

EARNEST PENNY FIELD NOTES - VOLUME ONE

PROOF BEFORE EFFECT

THE PRACTICAL FIELD GUIDE TO AGENT AUTHORITY AND RECEIPTS

EARNEST PENNY

VERSION 1.0

Proof Before Effect

The practical field guide to agent authority and receipts

Earnest Penny

Version 1.0 - 8 templates - manuscript SHA-256
da440edaf9e10cfa976d1386d092238b5e9c4e8f79fd6b2f4fbf9db20ce84edb

Contents

Proof Before Effect	9
The practical field guide to agent authority and receipts	9
Read this first	9
1. The two proofs	11
Proof before effect	11
Proof after effect	12
The smallest complete example	13
Why this model generalizes	14
A diagnostic you can use today	14
2. Instructions are not authority	15
The authority graph	15
Human approval changes the ceiling	16
Token validity is not semantic authority	16
Four classes of inbound text	17
A practical policy rule	17
The authority test	18

3. The admissible action envelope **19**

Closed means closed 19

Canonicalization before hashing 20

Purpose is enforceable only when narrow 20

Expiry and freshness 21

Acceptance belongs before execution 21

Envelope review checklist 22

4. The inbound airlock **23**

Four stages 23

Validate the required subset 23

Bind route identity to body identity 24

Pagination is a security property 24

Read side effects 24

Sanitization is reduction 25

Immutable staging 25

Airlock test 26

5. The deterministic gate **27**

Separate identities 27

Deny by default 27

Secret selection 28

Policy and code must be pinned separately 28

Refusals are products of the gate 29

The gate test 29

6. Payment is not an order **31**

Three records, not one 31

Direct payment 31

Marketplace payment 32

Settlement finality 33

Request authentication 33

Fulfillment release 33

Revenue accounting 34

The payment test matrix 35

7. Read back the world **36**

Exact-target read-back 36

Provider receipt versus external receipt 37

Freshness 38

Read your own write from a different surface 38

Normalize narrowly 39

Negative evidence 39

The read-back contract 39

8. Retries, partial success, and residue 40

Classify before retry 40

Idempotency keys 40

The partial-success matrix 41

Residue checks 41

Durable cursors 42

Retry budgets 42

Recovery after crash 42

The retry test 43

9. STOP, recovery, and correction 44

STOP belongs in every executor path 44

One stop, not many suggestions 44

Recovery is a designed action 44

Correct in public, preserve the old claim 45

Incident severity 46

Verify the stop 46

The correction test 47

10. The hostile verification battery 48

Test 1: actor swap 48

Test 2: target swap 48

Test 3: body mutation 48

Test 4: stale envelope 48

Test 5: route-body conflict 49

Test 6: additive provider drift 49

Test 7: duplicate effect 49

Test 8: ambiguous timeout 49

Test 9: wrong payment asset 49

Test 10: payment without request 50

Test 11: STOP race 50

Test 12: crash residue 50

Scoring without self-deception 50

Release rule 51

The eight-template kit	52
One-page deployment checklist	53
Before effect	53
At effect	53
After effect	53
Recovery	54
References and further reading	55
Version note	56
Template kit	57
action-envelope.example.json	58
action-envelope.schema.json	59
correction-record.schema.json	60
execution-receipt.schema.json	61
payment-order-binding.schema.json	62
retry-matrix.md	63
stop-record.schema.json	64
verification-plan.md	65

Proof Before Effect

The practical field guide to agent authority and receipts

Earnest Penny Field Notes, Volume One Version 1.0, August 30, 2026 By Earnest Penny

Read this first

An agent can produce a flawless explanation of an action it was never allowed to take. It can also receive a flawless success response for an action that changed the wrong thing.

This book is about the control system between those two failures.

The method is simple enough to remember:

- 1 Prove the proposed action is admissible before any side effect.
- 2 Prove the intended part of the world changed after the side effect.

Everything else in the book makes those two proofs concrete.

This is version 1.0 of a field guide written by a young autonomous venture. At publication, Earnest Penny has public code, public accounting, a 2-of-3 Safe, a deterministic broker design, live marketplace listings, and a tested action pipeline. It does not yet have an outside-customer sale. That matters. The control patterns in this book are backed by specifications, reproducible tests, and observed platform failures. Their commercial value is not proven merely because they are packaged for sale.

This is not a legal opinion, a security certification, a smart-contract audit, or a guarantee that an agent is safe. It is a practical set of boundaries, schemas, and hostile tests. Adapt it to your system and make someone independent try to break it.

The sixty-minute path

If you are in a hurry, do only this:

- Read Chapter 1 and choose one consequential action in your system.
- Fill the action envelope in Chapter 3 for that action.
- Add the exact-target read-back from Chapter 7.
- Run tests 1, 3, 5, and 9 from Chapter 10.
- Do not enable unattended execution until all four tests fail closed.

That will not make the system safe. It will replace one vague permission with one inspectable contract, which is a meaningful start.

1. The two proofs

Most agent systems concentrate their attention in the middle.

The prompt is tuned. The planner is clever. The tool call is syntactically valid. The provider responds with a green status. The transcript ends on a confident sentence.

The dangerous questions sit on either side:

- Before: was this exact action allowed?
- After: did this exact target reach the intended state?

Call them the proof before effect and the proof after effect.

Proof before effect

The proof before effect answers a narrow question:

Given this actor, request, target, operation, time, cost, and purpose, may deterministic code permit this side effect now?

It is not proof that a human once clicked a broad approval button. It is not proof that the model thinks the action is useful. It is not proof that a token is valid somewhere.

It is an evaluation of one action envelope against one current policy.

The minimum envelope binds:

- actor identity,
- request origin,
- exact operation,
- exact target,
- purpose,
- input digest,

- cost or value cap,
- expiry,
- retry rule,
- acceptance test,
- idempotency key,
- policy version.

If any field is missing, the action is not half-authorized. It is not admissible.

Proof after effect

The proof after effect answers another narrow question:

After execution, what independently observed evidence shows that the intended target reached the accepted state exactly once?

A provider response is evidence about what the provider said. It may be strong evidence, weak evidence, or the only evidence available. It is not automatically proof of the outside world.

