

Plugins

From manifest to signed package

Technical reference for the sandbox, the capability registry, the signing chain and the four execution paths, followed by a step-by-step authoring guide.

Contents

UNDERSTANDING

1.	The mental model	3
2.	The execution model	4
3.	Anatomy of a plugin	5
4.	The four execution paths	6
5.	The bridge protocol	7
6.	Contract and implementation	9

EXTENDING

7.	Contributions	11
8.	Where a panel opens	12
9.	Theme, language and snapshot	13
10.	Storage and bundled assets	14
11.	The R engine	16

GOVERNING

12.	Capabilities and consent	18
13.	Signing, author identity and provenance	19
14.	The access log	21
15.	Conflicts, dependencies and compatibility	22
16.	The manager and the entry points	23

REFERENCE

17.	Catalogue of host methods	25
18.	The manifest, field by field	29
19.	Publishing a plugin	30
20.	The development workshop	31
21.	Four typical plugins	32
22.	Troubleshooting	35
23.	Limits and persistence	37

WRITING

24.	From nothing to a running plugin	41
25.	Choosing your capabilities	42
26.	Writing a panel interface	43
27.	A complete plugin	45
28.	Checklist before publishing	48

Understanding

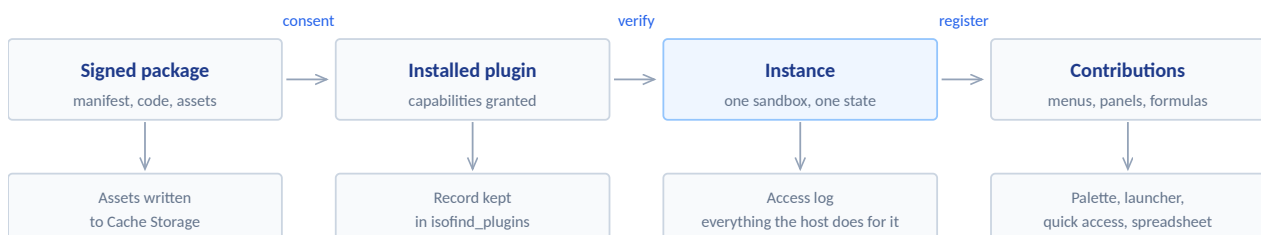
What a plugin is, what it is allowed to touch, and what you may rely on when you write one.

1. The mental model

Six words describe the whole system, provided you do not confuse the first two. The manifest is text the author writes; the granted capability is a decision the user makes. The first asks, the second disposes, and no line of plugin code bridges that gap.

Word	What it is	Where it lives
Package	An <code>.isplugin</code> archive, which is a zip: a manifest at the root, an entry point, optionally an icon and an assets folder. This is the only form a plugin travels in.	On disk, or in the catalogue.
Manifest	What the plugin declares about itself: identity, version, entry point, the capabilities it asks for and the contributions it brings. Text, therefore a request, never an authorisation.	<code>manifest.json</code> , kept verbatim at install time, <code>signature</code> field included.
Capability	A named right, requested by the manifest and granted by box at install time. Whatever was not ticked is refused at runtime.	On the installed plugin's record, in <code>isofind_plugins</code> .
Instance	A running plugin: its sandbox, its granted capabilities, its signature verdict, its live contributions. Two loads of the same plugin share nothing.	In memory, one per loaded plugin.
Contribution	What the plugin adds to the application: a menu entry, a panel, a spreadsheet formula. The host registers it under the plugin's name, and removes it with the plugin.	In the host registries, never in the sandbox.
Sandbox	The iframe the plugin code runs in, without <code>allow-same-origin</code> , therefore on an opaque origin. The plugin gets a whole document there, and nothing else.	Under <code>document.body</code> , permanently, revealed when a panel opens.

The life cycle



Unloading removes the contributions and destroys the instance.
Uninstalling also removes the plugin's storage and its cached assets.

Three ideas that explain most of the design

- 1. The manifest asks, the host decides.** Nothing a plugin writes grants it a right. A manifest declaring an unknown capability sees it ignored, a plugin calling a method whose capability was not ticked gets a refusal, and a panel that carries a dock zone is not thereby excused from `ui:panel`. What protects is never an instruction, but a method that was not answered.
- 2. The plugin shares nothing with the page.** It lives on an opaque origin: no host `document`, no `localStorage`, no cookies, and no request to the local server, which the content security policy closes off independently of capabilities. Everything it obtains goes through a message protocol the host alone arbitrates.
- 3. Confinement and verification answer two different questions.** The sandbox says what a plugin can do; the signature says where it came from and whether its contents are the ones that were signed. Neither replaces the other. A plugin can be perfectly confined and still malicious within its remit, and a plugin signed by a known author remains confined exactly like the others.

2. The execution model

A plugin does not run in the page. That is the decision everything else follows from, and it comes at a price worth knowing before you start writing.

The sandbox is an iframe carrying `sandbox="allow-scripts"`, without `allow-same-origin`. Its document does not come from a URL but from a `srcdoc` the host builds, which is how a content security policy computed for that instance, and the theme tokens in force at load time, get injected into it. Because the origin is opaque, the host itself cannot read the `contentDocument` of the iframe it has just created: the only channel, in either direction, is `postMessage`.

```
default-src 'none';
script-src 'unsafe-inline' 'unsafe-eval';
style-src 'unsafe-inline';
connect-src 'none' | https:      // depending on the network capability and air-gap mode
```

`connect-src` is the only variable term, and it only ever opens onto `https:`, never onto the local origin. A plugin therefore cannot call the IsoFind API directly, whatever capabilities it holds: what it is granted goes through host methods, which are named and counted. In air-gap mode the term falls back to `'none'` whatever capability was requested, and the corresponding box appears greyed out at install time rather than simply missing.

The sample snapshot, and why it exists. Plugins read data synchronously, in the middle of a render. Nothing that crosses an isolation boundary can be. Rather than force every author into a full rewrite in asynchronous style, the host delivers a copy of the samples in the initialisation message, and the plugin reads it as an ordinary array.

That copy travels through no request: it is copied in memory into the sandbox. The server log would therefore never see it, and would suggest the plugin read nothing while it holds the entire database. It is declared explicitly to the access log, and the host waits for the verdict before delivering. A revocation issued afterwards does not claw back what has already gone: it only bites at the next load.

What is deliberately missing

Access to the host DOM. A plugin draws only in its own document. What the application shows around it, modal header, dock tab, close button, belongs to the host. That is what makes it possible to move a panel from one surface to another without the plugin having to know.

Network access to the local server. It has been tried, and it does not work: the content security policy blocks the way out of the sandbox towards the local server, whatever capability was granted. A plugin that needs data goes through [samples:read](#) and the snapshot, or it bundles what it needs.

Two panels of the same plugin on screen at once. A plugin has one sandbox, hence one iframe, hence one rectangle on screen. Opening the second closes the first, explicitly, through whichever surface it was in.

Host execution without saying so. The native path exists, but it is declared in the manifest and shown on the plugin's record. A plugin cannot slip into it quietly, and the user granting capabilities sees what they are consenting to.

3. Anatomy of a plugin

A plugin is a zip archive with the [.isplugin](#) extension. It holds a manifest at the root, and everything that manifest points at.

```
my-plugin.isplugin
├─ manifest.json      // required, at the root, under that exact name
├─ plugin.js          // the entry point the manifest names
├─ icon.png           // optional
└─ assets/            // optional, extracted at install time
    ├─ engine.wasm
    └─ reference.json
```

The manifest is the only file whose name and location are fixed. The entry point, the icon and the assets folder are named by fields, and the field is what counts: assets are written under the declared prefix rather than a hard-coded [assets/](#), because the signer relies on the manifest too. Two different conventions for naming the same files would produce two different hash maps, hence a permanent and inexplicable refusal.

```

{
  "id": "my-plugin",                // stable, used as a key everywhere
  "name": "My Plugin",              // catalogue and consent screen
  "version": "1.4.0",
  "author": "Author name",
  "description": "Short description",
  "category": "analysis",           // analysis | visualization | data | tools | science

  "entry": "plugin.js",             // the extension decides the language
  "capabilities": ["samples:read", "ui:menu", "ui:panel"],

  "contributes": {
    "menus": [ { "id": "open", "label": "Open the panel" } ],
    "panels": [ { "id": "main", "label": "My Plugin", "dock": "right" } ]
  },

  "assets": "assets/",              // written to Cache Storage at install time
  "dependencies": ["other-plugin"],
  "icon": "icon.png",

  "signature": { ... }              // added by signing, never written by hand
}

```

The installed manifest is not rebuilt. At install time some fields are copied flat onto the plugin record for the list and the badges, but the original manifest is kept whole alongside, `signature` field included. That is what, and only what, the verifier recomputes. A record rebuilt from the flattened fields would no longer verify, and the plugin would stand accused of tampering with nothing having changed.

Metadata: English in the catalogue, bilingual at runtime

The `name` and `description` fields and the `contributes` labels are written in English. These are the strings the catalogue, the moderators and the consent screen see, on the same footing as the rest of code meant to circulate. The interface the plugin draws for itself is free, and the shipped plugins pick their language at runtime by reading the application's.

Legacy fields

Three first-generation names are still read, and should no longer be written. `main` stands in for `entry`. `permissions` stands in for `capabilities`. `language` no longer has a reason to exist, the language being derived from the entry point's extension, and it is consulted only when that extension settles nothing.

4. The four execution paths

The same manager installs, loads and unloads four kinds of plugin, which do not run in the same place. Confinement differs; consent does not. All four go through the same capability screen and the same signature check.

Path	Where the code runs	Confinement	What selects it
------	---------------------	-------------	-----------------

Sandbox	In an iframe without <code>allow-same-origin</code> , on an opaque origin. Every interaction goes through the message protocol.	Full. No DOM, no storage, no local network.	The default: a <code>.js</code> entry point.
Python	In the page, under Pyodide, in a dedicated namespace where <code>isofind</code> is bound to a capability holder.	None. The capability barrier gives consent and consistency, not isolation.	A <code>.py</code> entry point.
Native	In the page, through <code>new Function(code)(plugin)</code> . The code receives its own record and installs itself.	None.	<code>"runtime": "isofind-native"</code> in the manifest.
Workshop	Like the sandbox path, but the code is read from a working folder and reloaded on every change.	Full for the plugin; the service reading the folder, however, is not.	A folder opened in the workshop, in the local zone only.

Two paths run in the page, and it shows. Python and native are not confined. They keep signature verification, capability consent and the access log, which answers "where did this come from" and "what did the user grant", but not "what could it reach if it decided otherwise". Loading says so plainly in the console, and the interface marks it. A plugin that does not need to leave the sandbox should not leave it.

The language is derived from the entry point's extension, not from a field. The manifest is sealed, it has no language field, and a declaration contradicting the extension would send someone hunting for a long time over why a `.py` file goes through the JavaScript bridge.

