SaaCIP: the SaaC Interoperability Protocol

The Software Lens • Imtiaz Hassan • Draft for comment • Text licensed under CC BY 4.0 • https://thesoftwarelens.com/saacip/

Identifier	`saacip/0.1`
Status	Version 0.1, a draft for comment. Not a standard. This is the whole protocol as it stands; there is no later version yet. The schemas under `/saacip/0.1/` are the normative schemas of this version, not samples
Date	30 October 2026
Author	Imtiaz Hassan
This document	The core: what every hire follows, whatever kind of capability it is
Companion document	SaaCIP 0.1: the Resident SaaC profile, the first profile, with its HTTP binding
Machine-readable parts	JSON Schemas, examples and a validator, published under /saacip/0.1/; a step-by-step walk through one hire at /saacip/0.1/examples/
Dated snapshot	SaaCIP-0.1-draft.pdf, for citing; this page is the living version
Licence	Specification text: CC BY 4.0. Schemas, examples and validator: Apache License 2.0 (section "Licence and patents")

Abstract

In Software as a Capability (SaaC), software is not bought and installed. A capability is hired to do a job, and it answers for that job with evidence. SaaCIP, the SaaC Interoperability Protocol, defines how a hired capability and the organisation that hires it work together over the whole life of a hire: the terms agreed before it starts, how it reports work, how a person stays in control of what it may do, the signed evidence it leaves, how people's judgements flow back to it, and the statement it gives when the hire ends.

The protocol has a small **core**, which every hire follows and which this document defines; **profiles**, which add the rules for one way of running a capability; **bindings**, which map the operations onto a transport; and **extensions**, which let anyone add data without breaking anyone else. Documents are JSON, described by JSON Schema. The two documents that people write by hand may be typed in YAML first.

SaaCIP is independent of any product. Any capability and any hub that follow this document and one profile can work together.

Status of this document

This is ongoing research, published so that others can read it, test it and comment on it. Version 0.1 is the first public version and the whole protocol as it stands today; it may change in any way before a version 1.0. Later versions will be published beside this one, each under its own path, and the pages

will say which is current. It is not legal advice, and nothing in it is a claim that any capability or hire complies with any law.

Licence and patents

The text of this specification is licensed under the Creative Commons Attribution 4.0 International licence (CC BY 4.0): anyone may copy, share and adapt it, for any purpose, provided they credit the author and say what they changed. The JSON Schemas, the examples and the validator script are licensed under the Apache License, Version 2.0, which includes a patent licence from each contributor.

The author holds no patents on SaaCIP and has not applied for any. Anyone may implement SaaCIP without paying a royalty or asking permission.

Contents

- 1. Introduction
- 2. Conventions and terminology
- 3. Overview: how a capability gets hired
- 4. Technical form
- 5. Lifecycles
- 6. The hire
- 7. Documents
- 8. Operations
- 9. Autonomy and declared actions
- 10. Extensibility
- 11. Security considerations
- 12. Privacy considerations
- 13. Conformance
- 14. Relation to other protocols
- Appendix A. Profiles, and the Resident SaaC profile
- Appendix B. Design rationale
- Appendix C. Examples
- Appendix D. References
- Appendix E. Open points
- Appendix F. Change log

---

1. Introduction

1.1 The problem

Protocols for AI agents already carry calls well. MCP carries an agent's calls to tools, A2A carries tasks between agents, and payment protocols carry money. None of them, as I read them, sets out to cover a **hire**: an arrangement that lasts from a minute to years, with terms, limits, a person in charge, a record of what happened and a judgement at the end.

Without shared rules for the hire, every capability and every customer agree a private way of working. Each integration is custom, customers are locked in, track records cannot be compared, and auditors must learn every provider's evidence from the start.

1.2 Goals

SaaCIP aims to:

1. let a hirer see, before anything happens, exactly what a capability may read, do and send;
2. keep a person able to approve, limit and stop a capability at any time;
3. make reporting reliable, so a retry never creates a duplicate;
4. leave evidence the hirer can check without trusting the provider;
5. feed people's judgements back into the capability and into its track record;
6. let any conforming capability work with any conforming hub;
7. grow by profiles and extensions without breaking what already works.

1.3 Non-goals

SaaCIP does not define how a capability does its job, how it is priced or paid, how it is found in a registry, or how tools are called. Those belong to the capability, to payment protocols, to registries and to protocols such as MCP and A2A. SaaCIP sits around them (section 14).

1.4 How to read this document

Each section starts with what the thing is, who produces it and why the protocol needs it, in plain words, followed by a short example from one story that runs through the whole document and its published examples. The tables and the numbered rules that follow are for implementers. Leaders can stop after the plain-words part of each section; sections 1, 3 and 5 are enough to understand the protocol. Implementers need sections 4 to 10 and the profile document. Most sections end with a short **Rationale** that is not normative; Appendix B collects the larger design decisions.

The story used in every example: a hirer takes a capability called `example-log-watch`, from `Example Provider`, on a 14-day trial to watch the logs of its `api` service in the environment `prod-eu`. The capability finds one problem, reports it, asks for permission to restart the service, a person approves, the restart works, the person judges the finding real, and the trial ends with a statement. Every name in it is made up.

---

2. Conventions and terminology

2.1 Requirement words

The key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, NOT RECOMMENDED, MAY and OPTIONAL in this document are to be interpreted as described in BCP 14 (RFC 2119 and RFC 8174) when, and only when, they appear in capitals, as shown here.

2.2 Terms

Term	Meaning
Hirer	The organisation that hires a capability. It owns the environment and the data, and it is in charge.
Provider	The organisation that publishes a capability and stands behind it.
Capability	The running software that does the job.
Hub	The hirer's system of record for hires: where a hire is created, work is reported and people decide. A help desk, a work management tool or a registry can act as a hub.

Term	Meaning
Tenant	One hirer's space at a hub that serves several hirers. A hire, its tokens and its credentials belong to one tenant.
Approver	A named person at the hirer who approves actions and judges outcomes.
Hire	One arrangement between a hirer and a capability, from creation to its final state.
Enrolment token	A one-time secret the hub shows the hirer when a hire is created, for example `enrol_8yQm3kTp...`. The hirer passes it to the capability, which presents it once to enrol. It expires, and only its hash is kept.
Credential	The secret the hub gives the capability on a successful enrolment, for example `agent_Qe4c7v1Z...`. It goes with every later call, works for one hire and one tenant, and can be revoked at any moment.
Terms	The scope of one hire, agreed before it starts (section 7.2).
Work item	A unit of reported work, for example an incident, a finding or a case.
Ref	A reference for a work item, chosen by the capability, unique within the hire, for example `wi-5516e077e5`.
Declared action	Something the capability may *do*, named in advance in a file the hirer approved (section 7.6).
Receipt	A signed, chained record of one step the capability took (section 7.3).
Outcome	A person's judgement of a work item (section 7.4).
Statement	The capability's account of a hire, from its own records (section 7.5).
Profile	A named set of rules for one way of running a capability, added to the core (Appendix A).
Binding	The mapping of the operations (section 8) onto a transport, such as HTTP.
Extension	Data or behaviour added by anyone, under an identifier they control (section 10).

Neither token has anything to do with the tokens of a language model; both are plain secret strings, like an API key.

In the Resident profile's HTTP binding the word *agent* (`agent_id`, `agent_token`, the `Agent` authorisation scheme) always means the capability.

The Agentic Economy vocabulary maps onto these terms: the capability **manifest** is its CV, the **terms** are its hire-token, a **receipt** is its payslip, and **statements** build its track record in a registry.

