BLUE FAIRY DEVELOPMENT

BlueFairy API Gateway

REST response caching, per-IP rate limiting and signed webhooks for WordPress

Version 1.0.4 · GPL-2.0-or-later · WordPress 6.6+ · PHP 8.1+
bluefairydevelopment.com

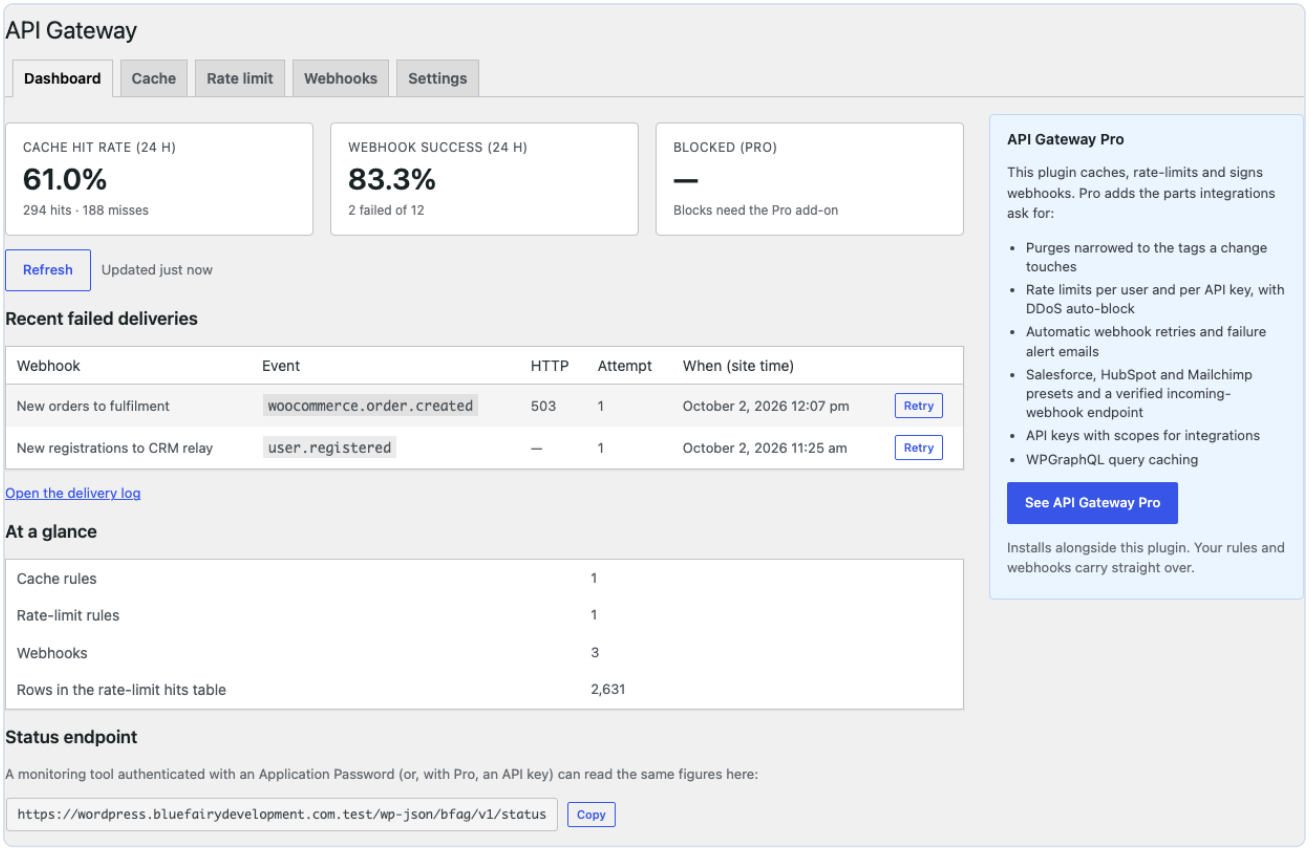
BlueFairy API Gateway — Feature Guide

Version 1.0.4 | GPL-2.0-or-later | PHP 8.1+ | WordPress 6.6+

BlueFairy API Gateway sits in front of the WordPress REST API and does three things: it caches anonymous responses, it rate-limits clients, and it sends signed webhooks when content changes. Everything is managed from one screen, **Settings → API Gateway**, with five tabs: **Dashboard**, **Cache**, **Rate limit**, **Webhooks** and **Settings**. A panel on the right of every tab describes the Pro add-on and disappears the moment Pro is active.

Every action on that screen — a saved rule, a deleted webhook, a purge — is protected by a nonce and by the `bfag_manage_gateway` capability, which the plugin grants to the Administrator role on activation.

Dashboard



The dashboard: cache hit rate, webhook success, recent failed deliveries with Retry

Three cards, each read from a transient that the five-minute KPI cron recomputes, so opening the dashboard never runs aggregate queries itself.

Card	Figure	Sub-line
Cache hit rate (24 h)	hits ÷ (hits + misses) over the last 24 hours, as a percentage	`N hits · M misses`
Webhook success (24 h)	successful ÷ (successful + failed) **outgoing** deliveries over the last 24 hours — an inbound row (Pro's receiving endpoint) is someone else's call, not a delivery this site made, and is counted in neither figure	`F failed of T` — T counts pending and retrying deliveries too
Blocked (Pro) / **Active blocks**	without Pro the card shows `—` and says Blocks need the Pro add-on; with Pro it is the number of addresses currently blocked (the KPI array may rename it through `blocked_label`, and Refresh applies the name from the answer too — without one the card says Active blocks again)	with Pro: `R hits in the rate-limit log` — the row count of the hits table, as of the last KPI run

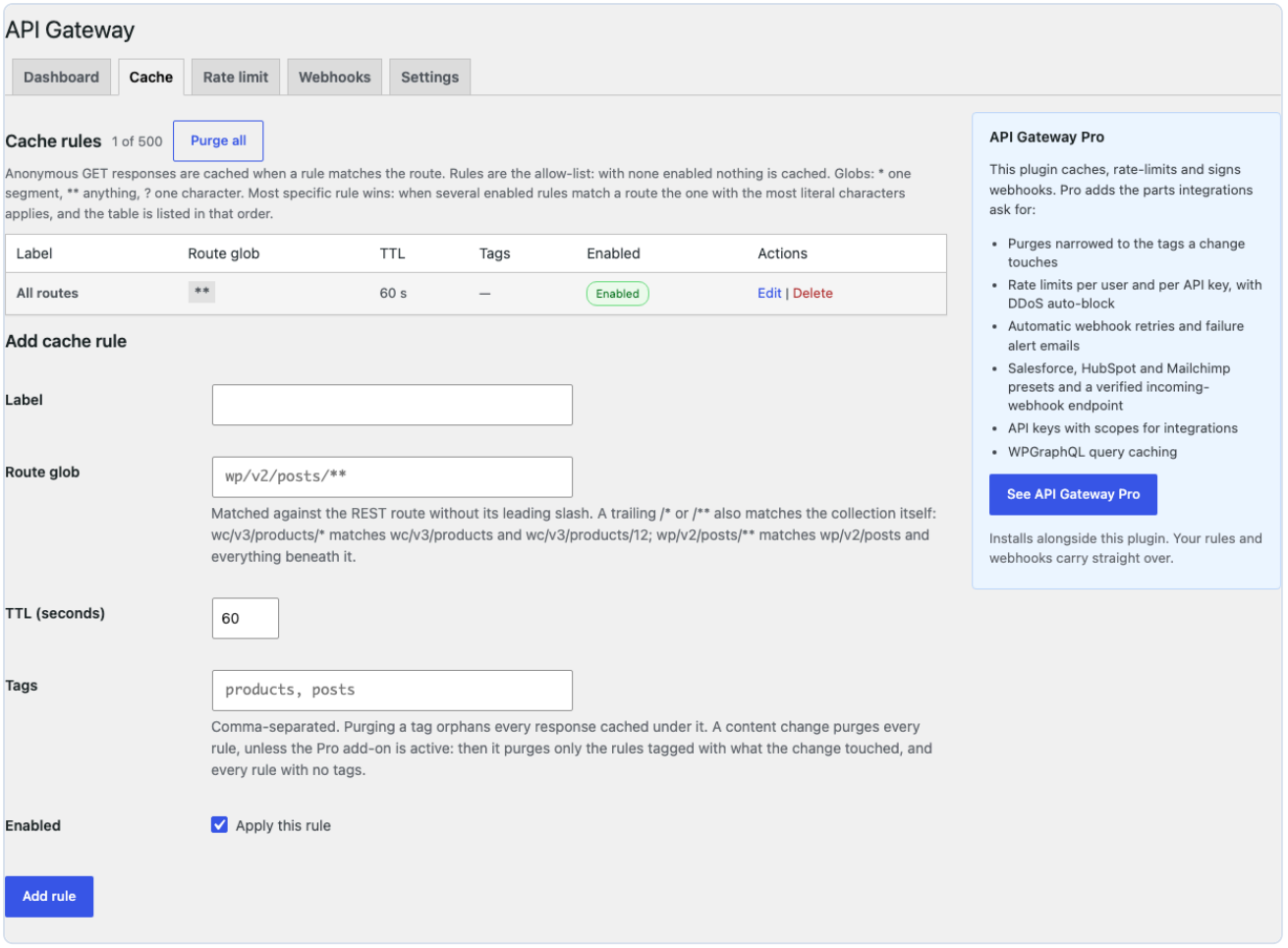
A card shows `—` when there is nothing to divide: no requests, or no finished deliveries. Below the cards, **Refresh** recomputes the figures immediately (the sub-line says Updated N ago, or Not computed yet before the first cron run, and Updated just now once Refresh has answered — said once, not followed by a second Updated.).