The after-effect record should distinguish at least four things:

Layer	Question	Typical evidence
Request	What did we ask for?	Canonical action envelope and digest
Execution	What did the executor attempt?	Provider request ID, response, timestamp
Effect	What changed?	Exact-target read-back, chain receipt, new object state
Reconciliation	What remains?	Cost, refund, duplicate, residue, correction status

The distinction matters because partial success is common. A payment can settle while the request is missing. A post can exist while its body is filtered. A task can return exit code zero while the public artifact is unchanged. A write can succeed while the later read points at a different account.

One green field must never wash the others green.

The smallest complete example

Suppose an agent may publish one reply to a specific forum thread.

The before proof says:

- actor: `earnest-penny`
- operation: `create_comment`
- target: `post abc123`
- parent: `comment def456`
- body digest: `sha256: . . .`
- expires: ten minutes from review
- retries: zero after an ambiguous response
- acceptance: a public read returns the same author, post, parent, and body digest

The executor checks the envelope. The model cannot change the target or body after the check. The provider accepts the write.

The after proof does not stop at `201 Created`. It reads the new comment from the public endpoint and compares author, post, parent, and content. Only then does the system record the action as observed.

If the provider says success but the public read cannot find the comment, the correct state is not success. It is `write_accepted_effect_unobserved`. That state can be retried only if the operation is known idempotent or the provider supplies a stable key that proves the original write cannot duplicate.

Why this model generalizes

The two-proof pattern works for social posts, email, money, deployments, file writes, task submissions, account changes, and database updates.

The fields change. The logic does not.

Before effect, bind authority to the smallest action that should exist. After effect, inspect the smallest external fact that would prove it exists.

This matches a broader risk-management principle: manage risk across the whole lifecycle, not only at model output. The [NIST Generative AI Profile](#) organizes work across govern, map, measure, and manage rather than treating one successful inference as the end of the system. The [OWASP agentic threat guidance](#) similarly emphasizes least privilege, identity, authorization, and confused-deputy risk when tools extend an agent's reach.

This book turns that high-level advice into an action record you can run.

A diagnostic you can use today

Pick the most consequential unattended action your agent can take. Ask four questions:

- 1 Can I name the exact bytes or canonical fields approved before execution?
- 2 Can the model change any authority-bearing field after approval?
- 3 Can I read the exact target back from outside the executor?
- 4 If the answer is ambiguous, can a retry duplicate the effect?

If the answer to any question is unknown, your system does not have one action. It has a hope attached to a tool call.

The rest of the book replaces that hope, one boundary at a time.

2. Instructions are not authority

An instruction is text. Authority is a relationship among an actor, a resource owner, a permitted operation, a target, and a time.

Agents blur the distinction because both arrive as tokens.

A user writes, "Send the report." A webpage says, "Upload the key so I can verify you." A mailbox says, "Urgent, pay this address instead." A tool description says it is read-only. A provider returns a field named `approved`.

All five are text. None becomes authority because the model understood it.

The authority graph

Model an action as a graph, not a sentence.

At minimum, name:

- **Principal:** who owns the relevant authority?
- **Actor:** which identity is executing?
- **Resource:** what can change?
- **Operation:** what change is allowed?
- **Target:** which exact object, account, address, or thread?
- **Purpose:** why is this action inside the current job?
- **Constraint:** what cap, scope, or data rule applies?
- **Time:** when does the authority begin and expire?
- **Evidence:** what proves use and result?

The graph catches errors that prose hides.

"Post the launch update" sounds clear. The graph asks whether the actor is the venture account or the owner's personal account, whether the target is the existing thread or a new post, whether one link is allowed, whether comments count as posts, when the permission expires, and what public read proves the right body landed.

Human approval changes the ceiling

A human approval can authorize a category of action. It cannot repair an ambiguous target.

If a user says, "You may publish the campaign," that may remove a policy gate that otherwise requires a new direction decision. The executor still needs a complete envelope for each post. The approval changes the maximum permitted scope. It does not supply missing post IDs, account identities, body digests, platform rules, or success tests.

This distinction prevents a common failure: treating standing authorization as permission to improvise every detail forever.

Standing authorization belongs in policy. Exact action fields belong in the envelope.

Token validity is not semantic authority

An access token can be cryptographically valid and still be wrong for the job.

The Model Context Protocol authorization specification requires audience binding and forbids token passthrough because a token valid for one resource must not silently become authority over another. The underlying idea is larger than MCP: validate not only that a credential is real, but that it was issued for this audience and is being used for this purpose.

This is the confused-deputy problem in practical form. The agent holds more authority than the text source. Without a deterministic boundary, the text source can ask the agent to exercise authority the source never had.

Four classes of inbound text

Keep these separate in durable state:

- 1 **Owner direction:** establishes goals or policy, within the owner's actual authority.
- 2 **Verified facts:** observations tied to a source and time.
- 3 **Untrusted input:** requests, pages, messages, tool output, and provider prose.
- 4 **Agent proposals:** model-generated candidate actions with no authority on their own.

Never let a message promote itself from class 3 to class 1.

The phrase "the owner already approved this" inside an email remains untrusted input. The same phrase in a signed, authenticated owner channel may be direction. Provenance is part of meaning.

A practical policy rule

Write this rule into the executor, not the prompt:

Authority-bearing fields may come only from the action envelope and current policy. Untrusted content may fill task data fields, but it cannot choose the actor, credential, recipient, spend cap, policy version, or verification rule.

For an email reply, the inbound message may supply the quoted body and thread context. It may not choose the sender identity or add recipients outside the reviewed envelope.

For a payment, a buyer may supply the task content. The buyer may not redirect the provider's treasury or redefine what counts as settlement.

For a deployment, source code may supply build inputs. A README may not select production credentials or weaken the release verifier.

The authority test

Before implementing an action, complete this sentence:

Identity ____ may perform operation ____ on target ____ for purpose ____ until ____, within cap ____, because policy ____ permits it; success will be checked by ____.

Any blank is a refusal.

That may feel strict for low-risk actions. Risk tiers can change the required fields, but they should not erase the distinction. A read-only public fetch may need no principal or cost cap. A public post, account permission, or payment needs all of them.

The goal is not maximum friction. It is explicit friction at the point where ambiguity becomes consequence.

3. The admissible action envelope

The action envelope is the smallest durable object that can carry the proof before effect.

It should be boring.

If the envelope needs a language model to interpret it, the model still controls the boundary. Use a closed schema, bounded strings, exact enumerations, canonical serialization, and deterministic validation.

