# Call Baxter

**Runtime, Telegram tooling, and WordPress bridge**

*Call Baxter contributors*

# Contents

# 1. call-baxter monorepo

Monorepo layout:

- `packages/call-baxter`:
- generic callback/runtime scaffold
- webhook ingress, pollers, schedules, rules, and actions
- Telegram integrated as a plugin
- `packages/tg-agent-cli`:
- Telegram projection/index database and CLI
- browse chats, messages, callbacks, media, and action history from `tg.db`
- `packages/wordpress-plugin`:
- WordPress bridge plugin for mirroring Telegram events into WordPress posts
- idempotent media upload, hashtag routing, and gallery support
- `packages/social-plugin`:
- Bluesky, Mastodon, and X publishing plugins
- native provider actions plus durable Telegram mirror/album-flush actions
- `infra`:
- Docker/Compose and Caddy deployment assets
- `rules.d`:
- operator rule files
- `services/rust-baxter`:
- Rust service/docs kept in the same repository

## 1.1 Runtime split

```
Telegram webhook or poll
  -> call-baxter telegram plugin
  -> persist into tg-agent-cli projection db
  -> emit telegram.* events
  -> call-baxter rules/actions
  -> tg-agent-cli operates directly on tg.db for browse/operator commands
```

## 1.2 Quickstart from monorepo root

Run the canonical verification gate before changing or releasing the repository:

```
./scripts/verify.sh
```

This root gate runs lockfile validation, all package tests, ruff, mypy, coordinated release-version validation, package builds, installation tests for the built wheels, Compose validation, Docker hardening checks, infra todo checks, repository hygiene checks, documentation command checks, quality/architecture todo checks, and coverage floors.

After the wheels have been built, their focused installed-artifact smoke test is:

```
./scripts/smoke-packages.sh
```

The complete release gate additionally starts both shipped container configurations:

```
./scripts/verify-release.sh v0.8.0
```

Focused infra todo verification is also available with:

```
./scripts/verify-infra-todos.sh
```

Focused quality/architecture todo verification is available with:

```
./scripts/verify-qual-arch-todos.sh
```

## 1.3 Deploy from the monorepo

For the complete Telegram-to-WordPress example, use the self-contained `docker/` bundle and start in polling mode:

```
cd docker
cp .env.example .env
# Edit .env with the Telegram token and database passwords.
mkdir -p state/{run,data,caddy_data,caddy_config,mysql,wordpress}
sudo chown -R 10001:10001 state/run state/data
sudo chmod 0770 state/run state/data
docker compose up -d --build
```

Then follow `docker/TELEGRAM_WORDPRESS_TUTORIAL.md` to install WordPress, create the integration user and application password, verify the REST API, publish a real `#blog` message, and optionally switch to HTTPS webhook mode.

Use `infra/` only for the minimal Call Baxter + Caddy layout. That image includes the social publishing plugins, but does not contain the WordPress plugin and does not run WordPress or MariaDB.

## 1.4 Coordinated release

Release `v0.8.0` ships `call-baxter`, `tg-agent-cli`, `cb-wordpress-plugin`, and `cb-social-plugin` as one coordinated package set. The first three remain the TGEX/`telegras2` replacement surface; the social package adds independently configurable Bluesky, Mastodon, and X publishing actions.

This minor release adds the social publishing plugin family and per-provider documentation while retaining the modular `cb` CLI, Telegram reingestion, redacted runtime configuration, and the operational acceptance metadata established by the 0.7 release line.

The `call-baxter` package installs compatible `telegras` and `telegras2` launchers. Existing webhook and polling deployments can retain the legacy command names, environment aliases, webhook path, `/healthz`, and `/internal/introspection/*` clients while the process runs the Call Baxter runtime.

Install the coordinated release artifacts into the same environment, preserve the existing TGEX environment, and initialize compatibility state:

```
telegras2 db-init
telegras2 doctor
telegras2 migrate-history --dry-run
telegras2 migrate-history
telegras2 routes
telegras2 start
```

The full compatibility matrix, cutover checklist, limitations, and rollback procedure are documented in `packages/call-baxter/docs/tgex-migration.md` and in the MkDocs page **call-baxter → Replace TGEX**.

## 1.5 Documentation

The repository includes a root `mkdocs.yml` configured for Material for MkDocs. The site is built from `docs/` wrapper pages that include the package-local markdown, so the package docs remain the canonical source. The published site provides client-side search with suggestions, highlighting, and shareable result URLs.

The documentation toolchain is pinned in `tools/docs/uv.lock`. Use the repository scripts instead of an unpinned global MkDocs installation:

```
# Live local preview
uv run --project tools/docs --locked mkdocs serve

# Strict HTML/search-index build into site/
./scripts/build-docs.sh

# Strict HTML validation plus the combined A4 PDF
./scripts/build-docs-pdf.sh
```

The PDF is written to `site/assets/call-baxter-documentation.pdf`. CI uploads it as a workflow artifact, and the Pages workflow publishes the same file beside the HTML site.

## 1.6 Automation

- `.github/workflows/ci.yml` runs package tests, linting, type checks, package builds, the reproducible HTML/PDF docs build, and the existing full `scripts/verify.sh` gate.
- `.github/workflows/docs.yml` deploys the searchable MkDocs site and generated PDF to GitHub Pages from `main` or `master`.
- `.github/workflows/release.yml` runs the complete release gate for tags matching `v*`, then publishes all wheels, source archives, the verified documentation PDF, and a `SHA256SUMS` manifest. Publishing is idempotent, so a failed or interrupted run can be rerun without recreating the tag.

## 1.7 Operator access and zrok command allowlist

Set `CALL_BAXTER_API_KEY` for non-ingress `/v1/*` operator routes such as reload, action execution, event publishing, poller runs, and query endpoints.

Set `CALL_BAXTER_INGRESS_TOKEN` separately for webhook/ingress execution routes so webhook callers do not gain operator privileges.

The `zrok.run` action is constrained by `allowed_subcommands` in the `call_baxter.plugins.zrok` plugin config. The default allowlist is `share`, `status`, and `version`; add more commands only when an operator rule genuinely needs them.

```yaml
plugins:
  - module: call_baxter.plugins.zrok
    config:
      allowed_subcommands:
        - share
        - status
        - version
```

## 1.8 Docker deployment layouts

| Layout | Contents | Intended use |
|---|---|---|
| `docker/` | Call Baxter, Telegram projection, WordPress plugin, MariaDB, WordPress, optional Caddy profile | complete Telegram-to-WordPress deployment |
| `infra/` | Call Baxter, Telegram projection, Caddy | minimal runtime/webhook deployment without the bundled WordPress bridge |

The `docker/` example uses Telegram polling by default so the first deployment needs no public domain. Caddy starts only with `docker compose --profile webhook ...` after webhook mode is configured.

# 2. Deploy

## 2.1 Deployment guide

Call Baxter has two Docker Compose layouts with deliberately different scopes.

### Choose the correct stack

| Layout | Services | Best for | WordPress bridge included |
|---|---|---|---|
| `docker/` | Call Baxter, MariaDB, WordPress, optional Caddy | complete Telegram-to-WordPress deployment and first-run tutorial | yes |
| `infra/` | Call Baxter and Caddy | minimal runtime/proxy deployment when WordPress is external or unused | no |

> **Do not use the minimal stack for the bundled WordPress example**
>
> The `infra/` image installs `call-baxter` and `tg-agent-cli`, but not `cb-wordpress-plugin`. It also does not run WordPress or MariaDB. Use `docker/` for the end-to-end example.

### Recommended first deployment

Start with Telegram long polling. It requires no public DNS or HTTPS and isolates initial failures to the bot token, runtime, rules, and WordPress credentials.

```
Telegram getUpdates
  -> call-baxter
  -> telegram.message.new
  -> wordpress.mirror_telegram_message
  -> WordPress REST API
```

Once this works, switch to webhook mode:

```
Telegram HTTPS webhook
  -> Caddy secret-token check
  -> call-baxter Unix socket
  -> same event/rule/action path
  -> WordPress REST API
```

### Tutorials and references

- Hands-on Docker deployment: Telegram to WordPress
- Minimal Call Baxter + Caddy deployment
- WordPress plugin integration
- Telegram plugin
- TGEX migration

### Operational sequence

```
1. Verify repository
2. Create Telegram bot
3. Configure docker/.env
4. Start MariaDB + WordPress
5. Create WordPress integration user + application password
6. Verify REST authentication
7. Start Call Baxter in polling mode
8. Send #blog test message
9. Verify runtime, projection, publication mapping, and WordPress
10. Optionally switch to Caddy webhook mode
11. Back up state before upgrades
```

The full tutorial includes exact commands and failure diagnostics for each stage.

## 2.2 Hands-on deployment: Telegram to WordPress with Docker Compose

This tutorial starts with the easiest reliable path—Telegram long polling—and then shows how to switch the same stack to a public HTTPS webhook.

The finished flow is:

```
Telegram message
  -> telegram.updates poller or HTTPS webhook
  -> call-baxter Telegram plugin
  -> telegram.message.new / telegram.message.edited
  -> mirror rules
  -> cb-wordpress-plugin
  -> WordPress REST API
  -> draft or published WordPress post
```

### What the example stack runs

| Service | Purpose | Host exposure in the first-run polling setup |
|---|---|---|
| `call-baxter` | Telegram ingestion, rules, persistence, WordPress actions | none; Unix socket only |
| `db` | MariaDB for WordPress | none |
| `wordpress` | WordPress application and REST API | `http://localhost:8080` by default |
| `caddy` | Public HTTPS Telegram webhook proxy | disabled unless the `webhook` profile is enabled |

Persistent state is written below `docker/state/`.

### Deployment model used in this tutorial

This guide assumes the deployment host builds the image from a reviewed Git revision. The normal source-to-runtime path is:

```
workstation
  -> git push origin main
  -> deployment host: git pull --ff-only
  -> verify exact Git SHA
  -> docker compose build --pull call-baxter
  -> docker compose up -d
```

This is the simplest model for the repository as it is currently structured: the Compose file has a local `build:` definition and the Dockerfile installs `call-baxter`, `tg-agent-cli`, and `cb-wordpress-plugin` together. Do not copy a locally built image to the server unless you deliberately switch to an image-registry workflow with immutable tags or digests.

On the deployment host, update and record the revision before changing containers:

```
git fetch origin
git checkout main
git pull --ff-only origin main
git rev-parse HEAD
```

For a production registry workflow later, build once in CI, push a tag based on the Git SHA, and deploy by digest instead of rebuilding independently on every host.

## Sample credentials used below

All credentials in this block are **fake examples**. They are intentionally unsuitable for a real deployment. Replace every one of them before starting a public or persistent instance.

```
# Telegram example only; not a real usable bot token.
TGCLI_BOT_TOKEN=123456:EXAMPLE_ONLY_REPLACE_ME

# WordPress integration account created later in this tutorial.
WP_USERNAME=wp-bot
WP_APP_PASSWORD="demo wp application password change me"

# Local demo database credentials only.
WORDPRESS_DB_NAME=wordpress
WORDPRESS_DB_USER=wordpress
WORDPRESS_DB_PASSWORD=demo-db-password-change-me
WORDPRESS_DB_ROOT_PASSWORD=demo-root-password-change-me

# Publication policy used by the example.
WP_PUBLISH_STATUS=draft
WP_REQUIRED_HASHTAG="#blog"
WP_CATEGORIES=
WP_TAGS=
```

The sample values make the configuration shape easy to recognize; the actual setup steps below generate or obtain proper secrets instead of reusing these strings.

## Choose an ingestion mode

| Mode | Use it when | Public DNS/HTTPS required | Compose command |
|---|---|---|---|
| Polling | first setup, local machine, private server, development | no | `docker compose up -d --build` |
| Webhook | public production ingress with lower delivery latency | yes | `docker compose --profile webhook up -d --build` |

Start with polling. Telegram does not allow `getUpdates` polling while a webhook is registered, so the polling steps explicitly remove any old webhook before testing.

## Prerequisites

You need:

- Docker Engine with Docker Compose v2;
- a Telegram account;
- a Telegram bot token created with `@BotFather`;
- a browser for the initial WordPress setup;
- `curl` for the verification commands;
- optional `jq` for readable JSON output.

Run all commands from the repository root unless a step says otherwise.

## 1. Verify the repository before deployment

```
./scripts/verify.sh
```

The release gate checks tests, linting, typing, package builds, Compose configuration, Docker hardening invariants, documentation commands, and coverage floors.

## 2. Create a Telegram bot

In Telegram:

1. Open a chat with `@BotFather`.
2. Send `/newbot`.
3. Choose a display name and a username ending in `bot`.
4. Copy the bot token into a password manager.
5. Open the new bot chat and press **Start** so you can send it direct test messages later.

Treat the token like a password. Anyone who has it can operate the bot API.

## 3. Prepare `docker/.env`

```
cd docker
cp .env.example .env
```

Generate three safe values:

```
python - <<'PY'
import secrets

print("TELEGRAM_WEBHOOK_SECRET=" + secrets.token_urlsafe(48))
print("WORDPRESS_DB_PASSWORD=" + secrets.token_hex(24))
print("WORDPRESS_DB_ROOT_PASSWORD=" + secrets.token_hex(24))
PY
```

Edit `docker/.env` and set at least the following. The values shown here retain the same fake/example shape from above; replace the Telegram and database credentials with your real generated values before starting the stack:

```
TELEGRAM_POLL_ENABLED=true
TGCLI_BOT_TOKEN=123456:EXAMPLE_ONLY_REPLACE_ME

WORDPRESS_DB_PASSWORD=demo-db-password-change-me
WORDPRESS_DB_ROOT_PASSWORD=demo-root-password-change-me

WP_USERNAME=wp-bot
WP_APP_PASSWORD="replace-after-wordpress-setup"
WP_PUBLISH_STATUS=draft
WP_REQUIRED_HASHTAG="#blog"
WP_CATEGORIES=
WP_TAGS=
```

For an actual deployment, the resulting file should contain the real BotFather token and the generated database passwords, not the demonstration strings above.

`WP_CATEGORIES` and `WP_TAGS` are comma-separated numeric WordPress term IDs. Docker supplies environment variables as strings, and Call Baxter converts these values to integers before sending them to WordPress; taxonomy names such as `News` are not accepted as IDs.

To retrieve category IDs without enabling the WordPress REST API, query WordPress directly from its container:

```
docker compose exec -T wordpress php -r 'require "/var/www/html/wp-load.php"; foreach
(get_terms(["taxonomy"=>"category","hide_empty"=>false]) as $term) echo $term->term_id."\t".$term->name."\t".
$term->slug."\n";'
```

Copy the desired `term_id` values into `WP_CATEGORIES=4,9` (or the hashtag-specific `WP_*_CATEGORY_ID` variables) and restart Call Baxter.

The Call Baxter container also exposes a `cb` executable backed by the installed Call Baxter CLI and configured for its Unix socket by default:

```
docker compose exec call-baxter cb ingress runs --limit 20
docker compose exec call-baxter cb ingress reingest ing:123
```

`WP_REQUIRED_HASHTAG` accepts a single hashtag such as `#blog`; leave it empty to disable required-hashtag filtering. `WP_CATEGORIES` and `WP_TAGS` accept comma-separated WordPress term IDs such as `4,9`. They are fallback terms; the default Docker config also maps `#blog` and `#news` through `WP_BLOG_CATEGORY_ID` and `WP_NEWS_CATEGORY_ID`.

For the first run, leave the webhook domain values as placeholders. Caddy is in an optional Compose profile and will not start in polling mode.

## 4. Prepare persistent directories

The Call Baxter container runs as UID/GID `10001:10001`. Bind-mounted runtime directories must therefore be writable by that identity.

```
mkdir -p state/{run,data,caddy_data,caddy_config,mysql,wordpress}
sudo chown -R 10001:10001 state/run state/data
sudo chmod 0770 state/run state/data
```

On a rootless Docker installation, map the directory ownership to the container UID used by your runtime instead of blindly using the host UID `10001`.

## 5. Validate the rendered Compose configuration

```
docker compose config --quiet
```

A successful command prints nothing and exits with status `0`.

Inspect the service plan when diagnosing interpolation problems:

```
docker compose config --services
docker compose config | less
```

In polling mode, `caddy` is not part of the default service set because it belongs to the `webhook` profile.

## 6. Start MariaDB and WordPress

```
docker compose up -d db wordpress

docker compose ps
docker compose logs --tail=100 db wordpress
```

Wait until the WordPress setup page answers:

```
until curl --fail --silent --show-error \
  "http://localhost:${WORDPRESS_PORT:-8080}/wp-admin/install.php" \
  >/dev/null; do
  printf 'waiting for WordPress...\n'
  sleep 3
done
```

Open this URL in a browser:

```
http://localhost:8080
```

When deploying to another machine, replace `localhost` with that server's address and ensure the configured `WORDPRESS_PORT` is reachable only from trusted networks during setup.