Recent failed deliveries lists the ten newest deliveries with status `failed`: webhook, event, HTTP status (`—` for a transport error), attempt number, time (in the site's timezone and formats — the header says site time), and a **Retry** button that resends the stored payload. The button follows the delivery log's rule: an **outgoing** `failed` (or Pro `retrying`) row that was not truncated. A delivery whose stored payload was truncated shows Truncated — cannot retry instead; an inbound row (Pro's receiving endpoint) is listed without a button, because the plugin will not sign someone else's payload with the site's secret. Open the delivery log goes to the full log on the Webhooks tab.

At a glance gives the raw counts: cache rules, rate-limit rules, webhooks, rows in the rate-limit hits table (from the last KPI run — a dash until the first one — so opening the dashboard never counts that table live).

Status endpoint shows the URL of `GET /bfag/v1/status` with a **Copy** button. A monitor authenticated with the gateway capability reads the same figures there; that route is never rate-limited.

Cache



The Cache tab: the seeded All routes rule, its lifetime, and Purge all

The rules table

Cache rules shows every rule with its label, route glob, TTL, tags, an **Enabled** / **Disabled** badge that toggles the rule when clicked, and **Edit** | **Delete** links. The table is in match order: **most specific rule wins** — when several enabled rules match a route, the one with the most literal (non-wildcard) characters in its glob applies, a tie going to the older rule — so a `wp/v2/posts/**` rule added beside the seeded `**` rule takes the posts routes and `**` keeps the rest. The counter beside the heading reads `1 of 500` on a new site: the seeded rule All routes (`**`, 60 s) is the first of up to 500 — the same with or without Pro. Once all 500 are in use the add form gives way to This site allows 500 cache rules and all of them are in use. Edit or delete one to add another. Edit the seeded rule freely — narrow its glob, lengthen its TTL, disable it.

Purge all orphans every cached response at once (a confirmation dialog first). It works without JavaScript, through a nonce URL.

Rules are the allow-list: a request is cached only when an enabled rule matches its route. Disable the only rule and nothing is cached.

The add / edit form

Field	What it does
Label	Up to 100 characters, for your own reference.
Route glob	Required. Matched against the REST route with its leading slash removed, so write `wp/v2/posts/**`, not `/wp-json/wp/v2/posts/**`. `*` matches one path segment, `**` matches anything including `/`, `?` matches one character. Everything else is literal; letters, digits and `/ * ? . - _ % : @ = & + ~` are allowed. A trailing `/*` or `/**` also matches the collection itself: `wc/v3/products/*` matches `/wc/v3/products` and `/wc/v3/products/12` but not `/wc/v3/products/12/variations`; `wp/v2/posts/**` matches `/wp/v2/posts`, `/wp/v2/posts/1` and `/wp/v2/posts/1/revisions`. A wildcard in the middle is never optional (`wp/v2/*/1` does not match `/wp/v2/1`).
TTL (seconds)	1–86400. A value outside the range is clamped to it.
Tags	Comma-separated. Lower-cased and reduced to `[a-z0-9_:-]`, at most 50 characters each, at most 20 per rule. Purging a tag orphans every response cached under it. **A rule with no tags is purged on every content change; add tags to narrow purging to those tags** (an untagged rule is keyed on the implicit tag `all`). In the free plugin tags matter only for a manual purge by tag (`POST /bfag/v1/cache/purge` with `tags`); the Pro add-on uses them to narrow the automatic purge on content change.
Enabled	Apply this rule.

A rejected save (an invalid glob) comes back as Cache rule not saved: the route glob may contain only letters, digits and / * ? . - _ % : @ = & + ~ (up to 500 characters). Your entries are kept in the form below.; a successful one as Cache rule saved.

How the cache decides: HIT, MISS or BYPASS

Every REST response gains an `X-BF-Cache` header. The decision is made in two places: before the request is dispatched (serve), and after the response is built (store).

1. Is the request cacheable? Any one of these makes it `BYPASS`:

- Enable response cache is off;
- the method is not `GET` or `HEAD`;
- a user is logged in (cookie authentication);
- the request carries an `Authorization`, `X-BF-API-Key` or `X-WP-Nonce` header — such a response may be personal;
- the route is under `/bfag/v1` (the plugin's own endpoints);
- a plugin returned `true` from the `bfag_cache_bypass` filter.

2. Does an enabled rule match the route? Rules are tried in id order and the most specific match wins (see The rules table). No match: `BYPASS`.

3. Is there a stored entry for this key? The key is an `md5` of the method, the route, the query parameters sorted by name at every depth (with `_wpnonce` and `_locale` removed, because they vary per visitor without changing the response), the lower-cased `Accept` header, the global cache generation

and the generation of every tag the rule carries (a rule with no tags counts as carrying the one tag `all`). Found: the stored status, headers and body are served with `X-BF-Cache: HIT`, and a hit is counted. The request never reaches the REST controller.

4. Otherwise the controller runs, and the response is examined. Any one of these makes it `BYPASS` rather than stored:

- the status is not 2xx;
- `Content-Type` contains `text/event-stream`;
- the response sets a cookie — a `Set-Cookie` header on the response, or one PHP has queued through `setcookie()` (how WordPress and WooCommerce set theirs, for example a guest's cart session). A plugin that sets a cookie on every request therefore keeps REST responses from being stored; `X-BF-Cache: BYPASS` shows it;
- `Cache-Control` carries `no-store`, `private` or `no-cache`, or a `max-age=0` / `s-maxage=0` (the plugin cannot revalidate, so a response that must be revalidated is simply not stored).

Everything else is stored for the rule's TTL and labelled `X-BF-Cache: MISS`, and a miss is counted. The next identical request is a `HIT`.

A rule may also carry extra response headers (a JSON object in the `headers` column of `bfag_cache_rules`); they are added to every response the rule matches, cached or not. The admin form does not expose that column.

Where responses are stored

With a persistent object cache (Redis, Memcached — `wp_using_ext_object_cache()` is true) responses go to the object-cache group `bfag_responses`, any size, for the rule's full TTL. Without one they go to transients named `bfag_resp_<key>`, which live in the options table — two rows each — and three limits apply there: a response over **64 KB** is not stored at all (it is served and labelled `MISS` every time; logged once per request when debug logging is on), an entry lives at most **one hour** whatever the rule's TTL, and `bfag_cron_prune_responses` deletes expired pairs every five minutes, 1,000 at a time — every expired `bfag…` transient, not only the responses: the limited markers, the parked forms, the one-time secrets and the Pro add-on's `bfagp_` transients go in the same sweep. The cap and the cleanup exist because a purge only orphans a transient (its key is an md5, so the generation it embeds cannot be read back) and core sweeps expired transients only once a day; without them every purge left every stored response in the options table until then. The Settings → Cache group says as much. Use a persistent object cache on a busy site.

How purging works

No entry is ever enumerated or deleted one by one. Each key embeds the **global generation** (option `bfag_cache_generation`) and the generation of each tag (all of them in the one option `bfag_tag_generations`). A purge bumps a generation, every future key is different, and the old entries are never looked up again. A purge of everything also flushes the `bfag_responses` group on an object cache that supports it; a tag purge relies on the tag generations alone.