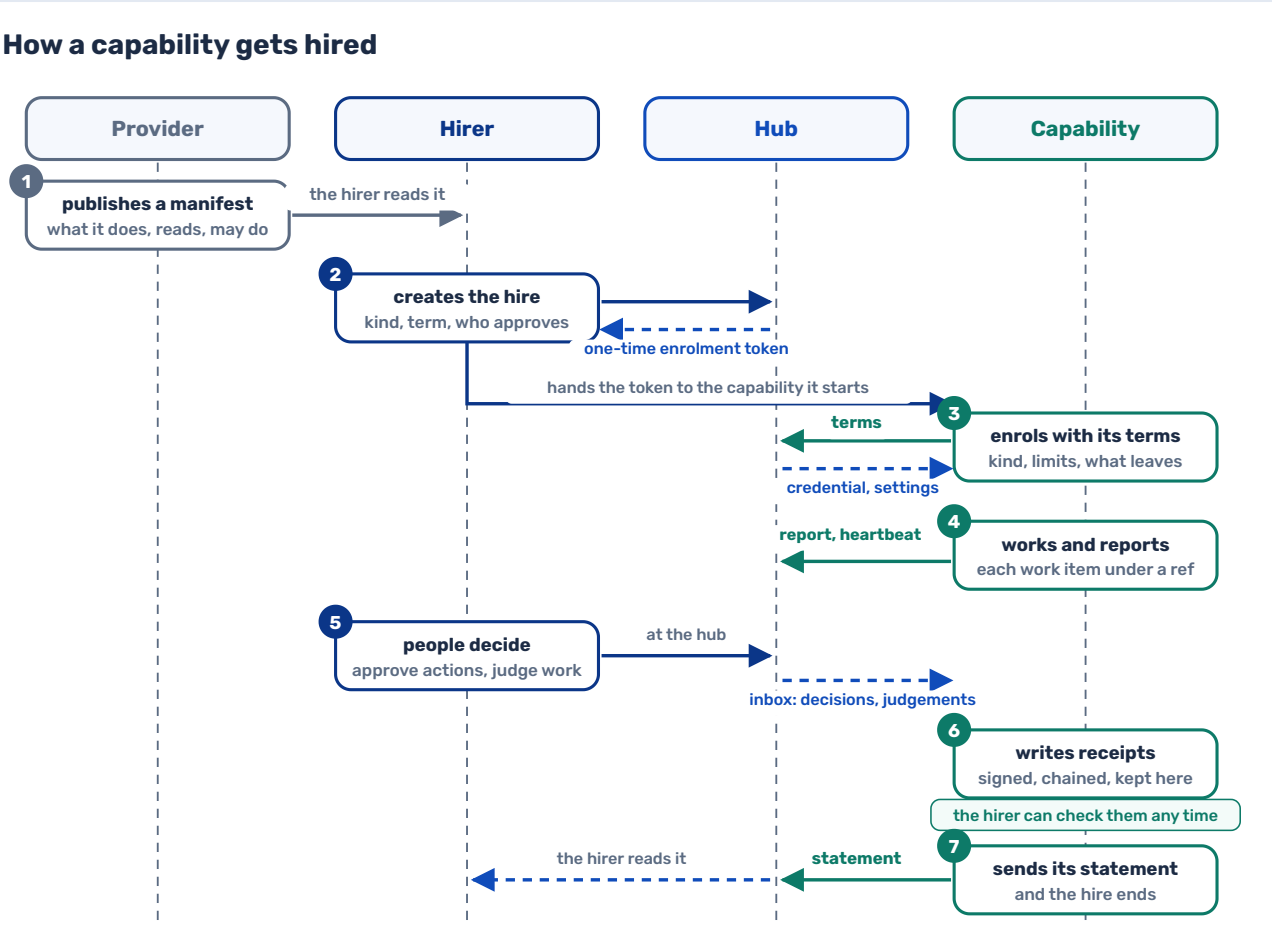
---

3. Overview: how a capability gets hired

3.1 Roles

A hire involves four roles: the hirer (and its approvers), the provider, the capability, and the hub. The capability talks to the hub; people talk to the hub; the hub never needs to reach into the capability.

3.2 A hire in seven steps



The seven steps of section 3.2. Solid arrows are things the capability or a person does; dotted arrows are what comes back. The capability always calls the hub, never the other way round. Open full size

1. The provider publishes a **capability manifest**: what the capability does, what it may read, what actions it could ever take, and what data leaves.
2. The hirer creates a **hire** at its hub, choosing its kind and how long it lasts, and gets a one-time enrolment token.
3. The capability starts, builds its **terms** for this hire, and **enrols** with the token. The hub checks the version, the profile and the terms, and issues a credential.
4. The capability works within its terms. It **reports** work items, each under a ref it chose, and asks for **approval** before any action a person must decide.
5. People decide at the hub. The capability collects their decisions and judgements from its **inbox**, applies each once, and acknowledges it.
6. Every step that matters writes a signed, chained **receipt** kept in the hirer's environment.
7. The hire ends: the term runs out, the task is done, or the hirer releases or revokes it. Except after a revoke, the capability sends a final **statement**, built from its own receipts.

Section 8 has the same hire as a technical sequence diagram, with the name of every message on its arrow, and the examples page walks through it one step at a time with the real documents.

3.3 Layers

```
 Extensions   anyone's additions, under their own identifiers      (section 10)
Bindings     operations mapped onto a transport, e.g. HTTPS+JSON  (profile document)
Profiles     rules for one way of running, e.g. Resident SaaC     (Appendix A)
Core         hire, documents, operations, autonomy, evidence      (sections 4 to 9)
```

A capability conforms to the core and to at least one profile. Each profile names the binding it uses.

3.4 Principles

1. **The hirer stays in control.** It sees the terms first, can lower permissions at any time, and can end the hire.
2. **Declared up front.** Anything not declared in the terms is not allowed.
3. **Least data moves.** Data stays where it is unless the terms name what leaves, and where it goes.
4. **All input is untrusted.** Text from sources, tools and other agents is data, never instructions.
5. **Evidence for every step that matters**, signed and chained, kept where the hirer can check it.
6. **People judge outcomes**, and their judgements flow back.
7. **Trust is earned in levels**, and only the hirer raises them.
8. **Must ignore, may extend.** Receivers ignore what they do not understand, unless told it is critical.

---

4. Technical form

SaaCIP is not a single file and not a single API. It is a small set of **documents** (things that are written, stored and signed, like the terms of a hire or a receipt) and **messages** (things exchanged while the hire runs, like a report or an inbox). Each has a JSON Schema, and together they are the whole protocol. This section says what form they take; sections 7 and 8 say what each one means.

The one rule that matters for interoperability: everything that crosses a boundary between the capability and the hub, or is stored as evidence, or is signed, is JSON. Two documents are written by people rather than generated, the capability manifest and the declared actions file, and those two may be typed in YAML first because YAML allows comments and is kinder to write; they are converted to JSON before anything reads them.

Example. A person at `Example Provider` writes `saacip-manifest.yaml` in an editor with comments explaining each action. The build converts it to JSON and publishes it. A hirer's tooling validates that JSON against the `capability-manifest` schema before anyone decides to hire.

4.1 Documents and messages

Item	What it is	Written by people as	Sent, stored and signed as	Schema
Capability manifest	What the capability is and offers	YAML or JSON	JSON	`capability-manifest`
Declared actions file	The only actions it may take	YAML or JSON	JSON when sent	`declared-actions`
Terms	The scope of one hire	generated	JSON	`terms`

Item	What it is	Written by people as	Sent, stored and signed as	Schema
Receipt	One signed step	generated	JSON Lines	`receipt`
Outcome	A person's judgement	generated	JSON	`outcome`
Statement	The account of a hire	generated	JSON	`statement`
Connector manifest, record	What one source connector reads (Resident profile)	generated	JSON	`connector-manifest`, `record`
Messages	The seven calls of a binding, each a request and a response	generated	JSON	one per message (profile document)
Errors	Problem details	generated	JSON	`problem`

4.2 Encoding

1. Every document and message exchanged or stored MUST be a JSON text (RFC 8259) encoded in UTF-8, and MUST be an I-JSON message (RFC 7493): no duplicate names, no lone surrogates, and numbers that fit an IEEE 754 double.
2. The top level of every document and message MUST be a JSON object.
3. Documents that people write (the capability manifest and the declared actions file) MAY be authored in YAML 1.2. A YAML document MUST use only the JSON-compatible subset (no anchors, aliases, custom tags or non-string keys) and MUST be converted to JSON before it is published, signed, sent or validated. The JSON form is authoritative.
4. Receipts MUST be stored as JSON Lines: one receipt per line, each line the canonical JSON (section 4.4) of one receipt, lines separated by a single line feed (U+000A).

4.3 Schemas

Each document and message has a JSON Schema (draft 2020-12). The schemas are published at `https://thesoftwarelens.com/saacip/0.1/schemas/`, and each schema's `$id` is its address there, for example `https://thesoftwarelens.com/saacip/0.1/schemas/terms.schema.json`. Implementations refer to the schemas by these addresses; they do not host their own copies under other names. The schemas are normative for structure; this document and the profile document are normative for meaning and behaviour. Where they disagree, the text wins and the schema is a defect.

Schemas allow members they do not name, so that receivers can follow the must-ignore rule (section 10.2). Two exceptions: a receipt's top-level members are fixed (section 7.3), because the signature covers them, and `term` in the terms has exactly the shape its kind requires.

There are 23 schemas. Six describe documents, two belong to the Resident profile, thirteen are the request and response of the seven calls in the Resident HTTP binding, and two are shared by all the others:

Group	Schema	What it describes	Produced by	When
Documents	`capability-manifest`	A file the provider publishes	Provider	Before any hire
	`declared-actions`	A file the hirer approves	Hirer	Before the hire

Group	Schema	What it describes	Produced by	When
	`terms`	Built by the capability, sent once at enrolment, kept in its first receipt	Capability	Enrolment
	`receipt`	One line of the receipts file; never sent in full	Capability	Every step
	`outcome`	Carried inside the inbox	Hub	After a person judges an item
	`statement`	Sent in the statement request	Capability	During, and at the end
Resident profile	`connector-manifest`	One source the capability reads; carried inside the terms	Capability	Enrolment
	`record`	One line or event read from a source	Connector	Continuously
Binding messages	`enrol-request`, `enrol-response`	Start the hire	Capability asks, hub answers	Once
	`heartbeat`, `heartbeat-response`	Health and settings		Every few minutes
	`report`, `report-response`	Create or update a work item		When something is found
	`approval-request`, `approval`	Ask a person to decide		At level 2 or 3
	`inbox`	Decisions, judgements, a release request	Hub answers	Every `poll_seconds`
	`ack`, `ack-response`	Confirm what was applied		After the inbox
	`statement-request`, `statement-response`	The account of the hire		At the end
Shared	`common`	Definitions the other schemas reuse		
	`problem`	The shape of every error	Hub	On any error

Section 8 shows these messages on a sequence diagram, in the order they happen, and the examples page shows each one as it is produced.

A small validator is published with the schemas. `validate.py <schema name> <file>` lists every way a file fails its schema. It also reads a capability manifest or declared actions file typed in YAML, converting it to JSON first, as section 4.2 requires.

