

ASP: una *skill* operativa di pre-mortem per agenti di programmazione basati su LLM

Un resoconto di esperienza

Carlos Ulisses Flores

Codex Hash Research Laboratory

ORCID: 0000-0002-6034-7765

Codex Hash Research Report 2026-01

Data: 17 novembre 2026

Riassunto

Gli assistenti di programmazione basati su LLM producono spesso codice *localmente corretto* che omette modi di fallimento che un ingegnere esperto solleverebbe in un *pre-mortem*, e la produzione si rompe in modi costosi. Chiamiamo *shadow point* questi elementi mancanti. Questo resoconto descrive Anticipating Shadow Points (ASP), un protocollo di pianificazione in 13 fasi distribuito come *skill* di Claude Code che integra il *pre-mortem* di Klein (2007) con la tassonomia dei fallimenti multi-agente di Cemri et al. (2025), controlla il piano con un validatore estraneo al contesto del pianificatore, secondo la letteratura sui limiti dell'autocorrezione (Huang et al., 2024; Tyen et al., 2024), e, opzionalmente, affida l'esecuzione a un sottoprocesso figlio con contratto su file. Documentiamo il progetto, il rilascio come *plugin open source* e osservazioni preliminari da cinque valutazioni di ingegneria, lette dall'autore senza cecità né trascrizioni conservate. Uno studio preregistrato ha misurato il protocollo su sei repository pubblici: 90 esecuzioni con Claude Opus 5.5, rischi di riferimento elencati alla cieca da tre modelli di altri fornitori e due giudici ciechi. ASP ha coperto il 48% dei rischi di riferimento, contro il 66% di Claude Code semplice invitato a elencare, prima di pianificare, cosa può andare storto. È rimasto indietro in tutti i sei compiti: nessun beneficio misurabile secondo la regola prefissata. Quella richiesta ha portato la copertura, rispetto a una semplice richiesta di pianificazione, dal 56% al 66% ($p = ,001$), un confronto secondario preregistrato che uno studio confermativo, riportato a parte, ha già verificato come domanda principale.

Parole chiave: agenti di programmazione LLM; pre-mortem; modi di guasto multi-agente; Claude Code; protocolli operativi; preregistrazione

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

Indice

1. Introduzione 5
1.1. Contributi 7
2. Contesto 8
2.1. Pre-mortem e prospective hindsight 8
2.2. Modi di fallimento multi-agente (MAST) 9
2.3. Limiti dell'autocorrezione e isolamento del validatore 10
2.4. Il lavoro correlato più vicino: PreFlect 11
2.5. La genealogia classica dell'ingegneria del software e della qualità 12
3. Il protocollo ASP 17
3.1. Panoramica 17
3.2. Posizionamento architetturale 18
3.3. Iron Law 20
3.4. Il contratto dei micro-TODO 21
4. Rilascio operativo 22
5. Osservazioni preliminari 23
5.1. L'insieme di valutazioni 24
5.2. Che cosa è stato osservato in esecuzioni reali (autore, non in cieco) 24
5.3. Che cosa mostra il giudicatore deterministico di sé 25
5.4. Il risultato sul codice di uscita di claude -p 28
5.5. Una misurazione preregistrata (2026-11) 30
6. Ipotesi di progetto 33
7. Minacce alla validità 35
8. Limiti e lavori futuri 37
9. Conclusione 38
10. Dichiarazioni 40
Ringraziamenti 41
Riferimenti bibliografici 42
Appendice A — Stato della verifica delle citazioni 48
Appendice B — Insieme di valutazioni e riproducibilità 50
Nota Tecnica 53
Sull'autore 55

1. Introduzione

Nella pratica dell'autore, il lavoro ingegneristico di routine — aggiungere una colonna a un database di produzione, distribuire una funzione che chiama un'API esterna, rifattorizzare un'utility usata in trenta file — è oggi spesso affidato ad assistenti di programmazione basati su IA. Quando l'assistente fa la cosa giusta, la giornata dell'ingegnere si libera. Quando sbaglia in modo *sottile* — scrivendo codice che compila, supera i propri test, va in produzione, ma omette in silenzio un passo che un ingegnere senior avrebbe segnalato in anticipo — la produzione si rompe in modi costosi da diagnosticare. La scienza delle decisioni ha una tecnica proprio per questa lacuna, il pre-mortem (Klein, 2007), e la ricerca sugli agenti dispone ora di un catalogo di come falliscono i sistemi LLM multi-agente (Cemri et al., 2025). La nostra tesi è che le due cose appartengano allo stesso posto a runtime: prima di scrivere codice, un agente dovrebbe essere costretto a eseguire un pre-mortem contro una tassonomia di fallimenti fissa, e il suo piano dovrebbe essere controllato da un validatore che non può vedere il ragionamento del pianificatore.

Il modo di fallimento degli agenti di programmazione LLM che ha motivato ASP non è il codice *sbagliato*. È il codice *incompleto*: l'agente fa ciò che gli è stato chiesto, ma non ciò che qualcuno con una profonda conoscenza del dominio si sarebbe chiesto *prima di cominciare*. Il ritardo di replica durante una migrazione di schema. L'*header* CORS sulla *edge function*, che dice al browser quali altri siti possono chiamarla. La *policy* di sicurezza a livello di riga (RLS), che limita le righe che ogni utente può leggere e va aggiornata per corrispondere alla nuova colonna. Chiamiamo questi elementi mancanti **shadow point** — modi di guasto ovvi per un professionista che ne è stato scottato, invisibili a un agente che non lo è stato.

Due filoni di letteratura convergono su questa lacuna. Il primo, nella scienza delle decisioni, è il **pre-mortem**: un esercizio strutturato in cui un team immagina che un progetto sia già fallito ed

elenca le cause prima di agire. La tecnica è stata resa operativa per i team di progetto da Klein (2007) e fondata sul precedente lavoro empirico di Mitchell, Russo e Pennington (1989) sulla *prospective hindsight*; Klein (2007) la riassume come un aumento di circa il 30% della capacità di individuare le ragioni di un esito futuro, una cifra che non siamo riusciti a ricalcolare dall'articolo del 1989 (Appendice A). Il secondo, nella ricerca sugli agenti, è la recente caratterizzazione dei modi di fallimento multi-agente di Cemri et al. (2025), che hanno derivato una tassonomia di 14 modi, la Multi-Agent System Failure Taxonomy (MAST), da 150 tracce iniziali di sistemi LLM multi-agente (accordo tra annotatori κ = ,88) e hanno rilasciato un corpus di oltre 1.600 tracce annotate. La critica del piano di un agente prima che agisca esiste già (Sezione 2.4); ciò che non abbiamo trovato sono i due filoni integrati come **disciplina di runtime**, con artefatti con nome e un validatore che non condivide il contesto del pianificatore, che un agente di programmazione possa essere costretto a seguire a ogni compito non banale.

Questo articolo riferisce su **Anticipating Shadow Points (ASP)**, un protocollo di pianificazione in 13 fasi distribuito come *skill* di Claude Code che combina il ragionamento da pre-mortem con la *checklist* MAST a 14 modi, separa pianificazione e validazione con isolamento dei *prompt* e, nella sua modalità --goal *opt-in*, affida l'esecuzione a un sottoprocesso figlio con un contratto basato su file (per impostazione predefinita l'esecuzione resta nella sessione dell'utente). Abbiamo rilasciato la *skill*, distribuita attraverso tre percorsi di installazione (*marketplace* dei *plugin*, modalità sviluppatore, *script* autonomo), l'abbiamo sottoposta a cinque valutazioni ingegneristiche e riportiamo il progetto e ciò che le valutazioni possono e non possono mostrare. Queste cinque valutazioni sono preliminari e non misurate: GREEN (l'agente con la disciplina di ASP) su cinque compiti e RED (a cui si è chiesto di pianificare senza di essa) su tre, un'esecuzione ciascuno, un solo modello, letto dall'autore senza procedura in cieco, con le trascrizioni grezze

non conservate. Le riportiamo come osservazione qualitativa. Una misurazione preregistrata è arrivata dopo (Sezione 5.5): su sei compiti, le fasi di pianificazione di ASP hanno coperto meno rischi di riferimento di una richiesta di una frase. Le quattro scelte di progetto che trattiamo come ipotesi, e una quinta misurata nell'articolo complementare, sono nella Sezione 6.

1.1. Contributi

Questo resoconto di esperienza contribuisce con:

1. **Il protocollo ASP come progetto documentato** — tredici fasi, ciascuna delle quali emette un artefatto con nome, organizzate attorno al principio che nessuna implementazione inizia prima che gli shadow point siano stati portati alla luce.
2. **Dodici vincoli operativi che chiamiamo "Iron Law"** (Sezione 3.3), ciascuno legato a una fonte primaria della letteratura o a una decisione di progetto documentata. Due hanno un controllo a livello di *shell* nel *template* del *runner* che all'agente viene detto di eseguire (l'output JSON della Iron Law 12, un'impostazione predefinita del *template* che l'operatore può sovrascrivere, e il *flag* ASP_IN_GOAL della Iron Law 9, impostato per il figlio perché ASP non riparta al suo interno, il cui controllo all'ingresso è uno snippet Bash che la *skill* dice all'agente di eseguire); gli altri dieci sono istruzioni a livello di *prompt*.
3. **Osservazioni preliminari e qualitative** da cinque valutazioni ingegneristiche, e un giudicatore deterministico con provenienza crittografica, messo alla prova su *fixture* costruite (piani di esempio che l'autore ha scritto per collaudarlo) e pronto a valutare trascrizioni reali senza fidarsi del giudizio dell'autore (gli elenchi attesi restano

comunque dell'autore, e il file laterale di sinonimi facoltativo non è fornito per alcun compito).

4. **Un risultato empirico separato** sulla semantica del codice di uscita di claude -p (Claude Code eseguito in modo non interattivo: entra un *prompt*, esce un risultato) nel rifiuto garbato dell'agente, caratterizzato nell'articolo complementare Flores (2026b). Il risultato ha motivato due delle Iron Law ma merita una caratterizzazione indipendente su un N più grande.

Questo è un **resoconto di esperienza** nel senso dell'ingegneria dei sistemi, non un articolo di *benchmark*. Documentiamo un sistema che abbiamo costruito, una metodologia che abbiamo adottato e ciò che abbiamo osservato; non rivendichiamo un contributo empirico solido alla letteratura sui *benchmark* degli agenti LLM. La Sezione 7 elenca nel dettaglio le minacce alla validità.

2. Contesto

Passiamo in rassegna quattro filoni di lavoro recente che informano direttamente il progetto di ASP (§§2.1–2.4), poi collochiamo ASP entro le discipline più antiche di ingegneria del software e della qualità che esso traspone (§2.5). Manteniamo breve la rassegna del lavoro recente rispetto ai tipici articoli sui sistemi di agenti, perché il contributo qui è operativo e il contesto bibliografico è al suo servizio.

