Gmail **Fillipe Guerra <fillipe.backup@gmail.com>**

IA_Autonoma_Parte12

1 mensagem

Fillipe Guerra <fillipe.backup@gmail.com> 28 de outubro de 2025 às 11:16
Para: Fillipe Augusto Gomes Guerra <fillipe182@hotmail.com>, Fillipe Guerra <fillipe.backup@gmail.com>

Aqui vai o **pacote extra** que faltava pra AION ficar **PERFEITA e cada vez mais autônoma**:

1. **Extrator de Entidades local** (sem OpenAI) plugado nos eventos de *retrieval*, alimentando teu **heatmap** e melhorando a curadoria.
2. **Treino Adaptativo (LoRA/QLoRA)** disparado **automaticamente** por metas (queda de nDCG / alta de fallback) ou **manualmente** no painel.
3. **Rotas, cron, UI e configs**, tudo pronto pra colar no Replit.

Se algum arquivo conflitar com o que já te passei, **substitui** por estes (são supersets compatíveis).

✅ Extra A — Entidades locais (wink-nlp) integradas aos eventos

1) Dependências

```
pnpm add wink-nlp @wink-nlp/model-general
```

2) Utilitário de entidades

```
server/nlp/entities.ts

import winkNLP from "wink-nlp";
import model from "@wink-nlp/model-general";

const nlp = winkNLP(model);
const its = nlp.its;
const as = nlp.as;

export type EntityHit = { text: string; type: string };

export function extractEntities(text: string, top = 12): EntityHit[] {
  const doc = nlp.readDoc(text || "");
  // Entidades nomeadas + noun-phrases simples (keyphrases)
  const ents = doc.entities().out(
    its.detail as any
  ) as { value: string; type: string }[];

  const nounPhrases = doc
    .phrases()
    .filter((p) => p.out(its.type) === "NounPhrase" && p.out().length > 3)
    .out(as.array);

  const bag = new Map<string, number>();
  for (const e of ents) bag.set(e.value, (bag.get(e.value) || 0) + 3); // NER mais forte
  for (const p of nounPhrases) bag.set(p, (bag.get(p) || 0) + 1);

  return [...bag.entries()]
    .sort((a, b) => b[1] - a[1])
    .slice(0, top)
```

```
    .map(([text]) => ({ text, type: ents.find((e) => e.value === text)?.type || "NP" }));
}
```

3) Hook pra registrar *retrieval events* com entidades

Chama este helper no ponto onde você já monta os resultados de RAG (antes de responder).
`server/aion/events.ts`

```
import { db } from "../db";
import { aiEvents } from "@shared/schema.ai.metrics";
import { extractEntities } from "../nlp/entities";

export async function logRetrievalEvent(tenantId: string, query: string, topK: { text: string; rank:
number; rel?: number }[]) {
  // Extrai entidades da query + primeiro contexto concatenado
  const sample = [query, ...topK.slice(0, 3).map((x) => x.text)].join("\n\n").slice(0, 7000);
  const ents = extractEntities(sample, 20).map((e) => e.text);

  await db.insert(aiEvents).values({
    tenantId,
    kind: "retrieval",
    value: topK[0]?.rel ?? 0,
    meta: {
      q: query.slice(0, 512),
      rank: topK[0]?.rank ?? 1,
      rel: topK[0]?.rel ?? 0,
      entities: ents
    }
  });
}
```

Uso (em qualquer pipeline de resposta):

```
// depois que tiver topK (documentos/chunks) — cada item com {text, rank, rel}
await logRetrievalEvent(tenantId, userQuery, topK);
```

Pronto: teu **heatmap de entidades** (na Fase 5) passa a ficar **alimentado automaticamente** e **sem custo externo**.

✅ Extra B — Treino Adaptativo (LoRA/QLoRA) automático e sob demanda

A ideia: quando a **taxa de fallback** subir acima de `τ_f` ou **nDCG** cair abaixo de `τ_n`, a AION dispara um **job** de adaptação (LoRA) usando **teus próprios logs** (sem OpenAI).
Também adiciono **botões** no painel pra iniciar/cancelar/checar status.

