Veritrace

Human-reviewed LLM rule authoring and symbolic Jason traceability

Lorenzo Leoni
lorenzo.leoni5@studio.unibo.it
Alma Mater Studiorum – University of Bologna

Intelligent Software Engineering – Academic Year 2025/2026

Contents

Declaration on the Use of AI

Generative AI tools were used as assistance during software development, user-interface implementation, debugging, and report drafting/revision. All generated material was reviewed, edited, and validated by the author. Benchmark traces and deterministic test results were produced by the project tooling, not by AI generation. The author takes responsibility for the final report and for the submitted software artifact.

1 Abstract

Veritrace is a local prototype for reviewing LLM-generated rule drafts before they are allowed to run inside a Jason multi-agent system. The core principle is to let the model support ambiguous authoring work without letting it decide the outcome of a case.

The project starts from source text, extracts claims, turns one selected claim into a candidate symbolic rule, asks a human to review it, promotes the approved rule, compiles it into AgentSpeak, and then runs the case through Jason. The final result is not just a label such as `high_risk` or `notify_authority`. It is a structured trace showing which rule fired, which evidence was used, what was missing, and which next steps the planner emitted.

Veritrace was built around a practical question: if an LLM can draft useful rules from text, how can those rules enter a runtime system without becoming hidden, unreviewed behaviour? The answer in this prototype is a gated pipeline. The LLM drafts, the human reviews, the backend promotes and validates, and Jason performs the runtime reasoning.

The review console supports two demo areas. The Cardiac path is the main authoring demo: it has a prepared CMR Mass Score source, claim, and candidate rule for the fast walkthrough, and it can also be driven through the live LLM pipeline. GDPR is stronger on the runtime side: it has approved rules, approved plan artifacts, benchmark cases, and expected traces for compliance scenarios such as missing legal basis, special-category processing, and late breach notification.

The result is an exam prototype, not a production compliance or medical tool. Its purpose is to show how LLM-assisted authoring, human review, symbolic execution, and trace inspection can be connected in one reproducible local workflow.

2 Domain and Core Concepts

The domain is traceable rule authoring for decision-support systems. In this kind of system the hard part is not only producing an answer. The hard part is being able to say where the answer came from: which source text suggested the rule, who accepted it, which executable artifact was produced, and which runtime evidence triggered it.

Without that chain, an LLM-based system becomes difficult to inspect. A model may summarise a source, invent a rule-like statement, and then explain its own answer, but those steps are usually mixed together in a single opaque generation. Veritrace separates them into visible artifacts.

The Cardiac and GDPR examples exercise different parts of the workflow. The Cardiac demo shows source-to-claim-to-rule authoring from a technical source. The GDPR cases show runtime reasoning, because compliance decisions naturally require evidence, missing-data handling, source references, and next steps. The project does not attempt to solve either domain completely; it uses them as realistic enough examples for the architecture.

Concept	**Meaning in Veritrace**
Source document	Text the rule should come from, identified by source id, domain, type, and body.
Claim	A candidate interpretation extracted from the source, with a quote, candidate meaning, and review flag.
Candidate rule	JSON draft produced from a claim. It contains conditions, conclusions, source metadata, and review metadata. It starts as `draft`.

Reviewed rule	The same candidate after a human decision: approved, rejected, or marked as needing revision.
Approved rule	A reviewed rule saved under `approved/rules/`. Only these rules are compiled into generated AgentSpeak.
Approved plan	A JSON artifact under `approved/plans/` mapping a decision to suggested next steps.
Jason MAS	The AgentSpeak runtime that evaluates case facts against approved symbolic rules and plans.
Structured trace	JSON result containing decision, risk, activated rules, used evidence, missing data, sources, human-review flags, and next steps.
Constrained explanation	Natural-language explanation produced after the trace exists. It explains the trace; it does not add new facts.

3 Design

Veritrace is built as a pipeline rather than a single agent that does everything. This shape makes the risky transitions visible. A source becomes a claim. A claim becomes a candidate rule. A candidate rule becomes a reviewed rule. A reviewed rule becomes an approved runtime artifact. Jason then turns case facts into a trace.

The LLM is useful in two phases: authoring, when text must be transformed into candidate structure, and explanation, when a structured trace should be verbalised. It is not used in the middle, where runtime authority is decided. That part belongs to the reviewer, the compiler, and Jason.

Logic-LM by Pan et al. separates LLM translation from symbolic solving, producing logic programs that a solver executes [1]. Veritrace shares the same separation pattern: the LLM handles natural-language authoring, while a symbolic runtime produces the final result. However, the role of each side differs. In Logic-LM the LLM translates every problem instance into a fresh logic program. In Veritrace the LLM drafts candidate rules once, at design time; those rules are reviewed by a human before they become executable logic. The symbolic side is also different: instead of a single solver call, Veritrace runs a Jason multi-agent system that selects plans, checks evidence, and coordinates four agents.

The paper by Ciatto et al. on GenAI plan generation for BDI agents motivated the plan layer [2]. The idea of using generative AI to produce plan-like artifacts for a BDI runtime is present in both projects. In Veritrace, however, plans are first-class approved artifacts that pass through the same review-compilation pipeline as rules. They are not live model output injected into the runtime. Every approved plan must be a validated JSON artifact, compiled into AgentSpeak, and gated by an `approved_plan(PlanId)` belief before the planner can trigger it. Handwritten fallback plans exist alongside generated ones, so the runtime keeps working when no approved plan is available.

This makes the prototype less autonomous than a fully agentic system, by design. For this project, being inspectable mattered more than being fully automatic.

4 Architecture

4.1 System Context

The system has one human actor: the reviewer, who works through the browser console. The system boundary includes the Node.js backend (which handles review logic, promotion, compilation, and MAS orchestration) and the Jason runtime (which executes AgentSpeak plans). Two external actors sit outside this boundary: the LLM provider, called only for structured authoring tasks, and the filesystem artifact store, which holds approved rules and plans as committed JSON.

