Enable on-prem healthchecks

how can we enable API Healthchecks for services deployed in on-prem VM environment. Code used is python and python libraries like FASTAPI.

can we use python libraries that enable and expose endpoints on python server for healthchecks. Please go into details and give examples on how can we enable on-prem deployed healthchecks?

Excellent question — this is a very common and important concern when deploying **Python-based microservices** (like those using **FastAPI**) in **on-prem environments** (e.g., VMs, bare metal, or private datacenters).

Let's go step-by-step and look at **how to implement health checks**, **how to expose them via HTTP endpoints**, and **how to make them operational in on-prem infrastructure**.

🧠 1. Why Health Checks Are Needed

A **health check** lets orchestration systems, load balancers, and monitoring tools determine whether your service is running correctly.

Typical uses:

- Load balancers (NGINX, HAProxy, F5) use `/health` endpoints to decide if a backend VM is healthy.
- Monitoring tools (Prometheus, Zabbix, Datadog) poll these endpoints to generate alerts.
- CI/CD pipelines use them to verify post-deployment status.

🧩 2. Health Check Types

You'll usually want two separate endpoints:

Type	Purpose	Typical Endpoint	Example Check
Liveness	Is the app running?	`/live`	Always returns 200 unless the process is dead or frozen
Readiness	Is the app ready to serve requests?	`/ready`	Checks DB connections, external APIs, etc.

⚙️ 3. Using FastAPI's Built-in Capability

FastAPI doesn't have built-in "healthcheck" middleware like some frameworks, but it's easy to implement using regular routes.

Example: Simple Liveness and Readiness

```
python

from fastapi import FastAPI
import asyncio
import asyncpg  # Example: PostgreSQL connection check

app = FastAPI()

# Example: Database pool
db_pool = None

@app.on_event("startup")
async def startup_event():
    global db_pool
    db_pool = await asyncpg.create_pool(
        user="myuser",
        password="mypassword",
        database="mydb",
        host="127.0.0.1"
    )

@app.on_event("shutdown")
async def shutdown_event():
    await db_pool.close()

@app.get("/health/live")
async def live():
    """Liveness probe — is the app running?"""
    return {"status": "alive"}

@app.get("/health/ready")
async def ready():
    """Readiness probe — can we connect to dependencies?"""
    try:
        async with db_pool.acquire() as conn:
            await conn.execute("SELECT 1")
        return {"status": "ready"}
    except Exception as e:
        return {"status": "unready", "error": str(e)}
```

✅ Result:

- **`/health/live`** → Returns **`{"status": "alive"}`** if process is running.
- **`/health/ready`** → Returns **`{"status": "ready"}`** only if DB and dependencies are available.

🧰 4. Using a Dedicated Library (Optional)

You can use the library **`fastapi-health`**↗ to make this cleaner.

Install:

```
bash

pip install fastapi-health
```

Example:

```python
from fastapi import FastAPI
from fastapi_health import health

app = FastAPI()

async def is_database_online():
    # You can implement a real DB ping here
    return True

app.add_api_route("/health", health([is_database_online]))
```

This automatically returns:

- `{"healthy": true}` with HTTP 200
- Or `{"healthy": false}` with HTTP 503 if any check fails

🏗️ 5. Exposing Health Checks On-Prem

Unlike Kubernetes (which automatically scrapes `/health`), **on-prem** environments require **manual configuration**:

a) Via Load Balancer

If you use **NGINX**, **HAProxy**, or **F5**, configure a health check like:

```nginx
location /health {
    proxy_pass http://127.0.0.1:8000/health/live;
}
```

and in your backend configuration:

```nginx
server 127.0.0.1:8000;
health_check uri=/health/ready interval=5s fails=3 passes=1;
```

b) Via Monitoring Agent

If you have a monitoring tool running on the VM (e.g., **Prometheus node exporter**, **Zabbix agent**, or **custom cron script**):

- Use a simple HTTP GET to `/health/ready`
- Raise alerts if the status code != 200

Example with `curl`:

```bash
curl -f http://localhost:8000/health/ready || echo "Service unhealthy"
```

This can be scheduled via cron or integrated into your monitoring platform.

