

ASP: An Operational Pre-Mortem Skill for LLM Coding Agents

An Experience Report

Carlos Ulisses Flores

Codex Hash Research Laboratory

ORCID: 0000-0002-6034-7765

Codex Hash Research Report 2026-01

Date: November 17, 2026

Abstract

LLM coding assistants often produce *locally correct* code that omits failure modes a senior engineer would surface in a pre-mortem, and production then breaks in ways that are expensive to debug. We call these missing items *shadow points*. This experience report describes Anticipating Shadow Points (ASP), a 13-phase planning protocol shipped as a Claude Code skill that integrates Klein's (2007) pre-mortem with the multi-agent failure taxonomy of Cemri et al. (2025), checks the plan with a validator that does not share the planner's context, following the self-correction-limit literature (Huang et al., 2024; Tyen et al., 2024), and, in an opt-in mode, hands execution to a child subprocess with a file-based contract. We document the design, its deployment as an open-source plugin, and preliminary observations from five engineering evaluations, read by the author without blinding and with transcripts not preserved. A pre-registered study then measured the protocol on six public repositories: 90 trials on Claude Opus 5.5, reference risks listed blind by three models of other vendors, and two blind judges. ASP covered 48% of the reference risks against 66% for plain Claude Code asked to list the ways the change could go wrong before planning. It was behind on all six tasks: no measurable benefit under the rule fixed in advance. That single request raised coverage over a plain planning request from 56% to 66% ($p = .001$), a pre-registered secondary comparison that a confirmatory study, reported separately, has since tested as its primary question.

Keywords: LLM coding agents; pre-mortem; multi-agent failure modes; Claude Code; operational protocols; pre-registration

Resumo

Assistentes de código baseados em LLM costumam produzir código *localmente correto* que omite modos de falha que um engenheiro sênior levantaria num pré-mortem, e a produção quebra de formas caras. Chamamos esses itens ausentes de *shadow points*. Este relato descreve o Anticipating Shadow Points (ASP), um protocolo de planejamento em 13 fases distribuído como *skill* do Claude Code, que integra o pré-mortem de Klein (2007) à taxonomia de falhas multiagente de Cemri et al. (2025), confere o plano com um validador que não compartilha o contexto do planejador, conforme a literatura sobre limites da autocorreção (Huang et al., 2024; Tyen et al., 2024), e, num modo opcional, entrega a execução a um subprocesso filho com contrato em arquivo. Documentamos o desenho, a implantação como *plugin* de código aberto e observações preliminares de cinco avaliações de engenharia, lidas pelo autor sem cegamento nem transcrições preservadas. Um estudo pré-registrado mediu o protocolo em seis repositórios públicos: 90 rodadas com o Claude Opus 5.5, riscos de referência listados às cegas por três modelos de outros fornecedores e dois juízes cegos. O ASP cobriu 48% dos riscos de referência, contra 66% do Claude Code simples com o pedido de listar, antes de planejar, o que pode dar errado. Ficou atrás nas seis tarefas: nenhum benefício mensurável pela regra fixada antes. Esse pedido elevou a cobertura sobre um pedido simples de planejamento de 56% para 66% (p = ,001), uma comparação secundária pré-registrada que um estudo confirmatório, relatado à parte, já testou como pergunta principal.

Palavras-chave: agentes de código LLM; pré-mortem; modos de falha multiagente; Claude Code; protocolos operacionais; pré-registro

Contents

1. Introduction 5
1.1. Contributions 7
2. Background 8
2.1. Pre-mortem and prospective hindsight 8
2.2. Multi-agent failure modes (MAST) 9
2.3. Self-correction limits and validator isolation 10
2.4. Closest related work: PreFlect 11
2.5. Classical software and quality-engineering lineage 12
3. The ASP Protocol 16
3.1. Overview 16
3.2. Architectural positioning 17
3.3. Iron Laws 17
3.4. The micro-TODO contract 19
4. Operational Deployment 20
5. Preliminary Observations 22
5.1. The evaluation set 22
5.2. What was observed in real dispatches (author, non-blind) 23
5.3. What the deterministic adjudicator shows about itself 24
5.4. The claude -p exit-code finding 26
5.5. A pre-registered measurement (2026-11) 28
6. Design Hypotheses 31
7. Threats to Validity 32
8. Limitations and Future Work 34
9. Conclusion 35
10. Declarations 37
Acknowledgments 38
References 39
Appendix A — Citation verification status 45
Appendix B — Evaluation set and reproducibility 46
Technical Note 49
About the Author 51

1. Introduction

In the author's practice, routine engineering work — adding a column to a production database, deploying a function that calls an external API, refactoring a utility used across thirty files — is now often handed to AI coding assistants. When the assistant gets it right, the engineer's day is freed. When it gets it *subtly* wrong — by writing code that compiles, passes its tests, ships, but quietly omits a step a senior engineer would have flagged in advance — production breaks in ways that are expensive to debug. Decision science has a technique for exactly that gap, the pre-mortem (Klein, 2007), and agent research now has a catalogue of how multi-agent LLM systems fail (Cemri et al., 2025). Our thesis is that the two belong together at runtime: before writing code, an agent should be made to run a pre-mortem against a fixed failure taxonomy, and its plan should be checked by a validator that cannot see the planner's reasoning.

The failure mode of LLM coding agents that motivated ASP is not *bad* code. It is *incomplete* code: the agent does what was asked, but not what someone with deep domain context would have asked themselves *before starting*. The replica lag during a schema migration. The CORS header on the edge function, which tells browsers which other sites may call it. The row-level security (RLS) policy, which limits the rows each user can read and needs updating to match the new column. We call these missing items **shadow points** — failure modes obvious to a practitioner who has been burnt by them, invisible to an agent that has not.

Two streams of literature converge on this gap. The first, in decision science, is the **pre-mortem**: a structured exercise in which a team imagines a project has already failed and enumerates causes before acting. The technique was operationalized for project teams by Klein (2007) and grounded in the earlier empirical work of Mitchell, Russo and Pennington (1989) on prospective hindsight; Klein (2007) summarises it as raising the ability to identify reasons for a future outcome by about

30%, a figure we could not re-derive from the 1989 paper (Appendix A). The second, in agent research, is the recent characterization of multi-agent failure modes by Cemri et al. (2025), who derived a 14-mode taxonomy, the Multi-Agent System Failure Taxonomy (MAST), from 150 seed traces of multi-agent LLM systems (inter-annotator agreement $\kappa = .88$) and released a corpus of 1,600+ annotated traces. Critique of an agent's plan before it acts already exists (Section 2.4); what we did not find is the two streams integrated as a **runtime discipline**, with named artifacts and a validator that does not share the planner's context, that a coding agent can be made to follow on every non-trivial task.

This paper reports on **Anticipating Shadow Points (ASP)**, a 13-phase planning protocol shipped as a Claude Code skill that combines pre-mortem reasoning with the MAST 14-mode checklist, separates planning from validation under prompt isolation, and, in its opt-in --goal mode, hands execution to a child subprocess with a file-based contract (by default execution stays in the user's session). We deployed the skill, distributed it through three install paths (plugin marketplace, dev mode, standalone script), ran it through five engineering evaluations, and report the design and what the evaluations can and cannot show. These five evaluations are preliminary and unmeasured: GREEN (the agent given the ASP discipline) on five tasks and RED (asked to plan without it) on three, one dispatch each, a single model, read by the author without blinding, with the raw transcripts not preserved. We report them as qualitative observation. One pre-registered measurement came later (Section 5.5): on six tasks, the planning phases of ASP covered fewer reference risks than a one-sentence request. The four design choices we treat as hypotheses, and a fifth measured in the companion paper, are in Section 6.

1.1. Contributions

This experience report contributes:

1. **The ASP protocol as documented design** — thirteen phases, each emitting a named artifact, organised around the principle that no implementation begins before shadow points have been surfaced.
2. **Twelve operational constraints we call "Iron Laws"** (Section 3.3), each tied to a primary literature source or to a documented design decision. Two have a shell-level check in the runner template the agent is told to run (Iron Law 12's JSON output, a template default the operator can override, and Iron Law 9's ASP_IN_GOAL flag, set for the child so that ASP does not start again inside it, whose check at entry is a Bash snippet the skill tells the agent to run); the other ten are prompt-level instructions.
3. **Preliminary, qualitative observations** from five engineering evaluations, and a deterministic adjudicator with cryptographic provenance, exercised on constructed fixtures (sample plans the author wrote to test it) and ready to score real transcripts without trusting the author's adjudication (the expected lists are still the author's, and the optional synonym sidecar is not provided for any task).
4. **A separate empirical finding** on the exit-code semantics of claude -p (Claude Code run non-interactively: one prompt in, one result out) under graceful agent refusal, characterised in the companion paper Flores (2026b). The finding motivated two of the Iron Laws but warrants independent characterization at larger N.

This is an **experience report** in the systems sense, not a benchmark paper. We document a system we built, a methodology we deployed, and what we observed; we do not claim a robust empirical

contribution to the LLM-agents benchmark literature. Section 7 enumerates the threats to validity in detail.

2. Background

We review four threads of recent prior work that directly inform ASP's design (§§2.1–2.4), then situate ASP within the older software- and quality-engineering disciplines it transposes (§2.5). We keep the recent-work review short relative to typical agent-systems papers, because the contribution here is operational and the literature context is in service of that.