## 7. Complete the WordPress setup

Create the initial administrator through the WordPress setup screen.

Then create a dedicated integration user:

1. Open **Users → Add New**.
2. Use the username configured as `WP_USERNAME`, for example `wp-bot`.
3. Give it the **Editor** role for text-only publishing. Use a role with upload capability when mirroring Telegram media.
4. Save the user.
5. Open that user's profile.
6. Under **Application Passwords**, create one named `call-baxter`.
7. Copy the generated password immediately; WordPress displays it only once.

Update `docker/.env`:

```
WP_USERNAME=wp-bot
WP_APP_PASSWORD="xxxx xxxx xxxx xxxx xxxx xxxx"
```

The spaces in a WordPress application password are accepted. Keep the value quoted in the env file.

## 8. Verify WordPress REST authentication

Load the env file into the current shell:

```
set -a
. ./.env
set +a
```

Verify the credentials through the REST API:

```
curl --fail-with-body --silent --show-error \
  --user "${WP_USERNAME}:${WP_APP_PASSWORD}" \
  "http://localhost:${WORDPRESS_PORT}/wp-json/wp/v2/users/me?context=edit"
```

Expected result: a JSON user object for `wp-bot`.

Common failures:

- `401`: wrong username or application password;
- `403`: the user lacks the required capability;
- HTML instead of JSON: the WordPress installation is incomplete or the URL is wrong;
- no **Application Passwords** section: finish the site setup and use HTTP only for local testing; production WordPress should be served over HTTPS.

## 9. Start Call Baxter in polling mode

Before polling, remove an old Telegram webhook if this bot was previously used elsewhere:

```
curl --fail-with-body --silent --show-error \
  --request POST \
  "https://api.telegram.org/bot${TGCLI_BOT_TOKEN}/deleteWebhook" \
  --data-urlencode "drop_pending_updates=false"
```

Build the Call Baxter image on this deployment host, then start it:

```
# Optional but useful for an audit trail.
DEPLOY_SHA="$(git -C .. rev-parse HEAD)"
printf 'deploying Call Baxter source revision %s\n' "$DEPLOY_SHA"

docker compose build --pull call-baxter
docker compose up -d call-baxter

docker compose ps
docker compose images call-baxter
docker compose logs --tail=150 call-baxter
```

The bundled `docker/Dockerfile` installs all three Python packages needed for this flow: `call-baxter`, `tg-agent-cli`, and `cb-wordpress-plugin`. The separate minimal `infra/` image is not sufficient for this tutorial because it deliberately omits the WordPress plugin.

The Docker runtime config enables the `telegram.updates` schedule when `TELEGRAM_POLL_ENABLED=true`.

Verify daemon health over its Unix socket:

```
docker compose exec call-baxter \
  cb \
  --config-file /srv/config/call-baxter.yml \
  --daemon-url http+unix:///srv/run/cb.sock \
  health
```

Inspect loaded plugins, rules, and schedules:

```
docker compose exec call-baxter \
  cb --config-file /srv/config/call-baxter.yml \
  --daemon-url http+unix:///srv/run/cb.sock \
  plugins list

docker compose exec call-baxter \
  cb --config-file /srv/config/call-baxter.yml \
  --daemon-url http+unix:///srv/run/cb.sock \
  rules list

docker compose exec call-baxter \
  cb --config-file /srv/config/call-baxter.yml \
  --daemon-url http+unix:///srv/run/cb.sock \
  schedules list
```

You should see:

- the Telegram plugin;
- the WordPress plugin;
- `mirror-telegram-message`;
- `mirror-telegram-edit`;
- `flush-media-groups`;
- the enabled `telegram-default` polling schedule.

## 10. Send the first Telegram-to-WordPress post

Send this direct message to the bot:

```
#blog First Call Baxter post

This draft travelled from Telegram through Call Baxter into WordPress.
```

The parser uses the first non-empty line as the title. Leading hashtags on that line are removed from the generated title, so the WordPress title becomes:

```
First Call Baxter post
```

The remaining text becomes the post body. Because the example uses `WP_PUBLISH_STATUS=draft`, the post appears under **Posts → Drafts**.

Follow the runtime while testing:

```
docker compose logs -f call-baxter
```

Stop log following with `Ctrl-C`; the container keeps running.

## 11. Verify all three persistence layers

**WordPress**

```
curl --fail-with-body --silent --show-error \
  --user "${WP_USERNAME}:${WP_APP_PASSWORD}" \
  "http://localhost:${WORDPRESS_PORT}/wp-json/wp/v2/posts?context=edit&search=First%20Call%20Baxter%20post"
```

**Telegram projection database**

```
docker compose exec call-baxter \
  tg --database-url /srv/data/tg.db messages list
```

**Call Baxter runtime records**

```
docker compose exec call-baxter \
  cb --config-file /srv/config/call-baxter.yml \
  --daemon-url http+unix:///srv/run/cb.sock \
  events list --kind telegram.message.new

docker compose exec call-baxter \
  cb --config-file /srv/config/call-baxter.yml \
  --daemon-url http+unix:///srv/run/cb.sock \
  actions list --kind wordpress.mirror_telegram_message
```

Expected state files:

```
state/data/cb.db
state/data/tg.db
state/data/wp-publications.db
state/data/tg-media/
```

## 12. Test idempotent edits

Edit the original Telegram message instead of sending a new one:

```
#blog First Call Baxter post

This body was edited in Telegram and should update the existing WordPress draft.
```

The `mirror-telegram-edit` rule sends `telegram.message.edited` to the same mirror action. The publication mapping in `wp-publications.db` should update the existing WordPress post rather than create a duplicate.

## 13. Test publishing policy

The example configuration supports status routing:

```
status_by_hashtag:
  "#draft": draft
  "#publish": publish
```

To test immediate publication, send:

```
#blog #publish Published from Telegram

This message requests WordPress status publish.
```

Use `#publish` only after the bot user, publication policy, and review process are ready for direct publishing. Keeping the default status as `draft` is safer.

## 14. Test a Telegram media group

Send an album containing two or more images. Put the caption and `#blog` hashtag on one album item.

The single `mirror-telegram-message` rule handles both ordinary messages and media-group fragments. Because the WordPress plugin is configured with `media_group_wait_seconds: 45`, grouped fragments are staged by `chat_id + media_group_id`. The heartbeat rule later runs `wordpress.flush_due_media_groups` and creates one assembled WordPress post.

Do **not** add a second broad `telegram.message.new` staging rule. Two rules matching the same event would invoke the mirror action twice.

Watch the staging behavior:

```
docker compose logs -f call-baxter
```

After at least 45 seconds plus the heartbeat interval, verify the resulting post and media library in WordPress.

## 15. Switch from polling to an HTTPS webhook

Only do this after the polling flow works.

### 15.1 Prepare public DNS and firewall

Point `PUBLIC_DOMAIN` to the Docker host and allow inbound TCP ports `80` and `443`. Telegram must be able to reach the resulting HTTPS URL.

Update `docker/.env`:

```
TELEGRAM_POLL_ENABLED=false
PUBLIC_DOMAIN=bot.example.org
ACME_EMAIL=ops@example.org
TELEGRAM_WEBHOOK_PATH=/webhooks/telegram/main
TELEGRAM_WEBHOOK_SECRET=replace-with-generated-secret
```

The webhook secret must contain only characters accepted by Telegram. The `secrets.token_urlsafe()` command from step 3 produces a suitable value.

### 15.2 Recreate Call Baxter and start Caddy

```
docker compose up -d --build --force-recreate call-baxter
docker compose --profile webhook up -d caddy

docker compose ps
docker compose logs --tail=150 caddy call-baxter
```

Verify the public health endpoint:

```
curl --fail --silent --show-error \
  "https://${PUBLIC_DOMAIN}/healthz"
```

### 15.3 Register the webhook

```
curl --fail-with-body --silent --show-error \
  --request POST \
  "https://api.telegram.org/bot${TGCLI_BOT_TOKEN}/setWebhook" \
  --data-urlencode "url=https://${PUBLIC_DOMAIN}${TELEGRAM_WEBHOOK_PATH}" \
  --data-urlencode "secret_token=${TELEGRAM_WEBHOOK_SECRET}" \
  --data-urlencode 'allowed_updates=["message","edited_message"]'
```

Inspect Telegram's current webhook state:

```
curl --fail-with-body --silent --show-error \
  "https://api.telegram.org/bot${TGCLI_BOT_TOKEN}/getWebhookInfo"
```

Expected fields include the configured URL and no persistent `last_error_message`.

Caddy accepts only `POST` requests on `TELEGRAM_WEBHOOK_PATH`, verifies `X-Telegram-Bot-Api-Secret-Token`, rewrites the request to `/v1/webhooks/telegram.update`, and forwards it over the shared Unix socket.

## 16. Upgrade the deployment

From the repository root:

```
git pull --ff-only
./scripts/verify.sh
cd docker
docker compose build --pull call-baxter
docker compose up -d call-baxter wordpress db
```

When using webhook mode:

```
docker compose --profile webhook up -d caddy
```

Before upgrading, back up at least:

```
state/data/
state/mysql/
state/wordpress/
```

## 17. Backup and restore

Stop write-heavy services before a filesystem-level backup:

```
docker compose stop call-baxter wordpress db

tar --create --gzip \
  --file "call-baxter-backup-$(date +%Y%m%d-%H%M%S).tar.gz" \
  state/data state/mysql state/wordpress

docker compose start db wordpress call-baxter
```

In webhook mode, Caddy may remain up but will return an upstream error while Call Baxter is stopped. For a quiet maintenance window, stop Caddy too.

Restore only into a stopped stack and preserve directory ownership.

### Reconcile interrupted WordPress media uploads

The deployment records every successful media upload in `/srv/data/wp-publications.db`. The heartbeat flush automatically reviews interrupted create attempts older than `WP_ORPHAN_MEDIA_CLEANUP_AFTER_SECONDS` and processes at most `WP_ORPHAN_MEDIA_CLEANUP_LIMIT` publications per pass.

Run a manual dry-run before an accelerated cleanup:

```
cat > state/data/wp-media-cleanup.json <<'JSON'
{
  "state_db_path": "/srv/data/wp-publications.db",
  "older_than_seconds": 0,
  "limit": 50,
  "dry_run": true
}
JSON

docker compose exec call-baxter \
  cb --config-file /srv/config/call-baxter.yml \
  --daemon-url http+unix:///srv/run/cb.sock \
  actions execute \
  --kind wordpress.cleanup_orphan_media \
  --args-file /srv/data/wp-media-cleanup.json
```

The cleanup first searches WordPress by the deterministic publication slug. It keeps and maps tracked media when the post exists. It deletes media only when the upload belongs to an interrupted create and no matching post exists. Repeat with `"dry_run": false` to apply the reviewed cleanup.

Back up `state/data/wp-publications.db` together with WordPress. Deleting this state database removes the evidence needed for safe reconciliation.

## 18. Troubleshooting decision tree

```
No WordPress post
|
+-- Did Telegram deliver an update?
|   +-- polling: delete any old webhook, inspect telegram-default schedule
|   +-- webhook: inspect getWebhookInfo and Caddy logs
|
+-- Did Call Baxter emit telegram.message.new?
|   +-- no: inspect bot token, allowed updates, projection DB, ingress errors
|
+-- Did mirror-telegram-message match?
|   +-- no: check rule loading and source/kind
|
+-- Did the action run?
|   +-- skipped: message lacks #blog or chat/channel policy rejected it
|   +-- failed: inspect WordPress URL, user, application password, capabilities
|
+-- Did WordPress accept the request?
    +-- 401: credentials
    +-- 403: role/capability or security plugin
    +-- 404: wrong WP_BASE_URL or REST routing
    +-- 5xx: WordPress/PHP/database logs
```

### Polling receives nothing

```
curl --silent --show-error \
  "https://api.telegram.org/bot${TGCLI_BOT_TOKEN}/getWebhookInfo"

docker compose exec call-baxter \
  cb --config-file /srv/config/call-baxter.yml \
  --daemon-url http+unix:///srv/run/cb.sock \
  schedules list

docker compose logs --tail=250 call-baxter
```

Confirm that:

- `TELEGRAM_POLL_ENABLED=true`;
- no webhook URL is registered;
- the bot token is correct;
- the bot chat has been started;
- no other process is consuming `getUpdates` for the same bot.

### Webhook returns `403 forbidden`

Confirm that:

- the registered URL exactly matches `https://${PUBLIC_DOMAIN}${TELEGRAM_WEBHOOK_PATH}`;
- the secret registered with Telegram matches `TELEGRAM_WEBHOOK_SECRET`;
- Caddy was recreated after editing `.env`;
- Telegram is sending the expected secret-token header.

### WordPress REST authentication fails

```
curl --verbose \
  --user "${WP_USERNAME}:${WP_APP_PASSWORD}" \
  "http://localhost:${WORDPRESS_PORT}/wp-json/wp/v2/users/me?context=edit"
```

Create a fresh application password rather than reusing the normal WordPress login password.

### Message is skipped

The default policy requires `#blog`. Inspect the action result and make sure the hashtag is present in either the message text or media caption.

### Duplicate media-group handling

The example intentionally contains one new-message mirror rule and one heartbeat flush rule. If you copied an older `stage-media-group.yml`, remove or disable it so each Telegram update reaches `wordpress.mirror_telegram_message` only once.

### Reset the example completely

This deletes all local WordPress, MariaDB, and Call Baxter state:

```
docker compose --profile webhook down
sudo rm -rf state
```

Then restart from step 4. Do not use this command on a deployment whose data must be preserved.

## Configuration map

| Requirement | File or variable |
| --- | --- |
| Bot token | `docker/.env` → `TGCLI_BOT_TOKEN` |
| Polling versus webhook | `docker/.env` → `TELEGRAM_POLL_ENABLED` |
| Poll interval and long-poll timeout | `docker/.env` → `TELEGRAM_POLL_*` |
| Public webhook URL | `PUBLIC_DOMAIN` + `TELEGRAM_WEBHOOK_PATH` |
| Webhook verification | `TELEGRAM_WEBHOOK_SECRET` and `docker/config/Caddyfile` |
| WordPress REST credentials | `WP_BASE_URL`, `WP_USERNAME`, `WP_APP_PASSWORD` |
| Publication hashtag/status/category policy | `docker/config/call-baxter.yml` |
| Event-to-action routing | `docker/rules.d/*.yml` |
| Runtime and projection state | `docker/state/data/` |
| WordPress content and database | `docker/state/wordpress/`, `docker/state/mysql/` |

## Security checklist before production

- Keep `docker/.env` outside version control and readable only by the deployment operator.
- Use a dedicated WordPress integration user and application password.
- Keep `WP_PUBLISH_STATUS=draft` until direct publishing is explicitly desired.
- Rotate the Telegram token immediately if it is exposed.
- Use webhook mode only behind valid HTTPS and the secret-token check.
- Restrict the host's WordPress port or place WordPress behind a separate trusted reverse proxy.
- Back up MariaDB, WordPress files, and all SQLite state before upgrades.
- Review `docker compose config` for accidentally interpolated secrets before sharing output.

## Upstream references

- Telegram Bot API: https://core.telegram.org/bots/api
- Telegram webhook guide: https://core.telegram.org/bots/webhooks
- WordPress REST API authentication: https://developer.wordpress.org/rest-api/using-the-rest-api/authentication/
- WordPress application-password endpoints: https://developer.wordpress.org/rest-api/reference/application-passwords/

# 3. call-baxter

## 3.1 call-baxter

A generic callback/runtime scaffold with:

- a daemon exposing a stable HTTP control plane
- a CLI built with `pydantic-settings` subcommands
- explicit plugin loading from config
- YAML rules as operator-friendly source of truth
- SQLite persistence for events, actions, and poll schedules
- typed context contracts for event yields and action expects
- a Telegram plugin that uses the same event-production pipeline for webhook ingress and polling

### Quick start

```
python -m venv .venv
. .venv/bin/activate
pip install -e .[dev]

call-baxter --config-file examples/config.yml health
call-baxter --config-file examples/config.yml plugins list
call-baxter --config-file examples/config.yml context event show telegram.message.new
call-baxter --config-file examples/config.yml ingress run --name telegram.update --payload-file examples/
telegram-update.json
call-baxter --config-file examples/config.yml events list
```

Run the daemon over TCP for local development:

```
call-baxterd --config-file examples/config.yml --port 8943
call-baxter --config-file examples/config.yml --daemon-url http://127.0.0.1:8943 health
```

Run the daemon over a Unix socket for deployment:

```
call-baxterd --config-file ../../infra/deploy/call-baxter.yml --database-url ../../infra/data/cb.db --uds ../../
infra/run/cb.sock
call-baxter \
  --config-file ../../infra/deploy/call-baxter.yml \
  --daemon-url http+unix:///absolute/path/to/cb.sock \
  health
```

The UDS-aware `call-baxter health` command is suitable for local service checks and Docker health checks without opening a TCP listener. When `CALL_BAXTER_API_KEY` is set for the daemon, the CLI automatically uses the same environment variable. An explicit `--daemon-api-key` value is also supported. HTTP authentication and server errors make the health command exit non-zero.

