Ensuring Trust in AI with Agent Contracts

A new approach to specification, verification and certification of agentic systems

relari

PUBLISHED BY Relari Inc.

Written by Pasquale Antonante and Yi Zhang.

Revision 2025-03-10
Cover design by Freepik

Contents

I AI Software Requirements

1. Introduction

In 2011, venture capitalist Marc Andreessen captured the tech world's imagination with his blog post titled "Why Software Is Eating the World."[1] Just a few years later, in 2017, NVIDIA's CEO Jensen Huang proclaimed that "Software Is Eating the World, but AI Is Going to Eat Software."[2] Both predictions have materialized with striking accuracy, but in ways few anticipated.

AI has fundamentally transformed the very fabric of software. We use AI to write software, and we embed AI into software itself. Software is now being developed faster than ever. At the same time, it is becoming more complex and more powerful. Natural language, images, and audio are no longer just data, they have become the primary way we interact with software.

The paradigm shift is profound: instead of explicit programming, we now craft prompts and let AI interpret our intent. This creates more powerful but less predictable systems. AI doesn't execute fixed instructions—it infers meaning, introducing a new dimension of uncertainty into software behavior.

This transformation presents a critical challenge: how do we verify systems whose core components operate on statistical inference rather than deterministic logic? When software behavior depends on how AI interprets natural language, how can we ensure consistent, correct outcomes?

Traditional software verification relied on systematic testing and deterministic execution. Engineers could trace every bug to specific code. Modern AI-powered systems operate differently—their behavior emerges from statistical models trained on vast datasets, not from explicit instructions. This fundamental shift makes correctness both harder to define and more difficult to enforce.

Today's software operates in a new reality of natural language inputs and probabilistic decision-making. Bugs no longer manifest simply as syntax errors or logic flaws—they emerge as subtle misalignments between human intent and machine interpretation. This creates an unprecedented verification crisis that demands new solutions.

1.1 A New Form of Bug: Prompt Misinterpretation

To every software engineer, the definition of a bug is clear: an unexpected behavior caused by a logical, syntactic, or semantic error. By this definition, LLM-powered systems might appear bug-free while still producing incorrect outputs.

Consider a question-answering system that, despite having all the necessary information to provide a correct response, it can still hallucinate an answer. Or imagine a customer support agent that processes a refund request but skips the eligibility check, approving refunds that should have been denied. In both cases, the system isn't failing due to a broken function or a missing logic gate: it's failing in a more subtle way that isn't captured by traditional definitions of a bug.

These failures stem from the LLM's misinterpretation of (or not adhering to) the prompt. The process of instructing an LLM is inherently lossy: a developer has a clear functionality in mind but must express it through natural language, which is ambiguous by nature. When the LLM then attempts to reconstruct the intended behavior from this prompt, sometimes it introduces errors by misinterpreting instructions or filling in missing details in unexpected ways.

1.1.1 A New Definition of Bug

Given this shift, we propose an updated definition of a bug:

> *A bug is any unintended, incorrect, or misaligned behavior in a software system, such as hallucinations, logical inconsistencies, or responses that violate expected intent or alignment.*

This broader definition acknowledges that LLM-powered software does not fail in the same way as traditional deterministic systems. Instead, its failures emerge from the model's probabilistic nature, its reliance on training data distributions, and the challenges of aligning it with developer and user expectations.

To understand why LLM-related bugs require a different approach, let's explore how they differ from traditional software bugs.

Soft Failures Instead of Hard Failures In conventional software, bugs often manifest as crashes, incorrect calculations, or broken logic—clear, deterministic failures that can be traced to a specific line of code. LLM-related bugs, on the other hand, tend to be soft failures. The system runs smoothly, and the output looks correct, but it might be subtly incorrect, misleading, or inappropriate. These failures can be harder to detect and measure.

Distributional Bugs Traditional bugs are often tied to a specific faulty instruction in the code. In contrast, LLM failures often arise from the statistical properties of the underlying model's training data. Issues like factual hallucinations, bias amplification, or inconsistent responses across similar inputs stem from the model's internal probability distribution rather than a specific logical flaw.

Contextual and Dynamic Bugs The correctness of an LLM's response is highly dependent on context. The same answer might be helpful in one scenario but completely inappropriate in another. For example, a customer support chatbot may handle 90% of cases correctly but fail catastrophically on edge cases—such as offering refunds where it shouldn't or giving legally incorrect advice. Unlike traditional software, where correctness is more rigidly defined, debugging LLM systems requires evaluating their responses across diverse contexts.

Non-Deterministic Bugs When a bug is found in traditional software, reproducing it is usually straightforward—providing the same input yields the same incorrect output (with some exceptions exceptions). With LLMs, the same input might generate different responses on different runs. This non-deterministic nature makes debugging significantly more challenging, as failures can appear sporadically and unpredictably.

Failures of Alignment Traditional software bugs typically occur when code does not meet explicit functional requirements. LLM failures, however, often stem from misalignment with implicit goals—such as ethical considerations, brand voice, or user expectations. These failures are harder to diagnose because they do not always violate a strict logical rule but rather a broader, sometimes subjective, intent.

Evaluation and Debugging Challenges Since LLM errors are often probabilistic and context-sensitive, traditional debugging techniques—like unit tests—are insufficient. Instead, evaluating and improving LLM performance requires new approaches, such as automated evaluation pipelines, adversarial testing, and continuous monitoring to track model quality, robustness, and failure rates.

1.2 What This Means for Software Development

The rise of LLM-powered applications changes how we think about debugging and quality control. Rather than treating issues as discrete bugs to be fixed, we must shift toward managing failure rates. Here's what that means in practice:

- Debugging becomes less about fixing broken code and more about refining the system's inputs and constraints. Since LLM outputs depend on prompts, improving performance often involves careful prompt engineering, dataset curation, and fine-tuning rather than modifying program logic.
- Monitoring and feedback loops take center stage. Given the probabilistic nature of LLMs, real-time monitoring and post-deployment evaluation become crucial to catching and mitigating failures dynamically.
- Human-in-the-loop interventions become essential. Unlike traditional software, where bugs are fixed in code, LLM-based applications often require human oversight—such as reinforcement learning with human feedback (RLHF) or manual correction workflows—to align behavior with expectations.
- New debugging tools are needed. Traditional debugging tools aren't designed for non-deterministic, distributional errors. Instead, new approaches like synthetic test cases, adversarial testing, and statistical failure analysis are required to improve reliability.

In essence, LLM-powered software redefines what it means for a system to be "correct." The challenge isn't just eliminating bugs in the traditional sense—it's about continuously steering a probabilistic model toward reliability, consistency, and alignment with human intent.

1.3 The Verification Crisis in Probabilistic Systems

Traditional software verification methods break down when applied to AI-driven systems due to several fundamental characteristics of probabilistic computing.

- the input space becomes effectively infinite as natural language prompts;
- the system's behavior emerges from complex statistical patterns rather than programmed logic;
- execution is non-deterministic, meaning that the same input may not always produce the same output.

The consequences can manifest in critical failures across AI applications. A customer support bot might refund a customer that is not eligible or a code generator might misinterpret the query and implement the wrong function. These failures stem not from implementation bugs in the classical sense, but from misalignments between the system's learned behavior and operational requirements.

1.4 Specification-Driven Verification

To address these verification challenges, we propose a specification-driven approach. This methodology enables systematic verification of AI-driven systems by establishing clear, measurable criteria for correct behavior while acknowledging the inherent uncertainty in their operation. Rather than attempting to verify every possible output—an intractable task given the infinite input space—specification-driven verification focuses on ensuring that system behaviors conform to well-defined constraints and properties across statistical distributions of inputs and outputs.

At its core, this approach requires the definition of *contracts*, the unit of testing. Contracts are specifications that capture both functional requirements and behavioral constraints. These specifications serve as a bridge between human intent and machine behavior, providing a framework for systematic testing and runtime monitoring. By expressing expected behaviors as formal contracts, we can automatically detect violations and measure the system's adherence to desired properties, even in the presence of non-deterministic outputs.

In the next section we will explore the role of contracts in the specification-driven verification framework.

2. Requirements Engineering

Before diving into agent contracts and how to verify AI systems, let's spend some time what requirements are and how to write them.

2.1 Functional Requirements

Every software project has requirements, though their initial form varies significantly across teams. While some organizations start with clear, well-defined detailed specifications of desired system features, others start with just a vague idea of the product they want to create (typical of startups). However, regardless of these different starting points, success depends on all team members ultimately developing a clear, unified vision of the project's deliverables.

There are two types of requirements: *functional* and *nonfunctional*[3]. Functional requirements define the specific behaviors and functions of a system, detailing what the system should do to meet the needs of its users. In contrast, nonfunctional requirements specify the quality attributes, system performance, and constraints under which the system must operate, such as security, usability, and reliability. In this document, we focus primarily on functional requirements, as they are essential for understanding the core capabilities that the software must deliver to fulfill stakeholder expectations, but we will also briefly touch on nonfunctional requirements.

2.2 Role of Functional Requirements

In today's sofware development, it is common to start with a rough idea of the product and then iterate on it based on feedback from the market/users. However, writing functional requirements before building a product offers several significant advantages:

Define the solution's boundaries functional requirements provide a clear roadmap for the development team (even if the team is small) and a clear expectation for the sales team, ensuring everyone understands what needs to be achieved and in which time horizon. Requirements can (and should) be updated as the product evolves in the shareholders mind.

Quality benchmarks Functional requirements serve as quality benchmarks (evaluation), allowing for objective measurement and testing of the software's behavior and the maturity of the features.

