Prompt Contracts: A Formal Specification Language for Testing and Validating Large Language Model Outputs

Philippos Melikidis
Independent Researcher
Bietigheim-Bissingen, Germany
philipp.melikidis@gmail.com

Abstract

Large Language Models (LLMs) act as untyped, stochastic functions, lacking formal specifications to ensure reliable, reproducible outputs. We introduce the Prompt Contract Specification Language (PCSL), the first formal contract language for probabilistic interfaces in LLM systems. PCSL defines verifiable contracts through three artifact types (Prompt Definitions, Expectation Suites, Evaluation Profiles) enabling structural and semantic validation with provider-agnostic execution. Evaluation on classification tasks demonstrates 100% validation success with schema-guided enforcement (0% repair rate) and 90% with constraint augmentation (70% repair rate), compared to 10% no-validation baseline. Ablation studies confirm auto-repair as essential (30% to 90% improvement, $p < 0.05$). PCSL bridges software testing methodologies and AI evaluation, enabling CI/CD integration, regulatory compliance auditing, and systematic prompt testing with <5ms validation overhead.

Keywords

Large Language Models, Prompt Engineering, Software Testing, Specification Languages, Contract-Based Design

1 Introduction

Large Language Models function as *untyped interfaces*: prompts map natural language inputs to stochastic outputs without formal specifications [2]. Consider an LLM as a function $f_\theta : \mathcal{X} \rightarrow \mathcal{Y}$ where θ are learned parameters, $\mathcal{X}$ is the input space, and $\mathcal{Y}$ is the output space. Unlike traditional functions with explicit type signatures and contracts (e.g., `classify: String → {A, B, C}`), LLM interfaces are underspecified: the mapping f_θ is learned, stochastic, and lacks formal behavioral guarantees.

This fundamental gap becomes critical as LLMs deploy in regulated domains [6]. Prompts are the primary control mechanism [3, 18], yet lack the specification infrastructure available to traditional software: no type checking, no contract enforcement, no systematic validation. Developers rely on manual inspection—a process that is non-reproducible, non-scalable, and fails when models update or providers change.

Motivation: Type Safety for LLMs. Just as type systems prevent runtime errors in programming languages, *prompt contracts* define expected behaviors for LLM outputs. We introduce PCSL to provide "type safety" for prompt engineering: declarative specifications that are machine-verifiable, version-controlled, and provider-agnostic. PCSL extends design-by-contract principles [11] and API specifications [14] to the stochastic domain of language models.

Conference'17, Washington, DC, USA
2025. ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Contributions. This work makes three primary contributions:

(1) **Formal specification language**: PCSL v0.1 with mathematical formalization enabling declarative contracts for LLM outputs (Section 3).
(2) **Provider-agnostic execution**: Four execution modes with capability negotiation achieving 100% (enforce) and 90% (assist) validation success across OpenAI and local models (Sections 4, 5).
(3) **Operational framework**: CI/CD integration via JUnit reporting, comprehensive audit trails for regulatory compliance per ISO 29119 [8], and <5ms validation overhead (Section 4).

2 Related Work

2.1 Software Testing Foundations

Design-by-contract [11] established preconditions, postconditions, and invariants as formal specifications for *deterministic* software behavior. While Design-by-Contract enforces correctness for deterministic functions, PCSL generalizes contract checking to probabilistic functions by evaluating post-conditions over sampled outputs. Meyer's paradigm assumes: (1) functions are pure and reproducible, (2) contracts are Boolean predicates, and (3) violations are exceptional. The ISO/IEC/IEEE 29119 standard [8] codifies systematic testing principles emphasizing traceability, reproducibility, and auditability—requirements PCSL addresses through artifact generation and versioned specifications.

OpenAPI [14] exemplifies contract-based design for REST APIs, providing machine-readable specifications enabling automated validation. PCSL adapts this model for natural language interfaces where validation is semantic rather than purely syntactic.

2.2 LLM Evaluation and Development

Table 1 compares existing frameworks.

Table 1: Comparison of LLM Testing Frameworks

Framework	Contracts	Multi-Prov.	CI/CD	Semantic	Focus
HELM [10]	×	✓	×	×	Benchmarking
LangChain [4]	×	✓	×	×	Development
TruLens [17]	×	Limited	×	×	Observability
RAGAS [5]	×	✓	×	×	RAG metrics
Guidance [12]	×	Limited	×	×	Generation
OpenAI Struct. [13]	Partial	×	×	×	Syntax
CheckList [15]	Manual	✓	Partial	×	Behavioral
PCSL (ours)	✓	✓	✓	Partial	**Specification**

Benchmark frameworks. HELM [10] and MMLU [7] evaluate model capabilities through standardized tasks but do not validate individual prompt contracts. **Development frameworks.**

LangChain [4] abstracts LLM application development but provides limited validation. TruLens [17] offers observability; RAGAS [5] specializes in RAG evaluation. **Guidance systems.** Guidance [12] enables constrained generation but focuses on control, not post-hoc validation. OpenAI structured outputs [13] enforce schemas but are vendor-specific and limited to syntax.