2.1. Pre-mortem and prospective hindsight

The pre-mortem technique asks a team to imagine, before commitment, that a project has already failed catastrophically, and to enumerate causes. The empirical basis is Mitchell, Russo and Pennington (1989), who showed that imagining a future event as certain (rather than uncertain) yields explanations that are longer and more episodic; Klein (2007) summarises the effect as about a 30% gain in correctly identifying reasons for a future outcome. Klein (2007) operationalized the finding as the project-team "pre-mortem". Its experimental base is thin: we found no replication of the roughly 30% effect that Klein (2007) attributes to Mitchell et al. (1989), and no meta-analysis of it, in Crossref, arXiv or a general web search (November 2026). That is an absence in those indexes, not proof that none exists, and it says nothing about experiments on other effects of the pre-mortem. Risk-assessment standards catalogue neighbouring techniques — scenario analysis and the structured what-if technique (SWIFT) — but the table of contents of IEC 31010:2019, Annex B, does not list the pre-mortem by name. The term *prospective hindsight* has also been reused for an unrelated reinforcement-learning signal (J. Zhang et al., 2026); we use it only in Mitchell et al.'s sense. Criticism of an agent's plan before it acts already exists (PreFlect;

Wang et al., 2026; §2.4). What we did not find is the integration ASP makes: a pre-mortem run against a fixed failure taxonomy, recorded as named artifacts and reviewed by a validator that does not share the planner's context.

2.2. Multi-agent failure modes (MAST)

LLM agents interleave reasoning and acting in a single loop (Yao et al., 2023), and are increasingly composed into multi-agent workflows whose failure modes compound. Cemri et al. (2025) introduce **MAST** with **MAST-Data**, a corpus of 1,600+ annotated traces from seven popular multi-agent frameworks such as AG2 (formerly AutoGen; Wu et al., 2023). From rigorous analysis of 150 seed traces (inter-annotator agreement $\kappa = .88$), they derived fourteen failure modes organised into three categories: specification and system design (5 modes), inter-agent misalignment (6 modes), and task verification and termination (3 modes); the tables of Section 3 cite these modes by their codes, FM-1.1 to FM-3.3. The 14-mode list is the empirical grounding of ASP's failure-mode checklist: ASP adopts the mode names verbatim and operationalizes each into a pre-mortem prompt plus mitigation in its companion skill files. Two concurrent works extend the taxonomy direction: Zhu et al. (2025) introduce AgentDebug and the AgentErrorTaxonomy (memory / reflection / planning / action / system-level), and S. Zhang et al. (2025), an ICML 2025 Spotlight, opens **automated failure attribution** with the Who&When dataset. MAST is therefore one taxonomy among several: TRAIL formalises a taxonomy of agentic errors over 148 human-annotated traces (Deshpande et al., 2025), AgentRx annotates 170 failed trajectories with a critical step and a category from a grounded-theory taxonomy (Barke et al., 2026), and AgenTracer reports that current reasoning LLMs attribute failures in multi-agent traces with accuracy generally below 10% (G. Zhang et al., 2025). ASP uses MAST as a prompt

library for anticipating failures, not as a classifier of observed ones, so the choice of taxonomy bounds what Phase 2 asks about rather than what it can detect.

2.3. Self-correction limits and validator isolation

The case for separating the validator from the planner rests on three primary works that find LLMs unreliable at critiquing themselves under shared context. These findings motivate, but do not establish, the hypothesis that separating the validator's prompt from the planner's transcript improves validation in ASP. Huang et al. (2024, ICLR) show that intrinsic self-correction without external feedback often degrades performance. Tyen et al. (2024) decompose self-correction into localization and correction and show empirically that LLMs cannot *localize* their own reasoning errors but can correct them given the location externally — *localization is the bottleneck*. Zheng et al. (2023, NeurIPS D&B) document that strong-LLM judges over-favour same-family-model outputs (a "self-enhancement bias") even when objectively judging quality. Kamoi et al. (2024) survey this literature and conclude that reliable self-correction of reasoning errors requires external feedback or verifiers rather than the model's unaided judgment; intrinsic self-feedback methods such as Self-Refine (Madaan et al., 2023) improve some generation tasks but do not close the localization gap that Tyen et al. (2024) isolate. The mechanism we assume, and do not test here (Section 6), is that a validator sharing context with a planner inherits the planner's confidence calibration on shared content, including erroneous content the planner is confident about. ASP's design responds by instructing prompt isolation between planner and validator (Iron Law 2 below). Kambhampati et al. (2024) argue, for planning, that auto-regressive LLMs cannot by themselves plan or self-verify, but can serve as approximate knowledge sources inside an "LLM-Modulo" loop with external verifiers. ASP's executable acceptance tests are verifiers of that external kind.

Its validator is not: it is a second prompt to the same model, which separates context but not the model.

2.4. Closest related work: PreFlect

The closest concurrent work is **PreFlect** (Wang et al., 2026), which similarly shifts the self-reflection paradigm from post-hoc correction to pre-execution criticism of agent plans, distilling planning errors from historical agent trajectories and adding a dynamic re-planning mechanism. The high-level similarity is significant: both ASP and PreFlect identify the post-execution-only nature of conventional reflection as a limitation and shift criticism *before* the action. ASP differs from PreFlect along four axes:

1. **Prompt-isolated validator.** PreFlect distils errors into the agent's own pre-execution criticism; ASP instructs a separate-prompt validator subagent that is not given the planner's transcript.
2. **Process-level isolation for the executor.** In its opt-in --goal mode ASP spawns the executor as a claude -p subprocess with a file-based contract; the PreFlect paper describes no equivalent process boundary.
3. **Explicit runtime constraints.** ASP states twelve Iron Laws, two of them guarded by executable checks and ten as prompt-level rules; PreFlect is presented as a methodology rather than a constraint system.
4. **An empirical CLI-level finding** characterising claude -p exit semantics under graceful refusal, motivating two of the Iron Laws; this is reported separately in Flores (2026b). The phenomenon itself is documented by the vendor and in public issue reports

(catalogued in Flores, 2026b, §2); the contribution there is a measured count and a proposed typed termination contract, not a discovery.

2.5. Classical software and quality-engineering lineage

The four threads above place ASP within the recent LLM-agents literature. ASP's machinery is, however, a transposition of software- and quality-engineering disciplines that predate LLMs by decades. What is new in ASP is not the disciplines but their transposition to a non-deterministic agent runtime and the mechanisms that enforce them there.

Mandatory postcondition checks: ASP and Design by Contract. ASP's micro-TODO schema (§3.4) is Design by Contract (Meyer, 1992) relocated from the routine boundary to the agent-task boundary. Meyer's contract assigns obligations across a caller/routine boundary: the caller must establish the PRECONDITION, which the routine is then entitled to *assume* — Meyer's non-redundancy principle forbids the body from re-checking it — and the routine must guarantee the POSTCONDITION on which the caller relies. Of the schema's seven fields, PRECONDITION, ACTION and POSTCONDITION map onto this contract directly, with ACTION as the routine body the pre/post pair wraps. Meyer (1992) also provides for monitoring assertions at run time, postconditions included, as an option chosen at compilation; by default only preconditions are monitored. ASP turns that option into the rule: because the obligor is a non-deterministic model rather than a compiled routine, every step's postcondition is checked by an external ACCEPTANCE-TEST, a command run outside the model. The trade-off: ASP makes postcondition monitoring mandatory, paying for a check on every step, in time and tokens, to get explicit evidence that the step was completed, because nothing guarantees that a model kept its contract. The rule is inspired by Design by Contract but distinct from Meyer's rejection of redundant precondition checks, which concerns a routine re-checking what its caller already

guaranteed. The FALSIFICATION-TEST adds the Popperian stance (Popper, 1959/2002) that a claim earns confidence by surviving a disconfirmation attempt rather than by a confirming check alone; it works as a negative test beside the contract, not as a clause of it. The same stance produces the exit-code discipline of Iron Laws 11–12. The Iron Laws, in turn, are stated as the analogue of DbC's third contract element — the class invariant — rather than as a per-task pre/post condition; ten of them are instructions the agent is told to keep, not checks that run.

Failure-mode enumeration. Shadow-point detection (Phase 2) is a prospective failure-mode enumeration in the tradition of failure modes and effects analysis (FMEA; MIL-STD-1629A, 1980; IEC 60812:2018). It is deliberately *not* full FMEA: ASP performs no per-component decomposition and no risk-priority scoring (severity × occurrence × detection), so the debt is to FMEA's prospective-enumeration discipline, not its bottom-up mechanics. Three inputs to Phase 2 occupy roles worth separating: the pre-mortem (Klein, 2007; §2.1) supplies the *philosophy* — imagine the failure as already certain; the MAST fourteen-mode taxonomy (Cemri et al., 2025; §2.2) supplies the *library* of failure modes; and the forced enumeration against that fixed library is closer, structurally, to the guideword prompting of hazard and operability studies (HAZOP; IEC 61882:2016), where a fixed list of words drives the search for deviations, than to component-by-component FMEA. What ASP adds is the substrate — the "component" whose failure modes are enumerated is an LLM plan; the planner runs the enumeration (Phase 2), and a validator that does not share the planner's context reviews the resulting list (Phase 5; §2.3).

Mistake-proofing. A subset of the Iron Laws (§3.3) are poka-yoke (Shingo, 1986), devices that make a mistake physically impossible, at the protocol level: the runner template in claude-p-goal-runner.md sets ASP_IN_GOAL for the child and passes --output-format json by default (Iron Laws 9 and 12); Iron Law 10's file-based contract is an instruction. These checks live

in a Bash template the agent is told to run, not in a hook (a command that the runner itself executes) outside the agent's control, and they are not the physical impossibility of true poka-yoke — an environment flag can be unset or fail to propagate — and the remaining laws are procedural gates (Iron Laws 1, 2, 4, 5) or verification rules that still depend on the operator, of which Iron Law 11 ("never trust $?", the shell's record of the last exit code) is the clearest case: precisely the operator-vigilance dependence poka-yoke is defined against. The borrowing is the mistake-proofing *stance* applied to two shell-level checks in a runner script the agent is instructed to emit, not a claim that ASP eliminates operator vigilance.