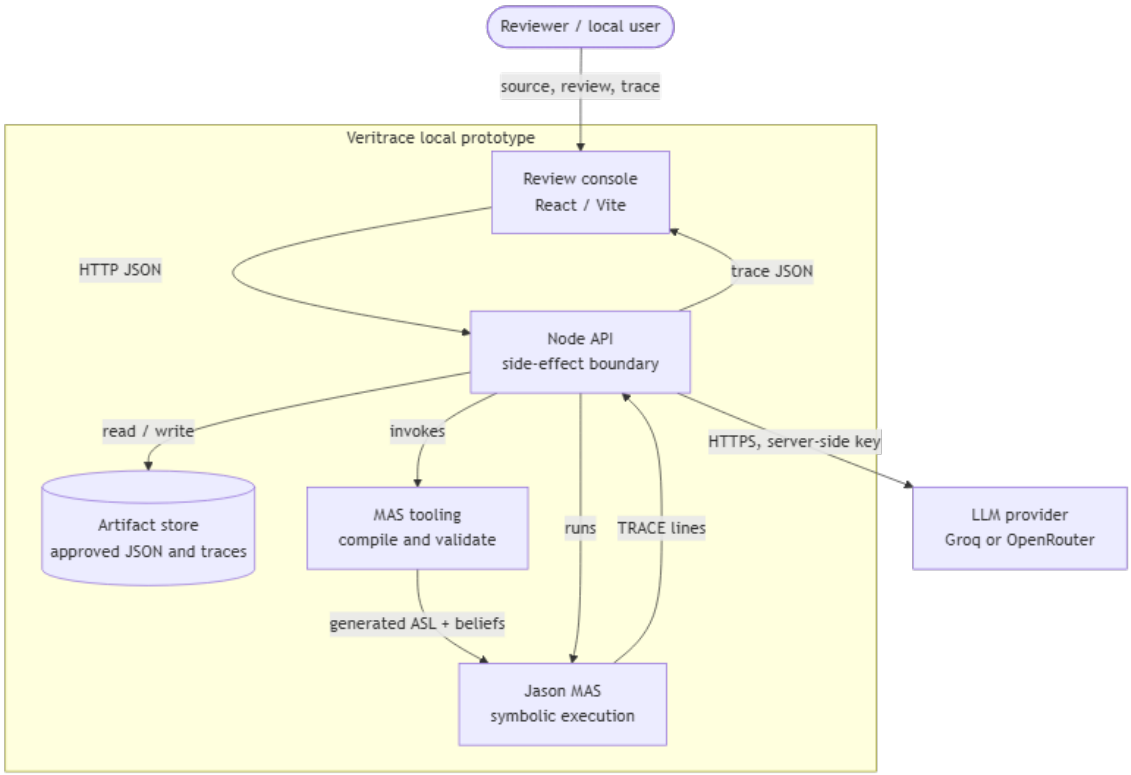

Figure 1: System context diagram. The reviewer interacts exclusively through the browser; the Node.js backend handles review, promotion, compilation and orchestration; the Jason runtime executes AgentSpeak plans. Two external actors (LLM provider and version-controlled filesystem artifact store) are outside the system boundary.

The key property of this boundary is that the LLM never communicates with the runtime directly. Every model response returns to the backend as candidate data; approval, compilation, and execution happen inside the boundary without further LLM involvement. The reviewer is therefore the only actor who can promote an artifact from candidate to executable. The filesystem artifact store doubles as the version-controlled repository, so every promoted artifact is traceable through git history.

4.2 Component Structure

Inside the project, the work splits into a small number of components. The browser and backend prepare artifacts; the Jason MAS executes four cooperating agents.

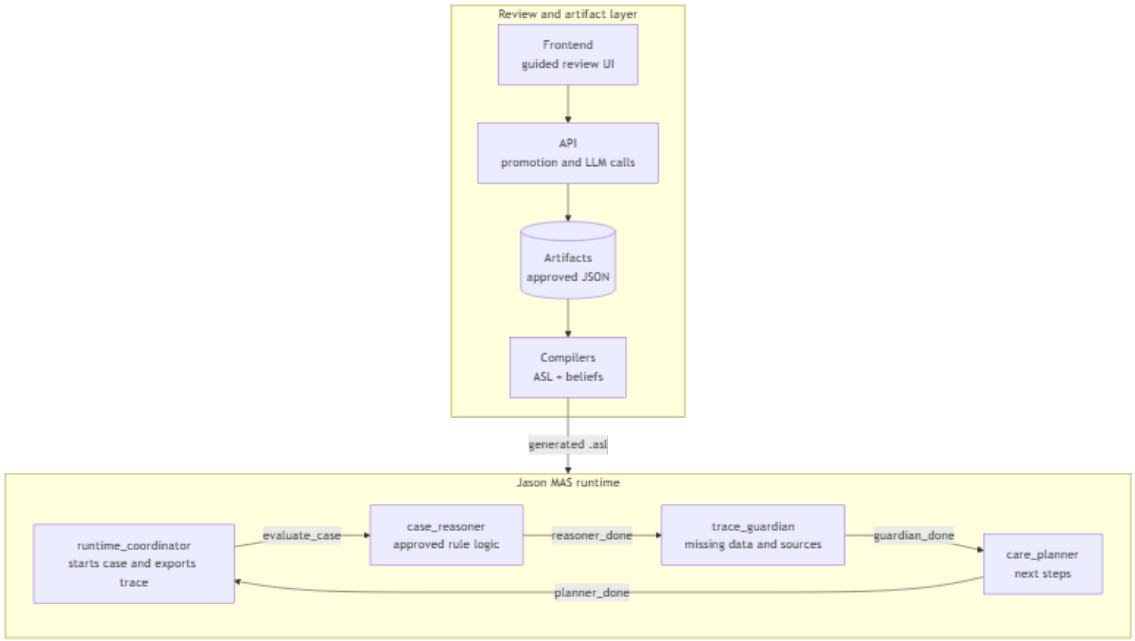

Figure 2: Component structure. The review and artifact layer prepares executable artifacts; the Jason runtime consumes them.

The frontend is a guided review interface: it selects the demo, shows the source, displays candidate rules as readable cards, exposes the review controls, and renders traces.

The backend is where the side effects happen: it calls the LLM, writes approved artifacts, compiles rules and plans, invokes Jason, loads expected traces, runs custom cases, and records audit events.

The artifact store is file-based. This would not be sufficient for a real multi-user governance tool, but it suits this prototype because every important object can be opened, diffed, committed, or regenerated.

The MAS tooling connects the JSON world with the AgentSpeak world: approved rules and plans are validated and compiled into `.asl` files and belief files.

5 Sequence Diagrams

This section contains the interaction sequences of the three main flows.

5.1 Full LLM Authoring Flow

The user selects a demo and starts from the source document. The backend sends the source to the LLM and asks for structured claims. The user selects one claim, then asks the backend to draft a candidate rule from it. The generated rule is still only a draft: it can be inspected, but it cannot run.

