Net0

The Assistant

Installation Guide

Putting the assistant on a government web page

August 2026

embed.agentic.ae

Contents

What this is

Two ways to put the assistant on a government web page, and how to install each.

Neither is specific to one ministry. Everywhere this document writes `{PLATFORM_ORIGIN}`, `{PORTAL_ORIGIN}` or `{WIDGET_KEY}`, substitute the values your platform team gives you; the last chapter lists what a ministry has to supply before its first install and what the platform team does in return.

Route A — the browser extension

An extension for Chrome and for Edge that puts the assistant on the page in front of you. Nothing changes on the ministry's website: the extension injects the panel from the browser side, and it appears only on the sites the person using it has switched on, one at a time.

Use this when

- the website cannot be edited yet, or a change to it needs a release you do not have;
- you are showing the assistant to somebody and want it on their own portal, live;
- staff need to work inside a ministry portal — including **filling the ministry's own form** from a conversation a citizen already had, which is the second half of this document and the reason the extension exists rather than being a demonstration toy.

It is an internal tool. It is not on the Chrome Web Store and is not meant to be: it is distributed inside the organisation, either by hand or by browser policy.

Route B — one script tag

A single `<script>` on the ministry's own page. This is the permanent integration and the one a citizen should eventually meet. It needs somebody who can edit the page and deploy it.

Use this when the site is yours to change and the assistant is going to stay.

Which to install

	Extension	Script tag
Changes the ministry's website	No	Yes — one tag
Who sees the assistant	Only the person who installed it	Every visitor
Works on a site you cannot edit	Yes	No
Can fill a ministry form	Yes	No
Survives the browser being reinstalled	Only if deployed by policy	Yes

	Extension	Script tag
Right answer for a pilot	Yes	
Right answer for production		Yes

They are not alternatives so much as a sequence. Most ministries start with the extension, because it can be running twenty minutes after the download finishes, and move to the script tag once the assistant has earned a place on the page.

What you need before you start

`{PLATFORM_ORIGIN}`	Where the assistant is served from, e.g. `https://assistant.example.gov.ae`.
`{WIDGET_KEY}`	Marks the embed as registered, which is what unlocks a durable identity, attachments and the dashboard. It travels in the page and a citizen can read it — it is not a secret.
`{PORTAL_ORIGIN}`	The ministry site you want the assistant on. Extension only.
A staff token	Only to fill forms. Nothing else in the extension asks for one, and with no token the extension is an embed host and nothing more.

Ask your platform team for the first two. An origin the platform does not recognise resolves to exactly what no key resolves to, so a key from one environment does not work in another — which is the intended behaviour and not a fault to work around.

Installing the extension

Chrome and Edge take the same package by the same steps. Where they differ, both are given.

Download the one for your browser, unzip it, and keep the folder somewhere permanent — the browser loads the extension **from that folder every time it starts**, so a package unzipped into Downloads and later tidied away stops working, and the failure looks like the extension having been uninstalled.

Browser	Package
Google Chrome 116 or later	`assistant-extension-chrome.zip`
Microsoft Edge 116 or later	`assistant-extension-edge.zip`

The two are the same code and differ only in the pinned identity, so both can be installed on one machine without either shadowing the other.

Chrome

1. Unzip the package into a folder you will keep, e.g. `C:\Assistant\chrome`.
2. Open `chrome://extensions`.
3. Turn on **Developer mode**, top right.
4. Press **Load unpacked** and choose the folder from step 1 — the folder that *contains* `manifest.json`, not the file itself and not the zip.
5. The extension appears in the list. Pin it: press the jigsaw-piece icon beside the address bar and press the pin beside the assistant.

Edge

1. Unzip the package into a folder you will keep, e.g. `C:\Assistant\edge`.
2. Open `edge://extensions`.
3. Turn on **Developer mode**, in the left-hand column.
4. Press **Load unpacked** and choose the folder from step 1.
5. Pin it from the jigsaw-piece icon beside the address bar.