Risk management When a feature does not meet the functional requirements, it can be identified early, allowing for better risk management and problem-solving before development begins.

While user feedback is valuable, starting with well-defined functional requirements provides a solid foundation for development, reducing the risk of costly changes and ensuring the final product meets both user needs and business objectives. Functional requirements don't need to be perfect or exhaustive at the beginning, they need to be good enough to be used as a guide and evolve as the product evolves.

Examples of functional requirements:

- A user must be able to login to the system using their username and password.
- The system should allow users to search for products by name or category.
- When a user places an order, the system must generate an order confirmation email.
- The system should calculate and display the total price of a shopping cart.

2.3 Desirable Properties of Requirements

The following list of desirable properties of requirements can guide basic requirements analysis. The software engineer seeks to establish any of these properties that do not hold yet. Each requirement should:

- Relevant: represent true, actual stakeholder needs;
- Unambiguous: interpretable in only one way;
- Testable: meaning that compliance/satisfiability can be clearly demonstrated;

The overall collection of requirements should be:

- Complete: The requirements adequately address boundary conditions, exception conditions, and security needs;
- Concise: No extraneous content or redundancy;
- Internally Consistent: No requirement conflicts with any other;
- Externally Consistent: No requirement conflicts with external resources;

2.4 Defining Functional Requirements for AI Applications

When developing AI applications, how do we effectively define functional requirements? The approach needs to account for the unique challenges and complexities of AI systems while maintaining clarity and testability.

A functional requirement in AI systems consists of three core components:

- User Input/Intent: What the user provides or wants to accomplish
- Expected Output: The desired result or response
- Expected Behavior: How the system should process the input to produce the output

We refer to these functional requirements as *scenarios* in AI applications. This terminology emphasizes that each requirement represents a specific interaction between the user and the system. A scenario combines the user's input/intent with a set of *contracts* - formal specifications that encode the expected behavior and output for that interaction. The complete collection of documented scenarios form the system's *specification*.

Taking inspiration from the SOTIF (Safety Of The Intended Functionality) standard[4] used in autonomous driving, we can categorize scenarios in a way similar to how autonomous vehicle companies analyze their systems. The automotive industry recognized that traditional software development approaches were insufficient for guaranteeing the safety of AI-based systems like self-driving cars.

To understand the coverage and effectiveness of our specifications, we can categorize all possible scenarios into four zones, similar to SOTIF's approach:

1. **Known/Correct**: Scenarios we've specified where the system behaves as intended
2. **Known/Incorrect**: Scenarios we've specified where the system fails requirements
3. **Unknown/Correct**: Unspecified scenarios where the system behaves appropriately
4. **Unknown/Incorrect**: Unspecified scenarios where the system fails

A scenario is "known" if it exists in our specification and "unknown" if it doesn't. It is "correct" if the system produces the intended output and follows the expected behavior, "incorrect" otherwise.

The primary purpose of documenting scenarios is to systematically sample the space of all possible interactions, helping us understand and define the boundaries between these four zones, enabling the *verification* of the system.

Beyond defining requirements, we can use contracts to actively guide system behavior by establishing constraints that must be respected during execution. This process of enforcing and verifying adherence to contracts is called *certification*.

Certification serves two critical purposes:

1. **Runtime Detection**: identifies misbehaviors and provides explanations to help correct system behavior, reducing the size of the Known/Incorrect zone
2. **Formal Verification**: provides guarantees about system correctness, enabling the use of AI systems in more critical applications where reliability is paramount

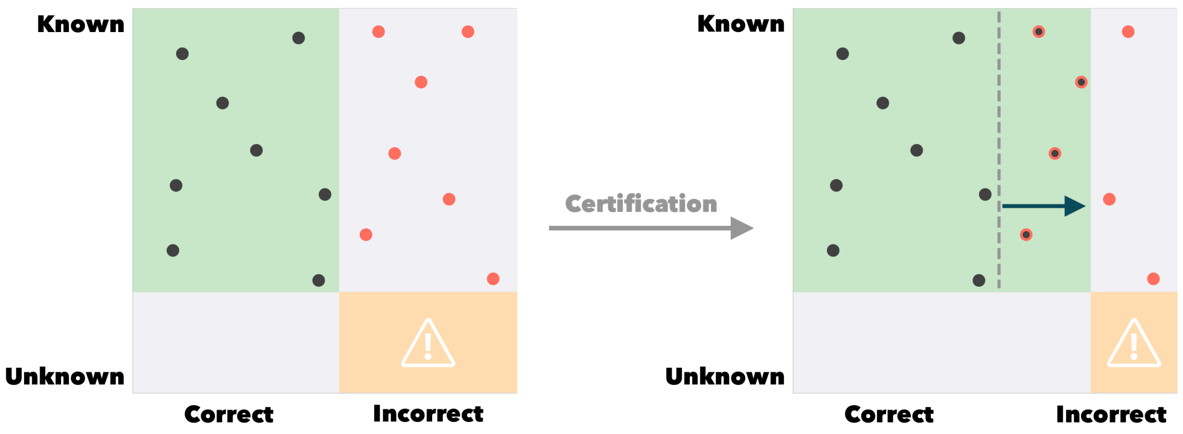


Figure 2.1: AI ODD and certification. The square represent the space of all possible scenarios, the circles represent a scenario in the specifications. The Known/Correct zone is the one where the system behaves as intended.

Now that we have a clear understanding of what requirements are and how to write them, let's dive into agent contracts and how to verify AI systems.

II

Agent Contracts

3. Agent Contracts: Defining System Requirements

In the specification-driven verification framework, contracts serve as the fundamental building blocks for defining system requirements. These contracts encode both the expected outputs and behaviors of the system under test, providing a formal mechanism for verification and validation.

3.1 Understanding Contract Components

An agent contract consists of three essential components that work together to define complete system behavior:

1. **Preconditions**: Conditions that must be satisfied before the contract can be evaluated
2. **Postconditions**: Conditions that must be satisfied after the contract execution
3. **Pathconditions**: Conditions that must be satisfied during the contract execution

To illustrate these concepts, consider a customer support agent handling refund requests. The expected output might be a simple confirmation message: *"Refund processed successfully."* However, this output alone doesn't guarantee correct behavior. The agent must also verify the customer's eligibility (pathcondition). Also, if the user provides the wrong information, the agent is not at fault for not being able to process the refund.

All the three components are important for flexibility, and should be considered when writing a contract. But not all of them are required for every contract, as we will see in the examples, we can omit any of them if they are not relevant to the contract.

3.1.1 Preconditions

Preconditions define the initial requirements that must be satisfied before a contract can be evaluated. They act as prerequisites that the caller (typically the user or another system) must fulfill before the agent can be expected to perform its task correctly.

For example, in a refund request scenario, typical preconditions include:

- The customer must provide a valid order ID,
- The customer must submit proof of damage or defect,

- The return request must be within the allowed time window.

If any precondition is not met, the agent cannot be held responsible for failing to complete the task. This is important because it establishes clear boundaries of responsibility between the caller and the agent. The agent's performance can only be fairly evaluated when all preconditions are satisfied.

3.1.2 Postconditions

Postconditions specify the required state of the system after execution completes. They act as a contract between the system and its users, guaranteeing certain properties of the output. Postconditions generally cover:

- **Quality metrics**: Measures of output correctness and completeness
- **Compliance**: The output must adhere to security, privacy and regulatory requirements

For example, in a refund processing system, key postconditions include:

- The refund amount must exactly match the original purchase price
- A unique refund confirmation number must be generated and provided to the customer
- All transaction details must be logged according to financial compliance standards
- The customer must receive clear confirmation of the refund status and next steps

3.1.3 Pathconditions

Pathconditions define the required sequence of actions, decisions, and state changes that must occur during system execution. For AI systems in particular, verifying this execution path is critical - it's not enough to just get the right output, the system must follow the correct process to get there. Key aspects that pathconditions typically verify include:

- **Tool usage**: Which tools must be called and in what order
- **Decision making**: What choices the system should make
- **State management**: How the system's state should evolve
- **Error handling**: How the system should detect and respond to problems

For example, in a refund processing scenario, pathconditions would verify that:

- The system first validates customer eligibility by checking return policy rules
- The refund amount is calculated correctly based on the original purchase price
- All steps of the transaction are logged for audit purposes
- The customer is kept informed of the refund status throughout the process

3.2 Requirements Levels

Requirements in contracts can be categorized into two levels: **MUST** and **SHOULD**. MUST requirements represent mandatory conditions that the system absolutely has to fulfill - failing to meet a MUST requirement constitutes a violation of the contract. For example, "The system MUST verify the customer's identity" means the system has failed if it skips this verification step.

SHOULD requirements, on the other hand, act as quality measures and guidelines for system improvement. While not strictly required, they help evaluate if the system is getting better within the boundaries established by the MUST requirements. For instance, "The system SHOULD respond within 2 seconds" indicates a desirable performance target, but exceeding this time doesn't necessarily mean the system is broken.

This two-level approach allows contracts to clearly distinguish between essential requirements that ensure correct behavior and aspirational goals that drive continuous improvement. Teams can focus first on meeting all MUST requirements to guarantee basic functionality, then work on optimizing SHOULD requirements to enhance the system's overall quality.

3.3 Why Contracts Matter

Contracts give us a powerful way to make sure AI systems work correctly. Unlike traditional testing that only looks at final outputs, contracts check the entire process from start to finish.

Think about a customer service AI that processes refunds. Traditional testing might only verify that the refund was issued correctly. But what if the AI skipped important steps along the way? For example, it might issue a refund without checking if the customer was eligible first. The output looks correct (the customer got their refund), but the process was wrong.