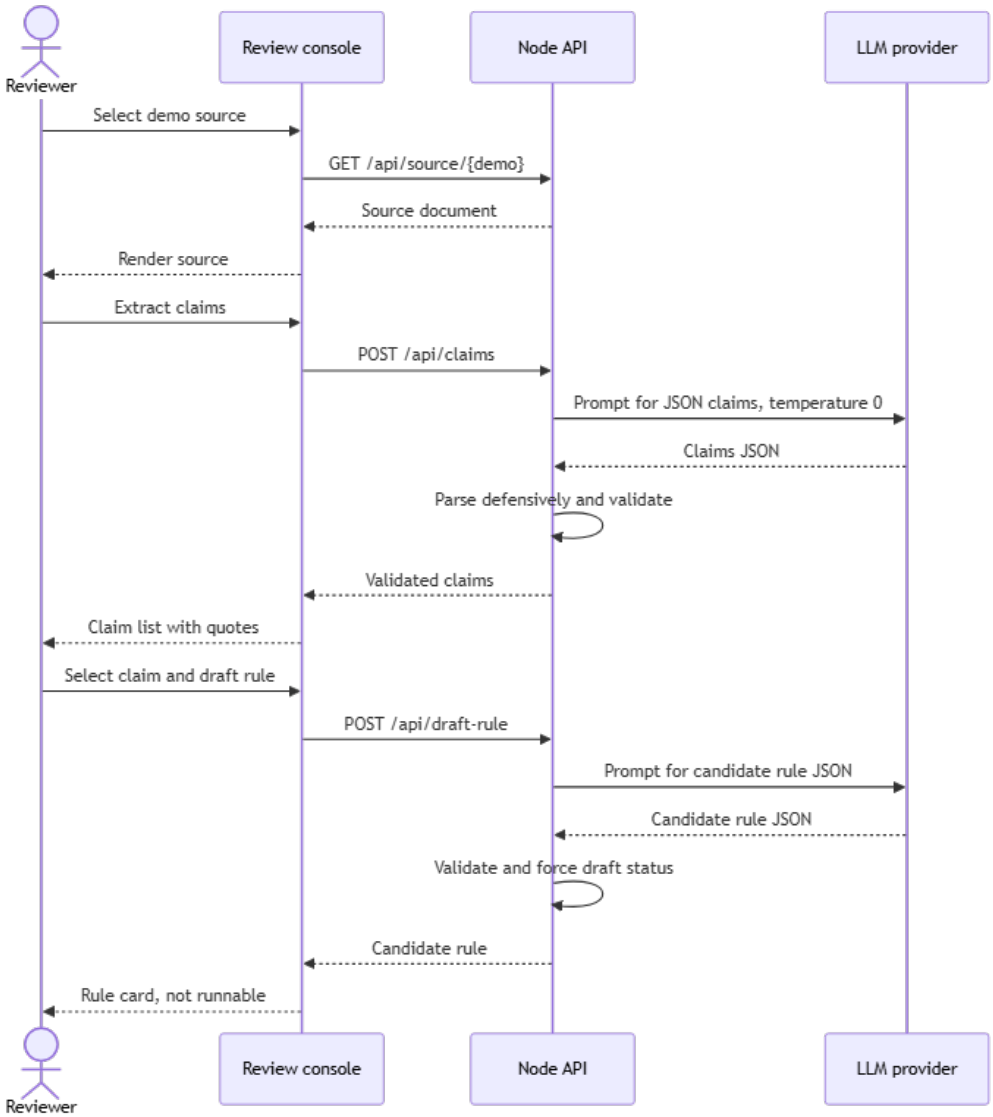

Figure 3: Full LLM authoring flow. The model drafts structure, but the output remains a draft until review and promotion.

The fast walkthrough variant loads the prepared Cardiac CMR Mass Score source, claim, and candidate rule instead of making live provider calls. This path makes the demonstration independent from provider latency, rate limits, or a missing API key.

5.2 Review and Promotion Flow

The reviewer can approve, reject, or request revision; only an approved rule enables promotion. Approval alone, however, is not enough. The artifact must also compile and validate before it remains promoted.

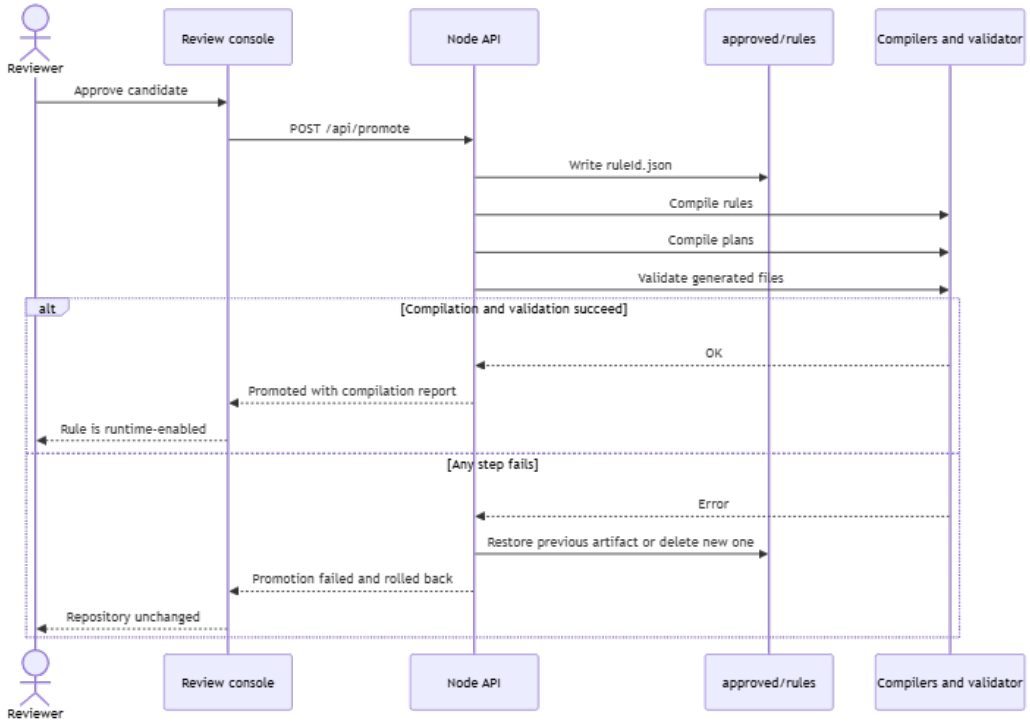

Figure 4: Promotion with rollback. A rule that does not compile is never left behind in the repository.

Rollback matters because otherwise the repository could contain an approved JSON file that does not actually compile. For a small filesystem-based prototype this is a practical consistency guard.

5.3 Runtime Evaluation Flow

Runtime evaluation has two modes. The UI can load expected benchmark traces from `expected/traces/`, which keeps the demo stable and shows the trace contract Jason must satisfy. For custom AgentSpeak facts the backend runs Jason live. The CLI and CI golden-case tests also run the MAS live against all benchmark cases.