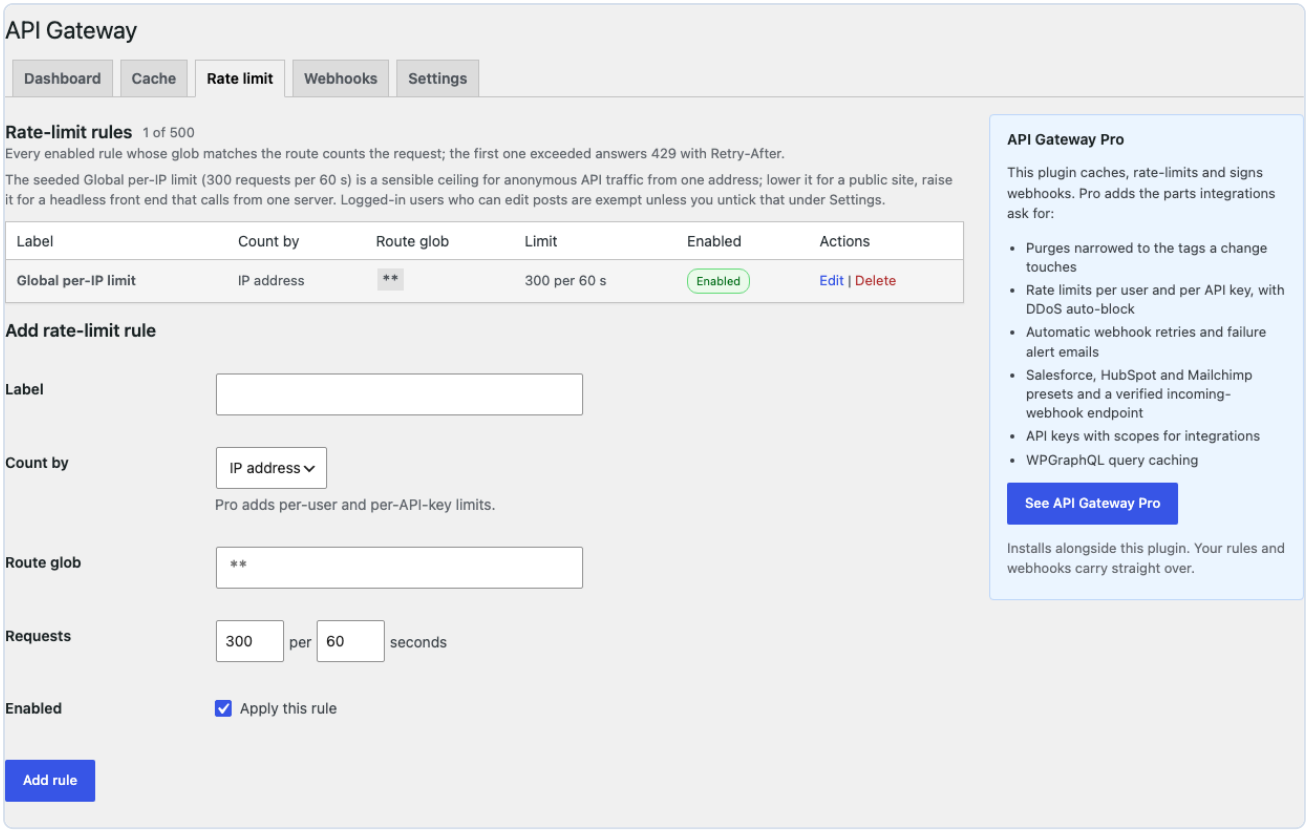
Trigger	What is purged
A post is saved, trashed or deleted (revisions and autosaves ignored)	everything
A term is created, edited or deleted	everything
A comment is inserted or changes status	everything
Purge all on the Cache tab	everything
`POST /bfag/v1/cache/purge` with no `tags`	everything
`POST /bfag/v1/cache/purge` with `tags`	only responses cached under those tags

In the free plugin every content change flushes everything, because it cannot tell which cached responses a change touches. The `bfag_cache_purge_tags` filter lets an add-on answer with the tags a change touches instead; the Pro add-on does exactly that, and its answer always includes `all`, the implicit tag of every untagged rule. So with Pro a rule with no tags is still purged on every content change; add tags to narrow purging to those tags.

Statistics

Hits and misses are counted in five-minute GMT buckets in `bfag_cache_stats` with a single `INSERT … ON DUPLICATE KEY UPDATE`, so a counter is never a read-modify-write race. With a persistent object cache they are counted there instead and written to the table by the five-minute KPI cron (see Cron and WP-CLI notes). The dashboard and `/bfag/v1/status` sum the last 24 hours.

Rate limit



The Rate limit tab: the seeded per-IP rule, 300 requests per 60 seconds

The rules table

Rate-limit rules shows label, **Count by** (IP address), route glob, limit as `N per M s`, the **Enabled** toggle badge and **Edit** | **Delete**. The counter reads `1 of 500`: the seeded rule Global per-IP limit (300 per 60 s on `**`) is the first of up to 500, with or without Pro. The note under the table says what it is for: a sensible ceiling for anonymous API traffic from one address; lower it for a public site, raise it for a headless front end that calls from one server. Edit it to taste.

The add / edit form

Field	What it does
Label	Up to 100 characters.
Count by	IP address is the only choice in the free plugin. The note under it says Pro adds per-user and per-API-key limits.
Route glob	Same syntax as a cache rule's.
Requests … per … seconds	Requests 1–65535; window 1–86400 seconds. The add form pre-fills the seeded rule's figures, 300 per 60 s. Unlike the cache TTL, a value outside its range is **rejected**, not clamped — a rule that silently became "1 request per day" would be a worse surprise than a refused save.
Enabled	Apply this rule.

A rejected save names its cause: Rate-limit rule not saved: that "Count by" subject type is not available (per-user and per-API-key limits need the Pro add-on)., Rate-limit rule not saved: the route glob may contain only letters, digits and / * ? . - _ % : @ = & + ~ (up to 500 characters). or Rate-limit rule not saved: requests must be 1–65535, the window 1–86400 seconds and the penalty 0–2592000 seconds. — each ending Your entries are kept in the form below.

How the counter works

The limiter runs on `rest_pre_dispatch` at priority 5 — ahead of the cache at priority 10 — so a limited client is refused before it can be served from cache, and cached hits count towards the limit like any other request.

1. The route is normalised: leading slash removed, lower-cased. WordPress matches REST routes case-insensitively, so `/wp/v2/Posts` is the same endpoint as `/wp/v2/posts` and is treated as such everywhere below. If Enable rate limiting is off, or the route is `/bfag/v1/status` (in any case), nothing is counted. Nor is anything counted for an internal dispatch — WP-CLI, WP-Cron, or a `rest_do_request()` made from admin-ajax or the front end — because those run as the server itself, and a penalty rule could otherwise block 127.0.0.1. Only the top-level HTTP REST request (`wp_is_serving_rest_request()` / `REST_REQUEST`) is limited.
2. The client IP is resolved (see Trusted proxies below). An unknown address is not counted. An `ip` rule counts an IPv4 address on its own and an IPv6 address by its /64 network (`2001:db8:1:2::/64`), because one host normally holds a whole /64 and could otherwise send each request from a fresh address. A logged-in user who can edit posts, in their own browser session (cookie and REST nonce), is exempt unless Exempt editors is unticked under Settings → Rate limit: the block editor, the media library and the customiser all drive the REST API from one browser. A request authenticated by an API key or an application password is never exempt, whatever the role of the user behind it, so a leaked administrator key is still limited and can still be blocked. Add-ons can narrow the exemption with the `bfag_rate_limit_exempt` filter.
3. `bfag_rate_limit_is_blocked` is asked; the free plugin has no block list, so this is always `false`. (Pro answers `true` for a blocked IP and the request gets `403 bfag_blocked` before any rule is counted.)
4. **Every** enabled rule whose glob matches the route is applied, in id order. For each, one row is **inserted** into `bfag_rate_limit_hits` (subject, the normalised route, a 16-hex-character SHA-256 prefix of the **rule key** `rule:<id>`, GMT timestamp) and **then** the rows for that subject and rule newer than `now − window` are counted. Insert-then-count means every request's own row is in the count it reads, so a hundred concurrent requests cannot all read "59" and all pass.
5. The first rule whose count exceeds its limit answers the request:

```
 HTTP/1.1 429 Too Many Requests
Retry-After: <window seconds>
X-BF-RateLimit-Limit: <max requests>
X-BF-RateLimit-Window: <window seconds>

{
  "code": "bfag_rate_limited",
  "message": "Too many requests. Please slow down and try again shortly.",
  "data": { "status": 429 }
}
```

It is a sliding window: a client allowed 300 per 60 s who sends 300 requests in the first second is refused for the next 59 seconds, and admitted again one request at a time as the oldest rows age out. The request that crosses the limit leaves a marker for the rest of the window (in the object cache when it is persistent, else a `bfag_limited_*` transient), and every further request from that client under that rule is refused from the marker alone — no row written, no count — so a client hammering a 429 costs nothing in the hits table. With a persistent object cache the count itself is an atomic `wp_cache_incr()` in fixed windows of the rule's length rather than a `COUNT(*)`: the hit row is still written (the admin's hit log reads it), and a client may get up to twice the limit across one window boundary, which is the usual trade for a counter that costs no query. The budget is **per subject and per rule**, not per path: every route a rule matches draws on the same allowance, so the `**` seeded rule means "300 per minute in total", and neither a crawl of 10,000 distinct URLs nor `/wp/v2/Posts` spelled ten different ways gets a fresh budget. Give a route its own allowance with its own rule. The `route` column of the hit log still shows the real route each hit came from.

If the insert fails (a database fault) the limiter fails open — a database problem must not turn into a site-wide 429 — and logs it when debug logging is on. Timestamps are written and compared in GMT from PHP's clock, never MySQL's.

