VPS Local: Construindo uma Infraestrutura de IA e Dados

Prof. Dr. Walmes Marques Zeviani
Especialização em Inteligência Artificial Generativa - UFPR

2026-04-24

i Objetivos do Tutorial

Este documento contém o passo a passo para a criação de um ambiente de desenvolvimento local (Local VPS) robusto, isolado e reprodutível. Ao final deste guia, os alunos terão construído uma infraestrutura local contendo serviços de Inteligência Artificial, análise de dados e automação de fluxos de trabalho.

i Resultados Esperados

- Máquina Virtual (VM) Ubuntu configurada localmente.
- Docker e Docker Compose instalados e configurados.
- Portainer em execução para o gerenciamento visual dos contêineres.
- Stacks de IA (Ollama, Open WebUI, LiteLLM), Dados (RStudio, Shiny) e Automação (n8n) operacionais e interconectadas em uma rede Docker local.

! Ambiente de Execução

Este tutorial foi desenvolvido e homologado para sistemas operacionais **Linux**. Os comandos aqui presentes levam em consideração distribuições baseadas em Debian/Ubuntu, com ênfase especial no Linux Mint.

1. Instalação do Multipass

O Multipass é uma ferramenta da Canonical que permite a criação e o gerenciamento de Máquinas Virtuais (VMs) Ubuntu de forma simples e rápida (documentação oficial: https://canonical.com/multipass/install).

O Linux Mint bloqueia o gerenciador de pacotes `snap` por padrão. Como o Multipass é distribuído oficialmente via Snap, é necessário desbloqueá-lo antes da instalação.

1.1. Desbloqueio e Instalação no Linux Mint

1. **Remova o bloqueio do Snap:** O Linux Mint utiliza um arquivo de preferência que impede a instalação de pacotes Snap. Edite ou remova este arquivo:

```
# Opção 1: Comentar as linhas do arquivo
sudo nano /etc/apt/preferences.d/nosnap.pref

# Opção 2: Remover o arquivo (recomendado para seguir o tutorial)
# sudo rm /etc/apt/preferences.d/nosnap.pref
```

2. **Atualize os pacotes e instale o `snapd`:**

```
sudo apt update && sudo apt install snapd -y
```

3. **Instale o Multipass:**

```
sudo snap install multipass
```

4. **Verifique a instalação:** A execução bem-sucedida do comando abaixo retornará a versão instalada.

```
multipass --version
```

2. Criação da Máquina Virtual

O Multipass permite especificar qual versão do Ubuntu será utilizada. Para um ambiente de laboratório (rodando Docker, Ollama, etc.), recomenda-se a versão `24.04 LTS (noble)`, pois conta com bibliotecas e kernel atualizados.

2.1. Escolha da Versão do Ubuntu

Para visualizar as versões disponíveis:

```
multipass find
```

2.2. Provisionamento da VM

Execute o comando abaixo para instanciar a VM. O *alias* `24.04` garante a versão 24.04 LTS (noble). O comando abaixo aloca 8GB de RAM, 40GB de disco e 4 CPUs virtuais, valores recomendados para a execução adequada dos serviços de IA.

```
multipass launch 24.04 --name lab-genai --memory 8G --disk 40G --cpus 4
```

2.3. Exame da Máquina Virtual

Comandos úteis para monitorar os recursos da VM criada:

```
# Exibe informações gerais da VM (IP, uso de disco, RAM)
multipass info lab-genai

# Acessa o shell da VM para verificar partições
multipass shell lab-genai
df -h

# Verifica o consumo de disco pelos contêineres Docker (necessário ter o Docker instalado)
docker system df
```

3. Instalação do Docker e Docker Compose

O ambiente virtual atuará como um servidor de contêineres. A instalação via script de conveniência oficial é a forma mais ágil para laboratórios de testes.

3.1. Acesso à Máquina Virtual

Acesse o terminal da VM recém-criada:

```
multipass shell lab-genai
```

> Dica
>
> Para descobrir o IP da sua VM no terminal do sistema hospedeiro (Host), utilize o comando:
>
> ```
> multipass list
> ```
>
> Anote o IP (ex: `10.106.126.40`), pois ele será utilizado frequentemente ao longo do tutorial para acessar os serviços.

3.2. Instalação e Configuração de Permissões

Execute os comandos abaixo **dentro do shell da VM**.

