Cascale

HIGG ASSESSMENT MODEL

Evaluating Assessments via the HAM HTTP API

Version ham-v0.4.0 · 25 September 2026

This document explains how to call the stl-local-server HTTP API to evaluate a Higg assessment against a pinned HAM version. The audience is integrators wiring an application or service against a hosted HAM evaluator rather than embedding HAM into their…

Contents

When to Use the API vs. the Bundle **3**
Resource Model **3**
Typical Integrator Flow **3**
Filtering the response 4
Evaluating a Higg-platform assessment by reference 4
Inspecting a single block's source 5
Version Selection **5**
Error Model **6**
Refresh and Hot-Reload **6**
Deployment **6**
What's NOT in This API **7**
Sanity-checking against the QA tool **7**

This document explains how to call the `stl-local-server` HTTP API to evaluate a Higg assessment against a pinned HAM version. The audience is integrators wiring an application or service against a hosted HAM evaluator rather than embedding HAM into their own runtime.

If you instead want to consume the HAM bundle directly in-process (load `model.json.gz`, build your own evaluator, run blocks in your own JavaScript environment), `usage.md` is the document you want.

When to Use the API vs. the Bundle

The two paths produce the same numeric result for the same `(version, rfi_pid, assessment)` triple. Choose based on operational fit:

- **Use the HTTP API** when you want HAM versioning, bundle distribution, evaluator pooling, and the platform-fetch helper handled for you. The server picks up new releases on a refresh tick — no application redeploy required. Most integrations should start here.
- **Use the bundle directly** when you need to embed evaluation into a hot path that can't afford an HTTP round trip per assessment, when you're operating in an environment without outbound HTTP to a HAM server, or when you want to extend evaluation behaviour beyond what the server exposes. See `usage.md`.

A single application can mix the two — use the API for online scoring, bundle-embedded evaluation for batch backfills, for example.

Resource Model

```
GET   /v1/health
GET   /v1/ham/versions
GET   /v1/ham/{version}/cadences
GET   /v1/ham/{version}/schema
GET   /v1/ham/{version}/cadences/{rfi_pid}/blocks/{name}
POST  /v1/ham/{version}/cadences/{rfi_pid}/evaluate
POST  /v1/ham/{version}/platform/evaluate
```

The `{version}` path segment is resolved against the server's current manifest at request time — see *Version Selection* below. The `{rfi_pid}` segment names a cadence (e.g. `fem2025`, `fem2026`).

All request and response bodies are JSON. The server emits `Content-Type: application/json; charset=utf-8` on every response.

Typical Integrator Flow

The minimum useful sequence is:

1. **Discover available versions.** `GET /v1/ham/versions` returns `{ "versions": ["1.2", "1.3", "next"] }`. Pick the version your application has agreed to pin to.
2. **(Optional) Discover available cadences for that version.** `GET /v1/ham/1.3/cadences` returns `{ "version": "1.3", "cadences": ["fem2024", "fem2025", "fem2026"] }`. The exact list depends on which cadences the HAM build included.

3. **(Optional) Fetch the schema.** `GET /v1/ham/1.3/schema` returns the raw JSON schema. Useful for code-generation or input validation; not required for evaluation.
4. **POST an assessment to be evaluated.**

```
POST /v1/ham/1.3/cadences/fem2025/evaluate
Content-Type: application/json

{
  "assessment": { ... flat answer map ... },
  "select": ["fep_achievement", "market_eeci"]
}
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "version": "1.3",
  "rfi_pid": "fem2025",
  "result": { ... evaluated blocks, optionally filtered by `select` ... }
}
```

The `assessment` request field is the flattened answer map — keys are the question's ref IDs (`fp-step-1`, `enelectricity`, etc.) and values are the facility's responses. The exact shape is what the HAM evaluator expects as its evaluation context; the response schema is what `schema.json` describes.

Filtering the response

The `select` request field accepts a list of dotted paths into the result. When provided, the server walks each path and assembles a minimal subset of the full evaluation. Omit `select` (or pass `null`) to receive every field every block produced.

Use `select` to keep responses small when only a handful of fields matter for your UI surface — for example, a dashboard card that needs `assessment.market_eeci` and a handful of GHGP fields can skip the per-row `EnergyUse` block payload.

Evaluating a Higg-platform assessment by reference

When the assessment you want to evaluate already lives in the Higg platform, you can have the server fetch it for you in a single call:

```
POST /v1/ham/1.3/platform/evaluate
Content-Type: application/json

{
  "env": "stage",
  "account": "<account-id>",
  "assessment": "femsurvey:<uuid>",
  "user": "<email>",
  "select": [...]   // optional, same semantics as above
}
```

Response (200):

```
{
  "version": "1.3",
```

```
  "rfi_pid": "fem2025",
  "result": { ... },
  "answers": { ... the flattened answer map the server fetched ... }
}
```

The `rfi_pid` is derived from the platform record — the server picks the correct cadence — so you don't pass it on this endpoint. The `answers` field on the response is the same shape you'd pass to `/cadences/{rfi_pid}/evaluate`; surface it in a debug view, log it against a request ID for replay, or feed it back into a direct evaluate call if you need to make a synthetic change.

This endpoint is gated on the server having a platform access token configured (see *Deployment* below). If the token isn't present the endpoint returns `500` with an error message.