Trusted proxies

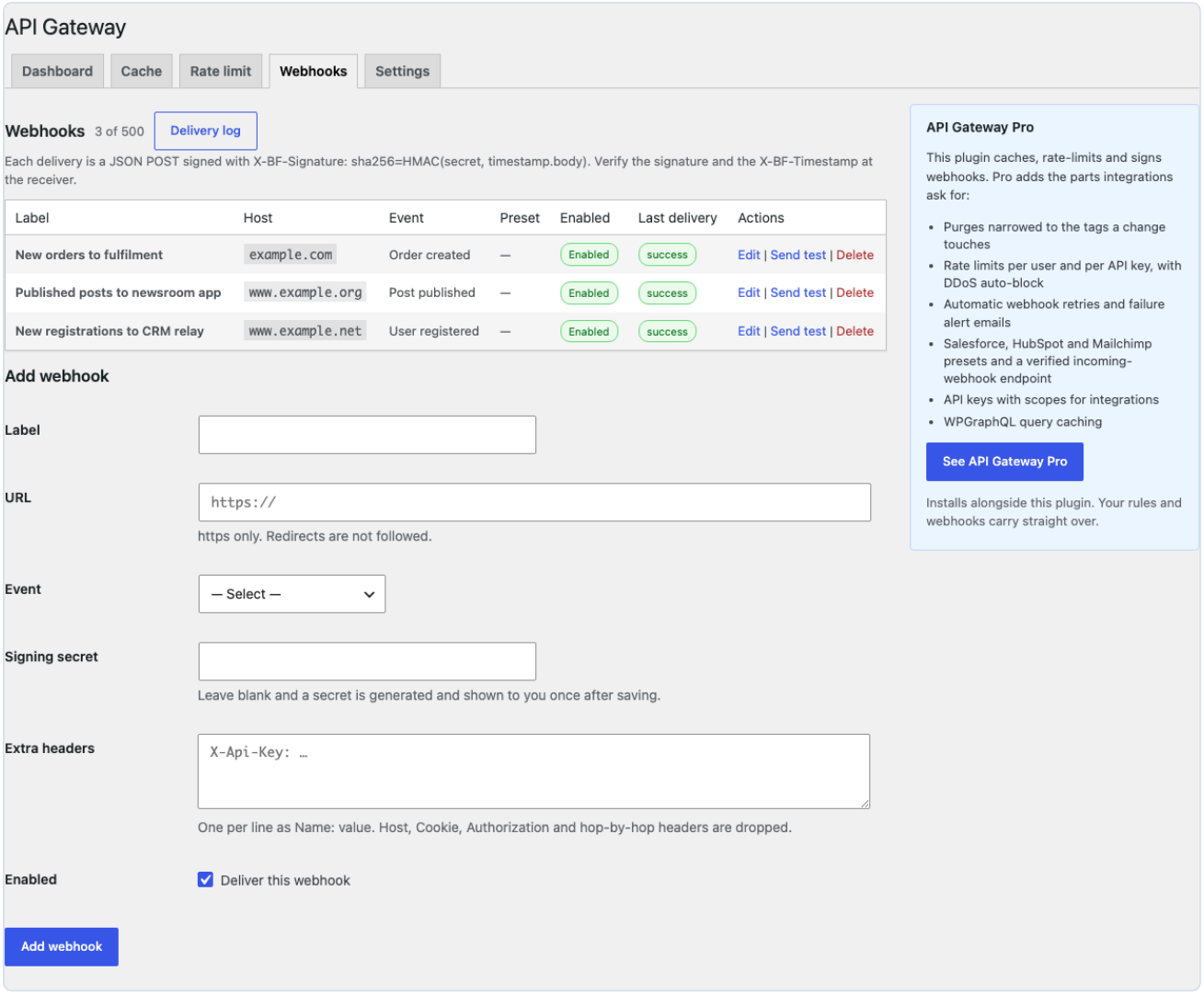
The client IP is `REMOTE_ADDR` — unless `REMOTE_ADDR` is inside one of the addresses or CIDR ranges listed under **Settings → Rate limit → Trusted proxies**. Only then is **one** forwarding header read: the one chosen under **Client address header** (`X-Forwarded-For` by default; `CF-Connecting-IP` , `X-Real-IP` or `True-Client-IP`). For `X-Forwarded-For` the address taken is the right-most one in the comma-separated list that is not itself a trusted proxy, read from the right; if an entry reached before that address is not a valid IP the header is ignored. Entries to the left of the address taken are never read. For the single-value headers the value is taken when it is a valid IP. Any header not chosen is ignored whatever it says, and an invalid value falls back to `REMOTE_ADDR` .

Trusting the headers unconditionally would let any visitor pick their own rate-limit bucket by sending a forged `X-Forwarded-For` , so behind Cloudflare or a load balancer you must list the proxy ranges — otherwise every visitor shares the proxy's address and one bucket. Reading several headers in a fixed order would be the same hole one step back: a proxy that only manages `X-Forwarded-For` passes a visitor's own `X-Real-IP` straight through, and a higher-ranked header would outrank the address the proxy actually wrote. So pick the header your proxy writes (Cloudflare: `CF-Connecting-IP`) and only that one is believed.

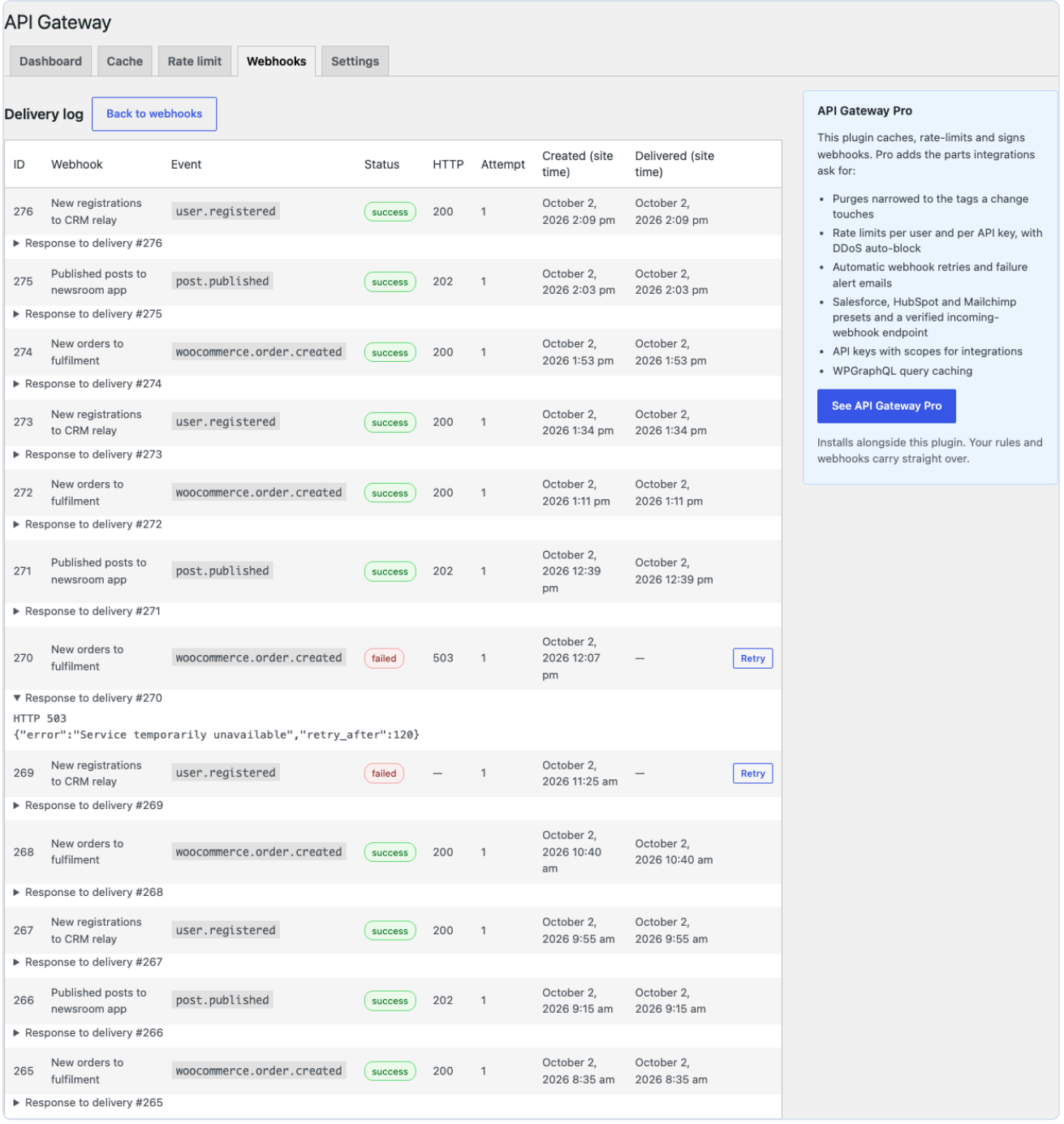
Keeping the hits table small

`bfag_cron_prune_hits` runs every five minutes and deletes hits older than the longest enabled window (at least 60 s), 10,000 rows at a time, at most 50 batches per run. If WP-Cron has been asleep long enough for the table to pass ten million rows, the oldest rows are deleted, by id range in slices of 10,000 (at most 50 slices a run), until the newest ten million remain; with debug logging on, the number deleted is logged. The table is never truncated: a range delete of the oldest ids does not stall the inserts at the other end, and it forgives only the oldest hits, which is better than every request's `COUNT` degrading further. That check reads an upper bound from the two ends of the primary key (`MAX(id) - MIN(id) + 1`), not a `COUNT(*)` and not the storage engine's estimate, which MySQL 8 can hold stale for a day; the bound over-counts only by rows deleted out of the middle of the id range, which the oldest-first prune never does. The one exact count is the KPI cron's, every five minutes, and that is the figure the dashboard shows.

Webhooks



The Webhooks tab: three webhooks, their events and the status of the last delivery



The delivery log: every delivery with its HTTP status, and the receiver's response opened on a failure

The webhooks table

Webhooks lists label, target host, event, preset (`—` in the free plugin), the **Enabled** toggle badge, the status of the last delivery (`success`, `failed`, `pending`; click it for that webhook's log), and **Edit** | **Send test** | **Delete**. The counter reads `N of 500`: a site may keep 500 webhooks, with or without Pro, and at 500 the form gives way to This site allows 500 webhooks and all of them are in use. **Delivery log** opens the log for every webhook.

Deleting a webhook deletes its delivery log with it, inside one transaction.