The Python path, in one paragraph

An `isofind` facade is defined once in the shared runtime, aligned on the bridge surface: same methods, same capabilities, two languages. Each plugin gets an instance bound to its own capability holder, and its code runs in a namespace kept alive after loading finishes. Without that, callbacks registered for a menu or a formula would no longer be callable once the first execution ended. Unloading frees the namespace and destroys the retained `PyProxy` objects, failing which the Python runtime would hold the registered functions forever.

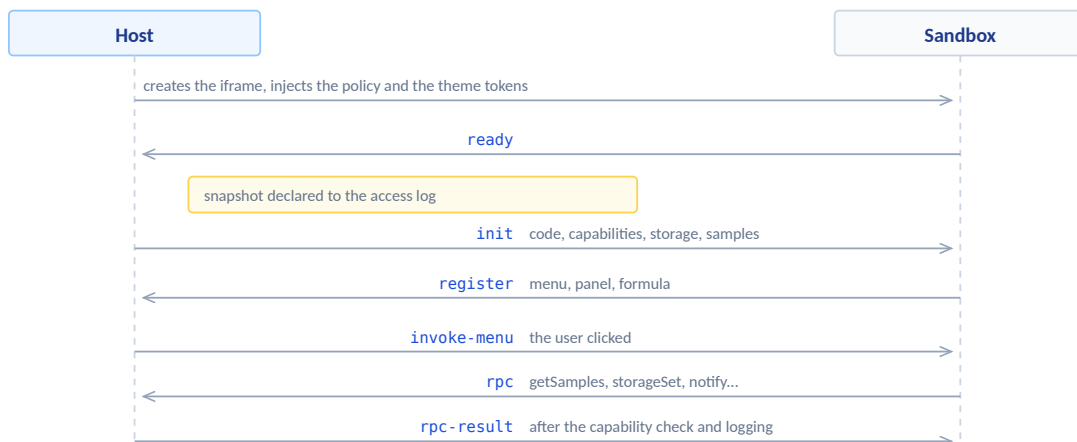
5. The bridge protocol

Twelve messages, and nothing else. Each carries a type in the `t` field and the plugin's identifier. The host recognises the sender by the message's source window, not by what the message declares: a plugin claiming to be another would still be speaking from its own iframe.

To the sandbox	What it carries
<code>init</code>	The plugin code, its granted capabilities, its name, the base URL of its assets, the snapshot of its storage, the snapshot of the samples, and the R engine setting.

theme	The recomputed token sheet, on every theme switch.
rpc-result	The answer to a call, with its identifier, a success flag, and the value or the error message.
invoke-menu	The user clicked one of the plugin's menu entries.
open-panel	A panel must paint itself. Sent both by an explicit open and by the dock factory.
close-panel	The panel is no longer visible. The plugin's state is not destroyed for that.
invoke-formula	A cell asked for a contributed formula, with its arguments.
To the host	What it carries
ready	The sandbox document is ready to receive its initialisation.
register	A contribution: <code>formula</code> , <code>menu</code> or <code>panel</code> , with its spec.
rpc	A host method call, with its name and arguments.
request-open-panel	The plugin asks for one of its panels to be opened.
formula-result	A formula's value, which goes into the cache and triggers a sheet recalculation.
log	A trace, re-prefixed with the plugin's name in the host console.
error	An uncaught error in the sandbox.

One load, from first message to first click



What the bridge does for every call

An incoming call is handled in four steps, in this order. The method is looked up in the host method table: absent from the table, it is refused, which makes refusal implicit rather than declared somewhere else. The capability that method requires is checked against the granted set. An access declaration scope is opened around the execution, naming the plugin, the capability and the method. Then the answer goes back, waiting for resolution if the method returns a promise.

That last point is worth stating. Answering without waiting would send the sandbox a promise serialised as an empty object, and the plugin would believe its computation finished with a null result. It is the hardest failure to diagnose, the one where nothing signals the error.

What the sandbox does on its side

Plugin code is not evaluated at document level. It is wrapped in a function that takes two arguments, and the call looks like this:

```
new Function('isofind', 'plugin', '"use strict";\n' + code)(isofind, context);
```

`isofind` is the surface described in section 17. `plugin` carries what initialisation delivered about the instance: its identifier, its name, the base URL of its assets, the list of granted capabilities, and a notification shortcut. A plugin therefore has nothing to import and nothing to look up in `window`.

That scope is a convenience, and the file's own comment says as much: real isolation comes from the iframe's opaque origin, not from an anonymous function. A plugin writing into `window` would only inconvenience itself.

Two guards frame reception. A message whose source is not the host window is ignored: without that test, a sibling iframe reaches this one through `parent.frames[i]`, which is accessible across origins, and injects code that would then run under the capabilities granted to this plugin, since the host attributes a call to the instance whose window sent it and not to the one that decided to send. And a second `init` is refused: it would reinstall the surface and run different code in the same context, hence with the storage and samples already delivered to the first.

Contributions stay local. A registered formula keeps its function inside the sandbox; only its spec goes to the host, which takes care of the synchronous wrapper, the picker and the help entry. The same holds for a panel: the host receives only its identifier and label, the render function never crosses the boundary.

6. Contract and implementation

An installed plugin is frozen code that quotes names, and those names are resolved at load time, sometimes months later, on a version of the application that did not exist when it was written. The line between what is promised and what is free to move therefore deserves to be explicit.

Object	Contract	Implementation
Manifest	The names of the fields read, the capability vocabulary, the shape of <code>contributes</code> , the fact that the installed manifest is kept verbatim.	The fields copied flat onto the record, the order of the install steps, the wording of the messages.
Bridge	The name and meaning of the twelve messages, the names of the host methods and their capabilities, the fact that an ungranted capability makes the call fail.	The transport, the call numbering, how the snapshot is assembled, the exact wording of error messages.
Contributions	The three kinds, their capability, the fact that a contribution disappears with its plugin leaving no trace.	The surface it lands on, the ordering in lists, the icon, the default dock zone.

Assets	The URL prefix, the fact that they are served offline and never downloaded, the hash key, which is the path inside the archive.	The cache name, the headers set, when extraction happens.
Theme	The list of exposed tokens, and the fact that they exist in the sandbox document from the first render.	Their values, which change with the theme, and the injection mechanism.

What breaks already-installed plugins

Renaming a menu or panel identifier. Quick-access ordering and hiding are stored under a key joining the plugin identifier to the entry identifier. A rename loses the user's preference, with no error, and their entry reappears at the end of the list after having been hidden.

Removing a capability from the known vocabulary. It is not refused, it ceases to exist for everyone: installation drops it from the consent screen with a console warning, and the corresponding calls then fail on a capability the user never had a chance to grant.

Changing the asset prefix or the cache name. Already-installed plugins keep a base URL computed under the old prefix, and their bundled engine finds nothing. The service worker then answers with an explanation rather than a silent 404, but a reinstall is required.

Touching the assets folder after signing. Hashes are recomputed from Cache Storage, that is, from the bytes that will actually be served. A file added, removed or modified makes the map diverge, and the plugin is refused at load time.

Removing a theme token from the exposed list. A plugin using it falls back to its fallback value, when it thought to provide one, or loses its colours. The list is closed for that reason: exposing everything would turn the whole design system into a public surface that could no longer evolve.

What the workshop lets you check, and nothing else did. The manifest declares what a plugin brings, and the code implements it. Two distinct things, therefore two things that can diverge. A contribution declared but never registered may be no more than work in progress. A contribution registered without having been declared is worse: it escapes the consent screen, where the user will never have seen it. The inspector in section 20 puts the two lists side by side.

Extending

What a plugin adds to the application, where it lands, and what it has to work with over time.

7. Contributions

Three kinds, and only three. Each requires a capability, is declared in the manifest, registered at runtime, and removed without residue when the plugin unloads.

Contribution	Capability	What it places in the application
Menu	<code>ui:menu</code>	A quick-access entry in the sidebar, reorderable and hideable, and a line in the launcher. The click comes back to the plugin as a message.
Panel	<code>ui:panel</code>	An entry in the shell registry, the same door core panels use: Panels menu, palette, universal search, docks and centre zone.
Formula	<code>ui:formula</code>	A function name in the spreadsheet, an entry in the formula help, and an entry in the function picker, grouped under a chosen heading.

```
// All three, from the plugin side. The signature is the same in Python,
// in snake case: register_menu, register_panel, register_formula.

isofind.registerMenu('open', 'Open the panel', function () {
  isofind.openPanel('main');
});

isofind.registerPanel('main', 'My Plugin', function (ctx) {
  // The render receives { root, id }: ctx.root is what to populate.
  var root = ctx.root;

  root.textContent = 'Hello';
});

isofind.registerFormula('MY_CALC', function (a, b) { return a * b; },
  { group: 'My plugin', label: 'Product',
    syntax: 'MY_CALC(a, b)', example: 'MY_CALC(2, 3)' });
```

The formula is a special case

The spreadsheet evaluates its formulas synchronously, in the page, while the sandbox only ever answers asynchronously. The host therefore places a synchronous wrapper in the page that serves the cached value when there is one, and otherwise requests the computation and returns `NaN`. The answer fills the cache and triggers a sheet refresh, batched so that several formulas resolving together do not multiply recalculations. Consistency is thus deferred, and exact after one recalculation cycle.

Two guards frame the name. It must match `^[A-Z][A-Z0-9_]{1,30}$`: a formula is a spreadsheet identifier, not an address in `window`, and without a shape constraint a plugin could name any global object in the application. And the name must be free, or already held by a plugin wrapper. The original check only covered

functions: any global that was not one, service objects first among them, could therefore be silently overwritten by a plugin that was otherwise confined.

Contributing a formula is a capability. For a long time it was the only one of the three contributions to require nothing, which made no sense: a formula places a name in the page and an entry in the spreadsheet picker, which is at least as visible as a menu.

8. Where a panel opens

A plugin panel can live in three places: the tabbed centre zone, a side or bottom dock, a floating window. The plugin knows nothing about it and does not need to. It declares a panel, it paints when asked, and the host decides the surface.

The dock zone is declared in the manifest, under `contributes.panels[].dock`, not in the JavaScript call: that call takes three simple arguments and carries no such information. The default is `right`, which is where plugin panels already live.

Value	Effect
<code>left</code>	Left dock, alongside the core panels.
<code>right</code>	Right dock. The default when the field is absent.
<code>bottom</code>	Bottom dock, for a console or a log.

The field does not decide whether the panel exists, only its default zone. Every declared panel enters the registry, with or without `dock`. Without that rule, a panel stayed confined to the modal: absent from the Panels menu, the palette, the launcher, and above all from the centre zone, which is the only surface large enough for a console or an editor. Three hundred and forty pixels of width are not enough for an editor, and that is what had left them in the modal.

One iframe, therefore one visible panel

A plugin has one sandbox. Two of its panels therefore cannot be on screen together: the second retargets the iframe, and the first would become an empty rectangle with nothing to say so. The host therefore closes the other explicitly, and through the surface it is in, centre zone first, dock second. A surface that disappears on a gesture makes sense; an empty surface that stays open gets blamed on the plugin.

The iframe is never moved. It lives permanently under `document.body`, in `position: fixed`, hidden as long as no container targets it. Opening a panel does not reparent it: a synchronisation loop copies the target container's rectangle onto the iframe's coordinates on every frame. Reparenting an iframe reloads it, and the plugin would lose its state on every tab change.

