Epi: An Epistemic Type System for Containing LLM Hallucination in Generated Code

Randerson Oliveira Melville Rebouças
Postgraduate Program in Informatics in Education (PPGIE), Universidade Federal do Rio Grande do Sul (UFRGS)
Porto Alegre, Brazil
randerson.melville@ufrgs.br

Dante Augusto Couto Barone
Postgraduate Program in Informatics in Education (PPGIE), Universidade Federal do Rio Grande do Sul (UFRGS)
Porto Alegre, Brazil
barone@inf.ufrgs.br

Eliseo Reategui
Postgraduate Program in Informatics in Education (PPGIE), Universidade Federal do Rio Grande do Sul (UFRGS)
Porto Alegre, Brazil
eliseo.reategui@ufrgs.br

Abstract

We introduce **Epi**, a domain-specific language whose type system makes the boundary between deterministic computation and LLM-inferred values explicit and enforceable at compile time. From a single `.epi` source, the Epi transpiler emits a complete Next.js project: database schema, API routes, authorization middleware, runtime validators, LLM call wrappers, and human-review checkpoints. Every AI-inferred field carries a type-level contract (an enum, a numeric range, a confidence threshold, an optional Bayesian prior) that the transpiler discharges in the generated code: the developer does not write the validation, the type does. We argue that LLM hallucination cannot be prevented, but that it can be *contained*, by construction, at a type-system boundary in generated code. We exercise Epi end-to-end on one source compiled against Anthropic Claude and Qwen 2.5 3B (local, via Ollama), where the boundary intercepts a low-confidence misclassification before it reaches a database of record, and probe it on 40 curated inputs: no value outside a declared enum ever reached the persistence path, while containment of merely *wrong* values is bounded by the model's confidence calibration. Unlike typed-LLM interfaces such as BAML and DSPy, which give structure to individual model calls, Epi propagates one epistemic contract across the whole generated application, from the database column to the human-review checkpoint. A mechanized soundness proof and a controlled comparison against hand-written baselines remain future work. Epi is open-source under the Apache 2.0 license.

Keywords

Type Systems, Domain-Specific Languages, Large Language Models, Code Generation, Information-Flow Control

1 Introduction

Large Language Models (LLMs) are increasingly embedded in production software, performing classification, extraction, summarization, and generation alongside conventional business logic. Yet the languages we build that software with were designed for deterministic computation: a function, given the same input, returns the same output. That assumption fails the moment an LLM enters the call graph, because the model may confidently return content that is fluent but unfaithful to its input, a failure mode surveyed as *hallucination* [6].

Throughout this paper we call a value *epistemic* when it is produced by LLM inference and therefore carries uncertainty, in contrast to a *rigid* value produced by ordinary deterministic code. We identify three concrete problems that arise when the two kinds of value are built in a conventional language whose type system cannot tell them apart. We phrase them against one common target, a TypeScript web application with a Prisma-managed database, though they are properties of the host language's type discipline rather than of any stack.

P1: No type-level separation between rigid and epistemic values. Consider a clinical-triage record with two fields: `status`, written by application code, and `risk`, produced by an LLM classifier. Declared as `status: String` and `risk: String`, the two are indistinguishable to the type checker. If the model returns `"Unknown"` instead of an expected label (`"High"`, `"Medium"`, `"Low"`), the value reaches the database with no type-level alarm, because nothing in the type of `risk` records that it came from inference and must be validated.

P2: Forced stack fragmentation. One AI-augmented feature forces the developer to hold several mental models at once: in the stack above, Prisma for the schema, TypeScript for the API, React for the UI, and a vendor SDK for the LLM call. The combination is one among many, but the difficulty is general: a single conceptual decision is spread across several grammars and toolchains, with no one place that states the epistemic contract.