Contracts solve this problem by checking three things:

1. **Starting conditions**: Was the request valid to begin with?
2. **Process steps**: Did the AI follow the right procedure?
3. **Final results**: Was the output correct?

Here's a real-world example: When a customer asks "Can I get a refund for my broken headphones?", a contract can verify that:

- The AI checked if the purchase was within the return period (a required step)
- The AI asked for proof of purchase (another required step)
- The AI issued the correct refund amount (the final output)

A possible example of such contract is shown in Contract 3.1.

Contracts also help us understand who's at fault when things go wrong. If a customer asks for a refund but doesn't provide an order number, the AI can't be blamed for failing to process the refund. The contract's starting conditions weren't met.

This becomes especially important for systems like chatbots that handle all kinds of user inputs. When a user says something unexpected like "My elephant is purple," the contract can recognize that no valid request was made, rather than treating the AI's inability to process a refund as a failure.

By checking the whole process—not just the end result—contracts give us confidence that AI systems are doing the right things for the right reasons, even in complex situations where simply looking at the output isn't enough.

3.3.1 Open-source package

Relari released an open-source package for writing and verifying contracts in natural language. The package provides a simple yet powerful interface for defining contracts, running verifications, and analyzing results. It supports both offline verification through test scenarios and runtime certification of live systems.

Key features include:

- Natural language contract definitions using YAML or Python
- Flexible verification strategies for different use cases

Refund

Scenario

Refund request for order Order n. 114-9587331-019785

Precondition

- The user **MUST** provide a valid order ID
- The user **MUST** provide a picture of the damaged product

Pathcondition

- The system **MUST** check if the user is eligible for a refund
- The system **MUST** check if the user has a valid order ID and documentation
- The system **MUST** update the database to reflect the refund

Postcondition

The system **MUST** generate a refund receipt and **MUST** send it to the user

Contract 3.1: Example of contract for a customer support agent processing refund requests.

- Integration with popular LLM frameworks and APIs
- Comprehensive test reporting and analytics
- Runtime monitoring and certification capabilities

The package is available on GitHub: https://github.com/relari-ai/agent-contracts. This open-source tool makes it easier for teams to start implementing contract-based verification in their AI systems without having to build verification infrastructure from scratch.

3.4 Contract Language and Specification

Contracts can be written in different ways, from everyday language to specialized code. Natural language contracts are easy to understand and offer major benefits for teams building AI systems. For example, a customer service manager can read and approve a contract that states *"The system must verify the customer's identity before processing any refund request"* without needing technical expertise. This clarity helps teams align on expectations and reduces misunderstandings between business and technical stakeholders. When a new team member joins, they can quickly understand system requirements by reading these plain-language contracts instead of deciphering complex code.

On the other hand, more technical approaches have their place too. For simple checks, code might be clearer: `if refund_amount > order_total: raise Error`. Between these extremes, domain-specific languages offer a middle path. A financial system might use a specialized language for transaction rules that both business analysts and developers can work with. The best approach is often to mix these methods based on who needs to understand each part of the contract. Critical security requirements might use precise formal language, while customer-facing features use natural language that product managers can easily review. This balanced approach ensures everyone stays on the same page while still enabling automated verification of system behavior.

3.5 Best practices

3.5.1 Writing Effective Contracts

When writing agent contracts, it's crucial to follow several key principles that ensure they provide meaningful verification while remaining practical to implement and maintain:

1. **Clearly define the scope of each condition**
   - Be specific about what each contract is intended to verify
   - Establish clear boundaries between different contracts
   - Use descriptive names that communicate the contract's purpose
   - Document the intent behind each contract for future reference
2. **Ensure conditions are measurable and verifiable**
   - Avoid subjective criteria that cannot be consistently evaluated
   - Define concrete success criteria for each condition
   - Leverage deterministic verification methods whenever possible
3. **Consider both functional and non-functional requirements**
   - Include quality metrics when relevant
   - Address security and privacy requirements explicitly
   - Incorporate accessibility and usability criteria when appropriate
4. **Account for edge cases and error conditions**
   - Identify boundary conditions where behavior might change
   - Define expected behavior for invalid or unexpected inputs
   - Specify appropriate error handling and recovery mechanisms
   - Include contracts that explicitly test failure modes and recovery paths
5. **Maintain consistency across related contracts**
   - Ensure terminology is used consistently across all contracts
   - Avoid contradictory conditions between different contracts
   - Create hierarchies or groupings of related contracts
   - Establish naming conventions and structural patterns
6. **Balance precision with adaptability**
   - Make contracts specific enough to catch real issues but not so rigid they break with minor, acceptable variations
   - Design contracts that can accommodate expected evolution of the system
   - Document changes to contracts

The combination of preconditions, postconditions, and pathconditions provides a comprehensive framework for specifying and verifying system behavior. When crafted thoughtfully, these contracts enable both rigorous automated testing during development and continuous runtime verification of AI-driven systems in production. Effective contracts serve as both documentation of system requirements and executable specifications that can detect deviations from expected behavior.

3.5.2 Contracts Lifecycle

In the rapidly evolving landscape of AI systems, traditional development approaches often struggle to keep pace[1]. The speed at which AI capabilities advance demands an agile approach to both development and verification. Contract-based verification embraces this philosophy rather than becoming a bottleneck.

When implementing new AI features, teams should develop contracts incrementally alongside the functionality itself. Rather than attempting to create a comprehensive specification upfront, focus on adding a small set of targeted contracts for each new feature. During active development, verification can be limited to only those contracts relevant to the feature under development, streamlining the testing process and providing quick feedback loops.

Once a feature passes verification, its contracts should be integrated into the broader regression testing suite. This ensures that future development doesn't inadvertently break previously verified functionality while keeping the active development process lightweight and focused.

Customer involvement is crucial throughout this lifecycle. Since many stakeholders lack technical expertise in AI systems or programming, contracts must support natural language specifications. This accessibility allows product owners, domain experts, and end users to directly participate in defining and reviewing contracts without requiring visibility into the underlying code or AI architecture.

For example, a customer service team implementing a new refund processing feature might start with basic contracts covering the happy path scenarios. As development progresses, they can add contracts for edge cases and error handling. Throughout this process, business stakeholders can review the natural language contract specifications to ensure they accurately capture business rules and customer expectations.

This agile approach to contract management balances thoroughness with practicality. By focusing verification efforts where they matter most at each stage of development, teams can maintain development velocity while still building a comprehensive verification framework over time. The result is a system that evolves quickly while maintaining reliability and alignment with customer needs.

[1]For example writing very precise sequence of tool calls is hard, time-consuming and it's error prone.

4. Verification

The aim of this chapter is to explore how we can use contracts, a language to define agents' behavior, to verify that AI agents adhere to their expected behavior.

4.1 Verifying Contracts with Traces

When we execute our AI agents, we can produce detailed execution reports through what is called observability. We need to observe the system's entire journey from input to output, capturing all the decisions and actions it takes along the way.

4.1.1 Telemetry

Telemetry refers to the automated collection, transmission, and measurement of data from remote sources. In the context of AI systems, telemetry involves gathering detailed information about an agent's operations, performance, and behavior during execution through **traces** and **spans**.

A trace represents the complete journey of a request through the system, from start to finish. Each trace is made up of multiple spans - individual operations or units of work that occur during the request's lifecycle. For example, when an AI agent processes a user query, the overall trace might include spans for:

- Parsing the initial input
- Making API calls to external services
- Running inference on an LLM
- Generating the final response

Spans contain rich metadata about each operation, including:

- Start and end timestamps
- The operation name and type
- Any errors or exceptions that occurred
- Custom attributes and tags

Open standards like OpenTelemetry[5] have emerged to standardize this process across different systems and platforms. OpenTelemetry provides a unified set of APIs, libraries, and agents for collecting traces, metrics, and logs from applications. This standardization makes it easier to implement consistent observability practices regardless of the underlying infrastructure or programming language.

By leveraging these standards, developers can:

- Track every function call and decision point within the AI system through detailed spans
- Measure performance metrics like latency and resource usage across traces
- Capture contextual information about the environment in span attributes
- Correlate events across distributed systems by following trace contexts

This comprehensive trace and span collection forms the foundation for effective contract verification, providing the raw material needed to reconstruct and analyze an AI agent's behavior.

4.1.2 State-Action graph

While traces and spans provide incredibly detailed information about system behavior, this granularity can actually make validation more challenging. A single trace might contain hundreds of spans capturing every API call, database query, and computation. This level of detail, while valuable for debugging and performance analysis, can obscure the higher-level behavioral patterns we need to verify against our contracts.

To address this, we transform the detailed trace data into a more abstract state-action graph representation. This transformation distills the essential decision points and actions from the raw telemetry data, filtering out implementation details that aren't relevant for contract verification. For example, a function parsing JSON produced by an LLM might be condensed into the LLM generation call, as the LLM's output is the actual JSON parsed.

A **state-action graph** is a structured representation of an AI system's behavior during execution. It models the system's journey as a series of **states** (points where the system has a particular configuration or is making a decision) connected by **actions** (operations that transition the system from one state to another).

In this graph:

- Nodes represent states of the system, capturing relevant context or information available at that point (*e.g.,* the agent name, the a specific macro operation like vector search, etc.)
- Edges represent actions taken by the system, such as API calls, tool usage, or internal computations
- The path through the graph shows the sequence of decisions and actions taken during execution

State-action graphs provide several advantages for verification:

- They make the system's decision-making process explicit and inspectable
- They allow for identifying critical decision points where contracts must be enforced
- They enable reasoning about alternative paths the system could have taken
- They provide a concrete artifact that can be analyzed against contract specifications

By transforming raw telemetry data into a state-action graph, we create a structured view of system behavior that's amenable to formal verification against our contract requirements.

