ChatGPT

Technical Audit of `ros2-semantic-contracts` Repository

A) Executive summary

`ros2-semantic-contracts` is an ambitious effort to make the implicit semantics of ROS 2’s core behaviours explicit and testable. It offers a layered specification (Core, Global, System) covering actions, lifecycle, parameters, executors/composition and system-level contracts. The repository couples those specifications with a harness designed to capture evidence from baseline implementations (ROS 2 Jazzy + rclcpp + Nav2), evaluate trace events against declared expectations and report divergences. Documentation clearly differentiates between normative statements and hypotheses and emphasises that specifications must be backed by upstream documentation or reproducible evidence [1]. However, the project is unfinished: many specifications are unvalidated, scenario bundles contain stubs, provenance links are incomplete, and the harness re-implementation is still underway. The absence of CI, missing scenario semantics and undefined normative-to-scenario mappings currently limit its credibility. Without closing those gaps, its “political” reception by ROS 2 maintainers is likely to be sceptical.

Overall rating

Category	Assessment
Technical correctness	**Medium** – core contracts capture many expected behaviours (e.g., unique goal IDs [2], lifecycle states [3], allowed parameter types [4]), but numerous clauses are labelled `UNVALIDATED` and lack scenarios or evidence.
Documentation quality	**Medium** – intent, philosophy and methodology documents provide clear rationale and constraints; the separation between normative contracts and measurement harness is well described. However, the spec files often lack context links, and no consolidated list of defined terms is provided.
Completeness	**Low** – many spec clauses are unvalidated; numerous scenarios (A03–A15, L01–L10, P02–P10, etc.) are placeholders with no operations or expectations [5]. Provenance files for scenario semantics are missing, and the reference implementation is empty.
Alignment with upstream docs	**Medium** – several core claims map to design articles (e.g., unique goal IDs [2], lifecycle state machine [3], composition services [6]). But other claims (e.g., action supersession semantics, parameter deletion semantics, executor shutdown behaviour) lack explicit upstream support and are hypothesised.
Repo & link hygiene	**Low** – the repo tree is unusual (many empty or stub files); no CI configuration is present; external links sometimes point to outdated docs; some anchors may be missing.

Category	Assessment
Credibility signals	**Mixed** – the insistence on measurement and traceability is a strong credibility signal, but the lack of implemented scenarios and the ongoing rebuild of the harness reduce confidence.

B) Strengths

- **Clear intent and philosophy.** The project explicitly states what it is (a semantic contract layer derived from ROS 2 baseline behaviour) and what it is not (not a tutorial, best-practices guide or redesign) [7] . This clarity helps avoid misinterpretation.
- **Separation of concerns.** The methodology distinguishes between normative specifications, measurement scenarios, oracle definitions and harness implementation [8] . It requires each normative clause to be testable and linked to upstream evidence or baseline measurements [9] .
- **Layered specification.** Dividing contracts into Core (physics-invariant), Global (wire-protocol) and System (runtime) layers provides a logical structure [10] . This helps maintainers focus on their area of interest.
- **Contributor guidelines.** CONTRIBUTING.md clearly lists accepted contributions (divergence reports, missing semantics, oracle scenarios) and disallowed ones (design proposals, subjective best practices) [11] . This gatekeeping helps maintain scope and reduces bikeshedding.
- **Harness redesign.** The harness contract forbids backends from embedding semantics or verdicts [12] , ensures append-only JSONL trace files, and defines standard operations and exit codes. The ongoing rebuild plan identifies legacy issues and proposes a clean Rust-based core and thin backends [13] .

C) Critical issues

1. **Lack of implemented scenarios and end-to-end traceability.** The majority of scenario bundles in `harness/scenarios_*.json` are placeholders with empty `ops` and `expects` sections; only A01/A02 and H00 scenarios are implemented [5] . Without scenario semantics, the normative clauses are not testable.
2. **Missing scenario semantics definitions.** The `docs/provenance` folder advertises `scenario semantics` definitions, but these files are absent. There is no mapping from spec IDs to scenario semantics or harness implementations, so traceability (Spec → Scenario → Evidence) is broken.
3. **Orphaned normative clauses.** Many clauses in the specs lack scenario IDs or provenance entries. For example, the lifecycle global contract claims that publishers are gated while inactive [14] , but no scenario validates this. The composition contract requires unique node IDs and rejection of duplicate names [15] , but again there is no scenario.
4. **Unvalidated hypotheses mislabelled as normative.** Some clauses are marked `BASELINE` but rely on a single baseline implementation rather than upstream specification. For instance, the action supersession semantics require that a superseded goal be reported as `CANCELED`, yet the design article does not mandate this; it is thus a hypothesis that needs validation.
5. **Absent reference implementation.** The `reference` folder hints at a `ros2_semantics_core` package, but there is no documentation or code, leaving the question of how specifications will be enforced at runtime.
6. **No automated CI or link checking.** The project lacks a continuous integration setup. Without automation, it is difficult to verify that specs and scenarios stay in sync, and broken links remain unnoticed.

