Verifiable Completion Contracts: A Declarative Specification for Outcome-Driven Termination in Autonomous Systems

A Preprint

Aldwin Polanco
aldwin78@gmail.com

January 12, 2026

Abstract

Autonomous agent and workflow frameworks built on large language models (LLMs) can plan and execute multi-step tasks with tool use, yet they routinely fail a critical systems property: *correct termination*. Many systems declare runs "complete" based on procedural conditions such as task exhaustion, iteration limits, or agent self-assessment rather than outcome correctness. This paper introduces *Verifiable Completion Contracts* (VCC), a declarative specification format that defines the necessary and sufficient conditions for declaring an autonomous run complete. A VCC instance specifies required deliverables, acceptance criteria with severity levels (`MUST/SHOULD/MAY`), validator bindings, evidence requirements, and provenance policy. The central invariant is a *termination law*: a run may terminate successfully if and only if all `MUST` acceptance criteria are satisfied. VCC synthesizes principles from design-by-contract, lightweight formal methods, acceptance testing, CI/CD gate enforcement, and requirements traceability. We formalize VCC semantics, present an algorithmic integration pattern for contract-driven convergence, and provide a reference schema and adapter interfaces for implementation. The result is an open specification for building autonomous systems that terminate based on verified outcomes rather than procedural progress.

1 Introduction

LLM-based autonomous systems have progressed from single-shot generation to multi-step execution with planning, delegation, and tool invocation. Despite these advances, most systems still lack a rigorous and enforceable definition of "done." In practice, termination is commonly governed by procedural criteria: a task list has been executed, a recursion or iteration limit is reached, a token or budget ceiling is hit, or the agent asserts completion. Such criteria are not equivalent to correctness.

This paper addresses the question: *How can an autonomous execution system define and enforce successful termination such that a run completes if and only if objective acceptance criteria are satisfied?*

We propose **Verifiable Completion Contracts (VCC)**, a declarative specification format that defines the necessary and sufficient conditions for declaring an autonomous run complete. VCC is released as an open specification to encourage adoption across diverse autonomous systems. A reference implementation is available in SDLAF (Software Development Lifecycle Automation Framework) v3.0. The central design choice is a *termination law*: a run may terminate successfully if and only if no `MUST` acceptance criteria are failing.

Contributions. This paper makes three contributions:

1. A *contract model* (VCC) that specifies required deliverables, acceptance criteria with severity, validator bindings, evidence requirements, and provenance policy.

2. A *formal semantics* for contract satisfaction and correct termination, expressed in a minimal mathematical model.
3. An *integration pattern* that embeds contract-driven convergence into existing refinement loops without creating a new runtime, using bounded budgets and stagnation detection.

2 Problem Definition: Procedural vs. Correct Termination

Many contemporary agent and orchestration systems expose termination controls primarily as recursion or iteration limits and ad hoc stop conditions. For example, LangGraph documents a recursion limit that raises an error when a step budget is exceeded and recommends setting higher recursion limits for complex graphs [1, 2, 3]. While such mechanisms are essential safety features, they are *procedural*: they bound execution but do not define correctness.

We distinguish:

- **Procedural termination**: A run ends because execution has progressed to a stopping condition (e.g., tasks exhausted, recursion limit reached).
- **Correct termination**: A run ends successfully only when outcome requirements are satisfied and verified.

The absence of a machine-enforceable completion contract yields a systematic failure mode: *silent success*. Systems may produce deliverables and even run validators, yet still mark runs complete despite failing requirements.

3 Related Work and Foundations

VCC is a synthesis of established software engineering and structured-generation foundations, adapted to autonomous LLM-driven systems.

3.1 Design by Contract and Acceptance Criteria

Design by Contract (DbC) formalizes correctness through preconditions, postconditions, and invariants that are checked at module boundaries [4]. VCC treats *acceptance criteria* as postconditions over produced deliverables and *provenance policy* as invariants about traceability and evidence.

3.2 Lightweight Formal Methods and Executable Specifications

Specification languages and lightweight formal methods provide precise, checkable descriptions of desired behaviors (e.g., TLA$^+$ [5, 6] and Alloy [7]). VCC adopts a deliberately lightweight stance: specifications are executable through validator adapters and machine-checkable evidence rather than theorem proving.

3.3 Continuous Delivery Gates and Requirements Traceability

Acceptance testing and continuous delivery practices enforce quality through automated checks and gated releases [8]. Requirements traceability research emphasizes that requirements quality depends on explicit links between claims, deliverables, and evidence [9, 10]. VCC operationalizes traceability as first-class policy: criteria bind to deliverables, validators emit evidence records, and provenance requirements are enforced before delivery.

