SPECIFICATION · VERSION 0.1 · DRAFT FOR DISCUSSION

MICP

Managed Integration & Credential Protocol

A way to describe an integration so that a machine can use it without ever holding the key.

25 November 2026

Status: proposed · reference implementation in progress

Comments welcome before publication of 0.1

ABSTRACT

Agents can call tools, but they still cannot use integrations safely. To let an agent read a mailbox or create a customer, someone ends up putting a key in an environment variable, a configuration file, or — in the worst case — the prompt itself. This document describes MICP, a protocol for declaring an integration as a self-contained pack: how to obtain its credential, what each operation accepts and returns, which secret it uses, who may use it, what gets recorded, and how to avoid performing the same effect twice. The pack contains references to secrets, never secrets. A secret is decrypted on the server, for the duration of a single call, and never reaches the model, the log, or the sandbox. The result is an integration that any agent — or any ordinary deterministic program — can use, without ever being trusted with a key.

1 A familiar morning

Someone wants an agent to send a weekly summary by email. The demo takes ten minutes. The key is pasted into a text box, because that is the fastest way to make it work, and it does work. Three weeks later nobody remembers that the key is now in a chat transcript, in the vendor's request logs, in the run history, and in a backup somewhere.

This is not carelessness. It is what happens when a system has no place to put a secret.

The pattern repeats with every integration. Give an agent access to Stripe and someone must decide whether it can refund. Give it access to a database and someone must decide whether it can write. Give it access to a mailbox and someone must decide whose mailbox. In practice these decisions are rarely made explicitly; they are implied by whoever wrote the code last, and they are invisible six months later when an auditor asks what the agent did with the credentials.

Meanwhile the failing is not in the model. Modern models are competent at choosing a tool and filling its arguments. The failing is in the layer underneath: an integration is a piece of software with a key inside it, and there is no agreed way to describe that piece so that it can be checked, limited, versioned, tested and audited.

Three failures worth naming

The key travels. Secrets end up in environment variables, shell histories, CI settings, screenshots, spreadsheets. Each place is a copy that nobody tracks.

Nobody defined what is allowed. An integration given to an agent is usually given whole. There is no sentence in the system that says “this agent may send email but not read the mailbox”, because there is no place to write that sentence.

Effects are not repeatable. When a long job fails halfway and is resumed, the steps that already succeeded run again. Sometimes that means a duplicate record. Sometimes it means two invoices.

2 What already exists, and what it leaves open

It is worth being precise here, because the space is crowded and it is easy to propose something that already has a name.

Tool-calling protocols define how an agent discovers a tool and invokes it. They answer the question “what can this thing do, and how do I call it?”. They deliberately say nothing about where a credential lives, who is allowed to use it, or what should be recorded afterwards. That silence is correct: it is a different problem.

API description formats describe an HTTP interface: paths, parameters, schemas, error codes. They are excellent at describing an API and useless at describing an author-

isation model, because an API does not have one — a caller either has a token or does not.

Workflow tools combine both and add a user interface. Their integration catalogue is a set of nodes with a form, and the knowledge of how to obtain the credential lives in their help pages. It works for a human clicking through a flow. It assumes a human is there, wiring each step, and it does not produce something another system can consume.

Between these three, a gap stays open. Nobody defines the object “an integration usable by an autonomous system”: what it accepts, what it returns, which secret it uses, who may call it, what is recorded, and how a duplicate is prevented. That object is what MICP describes.

> ONE SENTENCE
>
> A tool-calling protocol tells a machine how to knock on a door. MICP describes the door, who holds the key, what is allowed to pass through, and what is written in the visitors' book.

3 The idea

An integration is described in a single file, called a **pack**. A pack is data, not code. It can be read by a person, checked by a program, compared between versions, and sent to someone else without carrying a credential.

Inside the pack, each operation is a **cell**. A cell is the smallest thing that can be authorised, tested, and audited. It has a name like `email.enviar` or `stripe.cliente_criar`, a declared input, a declared output, and a mapping from the input to exactly one HTTP request.