7. **Repository structure confusion.** The harness and specs are interleaved with many stub files, making navigation difficult. Namespaces (e.g., `docs/spec/global` vs `docs/spec/system`) are clear, but mixing normative specs with baseline measurement notes and placeholder scenarios confuses new contributors.
8. **Unclear update/versioning policy.** The methodology promises stable versioning and refusal to silently change semantics [16], but no version tags or release notes are present in the repo.

D) Traceability audit results

Orphaned specifications

Several normative clauses do not map to scenario IDs or provenance entries:

Spec file	Clause summary	Observation
`docs/spec/core/action_core.md`	Cancellation semantics: a goal cancellation request must allow any terminal outcome and not change the terminal state [17].	No scenario (`A*`) exercises cancellation and verifies terminal state correctness; spec lacks `scenario:` tag.
`docs/spec/global/action_global.md`	Supersession semantics: preemption of an action should be represented as `CANCELED` [18].	No scenario tests this; no upstream citation.
`docs/spec/system/composition.md`	Atomic unload semantics: unloading a node must complete without side effects [19].	No scenario exists (no `S*` scenario).
`docs/spec/system/executor.md`	Bounded spin modes and idempotent shutdown [20].	No scenario `S*` verifying different spin calls or shutdown idempotency.
`docs/spec/system/system.md`	Hidden execution must be disallowed; there must be no hidden spinners [21].	No scenario ensures that nodes don't spin implicitly.

Orphaned scenarios

The harness contains scenario bundles with IDs that have no corresponding spec ID or normative clause. For example:

- In `scenarios_P.json`, scenario `P01_null_parameter_exists` references `spec_id: parameter-null-exists`, but this ID is not defined in any spec file.
- Several `G` and `L` scenarios mention spec IDs that are absent.

These orphaned scenarios suggest a misalignment between scenario planning and spec authoring.

Mismatched IDs

Where scenario IDs exist, the mapping is inconsistent. For instance, `A01_unique_goal_identity` expects the goal to be rejected and checks for absence of status/result events [5], but the corresponding normative clause (Action Core: unique goal identity [22]) requires that a goal UUID be unique and not reused; it does not prescribe rejection behaviour. The scenario conflates uniqueness with rejection and thus mismatches the spec.

Untestable clauses

Some normative clauses rely on timing or behaviour that is currently unmeasurable:

- **Monotonic status ordering**: the action core spec requires that status messages are published in non-decreasing order [23]. Without a definition of the observation window or guidance on sampling frequency, this is untestable.
- **Settling windows for composition**: the composition spec mentions a “settling window” for load/unload operations [24] but the harness has no concept of settling windows. Without measurement definitions, this clause cannot be enforced.
- **Error resolution semantics**: the lifecycle core spec requires deterministic resolution of error states [25]. Without a scenario to generate and observe error transitions, this clause remains hypothetical.

E) Spec-vs-upstream comparison table

The table below summarises major domains, highlights key normative claims in the repo, and compares them with explicit statements from upstream ROS 2 design articles or documentation.

Domain	Normative claim (repo)	Upstream support	Assessment
Actions	Goals must have unique UUIDs; clients generate IDs and servers must not reuse them [22].	The ROS 2 actions design article states that action clients are the sole entities generating goal IDs and a UUID is used to mitigate collisions [2].	Matches upstream; explicit.
	Terminal states are immutable: once a goal reaches SUCCEEDED/ ABORTED/CANCELED it cannot transition to another terminal state [26].	The goal state machine lists three terminal states and shows transitions from active to terminal but does not describe transitions between terminal states [27]. There is no explicit statement forbidding terminal override; this is a reasonable interpretation but needs evidence.	Upstream silent; labelled hypothesis.
	Cancel requests are advisory; cancellation may result in any terminal state and servers are free to ignore cancellations [17].	The cancel goal service description notes that a cancel request returns a list of goals to be attempted and that the status and result indicate whether the goal was canceled, aborted, etc. [28]. It does not forbid servers from aborting or succeeding a goal despite cancellation.	Upstream implicitly allows any outcome; claim is consistent.