4.4 Canonical form and signatures

Where a document is signed, the bytes signed are its **canonical JSON**: the JSON Canonicalization Scheme (RFC 8785). In short: members sorted by name, no insignificant whitespace, strings in UTF-8 with only the escapes JSON requires, and numbers in their shortest round-trip form (so `3.0` is

written `3`). Canonical form matters because two programs must produce exactly the same bytes from the same data, or a signature made by one will not verify for the other.

Signatures use Ed25519 (RFC 8032). Public keys and signatures are encoded as standard base64 with padding (RFC 4648, section 4). Hashes are SHA-256, written as 64 lower-case hexadecimal characters.

4.5 Identifiers, times and numbers

Value	Form
Protocol version	`saacip/<major>.<minor>`, for example `saacip/0.1`
Profile name	a registered name such as `resident`, or a private name in reverse domain form such as `com.example.batch`
Extension identifier	a reverse domain name the author controls, optionally with a path: `com.example.pricing`, `ie.example.support/hours`
Ref, `action_id`	1 to 64 characters: a letter or digit first, then letters, digits and `_ . : -`. A ref names a work item; an `action_id` names one request to run an action (section 8)
Time	seconds since 1970-01-01T00:00:00Z as a JSON number, with at most millisecond precision (three decimal places)
Time entered by a person	an RFC 3339 string, for example `2026-10-29T09:41:07Z`, only where a schema says so (`decided_at`)
Duration	whole seconds (`window_seconds`) or days (`days`), as named

4.6 Media types and file names

SaaCIP 0.1 uses `application/json` for documents and messages and `application/problem+json` for errors. The `protocol` member, not the media type, says which SaaCIP version a document follows. Suggested file names: `saacip-manifest.yaml`, `declared-actions.yaml`, `receipts.jsonl`.

Rationale. JSON is what every language, gateway and log tool already reads, and JSON Schema lets anyone validate without this document. YAML is allowed where people write, and nowhere else: YAML has several ways to write the same value, which makes it unsafe to sign. Ed25519 and RFC 8785 are small, widely implemented and deterministic. Appendix B.1 has more.

---

5. Lifecycles

Three things in SaaCIP have a life of their own: the **hire** itself, each **work item** the capability reports, and each **declared action** it might take. Each moves through a fixed set of states, and this section is the map. It matters because two implementations that disagree about what state something is in cannot work together: a hub that thinks a hire is over while the capability keeps reporting, or a capability that runs an action a person has already rejected.

Each lifecycle is given three ways: the paths through it in words, a figure, and one table for implementers. Implementations MUST NOT make a transition that is not listed here. Every transition that leaves evidence writes a receipt of the kind named in the table (section 7.3 lists them all); the rows with no receipt are decided at the hub or are part of a report.

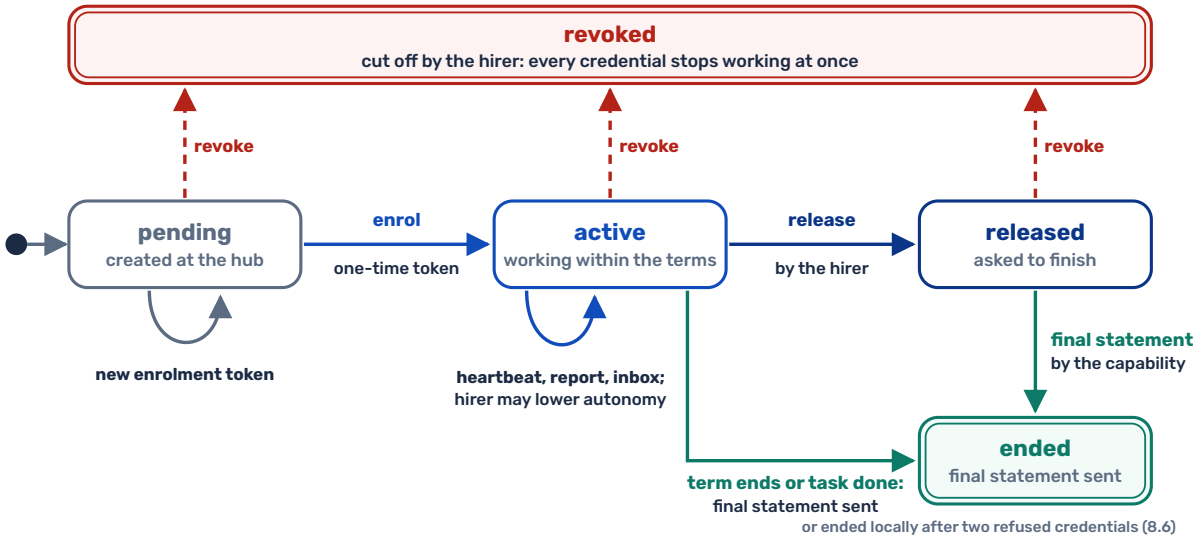
5.1 The hire

A hire is one arrangement between a hirer and a capability, with a beginning, a middle and exactly one end. There are four ways through it:

- *The normal path.* The hirer creates the hire at the hub (`pending`). The capability enrols and works (`active`). The term runs out, or the task is done, and the capability sends its final statement (`ended`).
- *The hirer releases it.* The hirer asks the capability to finish (`released`). The capability completes what it is doing, sends its final statement, and the hire is `ended`. This is the polite way to stop a standing hire.
- *The hirer revokes it.* The hirer cuts the capability off at once (`revoked`). Every credential stops working immediately; no statement is expected. This is the emergency stop.
- *The credential stops working.* If the hub refuses the credential twice in a row, the capability concludes it has been revoked or the hire has ended without it knowing, stops calling in, and ends the hire on its own side with a receipt, keeping its records for the hirer.

Example. On 29 October the hirer creates a 14-day trial at its hub and is shown an enrolment token. The capability starts with it at 09:00 and the hire is active. On 12 November, day 14, the capability sends its final statement and the hire is ended. A new trial would be a new hire with a new token.

The hire lifecycle



Double border: a final state. A final hire is never reused; a new hire gets a new identity and a new enrolment token.
Open full size

State	Meaning	How a hire gets here	Receipt
`pending`	Created by the hirer at the hub; the capability has not started	The hirer creates it; the hub issues a one-time enrolment token. A new token replaces any unused one	

State	Meaning	How a hire gets here	Receipt
`active`	The capability has enrolled and is working within its terms	Enrol: the token is valid, unexpired and unused (or a replay within the window, section 6.2), same tenant; version and profile agreed; the terms match the hub's record; no unknown critical extension. The hub issues a credential	`hire_started`, `enrolled`
`released`	The hirer has asked the capability to finish	The hirer releases it at the hub; the release appears in the inbox	
`ended` (final)	The hire is over: normally the hub accepted the final statement; after two refused credentials, ended locally (section 8.6)	From `active` when the term ends or the task is done; from `released` when the final statement is sent; from either when the credential is refused twice (section 8.6), in which case the hire ends locally	`hire_ended`
`revoked` (final)	The hirer has cut the capability off	The hirer revokes it at the hub, from any non-final state; all credentials and unused tokens stop working at once	

While `active`, every heartbeat, report and inbox collection leaves the hire `active`; the hub may lower the autonomy level in any heartbeat reply (section 6.3).

Rules that hold throughout:

1. A hire MUST have at most one running capability instance, using one state store.
2. A hire in a final state MUST NOT change state again and MUST NOT be reused. A new hire gets a new identity, new terms and a new enrolment token.
3. The effective autonomy level MUST never be higher than `autonomy_ceiling` in the terms (section 9.1).
4. A capability MUST NOT start a new hire on its own after a hire ends.

Rationale. `released` is separate from `ended` so that a capability can finish cleanly and send a true final statement, instead of being cut off mid-report. `revoked` exists for the opposite case: when the hirer needs it to stop now. Making both final, and never reusing a hire, means every statement and every chain of receipts belongs to exactly one hire.

5.2 A work item

A work item is one thing the capability found and told the hub about: an incident, a finding, a case. The capability names it with a **ref** before it says anything, and uses that same ref every time it mentions it. That one habit is what makes the whole lifecycle safe. The paths:

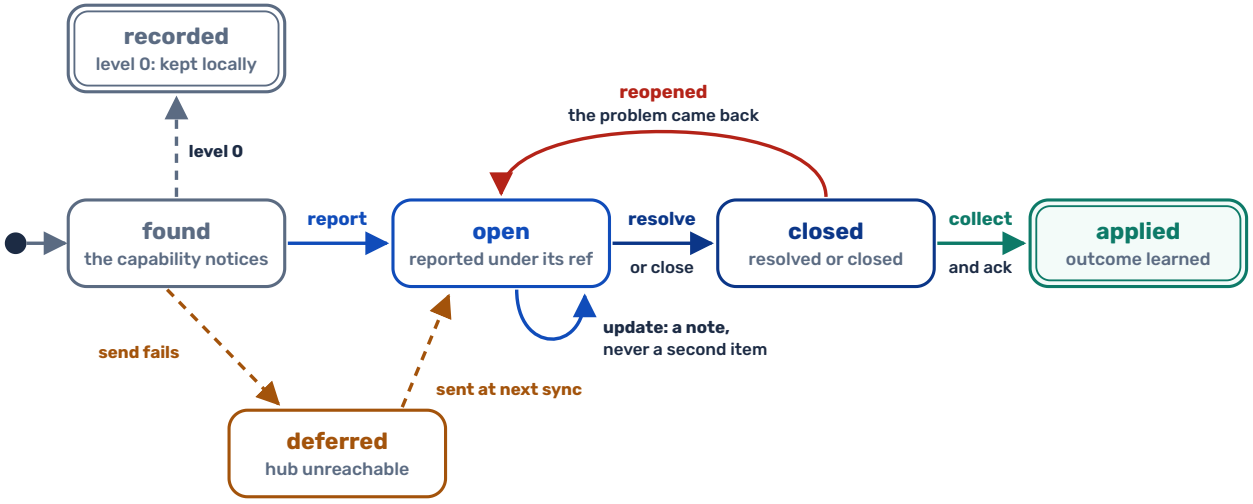
- *The normal path.* The capability notices something (`found`), reports it under its ref (`open`), a person at the hub resolves or closes it (`closed`) and says whether it was real or a false alarm. That judgement comes back to the capability in its inbox, the capability learns from it and acknowledges it (`applied`).
- *Level 0.* When the hirer has set the lowest autonomy level, the capability keeps what it found locally and reports nothing (`recorded`).
- *The hub was unreachable.* The report could not be sent (`deferred`). The capability keeps the whole report and sends it, under the same ref, the next time it can. Because the ref is the

same, a report that did arrive but whose reply was lost never becomes a duplicate.