The stacking level follows the surface. A docked panel sits very low, so that a real application modal covers and dims it like the rest of the shell. The plugin's own modal sits just above its own backdrop. A single high level for both cases produced both faults at once: the docked panel floated above native modals, and the iframe fell below its own backdrop in modal mode.

Two races, and how they were lost

The shell is not there yet. It installs when the DOM is ready; a plugin declares itself as soon as its sandbox answers, which depends on its size and what it loads. Light plugins arrived after the shell and registered; those that answered early found the door closed and fell back to the modal, permanently, with no signal beyond a console warning. Registration therefore waits for the shell rather than giving up, and replaying a registration has no side effects.

The panel is not declared at load time. Some plugins declare everything up front, others build their surface on first use, and they are right to. A panel that is requested but not yet registered therefore enters the registry at that moment, rather than falling back to the modal because the ordering did not suit it.

The modal remains, as a last resort. On a page with neither dock nor centre zone, it is the only surface available.

9. Theme, language and snapshot

A plugin does not hard-code colours. It writes `var(--surface-1)` like a native component, and the host guarantees those variables exist in the sandbox document. Since the origin is opaque, the only route is the `srcdoc` at creation time, then a message on every theme switch. Without the initial values, the plugin would render once before the message arrived, hence with no colours at all.

The exposed tokens

Family	Tokens
Surfaces	<code>--surface-1</code> , <code>--surface-2</code> , <code>--surface-3</code> , <code>--surface-inverse</code> , <code>--surface-inverse-soft</code>
Text	<code>--text-primary</code> , <code>--text-secondary</code> , <code>--text-muted</code> , <code>--text-inverse</code> , <code>--text-on-solid</code>
Borders	<code>--border-subtle</code> , <code>--border-primary</code> , <code>--border-strong</code>
Shadows	<code>--shadow-1</code> to <code>--shadow-4</code>
Brand	<code>--brand-primary</code> , <code>--brand-primary-dark</code> , <code>--brand-primary-soft</code> , <code>--brand-accent</code> , <code>--brand-on-primary</code> , <code>--brand-halo</code> , <code>--brand-tint-50</code> , <code>--brand-tint-100</code>
Tones	<code>--tone-{vert, ambre, orange, rouge, bleu, cyan, violet}-{bg, bg-strong, solid, solid-strong, solid-deep, text, border, on-solid}</code>
Legacy	<code>--border-color</code> , <code>--bg-primary</code> , <code>--bg-secondary</code> , <code>--surface</code> , <code>--text</code> , <code>--hover</code>

The list is deliberately closed. The last six names are first-generation aliases, exposed during the transition because already-published plugins use them: cutting them for the sake of tidiness would break what exists. They will go once those plugins have migrated.

A token value comes from a host stylesheet, but it crosses a security boundary. Only what looks like a colour or a shadow gets through, never a brace or a semicolon, which would allow escaping the declaration and writing

arbitrary style into the sandbox document.

Language

The application's language belongs to no plugin, but every plugin reads it to display itself in the right one. It therefore travels with the storage snapshot, under the `isofind_language` key, as soon as `storage:read` is granted. Refusing it would force plugins to fall back to a default, or to throw. Because it arrives in the snapshot, it is readable with no round trip, at the first render, through the compatibility facade.

The sample snapshot

It is delivered with the initialisation message when `samples:read` is granted, and is empty otherwise. Empty, not absent: the plugin reads the key without wondering whether it exists, and removing it would make the plugin throw instead of letting it show an empty list.

A sample arrives with its isotopic measurements attached. The `getAnalyses` method does not query another source: it flattens the samples' measurements, each becoming a row carrying its sample's context. There is no separate analysis store, an analysis being an isotopic measurement.

10. Storage and bundled assets

A plugin that needs to remember something could not: `localStorage` is unreachable from an opaque origin, and that was one of the two reasons the host path survived. Storage is therefore offered by the bridge, under capability.

Call	Capability	Effect
<code>isofind.storage.get(key)</code>	<code>storage:read</code>	Returns the value, or nothing.
<code>isofind.storage.keys()</code>	<code>storage:read</code>	The plugin's keys, without the prefix.
<code>isofind.storage.set(key, value)</code>	<code>storage:write</code>	Writes, or rejects when the quota is reached.
<code>isofind.storage.remove(key)</code>	<code>storage:write</code>	Removes a key.

All four return a promise, since they cross the boundary. Two shortcuts sit on top, `getJSON` and `setJSON`, because nearly every use of storage is hand-serialised JSON. And the compatibility facade exposes a synchronous `localStorage` over all of it, backed by the snapshot delivered at initialisation, described in section 17.

The namespace is derived from the instance on the host side, never from an argument the plugin supplies. That is the only thing that matters here: if the key were freely chosen, one plugin would read another's space, and the partitioning would be decorative. Values live under `isofind_plugin_store:<id>:<key>`, with a ceiling of two megabytes per plugin. The ceiling is not a courtesy: without it a plugin fills the origin's storage, and the whole application stops being able to write, its own preferences included.

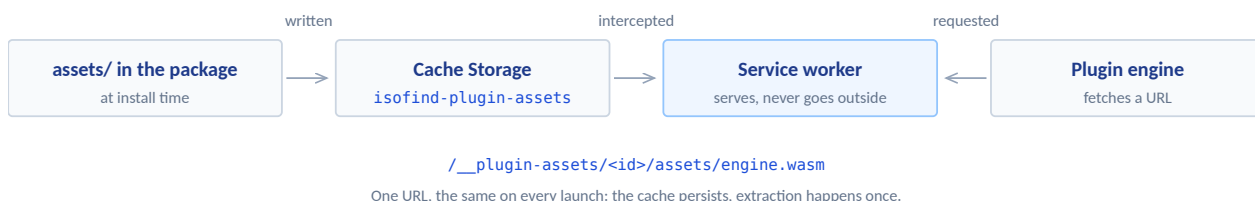
Disabling is not uninstalling. A plugin's storage is purged on uninstall, never on unload. Disabling a plugin makes it silent; it should not make it lose its memory. Conflating the two would destroy the work of a user who merely wanted to put a plugin to sleep.

Bundled assets

A WebAssembly engine loads its binaries by URL. It cannot read a Cache Storage, and you cannot hand it a byte array: it issues a request, and somebody has to answer. That is the job of asset provisioning, which is a platform capability and not an arrangement peculiar to one plugin.

At install time the files in the folder named by `assets` are extracted from the package and written to Cache Storage under virtual URLs prefixed with `/__plugin-assets/<id>/`. The plugin receives that base in its initialisation message, as `assetsBaseUrl`, and loads what it needs from there. There is no `vendor` folder, no placement script, and nothing for the user to do.

An asset's route, from archive to engine



The service worker never goes to the network

There is no fallback. A service worker that, failing to find an asset in cache, went looking for it outside would be an unannounced network egress in software that promises to have none. Missing from the cache, the asset is missing: it answers 404, and says why, in plain words, because a WebAssembly engine handed a silent 404 fails on an incomprehensible compilation error and the user concludes the plugin is corrupt.

This is also what protects the signature. The hashes were verified against the content written to the cache. Serving anything else, even the genuine file fetched from elsewhere, would make verification decorative. And only URLs under the prefix are intercepted: the application, the API and static files follow their usual route, a service worker that caches what it was not asked to cache being one you end up uninstalling by hand on some update day.

Purge precedes write. Without it, assets from an earlier version survive alongside the new ones: writing to the cache overwrites identical URLs but does not remove those that have disappeared from the archive. The recomputed hash map then carries extra entries, no longer matches the one that was signed, and the plugin stands accused of tampering when it was merely dragging its predecessor's leftovers.

The hashes

Every cached asset is digested with SHA-256, and the resulting map is keyed by the path inside the archive, as the signer saw it, so `assets/engine.wasm` and not the virtual URL the cache files it under. A URL path is

encoded: an accented or spaced file name comes back percent-escaped, and without decoding the key would not match, the file appearing to be missing.

An unreadable cache is not an absence of assets. Returning an empty dictionary would fail the signature with a misleading reason; the absence of a map is returned instead, which the verifier turns into an explicit refusal rather than presumed tampering.

11. The R engine

`compute:r` grants execution of arbitrary R code, and that is deliberate. R runs in WebAssembly, in a worker with no DOM and no tokens: it computes, it cannot reach the application. The capability stays separate from data access, and holding it grants neither `samples:read` nor the right to write anything.

The engine does not live in the sandbox. WebR creates a worker of its own, and an opaque origin cannot reliably create one; it also weighs several tens of megabytes, which will not be reloaded once per plugin. It therefore lives in a host worker, and the sandbox presents the usual surface so that plugin code calls it unchanged.

Call	Effect
<code>WebRManager.charger(request)</code>	Boots the engine: pinned version, and optionally the URL of the WebR bundled in the plugin's assets. Returns an engine object. This is the only boot path.
<code>engine.evalR(code)</code>	Evaluates and returns an object whose <code>toJs</code> gives the converted value.
<code>engine.evalRVoid(code)</code>	Evaluates and returns nothing, for side effects.
<code>engine.evalRBoolean(code)</code>	Evaluates and returns a boolean.
<code>engine.captureR(code, options)</code>	Evaluates while capturing console output, and returns the output with the value.
<code>engine.installPackages(packages, options)</code>	Installs R packages in the plugin's environment.
<code>isofind.r.eval(code, options)</code>	Direct route, once the engine has booted. Returns the text output; <code>{ value: true }</code> adds the converted value, which avoids serialising large R objects on every call.
<code>isofind.r.text(code)</code>	The output alone, concatenated. That is what a console wants.

The plugin identifier used to resolve the environment comes from the instance, never from an argument. It is what determines the R environment the code runs in, and letting the caller choose would amount to letting it read another plugin's environment on the same runtime.

The bundled engine URL is honoured only if the plugin's signature covers its assets. An engine loaded on the strength of a path declared in an unverified manifest is not a sovereign engine, it is an executable chosen by whoever built the package.

One example running in production. The PHREEQC console runs PHREEQC inside WebR, with no external executable at all. It declares `samples:read` , `ui:menu` , `ui:notify` , `compute:r` , `storage:read` and `storage:write` , and bundles its engine in its assets.

Governing

What a plugin is allowed to touch, where it came from, and what the log keeps of it.

12. Capabilities and consent

A signed package says who wrote the plugin. It does not say what that plugin touches, and that is the other question. It is reduced to a handful of named grants, requested by the manifest, presented box by box at install time, and checked on every call.