### Configuration behavior

Every YAML string value supports environment expansion:

```
plugins:
  - module: call_baxter.plugins.telegram
    config:
      bot_token: ${TELEGRAM_BOT_TOKEN}
      api_base: ${TELEGRAM_API_BASE:-https://api.telegram.org}
```

`${VAR}` expands to the environment value or an empty string. `${VAR:-default}` uses the default when the variable is unset or empty. The daemon watches its configuration file by default and atomically reloads valid changes. Invalid replacements are logged and the active runtime remains unchanged. Configure the watcher with:

- `CALL_BAXTER_CONFIG_RELOAD_ENABLED=false` to disable it.
- `CALL_BAXTER_CONFIG_RELOAD_SECONDS=1.0` to change the polling interval.

The built-in `call_baxter.plugins.file_out` helper provides a bounded local text sink for rules that need to emit reports, prompts, or handoff files. It must be configured with an explicit base directory; action paths are relative to that directory and escapes are rejected:

```
plugins:
  - module: call_baxter.plugins.file_out
    config:
      base_dir: /srv/data/file-out
      max_bytes: 1048576
      create_parents: true
```

Rules can then invoke `file.out` with `path`, `content`, and optional `mode: append`. The default mode is `write`. Because append is available, the action is intentionally classified as a local non-idempotent side effect and is not action-replay eligible.

Webhook endpoints acknowledge asynchronously by default, while ordinary ingress stays synchronous:

```
POST /v1/webhooks/{name}            # async by default
POST /v1/webhooks/{name}?async=false
POST /v1/ingress/{name}             # sync by default
POST /v1/ingress/{name}?async=true
GET  /v1/ingress/results/{receipt}
```

Async calls return `{"accepted": true, "ref": "ing:...", "status": "accepted"}`. Receipt state is process-local and intended for short-lived diagnostics, not durable workflow storage. Terminal receipts are trimmed when the in-process limit is reached; accepted and processing receipts are never evicted before their work finishes.

The operational acceptance matrix and exact regression-test mapping live in `docs/operational-todo-validation.md`.

## What this scaffold proves

- The common ground is a normalized event envelope, not a Telegram-shaped payload.
- Plugins contribute event definitions, action definitions, pollers, and ingress adapters.
- Rules are loaded from YAML, matched against events, and trigger actions.
- Schedules are synced from config and can run due pollers.
- Telegram can be folded into this runtime for event-driven automations while still keeping a separate Telegram projection/index in `tg-agent-cli`.

See the docs folder for architecture, CLI semantics, schema, daemon API, Telegram notes, interop notes, and deployment guidance.

# 3.2 Architecture

## Runtime model

The runtime is built around four primitives:

- `EventEnvelope`: the normalized fact published into the runtime
- `RuleSpec`: declarative match + actions loaded from YAML
- `ActionInvocation`: a structured action request with optional `on_success` / `on_error` follow-ups
- `ScheduleSpec`: a persisted poller schedule

## Functional division

### Daemon

The daemon owns:

- loading config and plugin modules
- loading YAML rules from `rules.d`
- keeping the SQLite state
- serving HTTP control routes
- running ingress adapters, pollers, and action execution

### CLI

The CLI is a thin operator surface. It can:

- operate locally against the same runtime implementation
- or speak HTTP to the daemon using `--daemon-url`

This keeps the command surface stable while the daemon stays the source of truth when running as a long-lived service.

## Context contracts

Each event definition declares a `yields` context schema. Each action definition declares an `expects` context schema.

The runtime validates both before persisting or executing work. This provides a low-friction but explicit contract for:

- what a plugin promises to emit
- what an executor requires to run

## Plugins

Plugins are discovered from an explicit list in the YAML config using `importlib.import_module(...)` and a `register(registry)` function.

This start version intentionally avoids depending on entry-point discovery, but leaves room for it later.

Configured plugin modules must expose a callable `register()` function. Event kinds, action kinds, ingress names, and poller names are unique runtime contracts; a second registration now rejects startup or reload rather than silently replacing the first implementation.

## Deterministic rule policy

Rule matching is independent of rule-directory iteration and caller list order. Matching candidates are evaluated by descending integer `priority` (default `0`), with lexical `rule.id` as the stable tie-breaker. Rules that do not opt into policy fields still all execute when they match; only their relative order is made deterministic.

An optional `conflict_group` makes matching rules mutually exclusive for one event. The first candidate in deterministic rule order owns the group, and later matches in that group are recorded as `conflict_skipped`. Ownership is established before rate-limit evaluation, so a throttled or malformed higher-priority policy cannot silently fall through to a lower-priority action and create operator spam.

Optional rule-level `rate_limit` uses a fixed UTC window and a scalar value selected by `key_attr` from `event.attrs` (dotted paths are supported). `max_count` successful policy admissions are allowed per `(rule id, typed key, window)`. Windows are aligned to Unix epoch boundaries using runtime receipt time, not caller-controlled event timestamps. Integer and non-empty string keys are supported and type-prefixed in storage; missing, `null`, boolean, container, or otherwise invalid keys fail closed without creating an action record. SQLite `BEGIN IMMEDIATE` serialization makes the counter shared across daemon threads and processes rather than giving each worker an independent bucket.

Rules can define reusable `variables`. Variable definitions are resolved once from `event.*` and other `vars.*` values before actions run. References are validated for undefined names and cycles during rule load/reload, and runtime-missing event values fail closed instead of leaving template text in an operator action. Exact substitutions such as `{{ vars.chat }}` preserve integers, objects, lists, and `null` values; string interpolation converts resolved values to text. Existing `event.*` and action-follow-up `result.*` templates remain compatible.

Policy decisions are separately persisted as `selected`, `conflict_skipped`, `rate_limited`, `rate_limit_key_missing`, or `rate_limit_key_invalid`. A `selected` outcome means policy admitted the rule; action success or failure remains authoritative in the existing durable action records.

## Reload integrity

The runtime remembers the absolute configuration-file path. `POST /v1/reload` rereads that file, expands environment variables, loads every plugin, and strictly validates the full rule set before publishing any replacement state.

Reload uses a prepare/commit sequence:

1. parse the new configuration;
2. build a replacement registry and rule list;
3. reconcile all schedule rows in one SQLite transaction;
4. atomically swap the in-memory configuration, registry, and rules.

Malformed rules, duplicate rule ids, plugin-load errors, or schedule transaction failures leave the active runtime unchanged. Schedules removed from configuration are disabled instead of continuing from stale database rows.

## Schedule execution leases

Due schedules are claimed in SQLite before their poller runs. The claim prevents the background loop, manual `run-due` calls, concurrent threads, or multiple daemon replicas from starting the same schedule simultaneously. Claims expire so another process can recover work after a crash. `execution_lease_seconds` can override the default lease for an individual schedule.

While a poller is running, a lightweight heartbeat renews the claim. The default heartbeat interval is the smaller of 30 seconds and one third of the lease. `execution_heartbeat_seconds` can shorten it, but it is capped at one third of the lease so multiple renewal opportunities remain. A claim cannot be renewed or completed after expiry, and a worker that loses ownership reports an execution error rather than marking the schedule complete.

The renewable lease provides at-most-one live owner while SQLite remains reachable. Backoff state is updated only after fenced completion, so an obsolete worker cannot clear or impose retry policy for a newer owner. It does not transactionally roll back arbitrary external side effects, so pollers should still use stable upstream offsets or idempotency keys. A database outage lasting longer than the lease is reported as a lease-loss error and may permit recovery by another replica.

## SQLite initialization

WAL mode is enabled during serialized schema initialization with bounded lock retries. Ordinary database connections never attempt to change journal mode, avoiding first-use races between daemon replicas while retaining foreign-key enforcement and a five-second busy timeout.

## Failure semantics

Ingress and poller event-publication errors propagate to the caller. Webhooks therefore receive an error response instead of a misleading success, and scheduled pollers enter their configured error backoff. Rule files are strict at runtime, while the standalone loader retains a lenient mode for diagnostics.

Parallel rules wait for every submitted action and propagate the first contract or template exception after collecting completed audit references. Executor-level failures remain normal `ActionResult(ok=False)` records.

For rules configured with both `execution_mode: parallel` and `stop_on_error: true`, actions execute sequentially. Starting every action concurrently would make the stop guarantee impossible to honor.

Action follow-ups are recursive. Each `on_success` or `on_error` action renders against the result of its immediate parent, and its own follow-ups run in turn. A failed follow-up marks the complete chain failed; this allows `stop_on_error` to stop later top-level actions when delivery, acknowledgement, or recovery work did not succeed.

## 3.3 CLI surface

`cb` is the operator CLI for Call Baxter. Running `cb` with no arguments, or running a command group such as `cb events` without a subcommand, prints the relevant help and exits successfully. CLI parsing uses the standard Python `argparse` surface; malformed input is reported as normal usage text, and runtime/configuration failures are reduced to one `cb: error: ...` diagnostic instead of a Pydantic validation dump or Python traceback.

Global connection options must come before the command:

```
cb [--config-file PATH] [--database-url URL_OR_PATH] [--daemon-url URL] COMMAND ...
```

Local mode requires a runtime config. You may either pass it explicitly or set the operator environment once:

```
export CB_CONFIG_FILE=/srv/config/call-baxter.yml
export CB_DATABASE_URL=/srv/data/cb.db

cb health
cb plugins list
cb rules list
```

Remote daemon mode does **not** require a local config file:

```
cb --daemon-url http://127.0.0.1:8943 health
cb --daemon-url http+unix:///srv/run/cb.sock health
```

If daemon admin authentication is enabled, set `CALL_BAXTER_API_KEY` or pass `--daemon-api-key TOKEN`. `CB_DAEMON_URL` can provide the daemon URL by environment. YAML string values continue to support `${VAR}` and `${VAR:-default}` expansion.

Output defaults to stable pretty JSON for script compatibility. `--output text` provides a compact operator-oriented form for simple lists and records.

### Health, config, and reload

```
cb health
cb config show
cb reload
```

`cb config show` prints the active effective configuration in both local and remote daemon mode. It includes the resolved config path and post-environment-expansion values, but recursively replaces passwords, tokens, API keys, authorization values, credentials, and private keys with `<redacted>`. Remote mode reads the authenticated daemon `GET /v1/config` endpoint, so the displayed config is the runtime's current configuration after reload rather than an unrelated local file.

`health` exits non-zero unless the daemon/runtime returns `"ok": true`, which keeps it useful for Docker and service-manager checks. Daemon startup logs also state the resolved config path.

### Plugins and rules

```
cb plugins list
cb rules list
cb rules list --kind telegram.message.new
```

### Context contracts

```
cb context event list
cb context event show telegram.message.new
cb context action list
cb context action show telegram.send_message
```

These commands expose the data contracts available to rules, templates, and action invocations.

## Events

Preferred syntax uses positionals for the required identity and payload arguments:

```
cb events list [--kind KIND]
cb events publish KIND SOURCE ATTRS_FILE [--subject-ref REF] [--occurred-at ISO_TIMESTAMP]
```

Example:

```
cb events publish system.heartbeat manual ./heartbeat.json
```

The pre-0.7.8 flag spelling remains accepted for compatibility:

```
cb events publish --kind KIND --source SOURCE --attrs-file ATTRS_FILE
```

`ATTRS_FILE` must contain a JSON object.

## Actions

```
cb actions list [--kind KIND]
cb actions execute KIND ARGS_FILE [--subject-ref REF]
```

The old form remains accepted:

```
cb actions execute --kind KIND --args-file ARGS_FILE [--subject-ref REF]
```

`ARGS_FILE` must contain a JSON object.

## Ingress adapters

```
cb ingress list
cb ingress runs [--ingress-name NAME] [--limit N]
cb ingress run NAME PAYLOAD_FILE
cb ingress reingest INGRESS_REF
```

Compatibility form:

```
cb ingress run --name NAME --payload-file PAYLOAD_FILE
```

Ingress adapters normalize source-native payloads into internal `EventEnvelope` instances.

## Pollers

```
cb pollers list
cb pollers run NAME [CONFIG_FILE]
```

The old `cb pollers run --name NAME [--config-file CONFIG_FILE]` spelling remains accepted.

## Schedules

```
cb schedules list
cb schedules sync
cb schedules run-due
```

Schedules are sourced from the YAML config and mirrored into SQLite.

## Policy evidence

```
cb policy outcomes [--event-ref REF] [--rule-id ID] [--outcome OUTCOME] [--limit N]
cb policy simulate EVENT_REF
cb policy replay EVENT_REF
cb policy replay-eligibility EVENT_REF [--limit N]
```

These are read-only policy/audit operations; they do not add an action-replay execution path.

## Plugin-specific CLI boundaries

Plugin-specific operator flows live outside the generic command modules. Telegram currently owns its dedicated helper namespace:

```
cb telegram reingest EVENT_REF
```

This takes the stored event's `attrs.raw_update` and sends it back through the `telegram.update` ingress. `cb events reingest-telegram EVENT_REF` remains as a compatibility alias, but its implementation lives in the Telegram CLI extension rather than the generic events code.

Generic plugin contracts remain generic: use `cb actions`, `cb ingress`, and `cb pollers` for plugin-defined action/ingress/poller names. Separately packaged integrations can ship their own small operator CLI; for example the WordPress package provides `cb-wp verify` and `cb-wp categories`.

# 3.4 Daemon HTTP API

## Health, config, and reload

- `GET /v1/health`
- `GET /v1/config`
- `POST /v1/reload`

`GET /v1/config` returns the active effective runtime configuration, including the resolved configuration path and post-environment-expansion values. Secret-like fields such as passwords, tokens, API keys, authorization values, credentials, and private keys are recursively replaced with `<redacted>`. The route is an admin `/v1/*` endpoint and therefore uses the same `CALL_BAXTER_API_KEY` protection as the other control-plane reads when admin authentication is enabled. Daemon startup logs also state the resolved config path without dumping secret values.

`POST /v1/reload` rereads the original YAML configuration file. It validates plugins and the complete rule set, reconciles schedules transactionally, disables removed schedules, and only then swaps the live runtime state. Invalid or duplicate rules return an error and leave the current runtime active.

The daemon also watches the configuration file by default. Changes are reloaded through the same validated atomic path. Set `CALL_BAXTER_CONFIG_RELOAD_ENABLED=false` to disable the watcher or `CALL_BAXTER_CONFIG_RELOAD_SECONDS` to change its interval.

For a daemon bound only to a Unix socket, run:

```
call-baxter \
  --config-file /etc/call-baxter/config.yml \
  --daemon-url http+unix:///run/call-baxter/cb.sock \
  health
```

When daemon admin authentication is enabled, set `CALL_BAXTER_API_KEY` for both the daemon and the health-check process, or pass `--daemon-api-key`. The HTTP client raises on 4xx/5xx responses and `health` exits non-zero unless the response contains `"ok": true`, so Docker does not mistake an authentication error for a healthy service.

## Registry and rule inspection

- `GET /v1/plugins`
- `GET /v1/rules`
- `GET /v1/context/events`
- `GET /v1/context/actions`

## Events and actions

- `GET /v1/events`
- `POST /v1/events/publish`
- `GET /v1/actions`
- `POST /v1/actions/execute`

## Ingress and pollers

- `GET /v1/ingress`
- `GET /v1/ingress/runs` — list durable ingress ledger entries
- `POST /v1/ingress/runs/{ingress_ref}/reingest` — rerun a stored payload through its original ingress
- `POST /v1/ingress/{name}` — synchronous by default; add `?async=true` for a receipt
- `POST /v1/webhooks/{name}` — asynchronous by default; add `?async=false` for the result
- `GET /v1/ingress/results/{receipt_ref}` — inspect an in-process async receipt
- `GET /v1/pollers`
- `POST /v1/pollers/{name}/run`

Async ingress validates authentication and the ingress name before returning HTTP 200. Processing then runs outside the ASGI event loop. Start logs include the ingress and a safe payload-kind classification. Completion logs include emitted event references with their source/kind, matched-rule identifiers, action statuses, and failures. A receipt progresses through `accepted`, `processing`, and either `complete` or `failed`.

Receipt storage is process-local and capped for diagnostics. Only terminal `complete` or `failed` receipts are eligible for trimming; accepted and processing receipts remain available until their work reaches a terminal state.

Ingress runs are also recorded durably in SQLite before handler execution. The ledger preserves the ingress name, original payload, terminal status, result, and error text. Use `call-baxter ingress runs` to find a run and `call-baxter ingress reingest ing:123` to reintroduce that payload under the currently loaded config and rules.