1. **Baixe e execute o instalador oficial:**

   ```
   curl -fsSL https://get.docker.com | sh
   ```

2. **Configure as permissões de usuário:** Adicione o usuário atual (ubuntu) ao grupo do docker para evitar o uso de `sudo` em todos os comandos.

   ```
   sudo usermod -aG docker $USER
   ```

3. **Aplique a mudança de grupo:**

```
newgrp docker
```

4. **Verifique a instalação:**

```
docker --version
docker compose version
docker run hello-world
```

Se a mensagem **"Hello from Docker!"** for exibida, o ambiente está configurado corretamente.

4. Gestão de Contêineres com Portainer

O Portainer (https://www.portainer.io/) fornece uma interface visual amigável para gerenciamento de contêineres, imagens e volumes.

4.1. Preparação de Diretórios

Antes de subir o Portainer, crie um diretório que servirá como repositório para os arquivos de configuração (`docker-compose.yml` e `Dockerfile`) de todas as stacks.

```
# Cria o diretório para armazenar as configurações (Stacks)
sudo mkdir -p /opt/stacks

# Ajusta as permissões para o usuário atual
sudo chown -R 1000:1000 /opt/stacks

# Cria um arquivo de teste
echo "Teste de leitura do Portainer" > /opt/stacks/readme.txt
```

4.2. Execução do Portainer

Execute o comando abaixo para instanciar o Portainer. O mapeamento `/opt/stacks:/opt/stacks` permite que o Portainer tenha acesso aos arquivos de configuração criados na etapa anterior.

```
docker run -d \
  --name portainer \
  --restart=always \
  -p 9000:9000 \
  -p 9443:9443 \
  -v /var/run/docker.sock:/var/run/docker.sock \
  -v portainer_data:/data \
  -v /opt/stacks:/opt/stacks \
  portainer/portainer-ce:latest
```

> **Nota**
>
> **Acesso ao Portainer:** Abra o navegador no sistema hospedeiro e acesse: `http://<IP_DA_VM>:9000`. Crie a senha inicial do administrador para prosseguir.

5. Configuração de Rede Compartilhada e IPv4

Para que os contêineres de diferentes Stacks (IA, Dados, Automação) se comuniquem utilizando os nomes dos serviços (DNS interno do Docker), é necessário criar uma rede externa dedicada.

5.1. Criação da Rede

No terminal da VM, execute:

```
docker network create lab-net
```

5.2. Forçar IPv4 (Correção de Timeout)

Muitas vezes, a resolução de nomes via IPv6 pelo Docker em VMs Multipass falha, gerando erros de `network unreachable` ao fazer o pull de imagens. Para estabilizar as conexões, desativaremos o IPv6.

1. **Correção Imediata:**

   ```
   sudo sysctl -w net.ipv6.conf.all.disable_ipv6=1
   sudo sysctl -w net.ipv6.conf.default.disable_ipv6=1
   sudo sysctl -w net.ipv6.conf.lo.disable_ipv6=1
   sudo systemctl restart docker
   ```

2. **Correção Permanente:** Abra o arquivo `/etc/sysctl.conf`:

   ```
   sudo nano /etc/sysctl.conf
   ```

   Adicione as linhas a seguir ao final do arquivo:

   ```
   net.ipv6.conf.all.disable_ipv6 = 1
   net.ipv6.conf.default.disable_ipv6 = 1
   net.ipv6.conf.lo.disable_ipv6 = 1
   ```

   Aplique as alterações com o comando `sudo sysctl -p`.

6. Stack de IA: Ollama + Open WebUI

O Ollama é responsável por rodar os Modelos de Linguagem de Larga Escala (LLMs) localmente. O Open WebUI atua como uma interface de chat similar ao ChatGPT.