2.1. Pre-mortem e prospective hindsight

La tecnica del pre-mortem chiede a un team di immaginare, prima di impegnarsi, che un progetto sia già fallito in modo catastrofico, e di elencarne le cause. La base empirica è Mitchell, Russo e Pennington (1989), che hanno mostrato che immaginare un evento futuro come certo (anziché

incerto) produce spiegazioni più lunghe e più episodiche; Klein (2007) riassume l'effetto come un guadagno di circa il 30% nell'individuare correttamente le ragioni di un esito futuro. Klein (2007) ha reso operativo il risultato come "pre-mortem" di team di progetto. La sua base sperimentale è esile: non abbiamo trovato alcuna replica dell'effetto di circa il 30% che Klein (2007) attribuisce a Mitchell et al. (1989), né alcuna meta-analisi di esso, in Crossref, arXiv o in una ricerca web generale (novembre 2026). Si tratta di un'assenza in quegli indici, non della prova che non ne esistano, e non dice nulla sugli esperimenti relativi ad altri effetti del pre-mortem. Gli standard di valutazione del rischio catalogano tecniche affini — l'analisi di scenario e la tecnica strutturata what-if (SWIFT) — ma l'indice dell'IEC 31010:2019, Allegato B, non elenca il pre-mortem per nome. Il termine *prospective hindsight* è stato inoltre riutilizzato per un segnale di apprendimento per rinforzo non correlato (J. Zhang et al., 2026); noi lo usiamo solo nel senso di Mitchell et al. La critica del piano di un agente prima che agisca esiste già (PreFlect; Wang et al., 2026; §2.4). Ciò che non abbiamo trovato è l'integrazione che fa ASP: un pre-mortem eseguito contro una tassonomia di fallimenti fissa, registrato come artefatti con nome e rivisto da un validatore che non condivide il contesto del pianificatore.

2.2. Modi di fallimento multi-agente (MAST)

Gli agenti LLM alternano ragionamento e azione in un unico ciclo (Yao et al., 2023) e sono sempre più composti in flussi di lavoro multi-agente i cui modi di fallimento si sommano. Cemri et al. (2025) introducono **MAST** con **MAST-Data**, un corpus di oltre 1.600 tracce annotate provenienti da sette *framework* multi-agente diffusi come AG2 (già AutoGen; Wu et al., 2023). Da un'analisi rigorosa di 150 tracce iniziali (accordo tra annotatori κ = ,88), hanno derivato quattordici modi di fallimento organizzati in tre categorie: specifica e progetto del sistema (5 modi), disallineamento tra agenti (6 modi) e verifica e terminazione del compito (3 modi); le

tabelle della Sezione 3 citano questi modi con i loro codici, da FM-1.1 a FM-3.3. L'elenco dei 14 modi è il fondamento empirico della *checklist* di modi di fallimento di ASP: ASP adotta alla lettera i nomi dei modi e rende operativo ciascuno in un *prompt* di pre-mortem più una mitigazione nei file della *skill* complementare. Due lavori concomitanti estendono la direzione della tassonomia: Zhu et al. (2025) introducono AgentDebug e la AgentErrorTaxonomy (memoria / riflessione / pianificazione / azione / livello di sistema), e S. Zhang et al. (2025), un ICML 2025 Spotlight, apre l'**attribuzione automatica dei fallimenti** con il *dataset* Who&When. MAST è dunque una tassonomia tra le altre: TRAIL formalizza una tassonomia di errori agentici su 148 tracce annotate da esseri umani (Deshpande et al., 2025), AgentRx annota 170 traiettorie fallite con un passo critico e una categoria tratta da una tassonomia fondata sui dati (grounded theory) (Barke et al., 2026), e AgenTracer riporta che gli attuali LLM di ragionamento attribuiscono i fallimenti nelle tracce multi-agente con un'accuratezza generalmente inferiore al 10% (G. Zhang et al., 2025). ASP usa MAST come libreria di *prompt* per anticipare i fallimenti, non come classificatore di quelli osservati, quindi la scelta della tassonomia delimita ciò che la Fase 2 chiede anziché ciò che può rilevare.

2.3. Limiti dell'autocorrezione e isolamento del validatore

Le ragioni per separare il validatore dal pianificatore poggiano su tre lavori primari secondo cui gli LLM non sono affidabili nel criticare se stessi sotto contesto condiviso. Questi risultati motivano, ma non stabiliscono, l'ipotesi che separare il *prompt* del validatore dalla trascrizione del pianificatore migliori la validazione in ASP. Huang et al. (2024, ICLR) mostrano che l'autocorrezione intrinseca senza *feedback* esterno spesso degrada le prestazioni. Tyen et al. (2024) scompongono l'autocorrezione in localizzazione e correzione e mostrano empiricamente che gli LLM non riescono a *localizzare* i propri errori di ragionamento ma possono correggerli se la

posizione viene fornita dall'esterno — *la localizzazione è il collo di bottiglia*. Zheng et al. (2023, NeurIPS D&B) documentano che i giudici LLM forti favoriscono eccessivamente gli output di modelli della stessa famiglia (un "self-enhancement bias") anche quando giudicano oggettivamente la qualità. Kamoi et al. (2024) passano in rassegna questa letteratura e concludono che un'autocorrezione affidabile degli errori di ragionamento richiede *feedback* esterno o verificatori anziché il solo giudizio del modello; metodi di autofeedback intrinseco come Self-Refine (Madaan et al., 2023) migliorano alcuni compiti di generazione ma non colmano il divario di localizzazione isolato da Tyen et al. (2024). Il meccanismo che assumiamo, e che qui non mettiamo alla prova (Sezione 6), è che un validatore che condivide il contesto con un pianificatore erediti la calibrazione della fiducia del pianificatore sui contenuti condivisi, compresi i contenuti errati di cui il pianificatore è sicuro. Il progetto di ASP risponde istruendo l'isolamento dei *prompt* tra pianificatore e validatore (Iron Law 2 più avanti). Kambhampati et al. (2024) sostengono, per la pianificazione, che gli LLM auto-regressivi non possono da soli pianificare né auto-verificare, ma possono servire come fonti approssimate di conoscenza dentro un ciclo "LLM-Modulo" con verificatori esterni. I test di accettazione eseguibili di ASP sono verificatori di quel tipo esterno. Il suo validatore non lo è: è un secondo *prompt* allo stesso modello, che separa il contesto ma non il modello.

2.4. Il lavoro correlato più vicino: PreFlect

Il lavoro concomitante più vicino è **PreFlect** (Wang et al., 2026), che analogamente sposta il paradigma dell'autoriflessione dalla correzione a posteriori alla critica dei piani dell'agente prima dell'esecuzione, distillando gli errori di pianificazione dalle traiettorie storiche degli agenti e aggiungendo un meccanismo di ripianificazione dinamica. La somiglianza di fondo è significativa:

sia ASP sia PreFlect individuano nel carattere solo post-esecuzione della riflessione convenzionale un limite e spostano la critica *prima* dell'azione. ASP si differenzia da PreFlect lungo quattro assi:

1. **Validatore isolato a livello di *prompt*.** PreFlect distilla gli errori nella critica pre-esecuzione dell'agente stesso; ASP istruisce un sottoagente validatore con *prompt* separato a cui non viene data la trascrizione del pianificatore.
2. **Isolamento a livello di processo per l'esecutore.** Nella sua modalità --goal *opt-in*, ASP avvia l'esecutore come sottoprocesso claude -p con un contratto basato su file; l'articolo di PreFlect non descrive un confine di processo equivalente.
3. **Vincoli di runtime espliciti.** ASP enuncia dodici Iron Law, due delle quali protette da controlli eseguibili e dieci come regole a livello di *prompt*; PreFlect è presentato come una metodologia anziché come un sistema di vincoli.
4. **Un risultato empirico a livello di CLI** che caratterizza la semantica del codice di uscita di claude -p nel rifiuto garbato, e che ha motivato due delle Iron Law; è riportato separatamente in Flores (2026b). Il fenomeno stesso è documentato dal fornitore e in segnalazioni pubbliche di *issue* (catalogate in Flores, 2026b, §2); il contributo lì è un conteggio misurato e un contratto di terminazione tipizzato proposto, non una scoperta.

2.5. La genealogia classica dell'ingegneria del software e della qualità

I quattro filoni sopra collocano ASP nella recente letteratura sugli agenti LLM. I meccanismi di ASP sono tuttavia una trasposizione di discipline dell'ingegneria del software e della qualità che precedono gli LLM di decenni. Ciò che è nuovo in ASP non sono le discipline ma la loro trasposizione in un runtime di agenti non deterministico e i meccanismi che le fanno rispettare lì.

Controlli obbligatori delle postcondizioni: ASP e il *Design by Contract*. Lo schema dei micro-TODO di ASP (§3.4) è il *Design by Contract* (Meyer, 1992) spostato dal confine della routine al confine del compito dell'agente. Il contratto di Meyer assegna obblighi attraverso un confine tra chiamante e routine: il chiamante deve stabilire la PRECONDIZIONE, che la routine ha poi diritto di *assumere* — il principio di non ridondanza di Meyer vieta al corpo di ricontrollarla — e la routine deve garantire la POSTCONDIZIONE su cui il chiamante fa affidamento. Dei sette campi dello schema, PRECONDITION, ACTION e POSTCONDITION corrispondono direttamente a questo contratto, con ACTION come corpo della routine che la coppia pre/post racchiude. Meyer (1992) prevede anche il monitoraggio delle asserzioni a tempo di esecuzione, postcondizioni comprese, come opzione scelta in compilazione; per impostazione predefinita si monitorano solo le precondizioni. ASP trasforma quell'opzione in regola: poiché l'obbligato è un modello non deterministico anziché una routine compilata, la postcondizione di ogni passo è verificata da un ACCEPTANCE-TEST esterno, un comando eseguito fuori dal modello. Il compromesso: ASP rende obbligatorio il monitoraggio delle postcondizioni, pagando un controllo a ogni passo, in tempo e in token, per ottenere un'evidenza esplicita che il passo è stato completato, perché nulla garantisce che un modello abbia rispettato il proprio contratto. La regola si ispira al *Design by Contract* ma è distinta dal rifiuto di Meyer dei controlli ridondanti delle precondizioni, che riguarda una routine che ricontrolla ciò che il chiamante ha già garantito. Il FALSIFICATION-TEST aggiunge la posizione popperiana (Popper, 1959/2002) secondo cui un'affermazione guadagna fiducia sopravvivendo a un tentativo di confutazione anziché da un controllo di conferma soltanto; funziona come un test negativo accanto al contratto, non come una clausola di esso. La stessa posizione produce la disciplina dei codici di uscita delle Iron Law 11–12. Le Iron Law, a loro volta, sono enunciate come l'analogo del terzo elemento del contratto del DbC — l'invariante

di classe — anziché come una pre/postcondizione per singolo compito; dieci di esse sono istruzioni che l'agente deve rispettare, non controlli che vengono eseguiti.

