W3PATTERNS

BRS-Core Product Guide

Install, configure, operate, and extend the Business Reliability System runtime.

Version 1.0.0
Customer Edition
August 2026

Contents

1. Product overview
2. Requirements and responsibilities
3. Installation
4. Quick start
5. Running BRS-Core directly
6. Optional systemd production setup
7. Prometheus integration
8. Verifier architecture and contract
9. Included examples
10. Configuration
11. Endpoints
12. Prometheus metrics
13. Operations
14. Troubleshooting
15. Upgrades and rollback
16. Directory structure
17. Security and licensing reminders

1. Product overview

BRS-Core is a Business Reliability System runtime and Prometheus exporter. It turns independently useful business verifiers into continuously scheduled, isolated, observable workflows.

Each verifier is an ordinary CommonJS JavaScript program. It owns its dependencies, configuration validation, authentication, business assertions, cleanup, diagnostics, and direct command-line behavior. A verifier must run without BRS-Core, Prometheus, Grafana, systemd, or another BRS-Core component.

BRS-Core provides four focused runtime responsibilities:

1. Discover top-level verifier entry files.

2. Schedule and isolate their executions.

3. Store the latest result for each verifier in memory.

4. Export those results through Prometheus metrics and a read-only JSON API.

```
one verifier runs independently
    -> BRS-Core discovers many verifiers
    -> schedules and isolates them
    -> records their latest results
    -> exports them as Prometheus metrics
```

The runtime listens after verifier discovery, validation, and registration. Initial verifier executions start asynchronously after the HTTP server is available. Different verifiers may execute concurrently. The same verifier never overlaps itself. If an execution exceeds its configured timeout, BRS-Core terminates the isolated process group, stores a timeout result, and allows a future scheduled run after containment completes.

All runtime state is held in memory. Restarting BRS-Core clears stored results and starts a new initial execution of every registered verifier. Prometheus retains previously scraped history according to the customer's Prometheus configuration.

Product boundary

BRS-Core is intended to integrate with an organization's existing monitoring environment. It does not install or manage:

- Prometheus
- Grafana
- Alertmanager
- Blackbox Exporter
- Dashboards or alert rules
- Alert delivery
- HTTPS or TLS termination
- Host or infrastructure monitoring

2. Requirements and responsibilities

Core requirements

- Node.js 24 or newer
- A reviewed BRS-Core v1.0.0 source checkout or release archive
- Read access to the active verifier collection
- Network access required by each active verifier
- Network access from the intended Prometheus scraper when Prometheus

integration is used

The reference production installation uses Linux and systemd. systemd is optional when BRS-Core is run directly. Git is required for a source checkout; it is not required when deploying a prepared release archive.

BRS-Core does not require Docker or a local Prometheus installation. The root package currently has no third-party runtime dependency, although `npm ci` is recommended during deployment to verify the root lockfile.

Customer responsibilities

The customer determines which systems verifiers may access and supplies the appropriate accounts, permissions, credentials, certificates, routing, and firewall rules. Production verifier accounts should use the narrowest practical privileges.

Customers own verifier code, configuration, business logic, credentials, and data created by or for them. Verifier authors are responsible for safe diagnostic logging and must not log passwords, tokens, cookies, authorization headers, private keys, customer records, environment contents, or sensitive response bodies.

3. Installation

3.1 Verify and extract a release archive

Place `brs-core-v1.0.0.tar.gz` and `SHA256SUMS` in the same directory, then verify the archive before extraction.

Linux:

```
sha256sum --check SHA256SUMS
```

macOS:

```
shasum -a 256 --check SHA256SUMS
```

Extract and enter the release directory:

```
tar -xzf brs-core-v1.0.0.tar.gz
cd brs-core-v1.0.0
```

3.2 Verify Node.js and the release

```
node --version
npm ci
npm run check
npm test
```

Node.js must report version 24 or newer. The release test suite validates the runtime contract and packaged verifier examples without enabling those examples automatically.

3.3 Choose the active verifier directory

The packaged `verifiers/` directory is empty by default. BRS-Core discovers every top-level `.js` file in the configured directory, so only place approved active verifiers there.

For long-lived production deployments, a separately managed directory such as `/var/lib/brs-core/verifiers` keeps customer verifiers independent from replaceable Core release directories. Set `BRS_VERIFIERS_DIRECTORY` to the chosen absolute path.

Each verifier that owns external dependencies must have those dependencies installed from its own same-named implementation directory:

```
cd /path/to/verifiers/client-login
npm ci
```

Do not add verifier dependencies to the BRS-Core root package.

