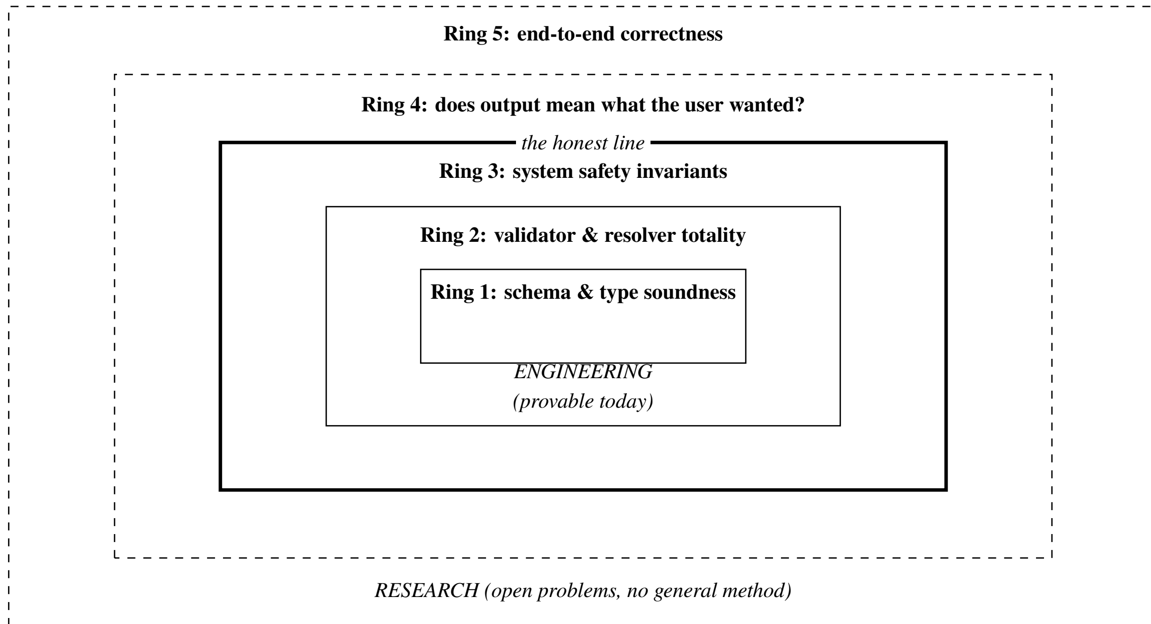
For most LLM-mediator projects in regulated industries, the honest answer is: B (or Event-B) is the responsible default, with Dafny or F* as engineering-driven alternatives, and Lean 4 or Coq reserved for cases where the deeper expressiveness is genuinely needed.

## 8 Formal verification: the honest taxonomy

The phrase "formally verified" is used loosely in marketing and precisely in the literature, and the gap between the two does real damage. A system advertised as "verified" may have a proved schema and an unproved everything-else; a skeptical regulator who has been burned before will discount the whole claim. This section draws the honest line. We organize verifiability into five concentric rings, from the innermost (most tractable, provable today with off-the-shelf tools) to the outermost (open research problems with no general method). The boundary between Ring 3 and Ring 4 is the honest line: inside it is engineering, outside it is research.

**Five rings of verifiability**

*inner rings are provable today; the honest line separates engineering from research*




*Rings 1 to 3 are tractable with tools that exist today. Rings 4 and 5 are genuine research frontiers; claiming to have "solved" them is the most common form of verification overclaim.*

### 8.1 The three engineering rings (inside the honest line)

**Ring 1: schema and type soundness.** The innermost ring. Can you prove that outputs conform to a declared structure, that values come from declared enumerations, that types are respected? Yes, routinely, with tools available today. A Zod schema, a JSON Schema validator, a TypeScript type-checker: each gives you Ring 1 for a few hours of work. Every architecture from B onward operates at least at this ring. There is no excuse for a high-stakes system not to reach it.

**Ring 2: validator and resolver totality.** The second ring. Can you prove that the validation and resolution functions are total, that they always return a definite answer (a parsed value or a typed error) and never crash, loop forever, or throw an unhandled exception? Yes, with modest additional effort. Totality checkers exist for most languages; in a dependently-typed or refinement-typed setting, totality is enforced by the type system. This ring matters because a validator that can crash is a denial-of-service vector and a source of undefined behavior.