Enumerazione dei modi di guasto. Il rilevamento degli shadow point (Fase 2) è un'enumerazione prospettica dei modi di guasto nella tradizione dell'analisi dei modi di guasto e dei loro effetti (FMEA; MIL-STD-1629A, 1980; IEC 60812:2018). Non è deliberatamente una FMEA completa: ASP non esegue alcuna scomposizione per componente né alcun punteggio di priorità di rischio (gravità × occorrenza × rilevabilità), quindi il debito è verso la disciplina di enumerazione prospettica della FMEA, non verso i suoi meccanismi bottom-up. Tre input della Fase 2 occupano ruoli che vale la pena distinguere: il pre-mortem (Klein, 2007; §2.1) fornisce la *filosofia* — immaginare il fallimento come già certo; la tassonomia MAST a quattordici modi (Cemri et al., 2025; §2.2) fornisce la *libreria* di modi di guasto; e l'enumerazione forzata rispetto a quella libreria fissa è più vicina, strutturalmente, al prompting con parole guida degli studi di pericolo e operabilità (HAZOP; IEC 61882:2016), in cui un elenco fisso di parole orienta la ricerca delle deviazioni, che alla FMEA componente per componente. Ciò che ASP aggiunge è il substrato — il "componente" di cui si enumerano i modi di guasto è un piano LLM; il pianificatore esegue l'enumerazione (Fase 2), e un validatore che non condivide il contesto del pianificatore rivede l'elenco risultante (Fase 5; §2.3).

Prevenzione degli errori. Un sottoinsieme delle Iron Law (§3.3) sono *poka-yoke* (Shingo, 1986), dispositivi che rendono un errore fisicamente impossibile, a livello di protocollo: il *template* del *runner* in claude-p-goal-runner.md imposta ASP_IN_GOAL per il figlio e passa --output-format json per impostazione predefinita (Iron Law 9 e 12); il contratto basato su file della Iron Law 10 è un'istruzione. Questi controlli vivono in un *template* Bash che all'agente viene detto di eseguire, non in un *hook* (un comando che l'esecutore stesso lancia) al di fuori del controllo dell'agente, e

non sono l'impossibilità fisica del vero *poka-yoke* — un *flag* d'ambiente può essere rimosso o non propagarsi — e le restanti leggi sono porte procedurali (Iron Law 1, 2, 4, 5) o regole di verifica che dipendono ancora dall'operatore, di cui la Iron Law 11 ("mai fidarsi di $?", il valore in cui la *shell* registra l'ultimo codice di uscita) è il caso più chiaro: proprio la dipendenza dalla vigilanza dell'operatore contro cui il *poka-yoke* è definito. Il prestito è la *posizione* di prevenzione degli errori applicata a due controlli a livello di *shell* in uno *script runner* che all'agente viene chiesto di emettere, non l'affermazione che ASP elimini la vigilanza dell'operatore.

Verifica e validazione indipendenti. Il validatore di ASP isolato a livello di *prompt* (Iron Law 2) trasferisce la mossa che definisce la Verifica e Validazione Indipendenti (IV&V; IEEE Std 1012-2016) — un verificatore mantenuto indipendente dallo sviluppatore, nei termini dell'IEEE 1012 dal punto di vista tecnico, gestionale e finanziario — dalla separazione organizzativa alla separazione *di prompt*: il sottoagente validatore non condivide alcun contesto con il pianificatore che controlla. La giustificazione specifica per gli LLM di tale isolamento è la letteratura sull'autocorrezione della §2.3; la *forma* della contromossa è quella dell'IV&V.

Vocabolario della fidatezza. Il comportamento dei codici di uscita alla base delle Iron Law 11–12 ha un nome esatto nella tassonomia della fidatezza: un exit 0 su un rifiuto garbato è un *fallimento non segnalato* (Avizienis et al., 2004), né un crash né un arresto fail-silent (che si ferma senza emettere nulla). Il termine, la sua letteratura precedente e la misura sono sviluppati nell'articolo complementare (Flores, 2026b); questa sezione importa solo il vocabolario e non riapre qui la questione della novità.

Garanzia empirica. La preferenza per i test di accettazione e di falsificazione rispetto ai codici di stato di cui ci si fida ha una base empirica nella patologia adiacente. Yuan et al. (2014) hanno trovato che una larga maggioranza dei fallimenti catastrofici nei sistemi distribuiti ad alta intensità

di dati risale alla gestione difettosa di errori non fatali che *erano* stati segnalati. Il caso del codice di uscita di ASP è il complemento — il segnale era *assente* — quindi i due non condividono una tesi; la stretta lezione che condividono è che un canale di stato non è un sostituto dell'esito, cosa che un ACCEPTANCE-TEST esterno esiste per imporre.

Tabella 1 mappa ciascun meccanismo di ASP al suo antenato classico e indica che cosa ASP aggiunge.

Tabella 1
I meccanismi di ASP e i loro antenati classici

Meccanismo di ASP	**Antenato classico**	**Che cosa ASP aggiunge (e *non* rivendica)**
micro-TODO PRE/ACTION/POST + ACCEPTANCE-TEST	*Design by Contract* (Meyer, 1992)	un obbligato non deterministico la cui postcondizione è verificata da un comando esterno a ogni passo, mentre il monitoraggio a tempo di esecuzione di Meyer è un'opzione; non l'idea di contratto
micro-TODO FALSIFICATION-TEST	Falsificazione popperiana (Popper, 1959/2002)	un passo di confutazione al confine del compito dell'agente; un test negativo accanto al contratto, non una clausola di esso
Rilevamento degli shadow point (Fase 2)	Enumerazione in stile FMEA (MIL-STD-1629A; IEC 60812:2018) + parole guida HAZOP, sulla libreria MAST (Cemri et al., 2025)	enumerazione su un piano LLM da parte del pianificatore, rivista da un validatore isolato a livello di *prompt*; nessun punteggio di priorità di rischio, nessuna scomposizione per componente
Iron Law 9 e 12 come controlli a livello di *shell*	*Poka-yoke* (Shingo, 1986)	due controlli in uno *script runner* che all'agente viene chiesto di emettere; non impossibilità fisica, e la Iron Law 10 è un'istruzione
Validatore isolato a livello di *prompt* (Iron Law 2)	IV&V (IEEE Std 1012-2016)	indipendenza organizzativa riformulata come isolamento di *prompt*; non IV&V
Mai fidarsi di $? (Iron Law 11–12)	Fallimento non segnalato (Avizienis et al., 2004); Yuan et al. (2014)	caratterizzato nell'articolo complementare (Flores, 2026b); non la classe di fallimento

Letta nel suo insieme, la sezione avanza una pretesa modesta: i meccanismi di ASP sono un'*integrazione* di discipline consolidate di affidabilità e ingegneria della qualità — contratto,

enumerazione dei modi di guasto, IV&V, prevenzione degli errori — in un unico protocollo il cui contributo è operativo anziché concettuale. Ciò che tale integrazione aggiunge oltre i suoi antenati è la loro applicazione in un runtime di agenti non deterministico: le Iron Law (due delle quali protette da controlli eseguibili), l'isolamento a livello di *prompt* e di processo della §3.2 e il giudicatore deterministico riproducibile della §5, con il risultato sul codice di uscita di claude -p caratterizzato separatamente nell'articolo complementare. Questa è la tesi del resoconto di esperienza della §1, non una pretesa di novità rispetto alla letteratura classica.

3. Il protocollo ASP

3.1. Panoramica

ASP è un protocollo in 13 fasi che procede dall'acquisizione (*intake*) fino all'approvazione della consegna. Ogni fase emette un artefatto con nome (Tabella 2). Il principio che organizza le fasi è che nessuna implementazione inizia prima che gli shadow point siano stati portati alla luce (Fase 2), rivisti da un validatore isolato a livello di *prompt* (Fase 5; la modalità --quick salta questa revisione, quella predefinita e la --paranoid la mantengono) e approvati dall'utente (Fase 7); l'esecuzione resta per impostazione predefinita nella sessione dell'utente e, con il *flag* --goal *opt-in*, è delegata a un sottoprocesso figlio con un contratto basato su file (Fase 9).

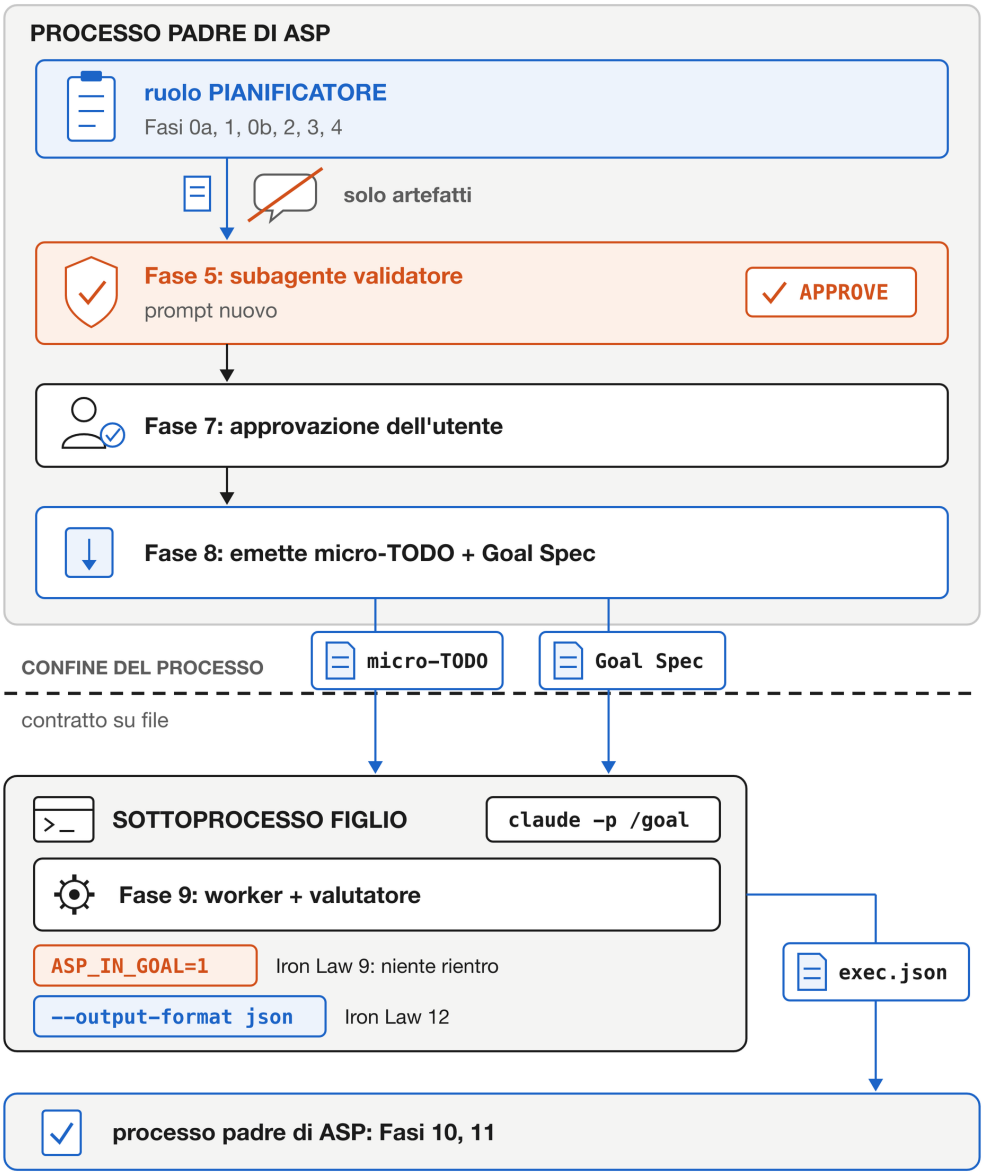
Tabella 2
Fasi di ASP e artefatti che emettono. La fase 0 ha due sotto-fasi (0a prima della ricerca, 0b dopo), perciò tredici fasi occupano quattordici righe