P3: No grammar construct that requires a hallucination contract. Output validation, retry strategies, confidence thresholding, fallback dispatch, and audit trails are the safeguards a developer would add around an LLM call. The language-design problem is that no construct in a general-purpose host grammar *requires* any of them: each is an ordinary library call the developer may write, forget, or remove, and the type checker never objects. We want a grammar in which these safeguards are obligations attached to the epistemic type rather than optional additions.

This paper introduces **Epi**, the Epistemic Programming Interface, a domain-specific language and transpiler addressing these problems together. Its type system separates the two domains at the type level, and not merely descriptively: whenever an epistemic value crosses into rigid application state, the generated code requires runtime validation by construction. Epi also defines a compact language built from five declaration forms: `Entity` for typed schemas, `Guard` for authorization predicates, `Pulse` for AI execution units, `Pipeline` for composed flows, and `Lens` for user-interface surfaces. We do not claim this set is minimal or complete; it is the set the current transpiler lowers, and four of the five are exercised in Section 5. The language is specified by an EBNF grammar processed by the Lark toolkit [14] and presented in Section 3, from which the

transpiler generates a three-layer architecture where the LLM is excluded from the deterministic layers and may act only within constraints they emit.

We demonstrate the approach on an end-to-end worked example in which one `.epi` program runs against both a frontier cloud LLM and a small local open-weight model, intercepting a low-confidence misclassification before it reaches persistent state, and then on a curated set of 40 inputs that separates what the boundary guarantees structurally from what it merely mitigates. Neither is a controlled study against a hand-written baseline, which remains future work.

The central claim of this paper is intentionally limited in scope: **LLM hallucination cannot be prevented, but it can be contained by construction at a type-system boundary in generated code.**

2 Background and Related Work

Epi sits at the intersection of information-flow control, probabilistic programming, and typed tools for LLM-augmented development. Information-flow control (IFC) tracks how values in distinct domains, classically *public* and *secret*, propagate through a program, rejecting those that leak one into the other. Sabelfeld and Myers [13] survey its language-based form, where the type system enforces *non-interference*: observable outputs must not depend on secret inputs. Epi's Rigid/Epistemic distinction is structurally analogous, with *epistemic provenance* taking the place of confidentiality; Russo and Sabelfeld [12] likewise contrast dynamic and static flow-sensitive analyses encoding non-interference between domains. The transpiler enforces the property by emitting validation at every crossing.

Probabilistic languages such as ProbZelus [2] and SlicStan [5] treat distributions as first-class and separate deterministic computation from inference. Epi shares the concern that uncertainty belong in the language rather than hide inside ordinary values, but orchestrates calls to pre-trained endpoints instead of specifying models or compiling inference procedures.

Closest are typed interfaces for LLMs: BAML [4] provides typed signatures for LLM functions, and DSPy [7] compiles declarative signatures into optimized prompting strategies. Both structure interactions with language models at the level of individual calls or prompt modules. Epi extends the idea to whole-application generation: one specification yields database structures, middleware, validators, and interface scaffolding. It thus also connects to full-stack DSLs such as Wasp [15], which targets React and Node from a high-level specification but encodes no type-level distinction for AI-inferred values. What separates Epi from all three is the epistemic discipline itself: audit-by-construction is a property of the type, not a feature the developer can forget.