The add / edit form

Field	What it does
Label	Required, 1–200 characters.
URL	Required. `https://` only, and it must pass WordPress's `wp_http_validate_url()` — a loopback, private or otherwise unsafe address is refused. Redirects are not followed when delivering. (A filter, `bfag_webhook_allow_http`, permits plain `http://` for local development.)
Event	Required; one of the events below, grouped as Content, Users and — when WooCommerce is active — WooCommerce.
Signing secret	Leave blank on creation and a random 64-character hexadecimal secret is generated and **shown to you once**, on the Webhooks tab right after saving, in a box with a **Copy** button — copy it into your receiver then, because the stored secret is never displayed again. Or set your own (up to 255 characters). On edit, blank keeps the current secret; **Generate a new secret** replaces it — deliveries fail at the receiver until it is updated there — and the new secret is shown once after saving in the same way. Until that page shows it, the secret waits up to two minutes in a transient, encrypted with a key derived from your site's salts, so neither the database nor a shared object cache holds it in the clear.
Extra headers	One per line as `Name: value`. Only entries whose name is `[A-Za-z0-9-]` are shown here or read from here; anything else stored on the webhook (a Pro preset's configuration, which holds its credentials) is neither displayed nor touched by an edit. Sent with every delivery after sanitising: names must be `[A-Za-z0-9-]`; `Host`, `Content-Length`, `Transfer-Encoding`, `Connection`, `Cookie`, `Authorization`, `Proxy-Authorization` and anything beginning `X-Forwarded-` are dropped, as is any value containing a line break, and any name the plugin sets itself (`Content-Type`, `User-Agent`, `X-BF-Event`, `X-BF-Delivery`, `X-BF-Timestamp`, `X-BF-Signature`). Each rejection is logged when debug logging is on.
Preset	Appears only with the Pro add-on.
Max retries	Appears only with the Pro add-on (stored as 3 otherwise). Its help text: Automatic retries after a failed delivery (Pro). 0 = none.
Enabled	Deliver this webhook.

An invalid form comes back with the notice for the field that failed: URL refused: it must start with https:// and resolve to a public address (localhost, 127.0.0.1, private ranges and non-standard ports are refused)., Webhook not saved: the label must be 1–200 characters., Webhook not saved: choose an event from the list (a WooCommerce event disappears when WooCommerce is deactivated). or Webhook not saved: check the extra headers (one Name: value per line) and the secret (255 characters at most). — each ending Your entries are kept in the form below. With a preset chosen (Pro) the URL may be left blank: the preset builds the request; if the preset is removed later, deliveries go to the URL as plain signed JSON.

Events

Event key	Label	Fires on
`post.published`	Post published	`transition_post_status` — only when the new status is `publish` and the old one was not
`post.updated`	Post updated	`post_updated`
`post.deleted`	Post deleted	`before_delete_post`
`user.registered`	User registered	`user_register`
`user.updated`	User profile updated	`profile_update`
`comment.created`	Comment created	`wp_insert_comment`
`woocommerce.order.created`	Order created	`woocommerce_new_order`
`woocommerce.order.status_changed`	Order status changed	`woocommerce_order_status_changed`
`woocommerce.order.completed`	Order completed	`woocommerce_order_status_completed`
`woocommerce.product.updated`	Product updated	`woocommerce_update_product`

The payload's `object` carries these fields:

Events	`object` fields
the three `post.*` events	`id`, `type`, `status`, `title`, `slug`, `link`, `author_id`, `date_gmt`, `modified_gmt`
`user.registered`, `user.updated`	`id`, `login`, `email`, `display_name`, `roles`
`comment.created`	`id`, `post_id`, `author`, `author_email`, `approved`, `date_gmt`, `type`
`woocommerce.order.created`, `woocommerce.order.completed`	`id`, `status`, `currency`, `total`, `customer_id`, `email`, `created_gmt`
`woocommerce.order.status_changed`	the order fields plus `from_status`, `to_status`
`woocommerce.product.updated`	`id`, `name`, `sku`, `status`, `price`, `type`

The four WooCommerce events are offered only while WooCommerce is active. Post events fire only for **public** post types, never for revisions or autosaves. The post's status is not checked: updating or

deleting a draft or private post of a public type is delivered, with its `status` in the payload. Payloads are built from an allow-list of fields per object: never a raw `$userdata` array (which carries the password hash), never `$_SERVER`, never another user's email. A payload of `null` — a post whose type is not public, a user that could not be loaded — means no delivery.

The payload

```
{
  "event": "post.published",
  "site": "https://example.com",
  "occurred_at": "2026-09-30T14:03:11+00:00",
  "object": {
    "id": 42,
    "type": "post",
    "status": "publish",
    "title": "Autumn planting guide",
    "slug": "autumn-planting-guide",
    "link": "https://example.com/autumn-planting-guide/",
    "author_id": 3,
    "date_gmt": "2026-09-30 14:03:11",
    "modified_gmt": "2026-09-30 14:03:11"
  }
}
```

Send test delivers `{"event":"test","site":…,"occurred_at":…,"object":{"webhook_id":7,"label":"…"}}` to one webhook and reports Test delivery succeeded or Test delivery failed; either way the attempt is in the log.

When and how a delivery is sent

The WordPress action for an event is attached only while an enabled webhook subscribes to that event, so a site with no webhooks pays nothing on `save_post`. During a request the plugin only **collects** payloads, deduped per event and object id — a bulk import that saves one post ten times queues one delivery. The queue is flushed once, on `shutdown`, after the response to the browser has been sent. Under PHP-FPM the response is handed to the web server first (`fastcgi_finish_request()`; the `bfag_webhook_finish_response` filter turns that off for a plugin that prints on `shutdown`). Delivery is synchronous from there for at most **twenty** deliveries and at most **fifteen seconds** of wall clock: one `wp_remote_post()` per webhook, five-second timeout, no redirects, `reject_unsafe_urls`. Anything beyond twenty, or still unsent when the fifteen seconds are up (a dead endpoint at the five-second timeout would otherwise hold a PHP-FPM worker for 100 s), is written to the delivery log as `pending`, due now, and one WP-Cron single event (`bfag_cron_flush_pending`) sends those in batches of twenty, batch after batch for up to twenty seconds per run, re-arming itself only while rows remain — so a bulk import of 500 posts with three webhooks never ties up the PHP worker that saved them, and the queue drains at the endpoint's speed rather than twenty rows per cron spawn. Under WP-CLI or WP-Cron, where there is no response to wait for, the queue is flushed as soon as it passes twenty rather than held in memory. A queued row the cron never reaches is pruned after a day. A queued delivery whose payload is longer than Max stored payload (bytes) cannot be sent at all — only the truncated stored copy is left —

so it is recorded as `failed` with the reason as its response, and reported as a final failure (with Pro, the failure alert). A `save_post` triggered by a receiver writing back during delivery is ignored rather than queued, so two sites pointing at each other cannot loop.

Every delivery row is written as `pending` **before** the request is sent and updated after it, so a fatal mid-send leaves a visible pending row rather than nothing.

Request headers

Header	Value
`Content-Type`	`application/json`
`User-Agent`	`BlueFairy-API-Gateway/1.0.0`
`X-BF-Event`	the event key, e.g. `post.published`
`X-BF-Delivery`	the delivery row id — the same id on every retry of that delivery, so a receiver can deduplicate
`X-BF-Timestamp`	Unix time, in seconds, when the request was signed
`X-BF-Signature`	`sha256=` followed by the signature below

plus your sanitised extra headers.

How the signature is computed

```
X-BF-Signature: sha256=<hex HMAC-SHA256(
    key     = secret,
    message = X-BF-Timestamp + "." + body
)>
```

The message is the timestamp **as sent in the header**, a single `.`, and the **exact request body bytes**. A retry re-signs the stored bytes with a fresh timestamp, so the signature changes while the body and `X-BF-Delivery` do not.

Verifying at the receiver

Read the raw body before any framework parses it, rebuild the message, compare in constant time, and reject stale timestamps to defeat replay.

```php
<?php // PHP receiver
$secret    = getenv( 'BF_WEBHOOK_SECRET' );
$body      = file_get_contents( 'php://input' );
$timestamp = $_SERVER['HTTP_X_BF_TIMESTAMP'] ?? '';
$signature = $_SERVER['HTTP_X_BF_SIGNATURE'] ?? '';

if ( ! ctype_digit( $timestamp ) || abs( time() - (int) $timestamp ) > 300 ) {
    http_response_code( 401 );
    exit( 'stale timestamp' );
}

$expected = 'sha256=' . hash_hmac( 'sha256', $timestamp . '.' . $body, $secret );
if ( ! hash_equals( $expected, $signature ) ) {
    http_response_code( 401 );
    exit( 'bad signature' );
}

$payload = json_decode( $body, true );
// $_SERVER['HTTP_X_BF_EVENT'] names the event;
// $_SERVER['HTTP_X_BF_DELIVERY'] is the delivery id.
http_response_code( 200 );
```

```
 // Node receiver (Express)
import { createHmac, timingSafeEqual } from 'node:crypto';
import express from 'express';

const app = express();
const secret = process.env.BF_WEBHOOK_SECRET;

app.post('/hooks/wordpress', express.raw({ type: '*/*' }), (req, res) => {
  const timestamp = req.get('X-BF-Timestamp') ?? '';
  const signature = req.get('X-BF-Signature') ?? '';

  if (!/^\d+$/.test(timestamp) || Math.abs(Date.now() / 1000 - Number(timestamp)) > 300) {
    return res.status(401).send('stale timestamp');
  }

  const expected = 'sha256=' + createHmac('sha256', secret)
    .update(timestamp + '.')
    .update(req.body)            // the raw Buffer, not a re-serialised object
    .digest('hex');

  const a = Buffer.from(expected);
  const b = Buffer.from(signature);
  if (a.length !== b.length || !timingSafeEqual(a, b)) {
    return res.status(401).send('bad signature');
  }

  const payload = JSON.parse(req.body.toString('utf8'));
  res.sendStatus(200);
});
```

Answer with any 2xx status to mark the delivery `success`; anything else, or no answer within five seconds, marks it `failed`.