Edge shows “This extension is not from the Microsoft Store” and asks you to keep or remove it. Keep it. There is no store listing, deliberately — see *Distribution* below.

What it asks for at install time

Nothing about your browsing. The permissions it holds on installation are storage, the ability to inject its own code when asked, and the ability to act on the tab you are looking at when you press the toolbar button.

It has access to no website until you grant one. The whole tool is “put the assistant on the page I am looking at”, so a permission over every site you ever open — granted once, at install, in exchange for a panel you wanted on four of them — would be the wrong trade. Access is requested per site, at the moment you switch that site on, where you can see which site it is. The browser will ask, by name; Chrome words it *“Read and change your data on* `portal.example.gov.ae` *”*.

Revoking is the same door in reverse: switch the site off in the popup, or remove the host from the extension’s own **Site access** settings on the extensions page.

Distribution

Loading unpacked is right for a pilot and wrong for forty machines. Three routes, in the order ministries normally take them:

By hand. The steps above. Fine for a handful of people who can be told when there is a new version.

By browser policy, which is how this should reach a ministry's estate. Both browsers can install an extension from a URL your organisation controls, keep it updated, and stop a user removing it. The package is served as a `.crx` alongside an update manifest; the pinned identity in each build is what makes this possible, because the extension id is the same on every machine.

Browser	Policy
Chrome	`ExtensionInstallForcelist`, or `ExtensionSettings` with `installation_mode: force_installed`
Edge	`ExtensionInstallForcelist` under the Edge policy tree

Ask the platform team for the `.crx` and the update URL; they are not on the download page, because a self-hosted `.crx` is signed with a private key and a download page is the wrong place to hand one out.

By store. Neither browser's store is used today and neither is planned. A public listing invites the public to install an internal tool.

Updating

An unpacked extension does not update itself. Replace the contents of the folder with the new package and press **Reload** under the extension on the extensions page, or restart the browser.

Your settings survive an update — they are held in the browser profile, not in the folder — with one deliberate exception. **The sites you enabled do not survive being uninstalled and reinstalled**, because the browser drops the host permissions with the extension. That is the permission model working, and it means a reinstall is followed by switching your site back on.

First run

Press the toolbar button. Everything is configured in the popup and nowhere else.

The popup is also the only place a site is switched on and the only place a fill run starts, and that is not an arrangement of convenience: the browser will only grant access to a site in response to a human press, and this is the one surface the extension has that has one.

Step 1 — Point it at the platform

Under **Platform**:

Field	Value
Origin	`{PLATFORM_ORIGIN}`
Widget key	`{WIDGET_KEY}`
Staff token	Leave empty unless you are filling forms — Chapter 4

Fields save when you leave them, not as you type. The popup says *"Saved. Reload any open page to pick it up."*

Step 2 — Switch on the site you want it on

Open the ministry site in a tab, then open the popup. It reads the tab's address and shows the origin under **This site**.

Press **Switch on for this site**. The browser asks for access to that host, by name. Accept it. If you decline, the popup says so rather than saving a setting the extension cannot act on.

The assistant appears on the next page load, and on any tab of that site already open.

> *Some pages cannot host it and no extension can change that: the browser's own pages (`chrome://`, `edge://`), the extension store, and local files. The popup says "This page cannot host the assistant" rather than offering a switch that would do nothing.*

Step 3 — Choose how it looks

Under **Panel**:

Field	Default	
Language	Follow the page	Takes the host document's own `lang`, so an Arabic page opens an Arabic assistant.
Launcher	Centred pill	A pill at the foot of the page, a corner bubble, or nothing.
Size when opened	Expanded	Docked, expanded or full screen.

If the panel does not appear

Work down this list. The first two causes account for nearly everything.

The site is not switched on. The popup's top section says which. A page loaded before you switched it on needs a reload.

The widget key is not registered for this origin. Ask your platform team to confirm the key is issued for `{PLATFORM_ORIGIN}`. A key presented from an origin the platform does not serve resolves to exactly what no key resolves to, on purpose.

