Plano — Agente de IA de Atendimento + Pré-qualificação (genérico, multi-nicho)

O que você vai construir

Um **agente de IA genérico e configurável** que atende no WhatsApp. O núcleo universal é:

> **cliente diz o que precisa → o agente atende e faz uma pré-qualificação → gera um resumo da conversa (o que foi decidido / a confirmação) → envia para um número cadastrado → grava tudo no banco.**

O **mesmo sistema** serve para vários nichos. Quem define o comportamento, os produtos/serviços e o destinatário do resumo é o **dono do negócio, na tela de configurações / formulário do prompt**. Exemplos que usaremos:

- **Clínica de estética** (exemplo fictício) — cliente tira dúvidas sobre procedimentos, o agente pré-qualifica e passa para a recepção humana concluir.
- **Loja de carros** — cliente diz o que procura, o agente pré-qualifica (modelo, faixa de preço, troca), resume e manda para o vendedor.
- **Outros nichos** — só trocam catálogo, prompt e número do resumo.

A ideia central: **o mesmo motor vira negócios diferentes só mudando configuração** (catálogo + prompt + número do resumo). Nada de domínio "hardcoded".

Decisões de projeto

- **Genérico e dirigido por configuração.** Nada de domínio fixo no código; clínica/loja são apenas dados.
- **Comportamento configurável pelo formulário do prompt:**
  - **Modo Qualifica (padrão/universal):** atende, tira dúvidas com base no catálogo, pré-qualifica, colhe o nome, **gera resumo e envia ao número cadastrado** (handoff). Quem fecha a venda é o humano.
  - **Modo Completo (opcional, p/ quem agenda):** além do acima, **marca sozinho na agenda** (slots, anti-choque). Ligado por configuração.
- **Catálogo de produtos/serviços = dado configurável** por negócio (tela de configurações).
- **Resumo:** salvo no painel + enviado ao **número cadastrado**.
- **IA: OpenAI** `gpt-4o-mini`.

Parte A — Arquitetura

Stack: **Backend** Express + TS (`backend/`), **Frontend** Next.js 14 (`frontend/`), **Banco** Supabase/PostgreSQL, **IA** OpenAI, **WhatsApp** Meta Cloud API + Z-API + Evolution API. Deploy do backend no Railway (ou similar).

Componentes principais (o que você vai construir):

Componente	Arquivo	Papel
Servidor + webhooks (Meta/Z-API/Evolution)	`backend/src/index.ts`	Recebe as mensagens; o núcleo é o `processarMensagemRecebida`
Envio WhatsApp (dispatcher + providers)	`backend/src/services/whatsapp/`	Um dispatcher que delega para o provider ativo
Chamada à IA	`backend/src/services/ai.service.ts` (`generateReply`)	Fala com a OpenAI usando o `SYSTEM_PROMPT` montado
Histórico de conversa (TTL 30 min)	`backend/src/services/conversas.service.ts`	Mantém o contexto recente da conversa
Roteamento por negócio (multi-tenant)	`backend/src/services/negocio-routing.ts`	Descobre a qual negócio a mensagem pertence
Cliente Supabase	`backend/src/lib/supabase.ts`	Conexão com o banco
Jobs (`setInterval`)	`backend/src/jobs/*.ts`	Tarefas periódicas (ex.: lembretes)
Auth (NextAuth) + shell do dashboard	`frontend/src/lib/auth*.ts` , `dashboard/layout.tsx`	Login e layout do painel
Painel de prompt + simulador	`frontend/src/lib/prompt-builder.ts` , `/api/configuracoes/simular-chat`	Editar e testar o prompt

Provider Evolution API

O dispatcher de WhatsApp (`backend/src/services/whatsapp/`) é feito por uma **interface comum** (`sendMessage` , `receiveWebhook`); assim, cada canal é um arquivo separado: `meta.provider.ts` , `zapi.provider.ts` e `evolution.provider.ts` . Pontos de atenção da Evolution:

- **Modelo de credenciais diferente dos outros dois:** em Meta e Z-API, **o cliente final cadastra as próprias credenciais** (conta Meta dele, instância Z-API dele) na tela de Conexão. Em Evolution, **não há credencial do cliente** — existe uma **única conta/API Evolution master (self-hosted), controlada pelo administrador do sistema**, cujo endpoint e API key ficam só no `.env` do backend (não são expostos nem pedidos ao cliente). O backend cria uma **instância Evolution por negócio** dentro dessa conta master (via chamada à API da Evolution) e o cliente final só interage com o **QR Code** gerado para aquela instância, pareando o próprio número — fluxo igual ao WhatsApp Web.
- **Payload de webhook próprio** — precisa de rota dedicada (`/webhook/evolution`) que normaliza o formato da Evolution para o mesmo shape interno usado por Meta/Z-API antes de cair em `processarMensagemRecebida` . Como todas as instâncias vivem na mesma conta master, o webhook identifica o negócio pelo `instance_name` recebido no payload (equivalente ao roteamento por `phone_number_id` /telefone nos outros providers).
- **Escolha do provider em runtime:** o dispatcher lê `negocio.whatsapp_provider` e delega para o provider correto — mesmo padrão de roteamento por negócio (`negocio-routing.ts`).
- **Sem custo por mensagem / self-hosted:** não depende da Meta Cloud API oficial; útil como opção mais barata, sem o dono do negócio precisar ter conta própria em nenhum provedor.

Padrão-chave (fácil de entender e ensinar): a IA escreve **tokens** no texto (ex.: `HANDOFF:...`); o backend detecta o token, executa a ação e **remove o token** antes de responder ao cliente. E injeta **contexto** com marcas (`[NEGOCIO]` , `[CATALOGO]` , `[CLIENTE]` , ...). Não usamos "function calling" — tokens estruturados são mais simples de aprender e depurar.

Criar: modos configuráveis, **catálogo genérico**, **handoff/resumo para número cadastrado**, **agenda/slots** (só Modo Completo), **provider Evolution API**.

Parte B — Desenho do agente

B.1 Modelo de dados

Tabela	Papel
`negocios`	tenant; credenciais WhatsApp, prompt, horários, **modo**, **número(s) do resumo**
`usuarios`	login do painel
`contatos`	lead/cliente final (nome, novo/recorrente, histórico)
`atendimentos`	registro de cada conversa qualificada (resumo, status, item de interesse, data_hora opcional)
`conversas`	histórico de chat
`catalogo_itens`	**produtos/serviços configuráveis** por negócio
`destinatarios`	número(s) que recebem o resumo
`lembretes`	só Modo Completo

`negocios` (config central, + `config` JSONB):

- `modo_atendimento`: `'qualifica'` | `'completo'`. É só um **PRESET inicial** de capacidades (qualifica = núcleo; completo = liga as de agenda). O comportamento real é definido por `capacidades` (abaixo), que o dono pode customizar individualmente.
- `capacidades` (JSONB) — liga/desliga cada habilidade do agente, **independente do modo**. Cada chave ativa um bloco de contexto e/ou um token no prompt (ver B.6):
  - `consultar_catalogo` — **padrão: `true`, sempre ativo** (é a base).
  - `consultar_agenda` — padrão: `false` (só se for usar agendamento).
  - `criar_agendamento` — padrão: `false`.
  - `consultar_agendamento_cliente` — padrão: `false`.
  - `cancelar_agendamento` — padrão: `false`.
  - `enviar_resumo` — **padrão: `true`, sempre ativo** (é a base).
- `whatsapp_provider`: `'meta'` | `'zapi'` | `'evolution'` — define qual provider o dispatcher usa para aquele negócio. Cada provider tem seu próprio bloco de credenciais (ver abaixo).
- `tipo_negocio` (texto livre: "clínica de estética", "loja de carros"...) — entra no prompt.
- Horários (`horario_inicio/fim`, `sabado_modo`, `domingo_modo`, `timezone`), `duracao_slot_min`, `antecedencia_minima_horas`, `dias_futuros_max` (capacidades de agenda).
- `prompt_customizado` (tom/regras/FAQ) e `config` JSONB (ex.: dados de pagamento/PIX).

Credenciais de WhatsApp por provider (armazenadas em `negocios` ou em `config` JSONB, conforme o provider ativo):

- **Meta Cloud API:** `meta_verify_token`, `meta_access_token`, `meta_phone_number_id`. **Credenciais do próprio cliente** — ele cadastra os dados da conta Meta dele na tela de Conexão.
- **Z-API:** `zapi_instance_id`, `zapi_token`, `zapi_client_token`. **Credenciais do próprio cliente** — ele cadastra os dados da instância Z-API dele.
- **Evolution API: NÃO é credencial do cliente.** A instância Evolution roda numa **conta master única, controlada pelo administrador do sistema** (self-hosted). O backend cria/gerencia uma instância por negócio nessa conta master (via API da Evolution), e o cliente final **só enxerga o QR Code** para parear o próprio WhatsApp — igual a escanear para abrir o WhatsApp Web. Ele não vê nem precisa saber `evolution_api_url` / `evolution_api_key` (essas ficam no `.env` do backend, globais, não por negócio); o que é por negócio é só o `evolution_instance_name` (gerado internamente) e o status da conexão (conectado/desconectado + QR atual).