4. Quick start

The fastest evaluation path is to run the dependency-free skeleton directly. From the release root:

```
node examples/verifiers/skeleton.js
```

The command prints formatted JSON containing a summary and two workflow steps. It exits with status `0` when all steps pass and a nonzero status when the workflow fails or cannot execute. BRS-Core and Prometheus do not need to be running.

To see the same verifier under Core, activate a copy in the packaged active directory:

```
cp examples/verifiers/skeleton.js verifiers/skeleton.js
cp -R examples/verifiers/skeleton verifiers/skeleton
```

Start Core with explicit local evaluation settings:

```
export BRS_VERIFIERS_DIRECTORY="$PWD/verifiers"
export BRS_LISTEN_HOST=127.0.0.1
export BRS_LISTEN_PORT=9124
node exporter/exporter.js
```

In another terminal:

```
curl --fail http://127.0.0.1:9124/health
curl --fail http://127.0.0.1:9124/metrics
curl --fail http://127.0.0.1:9124/api/v1/metrics
```

Stop the interactive process with `Ctrl-C`.

5. Running BRS-Core directly

BRS-Core reads configuration from `process.env`. It does not automatically load a root `.env` file. You may export variables directly or explicitly load an environment file in the shell before starting Core.

To use the shipped template:

```
cp .env.example .env
```

Edit `.env`, then load it explicitly and start Core:

```
set -a
. ./.env
set +a
node exporter/exporter.js
```

Real `.env` files are ignored by the repository and must not be committed.

Startup sequence

BRS-Core performs these actions in order:

1. Read and validate runtime environment variables.

2. Discover direct `.js` files in `BRS_VERIFIERS_DIRECTORY`.

3. Validate verifier IDs, labels, schedules, and exported functions.

4. Register verifiers with an initial `pending` state.

5. Start the HTTP server and scheduler.

6. Launch initial verifier executions asynchronously.

One invalid verifier prevents startup. This fail-fast behavior avoids silently operating with an incomplete verifier collection.

Interactive diagnostics

Core lifecycle messages, verifier output, completed-run summaries, contract errors, and timeout errors appear in the terminal. The runtime uses ordinary Node.js stdout and stderr and does not require a BRS-specific logging service.

6. Optional systemd production setup

The following reference setup is for a standalone Linux host. Adapt ownership, paths, hardening, and configuration management to organizational standards.

6.1 Create the service account and directories

```
sudo useradd \
  --system \
  --user-group \
  --home-dir /var/lib/brs-core \
  --create-home \
  --shell /usr/sbin/nologin \
  brs-core

sudo install -d -o brs-core -g brs-core -m 0755 /opt/brs-core
sudo install -d -o root -g brs-core -m 0750 /etc/brs-core
```

Deploy the reviewed release contents into `/opt/brs-core`, then run:

```
cd /opt/brs-core
sudo -u brs-core npm ci
sudo -u brs-core npm run check
sudo -u brs-core npm test
```

6.2 Install runtime configuration

```
sudo install \
  -o root \
  -g brs-core \
  -m 0640 \
  /opt/brs-core/.env.example \
  /etc/brs-core/brs-core.env

sudoedit /etc/brs-core/brs-core.env
```

The reference unit reads `/etc/brs-core/brs-core.env` through systemd. Core itself does not know or depend on that file path.

6.3 Install verifier dependencies

Deploy each active verifier as a top-level entry file plus its same-named implementation directory. Install dependencies as the service account:

```
cd /var/lib/brs-core/verifiers/client-login
sudo -u brs-core npm ci
```

Repeat for each dependency-owning verifier. Set `BRS_VERIFIERS_DIRECTORY=/var/lib/brs-core/verifiers` when using this durable collection.

6.4 Install and start the service

Confirm the approved Node.js path:

```
command -v node
node --version
```

The reference unit uses `/usr/bin/node`. If Node.js is installed elsewhere, change `ExecStart` to the approved absolute path before installation.

```
sudo install \
  -o root \
  -g root \
```

```
  -m 0644 \
  /opt/brs-core/deploy/systemd/brs-core.service \
  /etc/systemd/system/brs-core.service

sudo systemctl daemon-reload
sudo systemctl enable --now brs-core.service
sudo systemctl status brs-core.service --no-pager
```

The reference service runs as `brs-core`, restarts on failure after five seconds, uses `SIGTERM` for shutdown, and permits 90 seconds for service stop.