The definition worth remembering is this:

```
a cell is a pure function whose environment holds the secret.

  typed input → validation → secret resolution → one deterministic request
              → response validation → audit record → typed output
```

Read that again, because everything else follows from it. The function is pure in the sense that matters: the same input, at the same version, produces the same request. It does not decide, improvise, retry with a different URL, or read anything it was not given. The only thing it has that the caller does not is the secret, and it uses that secret for a few milliseconds, inside the server, and then forgets it.

Why cells and not “a Slack integration”

The temptation is to describe whole services. It feels natural to say “the Slack integration”, and then the agent must choose both the service and the operation, and the permission covers everything. Splitting by operation costs a little more description and

buys three things: an operation can be granted without granting its neighbours, an operation can be fixed without touching the others, and an operation can fail its test alone.

4 What a pack contains

A pack has two halves. The first half is how to get the credential. The second half is what to do with it. Both are needed; most systems describe only the second and leave the first to a help article.

```
{
  "micp": "0.1",
  "integracao": {
    "apelido": "email_elizeu",
    "descricao_modelo": "Sending account of Eliseu. Use when the request mentions 'Eliseu's
email'.",
    "auth": { "esquema": "basic", "segredo": "$cofre:email_elizeu.app_password" },
    "config": { "email": "eliseu@example.com" },
    "destinos_permitidos": ["smtp.gmail.com", "accounts.google.com"]
  },
  "provisionamento": { "variantes": [ ... ] },
  "celulas": [ ... ]
}
```

The field that matters most is the quiet one: `auth.segredo` is not a key. It is an address — `$cofre:email_elizeu.app_password` — pointing at a vault entry. A pack with a real key inside is rejected by validation, which means a pack can be shared, published, reviewed and handed to a colleague without anyone having to trust the channel it travelled through.

`destinos_permitidos` is the other quiet field, and it is a security boundary rather than a convenience. A pack declares the hosts it will contact; an executor that finds itself sending a request anywhere else refuses to run. Without this clause a malicious pack is an exfiltration tool with good documentation.

5 Getting the credential

Here is the part that every other format leaves out, and the part that consumes the most human time in practice: obtaining the key.

The comfortable case is a service that gives you a token on a page. The uncomfortable case is a service that requires you to create a project, enable an API, create an identity, download a JSON file, and share a document with that identity before anything works. For a person who does this once, it is a confusing hour. For a product that asks a thousand people to do it, it is the difference between adoption and abandonment.

So the pack declares the procedure, in the same way it declares an operation:

```
"passos": [
  { "n": 1, "acao": "abrir", "url": "https://console.cloud.google.com/projectcreate",
    "instrucao": "Create a project", "obtem": "project_id" },
  { "n": 2, "acao": "abrir", "url": "https://console.cloud.google.com/apis/library/
gmail.googleapis.com",
    "instrucao": "Enable the Gmail API", "espera": "120s",
    "verificar": "GET /v1/projects/{project_id}/services/gmail.googleapis.com" },
  { "n": 3, "acao": "criar_automatico",
    "api": "iam.googleapis.com/projects/{project_id}/serviceAccounts",
    "instrucao": "The executor creates the service account", "obtem": "service_account" },
  { "n": 4, "acao": "criar_automatico",
    "api": "iam.googleapis.com/.../serviceAccounts/{account}/keys",
    "instrucao": "The executor creates the key and stores it in the vault",
    "obtem": "key_json", "segredo": true } ]
```

A step either opens a page and waits for the human, collects a simple value, performs an API call on the human's behalf, or waits for a propagation delay and then verifies it. Each step declares what it produces and how to check it, so a mistake is caught at step two rather than at the first real execution.

Notice step three: when the credential can be created by API, the protocol does not ask. It creates. The person never sees the service-account JSON, because there is no reason for them to see it — it goes from the provider's response into the vault.