Inspecting a single block's source

`GET /v1/ham/{version}/cadences/{rfi_pid}/blocks/{name}` returns the compiled source for a single block:

```
{
  "name": "assessment",
  "kind": "Assessment",
  "context": {
    "<field>": "<source expression as a string>",
    ...
  }
}
```

This is the same data the bundle's `model.json.gz` carries for that block, served per-block so a debug surface doesn't need to fetch the full model. Use this when you want to display "where does this number come from" in a UI — render the block's `context` next to the evaluated output.

Version Selection

The `{version}` path segment accepts three forms:

- **Exact numeric.** `1.2` resolves to manifest entry `1.2` if it exists.
- **Major-only numeric.** `1` resolves to the *highest minor* under `1.*` in the manifest. Useful when an application wants to track a major version without redeploying for each minor bump. Major-only resolution is numeric-only — `1` will never resolve to a channel (`next`, `preview`).
- **Channel name.** `next` resolves to the manifest entry named `next`, if present. Channels are conventionally lowercase identifiers (letters, digits, hyphens; must start with a letter). `next` is the standard channel for the latest-in-progress release.

If none of these resolve, the server returns `404` with `{"error": "version not found: <requested>"}`.

Version resolution happens on every request; the server does not cache resolution per-client. A request for `1` will start returning results from a freshly-released `1.4` as soon as the server's next refresh tick picks up the new manifest — no client-side change needed.

Error Model

Every error response is a JSON object with a single `error` field holding the human-readable message.

HTTP status	When it fires	`error` shape
`400`	Invalid version format (e.g. `1.x`)	`"invalid version format: 1.x"`
`400`	Evaluation error (malformed input, missing required field, JS exception)	`"evaluation error: <details>"`
`404`	Unknown version	`"version not found: 99"`
`404`	Cadence not in this version's bundle	`"cadence not found: <rfi_pid>"`
`500`	Storage backend failure (B2 unreachable, etc.)	`"storage error: <details>"`
`500`	Bundle decode failure	`"artifact decode error: <details>"`
`500`	Evaluator pool / channel failure	`"internal error: <details>"`

The error message string is intended to be human-readable. It is *not* a stable contract — don't pattern-match against the wording. For programmatic dispatch, use the HTTP status code.

A bad evaluation context (missing fields, wrong types) surfaces as `400` with an `evaluation error: ...` message that quotes the underlying evaluator error. These are typically actionable client bugs.

Refresh and Hot-Reload

The server polls the artifact bucket on a fixed interval (default 60 seconds; configurable via `STL_REFRESH_SECS`). On each tick it re-reads the manifest and re-validates ETags on the bundles it has cached. Behaviour:

- **New version added to manifest:** the new version becomes resolvable on the next request. No restart.
- **Existing version's bundle revised (patch release):** the in-memory cached bundle for that version is evicted. The next request for that version triggers a lazy refetch from the storage backend. No restart.
- **Channel re-pinned (e.g. `next` now points at a different artifact):** treated the same as a patch revision — the server evicts and refetches.

Practical consequence: rolling out a new HAM release does not require a server restart or any application-side coordination. The server picks the release up within the refresh interval.

Set `STL_REFRESH_SECS=0` to disable the refresh task entirely — the server then uses whatever bundles it had at startup until restarted. Don't disable this in production.

Deployment

A full deployment guide is available from Cascale for organisations hosting their own HAM server. The essentials:

- **Storage backend.** The server reads HAM bundles from either a local directory (`STL_STORAGE=local` + `STL_LOCAL_ROOT`) or an S3-compatible bucket (`STL_STORAGE=s3` plus `STL_S3_ENDPOINT`, `STL_S3_BUCKET`, optionally `STL_S3_REGION` and `STL_S3_CACHE_DIR`). Production uses the S3 backend pointed at Backblaze B2.

- **Bind address.** `STL_BIND` (default `0.0.0.0:8080`).
- **Refresh interval.** `STL_REFRESH_SECS` (default 60s, see above).
- **Platform fetch.** `STL_PLATFORM_TOKEN_ACCESS` is required if you want `/platform/evaluate` to work. Without it the endpoint returns 500.
- **Authentication.** The server itself is open by default. In production we front it with Caddy + oauth2-proxy — see the deployment guide. Don't expose the server unauthenticated to the public internet.

The server is a single static binary and ships as a systemd unit; the deployment guide has the canonical unit definition.

What's NOT in This API

A few things you might expect that aren't here, plus the answer for each:

- **No write endpoints.** The API is read-only — assessments are evaluated, not persisted. Use the Higg platform for storage.
- **No async / job submission.** Every `evaluate` call runs synchronously. Typical p50 is well under 100ms; p99 depends on cadence size and JS expression density. If you need to evaluate thousands of assessments, run them in parallel — the server pools evaluators per `(version, rfi_pid)` pair.
- **No batch endpoint.** Submit one assessment per request. If throughput is a concern, parallelism on the client side is the right answer; the server can handle simultaneous evaluations against the same `(version, rfi_pid)` pool.
- **No model-introspection endpoints beyond schema and per-block source.** If you need richer introspection, consume the bundle directly — `model.json.gz` has the full block tree.
- **No webhook / change-notification surface.** Poll `/v1/ham/versions` if you need to detect new releases from the client side.

Sanity-checking against the QA tool

The server's root URL (`/`) serves a side-by-side debug UI. Pointing a browser at the server's bind address lets you eyeball a `platform/evaluate` round-trip — flat answers in one pane, evaluated result in the other. Useful for verifying that the version your application is pinned to behaves the way you expect against a known assessment before wiring it into production.