Independent verification and validation. ASP's prompt-isolated validator (Iron Law 2) transposes the defining move of Independent Verification and Validation (IV&V; IEEE Std 1012-2016) — a verifier kept independent of the developer, in IEEE 1012's terms technically, managerially and financially — from organisational separation to *prompt* separation: the validator subagent shares no context with the planner it checks. The LLM-specific justification for that isolation is the self-correction literature of §2.3; the *form* of the counter-move is IV&V's.

Dependability vocabulary. The exit-code behaviour behind Iron Laws 11–12 has an exact name in the dependability taxonomy: an exit 0 on a graceful decline is an *unsignalled failure* (Avizienis et al., 2004), neither a crash nor a fail-silent stop (one that halts and emits nothing). The term, its prior art, and the measurement are developed in the companion paper (Flores, 2026b); this section imports only the vocabulary and does not re-open the novelty question here.

Empirical warrant. The preference for acceptance and falsification tests over trusted status codes has an empirical basis in the adjacent pathology. Yuan et al. (2014) found that a large majority of catastrophic failures in distributed data-intensive systems trace to faulty handling of non-fatal errors that *were* signalled. ASP's exit-code case is the complement — the signal was *absent* — so

the two do not share a thesis; the narrow lesson they do share is that a status channel is not a proxy for outcome, which is what an external ACCEPTANCE-TEST exists to enforce.

Table 1 maps each ASP mechanism to its classical ancestor and states what ASP adds.

Table 1
ASP mechanisms and their classical ancestors

ASP mechanism	**Classical ancestor**	**What ASP adds (and does *not* claim)**
micro-TODO PRE/ACTION/POST + ACCEPTANCE-TEST	Design by Contract (Meyer, 1992)	a non-deterministic obligor whose postcondition is checked by an external command on every step, where Meyer's run-time monitoring is an option; not the contract idea
micro-TODO FALSIFICATION-TEST	Popperian falsification (Popper, 1959/2002)	a disconfirmation step at the agent-task boundary; a negative test beside the contract, not a clause of it
Shadow-point detection (Phase 2)	FMEA-style enumeration (MIL-STD-1629A; IEC 60812:2018) + HAZOP guidewords, over the MAST library (Cemri et al., 2025)	enumeration over an LLM plan by the planner, reviewed by a prompt-isolated validator; no risk-priority scoring, no per-component decomposition
Iron Laws 9 and 12 as shell-level checks	Poka-yoke (Shingo, 1986)	two checks in a runner script the agent is instructed to emit; not physical impossibility, and Iron Law 10 is an instruction
Prompt-isolated validator (Iron Law 2)	IV&V (IEEE Std 1012-2016)	organisational independence recast as prompt isolation; not IV&V
Never-trust-$? (Iron Laws 11–12)	Unsignalled failure (Avizienis et al., 2004); Yuan et al. (2014)	characterised in the companion paper (Flores, 2026b); not the failure class

Read together, the section makes a modest claim: ASP's machinery is an *integration* of established reliability and quality-engineering disciplines — contract, failure-mode enumeration, IV&V, mistake-proofing — into one protocol whose contribution is operational rather than conceptual. What that integration adds beyond its ancestors is their application in a non-deterministic agent runtime: the Iron Laws (two of them guarded by executable checks), the prompt- and process-level isolation of §3.2, and the reproducible deterministic adjudicator of §5, with the claude -p exit-code

finding characterised separately in the companion paper. This is the experience-report claim of §1, not a novelty claim against the classical literature.

3. The ASP Protocol

3.1. Overview

ASP is a 13-phase protocol that progresses from intake through delivery sign-off. Each phase emits a named artifact (Table 2). The principle organising the phases is that no implementation begins before shadow points have been surfaced (Phase 2), reviewed by a prompt-isolated validator (Phase 5; the --quick mode skips this review, the default and --paranoid modes keep it), and approved by the user (Phase 7); execution stays in the user's session by default and, with the opt-in --goal flag, is delegated to a child subprocess with a file-based contract (Phase 9).

Table 2
ASP phases and their emitted artifacts. Phase 0 has two sub-phases (0a before research, 0b after it), so thirteen phases occupy fourteen rows

#	Phase	Artifact	Design rationale
0a	Pre-research Q&A	intake block	Anthropic Skills best-practice: verifiable intermediate outputs
1	Parallel research	Research summary + recall block	Subagent dispatch pattern
0b	Post-research Q&A	Refined intake (skippable)	Same as 0a
2	Shadow-point detection	Categorised list + mitigations	Klein (2007) + Cemri et al. (2025)
3	Project Charter	Filled charter	Durable artifact-based handoff
4	Macro plan + register	Macro plan + deliverables D01–D*n*	Plan-and-Act (Erdogan et al., 2025)
5	Independent validator	APPROVE / REVISE	Huang et al. (2024) + Tyen et al. (2024) + Zheng et al. (2023)
6	Internal loop (cap=3)	Final-revision plan	Bounded iteration; FM-1.5
7	User approval	Go / no-go	Human-in-the-loop gate

#	Phase	Artifact	Design rationale
8	Micro-TODO + Goal Spec	One TaskCreate task (Claude Code's task-list tool) per step + Goal Spec (the written completion condition for the executor)	Executor handoff
9	Execution, in session or, with ASP's opt-in --goal flag, in a child subprocess that runs Claude Code's /goal command	Tasks marked completed with evidence	Process-level isolation in the opt-in mode; see Flores (2026b)
10	Per-deliverable sign-off	Each deliverable accepted or rejected	Acceptance contract
11	Execution Report + memory	Closure artifact + memory write-back	Reflexion-style memory (Shinn et al., 2023)
12	(Opt-in) public release	Plugin marketplace + GitHub release	Distribution layer

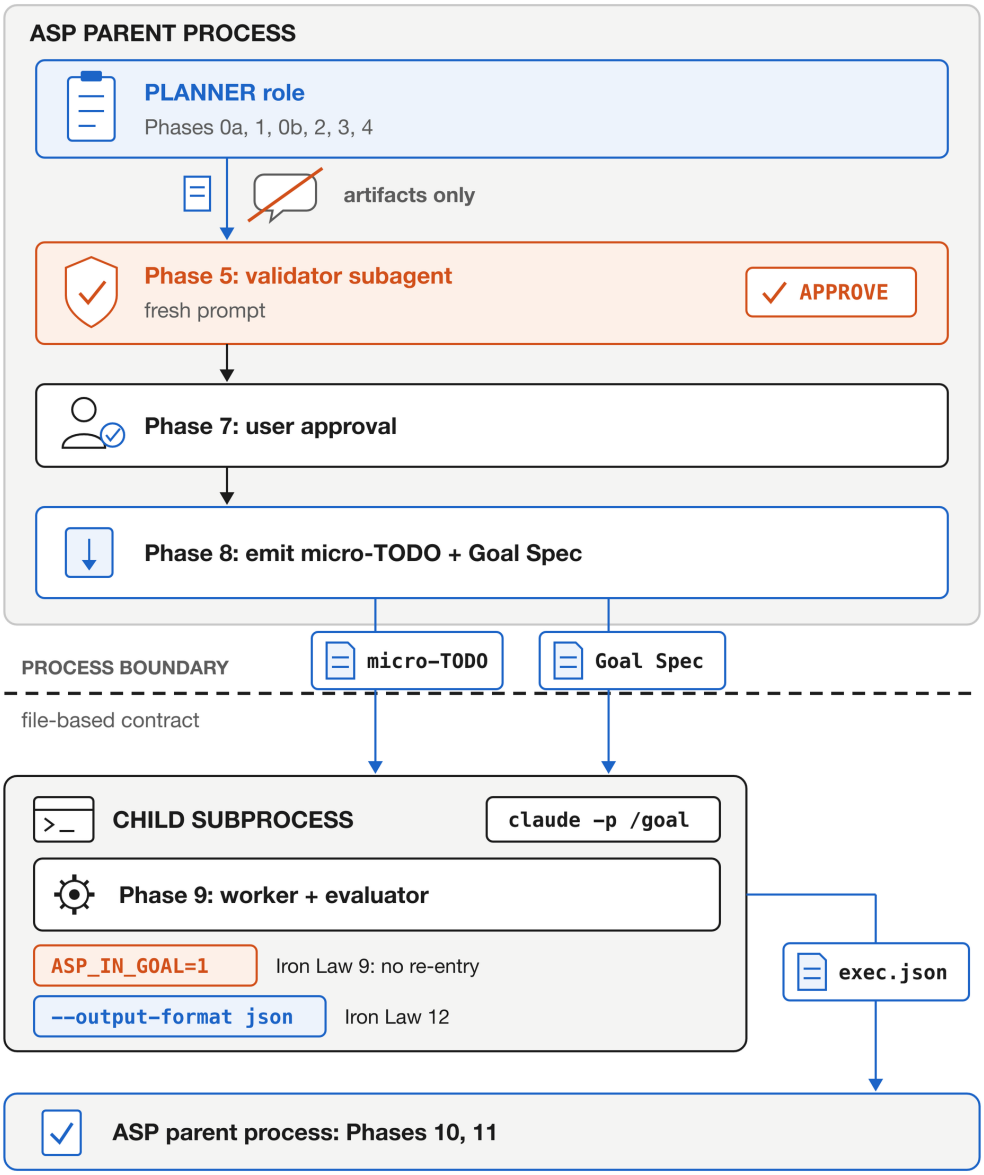
3.2. Architectural positioning

ASP separates three agent roles with explicit artifact-only handoffs (Figure 1).

3.3. Iron Laws

We state twelve operational constraints and call them Iron Laws (Table 3). Iron Laws 9 and 12 have shell-level checks in the runner template and the entry snippet the agent is told to run; the others are rules the skill instructs the agent to follow. Compliance with either kind has not been measured. Each is mapped to its derivation: a primary literature source, a documented Anthropic recommendation, or an empirical design decision.