Domain	Normative claim (repo)	Upstream support	Assessment
	Supersession semantics: if one goal supersedes another, the superseded goal should be reported as `CANCELED` [18].	The action design article does not mention supersession; no normative statement about how preemption should be represented.	Upstream silent; treat as baseline hypothesis.
	Result retention must not be zero (server must cache results) and the retention period must be configurable and documented [29].	The design article suggests that the server should cache results and discard them after a configurable timeout; the timeout may be set to 0 to discard immediately [30].	Repo contradicts upstream (upstream allows zero retention).
Lifecycle	Four primary states (Unconfigured, Inactive, Active, Finalized) and six transition states; only specified transitions are allowed [31] [32].	The managed nodes design article lists the same states and transitions [3] [33].	Matches upstream.
	While inactive, a node must not process data or respond to managed services [14].	The lifecycle article states that in the inactive state the node will not receive execution time and any managed service requests will fail immediately [34].	Matches upstream.
	Transitions are strictly controlled; only one active transition can occur at a time [35].	The design article lists transitions but does not explicitly forbid concurrent transitions.	Upstream ambiguous; treat as baseline requirement.
	Error resolution semantics: after an error the node must transit to Unconfigured or Finalized with deterministic cleanup [25].	The design article describes `ErrorProcessing` and possible transitions back to `Unconfigured` or to `Finalized` [36], but it does not define deterministic behaviour.	Upstream partially silent; some support but details need validation.
Parameters	Parameter names are case-sensitive and must not be fabricated by clients [37].	The parameter design article emphasises that parameters are addressed by node name and parameter name [38] but does not mention case sensitivity; rclcpp implements case-sensitive names.	Upstream ambiguous; baseline implementation is case-sensitive.

Domain	Normative claim (repo)	Upstream support	Assessment
	Undeclared parameters may be allowed; unknown parameters must return `NOT_SET` when queried and dynamic typing applies [39] [40].	rclcpp notes that when undeclared parameters are allowed and a parameter is set without being declared, it will be dynamically typed, and getting an undeclared parameter returns `NOT_SET` [41].	Matches upstream.
	Atomic parameter updates: sets of parameter updates must be applied atomically and roll back on failure [42].	The design article advocates for atomic set/get of parameter groups [43], but it does not specify rollback semantics on failure.	Upstream partially supports atomic operations; rollback behaviour is unspecified.
	Parameter deletion support: ability to delete parameters individually or groups [44].	The parameter design article lists unset/unset groups as a feature [45].	Matches upstream in intent.
Composition	A container must expose `~/_container/load_node`, `~/_container/unload_node`, and `~/_container/list_nodes` services; `load_node` must reject duplicate names and assign a unique ID [46] [15].	The launch design article states the same services, warns that services are hidden to avoid collisions, and states that duplicate names must be rejected; each loaded node must be assigned a unique ID [6].	Matches upstream.
	Load/unload operations must be atomic and provide human-readable failure messages [24] [19].	Upstream mentions that loading nodes may be parallel or sequential and that failure should be handled, but does not require human-readable messages or atomicity [47].	Repo adds stricter requirements; treat as baseline enhancements.
Executors	Executions must be bounded: support for spin forever, spin once and spin with timeout; shutdown must be idempotent and bounded [20].	ROS 2 executor documentation describes Single-Threaded and Multi-Threaded executors, but does not specify bounded spin or idempotent shutdown [48].	Repo extends beyond upstream; these are hypotheses needing validation.

Domain	Normative claim (repo)	Upstream support	Assessment
	Callback groups must be declared and concurrency model documented; no hidden spinners [49] [50].	ROS 2 docs explain callback groups and that Multi-Threaded executor processes callbacks in parallel respecting callback groups [51], but there is no prohibition on hidden spinners or requirement for documentation.	Repo imposes stricter requirements; upstream silent.