4.1.3 Contract Verification

With the trace data collected and transformed into a state-action graph, we can now perform the actual contract verification. This process involves systematically analyzing the graph against our contract specifications to ensure the AI system behaved correctly.

The verification process examines three key aspects of the contract:

1. **Preconditions**: We analyze the initial state nodes to verify that all required preconditions were met before the system began processing. For example, in a refund scenario, we would check that the customer provided a valid order ID and any required documentation.
2. **Pathconditions**: By walking through the state-action graph, we verify that the system followed all required steps in the correct order. Each action edge in the graph should correspond to a required operation or decision point specified in the contract. For instance, we can verify that a customer service AI checked the return policy before processing a refund by looking for the appropriate Knowledge Agent interaction in the graph.
3. **Postconditions**: Finally, we examine the terminal states to confirm that the system's output satisfies all postconditions. This includes both the correctness of the result and any required notifications or follow-up actions.

The verification process can detect various types of contract violations:
- Missing steps: Required actions that don't appear in the graph
- Incorrect ordering: Actions performed in the wrong sequence
- Invalid transitions: State changes that violate contract constraints
- Incomplete results: Outputs that don't meet all postconditions

For example, consider our customer refund contract from earlier. The state-action graph would let us verify that:
- The system properly validated the order ID using the Database Agent
- The Knowledge Agent was consulted to check refund eligibility
- The customer was offered both refund options (store credit or original payment method)
- A valid refund ID was generated and communicated to the user

This systematic verification approach ensures that AI systems aren't just producing correct outputs by chance, but are following the specific processes and procedures we've defined as necessary for proper operation. The state-action graph provides the concrete evidence needed to certify that each contract requirement was satisfied.

4.2 User Simulation

Verifying chatbots presents unique challenges since testing requires simulating entire multi-turn conversations to properly evaluate the system's behavior. To address this complexity, contracts can be effectively combined with a user simulator, here called a **testbot**.

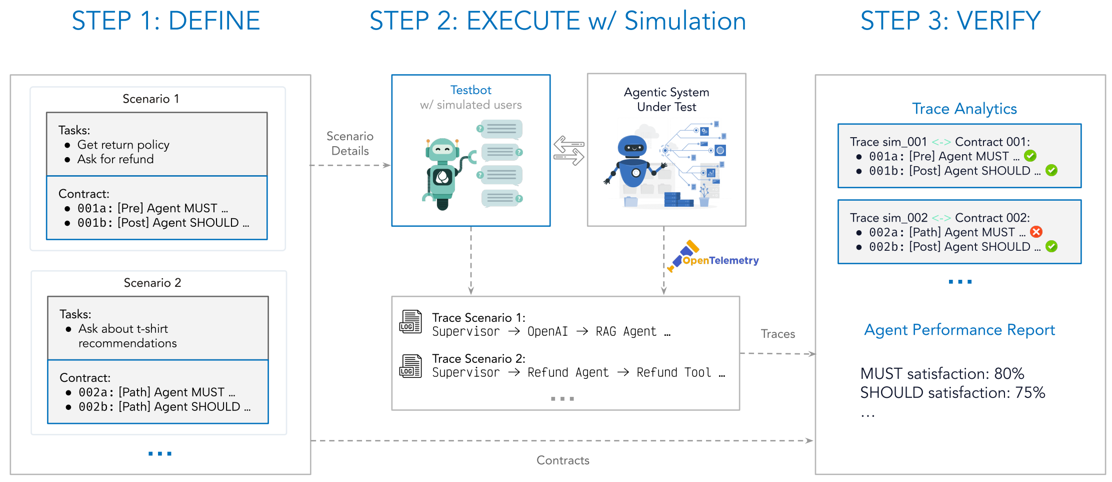

STEP 1: DEFINE
STEP 2: EXECUTE w/ Simulation
STEP 3: VERIFY
Scenario 1
Tasks:
• Get return policy
• Ask for refund
Contract:
• 001a: [Pre] Agent MUST …
• 001b: [Post] Agent SHOULD …
Scenario 2
Tasks:
• Ask about t-shirt recommendations
Contract:
• 002a: [Path] Agent MUST …
• 002b: [Path] Agent SHOULD …
…
Scenario Details
Testbot w/ simulated users
Agentic System Under Test
OpenTelemetry
Trace Scenario 1:
Supervisor → OpenAI → RAG Agent …
Trace Scenario 2:
Supervisor → Refund Agent → Refund Tool …
…
Traces
Contracts
Trace Analytics
Trace sim_001 <-> Contract 001:
• 001a: [Pre] Agent MUST … ✓
• 001b: [Post] Agent SHOULD … ✓
Trace sim_002 <-> Contract 002:
• 002a: [Path] Agent MUST … ✗
• 002b: [Post] Agent SHOULD … ✓
…
Agent Performance Report
MUST satisfaction: 80%
SHOULD satisfaction: 75%
…

Figure 4.1: Verification workflow for a conversation agent with simulation testbot.

The simplest form of a testbot is a predefined set of questions and expected responses. This approach, while limited in flexibility, provides a consistent baseline for testing basic functionality. For example, a testbot might systematically work through a fixed script of common customer service scenarios, checking that the chatbot responds appropriately at each step.

However, more sophisticated testbots can act as true user simulators, dynamically generating conversations based on configurable parameters and leveraging contract specifications to target specific test scenarios:

1. **Intents**: The testbot can be configured with a set of high-level goals or intents that real users might have, such as:
   - Requesting a refund
   - Resetting a password
   - Tracking an order
   - Updating account information
2. **Success Criteria**: For each intent, the testbot defines specific conditions that indicate successful completion. For instance, a refund request might require:
   - Verification of order details
   - Confirmation of refund eligibility
   - Receipt of a refund confirmation number
3. **Interaction Traits**: To test system robustness, testbots can simulate different user interaction patterns and behaviors:
   - Reasonable users who provide complete and accurate information (typical case)
   - Terse users who provide minimal information or one-word responses
   - Distracted users who switch topics or forget previous context
   - Angry or anxious users who express frustration or urgency

By understanding the contract requirements, testbots can strategically probe system behavior by:

- Deliberately violating preconditions to verify proper error handling
- Testing boundary conditions specified in the contract
- Exploring alternative paths through the contract's decision tree
- Validating that each required step in the contract is properly executed

This simulation-based approach enables not just basic functional testing, but also helps verify how the system handles edge cases and unusual interaction patterns. For example, a testbot simulating a confused user might switch between multiple intents mid-conversation, testing the system's ability to maintain context and guide the interaction back on track while still fulfilling contract obligations.

The combination of testbots and contracts enables both comprehensive testing at scale and verification of system consistency. By running the same scenarios multiple times with different personality traits, we can measure how reliably the system achieves its goals across varying interaction patterns. This approach helps identify areas where the system's behavior might be inconsistent or where certain user types consistently trigger contract violations.

4.3 Verifying Probabilistic Systems

A fundamental challenge when working with probabilistic AI systems is their inherent variability: given the same input, they may produce different outputs across multiple runs. This non-deterministic behavior makes traditional verification approaches insufficient for many applications, as one single passed test does not guarantee that the system is correct all the time. The challenge becomes even more complex when dealing with chatbots and conversational AI, where verification requires simulating entire multi-turn conversations to properly evaluate the system's behavior.

However, contracts (especially when combined with testbots) provide an effective solution to this verification challenge by building on the verification approaches discussed earlier:

1. **Pre/Post-condition Verification**: Contracts can specify acceptable ranges for outputs rather than exact matches. For example, a refund contract might require that the refund amount falls within a valid range and follows business rules, without mandating specific response text.
2. **Systematic Testing**: The testbot approach allows us to systematically verify these contracts across many conversation paths. By simulating different user personalities and interaction patterns, we can build confidence that the system maintains consistency where it matters while allowing for natural variation in exact responses.

Through repeated verification of the same contracts across similar or identical inputs, we can:

- Measure the frequency of different response patterns
- Establish statistical confidence intervals for system behavior
- Identify areas where variation exceeds acceptable bounds
- Ensure that any inconsistency remains within the limits defined by our specifications
- Pinpoint specific components or interactions that require improvement
- Prioritize development efforts based on statistical impact

The statistical nature of this testing approach provides valuable insights for system designers. By analyzing patterns in contract violations and performance variations, developers can identify which components most urgently need attention and what specific aspects of the system behavior need refinement.

Additionally, by categorizing requirements into MUST (core functional requirements) and SHOULD (quality-of-service requirements) levels, we can track system improvement over time. This allows us to:

- Measure baseline compliance with essential MUST requirements
- Track progress in meeting SHOULD requirements across development cycles
- Quantify quality improvements through statistical metrics
- Set clear improvement targets for subsequent iterations

This comprehensive approach transforms probabilistic behavior from a verification challenge into a measurable and manageable aspect of system performance. Rather than requiring perfect reproducibility, we can define and verify appropriate bounds of variation while still ensuring that core business logic and user experience requirements are consistently met. The combination of well-defined contracts and systematic testbot verification provides a framework for building confidence in AI systems despite their inherent variability, while also guiding continuous improvement efforts.

5. Certification

5.1 Certification

The development of agentic AI systems involves a fundamental tradeoff between generality and control. General agents, which utilize broad an generic LLM prompts, can handle a wide range of tasks but operate with relatively low reliability as we rely of the ability of the LLM fill the gaps in the prompt to perform the task. On the opposite end, agents with specific and detailed prompts achieve high reliability but can only handle a small subset of tasks (because of the overspecialization of the prompt).

