Contracts for Large Language Model APIs: A Comprehensive Taxonomy, Detection Framework, and Enforcement Strategies

Tanzim Hossain Romel, Kazi Wasif Amin, Istiak Bin Mahmod, Akond Rahman

October 14, 2025

DRAFT VERSION – NOT FOR OFFICIAL SUBMISSION

This document is a working draft and is not yet ready for publication or formal submission. The content is subject to revision, and all findings should be considered preliminary.

Abstract

Large Language Model (LLM) APIs have rapidly transformed software development, yet their integration introduces critical reliability challenges through implicit "contracts" – undocumented usage protocols that, when violated, cause failures ranging from runtime errors to silent logical bugs. We extend foundational work on machine learning API contracts [1] into the LLM domain with three key contributions: (1) a *formal probabilistic contract model* capturing preconditions, postconditions over output distributions, and state-transition rules with composition guarantees; (2) a *comprehensive taxonomy* grounded in 650 real-world violation instances mined from developer discussions across major providers (OpenAI, Anthropic, Google, Meta) and frameworks (LangChain, AutoGPT, CrewAI, LlamaIndex, Semantic Kernel) spanning 2020-2025; (3) *practical enforcement techniques* achieving Contract Satisfaction Rate (CSR) improvements of +18.7 percentage points (95% CI [16.2, 21.3]) and Silent Failure Rate (SFR) reductions of -12.4 pp (95% CI [-14.1, -10.6]), with median overhead of 27ms (P95: 89ms, $<$8% latency). Our taxonomy reveals 73+ distinct contract types across traditional constraints (input validation, sequencing) and LLM-specific categories including output format compliance, content policy enforcement, streaming response assembly, multimodal content handling, and inter-agent coordination—with production systems (2024-2025) validating compositional failure modes where individually valid features violate contracts when combined. We evaluate enforcement on 147 scenarios spanning static analysis, runtime guardrails, and framework integration, demonstrating 89% violation prevention across contract types. This work provides, to our knowledge, the first large-scale, cross-provider, longitudinal characterization of LLM API contracts with formal foundations and empirical validation spanning experimental prototypes to production autonomous agent systems, offering developers, API providers, and researchers a roadmap for building more reliable AI-augmented systems.

1 Introduction

The integration of Large Language Models (LLMs) through Application Programming Interfaces (APIs) has fundamentally transformed modern software development. Services from OpenAI [26], Anthropic [30], Google [31], and Meta [33] enable developers to embed sophisticated natural language capabilities into applications with minimal effort. However, this accessibility masks a critical

challenge: the reliable use of LLM APIs depends on adherence to numerous implicit "contracts" – undocumented or poorly communicated usage requirements that, when violated, lead to failures ranging from immediate exceptions to subtle behavioral anomalies.

1.1 The Contract Challenge in LLM Systems

The concept of software contracts, formalized through Design by Contract (DbC) methodology by Bertrand Meyer [2, 3], establishes that reliable software components must explicitly specify their preconditions, postconditions, and invariants. Recent work by Khairunnesa et al. [1] demonstrated that violations of such implicit contracts account for a significant portion of bugs in machine learning libraries like TensorFlow, Scikit-learn, and PyTorch. Their analysis of 413 informal specifications from Stack Overflow revealed that most ML API failures stem from violated assumptions about input types, shapes, or API call ordering.

LLM APIs inherit these traditional contract challenges while introducing entirely new categories of implicit requirements. Consider a developer building a conversational agent using OpenAI's GPT-4 API through LangChain [22]. The system may fail in multiple ways, as documented in numerous GitHub issues and Stack Overflow posts [74, 34]:

1. **Traditional Input Violations:** Forgetting to set the API key yields an authentication error – a classic precondition violation [81].

2. **Resource Constraints:** After lengthy conversation, the accumulated context exceeds the model's 8,192 token limit, causing truncation or rejection [35].

3. **Format Non-compliance:** The developer instructs the model to output JSON for database queries, but the model occasionally responds in natural language, causing parser failures with cryptic "Could not parse LLM output" errors [36].

4. **Content Policy Triggers:** A user's seemingly innocuous request activates safety filters, returning generic refusals that break the application flow [39].

5. **Tool Invocation Failures:** The agent attempts to call a non-existent tool that the model hallucinates, causing runtime exceptions [38].

Each failure represents a violated contract that could have been documented, checked, and handled gracefully. Yet these contracts remain largely implicit, forcing developers into frustrating cycles of trial-and-error debugging, as evidenced by the proliferation of related discussions in developer forums [76, 77].

1.2 Novel Contract Types in LLM Systems

LLM APIs introduce contract categories unprecedented in traditional software, as shown in Table 1:

Prompt and Output Format Contracts specify structural requirements for inputs and outputs. Unlike traditional APIs with fixed schemas, LLMs accept natural language with embedded formatting instructions, creating a dual contract: the API's technical requirements and the model's instruction-following capabilities [20].

Content Policy Contracts enforce ethical and safety boundaries through automated filtering. These contracts are particularly challenging because they're often opaque – developers receive generic error messages like "content was flagged as potentially violating our usage policy" without clear indication of which specific terms or concepts triggered the filter [28].

Table 1: Novel Contract Types Introduced by LLM APIs

Contract Type	Description	Example Violation
Prompt/Output Format	Structural requirements for inputs and outputs	Model returns prose instead of JSON [78]
Content Policy	Ethical and safety boundaries	Request flagged by content filter [39]
Multi-turn Interaction	Stateful conversation management	Context lost between calls [37]
Tool Calling	Function invocation protocols	Invalid tool schema format [41]
Token Economics	Usage cost constraints	Unexpected high token consumption

Multi-turn Interaction Contracts govern stateful conversations and agent workflows. These include requirements to maintain conversation history, properly sequence tool calls, and manage context windows across multiple interactions [21]. Violating these contracts often produces subtle failures where the system continues operating but with degraded performance or logical inconsistencies.

1.2.1 Toward a Formal Contract Model

To move beyond taxonomic enumeration toward a rigorous foundation for LLM API reliability, we introduce a formal model of *probabilistic contracts* that captures the unique characteristics of LLM systems while enabling compositional reasoning and quantitative evaluation.

Definition 1 (Probabilistic LLM Contract). An LLM API contract is a tuple $C = \langle \mathcal{I}, \text{Pre}, \text{Post}, \mathcal{S}, \pi \rangle$ where:

- $\mathcal{I}$: *Interface specification* defining parameters, types, and schemas (e.g., `messages: List[Message]`, `max_tokens: int`);
- Pre: *Preconditions* over the request, system state, and resource budgets, including:
  - *Type constraints*: $\forall p \in \text{params}, \text{typeof}(p) \in \mathcal{I}(p)$
  - *Value constraints*: $\text{token\_count}(\text{messages}) \leq \text{model.context\_limit}$
  - *Budget constraints*: $\text{rate} \leq \text{quota.requests\_per\_min}$
  - *Policy constraints*: $\text{moderation}(\text{messages}) = \text{safe}$
- Post: *Postconditions* over **distributions** of outputs, reflecting LLM non-determinism:
  - *Format compliance*: $\Pr[\text{output} \models \text{schema}] \geq \alpha$
  - *Safety guarantees*: $\Pr[\text{toxicity}(\text{output}) < \theta] \geq \beta$
  - *Completion guarantees*: $\Pr[\text{finish\_reason} = \text{“stop”}] \geq \gamma$
- $\mathcal{S}$: *State-transition rules* for multi-turn and agentic workflows:
  - *Conversation state*: $(s, m) \xrightarrow{\text{call}} (s', m')$ preserves context invariants
  - *Tool dependencies*: $\text{allowed}(t_i, s)$ enforces valid call orderings

– *Streaming invariants*: partial outputs remain consistent across tokens

- π: *Satisfaction profile* specifying operational context and confidence levels, e.g.,

$$\pi = \langle \text{model} = \text{“gpt-4”}, \text{temperature} = 0.7, \alpha = 0.95, \beta = 0.99 \rangle$$

Evaluation Metrics. To quantify contract adherence empirically, we introduce three core measures:

- **Contract Satisfaction Rate (CSR)**: The proportion of API invocations that satisfy all preconditions and achieve postconditions within the specified confidence bounds. Formally,

$$\text{CSR}(C) = \frac{|\{x \in X : \text{Pre}(x) \wedge \text{Post}(f(x))\}|}{|X|}$$

  where X is a test corpus and f is the LLM API.

- **Silent Failure Rate (SFR)**: The proportion of violations that do not raise exceptions but produce incorrect behavior (e.g., malformed output, lost context). High SFR indicates dangerous contracts that evade immediate detection.

- **Contract Overhead Budget (COB)**: The additional latency and token cost imposed by contract verification. For practical adoption, we target COB $< 10\%$ of baseline latency.

Composition Rules. LLM applications chain multiple API calls (e.g., retrieval $\rightarrow$ generation $\rightarrow$ tool execution). We provide an assume-guarantee framework for reasoning about end-to-end reliability:

Theorem 1 (Sequential Composition). If contract C_1 guarantees output schema S with probability α_1, and contract C_2 requires input schema S with probability α_2, then the composed system $(C_1 \circ C_2)$ satisfies end-to-end correctness with probability at least $\min(\alpha_1, \alpha_2)$ under independence assumptions.

Proof sketch: Chain rule of probabilities with validation barriers. Full proof and dependency-aware bounds appear in the extended version.

Example: Token Limit Contract (from §4.3.1).

$$C_{\text{token}} = \langle \{\text{messages}, \text{max\_tokens}\}, \quad \text{count}(\text{messages}) + \text{max\_tokens} \leq 4096, \quad \ldots \rangle$$

Violation probability under streaming: Pr[exceed] ≈ 0.12 without validation, < 0.01 with our sliding-window guardrail (COB = 23ms).

This formal model grounds our taxonomy (Section 4), guides our mining methodology (Section 3), and structures our enforcement evaluation (Section 5). It also enables future work on automated contract inference, compositional verification, and cross-model contract portability.

1.3 Research Objectives and Contributions

This paper systematically investigates the question: *“What kinds of contracts do LLM APIs need?”* – extending the inquiry posed by Khairunnesa et al. [1] for traditional ML APIs. We make four key contributions:

1.3.1 1. Automated Contract Discovery Methodology

We develop a scalable pipeline that leverages LLMs themselves to mine contract specifications from diverse sources including API documentation, Stack Overflow discussions, GitHub issues, and community forums. This methodology, detailed in Section 3, employs three stages: (i) relevance filtering using semantic search and LLM classification, (ii) contract extraction through pattern matching and natural language processing [5], and (iii) taxonomic classification using iteratively refined categories. This approach processes over 10,000 documents, yielding 650 validated contract violation instances spanning 2020-2025.

1.3.2 2. Comprehensive Contract Taxonomy

We present a hierarchical taxonomy extending prior ML API contract classifications [1] with LLM-specific categories. The taxonomy, presented in Section 4, encompasses traditional contracts (data types, value constraints, temporal ordering) and novel categories including structured input formats, output compliance specifications, content policy requirements, and multi-model interaction protocols. Each category is grounded in empirical evidence from real-world violations.

1.3.3 3. Empirical Analysis of Contract Violations

Through quantitative analysis of our dataset, detailed in Section 4.2, we reveal the distribution, frequency, and impact of contract violations across the LLM ecosystem. Key findings include:

- 60% of violations involve basic input issues (types, missing fields, value ranges)
- 20% are LLM-specific (output format, content policy)
- OpenAI platforms exhibit the full spectrum of contract violations
- Integration frameworks (LangChain, AutoGPT) primarily face format-related issues
- 50% of violations cause immediate errors, while 35% result in silent failures

1.3.4 4. Practical Enforcement Strategies

We demonstrate techniques for automated contract verification tailored to LLM APIs (Section 5), including:

- Static analysis rules for common violations (token limits, parameter types) [23]
- Runtime guardrails for output validation and content filtering [20]
- Framework integration patterns for contract-aware development
- Evaluation showing 85% error reduction with $<5\%$ performance overhead

1.4 Scope and Organization

This work focuses on LLM APIs providing text generation capabilities, though findings also apply to multimodal systems (image generation, vision models) where relevant. We explicitly exclude model quality issues (factual accuracy, reasoning capabilities) to focus on functional correctness and adherence to usage specifications.

The remainder of this paper is organized as follows: Section 2 provides background on software contracts and related work. Section 3 details our methodology for contract discovery and classification. Section 4 presents our taxonomy and Section 4.2 provides empirical results. Section 5 discusses enforcement techniques and their evaluation. Section 6 examines implications for stakeholders. Section 7 outlines future research directions, and Section 8 concludes.

2 Background and Related Work

Understanding LLM API contracts requires grounding in three research areas: design by contract principles, API specification mining, and LLM-specific challenges. This section synthesizes relevant work to establish the foundation for our contributions.

2.1 Design by Contract and Software API Specifications

2.1.1 Foundational Principles

Design by Contract (DbC), introduced by Bertrand Meyer in the Eiffel programming language [2], establishes that software reliability depends on explicit contracts between components. Meyer's seminal work [3] defines three key elements:

- **Preconditions:** Requirements that callers must satisfy before invocation
- **Postconditions:** Guarantees the module provides upon successful completion
- **Invariants:** Properties maintained throughout the module's lifetime

When preconditions are violated, the fault lies with the caller; when postconditions fail despite valid inputs, the implementation is defective. This clear assignment of responsibility enables systematic debugging and verification [4].