F) Harness readiness review

The harness design is one of the strongest parts of this repository, but it is not yet functional:

- **Contracted behaviour.** The harness contract specifies that backends emit append-only JSONL traces, support a small set of operations (`send_goal`, `wait`, etc.), emit lifecycle events (`run_start`, `scenario_start`, etc.), and never emit verdicts or assertions [12]. The stub backend contract implements these rules for stub scenarios [52]. This separation ensures backends do not encode semantics, which is key for credible measurement.
- **Core runner and backends.** The `run_harness.sh` script builds the Rust-based core and the selected backend, then executes scenarios and produces a trace and report [53]. However, the code for the core and `backend_ros` is missing from our audit (GitHub navigation limitations); their behaviour cannot be verified.
- **Scenario schema and trace schema.** JSON schemas define valid operations, expectations and trace events, forbidding `assertion` or verdict fields [54] [55]. This provides machine-checkable constraints.
- **Scenario coverage.** Only two ROS semantics scenarios are defined (A01/A02) plus one harness self-test (H00) [56]. Most other scenarios are placeholders, so the harness cannot yet validate the majority of specs.
- **Observation and settling windows.** The methodology document defines observation windows, settling windows and verdict definitions [57], but the harness implementation does not enforce them. Scenarios have no fields for window durations, and the stub backend emits events without timing semantics. Without windows, timing-related clauses are untestable.
- **Capability declarations.** The methodology mentions the need for capability declarations so scenarios can be skipped if a backend lacks support [8]. There is no implementation of capability descriptors in the harness or scenario schema.

Overall, the harness provides a sound conceptual foundation, but its current state is insufficient to test most specifications.

G) Repository hygiene and contributor experience

- **Structure and naming.** The repository uses a logical directory structure (`docs/spec/core`, `docs/spec/global`, `docs/spec/system`, `harness/`, `docs/provenance/`, `reference/`). However, many directories contain placeholder files or are empty (e.g., `reference/ros2_semantics_core`). This gives the impression of an unfinished project.
- **No continuous integration.** There are no GitHub Actions or similar CI pipelines. Contributors cannot run automated tests or link checkers, and regressions will not be caught early.

- **Link hygiene.** Many links in the spec point to ROS 2 design articles or docs; some of these links reference general pages rather than anchored sections, and there are no automated checks for anchor validity. During the audit, we observed that external links from the composition spec to design articles were valid and matched content [6], but others (e.g., to parameter API) could become stale.
- **Templates and guidelines.** There is a CONTRIBUTING guide and a clear philosophy, but there are no issue or pull request templates to guide contributors. There is also no `CODE_OF_CONDUCT.md`.
- **License and copyright.** The repository is licensed under Apache 2.0 [58], which is permissive and standard.
- **API connectors.** The repository has no integration with the provided `ros2-systems-operability` internal connector; this is appropriate for an open specification, but those connectors could be used to fetch internal baseline traces.

H) Reception analysis ("How this will be received")

ROS 2 core maintainers

- **Likes:** The focus on reproducible evidence and the willingness to back normative claims with upstream citations or baseline measurements will resonate. The layered structure echoes established interfaces (e.g., separation of DDS wire protocol vs application semantics), and the harness's emphasis on not encoding semantics in backends matches ROS 2's modularity goals.
- **Objections:** Core maintainers may object to normative claims that contradict or extend the design articles without clear justification. For example, insisting that result retention cannot be zero conflicts with the action design article, which allows immediate discarding [30]. Claims about executor behaviour (bounded spin, no hidden spinners) and composition semantics (settling windows, human-readable errors) go beyond upstream specs and could be seen as overreach. They may also be wary of the project's baseline selection (ROS 2 Jazzy + rclcpp + Nav2) as a source of truth; emphasising that baseline observations are not normative could reduce friction.
- **Tone:** The philosophy document sometimes adopts an adversarial tone, stating that the repository refuses to be a tutorial and rejects community conventions [59]. Softening this language (e.g., "this repository complements existing tutorials by focusing on testable semantics") would reduce defensiveness.

Ecosystem maintainers (e.g., Nav2, MoveIt)

- **Likes:** Having explicit contracts can help maintainers ensure their packages behave consistently across ROS 2 releases. The use of Nav2 as an "evidence amplifier" [60] could highlight real-world issues.
- **Risks:** If the project prescribes behaviours that diverge from current practice (e.g., requiring specific error messages or unique node IDs), it may create pressure on packages to conform or risk being labelled "non-compliant." Without broad community buy-in, maintainers might ignore or oppose the project.