The full vocabulary

Capability	What it grants	
<code>samples:read</code>	Read the loaded samples: the snapshot delivered at initialisation, and the measurements derived from it.	read
<code>samples:write</code>	Create new samples in the project.	writes
<code>analysis:run</code>	Trigger the analysis engine with parameters.	writes
<code>config:read</code>	Read the application settings.	read
<code>config:write</code>	Save application settings. Distinct from the plugin's own storage, which is <code>storage:write</code> .	writes
<code>ui:notify</code>	Show messages on screen.	read
<code>ui:menu</code>	Place menu entries, hence a quick-access entry and a line in the launcher.	read
<code>ui:panel</code>	Declare and open panels, in the docks and in the centre zone.	read
<code>ui:formula</code>	Add formulas to the spreadsheet, with their help entry and their group in the picker.	read
<code>storage:read</code>	Read back its own storage, and know the application's language.	read
<code>storage:write</code>	Keep its own settings, within its quota.	writes
<code>compute:r</code>	Compute in an isolated R engine, with no data access.	read
<code>network</code>	Open outbound connections, over <code>https:</code> only, and never towards the local origin. Refused outright in air-gap mode.	writes

A capability outside the vocabulary exists for nobody

No code refuses it: it is simply dropped from the consent screen, with a console warning. Showing it would mean having the user consent to a label invented by the plugin's author, which has the appearance of control without the substance.

The vocabulary is not quite the workflow one. The two systems share `samples:read`, `samples:write` and `ui:notify`, which mean the same thing and have no reason to carry two names. The rest diverges because the surfaces differ: a workflow writes files and runs solvers, a plugin places formulas and computes in R. A plugin that declares a workflow task, on the other hand, files its capability in the workflow vocabulary, which is the one its host registry uses.

How consent is presented

Every box is ticked by default, except those air-gap mode makes impossible to honour, which appear greyed out with the reason rather than removed. What the user unticks will be refused at runtime, with a message naming the missing capability. Cancelling the screen cancels the whole install.

The plugin's name, its capabilities and their labels arrive in a window built by string concatenation. This is the most sensitive moment of the life cycle, the one where the user grants rights, and everything coming from the manifest is escaped before being shown.

Where the capability is checked

In two places, and they do not overlap. The bridge checks on every incoming call, before looking the method up. And panel registration checks `ui:panel` at declaration time, not only at opening: a panel carrying a dock zone is never opened by the plugin, the shell calls its factory. The manifest declares the zone, it cannot declare the right, since it is written by the plugin's author.

13. Signing, author identity and provenance

The signature answers two questions at once: who produced this package, and is what is about to run the same as what was produced. The second is the less obvious one, and it is what dictates where the hashes are computed.

What is signed

The manifest, the entry point code, and the asset hash map. The manifest carries that map for information, under `signature.assets_sha256`, but the host does not believe it: it recomputes everything from Cache Storage, that is, from the bytes that will actually be served to the plugin and executed by it. That is the only place where the computation means anything. Verifying the downloaded archive would prove it was sound at the moment of download, not that what will run is sound.

The declared map is still used, once. Compared with the one found, it makes it possible to name the offending file: missing from the cache, extra, or different in content. A signature refusal is otherwise opaque, it says there is a discrepancy, never which, and naming the file is the only thing useful to whoever has to fix it.

Who requires a signature

The requirement is decided by the caller, the only party that knows the provenance of the install and the state of developer mode. It is no longer a global property of the bridge: a policy writable on a `window` object can be disarmed in one line from any script in the page. A silent registration counts as requiring, failing which forgetting the field would quietly reopen what is being closed here.

Situation	Signature required
The ordinary case	Yes.
<code>verifySignatures</code> set to false	No. Only an explicit refusal lifts the requirement; a setting that is missing, corrupt or from an earlier version counts as requiring.
Developer mode, local plugin	No, and the exemption is traced in the console. An exemption that leaves nothing behind is one you forget having granted.
Developer mode, plugin from the catalogue	Yes. A catalogue plugin is signed by construction: if its verification fails, what would run is no longer what was signed, and no local setting has any business covering that case.

Local means anything not installed from the catalogue, file import included. Only the catalogue install path stamps that provenance on the record.

The verifier can only be installed once. Allowing it to be replaced would let any script in the page return a favourable verdict at will, which would empty verification of meaning with nothing visible changing.

Unavailable is not the same as valid. A verification service that cannot be reached, an error response, a manifest with no `signature` field: all return an invalid verdict, with distinct reasons. It is then the requirement that decides whether the plugin loads anyway, and the interface shows "unverified" rather than nothing.

Author identity

To sign, you need a key and a certificate. The key is produced by the local service, in Python, and written encrypted into the data directory. It travels over no network, which is the only property that matters: a signing key that exists in two copies proves nothing any more, and revoking the certificate of an author who never held their own key revokes nothing at all. The interface states that constraint rather than hiding it, failing which an author who does not understand it ends up emailing their key, believing they are being helpful.

Step	What happens
1	Name, organisation, email, passphrase of at least twelve characters. The key and the certificate request are produced locally.
2	The request goes to IsoFind. It carries only the public key and the name: it can circulate safely, it cannot sign anything.
3	The returned certificate is installed. The authority chain is already bundled in every IsoFind install: emailing it added nothing but an opportunity to forget a file, and that was the most frequent mistake.

An author who already has a key imports it rather than creating another. That is the only way to keep signing updates to already-published plugins: those would stay valid, but their fixes would never reach their users. The imported key is rewritten encrypted with the supplied passphrase, the file is not copied as is: nothing would guarantee it was encrypted, and a signing key sitting in the clear on a disk is a key already lost.

Certificates expire. A warning appears in the last sixty days, and a signature produced with an expired certificate is refused. Regenerating a key is for loss or compromise only, and revocation of the old certificate must be requested at the same time: without it, whoever holds the stolen key goes on signing in its owner's name.

Signing

The archive is signed locally, then verified before being handed back. An archive that does not verify is not returned: letting it out would mean it gets published and the problem is discovered at its users' machines. The signed archive stays in memory rather than being downloaded straight away, because an author usually moves on to submission and a round trip through the disk would only add an opportunity to submit the wrong file, typically the unsigned version left next to it.

14. The access log

The log answers "what touched this sample". A plugin is the case where the answer is most reliable: it lives in a sandbox with no same origin, it does not speak HTTP, and it therefore cannot set an identity header itself. What the host declares there is what the host did.

The scope is opened around the call, not inside each method. It is the only place that knows both the instance and the required capability, and a method that had to repeat it would one day fail to, with nothing to signal it, while an undeclared access looks exactly like an access that never happened.

Declared field	Contents
<code>genre</code>	<code>plugin</code>
<code>id</code>	The plugin identifier, used as the join key.
<code>capacite</code>	The capability required by the method called.
<code>tache</code>	The name of the host method.

These values feed the same header envelope as the workflows, and the server treats them the same way: a declaration, not a proof, with an ambiguity marker when two consumers interleave their requests. The plugin bridge is in fact one of the two non-workflow consumers to open a scope explicitly, because it is the only one that knows which plugin it is serving.

Declaring the snapshot

The sample snapshot travels through no request. The server middleware would therefore never see it, and the log would suggest the plugin read nothing while it holds the entire database. A separate declaration is therefore issued, with nature `instantane`, the list of identifiers concerned, the volume, and a note saying what the line means: a one-off delivery at initialisation whose later reading is observable nowhere.

The host waits for the verdict before delivering. That is what makes a revocation effective on a route that does not pass through the middleware. If the log cannot be reached, delivery happens anyway: depriving a plugin of its data because a registry stumbled would be a worse failure than the missing line you wanted to write.

15. Conflicts, dependencies and compatibility

Three checks run before consent, in this order: edition compatibility, dependencies, then conflicts. The first two can install something else or refuse; the third warns and leaves the decision to the user.

Conflicts

What is looked for	Why
Global variables	Two plugins placing the same name overlap, and the second wins without the first knowing.
Menu names	Two identical entries in quick access are indistinguishable once placed.
Shared capabilities	Two plugins that can write the same data can undo each other.
Category	Flags likely functional duplicates, without forbidding them.
Code patterns	Five patterns found in the code, with their severity.

Pattern	Severity	What it would do
<code>localStorage.clear()</code>	Critical	Wipes the whole application's storage, not just its own.
<code>document.write(</code>	High	Can erase the page.
<code>eval(</code>	High	Runs code built at runtime.
<code>innerHTML = ... <script</code>	High	Injects a script through a string.
<code>\${ ... eval(</code>	High	Dynamic evaluation hidden inside a template.

This scan is a warning, not a verdict. It works on the text of the code, so it flags a dangerous plugin and a plugin that handles those patterns for a legitimate reason equally well. Its job is to make the user ask the question before installing, not to decide on their behalf.

Dependencies

A plugin may declare other plugins it needs. They are resolved before it, depth first, with a cycle guard: the current plugin is marked on entry, so that a dependency pointing back at it does not trigger its reinstallation. What is present in the catalogue is offered for installation, in a single window; what is not is named, and left to install by hand. Cancelling that window cancels the whole install.

Edition compatibility

A plugin may declare which IsoFind editions it works with. Blocking only happens when the current edition is known and absent from the declared list: a plugin that declares nothing runs everywhere, and an undetermined edition does not block. A check that refused in case of doubt would prevent installation on exactly the configurations it cannot name.

Air-gap mode

It does not merely cut the network, it closes the doors one by one, each with its own message.

What is closed	What remains
The online library and the remote registry	The first-party catalogue, bundled, code included.
Sharing and submission to the catalogue	Packaging and signing, which are local.
Downloading Pyodide from a CDN	The local bundle served by the application, which is the sovereign default. The CDN was only ever a development fallback.
The network capability	Nothing. The content security policy falls back to ' none ' whatever box was ticked.

16. The manager and the entry points

Four surfaces give access to plugins, and they serve different moments. The manager administers, the launcher opens, quick access shortens, the palette finds.

The plugin manager

One window, six views: installed plugins with their badges and settings, the library, installation from a file, the creation assistant, the built-in documentation and the platform settings. That is where enabling, disabling, updating, revising quick access and uninstalling happen.

Reloading is decided on close, not on click. An install removes its progress window from code, and a window removed by code fires no close handler: the reload decision therefore depended on the user's gesture, when it depends on the state of the application. It now hangs on an observable fact, the disappearance of every plugin window from the document, which covers the closing routes you would forget to instrument.

There are two kinds of reload. The soft one unloads every contribution then reloads the active plugins, which guarantees a consistent state with no orphan contribution and no duplicate. The hard one reloads the document, and is required by operations that may have written new assets to the cache. One decision is taken for a whole batch of operations, which allows several installs to be chained.

The Plugins panel, or launcher

It does not rebuild a list, it makes visible what the registries hold. Actions come from the plugin command category, panels from the panel registry filtered on their source, and the whole is grouped by plugin. A plugin brought in by an add-on appears there on the same footing as any other, without anything here having to know.

Every row says what it is in plain words, action or panel, to the right of the name. An extension often gives its panel and its action the same name: the list then showed the same label twice, distinguished only by a sixteen-pixel icon. The word says where you are going, the icon merely reminds you.

Opening a panel from the launcher goes through the placement arbiter, never through a direct open. It alone frees the other surfaces: without it, opening in the centre a panel that is already floating would make two live instances, each with its own state and its own registry subscription.

Quick access

The sidebar's Plugins section has a single owner, which both sources of entries delegate to, the bridge for sandboxed plugins and the contributions registry for host plugins. Ordering, hiding and persistence are therefore consistent whatever the origin. Hiding an entry removes it from quick access without disabling the plugin: its menu remains launchable from the launcher and from the manager.

