ChatGPT

Diagnóstico Completo dos Problemas Identificados

1. Batching (Agrupamento de Mensagens) Não Operando Corretamente

Apesar do código implementar uma lógica de atraso para agrupar mensagens recebidas em curto intervalo, na prática o agente não está **esperando** para responder e acaba respondendo múltiplas vezes rapidamente. Possíveis causas e observações:

- **Atraso Configurado vs. Atraso Efetivo:** No código Convex, a função `generateAiReply` aplica um delay (`batchingDelay`) antes de processar a resposta [1]. O valor padrão é `3000ms` (3 segundos) caso não haja configuração específica [2]. Embora tenha sido mencionado um desejado *batching* de 120 segundos, é provável que a configuração atual esteja mantendo 3 segundos – o que é muito curto. Isso explicaria por que o agente respondeu quase imediatamente a cada mensagem do usuário, ao invés de juntar todas em uma única resposta.

- **Mensagens Rápidas em Sequência:** O usuário mandou três mensagens em poucos segundos (`"Sou o Guilherme"`, `"da Iceberg Marketing"`, `"assessoria"`). Com apenas ~3s de espera, o primeiro texto pode ter sido processado separadamente antes das outras chegarem. Os logs sugerem que o agente respondeu **três vezes** (às 20:55:24, :26 e :27) – indicando que cada entrada acabou disparando uma resposta. Isso é sintoma de falha no agrupamento: idealmente, as três mensagens do usuário deveriam ter sido capturadas juntas e respondidas com **uma mensagem** do assistente.

- **Mecanismo de Lock/Cooldown:** O Convex utiliza um lock de processamento para evitar concorrência. A função `checkAndSetProcessing` marca a sessão como em processamento e aplica um *cooldown* de 5s entre respostas [3] [4]. No cenário ideal: a primeira mensagem do usuário aciona o processamento e bloqueia as próximas até concluir; as subsequentes seriam ignoradas (*skipped*) durante o atraso e depois consideradas na resposta única. Porém, parece que esse mecanismo não funcionou plenamente:

- Pode ter ocorrido uma corrida de timing: se as mensagens chegaram quase juntas, talvez o lock não estivesse estabelecido a tempo para a segunda mensagem, permitindo duas instâncias paralelas. Isso resultaria em múltiplas respostas.
- Mesmo quando o lock funcionou (ignorando mensagens 2 e 3 durante processamento da 1), notamos que a terceira mensagem do usuário possivelmente ficou sem ser incluída na resposta. O código atual simplesmente ignora (*return skipped*) mensagens recebidas durante o lock, sem re-agendá-las [5]. Assim, a última mensagem `"assessoria"` pode ter ficado sem tratamento na primeira resposta, e depois o sistema não voltou para processá-la. Isso explicaria por que o agente repetiu a pergunta em outra resposta logo em seguida – tentando "alcançar" a informação faltante.

Conclusão: É necessário revisar a lógica de *batching*. Para melhorar: - Aumentar o `batchingDelay` configurado (ex.: 30s a 120s) para garantir que mensagens consecutivas do usuário sejam coletadas antes de responder. Verificar na tabela `ai_configurations` se `batchingDelayMs` está definido conforme esperado (120000ms se 2 minutos for desejado). - Ajustar o tratamento de mensagens ignoradas durante lock: em vez de simplesmente descartar (`skipped`), talvez armazená-las ou **re-disparar** o processamento ao final do atraso. Atualmente, se uma mensagem é ignorada por cooldown, ela pode nunca ser respondida, causando comportamento incoerente. - Testar cenários de mensagens rápidas para confirmar que apenas uma resposta consolidada seja enviada.