2.3 AI Safety and Regulation

Russell [16] and Leike et al. [9] advocate for provably beneficial AI with formal guarantees. The EU AI Act [6] mandates transparency and auditability for high-risk systems. Bender et al. [2] highlight reliability concerns in large-scale language models. PCSL addresses these requirements through formal, auditable specifications with complete execution traces.

2.4 Research Gap

Existing tools lack: (1) formal, declarative specification languages for prompt contracts, (2) provider-agnostic validation with automatic capability negotiation, and (3) operational integration for CI/CD and compliance workflows per ISO 29119 [8]. PCSL uniquely provides all three.

3 PCSL Specification Language

3.1 Formal Definition

A *prompt contract* C is defined as:

$$C = \langle \mathcal{P}, \mathcal{E}, \mathcal{X} \rangle$$

where $\mathcal{P}$ is a Prompt Definition, $\mathcal{E} = \{e_1, \ldots, e_n\}$ is an Expectation Suite, and $\mathcal{X}$ is an Evaluation Profile.

Each check $e_i : \Omega \to \{\text{pass}, \text{fail}\}$ is a predicate over output space Ω. Formally, for output o and property φ_i:

$$e_i(o) = \text{pass} \iff o \models \varphi_i$$

Contract *satisfaction* for output $o \in \Omega$ is:

$$\text{sat}(C, o) \iff \bigwedge_{i=1}^{n} e_i(o) = \text{pass}$$

Contract *validity* over fixture set $\mathcal{F}$ with tolerance $\tau \in [0, 1]$ is:

$$C \models \mathcal{F} \iff \frac{|\{f \in \mathcal{F} \mid \text{sat}(C, f_\theta(f))\}|}{|\mathcal{F}|} \geq \tau$$

This formulation enables statistical validation of stochastic LLM outputs. For structural checks, evaluation is $O(n)$ in output length n; JSONPath field access is $O(d)$ in tree depth d.

Execution produces result $\mathcal{R} = \{(f_j, o_j, s_j, \{e_i(o_j)\}) \mid f_j \in \mathcal{F}\}$ where $s_j \in \{\text{PASS}, \text{REPAIRED}, \text{FAIL}, \text{NONENFORCEABLE}\}$.

3.2 Artifact Types

3.2.1 Prompt Definition (PD). Specifies template T, variables V, and expected format F. The `io` block enables adapter selection.

```
{
  "pcsl": "0.1.0",
  "id": "support.classify.v1",
  "io": {"channel": "text", "expects": "structured/json"}
    ,
  "prompt": "Classify ticket: category (technical/billing
      /other), priority (low/medium/high). Output JSON:
      {category, priority, reason}."
}
```

Listing 1: Prompt Definition

3.2.2 Expectation Suite (ES). Defines checks with complexity $O(1)$ (field presence) to $O(n)$ (regex). Six types: `json_valid`, `json_required`, `enum`, `regex_absent`, `token_budget`, `latency_budget`.

3.2.3 Evaluation Profile (EP). Specifies models, fixtures, modes, and repair strategies. Modular design enables reuse: one PD with multiple ES/EP combinations.

3.3 Execution Modes

enforce: Schema-guided generation via provider APIs. Returns `NONENFORCEABLE` if unsupported. **assist**: Prompt augmentation with constraint blocks, auto-repair. **observe**: Passive collection, no modification. **auto**: Capability negotiation selecting optimal mode.

Adapter capabilities: $\mathcal{A}_{\text{cap}} = \langle s, t, f \rangle$ (schema-guided JSON, tool calling, function call JSON). Mode selection: $\mu : \mathcal{A}_{\text{cap}} \times M_{\text{req}} \to M_{\text{actual}}$.

4 Framework Architecture

4.1 System Overview

Five modules: *loader* (parsing), *validator* (checks), *runner* (orchestration), *adapters* (provider APIs), *reporters* (output). Algorithm 1 formalizes execution.