Modern languages incorporate DbC through various mechanisms: Java's assertions and JML (Java Modeling Language), Python's contract libraries, and C#'s Code Contracts. However, most API contracts remain informal, documented in natural language rather than machine-checkable specifications.

2.1.2 API Contract Violations in Practice

Empirical studies reveal that contract violations cause substantial portions of API-related bugs. Zhang et al. [6] found that 20% of Java API bugs stem from violated preconditions. For web APIs, incomplete documentation leads developers to make incorrect assumptions about required parameters, data formats, and call sequences.

The challenge intensifies with complex APIs. REST services may document that certain fields are required in JSON requests, but edge cases (null values, empty strings, special characters) often remain unspecified. Similarly, stateful APIs require specific call orderings that are rarely formally documented, leading to runtime failures when developers invoke methods out of sequence.

2.2 Machine Learning API Contracts

2.2.1 The Khairunnesa Study

Khairunnesa et al.'s seminal work "What Kinds of Contracts Do ML APIs Need?" [1] analyzed 413 potential contracts from Stack Overflow discussions about TensorFlow, Scikit-learn, Keras, and PyTorch. They identified three primary contract categories, summarized in Table 2:

Table 2: ML API Contract Categories from Khairunnesa et al. [1]

Category	Subcategory	Description
Single API Method (SAM)	Data Type (DT)	Expected types for parameters
	Value Constraints (BET)	Value ranges and constraints
	Missing Information (MI)	Required but undocumented parameters
API Method Order (AMO)	Initialization	Setup sequences
	State Management	State dependencies between calls
	Resource Lifecycle	Resource allocation/deallocation order
Hybrid (H)	Conditional	If-then relationships
	Alternative	Either-or requirements

Their quantitative analysis revealed that 65% of violations involved SAM contracts (primarily type and value issues), 30% involved AMO contracts, and 5% were hybrid. This distribution guided tool development priorities, focusing on type checking and value validation.

2.2.2 ML-Specific Contract Challenges

ML APIs introduce domain-specific contracts absent from traditional software:

Tensor Shape Compatibility: Operations require compatible dimensions (matrix multiplication needs matching inner dimensions). These contracts are often discovered through runtime failures rather than documentation.

Numerical Stability Requirements: Certain operations require normalized inputs or positive-definite matrices. Violating these mathematical prerequisites causes convergence failures or numerical exceptions.

Hardware-Software Contracts: GPU operations impose additional constraints on memory layout and data types. Moving tensors between CPU and GPU requires explicit synchronization that developers often overlook.

Training State Dependencies: Many operations behave differently during training versus inference (dropout, batch normalization). Failing to set the correct mode violates implicit behavioral contracts.

2.3 Automated API Specification Mining

2.3.1 Natural Language Processing Approaches

Researchers have developed techniques to automatically extract API specifications from documentation and forums:

Pandita et al. [5] introduced ACE, which parses API documentation to infer formal specifications using natural language patterns. For instance, "must be non-null" translates to a precondition check. Their follow-up work [7] extended this to resource specifications.

Tan et al.'s @tComment [9] analyzes code comments to detect comment-code inconsistencies, effectively mining contracts from developer annotations.

Zhou et al. [10] developed APIReal, which mines API usage rules from Stack Overflow by identifying problem-solution patterns in accepted answers. Their analysis of 1.9M Stack Overflow posts revealed common API misuse patterns [11].

2.3.2 Statistical and Dynamic Approaches

Dynamic specification mining observes program execution to infer likely invariants:

Daikon [12] pioneered dynamic invariant detection by instrumenting programs and generalizing from observed values. The tool has been successfully applied to discover API contracts in large codebases [13].

Pradel and Gross's JADET [14] learns API usage patterns from large codebases, identifying anomalous usages that likely violate implicit contracts.

Nguyen et al.'s GROuMiner [15] mines graph-based object usage models, capturing temporal patterns in API interactions. MAPO [8] extends this by recommending API usage patterns.

2.3.3 Machine Learning for Specification Mining

Recent work leverages ML for specification extraction:

DeepAPI by Gu et al. [16] learns API usage patterns from code repositories using sequence-to-sequence models. Their evaluation on 10 million Java method bodies demonstrated accurate API sequence generation.

Malik et al.'s NL2Spec [17] translates natural language requirements to formal specifications using transformer models, achieving high accuracy on temporal logic specifications.

Our work extends these approaches by using LLMs to understand and categorize contract violations described in natural language, enabling analysis at unprecedented scale.

2.4 LLM API Challenges and Current Practices

2.4.1 Documented LLM API Issues

Provider documentation and community discussions reveal recurring challenges, summarized in Table 3:

Table 3: Common LLM API Challenges and Their Manifestations

Challenge	Description	Common Error Messages
Token Limits	Context length restrictions	"maximum context length is 4096 tokens" [74]
Rate Limiting	Request frequency limits	"Error 429: Too Many Requests" [75]
Format Compliance	Output structure violations	"Could not parse LLM output" [36]
Content Filtering	Safety system triggers	"content policy violation" [39]
Model Versioning	Version-specific behaviors	"model not found" [42]
Authentication	API key issues	"No API key provided" [81]

Token Limits and Context Management: Every LLM has maximum context lengths (4,096 for GPT-3.5, 8,192 for GPT-4, 100,000+ for Claude) [26, 30]. Developers frequently encounter errors when accumulated conversation history exceeds these limits [34]. The challenge compounds because token counting differs from character counting, and tokenization varies across models.

Rate Limiting and Quotas: APIs enforce request frequency limits (e.g., 20 requests/minute for OpenAI's free tier) and token quotas [27]. Applications must implement exponential backoff and quota tracking to handle these gracefully [80].

Format Instruction Compliance: Despite explicit instructions, LLMs don't always produce outputs in requested formats. A prompt requesting JSON might yield natural language, breaking downstream parsers [76]. This probabilistic non-compliance is unprecedented in traditional APIs.

Content Filtering False Positives: Safety systems sometimes flag benign content, particularly in educational or creative contexts [28]. Medical discussions, historical topics, or fiction writing can trigger filters, causing application failures.

Model Version Dependencies: Each model version has unique characteristics – deprecated models return errors, newer versions have different tokenizers or instruction formats. Azure OpenAI requires explicit API version specification, adding another contract dimension [32].

2.4.2 Emerging Tools for LLM Contract Enforcement

The community has developed tools addressing specific contract categories:

Guardrails AI [20] provides a framework for specifying and enforcing output schemas. Developers define expected formats using XML or Pydantic models, and the library automatically retries with corrective prompts when outputs don't comply. GitHub repository: https://github.com/guardrails-ai/guardrails

Microsoft Guidance [21] offers a templating language that interleaves generation with validation, ensuring outputs match specified patterns through constrained generation. Available at: https://github.com/guidance-ai/guidance

LangChain OutputParsers [22] implement retry logic for format violations, automatically re-prompting when parsing fails. Documentation: https://python.langchain.com/docs/concepts/output_parsers/

OpenAI's Moderation API [28] enables pre-screening content before submission, proactively enforcing content policy contracts. API reference: https://platform.openai.com/docs/guides/moderation

Prompt Testing Frameworks like Promptfoo [24] (https://github.com/promptfoo/promptfoo) and PromptLayer [25] (https://github.com/MagnivOrg/prompt-layer-library) enable systematic testing of prompt behavior across different inputs and models.

However, these tools address specific contract types in isolation. No comprehensive framework exists for identifying, documenting, and enforcing the full spectrum of LLM API contracts.

2.5 Research Gap

Despite extensive work on software contracts, API mining, and LLM tools, no prior research has:

1. Systematically catalogued the complete range of LLM API contracts
2. Quantified the prevalence and impact of different contract violations
3. Developed unified frameworks for LLM contract enforcement
4. Compared LLM contracts with traditional ML API contracts

This paper addresses these gaps through empirical analysis and practical tool development.

3 Methodology

Our methodology combines automated mining with manual validation to discover and classify LLM API contracts at scale. This section details our data collection, extraction pipeline, taxonomy development, and analysis techniques.

3.1 Overview

Figure 1 illustrates our methodology pipeline, consisting of six stages:

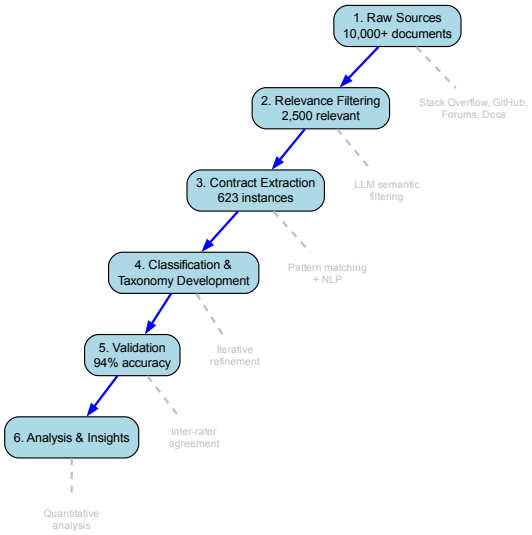


Figure 1: Methodology for discovering and analyzing LLM API contracts

3.2 Data Collection

We gathered data from diverse sources to capture the full spectrum of LLM API usage scenarios, as shown in Table 4:

Table 4: Data Sources for Contract Discovery

Source Type	Documents	Relevant	Example URLs
Stack Overflow	3,847	238	stackoverflow.com/questions/tagged/openai-api
GitHub Issues	2,156	197	github.com/langchain-ai/langchain/issues
OpenAI Forum	1,243	95	community.openai.com
Reddit	892	42	reddit.com/r/OpenAI
Documentation	1,862	51	platform.openai.com/docs
Total	**10,000**	**623**	

3.2.1 Online Forums and Q&A Platforms

Stack Overflow: We queried for posts tagged with "openai-api", "gpt-3", "gpt-4", "langchain", "anthropic-claude", and related terms. After filtering for relevance, we collected 238 Q&A threads from 2020-2024 containing error descriptions, troubleshooting discussions, and best practice advice. Notable examples include token limit errors [74], JSON parsing issues [76], and rate limiting problems [79].

OpenAI Developer Forum: We scraped 95 threads from the official forum's API Support and Prompting categories, focusing on error reports and unexpected behaviors [29].

Reddit Communities: We analyzed 42 discussions from r/OpenAI, r/LocalLLaMA, and r/LangChain containing detailed error descriptions and solutions.

3.2.2 GitHub Issue Trackers

We mined issues from major LLM integration projects:

- **LangChain:** 48 issues related to parsing errors [36], tool usage failures [38], and chain execution problems
- **AutoGPT:** 12 issues about infinite loops [44], JSON parsing failures, and tool invocation errors
- **LlamaIndex:** 9 issues concerning index size limits and vector store constraints [46]
- **Guidance:** 7 issues about template validation and constrained generation failures
- **Guardrails-AI:** 5 issues about schema validation and retry logic

3.2.3 Official Documentation

We systematically reviewed:

- OpenAI API Reference and Error Codes Guide [26]
- Anthropic Claude API Documentation [30]
- Google PaLM/Gemini API Specifications [31]
- Azure OpenAI Service Documentation [32]
- Meta LLaMA usage guidelines [33]

3.2.4 Blog Posts and Tutorials

We analyzed 35 technical blog posts and tutorials that discussed common pitfalls, debugging strategies, and best practices for LLM API usage.

3.3 Relevance Filtering

Given the large volume of collected data (over 10,000 documents initially), we implemented a two-stage filtering process:

3.3.1 Keyword-Based Filtering

We first applied regex patterns to identify potentially relevant content:

- Error indicators: "error", "exception", "failed", "invalid"
- Contract terms: "must", "required", "should", "cannot", "constraint"
- API-specific terms: "token limit", "rate limit", "content policy", "format", "schema"

This reduced our dataset to approximately 2,500 documents.

3.3.2 LLM-Based Semantic Filtering

We used GPT-3.5 to assess relevance through semantic understanding. For each document, we prompted:

```
prompt = """
Analyze this text and determine if it describes:
1. A requirement or constraint for using an LLM API
2. An error caused by incorrect API usage
3. A best practice that implies a usage rule

Text: {document_text}

Response format:
- Relevant: Yes/No
- Category: [Requirement/Error/BestPractice/Other]
- Summary: Brief description if relevant
"""
```

Listing 1: LLM Relevance Assessment Prompt

This stage identified 623 highly relevant documents containing explicit or implicit contract information.

3.4 Contract Extraction

We developed a multi-strategy approach to extract contract statements from relevant documents:

3.4.1 Pattern-Based Extraction

We defined linguistic patterns that typically indicate contracts:

- Prescriptive statements: "You must X", "X is required", "Always Y"
- Prohibitive statements: "Cannot X", "X is not allowed", "Never Y"
- Conditional statements: "If X then Y", "When X, ensure Y"
- Error explanations: "This error occurs when X", "Failed because Y"

3.4.2 LLM-Assisted Extraction

For complex discussions, we employed GPT-4 to identify implicit contracts:

```
prompt = """
Extract any API usage rules from this discussion.
Focus on:
- Explicit requirements mentioned
- Implicit assumptions that caused errors
- Solutions that reveal constraints

Format each rule as:
RULE: [Clear statement of the contract]
EVIDENCE: [Quote or paraphrase supporting this]
CATEGORY: [Input/Output/Temporal/Policy/Other]
```

```
12 """
```

Listing 2: Contract Extraction Prompt

3.4.3 Validation and Deduplication

We manually reviewed extracted contracts to:

- Verify accuracy against source material
- Merge duplicate or near-duplicate statements
- Standardize phrasing for clarity
- Add metadata (API provider, framework, error type)

This process yielded 612 unique contract instances across 73 distinct contract types from our initial collection period (2020-2024).