3 The Epistemic Type System

Rigid types are deterministic: the eight base types listed as `rigid_type` below. **Epistemic types** are AI-inferred and carry mandatory validation parameters, each parameterized by its own constraints (allowed values, max tokens, dimensions) plus optional attributes such as `strict`, `prior`, and `confidence_threshold`. Writing `ID` for identifiers and eliding the lexical productions, the core of the EBNF is:

```
decl ::= entity_decl | guard_decl | pulse_decl
  | pipeline_decl | lens_decl
entity_decl ::= "Entity" ID "{" field+ "}"
field ::= ID ":" type
type ::= rigid_type | epistemic_type
rigid_type ::= "UUID" | "Text" | "Int" | "Float"
  | "Decimal" | "Bool" | "DateTime" | "JSON"
epistemic_type ::= ai_kind "(" args ")"
ai_kind ::= "AI.Enum" | "AI.Text" | "AI.Score"
  | "AI.Classification" | "AI.Embedding"
enum_args ::= id_list ("," enum_attr)*
enum_attr ::= "strict" ":" bool
  | "prior" ":" distribution
  | "confidence_threshold" ":" number
guard_decl ::= "Guard" ID "{" cond "}"
pulse_decl ::= "Pulse" ID pulse_body
pulse_body ::= "{" input protect?
  (process | trace_decl+) output "}"
trace_decl ::= "Trace" ID trace_body
trace_body ::= "{" execute expose?
  checkpoint? "}"
pipeline_decl ::= "Pipeline" ID
  "{" flow on_error? "}"
lens_decl ::= "Lens" ID "{" lens_body "}"
```

The epistemic domain is therefore a syntactic category, not a convention: no production allows an `AI.*` constructor where a `rigid_type` is expected, or the reverse, so the distinction survives parsing and reaches the generator as a node tag rather than a naming discipline.

A field declared with an epistemic type imposes obligations on the generated code:

```
risk: AI.Enum(High, Medium, Low, strict: true,
              confidence_threshold: 0.85)
```

Here `High`, `Medium`, and `Low` are the only admissible labels; `strict` forbids anything outside that set; `confidence_threshold` is the minimum self-reported confidence below which the value is routed to human review instead of persisted. `AI.Score(min: 0, max: 5)` applies the same discipline to numeric inference, and the transpiler emits the matching bound check. In both cases the declaration is a *contract*, not a hint.

The single `risk` declaration requires the transpiler to emit four artifacts: (a) a **database column** of compatible storage type; (b) a **runtime validator**, emitted as a Zod [8] schema that parses the incoming value and raises an error when it falls outside the declared enum; (c) an **LLM inference call** wrapped to return JSON with the value and a `_confidence` field in $[0, 1]$; and (d) a **confidence-aware dispatch** that routes to a Checkpoint endpoint for human review when the reported confidence is below the threshold, and otherwise validates and persists.

The confidence in (c) is *elicited*: the transpiler appends a system-prompt instruction requiring the model to report it, rather than deriving it from token log-probabilities, which providers do not expose uniformly. When the model omits it, the generated code substitutes 0.5, below every threshold declared here, so an absent confidence fails towards review.

None of these artifacts is optional: the type itself forces their emission, so the *type carries an obligation that the transpiler discharges in the generated code*, and in Pierce's terms [11] the transpiler refuses

to emit programs that persist unvalidated AI output. The invariant, informal at this stage for the reasons given in Section 6, is that for every well-typed Epi program, every path from an `AI.*`-typed expression to a persistence sink passes through a validator on that type.

Crucially, the typed-signature DSLs discussed in Section 2 generate only artifact (c) above; Epi additionally generates (a), (b), and (d) as structural consequences of the same declaration.

These obligations are stated for `AI.Enum`, the strongest epistemic type; the others follow the same pattern with weaker guarantees. `AI.Classification` constrains a declared label set but admits more than one label per inference, `AI.Embedding` fixes the emitted vector's dimension, and `AI.Text` bounds only length and format, since free-form generation admits no enumeration of admissible values. Containment therefore degrades with the type: what the boundary can reject is exactly what the type constrains, so an `AI.Text` field still obtains (a), (c), and (d), including the review path, but its validator has no membership test to perform. This is a limit of the discipline, not of the implementation, and it is why the evaluation below uses an enumerated field.

3.1 The Five Primitives

Of the five declaration forms, only **Pulse** and the epistemic fields within **Entity** invoke the model; Guard, Pipeline, and Lens are deterministic. That asymmetry is what keeps conventional computation and probabilistic inference apart in the language itself.

