ChatGPT

Implementação da Evolution API v2 para Automação de WhatsApp com Docker, OpenAI e Integrações

Hoje em dia, oferecer atendimento automatizado no WhatsApp utilizando IA pode alavancar a eficiência no relacionamento com clientes. Neste artigo, detalharemos como implantar a **Evolution API v2** – uma plataforma robusta e gratuita de automação de mensagens – em uma arquitetura Docker Compose de instância única, integrando banco de dados, cache, fila de mensagens e serviços de IA. Abordaremos as configurações de **PostgreSQL e Redis** para persistência e desempenho, o uso do **RabbitMQ** para eventos assíncronos, a integração com o construtor de bots **Typebot** e com a **API OpenAI** para respostas inteligentes. Também discutiremos uma estratégia de **RAG (*Retrieval Augmented Generation*)**, combinando o histórico de mensagens do cliente e de todos os atendimentos para enriquecer as respostas da IA. Por fim, apresentamos considerações avançadas de **performance, segurança e alta disponibilidade**, e um arquivo *docker-compose.yml* de exemplo com configurações otimizadas e persistentes. Vamos começar!

Visão Geral da Evolution API v2

A Evolution API v2 é uma plataforma open-source projetada para ir além de um simples gateway de mensagens WhatsApp™. Ela foi concebida para **capacitar pequenos negócios e projetos com recursos limitados** a integrarem múltiplos serviços e automatizarem interações de forma abrangente [1] [2]. Inicialmente criada como uma API de controle do WhatsApp baseada em CodeChat/Baileys, a Evolution API evoluiu para suportar não apenas o **WhatsApp** (via API não-oficial com Baileys e também a API oficial Business), mas também integrar com ferramentas como **Typebot, Chatwoot, Dify e OpenAI** [3]. Em outras palavras, ela permite construir chatbots e fluxos inteligentes que conectam o WhatsApp a serviços de IA generativa (ChatGPT), a plataformas de atendimento ao cliente (Chatwoot), a construtores de fluxo conversacional (Typebot, Flowise, Dify), entre outros, tudo por meio de uma API unificada.

Alguns pontos de destaque da Evolution API v2:

- **API Unificada Multi-serviço:** fornece endpoints REST para gerenciar instâncias de WhatsApp, enviar/receber mensagens, além de gerenciar integrações com bots ou webhooks externos.
- **Gratuita e Aberta:** a plataforma é gratuita e pode ser auto-hospedada; seu código-fonte está disponível (repositório no GitHub) e conta com comunidade ativa [2].
- **Multicanal (Beta):** além do WhatsApp, há planos de suporte futuro para Instagram e Messenger [2].
- **Multi-instância:** é possível controlar múltiplas contas/números de WhatsApp (cada um configurado como uma "instância" separada via API). Em nosso cenário usaremos apenas uma instância inicialmente.
- **Extensibilidade via Webhooks e Filas:** a Evolution API pode emitir eventos detalhados (novas mensagens, atualizações, etc.) por Webhook ou fila (RabbitMQ/Kafka) para integrar com outros sistemas e automações em tempo real.
- **Modo Assistente AI:** suporte nativo a integração com **OpenAI** para criar assistentes virtuais no WhatsApp que respondem usando modelos GPT [4].

- **Chatbots de Fluxo:** suporte a bots externos como **Typebot** e **Flowise**, que podem ser usados para fluxos guiados ou RAG (busca em base de conhecimento) sem precisar codificar do zero.

Em resumo, a Evolution API v2 serve como o **cérebro** que conecta o WhatsApp a IA e outros serviços. Nos próximos tópicos, veremos como configurar cada componente para tirar proveito máximo dessa plataforma em um **ambiente Docker Compose** de instância única (ideal para MVPs ou início de projeto), garantindo persistência de dados e possibilidade de expansão futura.

Configuração do Banco de Dados e Cache (PostgreSQL + Redis)

Para manter um histórico consistente de conversas, contatos e outros dados importantes, usaremos um banco de dados relacional. A Evolution API suporta **PostgreSQL** ou **MySQL** como repositório de dados, utilizando o ORM Prisma para interagir com o banco [5]. Aqui optaremos pelo **PostgreSQL** por sua robustez e recursos avançados. Também utilizaremos o **Redis** como cache para melhorar a performance, conforme recomendado pela própria plataforma [6].

Banco de Dados (PostgreSQL)

Por que um DB externo? A Evolution API armazena informações críticas da aplicação no banco – instâncias configuradas, mensagens recebidas/enviadas, contatos, histórico etc. [5]. Sem um DB persistente, esses dados ficariam apenas em memória e seriam perdidos caso a API reiniciasse. Com o PostgreSQL, garantimos persistência e poderemos posteriormente fazer análises de histórico (por exemplo, geração de insights a partir de todas as conversas).

Configuração básica: Criaremos um container `postgres` usando a imagem oficial do PostgreSQL. No Docker Compose, definiremos variáveis de ambiente para setar usuário, senha e database inicial. No nosso exemplo, vamos criar um banco `evolution` para a API. É **fundamental** montar um volume no diretório de dados do Postgres (`/var/lib/postgresql/data`) para que os dados sejam preservados entre reinicializações do container.

Além disso, na Evolution API, precisamos habilitar e apontar a conexão com o DB via variáveis no arquivo `.env`. As principais configurações são:

- `DATABASE_ENABLED=true` – habilita o uso do banco de dados pela API [7].
- `DATABASE_PROVIDER=postgresql` – especifica Postgres como SGDB (poderia ser `mysql` se fosse o caso) [7].
- `DATABASE_CONNECTION_URI=` – URI de conexão com o banco, incluindo usuário, senha, host, porta e nome do DB [8]. Por exemplo:
  `postgresql://**usuario**:**senha**@postgres:5432/evolution?schema=public`
  (Note que usaremos o hostname do serviço Docker (`postgres`) em vez de `localhost`.)
- `DATABASE_CONNECTION_CLIENT_NAME=evolution_exchange` – nome do cliente de conexão para identificação (pode ser arbitrário, usado em logs e monitoramento) [9].
- Opções de persistência de dados: a Evolution API permite escolher quais dados salvar no banco. Para o nosso caso de uso (RAG e insights), é interessante salvar **tudo**, incluindo instância, mensagens, contatos, chats, etiquetas e histórico, configurando as variáveis a seguir como `true`: `DATABASE_SAVE_DATA_INSTANCE`, `DATABASE_SAVE_DATA_NEW_MESSAGE`, `DATABASE_SAVE_MESSAGE_UPDATE`, `DATABASE_SAVE_DATA_CONTACTS`, `DATABASE_SAVE_DATA_CHATS`, `DATABASE_SAVE_DATA_LABELS`, `DATABASE_SAVE_DATA_HISTORIC` [10]. Com isso, **todas as mensagens e eventos relevantes serão armazenados** no PostgreSQL, permitindo consultas futuras.

> **Dica:** A configuração acima garante que *todas* as mensagens (novas ou editadas), bem como contatos e conversas, fiquem salvas. Isso é perfeito para implementar o RAG posteriormente, pois teremos um repositório completo de conversas passadas para buscar informações relevantes.

Criação inicial do DB: O container do Postgres configurado com `POSTGRES_DB=evolution` já cria automaticamente o database. Alternativamente, poderíamos criar manualmente conforme a documentação (ex.: rodando `createdb evolution` no psql) [11], mas não será necessário aqui.

Cache (Redis)

Usaremos o Redis como um **cache in-memory** para acelerar operações e aliviar carga do banco. A Evolution API utiliza o Redis principalmente para armazenar dados temporários e melhorar a velocidade de acesso a informações de uso frequente [6]. Exemplos incluem cache de sessões do WhatsApp (se configurado), status de presença, ou simplesmente para guardar resultados de consultas pesadas momentaneamente.