6.5 View service logs

```
journalctl -u brs-core.service -n 100 --no-pager
journalctl -u brs-core.service -f
```

To filter a verifier using its stable ID:

```
journalctl -u brs-core.service --since "900 seconds ago" \
  | grep 'verifier=customer_checkout'
```

7. Prometheus integration

BRS-Core exposes Prometheus text exposition at `GET /metrics`. Adapt the shipped example and merge it into the organization's existing Prometheus configuration:

```
scrape_configs:
  - job_name: brs-core
    scrape_interval: 15s
    static_configs:
      - targets:
          - brs-core.example.internal:9124
```

BRS-Core does not modify or reload Prometheus. After applying the Prometheus configuration through the organization's normal process, verify the target:

```
up{job="brs-core"}
```

The expected value is `1`.

Listen-address and firewall decision

The conservative default is `BRS_LISTEN_HOST=127.0.0.1`. Use it when Prometheus runs on the same host or reaches Core through an approved local proxy.

When Prometheus runs on another host, bind to the host's private interface or to `0.0.0.0` according to local standards. Restrict TCP 9124 to authorized Prometheus or proxy source addresses using host and network firewalls.

`/metrics` is intentionally unauthenticated. The JSON API Basic Authentication setting does not protect `/metrics` or `/health`. Do not expose port 9124 directly to the public internet.

8. Verifier architecture and contract

8.1 Packaging convention

One verifier consists of a top-level executable entry file and a same-named implementation directory:

```
client-login.js
client-login/
    client-login.js
    .env.example
    package.json
    package-lock.json
    node_modules/
```

The top-level file is both the standalone command and Core's discovery target. The implementation directory owns source, configuration documentation, dependency manifests, lockfiles, and installed dependencies.

Discovery is flat and not recursive. Core loads only `.js` files directly inside `BRS_VERIFIERS_DIRECTORY`. It does not watch the directory. Restart Core after adding or changing a verifier.

8.2 Canonical module contract

An implementation exports this shape:

```
module.exports = {
    id: 'customer_checkout',
    label: 'Customer Checkout',

    schedule: {
        interval: 300,
        timeout: 60
    },

    async verify() {
        return {
            summary: 'The checkout workflow completed successfully.',
            steps: [
                {
                    name: 'place_order',
                    label: 'Place Order',
                    success: true,
                    durationSeconds: 0.42,
                    message: 'The order was accepted.',
                    checkedAt: new Date().toISOString()
                }
            ],
            observations: [
                {
                    name: 'items_processed',
                    value: 3,
                    labels: {
                        environment: 'test'
                    }
                }
            ]
        };
    }
};
```

Required verifier fields:

Field	Requirement
`id`	Lower snake case, starts with a letter, unique in the active directory

Field	Requirement
`label`	Non-empty stable human-readable label
`schedule.interval`	Optional positive integer seconds; defaults to Core configuration
`schedule.timeout`	Optional positive integer seconds; defaults to Core configuration
`verify()`	Required asynchronous function

Result requirements:

- `summary`: required non-empty stable string
- `steps`: required non-empty array
- `observations`: optional array; defaults to empty

An empty workflow is a contract error, not a pass. A malformed result or uncaught exception produces status `error`. A timeout produces status `timeout`.

Required step fields:

Field	Requirement
`name`	Lower snake case and unique within the result
`label`	Stable, non-empty human-readable label
`success`	JavaScript boolean
`durationSeconds`	Finite non-negative number
`message`	Stable, non-empty business-facing message
`checkedAt`	Valid ISO-8601 timestamp

Use stable identifiers, labels, and messages. Request IDs, timestamps, filenames, customer values, exception text, and other changing data do not belong in step messages or Prometheus labels.

8.3 Observations

Observations are optional numeric gauges:

- `name` uses lower snake case and becomes `brs_observation_<name>`.
- `value` is a finite number.
- `labels` may contain strings, booleans, or finite numbers.
- Core adds the `verifier` label; a verifier cannot override it.
- Two observations cannot produce the same name and label set in one result.
- Labels must remain bounded and low-cardinality.

Never use request IDs, filenames, email addresses, timestamps, database keys, or other unbounded values as observation labels.

8.4 Standalone execution

The entry file exports the verifier object and invokes the same `verify()` function when run directly. Standalone execution prints a useful structured result and returns a meaningful process exit status. This is the recommended first path for verifier development and diagnosis.