1) Dependências (lado Python, se quiser rodar no mesmo Replit)

```
pip3 install torch==2.3.1 transformers peft datasets accelerate bitsandbytes --upgrade
```

2) Estrutura

```
trainer/
 └─ lora/
     ├─ run_lora.py
     └─ data_builder.py
server/training/
 ├─ routes.train.ts
 └─ watcher.ts
```

3) Builder de dados (extrai pares instrução→resposta dos teus logs)

`trainer/lora/data_builder.py`

```
import json, sys, os

def build_jsonl(events_path, out_jsonl, max_samples=20000):
    """
    Espera um JSONL com eventos de conversa/resposta locais (sem PII sensível).
    Você pode exportar de aiEvents: kind in ['answer'] com meta {prompt, reply}.
    """
    n=0
    with open(out_jsonl, "w", encoding="utf-8") as w:
        with open(events_path, "r", encoding="utf-8") as f:
            for line in f:
                try:
                    ev = json.loads(line)
                    if ev.get("kind") == "answer" and ev.get("meta"):
                        prompt = ev["meta"].get("prompt") or ev["meta"].get("q")
                        reply  = ev["meta"].get("reply")  or ev["meta"].get("a")
                        if prompt and reply:
                            w.write(json.dumps({"instruction": prompt, "output": reply},
ensure_ascii=False) + "\n")
                            n += 1
                            if n >= max_samples: break
                except Exception:
                    pass
    print(json.dumps({"written": n}))
    return n

if __name__ == "__main__":
    build_jsonl(sys.argv[1], sys.argv[2])
```

4) Script de treino QLoRA (modelo local de base — ex.: `mistral-7b-instruct`/`llama-3-8b-instruct` ONNX/gguf não é necessário aqui porque treino é PyTorch)

trainer/lora/run_lora.py

```
import argparse, json, os
from datasets import load_dataset
from transformers import AutoModelForCausalLM, AutoTokenizer, TrainingArguments,
DataCollatorForLanguageModeling, BitsAndBytesConfig
from peft import LoraConfig, get_peft_model, prepare_model_for_kbit_training
from transformers import Trainer

def format_example(ex):
    # Estilo Alpaca
    prompt = f"### Instruction:\n{ex['instruction']}\n\n### Response:\n"
    return prompt, ex["output"]

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--base_model", required=True)  # ex: meta-llama/Meta-Llama-3-8B-Instruct
    ap.add_argument("--train_jsonl", required=True)
    ap.add_argument("--out_dir", required=True)
    ap.add_argument("--r", type=int, default=16)
    ap.add_argument("--alpha", type=int, default=16)
    ap.add_argument("--dropout", type=float, default=0.05)
    ap.add_argument("--epochs", type=int, default=2)
    args = ap.parse_args()

    os.makedirs(args.out_dir, exist_ok=True)

    bnb_config = BitsAndBytesConfig(load_in_4bit=True, bnb_4bit_quant_type="nf4",
bnb_4bit_compute_dtype="bfloat16")

    model = AutoModelForCausalLM.from_pretrained(args.base_model, device_map="auto",
quantization_config=bnb_config)
    tokenizer = AutoTokenizer.from_pretrained(args.base_model, use_fast=True)
    tokenizer.pad_token = tokenizer.eos_token
```

```
    model = prepare_model_for_kbit_training(model)
    peft_conf = LoraConfig(
        r=args.r, lora_alpha=args.alpha, lora_dropout=args.dropout,
        target_modules=["q_proj","k_proj","v_proj","o_proj"], bias="none", task_type="CAUSAL_LM"
    )
    model = get_peft_model(model, peft_conf)

    dataset = load_dataset("json", data_files=args.train_jsonl)["train"]

    def tokenize(ex):
        instr, out = format_example(ex)
        text = instr + out + tokenizer.eos_token
        tok = tokenizer(text, truncation=True, max_length=1024, padding="max_length")
        tok["labels"] = tok["input_ids"].copy()
        return tok

    tokenized = dataset.map(tokenize, batched=False, remove_columns=dataset.column_names)

    collator = DataCollatorForLanguageModeling(tokenizer, mlm=False)
    args_t = TrainingArguments(
        output_dir=args.out_dir,
        per_device_train_batch_size=2,
        gradient_accumulation_steps=8,
        logging_steps=20,
        num_train_epochs=args.epochs,
        learning_rate=2e-4,
        bf16=True,
        save_steps=200,
        save_total_limit=2,
        optim="paged_adamw_8bit",
        report_to="none"
    )

    trainer = Trainer(model=model, tokenizer=tokenizer, args=args_t,
                      train_dataset=tokenized, data_collator=collator)
    trainer.train()
    model.save_pretrained(os.path.join(args.out_dir, "adapter"))
    tokenizer.save_pretrained(os.path.join(args.out_dir, "adapter"))
    print(json.dumps({"status":"ok","out":args.out_dir}))

if __name__ == "__main__":
    main()
```

Resultado: um diretório `adapter/` com o **LoRA**. Na hora de inferir, você carrega o teu modelo base local e **mescla** o adapter (ou aplica **on-the-fly** com PEFT) — sem usar OpenAI.