No **Portainer**, vá em **Stacks** > **Add stack**. Dê o nome de `stack-ia` e utilize o conteúdo a seguir:

```yaml
services:
  ollama:
    image: ollama/ollama:latest
    container_name: ollama
    restart: always
    volumes:
      - ollama_data:/root/.ollama
    networks:
      - lab-net
    # A execução ocorrerá via CPU. Configurações para GPU exigem passthrough.

  open-webui:
    image: ghcr.io/open-webui/open-webui:main
    container_name: open-webui
    restart: always
    environment:
      - OLLAMA_BASE_URL=http://ollama:11434
      - ENABLE_CHANNELS=true
    ports:
      - "3000:8080"
    volumes:
      - open_webui_data:/app/backend/data
    networks:
      - lab-net
    depends_on:
      - ollama

networks:
  lab-net:
    external: true # Utiliza a rede global criada anteriormente

volumes:
  ollama_data:
  open_webui_data:
```

6.1. Adição de Modelos ao Ollama

O modelo `llama3.2:1b` é recomendado para processamento em CPU por ser extremamente leve. Como o Ollama roda dentro do Docker, os comandos devem ser enviados para dentro do contêiner.

Execute os comandos a seguir no terminal da VM para realizar o download dos modelos:

```
docker exec -it ollama ollama pull llama3.2:1b
docker exec -it ollama ollama pull qwen2-math:1.5b
```

```
# Modelo de Embedding
docker exec -it ollama ollama pull mxbai-embed-large
```

> **i** Nota
>
> **Teste Prático:** Para interagir diretamente no terminal, execute:
>
> ```
> docker exec -it ollama ollama run llama3.2:1b
> ```
>
> Acesse o Open WebUI no navegador: `http://<IP_DA_VM>:3000`. Crie seu usuário, selecione o modelo e inicie um chat.

7. Integração de Serviços Customizados (Langchain Adapter)

O Open WebUI possui um motor Python interno ("Pipes") que permite criar adaptadores para APIs externas, conectando a interface de chat com fluxos ou serviços personalizados desenvolvidos em Langchain.

7.1. Criação da Função (Pipe)

1. Acesse o **Open WebUI**.
2. Vá em **Workspace** > **Functions**.
3. Clique em **+** (Criar nova função) e nomeie como **"Pipe Fale Bíblia"**.
4. Insira o código Python abaixo, ajustando o IP ou nome do contêiner da sua API Langchain em `API_URL`.

```
"""
title: Fale Bíblia Pipe
author: Adaptador API Langchain
version: 0.1.0
"""

import requests
import json
from pydantic import BaseModel, Field
from typing import Union, Generator, Iterator

class Pipe:
    class Valves(BaseModel):
        # Substitua 'falebiblia' pelo nome ou IP do serviço alvo, caso necessário
        API_URL: str = Field(default="http://falebiblia:8008/falebiblia/invoke")
```

```
    def __init__(self):
        self.valves = self.Valves()
        self.type = "manifold"
        self.id = "fale_biblia"
        self.name = "Fale Bíblia"

    def pipe(self, body: dict) -> Union[str, Generator, Iterator]:
        user_message = body.get("messages", [])[-1].get("content", "")

        payload = {
            "input": {
                "input": user_message,
                "chat_history": [],
            }
        }

        headers = {"Content-Type": "application/json"}

        try:
            response = requests.post(
                self.valves.API_URL,
                json=payload,
                headers=headers,
                timeout=30,
            )
            response.raise_for_status()
            data = response.json()

            if "output" in data and "content" in data["output"]:
                return data["output"]["content"]
            else:
              return f"Erro: Resposta inesperada. Recebido: {json.dumps(data)}"

        except Exception as e:
            return f"Erro ao conectar no serviço: {str(e)}"
```

Salve e, ao iniciar um novo chat, o modelo “Fale Bíblia” estará disponível para seleção.

8. Stack LiteLLM + PostgreSQL

O LiteLLM unifica diferentes provedores de LLM e o PostgreSQL armazena logs e métricas de consumo de tokens.

8.1. Arquivos de Configuração

No terminal da VM, prepare os diretórios e crie a configuração do LiteLLM:

```
mkdir -p /home/ubuntu/stacks/litellm-stack
nano /home/ubuntu/stacks/litellm-stack/litellm-config.yaml
```

Cole o conteúdo abaixo no arquivo YAML:

```
# litellm-config.yaml
model_list:
  - model_name: llama3-1b
    litellm_params:
      model: ollama/llama3.2:1b
      api_base: http://ollama:11434

  - model_name: qwen2-math-1.5b
    litellm_params:
      model: ollama/qwen2-math:1.5b
      api_base: http://ollama:11434
```