A balanced approach emerges through trustworthy agents that combine general LLM prompts with agent contracts, maintaining the ability to handle many tasks while achieving high reliability. This approach leverages the the runtime certification to ensure that the agent performs the task as specified by the contract correctly.

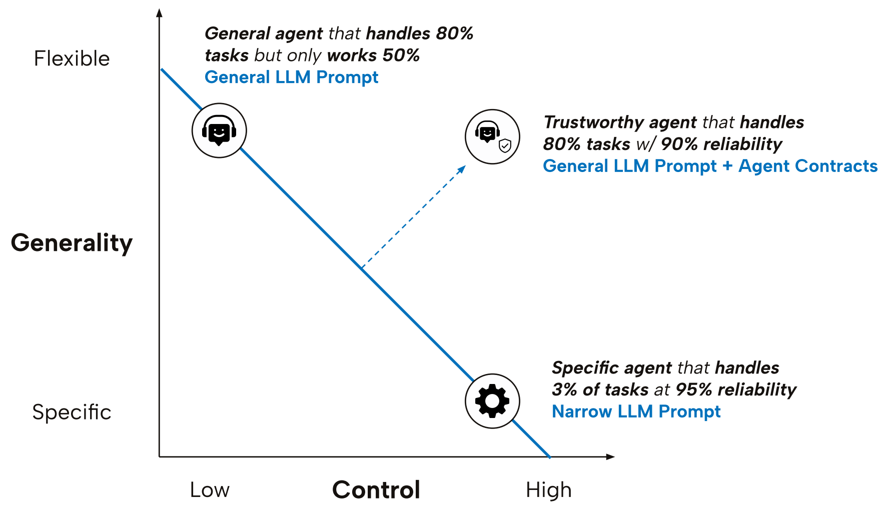

Flexible
Generality
Specific
Low
Control
High
General agent that handles 80% tasks but only works 50%
General LLM Prompt
Trustworthy agent that handles 80% tasks w/ 90% reliability
General LLM Prompt + Agent Contracts
Specific agent that handles 3% of tasks at 95% reliability
Narrow LLM Prompt

Figure 5.1: Tradeoff between generality and control in agentic AI systems.

5.2 Understanding Certification

Certification in AI systems refers to the process of attaching some form of proof, or certificates, to system outputs that verify their correctness according to predefined contracts. A certificate serves as a guarantee that the output satisfies all specified requirements - not just in terms of the final result, but also in how that result was obtained.

For example, when a customer service AI processes a refund request, the certificate would verify that:

- All preconditions were met (e.g., valid order ID, customer documentation)
- The correct process was followed (e.g., eligibility checks performed, database updated)
- The final output meets all postconditions (e.g., refund amount matches order, proper documentation generated)

This certification process transforms what would otherwise be a "black box" AI output into a verifiable result with a clear chain of evidence demonstrating its correctness. The certificate itself contains the proof that each contract condition was satisfied, and the evidence that the system followed the correct process, allowing any party to independently verify the system's behavior.

The power of certification lies in its ability to provide objective, measurable guarantees about system behavior. Rather than simply trusting that an AI system "probably" did the right thing, certification gives us concrete evidence that it definitively followed all required steps and produced a valid result according to our specifications.

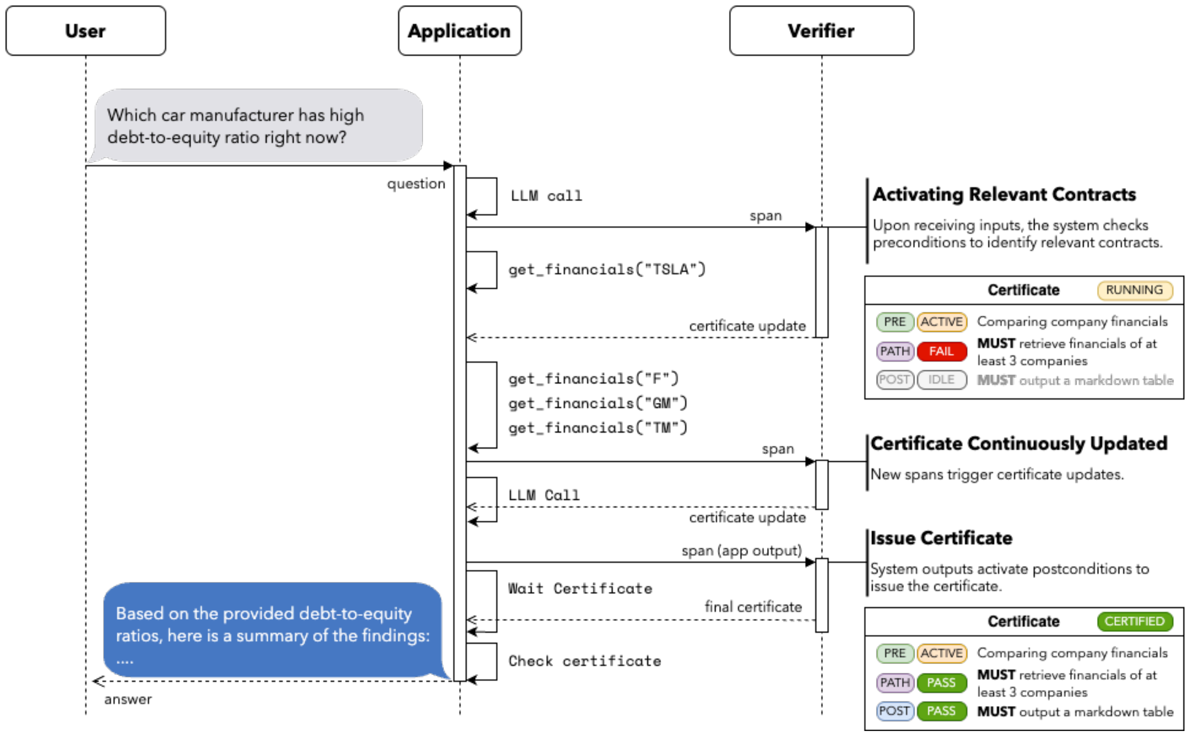

User
Application
Verifier
Which car manufacturer has high debt-to-equity ratio right now?
question
LLM call
span
get_financials("TSLA")
certificate update
get_financials("F")
get_financials("GM")
get_financials("TM")
span
LLM Call
certificate update
span (app output)
Wait Certificate
final certificate
Check certificate
Based on the provided debt-to-equity ratios, here is a summary of the findings: ....
answer
Activating Relevant Contracts
Upon receiving inputs, the system checks preconditions to identify relevant contracts.
Certificate RUNNING
PRE ACTIVE Comparing company financials
PATH FAIL MUST retrieve financials of at least 3 companies
POST IDLE MUST output a markdown table
Certificate Continuously Updated
New spans trigger certificate updates.
Issue Certificate
System outputs activate postconditions to issue the certificate.
Certificate CERTIFIED
PRE ACTIVE Comparing company financials
PATH PASS MUST retrieve financials of at least 3 companies
POST PASS MUST output a markdown table

Figure 5.2: The certification sequence diagram.

5.3 Benefits of Certification

Certification provides dual benefits for both consumers and developers of AI systems. When the system operates correctly, certification acts as a formal guarantee, giving consumers confidence that not only is the final output correct, but that it was produced through an appropriate and verified process. This is particularly valuable in high-stakes domains where trust and reliability are paramount. Conversely, when failures occur, the certification system serves as a diagnostic tool by precisely identifying which contracts were violated. This granular feedback helps developers understand whether the failure occurred in the preconditions (invalid inputs), postconditions (incorrect outputs), or pathconditions (improper process), enabling targeted improvements to either the system design or the recovery mechanisms. For example, if a customer service agent violates a pathcondition by skipping the eligibility check, the certification system can catch this error before the refund is processed, allowing for immediate correction rather than discovering the mistake after the fact.

5.4 Certification in Practice

Even though the contracts used for verification can be directly applied to certification, often verification contracts are written with a high specificity, as they are scenario-based. In verification contracts, we can be very specific about the input or user intent, for instance, we might write a contract that verifies the correct handling of order `ORDER_12345`, expecting all tool calls and responses to reference that exact order. This specificity allows us to write precise requirements and thoroughly test particular scenarios but it makes the contracts less general and harder to apply to new scenarios.

During runtime certification, we need more generalized contracts that can handle any order number or user input. To achieve this, we can either use automated methods to generalize our verification contracts, or we can directly write more general contracts specifically designed for certification, which we call *certification contracts*.

In these certification contracts, the preconditions serve to define broader scenarios (like "user requests a refund for an order") rather than specific instances (like "user requests a refund for order `ORDER_12345`"). The postconditions and pathconditions then specify the expected outputs and processes that should occur for any valid instance of that scenario, providing a framework for runtime validation of the system's behavior regardless of the specific inputs received.

Example of a certification contract:

The certification contracts like the one above are used to certify the system's behavior during runtime through a dedicated certification engine. The certification engine receives a continuous stream of telemetry spans from the system being monitored, for example collected using OpenTelemetry[5].

As each new trace begins, the engine first evaluates its initial spans against the preconditions of all available contracts to identify which contracts are relevant and should be actively monitored for this interaction. For each subsequent span received within an active trace, the engine updates its state tracking of the pathconditions for all matched contracts. This allows real-time monitoring of whether the system is following the expected sequence of steps and behaviors defined in the contracts. Once a trace completes its execution, the engine performs a final evaluation of all postconditions for the matched

Refund certificate (simple)

Precondition

The user requests a refund

Pathcondition

- The system checks if the user is eligible for a refund
- The system checks if the user has a valid order ID and documentation
- The system updates the database to reflect the refund

Postcondition

The system generates a refund receipt and sends it to the user

Contract 5.2: Example of a certification contract.

contracts. At this point, with the complete trace evaluated against preconditions, pathconditions, and postconditions, the engine can issue the final certificate.