#	Fase	Artefatto	Motivazione di progetto
0a	Q&A pre-ricerca	blocco intake	Buona pratica di Anthropic *Skills*: output intermedi verificabili
1	Ricerca parallela	Sintesi della ricerca + blocco di *recall*	*Pattern* di dispatch dei sottoagenti
0b	Q&A post-ricerca	*Intake* raffinato (saltabile)	Come 0a
2	Rilevamento degli shadow point	Elenco categorizzato + mitigazioni	Klein (2007) + Cemri et al. (2025)
3	*Project Charter*	*Charter* compilata	Passaggio durevole basato su artefatti
4	Piano macro + registro	Piano macro + deliverable D01–D*n*	Plan-and-Act (Erdogan et al., 2025)
5	Validatore indipendente	APPROVE / REVISE	Huang et al. (2024) + Tyen et al. (2024) + Zheng et al. (2023)
6	Ciclo interno (limite=3)	Piano della revisione finale	Iterazione limitata; FM-1.5
7	Approvazione dell'utente	Go / no-go	Porta con intervento umano (*human-in-the-loop*)
8	Micro-TODO + Goal Spec	Un task TaskCreate (lo strumento per la lista dei task di Claude Code) per passo + Goal Spec (la condizione di completamento scritta per l'esecutore)	Passaggio all'esecutore
9	Esecuzione, nella sessione o, con il *flag* --goal *opt-in* di ASP, in un sottoprocesso figlio che esegue il comando /goal di Claude Code	Task contrassegnati come completati con evidenza	Isolamento a livello di processo nella modalità *opt-in*; si veda Flores (2026b)
10	Approvazione per singolo deliverable	Ogni deliverable accettato o rifiutato	Contratto di accettazione
11	Rapporto di esecuzione + memoria	Artefatto di chiusura + scrittura in memoria	Memoria in stile Reflexion (Shinn et al., 2023)
12	Rilascio pubblico (*opt-in*)	*Marketplace* dei *plugin* + *release* GitHub	Livello di distribuzione

3.2. Posizionamento architetturale

ASP separa tre ruoli di agente con passaggi esclusivamente tramite artefatti (Figura 1).

Figura 1
La separazione di ASP in tre ruoli



Nota. Le fasi del pianificatore girano nel processo padre. Il validatore gira nel padre ma in un *prompt* di sottoagente nuovo che riceve solo artefatti, mai la trascrizione del pianificatore. Con il *flag* --goal *opt-in*, il caso qui disegnato, il *worker* gira in un sottoprocesso figlio claude -p con un contratto basato su file; per impostazione predefinita gira nella sessione dell'utente.

3.3. Iron Law

Enunciamo dodici vincoli operativi e li chiamiamo Iron Law (Tabella 3). Le Iron Law 9 e 12 hanno controlli a livello di *shell* nel *template* del *runner* e nello snippet d'ingresso che all'agente viene detto di eseguire; le altre sono regole che la *skill* istruisce l'agente a seguire. La conformità a entrambi i tipi non è stata misurata. Ciascuna è mappata alla propria derivazione: una fonte primaria della letteratura, una raccomandazione documentata di Anthropic o una decisione di progetto empirica.

Tabella 3
Le 12 Iron Law e la loro derivazione

#	Legge	Derivazione
1	Nessuna implementazione senza prima un passaggio sugli shadow point	Klein (2007) + Cemri et al. (2025)
2	Nessuna approvazione del piano senza un passaggio di validatore indipendente (*prompt* separato); la modalità --quick salta il validatore	Huang et al. (2024) + Tyen et al. (2024) + Zheng et al. (2023); mitiga FM-3.3
3	Nessun micro-TODO senza piano macro + *charter* + deliverable approvati dall'utente	Anthropic *Skills*: output intermedi verificabili
4	Nessun completamento di passo senza evidenza fresca del test di accettazione	Cemri et al. (2025) FM-3.1 + FM-3.2 + FM-2.6
5	Nessun deliverable accettato senza file/output di evidenza corrispondente	Cemri et al. (2025) FM-3.1
6	Nessuna chiusura di progetto senza rapporto di esecuzione + scrittura in memoria	Buffer di memoria di Shinn et al. (2023)
7	Nessun ciclo senza limite (limite predefinito=3 round di validatore)	Cemri et al. (2025) FM-1.5
8	Nessuna esecuzione di Fase 9 senza dispatch del valutatore ai checkpoint (tranne con --quick)	Erdogan et al. (2025) pianificatore/esecutore
9	Nessun ASP rientrante dentro il figlio claude -p /goal (protezione con variabile d'ambiente ASP_IN_GOAL)	Mitigazione dell'esplosione dei costi; decisione di progetto
10	Nessun avvio di claude -p /goal senza contratto basato su file	Mitigazione della divergenza di stato del sottoprocesso; decisione di progetto
11	Mai fidarsi di $? di claude -p per il successo/ fallimento semantico	Osservazione empirica, caratterizzata in Flores (2026b)
12	Tutte le invocazioni di claude -p devono usare --output-format json o stream-json	Corollario della Iron Law 11

3.4. Il contratto dei micro-TODO

Al confine tra pianificazione ed esecuzione (Fase 8), ASP emette un elenco di task contrattuale. Ogni task è creato tramite la primitiva TaskCreate con sette campi etichettati:

- **PRECONDITION** — invariante che deve valere prima del passo.
- **ACTION** — comando esatto o modifica letterale.
- **POSTCONDITION** — cambiamento di stato effettuato dal passo.
- **ACCEPTANCE-TEST** — comando e output atteso che dimostrano la postcondizione.
- **FALSIFICATION-TEST** — comando od osservazione che dimostrerebbe il fallimento del passo.
- **DEPENDS-ON** — identificatori dei task a monte (un grafo aciclico orientato, o DAG: dipendenze senza cicli).
- **DELIVERABLE-ID** — a quale deliverable del registro contribuisce questo passo.

Questo schema è ciò che rende verificabile l'elenco dei task: nessun task è contrassegnato come completato senza evidenza del test di accettazione (Iron Law 4); nessun deliverable è approvato senza evidenza corrispondente (Iron Law 5). Lo schema è condiviso con la Goal Spec che il sottoprocesso della Fase 9 consuma nella modalità --goal *opt-in*.

4. Rilascio operativo

ASP è distribuito come *plugin* di Claude Code attraverso tre percorsi di installazione, ciascuno adatto a un pubblico diverso:

1. ***Marketplace* dei *plugin*** (consigliato per gli utenti finali):

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

   Invocazione: /anticipating-shadow-points:asp <task> oppure semplicemente /asp.

2. **Modalità sviluppatore** (per i contributori): claude --plugin-dir ./plugins/anticipating-shadow-points carica la *skill* sul posto; utile quando si itera sulla *skill* stessa.

3. ***Script* di installazione autonomo** (per chi non usa il gestore di *plugin*): ./scripts/install.sh copia la *skill* nella directory delle *skill* di Claude Code dell'utente.

Il *plugin* usa il formato ufficiale di *manifest* .claude-plugin/plugin.json di Anthropic (Claude Code 2.1+; Anthropic, 2026a). Sta in una cartella propria, plugins/anticipating-shadow-points/, che contiene solo Markdown, l'icona e la licenza, così l'installazione non copia altro; la radice del repository funziona anche da *marketplace* proprio tramite .claude-plugin/marketplace.json.

Permessi. Dalla versione 1.0.4 del *plugin* il *runner* della Fase 9 della modalità --goal *opt-in* avvia il figlio claude -p **senza alcun *flag* di permesso**: il figlio gira con le impostazioni di permesso di Claude Code dell'utente stesso, e la *skill* non li concede, non li amplia né li aggira mai. Le versioni precedenti passavano --permission-mode bypassPermissions; ciò è stato rimosso dopo che la directory dei *plugin* di Claude, esaminando il *plugin*, ha segnalato che un *plugin* non deve concedersi permessi da sé (CHANGELOG.md, versione 1.0.4). In modalità non interattiva

un'azione che l'utente non ha preautorizzato viene negata anziché richiesta; il padre legge il campo permission_denials del risultato JSON del figlio e segnala all'utente ogni diniego anziché riprovare con un permesso più ampio. Questo rende l'esecuzione autonoma strettamente più debole che nel rilascio di maggio 2026, e più sicura: la *skill* può fare solo ciò che l'operatore ha già consentito. Dalla versione 2.0.0 la *skill* avvisa inoltre l'utente, prima di avviare il primo figlio, che questo gira con i permessi e il consumo dell'utente stesso, con budget e limite di turni dichiarati, e lo avvia solo dopo il suo OK; i file di lavoro del figlio vanno nella cartella dati del *plugin*, che Claude Code rimuove quando il *plugin* viene disinstallato. La *skill* stessa è implementata come *skill* in Markdown secondo il documento di buone pratiche delle Agent *Skills* di Anthropic (Anthropic, 2026b), con file complementari per la *checklist* MAST, i *prompt* di pre-mortem per singolo modo e i sette *template* di artefatto citati nella Sezione 3.1.

La documentazione è pubblicata come sito GitHub Pages, con traduzioni del README in portoghese, spagnolo, italiano ed ebraico assistite da macchina o da IA (docs/translations.md). Il codice sorgente completo — file della *skill*, *manifest* dei *plugin*, documentazione, artefatti di test, codice del *benchmark* — è concesso in licenza MIT; il testo del paper, le figure e i dati sono CC BY 4.0.

5. Osservazioni preliminari

Abbiamo sottoposto ASP a cinque valutazioni ingegneristiche strutturate che coprono domini eterogenei. Questa sezione separa tre cose: ciò che l'autore ha osservato in esecuzioni reali del modello (5.2), ciò che il giudicatore deterministico mostra di sé su *fixture* costruite (5.3) e ciò che non è stato affatto misurato.

5.1. L'insieme di valutazioni

La Tabella 4 elenca le cinque valutazioni ingegneristiche.

Tabella 4
Le cinque valutazioni ingegneristiche

ID	Dominio	Compito (abbreviato)	Dimensione attesa	Accettazione
E1	Migrazione di schema	Colonna NOT NULL aggiunta a user_profiles da 1M righe	10	≥ 80%
E2	*Deploy* su edge	*Edge function* che chiama un'API esterna con *rate limit*	10	≥ 80%
E3	Refactoring	Utility di formattazione delle date usata in oltre 30 file	10	≥ 80%
E4	*Policy* RLS	*Policy* di sicurezza a livello di riga che fa riferimento a una nuova colonna	10	≥ 80%
E5	Pianificazione	Cron job il cui *script* è in conflitto con un servizio disattivato	10	≥ 80% e punti 1 e 8 presenti

Ogni valutazione specifica un INPUT (un enunciato del compito come un ingegnere senior potrebbe formularlo a un collega), un elenco EXPECTED di shadow point che un professionista competente farebbe emergere e una soglia di ACCEPTANCE (predefinita 80%). E5 non è portabile: due dei suoi dieci punti attesi (1 e 8) dipendono dalla memoria privata di progetto dell'autore e non possono emergere dal solo INPUT, perciò misura il richiamo dalla memoria tanto quanto la pianificazione. Il testo completo di ciascuna valutazione è nel repository complementare.