2. Prompt Base do Convex Não Sendo Seguido pelo Modelo

O agente não parece estar obedecendo às instruções definidas no prompt base (por exemplo, **repetiu saudações e perguntas**, algo que o prompt proíbe). Pontos relevantes:

- **Armazenamento do Prompt Base:** Conforme identificado, o prompt base personalizado **não fica** na tabela `organizations` (como talvez esperado), mas sim em `ai_prompts` [6]. O código busca o prompt ativo desse org (onde `kind = "spin_sdr"` e `active = true`) [7]. Caso não encontre, utiliza um **prompt padrão** embutido no código [8] [9]. É importante verificar se para a organização em questão:
- Existe um registro em `ai_prompts` com `orgId` correspondente, `kind = "spin_sdr"` e marcado `active = true`.

- Se não houver, o sistema usará o conteúdo default. Isso não é necessariamente ruim, pois o default já contém as regras e metodologia SPIN. Porém, se um prompt custom era esperado e não foi salvo corretamente, o assistente estaria funcionando com instruções genéricas.

- **Formato do Prompt no Envio à API:** A função `getFullPrompt` concatena duas partes: uma **mensagem de sistema** (role do assistente e diretrizes) e um **prompt do usuário/metodologia** [9]. Esses dois trechos são unidos com um separador `"---"`. No momento de construir a chamada para a API do Gemini, o código insere esse texto completo no campo `text` da requisição [10]. Ou seja, todas as instruções (sistema + metodologia + histórico + instruções finais) são passadas em um bloco de texto único para o modelo.

  Possível problema: A API do Google (Gemini 2) pode não estar interpretando esse bloco da mesma forma que o OpenAI interpretaria prompts de sistema versus usuário. Não há separação explícita de papéis/roles – tudo vai como um prompt de texto contínuo. Isso pode reduzir a autoridade das regras (pode ser que o modelo trate até as "regras" como texto comum). Se o Gemini não der peso adequado às diretivas no meio do prompt, ele pode **ignorar restrições** como "não repetir saudações".

- **Evidência de Desobediência do Prompt:** No log, o agente repetiu **três vezes** uma saudação e a pergunta pelo nome completo, variando a forma. O prompt base define claramente:

- "Após a primeira mensagem, nunca repita cumprimentos ou se reapresente." [11] [12]
- "NUNCA repetir perguntas já respondidas" [13].

O fato de ele ter dito “Olá...” múltiplas vezes e pedido o nome de novo sugere que o modelo **falhou em seguir essas instruções**. Isso reforça a suspeita sobre o formato do prompt ou limitações do modelo: - O campo `parts` retornado pela API indica que a resposta veio segmentada. Provavelmente o Gemini devolveu o output dividido em pedaços (talvez cada frase como um part). O código porém só utiliza `parts[0]` [14]. É possível que as frases subsequentes (“Para começarmos, ... nome completo” etc.) estavam em `parts[1]` e `parts[2]` e **foram ignoradas no código** – assim, o sistema acabou enviando várias mensagens separadas, uma para cada *candidate/part*, em vez de uma única. Isso explicaria por que o assistente mandou fragmentos do tipo “Olá Guilherme...” seguido de outra mensagem com a pergunta. Deve-se verificar a estrutura exata retornada pelo Gemini: - Se `data.candidates[0].content.parts` contém múltiplos elementos, o código atual está enviando apenas o primeiro pedaço. Talvez os outros pedaços sejam importantes (no caso, continuação da resposta). O ideal seria concatenar todos os `parts` de `candidates[0]` para compor a resposta completa, garantindo que nada fique de fora.

- Outra hipótese é que *cada resposta duplicada* foi gerada separadamente (devido ao problema de batching mencionado). Porém, as frases são bem parecidas, sugerindo que **podem ser partes de uma mesma resposta** que saíram picadas. Ajustar o uso correto da resposta do modelo é crucial aqui.