A Pulse may also be decomposed into a sequence of *Trace* steps, each able to `Expose` intermediate reasoning fields and to pause at a `Checkpoint` for human review before the next step proceeds. Take a clinical-triage Pulse `TriageNote` that declares `Protect: Guard.CliniciansOnly`, reads a `Record`, writes `Record.risk`, and holds one Trace:

```
Trace Interpret {
  Execute: AI.reason(source: Input.note,
    prompt: file("interpret.md"))
  Expose: findings, severity
  Checkpoint: ReviewRequired(
    role: "Clinician")
}
```

Informally: `Protect` discharges the named Guard before any inference runs, so an unauthenticated request never reaches the model. The `Interpret` Trace issues one AI call and binds the fields listed in `Expose` from the model's JSON output. Because it declares a `Checkpoint`, execution suspends and those fields surface through a generated inspect/resume endpoint, resuming only after a user in the `"Clinician"` role approves or corrects them. Each paused step carries the exposed fields, the raw output, the reported confidence, the threshold that caused the pause, the reviewer's identity, and any correction injected downstream, so the record an audit needs is produced by the generated code rather than added to it. The emitted store holds it behind a save/get/update interface backed by an in-process map; binding that to a durable backend is a deployment concern the interface isolates, and the current release leaves it undone. The value written to `Record.risk` is thus the `AI.Enum` declared earlier, now validated and, when uncertain, reviewed. The mechanism targets high-stakes workflows, where auditing the final output alone is not enough.

An `AI.Enum` field may also declare a prior over its labels, in which case the transpiler emits a Bayesian-update function combining that prior with the model's reported likelihoods, so the posterior couples model output with domain base rates rather than overwriting them. Since both attributes may appear on one field, the composition order must be stated: today they do not compose. The dispatch of artifact (d) compares the raw reported confidence against `confidence_threshold`, while the Bayesian function sits beside the validator and exposes a posterior of its own. Comparing the threshold against that posterior is the natural composition and is left to future work.

We now describe how a Lark/EBNF grammar and these typing constraints become a runnable Next.js project.

4 Transpiler Architecture

The transpiler has three layers. The first parses the `.epi` source with the Lark grammar into an AST where every type node carries a domain tag, *rigid* or *epistemic*, assigned by the production that built it rather than inferred later, since the grammar of Section 3 already separates the categories. An `Entity` node thus exposes two derived views, one per domain, and that tag is the pivot of the design, because generator dispatch is a case split on it. The *Rigid Generator* consumes the rigid view and emits the Prisma schema, the route handlers, the authorization middleware derived from `Guard` conditions, and the Zod validators. The *Epistemic Generator* consumes the epistemic view and emits the call wrapper, the confidence extraction and threshold comparison, the Bayesian update when a prior is declared, and, for a Pulse decomposed into Traces, a trace store with `inspect` and `resume` endpoints per Trace.

Splitting the generators by epistemic domain rather than by output artifact is what carries the invariant. A full-stack generator is conventionally organized as one module per target file; under that split, the validator of an AI-inferred field is emitted by the schema module under a conditional and the dispatch by the route module under another, so nothing structural stops a later code path from lowering an epistemic field while skipping one of them. Making the domain the dispatch key removes that possibility: an epistemic node has exactly one lowering, and it emits the validator and the dispatch together with the call. The four artifacts of Section 3 thus come from one traversal rather than from the developer, and one `.epi` source replaces the several grammars identified in P2.

The LLM is formally excluded from Layers 1 and 2: the deterministic layers encode the constraints under which the application operates, and the epistemic layer may act only within them. These are not runtime checks added after the fact, but invariants fixed at generation time that the emitted runtime cannot violate without failing closed.

Layer 3 also emits a provider-agnostic client behind a uniform `llmCall` interface; environment variables select between Anthropic Claude [1] and any OpenAI-compatible endpoint, with Ollama [10] for local open-weight models. The contract is identical across providers: output must be valid JSON, carry a `_confidence` field, and parse against the Zod schema, so provider choice is a deployment concern rather than a code change.