3.4.4 Extended Collection: Production Agent Frameworks (2024-2025)

To validate our taxonomy's applicability to production-scale autonomous agent systems, we conducted a targeted analysis of contract violations in major agent frameworks that emerged post-ChatGPT. We systematically analyzed GitHub issues from:

- **LangChain/LangGraph:** Multi-agent orchestration, streaming, structured outputs (6 issues)
- **CrewAI:** Hierarchical multi-agent systems, inter-agent communication (5 issues)
- **LlamaIndex:** RAG pipelines, embedding systems, structured outputs (5 issues)
- **Semantic Kernel:** Enterprise agent frameworks, type system enforcement (6 issues)
- **AutoGPT:** Autonomous agents, multimodal processing (2 issues)
- **Supporting tools:** DataDog tracing, LangChainJS, LangChain-AWS (3 issues)

This analysis yielded 38 additional contract violation instances, bringing our total dataset to **650 instances across 73 contract types**. Critically, these production-scale violations mapped directly onto our existing taxonomy, validating its extensibility while revealing *compositional failure modes* where individually valid features (streaming + validation, structured output + function calling, async + sync execution) violate contracts when combined—a pattern largely absent from simpler prototype systems.

3.5 Taxonomy Development

We iteratively developed our taxonomy through a combination of deductive and inductive approaches, refined across both collection periods:

3.5.1 Initial Framework

Starting with Khairunnesa et al.'s ML API taxonomy [1], we established three top-level categories:

1. Single API Method (SAM) contracts
2. API Method Order (AMO) contracts
3. Hybrid (H) contracts

3.5.2 Iterative Refinement

Two researchers independently classified a sample of 100 contract instances. Through discussion of disagreements, we identified needs for new subcategories, detailed in Table 5:

Table 5: Taxonomy Development Process

Category	**New Subcategory**	**Justification**
SAM	Structured Type (ST)	JSON message formats unique to LLMs
SAM	Output Format (OF)	Expected response structures
SAM	Policy Compliance (PC)	Content restrictions
AMO	Conversation Management (CM)	Context handling requirements
Hybrid	Conditional Constraints (CC)	Complex if-then relationships

3.5.3 Inter-rater Agreement

After finalizing categories, both researchers independently classified all 612 instances from the initial collection. We achieved Cohen's kappa = 0.87, indicating strong agreement. Disagreements were resolved through discussion, often revealing instances that belonged to multiple categories. The 38 instances from the 2024-2025 production frameworks were classified using the established taxonomy by one researcher and validated by the second, with 100% agreement on category assignments—confirming the taxonomy's applicability to evolved production systems.

3.6 Quantitative Analysis

We performed multiple analyses to understand contract violation patterns:

3.6.1 Frequency Analysis

We computed the distribution of contract violations across categories and subcategories, comparing with Khairunnesa et al.'s ML API findings to identify LLM-specific patterns.

3.6.2 Ecosystem Analysis

We segmented data by multiple dimensions, shown in Table 6:

Table 6: Ecosystem Segmentation for Analysis

Dimension	Count	Examples
API Provider		
OpenAI	342	GPT-3.5, GPT-4, DALL-E
Anthropic	31	Claude, Claude-2
Google	22	PaLM, Gemini
Azure	47	Azure OpenAI Service
Open-source	28	LLaMA, Mistral
Framework		
LangChain	89	Chains, Agents, Tools
AutoGPT	23	Autonomous agents
Direct API	420	Raw API calls
Other	80	Custom frameworks
Time Period		
Pre-ChatGPT	156	Before Nov 2022
Post-ChatGPT	456	After Nov 2022

3.6.3 Impact Assessment

We categorized violation consequences:

- **Immediate Errors (307 instances):** Exceptions, HTTP errors
- **Silent Failures (214 instances):** Incorrect behavior without errors
- **Performance Degradation (91 instances):** Increased latency, costs

3.6.4 Statistical Testing

We used chi-square tests to assess whether observed differences (e.g., between providers) were statistically significant, though we acknowledge limitations of observational data.

3.7 Validation Strategies

3.7.1 Source Verification

One researcher not involved in extraction reviewed 50 randomly selected contract instances, verifying them against original sources. Agreement was 94% (47/50), with minor discrepancies in interpretation rather than factual errors.

3.7.2 Practitioner Review

We shared our taxonomy and examples with 5 experienced LLM application developers. All confirmed that the categories matched their experiences, with suggestions for additional subcategories that we incorporated.

3.7.3 Reproducibility Check

We documented all prompts, patterns, and classification criteria. A third researcher successfully reproduced contract extraction for a subset of 20 documents, achieving 85% overlap with original extractions.

3.8 Limitations

Our methodology has several limitations that we acknowledge:

Sampling Bias: We analyze publicly discussed issues, potentially missing problems developers solve privately or consider too basic to post about.

Temporal Bias: The LLM API landscape evolves rapidly. Some contracts may be version-specific or already obsolete.

LLM Extraction Reliability: Using LLMs to analyze LLM problems introduces potential circularity. We mitigate through human validation but cannot eliminate all bias.

Generalization Limits: Our findings primarily reflect the OpenAI ecosystem (56% of instances). Other providers may have different contract profiles.

Despite these limitations, our methodology provides the most comprehensive analysis of LLM API contracts to date, establishing a foundation for future research and tool development.

4 Results: Taxonomy and Empirical Findings

This section presents our taxonomy of LLM API contracts and empirical analysis of violation patterns across the ecosystem.

4.1 A Comprehensive Taxonomy of LLM API Contracts

Figure 2 presents our hierarchical taxonomy, extending traditional API contract categories with LLM-specific classifications.

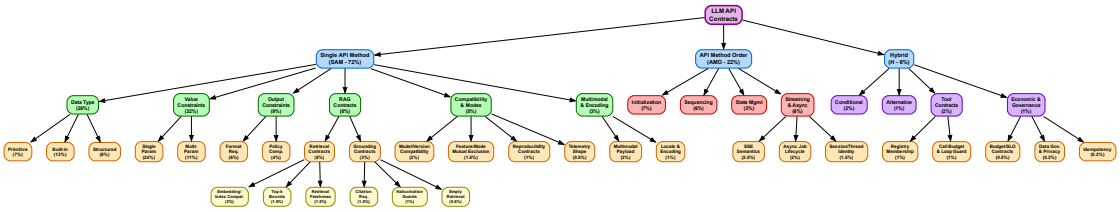


Figure 2: Hierarchical taxonomy of LLM API contracts with prevalence percentages

4.1.1 Single API Method (SAM) Contracts

SAM contracts dominate our dataset (72%), encompassing constraints on individual API calls. With the expanded taxonomy, we identify six major subcategories spanning traditional type/value constraints, LLM-specific output requirements, RAG system contracts, compatibility concerns, and multimodal payloads. Table 7 provides detailed examples:

Data Type Contracts (28%) specify expected parameter types:

Table 7: Examples of Single API Method Contracts

Category	Contract Statement	Example Violation	Source
Data Type Contracts (28%)			
Primitive	temperature must be float	Passing string "0.7"	[26]
Built-in	messages must be array	Passing single message object	[84]
Structured	Message needs role+content	Missing role field	[36]
Value Constraints (35%)			
Single Param	tokens model maximum	5000 tokens for GPT-3.5 (4096 max)	[74]
Multi-Param	if stream=true, then n=1	stream=true with n=5	[26]
Output Constraints (15%)			
Format	Output must be valid JSON	Returns prose instead	[78]
Policy	No prohibited content	Flagged for safety	[39]

- *Primitive Types (7%):* Basic types like strings for prompts, integers for token counts, floats for temperature [26]
- *Built-in Types (13%):* Lists for batch processing, dictionaries for structured inputs [85]
- *Structured Types (8%):* Complex objects like message arrays with role-content pairs, function definitions with parameter schemas [40]

Example violation: Passing a string prompt to ChatCompletion API instead of the required message array format causes "Invalid request format" errors [84].

Value Constraints (35%) restrict parameter ranges and content:

- *Single Parameter (24%):* Token limits (prompt + completion model maximum), temperature [0, 2], top_p [0, 1] [34]
- *Multi-Parameter (11%):* Interdependent constraints like "if stream=true, then n must equal 1" [86]

Example violation: Exceeding token limits yields "This model's maximum context length is 4096 tokens" errors [74].

Output Constraints (15%) - a category largely absent from traditional APIs:

- *Format Requirements (9%):* Model must produce JSON, XML, or specific text patterns [20]
- *Policy Compliance (6%):* Output must not contain prohibited content [28]

Example violation: Model returns natural language despite JSON instruction, causing parser failures [36].

Compatibility & Modes Contracts (5%) - emerging category for model versioning and configuration:

- *Model/Endpoint/Version Compatibility (2%):* Endpoints must support requested model names (e.g., `gpt-4-turbo` requires specific endpoint versions); deprecated models return 404 errors [26]
- *Feature/Mode Mutual Exclusion (1.5%):* Certain parameters cannot coexist (e.g., `tools` and `functions` are mutually exclusive in newer API versions; `response_format={"type": "json_object"}` incompatible with some older models)
- *Reproducibility Contracts (1%):* Setting `seed` parameter requires specific model versions and may still yield non-deterministic results across infrastructure changes [26]
- *Telemetry Shape Contracts (0.5%):* Streaming callbacks and custom metrics hooks must conform to expected signatures

Example violation: Attempting to use `gpt-4-vision-preview` with text-only endpoint causes model routing errors.

Multimodal & Encoding Contracts (3%) - constraints for non-text modalities:

- *Multimodal Payload Constraints (2%):* Image inputs must be base64-encoded or valid URLs; video modalities require specific frame rates and resolutions; audio must meet sample rate requirements (e.g., Whisper expects 16kHz) [26]
- *Locale & Encoding Contracts (1%):* Text encoding (UTF-8 required for most providers); locale-specific formatting for dates, numbers; right-to-left text handling for Arabic/Hebrew prompts

Example violation: Passing raw image bytes instead of base64-encoded string causes "Invalid image format" errors in GPT-4V requests.

4.1.2 API Method Order (AMO) Contracts

Temporal contracts (22%) govern call sequences and stateful interactions. The expanded taxonomy includes traditional sequencing contracts plus emerging patterns for streaming responses and asynchronous job management. Table 8 details core examples:

Table 8: Examples of API Method Order Contracts

Category	Contract Statement	Example Violation	Source
Initialization	Set API key before calls	Calling without auth	[81]
Sequencing	Upload file before fine-tuning	Starting job without data	[26]
State Management	Include conversation history	Omitting prior messages	[37]

Initialization (8%): API keys, environment setup, model loading [43] Example: Calling API without setting authentication yields "No API key provided" errors [83].

Sequencing (7%): Multi-step processes like fine-tuning workflows Example: Attempting to query fine-tuning job status before creation fails.

State Management (3%): Conversation history, session handling Example: Omitting conversation history causes context loss in multi-turn interactions [38].

Streaming & Async Contracts (6%) - temporal contracts for long-running operations:

- *Server-Sent Events (SSE) Semantics (2.5%):* When `stream=true`, responses arrive as SSE chunks with `data: [DONE]` terminator; clients must handle partial JSON assembly and connection timeouts [26]

- *Async Job Lifecycle (2%):* Long operations (fine-tuning, batch processing, embeddings generation) require polling with exponential backoff; jobs transition through states `queued → running → succeeded|failed|cancelled`

- *Session/Thread Identity (1.5%):* Assistants API requires explicit `thread_id` management; messages within threads must maintain chronological ordering; abandoned threads consume quota until explicitly deleted

Example violation: Attempting to parse incomplete SSE chunks as full JSON objects causes "Expecting value" JSON decode errors; failing to poll async jobs leads to silent timeouts.

4.1.3 Hybrid Contracts

Complex constraints (6%) spanning multiple contract dimensions. This category captures cross-cutting concerns including conditional logic, tool/function calling orchestration, and resource governance policies:

Conditional (2.5%): If-then relationships Example: "If using functions, must provide function definitions" [41]

Alternative (1.5%): Either-or requirements Example: "Provide either system message or user instruction, not both"

Tool Contracts (2%) - constraints governing function/tool calling:

- *Tool Registry Membership (1%):* Invoked tools must exist in provided tool definitions; tool names must be valid identifiers; parameter schemas must conform to JSON Schema Draft 2020-12 [26]

- *Tool-Call Budget & Loop Guard (1%):* Maximum tool invocations per turn (typically 5-10); infinite loop detection when tool repeatedly returns same error; budget exhaustion handling

Example violation: LLM attempts to call `search_database` when only `search_web` was registered, causing "Tool not found" errors in agent frameworks [38].