- **Substituição de Variáveis no Prompt:** Nota-se no prompt base a presença de `{{$now}}` para indicar data atual [15]. Não há implementação visível de substituição desse placeholder pela data real. Se isso não estiver sendo tratado, o modelo recebe literalmente “ **Data atual:** {{$now}}”, o que não faz sentido pra ele e pode introduzir ruído no entendimento. É importante corrigir esse placeholder (por exemplo, substituir por `new Date().toLocaleDateString()` antes de enviar ao modelo) ou removê-lo se não suportado, para não confundir a IA.

Conclusão: O prompt em si está bem elaborado, mas **há falhas na utilização dele**: - Garantir que o prompt custom de cada organização esteja sendo efetivamente lido (inserir via painel ou script na tabela `ai_prompts` se necessário). - Revisar como o prompt é enviado ao Gemini. Pode ser necessário usar o endpoint adequado (por exemplo, `generateMessage` se disponível, que aceita distinção de roles) ou ao menos enviar o texto de sistema de forma que o modelo entenda que são instruções. - Consertar a captura da resposta completa do modelo (juntando todos os parts retornados) para evitar dividir a mensagem do assistente em várias. - Com essas correções, espera-se que a IA pare de quebrar as regras definidas (ou pelo menos reduza as repetições). Ainda assim, vale lembrar que modelos diferentes têm nível de aderência variável às instruções – talvez ajustes de prompt (reforçar “NÃO cumprimente de novo” bem no final, por ex.) possam ajudar.

3. Coleta de Dados (Nome, Tipo de Pessoa) e Repetição de Perguntas

Outra razão do comportamento estranho é a lógica de coleta e interpretação de dados do prospect, que impacta quais perguntas o bot faz em seguida:

- **Detecção de Nome Incompleta:** O usuário disse “Sou o Guilherme...”. O parser de nomes (`analyzeCollectedData`) tem heurísticas que podem **não reconhecer “Guilherme”** nesse contexto [16]. De fato, a regex de nome exclui respostas contendo certas palavras como “sou” [17]. Assim, `"Sou o Guilherme"` não foi aceito como nome (por conter "Sou"). Resultado: a variável

`collectedInfo.name` ficou vazia. No contexto que o prompt montou, aparece **Nome: NÃO COLETADO** [18].

- **Identificação de Pessoa Física/Jurídica:** Curiosamente, o regex de `personType` marca a mensagem como PJ se encontrar palavras-chave como "empresa" ou até o termo "iceberg" [19]. No exemplo, `"Iceberg Marketing"` acionou esse detector. Ou seja, o sistema inferiu **Tipo: Jurídica** (provavelmente armazenou a própria frase do usuário em `personType`). Também detectou informações de **Empresa** pela presença de "Marketing/consultoria" etc. [20]. Então, após as três mensagens do usuário, o estado coletado possivelmente era:

- Nome: **não** coletado (lista vazia).
- Tipo: coletado (ex.: contém `"Sou o Guilherme da Iceberg Marketing assessoria"` como string – o que é meio estranho, pois deveria ser apenas "Jurídica" ou similar, mas o regex colocou a frase toda).
- Empresa: coletado (ex.: `"Iceberg Marketing assessoria"`).

- **Observação:** O fato de `personType` e `business` estarem preenchidos com as *próprias frases* do usuário pode confundir a formatação do contexto. O prompt insere literalmente essas listas no texto:

    - **Tipo:** "Sou o Guilherme da Iceberg Marketing assessoria" (em vez de simplesmente "Pessoa Jurídica").
    - Isso pode ter atrapalhado a compreensão do modelo ou pelo menos não é o ideal. Talvez fosse melhor normalizar esses campos (ex.: se detectou "iceberg" => deduzir Tipo = Jurídica; extrair nome da empresa separado).