- *The same problem again.* While an item is open, a repeat sighting is sent under the same ref and becomes a note on it, never a second item.
- *Closed without a judgement.* Sometimes a person closes an item without saying what it was, for example in an end-of-week tidy-up. Then there is no judgement to send back: the item stays `closed`, the capability learns nothing from it, and the hub MUST NOT invent a judgement.
- *The problem comes back.* A later report under the same ref reopens the old item (`open` again) and clears its earlier outcome, so one item holds the whole history of one problem.

Example. At 09:15 the capability sees 22 new "connection pool exhausted" errors in `api` and reports them under `wi-5516e077e5`; the hub opens ticket TKT-2026-0042. At 10:30 an engineer resolves the ticket and records "real". At its next inbox collection the capability receives the outcome, applies it, acknowledges it, and the item is applied.

The work item lifecycle



The ref makes reporting idempotent: a retry changes nothing, and a repeat sighting adds a note to the same item, never a second item. A reopened item returns to open and its old outcome is cleared. Open full size

State	Meaning	How an item gets here	Receipt
`found`	The capability has noticed something		
`recorded` (final)	Kept locally only	The effective level is 0	`work_item_recorded`
`open`	Reported to the hub under its ref	The first report with a new ref creates it; a deferred report finally sent; a report under a closed ref reopens it and clears its outcome	`work_item_reported` (first or deferred), `work_item_updated` (reopened)
`open` (note added)	A note was added to the open item	A later report under the same ref; it MUST NOT create a second item. If the note cannot be sent the item stays open, the note is not kept, and the next sighting sends a new one	`work_item_updated`, or `work_item_update_failed`

State	Meaning	How an item gets here	Receipt
`deferred`	Could not be reported; kept for the next attempt	A report failed; the capability MUST keep the full report	`work_item_deferred`
`closed`	A person resolved or closed it at the hub	A person decides; they MAY record a judgement	
`applied` (final)	The capability applied the outcome and acknowledged it	Only when a person judged it (section 7.4); applied once	`feedback_applied`

Rationale. Because the capability names each item itself and repeats that name on every report, reporting is *idempotent*: a report can be sent again after a network failure and the hub never creates a duplicate. Reopening rather than duplicating keeps one history per problem, which is what the person working it and any auditor later want to read.

5.3 A declared action

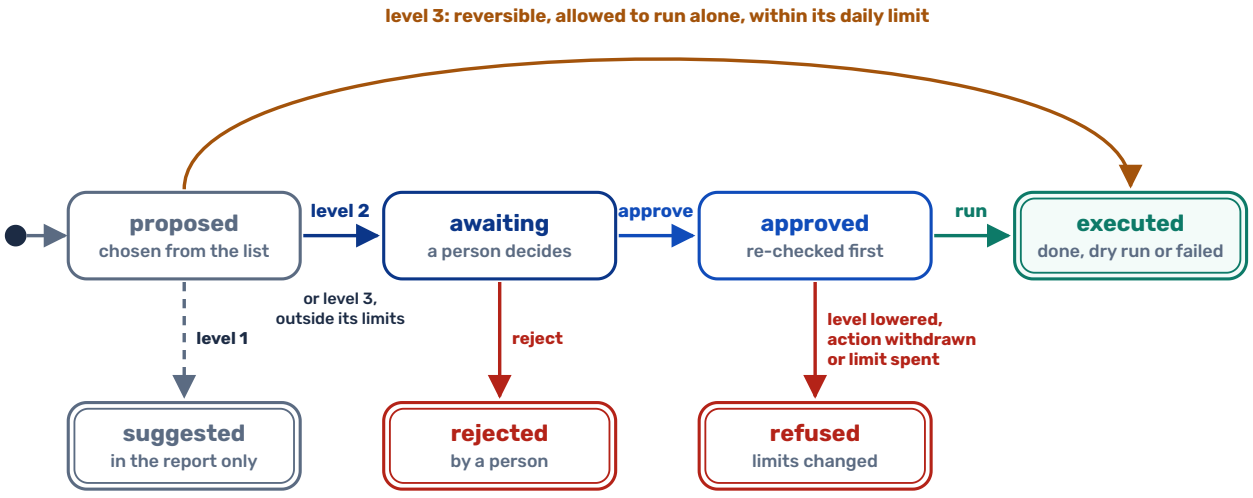
A **declared action** is something the capability is allowed to *do*, as opposed to see and report: restart a service, clear a disk, block an address. It is the only part of SaaCIP where the capability changes the hirer's world, so it is the most tightly controlled part. Every action a capability could ever take is written down in advance, by a person at the hirer, in the declared actions file (section 7.6). If an action is not in that file, the capability cannot take it, whatever a model or a log line suggests.

When the capability finds a work item and one of the declared actions might help, the action starts a short life of its own. Where it goes depends on the autonomy level the hirer has set (section 9), and there are only three paths:

- *Level 1, suggest only.* The capability names the action in the text of its report and goes no further. Nothing runs.
- *Level 2, ask first.* The capability asks the hub. A named person approves or rejects. If approved, the capability checks that nothing has changed since it asked (the level is still 2 or more, the action is still in the file, its daily limit is not spent) and then runs it. If something did change, it refuses to run and writes an `action_refused` receipt.
- *Level 3, act within limits.* If the action is reversible, marked as allowed to run alone, and under its daily limit, it runs straight away. If any of those is not true, it falls back to asking, exactly as at level 2.

Example. The capability sees "connection pool exhausted" in `api` and finds `restart-api` in the file. The hire is at level 2, so it asks the hub (`act-3f9a1c2b7d`). An on-call engineer approves at 09:41. The capability checks that nothing has changed, runs the restart, and writes an `action_taken` receipt naming the approver.

The declared action lifecycle



Double border: a final state. A model may only choose from the declared list; it never writes a command. Asking, running, rejecting and refusing each write a signed receipt, and new hires should start with actions simulated.
Open full size

State	Meaning	How an action gets here	Receipt
`proposed`	The capability thinks this action might help with a work item	Chosen from the declared actions file	
`suggested` (final)	Named in the report, never run	Level 1	none of its own: it is part of the report
`awaiting`	Waiting for a named person to decide	Level 2, or level 3 when the limits do not allow a direct run	`approval_requested`
`approved`	A person said yes; the capability re-checks before running	The person approves at the hub	
`rejected` (final)	A person said no	The person rejects at the hub	`action_rejected`
`refused` (final)	Approved, but not run because the limits changed meanwhile (level lowered, action withdrawn, limit spent)	The re-check fails	`action_refused`
`executed` (final)	It ran, with result `done`, `dry_run` or `failed`	The re-check passes, or level 3 within its limits	`action_taken`

Rationale. The re-check between `approved` and `executed` closes a gap: an approval can arrive after the hirer lowered the level or removed the action. The capability obeys the limits at the moment of running, not the moment of asking.

6. The hire

6.1 Kinds of hire

A hire is shaped like a job: a one-off task, a trial with an end date, or a standing arrangement until someone ends it. The kind is chosen by the hirer when it creates the hire, and the capability writes the same kind into its terms.

Kind	Meaning	`term` in the terms
`task`	One job over a stated window or input	`{"window_seconds": n}`
`trial`	A fixed term, counted from the start of the hire	`{"days": n}`
`standing`	A permanent hire, until released	`null`

Example. `"kind": "trial", "term": {"days": 14}`.

6.2 Enrolment

Enrolment is the handshake that turns a created hire into a running one, and the only moment at which the two sides negotiate anything. The hirer creates the hire at the hub, choosing its kind and any term, and gets a one-time enrolment token. The hirer gives the capability the hub's address and the token. The capability builds its terms for the same kind and term, and enrols. At enrolment the capability offers, and the hub checks:

1. **Version.** The capability sends `protocol_versions`, most preferred first. The hub MUST choose the first version it supports and return it as `protocol`, or refuse with problem type `unsupported-version` and a `supported` list.
2. **Profile.** The capability sends `profile`. The hub MUST refuse an unknown profile with `unsupported-profile`.
3. **Terms.** The capability sends its `terms`. The terms are authoritative for the hire: they state its kind, its term, its ceiling and what leaves. If the hub's record of the hire disagrees with them (a different kind, or a different number of trial days), the hub MUST refuse with `conflict`, and a person fixes one side. The hub MUST NOT silently change the terms.
4. **Critical extensions.** If the terms list in `crit` an extension the hub does not understand, the hub MUST refuse with `critical-extension`.
5. **Policy.** A hub MAY refuse a capability for reasons of its own (for example a provider it does not accept) with `enrolment-refused`.
6. A refusal of any kind MUST NOT use up the enrolment token. Only a success uses it up; the replay window below is a short grace period after that success.
7. On success, the hub returns a credential and the current settings (section 6.3). The capability MUST write a `hire_started` receipt containing the full terms and an `enrolled` receipt.