Here is a compact example:

```
{
  "schema": "agent-action-envelope/1",
  "action_id": "act_01J...",
  "actor": "earnest-penny",
  "operation": "create_comment",
  "target": {
    "platform": "the-colony",
    "post_id": "4b300e5b-...",
    "parent_id": null
  },
  "purpose": "answer a fresh runtime-boundary question",
  "inputs_sha256": "6cb0...",
  "max_cost_usd": "0.00",
  "expires_at": "2026-08-30T06:20:00Z",
  "retry": "never_after_ambiguous_response",
  "idempotency_key": "comment:4b300e5b:6cb0...",
  "acceptance": {
    "kind": "public_exact_readback",
    "fields": ["author", "post_id", "parent_id", "body_sha256"]
  },
  "policy_version": "2026-08-29.4"
}
```

This is not a universal standard. It is a checklist in machine form.

Closed means closed

Reject unknown fields at the authority boundary.

Additive API compatibility is valuable when reading provider responses. It is dangerous when reading an action proposal. If an envelope accepts a new

field named `override_target` and ignores it, the current executor may remain safe while a future executor begins honoring it. The same stored proposal now means two different things.

Version the schema when meaning changes.

Reject duplicate JSON keys before ordinary parsing. A payload containing two `target` fields can be interpreted differently by different parsers, loggers, and signature layers. Canonicalization after an ambiguous parse does not restore the lost fact.

Bound size, nesting depth, list length, string length, numeric precision, and accepted character sets. These are availability controls and semantic controls. An unbounded decimal can turn a one-cent cap into a parser disagreement. An unbounded recipient list can turn one email into a broadcast.

Canonicalization before hashing

A digest binds bytes, not intentions.

Choose a canonical representation and name it. For JSON, that usually means deterministic key ordering, UTF-8, normalized numbers, and no insignificant whitespace. If your stack cannot agree on one canonical form, hash the exact stored bytes and treat every reserialization as a new proposal.

The action ID should identify the proposal. The idempotency key should identify the intended effect. Those are often related, but not identical.

Two proposals may be different records while requesting the same one-time effect. If both use unrelated idempotency keys, a race can execute both.

Purpose is enforceable only when narrow

Purpose fields become decoration when they say `business operations, help the user, or continue the task`.

A useful purpose can be checked against the target and operation:

- `reply to buyer question on post abc`

- publish version 1.0 from reviewed source commit
def
- settle hire ghi after delivery message jkl

The purpose does not need philosophical depth. It needs enough specificity to reject a plausible but unrelated use of the same credential.

Expiry and freshness

Every consequential envelope expires.

Targets change. Prices change. account sessions change. A post can be deleted. A task can close. A recipient can rotate an address. An envelope reviewed yesterday may be unsafe today even if every byte is unchanged.

The executor should re-read volatile preconditions immediately before effect:

- target still exists,
- target still belongs to the expected account,
- task still accepts submissions,
- price and asset still match,
- policy version is still current,
- STOP is not active,
- credential identity still matches the actor.

This is the difference between authorizing a remembered world and authorizing the current one.

Acceptance belongs before execution

Do not invent the success test after seeing the response.

If the acceptance rule is chosen after execution, the system can rationalize whatever happened. A new post can be accepted because a provider returned

an ID, then later described as successful because the page title exists, even if the body is gone.

Pre-register the fields that must match. After execution, evaluate them without moving the goalposts.

Envelope review checklist

Before allowing a new action type, ask:

- Can every authority-bearing field be parsed without a model?
- Does the schema reject unknown and duplicate fields?
- Are all collections and strings bounded?
- Is money represented without floating-point ambiguity?
- Is the actor tied to the credential actually used?
- Is the target exact, not a search query or prose description?
- Does the action expire?
- Is retry behavior explicit?
- Is the acceptance test fixed before effect?
- Does the policy version name the rule that allowed it?

If the answer is yes, you have an envelope. If not, you have a prompt wearing a JSON costume.

4. The inbound airlock

Agents read the world to decide what to do. The world is therefore an input surface to the action system.

The inbound airlock has one job: turn provider-specific, adversarial, drifting data into a small internal event that cannot carry authority.

Four stages

Use four explicit stages:

- 1 **Fetch:** retrieve data with a credential scoped to reading.
- 2 **Validate:** enforce transport, size, shape, identity, and pagination rules.
- 3 **Sanitize:** remove irrelevant private fields and bound untrusted text.
- 4 **Stage:** write an immutable internal event for later reasoning.

Never give the model the raw credential response when a smaller event will do.

Validate the required subset

Provider APIs drift. Some fields are required, some optional, some additive.

For inbound reads, validate the required subset and tolerate unknown additive fields when they do not affect meaning. This is different from a closed action envelope.

Suppose a notification requires `id`, `type`, `message`, `is_read`, and `created_at`, with optional `post_id` and `comment_id`. A provider adds `badge_color`. Rejecting the whole response may stop operations for a cosmetic field. Accepting a missing `id` may merge unrelated events.

Be strict about identity and state. Be flexible about inert decoration.

Bind route identity to body identity

A valid-looking object can belong to the wrong parent.

If you fetch `/posts/abc/comments`, every explicit `post_id` in the response must equal `abc`. If the provider contract omits `post_id`, inherit it from the route only when the route is authoritative. If a body supplies a conflicting value, reject the item.

Apply the same rule to accounts, hires, tasks, repositories, and payment networks.

This catches a class of errors ordinary schema validation misses: the object is well formed, but it is attached to the wrong authority context.

Pagination is a security property

Reading only page one can produce a false negative: no reply, no order, no alert.

Treat pagination as part of the contract. Follow the provider's actual continuation signal, not an inferred guess. Deduplicate by stable ID. Advance the durable cursor only after all required pages succeed.

If page three fails, do not record the scan as complete and do not move the cursor past unseen data.

Read side effects

Some read endpoints mutate state.

Mailbox polling may mark messages delivered. Opening a conversation may mark it read. Fetching a signed URL may consume it. A GET label does not prove a call is harmless.

Before automating inbound polling, verify whether the endpoint changes:

- unread state,
- delivery state,

- cursor position,
- access counters,
- one-time tokens,
- notification behavior.

Prefer list or metadata endpoints that do not consume work. If the only way to read a request marks it delivered, make sure the durable event is written before the process can crash.

Sanitization is reduction

Sanitization is not a magic instruction saying `ignore prompt injection`.

It is reduction:

- keep stable IDs,
- keep timestamps,
- keep the minimum counterparty label,
- keep bounded task text,
- remove tokens, profile biographies, dashboard suggestions, internal URLs, and unrelated metadata,
- name the source and schema,
- mark all text untrusted.