The page blocks the panel. Rare, and there is a switch for it — see below.

The Content-Security-Policy switch, and when to touch it

Under **This site** there is a checkbox: *"Remove this site's Content-Security-Policy header."*

Leave it alone unless the panel will not load. The panel is drawn inside a frame that belongs to the extension, and an extension's own resources are normally exempt from the host page's policy — so this is an escape hatch for a page that blocks it anyway, not a step in the installation.

When it is ticked, that site's protection against script injection is off — for that site, for as long as it is ticked, until the browser restarts. It cannot be ticked for a site that is not switched on, and it is dropped when you switch the site off. If you find yourself needing it on a ministry portal, tell the platform team: the right fix is on the platform's side and this is the workaround while it is made.

Filling a ministry's own form

For a handful of services there is no application programming interface to file against — no contract the platform can call, only the ministry's own web form. For those, a citizen still has the whole conversation with the assistant, and a member of staff then completes the ministry's form from the answers already given, in their own signed-in session, with the extension typing and a person watching.

This is off by default and needs two things that nothing else in the extension needs: a staff token, and a decision.

Switching it on

Under **Filling forms**, *What the extension may do*:

Nothing — assistant only	The default. Nothing is typed into any form, ever.
Dry run	Resolves a real plan against the real page and paints what it *would* write. Writes nothing.
Fill the form	Writes values.

Then put your own staff token under **Platform**. It is the one genuinely sensitive value the extension holds, and it is why filling defaults to off: with no token, the extension is an assistant host and nothing more.

With both set, the popup lists what is waiting — one row per application whose conversation is finished and whose form has not yet been filed. Each row has **Open form**, which opens the ministry's own entry page for that service, and **Fill**.

The list is deliberately silent while filling is off. An operator who has not switched it on has not asked to see a list of citizens' applications, and fetching one would send their token for no reason.

What you see while it runs

A panel draws on the ministry's page: the step it is on, each control as it is written, and a count. It is not decoration — a run that halts has to say so somewhere, and a halt nobody can see is a run that appears to have worked.

A dry run paints the same panel with what it *would* have written.

Where it stops

Before the ministry's Submit, unless that service is explicitly configured otherwise *and* you confirm it in the same session. Both, every time. The default for every service is to fill every step and stop, so that the last act on a government form is a person's.

The confirmation is asked again on every run. Nothing remembers that you agreed once — a run that persisted "the operator already said yes" is how one confirmation files a second application, so the answer is never on record.

The rules it keeps

Each of these is a way to file the wrong thing on a citizen's behalf, so each is enforced in the code rather than described in a procedure. They are worth knowing because they explain what a refusal means when you see one.

1. **It never fills a control the plan does not name.** No matching on labels, no "nearest box to this text". Every field is written by the `name` or `id` captured from that form. A required field it cannot find **halts the run** and says which. A guess that put a date of birth in an expiry field would look exactly like success until a ministry acted on it.
2. **It never fills a page whose address does not match the step.** A form that looks right can belong to a different service — the same service has different numeric ids in staging and in production — so the address is checked on every step, not once at the start.
3. **It never presses Submit** without both the service's own policy and your confirmation.
4. **It never retries a step that may already have written.** Once the form has been advanced, the run makes no further decision about that step: there is no second press and no timeout that would produce one. A retry is how one application becomes two.
5. **The citizen's answers are never stored.** They are fetched for the step, held in memory, and gone when it finishes. Nothing is written to browser storage, and there is a test that reads the shipped code to confirm it.

When a run halts

A halt is the system working. The panel names the control it could not resolve; send that to the platform team, because the cause is a gap in what was captured from that form and the fix is on the platform's side.

The useful thing to do first is a **dry run**, which reaches every check above and none of the writes. It is the cheapest way to find out whether a service is completely captured without altering a government form to discover it.