The delivery log

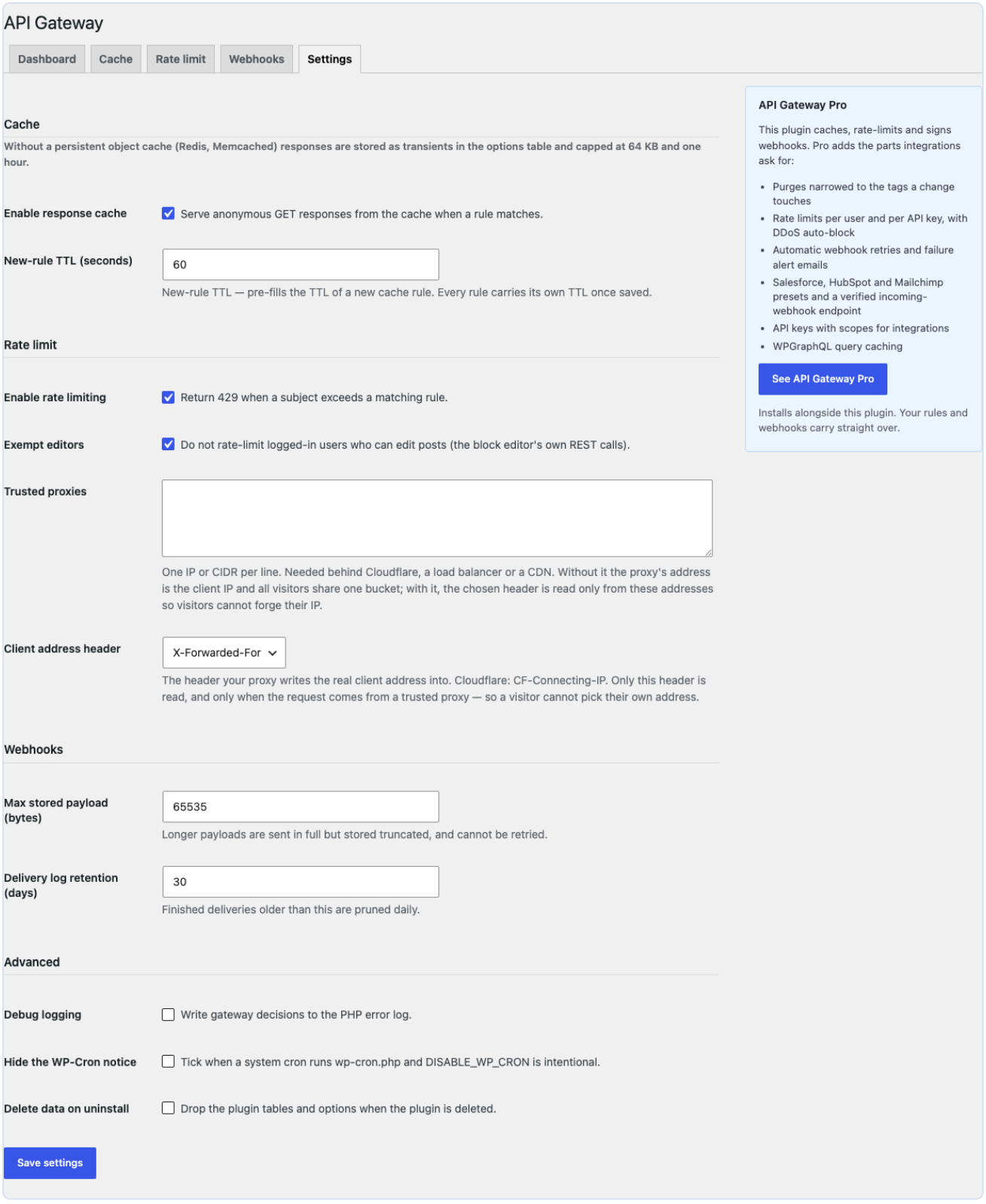
Columns: id, webhook, event, status badge (`pending`, `success`, `failed`; `retrying` with Pro), HTTP status, attempt, created, delivered (both in the site's timezone and formats — the headers say site time), and a **Retry** button on outgoing `failed` (or Pro `retrying`) deliveries that were not truncated — never on a successful or in-flight one, and the server refuses such a retry with a 400 even if asked directly. Under every finished row a **Response** disclosure opens the HTTP status and the response body. Fifty rows a page, newest first; filtered to one webhook when you arrive from its row. With more than fifty, a pager above the table (N deliveries, Page X of Y, first/previous/next/last) reaches every row, keeping the webhook filter, so an older failure can still be opened and retried.

- **Response** — the first 2,000 characters of what the receiver returned (`HTTP 500` then the body), or `Transport error: <message>` when there was no HTTP response at all (DNS, connection refused,

timeout). Send test reports the same status in its notice: Test delivery failed (HTTP 503) or (no HTTP response: transport error).

- **Truncated** — the stored payload was cut at Max stored payload (bytes). The receiver got the full body; the log did not keep it, so the delivery cannot be resent as signed. A truncated delivery that was queued (past the first twenty of a request) was never sent, and says so in its response.
- **Retry** resends the stored bytes as attempt + 1 with a new timestamp and signature. It is the only retry in the free plugin; automatic retries with back-off are a Pro feature. Only outgoing rows can be resent: an inbound row (Pro's receiving endpoint) was written by whoever called it, and the plugin will not sign someone else's payload with the site's secret.
- **Retention** — `bfag_cron_prune_deliveries` removes `success` and `failed` rows older than Delivery log retention once a day, 5,000 at a time. `pending` and `retrying` rows older than a day are removed whatever the retention, so a row the cron never reached cannot live for ever.

Settings



The Settings tab: cache, rate limit, webhook and advanced settings

General

This group appears only with the Pro add-on: nothing in the free plugin reads its two fields, so they are kept as hidden values rather than shown.

Field	Behaviour
Notification email	Where the Pro add-on's failure alerts are sent. Blank or invalid: the alert goes to the site admin email. The free plugin stores the address and sends nothing.
Failure threshold	1–100 consecutive failures before the Pro add-on alerts. Stored, unused by the free plugin.

Cache

Field	Behaviour
Enable response cache	Serve anonymous GET responses from the cache when a rule matches. Off: every response is `BYPASS`.
New-rule TTL (seconds)	1–86400. Pre-fills the TTL of a new cache rule in the add form. It is not a fallback: every rule carries its own TTL once saved, and the rule's TTL is the TTL.

Rate limit

Field	Behaviour
Enable rate limiting	Return 429 when a subject exceeds a matching rule. Off: nothing is counted.
Exempt editors	On by default. Do not rate-limit logged-in users who can edit posts in their own browser session (the block editor's own REST calls). Requests made with an API key or an application password are always limited.
Trusted proxies	One IP or CIDR per line (IPv4 or IPv6). Invalid lines are discarded on save. Needed behind Cloudflare, a load balancer or a CDN: without it the proxy's address is the client IP and all visitors share one bucket; with it, the chosen header is read only from these addresses so visitors cannot forge their IP. While it is empty and a request arrives carrying a forwarded-address header, a dismissible notice says so on the plugin's screen, the Plugins screen and the Dashboard. See Trusted proxies above.
Client address header	`X-Forwarded-For` (default), `CF-Connecting-IP`, `X-Real-IP` or `True-Client-IP`: the header your proxy writes the real client address into. Only this header is read, and only when the request comes from a trusted proxy — so a visitor cannot pick their own address. Anything else submitted is stored as the default.

Webhooks

Field	Behaviour
Max stored payload (bytes)	1024–16777215, default 65535. Longer payloads are sent in full but stored truncated, and cannot be retried.
Delivery log retention (days)	1–365, default 30. Finished deliveries older than this are pruned daily.

Advanced

Field	Behaviour
Debug logging	Write gateway decisions to the PHP error log, each line prefixed `[bfag]` with a JSON context: rejected headers and why, breached rules, refused rule saves, prune and rotation counts, failed deliveries. Off by default; while it is off the only line written is a failed table creation, which is always logged.
Hide the WP-Cron notice	Tick when a system cron runs `wp-cron.php` and `DISABLE_WP_CRON` is intentional.
Delete data on uninstall	Drop the plugin's six tables when the plugin is deleted (its options are removed on every uninstall). The Pro add-on's uninstall honours this same switch for its own tables.

Integer settings outside their bounds are clamped on save; an invalid notification email is stored as blank; unknown keys are dropped.