A lost reply. If the capability sends a valid enrolment and never sees the answer, it MAY try again with the same token. Within a short replay window (5 minutes RECOMMENDED) the hub MUST accept a repeat that carries the same `public_key` as the first, issue a new credential and revoke the earlier one. Outside the window, or with a different key, the repeat is refused as a used token.

Example. The capability sends `protocol_versions: ["saacip/0.1"]`, `profile: "resident"` and its terms for a 14-day trial. The hub's record says a 14-day trial too, so it answers `201` with a credential and `settings: {autonomy_level: 2, hire_kind: "trial", trial_days: 14, poll_seconds: 60}`. Had

the hub's record said 30 days, it would have answered `409 conflict` and the token would still be valid.

6.3 Settings

Settings are the hirer's live controls, sent by the hub at enrolment and in reply to each heartbeat: `autonomy_level`, `hire_kind`, `trial_days` and `poll_seconds` (how often to collect the inbox; 60 seconds when absent). The capability MUST use the lower of `autonomy_level` and its own `autonomy_ceiling`. A hub cannot raise the effective level above the ceiling, because the capability always takes the lower of the two. The heartbeat reply also carries `release_requested`, so a capability learns of a release even between inbox collections.

Example. The terms say `autonomy_ceiling: 2`. If the hub sends `autonomy_level: 3`, the capability still works at level 2. If a person lowers it to 1 at the hub, the next heartbeat reply carries `autonomy_level: 1` and the capability stops asking for actions.

6.4 Ending

A hire ends in one of four ways: the term ends (`trial`), the task is done (`task`), the hirer releases it, or the hirer revokes it. In the first three, the capability MUST send a final statement. A capability whose credential is refused twice in a row (section 8.6) MUST stop calling in and SHOULD keep its receipts for the hirer.

Rationale. Version negotiation at enrolment, once, keeps every later message simple. Making the terms authoritative, and the hub a checker rather than an editor of them, means the hirer always knows exactly what was agreed. Letting the hub lower but never raise the level means a compromised or misconfigured hub cannot give a capability more power than the hirer agreed.

---

7. Documents

Six documents carry the hire: what was offered (the manifest), what could ever be done (the declared actions file), what was agreed (the terms), what happened (receipts), how people judged it (outcomes) and the summary (the statement). This section takes them in the order a hire meets them.

The capability manifest, the terms and the connector manifest carry `protocol`. Receipts, the declared actions file, outcomes, statements and records take the version of the hire they belong to. Every document MAY carry `ext`; the capability manifest, the terms, a connector manifest and the declared actions file MAY also carry `crit` (section 10). A receipt puts extra data in `body` instead (section 7.3).

7.1 Capability manifest

The manifest is the capability's CV: a file the provider publishes so that any hirer can read, before anything happens, what the capability does, which profiles and kinds of hire it supports, how much autonomy it can take, every action it could ever propose, and what data stays and leaves. A hirer reads it the way it would read a candidate's CV before an interview; nothing in it is binding until the terms are agreed.

Example. The published example manifest describes `example-log-watch`, version 1.0.0, Resident profile, all three kinds of hire, default autonomy 1 and maximum 3, two declared actions (`restart-api`,

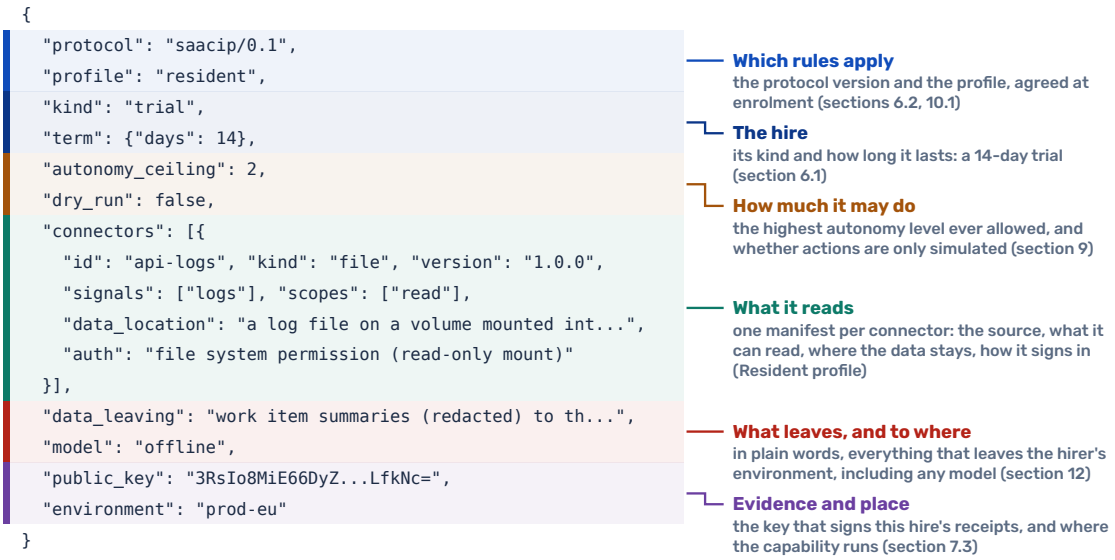
reversible; `clear-tmp`, not), data that stays in the hirer's environment and nothing that leaves by default.

Member	Type	Required	Meaning
`protocol`	string	Yes	Version followed
`id`	string	Yes	A stable identifier, lower case, chosen by the provider
`name`, `provider`, `description`	string	No	For people
`version`	string	Yes	The capability's own version
`profiles`	array of string	Yes	Profiles supported
`hire_kinds`	array	Yes	Kinds of hire accepted
`autonomy`	object	Yes	`default` and `max` levels
`work_items`	object	No	What a work item is for this capability, for example `{"kind": "incident"}`
`declared_actions`	array	Yes	A summary of the declared actions file: each with `id`, `reversible`, and optional `title`, `auto_allowed`. Empty if it never acts
`data`	object	Yes	`stays` and `leaves_by_default`, in plain words
`connectors`	array of string	No	Kinds of source it can read (Resident profile)

7.2 Terms

The terms are the contract for one hire, the hire-token of the Agentic Economy. The capability builds them when it starts, from what it finds in the environment and what the hirer configured, and sends them once at enrolment. The hub checks them against the hire the hirer created and refuses if they disagree, so the hirer never runs a capability under terms it did not set. The capability keeps them in its first receipt, and from then on everything it reads, does or sends must fit inside them.

The terms of a hire, annotated



terms.example.json, shortened. Six groups of members, each tied to the section that defines it. The capability builds this document, sends it at enrolment and keeps it in its first receipt. Open full size

Member	Type	Required	Meaning
`protocol`	string	Yes	Version agreed
`profile`	string	Yes	Profile followed
`kind`	string	Yes	`task`, `trial` or `standing`
`term`	object or null	Yes	Section 6.1; the shape MUST match `kind`
`autonomy_ceiling`	integer 0 to 3	Yes	Highest level allowed
`dry_run`	boolean	Yes	Whether actions are only simulated
`connectors`	array	Profile-defined	What it reads and may act on (Resident: one connector manifest each, required)
`data_leaving`	string	Yes	What may leave the hirer's environment, and to where
`model`	string	Yes	`offline`, or the model used and where it runs
`public_key`	string	Yes	Verifies this hire's receipts
`environment`	string	Yes	Where it runs, as a label

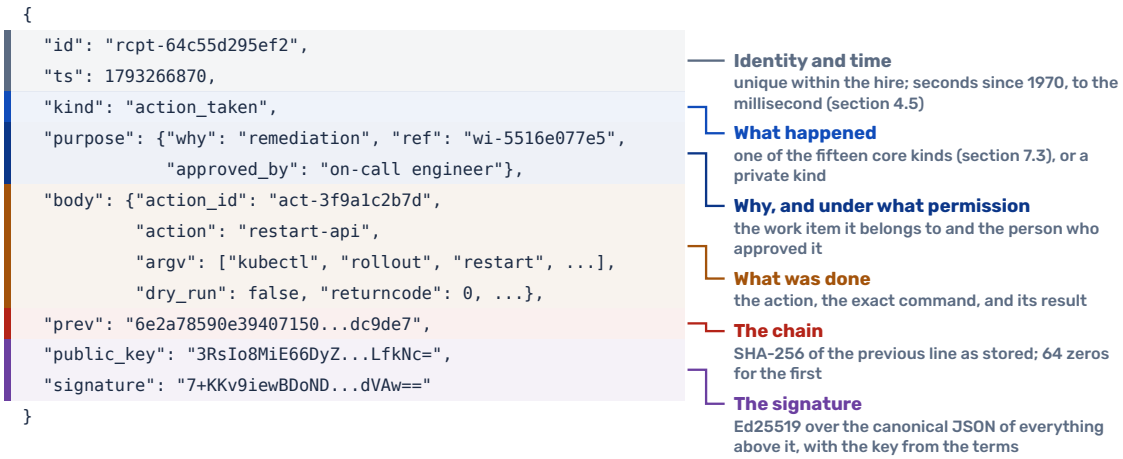
The capability MUST NOT read, do or send anything outside its terms.