🧩 6. Example Architecture

```bash
       ┌──────────────────┐
       │  Client Requests │
       └────────┬─────────┘
                │
         ┌──────▼──────┐
         │LoadBalancer │ ───▶ Performs health checks (/health/ready)
         └──────┬──────┘
                │
           ┌────▼────┐
           │ FastAPI App (VM) │ ───▶ /health/live, /health/ready
           └─────────┘
```

🧠 7. Advanced Example — Including External Dependencies

If your app depends on multiple services (like DB, Redis, third-party APIs), you can create a health aggregator:

```python
from fastapi import FastAPI
import aiohttp, asyncpg, aioredis

app = FastAPI()

async def check_db():
    try:
        conn = await asyncpg.connect("postgres://user:pass@localhost/db")
        await conn.execute("SELECT 1")
        await conn.close()
        return True
    except:
        return False

async def check_redis():
    try:
        redis = await aioredis.from_url("redis://localhost")
        pong = await redis.ping()
        await redis.close()
        return pong
    except:
        return False

async def check_external_api():
    try:
        async with aiohttp.ClientSession() as session:
            async with session.get("https://example.com/health") as resp:
                return resp.status == 200
    except:
        return False

@app.get("/health/ready")
async def ready():
    checks = {
        "database": await check_db(),
        "redis": await check_redis(),
        "external_api": await check_external_api(),
    }
    overall = all(checks.values())
    return {"status": "ready" if overall else "unready", "details": checks}
```

✅ 8. Summary — Recommended Setup for On-Prem FastAPI Healthchecks

Component	Recommendation
Library	Use **`fastapi-health`** or write custom **`/health`** routes
Endpoints	**`/health/live`**, **`/health/ready`**
Infra Integration	Load balancer health checks or monitoring probes hitting those endpoints
Return Codes	200 = Healthy, 503 = Unhealthy
On-Prem Deployment	Expose these ports internally only; don't open to the internet
Security	Optionally secure with IP allowlist or token if sensitive

---

Would you like me to show an example of **how to integrate this with an NGINX load balancer** for on-prem setup (so it automatically removes unhealthy VMs)?

yes show nginx load balancer for on-prem setup

Perfect — let's go step-by-step on how to integrate **FastAPI health checks** with an **NGINX load balancer** for an **on-prem deployment**.

This setup ensures that:

- Your NGINX load balancer only routes traffic to healthy FastAPI instances (running on separate VMs).
- NGINX automatically removes unhealthy instances until they recover.
- You can monitor or alert on health status using built-in or external tools.

🧩 1. Example Topology

Let's say you have:

Component	Hostname	IP	Service Port
VM1	app1.internal.local	10.0.0.11	8000
VM2	app2.internal.local	10.0.0.12	8000
NGINX LB	lb.internal.local	10.0.0.10	80

Each VM runs your **FastAPI app** (with **`/health/live`** and **`/health/ready`** endpoints).

⚙️ 2. NGINX Load Balancer Configuration

Below is a **full NGINX configuration** file example.

```nginx
# /etc/nginx/nginx.conf or /etc/nginx/conf.d/fastapi_lb.conf

upstream fastapi_backend {
    zone backend_zone 64k;

    # Define backend servers
    server 10.0.0.11:8000 max_fails=3 fail_timeout=10s;
    server 10.0.0.12:8000 max_fails=3 fail_timeout=10s;

    # Enable active health checks (NGINX Plus) or passive checks (open-source)
    # For NGINX OSS, passive health checks rely on failed requests.
}

server {
    listen 80;
    server_name lb.internal.local;

    # Load balancer health check route (optional)
    location /healthz {
        return 200 "Load balancer healthy\n";
    }

    # Health checks to backend servers (readiness endpoint)
    location /health {
        proxy_pass http://fastapi_backend/health/ready;
        proxy_set_header Host $host;
        proxy_connect_timeout 2s;
        proxy_read_timeout 2s;
    }

    # Application routes
    location / {
        proxy_pass http://fastapi_backend;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_connect_timeout 5s;
        proxy_read_timeout 10s;
    }
}
```

Explanation:

- **`max_fails=3 fail_timeout=10s`**: If a server fails 3 consecutive times, it's marked as "unhealthy" for 10 seconds.
- **`/health`** route calls each backend's **`/health/ready`**.
- Requests will automatically fail over to healthy nodes.

✅ **Result**:

- When **`/health/ready`** returns 503, NGINX stops routing traffic to that VM.
- When it recovers, it's added back to the pool automatically.

💎 3. If You're Using NGINX Plus

If you have **NGINX Plus**, you can use **active health checks** — NGINX itself periodically hits **`/health/ready`** on each backend.

```nginx
upstream fastapi_backend {
    zone backend_zone 64k;
    server 10.0.0.11:8000;
    server 10.0.0.12:8000;

    health_check uri=/health/ready interval=5s fails=3 passes=2;
}
```

This is **superior** because:

- It checks proactively.
- You get live metrics via the NGINX dashboard or API.
- No failed client requests are needed to mark a backend unhealthy.