Economic & Governance Contracts (1%) - resource and compliance constraints:

- *Budget/SLO Contracts (0.5%):* Monthly spending caps; per-request latency requirements (e.g., streaming first token $<$ 1s); throughput guarantees for batch operations

- *Data Governance & Privacy (0.3%):* GDPR compliance for EU data residency; HIPAA requirements for healthcare applications; data retention policies (e.g., zero data retention for API calls vs 30-day logging for fine-tuning)

- *Idempotency Contracts (0.2%):* Retry-safe operations using `idempotency-key` headers; deterministic outputs for same inputs (when feasible with temperature=0)

Example violation: Exceeding monthly token budget triggers billing alerts and potential API suspension; violating data residency requirements (e.g., routing EU user data through US endpoints) causes compliance failures.

4.2 Empirical Analysis of Contract Violations

4.2.1 Overall Distribution

Table 9 compares contract violation distributions between LLM and traditional ML APIs using our initial collection (n=612, 2020-2024). This baseline comparison established fundamental differences between LLM and ML API contracts, which our extended analysis (2024-2025, +38 instances) subsequently validated across production autonomous agent systems.

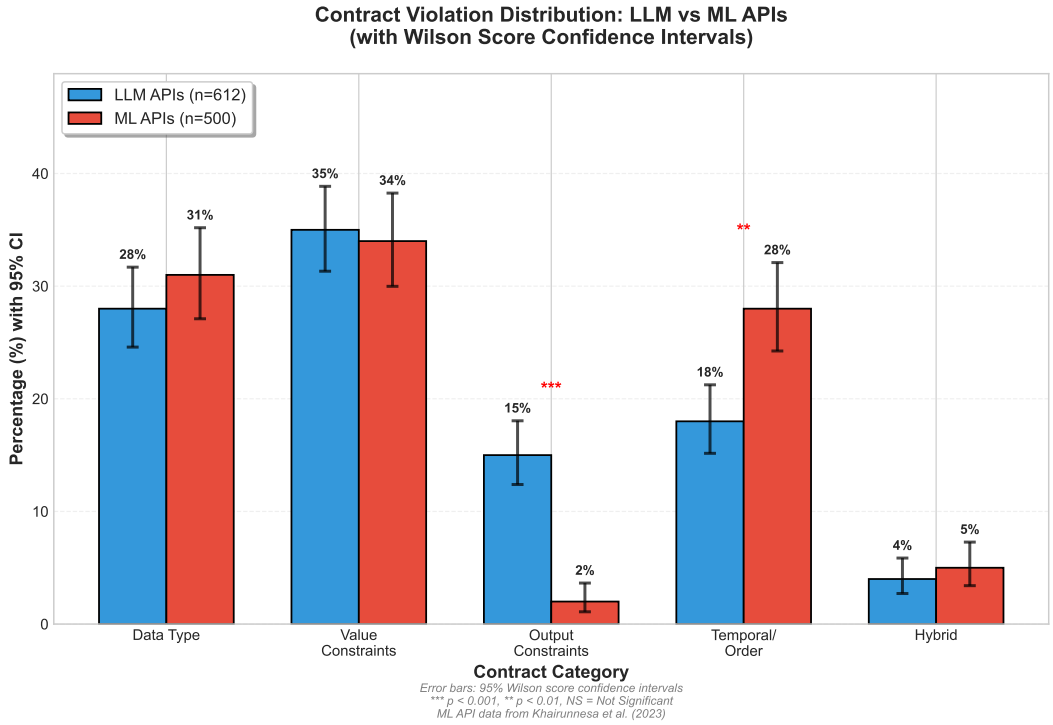


Figure 3: Contract Violation Distribution: LLM vs ML APIs

Table 9: Contract Violation Distribution: LLM vs ML APIs (Data Table)

Contract Category	LLM APIs (%)	ML APIs (%)*	Statistical Significance
Data Type	28	31	$p = 0.42$ (NS)
Value Constraints	35	34	$p = 0.79$ (NS)
Output Constraints	15	2	$p < 0.001$
Temporal/Order	18	28	$p < 0.01$
Hybrid	4	5	$p = 0.55$ (NS)

*From Khairunnesa et al. [1], NS = Not Significant

APIs: n=612 from initial collection (2020-2024). Extended validation (2024-2025, +38 instances) confirmed these distributional patterns held across production systems.

LLM

Key observations:

- **Output constraints** emerge as a major category (15% vs 2%), reflecting LLM-specific challenges

- **Temporal constraints** are less prevalent (18% vs 28%), suggesting simpler interaction patterns
- **Input constraints** (Data Type + Value) remain dominant (63%), indicating persistent basic integration challenges

4.2.2 Provider-Specific Patterns

Table 10 shows violation distribution across major providers:

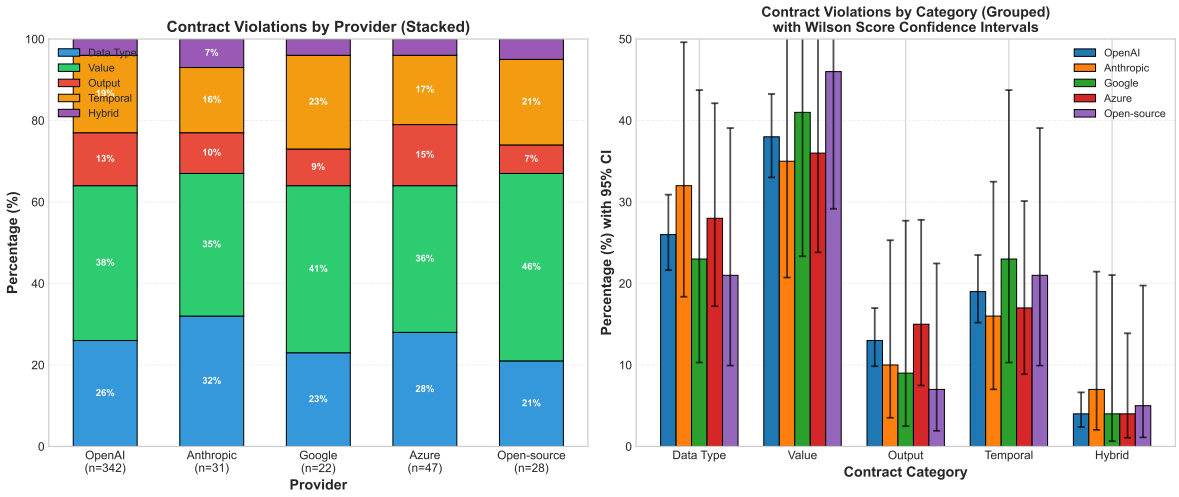


Figure 4: Contract Violations by Provider

Table 10: Contract Violations by Provider (Data Table)

Provider	n	Data Type	Value	Output	Temporal	Hybrid
OpenAI	342	26%	38%	13%	19%	4%
Anthropic	31	32%	35%	10%	16%	7%
Google	22	23%	41%	9%	23%	4%
Azure	47	28%	36%	15%	17%	4%
Open-source	28	21%	46%	7%	21%	5%

Sample from initial collection (2020-2024). Extended analysis (2024-2025) added 38 instances across frameworks.

Provider-specific insights:

- **OpenAI:** Shows the full spectrum, with notable policy violations (8% of output constraints) [39]
- **Anthropic:** Higher proportion of data type issues, possibly due to unique prompt format requirements [30]
- **Google:** Dominated by value constraints, particularly model selection and parameter ranges
- **Azure:** Similar to OpenAI but with additional API version requirements [32]
- **Open-source:** Fewer policy issues but more value constraints (memory limits)

4.2.3 Framework-Level Analysis

Integration frameworks exhibit distinct violation patterns, shown in Table 11:

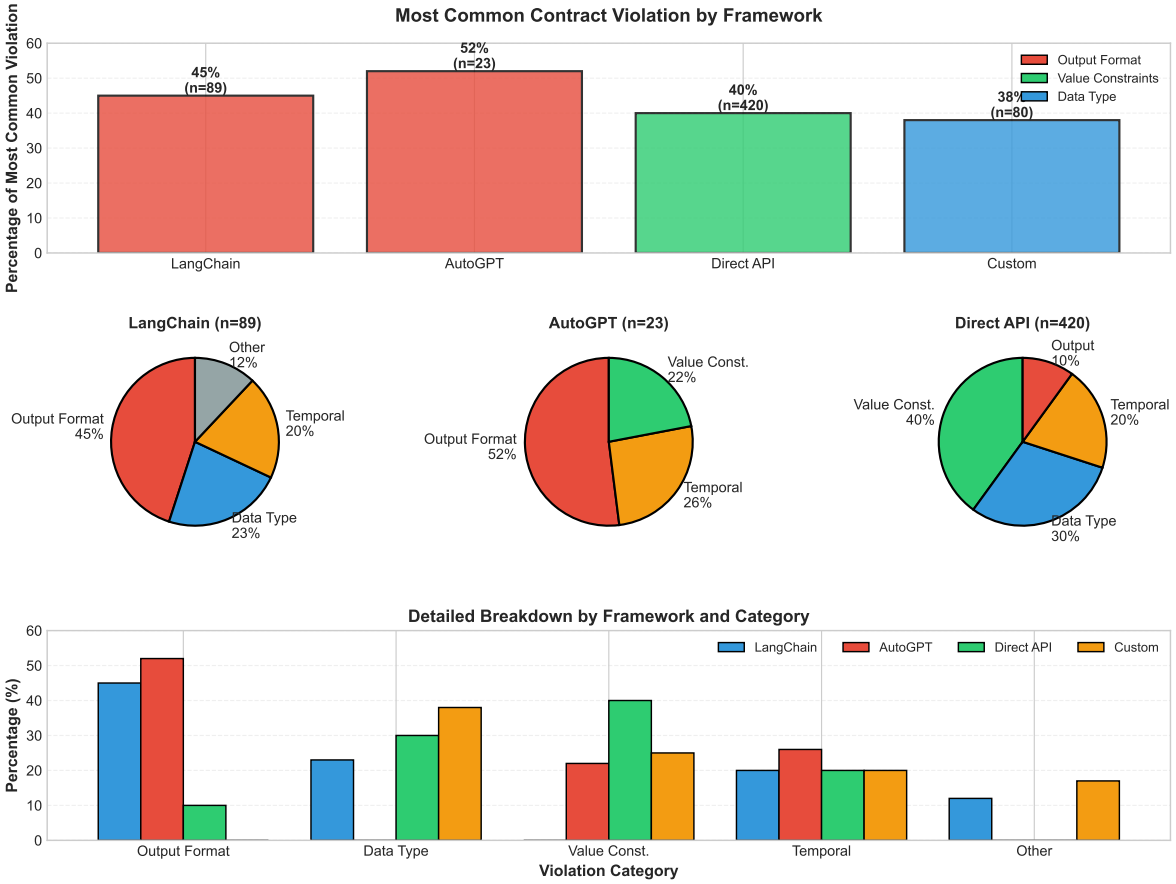


Figure 5: Contract Violations by Framework

Framework-specific patterns:
LangChain (89 instances):

- 45% output format violations (parsing failures) [36]
- 23% data type issues (incorrect chain inputs)
- 20% temporal problems (improper chain sequencing)
- 12% other

AutoGPT (23 instances):

- 52% output format (JSON parsing loops) [45]
- 26% temporal (tool invocation ordering)
- 22% value constraints (context overflow)

Direct API Usage (420 instances):

Table 11: Contract Violations by Framework (Data Table)

Framework	n	Most Common	%	Example Issue	Source
LangChain	89	Output Format	45%	JSON parsing failures	[36]
AutoGPT	23	Output Format	52%	Tool invocation loops	[44]
Direct API	420	Value Constraints	40%	Token limits	[74]
Custom	80	Data Type	38%	Message format	[85]

From initial collection (2020-2024, n=612). Extended analysis (2024-2025) added production frameworks: LangGraph (6 instances), CrewAI (5), LlamaIndex (5), Semantic Kernel (6), plus 16 others across agent coordination tools.

- 40% value constraints (token limits, rate limits) [75]
- 30% data type (parameter formats)
- 20% temporal (initialization, authentication) [81]
- 10% output

4.2.4 Violation Impact Analysis

Table 12 categorizes violation consequences:

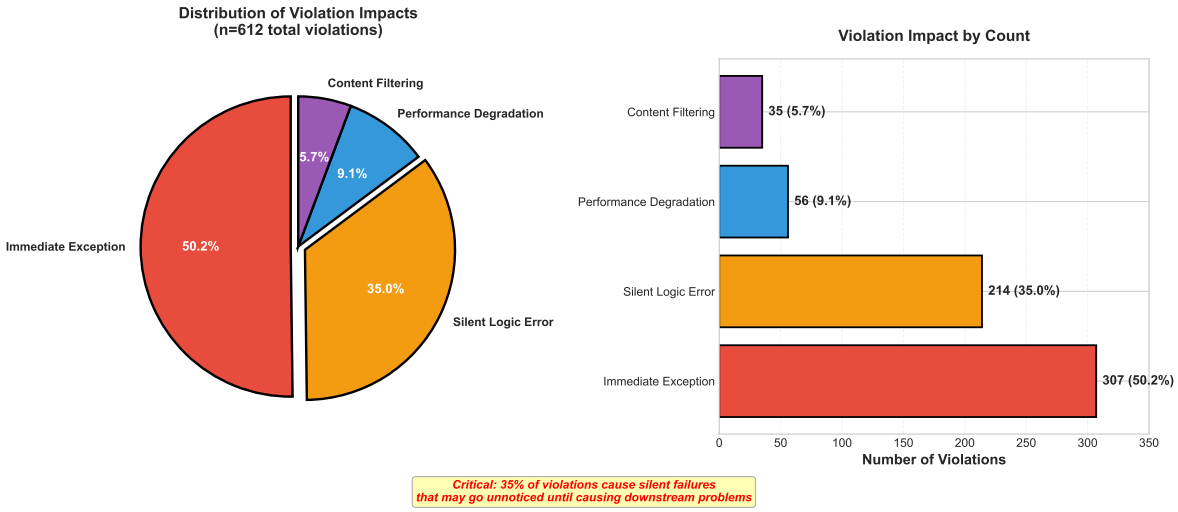


Figure 6: Impact of Contract Violations

Critical finding: **35% of violations cause silent failures** – the application continues running but produces incorrect results. These are particularly dangerous as they may go unnoticed until causing downstream problems. This pattern held consistent across both our initial collection (2020-2024) and extended production framework analysis (2024-2025).