The synchronous `/v1/events/publish` contract is unchanged.

`telegram.updates` schedules should configure an explicit `offset` or one of `projection_database_url`, `database_url`, or `tg_database_url`. The daemon logs a startup warning when an enabled Telegram polling schedule lacks all offset persistence.

## YAML environment expansion

All YAML string values support `${VAR}` and `${VAR:-default}`. The latter uses its default when the environment variable is unset or empty. Expansion happens before Pydantic validation and before every manual or automatic reload.

## Schedules

- `GET /v1/schedules`
- `POST /v1/schedules/sync`
- `POST /v1/schedules/run-due`

Schedule execution is protected by SQLite leases, so a manual `run-due` request cannot race the daemon background loop or another replica into executing the same due schedule twice.

## 3.5 Operational TODO validation

The eight deployment issues recorded in the repository-root `TODO.yml` were implemented for 0.7.0 and re-audited as release blockers for 0.7.1. Each item now carries `validated_in: 0.7.1` and one or more pytest node IDs in `regression_tests`.

### Acceptance matrix

| ID | Verified behavior | Regression evidence |
|---|---|---|
| `ASYNC-001` | A slow webhook returns HTTP 200 in under 50 ms, the real ingress → event → rule → action pipeline finishes afterward, background exceptions become failed receipts, and `/v1/events/publish` remains synchronous. | `test_async_ingress_todos_are_end_to_end_and_publish_stays_synchronous`; `test_receipt_retention_never_evicts_active_ingress` |
| `ASYNC-002` | `/v1/webhooks/*` defaults to async, `/v1/ingress/*` defaults to sync, and callers can override the mode with `?async=true\|false`. | `test_async_ingress_todos_are_end_to_end_and_publish_stays_synchronous` plus daemon route contract te |
| `ASYNC-003` | Start, completion, and failure logs include receipt identity, payload kind, event source/kind, matched rules, and action statuses. | `test_async_ingress_todos_are_end_to_end_and_publish_stays_synchronous` |
| `OPS-001` | Startup emits one warning for an enabled `telegram.updates` schedule without an explicit offset or projection database. | `test_telegram_poll_schedule_warns_at_startup_without_offset_persistence` |
| `OPS-002` | `sqlite:///absolute/path.db` resolves to and opens that exact path; ambiguous SQLite URI spellings are rejected. | `test_absolute_sqlite_uri_opens_the_requested_database`; `test_resolve_database_path_rejects_ambiguous_` |
| `OPS-003` | `${VAR}` and `${VAR:-default}` are present in CLI help and expand before config validation. | `test_cli_help_documents_environment_expansion`; `test_runtime_file_config_expands_env_variables` |
| `OPS-004` | Replacing the mounted config activates a new rule without restart; malformed replacement config is logged and leaves the active rules unchanged. | `test_config_file_changes_reload_live_and_invalid_config_is_atomic` |
| `OPS-005` | The real Uvicorn daemon is started on a Unix socket and reached through `call-baxter health`; admin authentication succeeds through `CALL_BAXTER_API_KEY`, while an unauthenticated client gets an HTTP error. | `test_cli_health_reaches_authenticated_daemon_over_real_uds` |

## Running the acceptance checks

Run the focused tests:

```
(cd packages/call-baxter && \
  uv run --extra dev python -m pytest -q tests/test_operational_todo_acceptance.py)

(cd packages/tg-agent-cli && \
  uv run --extra dev python -m pytest -q tests/test_cli_settings.py)
```

Validate that every TODO remains closed and references an existing pytest function:

```
bash scripts/verify-operational-todos.sh
```

The repository-wide release gate runs those tests, metadata validation, linting, typing, package builds, Compose validation, documentation builds, and coverage floors:

```
./scripts/verify.sh
```

## Scope and durability notes

Async receipts are intentionally process-local diagnostic records. They are not a durable queue and do not survive daemon restart. The acceptance requirement is that acknowledged work remains tracked until it completes or fails in the active process; active receipts are therefore excluded from retention trimming.

Automatic reload watches the configured YAML file. Rule-only edits should be followed by touching or atomically replacing that config file, or by calling `POST /v1/reload`.

# 3.6 Plugin system

## Start version

The scaffold intentionally starts with the simplest workable discovery model:

```yaml
plugins:
  - call_baxter.plugins.system
  - call_baxter.plugins.telegram
```

Each plugin module exports:

```python
def register(registry, config: dict[str, object] | None = None) -> None:
    ...
```

The daemon imports each configured module with `importlib.import_module(...)` and calls `register(registry, config)` when the function accepts a second argument.

Both simple and configured forms are supported:

```yaml
plugins:
  - call_baxter.plugins.system
  - module: call_baxter.plugins.telegram
    config:
      database_url: ./tg-state.db
      media_storage_mode: file
```

## Reload behavior

`POST /v1/reload` and `cb reload` currently do three things:

1. rebuild the registry
2. re-import configured plugins and call `register(...)`
3. re-load YAML rules and re-sync schedules

This is a safe start version because it does not attempt hot code reloading of arbitrary Python module changes. For code changes, restarting the daemon remains the clean path. For YAML changes, `cb reload` is usually enough.

## Optional future extensions

- optional Python entry-point discovery in addition to explicit module lists
- file watching for `rules.d/*.yml`
- separate plugin-owned projection stores for browse-heavy sources like Telegram

## Decorator-based plugin DSL (PluginBuilder)

New plugins can use `PluginBuilder` from `call_baxter.dsl` to reduce boilerplate.

```
from call_baxter.dsl import PluginBuilder
from call_baxter.registry import Registry

plugin = PluginBuilder("myplugin")

@plugin.event("myplugin.thing", description="A thing happened",
              yields={"ref": "string", "label?": "string"})
def _(): pass

@plugin.poller("myplugin.check", description="Poll for things")
def poll(config: dict) -> list:
    ...

@plugin.action("myplugin.do", description="Do something",
               expects={"target": "string"})
def do_action(plugin_config: dict, action, runtime):
    # plugin_config injected automatically when first param is named 'plugin_config'
    ...

def register(registry: Registry, config=None):
    plugin.register_all(registry, config)
```

### Field shorthand

`yields` and `expects` accept dicts of `"field_name": "type"`. Keys ending in `?` become optional fields; all others are required.

Supported types: `string`, `integer`, `number`, `boolean`, `object`, `array`, `datetime`.

### Available plugins

| Module | Poller(s) | Action(s) | Events |
|---|---|---|---|
| `call_baxter.plugins.system` | `system.heartbeat` | `log.write` | `system` |
| `call_baxter.plugins.telegram` | `telegram.updates` | `telegram.send_message`, `telegram.answer_callback_query`, `telegram.persist_update` | `telegra` |
| `call_baxter.plugins.http_poll` | `http.check` | `http.request` | `http.ch` |
| `call_baxter.plugins.docker_watch` | `docker.containers` | `docker.restart` | `docker` |
| `call_baxter.plugins.mqtt` | `mqtt.subscribe` | `mqtt.publish` | `mqtt.me` |
| `call_baxter.plugins.caldav` | `caldav.agenda` | `caldav.add_event`, `caldav.add_todo`, `caldav.complete_todo` | `caldav` `caldav` |
| `call_baxter.plugins.zrok` | — | `zrok.run` | — |
| `call_baxter.plugins.sonic_byte` | — | `sonic_byte.inspect`, `sonic_byte.render`, `sonic_byte.play` | — |

See Sonic Byte plugin for secure command configuration, sound-design options, and complete rule examples.

## 3.7 Sonic Byte plugin

The built-in `call_baxter.plugins.sonic_byte` plugin turns Call Baxter events into deterministic musical notifications by invoking the external `sonic-byte-codec` command.

The plugin deliberately does not copy the synthesizer into Call Baxter. It provides a small, constrained process boundary around the independently usable script and exposes three actions:

| Action | Purpose |
|---|---|
| `sonic_byte.inspect` | Return the lossless musical mapping of byte payloads as JSON. |
| `sonic_byte.render` | Render payloads to stereo WAV files below an allowlisted output directory. |
| `sonic_byte.play` | Trigger playback; detached by default so event processing is not blocked. |

### Prerequisite

Install the script from the sibling `scripting` project:

```
cd /home/user/code/scripting
./install-scripts.sh install sonic-byte-codec
sonic-byte-codec 0xAB --compact
```

The Call Baxter daemon user must be able to resolve `sonic-byte-codec` through `PATH`. Alternatively, configure an explicit command prefix:

```
plugins:
  - module: call_baxter.plugins.sonic_byte
    config:
      command:
        - uv
        - run
        - --script
        - /home/user/code/scripting/media/sonic-byte-codec.py
```

Command arrays are executed directly. No shell is used and action values cannot inject additional flags outside the plugin's structured option set.

### Recommended configuration

```
plugins:
  - call_baxter.plugins.system
  - module: call_baxter.plugins.sonic_byte
    config:
      command: sonic-byte-codec
      detached_playback: true
      allow_playback: true
      timeout: 30
      max_timeout: 120
      output_dir: /var/lib/call-baxter/sonic-byte
      allowed_players: [pw-play, paplay, aplay, ffplay, play]
      player: pw-play
      sound:
        root: C3
        waveform: fm
        space: room
        gain: 0.9
        cutoff: 7200
```

Important controls:

- `command`: executable string or argument array. Use an array for `uv run --script`.
- `cwd`: optional working directory for the child process.
- `output_dir`: root directory for relative WAV outputs.
- `allow_absolute_output`: disabled by default.
- `allow_playback`: permits or disables the `sonic_byte.play` action.
- `detached_playback`: defaults to `true` for non-blocking notifications.
- `allowed_players`: allowlist for per-action player selection.
- `max_payload_tokens`, `max_token_bytes`, `max_payload_bytes`: process-input limits.
- `sound`: default render/play parameters overridden by action arguments.

## Trigger playback from a rule

```yaml
id: telegram-sonic-byte-alert
enabled: true
match:
  source: telegram
  kind: telegram.message.new
  attrs:
    text: byte me
actions:
  - kind: sonic_byte.play
    args:
      payload: ["text:message received"]
      root: Bb2
      waveform: pulse
      space: dub
      gain: 0.85
```

Payload entries use the script's explicit token forms:

```yaml
payload: ["0xAB"]
payload: ["171", "0b00000001"]
payload: ["hex:deadbeef"]
payload: ["text:backup complete"]
payload: ["file:/var/lib/call-baxter/state.bin"]
```

`file:` payloads are read by `sonic-byte-codec` with the daemon user's permissions. Only use them with trusted rule/configuration authors.

## Render a retained WAV

```yaml
- kind: sonic_byte.render
  args:
    payload: ["text:deploy complete"]
    output: deploy-complete.wav
    root: C2
    waveform: saw
    space: room
```

Relative outputs remain inside `output_dir`; `..` escapes and non-WAV extensions are rejected before starting a child process. The action result contains `output_path` and the parsed JSON render report.

## Synchronous playback

Detached playback is appropriate for notifications. Use synchronous playback when a subsequent action must depend on completion:

```yaml
- kind: sonic_byte.play
  args:
    payload: ["0xAB"]
    wait: true
    timeout: 20
```

The action then returns the child exit code, stdout, and stderr. A timeout or non-zero exit is represented as a failed Call Baxter action and can activate the rule's `on_error` chain.

## 3.8 SQLite schema

The scaffold keeps the database intentionally small:

- `events` : persisted normalized event envelopes
- `actions` : action execution requests and results
- `schedules` : poller schedules mirrored from config

This is enough for:

- event history
- action audit history
- scheduled poll execution

If a source later needs a browse-optimized projection, it should be added as a plugin-owned projection model rather than inflating the generic core.

## 3.9 Telegram plugin sketch

The Telegram plugin in this scaffold contributes:

- event definitions:
- `telegram.message.new`
- `telegram.message.edited`
- `telegram.channel_post.new`
- `telegram.channel_post.edited`
- `telegram.business_message.new`
- `telegram.business_message.edited`
- `telegram.callback_query.new`
- `telegram.media.present`
- `telegram.message.reaction`
- action definitions:
- `telegram.send_message`
- `telegram.answer_callback_query`
- `telegram.persist_update`
- ingress adapter:
- `telegram.update`
- poller:
- `telegram.updates`

### Why this matters

If the runtime already exposes HTTP ingress, a Telegram webhook can hit the callback runtime directly. That means a dedicated Telegram daemon is no longer required for event-driven automation.

Webhook ingress and polling both flow through the same Telegram plugin update pipeline:

```
raw update
  -> optional projection persistence into tg-agent-cli db
  -> normalized telegram.* events
  -> generic call-baxter rule matching
```

### What `telegram.callback_query.new` does

This event represents an incoming callback query, typically triggered by a user pressing an inline keyboard button. It carries:

- `callback_query_id`
- `callback_ref`
- callback `data`
- optional `from_user_id`
- the raw update payload
- optional originating `chat_ref` / `message_ref`

That makes it the event to match when you want to:

- answer the callback query
- branch on button payload data
- correlate the button press with the message that contained the inline keyboard

Message events also expose optional `from_is_bot`. This is useful when a callback update includes the original bot-authored message and a rule must avoid recursively processing that synthetic parent message.

## Inline keyboards

`telegram.send_message` accepts an optional structured `reply_markup` object. For example:

```
- kind: telegram.send_message
  args:
    chat_id: "{{ event.attrs.chat_id }}"
    text: Choose an action
    reply_markup:
      inline_keyboard:
        - - text: Status
            callback_data: cmd:status
```

The Telegram API client also accepts a JSON string for compatibility, validates that it decodes to an object, and sends it as a JSON object in the Bot API request.

The client validates Telegram text boundaries before making a request:

- `telegram.send_message` requires 1–4096 characters;
- `telegram.answer_callback_query` accepts at most 200 characters.

This makes invalid rules fail as audited action results locally instead of depending on a remote Bot API rejection.

## What can move into `call-baxter`

Good fits for the generic runtime:

- webhook ingestion of raw Telegram updates
- `getUpdates` polling through the Telegram poller
- normalization of updates into events
- outbound actions like `sendMessage` and `answerCallbackQuery`
- YAML rules reacting to normalized message or callback-query events

Still valuable as a source-specific projection when needed:

- rich browse-first chat history UI
- Telegram-specific media storage policies
- advanced message/thread/entity indexing
- high-fidelity Bot API coverage across all update and media variants

So the pragmatic split is:

- use `call-baxter` as the event runtime and control plane
- keep `tg-agent-cli` as a separate browse/projection tool
- let the Telegram plugin persist into the Telegram projection DB directly when configured
- keep `telegram.persist_update` available as an explicit action for unusual flows

## 3.10 `call-baxter` + `tg` interoperability

### Recommended split

Keep the projects separate:

- `call-baxter`
- generic daemon / CLI / rule engine
- webhook ingress, pollers, schedules, and rule execution
- `tg-agent-cli`
- Telegram projection store
- browse-first CLI for chats, messages, callbacks, media, and actions

`tg-agent-cli` does **not** need a resident daemon for this integration model. The Telegram plugin inside `call-baxter` can receive webhook payloads or polled updates, persist those updates into the Telegram projection database immediately, and then emit the normalized `telegram.*` events that rules match on.

### Flow

```
Telegram webhook or poller
  -> call-baxter telegram plugin shared update pipeline
  -> persist raw update into tg projection db (optional but recommended)
  -> emit telegram.message.* / telegram.callback_query.new / telegram.media.present / telegram.message.reaction
  -> generic call-baxter rules match on telegram.*
```

### Plugin config example

```yaml
plugins:
  - call_baxter.plugins.system
  - module: call_baxter.plugins.telegram
    config:
      bot_token: "123456:abc"
      api_base: "https://api.telegram.org"
      projection_database_url: ./tg-state.db
      projection_media_storage_mode: file
      projection_media_base_dir: ./media
```

### Optional direct persist action

The plugin still exposes `telegram.persist_update` as an explicit action for manual or nonstandard flows, but the default interoperable path is to let the plugin persist before event publication.

### Event types contributed by the Telegram plugin

- `telegram.message.new`
- `telegram.message.edited`
- `telegram.channel_post.new`
- `telegram.channel_post.edited`
- `telegram.business_message.new`
- `telegram.business_message.edited`
- `telegram.callback_query.new`
- `telegram.media.present`
- `telegram.message.reaction`

`telegram.callback_query.new` represents an incoming button press / callback query. It carries the callback query id plus the callback data payload and, when Telegram supplied a parent message, the originating chat/message refs. That makes it the event to match when you want to answer a callback query, branch on button data, or correlate the press with the message that contained the inline keyboard.

## Poller

The plugin also contributes:

- `telegram.updates`

If `offset` is not given explicitly and a Telegram projection database is configured, the poller derives `offset = max(update_id) + 1` from the projection DB so repeated schedule runs stay incremental.