No Docker Compose, adicionaremos um serviço `redis` usando a imagem oficial do Redis. Podemos expor a porta 6379 apenas internamente (não é necessário acesso externo). É recomendável montar um volume para o Redis somente se quisermos persistir dados de cache (pode ser útil caso o Redis também seja usado para armazenar sessões ou filas que não queremos perder em um restart). Contudo, por padrão, usaremos o Redis apenas como cache volátil – se reiniciado, a API reconstruirá os dados essenciais conforme necessário.

Configurações no `.env` da Evolution API para habilitar o cache Redis:

- `CACHE_REDIS_ENABLED=true` – ativa o uso do Redis [12].
- `CACHE_REDIS_URI=redis://redis:6379/6` – URI de conexão com o Redis, apontando para o host do serviço Docker (`redis`). Note que utilizamos o **database 6** do Redis no URI (`/.../6`) apenas para isolar os dados da aplicação (poderia ser 0 ou outro índice; adotamos 6 conforme exemplo da documentação) [13].
- `CACHE_REDIS_PREFIX_KEY=evolution` – prefixo para chaves no Redis, caso um mesmo servidor Redis seja compartilhado com outras aplicações, evitando colisões de keys [14].
- `CACHE_REDIS_SAVE_INSTANCES=false` – mantém *false* para **não** salvar informações de credenciais de conexão do WhatsApp no Redis [15], já que optamos por persistir esses dados no PostgreSQL (via `DATABASE_SAVE_DATA_INSTANCE=true`). Assim, as credenciais de sessão do WhatsApp serão armazenadas no DB e no disco (explicado adiante), garantindo que uma queda não desconecte permanentemente o WhatsApp.
- `CACHE_LOCAL_ENABLED=false` – desabilita cache local em memória da API (vamos usar apenas o Redis como cache) [16].

Observação: Caso futuramente deseje-se maior resiliência das sessões do WhatsApp, poderíamos setar `CACHE_REDIS_SAVE_INSTANCES=true` para manter as credenciais no Redis (que, diferente do banco, é consultado mais rapidamente). Porém, isso exige um Redis persistente; como nosso Postgres já é persistente, manteremos as instâncias lá mesmo.

Persistência e Volume de Instâncias WhatsApp

Além do banco de dados, **é crucial persistir os dados de autenticação da instância do WhatsApp** (QR code, token de sessão, etc.). A Evolution API, quando usada com a biblioteca Baileys, armazena

arquivos de sessão em disco (no container). Para não precisar escanear o QR code toda vez que o contêiner sobe, montaremos um volume na pasta de instâncias do container Evolution API.

Segundo a documentação oficial, deve-se montar um volume em `/evolution/instances` dentro do container da API [17]. Dessa forma, mesmo que o container seja recriado, os arquivos de sessão do WhatsApp (relacionados ao número conectado) serão preservados. Iremos configurar isso no nosso compose (volume nomeado `evolution_instances`).

Gerenciamento de Eventos com RabbitMQ

Para tornar nossa solução escalável e reativa, incluiremos o **RabbitMQ** como um serviço de mensageria. A Evolution API integra-se ao RabbitMQ para **publicar eventos** do sistema em filas, o que possibilita processar tarefas de forma assíncrona e descentralizada [18]. Por exemplo, sempre que chega uma mensagem nova, a API pode publicar um evento em uma fila RabbitMQ; podemos ter consumidores separados (microserviços ou funções) ouvindo essas filas para **gerar insights**, **registrar logs** ou **acionar integrações adicionais** sem bloquear o fluxo principal.

Configuração do RabbitMQ na Evolution API

Optaremos por configurar o RabbitMQ no modo **global**, ou seja, com filas por tipo de evento ao invés de filas separadas por instância [19] [20]. Isso simplifica o gerenciamento: todos os eventos "message.received" por exemplo vão para uma única fila comum, facilitando um consumidor único processá-los de forma unificada, independentemente da instância de origem (no nosso caso inicial, só teremos uma instância de qualquer forma).

No `.env` da Evolution API, as principais variáveis para habilitar o RabbitMQ são:

- `RABBITMQ_ENABLED=true` – ativa a integração com RabbitMQ [21].
- `RABBITMQ_URI=amqp://admin:admin@rabbitmq:5672/default` – string de conexão ao RabbitMQ, incluindo credenciais, host e vhost. Usaremos `admin:admin` como usuário e senha padrão (configuraremos isso no container) e um virtual host chamado `default` [21].
- `RABBITMQ_EXCHANGE_NAME=evolution_exchange` – nome do *exchange* a ser utilizado pela Evolution API para publicar eventos [22]. Esse exchange será do tipo `topic` ou similar, conforme implementação interna, e as filas serão ligadas a ele.
- `RABBITMQ_GLOBAL_ENABLED=true` – habilita o modo global de filas unificadas por evento [22].

Além disso, a API permite selecionar quais eventos serão efetivamente publicados no RabbitMQ. Há diversas flags para cada tipo de evento (startup, novas mensagens, atualização de mensagens, contatos, chats, etc.) [20] [23]. Por padrão, se não especificarmos, muitos deles ficam `false` (desabilitados). **Para começar**, podemos habilitar os eventos mais importantes, como por exemplo:

- `RABBITMQ_EVENTS_MESSAGES_SET=true` (novo lote de mensagens recebidas) [24]
- `RABBITMQ_EVENTS_MESSAGES_UPSERT=true` (mensagens recebidas em tempo real) [24]
- `RABBITMQ_EVENTS_SEND_MESSAGE=true` (evento de envio de mensagem) [25]

E outros conforme a necessidade. No nosso caso de uso (automação de atendimento e análise de conversas), eventos relacionados a mensagens são os principais. Habilitando-os, a cada mensagem de cliente recebida ou resposta enviada, um evento correspondente será enfileirado – o que nos permite, por exemplo, ter um serviço separado consumindo essas filas para alimentar um banco de conhecimento de RAG ou gerar alertas.

> **Nota:** Habilitar eventos desnecessários consome recursos do RabbitMQ e da própria API, então ative apenas o que for usar. É possível posteriormente adicionar/remover eventos individualmente; inclusive, a Evolution API fornece um endpoint específico para configurar eventos por instância via API REST [26] [27], mas no modo global isso não é necessário.

Configuração do Serviço RabbitMQ (Docker)

No nosso Compose, adicionaremos o serviço `rabbitmq` usando a imagem oficial do RabbitMQ. Faremos algumas configurações importantes:

- **Credenciais e VHost:** Usaremos variáveis de ambiente do container para criar um usuário admin com senha admin, e um virtual host `default` já no startup. (Ex.: `RABBITMQ_DEFAULT_USER=admin`, `RABBITMQ_DEFAULT_PASS=admin`, `RABBITMQ_DEFAULT_VHOST=default`). Assim, a URI configurada acima já estará válida.
- **Persistência:** Vamos montar um volume em `/var/lib/rabbitmq` para persistir as filas e mensagens do RabbitMQ. Desta forma, se o container reiniciar, **não perderemos mensagens que não foram consumidas ainda**.
- **Portas:** Podemos expor a porta 5672 (protocolo AMQP) internamente para comunicação com a Evolution API. Opcionalmente, poderíamos expor a porta de management (15672) se quisermos acessar a interface web do RabbitMQ para monitoramento, mas neste artigo focaremos na operação headless via Docker.

Com RabbitMQ em ação, teremos uma arquitetura bem desacoplada: a Evolution API publicará eventos, e poderemos futuramente plugar consumidores para **processamento assíncrono**, como por exemplo um worker de IA que gera resumos das conversas ou extrai insights de leads.

Integração com Bots: Typebot e OpenAI

Agora entraremos no componente central da nossa automação: os **bots de atendimento ao cliente**. Vamos abordar duas abordagens que a Evolution API suporta – **Typebot** (bots de fluxo pré-definido) e **OpenAI** (assistente com IA generativa) – e como podemos utilizá-las, inclusive em conjunto.

