PV-PP Agent Governance Sidecar White Paper

From Available to Viable: A Packet-Driven Gate for Proposed Agent Actions

Version 0.1 — Public Draft

Prepared by Amundsen R&D LLC

Summary

This paper describes a working proof-of-concept for a PV-PP sidecar-style governance gate for agent actions. The prototype is not an email tool and not a production security layer. It is a common packet-driven runtime that accepts scenario-defined governance rules, evaluates proposed actions before execution, blocks or defers actions that fail governing predicates, and returns a trace explaining the decision. The first demonstration scenario is batch email attachment sending, where the sidecar blocks only proposed sends containing an Amundsen R&D confidentiality marker while allowing clean sends to proceed.

Core claim: tool layers tell an agent what is available; a governance sidecar asks whether a proposed action is admissible before it happens.

Contents

1. The Agent-Governance Problem

Agentic systems increasingly operate by selecting and invoking tools, APIs, routes, or workflow transitions. A tool registry can tell an agent what is technically available. That is necessary, but it is not sufficient. Availability does not establish that a proposed action is safe, authorized, adequate, recoverable, or valid in the current scenario.

The missing question is not merely: Can the agent call this tool? The missing question is: Should this proposed action be allowed to happen now, given the scenario, evidence, state, constraints, recovery obligations, and governing domains?

Layer	Question answered	Limitation
Tool registry / capability layer	What tools or actions are available?	Does not by itself determine whether a proposed action is admissible.
Ordinary scoring or optimization layer	Which available action scores best?	Can rank a high-throughput but inadmissible action above a viable one if used too early.
PV-PP sidecar governance layer	Is this proposed action admissible before execution?	Requires governance rules and scenario structure to be defined before runtime.

2. The Sidecar Idea

A sidecar is a separate governance component that sits beside or between an agent/workflow and the tool/action layer. The agent proposes an action. The sidecar evaluates that proposed action before execution. The sidecar returns a decision such as PROCEED, BLOCK, ESCALATE, or DEFER, together with a trace showing which predicates passed or failed.

```
Agent / workflow proposes action
        |
        v
PV-PP sidecar evaluates proposed action packet
        |
        v
Decision: PROCEED | BLOCK | ESCALATE | DEFER
        |
        v
Runtime/tool layer executes only actions that pass the gate
```

The sidecar pattern is deliberately modest. It does not require replacing the agent or the tool layer. It adds a pre-execution admission gate where governance rules can be evaluated before the action occurs.

3. What PV-PP Adds

PV-PP supplies a method for defining the governance rules that the sidecar evaluates. The method begins before runtime: identify the relevant Productive Powers, domains, constraints, state-validity checks, evidence and authority requirements, recovery obligations, and admissibility predicates for the scenario.

PV-PP governance element	Runtime role
Productive Powers	Capacities required to perform or authorize a proposed action.
Domains	Typed fields or conditions through which the powers are evaluated.
Governing domains	Domains whose failure cannot be offset by throughput, convenience, or score.
State-validity checks	Tests for stale, missing, conflicting, or untrusted state.
Evidence/authority checks	Tests that the sidecar has enough basis to allow the action.
Recovery requirements	Conditions where prevention or rollback must exist before execution.
Admissibility predicates	The pre-scalar gate that determines whether scoring or execution can proceed.

The sidecar is therefore not just a scoring service. Its central job is to construct and evaluate an admissible set of proposed actions. Scoring, if used at all, operates only after admissibility is established.

4. Common Runtime, Scenario-Specific Packet

The important implementation move is to avoid building a different sidecar for every application. The proof-of-concept uses a common packet-driven runtime. The runtime knows how to iterate through declared actions, declared rules, predicate sets, decision policies, scoring policies, and trace requirements. The scenario packet supplies the application-specific meaning.

Common runtime responsibility	Scenario packet responsibility
Load and validate the packet.	Define the scenario, interaction, and candidate actions.
Evaluate declared rules.	Define Productive Powers, domains, and predicate sets.
Map predicate failures to decisions.	Define BLOCK / ESCALATE / DEFER / PROCEED policy.
Score only admissible actions when scoring is enabled.	Define optional scoring weights or disable scoring.
Return trace and decision packet.	Define what trace fields matter for review.

The email demonstration is therefore not hard-coded email logic. It is the first scenario packet for a common runtime.