Safety/industrial users

- **Credibility:** The emphasis on deterministic behaviour, formal contracts and evidence can appeal to safety-critical users. The structured approach (capability declarations, observation windows) mirrors certification frameworks.
- **Challenges:** The current lack of completed scenarios and verified traceability undermines confidence. Industrial users need assurance that the measurement harness is trustworthy, which requires more maturity.

General community

- **Reception:** Many ROS developers are frustrated by ambiguous semantics and will welcome efforts to codify behaviour. However, the current state may appear overly theoretical. Clearer mapping to everyday issues (e.g., "why does my action sometimes not cancel?") and easy-to-run examples would help adoption.

Recommended wording tweaks

- Replace negative phrasing like "we refuse to be a tutorial" [59] with positive statements such as "this project complements existing tutorials by focusing exclusively on contractable semantics."
- When introducing baseline hypotheses, explicitly label them as such (e.g., "Based on current rclcpp behaviour, we hypothesise…"), and invite the community to validate or refute them.
- Acknowledge upstream ambiguity instead of asserting unsourced requirements. For example, for result retention, say "The ROS 2 design allows servers to discard results immediately [30]; our baseline retains them for at least one trace window to support introspection."

I) Prioritised action list

P0 – Blocks correctness/credibility

1. **Implement scenario semantics and mapping.** Create `docs/provenance/scenario/*.md` files defining the semantics of each scenario and link them to spec clauses. Implement missing scenarios (L, *P*, G, *S*) in the harness with `ops` and `expects`, so every normative clause has at least one validating scenario.
2. **Complete the harness rebuild.** Finalise the Rust core runner, add capability declarations and observation/settling window support, and ensure the `backend_ros` is functional. Add CI tests to run stub scenarios and verify trace compliance.
3. **Establish traceability tables.** For each spec file, add a table mapping normative clauses to scenario IDs and provenance entries. Ensure that each clause is tagged as `UNVALIDATED`, `BASELINE`, or `VALIDATED` with a link to the scenario and upstream citation.
4. **Remove or rephrase unsupported normative claims.** For example, revise the supersession semantics, result retention prohibition and executor shutdown clauses to either cite upstream design articles or clearly mark them as baseline hypotheses.

P1 – Improves auditability

1. **Add automated link checking and CI.** Use a GitHub Action to build the harness, run all scenarios against the baseline, and check for broken links in Markdown files.
2. **Provide a glossary of terms.** Add a document defining terms such as "scenario bundle," "observation window," "capability," etc., to improve readability and avoid ambiguity.
3. **Document reference implementation.** Explain the purpose of `reference/ros2_semantics_core`, what interfaces it exposes, and how it interacts with the harness.
4. **Versioning and release notes.** Introduce version tags (e.g., v0.1, v0.2) and a change log. This will allow consumers to track normative changes and manage compliance over time.

P2 – Nice-to-have

1. **Improve contributor experience.** Add issue and pull request templates, a code of conduct, and `README` files in subdirectories explaining their purpose.

2. **Expand upstream cross-references.** Cite additional sources such as REPs and rclcpp/rclpy API docs to support more clauses. Include optional `see also` sections in the specs for deeper reading.
3. **Provide real-world examples.** Offer examples of how to run the harness against a simple package, illustrating how divergence records help diagnose issues.

By addressing the P0 tasks, the project can move from a promising concept to a credible standardisation effort. Its success depends on delivering complete, verifiable contracts and maintaining a collaborative tone that invites the ROS 2 community to participate in validating and refining the semantics.

---

1 59 raw.githubusercontent.com
https://raw.githubusercontent.com/convyares-FCSL/ros2-semantic-contracts/main/docs/philosophy.md

2 27 28 30 Actions
https://design.ros2.org/articles/actions.html

3 33 34 36 Managed nodes
https://design.ros2.org/articles/node_lifecycle.html

4 38 43 45 Parameter API design in ROS
https://design.ros2.org/articles/ros_parameters.html

5 raw.githubusercontent.com
https://raw.githubusercontent.com/convyares-FCSL/ros2-semantic-contracts/main/harness/scenarios/scenarios_A.json

6 15 47 ROS 2 Launch System
https://design.ros2.org/articles/roslaunch.html

7 60 raw.githubusercontent.com
https://raw.githubusercontent.com/convyares-FCSL/ros2-semantic-contracts/main/docs/intent.md

