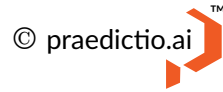


---

# Program Behavior Intelligence™ (PBI™)

## Technical Documentation



20260917

## Contents

|                                                                |           |
|----------------------------------------------------------------|-----------|
| <b>Program Behavior Intelligence™ (PBI™)</b>                   | <b>2</b>  |
| How it works . . . . .                                         | 3         |
| Rolling out PBI . . . . .                                      | 4         |
| Rolling out without a training phase (AUTO) . . . . .          | 5         |
| Documentation map . . . . .                                    | 5         |
| <b>PBI Deployment Guide</b>                                    | <b>6</b>  |
| Behavioral Enforcement Engine (BEE) . . . . .                  | 6         |
| Tolerance Levels . . . . .                                     | 6         |
| AUTO level . . . . .                                           | 7         |
| Common Deployment Pattern . . . . .                            | 7         |
| Java Protection Agent . . . . .                                | 8         |
| Node.js Protection Agent . . . . .                             | 8         |
| v24 / v26 . . . . .                                            | 8         |
| Python Protection Agent ( <b>Experimental</b> ) . . . . .      | 8         |
| Configuration Variables . . . . .                              | 9         |
| <b>PBI Concepts Glossary</b>                                   | <b>9</b>  |
| The Model . . . . .                                            | 9         |
| Verifier . . . . .                                             | 10        |
| Event Types . . . . .                                          | 10        |
| EVENT_STACK . . . . .                                          | 11        |
| EVENT_FILEPATH . . . . .                                       | 11        |
| EVENT_NETWORK . . . . .                                        | 11        |
| EVENT_SQL . . . . .                                            | 11        |
| EVENT_PROCESS . . . . .                                        | 11        |
| PBI SBOM™ (PBI Software Bill of Materials) . . . . .           | 11        |
| Training, Enforcement and AUTO . . . . .                       | 12        |
| Training Mode (PBI_LEVEL=TRAINING) . . . . .                   | 12        |
| Enforcement Mode (PBI_LEVEL=LOG, EXCEPTION, or KILL) . . . . . | 12        |
| AUTO (PBI_LEVEL=AUTO) . . . . .                                | 12        |
| The Ring Buffer . . . . .                                      | 12        |
| .dbg File . . . . .                                            | 12        |
| <b>PBI Explainer™</b>                                          | <b>13</b> |
| Running it . . . . .                                           | 13        |
| Browser UI . . . . .                                           | 13        |

|                                                                       |           |
|-----------------------------------------------------------------------|-----------|
| PBI ADA™: the agent interface . . . . .                               | 13        |
| <b>Getting Started with PBI ADA™</b>                                  | <b>14</b> |
| Prerequisites . . . . .                                               | 14        |
| Step 1: Start PBI Explainer™ . . . . .                                | 14        |
| Step 2: Connect your AI agent . . . . .                               | 14        |
| Option A: Claude Code . . . . .                                       | 14        |
| Option B: OpenCode . . . . .                                          | 15        |
| Step 3: Start analyzing . . . . .                                     | 16        |
| Browser UI . . . . .                                                  | 16        |
| <b>PBI Explainer™ MCP Tool Reference</b>                              | <b>16</b> |
| Tool Selection Guide . . . . .                                        | 16        |
| Model Summaries . . . . .                                             | 17        |
| inspect_model . . . . .                                               | 18        |
| search_sboms . . . . .                                                | 18        |
| search_verifiers . . . . .                                            | 19        |
| compare_models . . . . .                                              | 19        |
| verify_model . . . . .                                                | 20        |
| simulate_model_from_dbg . . . . .                                     | 20        |
| analyze_jar . . . . .                                                 | 21        |
| merge_model . . . . .                                                 | 21        |
| list_registered_events . . . . .                                      | 22        |
| Common Error Scenarios . . . . .                                      | 22        |
| <b>Reading and Auditing a Model</b>                                   | <b>22</b> |
| Classification Framework . . . . .                                    | 23        |
| Attack Indicator Heuristics . . . . .                                 | 23        |
| EVENT_SQL . . . . .                                                   | 23        |
| EVENT_FILEPATH . . . . .                                              | 23        |
| EVENT_NETWORK . . . . .                                               | 23        |
| EVENT_PROCESS . . . . .                                               | 23        |
| SBOM . . . . .                                                        | 24        |
| Interpreting a single model . . . . .                                 | 24        |
| Healthy model signals . . . . .                                       | 24        |
| Signals worth highlighting . . . . .                                  | 24        |
| Security-relevant PBI SBOM™ categories . . . . .                      | 24        |
| Remediation Guidance . . . . .                                        | 25        |
| <b>Troubleshooting</b>                                                | <b>25</b> |
| Reading the logs . . . . .                                            | 25        |
| BEE exits with “Failed to initialize logger” . . . . .                | 26        |
| BEE model fails to load because its directory doesn’t exist . . . . . | 26        |
| Shared-memory handshake fails . . . . .                               | 26        |
| Protection Agent never loads (no error at all) . . . . .              | 26        |
| Legitimate behavior flagged in protection mode . . . . .              | 27        |
| AUTO never switches to protection . . . . .                           | 27        |

## Program Behavior Intelligence™ (PBI™)

The cybersecurity platform from praedictio.ai is built on **Program Behavior Intelligence™ (PBI™)**, which prevents attacks by allowing only a program’s intended behavior. Knowing the typical usage patterns of our clients’ programs, PBI prevents any behavior outside the model, known or unknown, in real time. PBI builds an auditable deterministic model of intended behavior, not a trained AI model: enforcement is a strict

in-the-model-or-not decision, with no probability and no scoring.

praedictio.ai, Inc. products and categories of products may be covered by one or more patents in the U.S. and internationally, as is indicated below.

This information is provided to satisfy the virtual patent marking provisions of various jurisdictions, in particular it is intended to serve as notice under 35 U.S.C. § 287(a).

Items listed below may be sold individually or as part of a combination product.

|                  |                           |
|------------------|---------------------------|
| PBI Explainer™   | <b>US Patent 11989296</b> |
| PBI Enforcement™ | <b>US Patent 12547718</b> |

*Additional patents, pending and issued, in the U.S. and internationally are not listed.*

See also <https://praedictio.ai/legal.html> ↗

[Technical Overview Video https://praedictio.ai/technicaloverview](https://praedictio.ai/technicaloverview) ↗

PBI Features:

- Frictionless integration in development cycle
- Ensures security in run-time
- Program behaves as expected
- Prevents behavior outside the model, known or unknown

## How it works

### Pod / Node Environment

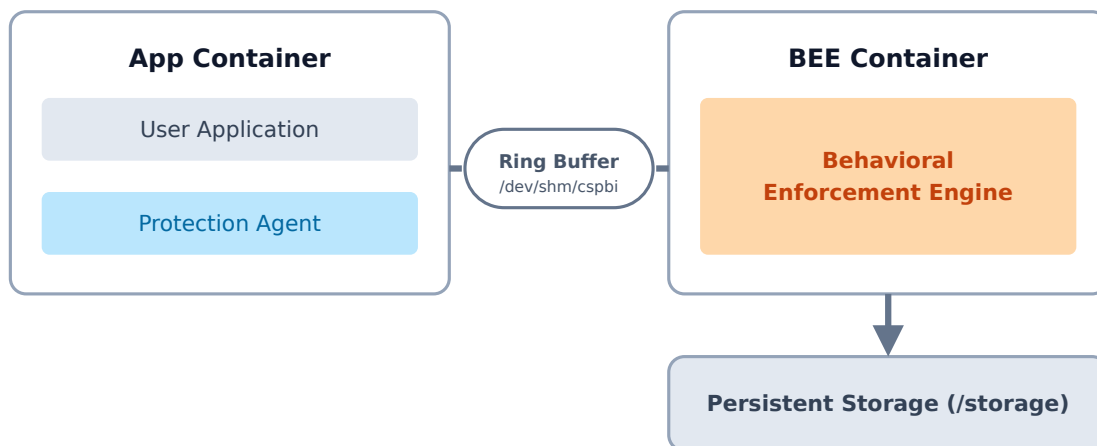


Figure 1: High Level Architecture

PBI requires **no rules or policies**. It frictionlessly learns what the correct behavior of an application is by running the standard QA suite, and any behavior that was unidentified during training is treated as behavior outside the model once protection is enabled.