## Wildcards and propagation

The runtime supports wildcard rule kinds such as `telegram.*` through shell-style matching, and an event can set `cancel_propagation: true` to stop matching after the first rule chain finishes.

## Replacing a TGEX deployment

For the compatibility launcher, legacy environment variables and routes, WordPress parity, cutover checklist, and rollback procedure, see Replace TGEX with Call Baxter.

# 3.11 Replace TGEX with Call Baxter

Call Baxter `v0.8.0` includes a TGEX/`telegras2` compatibility mode. It preserves the operational contracts used by a running TGEX deployment while executing the Call Baxter runtime, Telegram projection package, and WordPress plugin.

The compatibility layer is intended for a low-risk service cutover. It is not a source-level emulation of TGEX internals.

## Install the coordinated release

Install all four `v0.8.0` coordinated Python artifacts into the same environment:

```
python -m pip install \
  call_baxter-0.8.0-py3-none-any.whl \
  tg_agent_cli-0.8.0-py3-none-any.whl \
  cb_wordpress_plugin-0.8.0-py3-none-any.whl \
  cb_social_plugin-0.8.0-py3-none-any.whl
```

The `call-baxter` package installs both legacy command names:

```
telegras
telegras2
```

Native Call Baxter commands remain available as `call-baxter`, `cb`, and `call-baxterd`.

## Compatibility matrix

| TGEX surface | Call Baxter replacement | Compatibility notes |
|---|---|---|
| `telegras2 start` | Same command | Starts the Call Baxter ASGI application. `--host`, `--port`, and `--reload` remain accepted. |
| `telegras2 polling` | Same command | Uses the `telegram.updates` poller and the automatic Call Baxter schedule runner. `--stop-after-updates` is supported for finite/test runs. |
| Deployment validation | `telegras2 doctor` | Performs offline checks by default; add `--online` to verify the Telegram token and webhook state. JSON output is available for automation. |
| Historical TGEX data | `telegras2 migrate-history` | Imports interactions, route executions, and handler executions idempotently. `--dry-run`, `--limit`, and a separate `--source-database` are supported. |
| Webhook mode | Same configured path | `WEBHOOK_PATH=webhook/{secret}` and `WEBHOOK_PREFIX` are resolved to the old URL path. |
| `GET /healthz` | Same path | Returns the legacy service-shaped health response. Native health remains available at `/v1/health`. |
| Introspection API | Same `/internal/introspection/*` paths | Authentication and primary operational responses are preserved. Imported TGEX route/handler history is merged with current Call Baxter events/actions. |
| Telegram persistence | `tg-agent-cli` projection store | New updates, chats, messages, callbacks, media, actions, polling runs, and webhook snapshots are persisted. |
| Telegram CLI operations | `tg` CLI and legacy webhook commands | The dedicated `tg` CLI is the richer native operator interface. |
| WordPress mirroring | `cb-wordpress-plugin` | Supports text/entities, media, media groups, edits, idempotency, post types, default terms/author, filtering, dry-run, retry, and nonfatal media failures. |
| TGEX Python plugins | Call Baxter plugins and YAML rules | The Python plugin ABI is different. `PLUGINS` / `BOT_PLUGINS` values are detected and warned about, but not imported. Port custom plugins before final cutover. |

## Preserved environment variables

The launcher accepts the TGEX names and aliases below.

#### Runtime and Telegram

```
BOT_TOKEN / TELEGRAM_BOT_TOKEN / TGCLI_BOT_TOKEN
BOT_MODE / MODE
WEBHOOK_SECRET / BOT_WEBHOOK_SECRET / TELEGRAM_WEBHOOK_SECRET
WEBHOOK_PATH / BOT_WEBHOOK_PATH
WEBHOOK_PREFIX
WEBHOOK_URL / BOT_WEBHOOK_URL
TELEGRAM_WEBHOOK_HEADER_SECRET / WEBHOOK_HEADER_SECRET / BOT_WEBHOOK_HEADER_SECRET
PERSISTENCE_ENABLED / BOT_PERSISTENCE_ENABLED
DATABASE_URL / BOT_DATABASE_URL / TGCLI_DATABASE_URL
POLLING_TIMEOUT / BOT_POLLING_TIMEOUT / BOT_POLLING_TIMEOUT_SECONDS
POLLING_IDLE_SLEEP / BOT_POLLING_IDLE_SLEEP / BOT_POLLING_IDLE_SLEEP_SECONDS
POLLING_ERROR_BACKOFF / BOT_POLLING_ERROR_BACKOFF / BOT_POLLING_ERROR_BACKOFF_SECONDS
HOST / BOT_HOST
PORT / BOT_PORT
RELOAD / BOT_RELOAD
LOG_LEVEL / BOT_LOG_LEVEL
INTROSPECTION_ENABLED / BOT_INTROSPECTION_ENABLED
INTROSPECTION_TOKEN / BOT_INTROSPECTION_TOKEN
PLUGINS / BOT_PLUGINS
```

#### WordPress bridge

```
WORDPRESS_BRIDGE_ENABLED
WP_BASE_URL
WP_USERNAME
WP_APP_PASSWORD
WP_POST_TYPE
WP_POST_STATUS / WP_PUBLISH_STATUS
WP_CATEGORIES
WP_TAGS
WP_DEFAULT_AUTHOR
WP_DRY_RUN
WP_USE_FEATURED_MEDIA
WP_REQUIRED_HASHTAG / REQUIRED_HASHTAG
TG_ALLOWED_CHAT_IDS / WP_ALLOWED_CHAT_IDS
TG_ALLOWED_CHANNEL_USERNAMES / WP_ALLOWED_CHANNELS
TG_REQUIRE_CHANNEL_POST / WP_REQUIRE_CHANNEL_POST
TG_MEDIA_RETRY_ATTEMPTS
TG_MEDIA_RETRY_BACKOFF_SECONDS
TG_MEDIA_GROUP_WAIT_TIMEOUT_SECONDS
```

Call Baxter also recognizes `TELEGRAM_API_BASE`, `TG_MEDIA_STORAGE_MODE`, `TG_MEDIA_BASE_DIR`, and `BRIDGE_DOWNLOAD_DIR` for the replacement projection/media paths.

## Generated compatibility state

By default, compatibility mode writes its generated runtime configuration beneath:

```
./data/call-baxter-tgex/
├── call-baxter-tgex.yml
├── call-baxter.db
├── rules.d/
├── media/
└── wp-publications.db
```

Override the directory with `CALL_BAXTER_TGEX_STATE_DIR`.

The generated YAML contains resolved credentials and is created with mode `0600`. Keep the state directory private and out of source control.

The Telegram projection database continues to use `DATABASE_URL`. TGEX compatibility currently supports SQLite database URLs and filesystem paths only. `sqlite+aiosqlite:///...` and `sqlite:///...` are converted to their underlying file path.

The projection audit tables are named `tg_route_executions` and `tg_handler_executions`. This avoids a schema collision with TGEX's different tables named `route_executions` and `handler_executions`. A schema-version-2 migration renames only older `tg-agent-cli` tables whose columns positively identify them; genuine TGEX tables remain untouched for historical import.

## Cutover procedure

### 1. Back up the running service

```
cp -a data data.pre-call-baxter
cp -a .env .env.pre-call-baxter
```

Also capture the active Telegram webhook before changing the process:

```
telegras2 webhook-info --output-format json > webhook.pre-call-baxter.json
```

### 2. Stop TGEX and install `v0.8.0`

Stop the old process without deleting its database or media directories, then install the four coordinated wheels.

### 3. Validate environment translation

Initialize the replacement stores and generated configuration:

```
telegras2 db-init
```

Run the deployment doctor before starting the replacement process:

```
telegras2 doctor
telegras2 doctor --online
```

The offline doctor validates local configuration without contacting Telegram. The online mode additionally calls `getMe` and `getWebhookInfo`.

Preview and import historical TGEX records:

```
telegras2 migrate-history --dry-run
telegras2 migrate-history
```

The importer is idempotent. It records source-to-target mappings in `tg_legacy_imports`, skips rows already migrated, links interactions already present by Telegram update ID, and imports media metadata without downloading historical files.

Inspect generated rules without contacting Telegram:

```
telegras2 routes
```

When WordPress is enabled, verify `WP_DRY_RUN=true` first. This exercises normalization, filtering, and post preparation without making WordPress requests.

### 4. Start in the existing mode

Webhook deployment:

```
MODE=webhook telegras2 start
```

Polling deployment:

```
MODE=polling telegras2 polling
```

The Call Baxter daemon now executes enabled schedules automatically, so the Telegram polling and media-group flush schedules do not require a separate cron or `/run-due` caller.

### 5. Verify old and native endpoints

```
curl --fail http://127.0.0.1:8000/healthz
curl --fail http://127.0.0.1:8000/v1/health
curl --fail \
  -H "Authorization: Bearer ${INTROSPECTION_TOKEN}" \
  http://127.0.0.1:8000/internal/introspection/interactions
```

The native operator UI is available at `/ui`. The legacy introspection UI path redirects there.

For an additional webhook authenticity layer, configure `TELEGRAM_WEBHOOK_HEADER_SECRET`. The compatibility endpoint then requires a matching `X-Telegram-Bot-Api-Secret-Token` header, and `telegras2 set-webhook` passes the same secret token to Telegram.

#### 6. Verify Telegram delivery

For webhook mode, send one test update through the existing webhook path and confirm it appears in both:

```
/internal/introspection/interactions
/v1/events
```

For polling mode, a finite smoke run is available:

```
telegras2 polling --stop-after-updates 1
```

#### 7. Enable WordPress writes

Set `WP_DRY_RUN=false`, restart the process, and publish a controlled Telegram message. Confirm:

- the expected post type and status;
- configured categories, tags, and author;
- media behavior according to `WP_USE_FEATURED_MEDIA`;
- edits update the existing post;
- duplicate deliveries do not create duplicate posts;
- media-group fragments produce one post after the configured wait period;
- a failed Telegram media download is recorded in `media_errors` while the text post still succeeds.

## Historical data migration

`telegras2 migrate-history` imports the TGEX tables below when their expected columns are detected:

```
telegram_interactions -> tg_updates, tg_chats, tg_users, tg_messages, ...
route_executions      -> tg_route_executions
handler_executions    -> tg_handler_executions
```

Original TGEX tables are never dropped or rewritten. Timestamps, statuses, errors, sources, matched routes, handler results, and durations are retained. The compatibility introspection routes merge imported audit history with new Call Baxter runtime activity.

Keep the pre-cutover backup even after a successful migration. The importer normalizes Telegram payloads into the new browse-oriented schema, so it is not a byte-for-byte database conversion.

## Introspection differences

The following paths are retained:

```
GET    /internal/introspection/config
GET    /internal/introspection/webhook/status
DELETE /internal/introspection/webhook
GET    /internal/introspection/ui
GET    /internal/introspection/interactions
GET    /internal/introspection/interactions/{id}
GET    /internal/introspection/route-executions
GET    /internal/introspection/route-executions/{id}
GET    /internal/introspection/handlers/stats
GET    /internal/introspection/chats
GET    /internal/introspection/chats/{chat_id}/history
```

Interaction and chat endpoints are backed by the Telegram projection database. Route execution and handler statistics endpoints combine imported TGEX audit rows with current Call Baxter events/actions. Imported executions retain their positive projection IDs; current runtime events use negative compatibility IDs to avoid collisions. TGEX-specific fields that were never persisted cannot be reconstructed.

## Custom plugin migration

A TGEX plugin configured through `PLUGINS` or `BOT_PLUGINS` cannot be loaded safely into Call Baxter because registration, route matching, dependency injection, and persistence hooks differ.

Port each custom plugin as one or both of:

1. a Call Baxter plugin that registers event, ingress, poller, or action definitions;
2. YAML rules that connect existing Telegram events to actions.

Do not remove the TGEX process permanently until every configured custom plugin has either been ported or explicitly retired.

## Rollback

1. Stop the Call Baxter compatibility process.
2. Restore the old environment and original TGEX executable/image.
3. Restore the pre-cutover database if the old application must see an exact filesystem snapshot.
4. Restore the prior webhook URL only when it changed.
5. Start TGEX and confirm `GET /healthz` plus one test update.

The generated `CALL_BAXTER_TGEX_STATE_DIR` can remain for diagnosis; TGEX does not use it.

## Move to native Call Baxter configuration

After compatibility mode is stable, migrate intentionally to native configuration:

- replace `telegras2 start` with `call-baxterd --config-file ...`;
- move generated plugin configuration into an operator-owned YAML file;
- move generated rules into the project rules directory;
- use `/v1/*`, `/ui`, and the `tg` CLI as the primary operator surfaces;
- remove legacy environment aliases and introspection clients only after consumers have moved.

This second step is optional. Compatibility mode is a supported deployment surface for the `v0.8.0` release.

# 3.12 Call Baxter deployment with Caddy and Docker Compose

## Scope

The `infra/` layout is the minimal Call Baxter deployment:

- `call-baxter` runtime;
- `tg-agent-cli` projection support;
- Caddy HTTPS ingress;
- no bundled WordPress plugin, MariaDB, or WordPress service.

Use the full `docker/` layout and **Deploy → Telegram to WordPress with Docker** for the complete bridge example.

## Goal

Keep Call Baxter off the public TCP surface and expose only a narrow Telegram webhook path through Caddy.

## Runtime layout

```
Telegram
  -> https://PUBLIC_DOMAIN${TELEGRAM_WEBHOOK_PATH}
  -> Caddy
  -> unix:///srv/run/cb.sock
  -> Call Baxter daemon
  -> Telegram plugin persistence + event emission
  -> tg-agent-cli projection database at /srv/data/tg.db
```

## Bound host paths

```
infra/run/cb.sock
infra/data/cb.db
infra/data/tg.db
rules.d/
infra/caddy_data/
infra/caddy_config/
```

## Why this split

- Telegram webhook delivery requires a publicly reachable HTTPS endpoint.
- Caddy terminates TLS and reverse-proxies only the configured webhook and health paths.
- Call Baxter listens on a shared Unix socket rather than a public host port.
- The proxy checks `X-Telegram-Bot-Api-Secret-Token` before forwarding a webhook payload.
- Polling and webhook ingestion both normalize through the same Telegram plugin event path.

## Configure the minimal stack

```
cd infra
cp .env.example .env
```

Edit `.env` and set:

```
TGCLI_BOT_TOKEN=123456:replace-me
PUBLIC_DOMAIN=bot.example.org
ACME_EMAIL=ops@example.org
TELEGRAM_WEBHOOK_PATH=/webhooks/telegram/main
TELEGRAM_WEBHOOK_SECRET=replace-with-long-random-token
TELEGRAM_POLL_ENABLED=false
```

Prepare writable bind mounts:

```
mkdir -p run data caddy_data caddy_config
sudo chown -R 10001:10001 run data caddy_data caddy_config
```

Validate and start:

```
docker compose config --quiet
docker compose up -d --build
docker compose ps
docker compose logs --tail=150 call-baxter caddy
```

## Register the Telegram webhook

Load the env file:

```
set -a
. ./.env
set +a
```

Register the public URL:

```
curl --fail-with-body --silent --show-error \
  --request POST \
  "https://api.telegram.org/bot${TGCLI_BOT_TOKEN}/setWebhook" \
  --data-urlencode "url=https://${PUBLIC_DOMAIN}${TELEGRAM_WEBHOOK_PATH}" \
  --data-urlencode "secret_token=${TELEGRAM_WEBHOOK_SECRET}"
```

Inspect the result:

```
curl --fail-with-body --silent --show-error \
  "https://api.telegram.org/bot${TGCLI_BOT_TOKEN}/getWebhookInfo"
```

## Operational checks

```
curl --fail --silent --show-error "https://${PUBLIC_DOMAIN}/healthz"

docker compose exec call-baxter \
  cb --config-file /srv/config/call-baxter.yml \
  --daemon-url http+unix:///srv/run/cb.sock \
  health

docker compose exec call-baxter \
  tg --database-url /srv/data/tg.db messages list
```

## Upstream references

- Telegram Bot API: https://core.telegram.org/bots/api
- Telegram webhook guide: https://core.telegram.org/bots/webhooks
- Caddy reverse proxy: https://caddyserver.com/docs/caddyfile/directives/reverse_proxy
- Docker Compose environment variables: https://docs.docker.com/compose/how-tos/environment-variables/

# 4. tg-agent-cli

## 4.1 tg-agent-cli

A browse-first Telegram bot operator scaffold centered on a local projection database and CLI.

The recommended interoperable split is:

- `call-baxter` handles webhook ingress, polling, scheduling, and generic rule execution
- `tg-agent-cli` owns the Telegram projection database and browse-first CLI

The design assumes Telegram updates are ephemeral and update-driven, so the main state surface is a local SQLite database that normalizes chats, users, messages, callback queries, media assets, reactions, poll schedules, poll runs, actions, and audits.