Figure 1
ASP's three-role separation



Note. Planner phases run in the parent. The validator runs in the parent but in a fresh subagent prompt that receives only artifacts, never the planner's transcript. With the opt-in --goal flag, the case drawn here, the worker runs in a child claude -p subprocess with a file-based contract; by default it runs in the user's session.

Table 3
The 12 Iron Laws and their derivation

#	Law	Derivation
1	No implementation without shadow-point pass first	Klein (2007) + Cemri et al. (2025)
2	No plan approval without independent validator pass (separate prompt); the --quick mode skips the validator	Huang et al. (2024) + Tyen et al. (2024) + Zheng et al. (2023); mitigates FM-3.3
3	No micro-TODO without macro plan + charter + deliverables approved by user	Anthropic Skills: verifiable intermediate outputs
4	No step completion without fresh acceptance-test evidence	Cemri et al. (2025) FM-3.1 + FM-3.2 + FM-2.6
5	No deliverable accepted without evidence file/output matched	Cemri et al. (2025) FM-3.1
6	No project close without execution report + memory write-back	Shinn et al. (2023) memory buffer
7	No loop without cap (default cap=3 validator rounds)	Cemri et al. (2025) FM-1.5
8	No Phase 9 execution without evaluator dispatch at checkpoints (except under --quick)	Erdogan et al. (2025) planner/executor
9	No reentrant ASP inside claude -p /goal child (ASP_IN_GOAL env-var guard)	Cost-explosion mitigation; design decision
10	No claude -p /goal spawn without file-based contract	Subprocess state-divergence mitigation; design decision
11	Never trust $? from claude -p for semantic success/ failure	Empirical observation, characterised in Flores (2026b)
12	All claude -p invocations must use --output-format json or stream-json	Corollary of Iron Law 11

3.4. The micro-TODO contract

At the boundary between planning and execution (Phase 8), ASP emits a contractual task list. Each task is created via the TaskCreate primitive with seven labeled fields:

- **PRECONDITION** — invariant that must hold before the step.
- **ACTION** — exact command or literal edit.
- **POSTCONDITION** — state change effected by the step.
- **ACCEPTANCE-TEST** — command and expected output that prove the postcondition.

- **FALSIFICATION-TEST** — command or observation that would prove the step failed.
- **DEPENDS-ON** — upstream task identifiers (a directed acyclic graph, or DAG: dependencies without cycles).
- **DELIVERABLE-ID** — which deliverable in the register this step contributes to.

This schema is what makes the task list auditable: no task is marked completed without acceptance-test evidence (Iron Law 4); no deliverable is signed off without matching evidence (Iron Law 5). The schema is shared with the Goal Spec that the Phase 9 subprocess consumes in the opt-in --goal mode.

4. Operational Deployment

ASP ships as a Claude Code plugin and is distributed through three install paths, each appropriate for a different audience:

1. **Plugin marketplace** (recommended for end users):

   ```
   claude plugin marketplace add ulissesflores/anticipating-shadow-points
   claude plugin install anticipating-shadow-points@anticipating-shadow-points
   ```

   Invocation: /anticipating-shadow-points:asp <task> or bare /asp.

2. **Developer mode** (for contributors): claude --plugin-dir ./plugins/anticipating-shadow-points loads the skill in place; useful when iterating on the skill itself.

3. **Standalone install script** (for users who do not use the plugin manager): ./scripts/install.sh copies the skill into the user's Claude Code skills directory.

The plugin uses Anthropic's official .claude-plugin/plugin.json manifest format (Claude Code 2.1+; Anthropic, 2026a). It lives in its own folder, plugins/anticipating-shadow-points/, which holds only Markdown, the icon and the licence, so an install copies nothing else; the repository root doubles as a self-host marketplace via .claude-plugin/marketplace.json.

Permissions. Since plugin version 1.0.4 the Phase 9 runner of the opt-in --goal mode spawns the claude -p child with **no permission flag**: the child runs under the user's own Claude Code permission settings, and the skill never grants, widens or bypasses them. Earlier versions passed --permission-mode bypassPermissions; that was removed after the Claude plugin directory, reviewing the plugin, flagged that a plugin must not grant itself permissions (CHANGELOG.md, version 1.0.4). In non-interactive mode an action the user has not pre-allowed is denied rather than prompted; the parent reads the permission_denials field of the child's JSON result and reports any denial to the user instead of retrying with wider permission. This makes autonomous execution strictly weaker than in the May 2026 deployment, and safer: the skill can only do what the operator already allowed. Since version 2.0.0 the skill also tells the user, before the first spawn, that the child runs under their permissions and their usage, with the budget and turn cap stated, and spawns only after their OK; the child's scratch files go to the plugin's own data folder, which Claude Code removes when the plugin is uninstalled. The skill itself is implemented as a Markdown-based skill following the Anthropic Agent Skills best-practices document (Anthropic, 2026b), with companion files for the MAST checklist, per-mode pre-mortem prompts, and the seven artifact templates referenced in Section 3.1.

Documentation is published as a GitHub Pages site, with machine- or AI-assisted translations of the README into Portuguese, Spanish, Italian, and Hebrew (docs/translations.md). The full

source — skill files, plugin manifests, documentation, test artefacts, benchmark code — is MIT-licensed; the paper text, figures and data are CC BY 4.0.

5. Preliminary Observations

We ran ASP through five structured engineering evaluations spanning heterogeneous domains. This section separates three things: what the author observed in real model dispatches (5.2), what the deterministic adjudicator shows about itself on constructed fixtures (5.3), and what has not been measured at all.

5.1. The evaluation set

Table 4 lists the five engineering evaluations.

Table 4
The five engineering evaluations

ID	Domain	Task (abbreviated)	Expected size	Acceptance
E1	Schema migration	NOT NULL column added to 1M-row user_profiles	10	≥ 80%
E2	Edge deploy	Edge function calling rate-limited external API	10	≥ 80%
E3	Refactor	Date-formatting utility used across 30+ files	10	≥ 80%
E4	RLS policy	Row-level security policy referencing new column	10	≥ 80%
E5	Scheduling	Cron job whose script conflicts with disabled service	10	≥ 80% and points 1 and 8 present

Each evaluation specifies an INPUT (a task statement as a senior engineer might phrase it to a colleague), an EXPECTED list of shadow points that a competent practitioner would surface, and an ACCEPTANCE threshold (default 80%). E5 is not portable: two of its ten expected points (1 and 8) depend on the author's private project memory and cannot surface from its INPUT alone,

so it measures memory recall as much as planning. The full text of each evaluation is in the companion repository.

5.2. What was observed in real dispatches (author, non-blind)

On 2026-05-17 the evaluations were dispatched to fresh subagents: GREEN (with the ASP discipline injected: pre-mortem framing, MAST 14-mode forced enumeration, mitigation requirement) on all five, and RED (asked to plan as a senior engineer would, without any framework) on three, E1–E3; E4 and E5 were added afterwards and run in GREEN only. All ran on the session model of the time (Claude Opus 4.7; the model was not recorded per dispatch), one dispatch per condition. The subagents inherited the author's global agent instructions, so the RED condition was a competent baseline, not a naive one. The author read each plan against the expected list and recorded a coverage judgement; the summaries, with the subagent identifiers, are committed in tests/baseline-pressure-tests.md and tests/evals-summary.md. The raw transcripts were not committed and are no longer available. The RED prompts recorded there are shorter Portuguese phrasings of E1–E3 that omit details the expected lists later score: E2's RED prompt does not state the 100 req/s rate limit of Resend, the e-mail API the task calls, and E1's does not say that app code already references the new column in five query sites. Part of the RED shortfall may therefore be an artefact of the prompt, not of the absence of the protocol.

Under that non-blind reading, the three RED plans covered roughly 30–60% of their expected lists (mean near 47%, counting partial mentions as covered) and the GREEN plans covered all or nearly all of theirs. The committed summaries, written in May 2026, conclude that ASP is "reproducibly better"; a note above them withdraws that conclusion, which is not supported. These figures are an author's judgement on unpreserved text, scored against lists the same author wrote, with the

MAST vocabulary that the GREEN prompt injects appearing in those lists. They are reported as observation, not measurement, and no number in this paragraph is asserted by the replication layer.

5.3. What the deterministic adjudicator shows about itself

The benchmark/ package scores a plan against an expected list with a deterministic 4-layer matcher, the part that decides whether a plan mentions an expected point: exact name, synonym sidecar (an optional list of accepted synonyms), Jaccard sliding window (a word-overlap score over a moving stretch of the plan) and unmatched. On the published run every one of the 42 matches comes from the first layer (exact name); no synonym sidecar exists for any task, and the Jaccard layer never fires. Its one published run is a **mock-mode** run: the RED and GREEN texts are fixtures the author wrote to exercise the pipeline end to end (benchmark/tests/fixtures/mock_runs/); on them layers 2 and 3 never produce a match (layer 3's overlaps are computed and stay below the threshold), and only the unit tests exercise a match through those layers. The fixtures themselves were written deliberately weaker for RED and deliberately aligned with the expected list for GREEN. On those fixtures the matcher scores RED 4% and GREEN 80% on average — a statement about how the fixtures were written, not about ASP.