Two components cooperate to make this work:

1. **Protection Agent**, language-specific; instruments the application and reports what it does

2. **Behavioral Enforcement Engine (BEE)**, the runtime component of PBI Enforcement™: receives those events, builds the behavioral model, and enforces it

They exchange information via shared memory, so both must run on the same node (endpoint) or in the same pod, under the same effective user ID.

PBI\_LEVEL sets how BEE handles behavior outside the model. **TRAINING** learns it, while **LOG**, **EXCEPTION** and **KILL** protect against it. **AUTO** starts as **TRAINING** and switches itself to **LOG**. Usually you train first, then protect using the model training produced. See deployment-guide.md for the levels and concepts.md for the model/verifier/PBI SBOM™ vocabulary.

All the runtime protection code ships in a single container image (PBI Explainer ships separately, see explainer.md):

```
cybersentry.azurecr.io/cybersentry/pbi:${TAG}
```

Note: **cybersentry** is the container-registry namespace for praedictio.ai images. The product and company brand is praedictio.ai.

Supported platforms:

- x86-64 with AVX2 and FSGSBASE
- aarch64 (with its mandatory NEON) is still in development

Supported runtimes:

| Runtime | Versions           |
|---------|--------------------|
| Java    | LTS JDK 11–25      |
| Node.js | v20, v22, v24, v26 |
| Python  | 3.13, 3.14         |

## Rolling out PBI

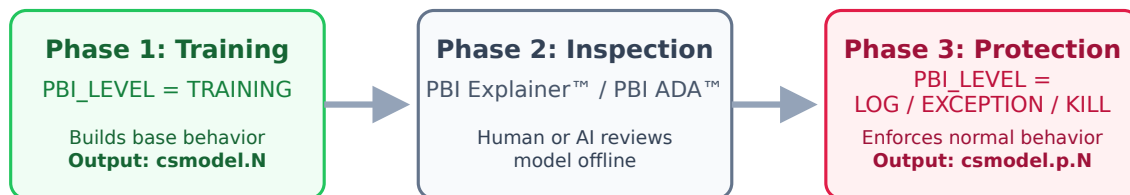


Figure 2: Operational Lifecycle

This is the recommended path from first install to production enforcement.

1. Identify the applications that need protection and their respective runtimes.
2. Read deployment-guide.md for the deployment pattern; each runtime has a sample manifest (java.yaml, node-20-22.yaml, node-24-26.yaml, python.yaml).

3. Deploy to a dev/test/staging environment first, to confirm compatibility and understand the impact on your applications.
4. Set the app container to **Training** (`PBI_LEVEL=TRAINING`).
5. Exercise the application thoroughly with integration and end-to-end tests, plus “captive normal use”. **Do not run penetration tests or any other attack-like activity during this phase**, or BEE will learn the attack as normal behavior.
6. Once training is complete, review the resulting model with the PBI Explainer to understand the application’s behavior and its dependencies.
7. Switch the app container to **Protection** (`PBI_LEVEL=LOG` or `PBI_LEVEL=KILL`), still in dev/test/staging. This surfaces false positives before they can affect production.
8. Watch `LOG_PATH` (default of `PBI_STORAGE_PATH/logs/pbi.log`) for `level=SEVERE` entries. Each one means either a genuine attack attempt (unlikely in this environment) or a gap in training coverage. BEE writes the behavior it saw into a model with a `.p` suffix, see `troubleshooting.md` for how to fold that behavior back into your model.
9. When no more `SEVERE` entries appear, promote to production.
10. Keep monitoring the logs and inspecting new models as you go.
11. In production, start at `PBI_LEVEL=LOG` and move to `PBI_LEVEL=EXCEPTION` or `PBI_LEVEL=KILL` once you are confident no legitimate behavior is being flagged.

**IMPORTANT** Do not use `PBI_LEVEL=EXCEPTION` in step 7. Resulting model will contain exception handling behavior which **IS NOT** legitimate behavior, rendering the resulting model unusable for further deployment.

## Rolling out without a training phase (AUTO)

When exercising the application in dev/test/staging isn’t practical, use `PBI_LEVEL=AUTO`. How it works is in `deployment-guide.md`.

1. Deploy as in steps 1–3, with the app container set to `PBI_LEVEL=AUTO` and `PBI_NO_VIOLATION_WINDOW` long enough to cover the application’s regular load.
2. Wait for BEE to log the switch to protection.
3. Review the `csmodel.N` saved at the switch with the PBI Explainer.
4. Handle `level=SEVERE` entries and the resulting `csmodel.p.N` model as in steps 8–10.

**IMPORTANT** Everything BEE sees before the switch, an attack included, becomes part of the model. Review the model from step 3 before relying on it.

## Documentation map

### Deploying and running PBI

| Document                         | Covers                                                                                                                       |
|----------------------------------|------------------------------------------------------------------------------------------------------------------------------|
| <code>deployment-guide.md</code> | BEE configuration, pod topology, per-runtime agent setup. The go-to deployment reference.                                    |
| <code>troubleshooting.md</code>  | Startup failures, shared-memory problems, log levels, recovering behavior from <code>.p</code> models                        |
| <code>concepts.md</code>         | Glossary: models, verifiers, event types, PBI SBOM™, training vs. enforcement vs. AUTO, ring buffer, <code>.dbg</code> files |

### Inspecting models

| Document                | Covers                                                                         |
|-------------------------|--------------------------------------------------------------------------------|
| explainer.md            | What PBI Explainer™ is, how to run it, the browser UI                          |
| tool-reference.md       | Full MCP tool reference, parameters, output shapes, pagination, tool selection |
| ada-onboarding.md       | Connecting an AI agent (Claude Code or OpenCode) to PBI Explainer              |
| security-model-audit.md | Classifying model differences, attack indicators, remediation guidance         |

### Agent persona

Folder **agent/** holds the operating instructions for PBI ADA™, the AI assistant persona. These files tell an agent *how to behave*; everything above tells it (and you) *what is true*. If you are reading as a human, you want the documents above.

## PBI Deployment Guide

How to deploy PBI™ alongside an application. For what PBI is and the recommended rollout path, start at README.md. For vocabulary (models, verifiers, PBI SBOM™), see concepts.md. When something doesn't start, see troubleshooting.md.

### Behavioral Enforcement Engine (BEE)

The Behavioral Enforcement Engine (BEE) is a native binary that runs as a sidecar to the monitored process. It reads events from the shared memory ring buffer, checks them against the learned behavioral model, and reacts according to the configured tolerance level, see Tolerance Levels.

BEE opens its shared memory file at `/dev/shm/cspbi`. Override with the `PBI_SHM_PATH` env var.

BEE stores its model/debug files in `/storage`. The full path depends on what is provided by the agent. Override the storage folder with `PBI_STORAGE_PATH` env var.

BEE exposes an observability endpoint at port 8888 (override with `--port`):

- `POST /save?bee_id=0`: writes the current model to file. BEE and the monitored process keep running, so this can be called at any point to capture the behavior learned so far.
- `GET /metrics`: Prometheus-compatible training/runtime statistics.

BEE maintains behavioral models, so persistent storage must be mounted at `/storage` (or wherever `PBI_STORAGE_PATH` points).

#### Optional BEE flags:

- `--dbg <path>`, writes the event feed to a separate `.dbg` file, usable later to recreate/verify a model.
- `--tag <n>`, sets a numeric tag on the running BEE instance.
- `--log <path>`, sets the BEE log file path (default `/storage/logs/pbi.log`).
- `--stream <lib> <config>`, loads an event-stream plugin `.so`.
- `--port <port>`, sets the BEE observability endpoint port (default 8888).

