AppSentinels
Business Logic Security

AppSentinels API Security Platform

Python Watcher Integration

Revision History

Revision	Date Modified	Author	Comments
1.0	10-Apr-2026	Sachin	Initial spec for Python agent integration
1.1	20-May-2026	Sachin	Support for OTEL for FASTAPI

Proprietary and confidential Information, ©2026, AppSentinels.ai

AppSentinels Python Watcher

The AppSentinels Python Watcher provides agent-based instrumentation using the trusted Hypertrace agent to enable API visibility and monitoring for Python applications. The approach leverages OpenTelemetry instrumentors bundled with the Hypertrace agent to automatically capture request and response data for every supported framework, with a one-time initialization in your application's entrypoint module.

> **ℹ Note**
>
> The Python runtime does not support transparent bytecode injection, so a small, two-line code change in your application's entrypoint module is required to initialize the Watcher. This is a one-time change per service and does not need to be repeated when framework versions or routes change.

Architecture Diagram

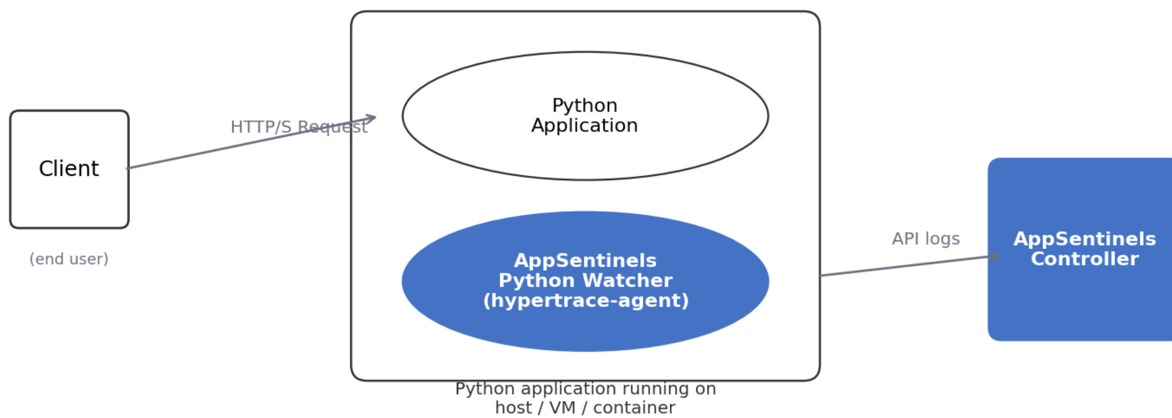

Figure 1 — AppSentinels Python Watcher topology

AppSentinels Watcher for Python Application

The AppSentinels Python Watcher instruments the application automatically to capture the API information below:

- Request and response headers
- Request and response bodies
- HTTP method, URI, status code, and latency

Host / VM Environment

The Python Watcher is delivered as a pip-installable package. Installation is identical on Linux and Windows and does not require root or administrator privileges if installed into a virtual environment alongside your application.

 Proprietary and confidential Information, ©2026, AppSentinels.ai

How to Install

Add the auto-instrumentation agent to your application's dependency file:

```
# requirements.txt
hypertrace-agent
```

Install:

```
pip install -r requirements.txt
```

This pulls in the Hypertrace agent and the full set of OpenTelemetry instrumentors as transitive dependencies — you do not need to list the individual instrumentation packages.

How to Configure

The Watcher is configured through a YAML file. Place it alongside your application (for example at /opt/appsentinels-watcher/config.yaml) and point the HT_CONFIG_FILE environment variable at it.

```
# /opt/appsentinels-watcher/config.yaml
service_name: appsentinels-watcher
enabled: true                          # Enable/disable the agent

# Reporting Configuration
reporting:
  # Please make it https instead of http in case secure connection
  # to controller
  endpoint: http://<Controller IP or FQDN>:<Port Number>
  protocol: grpc
  metricReporterType: None
  # HTTPS based config. Disable below 2 lines in case of http based
  # connection to controller
  #secure: true
  #cert_file: <Path of CA cert file which will validate the server certificate in case of
self signed cert>

# Data Capture Configuration
dataCapture:
  bodyMaxSizeBytes: 131072             # 128 KB - max size of request/response bodies to
capture
  httpHeaders:
    request: true                      # Enable/disable request headers capture
    response: true                     # Enable/disable response headers capture
  httpBody:
    request: true                      # Enable/disable request body capture
    response: true                     # Enable/disable response body capture
  allowedContentTypes:
    - application/json
    - application/xml
    - text/json
    - text/xml
    - application/x-www-form-urlencoded
    - application/graphql
```

Export the configuration path:

 Proprietary and confidential Information, ©2026, AppSentinels.ai

```
export HT_CONFIG_FILE=/opt/appsentinels-watcher/config.yaml
```

How to Activate

Activation requires two steps: a two-line initialization in your application's entrypoint module, and setting the HT_SERVICE_NAME environment variable before the process starts.

Add the following to the top of your entrypoint module (e.g. app.py for Flask/FastAPI, wsgi.py for Django). The Agent() + instrument() calls must run before any framework imports so that the instrumentors can monkey-patch them.