3.4 Structured Generation and Constrained Decoding for LLM Outputs

A large body of work targets *structural correctness* of LLM outputs via constrained decoding and grammar-/schema-guided generation. Recent systems evaluate constrained decoding frameworks against large corpora of real-world JSON Schemas [11]. XGrammar accelerates context-free grammar execution to reduce structured-generation overhead [12]. Libraries such as Outlines provide JSON Schema and grammar-constrained decoding in practice [13], while Guidance popularized programmatic constraints over LM generation [14, 15, 16]. LMQL introduces a query language that compiles constraints and control flow into efficient inference procedures [17]. In parallel, API-level structured outputs and tool calling expose schema-constrained interfaces for commercial models [18, 19, 20].

3.5 Guardrails and Output Validation Frameworks

Beyond structural correctness, guardrails and validation layers filter or score model inputs/outputs to improve reliability and safety [21]. Practical libraries operationalize such checks via schemas, regex constraints, and custom validators; for example, Jsonformer decomposes JSON generation into fixed-token filling with model-generated values [22]. These systems motivate VCC's emphasis on machine-checkable evidence and explicit acceptance criteria.

Positioning. VCC is *not* a replacement for constrained decoding. Instead, it defines a *completion contract* that can be enforced using multiple mechanisms: (i) constrained decoding (when available), (ii) post-hoc validation with bounded remediation, and (iii) human or hybrid gates. This separation is motivated by provider heterogeneity and the need to express *semantic* and *procedural* constraints (e.g., provenance, multi-artifact dependencies) that exceed the expressiveness of JSON Schema alone.

4 VCC Model and Semantics

4.1 Contract Elements

A VCC instance defines:

- **Deliverables**: required outputs with formats, dependencies, and ownership.
- **Acceptance criteria**: atomic properties evaluated by validators, each labeled with severity `MUST/SHOULD/MAY`.
- **Validators**: pluggable adapters capable of evaluating specific criterion types (e.g., schema, structure, traceability, provenance).
- **Evidence**: required records (reports, hashes, logs) produced during validation.
- **Gates, packaging, delivery**: constraints governing when deliverables may be bundled and delivered.
- **Provenance policy**: required run-record signals (e.g., inputs enumerated, tooling recorded, deliverable hashes recorded).

4.2 Formal Semantics

Let:

$$\mathcal{A} = \{a_1, \ldots, a_n\} \quad \text{required deliverables} \tag{1}$$
$$\mathcal{C} = \{c_1, \ldots, c_m\} \quad \text{acceptance criteria} \tag{2}$$
$$\mathcal{C}_M \subseteq \mathcal{C} \quad \text{criteria with severity MUST} \tag{3}$$

Each criterion $c \in \mathcal{C}$ is associated with a validator V_c that returns an evaluation result in $\{\texttt{PASS}, \texttt{FAIL}, \texttt{SKIP}\}$ and may produce evidence deliverables.

Definition 1 (MUST Satisfaction)**.** *A contract's MUST set is satisfied if and only if all MUST criteria evaluate to* `PASS`*:*

$$\text{Satisfied}(\mathcal{C}_M) \iff \forall c \in \mathcal{C}_M : V_c(c) = \texttt{PASS}.$$

Definition 2 (Correct Termination)**.** *A run terminates successfully if and only if the MUST set is satisfied:*

$$\text{TerminateSuccess} \iff \text{Satisfied}(\mathcal{C}_M).$$

Definition 3 (Resource-Bounded Termination)**.** *A run terminates when:*

$$\text{Terminate} \iff \text{Satisfied}(\mathcal{C}_M) \vee \text{Exhausted}(\mathcal{R})$$

with explicit failure state:

$$\text{TerminateFailure} \iff \text{Exhausted}(\mathcal{R}) \wedge \neg\text{Satisfied}(\mathcal{C}_M)$$

Procedural completion (task exhaustion, iteration limit reached) is *not* a success condition under this definition.

4.3 Formal Model

Let a VCC instance S be defined as a tuple:

$$S = \langle \mathcal{D}, \mathcal{C}, \mathcal{V}, \mathcal{P}, \mathcal{R} \rangle$$

where:

- $\mathcal{D}$: Set of deliverables with dependency DAG
- $\mathcal{C}$: Acceptance criteria partitioned by severity $\mathcal{C} = \mathcal{C}_M \cup \mathcal{C}_S \cup \mathcal{C}_Y$ (MUST, SHOULD, MAY)
- $\mathcal{V}$: Validator functions $V_c : c \to \{\texttt{PASS}, \texttt{FAIL}, \texttt{SKIP}\}$
- $\mathcal{P}$: Provenance policy requirements
- $\mathcal{R}$: Resource constraints (time, cost, iterations)

5 Contract-Driven Convergence Algorithm

We describe an integration pattern that embeds VCC evaluation into an existing refinement loop, yielding "run until right" behavior with bounded budgets, without introducing a parallel runtime.