`catalogo_itens`: `id`, `negocio_id`, `nome`, `descricao`, `preco` / `condicoes` (texto p/ casos como "6x de R$ X"), `ativo`. Editável na tela de configurações. Injetado no contexto como `[CATALOGO: ...]`.

`destinatarios`: `id`, `negocio_id`, `nome`, `whatsapp`, `ativo`. Um ou mais números que recebem o resumo/confirmação.

`atendimentos`: `id`, `negocio_id`, `contato_whatsapp`, `contato_nome`, `contato_tipo` (`novo` | `recorrente`), `item_interesse`, `status` (`em_andamento` | `handoff` | `agendado` | `cancelado` | `concluido` — `cancelado` é usado pela capacidade `cancelar_agendamento`; reservar `reagendado` para uso futuro), `data_hora` (capacidades de agenda), `resumo`, `criado_em`.

- Com capacidade de agenda ligada, índice único parcial `(negocio_id, data_hora) WHERE status NOT IN ('cancelado','reagendado')` → **anti-choque** (slot cancelado/reagendado volta a ficar livre).

B.2 Modos de atendimento (a configuração que muda tudo)

O backend lê `negocio.modo_atendimento` e ajusta:

- **Contexto injetado:** Modo Completo injeta `[AGENDA: proximos_livres=...]`; Modo Qualifica não.
- **Tokens/ações:**
  - **Qualifica:** `HANDOFF:{nome}|{item}|{tipo}` → grava `atendimento` `handoff`, gera resumo e **envia para os** `destinatarios`, avisa o cliente que o humano vai concluir.
  - **Completo:** `AGENDAMENTO_CONFIRMADO:{nome}|{item}|{data}|{hora}` → `marcarSlot()` (anti-choque) + resumo + envia aos `destinatarios`.
- O **prompt** recebe um bloco condicional ("MODO: qualifica/completo"). Assim o dono liga/desliga capacidade **só pela configuração**.

B.3 Serviço de agenda (só Modo Completo)

`backend/src/services/agenda.service.ts` : `slotsDisponiveis(negocioId, data)` , `proximosSlotsDisponiveis(negocioId, qtd)` (→ `[AGENDA]`), `marcarSlot(...)` (revalida + insere; constraint garante atomicidade; retorna conflito → IA reoferece). Reusa `horario.service.ts` como base.

B.4 Fluxo de uma mensagem (`processarMensagemRecebida`)

```
WhatsApp (Meta/Z-API/Evolution) → webhook → resolve negócio → lê capacidades
  → injeta contexto (cada bloco condicionado à capacidade ativa):
      [NEGOCIO: nome, tipo, modo, horario, pagamento]
      [CATALOGO: itens/preços configurados]              (consultar_catalogo)
      [CLIENTE: nome, tipo(novo/recorrente), ultimo_atendimento]  (se conhecido)
      [AGENDA: proximos_livres=...]                       (consultar_agenda)
      [AGENDAMENTOS_CLIENTE: id, servico, data, hora, status]
                  (consultar_agendamento_cliente E contato tem agendamento ativo)
  → histórico (30 min) → generateReply(OpenAI, ..., prompt_customizado)
  → detecta TOKENS (cada um só se a capacidade estiver ativa):
      NOME_CLIENTE:{nome}                                 → upsert contato
      HANDOFF:{nome}|{item}|{tipo}                         → resumo → envia aos destinatários
      AGENDAMENTO_CONFIRMADO:{...}        (criar_agendamento)→ marcarSlot() → resumo → envia
      CANCELAR_AGENDAMENTO:{atendimento_id}|{motivo}
                  (cancelar_agendamento) → localiza atendimento, status='cancelado',
                  libera o slot na agenda → resumo → envia aos destinatários
  → remove tokens → responde ao cliente → salva histórico
```

Notas:

- `[AGENDAMENTOS_CLIENTE]` é **leitura** (precisa estar disponível **antes** da IA responder), por isso é injetado como contexto e **não** é token — só ocorre se `capacidades.consultar_agendamento_cliente` estiver `true` e o contato tiver agendamento ativo.
- A IA só oferece horários quando `[AGENDA]` está presente. O backend sempre **revalida** antes de marcar e ao cancelar libera o slot.