**Ring 3: system safety invariants.** The third ring, and the outer boundary of the engineering territory. Can you prove that no execution of the system, regardless of what the language model outputs, ever violates a stated safety invariant (no payment exceeds the cap, no unauthorized counterparty is paid, no action without a recorded approval)? Yes, this is exactly what Architecture D's verified mediator delivers, at the cost of weeks to months of formal-methods work. This is the frontier of what is routinely achievable. It is hard, it is expensive, but it is engineering, not research: the methods exist, the tools exist, the track record exists.

### 8.2 The two research rings (outside the honest line)

**Ring 4: does the output mean what the user wanted?** The fourth ring crosses into research. Can you prove that the language model's output actually captures the user's intent, that when the user said

"pay the cloud provider" and the model proposed paying Acme Cloud, the model correctly understood which entity was meant? This is not provable with current methods. It depends on the model's internal representations, on the grounding of language in referents, on what "intent" even means formally. There are partial, heuristic approaches (confidence estimation, consistency checks, asking the model to justify itself) but none is a proof, and none should be presented as one.

**Ring 5: end-to-end correctness.** The outermost ring. Can you prove that the entire system, from the user's natural-language request through the model through the mediator to the executed action, does the globally right thing? This compounds the Ring 4 problem with every other source of specification gap. It is an open problem in the deepest sense: we do not have a formal definition of "the globally right thing" for an open-ended natural-language task, let alone a method to prove a system achieves it. Honest practitioners do not claim this ring.

**The inversion**

Notice what the rings invert. The intuitive worry about language-model systems is "how do I know the AI understood me?" (Ring 4) and "how do I know the whole thing does the right thing?" (Ring 5). Those are exactly the rings we cannot formally close. What we can close, completely and with mechanical proofs, are the rings the intuition treats as boring plumbing: structure (Ring 1), totality (Ring 2), and system safety invariants (Ring 3). The engineering response to this inversion is not to keep trying to verify the model. It is to design the system so that the model's correctness is never load-bearing. You verify the cage, not the animal. If the cage is sound, it does not matter that you cannot prove the animal is tame: it cannot do harm regardless. Architecture D is the cage made explicit.

## 9 Hybrid composition: what real stacks look like

The four architectures are reference points, not products. Real production systems compose them: they use Architecture B's schema at the boundary, Architecture C's grammar for the fields where it pays, and Architecture D's verified mediator for the irreversible final step, with the prompt-bound residue (Architecture A) still present as the system prompt that frames the whole interaction. A mature stack is a layered pipeline in which each layer adds one verifiable property and passes a cleaner artifact to the next.

**The seven-layer composition**

*each layer adds exactly one verifiable property and hands a cleaner artifact down*

data flows down, assurance accumulates

| Layer | Verifies |
|---|---|
| **1. Input layer.** *sanitize, de-inject untrusted text* | verifies: input well-formed; no smuggled control |
| **2. Prompt layer.** *system prompt + projected vocabulary* | verifies: nothing formal (English instructions) |
| **3. Decoder layer.** *grammar-constrained generation* | verifies: output in-grammar, by construction |
| **4. Validation layer.** *schema parse into typed objects* | verifies: shape, types, enumeration closure |
| **5. Ontology layer.** *resolve references, cross-field checks* | verifies: references resolve; constraints hold |
| **6. Mediator layer.** *policy gate + per-call proof (TCB)* | verifies: system safety invariants |
| **7. Execution layer.** *deterministic side effect + audit* | verifies: idempotency; complete audit trail |

*No single layer carries the whole burden. The model lives at layers 2 and 3; everything below treats its output as untrusted and progressively constrains it.*

## 9.1 Reading the stack

The stack is read top to bottom as the path a single request takes. The user's text enters at layer 1, where it is sanitized: untrusted content is neutralized so that a user cannot inject instructions that hijack the system prompt. Layer 2 assembles the prompt, including the projected vocabulary (the subset of the ontology relevant to this request, rendered as enumerations the model can see). Layer 3 generates the response under grammar constraints, so the output is in-grammar by construction. Layer 4 parses that output into typed objects, confirming shape and enumeration closure. Layer 5 resolves references against the live ontology (which database record does "Acme Cloud" denote?) and checks cross-field constraints that the grammar could not express. Layer 6, the mediator, applies the policy gate and discharges the per-call proof, the one layer that needs formal verification. Layer 7 performs the side effect deterministically and writes the audit record.