5.1 Termination Conditions

Let $\mathcal{F}_t \subseteq \mathcal{C}_M$ be the set of failing MUST criteria at iteration t. The loop must implement the following ordered checks:

1. **Success**: if $\mathcal{F}_t = \emptyset$, terminate with success.
2. **Budget exhaustion**: if time/cost/iteration ceilings are exceeded, terminate with failure and report $\mathcal{F}_t$.
3. **Stagnation**: if $\mathcal{F}_t$ does not change for k consecutive iterations (configurable), terminate with failure.
4. **Otherwise**: perform another refinement iteration and repeat.

5.2 Pseudocode

```
state <- init(output0)
for t in 0..maxPasses:
  failing <- EvaluateMustCriteria(state.output)
  if failing == {}:
     return SUCCESS
  if BudgetExceeded(state):
     return FAILURE(reason=budget, failing=failing)
  if Stagnant(failing, history, k):
     return FAILURE(reason=stagnation, failing=failing)
  proposal <- Refine(state.output, failing, context)
  state.output <- SelectBest(state.output, proposal, failing)
return FAILURE(reason=max_passes, failing=failing)
```

Contract-first stagnation. Stagnation is assessed primarily over $\mathcal{F}_t$ (contract progress), rather than only numeric quality scores, because correctness is defined by criteria satisfaction.

6 Implementation Considerations

VCC can be integrated as a module that:

- Runs an *integrity pass* over the contract (cross-reference validation; required validator coverage for MUST criteria).
- Binds acceptance criteria to available validator adapters.
- Emits evidence deliverables (validation reports, hashes) required by provenance policy.

This converts heuristic quality signals into enforceable acceptance criteria (e.g., a `MUST` criterion requiring score $\geq \tau$ when configured), while preserving extensibility to domain-specific validators.

Mode	Quality Scores	Threshold	Run Status
API	42–46 / 100	70	COMPLETED (invalid)
Local	41–61 / 100	50	COMPLETED (invalid)

Table 1: SDLAF v3.0.0-alpha regression motivating contract-driven convergence: procedural completion was incorrectly treated as success.

Reference Implementation. A production implementation of VCC is available in SDLAF (Software Development Lifecycle Automation Framework) v3.0. The implementation includes the reference JSON Schema, TypeScript adapter interfaces, and universal validators for structure, traceability, consistency, provenance, and AI-judged rubrics.

6.1 Reference Schema and Adapter Interfaces

A reference JSON Schema for VCC v1 was implemented to support machine validation and forward-compatible evolution (Schema ID: `https://sdlaf.dev/schemas/vcc/v1/schema.json`). A TypeScript reference interface set defines adapter boundaries (planning, execution, validation, packaging, delivery) to keep the orchestration kernel domain-agnostic. Both the schema and reference interfaces are provided as supplementary material with this manuscript.

7 Empirical Evaluation: SDLAF Contract-Driven Convergence Regression

This section reports a focused case study drawn from the SDLAF v3.0.0-alpha test regimen (2026-01-09), where runs were marked `COMPLETED` despite failing `MUST` acceptance criteria. The objective is to validate that VCC's semantics eliminate this failure mode by making acceptance satisfaction the sole success condition.

7.1 System Under Test

SDLAF is a production-grade autonomous workflow framework with a large codebase and extensive automated test coverage. The system implements iterative refinement, deliverable validation, and governance controls, but (prior to contract-driven convergence) allowed procedural completion to imply success.

7.2 Regression Scenario

In the reported tests, quality scores fell below configured thresholds, yet the run status was `COMPLETED`. Table 1 summarizes the observed failure.

7.3 Expected Post-Fix Behavior

Under VCC semantics (Definitions 1–3), a run can only succeed when no `MUST` acceptance criteria are failing. Therefore, the same scenarios must terminate as `FAILED` with an explicit snapshot of failing `MUST` criteria and supporting evidence. This behavior converts silent failure into auditable non-compliance.

7.4 Measurement Targets

For future benchmarking beyond this regression case study, the following metrics are proposed: (i) *Validity rate* of produced deliverables, (ii) *mean remediation iterations* until $\mathcal{C}_M$ is satisfied or budgets are exhausted, (iii) *latency and cost overhead* attributable to validation and remediation, and (iv) *failure transparency*, measured as the completeness of evidence records attached to failed criteria.