5.2. Che cosa è stato osservato in esecuzioni reali (autore, non in cieco)

Il 2026-05-17 le valutazioni sono state assegnate a sottoagenti nuovi: GREEN (con la disciplina ASP iniettata: inquadramento da pre-mortem, enumerazione forzata dei 14 modi MAST, obbligo di mitigazione) su tutte e cinque, e RED (a cui era chiesto di pianificare come farebbe un ingegnere senior, senza alcun *framework*) su tre, E1–E3; E4 ed E5 sono state aggiunte dopo ed eseguite solo in GREEN. Tutte sono girate sul modello di sessione dell'epoca (Claude Opus 4.7; il modello non

è stato registrato per singola esecuzione), un'esecuzione per condizione. I sottoagenti hanno ereditato le istruzioni globali dell'agente dell'autore, perciò la condizione RED era una base di riferimento competente, non ingenua. L'autore ha letto ogni piano rispetto all'elenco atteso e ha registrato un giudizio di copertura; i riepiloghi, con gli identificatori dei sottoagenti, sono sottoposti a *commit* in tests/baseline-pressure-tests.md e tests/evals-summary.md. Le trascrizioni grezze non sono state sottoposte a *commit* e non sono più disponibili. I *prompt* RED lì registrati sono formulazioni in portoghese più brevi di E1–E3 che omettono dettagli che gli elenchi attesi valutano poi: il *prompt* RED di E2 non indica il limite di frequenza di 100 req/s di Resend, l'API di posta elettronica che il compito chiama, e quello di E1 non dice che il codice dell'applicazione fa già riferimento alla nuova colonna in cinque punti di query. Parte del divario di RED può dunque essere un artefatto del *prompt*, non dell'assenza del protocollo.

Con questa lettura non in cieco, i tre piani RED hanno coperto all'incirca il 30–60% dei rispettivi elenchi attesi (media prossima al 47%, contando le menzioni parziali come coperte) e i piani GREEN hanno coperto tutti o quasi i propri. I riepiloghi sottoposti a *commit*, scritti a maggio 2026, concludono che ASP è "riproducibilmente migliore" (traduzione nostra); una nota sopra di essi ritira quella conclusione, che non è sostenuta. Queste cifre sono un giudizio dell'autore su testo non conservato, valutato rispetto a elenchi scritti dallo stesso autore, e con il vocabolario MAST iniettato dal *prompt* GREEN che compare in quegli elenchi. Sono riportate come osservazione, non come misura, e nessun numero di questo paragrafo è verificato dal livello di replicazione.

5.3. Che cosa mostra il giudicatore deterministico di sé

Il pacchetto benchmark/ valuta un piano rispetto a un elenco atteso con un *matcher* deterministico a 4 livelli, la parte che decide se un piano menziona un punto atteso: nome esatto, file laterale di sinonimi (un elenco facoltativo di sinonimi accettati), finestra scorrevole di

Jaccard (un punteggio di sovrapposizione di parole su un tratto mobile del piano) e non corrispondente. Nell'esecuzione pubblicata tutte le 42 corrispondenze provengono dal primo livello (nome esatto); non esiste alcun file laterale di sinonimi per alcun compito, e il livello di Jaccard non scatta mai. La sua unica esecuzione pubblicata è un'esecuzione in **modalità *mock***: i testi RED e GREEN sono *fixture* che l'autore ha scritto per esercitare la *pipeline* da un capo all'altro (benchmark/tests/fixtures/mock_runs/); su di esse i livelli 2 e 3 non producono mai una corrispondenza (le sovrapposizioni del livello 3 sono calcolate e restano sotto la soglia), e solo i test unitari esercitano una corrispondenza attraverso quei livelli. Le *fixture* stesse sono state scritte deliberatamente più deboli per RED e deliberatamente allineate all'elenco atteso per GREEN. Su quelle *fixture* il *matcher* valuta RED al 4% e GREEN all'80% in media — un'affermazione su come sono state scritte le *fixture*, non su ASP.

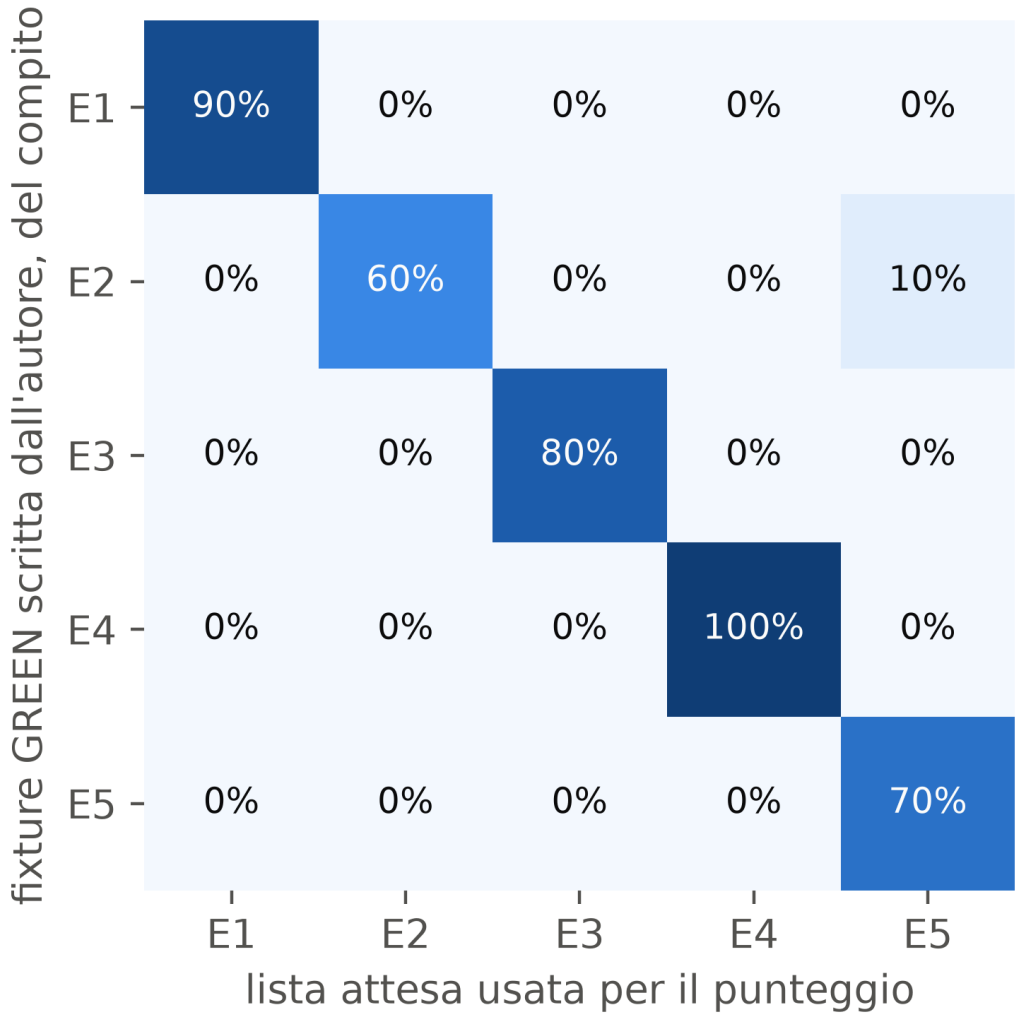
In simboli, il livello 3 valuta una finestra del piano rispetto a un punto atteso con l'Equazione 1, dove A è l'insieme dei token distinti della finestra e B quello del punto atteso; le barre verticali contano i token di un insieme. Una corrispondenza richiede un punteggio di almeno ,30.

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

La proprietà utile che l'esecuzione *mock* stabilisce è la **discriminazione**: valutare ciascuna *fixture* GREEN rispetto all'elenco atteso di un compito *diverso* abbassa la media dall'80% allo 0,5% (la cella fuori compito più alta 10%; Figura 2). È un'evidenza più debole di quanto sembri: l'unica corrispondenza fuori compito è il nome generico di una sola parola "Alerting", e le altre celle sono zero perché il livello 1 richiede che un intero nome atteso compaia alla lettera. Il *matcher* è inoltre più severo dei file di valutazione, che dicono a un valutatore umano di accettare le parafrasi. Il *recall* è una questione a parte, e le *fixture* mostrano un falso negativo: la *fixture* RED di E2

menziona un *rate limit*, i *retry* e il *logging* e ottiene comunque 0/10: la sua migliore sovrapposizione di finestra con il punto atteso “Rate-limit backoff” (il suo nome e la sua descrizione) è un Jaccard di ,11, sotto la soglia di ,30, e il *matcher* non fa *stemming* (“logging” non corrisponde a “logs”). Le parafrasi con bassa sovrapposizione lessicale sfuggono. La rivalutazione ha inoltre portato alla luce un difetto, ora corretto: la finestra scorrevole saltava gli ultimi token di un testo la cui lunghezza non cadeva sul passo; rivalutare ogni *fixture* con la correzione non cambia alcun valore di copertura, benché la migliore sovrapposizione di tre punti non corrispondenti salga leggermente (E2 RED “Observability / logs” da ,00 a ,02), restando ben al di sotto della soglia. Il livello 3 ha anche un tetto aritmetico: Jaccard divide per l’unione, e una finestra di 40 token delle *fixture* contiene da 27 a 40 parole di contenuto distinte (mediana 36); in qualsiasi finestra con 34 o più, un punto atteso con dieci o meno token distinti non può raggiungere ,30 nemmeno se il piano li ripete tutti, e 19 dei 50 punti attesi sono così brevi. Lo studio 2026-11 (Sezione 5.5) ha sostituito il livello 3 con il *recall* dei token prima che esistesse qualsiasi lista di riferimento. Sulle liste dei revisori stessi, nemmeno quella versione vedeva che due revisori nominavano lo stesso problema, per cui un giudice cieco è diventato il punteggio principale prima di raccogliere qualsiasi dato; il *matcher* è riportato come secondario.

Figura 2
Verifica dello strumento: ogni fixture GREEN costruita dall'autore valutata rispetto all'elenco atteso di ogni compito. Non è un effetto di ASP



La Sezione 5.5 riporta la misurazione; la Sezione 6 enuncia le ipotesi di progetto su cui essa incide.