## 9.2 The load-bearing principle

The principle that makes the composition sound is that no single layer carries the whole burden, and the model's correctness is never load-bearing. The language model operates at layers 2 and 3; it proposes. Every layer below it treats the model's output as untrusted input and adds a constraint the model could not be relied upon to satisfy. By the time a proposal reaches layer 7, it has been shaped by a grammar, validated against a schema, resolved against the ontology, and proved to satisfy the system invariants. If the model misbehaves at layer 2 or 3, the misbehavior is caught downstream; it cannot reach execution.

## 9.3 Where to stop

Not every system needs all seven layers. The right depth depends on stakes. A low-stakes system may implement only layers 1, 2, 4, and 7 (Architecture B with input sanitization). A high-stakes irreversible system implements all seven, with layer 6 carrying a real proof. The decision of how deep to build is

the subject of the next two sections: which architecture to pick for a given situation, and how to grow from a shallow stack to a deep one as the stakes rise.

## 10 The same task through all four architectures

To make the differences concrete, this section walks a single agentic-payment task through all four architectures, in one fixed workspace, and observes where each one catches (or fails to catch) the same underlying problem.

### 10.1 The workspace and the task

The workspace has three registered counterparties, each with specific payment capabilities:

- **Acme Cloud**, payable via Stripe, in EUR.
- **OpenAI**, payable via Stripe, in USD only. No EUR account exists for this counterparty.
- **Northwind**, payable via SEPA bank transfer to an IBAN, in EUR.

The mandate under which the agent operates is EUR-only: the agent may initiate payments denominated in EUR and no other currency. The task handed to the agent is: *Pay the OpenAI invoice.* This task is a trap. OpenAI can only be paid in USD, but the mandate permits only EUR. There is no correct payment; the only correct action is a clean, well-reasoned refusal. The interesting question is whether each architecture refuses correctly, refuses for the right reason, and can demonstrate that the refusal was correct.

**One task, four architectures**

*"Pay the OpenAI invoice" under a EUR-only mandate, where OpenAI is USD-only*

| Arch. | What the model emits | What the check does | Outcome |
|---|---|---|---|
| A. Prompt-Bound | Free text wrapping `{OpenAI, 1500, EUR}`; the model invents a plausible payment | Regex extracts JSON; JSON parses; shape looks valid. No check knows OpenAI lacks a EUR rail | Dispatched to the rail adapter, which fails late with "no rail-account for counterparty." Cause untraceable from the error |
| B. Schema-Bound | Typed `Payment{OpenAI, 1500, EUR}`; all fields are valid enumeration members | Schema parse succeeds (enums valid). The ontology layer resolves the rail and finds no (OpenAI, EUR) account: cross-field check fails | Clean typed rejection with a reason, or the model emits `NotProposable`. Caught before execution, with a diagnosis |
| C. Grammar-Constrained | With a per-request projected grammar, (OpenAI, EUR) is not even expressible; the only legal paths are USD (mandate-barred) or `NotProposable` | The grammar forbids the bad combination at decode time; the model cannot emit it. It is forced toward refusal | The model emits `NotProposable` by construction. The bad action never exists, not even transiently |
| D. Verified Mediator | Typed `Payment{OpenAI, 1500, EUR}` (the model may still propose it) | `canAuthorize` checks currency against the mandate and the existence of a (counterparty, currency) rail; no such rail exists, so authorization is false | Provably-correct refusal: the mediator returns refuse and a machine-checked witness that no mandate-compliant payment to OpenAI exists |

*All four except A avoid the bad payment. Only D produces evidence that the refusal was the correct decision, not merely an absence of action.*

### 10.2 Reading the four rows

In Architecture A, the model produces a plausible payment to OpenAI in EUR. Nothing in the validation path knows that OpenAI has no EUR account, that fact lives in the workspace's rail configuration, not in

the prompt or the regex. The proposal passes the structural check and is dispatched. The failure happens at the rail adapter, far downstream, with an error message that describes the symptom (no rail-account combination) rather than the cause (the model proposed an unexecutable action). An operator debugging this sees a payment-network error, not an agent-reasoning error.