5. Architecture of the Proof-of-Concept

The current proof-of-concept has three main artifacts:

Artifact	Purpose
Generic sidecar runtime prototype	A Python program that loads a packet, evaluates declared rules, returns per-action decisions, and can run as an HTTP server.
Batch email scenario packet	A strict JSON file defining the first demonstration scenario, rule set, candidate actions, decision policy, and scoring policy.
Volume Five implementation supplement	A documentation chapter explaining the rudimentary sidecar, the generic runtime, the commands, and the sample programs.

```
python3 PV-PP_Generic_Sidecar_Runtime_Prototype_v0_1_...py   --packet-json PV-
PP_Batch_Email_Attachment_Send_Scenario_Packet_JSON_v0_2_...json   --pretty
```

The same runtime can also run as a local HTTP sidecar server:

```
python3 PV-PP_Generic_Sidecar_Runtime_Prototype_v0_1_...py --serve --port 8081
```

6. Demonstration Scenario: Batch Email Attachment Sending

The first scenario is intentionally concrete. An agent receives a group of files and breaks them into multiple outbound emails because the full set cannot be sent in one message. The sidecar evaluates each proposed send action independently before the email tool is allowed to execute.

The governing rule is not "block every document containing the word confidential." That would be too crude. Law firms, contracts, and many ordinary business documents may use the word confidential in legitimate contexts. The scenario instead uses a proximity rule: block proposed sends where an attachment contains an Amundsen R&D term near a confidentiality term.

Proposed send	Attachment content pattern	Decision
send_email_001	Agenda and receipt; no Amundsen R&D confidentiality marker.	PROCEED
send_email_002	Draft strategy document containing Amundsen R&D near confidential/internal-only.	BLOCK
send_email_003	Legal memo uses "confidential" generically without Amundsen R&D proximity.	PROCEED

The proximity rule is scenario-defined, not hard-coded into the runtime:

```
rule_type: proximity_match_absent
terms_a: ["amundsen r&d", "amundsen rd", "amundsen r and d"]
terms_b: ["confidential", "restricted", "internal only"]
window_tokens: 8
failure_decision: BLOCK
```

7. Example Output and Interpretation

The observed proof-of-concept result is:

```
batch_decision: PARTIAL_PROCEED_WITH_BLOCKS
proceed_actions: send_email_001, send_email_003
blocked_actions: send_email_002
naive_scalar_winner: send_email_002
blocked_scalar_winner: true
```

This result is the point of the demonstration. The action with the highest ordinary scalar score is not allowed to proceed because it fails a governing predicate. Clean actions proceed. The blocked action is blocked with a predicate-level trace showing which rule matched and which attachment triggered the violation.

8. What the Prototype Proves and Does Not Prove

It proves	It does not prove
A common runtime can evaluate scenario-defined governance rules without hard-coding the application scenario.	It is not a production security system.
Proposed actions can be gated before tool execution.	It does not perform real document extraction, DLP, authentication, or TLS.
Violating actions can be blocked while clean actions proceed.	It does not guarantee that all confidential content will be detected.
Traceable decisions can be returned to a caller or reviewer.	It does not replace legal, security, or compliance review.

The prototype is best understood as an architectural proof-of-concept. It demonstrates the sidecar pattern and the packet-driven runtime shape. Production use would require a real document classifier, identity and authority controls, secure deployment, logging, policy governance, and integration with the actual tool layer.

9. Implementation Path

A practical implementation path is staged:

Stage	Purpose
Stage 1: Static sidecar gate	Evaluate declared rules over proposed actions, return decision and trace.
Stage 2: Scenario packet library	Add more packets: email, file routing, API calls, workflow approvals, reimbursement review.
Stage 3: Real classifiers and state sources	Replace text_sample fields with real extraction, metadata, DLP, and state-validation services.
Stage 4: Tool proxy integration	Force selected tool calls through the sidecar before execution.
Stage 5: Runtime governance layer	Add persistence, obligation tracking, recovery status, and human review queues.

10. Conclusion

The PV-PP sidecar proof-of-concept shows a practical way to move agent governance from abstract policy language into a runnable pre-execution gate. The key design is not an email filter. The key design is a common packet-driven runtime that evaluates scenario-defined governance predicates before proposed actions are executed.

That structure supports the central distinction: availability is not viability. An agent may have an available tool and still lack an admissible action. The sidecar pattern gives the system a place to ask that question before the action happens.