5.4. Il risultato sul codice di uscita di claude -p

Durante una batteria di quattro test contro il *runner* non interattivo claude -p, abbiamo osservato che il binario restituisce codice di uscita 0 anche quando l'agente rifiuta garbatamente l'obiettivo richiesto — l'*harness* (il programma che esegue l'agente) riporta una conclusione riuscita della conversazione nonostante l'agente rifiuti semanticamente il compito. Il codice di uscita riflette il

successo a livello di processo del binario della CLI, non il successo semantico del compito dell'agente. Le Iron Law 11 e 12 richiedono --output-format json o stream-json e la lettura di is_error, terminal_reason e stop_reason. Quei campi separano un'interruzione per limite di turni e un rifiuto del classificatore da un turno completato, ma non un rifiuto generato dal modello da una consegna; il *runner* segnala perciò anche le risposte il cui testo corrisponde a un *pattern* di rifiuto, un'euristica che l'articolo complementare mette alla prova.

Questo risultato è significativo sul piano osservativo per qualsiasi *toolchain* che incapsuli claude -p in una *pipeline* di *shell*. Poiché l'osservazione proveniva da un singolo scenario di rifiuto nella nostra batteria, lo **caratterizziamo in modo indipendente** nell'articolo complementare Flores (2026b), che riporta una misura su N=50 in quattro categorie di *prompt* che provocano un rifiuto (impossibile in un solo turno, capacità, sicurezza, ambiguità) su Claude Code 2.1.200 a luglio 2026. Il modello di sessione era Fable 5; Opus 4.8 ha risposto a 16 dei 50 *prompt*, compresi tutti i 12 *prompt* di sicurezza — coerentemente con un passaggio automatico a un modello di riserva, benché gli *envelope* (il record JSON che claude -p stampa alla fine di ogni esecuzione) registrino quale modello ha risposto, non perché. Ventiquattro prove sono uscite con 0, tutte con stop_reason: end_turn. L'articolo complementare conta in quattro modi i rifiuti tra di esse: 20 (40%, IC di Wilson al 95% 28–54%, un intervallo di confidenza adatto ai piccoli campioni) con l'etichetta non in cieco dell'autore; 19 (38%) con la maggioranza di un *codebook*, scritto dopo che l'autore aveva letto le risposte, e applicato da cinque valutatori LLM, quattro dei quali hanno restituito etichette identiche, tutti della stessa famiglia del modello sotto test, così che la maggioranza è un controllo del *codebook*, non una stima dell'accordo tra valutatori; 12 (24%) contando solo i rifiuti di sicurezza e di comandi distruttivi ed escludendo gli otto rifiuti motivati di compiti impossibili in un solo turno; e 9 (18%) con il rilevatore di frasi bloccato solo in inglese, che non coglie le risposte

scritte in portoghese. Un'istantanea precedente (maggio 2026, Claude Code 2.1.143, Opus 4.7) dava il 60% con lo stesso rilevatore, su un modello e una lingua di risposta diversi. Un secondo valutatore umano e una terza istantanea secondo una preregistrazione redatta sono i seguiti ancora aperti.

5.5. Una misurazione preregistrata (2026-11)

Lo studio è stato preregistrato nel repository (benchmark/study-2026-11/PROTOCOL.md), e il suo registro concatenato di hash ha ricevuto una marca temporale OpenTimestamps, un servizio pubblico di marcatura temporale, prima della prima esecuzione. Ha sostituito il Protocollo 2 di PREREGISTRATION.md, il piano precedente del repository per questa misurazione, mai depositato. Il disegno (Figura 3):

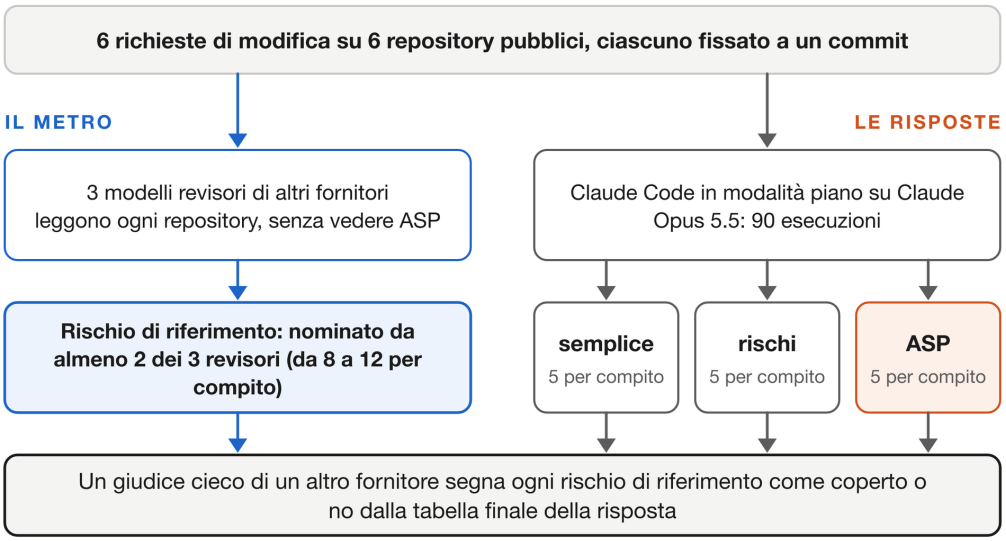
- **Compiti.** Sei richieste di modifica su sei repository pubblici, ciascuno fissato a un *commit*: cancellazione logica degli utenti (T1, *template* FastAPI), un riepilogo settimanale via e-mail (T2, Flask), collegare l'accesso Google a quello con password (T3, Django), ritentare le POST per impostazione predefinita (T4, la libreria HTTP ky), *deploy* senza interruzioni (T5, Django su Docker) e coautori per articolo (T6, Express con Prisma).
- **Condizioni.** Tre, ciascuna eseguita cinque volte per compito (90 esecuzioni) con claude -p su Claude Opus 5.5, in modalità piano (un'impostazione di Claude Code in cui l'agente può leggere i file ma non modificarli), da un clone nuovo e da un ambiente pulito, con lo stesso contratto di uscita: una tabella finale di al massimo dieci rischi.
    - **semplice:** "Plan this change." (pianifica questa modifica)
    - **rischi:** "Before planning, list the ways this change could go wrong." (prima di pianificare, elenca cosa può andare storto)

    - **ASP:** /asp --standard, fermato dopo che il piano aveva superato il suo unico passaggio di validatore.

- **Liste di riferimento.** Tre modelli di altri fornitori (Grok, GPT-OSS 120B, Gemini 3.8 Flash) hanno letto ogni repository e ogni richiesta senza vedere ASP e hanno scritto i rischi che si aspettavano. Un giudice cieco ha raggruppato le loro voci, e un rischio è entrato nella lista di riferimento quando almeno due revisori lo hanno nominato (da 8 a 12 per compito). I modelli hanno preso il posto di ingegneri indipendenti, cosa che il protocollo dichiara come un indebolimento.
- **Punteggio.** Il giudice principale (Gemini 3.8 Flash) ha visto solo le righe della tabella di ogni risposta, con le parole che rivelerebbero la condizione oscurate, e ha segnato ogni rischio di riferimento come coperto o no. Un secondo giudice (Grok) ha fatto lo stesso. I due hanno concordato sul 91,9% di 855 decisioni per voce (calcolato da benchmark/study-2026-11/judge/ e verificato in tests/test_study_2026_11.py).

Figura 3

Disegno dello studio 2026-11: i rischi di riferimento, scritti senza vedere ASP, e le risposte delle tre condizioni si incontrano solo presso il giudice cieco

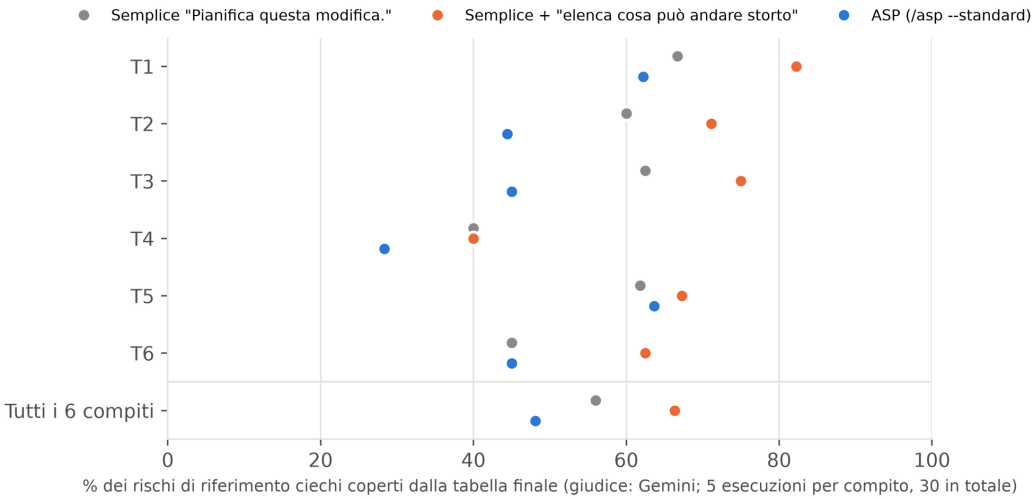


La Tabella 5 e la Figura 4 riportano la copertura per condizione.

Tabella 5
Copertura dei rischi e tempo per esecuzione nello studio 2026-11, per condizione

Condizione	Coperti (giudice principale)	Coperti (secondo giudice)	Tempo medio per esecuzione
semplice	56%	57%	58 s
rischi	66%	70%	50 s
ASP	48%	54%	175 s

Figura 4
Quota dei rischi di riferimento ciechi coperta dalla tabella finale di ogni condizione, per compito e in totale (giudice principale)



- **Primario.** ASP meno rischi ha dato −,18 (giudice principale) e −,16 (secondo giudice). È stato negativo su tutti e sei i compiti, con un test di permutazione unilaterale, stratificato per compito, p = 1,0 (quanto spesso rimescolare le etichette delle condizioni all'interno di ogni compito dà un divario altrettanto grande; il test chiede se ASP supera rischi). La regola di decisione fissata in anticipo dà *nessun beneficio misurabile*.

- **Secondario, preregistrato.** rischi meno semplice ha dato +,10 (p = ,001) e +,13 (p < ,001): chiedere, in una frase, cosa può andare storto ha aumentato la copertura.

- **Esplorativo.** Le risposte di ASP trattano alcuni rischi nelle ipotesi e nei deliverable anziché nella tabella finale. Quando il giudice principale ha letto la risposta intera, il divario si è ridotto a −,08 (66% contro 74%), e ASP è rimasto indietro su cinque dei sei compiti (verdetti in benchmark/study-2026-11/judge/gemini-full/, verificati in tests/test_study_2026_11.py).
- **Limiti.** Un modello di sessione (in una delle 90 esecuzioni l'*harness* ha chiamato anche un modello più piccolo dello stesso fornitore), sei compiti, liste di riferimento e giudici che sono tutti modelli, e la copertura come unico esito: non si è misurato se i rischi elencati siano reali né se portino a codice migliore.
- **Registro.** Ogni esecuzione, ogni verdetto e l'analisi sono nel repository, concatenati per hash, e run_all.py ricalcola ogni numero qui sopra.