Bots de Fluxo com Typebot

O **Typebot** é uma plataforma (open-source) de construção de chatbots conversacionais através de fluxos e formulários. Integrar o Typebot à Evolution API permite automatizar interações no WhatsApp com base em gatilhos (triggers) configurados, delegando ao Typebot a lógica das respostas [28]. Na prática, podemos criar um bot no Typebot (por exemplo, um fluxo de qualificação de lead ou um FAQ guiado) e configurar a Evolution API para acionar esse bot quando determinadas condições forem atendidas.

Como funciona a integração? Na Evolution API, cada instância WhatsApp pode ter bots associados. No caso do Typebot, há um endpoint `/typebot/create/{instance}` em que registramos as configurações do bot, incluindo: URL da API do Typebot, identificador público do bot, tipo de disparo (todas mensagens, por palavra-chave, etc.), expressão de gatilho, mensagem de saída, tempo de expiração de sessão, etc. [29] [30]. Uma vez configurado, a API monitora as mensagens recebidas. Quando uma mensagem atende ao *trigger* definido, o Evolution API passa a redirecionar as mensagens daquele contato para o bot do Typebot, e devolve ao WhatsApp as respostas geradas pelo bot, tudo automaticamente.

Configurações importantes do bot Typebot:

- **Trigger:** Podemos definir `triggerType` como `"keyword"`, `"all"` ou `"none"` [31]. Por exemplo, se quisermos que o bot atenda *todas* as mensagens, usaríamos `triggerType: "all"`. Ou podemos ativá-lo somente quando a mensagem contém certa palavra-chave/regex (como no exemplo em que o triggerValue é `^atend.*` para pegar quem digitou "atendimento") [32].
- **Operador de trigger:** (`triggerOperator`) se for keyword, podemos escolher contains, equals, regex etc. [33].
- **Expire:** tempo em minutos de inatividade após o qual a sessão do bot com aquele usuário expira [34]. Ex.: `expire: 20` minutos.
- **keywordFinish:** uma palavra-chave para o cliente digitar e encerrar o bot manualmente [35] (ex.: "#SAIR" para sair do fluxo).
- **unknownMessage:** mensagem enviada se o bot não entender a entrada do usuário [36] (uma espécie de fallback dentro do fluxo).
- **delayMessage:** delay em ms para simular "digitando" antes do bot responder [36] – isso deixa a interação mais humanizada.
- **Debounce de mensagens:** o parâmetro `debounceTime` define um intervalo (em segundos) para **agrupar múltiplas mensagens do usuário em uma só interação** [37]. Por exemplo, `debounceTime: 10` faz a API aguardar até 10 segundos após a última mensagem recebida antes de enviar o pacote de mensagens ao bot. Isso é útil porque muitas vezes o usuário manda várias mensagens fragmentadas; com debounce, evitamos acionar o bot para cada uma isoladamente e esperamos ele "terminar de digitar" para processar tudo de uma vez.

Esses parâmetros fornecem um **alto grau de controle** sobre como e quando o Typebot entra em ação. Podemos ter vários bots configurados com gatilhos diferentes (por exemplo, um bot de FAQ que ativa quando usuário menciona "preço" ou "horário", outro bot de coleta de dados que ativa quando agente humano está offline, etc.). No nosso escopo, poderíamos configurar um único bot Typebot para fluxo básico de atendimento ou triagem inicial do cliente.

Hospedando o Typebot: Podemos usar a versão cloud do Typebot (app.typebot.io) e apenas apontar a URL e ID do bot. Contudo, preferiremos **auto-hospedar o Typebot** junto à nossa stack Docker para ter controle total (e evitar limitações da versão gratuita na nuvem). O Typebot self-hosted consiste em dois serviços Docker: o **builder** (onde criamos/configuramos os bots via interface web) e o **viewer** (que fornece a interface de execução do bot, usada via API). No nosso Compose, incluiremos ambos:

- `typebot-db`: container PostgreSQL exclusivo para o Typebot (ele precisa de um banco para armazenar bots, usuários, etc.). Usaremos um segundo banco Postgres para isolá-lo do banco da Evolution API (embora poderíamos usar o mesmo servidor com DBs separados, manteremos distinto por simplicidade). Chamaremos o DB de `typebot` e definiremos senha/usuario.
- `typebot-builder`: container do builder (imagem `baptistearno/typebot-builder`). Este normalmente roda na porta 3000. Mapearmos para uma porta acessível (ex.: 8080 do host para builder).
- `typebot-viewer`: container do viewer (imagem `baptistearno/typebot-viewer`). Também roda na porta 3000 internamente, então mapearmos para outra porta do host, digamos 8081.
- Os dois containers do Typebot dependerão do `typebot-db` estar pronto (faremos `depends_on` com condição de saúde).
- Montaremos um volume para o DB do Typebot (armazenar bots e resultados).
- Precisaremos fornecer um arquivo `.env` ou variáveis de ambiente para o Typebot. Principais configs:

- `SECRET_KEY`: chave secreta de 32 caracteres usada para criptografar dados sensíveis (gerar via `openssl rand -base64 24` como sugerido na doc oficial).
- `ADMIN_EMAIL`: email que será utilizado para criar a conta admin automaticamente (quem se registrar com esse email ganhará plano de admin). Deve-se usar esse email no primeiro login.
- `DEFAULT_WORKSPACE_PLAN=UNLIMITED`: garante que novos workspaces (incluindo o do admin acima) não fiquem limitados ao plano Free de 200 chats [38].
- Configurações de banco para o builder se conectar: podemos usar variáveis padrão do Postgres (ex: `PGHOST=typebot-db`, `PGDATABASE=typebot`, `PGUSER=postgres`, `PGPASSWORD=<senha>`). O compose oficial do Typebot já cria o DB e executa migrações automaticamente ao subir [39], contanto que essas vars estejam acessíveis.
- Opcional: configuração de provedor de autenticação (o Typebot suporta login via Google/GitHub etc., mas podemos ignorar por ora e usar email/senha).
- Opcional: se quisermos enviar emails (para convites, reset de senha), poderíamos configurar SMTP, mas manteremos simples (o admin pode usar a conta criada sem precisar enviar convites).

Uma vez que o Typebot esteja rodando, podemos acessar o **builder** via navegador (ex.: `http://localhost:8080`) para criar nossos bots. Crie um bot relevante (por exemplo, um fluxo para responder dúvidas frequentes). Depois, copie o *ID público* do bot (um código tipo `my-typebot-uoz1rg9`) e configure na Evolution API via endpoint ou diretamente no banco. Podemos também configurar via API REST: enviando um POST para `/typebot/create/instancia` com JSON como no exemplo (habilitado, URL do Typebot, nome do bot, triggers, etc.) [30]. Esse passo final integra de fato o bot: a partir daí, ao receber mensagens, a Evolution API chamará a API do Typebot (no viewer) para obter a próxima resposta conforme o fluxo definido.

Caso não seja necessário um bot de fluxo... Às vezes, podemos decidir **não usar o Typebot** se quisermos focar apenas em respostas com IA generativa (livres, não roteirizadas). No próximo tópico, exploraremos a integração com OpenAI e como implementar isso possivelmente sem um bot externo, usando a Evolution API para orquestrar tudo.

Assistente de IA com OpenAI (ChatGPT)

A Evolution API v2 oferece integração nativa com o OpenAI, permitindo criar assistentes virtuais que usam modelos como GPT-3.5 ou GPT-4 para responder aos clientes no WhatsApp [4]. Em outras palavras, a própria Evolution API pode atuar como "ponte" entre o WhatsApp e a API do ChatGPT, sem necessidade de um servidor intermediário para chamadas de IA.