5 Demonstration

We exercise the system end-to-end on a single educational scenario, as a worked example rather than a controlled study. The complete `.epi` source appears below with identifiers in English; the distributed program is identical but for Portuguese identifiers and prompt.

```
Entity Submission {
  id: UUID(auto), prompt: Text, answer: Text,
  evaluation: AI.Enum(Correct, Partial,
    Incorrect, strict: true,
    prior: Distribution(Correct: 0.40,
      Partial: 0.45, Incorrect: 0.15),
    confidence_threshold: 0.85)
}
Guard TeachersOnly {
  Condition: Auth.Role == "Teacher"
}
Pulse EvaluateAnswer {
  Input: Submission
  Protect: Guard.TeachersOnly
  Process:
    Execute: AI.classify(source: Input.answer,
      prompt: file("evaluate.md"),
      temperature: 0.1,
      on_low_confidence:
        Checkpoint.ReviewRequired(
          role: "Teacher"),
      on_fail:
        Fallback.ManualReview(Queue: "Teachers"))
  Output: Submission.evaluation
}
Pipeline AssessmentFlow {
  Flow: EvaluateAnswer
  On_Error: Retry(max: 2, backoff: exponential)
}
```

The epistemic field `evaluation` declares a prior and a confidence threshold; the Pulse routes low-confidence outputs to a teacher checkpoint and malformed ones to a manual-review queue. The transpiler emits 13 files totalling 477 lines, 287 of them TypeScript and Prisma schema.

We compile the same source twice and run both deployments unchanged, against **Anthropic Claude Sonnet 4** in the cloud and against **Qwen 2.5 3B Instruct** (Q4_K_M) served locally by Ollama on an Apple Silicon laptop with 16 GB RAM. The only difference is `EPI_AI_PROVIDER` in the deployment's `.env`. Both receive the same inputs against the prompt "What is photosynthesis?".

One input carries the demonstration. Given a textbook-correct answer, the small model returned `"Partial"` with a reported confidence of 0.8, a false negative that, once persisted, would have penalized the student unfairly. Because 0.8 falls below the `confidence_threshold: 0.85` declared on the field, the dispatch emitted in Layer 3 routed the case to a teacher checkpoint before it reached the database. Claude returned `"Correct"` at 0.93 on the same input and passed validation directly, so the two providers differ in accuracy and in latency (2.6 s against 1.1 s) while behaving identically at the boundary. A control input injecting malformed JSON exercises the other path: the generated Zod schema rejects it and `Fallback.ManualReview` queues the request. No line of this behaviour was written by hand; all of it follows from the type declaration, which is the containment claim in miniature. Hallucination is not prevented at the source, but it is prevented from silently crossing into persistent state.

Table 1: Boundary outcomes over 40 curated inputs, Qwen 2.5 3B, under two declarations of the AI call's source. The two `.epi` programs differ only in that token; enum, prior, threshold, Checkpoint and Fallback are identical.

	answer only	answer + question
Persisted without review	20	16
Routed to human checkpoint	20	24
Outside the declared enum, persisted	0	0
Wrong values persisted (30 gradable)	6	0
Wrong, had the output been unguarded	12	6

5.1 Containment on a Curated Input Set

A single scenario shows that the boundary can fire, not how often it fires when it should. We therefore exercised the same generated project on a curated set of 40 inputs, each with a gold label and a written justification. The set is unbalanced towards cases that test the boundary rather than confirm it: answers right but incomplete, answers fluent yet wrong on a single word, inputs with no defensible single label, and adversarial inputs attempting to force a label outside the declared enum. The items are in Portuguese, like the distributed program.

The harness does not reimplement the boundary: it imports the transpiler-emitted Pulse and records what that function returned, so threshold comparison, schema validation and dispatch to Checkpoint or Fallback all execute in generated code. Inference ran against Qwen 2.5 3B locally, three times per item at temperature 0.1, the outcome taken as the majority; 2 of the 40 items changed between repeats. Median latency was 621 ms (Table 1).

