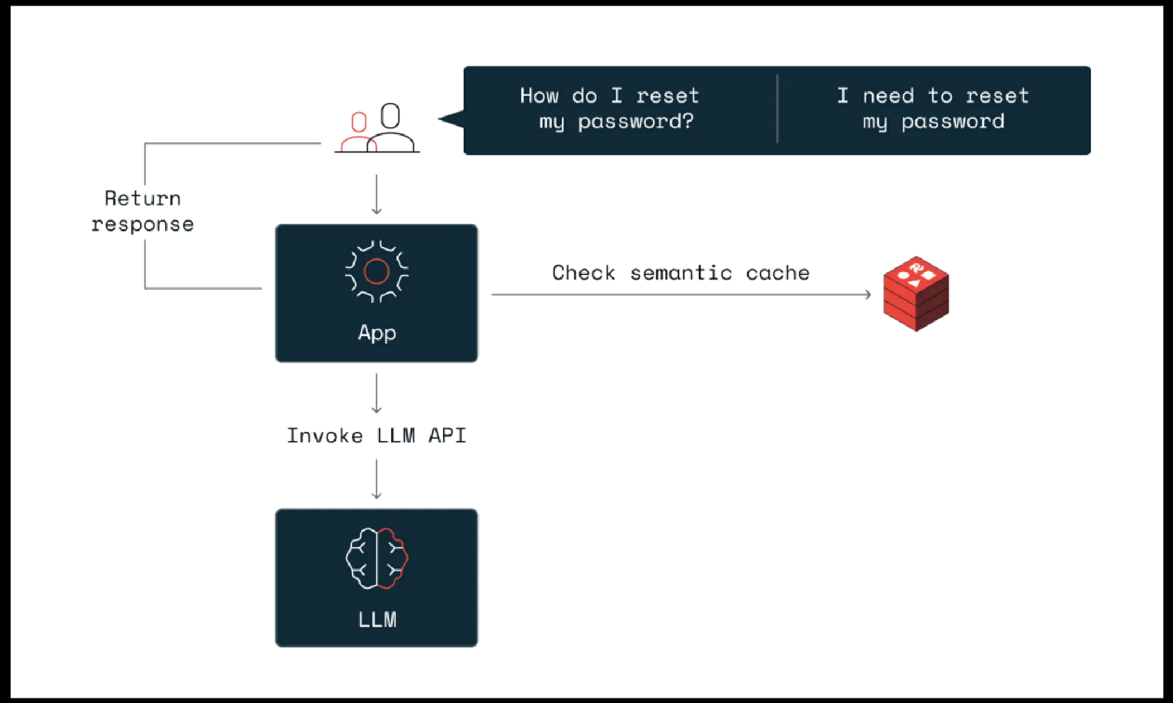
# LABORATÓRIO PRÁTICO: REDIS COMO CACHE SEMÂNTICO

## Sistemas Distribuídos e Otimização de Latência

Uma demonstração prática de como implementar cache semântico com Redis para reduzir latência e custos em aplicações de IA generativa. Abordagem passo a passo com Docker e LangChain.




LAB 01

# REQUISITOS DE INFRAESTRUTURA

Especificações Mínimas da Instância EC2

⚠ CONFIGURAÇÃO OBRIGATÓRIA

| INSTÂNCIA | MEMÓRIA RAM | ARMAZENAMENTO | SISTEMA OS |
|---|---|---|---|
| **t3.large** | **8 GB** | **32 GB** | **Ubuntu** |
| AWS EC2 Family | Mínimo para Ollama | SSD GP3 | 22.04 LTS |

**Por que esses requisitos?**

Instâncias menores não suportarão adequadamente o Ollama, Redis Stack e a aplicação Streamlit simultaneamente. O modelo de IA phi3 requer aproximadamente 2.3 GB de RAM dedicados.

# ARQUITETURA DO SISTEMA

COMPONENTES & COMUNICAÇÃO

## 1 Redis Stack

Banco de dados vetorial para armazenar embeddings e cache.