### Tolerance Levels

The tolerance level decides what BEE does with an event that is outside the model. Set it on the app container with the `PBI_LEVEL` env var; the agent passes it to BEE.

| PBI_LEVEL | Mode                            | Behavior                                                                                                            |
|-----------|---------------------------------|---------------------------------------------------------------------------------------------------------------------|
| TRAINING  | Training                        | Learns normal behavior, saves the model on shutdown                                                                 |
| LOG       | Protection (LOG)                | Logs the event outside the model but lets it through, saves the model on shutdown                                   |
| EXCEPTION | Protection (EXCEPTION)          | Throws an exception into the monitored process on an event outside the model, saves the model on shutdown           |
| KILL      | Protection (KILL)               | Terminates the monitored process on an event outside the model, then saves its model                                |
| AUTO      | Training, then Protection (LOG) | Starts as TRAINING and switches itself to LOG once no event outside the model has been seen for a configured window |

Note: TRAINING writes a `csmodel.N` model; the protection levels write a `csmodel.p.N` model. AUTO writes whichever of the two matches the level it is operating at. See `concepts.md`.

## AUTO level

PBI\_LEVEL=AUTO makes a separate training run optional. BEE learns the model from the load the application actually receives and decides by itself when to start protecting it, based on how often it still sees events outside the model.

1. **Training phase:** A BEE started with AUTO begins at TRAINING. Every event outside the model is learned and restarts the no-violation window. Under representative load these events are frequent right after startup and quickly become rare.
2. **Switch:** Once PBI\_NO\_VIOLATION\_WINDOW seconds (default 300) have passed without an event outside the model, the next event that matches the model switches BEE to LOG and saves the model learned so far as a Training model, `csmodel.N`.
3. **Protection (LOG) phase:** From here BEE behaves exactly like PBI\_LEVEL=LOG: events outside the model are logged at SEVERE and let through, and POST /save or shutdown writes a `csmodel.p.N` model.

Behavior to be aware of:

- The switch is only evaluated when an event matches the model. An application that receives no traffic stays at TRAINING, however long ago the last event outside the model was.
- The switch is one-way and ends at LOG. AUTO never moves to EXCEPTION or KILL.
- The AUTO state is kept in memory only. A restarted BEE loads the most recent Training model (or none) and starts again at TRAINING with a fresh window.

## Common Deployment Pattern

Every language agent follows the same pod/container topology:

1. **Init container** (`agent-install`) copies the Protection Agent files into a temporary, non-persistent `protectionAgent` volume, non-persistent so old code doesn't linger after an image update.
2. **BEE sidecar container**, persistent `storage` volume mounted at `/storage` (or wherever PBI\_STORAGE\_PATH points), port 8888 exposed.
3. **App container**, mounts the `protectionAgent` volume and sets one language-specific env var to load the agent. That env var applies to every process of that language in the container. If the container runs several apps in sequence, set it immediately before launching the one you want protected, not globally. Every app container also requires PBI\_MODEL\_PATH (e.g. `models/csmodel`), a path, relative to the BEE storage root, where the model file for this app will be stored/loaded. **Every directory component of that path must already exist** under the storage root. It also requires PBI\_LEVEL, see Tolerance Levels.

Each Protection Agent consists of two layers: a language-specific piece, and common native code distributed as `libpbi.so`. We provide gnu and musl variants for:

- **Java** (`/gnu/java`, `/musl/java`)
- **Node.js** (`/gnu/node-v{20,22,24,26}`, `/musl/node-v{20,22,24,26}`)
- **Python** (`/gnu/python`, `/musl/python`)

Before wiring any of this up, determine whether the image you're protecting was built with glibc (gnu) or musl. **If this is done incorrectly, the Protection Agent will not load.** Check with:

```
podman run --rm --entrypoint /bin/bash <image> ldd --version
```

If `musl` appears in the output, use the `/musl/...` paths; otherwise use `/gnu/...`

The provided sample manifests are intended only to illustrate the deployment concept and are not suitable for direct use. They contain several variables in the `${NAME}` format to highlight information the customer must provide, and they may omit certain elements needed for a proper deployment.

## Java Protection Agent

|                    |                                                                                                                                                                                                           |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Agent files        | <code>/gnu/java</code> or <code>/musl/java</code>                                                                                                                                                         |
| Env var            | <code>JAVA_TOOL_OPTIONS="--add-reads=java.base=ALL-UNNAMED<br/>--add-opens=java.base/java.nio=ALL-UNNAMED<br/>-agentpath:\${protectionAgent}/libpbi.so<br/>-javaagent:\${protectionAgent}/pbi.jar"</code> |
| Supported versions | LTS JDK 11–25                                                                                                                                                                                             |
| Sample manifest    | <code>java.yaml</code>                                                                                                                                                                                    |

## Node.js Protection Agent

|                    |                                                                                |
|--------------------|--------------------------------------------------------------------------------|
| Agent files        | <code>/gnu/node-v{20,22,24,26}</code> , <code>/musl/node-v{20,22,24,26}</code> |
| Env var            | <code>NODE_OPTIONS="-r \${protectionAgent}/agent.js"</code>                    |
| Supported versions | v20, v22, v24, v26                                                             |
| Sample manifest    | <code>node-20-22.yaml</code> / <code>node-24-26.yaml</code>                    |

### v24 / v26

Node v24 and v26 need one more env var on the app container: `NODE_OPTIONS` alone is not enough for these versions.

- `LD_PRELOAD="${protectionAgent}/libnode_patch.so"`

`libnode_patch.so` ships alongside the rest of the agent files in the `node-v24/node-v26` directories.

## Python Protection Agent (Experimental)

|                    |                                                      |
|--------------------|------------------------------------------------------|
| Agent files        | <code>/gnu/python</code> , <code>/musl/python</code> |
| Env var            | <code>PYTHONPATH="\${protectionAgent}"</code>        |
| Supported versions | 3.13, 3.14                                           |
| Sample manifest    | <code>python.yaml</code>                             |

## Configuration Variables

| Variable                | Set on                 | Description                                                                                                                                                               |
|-------------------------|------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PBI_LEVEL               | App container          | Tolerance level: <b>TRAINING</b> , <b>LOG</b> , <b>EXCEPTION</b> , <b>KILL</b> , <b>AUTO</b> . Required.                                                                  |
| PBI_NO_VIOLATION_WINDOW | App container          | Used only with <b>PBI_LEVEL=AUTO</b> : the number of seconds without an event outside the model before BEE switches from <b>TRAINING</b> to <b>LOG</b> . Defaults to 300. |
| PBI_STORAGE_PATH        | BEE container          | Storage root for models, debug files. Defaults to <code>/storage</code> ; must match wherever the persistent volume is mounted.                                           |
| PBI_SHM_PATH            | BEE and app containers | Shared memory file path. Defaults to <code>/dev/shm/cspbi</code> ; both sides must agree.                                                                                 |
| PBI_MODEL_PATH          | App container          | Model file path relative to the storage root, e.g. <code>models/csmodel</code> . Required.                                                                                |

## PBI Concepts Glossary

This document defines the core concepts behind Program Behavior Intelligence™ (PBI™), the vocabulary used throughout the rest of the documentation. For what PBI is and how to roll it out, see [README.md](#).

PBI's model is deterministic, not a trained AI model: at enforcement time every event either matches the model or does not, with no probability and no scoring.

### The Model

A **model file** (`csmodel`, `csmodel.N` where N is a number) is the artifact produced by BEE after each run. It encodes everything BEE learned about normal application behavior during the training period.

The model is a collection of **verifiers** - one representing all events observed that share (`event_id`, `event_type`) pair. When BEE runs in enforcement mode, every observed event is checked against its verifier. Events not matching a verifier are outside the model.

Model files are binary. They cannot be read directly, use PBI Explainer to inspect them.

A new model file is created with each shutdown of the **BEE**, and with each call to its `/save` endpoint (see [deployment-guide.md](#)). The models are incremental by design. Each model contains the behavior from the model initially loaded during **BEE** startup + new behavior observed during runtime. Under **PBI\_LEVEL=AUTO** one more model file is created, at the moment BEE switches from training to protection (see **AUTO** level). The standard file naming convention is `csmodel[.p].<number>`, where the `.p` designates a model generated while BEE was operating in **Protection** Mode. Both model types share an identical format, the naming convention solely determines which model is loaded at startup.