8 9 16 raw.githubusercontent.com
https://raw.githubusercontent.com/convyares-FCSL/ros2-semantic-contracts/main/docs/methodology.md

10 ros2-semantic-contracts/docs/spec at main · convyares-FCSL/ros2-semantic-contracts · GitHub
https://github.com/convyares-FCSL/ros2-semantic-contracts/tree/main/docs/spec

11 raw.githubusercontent.com
https://raw.githubusercontent.com/convyares-FCSL/ros2-semantic-contracts/main/CONTRIBUTING.md

12 ros2-semantic-contracts/harness/contracts/backend_contract.md at main · convyares-FCSL/ros2-semantic-contracts · GitHub
https://github.com/convyares-FCSL/ros2-semantic-contracts/blob/main/harness/contracts/backend_contract.md

13 ros2-semantic-contracts/harness/contracts/rebuild_plan.md at main · convyares-FCSL/ros2-semantic-contracts · GitHub
https://github.com/convyares-FCSL/ros2-semantic-contracts/blob/main/harness/contracts/rebuild_plan.md

14 raw.githubusercontent.com
https://raw.githubusercontent.com/convyares-FCSL/ros2-semantic-contracts/main/docs/spec/global/lifecycle.md

17 22 23 26 raw.githubusercontent.com
https://raw.githubusercontent.com/convyares-FCSL/ros2-semantic-contracts/main/docs/spec/core/action_core.md

[18] [29] raw.githubusercontent.com
https://raw.githubusercontent.com/convyares-FCSL/ros2-semantic-contracts/main/docs/spec/global/action.md

[19] [24] [46] raw.githubusercontent.com
https://raw.githubusercontent.com/convyares-FCSL/ros2-semantic-contracts/main/docs/spec/system/composition.md

[20] [49] [50] raw.githubusercontent.com
https://raw.githubusercontent.com/convyares-FCSL/ros2-semantic-contracts/main/docs/spec/system/executor.md

[21] raw.githubusercontent.com
https://raw.githubusercontent.com/convyares-FCSL/ros2-semantic-contracts/main/docs/spec/system/system_contract.md

[25] [31] [32] [35] raw.githubusercontent.com
https://raw.githubusercontent.com/convyares-FCSL/ros2-semantic-contracts/main/docs/spec/core/lifecycle_core.md

[37] [39] [40] raw.githubusercontent.com
https://raw.githubusercontent.com/convyares-FCSL/ros2-semantic-contracts/main/docs/spec/core/parameter_core.md

[41] Notes on statically typed parameters — rclcpp: Jazzy 28.1.16 documentation
https://docs.ros.org/en/jazzy/p/rclcpp/doc/notes_on_statically_typed_parameters.html

[42] [44] raw.githubusercontent.com
https://raw.githubusercontent.com/convyares-FCSL/ros2-semantic-contracts/main/docs/spec/global/parameters.md

[48] [51] Executors — ROS 2 Documentation: Foxy documentation
https://docs.ros.org/en/foxy/Concepts/About-Executors.html

[52] ros2-semantic-contracts/harness/contracts/stub_backend.md at main · convyares-FCSL/ros2-semantic-contracts · GitHub
https://github.com/convyares-FCSL/ros2-semantic-contracts/blob/main/harness/contracts/stub_backend.md

[53] raw.githubusercontent.com
https://raw.githubusercontent.com/convyares-FCSL/ros2-semantic-contracts/main/harness/scripts/run_harness.sh

[54] raw.githubusercontent.com
https://raw.githubusercontent.com/convyares-FCSL/ros2-semantic-contracts/main/harness/schemas/scenario.schema.json

[55] raw.githubusercontent.com
https://raw.githubusercontent.com/convyares-FCSL/ros2-semantic-contracts/main/harness/schemas/trace_event.schema.json

[56] raw.githubusercontent.com
https://raw.githubusercontent.com/convyares-FCSL/ros2-semantic-contracts/main/harness/scenarios/scenarios_H.json

[57] raw.githubusercontent.com
https://raw.githubusercontent.com/convyares-FCSL/ros2-semantic-contracts/main/docs/provenance/oracle/oracle_measurement.md

[58] raw.githubusercontent.com
https://raw.githubusercontent.com/convyares-FCSL/ros2-semantic-contracts/main/LICENSE