Two results follow, of different weight. The first is structural and never consults the gold labels: across every run, no value outside the declared enum reached the persistence path. This was not for lack of attempt. Against a deliberately weaker model, Llama 3.2 1B, one adversarial item drove the model to emit the invented label `CORRETÍSSIMO` on all three repeats, which the generated Zod validator rejected and routed to the manual-review queue; a second drove it to unparseable output, caught by the parse path of the same Fallback. Neither path was written by hand, and neither depends on the inference engine behaving well.

The second result is weaker and depends on the gold labels. Of the 30 items carrying a single defensible label, the model's output would have written a wrong value 12 times had it gone straight to the database; the boundary kept 6 of those out, at the cost of sending half of all items to review. Containment is therefore partial, bounded by something the type system does not supply: the self-reported confidence must actually track the errors. The 1B model illustrates the bound by violating it, reporting high confidence almost everywhere and so escaping with all 18 of its wrong values while sending only 22% of items to review.

The two columns isolate a language-level effect. They differ in one token of the source: whether the AI call declares `source: Input.answer` or `source: Input`, that is, whether the model sees the answer alone or the answer with the question it responds to. The epistemic contract is unchanged, yet under-specifying the source doubles the raw error rate and defeats the boundary on the errors that remain, because a verdict formed without the necessary information is still reported confidently. Source adequacy therefore belongs in the contract next to the threshold, a concrete instance of the calibration question of Section 6.

Three limitations bound these numbers. The items are synthetic, authored for this evaluation rather than collected, so they probe the boundary rather than estimate a field rate. The gold labels come from a single annotator, with no second pass and therefore no inter-rater figure; the structural result is unaffected, since it never consults a label. And containment is not accuracy: three adversarial items produced a wrong but confidently reported in-domain label, which the boundary passed. A type system can require that an uncertain value be reviewed and guarantee that an out-of-domain value is never written, but it cannot detect a confident, well-formed lie.

6 Discussion and Limitations

Epi's type system is specified in an implementation-oriented rather than mechanized form: the paper presents a working transpiler and an empirical demonstration, but no mechanized soundness proof. The invariant of Section 3, that every path from an `AI.*`-typed expression to a persistence sink passes through a validator for that type, can be read as non-interference between rigid and epistemic value spaces. A full treatment in a proof assistant such as Coq [3] or Agda [9] would require typing judgments, an operational semantics, and a soundness theorem, and is left as future work.

A second open question is how the `confidence_threshold` is chosen. It is a developer-supplied constant, which places the calibration burden on the developer and ignores that the compared quantity is a confidence the model reports about itself, elicited by prompt as described in Section 3 and not guaranteed to track its actual error rate. Section 5.1 shows the consequence directly: the same boundary that contained half of one model's errors contained none of a weaker model's, because that model reported high confidence almost everywhere. Three more principled routes are left to future work: deriving the threshold from a labelled validation set so it targets a chosen precision on reviewed cases; comparing it against the Bayesian posterior rather than the raw report, which also resolves the composition order left open in Section 3; and recalibrating reported confidences before comparison. In all three the language surface is unchanged, since the threshold is already a first-class attribute of the type; only the value bound to it would be computed rather than written by hand.

Two further limits belong to the prototype rather than the design: the FastAPI target is partially implemented and gated in the CLI, and the `Mood` declaration of Lens, whose tie to the epistemic discipline is weakest, is marked experimental.