If all requirements specified in a contract were satisfied throughout the trace's lifecycle, the engine generates a positive certificate - a formal verification that the interaction was fully compliant. However, if any requirement was violated at any stage, the engine produces a detailed negative certificate that precisely identifies which conditions failed, when they failed, and how they deviated from expectations. This granular feedback enables rapid diagnosis and correction of issues in the system's behavior.

6. Case Study

In this chapter we will see how we can use agents contracts to verify agentic systems.

6.1 Customer Support Agent

One of the most successful application of agentic systems is in the customer support domain. In this section we will use a customer support agent integrated with an online store's database, designed to provide comprehensive assistance to customers. The system has direct access to product information, inventory levels, order histories, and customer accounts, enabling it to handle a wide range of customer inquiries effectively. The agent can help customers find specific products, provide detailed product information, process new orders, and handle post-purchase support including refund requests. This deep integration with the store's systems allows the agent to provide real-time, accurate responses while ensuring all transactions and customer interactions follow established business rules and compliance requirements.

The vast domain in which customer support agents operate is well suited for agentic systems but, as we noticed in Chapter 2. also poses challenges, in particular regarding the verification.

6.1.1 How it works

Figure 6.1 shows the structure of the multi-agent system we are going to verify. There are four agents, each responsible for a different aspect of the customer support process:

1. *Supervisor agent* coordinates the actions of sub-agents that handle different aspects of the customer support process
2. *Knowledge Agent* queries and summarizes information in the company's knowledge base (Retrieval Augmented Generation, RAG)
3. *Database Agent* interacts with the company's database and performs transactions
4. *Customer Agent* interacts with the customer.

The architecture ensures that the agent has everything it needs to perform its task.

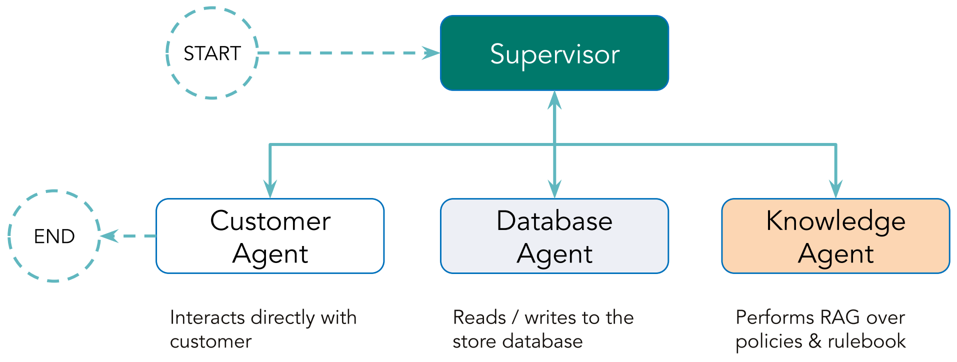

START
Supervisor
END
Customer Agent
Database Agent
Knowledge Agent
Interacts directly with customer
Reads / writes to the store database
Performs RAG over policies & rulebook

Figure 6.1: The structure of the multi-agent customer support system. The agents are connected by edges that represent the flow of information.

6.1.2 Contracts

One of the primary functions of a customer support agent is processing refund requests. We can use agent contracts to verify this critical use case.

To ensure refunds are handled properly, we must certify that the process follows all company policies. This involves several key steps: validating the customer's refund request with proper documentation, verifying eligibility based on return policies, and correctly processing the refund transaction.

To systematically verify this complex workflow, we can create a verification contract that specifies the exact requirements and steps that must be followed. This contract will help us ensure the refund process maintains consistency and compliance across all customer interactions. An example of such contract is shown in Contract 6.3.

6.1.3 Offline Verification

We can use a testbot (described in Section 4.2) to simulate customer interactions and generate a complete execution trace. Figure Figure 6.2 shows a conversation between a user (the testbot) and the customer support agent.

Refund scenario

Scenario

Refund request for order Order n. 114-9587331-019785

Precondition

- The customer **MUST** provide a valid order ID
- The customer **MUST** provide a picture of the damaged product

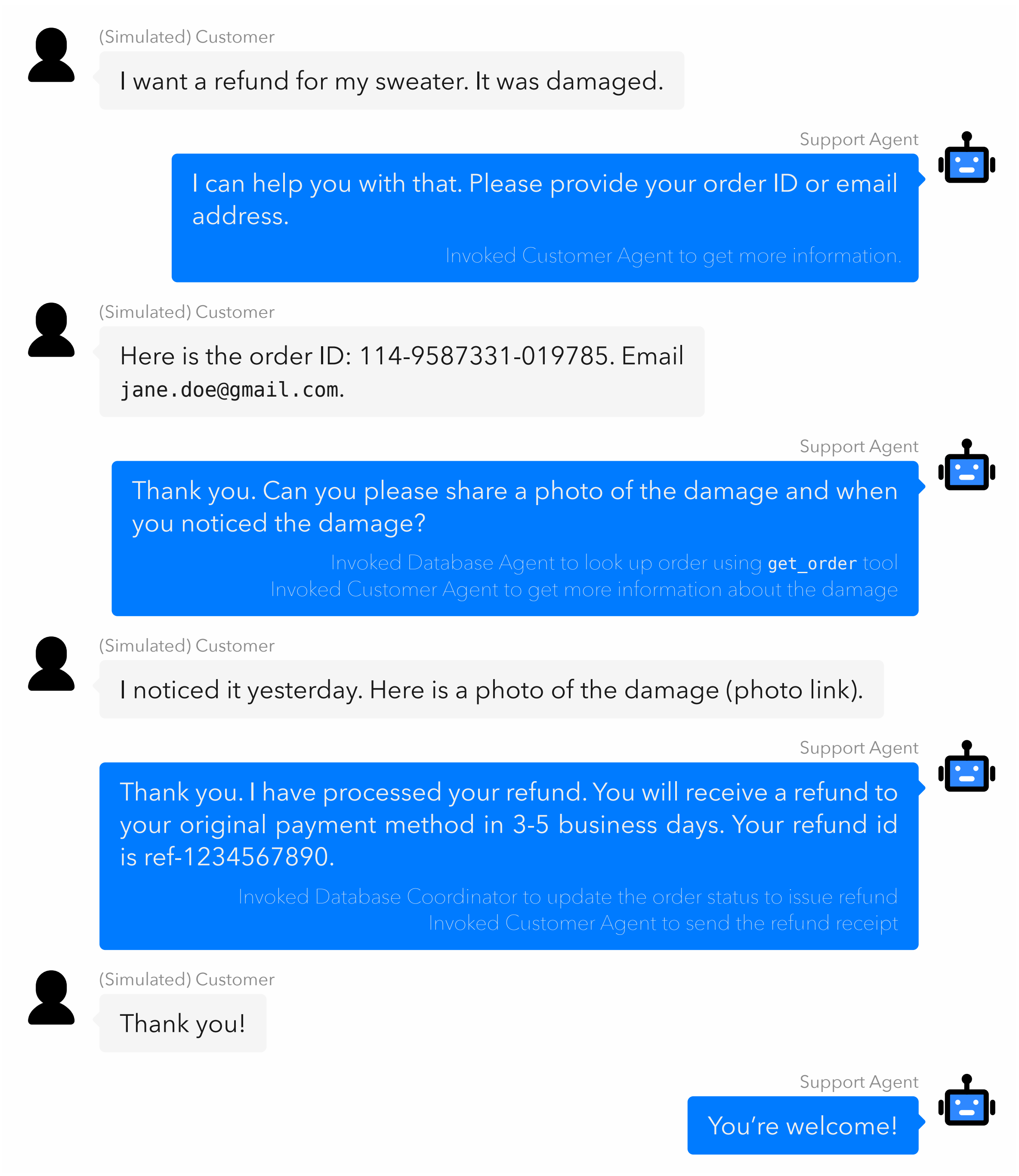
Pathcondition

- The system **MUST** check if the customer has a valid order ID using the Database Agent
- The system **MUST** check if the customer is eligible for a refund checking the return policy using the Knowledge Agent
- The system **MUST** verify if the customer wants refund to original payment method or store credit
- The system **MUST** perform the refund transaction using the Database Agent using the `process_refund` tool.

Postcondition

The system **MUST** generate a refund id and **MUST** send it to the user

Contract 6.3: A contract for a customer support agent processing a specific refund requests.



(Simulated) Customer

I want a refund for my sweater. It was damaged.

Support Agent

I can help you with that. Please provide your order ID or email address.

Invoked Customer Agent to get more information.

(Simulated) Customer

Here is the order ID: 114-9587331-019785. Email `jane.doe@gmail.com`.

Support Agent

Thank you. Can you please share a photo of the damage and when you noticed the damage?

Invoked Database Agent to look up order using `get_order` tool
Invoked Customer Agent to get more information about the damage

(Simulated) Customer

I noticed it yesterday. Here is a photo of the damage (photo link).

Support Agent

Thank you. I have processed your refund. You will receive a refund to your original payment method in 3-5 business days. Your refund id is ref-1234567890.

Invoked Database Coordinator to update the order status to issue refund
Invoked Customer Agent to send the refund receipt

(Simulated) Customer

Thank you!

Support Agent

You're welcome!

Figure 6.2: A conversation between a (simulated) user and the customer support agent.