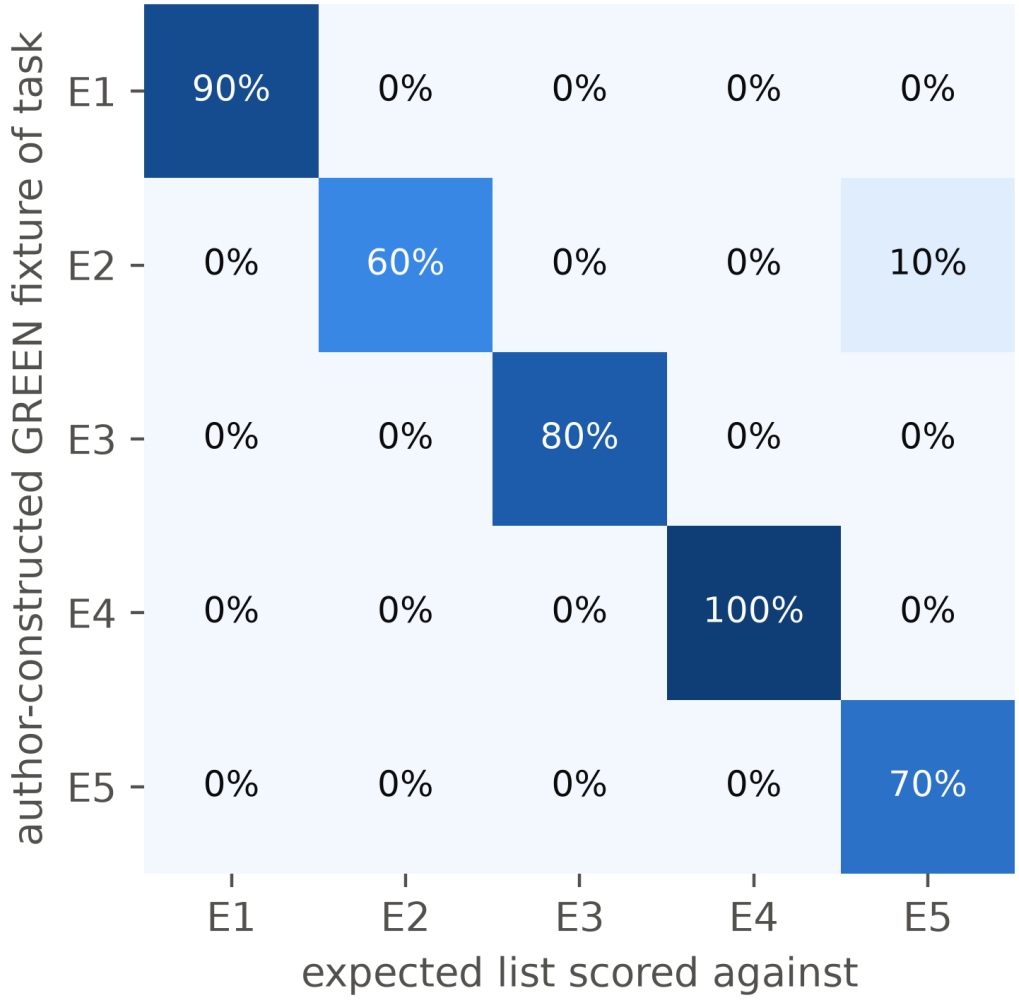
In symbols, layer 3 scores a window of the plan against an expected point by Equation 1, where A is the set of distinct tokens of the window and B that of the expected point; the vertical bars count the tokens of a set. A match needs a score of at least .30.

$$J = \frac{|A \cap B|}{|A \cup B|} \tag{1}$$

The useful property the mock run does establish is **discrimination**: scoring each GREEN fixture against a *different* task's expected list drops the mean from 80% to 0.5% (largest off-task cell 10%;

Figure 2). That is weaker evidence than it looks: the one off-task hit is the generic one-word name "Alerting", and the other cells are zero because layer 1 requires a whole expected name to appear verbatim. The matcher is also stricter than the evaluation files, which tell a human evaluator to accept paraphrases. Recall is a separate matter, and the fixtures show a false negative: the E2 RED fixture mentions a rate limit, retries and logging and still scores 0/10: its best window overlap with the expected point "Rate-limit backoff" (its name and description) is a Jaccard of .11, below the .30 threshold, and the matcher does no stemming ("logging" does not match "logs"). Paraphrase with low lexical overlap is missed. Re-scoring also exposed a defect, now fixed: the sliding window skipped the last few tokens of a text whose length did not land on the stride; re-scoring every fixture with the fix changes no coverage value, though the best overlap of three unmatched points rises slightly (E2 RED "Observability / logs" from .00 to .02), still far below the threshold. Layer 3 also has an arithmetic ceiling: Jaccard divides by the union, and a 40-token window of the fixtures holds 27 to 40 distinct content words (median 36); in any window with 34 or more, an expected point with ten or fewer distinct tokens cannot reach .30 even if the plan repeats all of them, and 19 of the 50 expected points are that short. The 2026-11 study (Section 5.5) replaced layer 3 with token recall before any reference list existed. On the reviewers' own lists even that version could not see that two reviewers named the same problem, so a blind judge became the primary scorer before any data was collected; the matcher is reported as secondary.

Figure 2
Instrument check: each author-constructed GREEN fixture scored against every task's expected list. Not an effect of ASP



Section 5.5 reports the measurement; Section 6 states the design hypotheses it bears on.

5.4. The claude -p exit-code finding

During a four-test battery against the claude -p non-interactive runner, we observed that the binary returns exit code 0 even when the agent gracefully refuses the requested goal — the harness (the program that runs the agent) reports a successful conversation end despite the agent semantically refusing the task. The exit code reflects process-level success of the CLI binary, not semantic

success of the agent's task. Iron Laws 11 and 12 require --output-format json or stream-json and reading is_error, terminal_reason and stop_reason. Those fields separate a turn-cap cut-off and a classifier refusal from a completed turn, but not a model-generated decline from a delivery; the runner therefore also flags replies whose text matches a decline pattern, a heuristic the companion paper tests.

This finding is observationally significant for any toolchain that wraps claude -p in a shell pipeline. Because the observation was from a single refusal scenario in our own battery, we **characterise it independently** in the companion paper Flores (2026b), which reports an N=50 measurement across four categories of refusal-provoking prompt (impossible-in-one-turn, capability, safety, ambiguity) on Claude Code 2.1.200 in July 2026. The session model was Fable 5; Opus 4.8 answered 16 of the 50 prompts, including all 12 safety prompts — consistent with an automatic fallback hand-off, although the envelopes (the JSON record claude -p prints at the end of each run) record which model answered, not why. Twenty-four trials exited 0, all with stop_reason: end_turn. The companion paper counts the declines among them four ways: 20 (40%, 95% Wilson CI 28–54%, a confidence interval suited to small samples) under the author's non-blind label; 19 (38%) under the majority of a codebook, written after the author had read the replies, and applied by five LLM raters, four of which returned identical labels, all of the lineage of the model under test, so the majority is a codebook check, not an inter-rater agreement estimate; 12 (24%) counting only the safety and destructive-command refusals and excluding the eight reasoned declines of impossible-in-one-turn tasks; and 9 (18%) under the locked English-only phrase detector, which misses the replies written in Portuguese. An earlier snapshot (May 2026, Claude Code 2.1.143, Opus 4.7) read 60% under the same detector, on a different model and reply language. A human second rater and a third snapshot under a drafted pre-registration are the open follow-ups.

5.5. A pre-registered measurement (2026-11)

The study was pre-registered in the repository (benchmark/study-2026-11/PROTOCOL.md), and its hash ledger was timestamped with OpenTimestamps, a public timestamping service, before the first trial. It replaced Protocol 2 of PREREGISTRATION.md, the repository's earlier plan for this measurement, which was never deposited. The design (Figure 3):

- **Tasks.** Six change requests on six public repositories, each pinned to one commit: soft-deleting users (T1, FastAPI template), a weekly e-mail digest (T2, Flask), linking Google and password sign-in (T3, Django), retrying POST by default (T4, the ky HTTP library), zero-downtime deploys (T5, Django on Docker), and co-authors per article (T6, Express with Prisma).
- **Conditions.** Three, each run five times per task (90 trials) with claude -p on Claude Opus 5.5, in plan mode (a Claude Code setting in which the agent may read files but not change them), from a fresh clone and a clean environment, with the same output contract: a final table of at most ten risks.
    - **plain:** "Plan this change."
    - **risk:** "Before planning, list the ways this change could go wrong."
    - **ASP:** /asp --standard, stopped after the plan had been through its single validator round.
- **Reference lists.** Three models of other vendors (Grok, GPT-OSS 120B, Gemini 3.8 Flash) read each repository and request without seeing ASP and wrote the risks they expected. A blind judge grouped their items, and a risk entered the reference list when at least two

reviewers named it (8 to 12 per task). Models stood in for independent engineers, which the protocol states as a weakening.

- **Scoring.** The primary judge (Gemini 3.8 Flash) saw only the rows of each answer's table, with words that would reveal the condition redacted, and marked each reference risk as covered or not. A second judge (Grok) did the same. The two agreed on 91.9% of 855 item decisions (computed from benchmark/study-2026-11/judge/ and asserted in tests/test_study_2026_11.py).