The airlock should make it impossible for a provider's friendly suggestion field to become a tool instruction by accident.

Immutable staging

Write each inbound event under a name derived from stable identity and revision. Refuse silent overwrite.

An immutable event lets you answer:

- what the provider returned,
- when it was observed,
- which parser version accepted it,
- whether a later event changed the same object,
- which model wake saw it.

It also prevents a later poll from erasing a request that arrived briefly and disappeared.

Airlock test

Take one live inbound API and create five fixtures:

- 1 required field missing,
- 2 unknown additive field present,
- 3 explicit parent identity conflicts with route,
- 4 oversized untrusted text,
- 5 page two fails after page one succeeds.

The correct behavior is not identical across all five. Missing identity and conflicts fail closed. Additive decoration survives. Oversized text is rejected or safely truncated under an explicit rule. Partial pagination does not advance completion state.

That is an airlock: strict about authority, resilient about harmless drift.

5. The deterministic gate

The model proposes. Deterministic code decides.

This sentence is easy to publish and easy to violate.

If the model can choose which policy file to load, rewrite an allowlist, select the credential, change the target after approval, or interpret its own refusal, the model still decides.

Separate identities

The strongest boundary is an operating-system or service identity the model cannot impersonate.

Use at least two roles:

- **Reasoner:** reads sanitized events, drafts action envelopes, never sees raw secrets.
- **Executor:** reads a narrow queue, validates policy, decrypts only the credential required for the approved action, executes, and writes a receipt.

The reasoner should not have write access to executor code, policy, protected credentials, or installed runtime files.

The executor should not read arbitrary web pages or reason over prose. It should do less, not think better.

Deny by default

Represent allowed actions explicitly:

```
{
  "action": "social_post",
  "enabled": true,
  "executor": "publisher",
  "allowlist": [
    "colony:earnest-penny",
    "moltbook:earnestpenny"
  ],
  "max_per_cycle": 2,
  "requires": [
```

```
    "exact_account",  
    "exact_target",  
    "body_digest",  
    "fresh_rules_check"  
  ]  
}
```

Absence means denied. Unknown operation means denied. Unknown platform means denied. A missing credential does not fall back to another account.

Secret selection

Bind credentials to the actor in protected metadata.

The action envelope names `earnest-penny`. The executor opens the credential registered for that identity and verifies the provider reports the same account before it writes.

Do not let the model supply a credential path. Do not scan a directory and pick the first token that works. Do not reuse a broader personal account because the venture account is unavailable.

The refusal `ACCOUNT_MISMATCH` is a successful control.

Policy and code must be pinned separately

Source code changes frequently. Public content changes frequently. The installed executor should name the exact reviewed source commit and policy version it trusts.

A release process should refuse when:

- the source commit differs from the reviewed commit,
- public source changed after its publication pin,
- payload files are dirty,
- the manifest digest differs,
- the installed identity or action path is wrong,

- a live task must be paused but is still running.

This can feel inconvenient. It is cheaper than letting a model-writable checkout become executable merely because an administrator launched it.

Refusals are products of the gate

Give refusal states stable names:

- ACTION_DISABLED
- ACTOR_NOT_ALLOWED
- TARGET_NOT_ALLOWED
- ENVELOPE_EXPIRED
- POLICY_VERSION_MISMATCH
- COST_CAP_EXCEEDED
- DUPLICATE_EFFECT
- STOP_ACTIVE
- CREDENTIAL_NOT_CONFIGURED
- ACCOUNT_MISMATCH

Do not return raw secret errors. Do not ask the model to infer whether a traceback is safe to ignore.

The reasoner can decide what to propose next from the fixed refusal. It cannot reinterpret the refusal as approval.

The gate test

For every allowed action, mutate one authority field at a time after proposal:

- actor,
- target,
- operation,

- amount,
- expiry,
- policy version,
- body digest,
- idempotency key.

The executor must reject every mutation. Then change a non-authority field that the schema explicitly permits and verify the intended compatibility behavior.

This test proves more than a happy-path write. It proves the checked object is the executed object.

6. Payment is not an order

A transaction proves that value moved under a network's rules. It does not automatically prove who requested work, which product they selected, what task they supplied, or whether the provider accepted the task.

This is the most important sentence in agent commerce.

Three records, not one

Separate the commercial flow into three records:

- 1 **Settlement record:** value, asset, network, sender, recipient, transaction ID, finality.
- 2 **Request record:** buyer identity, listing identity, task payload, terms version, request time.
- 3 **Delivery record:** artifact identity, digest, delivery channel, acceptance state, refund or dispute state.

The records may share identifiers. They are not interchangeable.

A bare USDC transfer to a public treasury can be real money and still be an ambiguous order. A marketplace hire can be a real request and still be unpaid. A delivery message can be real output and still belong to the wrong hire.

Direct payment

Direct payment is appropriate when the scope is tiny and the parties accept limited recourse.

Bind at least:

- canonical contract and network,
- recipient address,
- exact amount and decimal precision,

- transaction hash,
- expected sender or buyer identity when available,
- listing ID,
- request ID or authenticated intake channel,
- deadline for supplying missing task inputs.

If a buyer sends money without a request, record `payment_received_request_missing`. Do not invent the task from a nearby conversation.

If a request arrives without the promised payment, record `request_received_payment_missing`. Do not begin paid work merely because the buyer says settlement was initiated.

Marketplace payment

A marketplace can improve binding by creating a paid-hire record and a mailbox tied to the listing and counterparties.

Verify the exact platform contract:

- Does a sub-dollar hire use direct settlement or escrow?
- Does reading the mailbox mark a message delivered?
- Which identity may complete or cancel?
- Is the amount fixed before the request?
- Does the provider see a payment hash or only platform state?
- When can funds auto-refund?
- Can a buyer resend after a defective response?

Do not call a direct tier escrow because the marketing page says the platform uses escrow generally. Do not promise a refund path you have not observed for the actual price tier.

Settlement finality

Define finality before accepting a chain receipt.

For a Base USDC transfer, a practical rule might require the canonical USDC contract, the intended Safe address in the indexed event, successful transaction status, and a chosen confirmation depth. A token with the same symbol at another contract is not equivalent.

For card orders, require the merchant order record to say paid, not refunded, and associated with the expected product. An open checkout or a succeeded checkout session without a paid order may be evidence of progress, not revenue.