The truth is in the list, not in the DOM. The sidebar is shared and can be rebuilt by the application; entries are therefore reinjected from the list rather than on the assumption that the DOM persists, with a few deferred retries and then an observer that only reacts when an entry is genuinely missing.

The title bar button

The palette and the menu both assume you know where to look. A permanently visible button does not, and that is what was missing: nothing, natively, led to the extensions. It sits in the title bar rather than in a dock, because that is the only chrome present on every page, the workshop included, and because its registry releases its tools when a source unloads.

The commands

Command	Effect
<code>core.plugins.toggle</code>	Show or hide the launcher.
<code>core.plugins.authorIdentity</code>	Author identity and signing.
<code>core.plugins.checkBeforeSubmit</code>	Verify the signature before submitting.
<code>core.plugins.atelier</code>	Open the development workshop, local zone only.

Like every command, they enter the palette, the Actions panel and universal search, and each can be given whatever shortcut the user wants.

Reference

The catalogue of host methods, the manifest field by field, publishing, the workshop, typical plugins and the limits.

17. Catalogue of host methods

Everything a plugin can ask the application for, grouped by what it requires. This list serves as architecture: a missing method is something nobody will be able to do from a plugin, and one method too many is a door that will have to be defended.

The first column gives the call as an author writes it in the sandbox, on the `isofind` object. The second gives its equivalent on the shared Python runtime's facade. Underneath both sits the same host method, whose name appears in the access log and in refusal messages: `getSamples`, `storageSet`, `rEval`. The two surfaces are aligned deliberately, same methods and same capabilities for two languages.

Everything that crosses the boundary returns a promise. That is the one concession confinement demands, and it is unavoidable: nothing that crosses an isolation boundary can be synchronous. Contributions are the exception, since they stay local and merely announce.

No capability

Two only, and they do not leave the sandbox. `isofind.version` carries the protocol version, and `isofind.log(...)` writes to the host console, re-prefixed with the plugin's name.

Reading samples `samples:read`

Sandbox	Python	Arguments and return
<code>isofind.getSamples()</code>	<code>get_samples()</code>	The loaded samples. No arguments.
via compatibility	<code>get_samples_df()</code>	The same, as a pandas <code>DataFrame</code> on the Python side. Returns an empty frame when there is nothing.
via compatibility	<code>get_analyses()</code>	The isotopic measurements flattened, one row per measurement, carrying its sample's context. Derived from the samples, not from another source.

Writing to the project `samples:write`

Sandbox	Python	Arguments and return
<code>isofind.addSample(object)</code>	<code>add_sample(dict)</code>	Creates a sample. An object with no <code>name</code> is refused. Returns a boolean. From Python the object is serialised to cross the boundary, failing which it would arrive as a <code>Map</code> and be ignored.

Analysing `analysis:run`

Sandbox	Python	Arguments and return
<code>isofind.runAnalysis(params)</code>	<code>run_analysis(dict)</code>	Triggers the analysis engine. Returns its result, or nothing if no engine is mounted.

Application settings `config:read` , `config:write`

Sandbox	Python	Arguments and return
<code>isofind.getConfig(key)</code>	<code>get_config(key=None)</code>	One value, or the whole table when the key is omitted.
<code>isofind.setConfig(key, value)</code>	<code>set_config(key, value)</code>	Saves. Returns a boolean. Structures are serialised on the Python side.

Interface `ui:notify`

Sandbox	Python	Arguments and return
<code>isofind.notify(message, type)</code>	<code>notify(message, type='info')</code>	Shows a message. <code>info</code> , <code>success</code> , <code>warning</code> or <code>error</code> . <code>show_notification</code> is still accepted in Python, under its old name. The <code>plugin.notify</code> shortcut does the same thing.

Contributions `ui:menu` , `ui:panel` , `ui:formula`

Sandbox	Python	Arguments and return
<code>isofind.registerMenu(id, label, callback)</code>	<code>register_menu(...)</code>	Places a quick-access entry. The callback stays in the sandbox and will be invoked on click. Requires <code>ui:menu</code> .
<code>isofind.registerPanel(id, label, render)</code>	<code>register_panel(...)</code>	Declares a panel. <code>render</code> receives a <code>CONTEXT</code> , <code>{ root, id }</code> , not the element itself: it is <code>ctx.root</code> , an element of the sandbox document, that must be populated. Its return value is ignored, so returning an HTML string produces nothing and the panel stays blank. Requires <code>ui:panel</code> , checked at declaration time.
<code>isofind.openPanel(id)</code>	<code>open_panel(id)</code>	Asks for an already-declared panel to be opened. The host reveals the shell then sends back the render order.
<code>isofind.registerFormula(name, fn, spec)</code>	<code>register_formula(name, fn, group, label, syntax, example)</code>	Adds a formula to the spreadsheet. The two surfaces diverge here: the sandbox takes a spec object, with the keys <code>group</code> , <code>label</code> , <code>syntax</code> and <code>example</code> ; Python takes them as successive arguments. The name must match <code>^[A-Z][A-Z0-9_]{1,30}\$</code> and be free. A missing function throws on the spot, before any message is sent.

Plugin storage `storage:read`, `storage:write`

Sandbox	Host method	Arguments and return
<code>isofind.storage.get(key)</code>	<code>storageGet</code>	The value, as text, or nothing.
<code>isofind.storage.set(key, value)</code>	<code>storageSet</code>	Writes. The value is coerced to text. Rejects when the quota is reached.
<code>isofind.storage.remove(key)</code>	<code>storageRemove</code>	Removes a key.
<code>isofind.storage.keys()</code>	<code>storageKeys</code>	The plugin's keys, prefix stripped.
<code>isofind.storage.getJSON(key, fallback)</code>	<code>storageGet</code>	Convenience. Returns the fallback when the key is missing or the text is not readable JSON, rather than throwing.
<code>isofind.storage.setJSON(key, value)</code>	<code>storageSet</code>	Convenience, because nearly every use of storage is hand-serialised JSON.

Computing in R `compute:r`

Sandbox	Host method	Arguments and return
<code>WebRManager.charger(request)</code>	<code>rBoot</code>	Boots the engine and returns an engine object. This is the only boot path: calls on <code>isofind.r</code> assume an engine is already booted and fail otherwise.
<code>isofind.r.eval(code, options)</code>	<code>rEval</code>	Evaluates. Returns the text output; pass <code>{ value: true }</code> to also get the converted value, which avoids serialising large R objects on every call.
<code>isofind.r.text(code)</code>	<code>rEval</code>	Convenience. The output alone, concatenated. That is what a console wants.
<code>isofind.r.install(packages, repo)</code>	<code>rInstall</code>	Installs packages in the plugin's environment.
<code>isofind.r.reset()</code>	<code>rReset</code>	Resets the environment. The corresponding host method is not served in the shipped state: the call fails with "unknown method".
via compatibility	<code>rEvalVoid</code> , <code>rEvalBoolean</code> , <code>rCapture</code>	Evaluation with no return, boolean evaluation, and output capture with the value. Reached through the engine object <code>charger</code> returns.

Network `network`

No method. The capability opens nothing on the host side: it loosens the sandbox's content security policy, which then lets `https:` connections through. The plugin calls out itself, with the browser's own tools.

A method absent from the table is refused. There is no list of forbidden methods, only this list of served ones. That is what makes refusal the rule and service the exception, rather than the other way round.

The application's HTTP API is documented separately, and can be tried out. Most of the methods above end up calling a route on the local server. Those routes are described and executable from the Swagger interface the FastAPI backend exposes under `/docs`, with its ReDoc variant under `/redoc`, at the address the application runs on.

A plugin cannot call them: the content security policy closes the way out towards the local origin, capability or not. But reading them remains the fastest way to learn the exact shape of a sample, an isotopic measurement or an analysis result before writing the code that consumes it, rather than guessing it from one particular dataset.

The compatibility layer

Plugins written for the host path address `window.localStorage`, `IsoFindPluginAPI`, `PluginContributions`, `WebRManager`. None of that exists in an iframe on an opaque origin, and `localStorage` there throws on a mere read.

The sandbox therefore recreates those objects. This is not a convenience: without them, moving a plugin from the host path into confinement would require rewriting it, and confinement that forces a rewrite is confinement nobody adopts.

Recreated object	What it does, and what it cannot do
<code>localStorage</code>	A synchronous facade backed by the snapshot delivered at initialisation. Reads come from the cache; writes are immediate in memory and pushed to the host without waiting for confirmation, a failure being reported in the console rather than surfaced: no caller of <code>setItem</code> is written to receive one. The one limitation is that a plugin would not see a write made by another instance of itself, and there is only ever one. The native accessor is shadowed by an own property, redefining it in place being refused.
<code>IsoFindPluginAPI</code>	<code>getSamples()</code> returns a copy of the snapshot, synchronously, because that is how plugins call it. The list can therefore be stale if a sample is added while the plugin is running; <code>getSamplesAsync()</code> queries the host for anyone who cares about freshness. The rest delegates to the bridge.
<code>PluginContributions</code>	<code>addMenu</code> and <code>addPanel</code> announce the contribution just as the <code>isofind</code> methods would. <code>teardown</code> does nothing: the host tears down by source.
<code>WebRManager</code>	<code>charger()</code> boots the engine on the host side and returns an object presenting the same surface as WebR does locally: <code>evalR</code> , <code>evalRVoid</code> , <code>evalRBoolean</code> , <code>captureR</code> , <code>installPackages</code> . The returned objects keep <code>toJs</code> , <code>free</code> and <code>destroy</code> so that calling code goes on working without knowing that conversion already happened in the worker and there is nothing left to free.

One thing the layer cannot cover: `new Worker`, which WebR calls and which an opaque origin forbids. That is a browser limitation, not a missing method, and it is why the R engine lives in a host worker rather than in the sandbox. It also weighs several tens of megabytes, which will not be reloaded once per plugin.