Table Table 6.1 shows the results of the contract. The results reveal that while the refund was processed correctly, the customer support agent failed to adhere to two critical internal policies:

1. Verifying the product's eligibility for return
2. Offering the customer a refund as store credit

Both failing requirements are marked as **MUST** conditions, indicating that the system failed the verification step. Consequently, the system designer should take corrective actions (such as improving prompts) before deployment.

It's important to note that since we're using an LLM to plan and act based on human inputs, the behavior is non-deterministic. This means that repeated tests may yield different results, which is the aim of the next section.

	Result	Requirement	Reason
PRE	PASS	The customer MUST provide a valid order ID	-
	PASS	The customer MUST provide a picture of the damaged product	-
PATH	PASS	The system **MUST** check if the customer has a valid order ID using the Database Agent	-
	FAIL	The system **MUST** check if the customer is eligible for a refund checking the return policy using the Knowledge Agent	The system skipped calling the Knowledge Agent to check the eligibility.
	FAIL	The system **MUST** verify if the customer wants refund to original payment method or store credit	The system skipped asking the customer which option they want.
	PASS	The system **MUST** perform the refund transaction using the Database Agent using the `process_refund` tool.	-
POST	PASS	The system **MUST** generate a refund id and **MUST** send it to the user	-

Table 6.1: Results of the contract in figure

6.1.3.1 Explore Probabilistic Behavior

As described in Section 4.3, a single verification run cannot fully capture the statistical behavior of probabilistic systems like LLMs. Due to their inherent non-deterministic behavior, running this scenario multiple times will show that sometimes the system satisfies all conditions and sometimes it doesn't.

	Success	Requirement	Reason
PRE	100%	The customer MUST provide a valid order ID	-
	100%	The customer MUST provide a picture of the damaged product	-
PATH	100%	The system **MUST** check if the customer has a valid order ID using the Database Agent	-
	50%	The system **MUST** check if the customer is eligible for a refund checking the return policy using the Knowledge Agent	The system skipped calling the Knowledge Agent to check the eligibility.
	70%	The system **MUST** verify if the customer wants refund to original payment method or store credit	The system skipped asking the customer which option they want.
	100%	The system **MUST** perform the refund transaction using the Database Agent using the `process_refund` tool.	-
POST	80%	The system **MUST** generate a refund id and **MUST** send it to the user	-

Table 6.2: Results of the contract in figure

This information helps us identify which parts of the system are most likely to fail, allowing us to focus improvement efforts effectively.

For example, we could enhance the agent prompts to ensure the Knowledge Agent is called more proactively to check customer eligibility.

If we cannot guarantee the success of certain conditions, we can convert the verification contract into a certification contract.

6.1.4 Runtime Certification

The flexibility of agent contracts also allows us to use them to *certify* the system at runtime, as explored in Chapter 5..

To implement runtime certification, we must specify what we want the certifier to validate. Since refund requests are critically important, we can adapt the contract described in Contract 6.3 into a certification contract.

The first thing we should identify are the proeconditions, these conditions define when the contract is activated. In the refund scenario this means that *"the customer requests a refund"* and that they *"provide a valid order ID"*. Next we need to specify the pathconditions. These requirements will be checked and updated every time the agent under certification performs a new operation. As we saw in Section 3.1.3, these requirements describe what are some of the operations the agent must do to be correct. In our case we

Refund certificate

Precondition

- The customer requests a refund
- The customer provides a valid order ID

Pathcondition

- The system **MUST** check if the customer is eligible for a refund checking the return policy using the Knowledge Agent
- The system **MUST** verify if the customer wants refund to original payment method or store credit

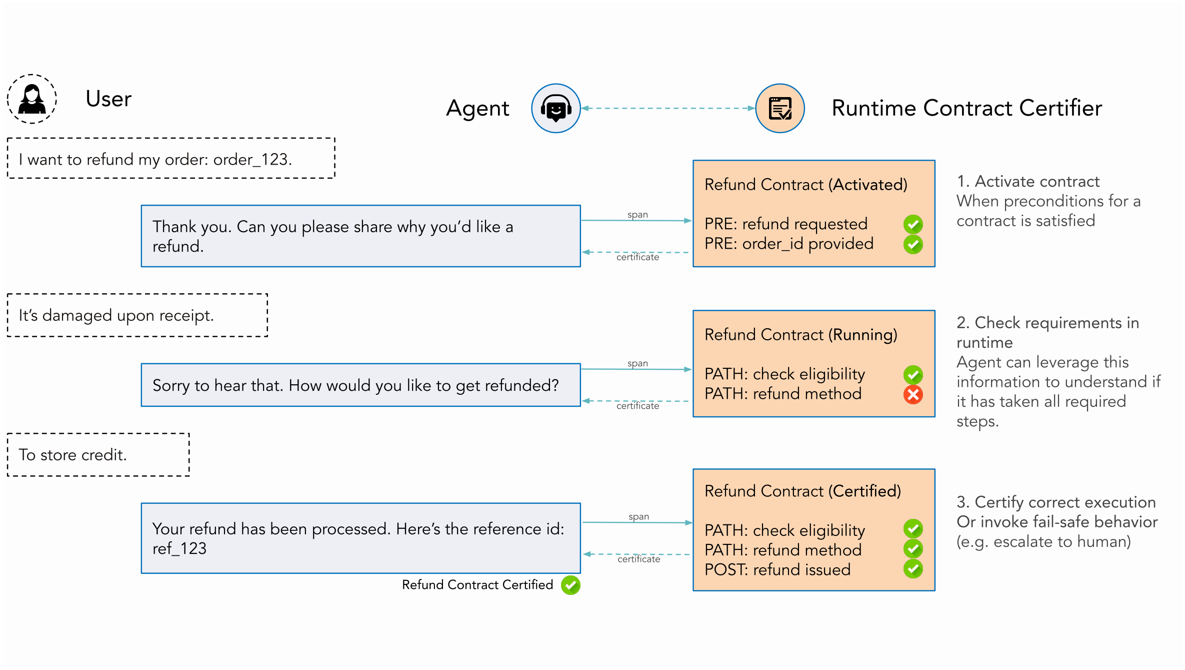
Postcondition

The system **MUST** send the refund iID to the user OR **MUST** notify the user is not eligible for the refund.

Contract 6.4: A certification contract for a customer support agent processing refund requests.

want to agent to (1) check for eligibility (to avoid refund when it should not) and, (2) to try to refund as store credit. Finally, we want to impose some postconditions (Section 3.1.2) to make sure the customer is properly notified. The final certification contract is shown in Contract 6.4.

As before we can try the certification process using the testbot. Figure 6.3 shows an example of how the certification process works for contract.



I want to refund my order: order_123.

Thank you. Can you please share why you'd like a refund.

Refund Contract (Activated)
PRE: refund requested ✓
PRE: order_id provided ✓

1. Activate contract
When preconditions for a contract is satisfied

It's damaged upon receipt.

Sorry to hear that. How would you like to get refunded?

Refund Contract (Running)
PATH: check eligibility ✓
PATH: refund method ✗

2. Check requirements in runtime
Agent can leverage this information to understand if it has taken all required steps.

To store credit.

Your refund has been processed. Here's the reference id: ref_123

Refund Contract (Certified)
PATH: check eligibility ✓
PATH: refund method ✓
POST: refund issued ✓

3. Certify correct execution
Or invoke fail-safe behavior (e.g. escalate to human)

Figure 6.3: Example of runtime certification.

When the user asked *"I want to refund an order"*, the certifier checked the preconditions. Since only the first precondition was true ("The customer requests a refund") the contract was not active yet. The customer support agent however, asked the user to provide the order ID. Once received the new information, the second precondition (The customer provides a valid order ID) becomes true, activating the contract. From here, every interaction between the agent and the user updates the preconditions, that one at the time get satisfied. When the customer support agent sends the final output, the certifier also check the postcondition, emitting the final certificate. As the agent satisfied all requirements, the process and the output is considered *certified*, or in other words correct.

This approach allows us to continuously monitor the system's behavior and ensure it adheres to critical requirements, even as the system evolves over time.

7. Automated Code Generation

With the rise of Large Language Models (LLMs), software development has undergone a dramatic transformation. Developers increasingly rely on LLMs to write code, with a growing trend toward building entire applications end-to-end using AI instead of human programmers. However, just as we needed clear requirements to communicate effectively with human developers, we face the same challenge with LLMs - ensuring they understand and implement exactly what we want. Agent contracts can be used as a powerful solution, providing a structured way to communicate requirements to LLMs and ensure the generated code aligns precisely with the intended functionality.

7.1 The Promise and Pitfalls of LLM-Based Code Generation

More recently, we've begun using Large Language Models (LLMs) to generate code directly from natural language descriptions. This approach shows tremendous promise, you can simply describe what you want, and the AI produces working code in seconds.

For instance, you might prompt an LLM "Write an application to summarize news articles found on the web". The LLM might generate a reasonable implementation, but this approach has significant limitations:

- **Ambiguity**: Natural language is inherently ambiguous.
- **Complexity management**: As requirements grow more complex, it becomes increasingly difficult for the LLM to keep track of all constraints and relationships between different parts of the system.
- **Verification challenges**: How do you verify that the generated code actually meets all your needs? The LLM might produce code that looks correct but misses critical edge cases or security considerations.

7.2 Agent Contracts: A New Specification Language

This is where agent contracts enter can support the code generation process. Agent contracts serve as a specification language specifically designed for as tool to communicate

Latest U.S. Politics News

Scenario

What are the latest U.S. Politics news from the New York Times?

Pathcondition

- The system **MUST** visit the website `https://www.nytimes.com/section/politics`
- The system **MUST** open every article from the last 24 hours

Postcondition

The system **SHOULD** generate a summary with bullet points and citation link

Contract 7.5: A contract for a news aggregator agent.

expected results, behaviors of the system and how to test them. They provide a structured way to express requirements that both humans and LLMs can understand and reason about. An agent contract isn't just a list of features–it's a formal agreement about what the system should do under various conditions.