In Architecture B, the typed proposal passes schema validation because every field is individually valid: OpenAI is a real counterparty, EUR is a real currency, 1500 is a positive integer. The catch happens one layer below the schema, in the ontology resolver, which looks up whether a (OpenAI, EUR) rail exists and finds none. This is a cross-field constraint the schema could not express, and it is exactly where the ontology layer earns its place. The result is a clean rejection with a reason, or, if the model is well-prompted, a NotProposable response the model produced itself.

In Architecture C, the leverage comes from projecting a per-request grammar. When the system assembles the request, it knows the mandate is EUR-only and it knows OpenAI's available currencies. It can build a grammar in which the only legal completions for a payment to OpenAI are USD (which the mandate bars, so it too is excluded) or a NotProposable response. The combination (OpenAI, EUR) is not in the grammar; the decoder cannot emit it. The bad action never exists, not even as a transient string that gets rejected. The model is structurally channeled toward the correct refusal.

In Architecture D, the model is still free to propose the EUR payment to OpenAI, the mediator does not constrain what the model says. But the mediator's authorization predicate checks both that the currency matches the mandate and that a rail exists for the (counterparty, currency) pair. Neither the existence of a EUR-OpenAI rail nor any other escape is available, so the predicate is false and the mediator refuses. Crucially, the refusal comes with a proof witness: the system can demonstrate, with a machine-checked argument, that no mandate-compliant payment to OpenAI was possible. This is the difference that matters to an auditor.

#### Why D wins this example

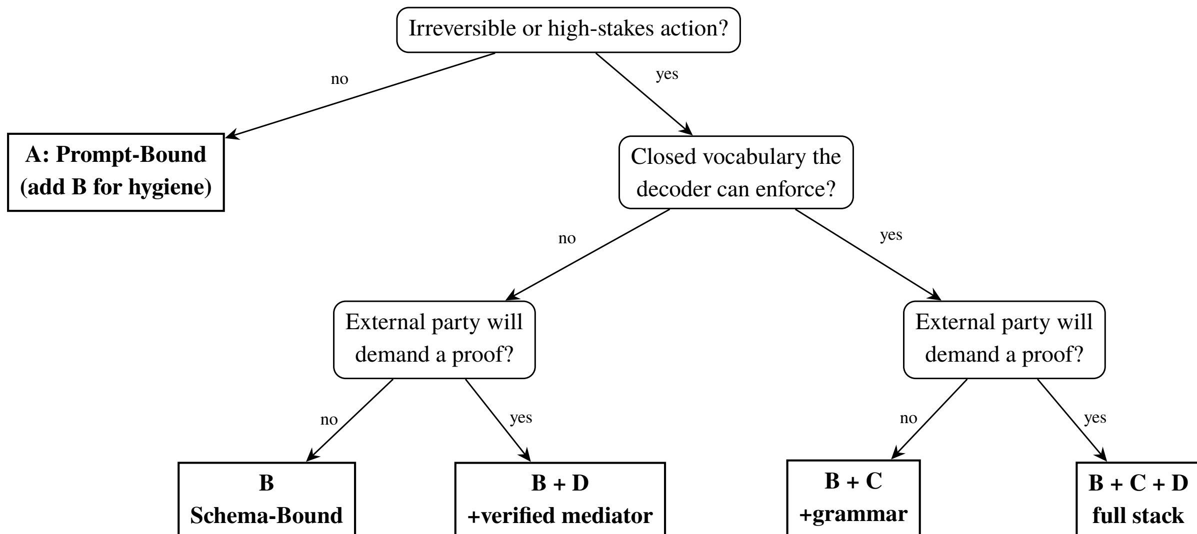
Architectures B, C, and D all avoid the bad payment; only A fails outright. So why prefer D? Because the question an auditor asks after the fact is not "did the system avoid the bad payment?" but "can you prove the system would always avoid it, for every input, including adversarial ones?" B and C avoid this particular bad payment through mechanisms (a cross-field check, a projected grammar) that are correct but not accompanied by a proof of completeness: you would have to trust that the check covers every case. D's refusal carries a machine-checked witness that the authorization predicate is false, and the MEDIATOR-SOUND theorem guarantees that no approved action ever violates the mandate, for all proposals. When the stakes are irreversible and the audience is adversarial, "we have a proof" beats "we have a check that handled this case."