- **Próxima Ação calculada incorretamente:** Na função que determina `nextAction`, o código só verifica se falta `personType` ou `business` [21]. Ele **não verifica se faltava o nome**. No caso, `hasPersonType = true` e `hasBusinessInfo = true`, então o código definiu **nextAction = "Iniciar qualificação SPIN sobre a necessidade específica"** [22]. Ou seja, indicou para o assistente que **já coletou as infos básicas e pode avançar** para perguntas de necessidade. **Porém, o nome não estava coletado de fato.** Isso gera um conflito:

- O contexto gerado listou "Nome: NÃO COLETADO" mas ao mesmo tempo disse que a próxima ação é pular para qualificação avançada.
- O modelo, ao ver "Nome: NÃO COLETADO", provavelmente decidiu que **deveria perguntar o nome** (já que o prompt inicial dizia ser obrigatório coletar nome completo). Então ignorou a instrução de *nextAction* e voltou para pedir o nome.

- Esse retorno inesperado fez o bot repetir a pergunta do nome – e como já havia dado um cumprimento antes, ele acabou repetindo também fórmulas de saudação junto com a pergunta, violando as diretrizes.

- **Repetição de Saudação:** Por que o modelo incluiu "Olá!" novamente ao perguntar o nome? Possivelmente porque:

- Considerou que estava iniciando uma *sub-interação* nova para coletar o nome, então tentou ser educado.

- Pode ter sido influenciado pelo formato do histórico (cada mensagem no histórico começa com " CLIENTE" ou " ASSISTENTE"). Talvez o modelo ficou incerto se devia continuar o diálogo ou reiniciar um cumprimento.
- De qualquer forma, isso reflete que o modelo não interpretou 100% que já estava em meio à conversa. Fortalecer a instrução “Não cumprimente novamente” ou ajustar o prompt de contexto poderia resolver. Por exemplo, talvez remover emojis/identificadores no histórico e usar um formato mais padrão de diálogo pudesse deixar claro para a IA que ela *é* o assistente e já saudou no início.

Conclusão: A lógica de coleta de dados e definição da próxima pergunta precisa de pequenos ajustes para evitar retrabalho: - Quando algum dado obrigatório estiver **faltando**, `nextAction` deve priorizar coletá-lo, independentemente de outros já coletados. No caso, faltando nome completo, a próxima ação deveria ser “Pedir nome completo”, mesmo que já saiba que é PJ. Atualmente essa condição não existe. - Melhorar a extração de nome: talvez detectar o primeiro nome mesmo em frases como "Sou Fulano" (ignorando o "sou"). Poderia usar outra abordagem, como se já houve uma interação anterior onde o usuário disse o nome, armazenar diretamente no contato ou nas variáveis de sessão. - Normalizar o output de `personType` e `business` para valores mais limpos (por exemplo, se regex bateu “iceberg” e “empresa”, definir personType = "Jurídica"; e extrair o nome da empresa separadamente “Iceberg Marketing Assessoria”). Assim o prompt de contexto fica mais claro (Tipo: Jurídica, Empresa: Iceberg Marketing Assessoria) ao invés de duplicar frases do usuário. - Com essas correções, o assistente não ficará indeciso entre coletar nome ou prosseguir, evitando fazer a mesma pergunta novamente. Ele coletará tudo na ordem certa: primeiro nome, depois tipo de pessoa/empresa, etc., conforme o roteiro pretendido.

4. Integração com Evolution API e Dashboard

Além da lógica do agente em si, há a questão dos dados de sessões e estatísticas do dashboard que deveriam vir da API da Evolution (plataforma de WhatsApp):

- **Webhook Configuração:** Foi confirmado que o endpoint de webhook do ROI Gem (WhatsApp) está implementado (`/api/webhook/whatsapp/roigem`). Ele foi testado manualmente com cURL e funcionou [23], porém é fundamental que a **Evolution** (provedor WhatsApp) esteja enviando as mensagens para esse endpoint. No dashboard da Evolution, é preciso registrar a URL do webhook para o *instance* do WhatsApp correspondente. Certifique-se de que:
- A URL configurada na Evolution está exatamente correta (incluso o caminho `/api/webhook/whatsapp/roigem` e usando HTTPS).
- O token de validação bate (`sharedToken` configurado). O código verifica um token para cada conta WhatsApp antes de aceitar o webhook [24] – se houver mismatch, as mensagens seriam descartadas.