18. The manifest, field by field

Field	Role	Default
<code>id</code>	Stable identifier. Used as the install key, the storage namespace, the asset URL prefix, the source of contributions and the join key in the log. Changing it amounts to publishing a different plugin.	required
<code>name</code>	Display name, in English: catalogue, moderation and consent screen.	required
<code>version</code>	Compared component by component for updates.	required
<code>entry</code>	Entry point. Its extension decides the language: <code>.py</code> for the Python path, everything else for the JavaScript one.	<code>plugin.js</code>
<code>capabilities</code>	The capabilities requested. Anything outside the known vocabulary is dropped at install time.	<code>[]</code>
<code>contributes.menus</code>	The declared menu entries, each with its identifier and label.	empty
<code>contributes.panels</code>	The declared panels. Each carries an identifier, a label, and optionally <code>dock</code> and <code>icon</code> .	empty
<code>assets</code>	The assets folder to extract into Cache Storage. The declared prefix is authoritative, for signing as well as for writing.	none
<code>dependencies</code>	Identifiers of other required plugins, resolved before installation.	empty
<code>runtime</code>	<code>isofind-native</code> asks for host-side execution, outside the sandbox. Any other value, or its absence, leaves the normal path.	absent
<code>category</code>	Used for catalogue filtering: <code>analysis</code> , <code>visualization</code> , <code>data</code> , <code>tools</code> , <code>science</code> .	<code>tools</code>
<code>icon</code>	Icon file inside the package.	<code>icon.png</code>
<code>author</code>	Name shown on the record. Not to be confused with the certificate identity, which is the authoritative one and the one moderators see.	unknown
<code>description</code>	Catalogue text, in English.	empty
<code>email</code>	Contact shown in the plugin's settings.	none
<code>pythonPackages</code>	Packages to load before execution, Python path only.	<code>numpy</code> , <code>pandas</code>
<code>keywords</code> , <code>repository</code> , <code>license</code> , <code>website</code>	Metadata shown, with no effect on loading.	none
<code>signature</code>	Added by signing. Carries the asset hash map, for information: the host recomputes it.	absent

Legacy fields, still read

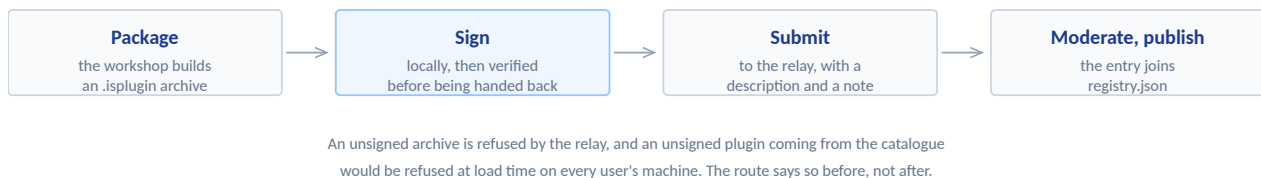
Field	Current equivalent
-------	--------------------

main	entry
permissions	capabilities
language	Derived from the entry extension. Consulted only when that extension settles nothing.

19. Publishing a plugin

Four steps, in this order, and each refuses what the previous one did not do.

From workshop to catalogue



Packaging

The workshop builds the archive from the working folder, named after the identifier and the version. Nothing forces you through it: a zip containing a valid manifest at the root will do. The workshop's advantage is that it refuses an invalid manifest before writing the file.

Signing

Signing requires an author identity on this machine, described in section 13. The archive goes to the local service with the passphrase, comes back signed, and is handed back only if it verifies. An archive that fails verification is not returned at all.

Submitting

Submission attaches the archive, a description and a note to the moderators. The identity shown to moderators comes from the certificate, not from what the author types in the form.

The proof of work the relay demands is computed by the relay, and no secret passes through the application window. The signature check, however, happens earlier: the server refuses an unsigned archive, but it does so after having had the proof of work computed. The user would therefore have waited, heated their processor, and collected a refusal. The application says so beforehand, and shows them the way out rather than leaving them at a wall.

The registry

The remote catalogue is a JSON document carrying the schema [isofind-plugin-registry/v1](#) and a list of entries. It is loaded when the library opens, outside air-gap mode, and merged with the bundled first-party catalogue. Where an identifier is shared, the local one wins, and any outage leaves the local one alone rather than emptying the library.

A remote entry does not carry its code, but a download address. Installing it follows the same route as a file import: assets written, conflicts checked, capabilities consented, signature verified.

20. The development workshop

A panel for writing a plugin from a working folder: manifest validation, hot reload, contribution inspection, packaging.

The workshop exists in the local zone only. This is not a display nicety. The service behind it reads and writes files on the machine; exposing it outside developer mode would amount to offering arbitrary file reads from the application window.

Hot reload

The folder's timestamp is polled every 1200 milliseconds. On the slightest change the plugin is reloaded, and it is unloaded first. Without that prior unload, the previous version's contributions would survive alongside the new ones: an identifier removed from the manifest would stay in the palette, and the author would think the manifest was being ignored.

The workshop plugin carries a provenance mark. It comes neither from the catalogue nor from a signed archive, and that origin must stay legible to the guards: a workshop plugin must never be able to pass for an installed one.

Closing the panel does not stop the polling. The plugin under development stays loaded, and hiding a surface should not stop reloading what you are in the middle of writing.

The inspector

It puts two lists side by side: what the manifest declares, and what the core registries actually hold for that source. It is the most useful debugging tool in the workshop.

State	What it means
implemented	Declared in the manifest and registered by the code. The normal case.
declared, awaiting activation	The core holds the declaration, the code has not replaced it yet. Either the plugin has not implemented it, or it is not enabled yet.
declared but absent	The manifest announces it, the core has no trace of it. Often work in progress, sometimes a typo in an identifier.
registered without declaration	The serious case. It works here, but it will escape the user's consent, who will never have seen that it existed.

The last case is explained, not merely coloured. A contribution appearing without having been announced is precisely what manifest declaration is meant to make impossible.

What else the workshop flags

A plugin declaring automatic startup gets a warning: its code runs at launch, before any invocation. That is allowed, and a moderator will ask why.

21. Four typical plugins

Four recurring shapes, written here in their shipped form rather than described. Manifest labels are in English; the plugin's interface picks its language at runtime.

Plugin code gets `isofind` and `plugin` in scope: nothing to import, nothing to look up in `window`. Everything that crosses the boundary returns a promise.

A panel, and nothing else

The shortest useful plugin: a menu entry that opens a panel, and a panel that shows what it has. Two capabilities are enough, plus the one that grants access to the data.

```
// manifest.json
{
  "id": "sample-counter",
  "name": "Sample Counter",
  "version": "1.0.0",
  "entry": "plugin.js",
  "capabilities": ["samples:read", "ui:menu", "ui:panel"],
  "contributes": {
    "menus": [ { "id": "open", "label": "Open Sample Counter" } ],
    "panels": [ { "id": "main", "label": "Sample Counter", "dock": "right" } ]
  }
}
```

```

// plugin.js
// The language travels with the storage snapshot: the compatibility
// facade returns it with no round trip, from the first render.
var fr = (localStorage.getItem('isofind_language') || 'fr') !== 'en';

isofind.registerPanel('main', 'Sample Counter', function (ctx) {
  // The render receives { root, id }: ctx.root is what to populate.
  var root = ctx.root;

  var p = document.createElement('p');
  p.style.color = 'var(--text-primary)';
  p.textContent = fr ? 'Comptage...' : 'Counting...';
  root.appendChild(p);

  // The call crosses the boundary, so it returns a promise. The
  // synchronous view of the snapshot is still available through
  // IsoFindPluginAPI.getSamples() for code already written that way.
  isofind.getSamples().then(function (samples) {
    var n = samples.length;
    p.textContent = fr ? n + ' échantillon(s) chargé(s)'
      : n + ' sample(s) loaded';
  }).catch(function (e) {
    p.textContent = e.message; // "Capacite refusee: samples:read"
  });
});

isofind.registerMenu('open', 'Open Sample Counter', function () {
  isofind.openPanel('main');
});

```

The root is emptied before every render. The panel therefore rebuilds itself entirely on each opening, and whatever state it wants to keep must live elsewhere, in a plugin variable or in its storage. Closing a panel destroys nothing: the content stays in place, it is simply rebuilt at the next display.

A plugin that remembers its settings

Storage belongs to the plugin and survives being disabled. It does not survive being uninstalled.

```

{
  "capabilities": ["ui:menu", "ui:notify", "storage:read", "storage:write"]
}

```

```

var DEFAULTS = { threshold: 3, unit: 'permil' };

function readSettings() {
  // getJSON returns the fallback when the key is missing or the text
  // is not readable JSON, rather than throwing mid-render.
  return isofind.storage.getJSON('settings', DEFAULTS);
}

function writeSettings(s) {
  return isofind.storage.setJSON('settings', s)
    .then(function () {
      isofind.notify('Settings saved', 'success');
    })
    .catch(function (e) {
      // The plugin quota is full, or storage:write was not granted.
      // Say so, rather than letting the user believe the setting
      // was taken.
      isofind.notify(e.message, 'error');
    });
}

```

A contributed formula

The function stays in the sandbox; only its spec goes to the host, which takes care of the synchronous wrapper, the function picker and the help entry.

```

isofind.registerFormula('D_EXCESS', function (d180, d2H) {
  return d2H - 8 * d180;
}, {
  group: 'Stable isotopes', // group in the picker
  label: 'Deuterium excess', // label in the help entry
  syntax: 'D_EXCESS(d180, d2H)',
  example: 'D_EXCESS(-8, -54)'
});

```

The name is a spreadsheet identifier: leading capital, then capitals, digits and underscores, thirty-one characters at most. A malformed or already-taken name is refused with a console warning, and the formula is not placed. The check covers every global in the page, not only functions.

The first evaluation of a cell returns an empty value, the answer comes back from the sandbox, and the sheet recalculates. An exception thrown inside the function also returns an empty value, without bringing the plugin down.

A plugin with a bundled engine

The engine travels in the package, it is written to the cache at install time, and it is loaded from the URL the host supplies. Nothing is downloaded, neither on the first launch nor on any later one.

```

{
  "id": "r-console",
  "name": "R Console",
  "version": "0.11.0",
  "entry": "plugin.js",
  "assets": "assets/",
  "capabilities": ["samples:read", "ui:menu", "ui:notify",
                  "compute:r", "storage:read", "storage:write"],
  "contributes": {
    "panels": [ { "id": "console", "label": "R Console", "dock": "bottom" } ]
  }
}

```

```

// The base URL was placed on the context at load time. It is
// deterministic, so it holds on later launches with no re-extraction.
//
// charger() is the only boot path: the engine lives in a host worker,
// because an opaque origin cannot create a Web Worker and because WebR
// will not be reloaded once per plugin.
WebRManager.charger({ baseUrl: plugin.assetsBaseUrl + 'webr/' })
  .then(function (engine) {
    return engine.evalR('R.version.string');
  })
  .then(function (obj) {
    return obj.toJs();
  })
  .then(function (version) {
    isofind.notify(version, 'info');
  })
  .catch(function (e) {
    // Asset missing from the cache: the service worker answers in
    // plain words rather than a silent 404, and the message is
    // readable here.
    isofind.notify(e.message, 'error');
  });

// Once the engine is up, the console uses the output shortcut.
function run(code) {
  return isofind.r.text(code);
}

```

The base URL is honoured only if the signature covers the assets. An engine loaded on the strength of a path declared in an unverified manifest is not a sovereign engine, it is an executable chosen by whoever built the package, and it will not run in an application worker.

22. Troubleshooting

The common failures, and where to start. Most of them show up within a minute in the console, which is deliberately talkative about refusals: a silent refusal gets hunted in the wrong file for half an hour.