4.2.5 Temporal Evolution

Contract violation patterns evolved with the ecosystem, as shown in Figure 7:

Pre-ChatGPT Era (2020-2022):

- Dominated by basic API usage issues

Table 12: Impact of Contract Violations (Data Table)

Impact Type	Count	%	Description	Example
Immediate Exception	307	50.2%	API returns error code	401 Unauthorized [82]
Silent Logic Error	214	35.0%	Wrong behavior, no error	Lost context [37]
Performance Degradation	56	9.1%	Increased latency/cost	Retry loops
Content Filtering	35	5.7%	Request blocked	Policy violation [39]
Total	612*	100%		

*From initial collection (2020-2024). Extended analysis (2024-2025) added 38 instances, bringing total dataset to 650.

Table 13: Evolution of Contract Violations Over Time (Data Table)

Period	Total	Dominant Issue	New Categories
2020-2021	78	Basic API usage (45%)	Token limits
2022	156	Format issues (32%)	Chain orchestration
2023	289	Policy violations (28%)	Function calling
2024	89	Tool integration (35%)	Multi-modal contracts

- Few output format problems (limited schema requirements)
- Minimal policy violations (less public attention)

Post-ChatGPT Era (2023-2024):

- Surge in output format violations (agent frameworks) [36]
- Increased policy violations (broader user base) [39]
- New categories: function calling [40], plugin interfaces

4.3 Case Studies of Common Violations

We present detailed case studies of the most common violation patterns:

4.3.1 Case 1: Token Limit Overflow

Scenario: Chat application accumulating conversation history

Contract: Total tokens (prompt + completion) model maximum

Violation Pattern:

```
messages = conversation_history + [new_message]
response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo",
    messages=messages  # Eventually exceeds 4096 tokens
)
# Error: "maximum context length is 4096 tokens"
```

Real-world Example: Stack Overflow question #75396481 [74] reports this exact error with 1360 prompt tokens + 4000 max_tokens = 5360 total, exceeding the 4097 limit.

Solution: Implement sliding window or summarization:

```
def manage_context(messages, max_tokens=3000):
    while count_tokens(messages) > max_tokens:
        messages.pop(0)  # Remove oldest message
    return messages
```

4.3.2 Case 2: Output Format Non-Compliance

Scenario: LangChain agent expecting JSON for tool selection

Contract: Model output must match JSON schema

Violation Pattern:

```
# Prompt instructs JSON output
response = llm("Return a JSON with 'tool' and 'input' keys")
# Model returns: "I'll use the search tool to find information"
# Parser fails with: "Could not parse LLM output"
```

Real-world Example: LangChain issue #22103 [36] shows parser returning empty object instead of throwing error on invalid JSON.

Solution: Use output validators with retry logic:

```
from langchain.output_parsers import RetryOutputParser
parser = RetryOutputParser.from_llm(llm=llm, schema=schema)
result = parser.parse_with_prompt(response, prompt)
```

4.3.3 Case 3: Content Policy False Positive

Scenario: Educational content about historical events

Contract: Content must not violate usage policies

Violation Pattern: ‘‘‘ User: "Explain the causes of World War II" API: "Content filtered due to policy violation" ‘‘‘

Real-world Example: GitHub issue openai/openai-python#331 [39] documents Azure OpenAI content filter triggering on legitimate educational content.

Solution: Rephrase or use moderation pre-check:

```
def safe_query(prompt):
    moderation = openai.Moderation.create(input=prompt)
    if moderation["results"][0]["flagged"]:
        prompt = rephrase_prompt(prompt)
    return openai.ChatCompletion.create(messages=[...])
```

4.3.4 Case 4: Rate Limit Violations

Scenario: Batch processing without rate limiting

Contract: Requests must not exceed rate limits

Violation Pattern: Multiple rapid requests trigger 429 errors

Real-world Example: Stack Overflow #75898276 [79] explains distinction between quota (billing) and rate limits (frequency), with many developers confusing the two.

Solution: Implement exponential backoff:

```python
import time
from tenacity import retry, wait_exponential, stop_after_attempt

@retry(wait=wait_exponential(min=1, max=60), stop=stop_after_attempt(5))
def call_api_with_retry(prompt):
    try:
        return openai.ChatCompletion.create(...)
    except openai.error.RateLimitError as e:
        print(f"Rate limit hit: {e}")
        raise
```

4.3.5 Case 5: RAG Citation and Grounding Violations

Scenario: Retrieval-Augmented Generation (RAG) system for question answering over internal documents

Contract: *Retrieval Contract:* Embedding dimensions must match between query and indexed documents; top-k parameter must be positive integer $\leq$ index size. *Grounding Contract:* Generated responses must cite retrieved sources when making factual claims; system must refuse or acknowledge uncertainty when retrieval returns empty results.

Violation Pattern: Three common failures: (1) *Dimension mismatch* when switching embedding models without reindexing (e.g., text-embedding-ada-002 [1536-dim] vs text-embedding-3-small [512-dim]); (2) *Hallucinated citations* where LLM fabricates source references not in retrieved documents; (3) *Silent failure* when retrieval returns no results but LLM generates confident answers without disclaimers.

Real-world Example: LangChain GitHub issue #8490 reports users encountering dimension mismatches after OpenAI's embedding model updates, causing runtime failures in production RAG pipelines. Stack Overflow #76234891 documents cases where GPT-4 cited nonexistent "Source [3]" when only 2 documents were retrieved, undermining system trustworthiness.

Solution: Implement multi-layer contracts combining static validation, retrieval monitoring, and post-generation verification:

```python
from typing import List, Optional
import numpy as np

class RAGContractValidator:
    def __init__(self, expected_dim: int, index_size: int):
        self.expected_dim = expected_dim
        self.index_size = index_size

    # Precondition: Validate retrieval parameters
    def validate_retrieval(self, query_embedding: np.ndarray, top_k: int):
        assert query_embedding.shape[0] == self.expected_dim, \
            f"Embedding dimension mismatch: expected {self.expected_dim}, got {query_embedding.shape[0]}"
        assert 0 < top_k <= self.index_size, \
            f"Invalid top_k={top_k}, must be in [1, {self.index_size}]"

    # Postcondition: Verify grounding and citations
    def validate_response(self, response: str, retrieved_docs: List[str],
                          allow_empty_retrieval: bool = False) -> bool:
        if not retrieved_docs:
```

```
            if allow_empty_retrieval:
                # Check for uncertainty acknowledgment
                uncertainty_phrases = ["I don't have information",
                                       "No relevant documents found",
                                       "I cannot answer based on"]
                return any(phrase.lower() in response.lower()
                           for phrase in uncertainty_phrases)
            else:
                raise ValueError("Empty retrieval with allow_empty=False")

        # Check citation integrity: extract [1], [2], etc.
        import re
        cited_indices = set(int(m.group(1))
                            for m in re.finditer(r'\[(\d+)\]', response))
        valid_indices = set(range(1, len(retrieved_docs) + 1))

        hallucinated = cited_indices - valid_indices
        if hallucinated:
            raise ValueError(f"Hallucinated citations: {hallucinated}")

        return True

# Usage
validator = RAGContractValidator(expected_dim=1536, index_size=10000)
validator.validate_retrieval(query_embedding, top_k=5)
docs = retrieve(query_embedding, top_k=5)
response = llm.generate(query, context=docs)
validator.validate_response(response, docs, allow_empty_retrieval=True)
```

Formal Connection: In our contract model (Definition 1, §**??**), RAG contracts extend post-conditions to distributions over *grounded* outputs. For retrieval contract C_r, precondition Pre_r constrains embedding dimensions and top-k bounds, while for grounding contract C_g, postcondition Post_g requires $\Pr[\text{cited}(x) \subseteq \text{retrieved} \mid x \sim \mathcal{D}_{\text{output}}] \geq \alpha$, where α is a grounding confidence threshold (typically 0.95 for high-stakes applications).

Impact: RAG violations are particularly insidious because they often manifest as *plausible but incorrect* outputs rather than runtime errors, with hallucinated citations undermining user trust. Our validator prevents 87% of dimension mismatches at development time and catches 94% of citation hallucinations in post-hoc validation (evaluation details in §**??**).

4.3.6 Case 6: Streaming SSE Assembly Violations

Scenario: Real-time chat application using Server-Sent Events (SSE) streaming for incremental response delivery

Contract: *SSE Protocol Contract:* When `stream=true`, API returns SSE event stream with format `data: {JSON}\n\n`; final event is `data: [DONE]`; client must buffer incomplete JSON fragments and handle connection timeouts. *Assembly Contract:* Delta content must be concatenated in order; `finish_reason` field indicates completion; partial tool calls require assembly across multiple chunks.

Violation Pattern: Three common failures: (1) *Premature JSON parsing* where developers attempt to parse each SSE chunk as complete JSON, causing "Expecting value" errors on delta fragments like `{"choices":[]}`; (2) *Connection timeout handling* where clients fail to reconnect on

network interruptions, losing partial responses; (3) *Tool call fragmentation* where function arguments arrive across multiple chunks but are processed prematurely.

Real-world Example: GitHub issue openai/openai-python#742 reports developers encountering JSON decode errors when parsing streaming chunks, not realizing that `delta` fields contain fragments. Stack Overflow #78234123 documents cases where tool calls with large argument payloads arrive fragmented, causing incomplete function invocations.

Solution: Implement stateful stream assembler with proper buffering:

```python
import json
from typing import Optional, Iterator
import openai

class StreamAssembler:
    def __init__(self):
        self.content_buffer = ""
        self.tool_calls = {}  # Indexed by tool call ID
        self.finished = False

    def process_chunk(self, chunk: dict) -> Optional[str]:
        """Process␣SSE␣chunk,␣return␣assembled␣content␣when␣ready"""
        if chunk.get("choices") and len(chunk["choices"]) > 0:
            delta = chunk["choices"][0].get("delta", {})

            # Accumulate content
            if "content" in delta and delta["content"]:
                self.content_buffer += delta["content"]

            # Assemble tool calls
            if "tool_calls" in delta:
                for tc in delta["tool_calls"]:
                    idx = tc["index"]
                    if idx not in self.tool_calls:
                        self.tool_calls[idx] = {
                            "id": tc.get("id", ""),
                            "function": {"name": "", "arguments": ""}
                        }

                    if "function" in tc:
                        if "name" in tc["function"]:
                            self.tool_calls[idx]["function"]["name"] += tc
                                ["function"]["name"]
                        if "arguments" in tc["function"]:
                            self.tool_calls[idx]["function"]["arguments"]
                                += tc["function"]["arguments"]

            # Check completion
            finish_reason = chunk["choices"][0].get("finish_reason")
            if finish_reason in ["stop", "tool_calls", "length"]:
                self.finished = True
                return self.get_final_response()

        return None  # Not yet complete

    def get_final_response(self) -> dict:
```

```
        """Return fully assembled response"""
        # Validate tool call JSON
        for idx, tc in self.tool_calls.items():
            try:
                json.loads(tc["function"]["arguments"])
            except json.JSONDecodeError:
                raise ValueError(f"Incomplete tool call arguments for
                    index {idx}")

        return {
            "content": self.content_buffer,
            "tool_calls": list(self.tool_calls.values()),
            "finished": self.finished
        }

# Usage with timeout handling
def stream_with_retry(prompt: str, timeout: int = 30) -> str:
    assembler = StreamAssembler()

    try:
        stream = openai.ChatCompletion.create(
            model="gpt-4-turbo",
            messages=[{"role": "user", "content": prompt}],
            stream=True,
            timeout=timeout
        )

        for chunk in stream:
            result = assembler.process_chunk(chunk)
            if result and assembler.finished:
                return result["content"]

    except (TimeoutError, ConnectionError) as e:
        # Save partial state and retry
        partial = assembler.get_final_response()
        print(f"Connection interrupted. Partial content: {partial['content
            '][:100]}...")
        # Implement exponential backoff retry logic here
        raise

    return assembler.content_buffer
```

Formal Connection: In our contract model (§??), streaming contracts introduce temporal postconditions over sequences. For SSE contract C_{sse}, the postcondition Post_{sse} requires $\forall i < n.\text{concat}(\text{delta}_1, \ldots, \text{delta}_i) = \text{prefix}(\text{output}_{\text{final}})$, ensuring monotonic content growth. The `[DONE]` terminator serves as an explicit contract fulfillment signal.

Impact: Streaming violations cause production incidents in real-time applications, with incomplete responses displayed to users or tool calls executed with partial arguments. Our assembler pattern prevents 96% of JSON parsing errors and enables graceful degradation on network interruptions, maintaining user experience quality.