### Highlights

- browse-friendly SQLite schema with message/media/action/callback separation
- `pydantic-settings` based CLI with nested subcommands
- media persistence policy with `skip`, `file`, and `blob` storage modes
- poll scheduling and hook registry extension points
- direct Telegram Bot API adapter for live status and outbound actions
- generated Telegram API models kept at the API edge

### Quickstart

```
python -m venv .venv
. .venv/bin/activate
pip install -e .[dev]
pytest -q

TGCLI_BOT_TOKEN=123456:abc tg --database-url ./state.db status
```

### Operator examples

```
tg --database-url ./state.db chats list
tg --database-url ./state.db messages list --chat-ref chat:-1001234567890
tg --database-url ./state.db message show msg:-1001234567890/9811
tg --database-url ./state.db callbacks list
tg --database-url ./state.db media list --message-ref msg:-1001234567890/9811
TGCLI_BOT_TOKEN=123456:abc tg send text --chat-ref chat:-1001234567890 --text "hello there"
```

### Integration note

Live operator commands such as `tg send text` should use the direct Telegram Bot API path by default and persist their own action history into `tg.db`. Reactive or scheduled behavior belongs in `call-baxter` rules/actions.

See `docs/cli.md`, `docs/architecture.md`, `docs/schema.md`, and `docs/call-baxter-integration.md` for the scaffold contract.

# 4.2 Architecture

## Core stance

The local SQLite projection and CLI are the core of this package.

The optional daemon is still present for testing or local HTTP workflows, but it is no longer required in the preferred `call-baxter` integration model. In that model, Telegram updates can enter through `call-baxter`, and `tg-agent-cli` is used as the projection/index layer only.

## Layers

```
api/
  Telegram Bot API adapter and generated Telegram request/response models

ingest/
  update normalization, media extraction, and persistence policy

store/
  SQLite schema and small DB helpers

services/
  query services, outbound action services, polling service, hook registry

cli/
  pydantic-settings command tree

daemon/
  FastAPI application exposing the same service operations over HTTP
```

## Division between CLI and daemon

### CLI

The CLI is optimized for:

- local inspection of already ingested state
- shell automation
- ref-based outbound actions
- manual polling or schedule management
- reproducible examples of modern `pydantic-settings` CLI composition

### daemon

The daemon remains useful for:

- local debugging of projection services over HTTP
- temporary integration before `call-baxter` is introduced
- service-style deployments that still want an HTTP wrapper around the projection DB

But the package no longer requires a daemon to stay useful.

## Polling model

Polling is represented explicitly instead of being hidden in a worker script.

```
tg_poll_schedules
  desired cadence and filters

tg_poll_runs
  execution evidence

tg_poll_schedule_claims
  cross-process execution ownership with expiring leases

PollingService
  live getUpdates call + normalization + hook emission
```

The scheduler model is intentionally simple:

- schedules have `interval_seconds`, `timeout_seconds`, `limit`, `allowed_updates`, `hook_names`, and `source`
- `run_due_schedules()` is the primitive loop
- external supervisors can call that periodically
- each due run is claimed atomically in SQLite, preventing two daemon requests or processes from running the same schedule concurrently
- the active worker renews its claim while `getUpdates` and normalization run; an expired claim can neither be revived nor completed, and lease loss is recorded on the poll run
- the next execution is scheduled from actual completion time rather than batch-start time, avoiding immediate reruns after a long poll
- hook names and Telegram polling bounds are validated before schedule persistence or API execution

This keeps the backend fixed and testable while leaving deployment policy outside the core library.

## Projection ordering

Telegram update rows remain an appendable audit history, while each normalized message row represents the newest known message version. Message upserts compare `edit_date` and `date`; a delayed original-message retry cannot replace a newer edit or its entity/media projections.

## Atomic projection visibility

Ingestion is split into two phases:

1. Telegram file metadata and bytes are fetched before a SQLite write transaction starts.
2. The update, chat, users, message, entities, callbacks/reactions, and media rows commit together in one `BEGIN IMMEDIATE` transaction.

File-mode media is downloaded into a unique `.part` file under a deterministic identity directory. A pre-existing file is reused only when a committed projection row already references it; otherwise each worker prepares its own copy. The prepared file is atomically renamed during the projection transaction. If any later row write fails, SQLite rolls back and cleanup removes only the exact filesystem inode finalized by that worker, and only after confirming that no committed row references the path. After a successful edit, old unreferenced local media files are removed. Housekeeping failures are conservative and do not turn an already committed ingest into a false failure. This avoids holding a database write lock during network I/O while making projection rows atomically visible.

## HTTP client lifetime

The daemon lazily creates at most one live and one offline Telegram API client and closes them during application shutdown. The Call Baxter Telegram plugin owns and closes the temporary clients it creates for polling, actions, and projection work.

## Media policy

Media persistence is normalized into `tg_message_media` and controlled by `MediaPolicy`.

```
storage_mode = skip
  metadata only

storage_mode = file
  bytes written under media base dir; DB links to local_path

storage_mode = blob
  bytes stored in SQLite BLOB column
```

Default scaffold policy is `file`, because it keeps the DB browse-friendly while avoiding unbounded database growth for larger assets.

WAL mode is established during serialized schema initialization with bounded retries. Normal connections do not change journal mode, which avoids concurrent-startup lock races.

## Hooking point

The scaffold does not ship a full callback DSL yet. Instead it offers a smaller extension point:

```
HookRegistry.register(name, callable)
PollingService(..., hook_registry=registry)
```

That gives later services a stable place to subscribe to polled update batches without entangling the core browse model with arbitrary callback semantics.

## 4.3 CLI contract

The CLI uses `pydantic-settings` nested CLI parsing with `CliSubCommand`, `CliPositionalArg`, `cli_parse_args`, `cli_kebab_case`, and `cli_implicit_flags`.

### Global flags

```
--database-url <path|sqlite:///path>
--bot-token <token>
--api-base <url>
--output-format short|json|yaml|table
--media-storage-mode skip|file|blob
--media-base-dir <dir>
--poll-timeout-seconds <int>
--poll-limit <int>
--poll-allowed-updates message,edited_message,callback_query,...
```

### Browse commands

```
tg status [--live]

# webhook inspection / mutation

tg webhook show
tg webhook set --url <https-url> [--allowed-updates message,callback_query,...]
tg webhook delete [--drop-pending-updates]

# chats / messages / updates

tg chats list [--search TEXT] [--limit N] [--type private|group|supergroup|channel]
tg chats show <chat-ref>

tg messages list [--chat-ref <chat-ref>] [--search TEXT] [--media-kind KIND] [--limit N]
tg message show <msg-ref>

tg updates list [--limit N] [--kind KIND] [--source webhook|poll|inject] [--status normalized|failed|ignored]
tg updates show <upd-ref>

# callback queries / media / actions

tg callbacks list [--message-ref <msg-ref>] [--answered|--no-answered] [--limit N]
tg callbacks show <cbq-ref>

tg media list [--message-ref <msg-ref>] [--chat-ref <chat-ref>] [--media-kind document|photo|video|...] [--
download-status pending|downloaded|failed|skipped] [--limit N]
tg media show <media-ref>

tg actions list [--subject-ref <ref>] [--action-kind KIND] [--limit N]
tg actions show <act-ref>
```

### Ingest commands

```
tg ingest update --file <json-file> [--bot-ref bot:main] [--source inject]
```

Semantics:

- `ingest update` replays a raw Telegram update into the normalized SQLite store.
- If `--bot-token` is available and media policy is not `skip`, the ingester attempts to resolve and persist file payloads.
- Without a live API client, media rows are still normalized and marked `pending`.

## Action commands

```
tg send text --chat-ref <chat-ref> --text <text>
tg send photo --chat-ref <chat-ref> --file <path> [--caption TEXT]
tg send document --chat-ref <chat-ref> --file <path> [--caption TEXT]

tg reply <msg-ref> --text <text>
tg react <msg-ref> --emoji <emoji> [--big]
tg ack <cbq-ref> [--text TEXT] [--alert]
tg delete <msg-ref>
tg pin <msg-ref> [--disable-notification]
tg unpin <msg-ref>
```

Semantics:

- All outbound actions require `--bot-token` or `TGCLI_BOT_TOKEN`.
- Every action is recorded in `tg_actions`, including failures.
- `reply`, `react`, `delete`, `pin`, and `unpin` operate on local stable refs.

## Polling commands

```
tg poll once [--timeout-seconds N] [--limit N] [--allowed-updates CSV] [--source NAME] [--hook-names CSV]

tg poll schedules list
tg poll schedules add --name <name> --interval-seconds <n> [--timeout-seconds N] [--limit N] [--allowed-updates
CSV] [--hook-names CSV] [--source NAME] [--disabled]
tg poll schedules run-due

tg poll runs list [--schedule-name NAME] [--limit N]
```

Semantics:

- `poll once` calls `getUpdates`, normalizes each returned update, optionally downloads media, records a `tg_poll_runs` row, and emits named hooks.
- `poll schedules add` only persists scheduling metadata; it does not talk to Telegram and therefore works without a live token.
- `poll schedules run-due` is the primitive scheduler loop: it finds enabled schedules with `next_run_at <= now`, executes them, advances offsets, and updates `next_run_at`.
- Hooks are registered in code via `HookRegistry`. The scaffold ships with a `noop` hook and leaves room for daemon-local service integrations.

## Output modes

```
--output-format short|json|yaml|table
```

- `short` is meant for operator use and shell pipelines.
- `json` / `yaml` preserve the structured contract.
- `table` is best-effort and follows the same normalized data models.

## 4.4 Daemon HTTP surface

Base path: `/v1`

### Status and webhook

```
GET    /status?live=false
GET    /webhook
POST   /webhook
DELETE /webhook?drop_pending_updates=false
```

### Browse endpoints

```
GET /chats
GET /chats/{chat_ref}

GET /messages?chat_ref=&search=&media_kind=&limit=
GET /messages/{message_ref}

GET /updates?kind=&source=&status=&limit=
GET /updates/{update_ref}

GET /callbacks?message_ref=&answered=&limit=
GET /callbacks/{callback_ref}

GET /media?message_ref=&chat_ref=&media_kind=&download_status=&limit=
GET /media/{media_ref}

GET /actions?subject_ref=&action_kind=&limit=
GET /actions/{action_ref}
```

### Ingest

```
POST /updates/ingest
```

Body:

```
{
  "bot_ref": "bot:main",
  "source": "inject",
  "update": {"update_id": 1, "message": {...}}
}
```

### Outbound actions

```
POST /actions/send-text
POST /actions/send-photo
POST /actions/send-document
POST /actions/reply
POST /actions/react
POST /actions/ack
POST /actions/delete
POST /actions/pin
POST /actions/unpin
```

### Polling

```
POST /poll/once
GET  /poll/schedules
POST /poll/schedules
POST /poll/schedules/run-due
GET  /poll/runs
```

Notes:

- `POST /poll/schedules` is purely local DB state and does not require a live token.
- `POST /poll/once` and `POST /poll/schedules/run-due` require `bot_token`.
- `GET /status?live=true` also requires `bot_token`; it does not silently fall back to offline status.
- Poll intervals must be positive, timeout must be non-negative, and Telegram limits must be between 1 and 100.
- Browse endpoint `limit` values must be between 1 and 1000.
- When `CALL_BAXTER_INGRESS_TOKEN` is configured, bearer authentication is required on every `/v1/*` endpoint and the authentication scheme is parsed case-insensitively.
- `GET /healthz` returns a simple daemon health probe.

## 4.5 Schema notes

The schema is intentionally browse-oriented.

### Main tables

```
tg_updates
  raw incoming updates with source + status

tg_chats
  stable chat catalog

tg_users
  stable user catalog

tg_messages
  normalized message rows, including reply/thread metadata

tg_message_entities
  rich text / caption entities

tg_message_media
  normalized media assets and storage state

tg_callback_queries
  callback query catalog

tg_message_reactions
  reaction history

tg_actions
  outbound action audit log

tg_poll_schedules
  persisted poll cadence and filters

tg_poll_schedule_claims
  expiring owner tokens that serialize due schedule execution

tg_poll_runs
  execution records for manual or scheduled polling

tg_route_executions / tg_handler_executions
  higher-level automation audit surface

tg_legacy_imports
  idempotency map for imported TGEX history
```

The `tg_` prefix on the execution tables is intentional. TGEX used the names `route_executions` and `handler_executions` for structurally different tables; keeping the projection tables prefixed allows both schemas to coexist in a single SQLite database during migration.

Schema version 3 adds `tg_poll_schedule_claims`. Existing version-2 databases are upgraded in place without dropping schedules or history.

The claim expiry is renewed by the active polling worker. Renewal and completion both require the same owner token and a lease that has not yet expired; a delayed worker cannot revive or finalize ownership after another process is eligible to recover the schedule.

## Browse-friendly queries this supports

```
-- latest chats
SELECT ref, type, COALESCE(title, username, first_name) AS label, last_seen_at
FROM tg_chats
ORDER BY last_seen_at DESC
LIMIT 50;

-- latest messages in a chat
SELECT ref, date_ts, COALESCE(text, caption) AS preview
FROM tg_messages
WHERE chat_id = ?
ORDER BY date_ts DESC
LIMIT 100;

-- media attached to a message
SELECT ref, media_kind, storage_mode, download_status, local_path
FROM tg_message_media
WHERE message_ref = ?
ORDER BY ordinal ASC;

-- callback queries for a message
SELECT ref, data, answered, received_at
FROM tg_callback_queries
WHERE message_ref = ?
ORDER BY received_at DESC;

-- actions against a ref
SELECT ref, action_kind, ok, created_at, error
FROM tg_actions
WHERE subject_ref = ?
ORDER BY created_at DESC;
```

## Media representation

`tg_message_media` is intentionally separate from `tg_messages` so that:

- one message can expose multiple normalized assets
- media persistence policy can evolve independently of message browse behavior
- downloads can be retried or backfilled without rewriting message rows

All rows produced by one normalized update are committed in one transaction. Network media retrieval happens before that transaction; local file finalization and the corresponding media row participate in the rollback protocol. Readers therefore observe either the previous complete projection or the new complete projection, not an intermediate mix.

Key columns:

```
telegram_file_id / telegram_file_unique_id
file_name / mime_type / size_bytes
width / height / duration_seconds
storage_mode
download_status
local_path
content_blob
```

## 4.6 `tg-agent-cli` as a projection store for `call-baxter`

### Recommended role

`tg-agent-cli` can stay a standalone browse/index tool even when `call-baxter` is the component that receives Telegram updates.

That means:

- `call-baxter`
- webhook ingress
- `getUpdates` poller
- generic rule matching
- generic action execution
- `tg-agent-cli`
- projection database schema
- update normalization into chats/messages/callbacks/media/reactions
- browse-first CLI over that projection

### Interop seam

The seam is intentionally small and library-shaped:

- `tg_agent_cli.cb_bridge.build_cb_events(update)`
- `tg_agent_cli.cb_bridge.persist_update_to_projection(...)`
- `tg_agent_cli.cb_bridge.projection_max_update_id(...)`

So the Telegram plugin inside `call-baxter` can:

1. normalize raw updates into `call-baxter` events
2. persist selected updates into the Telegram projection DB
3. derive incremental poll offsets from the Telegram projection DB

without needing a Telegram-specific daemon to be resident.

### Practical flow

```
Telegram webhook -> call-baxter ingress -> normalized telegram.* events
                       -> rule telegram.persist_update -> tg projection db

tg --database-url ./tg-state.db messages list --chat-ref chat:-100...
```

### Why this split is useful

- operator browsing stays Telegram-specific and rich
- callback/runtime orchestration stays generic
- the Telegram projection DB can grow independently of the generic `call-baxter` event log
- `call-baxter` only pays the projection cost when a rule chooses to persist the update

# 5. cb-wordpress-plugin

## 5.1 cb-wordpress-plugin

A call-baxter WordPress bridge plugin for rules that act on Telegram events.

Main actions: - `wordpress.mirror_telegram_message` - `wordpress.flush_due_media_groups` - `wordpress.cleanup_orphan_media`

Lower-level actions: - `wordpress.create_post` - `wordpress.upload_media`

The plugin is designed for rules like:

```yaml
id: mirror-telegram-message-to-wordpress
enabled: true
match:
  source: telegram
  kind: telegram.message.new
actions:
  - kind: wordpress.mirror_telegram_message
    args:
      update: "{{ event.attrs.raw_update }}"
      required_hashtag: "#blog"
```

For Telegram media groups, the normal new-message mirror rule is also the staging path: - configure `state_db_path` and `media_group_wait_seconds` on the plugin; - route each `telegram.message.new` event through one `wordpress.mirror_telegram_message` action; - let the plugin buffer messages carrying `media_group_id` by `chat_id + media_group_id`; - flush due groups with `wordpress.flush_due_media_groups`; - create exactly one WordPress post from the assembled bundle.