Symptom	Likely cause	Where to look
---------	--------------	---------------

The plugin does not load and the console mentions the signature	Signature required and invalid: unsigned archive, expired certificate, or content modified since signing	The badge on the record, which carries the reason. For a local plugin being written, developer mode lifts the requirement.
The signature refusal does not say what is wrong	A discrepancy between the signed map and the assets found in cache	The console names the files that are missing, extra or different. Reinstalling rewrites the assets.
The panel opens as a modal instead of joining a dock	The shell registry was not available, or the page has neither dock nor centre zone	The console at plugin load time. A page outside the boot manifest does not carry the shell.
A panel goes blank, and another closed by itself	Two panels of the same plugin: it has one iframe, and the second retargets it	Expected. A plugin shows one surface at a time.
The bundled engine fails on a compilation error	Its assets are not in the cache: never written, or cleared by a browser cleanup	The body of the 404 response, which says so in plain words. Reinstalling the plugin rewrites them.
A formula returns an empty value and never changes	Name rejected by the shape check, or already taken by another global	The console, which names the dropped formula. The name must match <code>^[A-Z][A-Z0-9_]{1,30}\$</code> .
A formula returns an empty value on the first evaluation, then the right one	Expected. The synchronous wrapper returns an empty value while waiting for the sandbox, then the sheet recalculates	Nothing to fix.
A call fails with "capability refused"	The box was unticked at install time, or the capability is outside the known vocabulary and was dropped	The plugin's settings in the manager, and the console warning issued at install time.
The plugin sees no samples	<code>samples:read</code> not granted, or access revoked at load time	The console reports a withheld snapshot. A revocation only bites at the next load.
The plugin's colours do not follow the theme	A token used that is not in the exposed list	The list in section 9. A token outside the list never reaches the sandbox document.
An entry has vanished from the sidebar	It was hidden with its cross, which does not disable the plugin	The plugin's settings. The action is still launchable from the launcher.
Nothing changes after an install	Reloading is decided when every plugin window closes, so that installs can be chained	Close the manager. An install that wrote assets calls for a document reload.
The workshop is nowhere to be found	It exists in the local zone only	Expected. The service behind it reads files on the machine.

Sharing refuses the archive before even sending it	It is not signed	Plugins menu, author identity and signing. The check happens before the proof of work, so as not to have it computed for nothing.
An R call fails with "R engine not booted"	The engine was never booted: <code>isofind.r</code> assumes one is already in place	Call <code>WebRManager.charger()</code> before anything else. It is the only boot path.
The panel loses its state when switching tabs	The root is emptied before every render, and the panel rebuilds entirely	Expected. State must live in a plugin variable or in its storage, not in the panel's DOM.
A Python plugin does nothing after loading	Its callbacks were freed, or its namespace did not survive	The console at unload time. A reloaded plugin loses the <code>PyProxy</code> objects of its previous version, which is intended.

The useful reflex. Look first at what the console said at load time, not at what it says when the failure surfaces. Nearly every platform refusal is pronounced there: signature, dropped capability, rejected formula, unavailable dock zone, assets missing from the cache. By the time the failure shows, the cause has often been named several seconds earlier.

23. Limits and persistence

The platform constants

Limit	Value	Reason
Storage quota per plugin	2 MB	Without it, a plugin fills the origin's storage and the whole application stops being able to write.
Formula name length	2 to 31 characters	Leading capital, then capitals, digits and underscores.
Workshop folder polling	1200 ms	Short enough not to wait, long enough not to reread a folder constantly.
Sheet recalculation batching	50 ms	Several formulas often resolve together; one recalculation per answer would be wasted.
Sidebar injection retries	0, 150, 500, 1500 ms	The sidebar may appear after the plugin, or be rebuilt afterwards.
Sidebar-not-found diagnostic	1600 ms	Said once, and clearly. A missing button with no explanation gets hunted for a long time.
Protocol version	2.0.0	Announced by the bridge at load time.

Stacking levels

Surface	Level	Why
Panel docked or in the centre	1	Below any native modal, so that a dimmed backdrop covers it normally, like the rest of the shell.
The plugin's own modal backdrop	11000	The dimmed background of its own window.
The iframe in modal mode	11001	Just above that backdrop. Since the iframe is no longer a child of the overlay but a separate element under <code>body</code> , too low a level would put it below its own backdrop.

What survives the page

Key	Contents
<code>isofind_plugins</code>	The installed plugins' records: identity, granted capabilities, code, original manifest, signature verdict, enabled state.
<code>isofind_plugin_settings</code>	The platform settings.
<code>isofind_quickaccess</code>	The ordering and hiding of quick-access entries.
<code>isofind_plugin_store:<id>:<key></code>	A plugin's storage. Purged on uninstall, never on unload.
<code>isofind_config</code>	The application configuration, read and written by <code>config:read</code> and <code>config:write</code> .
<code>isofind_language</code>	The language, passed to plugins with the storage snapshot.
<code>isofind-plugin-assets</code>	Cache Storage, not local storage: the plugins' assets, which sometimes weigh tens of megabytes and must persist between launches.

The platform settings

Setting	Default	Effect
<code>verifySignatures</code>	true	Only an explicit refusal lifts the signature requirement. A missing or corrupt setting counts as requiring.
<code>sandboxMode</code>	true	Confinement by default.
<code>devMode</code>	false	Lifts the signature requirement for local plugins only, and opens the workshop.
<code>airgapMode</code>	false	Closes the catalogue, sharing, the CDN and the network capability.
<code>autoUpdate</code>	false	Automatic updates from the catalogue.
<code>allowNetwork</code>	false	Global setting, distinct from the capability granted plugin by plugin.
<code>allowFileSystem</code>	false	Global setting.

<code>cacheLimit</code>	100	Cache ceiling, in megabytes.
<code>pyodideLocalPath</code>	empty	Explicit path to the Pyodide bundle. Takes precedence over probing for the bundled one, which itself takes precedence over the CDN.

Unreadable storage no longer throws: it falls back to these defaults, which are the strictest. Raising an exception to every caller, including the signature policy and air-gap detection, made security depend on the good health of a string.

Events

Event	When
<code>plugins.changed</code>	The list of installed plugins changes, with the total and the number enabled. Also emitted at startup, when the list goes from empty to full.
<code>registry.changed</code>	A command or a panel enters or leaves the registry. The launcher and the title bar button subscribe to it: without that relay you would have to reopen the panel to see what you had just installed, and the install would look as though it had failed.
<code>themechange</code>	Emitted by the theme manager, listened to by the bridge, which rebroadcasts the tokens to every open sandbox. Plugins follow along without being reloaded.

Surfaces exposed to the page

Object	Role
<code>PluginSandboxHost</code>	The bridge: load, unload, purge storage, install the verifier, read a verdict.
<code>PluginState</code>	The single source of truth on installed plugins. Exposed because the inspector, the shell menu and the Log panel consult it.
<code>PluginQuickAccess</code>	The single owner of the sidebar's Plugins section, whatever the origin of the entries.
<code>PluginContributions</code>	Contributions materialised on the host side, for plugins that run in the page.
<code>PluginAssets</code>	Write, purge, check presence and compute hashes for assets.
<code>IsoFindPluginAPI</code>	Access to the application's data, behind which both capability tables delegate.

Surfaces recreated inside the sandbox

Object	Role
<code>isofind</code>	The SDK surface, version 2.0.0: data, settings, notifications, storage, R, contributions, logging.
<code>plugin</code>	The instance context: identifier, name, asset base URL, granted capabilities, notification shortcut.
<code>localStorage</code>	Synchronous facade backed by the snapshot, shadowing the native accessor, which throws on an opaque origin.

IsoFindPluginAPI	Same name as on the host side, same surface, so that a plugin coming from the host path has nothing to rewrite.
PluginContributions	Announces contributions like the isofind methods. Its teardown does nothing, the host tearing down by source.
WebRManager	Boots the R engine in a host worker and presents its local surface.

Writing

From nothing to a signed plugin, in the order the steps actually happen, with what you need to know at each one.

24. From nothing to a running plugin

Six steps, and the first useful plugin fits in two files. This section takes them in order; the ones that follow come back to whatever deserves explaining.

Step	What you do	What it requires
1	Create a folder with a <code>manifest.json</code> and a <code>plugin.js</code> .	Nothing.
2	Open that folder in the workshop, which validates the manifest and loads the code.	Developer mode, and the local zone.
3	Write, save, watch. The plugin reloads itself.	Nothing.
4	Read the inspector, until every contribution is marked implemented.	Nothing.
5	Package, which produces an <code>.isplugin</code> archive.	A valid manifest.
6	Sign, then install or submit.	An author identity on this machine.

The smallest plugin that loads

A manifest needs three fields to be accepted, and a fourth to be useful. With no declared capability, installation asks for nothing and the plugin can do nothing but compute in its corner.

```
// manifest.json
{
  "id": "my-first-plugin",
  "name": "My First Plugin",
  "version": "0.1.0",
  "entry": "plugin.js",
  "capabilities": ["ui:notify"]
}
```

```
// plugin.js
// Nothing to import. isofind and plugin are already in scope, and the
// host runs the code in strict mode.
isofind.log('Loaded:', plugin.id, plugin.capabilities);
isofind.notify('Hello from ' + plugin.name, 'info');
```

If nothing happens, the application console will say why. It is deliberately talkative about refusals, and almost every platform failure is named there at load time, not at the moment you notice it.

Working in the workshop

The workshop reads the folder, polls its timestamp every 1200 milliseconds, and reloads the plugin as soon as a file moves. It unloads first, which stops a contribution removed from the manifest surviving alongside the new ones and making it look as though the manifest were being ignored.

Three things are visible there at all times. Manifest validity, with the list of errors when there are any. The inspector, which compares what the manifest declares with what the registries actually hold. And a warning if the plugin asks to run at startup, which is allowed but which a moderator will want justified.

Without the workshop the cycle is far longer. You can perfectly well zip the folder and install it from a file, but every fix then means rezzipping, uninstalling, reinstalling and reloading the application. The workshop is not an author's comfort, it is what makes development practical.

What you end up with

Packaging produces an archive named after the identifier and the version. It is not signed: an unsigned plugin is refused at load time as soon as verification is required, and refused by the submission relay in every case. Signing is therefore the next step, not a formality to be left for later.

25. Choosing your capabilities

The rule fits in one sentence: ask for what you use, nothing more. Every extra box is one more reason, for the user reading it, not to install. A plugin asking to read the database in order to show a unit converter will be unticked, and will then fail with nobody understanding why.

What the plugin wants to do	What it must ask for
Show a window of its own	<code>ui:panel</code>
Be reachable from the sidebar	<code>ui:menu</code>
Tell the user something	<code>ui:notify</code>
Read the loaded samples, or their measurements	<code>samples:read</code>
Create a sample	<code>samples:write</code>
Remember a setting between sessions	<code>storage:read</code> , <code>storage:write</code>
Know the application's language	<code>storage:read</code>
Add functions to the spreadsheet	<code>ui:formula</code>
Compute in R, with or without a bundled engine	<code>compute:r</code>
Read or write the application settings	<code>config:read</code> , <code>config:write</code>
Trigger the analysis engine	<code>analysis:run</code>

What requires nothing

Everything that stays inside the sandbox. Drawing an interface, reading theme tokens, computing, writing to the console through `isofind.log`, knowing its own identifier and the base URL of its assets. A pure computation plugin that gets its data from the user, through an input field or a paste, needs no data capability at all.