- Se essas configurações estiverem incorretas, pode ser que algumas mensagens não estejam chegando no Convex. Porém, como vimos logs de mensagens entrando, provavelmente o webhook está ok. Ainda assim, vale a revisão.

- **Consultas do Dashboard via Evolution:** Existem rotas Next.js para obter contatos, mensagens, análises, etc, possivelmente chamando a API da Evolution. Pelo código:

- Endpoints `/api/evolution/contacts`, `/messages`, `/spin-analysis`, `/instance-stats` estão implementados e fazem fetch para a API Evolution [25] [26]. Eles dependem de variáveis de ambiente `EVOLUTION_BASE_URL` e `EVOLUTION_API_KEY` para funcionar [27].

- Se essas chamadas "não funcionam", pode ser:

    - **Credenciais ausentes ou inválidas:** Verifique se em produção (Vercel) as env vars estão setadas. Localmente pode ter funcionado, mas em produção não.
    - **Diferença de Instance ID/Nome:** O sistema guarda contas WhatsApp em tabela `whatsapp_accounts` com `instanceId`, `baseUrl` e `token` [28] [29]. Certifique-se que o `instanceId` ali corresponde ao nome da instância na Evolution (p.ex. "qify-...") e que o token/API key está correto. A função de envio de mensagem usa esses dados para POSTar [30]. Se o envio de mensagens está funcionando, então a config da conta primária está certa. Mas as rotas de consulta talvez precisem do mesmo token ou de outra credencial?
    - **Dependência de APIs Internas:** Alguns endpoints podem precisar de serviços auxiliares. Por exemplo, `/spin-analysis` foi adaptado para usar Convex, mas se esperar algum cálculo do lado da Evolution, pode falhar se não implementado.
    - **Timeouts ou Erros silenciosos:** Os logs mencionam que consultas via Vercel às vezes dão timeout. As requisições à Evolution têm *timeout* configurado (ex.: 10s para fetch de instâncias [31]). Se a API Evolution demorar mais ou estiver indisponível, o dashboard não vai exibir dados. Adicione logs ou tratativas de erro nessas rotas para ver se está retornando alguma mensagem de erro da Evolution.

- **Dados de Sessão vs. Evolution:** Existe a expectativa de que dados como sessões ativas ou mensagens possam ser obtidos diretamente da Evolution em tempo real. Entretanto, note que **já estamos armazenando** contatos, sessões e mensagens no banco Convex (por meio do webhook) [32]. Portanto, o dashboard poderia usar diretamente esses dados locais (mais rápido) ao invés de bater na API externa para tudo. Se a ideia era usar a Evolution só como backup ou para estatísticas, talvez o não funcionamento seja porque não há implementação para de fato consultar lá (além do fetch de instâncias).

- Verifique o design pretendido: caso o dashboard tente chamar `/api/evolution/instance-stats` para, por exemplo, contar mensagens, isso requer que a Evolution tenha esse endpoint e retorne algo útil. Se não estiver retornando, pode ser que o caminho esteja errado ou a Evolution não ofereça certo dado via API.
- **Solução:** Considerar utilizar os próprios queries do Convex (por ex: contar sessões qualificadas, etc.) para popular o dashboard, já que os dados estão no banco interno. Isso elimina dependência de chamadas externas potencialmente quebrando.

Conclusão: Para a parte de Evolution/dashboard: - Confirmar configurações do webhook e credenciais da API Evolution no ambiente de produção. - Testar manualmente as rotas `/api/evolution/...` para ver se obtêm resposta (por exemplo, via cURL no console Vercel ou logs). Se der erro, depurar se é DNS, autenticação ou formato. - Decidir se o dashboard deve mesmo usar a API externa ou os dados internos – talvez combiná-los (usando Evolution para checar status de conexão da instância, e Convex para dados de conteúdo das conversas). Garantir que todas essas chamadas estejam funcionais ou implementar fallback.