8.5 Cleanup and isolation

Verifiers own browser sessions, database clients, network connections, temporary artifacts, and cleanup. Use `finally` blocks and bounded client timeouts.

Core runs each verifier in an isolated child process. Hard termination at the Core timeout cannot run verifier `finally` blocks or undo external side effects, so verifiers must be safe to retry and should avoid orphaning work outside their process group.

9. Included examples

The release includes one dependency-free skeleton and five working NYMG reference examples under `examples/verifiers/`. They remain outside the active `verifiers/` directory and are never enabled automatically.

Example	Capability	Verifier-specific dependency
`skeleton`	Contract, standalone execution, and local configuration pattern	None
`client-login`	Playwright login, authenticated dashboard, and logout workflow	`playwright` 1.62.1 plus a separately installed Chromium build
`public-api`	Read-only HTTP/JSON Customer Care verification	None; uses Node.js built-in `fetch()`
`database-read`	TLS MySQL query against a restricted reference view	`mysql2` 3.23.2
`sftp`	Restricted read-only SFTP file and CSV-header verification	`ssh2-sftp-client` 12.1.1
`daily-claims-report`	HTTP, MySQL, SFTP, CSV, freshness, totals, and reconciliation	`mysql2` 3.23.2 and `ssh2-sftp-client` 12.1.1

9.1 Skeleton

Run from the release root:

```
node examples/verifiers/skeleton.js
```

No dependency installation or configuration is required. Its optional `SKELETON_EXAMPLE_ENABLED` setting demonstrates verifier-local `.env` precedence without preventing immediate use.

9.2 NYMG client login

From `examples/verifiers`:

```
cp client-login/.env.example client-login/.env
cd client-login
npm ci
npx playwright install chromium
cd ..
node client-login.js
```

The verifier opens the policyholder login, authenticates the synthetic demo customer, confirms the authenticated policy experience, and signs out. Its four stable steps cover opening login, authenticating, verifying the authenticated experience, and ending the session.

9.3 NYMG public API

```
cp public-api/.env.example public-api/.env
node public-api.js
```

No `npm install` is required. The verifier checks service reachability, HTTP success, valid JSON, and the expected published Customer Care information.

9.4 NYMG database read

```
cp database-read/.env.example database-read/.env
cd database-read
npm ci
cd ..
node database-read.js
```

The example reads only `nymg.brs_reference_status` using a constrained public demo account. It verifies connection, result shape, application identity, `READY` status, and a valid non-negative policy count. Its public CA certificate is included, and TLS certificate verification is enabled by default.

9.5 NYMG SFTP

```
cp sftp/.env.example sftp/.env
cd sftp
npm ci
cd ..
node sftp.js
```

The example account is SFTP-only, chrooted, read-only, and unable to obtain a normal shell, upload, or delete. The verifier checks connection, file existence, non-empty regular-file metadata, download capability, and the exact CSV header. The included public demo key must never be reused for another system or account.

9.6 NYMG Daily Claims Report

This flagship reference verifier requires a one-time database prerequisite. A NYMG database administrator must review and install the two restricted views and grants in `examples/verifiers/daily-claims-report/database-setup.sql`. From the implementation directory, the documented pattern is:

```
mysql --host=<nymg-database-host> \
  --port=25060 \
  --user=<nymg-database-administrator> \
  --password \
  --ssl-mode=VERIFY_CA \
  --ssl-ca=certificates/ca-certificate.crt \
  < database-setup.sql
```

Review the SQL before execution and adjust the account host in the two `GRANT` statements if required by the deployment.

Then run from `examples/verifiers`:

```
cp daily-claims-report/.env.example daily-claims-report/.env
cd daily-claims-report
npm ci
cd ..
node daily-claims-report.js
```

The eleven-step workflow validates configuration, status headers, a restricted database connection and latest-run view, restricted SFTP access, file metadata, freshness, CSV structure, row count, exact decimal claim totals, and cross-system consistency.

The reference schedule is midnight in `America/New_York`. Before midnight plus the configured 900-second grace period, yesterday's completed report remains acceptable. At and after the deadline, today's report is required. The database completion time and SFTP modification time must also be within the configured clock tolerance.