## 11 Which architecture should you pick?

The four architectures are not a quality ranking; they are a cost-assurance trade-off. The right choice depends on the stakes of the action, the shape of the vocabulary, and whether an external party will demand proof. The decision tree below resolves most cases.

**Choosing an architecture**

*stakes, vocabulary, and proof obligation determine the floor*




*Every high-stakes path keeps Architecture B as its base. C is added when the vocabulary is closed enough to channel; D is added when an external party will demand proof.*

### 11.1 The questions, in order

The first question is whether the action is irreversible or high-stakes. If it is not, payments that a human reviews before they take effect, drafts, classifications, then Architecture A is acceptable, and adding the structural hygiene of Architecture B is a low-cost improvement rather than a necessity. The remaining questions only matter for high-stakes actions.

The second question is whether the vocabulary is closed enough for a decoder to enforce. If the set of legal values is finite and known (a fixed set of counterparties, a sealed enum of purposes), grammar-constrained decoding (C) can channel the model structurally, and it is worth the integration cost. If the vocabulary is open (free-text entity names that must be resolved at runtime), C buys less, and you rely on the ontology layer instead.

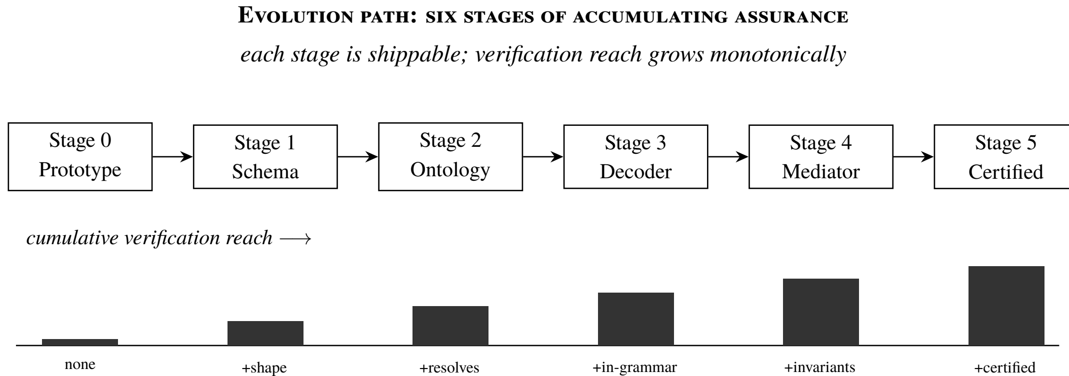
The third question is whether an external party, a regulator, an auditor, a court, will eventually demand a proof. If yes, the verified mediator (D) is the responsible floor, because only D produces the machine-checked evidence those parties expect. If no, a well-built Schema-Bound system (optionally with grammar constraints) is sufficient, and the cost of D is hard to justify.

### 11.2 The honest summary

For most production agents, Architecture B is the right answer, and it is where the majority of well-engineered systems should and do land. Architecture C is added when the vocabulary is closed and the integration cost is justified by the value of structural channeling. Architecture D is reserved for the genuinely irreversible, genuinely audit-bound systems where an adversarial reviewer will demand proof. Architecture A is a legitimate prototype and a legitimate choice for low-stakes outputs, but it is not a foundation for anything that moves money, changes infrastructure, or issues medical or legal instructions.

## 12 The evolution path

Few systems begin at Architecture D. The realistic trajectory is to start simple and deepen the stack as the stakes rise and the system earns the investment. The six stages below describe a path that adds one verifiable property at a time, with each stage a shippable system in its own right.




*Each stage strictly adds to the verification reach of the previous one. No stage retracts a guarantee an earlier stage established.*

### 12.1 Stage 0: prototype

Architecture A. A system prompt, an LLM call, a regex or JSON parser. The goal is to learn what the system needs to do. Verification reach: essentially none. This stage is appropriate for days-to-weeks of exploration and should not survive contact with high-stakes traffic.