Visão geral da integração: Assim como no caso do Typebot, há um sistema de bot/trigger. Precisamos primeiro **cadastrar as credenciais da API OpenAI** na Evolution API e depois configurar um "bot de IA" associando uma estratégia (modelos e prompts) a certos triggers. Os passos principais são:

1. **Cadastrar credencial OpenAI:** via endpoint `POST /openai/creds/{instance}` fornecemos um nome e a API Key secreta da OpenAI [40] [41]. A Evolution API armazena a chave (com segurança) e retorna um `openaiCredsId` (um ID interno).
2. **Criar bot OpenAI:** via `POST /openai/create/{instance}`, enviamos a configuração do bot [42] [43]. Aqui definimos:
3. `enabled: true` (ativado),
4. `openaiCredsId`: referenciando a credencial cadastrada acima [44],
5. `botType`: pode ser `"assistant"` ou `"chatCompletion"` [45].
    - *Assistant* refere-se a usar um assistente definido pelo OpenAI (por exemplo, se tivermos um ID de assistente finetunado ou função específica).

- *ChatCompletion* é o modo livre, onde passamos modelo, mensagens de sistema, etc. – será nossa escolha para usar o GPT-4 ou GPT-3.5 diretamente.

6. Se `botType: "chatCompletion"`, especificamos:
   - `model`: ex `"gpt-4"` ou `"gpt-3.5-turbo"` [46],
   - `systemMessages`: um array de mensagens de sistema (instruções de contexto, personalidade do assistente, etc.) [47],
   - `assistantMessages`: mensagem(s) iniciais do assistente (opcional, por ex. uma saudação) [47],
   - `userMessages`: exemplo de mensagem do usuário (opcional, para contexto ou few-shot learning) [48],
   - `maxTokens`: limite de tokens na resposta [49].
7. Opções de trigger e comportamento: muito similares às do Typebot:
   - `triggerType` e `triggerValue` – podemos usar `"all"` para que o bot OpenAI atenda todas as mensagens (exceto as que outros bots capturem) [50].
   - `expire`: tempo em minutos para expirar a sessão de conversa da IA [51] (funciona como tempo de "reset" de contexto; e.g. 20 min sem mensagens, o próximo input inicia um novo contexto do zero).
   - `keywordFinish`: se quiser uma palavra para o usuário encerrar a conversa com a IA (pode ser algo como "#SAIR") [52].
   - `delayMessage`: delay em ms antes de enviar a resposta (para simular digitando) [53].
   - `unknownMessage`: texto enviado se por algum motivo a IA não conseguir gerar resposta [54].
   - `listeningFromMe` / `stopBotFromMe`: similares ao Typebot, indicam se o bot ouve e/ou para quando **nós (operadores)** enviamos algo pelo próprio WhatsApp [55]. No nosso caso, como é 100% automatizado, pode ficar `false`.
   - `keepOpen`: se `false`, uma vez expirado tempo ou encerrado, nova mensagem inicia nova sessão (padrão). Se `true`, o bot nunca "reseta" para aquele contato, mantendo contexto indefinidamente [56]. Podemos deixar `false` para evitar que o contexto cresça ilimitado.
   - **Debounce (`debounceTime`):** novamente aparece essa opção, crucial para nosso cenário. Recomendamos definir, por exemplo, `debounceTime: 30` (segundos) ou mais [57]. Isso fará a Evolution API esperar até 30s após a última mensagem de um cliente antes de consultar a OpenAI, permitindo que o usuário mande várias mensagens seguidas (como detalhes da pergunta) e a IA receba tudo de uma vez. *(Você pode ajustar entre ~30 segundos a 2 minutos conforme o padrão de comportamento dos clientes; 30s é um bom começo para agrupar mensagens sem causar muita espera desnecessária.)*
   - `ignoreJids`: lista de contatos para os quais **não** ativar o bot [57] – por exemplo, números de atendentes humanos ou testers internos que não devem ser respondidos pela IA.

Uma vez criado e habilitado o bot OpenAI, a Evolution API irá monitorar as mensagens conforme o trigger. Se `triggerType: "all"`, praticamente **toda mensagem recebida do cliente** irá acionar o bot de IA (a não ser que algum outro bot com trigger específico seja configurado e detectado primeiro, ou se o bot estiver pausado).

E o contexto da conversa? A Evolution API gerencia **sessões de conversa com a IA**. Enquanto a sessão estiver aberta (status "opened"), ela provavelmente mantém o histórico de trocas recente para enviar ao modelo, permitindo conversas multi-turno. Podemos pausar ou fechar a sessão via endpoint `/openai/changeStatus/{instance}` passando o `remoteJid` do contato e status "paused"/"closed" [58] [59] – útil caso um atendente humano assuma o atendimento e queira que a IA

pare de responder naquele chat, por exemplo. No nosso caso, podemos deixar isso automático via expire ou controle manual se integrarmos com um sistema externo.

Além disso, podemos definir configurações padrão globais via `/openai/settings/{instance}` – por exemplo, definir um `openaiIdFallback` (um bot de fallback caso nenhum trigger específico ative) e habilitar `speechToText` [60] [61]. O `speechToText: true` é interessante: se ativado, **áudios recebidos serão automaticamente transcritos em texto** usando o Whisper (via OpenAI) e incluídos no payload do webhook [62] [63]. Isso poderia ser usado para permitir que a IA "entenda" áudios de voz dos clientes e responda em texto.

Para não nos estendermos demais, vamos supor que configuramos um assistente OpenAI usando o modelo GPT-4, com um prompt de sistema adequado (por exemplo: "Você é um assistente de suporte que atende os clientes de [Sua Empresa]. Seja educado e forneça informações úteis baseadas nas conversas anteriores.") e deixamos ele ativo para todas as mensagens de clientes.

Estratégia RAG: Aproveitando o Histórico de Mensagens

Uma limitação dos grandes modelos de linguagem é que seu conhecimento pode ser genérico e desconectado das particularidades do seu negócio. **É aqui que entra o RAG (Retrieval Augmented Generation)** – uma técnica para tornar as respostas da IA mais relevantes, "alimentando" o modelo com informações contextuais extraídas de uma base de conhecimento.

No contexto do nosso atendimento via WhatsApp, podemos identificar duas fontes de conhecimento valiosas para melhorar as respostas da IA:

- **Mensagens recentes da conversa atual:** já vimos que a Evolution API pode manter uma sessão com histórico recente para contexto. Garantir que as últimas mensagens do cliente (e do bot) estejam presentes na chamada ao modelo ajuda a IA a não perder o fio da meada e a evitar repetir perguntas. O próprio mecanismo de sessão do bot OpenAI cuida disso até certo ponto. Com `debounceTime` configurado, nos certificamos de enviar o bloco completo das últimas mensagens do cliente juntos. E com `expire` configurado razoavelmente, permitimos certa continuidade. Portanto, para o *RAG de curto prazo*, a configuração de sessão da Evolution API aliada ao armazenamento das mensagens no PostgreSQL já nos dá suporte.
- **Histórico completo de conversas e dados da empresa:** aqui falamos de *RAG de longo prazo*. Ao longo do tempo, teremos no banco PostgreSQL um volume grande de mensagens de diversos clientes (graças às variáveis `DATABASE_SAVE_DATA_*=true` que ativamos). Podemos extrair daí informações úteis: por exemplo, perguntas frequentes que já foram respondidas manualmente, detalhes de produtos, feedbacks, etc. Em vez de treinar um modelo (custo alto), podemos indexar essas mensagens em uma base de conhecimento pesquisável.

Como implementar isso? Uma abordagem comum é utilizar técnicas de **Embeddings + Vector Store**. Poderíamos, periodicamente, **gerar embeddings das mensagens armazenadas** (usando, por exemplo, a API de embeddings da OpenAI ou um modelo local) e armazená-las em um banco vetorial (pode ser uma tabela com pgvector no PostgreSQL, ou um serviço dedicado como Pinecone, Weaviate, etc.). Então, quando a IA for responder a uma nova pergunta, executamos uma busca por similaridade: procuramos no histórico quais mensagens (ou respostas anteriores) são semanticamente próximas à pergunta atual do cliente. Os resultados (digamos, os top 3 mais relevantes) podem ser fornecidos ao modelo na prompt (por exemplo, como part of system message: "Considere também as seguintes informações...").