- **Training** Models: `csmodel.<number>` (e.g., `csmodel.10`)
- **Protection** Models: `csmodel.p.<number>` (e.g., `csmodel.p.11`)

This convention is important as it determines which model will be loaded by the **BEE** during startup. At initialization the most recent **Training** model (the file with the highest numerical suffix that lacks the `.p`) will be selected.

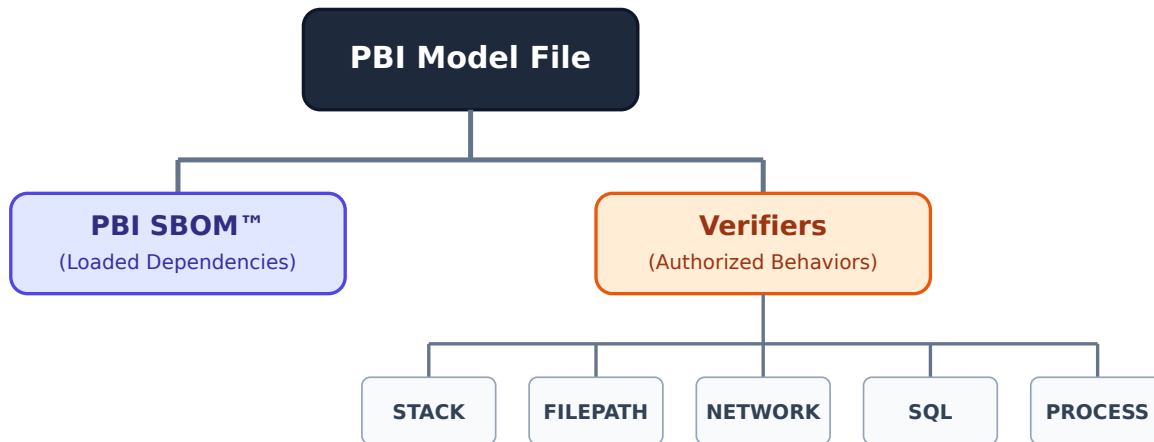


Figure 3: Anatomy of a PBI Model

## Verifier

A **verifier** is an internal structure representing BEE's learned observed behavior. It is used for validation of incoming events.

What counts as accepted, known behavior depends on the event type:

| Event type            | What must match to pass enforcement          |
|-----------------------|----------------------------------------------|
| <b>EVENT_STACK</b>    | Stacktrace only                              |
| <b>EVENT_FILEPATH</b> | Stacktrace + file path (common prefix match) |
| <b>EVENT_SQL</b>      | Stacktrace + normalized SQL pattern          |
| <b>EVENT_NETWORK</b>  | Stacktrace + network connection details      |
| <b>EVENT_PROCESS</b>  | Stacktrace                                   |

### Conditions that fall outside the model:

- The incoming stacktrace was never seen for this event (new call path) → outside the model
- The stacktrace is known but the parameter (file path / SQL query) doesn't match any learned pattern → outside the model
- The stacktrace and parameter match, but the request context doesn't match any learned context → outside the model

**One verifier per (event\_id, event\_type) pair.** Multiple stacktraces and parameter combinations for the same method are all recorded within the same verifier.

## Event Types

PBI instruments applications at multiple layers. Each layer produces a different event type:

## EVENT\_STACK

A **function/method call** observed in the monitored application. This is the broadest event type representing various captured behavior (default event type).

The verifier for an `EVENT_STACK` records the full call stack at the time of invocation.

## EVENT\_FILEPATH

A **file system access** to a specific path.

**Path matching uses prefix semantics.** If the model learned `/opt/activemq/` as an allowed path, any access to `/opt/activemq/config/broker.xml` will also pass. This means the model stores the most specific common prefix observed during training, not every exact path individually.

Filepath verifiers are important for security review, they show exactly which file-system areas the application is allowed to access under enforcement.

## EVENT\_NETWORK

An **outbound network connection** to a specific host/endpoint. Captured at the socket level.

Low counts are common in training runs where external services were not exercised. A model with very few network verifiers may under-enforce outbound connectivity.

## EVENT\_SQL

A **SQL query** sent to a database. Queries are normalized before being stored or compared:

- All literal values (strings, numbers, booleans, NULL, bind placeholders like `$1`) are replaced with `?`
- The query is uppercased
- Multi-statement queries are preserved (e.g. `UPDATE ... ; DELETE ...`)

Example: `SELECT * FROM users WHERE id = 42` is stored as `SELECT * FROM USERS WHERE ID = ?`

This means the verifier captures the **query structure**, not specific values. A query with the same structure but different parameter values will match. A structurally different query from the same call site will be flagged as behavior outside the model.

## EVENT\_PROCESS

A subprocess or OS process invocation (e.g. `exec`, `fork`). Rare in most application models.

---

## PBI SBOM™ (PBI Software Bill of Materials)

PBI SBOM™ is the **list of all modules, libraries, and packages that were loaded** during the training run. It is built automatically, no manual declaration required.

Each PBI SBOM™ entry stores two fields:

- **code\_reference**, the path or identifier of the loaded code (used as the unique key)
- **checksum**, a 256-bit hash of the loaded code at that path

If the same **code\_reference** is loaded again with a different **checksum**, the model updates the entry while raising an alert in protection mode. During **compare\_models**, PBI SBOM™ entries with the same path but different checksums indicate the code at that location changed between training runs, a potential supply-chain or tampering signal.

The PBI SBOM™ represents what was *actually loaded at runtime* during training, which is often a subset of what appears in `package.json` or `pom.xml`, lazy-loaded or conditionally-loaded packages only appear if they were exercised during training.

**Security use case:** The PBI SBOM™ lets you audit exactly which third-party code was running inside your application at the time of capture, including a cryptographic fingerprint of each loaded library. Unexpected packages or changed checksums indicate supply-chain risk.

---

## Training, Enforcement and AUTO

The `PBI_LEVEL` environment variable on the app container selects how BEE operates: Training mode, Enforcement mode, or AUTO.

### Training Mode (`PBI_LEVEL=TRAINING`)

BEE observes all events and builds the model. Nothing is blocked. Every observed pattern is recorded as a verifier.

**Goal:** capture a complete and representative model of normal behavior.

**Risk:** if attacks or unexpected behavior occur during training, they are learned as normal and will not be flagged in enforcement.

### Enforcement Mode (`PBI_LEVEL=LOG, EXCEPTION, or KILL`)

BEE checks every observed event against the model. If an event doesn't match any verifier, it is outside the model. What happens next depends on the level: BEE logs the event outside the model and lets it through (`LOG`), throws an exception into the monitored process (`EXCEPTION`), or terminates it (`KILL`). See `deployment-guide.md` for the full table.

### AUTO (`PBI_LEVEL=AUTO`)

BEE starts in Training mode and switches itself to Enforcement mode (`LOG`) once the application stops showing new behavior. See AUTO level.

---

## The Ring Buffer

The **ring buffer** is the shared-memory channel (`/dev/shm/cspbi` by default) through which the Protection Agent sends events to BEE.

---

### .dbg File

A **debug file** (`.dbg`) is an event log written by BEE when started with the `--dbg` flag. It records every raw event received from the ring buffer in chronological order.

`.dbg` files are used for:

- Verifying a capture against a model (`verify_model`)
- Replaying events (`simulate_model_from_dbg`)

A `.dbg` file must be written by a gracefully-stopped BEE, a hard kill (`SIGKILL`, `podman pod stop`) may leave the file incomplete or corrupt.

## PBI Explainer™

PBI Explainer™ is a standalone diagnostic tool for inspecting model files. It is **not** part of the runtime protection pod. It's a separate, self-contained image you run on demand against a directory of model files.

It serves two audiences from a single port: a browser UI and an MCP endpoint for AI agents.