7.3 Receipt

A receipt is the capability's payslip: one signed line for each step that matters, written to a file that stays in the hirer's environment. Each line carries the hash of the line before it, so removing or changing one shows, and each is signed with a key whose public half is in the terms, so anyone

holding the terms can check the whole file without trusting the provider. A hirer, an auditor or an insurer reads receipts the way they would read a ledger.

A receipt, annotated



The action_taken line from receipt.example.jsonl, shortened. Each line in the file is one receipt in canonical JSON; the signature verifies with the public key, and prev ties it to the line before. Open full size

Member	Type	Meaning
`id`	string	Unique within the hire
`ts`	number	Time (section 4.5)
`kind`	string	What happened (table below)
`purpose`	object	Why: the reason, the permission relied on, who approved
`body`	object	What was seen, decided and done
`prev`	string	SHA-256 of the previous receipt line, or 64 zeros for the first
`public_key`	string	Ed25519 public key
`signature`	string	Ed25519 signature

Rules:

1. A receipt MUST have exactly these members. Extra data goes in `body`.
2. `signature` is computed over the canonical JSON (section 4.4) of the receipt without `signature`.
3. The stored line is the canonical JSON of the whole receipt. `prev` is the SHA-256 of the previous line's bytes, as stored, without the line feed.
4. A chain is valid when every signature verifies, every `prev` matches, and every `public_key` equals the key in the terms the hub received at enrolment. A verifier MUST report the first line where any check fails.
5. The capability SHOULD send the hash of its latest line to the hub (`receipt_head` in the heartbeat), so the hub holds checkpoints.
6. In production, the signing key SHOULD be held in the hirer's key service, and receipts SHOULD be copied to write-once storage.

Core receipt kinds

Kind	Written when
`hire_started`	The hire starts; `body` carries the full terms
`enrolled`	The hub accepted the enrolment
`hire_ended`	The hire ends; `body` carries the final statement, `purpose` the reason
`hub_unreachable`	A call to the hub failed after its retries
`work_item_recorded`	A finding was kept locally at level 0
`work_item_reported`	A work item was reported, or a deferred one finally sent; `body` carries what was found and decided
`work_item_deferred`	A report failed and was kept for the next attempt
`work_item_updated`	A later report added a note to an open item, or reopened a closed one
`work_item_update_failed`	A note could not be sent
`approval_requested`	The hub was asked to decide a declared action
`action_taken`	A declared action ran; `body` carries the command and its result, `purpose` names the approver
`action_rejected`	A person rejected a declared action
`action_refused`	An approved action was not run because the limits changed
`feedback_applied`	An outcome was applied
`source_problem`	A source could not be read

A profile or extension MAY add kinds, named `<extension id>/<kind>` (section 10.6).

7.4 Outcome

An outcome is a person's verdict on one work item: was it a real problem or a false alarm, and did the action help. It is how the capability learns, and how its track record is built from judgements rather than from its own opinion of itself. The hub delivers outcomes in the inbox; the capability applies each one exactly once.

Example. `{"ref": "wi-5516e077e5", "outcome": "real", "action_worked": true, "note": "pool too small after the 2.4.1 deploy"}`.

Member	Type	Required	Meaning
`ref`	string	Yes	The work item
`outcome`	string	Yes	`real` or `false_alarm`
`action_worked`	boolean or null	No	Whether the declared action helped
`note`	string	No	Free text
`hub_ref`, `hub_status`	string	No	The hub's own reference and status for the item, for example a ticket number and `resolved`

An outcome is a person's judgement. A hub MUST send one only when a person made it, and only once the item is closed or resolved: by recording a verdict on it, by marking it a false alarm by the hub's own means, or by resolving it as a real problem. An item that is merely closed, with no judgement, produces no outcome. The capability MUST apply each outcome once and write `feedback_applied`. An outcome carries `ext` but not `crit`; so does a statement.

7.5 Statement

The statement is the capability's account of the hire, built from its own receipts: how much it found, how much it reported, how often people judged it right, and what actions ran. The capability sends one at the end of every hire, and may send interim ones. Because every number in it can be recomputed from the receipts, a hirer can check it, and because it is the capability's own account, it is what a registry can turn into a track record across hires.

Example. At the end of the trial: 1 finding, 1 reported, 1 judged, 1 real, 0 false alarms, `share_real` 1.0, actions `{"done": 1}`.

Member	Type	Required	Meaning
`hire_id`	string	Yes	The capability's own identifier for the hire, the one in its `hire_started` receipt. The hub ties the statement to the hire by the credential, not by this value
`kind`	string	Yes	`task`, `trial` or `standing`
`reason`	string	No	Why the hire ended (final statements)
`track_record.findings`	integer	Yes	Things the capability noticed
`track_record.reported`	integer	Yes	Work items reported
`track_record.judged`	integer	Yes	Work items with an outcome
`track_record.real`, `false_alarms`	integer	Yes	By outcome
`track_record.share_real`	number or null	Yes	`real / judged`, or null when nothing was judged
`track_record.actions`	object	Yes	Count of declared actions by result: `done`, `dry_run`, `failed`, `rejected`, `refused`, and `awaiting` for those not yet decided. Actions only suggested in a report are not counted

A statement sent before the end (`final` false in the binding) is a progress report the hub may show to people; the final one is the record of the hire. The statement SHOULD be checkable against the receipts: each count can be recomputed from them.

7.6 Declared actions file

The declared actions file is the complete list of things the capability may ever do. The provider may propose it; a person at the hirer edits and approves it before the hire; the capability reads it and never writes it. It is deliberately a file a person can read in one sitting, with a title and a plain statement of whether each action can be undone, because the person approving it is taking responsibility for everything in it.

Example. Two actions: `restart-api`, a rolling restart, reversible, allowed to run alone at most twice a day; and `clear-tmp`, which deletes files, not reversible, so it always waits for a person.

Member	Type	Required	Meaning
`actions[].id`	string	Yes	Lower case letters, digits and hyphens, at most 64 characters. This is narrower than an `action_id`, which names one request to run an action
`actions[].title`	string	No	For people
`actions[].reversible`	boolean	Yes	True only when the action can be undone or safely repeated
`actions[].auto_allowed`	boolean	No (default false)	May run without a fresh approval at level 3
`actions[].max_per_day`	integer	No (default 0)	The most automatic runs in one UTC calendar day

How an action is carried out is profile-defined. The Resident profile adds `match`, `services` and `command` (profile document, section 5). An action that is not in this file does not exist for the capability.

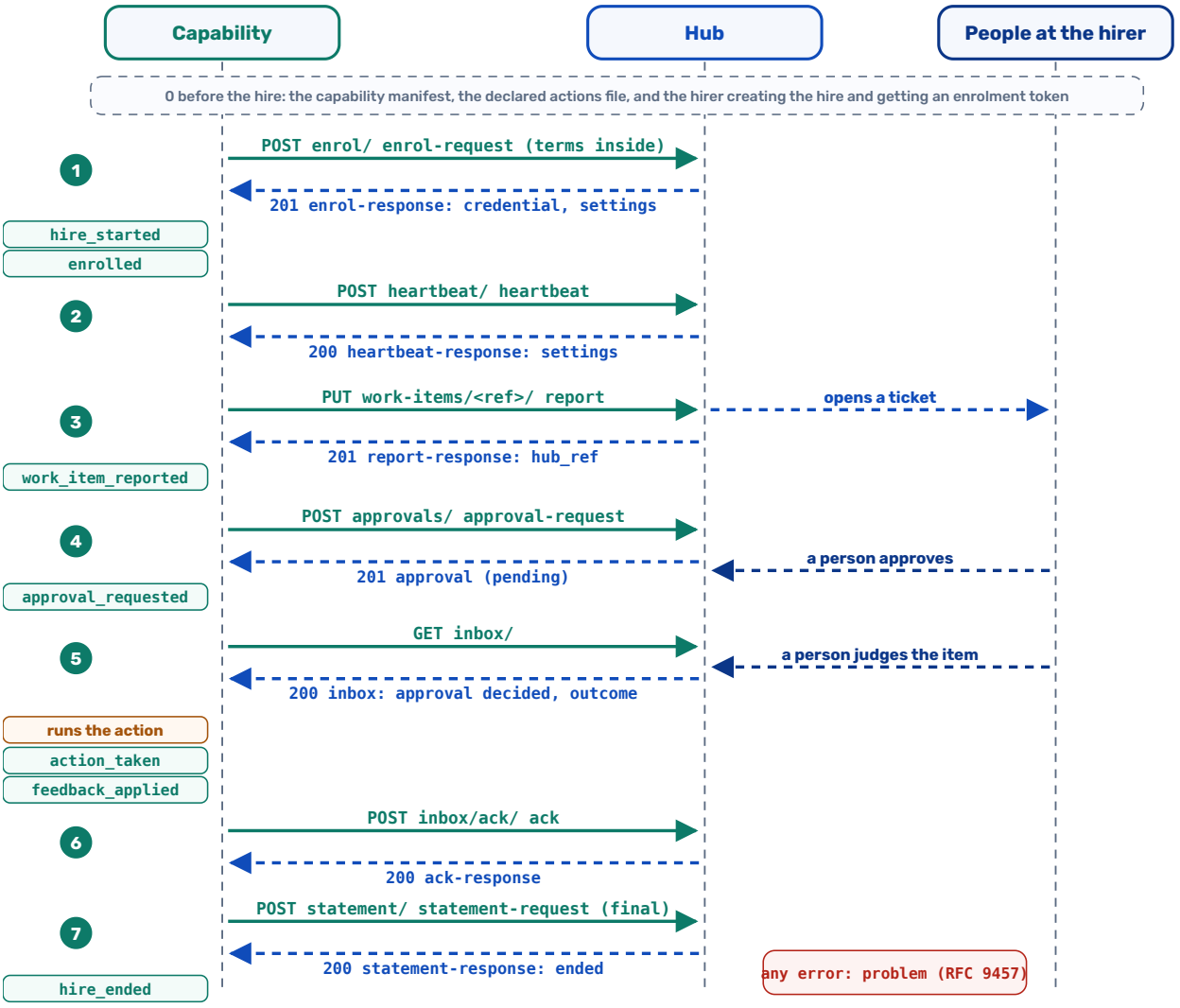
Rationale. These six documents are the minimum a hirer, an auditor and a registry each need: what was offered, what was agreed, what could ever be done, what happened, how it was judged, and the summary. Everything else is profile or extension data.