5) Rotas para disparar/cancelar/checar status

`server/training/routes.train.ts`

```
import type { Express, Request } from "express";
import { spawn } from "child_process";
import fs from "fs";
import path from "path";

let current: { pid: number; log: string } | null = null;

const ROOT = process.env.AION_TRAIN_ROOT || "./trainer";
const EVENTS_EXPORT = process.env.AION_EVENTS_JSONL || "./trainer/events.jsonl"; // gere a partir dos
seus aiEvents

export function registerTrainRoutes(app: Express) {
  app.post("/api/ai/train/lora/start", async (req, res) => {
    if (current) return res.status(409).json({ ok: false, msg: "job running" });

    // 1) Gera JSONL de treino a partir dos eventos (você pode exportar via query — aqui assumimos já
exportado)
    const outJsonl = path.join(ROOT, "lora", "train_data.jsonl");
```

```
    // 2) Dispara builder (opcional se você já exporta pronto)
    const builder = spawn("python3", ["./trainer/lora/data_builder.py", EVENTS_EXPORT, outJsonl]);
    let builderDone = false; let bErr = "";
    builder.stderr.on("data", (d) => (bErr += d.toString()));
    await new Promise<void>((resolve) => builder.on("exit", () => { builderDone = true; resolve();
}));

    if (!builderDone) return res.status(500).json({ ok: false, msg: "builder failed", err: bErr });

    // 3) Dispara treino
    const LOG = path.join(ROOT, "lora", "train.log");
    const base = process.env.AION_LORA_BASE || "mistralai/Mistral-7B-Instruct-v0.3";
    const out = path.join(ROOT, "lora", "out");
    const args = ["./trainer/lora/run_lora.py", "--base_model", base, "--train_jsonl", outJsonl, "--
out_dir", out];

    const p = spawn("python3", args);
    current = { pid: p.pid!, log: LOG };
    const w = fs.createWriteStream(LOG);
    p.stdout.pipe(w); p.stderr.pipe(w);

    p.on("exit", () => { current = null; });

    res.json({ ok: true, pid: p.pid, log: LOG });
  });

  app.get("/api/ai/train/lora/status", async (_req, res) => {
    res.json({ running: !!current, pid: current?.pid || null, log: current?.log || null });
  });

  app.post("/api/ai/train/lora/cancel", async (_req, res) => {
    if (!current) return res.json({ ok: false, msg: "no job" });
    try { process.kill(current.pid); } catch {}
    current = null;
    res.json({ ok: true });
  });
}
```

6) Watcher automático por metas (fallback ↑ / nDCG ↓)

server/training/watcher.ts

```
import cron from "node-cron";
import { db } from "../db";
import { aiDaily } from "@shared/schema.ai.metrics";
import fetch from "node-fetch";

const TENANT = process.env.PRIMARY_TENANT_ID!;
const TAU_F = Number(process.env.AION_TAU_FALLBACK || "0.18"); // 18% fallback-rate
const TAU_N = Number(process.env.AION_TAU_NDCG || "0.82");     // nDCG médio alvo

let coolDown = false;

export function startTrainWatcher(){
  // avalia a cada hora
  cron.schedule("0 * * * *", async ()=>{
    if (coolDown) return;
    const day = new Date().toISOString().slice(0,10);
    const rows = await db.select().from(aiDaily); // pegue do dia e de ontem para suavizar
    const last = rows.slice(-1)[0];
    if (!last) return;
    const ndcg = Number(last.ndcg || 0);
    const fr = Number(last.fallbackRate || 0);

    if (fr > TAU_F || ndcg < TAU_N){
      // dispara treino adaptativo
      try{
        await fetch("http://localhost:3000/api/ai/train/lora/start", { method:"POST" });
        coolDown = true;
```

```
        setTimeout(()=>{ coolDown=false; }, 1000*60*60*6); // 6h de resfriamento
      }catch(e){ /* log */ }
    }
  });
}
```

No bootstrap do servidor:

```
import { registerTrainRoutes } from "./training/routes.train";
import { startTrainWatcher } from "./training/watcher";

registerTrainRoutes(app);
startTrainWatcher();
```