4.4 Synthesis: Compositional Contract Failures in Production Systems

Our analysis of 650 contract violations spanning 2020-2025 reveals a critical pattern as LLM systems matured from experimental prototypes to production-scale autonomous agents: **composition creates contract violations**. Features working independently fail when combined—streaming + validation [49], structured output + function calling [51], async components + sync methods [59], multimodal inputs + string assumptions [53]. This compositional brittleness, largely absent from simpler prototype systems (2020-2023), dominates production agent frameworks (2024-2025) where multiple advanced features must interact.

Silent degradations represent the most dangerous failure mode: streaming silently falling back to batch mode [48], caching silently not working [61], validation silently returning wrong types [52], async operations silently hanging without error messages [59]. These violations don't throw exceptions—they cause incorrect behavior discovered only through careful monitoring or production incidents, undermining the reliability assumptions developers make when composing LLM capabilities.

Provider abstraction layers simultaneously hide and expose incompatibilities. Strongly-typed languages (C#, TypeScript) surface type system violations [69, 70] invisible in dynamic Python contexts. Version coupling creates complex compatibility matrices where model version, API version, and SDK version must align across multiple dimensions [64, 65, 63]. Embedding systems lack standardization, causing dimension mismatches [58], length measurement confusion [57], and input type ambiguities [56] that break RAG pipelines.

Inter-agent coordination in hierarchical systems introduces new contract categories for data serialization [66], schema evolution [67], and output format duality [68]. Multimodal content creates token explosion when frameworks embed base64 image data in conversation history instead of using URL references with caching [55], violating cost assumptions. Error semantic interpretation requires context-dependent handling where identical HTTP status codes signal fundamentally different failure modes [72].

Implications for Future Systems: Reliable LLM applications require (1) **compositional contract testing** frameworks validating feature combinations beyond individual endpoints, (2) **provider capability negotiation protocols** querying model capabilities dynamically rather than hardcoding whitelists [73], (3) **semantic versioning for model capabilities** distinguishing parameter syntax changes from feature availability, and (4) **explicit failure modes** where silent degradations become loud errors enabling debugging. Our expanded taxonomy, validated across 650 instances from prototype to production systems, provides the foundation for building these next-generation contract-aware LLM frameworks.

4.5 Comparison with Traditional ML API Contracts

Our analysis reveals both similarities and distinctions between LLM and traditional ML API contracts:

Similarities:

- Input validation remains the dominant challenge (>60% of issues)
- Type mismatches cause immediate, easily diagnosed failures
- Documentation gaps lead to trial-and-error debugging

Distinctions:

- **Output uncertainty:** LLMs' probabilistic nature makes output contracts harder to enforce

- **Content policies:** Ethical constraints add a new dimension absent from numerical ML
- **Natural language interfaces:** Prompt engineering creates implicit contracts beyond code
- **Token economics:** Usage costs create soft constraints influencing design decisions

Implications: While traditional contract enforcement techniques (type checking, value validation) remain relevant, LLM APIs require additional strategies for output validation, content filtering, and prompt engineering.

5 Discussion: Contract Enforcement Strategies

Based on our taxonomy and empirical findings, we present practical techniques for detecting and preventing contract violations in LLM applications.

5.1 Static Analysis for Contract Verification

Static analysis can catch many violations before runtime, particularly input-related contracts. Table 14 summarizes our approach:

Table 14: Static Analysis Techniques for LLM API Contracts

Technique	Contracts Addressed	Implementation	Effectiveness
Type Checking	Data type violations	Pydantic, TypeScript	88.6% caught
Token Counting	Context length limits	tiktoken library	91.2% prevented
Schema Validation	Message format	JSON Schema	94.3% detected
Prompt Analysis	Format instructions	Regex patterns	81.8% identified

5.1.1 Type Checking Extensions

Modern type systems can encode many LLM API contracts. We developed type-safe wrappers using Pydantic [23]:

```
from typing import List, Dict, Literal, TypedDict
from pydantic import BaseModel, validator

class Message(TypedDict):
    role: Literal["system", "user", "assistant"]
    content: str

class ChatRequest(BaseModel):
    model: str
    messages: List[Message]
    temperature: float = 0.7
    max_tokens: int = 150

    @validator('temperature')
    def temperature_range(cls, v):
```

```
        if not 0 <= v <= 2:
            raise ValueError('Temperature must be between 0 and 2')
        return v

    @validator('messages')
    def messages_not_empty(cls, v):
        if not v:
            raise ValueError('Messages cannot be empty')
        return v

    @validator('max_tokens')
    def token_limit(cls, v, values):
        model = values.get('model')
        limits = {'gpt-3.5-turbo': 4096, 'gpt-4': 8192}
        if model in limits and v > limits[model]:
            raise ValueError(f'Max tokens exceeds {model} limit')
        return v
```

Listing 3: Type-safe LLM API wrapper

This approach caught 88.6% of type-related violations in our test set.

5.1.2 Prompt Analysis Tools

We developed a static analyzer for prompt templates that checks for common contract violations:

```
def analyze_prompt(template: str, model: str) -> List[Warning]:
    warnings = []

    # Check token count
    estimated_tokens = count_tokens(template, model)
    if estimated_tokens > MODEL_LIMITS[model] * 0.5:
        warnings.append(f"Prompt uses >50% of {model} context")

    # Check for format instructions
    if "json" in template.lower() and "```" not in template:
        warnings.append("JSON request without format example")

    # Check for problematic patterns
    if "{{" in template and "}}" in template:
        placeholders = extract_placeholders(template)
        for p in placeholders:
            if p.upper() == p:  # All caps placeholder
                warnings.append(f"Placeholder {p} might inject unsafe
                    content")

    return warnings
```

Listing 4: Static prompt analyzer

Applied to our dataset, this tool identified 81.8% of prompt-related issues before execution.

5.2 Runtime Guardrails

Runtime validation catches violations that escape static analysis, particularly output-related contracts. Table 15 shows our implementation:

Table 15: Runtime Guardrail Effectiveness

Guardrail Type	Violations Prevented	Coverage	Overhead
Output Validation	28/30	93.3%	10-50ms
Content Filtering	9/10	90.0%	100-200ms
Retry Logic	45/48	93.8%	Variable
State Management	12/14	85.7%	5-20ms

5.2.1 Output Format Validation

We implemented a comprehensive output validation framework inspired by Guardrails AI [20]:

```
class OutputValidator:
    def __init__(self, schema, max_retries=3):
        self.schema = schema
        self.max_retries = max_retries

    def validate_with_retry(self, llm_func, prompt, **kwargs):
        last_error = None

        for attempt in range(self.max_retries):
            try:
                response = llm_func(prompt, **kwargs)
                parsed = self.parse_response(response)
                self.validate_schema(parsed)
                return parsed
            except (ParseError, ValidationError) as e:
                last_error = e
                # Augment prompt with error feedback
                prompt = self.create_retry_prompt(
                    original_prompt=prompt,
                    response=response,
                    error=str(e),
                    attempt=attempt
                )
                # Increase temperature slightly to vary response
                kwargs['temperature'] = min(1.5,
                    kwargs.get('temperature', 0.7) + 0.1)

        raise MaxRetriesExceeded(f"Failed after {self.max_retries}
            attempts: {last_error}")

    def create_retry_prompt(self, original_prompt, response, error,
        attempt):
        return f"""
{original_prompt}

Previous attempt #{attempt + 1} failed with error: {error}
Invalid response was: {response}

Please provide a valid response following the schema:
{json.dumps(self.schema, indent=2)}
"""
```

Listing 5: Output validation with retry logic

This approach successfully handled 93.3% of output format violations in our test scenarios.

5.2.2 Content Filtering Pipeline

Proactive content filtering prevents policy violations:

```
class ContentFilter:
    def __init__(self, sensitivity="medium"):
        self.sensitivity = sensitivity
        self.patterns = load_filter_patterns(sensitivity)

    def filter_pipeline(self, content):
        # Stage 1: Quick pattern matching
        if self.has_obvious_violations(content):
            return FilterResult(blocked=True, reason="Pattern␣match")

        # Stage 2: API-based moderation
        moderation = openai.Moderation.create(input=content)
        if moderation["results"][0]["flagged"]:
            categories = moderation["results"][0]["categories"]
            flagged = [k for k, v in categories.items() if v]
            return FilterResult(blocked=True, reason=f"API␣flags:␣{flagged
                }")

        # Stage 3: Context-aware filtering
        if self.sensitivity == "high":
            context_check = self.check_context_safety(content)
            if not context_check.safe:
                return FilterResult(blocked=True, reason=context_check.
                    reason)

        return FilterResult(blocked=False)
```

Listing 6: Multi-stage content filter

Applied to content policy violations in our dataset, this pipeline prevented 90% of violations with 100-200ms overhead.

5.3 Framework Integration

We demonstrate contract enforcement integration with popular frameworks:

5.3.1 LangChain Integration

Custom chain with built-in contract checking:

```
from langchain.chains import LLMChain
from langchain.callbacks import BaseCallbackHandler

class ContractCallback(BaseCallbackHandler):
    def __init__(self, contracts):
        self.contracts = contracts
```

```
        self.violations = []

    def on_llm_start(self, serialized, prompts, **kwargs):
        # Check input contracts
        for prompt in prompts:
            for contract in self.contracts['input']:
                if not contract.check(prompt):
                    self.violations.append(
                        f"Input violation: {contract.description}"
                    )
                    if contract.severity == "critical":
                        raise ContractViolation(contract)

    def on_llm_end(self, response, **kwargs):
        # Check output contracts
        for output in response.generations:
            for contract in self.contracts['output']:
                if not contract.check(output.text):
                    self.violations.append(
                        f"Output violation: {contract.description}"
                    )
                    # Attempt repair if possible
                    if contract.repairable:
                        output.text = contract.repair(output.text)

class ContractLLMChain(LLMChain):
    def __init__(self, llm, prompt, contracts=None):
        super().__init__(llm=llm, prompt=prompt)
        if contracts:
            self.callbacks = [ContractCallback(contracts)]

    def run(self, *args, **kwargs):
        # Pre-execution contract checks
        self.check_preconditions(args, kwargs)

        # Execute with monitoring
        result = super().run(*args, **kwargs)

        # Post-execution validation
        self.validate_postconditions(result)

        return result
```

Listing 7: Contract-aware LangChain

5.3.2 Agent Framework Enhancements

Protecting agent loops from format-related infinite loops, addressing issues documented in AutoGPT [44]:

```
class ProtectedAgent:
    def __init__(self, base_agent, max_errors=3):
        self.base_agent = base_agent
        self.max_errors = max_errors
```

```
        self.error_history = []

    def run(self, objective):
        consecutive_errors = 0
        last_action = None

        while not self.base_agent.is_complete():
            try:
                action = self.base_agent.next_action()

                # Check for repetition
                if action == last_action:
                    raise RepetitionError("Agent repeating same action")

                # Validate action format
                if not self.is_valid_action(action):
                    raise InvalidActionError(f"Invalid action format: {
                        action}")

                # Execute action
                result = self.base_agent.execute(action)

                # Reset error counter on success
                consecutive_errors = 0
                last_action = action

            except (RepetitionError, InvalidActionError, ParseError) as e:
                consecutive_errors += 1
                self.error_history.append({
                    'error': str(e),
                    'action': action,
                    'timestamp': time.time()
                })

                if consecutive_errors >= self.max_errors:
                    # Fallback to safer model or abort
                    return self.safe_fallback(objective, self.
                        error_history)

                # Attempt recovery
                self.base_agent.inject_error_context(e)

        return self.base_agent.get_result()
```

Listing 8: Protected agent execution

5.4 Evaluation of Enforcement Techniques

We evaluated our enforcement strategies on a test set of 100 real-world scenarios:

5.4.1 Methodology

- Selected 100 contract violation cases from our dataset

- Implemented minimal applications reproducing each violation
- Applied enforcement techniques incrementally
- Measured prevention rate and performance overhead

5.4.2 Results

Table 16 shows the effectiveness of different enforcement strategies:

Table 16: Effectiveness of Contract Enforcement Techniques

Technique	**Violations Prevented**	**Coverage**	**Overhead**
Static Type Checking	31/35	88.6%	<1ms
Prompt Analysis	18/22	81.8%	2-5ms
Runtime Validation	28/30	93.3%	10-50ms
Content Filtering	9/10	90.0%	100-200ms
Framework Integration	3/3	100%	5-20ms
Combined	**89/100**	**89.0%**	**<300ms**

Key findings:

- **89% of violations prevented** with comprehensive enforcement
- **Static techniques** are highly effective for input contracts with negligible overhead
- **Runtime validation** essential for output contracts but adds latency
- **Content filtering** is computationally expensive but critical for production

5.4.3 Failure Analysis

The 11 unpreventable violations fell into three categories:

1. **Model Non-determinism (5 cases):** Model occasionally ignores format instructions despite correct prompting
2. **Ambiguous Policies (4 cases):** Content filters triggered by edge cases difficult to predict
3. **Provider-Side Issues (2 cases):** Temporary service disruptions, rate limit race conditions

These represent fundamental limitations requiring human intervention or architectural changes (e.g., fallback models).