7 Conclusion and Future Work

LLM hallucination is a property of the inference, not of the surrounding code, and cannot be prevented at the type-system level. It can be *contained*. A type discipline distinguishing deterministic from epistemic provenance can require, by construction in generated code, that every AI-inferred value pass a validator, report its confidence, and surface for human review when uncertain. We have implemented this discipline as a language and a transpiler, exercised it against a frontier cloud model and a small local open-weight one, and shown that the boundary catches a misclassification that would otherwise reach a database of record and holds identically across both engines. Where BAML and DSPy make a single model call well-typed, Epi takes the crossing from inference into persistent state as the thing to be typed, which is why one declaration reaches the schema, the validator, and the review path at once. Three directions follow: a mechanized formalization, a controlled comparison against hand-written baselines, and a `confidence_threshold` computed from a labelled validation set or the declared `prior` rather than written by hand.

Artifact Availability

Epi is open-source under the Apache 2.0 license. The full source, the example program of Section 5, and the generated TypeScript artifacts are at https://github.com/RandMelville/epi-lang (`pip install epi-lang`); the containment harness of Section 5.1, its 40-item input set with gold labels and justifications, and the raw per-call results are in `eval/`. An archival snapshot is deposited on Zenodo at https://doi.org/10.5281/zenodo.21433256.

Acknowledgements

The authors thank Prof. Rosa Maria Vicari (PPGIE/UFRGS) for the conceptual discussion that led to the Trace+Checkpoint design, and Prof. Fernando Magno Quintão Pereira (Compilers Lab, UFMG) for pointing us to the information-flow-control literature as a structural analog. We also thank the anonymous reviewers, whose comments on grammar coverage, on the epistemic types left unexplored, and on the scope of the evaluation shaped this version. Generative AI tools assisted in drafting and revising this manuscript and in implementing parts of the accompanying software; all technical claims, the measurements of Section 5.1, and the final text were verified by the authors, who take full responsibility for the content.

References

[1] Anthropic. 2025. Claude API Documentation. https://docs.anthropic.com. Accessed: May 2026.

[2] Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. 2020. Reactive Probabilistic Programming. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 898–912. doi:10.1145/3385412.3386009

[3] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer, Berlin, Heidelberg. doi:10.1007/978-3-662-07964-5

[4] BoundaryML. 2024. BAML: A Domain-Specific Language for AI Engineering. https://github.com/BoundaryML/baml. Accessed: May 2026.

[5] Maria I. Gorinova, Andrew D. Gordon, and Charles Sutton. 2019. Probabilistic Programming with Densities in SlicStan: Efficient, Flexible, and Deterministic. *Proceedings of the ACM on Programming Languages* 3, POPL, Article 35 (2019), 30 pages. doi:10.1145/3290348

[6] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of Hallucination

in Natural Language Generation. *Comput. Surveys* 55, 12, Article 248 (2023), 38 pages. doi:10.1145/3571730

[7] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2023. DSPy: Compiling Declarative Language Model Calls Into Self-Improving Pipelines. arXiv:2310.03714 [cs.CL] https://arxiv.org/abs/2310.03714

[8] Colin McDonnell. 2024. Zod: TypeScript-First Schema Validation with Static Type Inference. https://github.com/colinhacks/zod. Accessed: May 2026.

[9] Ulf Norell. 2009. Dependently Typed Programming in Agda. In *Advanced Functional Programming (AFP 2008)*. Lecture Notes in Computer Science, Vol. 5832. Springer, 230–266. doi:10.1007/978-3-642-04652-0_5

[10] Ollama Inc. 2024. Ollama: Run Open-Weight LLMs Locally. https://ollama.com. Accessed: May 2026.

[11] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA.

[12] Alejandro Russo and Andrei Sabelfeld. 2010. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium (CSF 2010)*. IEEE Computer Society, 186–199. doi:10.1109/CSF.2010.20

[13] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19. doi:10.1109/JSAC.2002.806121

[14] Erez Shinan. 2024. Lark: A Parsing Toolkit for Python. https://github.com/lark-parser/lark. Accessed: May 2026.

[15] Wasp. 2024. Wasp: A DSL for Full-Stack Web Apps. https://wasp-lang.dev. Accessed: May 2026.