Do not install a second broad new-message staging rule beside the mirror rule. Both would match the same update and invoke the action twice.

See `docs/architecture.md`, `docs/cb-integration.md`, and `docs/e2e-telegram-webhook-to-wordpress.md` for the responsibility split, rule shape, and complete Docker deployment path.

New in this build: - crash-recoverable idempotency state keyed by Telegram message ref and media-group key - deterministic WordPress slugs used to reconcile a remote post after a process crash - publication leases that suppress concurrent duplicate delivery across workers - duplicate deliveries are skipped once already mirrored or reconciled - Telegram edits update the existing WordPress post instead of creating a new one - hashtag-based post status routing (`status_by_hashtag`) - hashtag-based category routing (`category_ids_by_hashtag`) - WordPress tag syncing from hashtags (`sync_tags_from_hashtags`) - richer Telegram entity-aware HTML rendering for the mirrored body - a durable media-upload ledger with deterministic post reconciliation and safe cleanup of proven-unattached create media

### Loading in call-baxter

Explicit module loading:

```yaml
plugins:
  - module: cb_wordpress_plugin.plugin
    config:
      base_url: https://example.com
      username: wp-bot
      app_password: application-password-here
      state_db_path: ./state/wp-publications.db
      publication_lease_seconds: 30
      publication_wait_seconds: 2
      orphan_media_cleanup_after_seconds: 86400
      orphan_media_cleanup_limit: 50
```

`state_db_path` is required for crash recovery and cross-worker duplicate suppression. Without it, the plugin can still publish, but only WordPress's normal slug behavior protects against duplicate creates.

`wordpress.flush_due_media_groups` performs bounded orphan-media reconciliation using the configured age and limit. Operators can also invoke `wordpress.cleanup_orphan_media` directly, first with `dry_run: true`, to inspect or accelerate cleanup.

Entry-point discovery (after installing this package into the same environment as `call-baxter`):

```
discover_entry_points: true
entry_point_groups:
  - call_baxter.plugins
plugins:
  - call_baxter.plugins.system
  - call_baxter.plugins.telegram
```

When discovery is enabled, this package contributes the `wordpress` entry in `call_baxter.plugins`.

## Operator helper CLI

The package installs `cb-wp`, a small WordPress REST helper that reuses the same environment variables as the Docker deployment. Prefer environment variables so the application password does not appear in shell history or the process list:

```
export WP_BASE_URL=https://example.com
export WP_USERNAME=wp-bot
export WP_APP_PASSWORD='xxxx xxxx xxxx xxxx xxxx xxxx'

# Verify that the application password authenticates successfully.
cb-wp verify

# Enumerate category term IDs and names for category_ids_by_hashtag / WP_CATEGORIES.
cb-wp categories
# 1 : Uncategorized
# 12 : News
```

`cb-wp verify` calls the authenticated `wp/v2/users/me?context=edit` endpoint. The helper never prints the configured application password. `cb-wp categories` walks category results in deterministic integer-ID order and prints exactly `<int> : <string>` per line.

# 5.2 Architecture

## Role in the call-baxter world

This package is a **call-baxter runtime plugin**. It is not a PHP plugin installed inside WordPress.

The intended flow is:

```
Telegram webhook/poller -> call-baxter telegram plugin -> telegram.* events
                                    -> call-baxter rules
                                    -> wordpress.mirror_telegram_message
                                    -> wordpress.flush_due_media_groups
                                    -> wordpress.cleanup_orphan_media
                                    -> WordPress REST API
```

## Main actions

`wordpress.mirror_telegram_message` - accepts either a full Telegram update object or a single Telegram message object - parses Telegram text/caption and supported media - applies hashtag filtering - derives title + slug + body HTML - can render basic Telegram entities as HTML for the body ( `bold` , `italic` , `underline` , `strikethrough` , `code` , `pre` , `url` , `text_link` ) - uploads supported media to WordPress - creates or updates a WordPress post - when `media_group_id` is present and staging is configured, defers publication instead of posting immediately - can route post status and categories from hashtags - can sync WordPress tags from hashtags

`wordpress.flush_due_media_groups` - loads staged media-group fragments from the plugin state store - assembles all parts for a given `chat_id + media_group_id` - applies filters once to the combined publication text - creates one WordPress post for the whole group - reuses the same post on later flushes when a media-group key is already mapped - reconciles sufficiently old media from interrupted create attempts before flushing

`wordpress.cleanup_orphan_media` - inspects the durable publication-media ledger - looks up the deterministic post slug before deleting anything - marks tracked media attached when the post exists - force-deletes only media proven to belong to an interrupted create with no matching post - excludes attempts protected by a live publication lease, claims each candidate before remote work, and renews that claim during cleanup - supports age thresholds, bounded batches, lease duration controls, and dry-run reporting

## Routing and taxonomy policy

The plugin keeps publication policy local to the WordPress side: - `status_by_hashtag` picks a post status from matching hashtags - `category_ids_by_hashtag` attaches configured category ids from matching hashtags - `sync_tags_from_hashtags: true` resolves hashtags into WordPress tags through the REST API

Current tag sync behavior: - hashtags are normalized into slugs - existing tags are looked up by slug - missing tags are created - hashtags already consumed by `status_by_hashtag` are excluded from tag sync

## Responsibility split

### Telegram plugin responsibility

The **Telegram plugin** should own transport-native facts and reusable grouping semantics: - identifying update/message kinds - normalizing raw Telegram payloads - recognizing `media_group_id` - eventually, publishing higher-level aggregate events that are reusable across many sinks

This is the cleaner long-term home for media-group reassembly, because the problem is Telegram-specific rather than WordPress-specific.

### WordPress plugin responsibility

The **WordPress plugin** should own publication policy: - title/body/slug generation - hashtag filtering for publication - media upload strategy - taxonomy/status routing for WordPress posts - post creation and update policy - practical fallback buffering when a downstream publication target needs a whole bundle before it can post

That is why this plugin now includes a small staging/flush mechanism: it solves the immediate WordPress publishing problem without forcing the Telegram plugin to become aggregate-aware first.

## Message splits without `media_group_id`

This is a different class of problem.

If content is split across unrelated Telegram messages with no stable transport-level grouping key, the WordPress plugin should **not guess**. That belongs either in: - an explicit authoring convention, - a Telegram-side aggregation rule, - or a higher-level editorial workflow.

`media_group_id` is reliable enough to buffer automatically. Arbitrary multi-message essays are not.

## Idempotency and edits

The plugin keeps publication mappings and attempts in the same SQLite state DB used for staged media groups.

Keys: - `tg-message:<chat_id>:<message_id>` - `tg-media-group:<chat_id>:<media_group_id>`

Behavior: - repeated delivery of the same Telegram message is skipped once mirrored - `edited_message` / `edited_channel_post` updates patch the existing WordPress post - source versions combine Telegram `edit_date` / `date` with `update_id`, so older edits are skipped even when they arrive after a newer one - mapped-post updates use the same publication lease as creates, preventing concurrent edits from racing each other - media-group flush stores both the group key and all participating message keys

### Crash-recovery protocol

For each Telegram message or media group, the plugin derives a stable publication key and a deterministic WordPress slug containing a hash of that key.

1. A worker atomically claims a short SQLite lease in `publication_attempts` and renews it while Telegram downloads, WordPress uploads, and post requests are active. `publication_heartbeat_seconds` may shorten the renewal interval; it cannot exceed one third of the lease.
2. Other workers wait for the mapping or return `wordpress.deferred` while that lease is active.
3. The owner creates the WordPress post and records the returned post id as `remote_created` only while its lease is still live.
4. Message and media-group mappings are committed together and the attempt becomes `mapped`; completion is fenced by owner token and lease expiry.
5. If the process dies after WordPress accepted the post but before local completion, the expired-lease owner queries WordPress by the deterministic slug, recovers the post id, and repairs every mapping without another create.

The remote HTTP call is deliberately outside the SQLite transaction. SQLite protects local ownership and completion; the deterministic WordPress slug provides reconciliation across the system boundary.

Each successful WordPress media upload is written to `publication_media` before post creation continues. On recovery, the plugin first checks the durable `remote_created` state or deterministic post slug. `remote_created` cannot be stolen by an edited-message worker; it is finalized directly from the stored post ID. Existing posts retain their tracked media and repair local mappings; absent posts cause create-attempt media to be deleted before another upload is attempted. The normal media-group flush also runs bounded reconciliation for entries older than `orphan_media_cleanup_after_seconds` (default: 86,400 seconds, limit 50). Cleanup excludes active leases at query time and claims each candidate again before destructive work, closing the selection-to-deletion race.

`state_db_path` must be configured and stored on durable media for this protocol. Back up the state DB together with WordPress. The defaults are a 30-second publication lease and a two-second duplicate-worker wait; both can be overridden with `publication_lease_seconds` and `publication_wait_seconds`.

Dry-run media-group flushes never consume staged data. Failed or concurrently active publications also remain staged for a later flush.

Staged media-group parts are version-aware as well: a delayed original part cannot replace a newer edited part or extend its flush deadline. If recovery lookup against WordPress fails, the local lease is marked failed and released immediately for retry.

State-store schema initialization is serialized within a process and WAL activation retries lock contention from other processes. Normal store connections do not attempt to change journal mode, so concurrent first-use actions can safely share a newly created database.

There remains a very small external-system ambiguity if the process dies after WordPress returns an upload response but before the ledger insert commits. Update-attempt media is tracked for diagnosis but is not automatically deleted because an existing post may already reference it. These cases are visible in `publication_media` and should be reviewed before manual deletion.

## Recommended call-baxter shape

In the call-baxter runtime, this package should be treated as a sink-side publication plugin:

- Telegram plugins emit `telegram.*` events.
- call-baxter rules decide which events should result in publication work.
- This plugin exposes WordPress-oriented actions that operate on Telegram payloads through those rules.

That keeps the WordPress side publication-focused instead of turning it into a transport adapter.

## 5.3 call-baxter integration

This package is meant to live on the call-baxter side of the boundary.

- `call-baxter` receives and routes Telegram events.
- Telegram-specific source plugins emit `telegram.*` events.
- call-baxter rules choose when WordPress actions fire.
- This plugin does **not** fetch Telegram updates itself.

### Recommended rule shapes

#### Mirror a normal Telegram message

```yaml
id: mirror-telegram-message
enabled: true
match:
  source: telegram
  kind: telegram.message.new
actions:
  - kind: wordpress.mirror_telegram_message
    args:
      update: "{{ event.attrs.raw_update }}"
      required_hashtag: "#blog"
```

#### Update a post when Telegram edits the message

```yaml
id: mirror-telegram-edit
enabled: true
match:
  source: telegram
  kind: telegram.message.edited
actions:
  - kind: wordpress.mirror_telegram_message
    args:
      update: "{{ event.attrs.raw_update }}"
      required_hashtag: "#blog"
```

#### Stage media-group fragments with the normal mirror rule

Configure publication staging once on the plugin:

```yaml
plugins:
  - module: cb_wordpress_plugin.plugin
    config:
      state_db_path: ./state/wp-publications.db
      media_group_wait_seconds: 45
      publication_lease_seconds: 30
      publication_wait_seconds: 2
      orphan_media_cleanup_after_seconds: 86400
      orphan_media_cleanup_limit: 50
```

The same state database also protects normal one-message publications. Before creating a post, the plugin claims a short lease for the Telegram message or media-group key. Concurrent workers wait for the owner to complete. After a crash, the next worker searches WordPress by the deterministic idempotency slug and repairs the local mapping instead of creating another post.

Successful media uploads are recorded before post creation continues. If a create attempt is interrupted, recovery keeps the media when the deterministic post exists and deletes the tracked upload when no post exists. The age threshold prevents cleanup from racing a slow but still-active operation.

Then keep a single `telegram.message.new` mirror rule. The action detects `media_group_id` and stages grouped fragments automatically. Do not add a second broad new-message rule for media groups, because both rules would match the same update and execute the mirror action twice.

#### Flush due media groups from a call-baxter scheduler

```yaml
id: flush-telegram-media-groups
enabled: true
match:
  source: system
  kind: system.heartbeat
actions:
  - kind: wordpress.flush_due_media_groups
    args:
      state_db_path: ./state/wp-publications.db
```

A dry-run flush reports due groups without deleting them. A later non-dry-run flush can still publish the staged bundle.

The flush action also performs a bounded pass over old interrupted create attempts. For an operator-controlled pass, execute `wordpress.cleanup_orphan_media` with `dry_run: true` first, then repeat without dry-run after reviewing the candidate count.

## Responsibility split

The practical split is:

- Telegram plugin responsibility
- webhook / polling transport
- Telegram-native event detection
- stable `telegram.*` events
- optionally aggregate native transport concepts like `media_group_id`
- WordPress plugin responsibility
- publication policy
- title / slug / body extraction
- hashtag routing
- taxonomy syncing
- WordPress media / post creation
- temporary staging when publication needs to wait for a grouped bundle

#### Where should media-group reassembly live?

Long term, transport-native grouping belongs most cleanly in the Telegram plugin. That keeps the grouping reusable for non-WordPress sinks too.

But in the current architecture, the WordPress plugin still owns a valid **publication-side** staging layer:

- Telegram emits one event per update.
- call-baxter rules route each new-message event through one mirror action.
- WordPress plugin stages by `chat_id + media_group_id` when it needs the whole bundle before publication.

That is a sane short-to-medium-term split because the staging is narrowly tied to publication timing rather than Telegram transport itself.

#### What should *not* live here?

Arbitrary multi-message editorial reconstruction without a stable key should not be guessed here. If a publication spans several unrelated Telegram messages with no `media_group_id`, that should be solved upstream:

- explicit Telegram-side aggregation
- authoring convention
- or editorial tooling

## 5.4 End-to-end Telegram-to-WordPress flow

The repository ships a complete Docker Compose example under `docker/`.

Use the MkDocs page **Deploy → Telegram to WordPress with Docker** or open:

```
docker/TELEGRAM_WORDPRESS_TUTORIAL.md
```

That tutorial is the canonical hands-on deployment guide. It starts with Telegram long polling, brings up MariaDB and WordPress, creates a WordPress application password, verifies the REST API, sends a real `#blog` message, and then switches the same deployment to an HTTPS webhook.

### Why the Docker bundle is the canonical example

The repository has two container layouts:

| Layout | Scope |
|---|---|
| `docker/` | Call Baxter + Telegram projection + WordPress plugin + MariaDB + WordPress + optional Caddy |
| `infra/` | Call Baxter + Telegram projection + Caddy only |

The minimal `infra/` image does not install `cb-wordpress-plugin` and does not run WordPress. It is appropriate when WordPress is external or the bridge is not needed, but it cannot demonstrate the bundled end-to-end flow by itself.

### Runtime path

```
Telegram webhook or getUpdates poller
  -> call-baxter Telegram plugin
  -> optional tg-agent-cli projection in /srv/data/tg.db
  -> telegram.message.new / telegram.message.edited
  -> call-baxter rule match
  -> wordpress.mirror_telegram_message
  -> WordPress REST API
  -> publication mapping in /srv/data/wp-publications.db
```

### Minimal plugin configuration

```yaml
plugins:
  - call_baxter.plugins.system
  - module: call_baxter.plugins.telegram
    config:
      bot_token: ${TGCLI_BOT_TOKEN}
      projection_database_url: /srv/data/tg.db
      projection_media_base_dir: /srv/data/tg-media
  - module: cb_wordpress_plugin.plugin
    config:
      base_url: ${WP_BASE_URL}
      username: ${WP_USERNAME}
      app_password: ${WP_APP_PASSWORD}
      publish_status: ${WP_PUBLISH_STATUS:-draft}
      required_hashtag: ${WP_REQUIRED_HASHTAG:-}
      categories: ${WP_CATEGORIES:-}
      tags: ${WP_TAGS:-}
      state_db_path: /srv/data/wp-publications.db
      media_group_wait_seconds: 45

rules_dir: /srv/rules.d
```

Set `WP_REQUIRED_HASHTAG` to a hashtag such as `#blog` to require it. Leave it unset or set it to an empty value to accept messages without a required hashtag. Set `WP_CATEGORIES` or `WP_TAGS` to comma-separated WordPress term IDs such as `4,9` when you want fallback categories or tags independent of message hashtags.

## New-message rule

One rule handles both ordinary messages and Telegram media-group fragments:

```yaml
id: mirror-telegram-message
enabled: true
match:
  source: telegram
  kind: telegram.message.new
actions:
  - kind: wordpress.mirror_telegram_message
    args:
      update: "{{ event.attrs.raw_update }}"
```

The action reads `required_hashtag`, `state_db_path`, and `media_group_wait_seconds` from plugin configuration.

> **Do not duplicate the new-message rule for media groups**
>
> When `media_group_wait_seconds` is greater than zero, `wordpress.mirror_telegram_message` automatically stages messages carrying a Telegram `media_group_id`. A second broad `telegram.message.new` rule would execute the action twice for the same update.