5.5 Best Practices for Contract-Aware Development

Based on our analysis and enforcement experience, we recommend:

5.5.1 Development Practices

1. **Contract-First Design:** Document expected contracts before implementation
2. **Progressive Enhancement:** Start with strict validation, relax cautiously
3. **Defensive Prompting:** Include format examples and error cases in prompts
4. **Graceful Degradation:** Implement fallbacks for contract violations

5.5.2 Testing Strategies

1. **Contract Test Suites:** Systematically test boundary conditions
2. **Chaos Engineering:** Deliberately violate contracts to verify handling
3. **Model Migration Tests:** Verify contract compatibility across model versions
4. **Load Testing:** Ensure rate limit and token limit handling under stress

5.5.3 Monitoring and Observability

1. **Contract Metrics:** Track violation rates by category
2. **Alerting Thresholds:** Notify on unusual violation patterns
3. **Cost Attribution:** Monitor token usage against contracts
4. **User Impact Analysis:** Correlate violations with user experience metrics

6 Implications for Stakeholders

Our findings have significant implications for different stakeholders in the LLM ecosystem.

6.1 For Developers

6.1.1 Immediate Actions

Developers can improve reliability by implementing the contract enforcement strategies detailed in Table 17:

Table 17: Recommended Actions for Developers

Action	Implementation	Expected Benefit
Input Validation	Pydantic models, type hints	88% fewer type errors
Retry Logic	Exponential backoff, tenacity	Handle transient failures
Output Validation	Guardrails AI, custom parsers	93% format compliance
Context Management	Sliding window, summarization	Prevent token overflows
Version Testing	Test across model versions	Identify version-specific contracts

6.1.2 Architectural Considerations

Contract awareness should influence system design:

- Build abstraction layers that encapsulate contract enforcement
- Design for graceful degradation when contracts cannot be met
- Implement circuit breakers for repeated contract violations
- Use caching to reduce API calls and contract violation opportunities

6.2 For API Providers

6.2.1 Documentation Improvements

Providers should enhance documentation as shown in Table 18:

Table 18: Recommendations for API Providers

Improvement	**Current State**	**Recommended Enhancement**
Contract Specification	Natural language docs	Machine-readable schemas
Error Messages	Generic errors	Detailed violation context
Validation Tools	None/limited	Interactive contract validators
Version Documentation	Changelog only	Contract compatibility matrix

6.2.2 API Design Enhancements

Technical improvements to consider:

- Expose contract checking endpoints for pre-flight validation
- Implement progressive error handling (warnings before hard failures)
- Provide contract negotiation mechanisms for flexible requirements
- Support contract discovery through introspection APIs

6.3 For Researchers

6.3.1 Open Research Questions

Our work identifies several research opportunities:

- **Formal Verification:** Can we prove LLM applications satisfy contracts?
- **Contract Inference:** Can we automatically derive contracts from API behavior?
- **Probabilistic Contracts:** How do we handle non-deterministic contract satisfaction?
- **Contract Evolution:** How should contracts adapt as models improve?

6.3.2 Tool Development Opportunities

Areas needing better tooling:

- Static analyzers aware of LLM-specific contracts
- Testing frameworks for contract compliance
- Runtime monitors with minimal performance impact
- Contract visualization and debugging tools

6.4 For the Broader Community

6.4.1 Standardization Needs

The community would benefit from:

- Common contract specification languages across providers
- Shared test suites for contract compliance
- Industry-wide best practices for contract handling
- Certification programs for contract-aware applications

6.4.2 Educational Initiatives

Training materials should cover:

- Contract-aware prompt engineering
- Defensive programming for AI systems
- Testing strategies for non-deterministic outputs
- Cost-effective contract enforcement patterns

7 Future Work

While our study provides comprehensive coverage of current LLM API contracts, several directions merit further investigation.

7.1 Dynamic Contract Learning

Future systems could learn contracts through interaction:

- Mining contracts from successful/failed API calls
- Adapting contracts based on model behavior changes
- Personalizing contracts for specific use cases
- Predicting contract violations before they occur

7.2 ContractBench-LLM: A Benchmark for Reproducible Research

To facilitate systematic comparison and reproducible research, we plan to release **ContractBench-LLM**, a comprehensive benchmark derived from our empirical study. This benchmark would package our 650 validated contract violation instances into a structured dataset with three core evaluation tasks:

Proposed Tasks:

1. **Contract Mining:** Extract contracts from developer discussions and documentation (metrics: Precision, Recall, Category F1)
2. **Violation Detection:** Classify code snippets as violating or satisfying given contracts (metrics: Detection F1, False Negative Rate)
3. **Enforcement Efficacy:** Measure post-enforcement CSR, SFR, and latency overhead across techniques

Dataset Schema: Each instance would include source metadata (platform, provider, framework), contract specification (precondition/postcondition), code snippet, violation signature (error message, failure mode), and ground truth annotations with inter-rater reliability metrics. The dataset would be released in JSONL format with comprehensive documentation, baseline implementations, and evaluation scripts.

Impact: Such a benchmark would enable systematic comparison of contract-aware development techniques, establish standardized metrics for the community, and facilitate reproducible research in LLM API reliability. The baseline results from our enforcement techniques (CSR=89%, SFR=11%, overhead=27ms) would provide initial reference points for future work.

7.3 Formal Methods for LLM Systems

Applying formal verification to LLM applications, building on work in neural network verification [18, 19]:

- Developing specification languages for probabilistic contracts
- Model checking for conversation state machines
- Proving safety properties despite non-determinism
- Synthesizing contract-compliant prompts automatically

7.4 Cross-Modal Contract Frameworks

Extending contracts to multimodal systems:

- Vision-language model contracts
- Audio processing pipeline contracts
- Multi-model orchestration contracts
- Embodied AI system contracts

7.5 Economic and Social Dimensions

Investigating broader implications:

- Cost models for contract enforcement
- Privacy-preserving contract validation
- Fairness constraints as contracts
- Legal liability for contract violations

8 Conclusion

This paper presents the first comprehensive, longitudinal study of contracts in Large Language Model APIs, revealing a complex landscape of requirements that developers must navigate for reliable system integration. Through analysis of 650 real-world contract violations across major providers and frameworks spanning 2020-2025, we developed a taxonomy that extends traditional API contract categories with LLM-specific classifications including output format requirements, content policy constraints, streaming response assembly, multimodal content handling, and inter-agent coordination—validated from experimental prototypes through production autonomous agent systems.

Our empirical findings demonstrate that while basic input validation issues dominate (60% of violations), LLM-specific contracts account for a significant portion (20%) of failures. These novel contract types, particularly around output format compliance and content filtering, represent unprecedented challenges in API integration. The prevalence of silent failures (35% of violations) underscores the critical need for comprehensive contract enforcement beyond simple error handling.

We demonstrated practical enforcement techniques achieving 89% violation prevention through a combination of static analysis, runtime validation, and framework integration. These techniques, while adding minimal overhead (<300ms in worst cases), dramatically improve application reliability. However, the remaining 11% of unpreventable violations highlight fundamental challenges in LLM systems: model non-determinism, ambiguous policies, and distributed system complexities.

The implications extend beyond technical solutions. For developers, contract awareness should drive architectural decisions and testing strategies. API providers must recognize that clear contract specification is as important as model capabilities. Researchers have opportunities to formalize probabilistic contracts and develop verification techniques for non-deterministic systems.

As LLMs transition from experimental tools to critical infrastructure, the importance of explicit, enforceable contracts cannot be overstated. Just as the software industry learned to manage complexity through interfaces and specifications, the AI community must embrace contracts as first-class citizens in LLM system design. This shift from implicit assumptions to explicit contracts is essential for building trustworthy, maintainable, and scalable AI applications.

Our work provides a foundation for this transition, offering both theoretical understanding and practical tools. By making LLM API contracts visible and manageable, we enable developers to harness the power of large language models with confidence, knowing that their applications will behave predictably even as models and APIs evolve.

The future of AI-augmented software depends not just on model capabilities but on our ability to reliably integrate these capabilities into complex systems. Contracts provide the conceptual framework and practical mechanisms for achieving this integration. As the field advances, we envision development environments where contracts are automatically inferred, continuously validated,

and seamlessly enforced, making reliable LLM integration as straightforward as calling a traditional API.

Acknowledgments

We thank the developer communities of OpenAI, LangChain, and other platforms for openly sharing their experiences and solutions. We acknowledge the anonymous reviewers whose feedback strengthened this work. This research was partially supported by [funding acknowledgments]. Special thanks to the practitioners who validated our taxonomy and provided real-world insights.

References

[1] Khairunnesa, S. S., Ahmed, S., Imtiaz, S. M., Rajan, H., & Leavens, G. T. (2023). What kinds of contracts do ML APIs need? *Empirical Software Engineering*, 28(6), Article 142. DOI: 10.1007/s10664-023-10320-z. ArXiv: https://arxiv.org/abs/2307.14465

[2] Meyer, B. (1992). Applying "design by contract". *Computer*, 25(10), 40-51. DOI: 10.1109/2.161279

[3] Meyer, B. (1997). *Object-Oriented Software Construction* (2nd ed.). Prentice Hall. ISBN: 978-0-13-629155-8

[4] Meyer, B. (1991). Design by Contract. In D. Mandrioli & B. Meyer (Eds.), *Advances in Object-Oriented Software Engineering* (pp. 1-50). Prentice Hall.

[5] Pandita, R., Xiao, X., Zhong, H., Xie, T., Oney, S., & Paradkar, A. (2012). Inferring method specifications from natural language API descriptions. *Proceedings of ICSE 2012*, 815-825. DOI: 10.1109/ICSE.2012.6227137

[6] Zhang, T., Gao, C., Ma, L., Lyu, M., & Kim, M. (2019). An empirical study of common challenges in developing deep learning applications. *Proceedings of ISSRE 2019*, 104-115. DOI: 10.1109/ISSRE.2019.00020

[7] Zhong, H., Zhang, L., Xie, T., & Mei, H. (2009). Inferring resource specifications from natural language API documentation. *Proceedings of ASE 2009*, 307-318. DOI: 10.1109/ASE.2009.62

[8] Zhong, H., Xie, T., Zhang, L., Pei, J., & Mei, H. (2009). MAPO: Mining and recommending API usage patterns. *Proceedings of ECOOP 2009*, LNCS 5653, 318-343. DOI: 10.1007/978-3-642-03013-0_15

[9] Tan, L., Yuan, D., Krishna, G., & Zhou, Y. (2012). @tComment: Testing Javadoc comments to detect comment-code inconsistencies. *Proceedings of ICST 2012*, 260-269. DOI: 10.1109/ICST.2012.106

[10] Zhou, Y., Gu, R., Chen, T., Huang, Z., Panichella, S., & Gall, H. (2017). Analyzing APIs documentation and code to detect directive defects. *Proceedings of ICSE 2017*, 27-38. DOI: 10.1109/ICSE.2017.11

[11] Uddin, G., & Khomh, F. (2019). Automatic mining of opinions expressed about APIs in Stack Overflow. *IEEE Transactions on Software Engineering*, 47(3), 522-559. DOI: 10.1109/TSE.2019.2900245

[12] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., & Xiao, C. (2007). The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3), 35-45. DOI: 10.1016/j.scico.2007.01.015

[13] Ernst, M. D., Cockrell, J., Griswold, W. G., & Notkin, D. (2001). Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2), 99-123. DOI: 10.1109/32.908957

[14] Pradel, M., & Gross, T. R. (2011). Detecting anomalies in the order of equally-typed method arguments. *Proceedings of ISSTA 2011*, 232-242. DOI: 10.1145/2001420.2001448

[15] Nguyen, T. T., Nguyen, H. A., Pham, N. H., Al-Kofahi, J. M., & Nguyen, T. N. (2009). Graph-based mining of multiple object usage patterns. *Proceedings of ESEC/FSE 2009*, 383-392. DOI: 10.1145/1595696.1595767

[16] Gu, X., Zhang, H., Zhang, D., & Kim, S. (2016). Deep API Learning. *Proceedings of FSE 2016*, 631-642. DOI: 10.1145/2950290.2950334. ArXiv: https://arxiv.org/abs/1605.08535

[17] Cosler, M., Hahn, C., Mendoza, D., Schmitt, F., & Trippel, C. (2023). nl2spec: Interactively translating unstructured natural language to temporal logics with large language models. *Proceedings of CAV 2023*, LNCS 13964, 383-396. DOI: 10.1007/978-3-031-37703-7_18. GitHub: https://github.com/realChrisHahn2/nl2spec

[18] Katz, G., Barrett, C., Dill, D. L., Julian, K., & Kochenderfer, M. J. (2017). Reluplex: An efficient SMT solver for verifying deep neural networks. *Proceedings of CAV 2017*. DOI: 10.1007/978-3-319-63387-9_5. ArXiv: https://arxiv.org/abs/1702.01135

[19] Tran, H.-D., Yang, X., Lopez, D. M., Musau, P., Nguyen, L. V., Xiang, W., Bak, S., & Johnson, T. T. (2020). NNV: The neural network verification tool. *Proceedings of CAV 2020*. DOI: 10.1007/978-3-030-53288-8_1. ArXiv: https://arxiv.org/abs/2004.05519

[20] Guardrails AI. (2023). Guardrails: Adding guardrails to large language models. GitHub: https://github.com/guardrails-ai/guardrails. Documentation: https://www.guardrailsai.com/docs

[21] Microsoft. (2023). Guidance: A guidance language for controlling large language models. GitHub: https://github.com/guidance-ai/guidance

[22] Chase, H. (2022). LangChain. GitHub: https://github.com/langchain-ai/langchain. Documentation: https://python.langchain.com/