Essa abordagem permite que a IA se baseie **em conhecimento específico acumulado**: tanto conteúdo fornecido por atendentes humanos em conversas passadas quanto dados cadenciados (podemos também incorporar documentos externos – manuais, políticas, etc. – ao índice de vetores, tornando o bot capaz de responder com dados da empresa).

Fluxo resumido do RAG no atendimento:

1. Cliente envia mensagem(s) -> Evolution API (com debounce) aguarda quieto.
2. Após X segundos sem novas mensagens do cliente, Evolution API consolida as últimas mensagens recebidas e gera um evento (via RabbitMQ, por exemplo, um evento `MESSAGES_SET` ou `SEND_MESSAGE` pedido de resposta).
3. Um **serviço externo de RAG** (que podemos desenvolver) consome esse evento. Ele então:
4. Toma a mensagem do cliente (ou as últimas N mensagens dele na conversa atual) como query.
5. Consulta o **índice vetorial** de conhecimento para encontrar textos relevantes (por exemplo, respostas de casos parecidos, informações correlatas).
6. Monta um prompt para o OpenAI combinando o contexto da conversa atual + os trechos encontrados no passo anterior (por exemplo: "O cliente perguntou: {mensagem atual}. Consulte as informações a seguir antes de responder: {trechos relevantes}. Responda de forma completa...").
7. Chama a API do OpenAI (diretamente, ou poderia até usar o próprio endpoint do Evolution API se o bot OpenAI estiver configurado – mas possivelmente melhor chamar direto para ter controle do prompt).
8. Recebe a resposta gerada e, via Evolution API, envia de volta essa mensagem ao cliente (usando o endpoint `/message/send` da Evolution API, que encaminha pelo WhatsApp).
9. Evolution API entrega a mensagem ao WhatsApp do cliente.

Apesar de parecer complexo, a infraestrutura que montamos facilita muito isso: o RabbitMQ entrega eventos em tempo real, o PostgreSQL guarda todo o histórico (que pode ser indexado por outra ferramenta), e a integração com OpenAI está disponível. **Uma alternativa no-code** seria usar o **Flowise**, que a Evolution API suporta nativamente. O Flowise permite construir fluxos tipo LangChain de forma visual – você pode, por exemplo, configurar um bot Flowise que ao ser acionado pega a pergunta, busca em um vetor de conhecimento (ele suporta conectar com bases de dados, arquivos PDFs, etc.) e formula uma resposta com GPT-4. A Evolution API integra com Flowise de forma semelhante ao Typebot [64], ou seja, poderíamos simplesmente apontar para um fluxo criado no Flowise. Isso evitaria desenvolver do zero o serviço de RAG – basta manter o Flowise alimentado com os dados (atualizando a base vetorial quando novas mensagens chegam, possivelmente via webhook ou script regular).

De qualquer modo, implementar RAG é um passo avançado que depende de já termos acumulado mensagens e de configurarmos essa camada de busca semântica. Para os primeiros testes locais, podemos iniciar sem ele, somente com as respostas diretas da IA. Conforme as conversas se acumulam, podemos então ativar o módulo de RAG para **maximizar o aproveitamento de leads e melhorar continuamente o atendimento**, usando as mensagens salvas como fonte de verdade para a IA.

Considerações Avançadas: Performance, Segurança e Alta Disponibilidade

Nossa solução integrada envolve diversos componentes (API, DB, cache, fila, serviços de bot). A seguir discutimos boas práticas para garantir **alto desempenho, segurança** dos dados e **disponibilidade** do serviço, mesmo conforme ele cresce em escala.

Desempenho e Escalabilidade

- **Cache Redis e otimizações:** já habilitamos o Redis para reduzir acessos ao banco. Certifique-se de dimensionar adequadamente a memória do Redis para que caiba os dados de cache mais frequentes. Monitorar métricas como hits/misses de cache pode orientar ajustes (por exemplo, habilitar `CACHE_LOCAL_ENABLED` se a latência de rede ao Redis for significativa, embora em docker local não seja). Além disso, a Evolution API suporta **Kafka** como alternativa ao RabbitMQ para alto volume de eventos (até integrou no v2.3.4) – se precisar processar eventos em escala massiva, Kafka pode ser considerado, mas para nosso caso RabbitMQ basta.
- **Banco de Dados:** Use índices no Postgres para consultas frequentes (a Prisma já define alguns índices úteis, mas para RAG talvez seja necessário índice full-text ou vetorial – podemos usar a extensão **pgvector** se optar por armazenar embeddings no próprio Postgres). Realize tunings básicos no Postgres (buffers, workers) conforme volume de dados crescer. Em produção, separar o DB em servidor dedicado pode melhorar latência.
- **Escalando instâncias da API:** A Evolution API pode rodar múltiplas instâncias em paralelo, mas atenção: se estiver usando a integração **Baileys** (WhatsApp não-oficial), apenas uma instância por número deve estar ativa para evitar conflitos de sessão. Entretanto, se usar a **API oficial do WhatsApp Business (Cloud API)**, você pode horizontalizar chamadas pois o WhatsApp Cloud API é stateless. A documentação menciona Docker Swarm para rodar 2+ contêineres em paralelo quando se precisa de escalabilidade ou redundância [65] . Em modo Swarm, você poderia ter n réplicas do serviço Evolution API atendendo a diferentes instâncias (ou até a mesma instância em modo ativo/standby dependendo do caso). Para isso, habilita-se o modo global do RabbitMQ (como fizemos) para centralizar eventos e um banco único para compartilhar estado. Em resumo, inicialmente rodamos standalone (1 máquina) [66] , mas a plataforma está pronta para escalar.
- **Filas e consumidores:** Ao usar RabbitMQ, podemos distribuir consumidores de eventos em outras máquinas ou serviços conforme a carga. Por exemplo, se muitos eventos de mensagem precisam ser processados por IA, pode-se rodar vários workers paralelos lendo da fila de mensagens para gerar respostas mais rapidamente a múltiplos clientes simultâneos.
- **Monitoramento:** Em termos de performance, implemente monitoramento: use a UI do RabbitMQ para ver filas, métricas do Redis (expor `redis:alpine` no 9121 para Prometheus, por ex.), logs do Evolution API (atentar a erros ou lentidão em chamadas OpenAI). A própria Evolution API não forneceu métricas prontas, mas você pode monitorar o container (CPU, memória) e eventualmente habilitar WebSocket ou webhooks de erro (`WEBHOOK_EVENTS_ERRORS`) para capturar exceções [67] .

Segurança

- **Autenticação da API: Nunca exponha a Evolution API sem proteção.** Utilize a chave de API global para proteger os endpoints. No nosso `.env` definimos `AUTHENTICATION_API_KEY` com um valor secreto [68] ; isso requer que todos os requests à API forneçam este chave (por header ou query param conforme doc) para serem autorizados. Assim, só seus sistemas internos ou front-ends autorizados podem usar a API – evita acesso indevido de terceiros.