6. Ipotesi di progetto

Evidenziamo cinque scelte di progetto che ritenevamo portanti (*load-bearing*). Le prime quattro sono ipotesi sul disegno di ASP, ciascuna enunciata con l'osservazione che l'ha motivata. Lo studio 2026-11 (Sezione 5.5) ha misurato un adattamento non interattivo delle Fasi da 0 a 6 nel loro insieme e non ha trovato benefici, il che pesa contro di esse prese insieme, senza verificarne nessuna da sola. A tutte e tre le condizioni è stato chiesto di dichiarare le proprie ipotesi anziché fare domande, il che sostituisce i giri di domande di ASP (Fasi 0a e 0b); l'uso interattivo non è stato valutato. La quinta si basa sulla misurazione dell'articolo complementare.

Ipotesi: una tassonomia forzata fa emergere più di un pre-mortem a forma libera. Nelle bozze dell'autore, il pre-mortem a forma libera si concentrava sulle categorie di rischio ovvie (dati, integrazione) e sottopesava quelle meno salienti (tempo/fuso orario, osservabilità, eccellenza

operativa, deriva dei contratti). Nessuna esecuzione ha messo alla prova il pre-mortem a forma libera come condizione (RED non aveva alcun *framework*), perciò ciò poggia sulla lettura delle bozze da parte dell'autore. Il *prompt* a forma libera ("immagina che il compito sia fallito tra 30 giorni; elenca oltre 10 cause") sarebbe allora necessario ma non sufficiente, e l'interrogazione MAST a 14 modi farebbe emergere categorie da cui il passaggio a forma libera si allontana.

Ipotesi: l'isolamento di *prompt* del validatore conta. Ci aspettiamo che i validatori che condividono il contesto con i pianificatori riproducano la fiducia del pianificatore sui contenuti condivisi. La letteratura convergente sui limiti dell'autocorrezione intrinseca, sul fallimento della localizzazione degli errori e sul self-enhancement bias dei giudici indica tutta questa direzione; non l'abbiamo messo alla prova dentro ASP.

Ipotesi: i passaggi basati su file reggono meglio alla pressione sul contesto di quelli basati sulla trascrizione. ASP scrive su disco *charter*, piano macro, registro dei deliverable e goal spec prima che l'esecuzione abbia inizio. Nella modalità --goal *opt-in* il figlio riceve su stdin (lo standard input) solo il testo della Goal Spec e restituisce un risultato JSON; il padre ricostruisce lo stato da quel risultato e dai file su disco, così nessuno dei due dipende da una trascrizione che potrebbe essere stata compressa o troncata. Non abbiamo confrontato i due tipi di passaggio.

Ipotesi: i vincoli enunciati (Iron Law) rendono una *skill* più difficile da saltare. Nella nostra esperienza, le *skill* che si leggono come "linee guida" vengono spesso saltate sotto pressione sul contesto. Le *skill* che emettono una *checklist* di punti non negoziabili, con *template* i cui campi vanno compilati prima della fase successiva, sono nel nostro uso più difficili da saltare. Il rispetto non è stato misurato.

Misurato altrove: trattare i codici di uscita di claude -p come segnali di successo semantico è una trappola silenziosa. Questo è il risultato riportato in Flores (2026b); lo menzioniamo qui perché l'implicazione di progetto (Iron Law 11 e 12) è stata rilevante per l'affidabilità del sistema.

7. Minacce alla validità

Trattiamo le minacce in modo esplicito nelle quattro categorie standard dei resoconti empirici di ingegneria del software.

Validità di costrutto. Nelle valutazioni delle Sezioni 5.1–5.2, lo stesso ricercatore che ha progettato il protocollo ha scritto gli elenchi degli shadow point attesi *e* ha giudicato a mano la copertura. Anche con una rubrica scritta, questa è la singola minaccia maggiore all'interpretabilità dell'osservazione. Il giudicatore deterministico che distribuiamo la mitigherebbe una volta applicato a trascrizioni reali e conservate; finora è stato eseguito solo su *fixture* costruite (§5.3), e non affronta la questione preliminare se gli elenchi attesi stessi siano sbilanciati verso ciò che ASP fa emergere. **La riscrittura in cieco da parte di revisori indipendenti è il seguito più forte.** Nello studio della Sezione 5.5 le liste di riferimento sono state scritte in cieco, da tre modelli di altri fornitori anziché da ingegneri, per sei compiti nuovi e non per questi cinque.

Validità interna. Con un'esecuzione per cella, GREEN su cinque compiti e RED su tre, le valutazioni delle Sezioni 5.1–5.2 al più suggeriscono che il protocollo possa cambiare le trascrizioni in questi casi; *non* sono sufficienti per stimare una dimensione dell'effetto, stabilire causalità o sostenere inferenza statistica, e per esse non è riportato alcun test statistico. Non è stata eseguita alcuna ablazione formale dei componenti (l'attribuzione qualitativa di ruolo portante alla *checklist* MAST nella §6 è un'ipotesi). La soglia di accettazione dell'80% è una scelta dell'autore e non è derivata dalla letteratura precedente.

Validità esterna. Per le valutazioni delle Sezioni 5.1–5.2: un solo modello sottostante (Claude Opus 4.7, presunto dalla sessione; non registrato per singola esecuzione), un solo ambiente (la workstation macOS dell'autore e la versione di Claude Code di maggio 2026, non registrata per singola esecuzione), cinque compiti ingegneristici tratti dall'esperienza di dominio dell'autore. Non rivendichiamo alcuna generalizzazione oltre queste condizioni.

Affidabilità. Tutti gli enunciati INPUT delle valutazioni, gli elenchi degli shadow point attesi, le *fixture* delle esecuzioni *mock* e il codice del giudicatore deterministico sono sottoposti a *commit* nel repository complementare; le trascrizioni delle esecuzioni del 2026-05-17 non sono state conservate, perciò le osservazioni della §5.2 non possono essere ricontrollate. Il pacchetto di *benchmark* costruisce per ogni esecuzione un *manifest* concatenato crittograficamente, così che qualsiasi byte modificato nei file che il *manifest* copre (ambiente, codice, input, output e giudizio) invalida l'hash del *manifest*; il file del *manifest* stesso e il report.md dell'esecuzione ne restano fuori.

Lo studio della Sezione 5.5. Esterna: un modello (Claude Opus 5.5) e sei compiti. Di costrutto: le liste di riferimento e i giudici sono modelli, al posto di ingegneri indipendenti, cosa che il protocollo dichiara come un indebolimento; la copertura è l'unico esito, quindi non si è misurato se i rischi elencati siano reali né se portino a codice migliore; e il contratto della tabella finale può non contare rischi che ASP tratta nelle ipotesi e nei deliverable (letta la risposta intera, il divario si è ridotto ma non si è chiuso). Interna: lo studio ha misurato le fasi di pianificazione del protocollo nel loro insieme, non i singoli componenti (Sezione 6). Ai revisori che hanno scritto le liste di riferimento è stato chiesto "the concrete ways this change could go wrong" (i modi concreti in cui questa modifica potrebbe andare storto), parole vicine a quelle della condizione rischi, il che può favorire quella condizione nella copertura. Il giudice principale ha anche scritto una delle tre liste

dei revisori, e 50 dei 57 rischi di riferimento portano la sua formulazione; il secondo giudice ha scritto un'altra lista, quindi nessuno dei due giudici è indipendente dal metro che applica.

8. Limiti e lavori futuri

I limiti di questo resoconto di esperienza corrispondono uno a uno alle minacce sopra. Il primo seguito indicato nelle versioni precedenti, uno studio confermativo del pre-mortem di una frase (la richiesta della condizione rischi, che mantiene la parte di elencazione di un pre-mortem e non chiede al modello di immaginare che la modifica sia già fallita), è ora riportato in Flores (2026c). Su dodici compiti fissati e due modelli di un solo fornitore, Claude Opus 5.5 e Claude Sonnet 5.5, la frase ha portato la copertura, rispetto a una semplice richiesta di pianificazione, dal 46,1% al 57,5% su 240 esecuzioni ($p < ,001$); tale affermazione è limitata a liste di riferimento scritte da modelli e a piani, non a codice. I seguiti prioritari che restano, in ordine:

1. **Ablazione formale**, ora che lo studio confermativo è concluso: prima di verificare quale componente del protocollo aiuta (*checklist* MAST, isolamento del validatore, *intake* della Fase 0a), verificare se qualcuno supera la frase unica.
2. **Un ingegnere umano su un campione delle liste di riferimento**, accanto alle liste scritte da modelli della Sezione 5.5 e di Flores (2026c), e, per lo studio della Sezione 5.5, una misura di precisione (i rischi elencati sono reali?) accanto alla copertura.
3. **Un secondo valutatore umano e una terza istantanea di claude -p** per il risultato complementare (Flores, 2026b, §8), questa sotto il Protocollo 1 di PREREGISTRATION.md.
4. **Valutazione cross-modello.** Rieseguire su Claude Sonnet, GPT-5, Gemini 2.5 e un modello a pesi aperti.

5. **Confronto diretto con PreFlect.** Eseguire ASP e una reimplementazione di PreFlect sugli stessi compiti.
6. **Giudicatore basato su *embedding*** come v0.2 del *benchmark*, con sentence-transformer fissato, per alzare il minimo di *recall* sulle parafrasi lessicalmente disgiunte.
7. **Replicazione su *benchmark* standardizzato.** I cinque compiti non sono un *benchmark* pubblico per agenti; replicare la direzione su SWE-bench (Jimenez et al., 2024) o su un'analoga suite standardizzata proverebbe se regge fuori dai compiti scelti dall'autore.

9. Conclusione

Abbiamo descritto ASP, una *skill* operativa di pre-mortem per agenti di programmazione LLM, integrando il risultato sulla *prospective hindsight* di Mitchell, Russo e Pennington (1989), la resa operativa del pre-mortem di Klein (2007) e la tassonomia dei fallimenti a 14 modi MAST di Cemri et al. (2025) come disciplina di runtime sotto un contratto di dodici Iron Law. La *skill* è *open source*, con tre percorsi di installazione. L'osservazione iniziale dell'autore, non in cieco, che i piani assistiti coprissero più modi di guasto non ha retto alla misura. In uno studio preregistrato su sei repository reali, le fasi di pianificazione del protocollo hanno coperto meno rischi dei revisori rispetto a Claude Code semplice invitato, in una frase, a elencare cosa può andare storto, e quella frase da sola ha superato una semplice richiesta di pianificazione. Da allora, uno studio confermativo ha verificato quel confronto come domanda principale (Flores, 2026c).

Abbiamo inoltre osservato che claude -p restituisce codice di uscita 0 anche quando l'agente rifiuta garbatamente l'obiettivo richiesto. Il fornitore documenta il comportamento per la sua API, come cataloga l'articolo complementare (Flores, 2026b, §2); quell'articolo aggiunge un conteggio al confine della CLI.

Con lo studio confermativo del pre-mortem di una frase ormai riportato (Flores, 2026c), il seguito più forte per ASP stesso è l'ablazione formale della Sezione 8. Il lavoro complementare più utile è un secondo valutatore umano e una terza istantanea preregistrata della misura su claude -p (Flores, 2026b).

10. Dichiarazioni

Conflitto di interessi. L'autore ha creato e mantiene ASP, il *plugin* che questo resoconto descrive, e dirige il laboratorio che lo pubblica. Lo studio preregistrato della Sezione 5.5 non ha trovato alcun beneficio misurabile del protocollo.