`PORTS: 6379, 8001`

## 2 Ollama Service

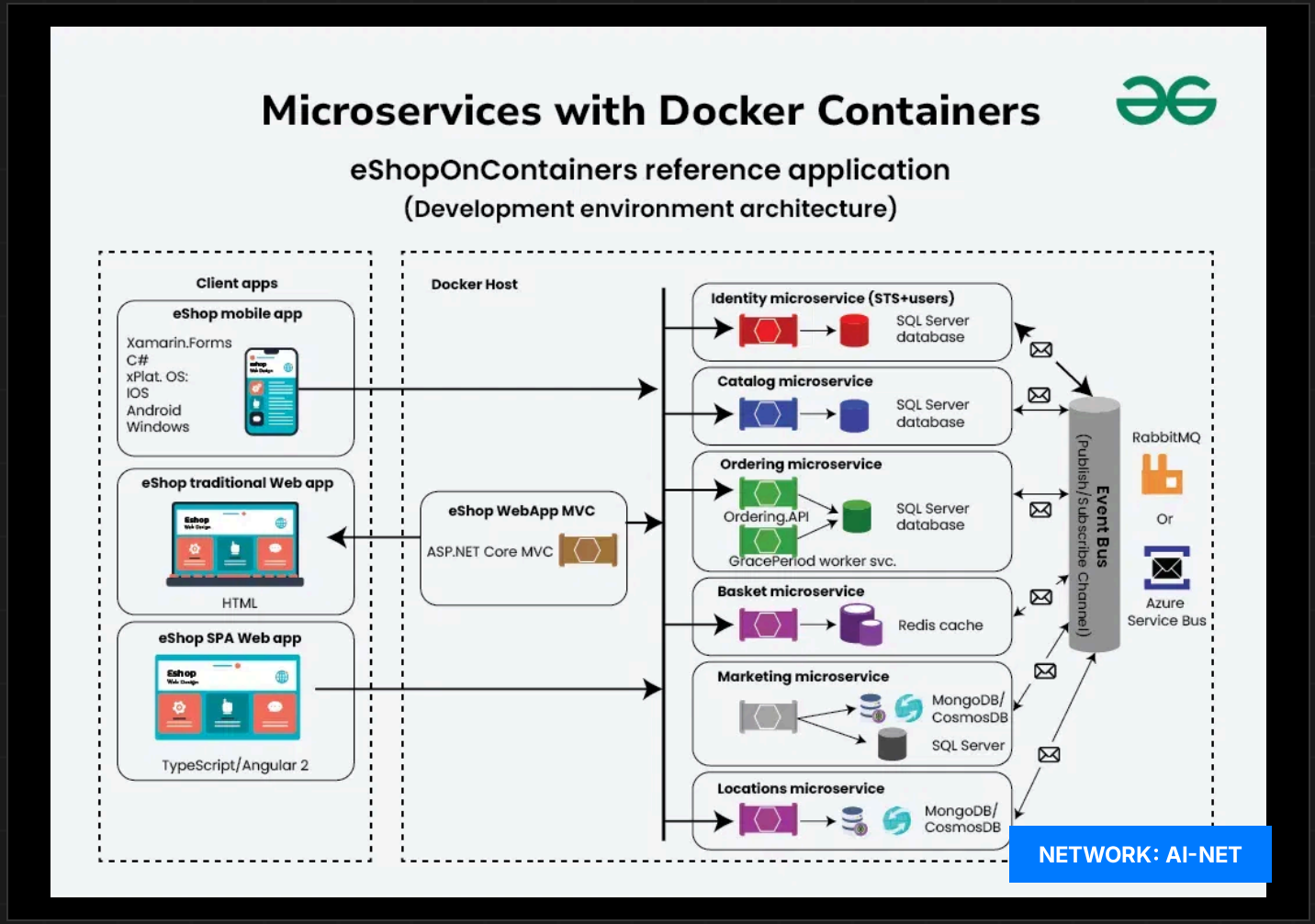
Motor de IA executando o modelo phi3 localmente.

`PORT: 11434 (Internal)`

## 3 Streamlit App

Frontend em Python com integração LangChain.

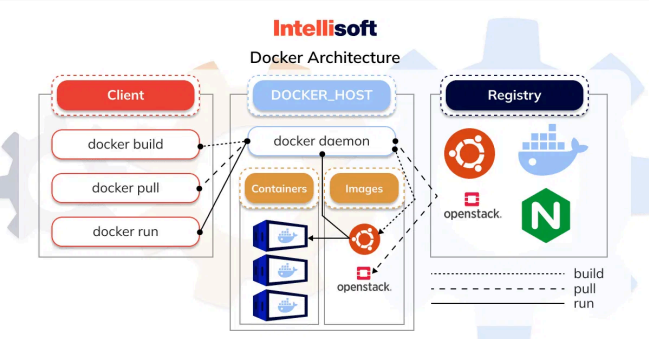
`PORT: 8501 (Public)`



# Passo 1

## INSTALAÇÃO DO DOCKER

Prepare sua instância EC2 instalando o Docker Engine e o plugin Compose.