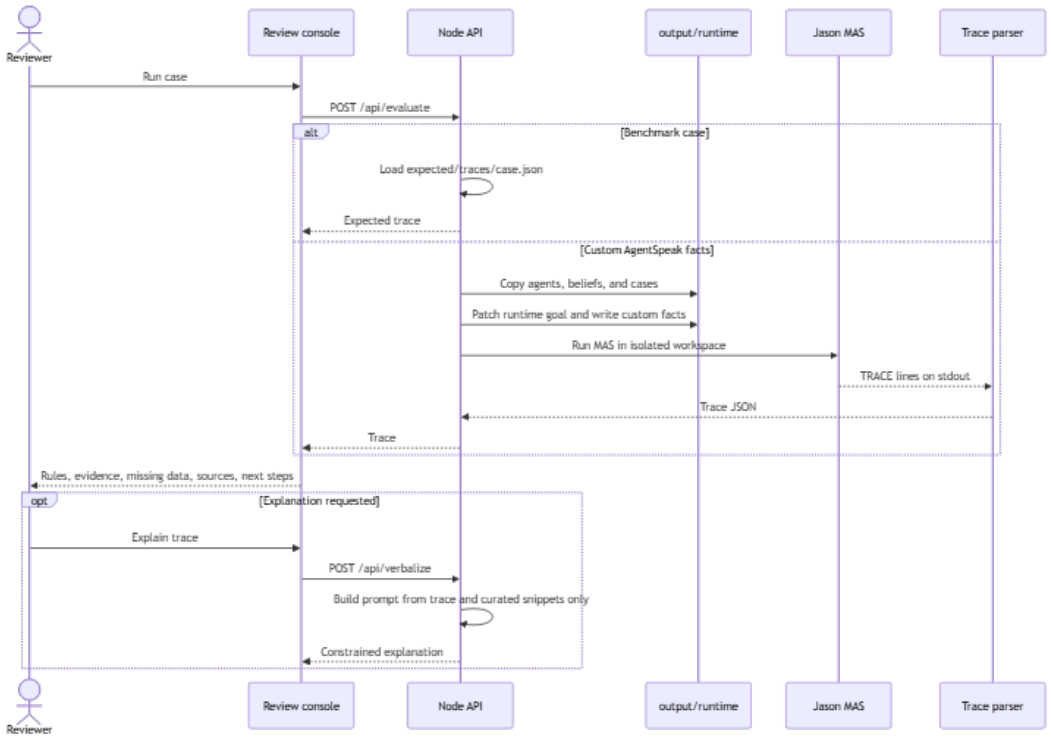

Figure 5: Runtime evaluation. Benchmark cases load a stable expected trace; custom facts run live in an isolated workspace.

The isolated workspace under `output/runtime/` matters: the runner copies `agents/`, `beliefs/`, and `cases/`, patches the copied runtime goal, and writes any custom facts into the copy. The checked-in MAS sources are never modified by user experiments.

6 AgentSpeak and Prolog

Two logic-programming technologies run in the MAS: Prolog-style belief queries for declarative conditions, and AgentSpeak plans executed by Jason for procedural agent behaviour. Both are needed because the compiler-generated plans must mix data checks (Prolog) with imperative communication and control flow (AgentSpeak).

6.1 Why a logic-based runtime

The project's central constraint is that a runtime decision must be explainable by construction. An imperative implementation could produce the same labels, but the explanation would have to be reconstructed separately, and would therefore be a second artifact that can drift from the code that actually ran.

A logic-based runtime avoids this. The condition under which a rule fires is the rule, and it is a readable logical formula. When Jason selects a plan, the reason it selected that plan is exactly the conjunction of literals in the plan context. For this reason, the compilation target of the pipeline is AgentSpeak rather than generated TypeScript.

6.2 Prolog: belief bases and case facts

Prolog appears in Veritrace through the layer Jason inherits from it: belief bases and logical consequence. In practical terms, this is the part of the system where facts are queried and rules are matched against the current case.

The clearest example is `usable_case_data/1`. It is defined in both `agents/case_reasoner.asl` and `agents/trace_guardian.asl`, because Jason agents have separate belief bases. The predicate answers a simple question: does this case contain any evidence that the runtime knows how to use?

Listing 1: Prolog-style belief rule used by the Jason agents.

```
usable_case_data(Case) :- score(Case, Metric, Value).
usable_case_data(Case) :- observed(Case, Modality, Finding).
usable_case_data(Case) :- ct_level(Case, Level).
usable_case_data(Case) :- pet_positive(Case).
usable_case_data(Case) :- processing(Case).
usable_case_data(Case) :- data_breach(Case).
```

The predicate deliberately abstracts over both domains: some clauses refer to cardiac facts, while others refer to GDPR facts. Adding a new evidence kind means adding a clause, not editing a procedural decision function.

The predicate is used as a safety condition rather than as a domain-specific medical or legal rule. `trace_guardian` relies on it to detect the dangerous case of a low-risk conclusion reached with no usable evidence, which is escalated to human review.

Case facts themselves are Prolog-style ground terms loaded from `cases/*.asl`. Domain constants such as thresholds live in belief files, for example `beliefs/cutoffs.asl`, as facts such as `cutoff(cmr_mass_score, 5)`. Keeping cutoffs as facts rather than literals in the rule allows the same compiled rule to be re-evaluated against a different threshold without recompilation.

6.3 AgentSpeak: plans, events, and the BDI cycle

Where Prolog supplies the declarative query layer, AgentSpeak supplies the procedural reasoning layer: what the agent does when a goal is posted, and how work is distributed across agents.

Syntax recap.

`f(a).`	belief / fact
`head :- condition1, condition2.`	inference rule
`+!g : context ← a1; a2.`	AgentSpeak plan
`!goal`	post internal goal
`.send(a, tell, f(1))`	broadcast belief
`true`	always-matching context

Every approved rule is compiled into an AgentSpeak plan for the achievement goal `+!evaluate_case(Case)`:

Listing 2: Generated AgentSpeak plan for one approved rule.

```
+!evaluate_case(Case)
  : score(Case, cmr_mass_score, Score) & cutoff(cmr_mass_score, Cutoff)
    & Score >= Cutoff & approved_rule(cmr_mass_score_above_cutoff)
  <- !emit_conclusion(Case, high, cmr_driven_high_suspicion,
                      cmr_mass_score_above_cutoff);
     !emit_evidence(Case, score(Case, cmr_mass_score, Score));
```

```
    .send(runtime_coordinator, tell, reasoner_done(Case)).
```

The three parts of the plan map onto the JSON artifact. The triggering event is the fixed runtime goal shown in the listing. The context after `:` is a Prolog-style query over the belief base, and it is where the rule's conditions end up. The body after `<-` is the sequence of AgentSpeak actions, and it is where conclusions and used evidence end up.

This mapping is the reason the design works. Plan contexts are declarative and therefore safe to generate from reviewed data, while plan bodies are procedural and therefore restricted to a small fixed vocabulary of goals, such as `!emit_conclusion` and `!emit_evidence`, plus one message send. The compiler never generates arbitrary procedural code.

Three further AgentSpeak features are relevant in the project. First, Jason tries applicable plans in source order; the compiler exploits this to implement deterministic precedence by emitting plans in priority order. Second, speech-act communication through `.send(..., tell, ...)` coordinates the four agents. Third, handwritten and generated code are separated: `case_reasoner.asl` defines only the stable runtime protocol and includes `agents/case_reasoner_generated.asl`.