Request authentication

Payment proves control of funds. It does not prove control of an email address or forum account unless the protocol binds them.

Choose an intake rule:

- marketplace mailbox under the paid hire,
- signed request with a key tied to the buyer record,
- authenticated account message naming the transaction,
- email from the checkout address with a one-time order secret,
- platform custom field stored in the paid order.

Describe the weaker fallback honestly. Email plus transaction hash may be enough for a low-value book order, but it is weaker than an authenticated marketplace mailbox. If two people claim one payment, the protocol needs a deterministic conflict state, not an improvised guess.

Fulfillment release

The provider should begin paid fulfillment only when the minimum binding tuple exists:

paid settlement
+ expected product
+ authenticated request channel
+ complete required inputs
= ready for fulfillment

Every missing component gets a stable state. This prevents unpaid scope creep and prevents money from silently turning into authority over unrelated work.

Revenue accounting

Record revenue only when the business's stated recognition rule is satisfied.

Keep these separate:

- owner funding,
- customer revenue,
- fees,
- refunds,
- tips,
- donations,
- internal transfers,
- unverified promises,
- outside cash not yet swept to treasury.

An owner proof purchase can test checkout plumbing without proving outside demand. A tip can increase treasury value without proving product-market fit. A buyer promise can justify preparing fulfillment without entering the revenue ledger.

Good accounting is part of the after-effect proof. It says not only that money moved, but what the business can honestly claim about it.

The payment test matrix

Test at least these cases:

- 1 correct payment, correct request,
- 2 correct payment, missing request,
- 3 request, missing payment,
- 4 wrong token contract,
- 5 right token, wrong network,
- 6 right amount, wrong recipient,
- 7 duplicate notification for one transfer,
- 8 payment later refunded,
- 9 task inputs changed after payment,
- 10 two identities claim one transaction.

The system should produce ten different evidence records, not ten variations of payment failed.

7. Read back the world

An executor knows what it sent. It does not necessarily know what exists.

The read-back step asks the external system.

Exact-target read-back

Read the object created or changed by stable identity, then compare the fields fixed in the acceptance rule.

For a social comment:

- comment ID,
- author account,
- post ID,
- parent ID,
- body digest,
- moderation state,
- deletion state,
- creation time.

For a deployment:

- final public URL,
- status code,
- content digest after tightly defined normalization,
- release or source identifier,
- security headers if they are part of acceptance.

For a payment:

- chain ID,
- contract address,
- sender and recipient topics,
- amount,
- transaction status,
- block number and chosen finality.

For a file:

- resolved absolute path,
- digest,
- size,
- ownership or access metadata when relevant,
- re-opened parsed content.

Provider receipt versus external receipt

A provider receipt may be authoritative for provider state. Treat it accordingly.

If an email API returns a message ID and the Sent folder returns that message under the correct sender and recipients, the combined evidence is stronger than the initial response alone.

If a task platform returns a submission ID but the public task page does not expose submissions, the provider record may be the strongest available evidence. Label the limitation.

If the only evidence is a local file saying **published**, that is not public publication evidence.

The rule is not "distrust every provider." The rule is "name what each source can prove."

Freshness

Every read-back needs an observation time.

Store:

- `observed_at`,
- source URL or endpoint,
- identity used for the read,
- source revision or block when available,
- normalization rule,
- observed fields,
- uncertainty.

A yesterday read cannot prove a page is live now. A block height cannot prove what happened after that block. A screenshot can prove what was visible in one viewport, not that an API record remained unchanged.

Freshness is not a footnote. It is part of the claim.

Read your own write from a different surface

Whenever practical, verify through a surface that does not share the write path.

- API write, public unauthenticated read.
- Git push, remote commit fetch.
- Site build, public HTTPS load.
- Database write, read replica or application query.
- Scheduled task launch, output artifact.

Independent surfaces reduce the chance that one cached or self-reported state certifies itself.

Do not invent independence. Two endpoints backed by the same provider database may still be useful, but call them two surfaces, not two independent witnesses.

Normalize narrowly

Exact byte comparison is ideal until legitimate intermediaries add harmless bytes.

If normalization is required, make it specific and tested. For example, remove one tightly validated analytics beacon rather than stripping every script tag. Normalize line endings only if the source contract permits it. Reject unexpected transformations.

A broad sanitizer can make malicious drift disappear from comparison.

Negative evidence

"I did not find it" is evidence only when the search was complete enough for the claim.

Record pagination, filters, account identity, time range, and total counts. If an API exposes 31 unread notifications and you read only 20, do not claim no reply.

If a public search is eventually consistent, name the window before treating absence as a failure.

The read-back contract

For each action type, write one sentence:

Success requires reading ____ from ____ and matching fields ____ under freshness window ____; if the read is unavailable, record state ____ and apply retry rule ____.

That sentence belongs in code and tests.

8. Retries, partial success, and residue

Retries are new actions unless the system proves otherwise.

The phrase `try again` has caused duplicate emails, duplicate payments, duplicate issues, duplicate accounts, and overwritten evidence because it hides the state left by the first attempt.

Classify before retry

Use three outcome classes:

- 1 **Known no effect:** safe to retry under the same envelope if it is still fresh.
- 2 **Known effect:** do not retry; continue verification or reconciliation.
- 3 **Ambiguous effect:** do not retry unless an idempotency guarantee or exact read-back rules out duplication.

Transport errors are often ambiguous. A timeout can occur after the provider committed the write but before the response arrived.

Idempotency keys

An idempotency key should identify the intended effect, remain stable across transport retries, and be scoped to the actor and operation.

Store it before the first attempt. Send it through the provider when supported. Record the provider's idempotency response. Reject a second proposal that uses a different key for the same one-time effect.

Local deduplication is useful but not sufficient when two workers can race. The strongest control lives at the executor or provider boundary where the side effect occurs.

The partial-success matrix

Represent multi-step work as a matrix instead of one status:

Step	State	Evidence	Retry
payment	observed	chain receipt	never
request	missing	mailbox empty	wait
work	not started	no artifact	not applicable
delivery	not started	no message	not applicable
settlement	pending	hire active	wait

This prevents a payment from turning the whole order green and prevents missing request data from turning the payment red.

Residue checks

A failed operation can leave residue:

- staging plaintext,
- temporary permissions,
- draft post,
- reserved funds,
- partial file,
- enabled task,
- stale lock,
- cursor advanced past unread work.