🧪 4. Testing Locally

Run your two FastAPI instances (simulate VMs):

```bash
uvicorn app:app --host 0.0.0.0 --port 8000
uvicorn app:app --host 0.0.0.0 --port 8001
```

Then modify `nginx.conf`:

```nginx
upstream fastapi_backend {
    server 127.0.0.1:8000;
    server 127.0.0.1:8001;
}
```

Test:

```bash
curl http://localhost/health
curl http://localhost/
```

Stop one instance:

```bash
kill <pid_of_one_server>
```

Then re-test. You'll see NGINX automatically routes to the healthy one.

🧰 5. Optional — Add Monitoring and Alerting

You can easily extend this to monitor health statuses using:

- **Prometheus + Blackbox exporter** polling `/health/ready`
- **Zabbix agent** executing `curl -f http://localhost/health/ready`
- **Email/SMS alert** when unhealthy

Example cron-based local probe:

```bash
*/1 * * * * curl -fs http://localhost/health/ready || echo "$(date): FastAPI down" >>
/var/log/health_failures.log
```

🧠 6. Security & Hardening

For on-prem environments:

- **Restrict `/health`** endpoints to internal network or LB IPs:

```nginx
location /health {
    allow 10.0.0.0/24;
    deny all;
    proxy_pass http://fastapi_backend/health/ready;
}
```

- **Use HTTPS** between LB ↔ backend if sensitive.
- **Use authentication** if `/health` exposes sensitive system data.

✅ 7. Summary

Component	Description
FastAPI app	Exposes `/health/live` and `/health/ready`
NGINX Load Balancer	Uses passive (OSS) or active (Plus) health checks
Failover logic	Unhealthy VMs are removed automatically
Integration	Works seamlessly with existing on-prem infra
Security	Restrict access, use internal networks, or HTTPS

Would you like me to show how to **make this setup observable** (e.g., integrate `/health` results into Prometheus or Grafana dashboards for visualization and alerting)?

how can we use appdynamics product for above scenario around checking healthcheck of services

Excellent — bringing **AppDynamics** into this scenario is a smart move.
AppDynamics provides **deep application performance monitoring (APM)** and can easily be extended to monitor **health checks** for on-premise FastAPI services (and their infrastructure, like NGINX and databases).

Let's go step by step and cover exactly how to integrate **AppDynamics** with your FastAPI + NGINX healthcheck setup.

🧭 1. AppDynamics Overview in This Context

AppDynamics can monitor your system on **three layers**:

Layer	Component	What AppDynamics Does
Application Layer (FastAPI)	Python app code	Monitors transactions, exceptions, response time, and health
Infrastructure Layer (VMs)	On-prem servers	Tracks CPU, memory, network usage
Load Balancer Layer (NGINX)	Load balancer	Collects throughput, latency, and backend health metrics

---

🧩 2. AppDynamics Setup for Python / FastAPI

AppDynamics provides a **Python agent** that instruments your app and sends metrics to the AppDynamics Controller.

🧱 Step 1 — Install the Python Agent

On each VM running FastAPI:

```bash
pip install appdynamics
```

(Or, if using the more recent naming:)

```bash
pip install appdynamics-bindeps
```

Then, download your **AppDynamics configuration file (`controller-info.xml`)** from the Controller UI.

Place it in your project (for example, under `appd/`).

🧩 Step 2 — Initialize AppDynamics in Your FastAPI App

In your main FastAPI entrypoint (`app.py` or `main.py`):

```
python

from appdynamics.agent import agent
import os

# Initialize AppDynamics Agent
agent.init(os.path.join(os.path.dirname(__file__), "appd/controller-info.xml"))

from fastapi import FastAPI
import asyncpg

app = FastAPI()

@app.get("/health/live")
async def live():
    return {"status": "alive"}

@app.get("/health/ready")
async def ready():
    try:
        conn = await asyncpg.connect("postgres://user:pass@localhost/db")
        await conn.execute("SELECT 1")
        await conn.close()
        return {"status": "ready"}
    except Exception as e:
        return {"status": "unready", "error": str(e)}
```

✅ **Now AppDynamics monitors:**

- Response times for `/health/*` and other endpoints
- Database query performance
- Exceptions and failed health checks

⚙️ 3. AppDynamics Metrics to Watch for Health

Once the agent is running, AppDynamics automatically creates metrics like:

Metric	Meaning
`Business Transaction Response Time`	Latency of API endpoints, including `/health/ready`
`Errors per Minute`	Shows if `/health/ready` endpoints start failing
`Availability %`	Derived metric you can define to track uptime
`Custom Metric: Health Status`	You can create one manually — see below

🧠 4. Sending Custom Health Metrics to AppDynamics

You can enhance observability by **publishing custom metrics** to AppDynamics, derived directly from your FastAPI health logic.