Figure 3
Design of the 2026-11 study: the reference risks, written without seeing ASP, and the answers of the three conditions meet only at the blind judge

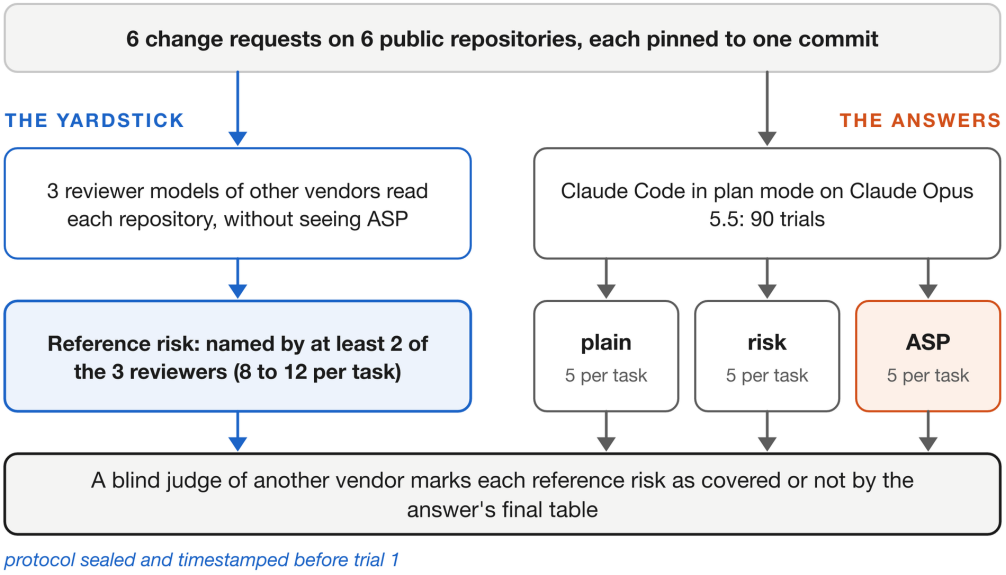
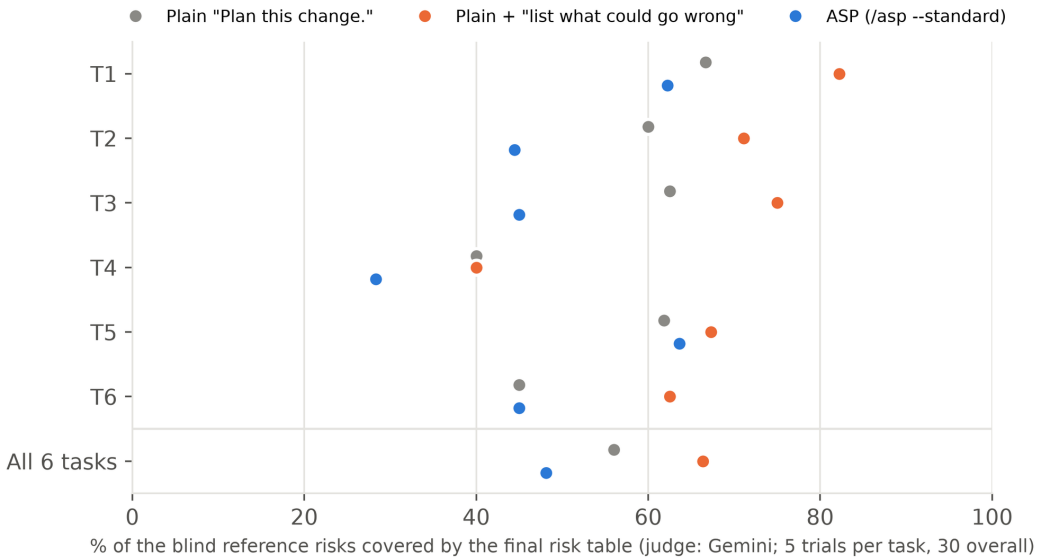


Table 5 and Figure 4 report coverage by condition.

Table 5
Risk coverage and time per trial in the 2026-11 study, by condition

Condition	Covered (primary judge)	Covered (second judge)	Mean time per trial
plain	56%	57%	58 s
risk	66%	70%	50 s
ASP	48%	54%	175 s

Figure 4
Share of the blind reference risks covered by each condition's final table, per task and overall (primary judge)



- **Primary.** ASP minus risk was −.18 (primary judge) and −.16 (second judge). It was negative on all six tasks, with a one-sided, task-stratified permutation p = 1.0 (how often shuffling condition labels within tasks gives as large a gap; the test asks whether ASP exceeds risk). The decision rule fixed in advance returns *no measurable benefit*.
- **Secondary, pre-registered.** risk minus plain was +.10 (p = .001) and +.13 (p < .001): asking for the ways the change could go wrong, in one sentence, raised coverage.
- **Exploratory.** ASP's answers handle some risks in their assumptions and deliverables rather than in the final table. When the primary judge read the whole answer, the gap narrowed to −.08 (66% against 74%), and ASP was still behind on five of the six tasks (verdicts in benchmark/study-2026-11/judge/gemini-full/, asserted in tests/test_study_2026_11.py).
- **Limits.** One session model (in one of the 90 trials the harness also called a smaller model of the same vendor), six tasks, reference lists and judges that are all models, and coverage

as the only outcome: whether listed risks are real, or lead to better code, was not measured.

- **Record.** Every trial, verdict and analysis is in the repository, chained by hash, and run_all.py re-derives every number above.

6. Design Hypotheses

We highlight five design choices we believed to be load-bearing. The first four are hypotheses about ASP's design, each stated with the observation that motivated it. The 2026-11 study (Section 5.5) measured a non-interactive adaptation of Phases 0 to 6 as a whole and found no benefit, which weighs against them taken together without testing any one alone. All three conditions were told to state their assumptions instead of asking, which replaces ASP's question rounds (Phases 0a and 0b); interactive use was not evaluated. The fifth rests on the companion paper's measurement.

Hypothesis: a forced taxonomy surfaces more than free-form pre-mortem. In the author's drafts, free-form pre-mortem clustered on obvious risk categories (data, integration) and underweighted less-salient ones (time/timezone, observability, operational excellence, contract drift). No dispatch tested free-form pre-mortem as a condition (RED had no framework at all), so this rests on the author's reading of drafts. The free-form prompt ("imagine the task has failed in 30 days; list 10+ causes") would then be necessary but not sufficient, and the MAST 14-mode interrogation would surface categories the free-form pass clusters away from.

Hypothesis: prompt isolation of the validator matters. We expect validators that share context with planners to reproduce the planner's confidence on shared content. The convergent literature on intrinsic self-correction limits, error-localization failure, and judge self-enhancement bias all point at this; we have not tested it inside ASP.

Hypothesis: file-based handoffs survive context pressure better than transcript-based ones. ASP commits charter, macro plan, deliverables register and goal spec to disk before execution starts. In the opt-in --goal mode the child receives only the Goal Spec text on stdin (standard input) and returns a JSON result; the parent reconstructs state from that result and from the files on disk, so neither side depends on a transcript that may have been compressed or truncated. We have not compared the two handoffs.

Hypothesis: stated constraints (Iron Laws) make a skill harder to skip. In our experience, skills that read as "guidelines" are often skipped under context pressure. Skills that emit a checklist of non-negotiables, with templates whose fields must be filled before the next phase, are in our use harder to skip. Compliance has not been measured.

Measured elsewhere: treating claude -p exit codes as semantic-success signals is a silent footgun. This is the finding reported in Flores (2026b); mentioning it here because the design implication (Iron Laws 11 and 12) was material to the system's reliability.

7. Threats to Validity

We treat threats explicitly across the four categories standard in empirical software-engineering reports.

Construct validity. In the evaluations of Sections 5.1–5.2, the same researcher who designed the protocol authored the expected-shadow-points lists *and* hand-adjudicated coverage. Even with a written rubric, this is the single largest threat to the observation's interpretability. The deterministic adjudicator we ship would mitigate this once applied to real, preserved transcripts; so far it has only been run on constructed fixtures (§5.3), and it does not address the prior question of whether the expected lists themselves are biased toward what ASP surfaces. **Blind re-authoring by**

independent reviewers is the strongest follow-up. The study of Section 5.5 had its reference lists written blind, by three models of other vendors rather than engineers, for six new tasks rather than these five.

Internal validity. With one dispatch per cell, GREEN on five tasks and RED on three, the evaluations of Sections 5.1–5.2 at most suggest that the protocol can change the transcripts on these cases; they are *not* sufficient to estimate an effect size, establish causality, or support statistical inference, and no statistical test is reported for them. No formal ablation of components has been run (the qualitative attribution of load-bearing-ness to the MAST checklist in §6 is a hypothesis). The acceptance threshold of 80% is the author's choice and not derived from prior literature.

External validity. For the evaluations of Sections 5.1–5.2: a single underlying model (Claude Opus 4.7, presumed from the session; not recorded per dispatch), single environment (the author's macOS workstation and the Claude Code version of May 2026, not recorded per dispatch), five engineering tasks drawn from the author's domain experience. We make no claim of generalisation beyond these conditions.

Reliability. All evaluation INPUT statements, expected-shadow-point lists, the mock-run fixtures and the deterministic-adjudicator code are committed to the companion repository; the 2026-05-17 dispatch transcripts were not preserved, so the observations of §5.2 cannot be re-checked. The benchmark package builds a cryptographically-chained manifest per run, so any byte edited in the files the manifest covers (environment, code, inputs, outputs and adjudication) invalidates the manifest hash; the manifest file itself and the run's report.md are outside it.

The study of Section 5.5. External: one model (Claude Opus 5.5) and six tasks. Construct: the reference lists and the judges are models, in place of independent engineers, which the protocol

states as a weakening; coverage is the only outcome, so whether the listed risks are real, or lead to better code, was not measured; and the final-table contract may miss risks that ASP handles in its assumptions and deliverables (read whole, the gap narrowed but did not close). Internal: the study measured the planning phases of the protocol as a whole, not any one component (Section 6). The reviewers who wrote the reference lists were asked for "the concrete ways this change could go wrong", words close to those of the risk condition, which may favour that condition on coverage. The primary judge also wrote one of the three reviewer lists, and 50 of the 57 reference risks carry its wording; the second judge wrote another list, so neither judge is independent of the yardstick it applies.

8. Limitations and Future Work

The limitations of this experience report match the threats above one-to-one. The first follow-up named in earlier versions, a confirmatory study of the one-sentence pre-mortem (the request of the risk condition, which keeps the listing part of a pre-mortem and does not ask the model to imagine that the change has already failed), is now reported in Flores (2026c). On twelve pinned tasks and two models of one vendor, Claude Opus 5.5 and Claude Sonnet 5.5, the sentence raised coverage over a plain planning request from 46.1% to 57.5% across 240 trials ($p < .001$); that claim is limited to model-written reference lists and to plans, not code. The highest-priority follow-ups that remain, in order, are:

1. **Formal ablation**, now that the confirmatory study is done: before testing which component of the protocol helps (MAST checklist, validator isolation, Phase 0a intake), test whether any of them beats the one sentence.