Flask / FastAPI — in app.py

```
from hypertrace.agent import Agent

agent = Agent()
agent.instrument()           # must run before framework imports

from flask import Flask      # or:  from fastapi import FastAPI
app = Flask(__name__)
```

Django — in wsgi.py

Django requires a slightly different placement. DJANGO_SETTINGS_MODULE must be set before agent.instrument() is called, and the agent must be instrumented before get_wsgi_application() is invoked:

```
import os
os.environ.setdefault("DJANGO_SETTINGS_MODULE", "settings")

from hypertrace.agent import Agent
agent = Agent()
agent.instrument()

from django.core.wsgi import get_wsgi_application
application = get_wsgi_application()
```

Set the service name. This value identifies your service in the AppSentinels Controller and must be unique per service:

```
export HT_SERVICE_NAME=<your-service-name>
```

Restart your application in a new shell. The Watcher will automatically instrument incoming requests, outbound HTTP/gRPC/DB calls, and forward the data to the AppSentinels Controller.

> ⚠ **Important**
>
> The HT_SERVICE_NAME environment variable is required even if you have set serviceName in config.yaml. Due to a known behaviour of the Hypertrace agent, the YAML key is not propagated to the service.name attribute on exported spans; the environment variable is. Without it, your service will appear in the Controller as "pythonagent".

 Proprietary and confidential Information, ©2026, AppSentinels.ai

How to Verify

At startup, the agent prints two lines you should see in your application's logs:

```
Config init complete - config state: {... 'service_name': '<your-name>'... }
Initialized otlp exporter reporting to `<controller-host>:9009`
```

Issue a request to your application and confirm the corresponding trace appears in the AppSentinels Controller dashboard within a few seconds. If no traces appear, see the Reference section for common failure modes and how to diagnose them.

How to Uninstall

- Remove hypertrace-agent from requirements.txt and re-run pip install -r requirements.txt, or run pip uninstall hypertrace-agent directly.
- Remove the two-line Agent() + agent.instrument() block from your entrypoint module.
- Remove the HT_CONFIG_FILE and HT_SERVICE_NAME environment variables from your process manager / systemd unit / container manifest.
- Restart the application. No framework code or route changes need to be reverted — the instrumentation is purely additive.

AppSentinels Watcher for Container-based Application

For containerized deployments (Docker, Kubernetes, AWS ECS, etc.), the Watcher is activated the same way as on a VM: the hypertrace-agent package is installed as part of the container image, the two-line initialization is added to the entrypoint module, and configuration is provided through environment variables or a mounted config file.

Dockerfile example

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY app.py .
CMD ["gunicorn", "--bind=0.0.0.0:8080", "app:app"]
```

docker-compose.yaml fragment

```
services:
  my-python-service:
    build: .
    environment:
      HT_CONFIG_FILE: /app/config.yaml
      HT_SERVICE_NAME: my-python-service
    volumes:
      - ./config.yaml:/app/config.yaml:ro
```

 Proprietary and confidential Information, ©2026, AppSentinels.ai

Kubernetes — ConfigMap-based configuration

On Kubernetes, ship the same config.yaml file via a ConfigMap and mount it into the application pod at the path referenced by HT_CONFIG_FILE. This keeps the configuration identical to the VM/host deployment.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: appsentinels-watcher-config
data:
  config.yaml: |
    service_name: appsentinels-watcher
    enabled: true
    reporting:
      endpoint: http://<Controller IP or FQDN>:<Port Number>
      protocol: grpc
      metricReporterType: None
    dataCapture:
      bodyMaxSizeBytes: 131072
      httpHeaders:  { request: true, response: true }
      httpBody:     { request: true, response: true }
      allowedContentTypes:
        - application/json
        - application/xml
        - text/json
        - text/xml
        - application/x-www-form-urlencoded
        - application/graphql
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-python-service
spec:
  template:
    spec:
      containers:
        - name: app
          image: my-python-service:latest
          env:
            - name: HT_CONFIG_FILE
              value: /etc/appsentinels/config.yaml
            - name: HT_SERVICE_NAME
              value: my-python-service
          volumeMounts:
            - name: appsentinels-config
              mountPath: /etc/appsentinels
              readOnly: true
      volumes:
        - name: appsentinels-config
          configMap:
            name: appsentinels-watcher-config
```

 Proprietary and confidential Information, ©2026, AppSentinels.ai

ℹ Note

The Python Watcher ships inside the application container image as a normal pip dependency — no sidecar or init container is required. Container orchestrators only need to mount the config file and set HT_CONFIG_FILE and HT_SERVICE_NAME.

OpenTelemetry Integration

Python

When Is This Needed?

Use this integration method when your Python application cannot use the standard hypertrace-agent due to a dependency conflict with other packages in your environment. This method exports traces over HTTP/JSON, removing the dependency conflict on the Python side entirely while providing the same request and response visibility in the AppSentinels Controller.