5. Logs e Observabilidade Limitados

Por fim, dificultando identificar tudo isso, os logs disponíveis são limitados: - **Convex:** Os logs do Convex aparecem apenas no console (e possivelmente no dashboard Dev do Convex, mas sem uma interface robusta de busca). Não há um sistema de monitoramento sofisticado integrado. Isso torna difícil rastrear cenários específicos a menos que reproduzidos manualmente. - **Vercel:** Logs de funções serverless (API routes Next.js) existem, mas sabemos que consultas longas podem atingir limite e não logar completamente. Além disso, se há falhas silenciosas (ex.: promessa rejeitada não tratada), podem nem aparecer claramente. - **Sugestão:** Adicionar alguns logs específicos temporariamente: - Logar quando `generateAiReply` está *skipping* uma mensagem (e por qual razão: lock ou cooldown) para confirmar se esse fluxo está ocorrendo como pensamos. - Logar o conteúdo completo enviado ao Gemini e o retorno (candidates/parts) para entender como a resposta está vindo e como está sendo truncada. - Logar chamadas às APIs Evolution (status code, tempo de resposta) para o dashboard, a fim de capturar onde está falhando. - Monitorando esses logs em tempo real durante um teste, será possível confirmar os pontos suspeitos acima e validar as correções.

Conclusão Geral e Próximos Passos

Mesmo com toda a estrutura implementada, pequenas falhas de integração e lógica estão impedindo o correto funcionamento do agente ROI Gem. Em resumo:

- Ajuste o mecanismo de *batching* para realmente agrupar mensagens próximas (aumentar delay e garantir reprocessamento adequado das ignoradas).
- Corrija a formação e utilização do prompt base para que o modelo Gemini respeite as regras (incluir todas as partes da resposta, tratar placeholders, talvez simplificar o formato do histórico).
- Revise a lógica de coleta de informações (nome completo, etc.) para não gerar perguntas repetidas ou conflituosas – sempre coletar dados obrigatórios antes de avançar no roteiro.
- Verifique a integração com a Evolution API (webhook registrado corretamente, env vars de credencial, e endpoints do dashboard) para garantir que dados externos e estatísticas funcionem como esperado.
- Melhore a observabilidade com logs durante os testes, pois assim será possível verificar se as correções surtiram efeito ou se outros ajustes serão necessários.

Implementando esses passos, a expectativa é que o agente passe a conduzir a conversação de forma **fluida e coerente**, sem repetições desnecessárias, e que o dashboard apresente as informações desejadas sobre as sessões e desempenho do bot. Boa depuração!

---

1 2 5 10 14 16 17 18 19 20 21 22 ai.ts
https://github.com/oguidomingos/qify-blackandwhite/blob/769f3317499d018216efd905b18761c21cc4cccc/convex/ai.ts

3 4 sessions.ts
https://github.com/oguidomingos/qify-blackandwhite/blob/769f3317499d018216efd905b18761c21cc4cccc/convex/sessions.ts

6 7 8 9 11 12 13 15 aiPrompts.ts
https://github.com/oguidomingos/qify-blackandwhite/blob/769f3317499d018216efd905b18761c21cc4cccc/convex/aiPrompts.ts

23 24 28 29 32 webhooks.ts.bak

https://github.com/oguidomingos/qify-blackandwhite/blob/769f3317499d018216efd905b18761c21cc4cccc/convex/webhooks.ts.bak

25 26 27 30 31 wa.ts

https://github.com/oguidomingos/qify-blackandwhite/blob/769f3317499d018216efd905b18761c21cc4cccc/convex/wa.ts