B.5 Resumo / Handoff (núcleo universal)

`backend/src/services/resumo.service.ts` : `gerarResumo(historico)` faz **2ª chamada OpenAI** (sumarização) → resumo curto e estruturado (cliente, novo/recorrente, item de interesse, o que foi decidido, próximo passo / horário). Gatilho: **qualquer token de ação** — `HANDOFF` , `AGENDAMENTO_CONFIRMADO` ou `CANCELAR_AGENDAMENTO` (no cancelamento, o resumo informa o que foi

cancelado e o motivo). Persiste em `atendimentos.resumo`, envia via `whatsappService.sendMessage()` para **todos os** `destinatarios` **ativos**, e aparece no painel.

B.6 Prompt do agente

Estrutura por passos, genérica e alimentada por config:

1. **Identidade e tom** (preenchidos pelo dono; ex.: acolhedor/tranquilizador).
2. **Como ler o contexto** (`[NEGOCIO]`, `[CATALOGO]`, `[CLIENTE]`, `[AGENDA]`, bloco **MODO**).
3. **Usar o catálogo** (`[CATALOGO]`) para preços/dúvidas — **nunca inventar** item/preço fora dele → oferecer "avaliação/atendimento humano".
4. **Novo vs recorrente** → `NOME_CLIENTE`.
5. **FAQ/objeções** (texto do dono).
6. **Regra de fecho conforme capacidades** (sempre `HANDOFF` para qualificação; se `criar_agendamento` ligado, ofertar `[AGENDA]` + `AGENDAMENTO_CONFIRMADO`; se `cancelar_agendamento` ligado, permitir `CANCELAR_AGENDAMENTO`).
7. **Formato exato dos tokens** (exemplos).

Capacidades → mecanismo (tokens estruturados, NÃO function calling):

Capacidade	Mecanismo	Observação
buscar_servicos	Contexto injetado `[CATALOGO: ...]`	base
verificar_agenda	Contexto injetado `[AGENDA: proximos_livres=...]`	capacidade de agenda
criar_agendamento	Token `AGENDAMENTO_CONFIRMADO:{nome}|{item}|{data}|{hora}`	
buscar_agendamento	Contexto injetado `[AGENDAMENTOS_CLIENTE: id, servico, data, hora, status]` quando o contato tem agendamento ativo. **Não é token** — é leitura, precisa estar disponível **antes** da IA responder	
cancelar_agendamento	Token `CANCELAR_AGENDAMENTO:{atendimento_id}|{motivo}`	
enviar_resumo_gerente	`resumo.service.ts`, disparado por **qualquer token de ação** (HANDOFF, AGENDAMENTO_CONFIRMADO, CANCELAR_AGENDAMENTO)	

Prompt-builder condicional por capacidade (não só pelo modo binário): o `prompt-builder` lê `negocios.capacidades` e monta o `SYSTEM_PROMPT` incluindo **somente** os blocos de instrução/token das capacidades **ativas** daquele negócio. Ex.: se `cancelar_agendamento` está `false`, o prompt **não** menciona o token `CANCELAR_AGENDAMENTO` nem ensina a IA a cancelar; se `consultar_agenda` está `false`, nenhum bloco de horários/agendamento entra. Assim, dois

negócios com o mesmo motor recebem prompts diferentes conforme suas capacidades. O `modo_atendimento` apenas pré-seleciona o conjunto inicial de capacidades.

Editável no painel + testável no **simulador** (`/api/configuracoes/simular-chat`). Clínica e loja de carros são dois preenchimentos diferentes do mesmo formulário.

B.7 Painel (frontend)

Telas do MVP:

- **Atendimentos/Resumos** — conversas qualificadas + resumo (e agendamentos no Modo Completo).
- **Contatos** — leads (novo/recorrente).
- **Catálogo** — CRUD de produtos/serviços do negócio.
- **Destinatários** — números que recebem o resumo.
- **Agenda** — calendário/lista (aparece no Modo Completo).
- **Configurações** — modo, tipo de negócio, horários, **prompt + simulador**, pagamento/PIX, e uma área **"Capacidades do agente"**.
- **Conexão** — Seletor de provider WhatsApp (**Meta / Z-API / Evolution**), com telas diferentes conforme a escolha:
  - **Meta / Z-API:** formulário para o **cliente colar as próprias credenciais** (conta Meta ou instância Z-API dele).
  - **Evolution: sem formulário de credencial** — o backend já cria a instância na conta master do administrador; o cliente só vê o **botão "Conectar" que gera o QR Code** e o status (conectado/desconectado).

Área "Capacidades do agente" (na tela de Configurações): 6 toggles individuais gravando em `negocios.capacidades` , cada um com texto explicando o que faz:

- **Consultar catálogo** (sempre ligado) — responder sobre produtos/serviços e preços.
- **Consultar agenda** — informar horários livres ao cliente.
- **Criar agendamento** — marcar a consulta/horário sozinho.
- **Consultar agendamento do cliente** — ver/relembrar o agendamento ativo do cliente.
- **Cancelar agendamento** — cancelar um agendamento a pedido do cliente.
- **Enviar resumo ao gerente** (sempre ligado) — mandar o resumo da conversa aos números cadastrados.