Step 1 — Update requirements.txt

Do not add opentelemetry-exporter-otlp — it can conflict with other packages in your environment. Add only the following three packages to your existing requirements.txt, keeping all other package versions unchanged:

```
opentelemetry-api
opentelemetry-sdk
opentelemetry-instrumentation-fastapi
```

Step 2 — Download the AppSentinels JSON Exporter

Download appsentinels_json_exporter.py from the URL below and place it in the same directory as your app.py:

```
https://downloads.appsentinels.ai/appsentinels-deployment/Python-
OTEL/appsentinels_json_exporter.py
```

This file contains the OTLP/HTTP JSON exporter, body and header capture middleware, and the tracing initialisation function. No additional dependencies are required.

Step 3 — Instrument Your FastAPI Application

Add the AppSentinels sections shown below into your app.py. All AppSentinels-specific lines are marked with ### AppSentinels logging comments so they are easy to locate and remove if needed.

Important: app.add_middleware(BodyCaptureMiddleware) must be called before FastAPIInstrumentor().instrument_app(app). If the order is reversed, the middleware runs outside the active span and no request/response attributes are captured.

```
from fastapi import FastAPI, Request

### AppSentinels logging - Section Start
from opentelemetry.instrumentation.fastapi import FastAPIInstrumentor
from appsentinels_json_exporter import appsentinels_init_tracing,
```

 Proprietary and confidential Information, ©2026, AppSentinels.ai

```
BodyCaptureMiddleware

appsentinels_init_tracing()
### AppSentinels logging - Section End

app = FastAPI()

### AppSentinels logging - Section Start
app.add_middleware(BodyCaptureMiddleware)
FastAPIInstrumentor().instrument_app(app)
### AppSentinels logging - Section End

# ... your existing routes below, no changes needed ...
```

Step 4 — Set Environment Variables

Set the following environment variables before starting your application:

```
export OTEL_EXPORTER_OTLP_ENDPOINT=http://<controller-host>:9009/v1/traces
export OTEL_SERVICE_NAME=<your-service-name>
```

OTEL_EXPORTER_OTLP_ENDPOINT — The AppSentinels controller address. Replace <controller-host> with the IP or hostname of your controller. The path /v1/traces must be included.

OTEL_SERVICE_NAME — The name used to identify your service in the AppSentinels Controller. Appears as the X-As-Sensor-Instance value in logged transactions. Use a unique value per service.

Step 5 — Verify

Send a request to any endpoint, then run the following command on the AppSentinels controller host to view the last transaction sent to the cloud:

```
dp-commands.sh dp-last-saas-log | jq .
```

In the output, confirm:

- HTTPReq.Uri is set to your application route (e.g. /api/orders), not /v1/traces
- HTTPReq.Body and HTTPResp.Body are populated for JSON requests
- Request and response headers are present (Req.Header.* and Resp.Header.*)
- Resp.Header.extra contains X-As-Sensor-Instance set to your OTEL_SERVICE_NAME value

Reference

The items below describe known behaviours of the underlying Hypertrace agent that customers have hit during initial rollout. Each entry lists the symptom and the resolution.

1. Agent initializes, logs look healthy, but no traces reach the Controller

The agent does not install a real OpenTelemetry TracerProvider unless agent.instrument() is called. If you see a ProxyTracerProvider when inspecting the SDK, every span is silently dropped.

Verify with:

 Proprietary and confidential Information, ©2026, AppSentinels.ai

```
python -c "from opentelemetry import trace;
print(type(trace.get_tracer_provider()).__name__)"
# Expected: TracerProvider
# If you see: ProxyTracerProvider  — agent.instrument() is missing or ran after framework
imports.
```

Resolution: ensure Agent() and agent.instrument() are the first executable statements in your entrypoint module, before any framework import.

2. Service appears in the Controller as "pythonagent"

The YAML configuration key serviceName is loaded into the agent's internal config but is not propagated to the OTel service.name resource attribute on exported spans.

Resolution: set the HT_SERVICE_NAME environment variable in the process environment. This value overrides the internal default and is carried through to the exported spans.

```
export HT_SERVICE_NAME=<your-service-name>
```

Note: OTEL_SERVICE_NAME (the OpenTelemetry standard variable) is not honoured by the Hypertrace agent; HT_SERVICE_NAME must be used.

3. Agent initializes but no HTTP spans are produced for a specific framework

The bundled instrumentor for that framework has declined to patch due to a version mismatch. Look for a DependencyConflict line in the application's startup logs:

```
DependencyConflict: requested "<framework> >= <min>, < <max>" but found "<framework> <too-
new>"
```

Resolution: pin the framework in requirements.txt to a version inside the supported range. Find the range with:

```
pip show opentelemetry-instrumentation-<framework> | grep Requires-Dist
```

4. Confirming the agent is shipping traces on the wire

From the host where the AppSentinels Controller listens, run a packet capture while issuing a request to your application:

```
sudo tcpdump -i any -nn 'tcp and port 9009'
```

A successful export appears as an outbound TCP PUSH from the application container's IP to the Controller on port 9009, with a payload of a few KB. If no such traffic is visible, the agent is either not producing spans (see item 1) or is pointing at the wrong endpoint (verify the reporting.endpoint setting).

 Proprietary and confidential Information, ©2026, AppSentinels.ai