Example:

```
python

from appdynamics.agent import api
from fastapi import FastAPI
import asyncpg
```

```
app = FastAPI()

@app.get("/health/ready")
async def ready():
    metric_path = "Custom Metrics|FastAPI|Health|Readiness"
    try:
        conn = await asyncpg.connect("postgres://user:pass@localhost/db")
        await conn.execute("SELECT 1")
        await conn.close()
        api.add_metric(metric_path, 1)  # 1 = healthy
        return {"status": "ready"}
    except Exception:
        api.add_metric(metric_path, 0)  # 0 = unhealthy
        return {"status": "unready"}
```

Now, in AppDynamics, you'll see a metric:

```vbnet
Custom Metrics | FastAPI | Health | Readiness
```

You can then:

- Create **dashboards** showing service health.
- Set **alerts** (Health Rules) for when this metric = 0.
- Combine multiple service checks into an overall **Application Health KPI**.

🧱 5. Integrating with NGINX Health Checks

If NGINX performs health checks against `/health/ready` (as shown earlier), you can use **AppDynamics Machine Agent** on the load balancer VM to monitor NGINX's status.

Install Machine Agent:

Download and install the **AppDynamics Machine Agent** on the NGINX host.

Then, enable **NGINX Monitoring Extension**:

```bash
wget https://github.com/Appdynamics/nginx-monitoring-extension/archive/master.zip
unzip master.zip
```

Configure:

```yaml
# monitor.xml
servers:
  - host: "localhost"
    port: 8080   # or your NGINX status port
    useSSL: false
    metrics:
      - nginx.status.active
      - nginx.status.accepts
      - nginx.status.handled
      - nginx.status.reading
      - nginx.status.writing
      - nginx.status.waiting
```

Enable NGINX status endpoint:

```nginx
location /nginx_status {
    stub_status on;
    allow 127.0.0.1;
    deny all;
}
```

✅ This lets AppDynamics collect:

- Active connections
- Requests per second
- Backend availability (if health checks fail)

🧩 6. End-to-End Architecture (with AppDynamics)

```bash
                     ┌─────────────────────────────┐
                     │      AppDynamics Cloud      │
                     │   (Controller Dashboard)    │
                     └──────────────┬──────────────┘
                                    │
                                    ▼
                     ┌──────────────────────────┐
                     │ Machine Agent (NGINX LB) │───► Monitors NGINX health metrics
                     └──────────────┬───────────┘
                                    │
                  ┌─────────────────┴──────────────┐
                  │  FastAPI VM 1, VM 2, ...        │
                  │  Python Agent Installed         │
                  │  Exposes /health endpoints      │
                  └─────────────────┬──────────────┘
                                    │
                              Dependencies
                        (DB, Redis, External APIs)
```

AppDynamics dashboard then shows:

- App response times (from Python agent)
- Health endpoint availability (custom metric)
- NGINX backend connection status (Machine Agent)
- Host performance (CPU, memory, network)

⚠️ 7. Health Rule & Alert Example in AppDynamics

In AppDynamics UI:

Health Rule Example:

- Name: `FastAPI Service Health`
- Affected Entities: Custom Metric → `Custom Metrics|FastAPI|Health|Readiness`
- Condition: If metric value = 0 for 3 consecutive intervals → Critical alert

Policy Actions:

- Send email/SMS/Slack message
- Trigger a webhook to your ticketing system

🧠 8. Optional — Combine with Synthetic Health Checks

If you want *end-to-end validation* (like a real user experience test), use **AppDynamics Synthetic Monitoring** (formerly Browser Synthetic Agent):

- Configure a synthetic job to `GET http://lb.internal.local/health/ready`
- Schedule every minute
- Alert if HTTP status != 200

That way, AppDynamics validates not only backend health but also **network and LB availability**.