One honest limit: a dry run stops before advancing, so it can only report on the step you are looking at. Pressing Next to see the next step's markup is itself a write — these portals save the form when they advance — and a "dry" run that did that would not be one.

What a filed application is called

The number the ministry issues is the only one that works at a counter, and the extension reads it off the confirmation page and sends it back to the platform. Until the ministry has issued one there is no number to quote, and the platform says so rather than showing its own record locator as if it were one.

Route B — one script tag

The permanent integration. No extension, nothing for a citizen to install, and the assistant is there for every visitor.

The tag

```
<script
  src="{PLATFORM_ORIGIN}/embed.js"
  data-platform-origin="{PLATFORM_ORIGIN}"
  data-widget-key="{WIDGET_KEY}"
  async
></script>
```

Around four kilobytes over the wire. It builds no frame and makes no request until a visitor opens the assistant, so a page that carries it and is never used pays only for the script.

`data-platform-origin` is the only attribute that must be present. Without it the script installs nothing and returns quietly, so a page copied into an environment with no assistant does not throw.

Check it is registered

```
curl -s -H 'x-embed-key: {WIDGET_KEY}' {PLATFORM_ORIGIN}/v1/embed/config
```

```
{ "trusted": true, "signIn": "uae_pass", "languages": ["en", "ar"] }
```

If `trusted` is `false`, stop. No sign-in is possible and the dashboard will not appear. The cause is almost always the origin: the key has to be issued for the origin the page is served from, and the platform compares origins exactly rather than by prefix — `https://example.gov.ae.attacker.test` begins with the string a careless check would look for.

Open it from your own buttons

This is the recommended entry point, and the reason is worth stating: the assistant's value is that somebody presses a thing they were already looking at and lands in a conversation about that service, rather than on a menu.

```
MoeiAssistant.open({ service: 'your-service-slug' });
MoeiAssistant.open({ view: 'dashboard' });
MoeiAssistant.open({ view: 'tracking' });
```

The ministry's existing "Start service" link stays where it is; this sits beside it as a second route to the same outcome.

The script also draws a launcher of its own — a pill at the foot of the page, a corner bubble, or nothing. It is a reasonable catch-all and it is not where the value is. Set `data-launcher="none"` if the ministry would rather have only its own buttons.

One thing the platform team must do

The assistant is drawn in a frame, and a frame is only allowed to be embedded by pages the platform names. **Your page's origin has to be on that list** before the tag will work, and it is set on the platform at deploy time rather than by anything on your page.

Send the platform team the exact origin — scheme, host and port if it is not the default. The symptom of a missing one is a panel that stays blank with a `frame-ancestors` message in the browser console, which reads like a fault in the assistant and is not.

Language

Language follows the host document's own `lang` attribute rather than a constant, so an English page opens an English assistant and an Arabic page an Arabic one, right to left, without a second integration. Override with `data-lang="ar"` if you need to.

Adding a ministry

Nothing in this document is specific to one ministry, and nothing in the extension is either. This chapter is what changes when a second one takes it — written down because the answer is short and being told it is short is the useful part.

What the ministry supplies

The origin of the portal	Exactly as served: scheme, host, and port if not the default.
A test account on it	With permission to open the forms in question, and to file nothing.
The services wanted	By name, as the portal names them.
Who will operate it	The people who will hold a staff token, if forms are to be filled.

What the platform team does

Issues a widget key for the ministry's origin, and adds that origin to the list of pages allowed to embed the assistant frame. Both are configuration, not code.

Captures the forms, if filling is wanted. Somebody walks each service on the ministry's portal and records every control, its options, and which order the steps come in. That capture becomes a *site pack* — the description of one portal that the extension resolves a form against — and it is the only per-ministry artefact of any size. One pack covers a whole portal, not one service.

Binds each field to the question the assistant already asks, so an answer given in a conversation has somewhere to go. A field that is required on the form and bound to nothing is refused at build time rather than discovered by an operator: a check runs on every commit and fails if any service configured to be filled could not actually be filled.