### Running it

Replace the placeholders below with your values:

| Placeholder                          | Description                                       |
|--------------------------------------|---------------------------------------------------|
| <code>&lt;/path/to/models&gt;</code> | Directory on your host containing the model files |
| <code>&lt;port&gt;</code>            | The port PBI Explainer™ listens on                |
| <code>&lt;tag&gt;</code>             | specific version of PBI Explainer™                |

```
podman run --rm -it \
  -e HOST_PATH=</path/to/models> \
  -v </path/to/models>:/storage \
  -p <port>:<port> \
  cybersentryregistry.azurecr.io/cybersentry/pbie:<tag> <port>
```

Two requirements are mandatory for the agent to work correctly inside the container:

- `-v </path/to/models>:/storage` sets the mount target, which must be exactly `/storage`. PBI Explainer™ resolves all model paths relative to this location.
- `-e HOST_PATH=</path/to/models>` must match the host directory you mounted above. When the agent passes an absolute host path (e.g. `</path/to/models>/csmodel.1`), PBI Explainer™ strips the `HOST_PATH` prefix and maps it to `/storage/csmodel.1` inside the container.

**Important:** all model files the agent references must live under `HOST_PATH`. Paths outside that directory are not remapped and will fail with a “file not found” error inside the container.

PBI Explainer™ serves both interfaces on the single port you passed:

| Service    | Address                                        | Purpose          |
|------------|------------------------------------------------|------------------|
| MCP server | <code>http://localhost:&lt;port&gt;/mcp</code> | AI agent tooling |
| Browser UI | <code>http://localhost:&lt;port&gt;/</code>    | Visual explorer  |

### Browser UI

Open `http://localhost:<port>/` to browse **Verifiers**, **PBI SBOM™ entries**, and **Registered Events** for any model file found under `/storage`. Pick a model from the selector, then filter or page through each tab. No AI agent required.

### PBI ADA™: the agent interface

PBI ADA™ (PBI Advanced Defensive Agent) is the persona an AI agent adopts by connecting to PBI Explainer™ MCP endpoint at `http://localhost:<port>/mcp`. It can explain, compare, and audit model files conversationally rather than through the browser UI.

- To connect an agent, see [ada-onboarding.md](#).
- For the tools it exposes, see [tool-reference.md](#).
- For judging what a model comparison actually means, see [security-model-audit.md](#).

## Getting Started with PBI ADA™

PBI ADA™ (PBI Advanced Defensive Agent) is an AI assistant pre-loaded with PBI™ expertise. Once connected to your PBI Explainer™ instance, it can analyze behavioral model files, audit security posture, assist with deployment, and answer PBI questions.

### Prerequisites

- The PBI Explainer™ container image (provided separately)
- Podman or Docker
- An AI agent: **Claude Code** or **OpenCode** (both covered below)

### Step 1: Start PBI Explainer™

Replace the placeholders below with your values:

| Placeholder                          | Description                                       |
|--------------------------------------|---------------------------------------------------|
| <code>&lt;/path/to/models&gt;</code> | Directory on your host containing the model files |
| <code>&lt;port&gt;</code>            | The port PBI Explainer™ listens on                |
| <code>&lt;tag&gt;</code>             | specific version of PBI Explainer™                |

```
podman run --rm -it \
  -e HOST_PATH=</path/to/models> \
  -v </path/to/models>:/storage \
  -p <port>:<port> \
  cybersentryiaregistry.azurecr.io/cybersentry/pbie:<tag> <port>
```

Two requirements are mandatory for the agent to work correctly inside the container:

- `-v </path/to/models>:/storage` sets the mount target, which must be exactly `/storage`. PBI Explainer™ resolves all model paths relative to this location.
- `-e HOST_PATH=</path/to/models>` must match the host directory you mounted above. When the agent passes an absolute host path (e.g. `</path/to/models>/csmodel.1`), PBI Explainer™ strips the `HOST_PATH` prefix and maps it to `/storage/csmodel.1` inside the container.

**Important:** all model files the agent references must live under `HOST_PATH`. Paths outside that directory are not remapped and will fail with a “file not found” error inside the container.

PBI Explainer™ serves both interfaces on the single port you passed:

| Service    | Address                                        | Purpose          |
|------------|------------------------------------------------|------------------|
| MCP server | <code>http://localhost:&lt;port&gt;/mcp</code> | AI agent tooling |
| Browser UI | <code>http://localhost:&lt;port&gt;/</code>    | Visual explorer  |

### Step 2: Connect your AI agent

#### Option A: Claude Code

Register the PBI Explainer™ MCP server once:

```
claude mcp add pbi http://localhost:<port>/mcp --transport http
```

Then start Claude Code in your working directory and give it this first prompt:

Read the MCP resource `pbi://doc/agent/ada-system-prompt.md` and use it as your operating instructions

Full documentation for PBI ADA™ is embedded in PBI Explainer™ and available as MCP resources. Claude Code will fetch individual files on demand as the conversation progresses.

## Option B: OpenCode

You will need the **PBI ADA™ context folder** (provided alongside this guide). It contains the markdown files that define knowledge and capabilities of PBI ADA™.

Place the folder anywhere your agent can read from, typically your working directory or the directory where you launch the agent.

```
ADA/
|-- README.md
|-- concepts.md
|-- deployment-guide.md
|-- troubleshooting.md
|-- tool-reference.md
|-- security-model-audit.md
|-- explainer.md
`-- agent/
    |-- ada-system-prompt.md
    |-- security-playbook.md
    `-- analyst-playbook.md
```

Configure the PBI Explainer™ MCP server in `~/.config/opencode/opencode.json` (create the file if it does not exist):

```
{
  "$schema": "https://opencode.ai/config.json",
  "mcp": {
    "pbi": {
      "type": "remote",
      "url": "http://localhost:<port>/mcp",
      "enabled": true
    }
  }
}
```

Then start OpenCode and give it this first prompt:

Read `ADA/agent/ada-system-prompt.md` and use it as your operating instructions

The agent will read additional files from the folder as needed during the conversation.

## Step 3: Start analyzing

Once PBI ADA™ has loaded its instructions, try any of the following. Reference model files by their path on your host machine, PBI Explainer™ remaps them automatically.

### Model overview

Give me a high-level summary of `csmodel.1`

### Security audit (two models)

Compare `csmodel.3` and `csmodel.p.4`. Did anything security-relevant change?

### Find specific behaviors

What network connections does this application make?

Show me all file path access patterns in the model.

### Package audit

What packages are loaded? Are any of them security-sensitive?

### Verify a recording

Does `run.dbg` show any violations against `csmodel.1`?

---

## Browser UI

Open `http://localhost:<port>/` in a browser to explore verifiers, PBI SBOM™ entries, and registered events visually without an AI agent. See `explainer.md` for details.

## PBI Explainer™ MCP Tool Reference

This document describes every tool exposed by the PBI Explainer™ MCP server, what each one does, what parameters it takes, and how to interpret its output. For how to connect a client to the MCP endpoint, see `ada-onboarding.md`.

All path parameters must be **absolute host paths**: PBI Explainer™ strips the `HOST_PATH` prefix and maps them under `/storage` inside the container (see `ada-onboarding.md`). Relative paths will not resolve, and paths outside `HOST_PATH` will not be remapped. Paginated output is indexed from 0 (`page=0`).