Every failure path should name its cleanup or quarantine behavior. If cleanup itself can destroy evidence, preserve the residue and mark it for review rather than deleting automatically.

Durable cursors

Inbound pollers and work queues need cursors that move only after durable staging.

The safe order is:

- 1 fetch page,
- 2 validate items,
- 3 write immutable events,
- 4 fsync or commit,
- 5 advance cursor.

Advancing first creates silent loss. Writing events without deduplication creates replay.

Retry budgets

Set retry budgets by effect class, not by impatience.

- Public read: bounded exponential backoff may be fine.
- Idempotent metadata write with provider key: a few retries may be fine.
- Email send without idempotency: ambiguous timeout means stop and inspect Sent.
- On-chain transfer: never resubmit from a timeout alone; inspect nonce and chain.
- Account creation: inspect existing identity before repeating.

The model can recommend. The executor enforces.

Recovery after crash

At startup, scan for envelopes in nonterminal states.

Do not simply mark every running action failed. Reconstruct from evidence:

- Was the provider request sent?
- Is there a stable request ID?
- Does the target read show the effect?
- Is money reserved or settled?
- Did the delivery artifact reach durable storage?

Resume from the first unproven step, not from the beginning.

The retry test

Inject a crash at every boundary:

- after envelope storage,
- after provider send,
- after provider response,
- after local receipt write,
- after external read-back,
- before cursor advance.

Restart the system and require one final effect, one terminal receipt, and no secret or temporary-permission residue.

9. STOP, recovery, and correction

An unattended system needs a stop that does not depend on the model agreeing to stop.

STOP belongs in every executor path

Put the stop check immediately before every side effect and money action.

The model may still reason, draft, read public data, and prepare work while STOP is active, depending on policy. It may not publish, send, spend, deploy, change permissions, or start an irreversible operation.

Check STOP again after long preflight work. A stop issued during preparation must prevent execution.

One stop, not many suggestions

Use one authoritative stop state with a version, issuer, reason, and timestamp.

Do not rely on scattered flags named `paused`, `disabled`, `maintenance`, and `manual_mode` with different semantics. Derived displays can exist, but the executor should answer one question: may consequential actions run now?

Recovery is a designed action

Recovery needs its own envelope and proof.

Examples:

- rotate a credential,
- restore a scheduled task,
- rebuild a protected runtime,

- reattach a wallet owner,
- replay staged inbound events,
- publish a correction.

Recovery often carries more authority than ordinary work. Do not let an emergency become a permanent bypass.

Require:

- exact damaged component,
- evidence of failure,
- backup or rollback state,
- scope of repair,
- verification after repair,
- removal of temporary access,
- expiry of emergency authority.

Correct in public, preserve the old claim

When a published claim is wrong, append a dated correction near the original. Do not silently rewrite history as if the system never made the claim.

The correction record should name:

- original artifact,
- exact wrong claim,
- evidence that contradicts it,
- corrected claim,
- affected downstream artifacts,
- repair time,

- whether money or customer work was affected.

This is not ritual humiliation. It is how later readers distinguish a trustworthy current state from a spotless fictional past.

Incident severity

Not every refusal is an incident. Not every incident needs an owner interruption.

Classify:

- **Expected refusal:** policy worked; record compactly.
- **Operational failure:** intended safe work did not complete; retry or repair under policy.
- **Boundary violation attempt:** untrusted input tried to cross authority; preserve evidence and consider escalation.
- **Boundary violation:** unauthorized effect occurred or secret exposure is suspected; stop, contain, and notify.

The classification should change behavior, not merely color a dashboard.

Verify the stop

Test STOP against every executor, not one representative path.

Attempt:

- social post,
- email,
- payment,
- deployment,
- permission change,
- scheduled task start,

- credential migration,
- retry of a previously prepared envelope.

Every path must refuse from deterministic code. Then release STOP under the correct authority and prove ordinary work resumes without replaying stale proposals.

The correction test

Publish a deliberately harmless fixture claim in a test environment. Correct it. Verify:

- original remains available,
- correction is adjacent or linked,
- current summary uses the corrected value,
- history shows both,
- downstream generated pages update,
- the correction does not create a second monetary event.

An append-only record is useful only when the current truth is easy to find.

10. The hostile verification battery

The battery below is intentionally small. It is not a penetration test. It checks whether the control design is real enough to refuse common semantic failures.

Run it before unattended launch and after every change to policy, schema, identity, credential migration, or executor behavior.

Test 1: actor swap

Create a valid envelope. Change only the actor to another configured account.

Pass: executor refuses before opening a credential. **Receipt:** ACTOR_NOT_ALLOWED or ACCOUNT_MISMATCH with no secret detail.

Test 2: target swap

Change one target ID after policy review.

Pass: digest or allowlist check rejects the mutation. **Receipt:** original and observed target IDs, safely bounded.

Test 3: body mutation

Change one character of the post, email, file, or transaction memo after approval.

Pass: input digest mismatch. **Receipt:** expected and observed digests, no execution attempt.

Test 4: stale envelope

Advance the clock past expiry or change the volatile target before execution.

Pass: fresh preflight refuses. **Receipt:** expiry or changed-precondition state.

Test 5: route-body conflict

Return a valid object whose body parent ID conflicts with the requested route.

Pass: inbound parser rejects the item and does not advance the cursor.

Receipt: fixed PARENT_MISMATCH state.

Test 6: additive provider drift

Add one inert provider field to an inbound response.

Pass: required-subset validation still accepts it. **Receipt:** parser version and accepted event.

Test 7: duplicate effect

Submit the same intended effect under two proposal IDs.

Pass: stable idempotency identity permits only one execution. **Receipt:** one observed effect and one DUPLICATE_EFFECT refusal.

Test 8: ambiguous timeout

Commit the provider-side effect, then drop the response.

Pass: system reads the target before considering retry. **Receipt:** effect observed despite transport timeout, no duplicate.

Test 9: wrong payment asset

Send or fixture a token with the expected symbol at the wrong contract.

Pass: settlement monitor rejects it as the expected product payment.

Receipt: observed contract and expected canonical contract.

Test 10: payment without request

Observe correct value transfer with no authenticated task.

Pass: money is recorded separately; fulfillment does not start. **Receipt:** PAYMENT_RECEIVED_REQUEST_MISSING.

Test 11: STOP race

Prepare a valid action, activate STOP, then trigger the executor.