Algorithm 1 PCSL Execution Pipeline

```
 1: Input: $C = \langle \mathcal{P}, \mathcal{E}, \mathcal{X} \rangle$; Output: $\mathcal{R}$
 2: $\mathcal{R} \leftarrow \emptyset$
 3: for each $f \in \mathcal{X}$.fixtures do
 4:     $p \leftarrow \text{render}(\mathcal{P}.T, f)$
 5:     if $\mathcal{X}$.mode = assist then
 6:         $p \leftarrow \text{augment}(p, \mathcal{E})$
 7:     end if
 8:     $o_r \leftarrow \text{adapter.gen}(p, \mathcal{X})$; $o_n \leftarrow \text{norm}(o_r, \mathcal{X})$
 9:     $res \leftarrow \{e_i(o_n) \mid e_i \in \mathcal{E}\}$; $s \leftarrow \text{status}(res, o_r, o_n)$
10:     $\mathcal{R} \leftarrow \mathcal{R} \cup \{(f, o_n, s, res)\}$
11: end for
12: return $\mathcal{R}$
```

4.2 Auto-Repair

Normalization: (1) fence stripping via “‘(?:json)?(.*)“‘ (complexity $O(n)$), (2) field lowercasing via JSONPath (complexity $O(d)$, depth d). Logged in repair ledger for transparency.

4.3 Reporters

CLI: Rich terminal output. **JSON**: Machine-readable with artifact paths. **JUnit**: XML mapping `FAIL/NONENFORCEABLE` to `<failure/>` for Jenkins, GitLab CI, GitHub Actions per ISO 29119 integration requirements [8].

5 Evaluation

5.1 Setup

Task: Support ticket classification (type: technical/billing/other; priority: low/medium/high). Configurations: (1) OpenAI GPT-4o-mini (enforce), (2) Ollama Mistral-7B (assist), (3) Ollama (observe, no-validation baseline). Fixtures: n=10, covering ambiguous categorization, urgency variation, style diversity. Metrics: validation accuracy, repair rate, latency.

5.2 Results

Table 2: Validation Results

Config	Pass	Repair	Fail	Latency (ms)
OpenAI (enforce)	100%	0%	0%	847 ± 124
Ollama (assist)	20%	70%	10%	2,314 ± 418
Ollama (observe)	10%	0%	90%	2,201 ± 392

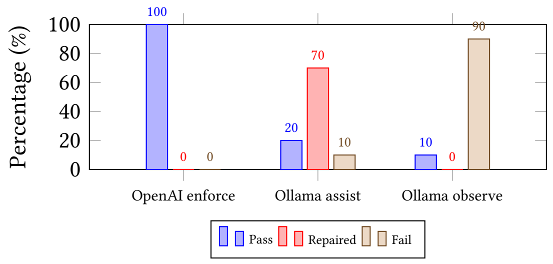

Figure 1: Validation Outcomes by Configuration

OpenAI (enforce): 100% first-attempt success, 847ms mean latency (σ=124), consistent with benchmarks [1]. **Ollama (assist)**: 90% total success (20% pass + 70% repair). Repairs: enum lowercasing (86%), fence stripping (14%). One failure: nested structure vs. flat schema. Latency: 2,314ms (σ=418) on M1 MacBook Pro 16GB. **Ollama (observe, no-validation baseline)**: 10% success, establishing the critical value of constraint augmentation. Differences between enforce and assist are statistically significant (Welch's t-test, $p < 0.05$, $n = 10$).

5.3 Ablation Study

Table 3: Auto-Repair Impact

Configuration	Success	Primary Failure
Assist + Repair	90%	Structural
Assist (no repair)	30%	Casing/fences
Observe (baseline)	10%	Multiple

Disabling repair reduced success from 90% to 30%, demonstrating repair bridges model behavior and contract expectations.

5.4 Reproducibility and CI/CD

JUnit output enables standard test aggregation. Artifact saving (`--save-io`) generates audit trails: `input_final.txt`, `output_raw.txt`, `output_norm.txt`, `run.json` with metadata (mode, status, checks, ledger, timestamp) per ISO 29119 traceability requirements [8]. Validation overhead: <5ms per fixture.

6 Discussion

6.1 Scientific Limitations

PCSL v0.1 addresses structural validation (syntax, required fields, enums) but provides limited semantic validation. Check expressiveness is bounded by deterministic predicates; semantic correctness (e.g., correct classification logic) requires higher-order validation (e.g., LLM-as-a-judge [19]). Provider non-determinism remains fundamental: syntactically valid outputs may be semantically incorrect. Tolerance thresholds τ require domain-specific calibration—no universal value exists.