10. Configuration

10.1 Core runtime variables

Variable	Default	Purpose
`BRS_VERIFIERS_DIRECTORY`	`/opt/brs-core/verifiers`	Directory containing top-level verifier entry files
`BRS_LISTEN_HOST`	`127.0.0.1`	HTTP listen address
`BRS_LISTEN_PORT`	`9124`	HTTP listen port
`BRS_SCHEDULER_TICK_SECONDS`	`1`	Scheduler polling interval; positive and may be fractional
`BRS_DEFAULT_INTERVAL_SECONDS`	`300`	Default verifier interval when omitted
`BRS_DEFAULT_TIMEOUT_SECONDS`	`30`	Default verifier timeout when omitted
`BRS_API_AUTH_ENABLED`	`false`	Require Basic Authentication for the JSON API only
`BRS_API_USERNAME`	Empty	JSON API username
`BRS_API_PASSWORD`	Empty	JSON API password

The default interval and timeout must be positive integer seconds. Boolean settings accept only `true` or `false`, case-insensitively.

All externally visible BRS-Core and verifier time values use seconds. When an underlying library expects milliseconds, Core or the verifier converts at the dependency boundary.

10.2 JSON API authentication

Authentication applies only to `GET /api/v1/metrics`:

```
BRS_API_AUTH_ENABLED=true
BRS_API_USERNAME=integration-user
BRS_API_PASSWORD=replace-with-a-secret
```

With authentication enabled:

- Correct credentials return HTTP 200.
- Missing or incorrect credentials return HTTP 401.
- A blank configured username or password returns HTTP 503.
- `/metrics` and `/health` remain unauthenticated.

Basic Authentication does not provide transport encryption. Use it only on a trusted encrypted network or behind an organizational HTTPS reverse proxy.

10.3 Verifier-local environment files

Each implementation directory should include a committed `.env.example`. A developer may copy it to an ignored `.env` for standalone use. The precedence rule is exact:

> An existing value in `process.env` always wins. The verifier-local `.env` supplies only missing values.

Core does not search for or load verifier `.env` files. It inherits its execution environment and passes that environment to isolated verifier processes. The verifier may load only its own adjacent `.env` before reading its configuration.

10.4 Example configuration summary

Client login

Required variables:

- `CLIENT_LOGIN_URL`
- `CLIENT_LOGIN_USERNAME`
- `CLIENT_LOGIN_PASSWORD`
- `CLIENT_LOGIN_SUCCESS_SELECTOR`

Optional variables and defaults:

- `CLIENT_LOGIN_USERNAME_SELECTOR=#email`
- `CLIENT_LOGIN_PASSWORD_SELECTOR=#password`
- `CLIENT_LOGIN_SUBMIT_SELECTOR=button[type="submit"]`
- `CLIENT_LOGIN_SIGN_OUT_SELECTOR=form[action="/logout"] button[type="submit"]`
- `CLIENT_LOGIN_ACTION_TIMEOUT_SECONDS=10`
- `CLIENT_LOGIN_NAVIGATION_TIMEOUT_SECONDS=30`
- `CLIENT_LOGIN_HEADLESS=true`

Public API

- Required: `PUBLIC_API_URL`
- Optional: `PUBLIC_API_TIMEOUT_SECONDS=10`

Database read

Required variables:

- `DATABASE_READ_HOST`
- `DATABASE_READ_PORT`
- `DATABASE_READ_DATABASE`
- `DATABASE_READ_USER`
- `DATABASE_READ_PASSWORD`

Optional variables and defaults:

- `DATABASE_READ_CONNECT_TIMEOUT_SECONDS=10`
- `DATABASE_READ_TLS_ENABLED=true`
- `DATABASE_READ_TLS_CA_FILE=ca-certificate.crt`
- `DATABASE_READ_TLS_REJECT_UNAUTHORIZED=true`

SFTP

Required variables:

- `NYMG_SFTP_HOST`
- `NYMG_SFTP_PORT`
- `NYMG_SFTP_USER`
- `NYMG_SFTP_PRIVATE_KEY_FILE`

- NYMG_SFTP_DIRECTORY
- NYMG_SFTP_TEST_FILE

Optional: NYMG_SFTP_CONNECT_TIMEOUT_SECONDS=10.