ubuntu@ip-172-31-0-1:~

```
# 1. Atualiza pacotes e instala dependências
$  sudo apt-get update && sudo apt-get install -y ca-certificates curl gnupg

# 2. Configura repositório oficial
$  sudo install -m 0755 -d /etc/apt/keyrings
$  curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -
o /etc/apt/keyrings/docker.gpg

# 3. Instala Docker e plugins
$  sudo apt-get update
$  sudo apt-get install -y docker-ce docker-ce-cli containerd.io docker-compose-
plugin

# 4. Adiciona usuário ao grupo (evita sudo)
$  sudo usermod -aG docker $USER
```

**IMPORTANTE: Saia do SSH (exit) e entre novamente para aplicar as permissões.**

# Estrutura do Projeto

ORGANIZAÇÃO DE ARQUIVOS

A organização correta é essencial para manter a separação entre infraestrutura (Docker) e aplicação (Python).

```
$ mkdir projeto-redis-ollama
$ cd projeto-redis-ollama
```

| | |
|---|---|
| `requirements.txt` | Lista de bibliotecas Python (LangChain, Redis, Streamlit). |
| `Dockerfile` | Receita para criar a imagem containerizada da aplicação. |
| `app.py` | Código principal com lógica de cache semântico e UI. |
| `docker-compose.yml` | Orquestrador que liga Redis, Ollama e App na mesma rede. |

# PASSO 3: CONFIGURAÇÃO DE DEPENDÊNCIAS

## DEFININDO O AMBIENTE DA APLICAÇÃO

/projeto-redis-ollama

### requirements.txt

```
streamlit
langchain
langchain-community
langchain-ollama
redis==4.6.0
```

Bibliotecas essenciais para interface (Streamlit), orquestração de LLM (LangChain) e conexão com banco vetorial (Redis).

### Dockerfile

```
FROM python:3.9-slim WORKDIR /app # Instala dependências do sistema RUN apt-get
update && apt-get install -y build-essential curl # Otimização de cache de build
COPY requirements.txt . RUN pip install --no-cache-dir -r requirements.txt # Copia
código fonte COPY app.py . EXPOSE 8501 # Healthcheck para orquestração robusta
HEALTHCHECK CMD curl --fail http://localhost:8501/_stcore/health || exit 1
ENTRYPOINT ["streamlit", "run", "app.py", "--server.port=8501", "--
server.address=0.0.0.0"]
```

**Boas Práticas:** Uso de imagem slim, limpeza de cache do pip, healthcheck configurado e entrypoint explícito.

# PASSO 4: CÓDIGO DA APLICAÇÃO

## IMPLEMENTANDO CACHE SEMÂNTICO (PARTE 1)

### RedisSemanticCache

Intercepta consultas e armazena **embeddings** (representações vetoriais) das perguntas, não apenas o texto exato.

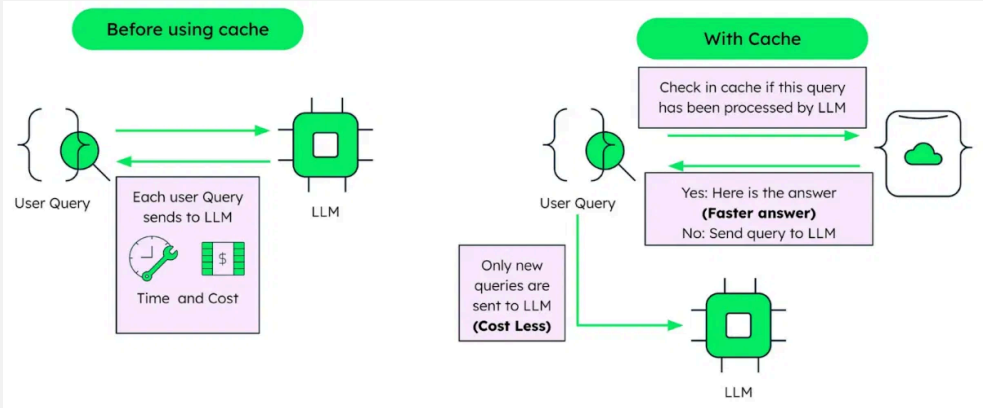
### Score Threshold (0.1)

Define a tolerância de similaridade. Quanto menor o valor, mais estrita é a comparação. Um valor de 0.1 permite variações sutis na frase.