### 12.2 Stage 1: schema

Introduce a typed schema (Zod, Pydantic, JSON Schema) at the boundary and enforce it with the provider's strict JSON mode. This is the jump to Architecture B. Verification reach: shape, types, enumeration closure. This is the single highest-leverage step in the entire path, and for many systems it is the last step they need.

### 12.3 Stage 2: ontology

Add an ontology layer below the parser: resolve entity references against live data, and check the cross-field constraints the schema could not express (does this counterparty have an account in this currency?). Verification reach: references resolve, cross-field constraints hold. This is where semantic mismatches, the (OpenAI, EUR) trap, get caught.

### 12.4 Stage 3: decoder

For the fields whose vocabulary is closed, add grammar-constrained decoding (Architecture C). The model can no longer emit out-of-grammar values for those fields, even transiently. Verification reach: in-grammar output by construction, for the constrained fields. Adopt this when the cost of even transient invalid proposals (latency, retries, audit noise) justifies the integration.

### 12.5 Stage 4: mediator

Introduce a verified mediator (Architecture D) for the final authorization step. Specify it in a proof assistant (B, Event-B, Lean, Dafny, or F*), prove the system safety invariants, and place the verified

component on the path to execution. Verification reach: no execution violates the system invariants, for all proposals including adversarial ones. This is the jump from engineering-with-tests to engineering-with-proofs.

### 12.6 Stage 5: certified extraction

Replace the hand-written mediator with code extracted (or compiled) from the proof artifact itself, so the running code is provably equivalent to the specification. In the B method, this is the refinement-to-B0-to-Ada/C pipeline; in Lean or F*, it is native extraction. Verification reach: the deployed code, not just the specification, carries the proof. This is the stage a transportation or medical safety regulator expects.

> **The trigger for stages 4 and 5**
>
> Stages 0 through 3 are driven by ordinary engineering judgment: each one pays for itself in reliability and debuggability, and most teams can justify them on those grounds alone. Stages 4 and 5 are different. They are expensive, they require formal-methods capability, and they pay off in a currency, machine-checked proof, that only certain audiences value. The trigger to climb to stage 4 or 5 is not internal: it is the appearance of an external party who will demand evidence. A regulator entering your domain, an auditor scoping a review, a court apportioning liability after an incident, a customer whose own compliance requires proof from their vendors. Until that party is on the horizon, stage 3 is often the responsible place to stop. Once they are, stages 4 and 5 stop being gold-plating and become the cost of doing business.

## 13 Closing

The space of ontology control layers over language models is organized by a single load-bearing idea: you do not verify the model, you verify the system around it. The model is an untrusted oracle that proposes structured terms; the architecture's job is to decide how much of the surrounding system is amenable to proof, and at what cost.

The four reference architectures are four answers to that question. Prompt-Bound verifies almost nothing and is honest only as a prototype or a low-stakes tool. Schema-Bound verifies structure and is the right default for the great majority of production agents. Grammar-Constrained verifies in-grammar output by construction and earns its place where the vocabulary is closed enough to channel. The Verified Mediator verifies system safety invariants with machine-checked proofs and is the responsible choice when actions are irreversible and an external party will demand evidence.

The five rings of verifiability draw the honest line. Structure, totality, and system safety invariants are inside the line: they are engineering, achievable today, with tools and track records that exist. Whether the model understood the user, and whether the whole system does the globally right thing, are outside the line: they are research, and claiming to have closed them is the most common form of overclaim. The discipline the rings impose is to stop trying to verify the model and instead design systems in which the model's correctness is never load-bearing.

The B method deserves its prominence in this account not because it is the most expressive formal method, it is not, but because it is the one with the deepest record of verifying deployed, safety-critical

infrastructure, and the one a transportation, medical, or banking regulator is most likely to have seen before. For a verified mediator in a regulated industry, B (or its modern successor Event-B) is the responsible default, with Dafny and F* as engineering-driven alternatives.

The through-line of the entire document is that programmable systems acting on the world must be provable where it counts. Not everywhere, the rings show that everywhere is not available, but at the layer where irreversible actions are authorized. The architectures, the formal methods, and the evolution path are all in service of placing a proof at exactly that layer, and being honest about everything around it that remains, for now, beyond proof.