Daily Claims Report

Required variables:

- DAILY_CLAIMS_REPORT_STATUS_URL
- DAILY_CLAIMS_REPORT_DATABASE_HOST
- DAILY_CLAIMS_REPORT_DATABASE_PORT
- DAILY_CLAIMS_REPORT_DATABASE_NAME
- DAILY_CLAIMS_REPORT_DATABASE_USER
- DAILY_CLAIMS_REPORT_DATABASE_PASSWORD
- DAILY_CLAIMS_REPORT_SFTP_HOST
- DAILY_CLAIMS_REPORT_SFTP_PORT
- DAILY_CLAIMS_REPORT_SFTP_USER
- DAILY_CLAIMS_REPORT_SFTP_PRIVATE_KEY_FILE
- DAILY_CLAIMS_REPORT_SFTP_DIRECTORY
- DAILY_CLAIMS_REPORT_SFTP_FILE
- DAILY_CLAIMS_REPORT_EXPECTED_OUTPUT_FILE
- DAILY_CLAIMS_REPORT_TIME_ZONE

Optional variables and defaults:

- DAILY_CLAIMS_REPORT_HTTP_TIMEOUT_SECONDS=10
- DAILY_CLAIMS_REPORT_DATABASE_CONNECT_TIMEOUT_SECONDS=10
- DAILY_CLAIMS_REPORT_DATABASE_TLS_ENABLED=true
- DAILY_CLAIMS_REPORT_DATABASE_TLS_CA_FILE=certificates/ca-certificate.crt
- DAILY_CLAIMS_REPORT_DATABASE_TLS_REJECT_UNAUTHORIZED=true
- DAILY_CLAIMS_REPORT_SFTP_CONNECT_TIMEOUT_SECONDS=10
- DAILY_CLAIMS_REPORT_GRACE_SECONDS=900
- DAILY_CLAIMS_REPORT_CLOCK_TOLERANCE_SECONDS=300

Relative CA and private-key paths in the included examples resolve from the verifier's implementation directory, not the caller's working directory.

11. Endpoints

BRS-Core exposes three HTTP GET endpoints on the configured host and port.

Endpoint	Format	Authentication	Purpose
`GET /health`	JSON	None	Core liveness and verifier runtime-state summary
`GET /metrics`	Prometheus text	None	Stable machine-readable metrics for scraping
`GET /api/v1/metrics`	JSON	Optional Basic Authentication	Read-only latest in-memory verifier results

Unknown paths return HTTP 404.

11.1 Health endpoint

```
curl --fail http://127.0.0.1:9124/health
```

The response includes:

- Core status: `ok` or `stopping`
- Registered verifier count
- Running verifier count
- For each verifier: ID, running state, interval, timeout, latest start and completion, next run, and status

Verifier statuses are `pending`, `pass`, `fail`, `error`, or `timeout`. Business failures do not make `/health` return a non-200 response. Alert on BRS metrics rather than treating HTTP health as workflow success.

11.2 Prometheus endpoint

```
curl --fail http://127.0.0.1:9124/metrics
```

The response uses Prometheus text exposition format. Messages and technical errors are deliberately absent from metric labels.

11.3 Version 1 JSON API

```
curl --fail http://127.0.0.1:9124/api/v1/metrics
```

With authentication enabled:

```
curl --user integration-user \
  http://127.0.0.1:9124/api/v1/metrics
```

The top-level JSON fields are `generated_at` and `verifiers`. Each verifier contains:

- `id`
- `label`
- `status`
- `summary`
- `started_at`
- `completed_at`
- `duration_seconds`

- `steps`
- `observations`

Each step contains `id`, `label`, `status`, `duration_seconds`, `message`, and `checked_at`. Each observation contains `name`, `value`, and `labels`.

A newly registered verifier is `pending` with null result and timing fields and empty step and observation arrays. Technical errors and timeouts have no synthetic steps or observations.

12. Prometheus metrics

All metric names use the `brs_` namespace.

12.1 Verifier metrics

Metric	Labels	Meaning
`brs_verifier_info`	`verifier`, `verifier_label`	Stable verifier metadata
`brs_verifier_running`	`verifier`	`1` while execution is running
`brs_verifier_last_run_outcome`	`verifier`, `outcome`	Latest outcome: `executed`, `error`, or `timeout`
`brs_verifier_last_run_duration_seconds`	`verifier`	Overall duration of the stored run
`brs_verifier_interval_seconds`	`verifier`	Configured schedule interval
`brs_verifier_timeout_seconds`	`verifier`	Configured execution timeout
`brs_verifier_last_run_started_timestamp_seconds`	`verifier`	Actual latest run start time
`brs_verifier_last_run_completed_timestamp_seconds`	`verifier`	Time latest results were stored
`brs_verifier_next_run_timestamp_seconds`	`verifier`	Next scheduled start
`brs_workflow_success`	`verifier`	`1` for pass; `0` for fail, error, or timeout

12.2 Step metrics

Metric	Labels	Meaning
`brs_workflow_step_info`	`verifier`, `step`, `verifier_label`, `step_label`	Stable step metadata
`brs_workflow_step_success`	`verifier`, `step`	`1` for pass and `0` for fail
`brs_workflow_step_duration_seconds`	`verifier`, `step`	Verifier-reported step duration
`brs_workflow_step_checked_timestamp_seconds`	`verifier`, `step`	Verifier-reported step completion time
`brs_workflow_step_run_completed_timestamp_seconds`	`verifier`, `step`	Stored run completion shared by its steps

Step metrics exist only for an executed verifier result. Technical errors and timeouts use verifier-level metrics and `brs_workflow_success=0`.