APP.PY

```python
from langchain.cache import RedisSemanticCache
from langchain.globals import set_llm_cache

set_llm_cache(RedisSemanticCache(
    redis_url=REDIS_URL,
    embedding=embeddings,
    score_threshold=0.1
))
```




FLUXO DE OTIMIZAÇÃO

# PASSO 4: CÓDIGO DA APLICAÇÃO (PARTE 2)

Lógica de Inicialização e Análise de Performance

```python
# Configuração de Conexão
REDIS_URL = "redis://redis-stack:6379"
MODEL_NAME = "phi3"

# Inicialização do LLM (Ollama)
llm = Ollama(
 model=MODEL_NAME,
 base_url="http://ollama-service:11434"
)

# Lógica de Medição de Tempo
start = time.time()
response = llm.invoke(prompt)
duration = time.time() - start

if duration < 1.0:
 st.info(f"🚀 CACHE HIT: {duration}s")
else:
 st.warning(f"🐢 MISS: {duration}s")
```

## CONECTIVIDADE

O parâmetro `base_url` é crucial. Ele aponta para o container **ollama-service** na porta interna **11434**, permitindo que a aplicação Python converse diretamente com o motor de IA dentro da rede Docker.

## ANÁLISE DE LATÊNCIA

O código utiliza o tempo de resposta como indicador da fonte da informação. A diferença de magnitude é a prova do funcionamento do cache.

| | |
|---|---|
| **TEMPO < 1.0S** | CACHE HIT (REDIS) |
| **TEMPO > 1.0S** | MISS (OLLAMA) |

# PASSO 5: ORQUESTRAÇÃO

## DEFININDO SERVIÇOS, VOLUMES E REDES

INFRASTRUCTURE AS CODE

DOCKER-COMPOSE.YML

```
services: # 1. Banco de Dados Vetorial redis-stack:
image: redis/redis-stack:latest ports: - "6379:6379" -
"8001:8001" volumes: - redis-data:/data networks: - ai-
net # 2. Motor de IA ollama-service: image:
ollama/ollama:latest volumes: -
ollama_models:/root/.ollama networks: - ai-net # 3.
Aplicação Frontend app: build: . ports: - "8501:8501"
depends_on: - redis-stack - ollama-service networks: -
ai-net volumes: redis-data: ollama_models: networks: ai-
net:
```

### SERVIÇOS INTERCONECTADOS

Define os três pilares da aplicação. O `depends_on` garante que o frontend só inicie após o banco de dados e a IA estarem prontos.

### PERSISTÊNCIA DE DADOS

Volumes nomeados (`redis-data` e `ollama_models`) garantem que os dados do cache e os modelos de IA (que são grandes) não sejam perdidos ao reiniciar os containers.

### REDE ISOLADA

A rede `ai-net` cria um canal seguro de comunicação. Os containers se enxergam pelo nome do serviço (ex: `http://ollama-service`), sem expor portas internas para a internet.

# PASSO 6: INICIALIZAÇÃO

Subindo a Infraestrutura e Configurando a IA

1

### Subir Containers

```
$ docker compose up -d --build
```

O parâmetro `--build` força a recriação da imagem Python. Aguarde o download das imagens base.

2

### Baixar Modelo LLM

```
$ docker exec -it ollama-service ollama pull phi3
```

Executa o comando `pull` dentro do container **ollama-service**. Tamanho aprox: 2.3 GB.

**Dica de Performance:** Se sua instância tiver recursos limitados, substitua `phi3` por `tinyllama` no comando acima e atualize a variável `MODEL_NAME` no arquivo `app.py`.