---

## Tool Selection Guide

| Goal                                       | Tool                                 |
|--------------------------------------------|--------------------------------------|
| Orient on a model you have not seen before | the six summaries, in pipeline order |

| Goal                                                         | Tool                                                                  |
|--------------------------------------------------------------|-----------------------------------------------------------------------|
| List every entry point the app is invoked through            | <code>summarize_entrypoints</code>                                    |
| List every loaded module, and identify the runtime           | <code>summarize_sbom</code>                                           |
| List every file path the app may touch                       | <code>summarize_filepaths</code>                                      |
| List every network destination it may reach                  | <code>summarize_network</code>                                        |
| List every SQL query it may run                              | <code>summarize_sql</code>                                            |
| List every command it may execute                            | <code>summarize_processes</code>                                      |
| Full detail / deep dive                                      | <code>inspect_model</code>                                            |
| Is library X or package Y in the model?                      | <code>search_sboms</code> with the library name                       |
| Does the app do X? (behavior question)                       | <code>search_verifiers</code> with a relevant keyword                 |
| What network calls does it make?                             | <code>search_verifiers</code> filtered by <code>EVENT_NETWORK</code>  |
| What files does it access?                                   | <code>search_verifiers</code> filtered by <code>EVENT_FILEPATH</code> |
| Does it use a SQL database?                                  | <code>search_verifiers</code> filtered by <code>EVENT_SQL</code>      |
| What changed between two models?                             | <code>compare_models</code> with both paths                           |
| Is this <code>.dbg</code> capture consistent with the model? | <code>verify_model</code> with model + dbg path                       |
| Create a model from a <code>.dbg</code> file                 | <code>simulate_model_from_dbg</code>                                  |
| Analyze a JAR against a Java model                           | <code>analyze_jar</code> with model + JAR path                        |
| Merge several training runs                                  | <code>merge_model</code> with all model paths                         |
| Enumerate every registered method or function signature      | <code>list_registered_events</code>                                   |

## Model Summaries

Six tools, one per facet of a model. Each takes a single `model_path` and returns its whole facet in one response, with no pagination.

| Tool                               | Answers                                      |
|------------------------------------|----------------------------------------------|
| <code>summarize_entrypoints</code> | How is this application invoked?             |
| <code>summarize_sbom</code>        | What code is loaded, and what runtime is it? |
| <code>summarize_filepaths</code>   | What files may it read and write?            |
| <code>summarize_network</code>     | What may it connect to?                      |
| <code>summarize_sql</code>         | What may it ask the database?                |
| <code>summarize_processes</code>   | What may it execute?                         |

Run together they are the orientation pipeline, the standard first look at any model. The order to call them in, and how to turn the six outputs into conclusions, is in `agent/analyst-playbook.md`.

### When to use:

- First look at any model file
- User asks “what is this model?” or “give me a summary”
- Establishing context before a deeper investigation

| Parameter               | Type   | Description                          |
|-------------------------|--------|--------------------------------------|
| <code>model_path</code> | string | Absolute host path to the model file |

### Key output to extract:

- Total verifier count and PBI SBOM™ entry count

- Event type breakdown (EVENT\_STACK / EVENT\_FILEPATH / EVENT\_NETWORK / EVENT\_SQL / EVENT\_PROCESS)
- Application identity: runtime, framework, entry point
- PBI SBOM™: list of loaded modules / packages
- Blast radius: the file paths, network destinations, SQL queries and commands that enforcement will permit

---

## inspect\_model

**Purpose:** Deep inspection of a model file. Far more detailed than the summaries, includes stacktraces and per-verifier analysis.

**When to use:**

- You need to examine specific call patterns or stacktraces
- Investigating a specific area the summaries surfaced

| Parameter  | Type   | Description                          |
|------------|--------|--------------------------------------|
| model_path | string | Absolute host path to the model file |

**Key output to extract:**

- Suspicious patterns if present
- Depth of behavioral coverage per module

**Note:** This is a model dump and produces orders of magnitude more output than the summaries. It is worth running only against a specific investigation goal, as the full output is too large to read end to end.

---

## search\_sboms

**Purpose:** Search the PBI SBOM™ entries (loaded module list) for specific library or package names.

**When to use:**

- “Is library X in the model?”
- “Was package Y loaded when this model was captured?”
- Checking for presence of a specific dependency (security review, license audit, unexpected package)

| Parameter  | Type   | Required | Description                                                                                       |
|------------|--------|----------|---------------------------------------------------------------------------------------------------|
| model_path | string | yes      | Absolute host path to the model file                                                              |
| query      | string | no       | Search term (package name, partial name, or path fragment).<br>Omit to list all PBI SBOM™ entries |
| page       | u32    | yes      | Page number for paginated results                                                                 |

**Key output to extract:**

- Matching PBI SBOM™ entries (module paths, names)
- Whether the package is present at all
- Multiple matches may indicate different versions or sub-packages

**Interpretation tips:**

- A package in the PBI SBOM™ means it was loaded during the training run, not necessarily that it is always loaded
- Absence from the PBI SBOM™ means it was never loaded during the captured period, not necessarily absent from the dependency tree

## search\_verifiers

**Purpose:** Search verifiers (behavioral monitors) by keyword, method names, module names, or event types.

**When to use:**

- “Does my app make SQL queries?”
- “What network calls does it make?”
- “Does it use the crypto module?”
- Investigating a specific method, class, or module’s behavior patterns

| Parameter               | Type   | Required | Description                                                                                                                                                                                  |
|-------------------------|--------|----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>model_path</code> | string | yes      | Absolute host path to the model file                                                                                                                                                         |
| <code>query</code>      | string | no       | Search term (method name, module, or path fragment). Omit to list all verifiers                                                                                                              |
| <code>event_type</code> | string | no       | Filter by event type: <code>EVENT_STACK</code> , <code>EVENT_FILEPATH</code> , <code>EVENT_NETWORK</code> , <code>EVENT_SQL</code> , <code>EVENT_PROCESS</code> . Invalid values are ignored |
| <code>page</code>       | u32    | yes      | Page number for paginated results                                                                                                                                                            |

**Key output to extract:**

- Matching verifier entries: event type, monitored method/path, call context
- For `EVENT_FILEPATH` verifiers: the exact file paths monitored
- For `EVENT_SQL` verifiers: the normalized SQL query templates monitored

**Useful query patterns:**

| Goal                                | query                                    | event_type                  |
|-------------------------------------|------------------------------------------|-----------------------------|
| List all file access verifiers      | <i>(omit)</i>                            | <code>EVENT_FILEPATH</code> |
| List all network verifiers          | <i>(omit)</i>                            | <code>EVENT_NETWORK</code>  |
| List all SQL verifiers              | <i>(omit)</i>                            | <code>EVENT_SQL</code>      |
| Find crypto-related methods         | <code>crypto</code> or <code>jose</code> | <i>(omit)</i>               |
| Find AWS-related methods            | <code>aws</code> or <code>s3</code>      | <i>(omit)</i>               |
| Find specific method                | exact or partial name                    | <i>(omit)</i>               |
| Find file access to a specific path | <code>/etc</code> or <code>.env</code>   | <code>EVENT_FILEPATH</code> |

## compare\_models

**Purpose:** Diff two model files, shows what was added or changed between them. Models can also come from unrelated applications.

**When to use:**

- “What changed between two training runs?”
- “Is this new model safe to deploy?”

- Validating that a model update captured the expected new behaviors without introducing unexpected ones
- Investigating a model regression

| Parameter                   | Type   | Description                                       |
|-----------------------------|--------|---------------------------------------------------|
| <code>model_one_path</code> | string | Absolute host path to the first (reference) model |
| <code>model_two_path</code> | string | Absolute path to the second (new) model           |

#### Key output to extract:

- Verifiers present/absent between models (indicates differences in observed behavior)
- PBI SBOM™ differences: new packages loaded, packages no longer loaded

#### Interpretation tips:

- Many new EVENT\_STACK verifiers after a code deploy = expected and healthy
- New EVENT\_FILEPATH or EVENT\_SQL verifiers = worth reviewing specifically, these indicate new file or database access patterns
- Removed verifiers = code paths no longer exercised during training (could be intentional, could be coverage gap)

### verify\_model

**Purpose:** Check whether a recorded event session (.dbg file) is consistent with a model. Reports any events in the .dbg that the model would flag as violations.

#### When to use:

- “Is this capture from a known-good run?”
- Validating that a test run or production trace is within the model
- Incident investigation: does this suspicious traffic match or deviate from the model?

| Parameter               | Type   | Description                                    |
|-------------------------|--------|------------------------------------------------|
| <code>model_path</code> | string | Absolute host path to the model file           |
| <code>dbg_path</code>   | string | Absolute path to the .dbg event recording file |

#### Key output to extract:

- Number of events in the capture
- Details of violating events: what method, what stacktrace, what parameters

#### Interpretation tips:

- Zero violations = capture is fully within the model
- Violations against the model = unrecognized behavior observed in the capture (could be due to insufficient model training or could indicate a real dangerous behavior)

### simulate\_model\_from\_dbg

**Purpose:** Build a new model file by replaying events from a .dbg recording. The resulting model represents exactly the behaviors captured in that debug session.