Five traps

The dock zone does not replace `ui:panel`. A panel declared with a zone is opened by the application shell, which never goes through the plugin's call. The capability is therefore checked at declaration time. The manifest declares the zone, it cannot declare the right, since the plugin's author writes it.

The language costs `storage:read`. It travels with the storage snapshot, and without that capability the plugin does not receive it. It will fall back to a default language, which is acceptable, but it is a choice to make knowingly.

`compute:r` grants no data. The R engine runs in a worker with no DOM and no tokens. A plugin that wants to analyse samples in R needs both capabilities, and will have to pass the data to the engine itself.

`network` is useless in air-gap mode. The content security policy falls back to no egress whatever box was ticked, and the box appears greyed out at install time. A plugin meant for isolated machines should not depend on it to work.

An invented capability is ignored. Anything outside the vocabulary in section 12 is dropped at install time, with a console warning. The plugin will install, and will then fail on a capability the user never had a chance to grant.

26. Writing a panel interface

The sandbox document is blank. No application stylesheet is loaded there, and nothing looks like IsoFind until the plugin writes it. What is provided is the palette: the theme tokens listed in section 9, injected at creation and refreshed on every switch.

Place your stylesheet once

```
function styles() {
  if (document.getElementById('mp-style')) return; // placed only once
  var st = document.createElement('style');
  st.id = 'mp-style';
  st.textContent = [
    // The prefix avoids any collision if the plugin gains a second
    // panel. Colours come from the tokens, never from a hard-coded
    // value: a dark theme would make the panel unreadable.
    '.mp{font:13px/1.55 system-ui,-apple-system,"Segoe UI",sans-serif;',
    '  color:var(--text-primary);padding:14px 16px;}',
    '.mp h3{margin:0 0 8px;font-size:14px;color:var(--brand-primary);}',
    '.mp-row{display:flex;gap:10px;padding:6px 0;',
    '  border-bottom:1px solid var(--border-subtle);}',
    '.mp-row:hover{background:var(--surface-3);}',
    '.mp-muted{color:var(--text-secondary);font-size:12px;}',
    '.mp button{border:1px solid var(--border-primary);border-radius:var(--rayon-2);',
    '  background:transparent;color:var(--text-primary);padding:6px 12px;',
    '  font:inherit;cursor:pointer;}',
    '.mp button:hover{background:var(--brand-tint-50);color:var(--brand-primary);}'
  ].join('');
  document.head.appendChild(st);
}
```

The application's geometry tokens, such as `--rayon-2`, are not part of the exposed list. Writing `var(--rayon-2)` is harmless, the browser's fallback value applying, but it is better to provide an explicit fallback when the shape matters.

Six rules that prevent the most common failures

The root is emptied before every render. The panel rebuilds itself entirely on each opening, and anything it wants to keep must live elsewhere: in a plugin variable, which lives as long as the instance, or in its storage, which survives a restart.

One visible panel at a time. The plugin has one iframe. Opening a second panel closes the first, explicitly. Two simultaneous surfaces call for two plugins.

No request to the local server. The content security policy blocks it, capability or not. Data comes through `samples:read`, or is entered by the user, or travels in the package.

No `localStorage` for anything large. The compatibility facade makes it usable, but it writes into the plugin's partitioned storage, capped at two megabytes. A reference dataset belongs in the assets folder, not in storage.

Events do not cross the boundary. A context menu drawn inside the sandbox will never meet the application's: they are two separate documents. That is good news, and it saves any precaution on that front.

Write both languages. A plugin's interface is read in French as well as in English. The language arrives in the storage snapshot, and reading it at load time rather than on every render means never having to think about it again.

27. A complete plugin

A real plugin, short, using most of what precedes: a panel, a menu entry, sample reading, a setting kept between sessions, and a notification. Nothing that does not appear in the earlier sections.

```
// manifest.json
{
  "id": "isotopic-spread",
  "name": "Isotopic Spread",
  "version": "1.0.0",
  "author": "Author name",
  "description": "Standard deviation of isotopic ratios, by element.",
  "category": "analysis",
  "entry": "plugin.js",
  "capabilities": ["samples:read", "ui:menu", "ui:panel",
                  "ui:notify", "storage:read", "storage:write"],
  "contributes": {
    "menus": [ { "id": "open", "label": "Open Isotopic Spread" } ],
    "panels": [ { "id": "main", "label": "Isotopic Spread", "dock": "right" } ]
  }
}
```

```

// plugin.js

// The language is read once, at load time: it arrives with the storage
// snapshot, so with no round trip.
var fr = (localStorage.getItem('isofind_language') || 'fr') !== 'en';
function t(f, e) { return fr ? f : e; }

// The setting lives in the plugin's storage. It survives being
// disabled, not being uninstalled.
var DEFAULT_MINIMUM = 3;
var minimum = DEFAULT_MINIMUM;

function styles() {
  if (document.getElementById('is-style')) return;
  var st = document.createElement('style');
  st.id = 'is-style';
  st.textContent = [
    '.is{font:13px/1.55 system-ui,-apple-system,"Segoe UI",sans-serif;',
    '  color:var(--text-primary);padding:14px 16px;}',
    '.is h3{margin:0 0 4px;font-size:14px;color:var(--brand-primary);}',
    '.is p{margin:0 0 12px;color:var(--text-secondary);font-size:12px;}',
    '.is-row{display:flex;gap:10px;padding:6px 0;',
    '  border-bottom:1px solid var(--border-subtle);}',
    '.is-el{flex:1;font-weight:600;}',
    '.is-v{font-variant-numeric:tabular-nums;}',
    '.is-n{color:var(--text-muted);font-size:12px;width:52px;text-align:right;}'
  ].join('');
  document.head.appendChild(st);
}

// The computation needs no capability: it works on what the host has
// already delivered.
function spread(samples) {
  var byElement = {};
  samples.forEach(function (s) {
    (s.isotope_data || s.isotopic_data || []).forEach(function (m) {
      var v = parseFloat(m.ratio_value);
      if (!isFinite(v) || !m.element) return;
      (byElement[m.element] = byElement[m.element] || []).push(v);
    });
  });

  return Object.keys(byElement).map(function (el) {
    var xs = byElement[el];
    var mean = xs.reduce(function (a, b) { return a + b; }, 0) / xs.length;
    var variance = xs.reduce(function (a, b) {
      return a + (b - mean) * (b - mean);
    }, 0) / xs.length;
    return { element: el, n: xs.length, sd: Math.sqrt(variance) };
  }).filter(function (r) {
    // A standard deviation over two measurements means nothing. The
    // threshold is a setting, because it depends on what the user is
    // analysing.
    return r.n >= minimum;
  }).sort(function (a, b) { return b.sd - a.sd; });
}

function paint(root, rows) {
  // The root arrives empty: the host clears it before every render.

```

```

var d = document.createElement('div');
d.className = 'is';

var h = document.createElement('h3');
h.textContent = t('Dispersion par élément', 'Spread by element');
d.appendChild(h);

var p = document.createElement('p');
p.textContent = t('Écart-type des rapports mesurés, à partir de '
    + minimum + ' mesures.',
    'Standard deviation of measured ratios, from '
    + minimum + ' measurements.');
d.appendChild(p);

if (!rows.length) {
    var empty = document.createElement('p');
    empty.textContent = t('Rien à montrer avec ce seuil.',
        'Nothing to show at this threshold.');
    d.appendChild(empty);
    root.appendChild(d);
    return;
}

rows.forEach(function (r) {
    var line = document.createElement('div');
    line.className = 'is-row';

    var el = document.createElement('span');
    el.className = 'is-el';
    el.textContent = r.element;

    var v = document.createElement('span');
    v.className = 'is-v';
    v.textContent = r.sd.toFixed(4);

    var n = document.createElement('span');
    n.className = 'is-n';
    n.textContent = r.n + ' ' + t('mes.', 'meas.');
    line.appendChild(el);
    line.appendChild(v);
    line.appendChild(n);
    d.appendChild(line);
});

root.appendChild(d);
}

isofind.registerPanel('main', 'Isotopic Spread', function (ctx) {
    // The render receives { root, id }: ctx.root is what to populate.
    var root = ctx.root;

    styles();

    var waiting = document.createElement('div');
    waiting.className = 'is';
    waiting.textContent = t('Lecture...', 'Reading...');
    root.appendChild(waiting);

```

```

// Two calls that cross the boundary, so two promises. The setting
// first, since the computation depends on it.
isofind.storage.getJSON('settings', { minimum: DEFAULT_MINIMUM })
  .then(function (s) {
    minimum = s.minimum || DEFAULT_MINIMUM;
    return isofind.getSamples();
  })
  .then(function (samples) {
    root.textContent = '';
    paint(root, spread(samples));
  })
  .catch(function (e) {
    // The message carries the missing capability when that is what
    // it is: "Capacite refusee: samples:read". Show it rather than
    // leaving an empty panel with no explanation.
    root.textContent = '';
    var err = document.createElement('div');
    err.className = 'is';
    err.style.color = 'var(--tone-rouge-text)';
    err.textContent = e.message;
    root.appendChild(err);
  });
});

isofind.registerMenu('open', 'Open Isotopic Spread', function () {
  isofind.openPanel('main');
});

// In a real version the setting would be changed from a field in the
// panel. The writing function is kept here to show how a full quota is
// handled: a refused write has to be reported, or the user believes the
// setting was saved.
function setMinimum(v) {
  return isofind.storage.setJSON('settings', { minimum: v })
    .then(function () {
      minimum = v;
      isofind.notify(t('Seuil enregistré', 'Threshold saved'), 'success');
    })
    .catch(function (e) {
      isofind.notify(e.message, 'error');
    });
}

```

28. Checklist before publishing

What a moderator will look at, and what costs the most to fix afterwards.

Point	What to check
Identifier	Stable and final. It serves as the install key, the storage namespace, the asset prefix and the source of contributions. Changing it amounts to publishing a different plugin, and losing users' storage.
Capabilities	None that is not used. Every extra box is a reason not to install, and a capability unticked out of caution will make the plugin fail later.

Contributions	All declared in the manifest, and all marked implemented by the inspector. A contribution registered without a declaration escapes the user's consent.
Metadata	Name, description and labels in English. These are the strings the catalogue, the consent screen and moderation see.
Interface	Bilingual, and built on the theme tokens. A plugin tested in the light theme only turns out unreadable for the next user.
Network	No egress that is not announced. A plugin that loads a resource from a CDN stops working on an isolated machine, which is the opposite of what the software promises.
Assets	In the folder named by <code>assets</code> , and nowhere else. The declared prefix is authoritative, for signing as much as for writing.
Startup	Do not ask to run at launch without a reason. It is allowed, the workshop flags it, and a moderator will ask why.
Errors	Every promise has a failure handler that shows the real message. An empty panel with no explanation gets blamed on the software.
Signature	The archive submitted is the one the service signed and handed back, not the unsigned version left next to it.
Identity	The certificate does not expire in the next few weeks. A signature produced with an expired certificate is refused.

The last reflex. Install your own signed archive on a fresh installation, as a user would, rather than stopping at the workshop. The workshop lifts the signature requirement, loads from a folder and writes no assets to the cache: three differences that are enough to make a broken plugin look like one that works.