- **Segredos e Keys:** Armazene as chaves sensíveis (.env do Evolution com API key, .env do Typebot com SECRET_KEY, credencial da OpenAI cadastrada) em local seguro. Em produção, use um cofre de segredos ou pelo menos não versionar esses arquivos.
- **Criptografia:** Use SSL nos pontos necessários. O WhatsApp Cloud API (se utilizado) requer webhook HTTPS e envio via servidores Facebook – mas se você usar Baileys (não-oficial), a conexão já é criptografada internamente pelo protocolo. No entanto, quando acessar a interface do Typebot (builder) ou Evolution API, faça-o de preferência via HTTPS. Você pode colocar um **proxy reverso** (Nginx, Traefik ou Caddy) na frente dos serviços web (Evolution API porta 8080, Typebot builder 8080/ viewer 8081) para prover HTTPS e autenticação adicional se preciso. A documentação do Typebot, por exemplo, traz exemplos de uso do Caddy para expor os subdomínios do builder e viewer com certificados HTTPS [69] [70].
- **Controle de Acesso:** No WhatsApp Business API oficial, se fosse o caso, tenha cuidado com o token e webhook. Na Evolution API com Baileys, restrinja quem pode enviar comandos (como instanciar um novo bot, etc.) via sua API key. Além disso, para evitar que o bot OpenAI gere respostas inadequadas, incorpore no `systemMessage` guidelines de moderação conforme a política da empresa (e você pode ativar filtros de conteúdo nas respostas usando a própria API da OpenAI para classificação, se necessário).
- **Atualizações e Patches:** Manter a Evolution API atualizada é importante para segurança e melhorias [71]. Por isso, incluiremos no Compose o serviço **Watchtower**, que monitora imagens Docker em execução e aplica updates automaticamente. Com o Watchtower, se uma nova versão da Evolution API for lançada, ele fará pull da imagem e recriará o container com downtime mínimo. *Nota:* Em produção, convém fixar versões e atualizar manualmente após testes, para evitar surpresas [72], mas em ambiente de desenvolvimento ou se você quiser sempre as últimas features, o Watchtower é útil. Configuraremos o Watchtower para limpar imagens antigas e fazer checagens periódicas (por exemplo, a cada 5 minutos).
- **Backup:** Garanta backup periódico do PostgreSQL (pelo menos diário ou em tempo real via WAL shipping) – como temos volume nomeado, podemos usar um container de backup ou uma solução externa. As mensagens no WhatsApp podem conter dados sensíveis, então esses backups devem ser armazenados com segurança (criptografados, se possível). Se usar MinIO/S3 para mídia, faça backup do bucket também.

Alta Disponibilidade e Resiliência

- **Reinício Automático e Health-checks:** Configuremos todos os containers com `restart: always` para que reiniciem automaticamente em caso de falha. Além disso, definimos *healthchecks* em alguns serviços críticos. Por exemplo, no Postgres adicionamos um healthcheck com `pg_isready` antes de liberar dependentes [73] [74]. Podemos adicionar healthchecks similares ao Redis (e.g. usar `redis-cli ping`) e à Evolution API (talvez verificando um endpoint de status, se disponível, ou checando periodicamente se o processo está respondendo). Ao definir healthchecks, podemos utilizar um utilitário como **Autoheal** – incluiremos um container `autoheal` (imagem `willfarrell/autoheal`) que monitora contêineres marcados como unhealthy e os reinicia automaticamente. Assim, se por exemplo a Evolution API travar (marcada unhealthy), o autoheal a reiniciará sem intervenção.
- **Redundância:** Numa arquitetura de produção, colocar o banco de dados em cluster (ex: Patróni para Postgres ou usar um serviço gerenciado) e rodar instâncias da Evolution API em múltiplos hosts com balanceamento de carga garante que mesmo se um nó cair, o serviço continue. O Docker Swarm ou Kubernetes podem cuidar disso; a doc oficial inclusive traz diretrizes de configuração em Swarm para HA [75] [76]. Para nosso escopo, mantemos uma instância única, mas com possibilidade de retomada rápida (graças a volumes e auto-restart).
- **Escalabilidade horizontal:** Com as integrações via RabbitMQ e possivelmente webhooks, podemos externalizar várias responsabilidades. Por exemplo, se o volume de requisições ao

OpenAI crescer muito (encarecendo ou ultrapassando limites), poderíamos integrar um modelo local via **Flowise** ou **Langchain**, ou adicionar uma camada de fila para chamadas OpenAI para nivelar a taxa. A arquitetura de filas nos protege de picos: mensagens entram na fila e são processadas conforme capacidade dos workers, sem sobrecarregar de uma vez a API principal.

- **Logs e observabilidade:** Por fim, armazenar logs de conversa (já temos no DB) e logs de sistema (logs dos containers) é importante para auditoria e para depurar problemas. Utilize volumes ou drivers de log apropriados. Em caso de indisponibilidade, os logs podem ajudar a diagnosticar se foi falha na conexão do WhatsApp, erro da OpenAI, etc., e tomar medidas (como talvez reiniciar a instância do WhatsApp via endpoint ou alertar suporte).

Recapitulando: nossa implantação prioriza manter tudo em pé automaticamente (restart, autoheal) e atualizado (watchtower), além de persistir tudo que for necessário para rápida recuperação em caso de falha. Com as precauções acima, teremos um sistema de atendimento automatizado **confiável e escalável**, pronto para evoluir com as necessidades.

Configuração com Docker Compose (Exemplo)

Finalmente, vamos consolidar todas as configurações em um arquivo `docker-compose.yml` comentado, juntamente com um arquivo `.env` **de exemplo**, para demonstrar a orquestração de todos os serviços mencionados. Essa configuração visa otimizar a implantação em um só host, com persistência de dados e facilidade de manutenção.

Arquivo `.env` de Exemplo

A seguir, um exemplo de arquivo de ambiente contendo as variáveis utilizadas pela Evolution API e pelo Typebot. **Substitua os valores conforme sua necessidade** (especialmente senhas, chaves e URLs reais em produção):

```
#======== Configurações da Evolution API v2 ========#
AUTHENTICATION_API_KEY=superchavesecreta123   # Chave para proteger os
endpoints da Evolution API

# Banco de Dados (PostgreSQL)
DATABASE_ENABLED=true
DATABASE_PROVIDER=postgresql
DATABASE_CONNECTION_URI=postgresql://postgres:MinhaSenhaForte@postgres:5432/
evolution?schema=public
DATABASE_CONNECTION_CLIENT_NAME=evolution_exchange
DATABASE_SAVE_DATA_INSTANCE=true
DATABASE_SAVE_DATA_NEW_MESSAGE=true
DATABASE_SAVE_MESSAGE_UPDATE=true
DATABASE_SAVE_DATA_CONTACTS=true
DATABASE_SAVE_DATA_CHATS=true
DATABASE_SAVE_DATA_LABELS=true
DATABASE_SAVE_DATA_HISTORIC=true

# Cache (Redis)
CACHE_REDIS_ENABLED=true
CACHE_REDIS_URI=redis://redis:6379/6
CACHE_REDIS_PREFIX_KEY=evolution
```

```
CACHE_REDIS_SAVE_INSTANCES=false
CACHE_LOCAL_ENABLED=false

# Fila de Mensagens (RabbitMQ)
RABBITMQ_ENABLED=true
RABBITMQ_URI=amqp://admin:admin@rabbitmq:5672/default
RABBITMQ_EXCHANGE_NAME=evolution_exchange
RABBITMQ_GLOBAL_ENABLED=true
# Habilitar eventos principais no RabbitMQ
RABBITMQ_EVENTS_MESSAGES_SET=true
RABBITMQ_EVENTS_MESSAGES_UPSERT=true
RABBITMQ_EVENTS_SEND_MESSAGE=true

# Integração OpenAI
OPENAI_ENABLED=true           # Habilita integração com OpenAI (é necessário
cadastrar credencial via API)
# (Outras configs OpenAI são feitas via endpoints da Evolution API, não
por .env)

# Integração Typebot
TYPEBOT_API_VERSION=latest    # Usa a versão mais recente da API do Typebot

# (Se usássemos Chatwoot, Dify, etc., aqui colocaríamos CHATWOOT_ENABLED,
DIFY_ENABLED etc. como necessário)

#======== Configurações do Typebot (Self-Hosted) ========#
# Banco de Dados do Typebot (PostgreSQL interno do Typebot)
PGHOST=typebot-db
PGDATABASE=typebot
PGUSER=postgres
PGPASSWORD=SenhaTypebot123

# Chave secreta e email admin do Typebot
SECRET_KEY=SUA_CHAVE_SECRETA_32CARACTERES_AQUI   # substitua por uma chave
aleatória de 32 chars
ADMIN_EMAIL=[email protected]                     # email para conta admin do
Typebot
DEFAULT_WORKSPACE_PLAN=UNLIMITED                 # dá plano ilimitado para
workspaces novos (remove limite de 200 chats)
# (Opcional: configurar SMTP se desejar envio de emails de confirmação/
convite no Typebot)
```

Alguns pontos a ressaltar no `.env` acima:

- Use senhas fortes em vez dos exemplos (`MinhaSenhaForte`, `SenhaTypebot123`) e altere `AUTHENTICATION_API_KEY` para um valor seguro.
- `DATABASE_CONNECTION_URI` aponta para o serviço `postgres` definido no Compose, com o usuário padrão `postgres` e a senha definida lá (no Compose adiante).
- As flags de `DATABASE_SAVE_DATA_*` estão todas `true` para máxima persistência.