## Edit rule

```yaml
id: mirror-telegram-edit
enabled: true
match:
  source: telegram
  kind: telegram.message.edited
actions:
  - kind: wordpress.mirror_telegram_message
    args:
      update: "{{ event.attrs.raw_update }}"
```

The publication mapping lets an edited Telegram message update its existing WordPress post.

## Media-group flush rule

```yaml
id: flush-media-groups
enabled: true
match:
  source: system
  kind: system.heartbeat
actions:
  - kind: wordpress.flush_due_media_groups
    args:
      state_db_path: /srv/data/wp-publications.db
```

A heartbeat schedule emits the system event that triggers this rule.

## First test message

```
#blog First Call Baxter post

This draft travelled from Telegram through Call Baxter into WordPress.
```

Expected behavior:

1. the required `#blog` hashtag passes publication policy;
2. the title parser removes leading hashtags and uses `First Call Baxter post`;
3. the remaining paragraph becomes the post body;
4. the default example creates a draft;
5. editing the original Telegram message updates that draft instead of creating another post.

## Verification layers

A complete deployment should be checked at four boundaries:

| Boundary | Check |
|---|---|
| Telegram transport | polling schedule or `getWebhookInfo` |
| Call Baxter | health, events, action results, loaded rules |
| Projection/mapping | `tg.db` and `wp-publications.db` |
| WordPress | authenticated REST request and resulting post |

The Docker tutorial provides exact commands for each check, plus webhook registration, backups, upgrades, and a troubleshooting decision tree.

# 6. cb-social-plugin

## 6.1 Social publishing plugins

`cb-social-plugin` packages three independent Call Baxter plugin entry points:

| Plugin | Native actions | Telegram mirror |
|---|---|---|
| `bluesky` | create/update/delete post, upload blob | `bluesky.mirror_telegram_message` |
| `mastodon` | create/update/delete post, upload media | `mastodon.mirror_telegram_message` |
| `x` | create/update/delete post, upload image | `x.mirror_telegram_message` |

Each provider also exposes `<provider>.flush_due_media_groups` for Telegram albums staged through `media_group_wait_seconds`.

Provider references:

- Bluesky
- Mastodon
- X

### Shared mirror contract

The mirror layer deliberately follows the existing WordPress operator shape without forcing the three remote protocols into identical semantics:

- `update` or `message` accepts Telegram raw update/message objects.
- `required_hashtag` optionally gates publication.
- `dry_run` renders the source decision without making a remote request.
- `state_db_path` persists `(provider, Telegram source)` mappings and source versions.
- `media_group_wait_seconds` stages Telegram albums before publication.
- `telegram_gateway_factory` or `telegram_bot_token` supplies Telegram file access for images.
- `media_required: true` turns otherwise reported media omissions into hard failures.

The initial media contract is image-only. This keeps the mirror deterministic while provider video and GIF APIs have materially different asynchronous/chunked processing models.

### Provider-specific behavior

#### Bluesky

Configuration accepts either `access_token` + `repo`, or `identifier` + `app_password`. The latter creates an AT Protocol session and uses the returned DID as the repository.

Mirrors derive a stable record key from the Telegram source key and publish with `putRecord`. Telegram edits therefore update the same AT Protocol record. URL facets are generated with UTF-8 byte offsets, and up to four downloaded images are embedded. The plugin currently enforces a 1,000,000-byte per-image mirror limit before upload.

#### Mastodon

Configuration requires `base_url` and a user `access_token`. Creates use `POST /api/v1/statuses` with a deterministic `Idempotency-Key`; Telegram edits use the mapped status ID with the status edit endpoint. Visibility, sensitivity, content warning, and language can be supplied in config or action arguments. Images upload through `/api/v2/media`.

#### X

Configuration requires a user `access_token`; `base_url` defaults to `https://api.x.com`. Posts use the v2 create/delete endpoints and images use the v2 media upload endpoint. Current X API post edits are represented as another `POST /2/tweets` request with `edit_options.previous_post_id`; the returned post ID becomes the current durable mapping.

`edit_policy: native` is the default. `ignore` retains the existing remote post, while `replace` explicitly deletes and recreates it. Native edits remain subject to X's own eligibility/window rules; replacement remains non-atomic.

## Idempotency boundary

Use `state_db_path` in production. Bluesky has an additional remote safety property because the mirror targets a deterministic record key. Mastodon sends an idempotency key on creates, but remote retention is bounded by the server implementation. X has no create idempotency key in this plugin, so a crash after remote creation but before local mapping persistence can still duplicate a post on retry.

## 6.2 Bluesky plugin

The `bluesky` entry point publishes through the AT Protocol PDS APIs and exposes both low-level post/blob actions and the shared Telegram mirror workflow.

### Configuration

Use either an already-issued access token plus repository DID, or an account identifier and an app password. `service_url` defaults to `https://bsky.social`; set it to the account's PDS when the account is hosted elsewhere.

```
plugins:
  - module: cb_social_plugin.bluesky
    config:
      identifier: "publisher.example"
      app_password: "${BLUESKY_APP_PASSWORD}"
      service_url: "https://bsky.social"
      state_db_path: "/srv/data/social-publications.db"
      media_group_wait_seconds: 2
```

For long-running deployments, keep credentials in the normal Call Baxter environment/secret boundary. Do not put app passwords or access tokens in committed rule files.

### Actions

| Action | Purpose |
|---|---|
| `bluesky.create_post` | Create an `app.bsky.feed.post` record. |
| `bluesky.update_post` | Upsert a post at a known record key with `putRecord`. |
| `bluesky.delete_post` | Delete a post record by record key. |
| `bluesky.upload_blob` | Upload raw bytes to the authenticated PDS. |
| `bluesky.mirror_telegram_message` | Mirror one Telegram message/update. |
| `bluesky.flush_due_media_groups` | Publish staged Telegram albums that are due. |

`bluesky.create_post` accepts `text` plus optional `created_at`, `langs`, `embed`, and `rkey`. `bluesky.update_post` requires `rkey` and `text` and accepts the same record fields.

### Telegram mirroring

The mirror derives a deterministic record key from the Telegram source key and writes with `com.atproto.repo.putRecord`. A Telegram edit therefore targets the same remote record rather than creating another post. When `state_db_path` is configured, Call Baxter also records the source version, remote URI/CID/rkey, and creation timestamp so stale or duplicate Telegram events can be suppressed locally.

```
actions:
  - kind: bluesky.mirror_telegram_message
    args:
      update: "{{ event.attrs.raw_update }}"
      required_hashtag: "#social"
```

URLs in mirrored text receive link facets with UTF-8 byte offsets. The original Telegram creation time is retained when an edit updates the record.

### Images

The alpha mirror supports Telegram photos and image documents. It embeds at most four images and rejects individual images larger than 1,000,000 bytes before upload, matching the Bluesky image embed limit. `media_required: true` makes a missing/failed image fatal; otherwise omissions are returned in `media_errors`.

Video, animation, alt-text extraction, mentions, replies, quotes, and external-card generation are not synthesized by this first mirror slice.

### Idempotency and failure boundary

The deterministic record key gives the mirror a stable remote target even if local publication state is lost. Native create/delete/blob actions remain classified as external mutations and are not considered generally replay-safe by Call Baxter.

Use `dry_run: true` to inspect the Telegram publication key, text, media count, and edit detection without authenticating to or mutating Bluesky.

## 6.3 Mastodon plugin

The `mastodon` entry point targets the standard Mastodon status and media APIs. It supports native status lifecycle actions and Telegram mirroring with durable source-to-status mappings.

### Configuration

Provide the instance base URL and a user access token with the write scopes required by that instance.

```
plugins:
  - module: cb_social_plugin.mastodon
    config:
      base_url: "https://social.example"
      access_token: "${MASTODON_ACCESS_TOKEN}"
      state_db_path: "/srv/data/social-publications.db"
      visibility: "unlisted"
      media_group_wait_seconds: 2
```

Optional defaults accepted by create/mirror actions include `visibility`, `sensitive`, `spoiler_text`, and `language`.

### Actions

| Action | Purpose |
| --- | --- |
| `mastodon.create_post` | Create a status with `POST /api/v1/statuses`. |
| `mastodon.update_post` | Edit a mapped status with `PUT /api/v1/statuses/:id`. |
| `mastodon.delete_post` | Delete a status. |
| `mastodon.upload_media` | Upload an attachment with `POST /api/v2/media`. |
| `mastodon.mirror_telegram_message` | Mirror one Telegram message/update. |
| `mastodon.flush_due_media_groups` | Publish staged Telegram albums that are due. |

Native create supports `media_ids`, visibility, sensitivity, content warning, language, and an optional caller-supplied `idempotency_key`. Native update supports the same editable status fields apart from visibility and create idempotency.

### Telegram mirroring

New Telegram messages receive a deterministic SHA-256-derived `Idempotency-Key` when Call Baxter creates the Mastodon status. When the Telegram source is edited, the durable mapping is used to update the same status ID.

```
actions:
  - kind: mastodon.mirror_telegram_message
    args:
      update: "{{ event.attrs.raw_update }}"
      required_hashtag: "#social"
```

Configure `state_db_path` in production. It is what lets the mirror distinguish an edit from a new publication and suppress duplicate/stale source versions across process restarts.

### Images

The alpha mirror uploads Telegram photos and image documents through `/api/v2/media`, then passes the returned attachment IDs to the status create/edit request. Mastodon's API can process larger media asynchronously; this mirror intentionally limits itself to images and does not attempt the video/GIF polling workflow yet.

If Telegram media cannot be acquired, the failure is reported in `media_errors` unless `media_required: true` is configured.

### Idempotency and failure boundary

Mastodon documents `Idempotency-Key` for status creation, and the mirror uses it for a stable Telegram source key. The server controls how long an idempotency key is retained, so the local `state_db_path` remains the durable duplicate-suppression authority for Call Baxter.

Native create/update/delete/media actions are still external side effects and are not exposed as generically replay-safe actions.

## 6.4 X plugin

The `x` entry point publishes through the current X API v2 post and media endpoints. It exposes native create/edit/delete/image-upload actions and the shared Telegram mirror workflow.

### Configuration

Use a **user-context** access token authorized to create and manage posts for the target account. `base_url` defaults to `https://api.x.com`.

```yaml
plugins:
  - module: cb_social_plugin.x_plugin
    config:
      access_token: "${X_ACCESS_TOKEN}"
      state_db_path: "/srv/data/social-publications.db"
      edit_policy: "native"
      media_group_wait_seconds: 2
```

Keep OAuth credentials outside committed configuration. App-only bearer tokens are not a substitute for the user authorization required by post mutations.

### Actions

| Action | Purpose |
|---|---|
| `x.create_post` | Create a post with `POST /2/tweets`. |
| `x.update_post` | Create the next eligible edit version using `edit_options.previous_post_id`. |
| `x.delete_post` | Delete an authored post with `DELETE /2/tweets/:id`. |
| `x.upload_media` | Upload an image with `POST /2/media/upload`. |
| `x.mirror_telegram_message` | Mirror one Telegram message/update. |
| `x.flush_due_media_groups` | Publish staged Telegram albums that are due. |

`x.update_post` requires `previous_post_id` and replacement `text`, with optional `media_ids`. X returns a new post ID for a successful edit; Call Baxter stores that ID as the new current remote mapping.

### Telegram edit policy

`edit_policy` controls what happens when a Telegram edit reaches an already-mapped X post:

- `native` (default) sends `edit_options.previous_post_id` and advances the local mapping to the new post ID returned by X.
- `ignore` leaves the remote post unchanged but records the newer Telegram source version, so the same edit is not retried forever.
- `replace` deletes the mapped post and creates a new one. This is non-atomic and is intended only as an explicit fallback strategy.

X decides whether a post remains edit-eligible. The platform documents bounded edit eligibility and post types that cannot be edited, so a `native` edit can be rejected even though the endpoint itself is supported. Call Baxter surfaces that HTTP failure rather than silently switching to a destructive replacement policy.

```yaml
actions:
  - kind: x.mirror_telegram_message
    args:
      update: "{{ event.attrs.raw_update }}"
      required_hashtag: "#social"
```

### Images

The alpha mirror uploads at most four Telegram photos/image documents, using the `tweet_image` media category, and references their media IDs in the post request. Video and animated-GIF media need the chunked upload/processing flow and are deliberately not emulated in this first slice.

### Idempotency and failure boundary

Unlike the Bluesky deterministic-record target and Mastodon's create idempotency header, this X plugin has no durable remote create-idempotency key. A crash after X accepts a create but before the local mapping commits can therefore duplicate a source on retry. Use `state_db_path` and monitor failed publications closely.

`replace` is also non-atomic: if deletion succeeds and the subsequent create fails, Call Baxter cannot restore the old post.

## 6.5 Call Baxter integration

### Plugin configuration

```yaml
plugins:
  - module: cb_social_plugin.bluesky
    config:
      identifier: "publisher.example"
      app_password: "${BLUESKY_APP_PASSWORD}"
      state_db_path: "/srv/data/social-publications.db"
      media_group_wait_seconds: 2

  - module: cb_social_plugin.mastodon
    config:
      base_url: "https://social.example"
      access_token: "${MASTODON_ACCESS_TOKEN}"
      state_db_path: "/srv/data/social-publications.db"
      visibility: "unlisted"

  - module: cb_social_plugin.x_plugin
    config:
      access_token: "${X_ACCESS_TOKEN}"
      state_db_path: "/srv/data/social-publications.db"
      edit_policy: "native"
```

Secrets above are placeholders; keep actual credentials in the normal Call Baxter secret/env boundary rather than committing them to configuration files.

### Mirror the same Telegram event to several providers

```yaml
id: mirror-telegram-social
enabled: true
match:
  source: telegram
  kind: telegram.message.new
actions:
  - kind: bluesky.mirror_telegram_message
    args:
      update: "{{ event.attrs.raw_update }}"
      required_hashtag: "#social"
  - kind: mastodon.mirror_telegram_message
    args:
      update: "{{ event.attrs.raw_update }}"
      required_hashtag: "#social"
  - kind: x.mirror_telegram_message
    args:
      update: "{{ event.attrs.raw_update }}"
      required_hashtag: "#social"
```

Add equivalent `telegram.message.edited` rules when edits should propagate. Bluesky and Mastodon update their existing mapped post. X follows its configured `edit_policy`; its native mode advances the mapping to the new post ID returned by the X edit request.

### Media groups

When `media_group_wait_seconds` is greater than zero, every mirror action stages Telegram messages with `media_group_id`. Schedule each configured provider's flush action against the same state DB:

```yaml
actions:
  - kind: bluesky.flush_due_media_groups
  - kind: mastodon.flush_due_media_groups
  - kind: x.flush_due_media_groups
```

Do not add a second broad album-staging mirror rule. As with the WordPress plugin, the ordinary mirror action itself is the staging path.

# 7. Contributing

## 7.1 Building the documentation

The documentation has three supported workflows: local preview, strict HTML validation, and combined PDF export.

### Reproducible toolchain

The docs dependencies are pinned in `tools/docs/uv.lock`. Commands use `uv run --project tools/docs --locked` so local and CI builds resolve the same MkDocs, Material, and PDF-renderer versions.

Do not replace the repository commands with an unpinned global MkDocs installation when validating a release.

### Local preview

```
uv run --project tools/docs --locked mkdocs serve
```

The `watch` configuration reloads changes in `docs/`, the package-local documentation directories, and the Docker tutorial sources.

### Strict HTML and search-index build

```
./scripts/build-docs.sh
```

Output:

```
site/
├── index.html
├── search/search_index.json
└── ...
```

The build fails for invalid navigation, broken internal links, unresolved snippets, or unrecognized links.

### Combined PDF build

```
./scripts/build-docs-pdf.sh
```

Output:

```
site/assets/call-baxter-documentation.pdf
```

The script performs two passes:

1. a strict HTML/link/search-index validation pass;
2. a PDF-enabled rendering pass followed by PDF header, size, and EOF integrity checks.

The passes are intentionally separate. WeasyPrint reports unsupported browser-only Material CSS rules while still producing a valid print document; enabling MkDocs strict mode during that rendering pass would promote those harmless renderer diagnostics to build failures.

### PDF styling

Print-specific rules live in:

```
tools/docs/pdf/styles.scss
```

The stylesheet defines A4 output, printable margins, local system fonts, wrapped code blocks, repeated table headers, and page-break avoidance for tables, figures, admonitions, and code blocks.

### CI artifacts

Both documentation workflows generate the PDF:

- `.github/workflows/ci.yml` uploads `call-baxter-documentation-pdf` as a workflow artifact;
- `.github/workflows/docs.yml` uploads the same standalone artifact and includes it in the GitHub Pages site.

## Workspace bridge commands

```
docs-site   strict HTML/search build
docs-pdf    strict HTML validation plus PDF export
docs-serve  local live preview
```