> PROGRESS
>
> The state of a provisioning run is stored: which variant was chosen, which step is current, which values are already obtained. A twenty-minute procedure can be interrupted and resumed days later. Without this, long setups die halfway, and with them the confidence that the product works.

Variants exist because the same goal has different roads. One provider may offer a five-minute personal password and a twenty-minute service account with different capabilities. The protocol does not choose; it declares both, each with its own trade-off, and lets the person decide with the information in front of them.

6 Where the secret lives

A vault entry is encrypted with a data key; the data key is encrypted with a master key; the master key is not in the database. That arrangement is unremarkable and it is also the one that gets skipped, with consequences that only appear during a restore.

```
secret       → AES-GCM under a per-secret data key
data key     → encrypted under the master key (version recorded in key_version)
master key   → file on the server; separate encrypted backup, stored elsewhere
```

The master key cannot encrypt itself. A database backup that travels alone is therefore worthless for those rows: the ciphertext arrives, and nothing can open it. The backup of the key has to exist separately, and it has to be tested — a vault that has never been opened from a backup is a promise, not a mechanism.

An obvious objection is that the person should hold the key, not the server. It is a reasonable instinct and it fails on a specific requirement: at three in the morning, when a scheduled job runs, nobody is awake to unlock anything. Automation needs the key to be reachable when the owner is asleep. What the owner keeps is everything that matters for trust — the ability to revoke, to restrict, and to see who used what and when — without being required to be present for the work to happen.

Decryption happens once per call, in memory. The plaintext is not written to disk, not placed in a log, not attached to a run event, and never handed to the model. A model that is told to “send the report from Eliseu's email” sees an integration named `email_elizeu` with a description; it does not see, and cannot ask for, the password.

7 Permission, record and repetition

Authorisation is per cell. A grant says that a named agent may use a named operation. Not the service, not a wildcard: the operation. “This agent may send email; it may not read the mailbox” is a sentence the system understands, stores, and enforces — and it is the sentence most teams discover they wanted only after an incident.

The default is permissive within a workspace: connecting an integration makes it available. Restriction is an explicit act. This is a deliberate trade — friction is placed where it is rare rather than where it is daily — and it is honest about its consequence: effective protection comes from the audit trail and from the read/write scope declared on each cell, not from a permission form nobody fills in.

Discovery is filtered by authorisation. When an agent asks what it can do, the answer omits the cells it may not use. Absence is stronger than refusal: an agent that cannot see an operation will not attempt it, will not hallucinate arguments for it, and will not fill its context with schemas it cannot act on.

Every execution is recorded: cell and version, integration alias, agent, run and step, outcome, error code, duration, timestamp. Never the secret; never a request body that could contain one.

Every execution carries an idempotency key. This is part of the contract rather than an option, because the alternative is silent duplication. When a long job is resumed after a failure, the steps that already succeeded are skipped — but the step that failed will run again, and “run again” must not mean “send a second email”. A repeated key returns the first result instead of performing the effect twice.

8 Beyond agents

Nothing in the design requires a language model. A pack can be read by an ordinary program, a scheduled script, a workflow engine, or a person with a command line. This is not an afterthought; it is a test of whether the abstraction is real.

An integration described this way is a recipe with a locked drawer. A team that distrusts agents can still use it: the operations are declared, the secrets are in a vault, the calls are recorded, and the same guarantees hold. A workflow tool could adopt packs as its node catalogue. A company with an internal platform could generate packs from its own OpenAPI specifications and stop hand-writing connectors. A supplier could publish a pack for its own API, and its customers would not need a developer to wire it.

That portability is the point of writing a protocol instead of a feature. A feature serves one product. A format that describes the integration — independent of who calls it and independent of whether the caller is intelligent — serves every product that has the same problem, which is all of them.

9 Conformance

An implementation is conformant when it passes twelve checks. They are deliberately unglamorous, and each one corresponds to a failure seen in practice.