✅ Summary

Layer	Tool	What It Monitors
FastAPI app	**AppDynamics Python Agent**	Endpoint performance, errors, health endpoint metrics
Custom metric	**AppDynamics API (add_metric)**	Explicit health status metric
NGINX load balancer	**AppDynamics Machine Agent + NGINX Extension**	Connection load, backend health
Synthetic monitoring	**AppDynamics Browser/Synthetic**	End-to-end service reachability
Alerts	**AppDynamics Health Rules**	Automatic alerts when service unhealthy

---

Would you like me to show a **dashboard example** (JSON or YAML definition) for visualizing health status of all FastAPI services inside AppDynamics (e.g., green/red indicator for each service)?

how can we use splunk capabilities for above scenario around checking healthcheck of services

Perfect — integrating **Splunk** into your FastAPI + on-prem + NGINX setup is one of the **best ways** to get **real-time observability** into your health checks, logs, and performance data.

Let's go in depth on how to use **Splunk's capabilities** for **monitoring health checks**, **alerting**, and **visualizing service health** for your Python FastAPI services deployed in on-prem VMs.

🧭 1. What Splunk Can Do in This Scenario

Splunk can monitor **three dimensions** of your setup:

Layer	Data Source	Purpose
Application (FastAPI)	Application logs, custom health metrics	Detect unhealthy services, response latency
Infrastructure (VMs)	OS logs, metrics	CPU/memory/network health
Load Balancer (NGINX)	Access/error logs	Identify failed upstreams, backend unavailability

---

🧱 2. Data Flow Overview

```scss
           ┌───────────────────────────────┐
           │        Splunk Platform        │
           │  (Enterprise / Cloud / HEC)   │
           └───────────────────────────────┘
                     │               │
                     ▼               ▼
        ┌────────────────────┐   ┌─────────────────────┐
        │ Splunk Universal   │   │ Splunk HEC (HTTP    │
        │ Forwarder on VMs   │   │ Event Collector)    │
        └────────────────────┘   └─────────────────────┘
                     │
                     ▼
              ┌─────────────────┐
              │ FastAPI App     │
              │ + Health Checks │
              │ + NGINX Logs    │
              └─────────────────┘
```

We can send:

- FastAPI healthcheck logs and custom metrics → Splunk
- NGINX access/error logs → Splunk
- Synthetic health pings → Splunk dashboards and alerts

⚙️ 3. Enable Logging from FastAPI Health Endpoints

Example FastAPI app with Splunk integration

We'll log every health check event with structured JSON that Splunk can easily index.

```python
from fastapi import FastAPI
import logging, json, time
import asyncpg

app = FastAPI()

# JSON logger
logger = logging.getLogger("health_logger")
handler = logging.FileHandler("/var/log/fastapi_health.log")
```

```
formatter = logging.Formatter('%(message)s')
handler.setFormatter(formatter)
logger.addHandler(handler)
logger.setLevel(logging.INFO)

@app.get("/health/ready")
async def ready():
    start = time.time()
    status = "ready"
    error = None
    try:
        conn = await asyncpg.connect("postgres://user:pass@localhost/db")
        await conn.execute("SELECT 1")
        await conn.close()
    except Exception as e:
        status = "unready"
        error = str(e)

    duration = round(time.time() - start, 3)
    log_data = {
        "event": "healthcheck",
        "status": status,
        "duration": duration,
        "timestamp": int(time.time()),
        "error": error
    }
    logger.info(json.dumps(log_data))
    return log_data
```

✅ Each healthcheck call writes JSON logs like:

```json
{"event": "healthcheck", "status": "ready", "duration": 0.031, "timestamp": 1735105100, "error": null}
```

🚀 4. Send Logs to Splunk

There are **two main ways** to send these logs to Splunk from your on-prem VMs.

Option A: Splunk Universal Forwarder (recommended on-prem)

1. **Install Forwarder:**

   ```bash
   wget -O splunkforwarder.tgz
   https://download.splunk.com/products/universalforwarder/releases/...
   tar -xvf splunkforwarder.tgz
   ./splunkforwarder/bin/splunk start --accept-license
   ./splunkforwarder/bin/splunk enable boot-start
   ```

2. **Configure input for your FastAPI logs:**
   `/opt/splunkforwarder/etc/system/local/inputs.conf`

   ```ini
   [monitor:///var/log/fastapi_health.log]
   index = fastapi_health
   ```

```
sourcetype = fastapi:health
host = app1.internal.local
```

3. **Configure forwarding to Splunk indexer:**

   `/opt/splunkforwarder/etc/system/local/outputs.conf`

```
ini

[tcpout]
defaultGroup = default-autolb-group

[tcpout:default-autolb-group]
server = splunk-indexer.internal.local:9997
```

4. **Restart forwarder:**

```
bash

./splunkforwarder/bin/splunk restart
```

✅ Now Splunk will ingest healthcheck logs continuously.