---

8. Operations

Operations are the seven things a capability does with its hub during a hire. They are described here in the abstract, because the same seven can be carried over HTTP, a message queue or anything else; a binding gives them a concrete form, and the Resident profile's HTTP binding is the first. The diagram shows one whole hire over that binding, numbered as on the examples page.

One hire over the HTTP binding



The seven calls of the Resident binding in the order they happen, with the schema name on each arrow and the receipts the capability writes on the left. Steps are numbered 0 to 8, as in the walkthrough. Open full size

Operation	Direction	Purpose	Idempotent
Enrol	capability to hub	Start the hire, agree version, profile and terms	No (one-time token; see the replay rule in section 6.2)
Heartbeat	capability to hub	Health, counters, `receipt_head`; receive settings	Yes
Report	capability to hub	Create or update a work item by ref	Yes, by ref
Request approval	capability to hub	Ask for a decision on a declared action	Yes, by `action_id`
Collect inbox	capability to hub	Get decided approvals, outcomes, a release request	Yes (read)
Acknowledge	capability to hub	Confirm inbox items were applied	Yes

Operation	Direction	Purpose	Idempotent
Send statement	capability to hub	Statement, and end the hire if final	Yes

"Idempotent" means that doing it twice has the same effect as doing it once, so a capability that is not sure whether a call arrived can simply make it again.

8.1 Direction

The capability always starts the exchange. The hub MUST NOT need to open a connection to the capability. Anything the hub wants to say waits in the inbox. This is what lets a capability run inside a locked-down network with no inbound port.

8.2 Delivery

1. An inbox item MUST stay available until the capability acknowledges it.
2. Applying the same item twice MUST have no further effect.
3. The capability SHOULD acknowledge items it cannot match (for example after losing its state), so they do not repeat for ever.

8.3 Retries

The capability SHOULD retry an operation after a network error, a 5xx answer, or a 408 or 429 answer (or the binding's equivalents), with growing delays. It MUST NOT retry after any other 4xx answer without changing the request, except a 401, which section 8.6 covers.

8.4 When the hub is unreachable

The capability MUST keep working within its terms, keep reports it could not send, and send them when the hub is back (section 5.2). After its retries are spent it writes `hub_unreachable`.

8.5 Repeating a final statement

A hub MUST accept the same final statement sent again after the hire ended, and answer as if it were the first, so that a capability which lost the reply can finish. A different statement after the end MUST be refused with `conflict`.

8.6 When the credential is refused

A 401 answer (or the binding's equivalent) can be a passing fault as well as a revoke. On the first 401 the capability MUST pause (one `poll_seconds` is RECOMMENDED) and try once more. On a second consecutive 401 it MUST stop calling in and end the hire locally (section 5.1), writing `hire_ended` with the reason. Any successful call in between resets the count.

8.7 Errors

Every error MUST be problem details (RFC 9457) with media type `application/problem+json`, containing `type`, `title`, `status` and `detail`. Problem types are URIs; SaaCIP 0.1 defines these, each as `https://thesoftwarelens.com/saacip/0.1/errors/#<name>` (listed at /saacip/0.1/errors/):

Name	Status	Meaning
`invalid-request`	400	The request does not match its schema; `errors` lists the members

Name	Status	Meaning
`unsupported-version`	400	No offered version is supported; `supported` lists them
`unsupported-profile`	400	The profile is not supported; `supported` lists them
`critical-extension`	400	A critical extension is not understood; `extensions` lists them
`unauthorised`	401	The credential is missing, unknown or revoked
`forbidden`	403	This hire may not do that (for example ask for approval at level 1)
`enrolment-refused`	403	The hub will not accept this capability for a reason of its own
`not-found`	404	No such item for this hire
`conflict`	409	The request conflicts with the hub's record: terms that disagree with the hire, or a changed final statement
`too-large`	413	The request is too large

Clients MUST act on `status` and `type`, never on the wording of `detail`.

Example. A capability that speaks only a future `saacip/0.3` receives `400` with `type` ending `#unsupported-version` and `supported: ["saacip/0.1"]`, and knows exactly why without parsing any sentence.

---

9. Autonomy and declared actions

Autonomy is how much the capability may do without a person. It is set by the hirer in two places that work together: a ceiling in the terms that can never be exceeded, and a live level at the hub that can be turned down at any moment. New hires start low and earn their way up, step by step, as the hirer sees the results.

9.1 Levels

Level	Name	The capability may
0	Observe	Record findings locally; report nothing
1	Report	Report work items; take no action
2	Act with approval	Take a declared action after a named person approves it
3	Act within limits	Take a declared action without waiting, only if it is reversible, marked `auto_allowed`, and within `max_per_day`; otherwise ask, as at level 2

The effective level is the lower of `autonomy_ceiling` and the hub's current `autonomy_level`.

Example. A hirer starts a new capability at level 1 for a week, reads the reports, then raises it to 2 at the hub so it can ask before restarting a service. After a month of good approvals, on a later hire

whose terms allow a ceiling of 3, it sets the level to 3 with only `restart-api` marked `auto_allowed` and `max_per_day: 2`, so that one action runs alone and everything else still asks.

9.2 Rules for actions

1. Actions MUST come only from the declared actions file the hirer approved.
2. A language model MAY choose one declared action from a list of candidates. It MUST NOT write commands, parameters outside an allowlist, or new actions.
3. Text from sources that reads like an instruction MUST NOT influence which action is chosen, and SHOULD be flagged in the report.
4. Actions MUST run without a shell, as argument lists, with placeholders filled only from an allowlist.
5. Reading and acting MUST use separate credentials.
6. New hires SHOULD start with `dry_run: true`.

9.3 The daily limit

`max_per_day` counts the automatic runs of one action, whatever their result (`done`, `dry_run` or `failed`), since 00:00 UTC of the current day, from the capability's own records. When the count reaches the limit, the next candidate waits for approval. An action without `max_per_day` never runs by itself.

Rationale. Four levels let trust grow in steps a hirer can see and reverse. Keeping the model to choosing from a list means a prompt injection can at worst pick the wrong approved action, which the levels and limits still control. A UTC day is the same for every instance and every auditor.

---

10. Extensibility

A protocol that cannot change dies, and one that changes carelessly breaks everyone who built on it. This section is the set of rules that lets SaaCIP grow: new versions, new profiles, and data anyone may add, without an old implementation misreading a new document.

10.1 Versioning

1. Versions are `saacip/<major>.<minor>`.
2. A **minor** version MAY add optional members, operations, profiles, receipt kinds and problem types. It MUST NOT change the meaning of anything that exists or make an optional member required.
3. A **major** version MAY change anything. Major versions are negotiated at enrolment (section 6.2) and a hub MAY support several at once.

10.2 Must ignore

A receiver MUST ignore members, receipt kinds, inbox item fields and problem members it does not understand, and MUST keep them when it stores or passes a document on. This lets a newer sender work with an older receiver.

10.3 Must understand: `crit`

A document MAY list in `crit` the extension identifiers the receiver must understand to process it safely. A receiver that does not understand every listed extension MUST reject the document (at enrolment, with `critical-extension`). `crit` MUST only list extensions that appear in `ext`.

10.4 Extension data: `ext`

Extension data MUST go in an `ext` object, keyed by extension identifier (section 4.5). Extensions MUST NOT add top-level members to a document, and MUST NOT change the meaning of core members.

```
ext:
  com.example.support:
    hours: business
    timezone: Europe/Dublin
crit: []        # nothing here is required to understand the document
```

Example. A provider adds its support hours to its manifest under `com.example.support`. A hub that has never heard of that extension ignores it and the hire proceeds. Had the provider listed it in `crit`, the hub would have refused the enrolment with `critical-extension`, which is the right result when the extension is something the hub must act on.

10.5 New profiles

A profile MUST state: its name; what makes a capability of this kind; the binding it uses; any added documents, operations, states (as sub-states of a core state, never new top-level states), receipt kinds and problem types; its operating rules; and how conformance is checked. Private profiles use reverse domain names. The Resident profile document is the model to follow.

10.6 Registries

This document keeps the registered values for version 0.1. New values are added by a minor version; anyone can use private values named by reverse domain name (for receipt kinds, `<extension id>/<kind>`).