Selecionar o `modo_atendimento` pré-marca os toggles (preset), mas o dono pode ligar/desligar cada um individualmente.

Parte C — Sequência de construção (do zero)

Projeto novo e vazio; cada módulo roda de ponta a ponta:

1. **Módulo 1 — Esqueleto:** monorepo (Express+TS / Next.js / Supabase), `/health`, conexão Supabase.
2. **Módulo 2 — WhatsApp:** webhooks Meta/Z-API/Evolution + envio (dispatcher `whatsapp/`, com `meta.provider.ts`, `zapi.provider.ts`, `evolution.provider.ts` e rota de webhook dedicada). Eco ponta a ponta com o provider escolhido.
3. **Módulo 3 — IA + Prompt:** `ai.service` + `conversas.service` + **formulário do prompt** + **catálogo** (`[CATALOGO]`). Tirar dúvidas funciona. Demonstrar clínica **e** loja de carros no mesmo motor.
4. **Módulo 4 — Contato + qualificação:** `contatos` / `atendimentos`, `NOME_CLIENTE`, novo vs recorrente, `[CLIENTE]`.
5. **Módulo 5 — Modo Qualifica + Resumo (MVP universal):** `modo_atendimento`, `destinatarios`, token `HANDOFF`, `resumo.service`, envio ao número cadastrado, salvar no painel.
6. **Módulo 6 — Modo Completo (agenda):** `agenda.service` (slots/anti-choque), `[AGENDA]`, `AGENDAMENTO_CONFIRMADO`. Ligado por config.
7. **Módulo 7 — Painel:** Atendimentos/Resumos, Contatos, Catálogo, Destinatários, Configurações (modo + prompt + simulador).
8. **Módulo 8 — Extras (opcional):** lembrete, cancelamento/remarcação, PIX, multi-tenant/planos.

Arquivos a criar

- `backend/src/index.ts` (servidor + fluxo `processarMensagemRecebida`)
- `backend/src/lib/supabase.ts` (cliente do banco)
- `backend/src/services/ai.service.ts` (IA + prompt)
- `backend/src/services/conversas.service.ts` (histórico)
- `backend/src/services/horario.service.ts` (base de slots)
- `backend/src/services/whatsapp/` — `whatsapp.service.ts` (dispatcher) + `providers/meta.provider.ts`, `providers/zapi.provider.ts`, `providers/evolution.provider.ts` (+ rota `/webhook/evolution`)
- `backend/src/services/agenda.service.ts`, `services/resumo.service.ts`, `services/atendimento-parser.ts`
- `backend/src/services/negocio-routing.ts`
- Migrations: `negocios` / `contatos` / `atendimentos` / `catalogo_itens` / `destinatarios` (+ índice anti-choque) e config (`modo_atendimento`, `whatsapp_provider`, etc.)
- Frontend: telas do painel (Atendimentos, Contatos, Catálogo, Destinatários, Configurações com prompt + simulador, Conexão com seletor de provider)

Verificação (ponta a ponta)

1. **Clínica (Modo Qualifica, simulador):** preencher catálogo + prompt da clínica, conversar, demonstrar intenção → conferir `atendimento` `status=handoff`, `resumo` salvo e **enviado ao número cadastrado**.
2. **Loja de carros (mesmo motor):** trocar catálogo + prompt para a loja → o mesmo fluxo qualifica e resume, provando a flexibilidade.
3. **Modo Completo:** ligar nas configurações → a conversa passa a ofertar `[AGENDA]` e marca; conferir gravação e bloqueio de slot.
4. **Anti-choque:** marcar o mesmo horário 2x → 2ª falha/reoferece.
5. **Resumo:** conferir banco, painel e chegada no WhatsApp dos destinatários.
6. **WhatsApp real:** conectar Z-API (QR) e repetir do cliente até o resumo no número cadastrado.
7. **WhatsApp via Evolution:** trocar `whatsapp_provider` para `'evolution'`, parear via QR na instância self-hosted, repetir o fluxo do cliente até o resumo — confirma que o dispatcher funciona igual independente do provider ativo.
8. **Capacidade desligada:** com `cancelar_agendamento` **off**, pedir explicitamente para cancelar no simulador → a IA **não** deve cancelar nada (o token nem está no prompt); deve fazer **handoff genérico** para um humano resolver. Confirma que o prompt-builder respeita as capacidades ativas.