- Em `RABBITMQ_URI`, configuramos `admin:admin` combinando com as credenciais definidas no serviço RabbitMQ.
- `OPENAI_ENABLED=true` apenas habilita a funcionalidade; lembre-se de cadastrar a chave da API OpenAI via endpoint após subir os containers.
- O bloco do Typebot (`PGHOST` etc.) será utilizado pelos containers `typebot-builder` e `typebot-viewer` para se conectarem ao banco. Reforçando: esses valores não interferem na Evolution API, pois são lidos somente pelos processos do Typebot (no Compose, amarraremos via `env_file`).
- `SECRET_KEY` **deve** ser substituída por um valor aleatório próprio. Sem isso, o Typebot recusará iniciar.
- `ADMIN_EMAIL` é importante para você conseguir acesso admin no Typebot. Use um email válido (o Typebot pode pedir confirmação via link, dependendo da configuração de email SMTP; se você não configurar SMTP, tente usar um email real e conferir os logs do container para extrair o link de confirmação, ou configure `DEFAULT_WORKSPACE_PLAN` como Unlimited para nem precisar confirmar, já que a conta inicial vira admin automaticamente).
- `DEFAULT_WORKSPACE_PLAN=UNLIMITED` evita a limitação de 200 chats do plano grátis no Typebot [38], o que é útil para ambiente self-hosted.

Arquivo `docker-compose.yml`

Abaixo, apresentamos o **docker-compose.yml** unindo todos os serviços: Evolution API, Postgres, Redis, RabbitMQ, Typebot (builder, viewer e banco) e os auxiliares (Watchtower e Autoheal). Esta configuração pode ser salva e executada diretamente. Cada serviço está comentado para explicar sua função:

```
version: "3.9"
services:
  # Serviço da Evolution API v2
  evolution-api:
    image: evoapicloud/evolution-api:v2.3.4   # usar versão estável mais
recente disponível
    container_name: evolution_api
    restart: always
    env_file:
      - .env                          # carrega as variáveis do nosso
arquivo .env
    ports:
      - "8080:8080"
# expõe API na porta 8080 (pode proteger via proxy depois)
    volumes:
      - evolution_instances:/evolution/instances   # volume para persistir
sessões WhatsApp
    depends_on:
      - postgres
      - redis
      - rabbitmq

  # Banco de Dados PostgreSQL para Evolution API
  postgres:
    image: postgres:15-alpine
    container_name: evolution_postgres
```

```
    restart: always
    environment:
      - POSTGRES_DB=evolution        # nome do banco a criar
      - POSTGRES_PASSWORD=MinhaSenhaForte   # senha do usuário 'postgres'
    volumes:
      - evolution_pgdata:/var/lib/postgresql/data   # volume para dados do
Postgres
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres -d evolution -h localhost"]
      interval: 10s
      timeout: 5s
      retries: 5

  # Servidor Redis para cache
  redis:
    image: redis:7-alpine
    container_name: evolution_redis
    restart: always
    volumes:
      - evolution_redisdata:/data    # (opcional: volume caso queira
persistência do dump RDB/AOF)
    # Redis possui healthcheck embutido na imagem oficial (ping)

  # Broker RabbitMQ para eventos
  rabbitmq:
    image: rabbitmq:3.12-management          # versão com plugin de
management (facilita monitoramento opcional)
    container_name: evolution_rabbit
    restart: always
    environment:
      - RABBITMQ_DEFAULT_USER=admin
      - RABBITMQ_DEFAULT_PASS=admin
      - RABBITMQ_DEFAULT_VHOST=default
    ports:
      - "5672:5672"               # porta AMQP (se precisar acessar
externamente)
      - "15672:15672"             # porta da interface web (http://
localhost:15672) - opcional
    volumes:
      - evolution_rabbitmq:/var/lib/rabbitmq  # volume para filas/dados
persistentes
    healthcheck:
      test: ["CMD", "rabbitmqctl", "status"]
      interval: 30s
      timeout: 5s
      retries: 5

  # Banco de Dados PostgreSQL para Typebot
  typebot-db:
    image: postgres:15-alpine
    container_name: typebot_postgres
```

```
    restart: always
    environment:
      - POSTGRES_DB=typebot
      - POSTGRES_PASSWORD=SenhaTypebot123    # senha do postgres do Typebot
(usar a mesma em .env PGPASSWORD)
    volumes:
      - typebot_pgdata:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres -d typebot -h localhost"]
      interval: 10s
      timeout: 5s
      retries: 5

  # Typebot Builder (interface de criação de bots)
  typebot-builder:
    image: baptistearno/typebot-builder:latest
    container_name: typebot_builder
    restart: always
    env_file:
      - .env              # reutiliza as vars de .env relevantes (PGHOST,
SECRET_KEY, etc.)
    ports:
      - "8081:3000"       # Porta do Builder UI (acessível em http://
localhost:8081)
    depends_on:
      typebot-db:
        condition: service_healthy   # aguarda DB do Typebot ficar pronto
antes
    # (o builder se inicializa executando migrações no DB na primeira vez)

  # Typebot Viewer (interface runtime dos bots, usada via API pelas
integrações)
  typebot-viewer:
    image: baptistearno/typebot-viewer:latest
    container_name: typebot_viewer
    restart: always
    env_file:
      - .env              # usa mesmas credenciais de DB e secret
    ports:
      - "8082:3000"
# Porta do Viewer (pode ser usada para chamadas API do bot)
    depends_on:
      typebot-db:
        condition: service_healthy

  # Watchtower - atualizador automático de containers
  watchtower:
    image: containrrr/watchtower:latest
    container_name: watchtower
    restart: always
    environment:
```

```
      - WATCHTOWER_CLEANUP=true            # remove imagens antigas após
update
      - WATCHTOWER_POLL_INTERVAL=300       # verifica atualizações a cada
300s (5 min)
      - WATCHTOWER_INCLUDE_STOPPED=false   # monitora apenas containers
rodando
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock

# (Observação: Por segurança, talvez usar uma versão fixa do watchtower e
limitar a update do que for desejado)

  # Autoheal - reinicia containers que entrarem em estado unhealthy
  autoheal:
    image: willfarrell/autoheal:latest
    container_name: autoheal
    restart: always
    environment:
      - AUTOHEAL_CONTAINER_LABEL=all
# monitora todos os containers (ou use label para específicos)
      - AUTOHEAL_INTERVAL=10           # verifica a cada 10s
      - AUTOHEAL_DEFAULT_STOP_TIMEOUT=10
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    # Importante: adicionar label "autoheal=true" nos containers que
desejamos monitorar
    labels:
      - autoheal=true

# Definição dos volumes persistentes
volumes:
  evolution_instances:
  evolution_pgdata:
  evolution_redisdata:
  evolution_rabbitmq:
  typebot_pgdata:
```

Acima, temos bastante informação. Vamos destacar alguns aspectos finais para garantir o funcionamento adequado:

- **Ordem de inicialização:** Definimos `depends_on` com healthcheck para garantir que o Evolution API só suba após o Postgres e RabbitMQ estarem prontos, e que o Typebot só suba após seu Postgres estar pronto. Isso evita erros de conexão na partida.
- **Portas mapeadas:** Usamos as portas 8080 (Evolution API), 8081 (Typebot Builder), 8082 (Typebot Viewer), além de 5672/15672 do RabbitMQ. Sinta-se livre para ajustá-las para evitar conflitos no seu host. Em produção, provavelmente o Builder/Viewer do Typebot ficariam atrás de um proxy e não expostos diretamente nessas portas brutas.
- **Watchtower e Autoheal:** Ambos requerem acesso ao socket Docker (montamos `/var/run/docker.sock`). O Watchtower atualizará todos os containers (a menos que restrinjamos por label ou config). O Autoheal, configurado com `AUTOHEAL_CONTAINER_LABEL=all`, observará todos. **Adicionamos uma label** `autoheal=true` nele mesmo para indicar que está

monitorando geral – mas na verdade o autoheal monitora outros containers, não a si próprio, então essa label não é tão crítica aqui (seria usada se optássemos por monitorar apenas containers com essa label). De qualquer forma, ao optar por `all`, garantir que os containers críticos tenham um healthcheck definido é essencial. No nosso compose, definimos healthchecks para Postgres e RabbitMQ. O Redis e Typebot usam checks internos simples. Poderíamos adicionar um healthcheck custom no Evolution API (por exemplo, checar se consegue pegar `/` ou `/docs` com código 200), o que seria recomendado para o autoheal saber reiniciá-lo se travar.

- **Volumes:** Nomeamos todos os volumes para fácil identificação. Você pode verificar se estão corretamente montados com `docker volume ls` e inspecionar. Eles garantem persistência de:
- Sessões WhatsApp (`evolution_instances`),
- Dados do Postgres principal (`evolution_pgdata`),
- (Opcional) Dump do Redis (`evolution_redisdata` – Redis por padrão armazena um snapshot RDB, então mantivemos),
- Filas do RabbitMQ (`evolution_rabbitmq`),
- Dados do Postgres do Typebot (`typebot_pgdata`).
- **Imagens e versões:** Fixamos o Postgres e Redis em versões específicas (15-alpine e 7-alpine). No Evolution API, usamos `evoapicloud/evolution-api:v2.3.4` hipotético (esse repositório reflete as versões mais recentes >=2.3; alternativamente poderíamos usar `atendai/evolution-api:v2.2.3` se estivéssemos em versão antiga). É importante usar uma tag de versão conhecida em produção para evitar atualizações breaking sem planejamento [72]. No Watchtower, usamos `latest` deliberadamente, já que ele se autoatualiza raramente e não é crítico quebrar. De qualquer forma, monitore os logs do Watchtower para saber quando algo foi atualizado – idealmente teste atualizações em ambiente separado antes de aplicar em produção.

Com o compose configurado, **basta rodar**: `docker-compose up -d` (ou `docker compose up -d` em versões mais novas) e aguardar. Os containers deverão iniciar em poucos instantes. Você pode acompanhar os logs iniciais com `docker-compose logs -f evolution-api typebot-builder typebot-viewer` etc., para verificar se tudo subiu corretamente.

Agora você terá: a Evolution API rodando na porta 8080 (tente acessar `http://localhost:8080/docs` para ver a documentação da API aberta, ou faça um GET em `/api/version` para checar a versão), o Typebot Builder na porta 8081 (acesse e crie um bot, usando o email configurado), e o WhatsApp instância aguardando conexão. Para conectar o WhatsApp via Baileys, utilize a API (com sua AUTH key) para criar uma instância e obter o QR code (endpoint `/instance/create` e depois pegue o QR em `/instance/qrcode`). Alternativamente, use a interface web do Evolution Manager se estiver disponível para facilitar essa configuração.

Testes iniciais: Com tudo no ar, teste enviar uma mensagem de WhatsApp para o número conectado. Se o bot OpenAI estiver configurado com trigger "all", a IA deve responder após o debounce configurado (pode levar ~30s após sua última mensagem). Observe no log do Evolution API as interações e eventuais chamadas à OpenAI. Caso tenha integrado o Typebot com um trigger, teste enviando a palavra-chave de disparo e veja o fluxo acontecendo.

Próximos passos: a partir deste ambiente, você pode iterar e melhorar. Por exemplo, ajustar prompts da IA, treinar o modelo com informações do seu negócio ou implementar o módulo de RAG sugerido para que a IA aproveite o histórico. À medida que coletar mais dados de conversas, analise-os (p. ex., usando queries SQL ou exportando para análises) para descobrir padrões e melhorar o atendimento (insights sobre dúvidas comuns, horários de pico, sentimento, etc.).

Em resumo, integramos uma **solução completa de atendimento automatizado via WhatsApp**, compondo a Evolution API v2 com persistência de dados, cache de performance, filas de eventos e chatbots inteligentes. Essa arquitetura, embora complexa, oferece flexibilidade máxima: você pode tanto conduzir fluxos pré-definidos (Typebot) quanto respostas livres com IA (OpenAI), e até combinar os dois (fluxos que acionam IA para certas perguntas). Com persistência robusta e monitoração, o sistema tende a se manter confiável e a **maximizar o aproveitamento de cada lead**, respondendo rápido e de forma contextualizada, 24×7. Boa implantação e bons resultados!

Referências Utilizadas:

- Documentação oficial da Evolution API v2 (introdução e conceitos) [3] [7]
- Documentação de requisitos: uso de PostgreSQL/MySQL e Redis [5] [13]
- Documentação de integrações: RabbitMQ (eventos globais), Typebot e OpenAI [18] [37] [4]
- Exemplo oficial de Docker Compose e variáveis de ambiente da Evolution API [17] [68]
- Exemplo de configuração do Typebot self-hosted (Portainer stack) [73] [74]
- Recomendações da Evolution API para ambientes standalone vs. escaláveis [66] [72]

---

[1] [2] [3] Introdução - Evolution API Documentation
https://doc.evolution-api.com/v2/pt/get-started/introduction

[4] [62] [63] OpenAI - Evolution API Documentation
https://doc.evolution-api.com/v2/pt/integrations/openai

[5] [7] [8] [9] [10] [11] Banco de Dados - Evolution API Documentation
https://doc.evolution-api.com/v2/pt/requirements/database

[6] [12] [13] [14] [15] [16] Redis - Evolution API Documentation
https://doc.evolution-api.com/v2/pt/requirements/redis

[17] [65] [66] [68] [75] [76] Docker - Evolution API Documentation
https://doc.evolution-api.com/v2/pt/install/docker

[18] [19] [20] [21] [22] [23] [24] [25] [26] [27] RabbitMQ - Evolution API Documentation
https://doc.evolution-api.com/v2/pt/integrations/rabbitmq

[28] [29] [30] [31] [32] [33] [34] [35] [36] [37] Typebot - Evolution API Documentation
https://doc.evolution-api.com/v2/pt/integrations/typebot

[38] Troubleshoot - Typebot Docs
https://docs.typebot.io/self-hosting/troubleshoot

[39] [69] [70] Docker - Typebot Docs
https://docs.typebot.io/self-hosting/deploy/docker

[40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60] [61] OpenAI - Evolution API Documentation
https://doc.evolution-api.com/v2/en/integrations/openai

[64] Flowise - Evolution API Documentation
https://doc.evolution-api.com/v2/en/integrations/flowise

[67] Variáveis de Ambiente - Evolution API Documentation
https://doc.evolution-api.com/v2/pt/env

[71] [72] Update - Evolution API Documentation
https://doc.evolution-api.com/v2/en/updates

73 74 Build Interactive Chat Forms with Typebot
https://formable.app/build-interactive-chat-forms-with-typebot/