#### When to use:

- “Turn this debug capture into a model I can use”
- Creating a model from a known-good recorded session without running a full training cycle
- Generating a reference model from a specific scenario (e.g. a load test, a specific user journey)

| Parameter         | Type   | Required | Description                                                  |
|-------------------|--------|----------|--------------------------------------------------------------|
| dbg_path          | string | yes      | Absolute path to the .dbg event recording file               |
| output_model_path | string | yes      | Absolute path where the generated model file will be written |

#### Key output to extract:

- **model\_path**: path of the generated model file
- **total\_events**: total number of events replayed from the .dbg
- **originally\_verified\_events**: number of events that were already verified at .dbg capture time against whatever model BEE was loaded then

**Note:** This tool does **not** take an existing model as input, it builds a new model from scratch using the .dbg as the event source. To check whether a .dbg matches an existing model, use **verify\_model** instead.

### analyze\_jar

**Purpose:** Analyze a Java JAR file against a Java model, identifies which methods in the JAR are and are not covered by the model.

#### When to use:

- Java applications only
- “Does the model cover my new JAR version?”
- “What methods in this JAR are not in the model?”
- Estimating coverage gaps before switching to enforcement mode

| Parameter  | Type   | Description                          |
|------------|--------|--------------------------------------|
| model_path | string | Absolute host path to the model file |
| jar_path   | string | Absolute path to the JAR file        |

#### Key output to extract:

- Methods in JAR covered by the model
- Methods in JAR not covered

### merge\_model

**Purpose:** Merge two or more model files into a single combined model. The merged model represents the union of all behaviors learned across the individual models.

#### When to use:

- Training was split across multiple runs (different code paths, different environments)
- Combining models from different application instances
- Building a comprehensive model from several partial training sessions

| Parameter                      | Type             | Description                                                |
|--------------------------------|------------------|------------------------------------------------------------|
| <code>model_paths</code>       | array of strings | List of absolute paths to model files to merge (minimum 2) |
| <code>output_model_path</code> | string           | Absolute path where the merged model file will be written  |

#### Key output to extract:

- Confirmation of merge success
- Output model path

**Note:** Merging models from different applications or very different training conditions is not recommended because the merged model may have a very broad model coverage and reduced enforcement effectiveness.

### `list_registered_events`

**Purpose:** List all method/function signatures that are registered in the model.

**When to use:**

- Getting a raw enumeration of every monitored event
- Checking if a specific method signature is registered
- Building a complete picture before a security audit

| Parameter               | Type   | Required | Description                          |
|-------------------------|--------|----------|--------------------------------------|
| <code>model_path</code> | string | yes      | Absolute host path to the model file |
| <code>page</code>       | u32    | yes      | Page number for paginated results    |

**Note:** This is a low-level enumeration. It is useful as an intermediate step when building a complete picture of a model, but rarely meaningful to read directly, `search_verifiers` and `summarize_entrypoints` give more interpretable views of the same underlying data.

### Common Error Scenarios

| Error                         | Likely cause                      | Suggested fix                                                                                 |
|-------------------------------|-----------------------------------|-----------------------------------------------------------------------------------------------|
| File not found                | Wrong or relative path            | Use valid absolute path                                                                       |
| Empty results from search     | Query term not present in model   | Try a broader or partial search term                                                          |
| <code>.dbg</code> parse error | Corrupted or truncated debug file | Check that the <code>.dbg</code> file was written completely (BEE must be stopped gracefully) |

## Reading and Auditing a Model

How to judge what a model contains and what its differences from another model mean. Vocabulary is defined in `concepts.md`; the tools that produce this data are in `tool-reference.md`.

## Classification Framework

Every difference or new behavior found should be assigned one of these four classes, with a stated reason. The class is a judgement, not a label to assert.

| Class                | Meaning                                                   | Typical signals                                                                                                                                                                                                |
|----------------------|-----------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>COVERAGE_GAP</b>  | Legitimate code path not exercised during training        | Stackframes in known app modules; no suspicious params; no new PBI SBOM™ entries                                                                                                                               |
| <b>NEW_BEHAVIOR</b>  | Legitimate new feature or dependency added since training | New PBI SBOM™ entries matching the expected tech stack; new verifiers in recognizable app code                                                                                                                 |
| <b>SECURITY_RISK</b> | Behavioral change that exposes a potential vulnerability  | Unexpected credential file access; novel SQL shape reachable from user input; new crypto usage in unexpected call path                                                                                         |
| <b>LIKELY_ATTACK</b> | Strong indicators of active exploitation                  | Injection patterns in <code>EVENT_SQL</code> ; outbound <code>EVENT_NETWORK</code> to unknown hosts from unexpected stack; <code>EVENT_PROCESS</code> spawning shells; PBI SBOM™ entries for offensive tooling |

A finding that sits between `SECURITY_RISK` and `LIKELY_ATTACK` is classified as **LIKELY\_ATTACK**, with the uncertainty stated explicitly.

## Attack Indicator Heuristics

Patterns worth actively looking for when classifying differences.

### EVENT\_SQL

- New param shape contains UNION, UNION ALL, DROP, or stacked statements absent from the model
- Param appears in a stacktrace that passes through HTTP request-parsing or user-input-handling code
- SQL pattern structurally longer or significantly different from trained variants

### EVENT\_FILEPATH

- New paths like: `~/.aws/credentials`, `~/.ssh/`, `.env`, `application.properties`, `/etc/passwd`, `/proc/`, `/etc/shadow`
- New paths inside `/tmp` or world-writable directories with executable-like names
- File access from a stacktrace not associated with app startup or config-loading code

### EVENT\_NETWORK

- New outbound connections from stacktraces not associated with known HTTP client libraries in the model
- Network calls from code paths that handle user input
- Connections to destinations not present in the model's network verifiers

### EVENT\_PROCESS

- Any new subprocess spawn is high priority
- Shell spawning (`/bin/sh`, `cmd.exe`, `powershell`, `bash`) from unexpected code paths

## SBOM

- New PBI SBOM™ entries matching known offensive tooling categories
- Build/package toolchain libraries (e.g., `pip`, `maven`, `npm` internals) appearing in a production protection model
- Libraries that do not match the application's stated domain

## Interpreting a single model

### Healthy model signals

- Verifier count is proportional to application complexity (hundreds to low thousands for typical microservices)
- PBI SBOM™ entries match the expected tech stack for the application
- `EVENT_SQL` does not contain suspicious queries and looks consistent with the app's function
- `EVENT_FILEPATH` parameters do not contain suspicious paths and file access patterns look reasonable
- `EVENT_NETWORK` looks consistent with expected network activities
- `EVENT_PROCESS` missing or can be explained as legitimate process spawning

### Signals worth highlighting

- **Sensitive file path access** in filepath verifiers: `.env` files, credential files (`~/.aws/credentials`, `application.properties`, `appsettings.json`), normal at startup but worth naming explicitly
- **Auth/crypto libraries** in PBI SBOM™: any JWT, OAuth, password hashing, or signing library, all sensitive behavior that should be stable across deployments
- **Unexpected libraries** in PBI SBOM™: packages that don't belong to the expected application domain (crypto miners, unusual network tools, build toolchain packages in a production model)
- **Very sparse model** (under a few hundred verifiers for a non-trivial app): training coverage was probably insufficient
- **Every process spawn** should be carefully investigated
- **Every network connection attempt** should be carefully investigated

### Security-relevant PBI SBOM™ categories

What matters is the *category* of behavior, matched against what's present in the PBI SBOM™, rather than any specific package name. Below are a few examples.