1. The pack declares the protocol version it follows.
2. Required integration fields exist, and the model-facing description is short enough to be read.
3. Every secret reference is a vault address, and no value in the pack resembles a real credential.
4. Cell identifiers are stable, and versions are integers that only increase.
5. Every input schema is non-empty and uses known types.
6. Every output schema exists — a cell that returns “something” cannot be composed.
7. The mapping draws only on input, configuration and secret; no other variable is reachable.
8. The transport is valid and consistent with the mapping.
9. An allowlist of destinations exists, and every host in the pack appears in it.
10. A verification call exists. An integration that cannot be tested is not conformant.
11. Provisioning steps are coherent: every value obtained is used, and fragile variants declare a fallback.
12. No value matches a known credential pattern.

The twelfth check contradicts the purpose of the document on purpose. It exists because the first thing people do, when a format feels convenient, is paste a real key into it “just to test”.

10 What this does not solve

Honesty about limits is cheaper than discovering them in production.

Three transports only — HTTP, SMTP, IMAP. A service requiring multipart uploads, WebSocket streaming or a proprietary protocol needs a new transport, and that is a decision with a review, not a configuration.

Provisioning stays partly human. Many providers do not expose an API for creating credentials. The protocol guides, validates and resumes; it cannot click for you on a console that has no API.

A centralised master key is a single point of trust. A per-user key is possible and incompatible with unattended execution; it belongs as an organisational variant, not as the default.

Version coexistence is specified in intention, not in experience. What happens to a running automation when a cell moves from version three to four is a policy question this draft states but has not lived through.

Trust between organisations is unresolved. A “verified” badge means whoever signed it is trusted, and building reputation between strangers is a different project.

11 Roadmap

The version numbers describe commitments, not hopes.

0.1 — the contract

Pack format, cells with input and output contracts, vault references, grants per cell, audit records, filtered discovery, conformance checks, and a validator that runs them. Three reference packs, one of them canonical (email with two provisioning variants; Telegram for the simplest possible case; a read-only payment integration for the case where scope matters).

0.2 — the guarantees

Idempotency keys end to end, versioned cells with declared deprecation, adapters that expose cells through the three transports a caller might use (tool-calling protocols, HTTP, and a command line inside a sandbox with a short-lived pass instead of a credential), and provisioning with resumable state.

0.3 — the ecosystem

Pack generation from published API specifications, so a specification becomes a pack without a developer. Signed packs and a review checklist, so a third party's pack can be accepted without reading every line. Import of existing community node definitions where licensing allows.

12 Status of this implementation

This document was written from a working system rather than ahead of one, which makes it possible to separate what exists from what is intended.

ELEMENT	STATUS
Pack manifests, operations as resource × action	implemented
Input and output contracts, mandatory verification call	implemented
Grants per agent, per-call audit	implemented
Discovery filtered by authorisation	implemented
Pack generation from OpenAPI specifications	implemented
Vault with envelope encryption and versioned master key	in progress
Provisioning with persisted progress	in progress
Idempotency per execution	0.2
Adapters (tool-calling protocol, HTTP, command line)	0.2

The last two rows are the ones that decide whether the protocol is usable in production or merely correct. A format that describes integrations perfectly but duplicates an effect when a job resumes has solved the easy half.

13 Why it is worth writing down

Every organisation that connects software to other software rebuilds the same four things: a way to describe the connection, a place to keep the key, a rule about who may use it, and a record of what happened. They rebuild them differently, badly, and privately, and then they rebuild them again when the person who wrote the first version leaves.

The value of a protocol is not elegance. It is that the fourth team to solve this problem inherits the decisions of the first three — including the decisions about what not to allow.

MICP is small on purpose. It describes an integration, not a platform; a contract, not an implementation; a place for the key, not the key itself. If it works, it disappears: people stop talking about how they connected the agent to the mailbox, because the connection is declared, the key is in the vault, and the record says what happened.

MICP · Managed Integration & Credential Protocol · version 0.1 · 25 November 2026 · This is a draft circulated for discussion. Criticism is more useful than agreement.