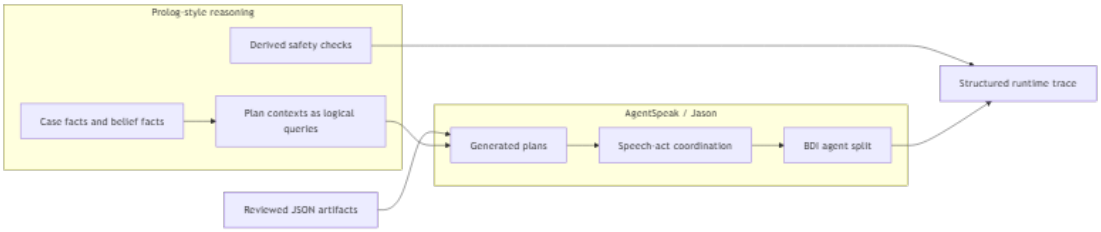

Figure 6: Role of Prolog-style reasoning and AgentSpeak/Jason in the runtime path.

7 JSON to AgentSpeak Compilation

The compiler is the mechanism that turns reviewed data into executable logic. It lives in `tools/mas/compile-rules.mjs` and runs as a single pass over `approved/rules/*.json`, producing three files. The LLM never generates `.asl` files: it generates a candidate JSON rule, the reviewer approves it, the backend stores it as an approved artifact, and only then does the compiler translate it.

7.1 Input artifact

An approved rule stores runtime logic as data. For the CMR Mass Score rule, the relevant part is:

Listing 3: Approved JSON rule excerpt.

```
{
  "ruleId": "cmr_mass_score_above_cutoff",
  "conditions": [
    "score(Case, cmr_mass_score, Score)",
    "cutoff(cmr_mass_score, Cutoff)",
    "Score >= Cutoff"
  ],
  "conclusions": [
    "risk(Case, high)",
    "decision(Case, cmr_driven_high_suspicion)",
```

```
    "activated_rule(Case, cmr_mass_score_above_cutoff)"
  ],
  "usedEvidence": [
    "score(Case, cmr_mass_score, Score)"
  ]
}
```

7.2 Compilation pipeline

The pass has five stages.

1. Selection and validation. Every JSON file in the rules directory is read. Non-runtime artifacts are filtered out; unsafe runtime artifacts abort the compilation rather than producing a partial file. The checks are that the rule is approved and `approvedForRuntime` is true, that the rule id is a valid AgentSpeak identifier, that every condition and conclusion parses as a valid AgentSpeak fragment, that the conclusions include `risk`, `decision`, and `activated_rule`, and that a source mapping exists. Validation is structural: the compiler does not judge whether a rule is clinically or legally correct. That remains the reviewer's job.

2. Priority ordering. Rules are sorted by `rulePriority`. An explicit `compilation.priority` wins; otherwise the compiler derives one from the rule type: `safety_guard` maps to 0, `missing_data_escalation` to 20, `threshold` to 50, and anything else to 100. Ties are broken by rule id, so the output is deterministic and diff-stable. Because Jason attempts applicable plans in the order they appear in the file, this ordering implements visible rule precedence.

3. Derived-predicate hoisting. Rules may declare shared helper predicates under `compilation.derivedPredicates`. The compiler collects these declarations across all rules, merges duplicates by name while accumulating which rules use them, and keeps only predicates used by at least two rules. Each survivor is emitted once at the top of the generated file as a Prolog clause annotated with the rules that depend on it. A helper used by a single rule is intentionally not hoisted, because keeping it inside the rule context preserves local readability.

4. Plan rendering. Each rule becomes one plan through a fixed template. The compiler extracts the risk and decision values from the conclusions, checks that the `activated_rule` conclusion equals the `ruleId`, and then renders the plan. Conditions become the plan context joined with `&`; the compiler appends `approved_rule(ruleId)`; risk, decision, and rule id become a single `!emit_conclusion` goal; each `usedEvidence` entry becomes one `!emit_evidence` goal; and the plan ends by notifying the coordinator with `reasoner_done(Case)`.

5. Emission. Three artifacts are written in one pass, each carrying a do-not-edit header. The generated agent file contains the executable rule plans. The approved-rules belief file contains one `approved_rule(...)` fact per rule, used as the runtime gate. The source-mapping belief file contains one `source_for_rule(...)` fact per rule, so the runtime can report which source supported an activation.

The source id is resolved from `runtimeImplementation.sourceMappingFact` when present, falling back to `source.sourceId`; if neither exists, compilation fails. A rule can therefore never reach the runtime without recorded provenance.

7.3 Compiler implementation

The compiler is the architectural boundary between reviewed artifacts and executable MAS behaviour. Its purpose is not only to translate JSON into AgentSpeak syntax, but to make the transition from human-approved data to runtime logic explicit, deterministic, and inspectable.

Technically, the compiler is a template-based source-to-source transpiler. Its input is not a programming language but a constrained JSON schema; its output is AgentSpeak. There is no AST, no optimisation pass, and no intermediate representation. Each approved rule independently fills a fixed AgentSpeak plan template:

Listing 4: Plan rendering template.

```
+!evaluate_case(Case)
  : {conditions} & approved_rule({ruleId})
  <- !emit_conclusion(Case, {risk}, {decision}, {ruleId});
     !emit_evidence(Case, {usedEvidence});
     .send(runtime_coordinator, tell, reasoner_done(Case)).
```

The template is fixed: conditions, risk, decision, and evidence are slots filled from the JSON artifact; `approved_rule(ruleId)` is injected automatically as the runtime gate.

The main architectural choices are the following.

- **Narrow input language.** An approved rule can express conditions, conclusions, used evidence, source metadata, review state, and compilation metadata. It cannot express arbitrary AgentSpeak plans. This keeps the JSON artifact close to what a reviewer needs to inspect: when the rule applies, what it concludes, which evidence it records, and which source supports it.
- **Narrow output interface.** Each approved rule becomes one AgentSpeak plan for `+!evaluate_case(Case)`. Conditions become the plan context; conclusions become calls to `!emit_conclusion`; and evidence entries become calls to `!emit_evidence`. The generated plan cannot introduce new runtime protocols, new agent messages, or arbitrary side effects.
- **Handwritten architecture, generated rule layer.** The compiler does not generate the MAS architecture. Handwritten AgentSpeak defines how conclusions are sent to the coordinator, guardian, and planner; how evidence is recorded; and when the reasoner reports completion. Generated AgentSpeak only fills one controlled extension point: the domain-rule plans included by `case_reasoner`. The inclusion is explicit: `case_reasoner.asl` ends with `{ include("agents/case_reasoner_generated.asl") }`, and the `.mas2j` project file references only the handwritten agents. The generated content is loaded transparently at startup and is physically separated to keep generated and hand-maintained code reviewable independently.
- **Runtime approval gate.** Approval is represented twice. It exists in the JSON artifact through `reviewStatus` and `approvedForRuntime`, and it is materialised in Jason as `approved_rule(ruleId)`. The compiler injects this belief into every generated plan context, so approval becomes a runtime precondition checked during plan selection.
- **Provenance preservation.** For every compiled rule, the compiler emits a `source_for_rule(ruleId, sourceId)` belief. The guardian can therefore include rule-source mappings in the final trace whenever a rule activates.
- **Deterministic precedence.** Rules are emitted in a stable order based on priority and rule id. This matters because Jason considers applicable plans in source order. The generated order is therefore part of the runtime semantics, and priority gives the project a simple, visible form of rule precedence.
- **Promotion consistency.** Compilation is part of the promotion workflow. After a reviewed rule is copied into the approved artifact area, the system compiles and validates

the runtime artifacts. If compilation fails, promotion is rolled back instead of leaving a broken approved rule in the repository.