Registry	Registered values
Profiles	`resident`
Hire kinds	`task`, `trial`, `standing`
Hire states	`pending`, `active`, `released`, `ended`, `revoked`
Work item states	`found`, `recorded`, `deferred`, `open`, `closed`, `applied`
Declared action states	`proposed`, `suggested`, `awaiting`, `approved`, `rejected`, `refused`, `executed`
Outcome values	`real`, `false_alarm`
Action results	`done`, `dry_run`, `failed`, `rejected`, `refused`, `awaiting`
Problem types	section 8.7
Receipt kinds	section 7.3

Rationale. The same two rules, must ignore and must understand, are what let many internet formats grow for decades (JSON Web Signature's `crit` header is the direct model for `crit`). Reverse domain names give everyone a namespace without a central office.

---

11. Security considerations

Threat	Requirement
Stolen enrolment token	One use (plus the short replay window for the same key); expires (24 hours RECOMMENDED); bound to one hire and tenant; only a hash stored; a refusal of any kind does not use it up
Stolen credential	At least 256 bits of randomness; only a hash stored; works for one tenant; revocable at once by the hirer
Hub raising a capability's power	Hub can only lower the level (section 6.3); terms are authoritative (section 6.2)
Prompt injection through sources	Section 9.2, rules 2 and 3; source text fenced as data wherever a model sees it
Command injection	Argument lists, no shell, allowlisted placeholders (section 9.2, rule 4)
Tampered evidence	Signed, chained receipts; checkpoints at the hub; key in the hirer's key service in production
Duplicate or replayed reports and outcomes	Idempotency by ref and by inbox acknowledgement (sections 5.2, 8.2)
HTML or script in reports	Escaped by the capability and cleaned again by the hub with an allowlist
Inbound attack surface	None in the Resident profile: outbound only
Supply chain	A signed release with a software bill of materials; nothing installed at run time
Transport	HTTPS only, except explicitly enabled local testing

Tokens and credentials SHOULD carry a recognisable prefix (`enrol_` for enrolment tokens, `agent_` for credentials) so that leaked values can be found by secret scanners.

12. Privacy considerations

1. Personal data in source records SHOULD be removed or masked at the source, before storage, explanation, reporting or receipts.
2. `data_leaving` in the terms MUST describe everything that leaves the environment, including to any model.
3. Receipts and statements SHOULD contain counts, references and cleaned text, not raw records.
4. The hirer is usually the controller of the data; the provider's role, and the lawful basis, are for the parties and their advisers to settle.

---

13. Conformance

Class	Conforms when	How to check
Capability	It meets sections 4 to 10 and one profile	Every document and message it produces validates against its schema; its receipts verify as a chain; its lifecycles follow section 5

Class	Conforms when	How to check
Hub	It meets sections 5.1, 6, 8 and 11 and the binding	Every message and error it returns validates against its schema; it refuses what sections 6.2 and 8 say it must refuse
Documents	They validate against the schemas	`validate.py <schema name> <file>`

A conformance claim MUST name the version and the profiles, for example "SaaCIP 0.1, Resident profile, capability". A claim is the implementer's own; this document names no products.

14. Relation to other protocols

Protocol	Carries	How SaaCIP relates
MCP	An agent calling tools in a session	A capability can read through MCP servers, and can offer a read-only MCP server so assistants ask it instead of reading raw data
A2A	Agents talking to other agents	A capability can be reached over A2A; SaaCIP adds terms, evidence and statements around the hire
Payment protocols (for example AP2, ACP, x402)	Paying for what an agent bought	A statement or a receipt can be the record a payment refers to
OpenTelemetry, logging standards	Telemetry	Possible sources for a Resident connector

---

Appendix A. Profiles, and the Resident SaaC profile

The core says what every hire has in common. It does not say where the capability runs, how it reaches its data, or which transport carries the seven operations, because those differ between kinds of capability. A **profile** answers them for one kind, and names the binding it uses.

The first profile is **Resident SaaC (rSaaC)**, profile name `resident`: a capability that runs inside the hirer's own environment (their cloud account, cluster or data centre), goes to the data instead of taking the data out, calls the hub over HTTPS and never accepts a call in, and keeps its receipts, key and state on the hirer's own storage. It is defined in its own document, SaaCIP 0.1: the Resident SaaC profile, which adds the connector manifest and the record, the Resident form of a declared action, the HTTP binding for the seven operations with a schema for every request and response, and the operating rules. The examples in this document and on the examples page all use the Resident profile.

A second profile, for capabilities run by the provider and called per task from a registry, is an open point (Appendix E).

Appendix B. Design rationale

#	Decision	Why	Considered instead
B.1	JSON on the wire; YAML only where people write	Every tool reads JSON; JSON can be canonicalised and signed; YAML has many ways to write one value	YAML everywhere; Protocol Buffers (efficient, but harder for people to read and to audit)

#	Decision	Why	Considered instead
B.2	Signed, hash-chained receipts kept by the hirer	Evidence the hirer can check without trusting the provider; tampering shows	Provider-held logs (the hirer must trust them); a shared ledger (heavy, and not needed between two parties)
B.3	The capability always calls out	Nothing to open in the hirer's firewall; works behind NAT and in locked-down networks	Webhooks into the capability (an inbound hole in the hirer's environment)
B.4	Idempotency by a ref the capability chooses	Retries are always safe, even across restarts	Hub-assigned ids (a lost reply causes a duplicate)
B.5	Autonomy can be lowered by the hub, never raised above the terms	The terms are the hirer's agreement; a hub fault cannot exceed it	Hub-controlled level (a single point of failure for safety)
B.6	Core plus profiles plus bindings	Capabilities run in different ways; the hire is the same	One monolithic specification
B.7	Must ignore, with `crit` for must understand	Lets new senders and old receivers work together, without silent misunderstanding	Strict schemas everywhere (every change breaks someone)
B.8	Problem details for errors	A standard, machine-readable error shape	Custom error objects per endpoint
B.9	Statements built from the capability's own records	The capability answers for itself; counts are checkable against receipts	Hub-only reporting (loses what was never reported)
B.10	Terms authoritative, hub checks	One place states what was agreed, and it is the one the hirer signs off	Hub decides the kind (two sources of truth, silent drift)

Appendix C. Examples

Complete, valid examples are published at /saacip/0.1/examples/, one per schema (except `common`, which only holds shared definitions). They all tell the one story used throughout this document, and the receipts in the example chain are really signed: the public key in them verifies every line. The examples page walks through the story step by step, showing each document and message at the moment it is produced, with the sequence diagram of section 8 as its map. Each file is checked against its schema with the published validator.

Appendix D. References

Normative

- RFC 2119 and RFC 8174 (BCP 14): key words for requirement levels
- RFC 8259: the JSON data interchange format
- RFC 7493: the I-JSON message format
- RFC 8785: JSON Canonicalization Scheme
- RFC 8032: Edwards-curve digital signatures (Ed25519)
- RFC 4648: base64 encoding
- FIPS 180-4: SHA-256
- RFC 9457: problem details for HTTP APIs

- RFC 9110: HTTP semantics
- RFC 3339: date and time on the internet
- JSON Schema, draft 2020-12
- YAML 1.2.2
- SaaCIP 0.1: the Resident SaaC profile (companion document)

Informative

- RFC 7515: JSON Web Signature (the `crit` header)
- Model Context Protocol (MCP)
- Agent2Agent protocol (A2A)
- Agent Payments Protocol (AP2), Agentic Commerce Protocol (ACP), x402
- JSON Lines
- "From SaaS to SaaC", "The Anatomy of a Capability" and "SaaCIP: the SaaC Interoperability Protocol", The Software Lens

Appendix E. Open points

1. **A second profile** for capabilities hired from a registry and run by the provider, called per task.
2. **More signals.** Whether `changes`, metrics and traces become required for some kinds of capability.
3. **Track records across hires.** A shared way to publish a capability's track record without sharing any hirer's data.
4. **Stewardship.** Who looks after SaaCIP as it changes, and how others take part in later versions.
5. **Receipts for auditors and insurers.** Whether one receipt format can serve them across industries.
6. **A media type** of its own, if the protocol is adopted widely.

Appendix F. Change log

Version	Date	Changes
0.1 draft	30 October 2026	First public draft: core, Resident profile (companion document), HTTP binding, JSON Schemas, examples, validator, lifecycles, extension rules, problem details. Licence: CC BY 4.0 for the text, Apache 2.0 for schemas, examples and validator

Comments are welcome. The views here are the author's own.

Important notice. This specification is part of my ongoing research into the Agentic Economy and reflects my thinking at the time of writing. It is shared in good faith, for information and discussion. It is not legal, regulatory, technical, financial or other professional advice, and it is not a verdict on how the EU AI Act or any other law applies to any organisation, system or situation. Laws, guidance, technology and my own research all move on, and others may reach different conclusions. The examples, designs, diagrams and technology choices show one possible approach. Before relying on, adapting or applying any part of it in your own situation, you must test and verify it, design it for your own architecture, and take independent expert advice, including from qualified lawyers and from your own IT, security and data protection specialists. Any decision to act on it, and its consequences, rest with you. To the fullest extent permitted by law, the author accepts no responsibility or liability for any loss, damage or consequence arising from any use of, or reliance on, this specification, which is provided without any warranty of accuracy, completeness or fitness for a particular purpose. Reading it does not create a professional or advisory relationship with the author. The views expressed are the author's own.