2. **A human engineer on a sample of the reference lists**, beside the model-written lists of Section 5.5 and of Flores (2026c), and, for the study of Section 5.5, a precision measure (are the listed risks real?) next to coverage.
3. **A human second rater and a third claude -p snapshot** for the companion finding (Flores, 2026b, §8), the latter under Protocol 1 of PREREGISTRATION.md.
4. **Cross-model evaluation.** Re-run on Claude Sonnet, GPT-5, Gemini 2.5, and an open-weight model.
5. **Head-to-head against PreFlect.** Run ASP and a PreFlect reimplementation on the same task set.
6. **Embedding-based adjudicator** as a v0.2 of the benchmark, with pinned sentence-transformer model, to raise the recall floor on lexically-disjoint paraphrases.
7. **Standardized-benchmark replication.** The five-task set is not a public agent benchmark; replicating the direction on SWE-bench (Jimenez et al., 2024) or a comparable standardized suite would test whether it survives outside the author's chosen tasks.

9. Conclusion

We described ASP, an operational pre-mortem skill for LLM coding agents, integrating Mitchell, Russo and Pennington's (1989) prospective hindsight finding, Klein's (2007) pre-mortem operationalisation, and Cemri et al.'s (2025) MAST 14-mode failure taxonomy as runtime discipline under a contract of twelve Iron Laws. The skill ships open source with three install paths. The author's early, non-blind observation that assisted plans covered more failure modes did not survive measurement. In a pre-registered study on six real repositories, the planning phases of the

protocol covered fewer of the reviewers' risks than plain Claude Code asked, in one sentence, to list the ways the change could go wrong, and that sentence alone beat a plain planning request. A confirmatory study has since tested that comparison as its primary question (Flores, 2026c).

We additionally observed that claude -p returns exit code 0 even when the agent gracefully refuses the requested goal. The vendor documents the behaviour for its API, as the companion paper catalogues (Flores, 2026b, §2); that paper adds a count at the CLI boundary.

With the confirmatory study of the one-sentence pre-mortem reported (Flores, 2026c), the strongest follow-up for ASP itself is the formal ablation of Section 8. The most useful companion work is a human second rater and a pre-registered third snapshot of the claude -p measurement (Flores, 2026b).

10. Declarations

Conflict of interest. The author created and maintains ASP, the plugin this report describes, and directs the laboratory that publishes it. The pre-registered study of Section 5.5 found no measurable benefit of the protocol.

Funding. This work received no external funding. The costs of using the models were paid by the author.

Author contribution. Single author: conception, design, data collection, analysis, writing and revision.

Use of AI tools. Language models are the subject of this report and take three roles in the study of Section 5.5: the agents under test, the writers of the reference lists, and the judges. An AI assistant was also used in programming, in the analysis, and in drafting and translating the text, under the author's direction, as the Acknowledgments detail. The author checked every number against the data and is responsible for the content.

Acknowledgments

This experience report was prepared in explicit human-in-the-loop collaboration with an AI assistant, used as a drafting, verification, and review partner. Methodology decisions, empirical design, and finalisation were performed by the human author. The collaboration model is documented in the companion repository's docs/ARCHITECTURE.md Parts 5–6.

The author thanks the maintainers of the obra/superpowers skill framework, the contributors of the open agentskills.io standard, and the Anthropic Claude Code team for the /goal primitive whose empirical behavior motivated Iron Laws 11 and 12.

References

Anthropic. (2026a). *Plugins reference* [Software documentation]. https://code.claude.com/docs/en/plugins-reference

Anthropic. (2026b). *Agent Skills best practices* [Software documentation]. https://platform.claude.com/docs/en/agents-and-tools/agent-skills/best-practices

Avizienis, A., Laprie, J.-C., Randell, B., & Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, *1*(1), 11–33. https://doi.org/10.1109/TDSC.2004.2

Barke, S., Goyal, A., Khare, A., Singh, A., Nath, S., & Bansal, C. (2026). *AgentRx: Diagnosing AI agent failures from execution trajectories*. arXiv:2602.02475. https://arxiv.org/abs/2602.02475

Cemri, M., Pan, M. Z., Yang, S., Agrawal, L. A., Chopra, B., Tiwari, R., Keutzer, K., Parameswaran, A., Klein, D., Ramchandran, K., Zaharia, M., Gonzalez, J. E., & Stoica, I. (2025). Why do multi-agent LLM systems fail? *Advances in Neural Information Processing Systems, 38*, 135676–135729. https://doi.org/10.52202/085713-4082

Deshpande, D., Gangal, V., Mehta, H., Krishnan, J., Kannappan, A., & Qian, R. (2025). *TRAIL: Trace reasoning and agentic issue localization*. arXiv:2505.08638. https://arxiv.org/abs/2505.08638

Erdogan, L. E., Lee, N., Kim, S., Moon, S., Furuta, H., Anumanchipalli, G., Keutzer, K., & Gholami, A. (2025). Plan-and-Act: Improving planning of agents for long-horizon tasks. In *Proceedings of the 42nd International Conference on Machine Learning* (Vol. 267, pp. 15419–15462). PMLR. https://proceedings.mlr.press/v267/erdogan25a.html

Flores, C. U. (2026a). *ASP — Anticipating Shadow Points: Claude Code plugin and replication package* (Version 2.0.0) [Computer software]. GitHub. https://github.com/ulissesflores/anticipating-shadow-points https://doi.org/10.5281/zenodo.20276631

Flores, C. U. (2026b). *Graceful refusals as unsignalled failures: Characterising claude -p exit-code semantics on agent declines* (Version 2.4) Technical note. Zenodo. https://doi.org/10.5281/zenodo.24065934

Flores, C. U. (2026c). *One sentence before the plan: A pre-registered confirmatory study of pre-mortem prompting in LLM coding agents* (Version 1.0) [Technical report]. Zenodo. https://doi.org/10.5281/zenodo.24410437

Huang, J., Chen, X., Mishra, S., Zheng, H. S., Yu, A. W., Song, X., & Zhou, D. (2024). *Large language models cannot self-correct reasoning yet*. ICLR 2024. https://arxiv.org/abs/2310.01798

IEEE. (2017). *IEEE standard for system, software, and hardware verification and validation* (IEEE Std 1012-2016). https://doi.org/10.1109/IEEESTD.2017.8055462

International Electrotechnical Commission. (2016). *Hazard and operability studies (HAZOP studies) — Application guide* (IEC 61882:2016, Edition 2.0). https://webstore.iec.ch/en/publication/24321

International Electrotechnical Commission. (2018). *Failure modes and effects analysis (FMEA and FMECA)* (IEC 60812:2018). https://webstore.iec.ch/en/publication/26359

International Electrotechnical Commission. (2019). *Risk management — Risk assessment techniques* (IEC 31010:2019, Edition 2.0). https://webstore.iec.ch/en/publication/59809

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. (2024). *SWE-bench: Can language models resolve real-world GitHub issues?* ICLR 2024. https://arxiv.org/abs/2310.06770

Kambhampati, S., Valmeekam, K., Guan, L., Verma, M., Stechly, K., Bhambri, S., Saldyt, L. P., & Murthy, A. B. (2024). Position: LLMs can't plan, but can help planning in LLM-Modulo frameworks. In *Proceedings of the 41st International Conference on Machine Learning* (Vol. 235, pp. 22895–22907). PMLR. https://proceedings.mlr.press/v235/kambhampati24a.html

Kamoi, R., Zhang, Y., Zhang, N., Han, J., & Zhang, R. (2024). When can LLMs actually correct their own mistakes? A critical survey of self-correction of LLMs. *Transactions of the Association for Computational Linguistics, 12*, 1417–1440. https://doi.org/10.1162/tacl_a_00713

Klein, G. (2007). Performing a project premortem. *Harvard Business Review*, *85*(9), 18–19.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Gupta, S., Majumder, B. P., Hermann, K., Welleck, S., Yazdanbakhsh, A., & Clark, P. (2023). *Self-Refine: Iterative refinement with self-feedback*. NeurIPS 2023. https://arxiv.org/abs/2303.17651

Meyer, B. (1992). Applying "design by contract." *Computer*, *25*(10), 40–51. https://doi.org/10.1109/2.161279

Mitchell, D. J., Russo, J. E., & Pennington, N. (1989). Back to the future: Temporal perspective in the explanation of events. *Journal of Behavioral Decision Making*, *2*(1), 25–38. https://doi.org/10.1002/bdm.3960020103

Popper, K. R. (2002). *The logic of scientific discovery*. Routledge. (Original work published 1959)

Shingo, S. (1986). *Zero quality control: Source inspection and the poka-yoke system* (A. P. Dillon, Trans.). Productivity Press. (Original work published 1985)

Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., & Yao, S. (2023). Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems, 36*, 8634–8652. https://doi.org/10.52202/075280-0377

Tyen, G., Mansoor, H., Cărbune, V., Chen, P., & Mak, T. (2024). LLMs cannot find reasoning errors, but can correct them given the error location. In *Findings of the Association for Computational Linguistics: ACL 2024*. https://arxiv.org/abs/2311.08516

U.S. Department of Defense. (1980). *Procedures for performing a failure mode, effects and criticality analysis* (MIL-STD-1629A). https://everyspec.com/MIL-STD/MIL-STD-1600-1699/MIL_STD_1629A_1556/

Wang, H., Cao, Y., Lin, L., & Chen, J. (2026). *PreFlect: From retrospective to prospective reflection in large language model agents*. arXiv. https://arxiv.org/abs/2602.07187