Before the first run completes, Core emits registration, running, interval, timeout, and next-run metrics. Last-run, workflow, step, and observation metrics appear only after the corresponding result is stored.

12.3 Observation metrics

Valid observations become gauges named:

```
brs_observation_<observation_name>
```

Every observation includes `verifier` plus its validated bounded labels.

The Daily Claims Report may expose these observations as progress permits:

- `brs_observation_daily_claims_report_file_size_bytes`
- `brs_observation_daily_claims_report_age_seconds`
- `brs_observation_daily_claims_report_file_rows`
- `brs_observation_daily_claims_report_database_rows`
- `brs_observation_daily_claims_report_file_claim_total`
- `brs_observation_daily_claims_report_database_claim_total`

12.4 PromQL examples

Overall workflow failures:

```
brs_workflow_success == 0
```

Failed steps:

```
brs_workflow_step_success == 0
```

Seconds since each verifier last completed:

```
time() - brs_verifier_last_run_completed_timestamp_seconds
```

Latest execution duration:

```
brs_verifier_last_run_duration_seconds
```

Results older than twice the configured interval:

```
time() - brs_verifier_last_run_completed_timestamp_seconds
  > 2 * brs_verifier_interval_seconds
```

These are integration examples. BRS-Core does not install dashboards or alert rules.

13. Operations

13.1 Scheduling behavior

Verifier intervals are measured from each run's start. Missed runs are not queued. Different verifiers may run concurrently, but a verifier never overlaps itself.

At timeout, Core terminates the isolated process group and waits for it to close before clearing the running state. A later scheduled execution may then run. Other verifiers continue independently.

13.2 Operational information surfaces

Surface	Intended use
`/health`	Is Core alive, and what are the current verifier runtime states?
`/metrics`	Stable bounded values for Prometheus, Grafana, and alerting
stdout/stderr/journald	Variable human-readable exceptions and diagnostic context

Do not place arbitrary error text, response bodies, or debugging context in Prometheus labels. Use the log stream for changing details.

13.3 Configuration and verifier changes

The active directory is not watched. After changing Core environment values, adding a verifier, or changing verifier code, restart the service:

```
sudo systemctl restart brs-core.service
sudo systemctl status brs-core.service --no-pager
journalctl -u brs-core.service -n 100 --no-pager
```

13.4 Operational backups

Core has no persistent runtime database to back up. Preserve these separately:

- Active customer verifier entry files and implementation directories
- Verifier package manifests and lockfiles
- `/etc/brs-core/brs-core.env`
- Production credentials, keys, and certificates in approved secret storage
- The installed systemd unit or configuration-management source
- The deployed BRS-Core release version or commit identifier

14. Troubleshooting

Core does not start

```
systemctl status brs-core.service --no-pager
journalctl -u brs-core.service -n 200 --no-pager
```

Check:

- Node.js is version 24 or newer.
- `ExecStart` uses the correct absolute Node.js path.
- `BRS_VERIFIERS_DIRECTORY` exists and is readable by the service account.

- Every top-level `.js` verifier exports a valid unique ID, stable label, positive schedule values, and `verify()`.
- Verifier package dependencies have been installed in the same-named implementation directory.
- Runtime environment values use valid types and ranges.

One invalid verifier prevents Core from starting.

Prometheus cannot scrape

```
ss -ltnp | grep ':9124'
curl --fail http://BRS_CORE_ADDRESS:9124/metrics
```

Check the configured listen address, Prometheus target, routing, host firewall, cloud firewall, security group, and source-address rules. If Core binds to `127.0.0.1`, a remote Prometheus host cannot connect directly.

JSON API returns HTTP 401

Authentication is enabled and the request omitted credentials or supplied an incorrect username/password. Use the configured Basic Authentication values.

JSON API returns HTTP 503

Authentication is enabled but the configured username or password is blank. Set both values and restart Core.