7) Inferência com Adapter (no teu motor local)

Onde você carrega o modelo base **local** (não-OpenAI), aplica o **adapter** quando existente:

```
// pseudocódigo: server/ai/local-llm.ts
import { AutoModelForCausalLM, AutoTokenizer } from "@xenova/transformers"; // ou python-serving
import { PeftModel } from "peft"; // se rodar via Python serve; do lado Node use servidor python

// Sugestão prática: subir um microserviço Python para inferência local já com PEFT (a mesma stack do
treino).
// A rota Node chama o microserviço via HTTP, mantendo autonomia total.
```

Resumo: toda a lógica de **ação corretiva** (LoRA) está **no teu lado** e se alimenta dos **teus eventos**. Quanto mais a AION conversa, menos fallback ela precisa.

🖥️ UI — Controles no Painel

1) Botões em Telemetria (ou Configurações da IA)

Adiciona no topo de `/ui/pages/admin/ai-telemetry.tsx`:

```
function Actions() {
  const [status,setStatus]=useState<any>({});
  async function refresh(){ setStatus(await (await fetch("/api/ai/train/lora/status")).json()); }
  async function start(){ await fetch("/api/ai/train/lora/start",{method:"POST"}); await refresh(); }
  async function cancel(){ await fetch("/api/ai/train/lora/cancel",{method:"POST"}); await refresh();
}
  useEffect(()=>{ refresh(); },[]);

  return (
    <div className="flex gap-2 items-center">
      <button className="px-3 py-2 bg-emerald-600 text-white rounded" onClick={start}>Iniciar
LoRA</button>
      <button className="px-3 py-2 bg-rose-600 text-white rounded" onClick={cancel}>Cancelar</button>
      <button className="px-3 py-2 bg-gray-700 text-white rounded" onClick={refresh}>Status</button>
      <span className="text-xs text-gray-400">Job: {status.running? `em execução (pid ${status.pid})`
: "parado"}</span>
    </div>
  );
}
```

E no componente principal:

```
<h1 className="text-2xl font-bold flex items-center justify-between">
  Telemetria & Métricas (AION)
  <Actions/>
</h1>
```

2) Parâmetros de disparo automático

No painel **Configurações da IA** (Fase 2), adiciona sliders/inputs para:

- `τ_fallback` (limite de fallback-rate),
- `τ_ndcg` (limite mínimo de nDCG),
- **cooldown** (horas entre retreinos).

Persistir em tabela de settings (já criada) e ler no `watcher.ts` (ou usar `.env` como acima).

📐 Notas matemáticas de robustez (resumo didático nos comentários)

- **Aprendizado Ativo**: escolha amostras para rótulo humano quando $|C-\tau|$ é mínimo (incerteza). Isso maximiza ganho de informação.
- **Regra de Rebuild**: após ingest significativo (ΔN chunks) ou após LoRA, executar `ann.rebuild()` para manter recall alto.
- **Budget-aware Fallback**: otimizador simples

  $\tau \min \text{Cost}(\tau) = p(C<\tau)\cdot c_{fallback} + p(C\geq\tau)\cdot c_{local}$

  varrer τ semanalmente e escolher o argmin histórico.
- **LoRA Gate**: dispara se $FR > \tau_f$ ou $nDCG < \tau_n$; usa **cooldown** para evitar thrashing.

🔚 Checklist final (colar e rodar)

1. `pnpm add wink-nlp @wink-nlp/model-general` e (se quiser treino) `pip3 install ...` conforme acima.
2. Criar arquivos:
   - `server/nlp/entities.ts`, `server/aion/events.ts`
   - `trainer/lora/data_builder.py`, `trainer/lora/run_lora.py`
   - `server/training/routes.train.ts`, `server/training/watcher.ts`
3. Registrar rotas/cron no **bootstrap** do servidor.
4. Adicionar `<Actions/>` na página de Telemetria (ou nas Configurações).
5. (Opcional) Configurar `.env`: `AION_TAU_FALLBACK`, `AION_TAU_NDCG`, `AION_TRAIN_ROOT`, `AION_LORA_BASE`, `AION_EVENTS_JSONL`.

Dessa forma, a AION:

- **extrai entidades localmente** e **enriquece a telemetria**;
- **se adapta sozinha** (LoRA), reduzindo **gradualmente** a dependência de qualquer API externa;
- dá a você **controles explícitos** para iniciar, cancelar e auditar o processo — sempre **single-tenant** e **com governança**.