The resulting design keeps responsibilities separated: the reviewer authorises a constrained symbolic artifact, the compiler translates it through a fixed template, and Jason executes the resulting logic inside the existing MAS protocol. This is the trust boundary of Veritrace: only reviewed, structurally valid, source-grounded artifacts can enter the runtime.

7.4 Output and approval gate

For the CMR artifact, the generated plan is:

Listing 5: Generated AgentSpeak output.

```
// Rule: cmr_mass_score_above_cutoff
+!evaluate_case(Case)
  : score(Case, cmr_mass_score, Score) & cutoff(cmr_mass_score, Cutoff)
    & Score >= Cutoff & approved_rule(cmr_mass_score_above_cutoff)
  <- !emit_conclusion(Case, high, cmr_driven_high_suspicion,
                      cmr_mass_score_above_cutoff);
     !emit_evidence(Case, score(Case, cmr_mass_score, Score));
     .send(runtime_coordinator, tell, reasoner_done(Case)).
```

The corresponding belief files contain the runtime gate and the source mapping:

Listing 6: Generated belief facts for the approved rule.

```
approved_rule(cmr_mass_score_above_cutoff).
source_for_rule(cmr_mass_score_above_cutoff,
                paolisso_2024_cmr_mass_score).
```

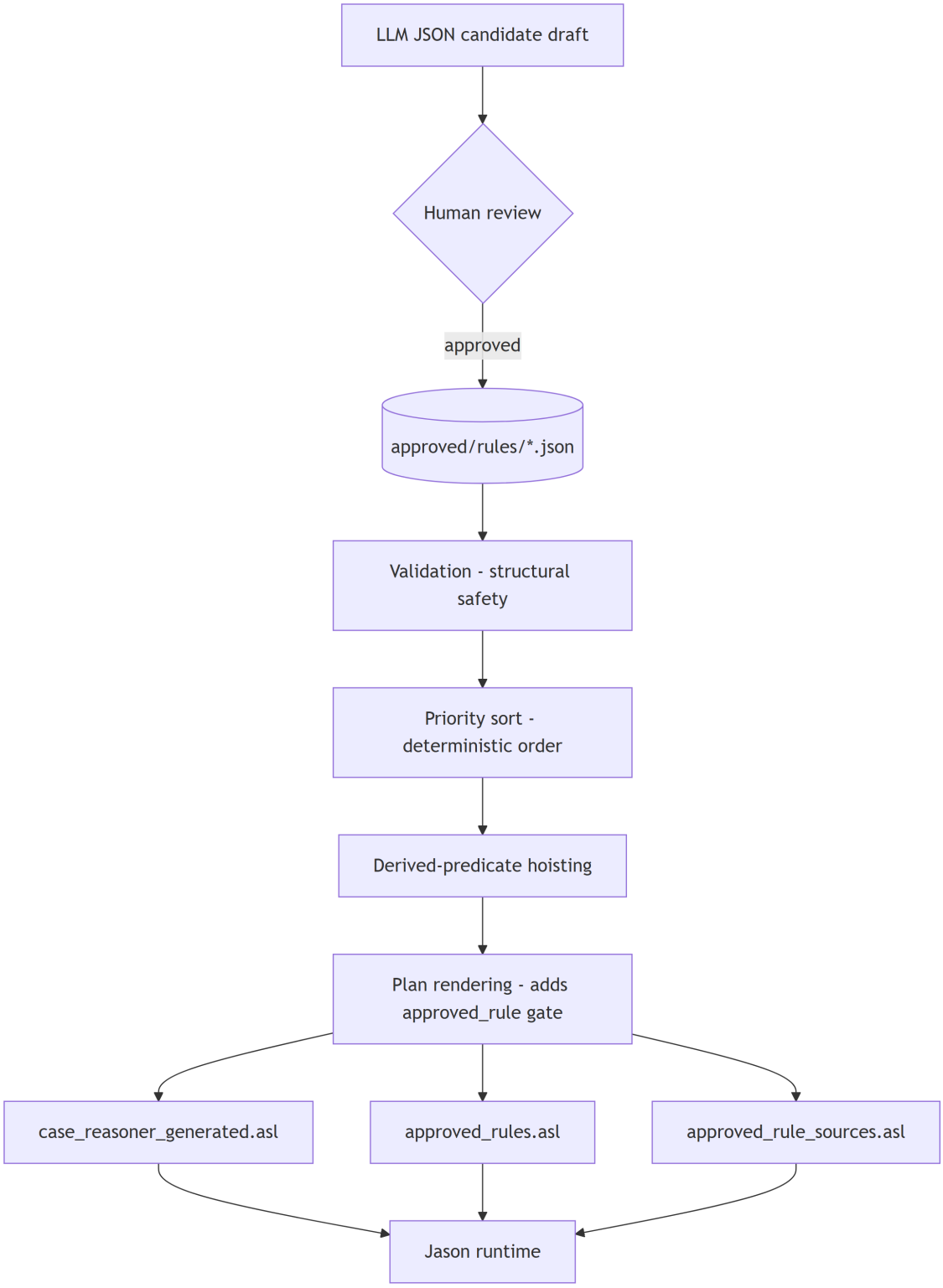

Figure 7: JSON-to-AgentSpeak compilation. Jason includes the generated file, but every rule remains gated by an `approved_rule` fact.

The injected `approved_rule(...)` literal is the most important line the compiler writes. Even if a rule-like plan exists in the generated file, it remains inert unless the corresponding fact is present in the generated belief file. Approval is enforced by runtime logic itself, not only by the promotion procedure that wrote the file.

8 BDI Agents in the MAS

The Jason runtime is small, but it is not a single monolithic agent. It uses a BDI-style split so that each agent has a clear responsibility in the reasoning path. This also makes the trace easier to explain during a demo, because the case visibly moves through reasoning, guarding, planning, and export.

Agent	Role	What it does
`runtime_coordinator`	Orchestrator	Starts `evaluate_case(Case)`, waits for the other agents, and prints the final `TRACE_*` lines consumed by the parser.
`case_reasoner`	Domain reasoner	Applies approved generated rule plans; emits `risk`, `decision`, `activated_rule`, and `used_evidence`.
`trace_guardian`	Safety / meta agent	Checks missing data, verifies activated rules are approved, emits source mappings, detects conflicting risk conclusions, and raises human-review flags.
`care_planner`	Planning agent	Converts decisions and risk levels into next steps. GDPR plans come from approved JSON artifacts; some Cardiac and fallback plans are handwritten.
`cardiac_mass_agent`	Reference artifact	Kept as a mono-agent reference from an earlier project stage; useful for comparison but not the main reviewed runtime path.