Verifier remains pending

Immediately after startup, initial execution proceeds asynchronously. Check `/health` for the running state and review logs. A slow initial verifier does not delay the HTTP interfaces.

Verifier reports error

An uncaught exception, malformed result, empty workflow, duplicate identifier, or other contract problem produces status `error`. Run the verifier directly to reproduce it and inspect safe terminal diagnostics:

```
node /path/to/verifiers/example.js
```

Verifier reports timeout

Core terminated the run after its verifier-specific timeout. Inspect remote service behavior, verifier diagnostics, and the verifier's own dependency timeouts. Repeated timeouts do not require restarting Core, but they indicate that the verifier or an external dependency needs attention.

Standalone verifier cannot find a dependency

Install from the verifier's same-named implementation directory:

```
cd /path/to/verifiers/example
npm ci
cd ..
node example.js
```

Do not install the dependency in the BRS-Core root package.

Verifier-local `.env` appears ignored

The verifier loads only its own adjacent `.env`. Existing process environment values override matching local values. Confirm the file is inside the same-named implementation directory and check for an already exported value.

15. Upgrades and rollback

BRS-Core release directories are replaceable. Customer verifiers and deployed configuration should be durable and independent from a specific release.

A reference layout is:

```
/opt/brs-core-1.0.0
/opt/brs-core-2.0.0
/opt/brs-core -> active version
/var/lib/brs-core/verifiers -> customer-owned verifier collection
/etc/brs-core/brs-core.env -> durable production configuration
```

Upgrade procedure

1. Record the active version or Git commit.

2. Verify the new release checksum.

3. Extract the new release beside the current release.

4. Run `npm ci`, `npm run check`, and `npm test` in the new release.

5. Install each active verifier's locked dependencies for the deployment.

6. Confirm the durable verifier and environment paths.

7. Stop or restart the service using the organization's deployment procedure.

8. Update `/opt/brs-core` atomically to the reviewed release.

9. Verify `/health`, `/metrics`, and the Prometheus target.

10. Keep the prior release until acceptance is complete.

This is an operating convention, not an automated upgrade manager.

Rollback procedure

1. Restore the previous reviewed release or repoint the active symlink.

2. Restore its matching verifier dependencies if they changed.

3. Restore the compatible configuration if necessary.

4. Restart `brs-core.service`.

5. Verify `/health`, `/metrics`, logs, and the Prometheus target.

16. Directory structure

Customer-relevant release paths:

```
brs-core-v1.0.0/
    LICENSE                   W3Patterns commercial license
    THIRD_PARTY_NOTICES       Third-party attribution and license notices
    README.md                 Product introduction and evaluation path
    .env.example              Core runtime configuration template
    package.json              Core commands and Node.js requirement
    package-lock.json         Locked root package metadata

    exporter/
        exporter.js           Discovery, scheduling, state, and HTTP runtime
        verifier-runner.js    Isolated verifier child-process runner

    verifiers/
        README.md             Active-directory behavior
        ...                   Customer-selected active verifiers

    examples/
```

```
    verifiers/              Skeleton plus five working NYMG examples
    prometheus/             Prometheus scrape example
    generic/                Reserved generic-pattern library
    nymg/                   Reserved opt-in NYMG reference area

deploy/
    systemd/                Reference brs-core.service unit

docs/
    BRS-Core-Product-Guide.md
    configuration.md
    installation.md
    json-api.md
    metrics.md
    operations.md
    release.md
    verifier-contract.md
```

Examples are never active automatically. An example becomes scheduled only when its top-level entry file and same-named implementation directory are copied into the configured active verifier directory, or when Core is explicitly pointed at `examples/verifiers` for evaluation.

17. Security and licensing reminders

- Keep TCP 9124 private and restrict it to approved scraper or proxy sources.
- `/metrics` and `/health` are unauthenticated.
- Use JSON API Basic Authentication only over a trusted encrypted path.
- Use least-privilege verifier accounts and bounded client timeouts.
- Never commit real `.env` files, production credentials, or private keys.
- Never log secrets or sensitive business data.
- Do not reuse the included NYMG public demo database password or SFTP key for another host, account, environment, or purpose.
- BRS-Core is proprietary W3Patterns commercial software governed by `LICENSE`.
- Customer-created verifier code and customer data remain customer-owned as described in `LICENSE`.
- Third-party dependencies remain governed by their respective terms in `THIRD_PARTY_NOTICES`.

For detailed contractual language, consult the legal files included with the release. For implementation-specific verifier behavior, consult each packaged example's README and `.env.example`.