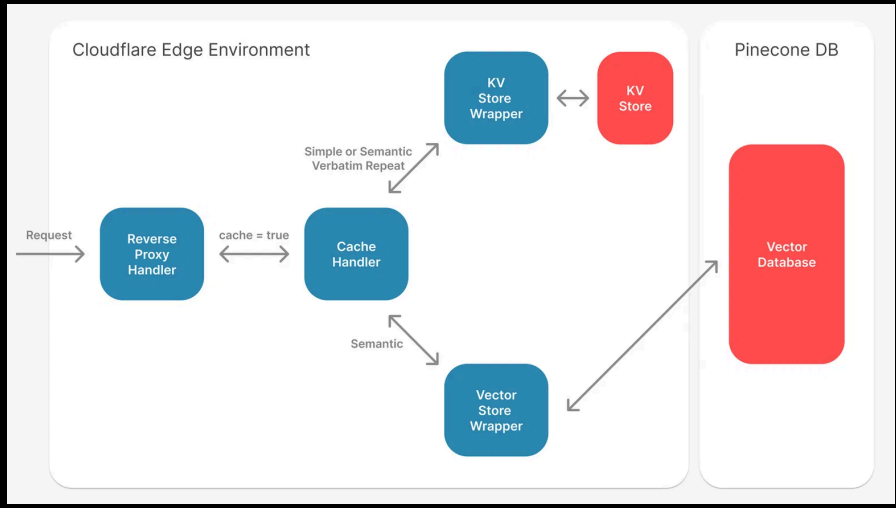
# PASSO 7: TESTANDO O SISTEMA

VALIDAÇÃO E ANÁLISE DE PERFORMANCE

| | |
|---|---|
| **STREAMLIT UI** | `http://SEU-IP-AWS:8501` |
| **REDIS INSIGHT** | `http://SEU-IP-AWS:8001` |

## Experimento de Cache

| | | |
|---|---|---|
| 1 | **Primeira Pergunta**<br>"O que são sistemas distribuídos?" | **MISS (5-10S)** |
| 2 | **Repetir Pergunta**<br>Mesma frase exata | **HIT (< 1S)** |
| 3 | **Pergunta Similar**<br>"Explique sistemas distribuídos" | **HIT (SEMÂNTICO)** |




FLUXO DE VALIDAÇÃO SEMÂNTICA

# RESULTADOS E CONCLUSÃO

## IMPACTO DO CACHE SEMÂNTICO EM PRODUÇÃO

Lab Finalizado

**95%**

REDUÇÃO DE LATÊNCIA

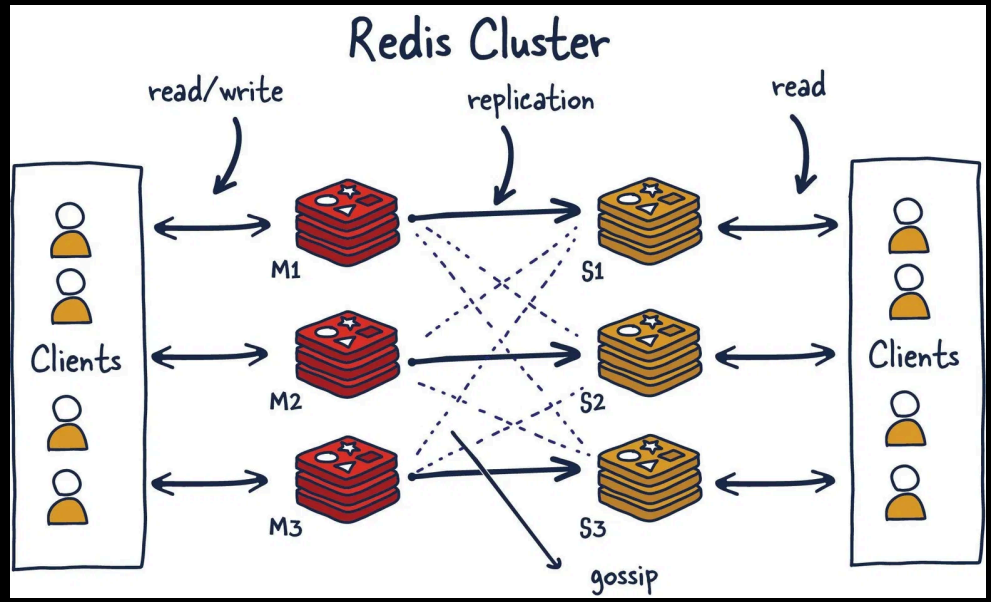
**80%**

ECONOMIA DE CPU

### APLICAÇÕES PRÁTICAS

- Chatbots Corporativos e FAQ
- Suporte Técnico Automatizado
- Assistentes Educacionais

> O laboratório demonstrou com sucesso como a arquitetura distribuída (Redis + Docker) otimiza custos e experiência do usuário em aplicações de IA Generativa.




ARQUITETURA ESCALÁVEL COM REDIS CLUSTER