8 Discussion

VCC reframes autonomy from intent-driven execution to outcome-driven verification. It is compatible with heterogeneous domains because validators are adapters: deterministic checks for structure, schemas, and provenance can coexist with AI-judged criteria where appropriate. The approach also clarifies failure semantics: if budgets are exhausted before MUST criteria pass, the system terminates with explicit failure and a machine-readable set of unmet requirements.

9 Limitations and Future Work

VCC does not guarantee convergence. Validator quality and coverage are critical, and AI-judged checks require careful calibration and reproducibility controls. Future work includes (i) learning-informed prioritization of refinements based on failing criteria, (ii) compositional contracts across multi-run programs, and (iii) formal analysis of convergence under different refinement operators.

10 Ethical Considerations

Contract-driven termination improves transparency by preventing silent success, but it may incentivize systems to optimize narrowly for criterion satisfaction. Criterion design should therefore include robustness, safety, and provenance requirements, and should be reviewed in high-stakes deployments. When AI-judged validators are used, the evaluation pipeline should be documented and, where possible, made reproducible (model identifiers, prompts, and decision traces).

11 Specification Availability

The VCC specification is released as an open standard under the MIT license. The following artifacts are provided:

- **JSON Schema**: `vcc-v1.schema.json` for machine validation
- **TypeScript Interfaces**: Reference adapter boundaries for planner, executor, validator, packager, and deliverer
- **Universal Validators**: Domain-agnostic validators for structure, traceability, consistency, and provenance
- **Example Contracts**: Research, software, and hardware domain examples

The specification, reference implementation, and examples are available at `https://github.com/SDLAF/vcc-spec`.

12 Conclusion

We introduced Verifiable Completion Contracts (VCC), a machine-verifiable completion contract for autonomous systems that defines successful termination as satisfaction of MUST acceptance criteria. By embedding contract evaluation into existing refinement loops, VCC enables bounded, outcome-driven convergence without introducing a parallel runtime. VCC is released as an open specification to encourage adoption and community contribution. As autonomous systems are deployed in more consequential settings, explicit completion contracts and verifiable termination semantics become foundational rather than optional.

References

[1] GRAPH_RECURSION_LIMIT (javascript) — langgraph documentation. Online documentation, 2025. RecursionLimit configuration and troubleshooting.

[2] GRAPH_RECURSION_LIMIT (python) — langgraph documentation. Online documentation, 2025. Recursion limit raises GraphRecursionError after maximum steps.

[3] Use the graph api — langgraph documentation. Online documentation, 2025. Loop termination conditions and recursion limits.

[4] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992. Metadata listed in Meyer's publication list.

[5] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, June 2002.

[6] Leslie Lamport, John Matthews, Mark Tuttle, and Yuan Yu. Specifying and verifying systems with TLA+. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, September 2002.

[7] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, April 2002.

[8] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[9] Orlena Gotel and Anthony Finkelstein. Modelling the contribution structure underlying requirements. In *International Workshop on Requirements Engineering: Foundations for Software Quality*, 1994.

[10] Orlena Gotel. Making requirements elicitation traceable. PDF, 1994.

[11] Saibo Geng, Hudson Cooper, Michał Moskal, Samuel Jenkins, Julian Berman, Nathan Ranchin, Robert West, Eric Horvitz, and Harsha Nori. Jsonschemabench: A rigorous benchmark of structured outputs for language models. *arXiv preprint arXiv:2501.10868*, 2025.

[12] Yixin Dong, Charlie F. Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. Xgrammar: Flexible and efficient structured generation engine for large language models. *arXiv preprint arXiv:2411.15100*, 2024.

[13] Outlines Contributors. Outlines documentation. Project Documentation, 2025.

[14] Microsoft Research. Guidance: Control lm output. Project Page, 2024.

[15] guidance-ai. Llguidance: Making structured outputs go brrr. Project Documentation, 2025.

[16] guidance-ai. Llguidance: Super-fast structured outputs. GitHub Repository, 2025.

[17] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. *arXiv preprint arXiv:2212.06094*, 2022.

[18] OpenAI. Introducing structured outputs in the api. OpenAI Blog, 2024.

[19] OpenAI. Structured model outputs: Structured outputs guide. OpenAI Platform Documentation, 2024.

[20] Anthropic. Introducing advanced tool use on the claude developer platform. Anthropic Engineering, 2024.

[21] Yi Dong, Ronghui Mu, Gaojie Jin, Yi Qi, Jinwei Hu, Xingyu Zhao, Jie Meng, Wenjie Ruan, and Xiaowei Huang. Building guardrails for large language models. *arXiv preprint arXiv:2402.01822*, 2024.

[22] 1rgs. Jsonformer: A bulletproof way to generate structured json from language models. GitHub Repository, 2023.