The flow is intentionally sequential. The coordinator asks the reasoner to evaluate the case; when the reasoner is done the guardian checks the trace conditions; when the guardian is done the planner adds next steps; only then does the coordinator export the trace. This is simpler than fully autonomous multi-agent negotiation, but it fits the project goal of making the reasoning path inspectable.

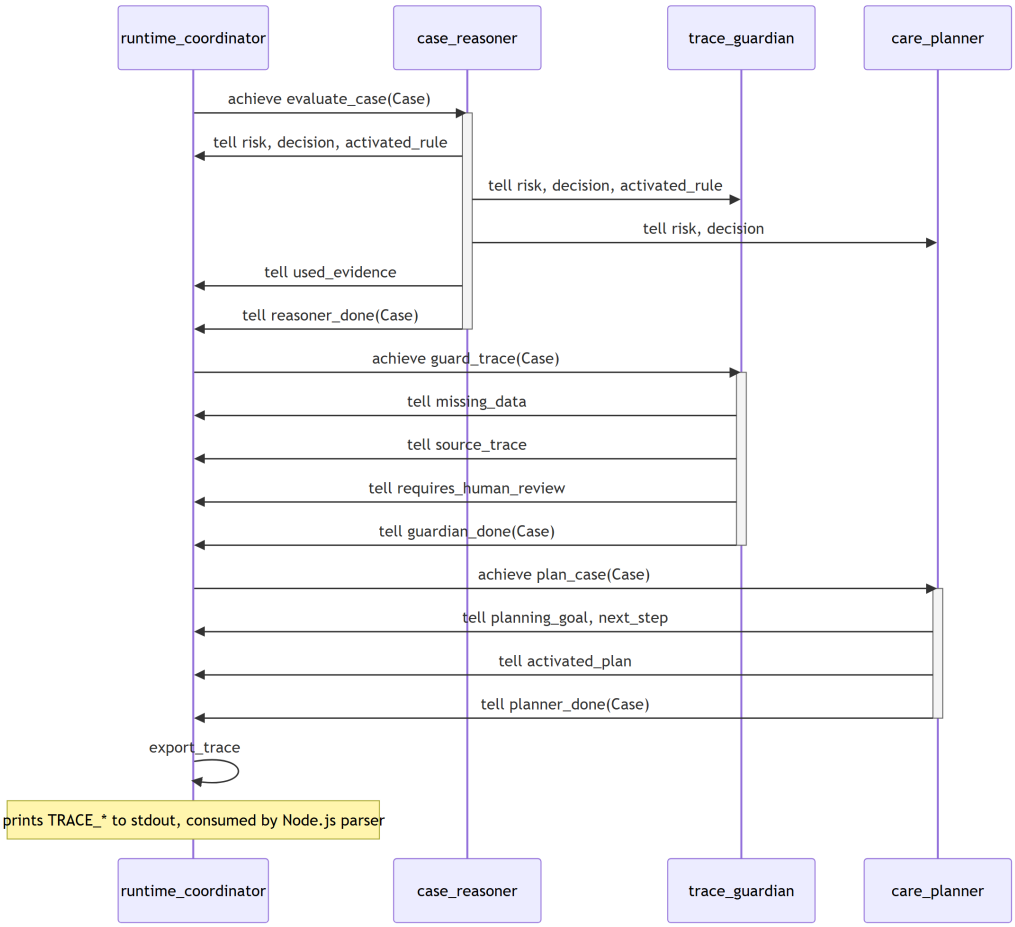

Figure 8: MAS agent communication protocol. The coordinator hands off control sequentially through three agent phases. During the reasoning phase the broadcaster sends conclusions to all other agents simultaneously.

The split also avoids mixing concerns. The rule compiler only needs to generate plans for `case_reasoner`. Source-grounding and approval checks stay in `trace_guardian`. Next-step logic stays in `care_planner`. If one part changes, the rest of the runtime protocol remains stable.

`trace_guardian.asl` is the component that would remain central if the project grew. It does not decide the domain case; it checks the reasoning path around the decision: missing data are recorded explicitly, activated rules must be approved, rule-to-source mappings are emitted into the trace, conflicting risk conclusions require human review, and a low-risk result without usable evidence is treated as unsafe.

The planner layer is intentionally mixed. GDPR next steps are generated from approved JSON plan artifacts, while some Cardiac and fallback plans still live as handwritten AgentSpeak in `care_planner.asl`: insufficient data, CT gray zone, generic high risk, and generic low risk. This keeps the prototype controllable while still demonstrating how approved plan artifacts work.

9 Where the LLM Is Used

The LLM is used in exactly three places: to extract candidate claims from a source document, to draft a candidate rule from a selected claim, and to verbalise an already computed trace.

It is not used while Jason evaluates a case. That separation is the main rule of the project: the model can help write or explain, but it does not approve, promote, compile, or execute runtime knowledge.

The backend also treats provider output defensively. Prompts demand JSON, requests use `temperature: 0`, and the parser recovers from common failures such as fenced JSON or extra text around a JSON object. This does not make the model authoritative; it only makes the authoring path less fragile during a demo.

10 DSPy Harness

The optional `llm-dspy/` module is not part of the runtime. It is used as an evaluation harness for the authoring layer: source document to claims, and claim to candidate rule. The motivation was that plain prompt files are hard to evaluate systematically. A textual prompt can look good, but it is difficult to tell whether a change actually improved source grounding, JSON validity, or rule safety.

Classic prompt/template approach	**DSPy approach used here**
Prompt text is edited manually and judged mostly by inspection.	Tasks are represented as signatures with explicit inputs and outputs.
A response can sound good but violate the artifact contract.	Metrics score concrete constraints such as JSON validity, source id match, quote match, draft status, and AgentSpeak fragment validity.
Prompt changes are hard to compare over time.	The same examples can be rerun in fixture or live mode to compare behaviour.
The prompt mixes instructions, examples, and evaluation criteria.	Signatures, examples, and metrics are separated, so each part has a clearer role.

The metrics are deliberately traceability-focused rather than fluency-focused. Claim extraction is scored on `sourceId`, claim presence, exact quote match, claim type, review requirement, and candidate meaning. Candidate rule drafting is scored on `reviewStatus = draft`, `approvedForRuntime = false`, source and quote match, condition/conclusion overlap, valid AgentSpeak-like fragments, missing-data behaviour, and human-review requirement.

The authoring layer should not be optimised for fluent prose alone. It should be optimised for producing artifacts that remain reviewable, source-grounded, and safe to reject before runtime. DSPy was a first step toward measuring that behaviour instead of relying only on hand-written templates and manual inspection.