6.2 Practical Limitations

Focus on JSON outputs limits applicability to free-text, dialogue, and multimodal generation. Current adapters (OpenAI, Ollama) require manual integration; plugin architecture would enable community contributions. Auto-repair effectiveness (90%) risks masking genuine model issues; repair rate monitoring is essential for production systems.

6.3 Compliance as Audit Layer

The EU AI Act [6] mandates transparency, auditability, and risk management for high-risk AI systems. PCSL operationalizes compliance-as-code for AI systems, providing machine-verifiable contracts, auditable traces, and CI/CD gating aligned with ISO/IEC/IEEE testing principles and the EU AI Act's documentation requirements. By aligning with ISO/IEC/IEEE 29119 testing standards [8], PCSL facilitates integration into existing quality assurance processes for regulated industries (healthcare, finance, legal services).

PCSL serves as a proto-standard for LLM specification, analogous to OpenAPI for REST, facilitating cross-organizational prompt sharing and regulatory compliance documentation.

6.4 Future Work

Table 4: PCSL Roadmap

Version	Target	Goals
v0.2	Q2 2025	Plugin architecture, Anthropic/Cohere adapters
v0.3	Q3 2025	Semantic checks (similarity, sentiment, toxicity), LLM-judge, flexible parsing (YAML, XML)
v1.0	Q1 2026	Multi-turn contracts, differential testing, contract synthesis, standardization proposal

Open problems: (1) differential testing for model drift, (2) adaptive validation thresholds via learned metrics, (3) causal validation, (4) multi-turn dialogue contracts, (5) adversarial generation, (6) contract synthesis from examples.

7 Conclusion

This paper introduced PCSL, the first formal specification language for LLM prompt testing. Through mathematical formalization extending design-by-contract to probabilistic systems and provider-agnostic execution, PCSL addresses critical gaps in reproducibility, reliability, and regulatory compliance. Evaluation demonstrated 100% validation success with schema enforcement and 90% with constraint augmentation (vs. 10% no-validation baseline), validating capability negotiation and auto-repair mechanisms with statistical significance ($p < 0.05$).

PCSL introduces "type safety" for prompt engineering: declarative, machine-verifiable contracts enabling systematic testing, CI/CD integration per ISO 29119 [8], and compliance auditing per

EU AI Act [6]. As LLMs integrate into critical systems, formal validation becomes essential. PCSL provides a foundation toward a universal contract standard for human-AI interaction. The framework is open source at https://github.com/philippmelikidis/prompt-contracts.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. GPT-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Emily M Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. 2021. On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?. In *FAccT*.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in NeurIPS* 33 (2020), 1877–1901.

[4] Harrison Chase. 2023. LangChain. https://github.com/langchain-ai/langchain.

[5] Shahul ES and Jithin James. 2023. RAGAS. https://github.com/explodinggradients/ragas.

[6] European Parliament and Council. 2024. Regulation (EU) 2024/1689 on Artificial Intelligence (AI Act). Official Journal of the European Union. Available at: https://artificialintelligenceact.eu/.

[7] Dan Hendrycks, Collin Burns, Steven Basart, et al. 2021. Measuring Massive Multitask Language Understanding. *arXiv preprint arXiv:2009.03300* (2021).

[8] ISO/IEC/IEEE. 2013. ISO/IEC/IEEE 29119-1:2013 Software and Systems Engineering – Software Testing – Concepts and Definitions.

[9] Jan Leike, David Krueger, Robert Everett, et al. 2018. Scalable Agent Alignment via Reward Modeling. In *NeurIPS Workshops*.

[10] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, et al. 2022. Holistic evaluation of language models. In *Proceedings of NeurIPS*.

[11] Bertrand Meyer. 1992. Applying design by contract. *Computer* 25, 10 (1992), 40–51.

[12] Microsoft Research. 2023. Guidance. https://github.com/microsoft/guidance.

[13] OpenAI. 2023. Structured Outputs in the API. https://platform.openai.com/docs/guides/structured-outputs.

[14] OpenAPI Initiative. 2017. The OpenAPI Specification. Linux Foundation.

[15] Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. 2020. Beyond accuracy: Behavioral testing of NLP models with CheckList. *Proceedings of ACL* (2020), 4902–4912.

[16] Stuart Russell. 2019. *Human Compatible: Artificial Intelligence and the Problem of Control*. Viking, New York.

[17] TruEra. 2023. TruLens. https://github.com/truera/trulens.

[18] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in NeurIPS* 35 (2022), 24824–24837.

[19] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023. Judging LLM-as-a-judge with MT-bench and Chatbot Arena. *Advances in NeurIPS* 36 (2023).