8.2. Docker Compose do LiteLLM

Crie uma nova Stack no Portainer chamada `stack-litellm` com o código abaixo:

```
services:
  db:
    image: postgres:16-alpine
    container_name: litellm_db
    restart: always
    ports:
      - "5432:5432"
    environment:
      POSTGRES_DB: litellm
      POSTGRES_USER: llmproxy
      POSTGRES_PASSWORD: dbpassword9090
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - lab-net
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -d litellm -U llmproxy"]
      interval: 5s
      timeout: 5s
      retries: 5
```

```
  litellm:
    image: ghcr.io/berriai/litellm:main-v1.80.5-stable
    container_name: litellm
    restart: always
    ports:
      - "4000:4000"
    depends_on:
      db:
        condition: service_healthy
    command:
      - "--config=/app/config.yaml"
      - "--detailed_debug"
    volumes:
      - /home/ubuntu/stacks/litellm-stack/litellm-config.yaml:/app/config.yaml
    networks:
      - lab-net
    environment:
      - DATABASE_URL=postgresql://llmproxy:dbpassword9090@db:5432/litellm
      - STORE_MODEL_IN_DB=True
      - LITELLM_MASTER_KEY=sk-1234567890
      - UI_USERNAME=admin
      - UI_PASSWORD=admin
      - LITELLM_SALT_KEY=segredo-super-seguro-lab
    healthcheck:
      test:
        - CMD-SHELL
          - python3 -c "import urllib.request; urllib.request.urlopen('http://
localhost:4000/health/liveliness')"
      interval: 30s
      timeout: 10s
      retries: 3
      start_period: 20s

networks:
  lab-net:
    external: true

volumes:
  postgres_data:
```

8.3. Verificação de Funcionamento

Execute requisições curl para verificar se a API do LiteLLM está respondendo corretamente:

```
# Teste de inferência utilizando a chave mestra e o modelo llama3-1b
curl http://localhost:4000/chat/completions \
  -H "Authorization: Bearer sk-1234567890" \
  -H "Content-Type: application/json" \
  -d '{
    "model": "llama3-1b",
   "messages": [{"role": "user", "content": "Quando surgiu o jogo de Xadrez?"}]
  }'
```

> **Nota**
>
> Para verificar a base de dados via cliente visual (ex: DBeaver) a partir do Host, utilize as seguintes credenciais de conexão PostgreSQL: - **Host:** `<IP_DA_VM>` (ex: 10.106.126.40) | **Port:** `5432` - **Database:** `litellm` | **User:** `llmproxy` | **Password:** `dbpassword9090`

9. Stack de Automação: N8N e API Auxiliar

O n8n é uma ferramenta de automação de fluxo de trabalho baseada em nós. Em conjunto, instalaremos um serviço de conversão de texto Markdown customizado.

9.1. API de Conversão para Telegram

Antes de subir a Stack, crie os arquivos do serviço customizado em Python que converterá o padrão Markdown tradicional para a versão compatível com a API do Telegram.

```
# Crie e acesse o diretório da API
mkdir -p /opt/stacks/telegram-converter
cd /opt/stacks/telegram-converter
```

Crie o arquivo `/opt/stacks/telegram-converter/requirements.txt`:

```
fastapi
uvicorn
telegramify-markdown
pydantic
```

Crie o arquivo `/opt/stacks/telegram-converter/main.py`:

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
import telegramify_markdown

app = FastAPI(title="API Conversor Markdown Telegram")
```

```
class TextInput(BaseModel):
    text: str

@app.get("/")
def health_check():
    return {"status": "online", "service": "Telegramify Converter"}

@app.post("/convert")
def convert_markdown(payload: TextInput):
    try:
        converted_text = telegramify_markdown.markdownify(payload.text)
        return {"original": payload.text, "converted": converted_text}
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
```

Crie o arquivo `/opt/stacks/telegram-converter/Dockerfile`:

```
FROM python:3.10-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY main.py .
EXPOSE 8000
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

9.2. Docker Compose do n8n

No **Portainer**, crie a Stack `stack-n8n`:

> ⚠ Aviso
>
> Substitua `<IP_DA_VM>` no campo `WEBHOOK_URL` pelo IP retornado no comando `multipass list`.

```
services:
  n8n:
    image: docker.n8n.io/n8nio/n8n
    container_name: n8n
    restart: always
    ports:
      - "5678:5678"
    environment:
      - N8N_HOST=0.0.0.0
```

```
      - N8N_PORT=5678
      - N8N_PROTOCOL=http
      - WEBHOOK_URL=http://<IP_DA_VM>:5678/  # Substitua <IP_DA_VM> pelo seu IP
      - NODE_ENV=production
      - N8N_ENFORCE_SETTINGS_FILE_PERMISSIONS=true
      - N8N_SECURE_COOKIE=false
      - GENERIC_TIMEZONE=America/Sao_Paulo
      - TZ=America/Sao_Paulo
    volumes:
      - n8n_data:/home/node/.n8n
      - ./local-files:/files
    networks:
      - lab-net

  converter:
    container_name: telegram-converter
    build:
      context: /opt/stacks/telegram-converter
      dockerfile: Dockerfile
    restart: always
    expose:
      - "8000"
    ports:
      - "8005:8000"
    networks:
      - lab-net

networks:
  lab-net:
    external: true

volumes:
  n8n_data:
```

Ao adicionar um nó "Ollama" no n8n, informe a **Base URL** como `http://ollama:11434`. Como ambos compartilham a rede `lab-net`, a conexão será estabelecida automaticamente.

10. Qdrant: Banco de Dados Vetorial

O Qdrant será utilizado para armazenar vetores resultantes de extração de embeddings, suportando aplicações de Geração Aumentada por Recuperação (RAG).

Crie a Stack `stack-qdrant`:

```yaml
services:
  qdrant:
    image: qdrant/qdrant:v1.16
    container_name: qdrant
    restart: always
    ports:
      - "6333:6333" # Porta REST (Dashboard, N8N, WebUI)
      - "6334:6334" # Porta gRPC
    environment:
      - QDRANT__SERVICE__API_KEY=senha-secreta-vetorial
      - QDRANT__SERVICE__ENABLE_STATIC_CONTENT=true
    volumes:
      - qdrant_data:/qdrant/storage
    networks:
      - lab-net
    healthcheck:
      test: ["CMD", "bash", "-c", "cat < /dev/null > /dev/tcp/localhost/6333"]
      interval: 30s
      timeout: 10s
      retries: 3

networks:
  lab-net:
    external: true

volumes:
  qdrant_data:
```

Integração no n8n: - Utilize o nó **Qdrant Vector Store**. - Host: `http://qdrant:6333` - API Key: `senha-secreta-vetorial` - Dashboard acessível em: `http://<IP_DA_VM>:6333/dashboard`.

11. Stack de Dados: RStudio + Shiny

A Stack consolida o ambiente de pesquisa de dados estatísticos com capacidades de desenvolvimento Web interativo via Shiny.

11.1. Arquivos de Configuração

Execute no terminal da VM:

```bash
mkdir -p ~/stacks/r-stack
cd ~/stacks/r-stack
mkdir -p projects/shiny-apps libs
sudo chown -R 1000:1000 projects libs
```

Crie o arquivo `~/stacks/r-stack/Dockerfile`:

```
FROM rocker/geospatial:latest

RUN /rocker_scripts/install_shiny_server.sh

RUN apt-get update && apt-get install -y \
    libxml2-dev \
    libcurl4-openssl-dev \
    && rm -rf /var/lib/apt/lists/*

# Instala pacotes do laboratório
RUN R -e "install.packages(c('remotes', 'httr2', 'ellmer'))"

EXPOSE 8787 3838
```

Crie a Stack `stack-rstudio` no Portainer ou via docker-compose:

```
services:
  r-server:
    build: /home/ubuntu/stacks/r-stack
    container_name: r_geospatial_shiny
    restart: always
    environment:
      - PASSWORD=suasenha
      - ROOT=true
      - USERID=1000
      - GROUPID=1000
      - R_LIBS_USER=/home/rstudio/R_libs
    ports:
      - "8787:8787"
      - "3838:3838"
    volumes:
      - /home/ubuntu/stacks/r-stack/projects:/home/rstudio/projects
      - /home/ubuntu/stacks/r-stack/projects/shiny-apps:/srv/shiny-server
      - /home/ubuntu/stacks/r-stack/libs:/home/rstudio/R_libs
    networks:
      - lab-net

networks:
  lab-net:
    external: true
```