Finanziamento. Questo lavoro non ha ricevuto finanziamenti esterni. I costi d'uso dei modelli sono stati pagati dall'autore.

Contributo dell'autore. Autore unico: ideazione, disegno, raccolta dei dati, analisi, stesura e revisione.

Uso di strumenti di IA. I modelli linguistici sono l'oggetto di questo resoconto e hanno tre ruoli nello studio della Sezione 5.5: gli agenti provati, gli autori delle liste di riferimento e i giudici. Un assistente di IA è stato usato anche nella programmazione, nell'analisi e nella stesura e traduzione del testo, sotto la direzione dell'autore, come precisano i Ringraziamenti. L'autore ha verificato ogni numero sui dati e risponde del contenuto.

Ringraziamenti

Questo resoconto di esperienza è stato preparato in esplicita collaborazione con intervento umano (*human-in-the-loop*) con un assistente di IA, usato come partner di stesura, verifica e revisione. Le decisioni metodologiche, il disegno empirico e la finalizzazione sono stati svolti dall'autore umano. Il modello di collaborazione è documentato in docs/ARCHITECTURE.md, Parti 5–6, del repository complementare.

L'autore ringrazia i manutentori del *framework* di *skill* obra/superpowers, i contributori dello standard aperto agentskills.io e il team di Claude Code di Anthropic per la primitiva /goal il cui comportamento empirico ha motivato le Iron Law 11 e 12.

Riferimenti bibliografici

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

Appendice A — Stato della verifica delle citazioni

Ogni affermazione nel corpo del testo cita una fonte primaria ove ne esista una. I paragrafi che seguono registrano quando ciascun gruppo di riferimenti è stato controllato.

I riferimenti presenti nella versione 1.0 sono stati verificati con un recupero diretto dell'URL il 2026-05-17; i riferimenti aggiunti nella revisione di luglio 2026 (Flores, 2026b; le voci della genealogia classica della Sezione 2.5) sono stati controllati al momento dell'aggiunta; e le voci aggiunte a novembre 2026 (Barke et al., 2026; Deshpande et al., 2025; G. Zhang et al., 2025; J. Zhang et al., 2026) sono state controllate rispetto ai loro record arXiv, e Cemri et al. (2025) rispetto al suo record negli atti di NeurIPS. Nello stesso mese Kambhampati et al. (2024) e Shinn et al. (2023) sono stati controllati rispetto ai loro record negli atti (PMLR e NeurIPS), e le loro voci ora citano quei record. Flores (2026c), anch'esso aggiunto a novembre 2026, è stato controllato rispetto al suo manoscritto nel repository complementare (paper/one-sentence-premortem.md). I nomi dei 14 modi MAST sono citati alla lettera da Cemri et al. (2025) ed elencati nel file della *skill* complementare plugins/anticipating-shadow-points/skills/anticipating-shadow-points/mast-checklist.md. Mitchell et al. (1989) è lo studio primario sulla prospective hindsight; la cifra del ≈30% è il riassunto che ne dà Klein (2007), che non siamo riusciti a ricalcolare dall'articolo del 1989, e Klein (2007) è anche la ricetta operativa.

I nove riferimenti introdotti nella §2.5 (Avizienis et al., 2004; International Electrotechnical Commission [IEC 60812:2018], 2018; International Electrotechnical Commission [IEC 61882:2016], 2016, controllato rispetto al catalogo IEC il 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) sono stati verificati rispetto alla loro fonte primaria — DOI, ISBN o voce di catalogo dello standard — il 2026-07-08.

Appendice B — Insieme di valutazioni e riproducibilità

Le cinque valutazioni ingegneristiche sono sottoposte a *commit* nel repository complementare, nella cartella 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)

Ogni file contiene l'enunciato INPUT, l'elenco EXPECTED degli shadow point e la soglia di ACCEPTANCE, tutti scritti dal progettista del protocollo. Il pacchetto di *benchmark* in benchmark/ fornisce:

- un giudicatore deterministico a 4 livelli (esatto / sinonimo / Jaccard / non corrispondente);
- una provenienza crittografica per ogni esecuzione (ambiente, codice, input, output e giudizio sottoposti ad hash in una catena in stile Merkle);
- un sottocomando verify che riesamina una directory di esecuzione e segnala ogni deriva a livello di byte nei file che il *manifest* copre;
- 31 test unitari che coprono il giudicatore, i parser delle valutazioni e delle risposte, il *manifest* e l'hasher, incluso il contratto di determinismo.

Riprodurre i numeri calcolati delle Sezioni 5.3–5.5 (nessuna chiamata a modelli). La Tabella B1 elenca ciò che il repository contiene per essi.

Tabella B1
Ciò che il repository contiene per le Sezioni da 5.3 a 5.5

Materiale	**Dove nel repository**	**Che cosa può verificare il lettore**
Le cinque valutazioni	plugins/anticipating-shadow-points/skills/ anticipating-shadow-points/evals/	Ogni compito, il suo elenco atteso e la sua regola di accettazione (Tabella 4)
I piani di esempio dell'autore (RED e GREEN) e l'esecuzione che li ha valutati	benchmark/tests/fixtures/mock_runs/ e benchmark/runs/published/	I piani che il *matcher* ha valutato e i punteggi alla base della Figura 2
L'esecuzione di luglio 2026 di claude -p della nota complementare	data/spike-2026-07-08/	I conteggi citati nella Sezione 5.4
Lo studio 2026-11	benchmark/study-2026-11/	Protocollo, compiti, liste di riferimento, 90 esecuzioni, verdetti di entrambi i giudici e registro concatenato di hash, alla base della Sezione 5.5 e della Figura 4
Il calcolo	analysis/results.py e benchmark/ study-2026-11/analyze.py	Come si calcola ogni numero delle Sezioni da 5.3 a 5.5
I controlli	tests/test_paper_numbers.py e tests/test_ study_2026_11.py	Che ciascuno di questi numeri corrisponde ai dati, e che l'esecuzione valutata è etichettata come mock
Il sigillo	output/hash-chain.md	Una sola impronta SHA-256 su codice, dati e risultati, così che ogni modifica successiva si veda

Un'impronta SHA-256 è un codice di 64 caratteri calcolato a partire da un file; cambiare un solo byte del file cambia il codice. Un unico comando, python run_all.py, eseguito dentro la cartella del repository, rifà tutto ciò che è nella Tabella B1 e non chiama nessun modello; richiede Python 3.11 o successivo, e il README del repository elenca i quattro comandi che scaricano il repository, installano ciò che serve e lo eseguono. Le osservazioni dell'autore nella Sezione 5.2 non possono essere riprodotte: gli output del modello da cui furono lette non sono stati conservati.

Raccogliere evidenza reale (costa chiamate a modelli). Installare il pacchetto di *benchmark* ed eseguire asp-benchmark run --mode real --model <model> seguito da asp-benchmark verify <run-id>. Il *runner* di questo pacchetto non è stato usato per alcun numero pubblicato. Lo studio 2026-11 della Sezione 5.5 ha *runner*, giudice e registro concatenato di hash propri in benchmark/study-2026-11/, e python run_all.py ricalcola i numeri di quello studio dai verdetti versionati, senza chiamare un modello. Il protocollo è specificato indipendentemente dall'implementazione Python in benchmark/PROTOCOL.md, così da poter essere reimplementato in un altro linguaggio.

Nota Tecnica

Versione 2.4 · novembre 2026 · prima pubblicazione il 18 maggio 2026

Serie Codex Hash Research Report 2026-01

Licenza Testo del paper, figure e dati: CC BY 4.0. Software complementare: MIT.

Il progetto ASP. ASP (Anticipating Shadow Points) è un *plugin open source* per Claude Code, l'agente di programmazione da riga di comando di Anthropic, che fa elencare all'agente ciò che potrebbe andare storto prima di scrivere codice. Il *plugin* e il materiale di replicazione stanno in un repository pubblico (Flores, 2026a), questo resoconto di esperienza descrive come funziona e una nota tecnica misura da sola una delle sue regole: non fidarsi mai del codice di uscita di claude -p (Flores, 2026b). La Sezione 5.5 riporta la prima misurazione di se ASP migliora i piani di un agente; un terzo articolo riporta lo studio confermativo del pre-mortem di una frase a cui essa ha portato (Flores, 2026c).

DOI. Un DOI (Digital Object Identifier) è un collegamento permanente a un'opera depositata. Zenodo, un archivio pubblico di ricerca gestito dal CERN, conserva questo resoconto di esperienza come record a sé: https://doi.org/10.5281/zenodo.24065968, che apre sempre la versione più recente; questa versione, la 2.4, è https://doi.org/10.5281/zenodo.24410420. Il *plugin*, i dati e il codice formano un altro record (Flores, 2026a), il cui DOI concept, https://doi.org/10.5281/zenodo.20276631, apre sempre la *release* più recente; la *release* 2.0.0 è quella che corrisponde a questo resoconto di esperienza. Le *release* di maggio 2026 di quel record (da 1.0.0 a 1.0.3) presentavano i punteggi dati dal *matcher* ai piani di esempio dell'autore come lettura di output reali del modello; la *release* 2.0.0 le sostituisce.

Correzioni. La versione 1.0 di questo resoconto presentava inoltre le cinque valutazioni come misurate e confondeva tre cose che la Sezione 5 ora separa: ciò che l'autore ha osservato in esecuzioni reali, ciò che il giudicatore mostra di sé su *fixture* costruite e ciò che non è stato misurato. La versione 2.0 ha corretto questo punto, ha aggiunto sopra i riepiloghi di maggio sottoposti a *commit* la nota che ritira la loro conclusione che ASP sia "riproducibilmente migliore" (Sezione 5.2) e ha corretto il difetto della finestra scorrevole della Sezione 5.3.

Traduzioni e sottomissione. Le traduzioni della documentazione sono descritte nella Sezione 4; i piani di sottomissione dipendono dai seguiti della Sezione 8.

Replicazione. Ogni numero che le Sezioni da 5.3 a 5.5 calcolano può essere ricalcolato dal repository senza chiamare un modello; l'Appendice B, Insieme di valutazioni e riproducibilità, elenca cosa contiene e come eseguirlo.

Sull'autore

Carlos Ulisses Flores è CTO e Ricercatore Capo presso il Codex Hash Research Laboratory, un laboratorio indipendente di R&S a San Paolo, in Brasile. Costruisce sistemi di produzione in cui il fallimento silenzioso è costoso — storicamente nel trading algoritmico, nell'IoT embedded e nella crittografia edge, e oggi negli strumenti di IA agentica. La *skill* ASP descritta in questo resoconto di esperienza è nata da quest'ultimo filone di lavoro; il risultato empirico riportato nella nota tecnica complementare è nato dall'uso del *runner* claude -p di Anthropic come nucleo di esecuzione autonoma e dall'averlo visto restituire codice di uscita 0 davanti a un rifiuto garbato dell'agente.

Scrive in portoghese, inglese, spagnolo e italiano.

Identificatori · ORCID: 0000-0002-6034-7765 · Lattes (registro nazionale brasiliano dei curricula dei ricercatori): 6905246706890561

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