The contracts examples in Contract 7.5 and Contract 7.6 clarify several things: (1) the user does not always specify the website but they can, (2) the term "latest" means "in the last 24 hours" and (3) the user expects bullet points with citations.

The list of contracts is not exhaustive, but it should give you an idea of how agent contracts can be used to communicate with LLMs with specific examples and detailed requirements.

Latest NVIDIA News

Scenario

What latest news about NVIDIA?

Pathcondition

- The system **MUST** visit the website `https://www.cnbc.com/quotes/NVDA?tab=news`
- The system **MUST** open every article from the last 24 hours

Postcondition

The system **SHOULD** generate a summary with bullet points and citation link

Contract 7.6: A contract for a news aggregator agent.

7.3 The Code Generation Process with Agent Contracts

When using agent contracts for code generation, the process works like this:

1. **Initial Description**: The human provides a relatively vague description of what the system should do—the high-level vision.
2. **Contract Definition**: For each specific aspect of the system, agent contracts are defined that specify both scenario-based behaviors and runtime constraints.
3. **Guided Generation**: The LLM uses these contracts as guardrails during the code generation process, ensuring that each piece of code it produces aligns with the specified contracts.
4. **Continuous Verification**: At each step of code generation, the system checks the emerging codebase against the agent contracts, identifying any areas of non-compliance.
5. **Feedback Loop**: When verification reveals issues, the specific contract violations are used to guide corrections to the generated code.

7.4 Why Agent Contracts Work Better

Agent contracts offer several advantages over traditional approaches:

1. **Reduced ambiguity**: By formalizing requirements in a structured format, we minimize misinterpretations.
2. **Continuous verification**: Contracts serve as an ongoing check throughout the development process, not just at the beginning or end.
3. **Focused feedback**: When issues arise, the contract violations provide specific guidance on what needs to be fixed.
4. **Scalability**: As system complexity grows, contracts help the LLM maintain awareness of all constraints without getting overwhelmed.
5. **Shared understanding**: Both humans and AI systems can interpret and reason about the same contract specifications.

7.5 Implementing Agent Contracts in Your Workflow

To adopt agent contracts in your own development process:

1. Start with a high-level description of your system's purpose and main functionality.
2. Break down this description into specific behaviors and constraints.
3. Formalize these as agent contracts, focusing on both normal scenarios and edge cases.
4. Use these contracts to guide your LLM-based code generation.
5. Implement automated verification that checks generated code against your contracts,iteratively improving the generated code.

For example, if you're building a customer support system, you might have contracts for all the main features of the system, like product search, order tracking, account management, and refund requests. Each contract would specify the expected behaviors and constraints for its domain.

8. Future Work

8.1 Future Directions

The field of agent contracts is still in its early stages, with many exciting possibilities on the horizon. In this chapter, we'll explore three promising directions for future research and development that could significantly enhance the effectiveness and adoption of agent contracts.

8.1.1 Formal Methods to Check Requirements

Today, most verification of agent contracts happens through formal specifications written in natural language or a developer write code to verify the requirement. Both have pros and cons, natural language is easy to write but ambiguous, code is hard to write but unambiguous. But what if we could mathematically prove that our systems meet their requirements through formal methods?

Formal methods offer a way to verify software correctness with mathematical precision. Unlike testing, which can only confirm the presence of bugs but never their absence, formal verification can provide absolute guarantees about system behavior.

For agent contracts, this means we could:

1. **Transform contracts into formal specifications**: Convert natural language contracts into mathematical formulas that can be analyzed by verification tools.
2. **Verify code against these specifications**: Use theorem provers or model checkers to mathematically prove that the generated code satisfies all contract requirements.
3. **Catch issues before deployment**: Identify logical contradictions, edge cases, and potential failures before the code ever runs in production.

For example, imagine a contract specifying that "a user cannot withdraw more money than their account balance." With formal methods, we could prove that no sequence of operations could ever violate this property, rather than just testing a few scenarios.

While formal methods require specialized knowledge and can be computationally intensive, they offer the highest level of assurance. As tools become more accessible and LLMs gain capabilities to assist with formal specification, we expect to see increasing integration of formal verification with agent contracts.

Formal methods could also be very useful for code generation tools. Historically formal methods have been used to prove correctness of human generated code. But with the formal methods based contracts, we could check that the LLM-generated code is indeed correct and implements the right behavior (*i.e.*, the contract).

8.1.2 Synthetic Contracts Generation

Writing good contracts is a skill that takes time to develop. One interesting research direction would be to automate the generation of contracts based on examples, from product use in staging/production environments, system descriptions, or even the existing code.

1. **Generate contracts from examples**: Given a few sample contracts, an AI system could generate additional contracts covering related scenarios.
2. **Expand contracts with edge cases**: Automatically identify potential edge cases not covered in the original contracts.
3. **Suggest contracts from system descriptions**: Generate initial contracts based on high-level descriptions of system functionality.
4. **Learn from production data**: Analyze real system usage patterns to suggest contracts that match actual behavior.
5. **Extract from existing code**: Reverse engineer contracts by analyzing implemented functionality and business logic.
6. **Evolve through feedback**: Refine generated contracts based on developer feedback and validation results.

For instance, if you write a contract for “user login,” a synthetic generation system might automatically suggest related contracts for “failed login attempts,” “password reset,” and “account lockout” scenarios.

This technology would make agent contracts more accessible to developers without extensive specification experience and help ensure more comprehensive coverage of system behaviors. The automated generation could also help identify gaps in contract coverage and suggest improvements based on real-world usage patterns.

8.1.3 From Scenario-based Contracts to Certification Contracts

As AI systems become more sophisticated, we’re seeing a shift from scenario-based contracts to certification contracts. While scenario-based contracts excel at guiding development by specifying behavior for particular use cases, they have limitations when it comes to runtime certification of AI systems.

The future of AI-to-AI interaction demands stronger guarantees about system properties. Consumers of AI outputs increasingly expect formal assurances about system behavior, compliance, and safety. This creates a need for certification contracts that can provide these higher-level guarantees.

However, writing effective certification contracts presents challenges. Developers must identify and formally specify general system properties rather than just individual scenarios. To address this, we propose several key developments:

1. **Generalize scenario-based contracts to certification contracts**: Leverage LLMs to help transform specific scenario contracts into broader certification contracts that capture general system properties.
2. **Enable compliance verification**: Develop reusable contract libraries that encode common compliance requirements. These could include contracts for regulations like GDPR and HIPAA, as well as industry-specific standards.
3. **Support composition**: Create mechanisms for complex systems to inherit certifications from their components. This hierarchical approach makes it feasible to verify large-scale systems by building on certified components.

For example, rather than writing separate contracts for each data access pattern, a certification contract might broadly state: *"This system complies with GDPR requirements for user data processing"* - with precise definitions of compliance requirements that can be automatically verified.

This evolution towards certification contracts will make them valuable beyond just development, enabling automated verification of regulatory compliance, security requirements, and other critical system properties. As AI systems increasingly interact with and depend on each other, these formal guarantees will become essential for building trustworthy AI ecosystems.

9. Conclusions

Agent contracts provide a comprehensive framework for specifying, verifying, and certifying AI systems at runtime. Throughout this report, we have explored how they serve as a crucial bridge between human requirements and AI implementation. By providing a structured specification language that both humans and AI systems can understand and work with, agent contracts help ensure alignment between intended functionality and actual implementation.

The rise of AI-based code generation has made effective specification methods more important than ever. Agent contracts offer a balanced approach, combining formal precision with practical flexibility. They enable developers to clearly communicate requirements while allowing AI systems room to leverage their capabilities within well-defined boundaries. We've seen how agent contracts provide multiple benefits:

- Clear communication of requirements between humans and AI systems
- Early detection of misalignments and potential issues
- Runtime verification and certification of AI behavior
- Structured guidance for AI code generation

Looking ahead, several promising developments are set to make agent contracts even more powerful: (1) integration with formal verification methods, (2) automated generation of contracts from examples and system behavior, (3) evolution toward certification contracts for stronger runtime guarantees.

As AI systems become more prevalent in software development, agent contracts will play an increasingly vital role in ensuring these systems remain reliable, safe, and aligned with human intentions. By continuing to advance contract-based development approaches, we can work toward a future where AI systems consistently deliver what users need while maintaining high standards of quality and trustworthiness.

The journey toward better AI specification and verification methods continues, but agent contracts provide a solid foundation for building more reliable and trustworthy AI systems. As we've demonstrated throughout this book, they offer practical tools for today while pointing the way toward even more powerful capabilities tomorrow.

Bibliography

[1] M. Andreessen, "Why software is eating the world." Accessed: Mar. 05, 2025. [Online]. Available: https://a16z.com/why-software-is-eating-the-world/

[2] M. T. Review, "Nvidia CEO: Software Is Eating the World, but AI Is Going to Eat Software." Accessed: Mar. 05, 2025. [Online]. Available: https://www.technologyreview.com/2017/05/12/151722/nvidia-ceo-software-is-eating-the-world-but-ai-is-going-to-eat-software/

[3] K. E. Wiegers and J. Beatty, *Software requirements*. Pearson Education, 2013.

[4] I. O. for Standardization (ISO), "ISO/PAS 21448:2019: Road vehicles - Safety of the intended functionality," 2019.

[5] OpenTelemetry, "OpenTelemetry." Accessed: Mar. 09, 2025. [Online]. Available: https://opentelemetry.io/