Cron and WP-CLI notes

Hook	Schedule	Work
`bfag_cron_prune_hits`	every five minutes	prune the rate-limit hit log; past ten million rows (by the `MAX(id) - MIN(id)` bound) delete the oldest so the newest ten million remain
`bfag_cron_prune_deliveries`	daily	prune finished deliveries past the retention setting, and `pending` / `retrying` rows older than a day
`bfag_cron_recompute_kpis`	every five minutes	flush the object-cache hit/miss counters (persistent cache only), then rebuild the dashboard transient (`bfag_kpis` , six-minute TTL)
`bfag_cron_prune_responses`	every five minutes	delete every expired `bfag…` transient (no persistent object cache) in slices of 1,000 pairs
`bfag_cron_flush_pending`	single event, on demand	send webhook deliveries queued past the twenty-per-flush limit, batch after batch of twenty for up to twenty seconds a run

The five-minute interval is registered by the plugin on `cron_schedules` as `bfag_five_minutes` (its label is translated only once `init` has run, so a schedule lookup before `init` never trips WordPress 6.7's early-text-domain notice). The four recurring events are scheduled sixty seconds after activation and cleared on deactivation, along with any pending flush event. The plugin books them only from `init` on, never during plugin loading, and any that goes missing (a cron wipe, a restore) is booked again on the next page load. They run on WP-Cron, which is driven by page views; on a quiet site the hit log is pruned less often than every five minutes, which is harmless because counting only ever looks back one window. If `DISABLE_WP_CRON` is true the plugin's screen shows a warning until you confirm, under Advanced, that a real cron runs `wp-cron.php` .

The plugin ships no WP-CLI commands of its own. Core's commands cover what you might need:

```
wp cron event list | grep bfag_
wp cron event run bfag_cron_prune_hits
wp cron event run bfag_cron_recompute_kpis
wp option get bfag_settings --format=json
wp transient delete bfag_kpis
```

In the free plugin the first 20 webhook deliveries of a request are sent at the end of that request; the rest are queued and sent by `bfag_cron_flush_pending`, a single event scheduled on demand and re-armed while queued rows remain.

Installation

1. Upload the `bluefairy-api-gateway` folder to `/wp-content/plugins/`, or install the zip from **Plugins** → **Add New** → **Upload Plugin**.
2. Activate it from the **Plugins** screen (PHP 8.1 or newer is required; the activation refuses otherwise).
3. Open **Settings** → **API Gateway**. The cache and the rate limiter are already on, with one seeded rule each; webhooks wait for you to add one.

Uninstalling

Deleting the plugin through the Plugins screen removes its options, every `bfag_` transient (the dashboard's `bfag_kpis` is deleted by name through the transients API as well as by the options-table sweep, so it goes even when a persistent object cache holds it), its four recurring cron events and any pending flush event, and the `bfag_manage_gateway` capability, on every site of a multisite network. Your rules, webhooks and delivery log — the six `bfag_` tables — are dropped **only** when Delete data on uninstall was ticked beforehand; by default they stay, because a delivery log is a record, not a cache. Deactivating removes only the cron events and the capability.

Extending

The Pro add-on is built entirely on these hooks; they are public API and changing one is a breaking change. `Plugin::service( $id )` hands out the shared instances (`settings`, `request`, `cache.store`, `cache.rules`, `cache.stats`, `cache.manager`, `cache.purger`, `ratelimit.rules`, `ratelimit.hits`, `webhook.repository`, `webhook.deliveries`, `webhook.dispatcher`, `webhook.events`, `kpi.cron`) once `bfag_booted` has fired.

Lifecycle

Hook	Type	Signature	Purpose
`bfag_loaded`	action	`()`	Fired by `Plugin::boot()` **before** any subsystem binds. Attach the subject-type, purge-tag, preset and any limit filters here — a filter attached after a repository has read it is attached too late. Does not fire while this plugin is inactive, so an add-on must also check the dependency on `plugins_loaded`.
`bfag_booted`	action	`()`	Fired after every subsystem is wired; `Plugin::service()` is populated.
`bfag_is_pro`	filter	`( bool $is_pro = false )`	`true` hides the Pro panel, reveals the Pro-only form fields and is reported by `GET /bfag/v1/status`.

Cache

Hook	Type	Signature	Purpose
`bfag_cache_rule_limit`	filter	`( int $limit = 500 )`	Maximum number of cache rules: 500 by default, with or without Pro (`Support\Limits::DEFAULT_LIMIT`; 1 before 1.0.1). Lower or raise it; at most 1,000 (the most rules the plugin reads).
`bfag_cache_purge_tags`	filter	`( string[]|null $tags = null, string $type, int $id )`	`$type` is `'post'`, `'term'` or `'comment'`. Return the tags the change touches to bump only those; return `null` to flush everything. Include `all`: a rule with no tags is keyed on it and is purged on every content change; only tagged rules are narrower.
`bfag_cache_bypass`	filter	`( bool $bypass = false, WP_REST_Request $request )`	Return `true` to force a request past the cache (`BYPASS`).
`bfag_cache_purged`	action	`( string $scope, string[] $tags )`	`$scope` is `'all'` or `'tags'`. Fired by every purge path: content hooks, the Cache tab, AJAX and the REST endpoint.

Rate limiting

Hook	Type	Signature	Purpose
`bfag_rate_limit_rule_limit`	filter	`( int $limit = 500 )`	Maximum number of rate-limit rules: 500 by default, with or without Pro (1 before 1.0.1); at most 1,000.
`bfag_rate_limit_subject_types`	filter	`( string[] $types = [ 'ip' ] )`	Subject types a rule may count by. Pro adds `user` and `apikey`.
`bfag_rate_limit_subject`	filter	`( string|null $subject, string $type, WP_REST_Request $request, string $ip )`	The subject a rule counts this request against. Default is the IP for type `ip`, `null` otherwise; `null` or `''` means the rule does not apply to this request.
`bfag_rate_limit_is_blocked`	filter	`( bool $blocked = false, string $ip, WP_REST_Request $request )`	Return `true` to answer `403 bfag_blocked` before any rule is counted.
`bfag_rate_limit_breached`	action	`( string $subject, array $rule, WP_REST_Request $request, string $ip, bool $forwarded_without_proxy )`	A subject newly exceeded a rule: fired once per window, when the marker is set, never for a request the marker refuses. The last argument is true when a forwarded-address header arrived but the sender is not a trusted proxy — the IP is a shared address and must not be blocked.
`bfag_rate_limit_refused`	action	`( string $subject, array $rule, string $ip, int $refused_so_far )`	Every request refused from the limited marker, with the running count of refusals of this subject under this rule in the current window (this one included; the breaching request is

Hook	Type	Signature	Purpose
			not a refusal). Fired with no database work — the count lives in the marker — so a DDoS threshold can mean "requests over the limit" without a `COUNT(*)` per `429`.
`bfag_rate_limit_max_window`	filter	`( int $seconds )`	The oldest hit the prune cron must keep; default is the largest enabled window (or 60). Raise it if you count across a longer window yourself.

Webhooks

Hook	Type	Signature	Purpose
`bfag_webhook_limit`	filter	`( int $limit = 500 )`	Maximum number of webhooks: 500 by default, with or without Pro (3 before 1.0.1); at most 1,000.
`bfag_webhook_events`	filter	`( array $events )`	Add events: `key => [ 'hook' => string, 'label' => string, 'accepted_args' => int, 'group' => string ]`. An add-on event's payload object is `[ 'id' => <first integer hook argument> ]` until `bfag_webhook_payload` fills it in.
`bfag_webhook_payload`	filter	`( array|null $payload, string $event, array $hook_args )`	The payload before it is signed and sent. Return `null` to suppress the delivery.
`bfag_webhook_request_args`	filter	`( array $args, array $webhook, string $event, array $payload )`	The `wp_remote_post()` arguments. May rewrite `body` and `headers`, set `url` to redirect the delivery, and set `headers['Authorization']` — the only place a credential header is accepted. The filter is trusted for credentials, not for transport: after it runs the headers are sanitised again, so a `Host`, `X-Forwarded-*` or line-break value it plants is dropped. Set `$args['bfag_skip']` to a non-empty string to cancel the delivery: it is logged as `failed` with that string as the response body, nothing is sent, and `bfag_webhook_delivery_failed` fires with `$is_final` true.
`bfag_webhook_allow_http`	filter	`( bool $allow = false )`	Permit `http://` targets (local development only).
`bfag_webhook_delivery_failed`	action	`( int $delivery_id, array $webhook, int $attempt, int|null $http_status, bool $is_final )`	A delivery failed. `$is_final` is **true** for the outcomes a retry can never change — the request-args filter cancelled the delivery (`bfag_skip` : the preset could not build a request from this payload), the URL is invalid or unsafe (logged as `Invalid or unsafe URL (nothing sent): <url>`), or the payload was longer than Max stored payload (bytes), so the stored copy is truncated and could not be resent (or, for a delivery queued past

Hook	Type	Signature	Purpose
			the first twenty of a request, could not be sent at all — its response reads Payload longer than Max stored payload (bytes): … (nothing sent).); a scheduler must leave such a row `failed`. It is **false** for any other non-2xx answer or transport error (`$http_status` null), which may be retried. Pro schedules retries and alerts from here; a listener registered with four accepted arguments keeps working.
`bfag_webhook_delivered`	action	`( int $delivery_id, array $webhook )`	A delivery succeeded.
`bfag_webhook_finish_response`	filter	`( bool $finish )`	Whether the shutdown flush calls `fastcgi_finish_request()` before sending (default: whenever PHP is not running as CLI). A plugin that prints on `shutdown` after this one returns `false`.