11 Implementation

11.1 Tech Stack

Part	**Technology**	**Why it is there**
Review console	React, Vite, TypeScript, Tailwind CSS	Guided browser UI for the demo.

Backend API	Node.js 22, Express, Zod	LLM calls, validation, promotion, runtime endpoints.
Symbolic runtime	Jason, AgentSpeak, Java 21+	Deterministic rule and plan execution.
Tooling	Node scripts, Gradle wrapper	Compile artifacts and run Jason locally.
Tests	Node test runner, Playwright	Artifacts, MAS cases, traces, UI integration flows.
LLM providers	Groq, OpenRouter	Claim extraction, rule drafting, trace verbalisation.
Optional evaluation	DSPy	Experiments on LLM authoring quality.
Deployment	Docker Compose	Reproducible local run.

The frontend runs on http://127.0.0.1:5173 and the backend on http://127.0.0.1:8787. During development Vite proxies API calls to the backend; provider calls stay server-side.

11.2 Repository structure

The repository is a monorepo because the UI, backend, artifacts, MAS runtime, and tests must evolve together.

Listing 7: Repository structure.

```
app/review-console/   React frontend and Node backend API
approved/rules/       Approved rule JSON artifacts
approved/plans/       Approved plan JSON artifacts
agents/               Handwritten and generated AgentSpeak agents
beliefs/              Generated belief files and source mappings
cases/                AgentSpeak benchmark/custom facts
expected/traces/      Golden JSON traces
tools/mas/            Validation, compilation, isolated runtime execution
tools/trace/          Jason trace parsing and trace validation
llm-dspy/             Optional DSPy harness
output/               Local traces, reports, and temporary workspaces
```

The backend exposes endpoints for the main workflow: health checks, claim extraction, rule drafting, approved artifact lists, promotion, compilation, trace loading, custom case evaluation, audit events, and trace verbalisation. The API is not a separate product surface; it exists to support the review console and the validation scripts.

Most objects are plain JSON, so candidate rules, approved rules, plans, traces, and audit details can all be inspected without custom tooling. Zod covers the review-console API contracts, while approved rule and plan artifacts also have custom Node validators under `tools/mas/`.

Trace export is deliberately simple. `runtime_coordinator.asl` prints parser-friendly lines such as `TRACE_CASE=`, `TRACE_RISK=`, `TRACE_ACTIVATED_RULES=`, and `TRACE_HUMAN_REVIEW=`, and the Node trace parser reads them back into JSON. The mechanism favours debuggability and reproducibility over a more complex integration protocol.

The main consistency risk is drift: if `approved/rules/` or `approved/plans/` changes, the generated AgentSpeak, belief files, and expected traces can become stale. The project handles this with validation commands, generated-artifact checks, MAS golden cases, and rollback during promotion.

12 Testing

Testing follows the same boundary as the design: deterministic parts are tested automatically, while parts depending on a live provider are either optional or covered by the fast walkthrough.

The automatic checks cover API schemas and custom JSON validators; compilation of approved artifacts into AgentSpeak and belief files; generated-artifact consistency; Jason MAS golden cases compared with expected traces; trace parser behaviour; Playwright integration tests for the guided UI and runtime API; and optional LLM end-to-end runs when a provider key is configured.

Listing 8: Main validation commands.

```
npm test
npm run validate:rules
npm run validate:plans
npm run validate:compilation
npm run compile
npm run test:mas
npm run validate:traces
npm run test:e2e:smoke
npm run test:e2e
```

For the review console:

Listing 9: Review-console validation commands.

```
npm --prefix app/review-console run typecheck
npm --prefix app/review-console run build
npm --prefix app/review-console run test:e2e:smoke
```

The integration tests avoid real LLM calls. They cover the fast walkthrough, GDPR runtime traces, custom AgentSpeak facts, next-step rendering, promotion rollback, duplicate promotion handling, and the approve-promote-evaluate path, giving the demo a reliable baseline even when the provider is unavailable.

CI repeats the same idea with Node.js 22, Java 21, and Python 3.12: it validates artifacts, builds the review console, runs Jason golden cases, checks generated files, and runs Playwright integration tests.

13 Deployment

Direct execution needs Node.js 22, npm, Java 21 or newer, and Git. Docker is optional but useful when the Node and Java setup should be isolated.

Listing 10: Local startup.

```
cd app/review-console
npm install
copy .env.example .env
npm run dev
```

The `.env` file holds provider configuration:

Listing 11: Provider configuration.

```
LLM_PROVIDER=groq
```

```
LLM_API_KEY=
LLM_MODEL=qwen/qwen3.6-27b
PORT=8787
```

Docker Compose offers three services: `tests`, `mas`, and `review-console`.

Listing 12: Docker Compose commands.

```
docker compose --env-file "app/review-console/.env" up -d --build review-console
docker compose run --rm tests
docker compose run --rm mas
docker compose down
```

14 Conclusion

Veritrace delivers a complete local workflow from source text to verified AgentSpeak trace. The four layers are each responsible for one thing: the LLM drafts from source; the reviewer approves or rejects; the compiler validates the artifact and transforms it into AgentSpeak; the Jason runtime executes it and emits a trace with provenance. No LLM output reaches the runtime without passing through human review and compilation validation.

Two design decisions make this different from a direct LLM prompt. First, the approval gate: the reviewer's `approved_rule` fact is a runtime condition, not just a UI flag — without it the plan context fails and the rule never fires. Second, provenance is built into the compilation: every rule carries its `source_for_rule` belief, so the trace guardian can map activated rules back to the source claim without relying on the LLM's memory.

The prototype works on curated examples. The artifact store is the filesystem, not a governance database — there are no authenticated reviewers, versioned rollbacks, or audit logs. The full LLM path depends on provider quality, and the DSPy harness is a prototype that needs more examples and stricter source-grounding metrics before it can guide model selection.

The main limitations suggest the next steps. Replacing the filesystem artifact store with a lightweight database would give authenticated reviewers, versioned artifacts, and audit history — all currently absent. The DSPy harness could also be exploited more thoroughly for systematic evaluation of LLM authoring quality across providers and prompts, replacing the current ad-hoc testing.

References

[1] Liangming Pan, Alon Albalak, Xinyi Wang, and William Wang. 2023. *Logic-LM: Empowering Large Language Models with Symbolic Solvers for Faithful Logical Reasoning*. Findings of the Association for Computational Linguistics: EMNLP 2023, pages 3806–3824. DOI: 10.18653/v1/2023.findings-emnlp.248.

[2] Giovanni Ciatto, Gianluca Aguzzi, Riccardo Battistini, Martina Baiardi, Samuele Burattini, and Alessandro Ricci. 2025. *Exploiting GenAI for Plan Generation in BDI Agents*. Frontiers in Artificial Intelligence and Applications, ECAI 2025, pages 3495–3502. DOI: 10.3233/FAIA251223.