| Category                  | Node.js examples                                                                           | Java examples                                                    | Python examples                                                         |
|---------------------------|--------------------------------------------------------------------------------------------|------------------------------------------------------------------|-------------------------------------------------------------------------|
| Auth / identity           | <code>passport</code> , <code>jsonwebtoken</code> , <code>jose</code> , <code>auth0</code> | <code>io.jsonwebtoken:jjwt</code> , <code>spring-security</code> | <code>PyJWT</code> , <code>authlib</code>                               |
| Password hashing          | <code>bcrypt</code> , <code>argon2</code>                                                  | <code>spring-security.crypto</code> , <code>argon2-jvm</code>    | <code>passlib</code> , <code>bcrypt</code>                              |
| Crypto primitives         | <code>node:crypto</code> , <code>jwa</code> , <code>jws</code>                             | <code>org.bouncycastle</code> , <code>javax.crypto</code>        | <code>cryptography</code> , <code>pycryptodome</code>                   |
| Secrets / config          | <code>dotenv</code>                                                                        | <code>spring-cloud-config</code> , <code>snakeyaml</code>        | <code>python-dotenv</code> , <code>dynaconf</code>                      |
| Cloud SDKs                | <code>@aws-sdk/*</code> , <code>@google-cloud/*</code> , <code>@azure/*</code>             | <code>software.amazon.awssdk</code> , <code>com.azure</code>     | <code>boto3</code> , <code>google-cloud-*</code> , <code>azure-*</code> |
| HTTP / network clients    | <code>axios</code> , <code>undici</code> , <code>node-fetch</code> , <code>got</code>      | <code>okhttp3</code> , <code>java.net.http</code>                | <code>requests</code> , <code>httpx</code> , <code>aiohttp</code>       |
| Build / package toolchain | <code>sigstore</code> , <code>tuf-js</code>                                                | <code>org.apache.maven</code> , <code>org.gradle</code>          | <code>pip</code> , <code>setuptools</code>                              |

## Remediation Guidance

| Class         | Action                                                                                                                                      |
|---------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| COVERAGE_GAP  | Identify the missing test scenario; add it to the test suite and retrain                                                                    |
| NEW_BEHAVIOR  | Confirm the change is intentional with the team; if legitimate, retrain to incorporate it                                                   |
| SECURITY_RISK | Trace the violating stackframe in source code; open a security ticket; do not promote to production until resolved                          |
| LIKELY_ATTACK | <b>Do not retrain.</b> Preserve both models as forensic evidence. Isolate the affected instance. Escalate to the security team immediately. |

Under `PBI_LEVEL=AUTO`, the Training model (`csmodel.N`) was learned from live traffic, not a controlled training run. Audit it like any other model instead of treating it as a trusted reference, because behavior you find in a protection model may already be in it.

Retraining on a model that captured an attack permanently poisons the model: the attack behavior becomes “normal” and will never be flagged again.

## Troubleshooting

Runbook for PBI deployments that fail to start, fail to protect, or flag behavior you believe is legitimate. Configuration itself is documented in `deployment-guide.md`.

### Reading the logs

BEE and the Protection Agent both log to `stdout`, so container logs are the first place to look:

```
podman logs <container>
kubectl logs <pod> -c bee
```

BEE also writes a log file, defaulting to `/storage/logs/pbi.log`. Every entry carries one of three levels:

| Level   | Meaning                                                                                                                                                                          |
|---------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| INFO    | Routine. In training mode, this includes BEE intercepting an event that isn't yet in the reference model, that's the training process working.                                   |
| WARNING | BEE startup and shutdown messages, model saves, and the <code>AUTO</code> switch from training to protection.                                                                    |
| SEVERE  | <b>In protection mode, BEE intercepted an event that isn't in the reference model.</b> Each one is either genuine behavior outside the model or a gap in your training coverage. |

The log file path is a fixed default: it does **not** follow `PBI_STORAGE_PATH`. If you point storage elsewhere, the log still goes to `/storage/logs/` unless you also pass `--log <path>`.

## BEE exits with “Failed to initialize logger”

**Symptom:** BEE prints `Failed to initialize logger` on stdout and exits immediately.

**Cause:** The log file’s parent directory doesn’t exist. BEE opens the log *file* for appending but never creates the directory holding it (default `/storage/logs/`), or the parent of whatever `--log` points at.

**Fix:** Create the directory before BEE starts.

## BEE model fails to load because its directory doesn’t exist

**Symptom:** BEE logs `PBI™ BEE model <path> loading failed`. The monitored process then aborts with:

`BEE Proxy initialization Failure: Kind(InvalidFilename)`

**Cause:** `PBI_MODEL_PATH` is a path relative to `PBI_STORAGE_PATH` (default `/storage`). BEE never creates missing parent directories, so `PBI_MODEL_PATH=models/csmode` fails to open when `${PBI_STORAGE_PATH}/models/` doesn’t exist in the container. The model load is aborted, BEE never confirms readiness over the ring buffer, and the app’s Protection Agent times out waiting for the handshake.

**Fix:** Create `${PBI_STORAGE_PATH}/<directory portion of PBI_MODEL_PATH>` before BEE starts.

## Shared-memory handshake fails

**Symptom:** BEE and the app don’t connect over shared memory. This shows up one of two ways:

- BEE fails to start while setting up shared memory, or
- the **app** prints the Protection Agent’s startup messages but never reaches `Native RingBuffer initialized`, stopping after `Waiting for "/dev/shm/cspbi"`.

A healthy app startup looks like:

```
Starting Java Protection Agent 1.0
JVM TI agent done
[Instant { ... }] Waiting for "/dev/shm/cspbi"
[Instant { ... }] "/dev/shm/cspbi" ready at XXXXXXXX
Native RingBuffer initialized
```

**Cause:** BEE and the app aren’t sharing the same `/dev/shm/cspbi`, most often a user-ID mismatch, a leftover file from a previous run, or the two containers not sharing `/dev/shm`.

**Fix:** Work through these in order:

- Run BEE and the app under the **same effective user ID**. If they differ, pin BEE’s UID in the pod spec:
 

```
securityContext:
  runAsUser: <uid>
```
- Verify `/dev/shm/cspbi` exists in **both** the app and BEE containers.
- Check the `/dev/shm` filesystem size, it must be slightly over 1 MB.
- Remove a stale `/dev/shm/cspbi` before restarting; in containers a pod restart gives a fresh `/dev/shm`.

## Protection Agent never loads (no error at all)

**Symptom:** No agent startup messages appear, the app runs unprotected, and nothing reports an error.

**Cause:** The wrong libc variant. A glibc/musl mismatch produces no error; the agent simply never loads.

**Fix:** Re-check whether the application image is glibc or musl and use the matching agent path, the check is in `deployment-guide.md`.

## Legitimate behavior flagged in protection mode

**Symptom:** BEE logs **SEVERE** entries for behavior you know is valid, it simply wasn't exercised during training.

**Cause:** The behavior isn't in the reference model. BEE wrote it into the model on shutdown, but that model carries the `.p` suffix (`csmodel.p.N`), marking it a *Protection* model. BEE only loads *Training* models (`csmodel.N`) as its reference model at startup, so a `.p` model is never picked up on its own. See `concepts.md` for the naming convention.

**Fix:** Fold the behavior back into your model one of two ways:

- **Rename** the file to drop the `.p` suffix, making it the most recent Training model.
- **Merge** it with your existing model using PBI Explainer's `merge_model` tool, writing the result to a filename without the `.p` suffix. See `tool-reference.md`.

Before doing either, confirm the flagged behavior really is legitimate, retraining on a model that captured an attack permanently poisons the model. Inspect the model with PBI Explainer to make that call `explainer.md`.

## AUTO never switches to protection

**Symptom:** BEE started with `PBI_LEVEL=AUTO` (its log shows `loaded with tolerance: AUTO`), but `PBI™ BEE auto level TRAINING -> LOG` never appears, and events outside the model keep being logged at **INFO**.

**Cause:** One of:

- Events outside the model still arrive more often than the no-violation window allows. Each one restarts the window. The `failures` counters on `/metrics` keep growing when this is the case.
- The application is idle. The switch is only evaluated when an event matches the model, so without traffic BEE stays at **TRAINING** however long it runs.

**Fix:** Keep the application under representative load. If `failures` keeps growing, training is still finding new behavior and the switch follows once it stops.