11.2. Teste de Aplicação Shiny + IA

1. Crie o diretório do app: `mkdir -p ~/stacks/r-stack/projects/shiny-apps/teste-ia`
2. Crie e salve o arquivo `app.R` dentro dele:

```
library(shiny)
library(ellmer)
library(bslib)

ui <- page_sidebar(
  title = "Dashboard Lab: R + Shiny + Ollama",
  theme = bs_theme(version = 5, bootswatch = "zephyr"),

  sidebar = sidebar(
    h4("Controles"),
    h5("🤖 Inteligência Artificial"),
    selectInput("model", "Modelo Ollama:", choices = c("llama3.2:1b")),
     textAreaInput("prompt", "Sua pergunta:", height = "100px", placeholder =
"Ex: O que é estatística?"),
    actionButton("btn_send", "Enviar para Ollama", class = "btn-primary", icon
= icon("paper-plane")),
    hr(),
    h5("📊 Teste de Reatividade"),
    sliderInput("bins", "Número de barras:", min = 5, max = 50, value = 20),
    actionButton("btn_regen", "Gerar Nova Amostra", class = "btn-success", icon
= icon("sync"))
  ),

  layout_columns(
    col_widths = c(12, 12),
    card(
      card_header("💬 Resposta do Ollama"),
     div(style = "min-height: 100px; padding: 10px; background-color: #f8f9fa;",
textOutput("ai_response"))
    ),
    card(
      card_header("📈 Histograma Dinâmico"),
      plotOutput("distPlot", height = "300px")
    )
  )
)

server <- function(input, output, session) {
  dados_reativos <- reactive({
```

```
    input$btn_regen
    rnorm(500)
  })

  output$distPlot <- renderPlot({
    x <- dados_reativos()
     hist(x, breaks = input$bins, col = "#007bc2", border = "white", main =
"Histograma", xlab = "Valor")
  })

  resposta_ia <- eventReactive(input$btn_send, {
    req(input$prompt)
   id <- showNotification("Consultando o Ollama...", duration = NULL, closeButton
= FALSE)
    on.exit(removeNotification(id), add = TRUE)

    tryCatch({
     chat <- chat_ollama(model = input$model, base_url = "http://ollama:11434")
      chat$chat(input$prompt)
    }, error = function(e) {
      paste("Erro de Conexão:", e$message)
    })
  })

  output$ai_response <- renderText({ resposta_ia() })
}

shinyApp(ui = ui, server = server)
```

Acesse a aplicação no navegador em `http://<IP_DA_VM>:3838/teste-ia/`. O pacote `ellmer` conseguirá se conectar diretamente ao Ollama usando a referência `http://ollama:11434` sem roteamento adicional.

12. Gestão da Máquina Virtual

12.1. Controle de Ciclo de Vida

Quando não estiver utilizando o ambiente e precisar liberar a memória RAM, encerre a máquina virtual:

```
multipass stop lab-genai
```

Para retomar o ambiente (os contêineres serão reiniciados automaticamente em função da política `--restart=always`):

```
multipass start lab-genai
```

Para obter o IP após reiniciar a máquina (pode ter sido alterado na nova sessão DHCP):

```
multipass list
```

13. Tabela de Conexões e Senhas

Quadro de referência para os serviços instalados:

Serviço	URL (Substitua IP_VM)	Usuário Padrão	Senha/Token Padrão
Portainer	`http://<IP_VM>:9000`	admin	Definida no 1º acesso
Open WebUI	`http://<IP_VM>:3000`	admin	Definida no 1º acesso
LiteLLM UI	`http://<IP_VM>:4000/ui/`	admin	admin
LiteLLM API	`http://<IP_VM>:4000`	-	sk-1234567890
n8n	`http://<IP_VM>:5678`	Usuário a cadastrar	Senha a cadastrar
RStudio	`http://<IP_VM>:8787`	rstudio	suasenha
Shiny Apps	`http://<IP_VM>:3838/<pasta>/`	-	-
Qdrant Panel	`http://<IP_VM>:6333/dashboard`	-	senha-secreta-vetorial

> ⚠ Aviso
>
> As senhas e tokens definidos nos scripts YAML são exemplificativos e focados na praticidade de um ambiente acadêmico ou de laboratório. Em um cenário real de produção (Deploy Externo), recomenda-se adotar soluções de Vault, segredos externos e variáveis de ambiente cifradas.