Option B: Directly via HTTP Event Collector (HEC)

If you want to push metrics from the app directly:

1. Enable HEC in Splunk (`Settings → Data Inputs → HTTP Event Collector`).
2. Get the token and endpoint (e.g., `https://splunk.company.com:8088/services/collector`).
3. Add Python code:

```
python

import requests, json, time

SPLUNK_HEC_URL = "https://splunk.company.com:8088/services/collector"
SPLUNK_TOKEN = "YOUR_TOKEN"

def send_to_splunk(payload):
    headers = {"Authorization": f"Splunk {SPLUNK_TOKEN}"}
    event = {
        "time": int(time.time()),
        "host": "app1.internal.local",
        "sourcetype": "fastapi:health",
        "event": payload
    }
    requests.post(SPLUNK_HEC_URL, headers=headers, data=json.dumps(event), verify=False)
```

Then call `send_to_splunk(log_data)` in your `/health/ready` endpoint.

📊 5. Configure NGINX Logs for Splunk

You can also monitor backend failures or healthcheck results from NGINX.

In `/etc/nginx/nginx.conf`:

```
nginx

log_format json_combined escape=json
    '{ "time":"$time_iso8601",'
    '"remote_addr":"$remote_addr",'
    '"status":"$status",'
    '"request":"$request",'
    '"upstream_status":"$upstream_status",'
    '"upstream_addr":"$upstream_addr",'
    '"request_time":"$request_time",'
    '"upstream_response_time":"$upstream_response_time" }';

access_log /var/log/nginx/access_json.log json_combined;
```

Then add this file to Splunk Forwarder's monitored inputs:

```
ini

[monitor:///var/log/nginx/access_json.log]
index = nginx_access
sourcetype = nginx:json
```

Now you'll see when **`/health/ready`** calls fail (non-200 statuses).

📈 6. Build Splunk Dashboards

Once data is flowing, you can build **real-time health dashboards**.

Example 1: Service Health Summary Panel

Search:

```
spl

index=fastapi_health event=healthcheck
| stats latest(status) as last_status by host
| eval color=if(last_status=="ready","green","red")
```

Visualize as a **table** or **single value visualization** with color-by-status.

Example 2: Healthcheck Latency Trend

```
spl

index=fastapi_health event=healthcheck
| timechart avg(duration) as avg_latency by host
```

Example 3: NGINX Upstream Failures

```
spl

index=nginx_access request="/health/ready"
| stats count by status, upstream_status, upstream_addr
```

This shows which backend services fail readiness checks.

🔔 7. Alerts in Splunk

You can create **alert rules** to detect when any service becomes unhealthy.

Example:

```
spl

index=fastapi_health event=healthcheck status="unready"
| stats count by host
```

Set an alert:

- Trigger when result count > 0
- Send email, Slack, or webhook notification
- Optionally trigger automatic remediation script (restart service)

🧠 8. Optional — Synthetic Health Monitoring in Splunk

Splunk **Synthetic Monitoring** (via **Splunk Observability Cloud**) can continuously hit your endpoints externally.

Example configuration:

- URL: `http://lb.internal.local/health/ready`
- Interval: every 1 minute
- Conditions:
  - HTTP status == 200
  - Response time < 500 ms

Results go into Splunk dashboards automatically, giving you **outside-in visibility**.

✅ 9. Summary Table

Layer	Data Source	Splunk Integration	What You Get
FastAPI healthchecks	JSON logs	Universal Forwarder / HEC	Live health status, errors, durations
NGINX load balancer	access/error logs	Forwarder	Backend failures, 503s
VM health	OS metrics	Forwarder or Splunk Infrastructure Monitoring	Resource-level alerts
Synthetic endpoint testing	HTTP probes	Splunk Observability Cloud	End-to-end uptime validation
Alerting & Dashboards	Splunk Core / ITSI		Visual health map, automated alerts

🧩 10. Example Dashboard Layout

- **Top Row**: Service health indicators (green/red per VM)
- **Middle Row**: Average latency (trend)
- **Bottom Row**: NGINX backend errors / failed readiness checks
- **Sidebar**: Recent error logs (for context)

Would you like me to show a **Splunk Dashboard JSON definition** (ready to import) that visualizes FastAPI service health status, average latency, and recent failures — all in one view?