Sets the submit policy per service. The default is to fill every step and stop before the ministry's own Submit. A service is moved off that default deliberately, one at a time, and never as a blanket setting.

What does not change

The extension. There is one codebase, one Chrome build and one Edge build, and a second ministry does not fork it. The portal it works on, the services it knows, the forms' field names and the order of

their steps are all data the extension fetches from the platform at run time and caches; a new ministry is a new pack, not a new extension.

The product name shown in the toolbar is a build setting rather than something compiled in, so a ministry that wants its own name on it gets one by changing a line of configuration and rebuilding — not by a code change and not by a second repository.

The order to do it in

1. Origin registered and widget key issued. The assistant now works on the ministry's portal through the extension, with filling off. This is a day's work and is worth doing on its own, because it is what the ministry will want to see first.
2. Forms captured for the services that matter. Dry runs against each, until every required field resolves.
3. Filling switched on for a named group of operators, stopping before Submit.
4. The script tag, when the ministry is ready to put the assistant on its own page for every visitor. The extension stays useful after this — it is how staff fill a form the platform has no contract to file against, and that does not change when a citizen-facing panel goes live.

Reference

Every setting in the popup

Setting	Default	What it does
Origin	—	Where the assistant is served from.
Widget key	empty	Marks the embed as registered. Not a secret.
Staff token	empty	Only for filling forms. Sent to the platform's fill endpoints and nowhere else.
Language	Follow the page	Or force English or Arabic.
Launcher	Centred pill	Pill, corner bubble, or none.
Size when opened	Expanded	Docked, expanded, full screen.
What the extension may do	Nothing	Nothing, dry run, or fill.
Sites	none	One grant per site, requested when you switch it on.
Remove this site's CSP	off	Escape hatch. Per site. Reset when the browser restarts.

Settings follow the browser profile, so somebody with the extension on two machines signed into the same profile configures it once. The sites you enabled are held as browser permissions rather than as a setting, which is why they do not survive a reinstall.

Troubleshooting

Symptom	Cause, in order of likelihood
No panel on the page	The site is not switched on; or the page was loaded before you switched it on — reload.
No panel, and the site is on	The widget key is not registered for this origin. Ask the platform team.
Panel frame stays blank	The extension's origin is not on the platform's embed allowlist, or the page's own policy is blocking it — see the CSP switch in Chapter 2.
Popup says "This page cannot host the assistant"	A browser page, the extension store, or a local file. No extension can reach these.
Nothing listed under "Waiting to be filled"	Filling is off; or no staff token; or nothing is genuinely waiting. The popup distinguishes all three.
A run halts naming a control	The form was not completely captured. Send the message to the platform team.
Extension vanished after a restart	The folder it was loaded from moved or was deleted.
Sites all switched off after reinstalling	Expected — the browser drops host permissions with the extension.

What the extension can and cannot see

It holds no access to any website until you grant one, per site, by name. Within a site you have granted, it can read and change the page — which is what filling a form means, and there is no smaller permission that would allow it.

It cannot see your other tabs. Opening the popup grants access to the one tab you are looking at for the life of that popup, which is how it reads the address at the top; there is no permission here that would let it enumerate the rest.

The panel's own conversation is inside a frame belonging to the platform. The citizen's session lives in there and nothing in the extension can reach it — which is correct, and is why filling a form authenticates as staff instead: reading an application an operator is already entitled to see is the same authority they have in the console, not a new one.

Requirements

Chrome	116 or later
Edge	116 or later

Firefox, Safari	Not supported. Both differ enough in the extension platform to be a separate build rather than a setting.
Operating system	Any the browser runs on.

Words used here

Widget key — marks an embed as registered for an origin. Travels in the page; not a secret.

Site pack — the captured description of one ministry portal: its addresses, its forms' controls, and the order of their steps. One per portal.

Fill plan — one application's answers, converted to the values that portal's form expects. Fetched per step, held in memory, never stored.

Submit policy — per service, whether the run may press the ministry's own Submit. Defaults to no.