Admin page

Hook	Type	Signature	Purpose
`bfag_admin_tabs`	filter	`( array $tabs )`	Add tabs: `id => [ 'label' => string, 'page' => object with render(): void and handle_actions(): void ]`.
`bfag_dashboard_kpis`	filter	`( array $kpis )`	Add or change dashboard figures before they are stored.
`bfag_kpi_blocked_today`	filter	`( int $blocked = 0 )`	The Active blocks card (Pro). Without Pro the dashboard shows Blocked (Pro) instead. `bfag_dashboard_kpis` may add `blocked_label` to rename the card, on render and on Refresh.
`bfag_settings_fields`	action	`( Settings $settings )`	Render extra `<tr>` rows inside the Settings form table.
`bfag_settings_saved`	action	`( array $clean, array $raw )`	After the base settings are stored; `$raw` is the whole unslashed `$_POST`.
`bfag_cache_page_after`	action	`()`	Render below the cache rules table and its add/edit form.
`bfag_ratelimit_page_after`	action	`()`	Render below the rate-limit rules table and its add/edit form.
`bfag_webhook_form_fields`	action	`( array $webhook )`	Extra rows in the add/edit webhook form, directly under the Preset row and above Max retries and Enabled, between two `<tbody>` row groups (print `<tr>` rows or your own `<tbody>`); `$webhook` never contains the secret.
`bfag_webhook_form_defaults`	filter	`( array $defaults )`	Values the Add webhook form opens with (Pro sets `max_retries`)
`bfag_webhook_presets`	filter	`( array $presets = [] )`	`slug => label`. Non-empty makes the form show a **Preset** select and store the choice.
`bfag_webhook_sanitize`	filter	`( array $data, array $raw )`	Sanitise your own form fields into the columns about to be stored.

Hook	Type	Signature	Purpose
`bfag_admin_notices`	filter	`( array $messages )`	Add notice codes for `?bfag_notice=<code>` : `code => [ 'success'|'error'|'warning'|'info', text ]` .
`bfag_admin_action_<action>`	action	`()`	A form `bfag_action` no base tab owns, fired after the `bfag_<action>` nonce and the capability have been verified.

Admin forms post back to `options-general.php?page=bfag-gateway&tab=<id>` with `bfag_action=<action>` and `wp_nonce_field( 'bfag_<action>' )` . AJAX actions are `bfag_ajax_purge_cache` , `bfag_ajax_refresh_kpis` , `bfag_ajax_retry_webhook` , `bfag_ajax_toggle_rule` and `bfag_ajax_dismiss_notice` , each with a nonce of the same name.

Route → scope map

`Capabilities::required_scope( $route, $method )` returns the API-key scope a route needs, and `Capabilities::scope_satisfied( $held, $required )` evaluates it. The map lives in the base so the two plugins cannot disagree about it; the Pro add-on enforces it.

Scope	Grants	Routes
`read`	Read any route	GET/HEAD/OPTIONS on every route outside `/bfag/v1` , and any method on the `/bfag/v1` routes not listed below (`/bfag/v1/status`)
`write`	Create, update and delete through any REST route (acts as the key's user)	POST/PUT/PATCH/DELETE on every route outside `/bfag/v1` — `wp/v2` , `wc/v3` , …
`cache:purge`	Purge the response cache	`POST /bfag/v1/cache/purge`
`webhook:read`	List webhooks and their deliveries	GET on `/bfag/v1/webhooks…`
`webhook:write`	Create, update, delete and retry webhooks	POST/PUT/PATCH/DELETE on `/bfag/v1/webhooks…`

`webhook:write` implies `webhook:read` ; nothing else implies anything — `write` does not imply `read` , and neither implies `cache:purge` . A key acts as the user it is bound to, so `write` on an administrator's key is as powerful as that administrator's REST access: grant it only to integrations that must change content. `Capabilities::SCOPES` lists the tokens and `Capabilities::SCOPE_LABELS` their one-line descriptions, which the Pro add-on's API-key form shows (`Capabilities::scope_label()` is the translatable twin).

The gateway's own two routes use `Capabilities::request_allowed( $request )` as their permission callback: the `bfag_manage_gateway` capability passes, and so does a request the Pro add-on authenticated with an API key whose scope covers the route (`cache:purge` for the purge endpoint, `read` for status) — even a key bound to Nobody. The base asks for the key's scope through the

`bfag_request_key_scope` filter, which the add-on answers from the key it authenticated; a scope a client plants in the query string or the body is never read.

What API Gateway Pro adds

You have a cache, a rate limiter and signed webhooks — up to 500 rules of each kind and 500 webhooks, the same as with Pro. What you do not have is anything that happens when a delivery fails at three in the morning, any way to stop a client that keeps hammering you, or any way to give an integration its own credentials.

That is what Pro adds. It installs **alongside** this plugin rather than replacing it — your rules, webhooks and settings carry straight over, and this plugin keeps doing exactly what it does now.

Feature	API Gateway Pro	API Gateway (free)
REST response cache, every response labelled `X-BF-Cache: HIT` / `MISS` / `BYPASS`	Yes	Yes
Cache rules, rate-limit rules, webhooks	Up to 500 of each	Up to 500 of each — the same limit
What a content change purges	Only the tags it touches: `post:<id>`, `type:<post_type>`, `term:<id>`, `tax:<taxonomy>`, `all`	Everything
Purge all button and `POST /bfag/v1/cache/purge`	Yes	Yes
WPGraphQL query caching (anonymous, error-free queries)	Yes	—
Per-IP sliding-window rate limiting with `429` and `Retry-After`	Yes	Yes
Count a rule by logged-in user or by API key	Yes	—
Penalty block — an address that breaches an IP rule is blocked for N seconds	Yes	—
DDoS auto-block, with a never-block list	Yes	—
Blocked IP addresses panel — see, lift and add blocks by hand	Yes	—
Dashboard Active blocks card	Live count of blocked addresses	Blocked (Pro) placeholder
Real client address behind trusted proxies	Yes	Yes
Signed outbound webhooks (`X-BF-Signature`, HMAC-SHA256)	Yes	Yes
Events — six WordPress, four WooCommerce	Yes	Yes
Delivery log with a manual **Retry** on failed deliveries	Yes	Yes
Automatic retries — 1 min, 5 min, then 30 min, by a five-minute sweep, up to each webhook's Max retries (0–10)	Yes	—
Retries per delivery — a site-wide default for new webhooks	Yes	—
Failure alert email — on the first failure, after N in a row, or after the last retry	Yes	— (sends no alerts)
Salesforce, HubSpot and Mailchimp presets — the delivery becomes a CRM record	Yes	—

Feature	API Gateway Pro	API Gateway (free)
Verified incoming webhooks at `/bfag/v1/incoming/<preset>`	Yes	—
Webhook REST API — list, create, update, delete, read the log, retry	Yes	—
Scoped API keys (`read` , `write` , `cache:purge` , `webhook:read` , `webhook:write`)	Yes	—
`GET /bfag/v1/status` for a monitoring tool	Yes	Yes
Updates through your licence key (Licence tab); no feature depends on the licence	Yes	— (no automatic updates: new versions are a download from the shop)
Admin tabs	7 (adds **API Keys** and **Licence**)	5

Why the missing rows cost you

A failed delivery stays failed, and nobody tells you. Your fulfilment service was down for two minutes at three in the morning. The new order is in your delivery log marked `failed`, and this plugin does nothing more about it: no second attempt, no email. It waits there until someone happens to open the dashboard and press **Retry** — by which time that order has sat unfulfilled for hours. Pro retries it for you — due one minute later, then five, then thirty, and sent by a sweep that runs every five minutes — up to the limit you set per webhook, and emails you at the moment you choose: on the first failure, after a run of failures, or only once the last retry has failed.

Your CRM receives JSON it cannot read. This plugin sends one signed JSON shape to one URL. Salesforce, HubSpot and Mailchimp each want their own format, their own endpoint and their own authentication, so today you need a relay service in the middle — another thing to pay for and another thing to go down. Pro's presets turn the delivery into a Salesforce Lead, a HubSpot contact, deal or ticket, or a Mailchimp subscriber directly, and verify the webhooks those vendors send back to you.

One blog post edit empties the product cache. This plugin has one answer to any content change: flush everything. Every post you publish turns your whole API cache cold again. Pro tags what each change touches, so editing a post leaves responses tagged `type:product` warm. If your front end speaks WPGraphQL, Pro caches those queries too; this plugin does not.

A client that is hammering you is only slowed down. The free limiter answers `429` and keeps counting, request after request, for as long as the client cares to keep sending them. Pro blocks the address outright — for the penalty you set on the rule, or, once you switch on auto-block, for an hour by default after a rule has refused the DDoS threshold number of its requests over the limit — and gives you a panel listing every block, its reason and its expiry, with **Unblock** and a form to block an address by hand.

One address means one budget for everyone behind it. Every rule in this plugin counts by IP address, so every user and every integration calling from one office or one server shares one allowance. Pro lets a rule count per logged-in user or per API key, so a partner integration gets its own allowance instead of sharing an office IP.

Every integration shares the administrator's login. Today a build server that needs to purge the cache holds a credential that can do anything the administrator can. Pro mints API keys with scopes — a deploy pipeline can purge the cache and nothing else, a dashboard can read webhooks without being able to change them — each one stored hashed, with an optional expiry and a revoke that takes effect at once.

Get Pro

https://wordpress.bluefairydevelopment.com/product/bluefairy-api-gateway-pro/

One payment, no subscription: the licence never expires, and the price includes 12 months of updates and support — renewing those after the first year is optional. Pro requires this plugin to be installed and active — it is an add-on, not a replacement — and adding Pro to the cart adds the free plugin with it.