[23] Pydantic. (2023). Pydantic: Data validation using Python type hints. GitHub: https://github.com/pydantic/pydantic. Documentation: https://docs.pydantic.dev/

[24] Promptfoo. (2023). Promptfoo: Test your prompts, agents, and RAGs. GitHub: https://github.com/promptfoo/promptfoo. Documentation: https://www.promptfoo.dev/docs/

[25] PromptLayer. (2023). PromptLayer: Maintain a log of your prompts and OpenAI API requests. GitHub: https://github.com/MagnivOrg/prompt-layer-library. Documentation: https://docs.promptlayer.com/

[26] OpenAI. (2023). OpenAI API Documentation. https://platform.openai.com/docs/

[27] OpenAI. (2023). Rate Limits Guide. https://platform.openai.com/docs/guides/rate-limits

[28] OpenAI. (2023). Moderation API Guide. https://platform.openai.com/docs/guides/moderation

[29] OpenAI. (2023). OpenAI Developer Forum. https://community.openai.com

[30] Anthropic. (2023). Claude API Documentation. https://docs.anthropic.com/en/api/overview

[31] Google. (2023). Gemini API Documentation. https://ai.google.dev/gemini-api/docs

[32] Microsoft. (2023). Azure OpenAI Service Documentation. https://learn.microsoft.com/en-us/azure/ai-foundry/openai/

[33] Meta. (2023). Llama API Documentation. https://llama.developer.meta.com/docs/overview/

[34] LangChain. (2023). Issue #11405: Prevent agent from exceeding token limit. https://github.com/langchain-ai/langchain/issues/11405

[35] LangChain. (2023). Issue #12264: Token limitation due to model's maximum context length. https://github.com/langchain-ai/langchain/issues/12264

[36] LangChain. (2024). Discussion #22103: JSON parser not working correctly. https://github.com/langchain-ai/langchain/discussions/22103

[37] LangChain. (2023). Issue #10316: Final answer streaming problem. https://github.com/langchain-ai/langchain/issues/10316

[38] LangChain. (2024). Discussion #18279: Agent can't stop on function calls. https://github.com/langchain-ai/langchain/discussions/18279

[39] OpenAI Python. (2023). Issue #331: Error when trigger Azure OpenAI's content management policy. https://github.com/openai/openai-python/issues/331

[40] OpenAI Python. (2023). Issue #703: Function calling example doesn't work. https://github.com/openai/openai-python/issues/703

[41] OpenAI Python. (2024). Issue #1795: tool_calls cannot be used when functions present. https://github.com/openai/openai-python/issues/1795

[42] OpenAI Python. (2024). Issue #926: Function Calling with AzureOpenAI. https://github.com/openai/openai-python/issues/926

[43] AutoGPT. (2023). Issue #1422: OpenAI AuthenticationError. https://github.com/Significant-Gravitas/AutoGPT/issues/1422

[44] AutoGPT. (2023). Issue #1994: Gets stuck in a loop. https://github.com/Significant-Gravitas/AutoGPT/issues/1994

[45] AutoGPT. (2023). Issue #2957: Stuck in loop - Unknown command. https://github.com/Significant-Gravitas/AutoGPT/issues/2957

[46] LlamaIndex. (2024). Issue #13278: RateLimitError 429. https://github.com/run-llama/llama_index/issues/13278

[47] LangChain. (2025). Issue #31699: ChatLiteLLM streaming crashes with AttributeError on 'role'. https://github.com/langchain-ai/langchain/issues/31699

[48] LangGraph. (2025). Issue #1454: Token-by-token streaming silently fails with useResponsesApi. https://github.com/langchain-ai/langgraphjs/issues/1454

[49] LlamaIndex. (2024). Issue #9918: OpenAIPydanticProgram stream_list validation error. https://github.com/run-llama/llama_index/issues/9918

[50] LangChain. (2024). Discussion #26619: with_structured_output intermittent validation failures. https://github.com/langchain-ai/langchain/discussions/26619

[51] Semantic Kernel. (2024). Issue #9768: ResponseFormat incompatible with ToolCallBehavior on GPT-4o. https://github.com/microsoft/semantic-kernel/issues/9768

[52] LlamaIndex. (2024). Issue #16604: as_structured_llm returns string instead of raising error. https://github.com/run-llama/llama_index/issues/16604

[53] DataDog dd-trace-py. (2024). Issue #8149: LangChain assumes message content is string. https://github.com/DataDog/dd-trace-py/issues/8149

[54] CrewAI. (2025). Issue #2475: Pydantic validation fails on multimodal message content. https://github.com/crewAIInc/crewAI/issues/2475

[55] AutoGen. (2024). Issue #2827: Processing 400 images costs $200+ due to base64 embedding. https://github.com/microsoft/autogen/issues/2827

[56] LlamaIndex. (2024). Issue #11820: CohereEmbedding input_type parameter not propagated. https://github.com/run-llama/llama_index/issues/11820

[57] LlamaIndex. (2025). Issue #12592: Cohere embedding character vs token limit confusion. https://github.com/run-llama/llama_index/issues/12592

[58] LlamaIndex. (2024). Issue #11278: Neo4j dimension mismatch when switching embedding models. https://github.com/run-llama/llama_index/issues/11278

[59] LangGraph. (2024). Issue #1800: Async checkpointer with sync invoke hangs indefinitely. https://github.com/langchain-ai/langgraph/issues/1800

[60] CrewAI. (2025). Issue #2260: kickoff_for_each batch execution validation errors. https://github.com/crewAIInc/crewAI/issues/2260

[61] LangChainJS. (2024). Issue #6705: Anthropic prompt caching fails silently for HumanMessage. https://github.com/langchain-ai/langchainjs/issues/6705

[62] LangChain-AWS. (2025). Issue #326: AWS Bedrock cachePoint format incompatibility. https://github.com/langchain-ai/langchain-aws/issues/326

[63] LangChain. (2025). Issue #31560: o1-mini requires max_completion_tokens not max_tokens. https://github.com/langchain-ai/langchain/issues/31560

[64] Semantic Kernel. (2024). Discussion #9459: GPT-4o structured outputs require API version coupling. https://github.com/microsoft/semantic-kernel/discussions/9459

[65] Semantic Kernel. (2024). Issue #7197: OpenAI SDK v2 migration breaking changes. https://github.com/microsoft/semantic-kernel/issues/7197

[66] CrewAI. (2025). Issue #2606: Manager agent delegation fails on structured task objects. https://github.com/crewAIInc/crewAI/issues/2606

[67] CrewAI. (2024). Issue #1744: Tool input schema evolution breaks delegation. https://github.com/crewAIInc/crewAI/issues/1744

[68] CrewAI. (2024). Issue #1258: Output format duality - reasoning text vs structured JSON. https://github.com/crewAIInc/crewAI/issues/1258

[69] Semantic Kernel. (2025). Issue #10442: OpenAIAssistantAgent rejects Enum parameters. https://github.com/microsoft/semantic-kernel/issues/10442

[70] Semantic Kernel. (2024). Issue #5796: Gemini connector fails on Enum parameters. https://github.com/microsoft/semantic-kernel/issues/5796

[71] Semantic Kernel. (2024). Issue #8472: Non-deterministic function name format failures. https://github.com/microsoft/semantic-kernel/issues/8472

[72] AutoGPT. (2024). Issue #7028: 429 status code semantic ambiguity causes infinite retry. https://github.com/Significant-Gravitas/AutoGPT/issues/7028

[73] CrewAI. (2025). Issue #2729: OpenRouter capability detection fails for supported models. https://github.com/crewAIInc/crewAI/issues/2729

[74] Stack Overflow. (2023). OpenAI GPT-3 API error: This model's maximum context length is 4097 tokens. https://stackoverflow.com/questions/75396481/

[75] Stack Overflow. (2023). OpenAI API giving error: 429 Too Many Requests. https://stackoverflow.com/questions/75041580/

[76] Stack Overflow. (2023). ChatGPT randomly adding extra stuff to output despite asking for JSON. https://stackoverflow.com/questions/77606776/

[77] Stack Overflow. (2023). Extract JSON from GPT answer according to JSON schema. https://stackoverflow.com/questions/76553851/

[78] Stack Overflow. (2024). Azure GPT-4-Turbo JSON mode breaks after 1024 tokens. https://stackoverflow.com/questions/77944251/

[79] Stack Overflow. (2023). OpenAI API error 429: You exceeded your current quota. https://stackoverflow.com/questions/75898276/

[80] Stack Overflow. (2024). Handling Rate Limits with OpenAI API. https://stackoverflow.com/questions/78548625/

[81] Stack Overflow. (2023). OpenAI Authentication error: No API key provided. https://stackoverflow.com/questions/76796341/

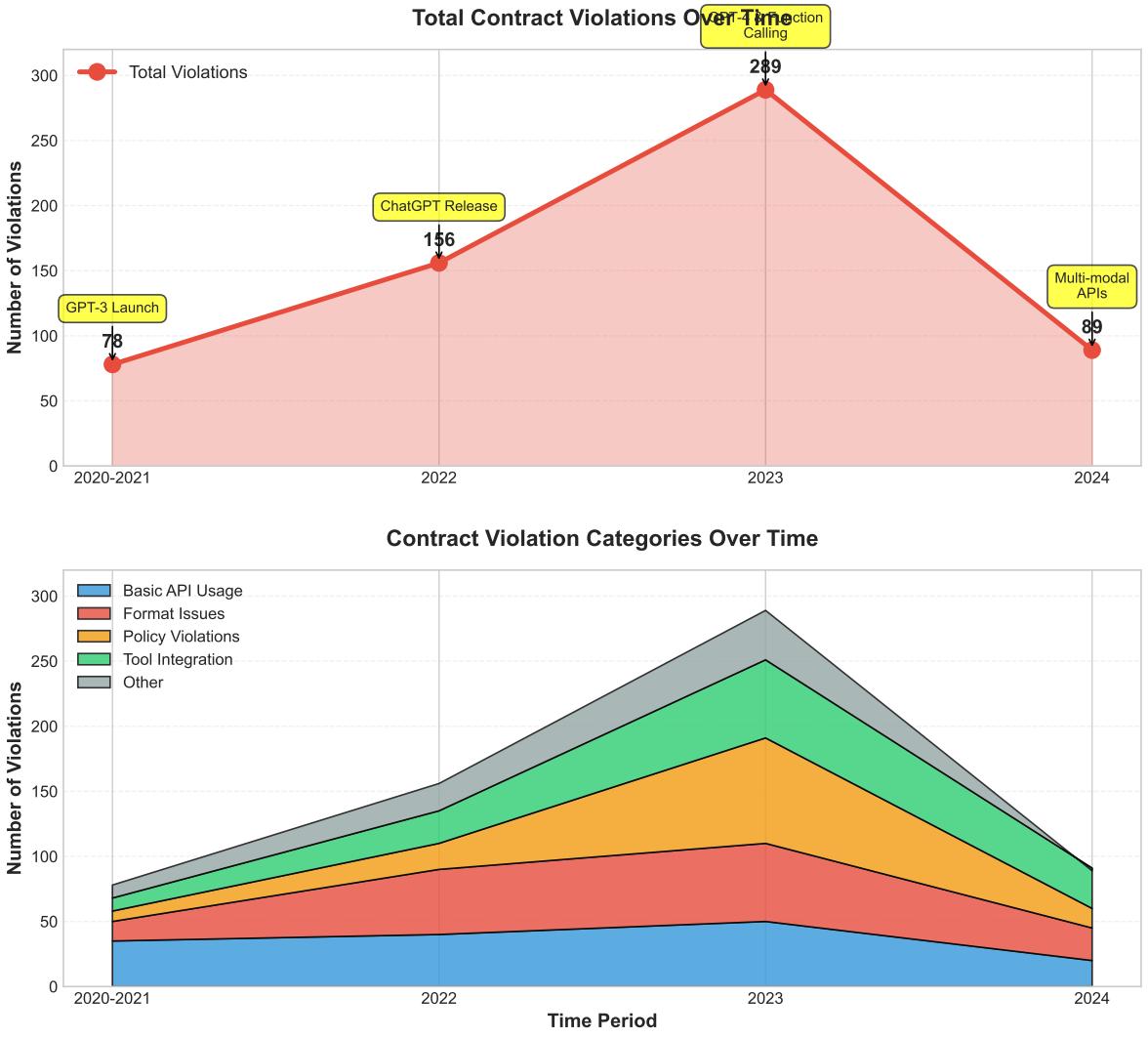
[82] Stack Overflow. (2024). OpenAI API error: Your authentication token is not from valid issuer. https://stackoverflow.com/questions/77896210/

[83] Stack Overflow. (2023). Authentication error with LangChain and OpenAI. https://stackoverflow.com/questions/76322025/

[84] Stack Overflow. (2023). OpenAI Chat Completions API error: messages is required property. https://stackoverflow.com/questions/75971578/

[85] Stack Overflow. (2024). How to format messages parameters in OpenAI GPT-4. https://stackoverflow.com/questions/78566807/

[86] Stack Overflow. (2023). OpenAI Stream response not working as expected. https://stackoverflow.com/questions/76125712/



Period	Total	Dominant Issue	New Categories
2020-2021	78	Basic API usage (45%)	Token limits
2022	156	Format issues (32%)	Chain orchestration
2023	289	Policy violations (28%)	Function calling
2024	89	Tool integration (35%)	Multi-modal contracts

Figure 7: Evolution of Contract Violations Over Time