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, S., Liu, J., Awadallah, A. H., White, R. W., Burger, D., & Wang, C. (2023). *AutoGen: Enabling next-gen LLM applications via multi-agent conversation*. arXiv. https://arxiv.org/abs/2308.08155

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). *ReAct: Synergizing reasoning and acting in language models*. ICLR 2023. https://arxiv.org/abs/2210.03629

Yuan, D., Luo, Y., Zhuang, X., Rodrigues, G. R., Zhao, X., Zhang, Y., Jain, P. U., & Stumm, M. (2014). Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)* (pp. 249–265). USENIX Association. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan

Zhang, G., Wang, J., Chen, J., Zhou, W., Wang, K., & Yan, S. (2025). *AgenTracer: Who is inducing failure in the LLM agentic systems?* arXiv:2509.03312. https://arxiv.org/abs/2509.03312

Zhang, J., Peng, X., Chen, Q., Li, Y., Hayashi, H., & Wu, C.-S. (2026). *Prospective hindsight: Self-calibrating reinforcement learning via prediction-reality gaps*. arXiv:2610.02740. https://arxiv.org/abs/2610.02740

Zhang, S., Yin, M., Zhang, J., Liu, J., Han, Z., Zhang, J., Li, B., Wang, C., Wang, H., Chen, Y., & Wu, Q. (2025). Which agent causes task failures and when? On automated failure attribution of LLM multi-agent systems. In *Proceedings of the 42nd International Conference on Machine Learning* (Vol. 267, pp. 76583–76599). PMLR. https://proceedings.mlr.press/v267/zhang25cq.html

Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E. P., Zhang, H., Gonzalez, J. E., & Stoica, I. (2023). *Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena*. NeurIPS 2023 D&B. https://arxiv.org/abs/2306.05685

Zhu, K., Liu, Z., Li, B., Tian, M., Yang, Y., Zhang, J., Han, P., Xie, Q., Cui, F., Zhang, W., Ma, X., Yu, X., Ramesh, G., Wu, J., Liu, Z., Lu, P., Zou, J., & You, J. (2025). *Where LLM agents fail and how they can learn from failures*. arXiv. https://arxiv.org/abs/2509.25370

Appendix A — Citation verification status

Every claim in the body cites a primary source where one exists. The paragraphs below record when each group of references was checked.

The references present in version 1.0 were verified by direct URL fetch on 2026-05-17; references added in the July 2026 revision (Flores, 2026b; the classical-lineage entries of Section 2.5) were checked when added; and the entries added in November 2026 (Barke et al., 2026; Deshpande et al., 2025; G. Zhang et al., 2025; J. Zhang et al., 2026) were checked against their arXiv records, and Cemri et al. (2025) against its NeurIPS proceedings record. In the same month Kambhampati et al. (2024) and Shinn et al. (2023) were checked against their proceedings records (PMLR and NeurIPS), and their entries now cite those records. Flores (2026c), also added in November 2026, was checked against its manuscript in the companion repository (paper/one-sentence-premortem.md). The MAST 14-mode names are quoted verbatim from Cemri et al. (2025) and listed in the companion skill file plugins/anticipating-shadow-points/skills/anticipating-shadow-points/mast-checklist.md. Mitchell et al. (1989) is the primary study of prospective hindsight; the ≈30% figure is Klein's (2007) summary of it, which we could not re-derive from the 1989 paper, and Klein (2007) is also the operational recipe.

The nine references introduced in §2.5 (Avizienis et al., 2004; International Electrotechnical Commission [IEC 60812:2018], 2018; International Electrotechnical Commission [IEC 61882:2016], 2016, checked against the IEC catalogue on 2026-11-07; IEEE [Std 1012-2016], 2017; Meyer, 1992; Popper, 1959/2002; Shingo, 1986; U.S. Department of Defense [MIL-STD-1629A], 1980; Yuan et al., 2014) were verified against their primary source — DOI, ISBN, or standard catalogue entry — on 2026-07-08.

Appendix B — Evaluation set and reproducibility

The five engineering evaluations are committed in the companion repository, in the folder plugins/anticipating-shadow-points/skills/anticipating-shadow-points/evals/:

- 01-supabase-migration.md (E1)
- 02-edge-function-deploy.md (E2)
- 03-refactor-shared-util.md (E3)
- 04-rls-policy-change.md (E4)
- 05-cron-skill-conflict.md (E5)

Each file contains the INPUT statement, the EXPECTED shadow-points list, and the ACCEPTANCE threshold, all authored by the protocol's designer. The benchmark package in benchmark/ provides:

- a deterministic 4-layer adjudicator (exact / synonym / Jaccard / unmatched);
- per-run cryptographic provenance (env, code, inputs, outputs and adjudication hashed into a Merkle-style chain);
- a verify subcommand that re-walks a run directory and reports any byte-level drift in the files the manifest covers;
- 31 unit tests covering the adjudicator, the evaluation and reply parsers, the manifest and the hasher, including the determinism contract.

Reproducing the computed numbers of Sections 5.3–5.5 (no model calls). Table B1 lists what the repository holds for them.

Table B1
What the repository holds for Sections 5.3 to 5.5

Material	Where in the repository	What a reader can check with it
The five evaluations	plugins/anticipating-shadow-points/skills/anticipating-shadow-points/evals/	Each task, its expected list and its acceptance rule (Table 4)
The author's sample plans (RED and GREEN) and the run that scored them	benchmark/tests/fixtures/mock_runs/ and benchmark/runs/published/	The plans the matcher scored and the scores behind Figure 2
The July 2026 claude -p run of the companion note	data/spike-2026-07-08/	The counts quoted in Section 5.4
The 2026-11 study	benchmark/study-2026-11/	Its protocol, tasks, reference lists, 90 trials, both judges' verdicts and hash ledger, behind Section 5.5 and Figure 4
The calculation	analysis/results.py and benchmark/study-2026-11/analyze.py	How each number of Sections 5.3 to 5.5 is computed
The checks	tests/test_paper_numbers.py and tests/test_study_2026_11.py	That each of those numbers matches the data, and that the scored run is labelled mock
The seal	output/hash-chain.md	One SHA-256 fingerprint over code, data and results, so that any later change shows

A SHA-256 fingerprint is a 64-character code computed from a file; changing a single byte of the file changes the code. A single command, python run_all.py, run inside the repository folder, redoes everything in Table B1 and calls no model; it needs Python 3.11 or newer, and the repository's README lists the four commands that download the repository, install what it needs and run it. The author's observations in Section 5.2 cannot be reproduced: the model outputs they were read from were not kept.

Collecting real evidence (costs model calls). Install the benchmark package and run asp-benchmark run --mode real --model <model> followed by asp-benchmark verify <run-id>. This package's runner was not used for any published number. The 2026-11 study of Section 5.5 has its own runner, judge and hash ledger in benchmark/study-2026-11/, and python run_all.py re-derives that study's numbers from the committed verdicts without calling a model. The protocol is specified independently of the Python implementation in benchmark/PROTOCOL.md so it can be re-implemented in another language.

Technical Note

Version 2.4 · November 2026 · first published 18 May 2026

Series Codex Hash Research Report 2026-01

License Paper text, figures and data: CC BY 4.0. Companion software: MIT.

The ASP project. ASP (Anticipating Shadow Points) is an open-source plugin for Claude Code, Anthropic's command-line coding agent, that makes the agent list what could go wrong before it writes code. Its plugin and replication material sit in one public repository (Flores, 2026a), this experience report describes how it works, and a technical note measures one of its rules on its own: never trust the exit code of claude -p (Flores, 2026b). Section 5.5 reports the first measurement of whether ASP improves an agent's plans; a third paper reports the confirmatory study of the one-sentence pre-mortem that it pointed to (Flores, 2026c).

DOI. A DOI (Digital Object Identifier) is a permanent link to a deposited work. Zenodo, a public research archive run by CERN, holds this experience report as its own record: https://doi.org/10.5281/zenodo.24065968, which always opens the newest version; this version, 2.4, is https://doi.org/10.5281/zenodo.24410420. The plugin, data and code are a separate record (Flores, 2026a) whose concept DOI, https://doi.org/10.5281/zenodo.20276631, always opens the newest release; release 2.0.0 is the one that matches this experience report. That record's May 2026 releases (1.0.0 to 1.0.3) presented the matcher's scores on the author's sample plans as a reading of real model output; release 2.0.0 replaces them.

Corrections. Version 1.0 of this report also presented the five evaluations as measured and blurred three things that Section 5 now separates: what the author observed in real dispatches, what the adjudicator shows about itself on constructed fixtures, and what has not been measured. Version

2.0 corrected this, added above the committed May summaries the note that withdraws their conclusion that ASP is “reproducibly better” (Section 5.2), and fixed the sliding-window defect of Section 5.3.

Translations and submission. The documentation translations are described in Section 4; submission plans depend on the follow-ups of Section 8.

Replication. Every number that Sections 5.3 to 5.5 compute can be recomputed from the repository without calling a model; Appendix B, Evaluation set and reproducibility, lists what is there and how to run it.

About the Author

Carlos Ulisses Flores is CTO and Chief Researcher at Codex Hash Research Laboratory, an independent R&D lab in São Paulo, Brazil. He builds production systems where silent failure is expensive — historically in algorithmic trading, embedded IoT and edge cryptography, and currently in agentic-AI tooling. The ASP skill described in this experience report grew out of that latter line of work; the empirical finding reported in the companion technical note grew out of using Anthropic's claude -p runner as an autonomous execution kernel and watching it return exit code 0 on a graceful agent refusal.

He writes in Portuguese, English, Spanish, and Italian.

Identifiers · ORCID: 0000-0002-6034-7765 · Lattes (Brazil's national registry of researcher CVs): 6905246706890561

Contact · c.ulisses@gmail.com · https://ulissesflores.com