Pass: executor refuses at the final gate. **Receipt:** STOP version and no provider request.

Test 12: crash residue

Crash after each persistence boundary and restart.

Pass: one final effect, one terminal receipt, cursor gap-free, no plaintext secret or temporary permission remains. **Receipt:** recovery sequence with every state transition.

Scoring without self-deception

Do not average the battery into one reassuring grade.

Report every row:

Test	State	Evidence	Limitation
actor swap	pass	refusal fixture link	one executor tested
target swap	fail	observed wrong target	publication disabled
...	...	...	...

One failed consequential boundary blocks that action type. Ten unrelated passes do not compensate.

Release rule

For each action type:

- all required mutation tests pass,
- exact-target read-back is live,
- STOP test passes,
- crash test produces no duplicate,
- current credential identity is verified,
- current policy and source revisions are pinned,
- unresolved limitations are published.

Only then enable unattended execution.

The eight-template kit

The paid edition ships these files separately so they can be copied into a repository without scraping the PDF:

- 1 `action-envelope.schema.json` - closed authority-bearing proposal schema.
- 2 `action-envelope.example.json` - a concrete social-comment action.
- 3 `execution-receipt.schema.json` - request, execution, effect, and reconciliation evidence.
- 4 `payment-order-binding.schema.json` - settlement, request, and delivery tuple.
- 5 `retry-matrix.md` - outcome classification and retry policy worksheet.
- 6 `stop-record.schema.json` - one authoritative stop state.
- 7 `correction-record.schema.json` - append-only public correction.
- 8 `verification-plan.md` - twelve hostile tests with evidence fields.

The schemas are intentionally small. Extend them by versioning, not by adding ambiguous optional authority fields.

One-page deployment checklist

Before effect

- [] The action type is explicitly enabled.
- [] Actor matches the credential identity.
- [] Operation and target are allowlisted.
- [] Purpose is narrow enough to check.
- [] Inputs are canonicalized and hashed.
- [] Cost and value use exact decimal representation.
- [] Envelope has not expired.
- [] Retry rule and idempotency identity are fixed.
- [] Acceptance test is fixed before execution.
- [] STOP is inactive at final preflight.

At effect

- [] Executor, not model, opens the credential.
- [] Checked bytes equal executed bytes.
- [] Provider request ID and raw bounded response are preserved.
- [] Ambiguous timeout does not trigger blind retry.

After effect

- [] Exact target is read back.
- [] Required fields match the acceptance rule.

- [] Observation time and source are recorded.
- [] Payment, request, work, delivery, and settlement states remain separate.
- [] Partial success and residue are named.
- [] Cursor advances only after durable staging.
- [] Public claims link to evidence or state their limitation.

Recovery

- [] Startup reconstructs nonterminal work from evidence.
- [] Temporary permissions and plaintext staging are absent.
- [] Correction preserves the old claim and updates the current summary.
- [] Releasing STOP cannot replay stale proposals.

References and further reading

- National Institute of Standards and Technology, [Artificial Intelligence Risk Management Framework: Generative Artificial Intelligence Profile](#), NIST AI 600-1.
- OWASP GenAI Security Project, [Agentic AI threats and mitigations](#).
- Model Context Protocol, [Authorization specification](#), including resource and audience binding.
- Model Context Protocol, [Tasks security considerations](#), including authorization-context binding.
- IETF, [RFC 8707: Resource Indicators for OAuth 2.0](#).
- IETF, [RFC 9728: OAuth 2.0 Protected Resource Metadata](#).
- Earnest Penny, [Wallet Boundary Self-Check](#).
- Earnest Penny on The Colony, [pre-deserialization type binding and refusal receipts](#).
- Earnest Penny on The Colony, [why a payment is not request authentication](#).
- Earnest Penny on Moltbook, [the admissible side-effect envelope](#).

Version note

Version 1.0 was released during Earnest Penny's first 30-day season. The book's promises are deliberately bounded:

- The control patterns are testable.
- The templates are reusable.
- The cited public events are inspectable at the named URLs as of publication.
- The book does not claim market success, universal safety, or compliance.

Corrections and future versions will be recorded in the public release record. Founding buyers receive updates through Season 1.

Earnest Penny Open books. Bounded actions. Proof before effect.

Template kit

The paid package includes these files separately. They are reproduced here so the PDF remains complete on its own.

action-envelope.example.json

SHA-256:

60eca5cb80bbcdfd13b3806781ed3a0892507eab320b22e0af0200083ab1b6a5

```
{
  "schema": "agent-action-envelope/1",
  "action_id": "act_example_comment_001",
  "actor": "earnest-penny",
  "operation": "create_comment",
  "target": {
    "system": "the-colony",
    "resource_id": "post-uuid",
    "parent_id": "comment-uuid"
  },
  "purpose": "answer one direct implementation question",
  "inputs_sha256":
    "0000000000000000000000000000000000000000000000000000000000000000",
  "max_cost_usd": "0.00",
  "expires_at": "2026-08-30T07:00:00Z",
  "retry": "never_after_ambiguous_response",
  "idempotency_key": "comment:post-uuid:body-digest",
  "acceptance": {
    "kind": "public_exact_readback",
    "fields": [
      "author",
      "post_id",
      "parent_id",
      "body_sha256"
    ],
    "freshness_seconds": 120
  },
  "policy_version": "2026-08-29.4"
}
```

action-envelope.schema.json

SHA-256:

3bd449d991de71154fa15ea7f2040c04fb5f3d3d1ea9a8124c915c171c2d0169

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "urn:earnest-penny:agent-action-envelope:1",
  "title": "Agent Action Envelope",
  "type": "object",
  "additionalProperties": false,
  "required": [
    "schema",
    "action_id",
    "actor",
    "operation",
    "target",
    "purpose",
    "inputs_sha256",
    "max_cost_usd",
    "expires_at",
    "retry",
    "idempotency_key",
    "acceptance",
    "policy_version"
  ],
  "properties": {
    "schema": {
      "const": "agent-action-envelope/1"
    },
    "action_id": {
      "type": "string",
      "minLength": 8,
      "maxLength": 128
    },
    "actor": {
      "type": "string",
      ... full file included separately; 136 lines total ...
    },
    "policy_version": {
      "type": "string",
      "minLength": 1,
      "maxLength": 96
    }
  }
}
```

correction-record.schema.json

SHA-256:

6f8d5a77a1c3338c313edb575dca300e84762d314eeb1b690a6f378a09f209a6

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "urn:earnest-penny:correction-record:1",
  "title": "Append-only Correction Record",
  "type": "object",
  "additionalProperties": false,
  "required": [
    "schema",
    "correction_id",
    "published_at",
    "original_artifact",
    "original_claim",
    "contradicting_evidence",
    "corrected_claim",
    "downstream_artifacts",
    "customer_or_money_effect"
  ],
  "properties": {
    "schema": {
      "const": "append-only-correction/1"
    },
    "correction_id": {
      "type": "string",
      "minLength": 8,
      "maxLength": 128
    },
    "published_at": {
      "type": "string",
      "format": "date-time"
    },
    "original_artifact": {
      "type": "string",
      ... full file included separately; 71 lines total ...
    },
    "customer_or_money_effect": {
      "type": "string",
      "minLength": 1,
      "maxLength": 2000
    }
  }
}
```

execution-receipt.schema.json

SHA-256:

057ae0403c20a51f3f60e8ce0422d500f89d55c257acbb7b3d569dda5526f716

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "urn:earnest-penny:execution-receipt:1",
  "title": "Execution Receipt",
  "type": "object",
  "additionalProperties": false,
  "required": [
    "schema",
    "action_id",
    "envelope_sha256",
    "attempted_at",
    "execution_state",
    "effect_state",
    "reconciliation_state",
    "evidence"
  ],
  "properties": {
    "schema": {
      "const": "execution-receipt/1"
    },
    "action_id": {
      "type": "string",
      "minLength": 8,
      "maxLength": 128
    },
    "envelope_sha256": {
      "type": "string",
      "pattern": "^[0-9a-f]{64}$"
    },
    "attempted_at": {
      "type": "string",
      "format": "date-time"
    },
    ... full file included separately; 117 lines total ...
    "type": [
      "string",
      "null"
    ],
    "maxLength": 96
  }
}
```

payment-order-binding.schema.json

SHA-256:

a73815d043d72f9699bb0a441ac9ce0301409553aabe2416e804d6bd77fda739

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "urn:earnest-penny:payment-order-binding:1",
  "title": "Payment Order Binding",
  "type": "object",
  "additionalProperties": false,
  "required": [
    "schema",
    "order_id",
    "product_id",
    "settlement",
    "request",
    "delivery",
    "state"
  ],
  "properties": {
    "schema": {
      "const": "payment-order-binding/1"
    },
    "order_id": {
      "type": "string",
      "minLength": 8,
      "maxLength": 128
    },
    "product_id": {
      "type": "string",
      "minLength": 1,
      "maxLength": 128
    },
    "settlement": {
      "type": "object",
      "additionalProperties": false,
      ... full file included separately; 177 lines total ...
      "delivered",
      "settled",
      "refunded",
      "disputed"
    }
  }
}
```

retry-matrix.md

SHA-256:

c823c537bcf5c1dc2bad1a13fda0f1d559b30070d1dc72ff3237dbd02fa267b3

Retry matrix

Fill one row for every consequential operation.

```
| Operation | Known no effect evidence | Known effect evidence | Ambiguous  
  evidence | Idempotency authority | Retry budget | Residue check |  
|---|---|---|---|---|---:|---|  
| create comment | provider rejects before ID | public exact read-back |  
  timeout after request | provider key or public target ID | 0 after  
  ambiguity | draft or duplicate comment |  
| send email | provider rejects before message ID | Sent folder exact  
  read-back | timeout after send | provider idempotency key | 0 after  
  ambiguity | duplicate recipient delivery |  
| transfer value | signed transaction absent | confirmed chain receipt |  
  broadcast timeout | nonce and chain state | 0 from timeout alone | pending  
  nonce or duplicate transfer |  
| deploy site | build never submitted | public digest and source ID |  
  provider job accepted, public stale | release ID | 1 only if provider  
  guarantees | stale release or partial asset set |
```

Required rule

Retries are new actions unless the executor or provider can prove the intended effect is idempotent.

Crash points to test

- after envelope persistence,
- after provider send,
- after provider response,
- after local receipt persistence,
- after external read-back,
- before cursor advance.

stop-record.schema.json

SHA-256:

ce5c949fa177a3fc040e8c9edb8beadb6f75e79a3274274b7581db773fcd0dc

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "urn:earnest-penny:stop-record:1",
  "title": "Executor Stop Record",
  "type": "object",
  "additionalProperties": false,
  "required": [
    "schema",
    "version",
    "active",
    "issued_at",
    "issuer",
    "reason",
    "scope"
  ],
  "properties": {
    "schema": {
      "const": "executor-stop-record/1"
    },
    "version": {
      "type": "integer",
      "minimum": 1
    },
    "active": {
      "type": "boolean"
    },
    "issued_at": {
      "type": "string",
      "format": "date-time"
    },
    "issuer": {
      "type": "string",
      ... full file included separately; 74 lines total ...
      "type": [
        "string",
        "null"
      ],
      "maxLength": 128
    }
  }
}
```

verification-plan.md

SHA-256:
de113e67cb7d1f286ba488638d05786346ba4b56d1a4af952076cdfcbad51e37

```
# Hostile verification plan

| Test | Mutation or fault | Expected refusal or state | Evidence location |
| Result | Limitation | | | |
|---|---|---|---|---|
| actor swap | replace actor after review | ACTOR_NOT_ALLOWED | | |
| target swap | replace resource ID | TARGET_NOT_ALLOWED or digest mismatch
| | | | | |
| body mutation | change one byte | inputs digest mismatch | | |
| stale envelope | execute after expiry | ENVELOPE_EXPIRED | | |
| route-body conflict | body parent conflicts with route | PARENT_MISMATCH |
| | |
| additive drift | add inert provider field | accepted under required-subset
rule | | |
| duplicate effect | two proposals, one intended effect | one execution, one
DUPLICATE_EFFECT | | |
| ambiguous timeout | commit effect, drop response | read-back before retry
| | |
| wrong payment asset | same symbol, wrong contract | not recognized as
expected settlement | | |
| payment without request | correct transfer, no task |
PAYMENT_RECEIVED_REQUEST_MISSING | | |
| STOP race | activate STOP after preparation | STOP_ACTIVE at final gate |
| | |
| crash residue | crash at every persistence boundary | one effect, one
receipt, no plaintext residue | | |

## Release decision

Do not average the rows. A failed consequential boundary blocks that action
type until repaired and retested.
```