AVALIAÇÃO PRELIMINAR

Problemas estruturais.

PREFÁCIO

Antes de avançar com a análise completa de SEO e AEO, é necessário corrigir um conjunto de problemas estruturais do site. Bots de motores de busca e LLMs, crawlers SEO e auditorias de performance dependem de uma infraestrutura estável e sinais técnicos consistentes para produzirem dados fiáveis. Enquanto as falhas documentadas a seguir persistirem, qualquer análise mede sobretudo o ruído que provocam, levando a conclusões erradas e recomendações desviadas.

Resumo dos problemas documentados

1. O servidor falha de forma imprevisível. Em alguns momentos, o site não carrega - o utilizador vê uma página partida ou um ecrã de erro. Afeta tanto visitantes humanos como motores de busca e assistentes de IA que aterrem nesses momentos.

2. O site demora, por vezes, quase 30 segundos a aparecer. Mesmo sem falha total, há momentos em que os ficheiros necessários para mostrar a página chegam muito devagar. O utilizador olha para um ecrã branco até meio minuto e desiste.

3. Quatro dos cinco sitemaps estão em falta. O sitemap é a lista oficial das páginas entregue ao Google. O robots.txt declara cinco, mas só existe um. Secções inteiras (tours, localizações, atrações, blog) podem nunca ser descobertas.

4. As versões em diferentes idiomas não estão ligadas entre si. O sistema que diz ao Google "esta página em português corresponde àquela em francês" aponta para endereços intermediários. O Google pode mostrar a versão errada a cada utilizador e o site pode parecer ter conteúdo duplicado.

5. Mais de 4.600 páginas redirecionam de forma incorreta. O site faz redireccionamentos usando técnicas que o Google trata como inferiores e que os assistentes de IA (ChatGPT, Claude, Perplexity) muitas vezes não seguem. Perde-se autoridade SEO acumulada e visibilidade em IA.

6. Cerca de 150 páginas entram em loop infinito. Por um bug, dizem ao browser para ir para a mesma página onde já está. O browser recarrega indefinidamente e o utilizador vê apenas piscar. Para motores de busca, este conteúdo é invisível.

Enquanto estes seis problemas persistirem, qualquer estudo aprofundado mede sobretudo o ruído que provocam, e não o desempenho real do site. Resolvidos, o site passa a poder ser analisado a sério.

Problemas de Estabilidade e Performance

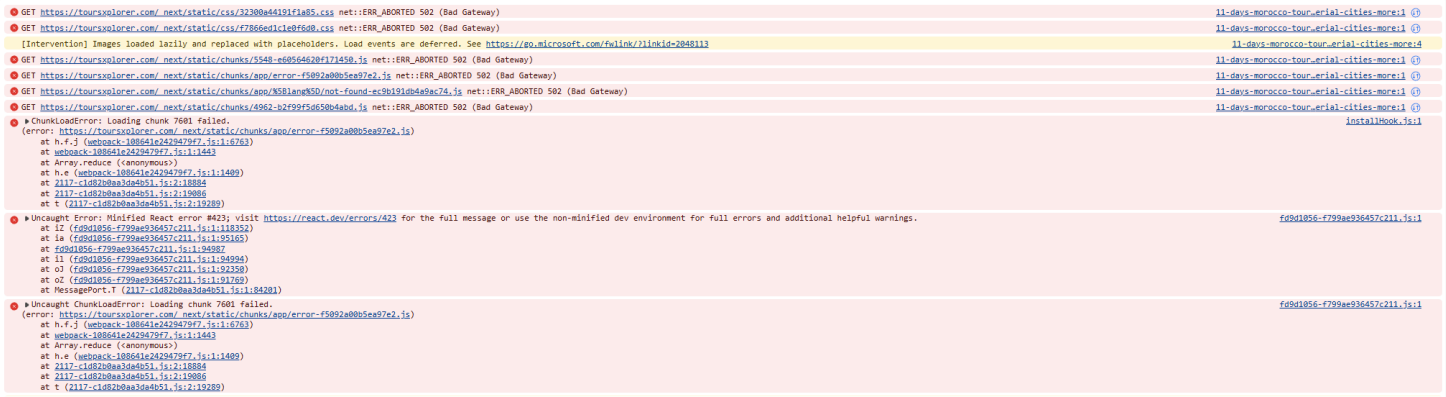
O site apresenta **instabilidade intermitente em produção** e **fraca performance em runtime**. O problema mais urgente não é de otimização - são falhas pontuais do servidor que, quando ocorrem, devolvem erros 502 Bad Gateway em assets estáticos críticos e partem a página logo no primeiro carregamento.

Problemas de estabilidade (críticos)

Uma breve análise revela uma aplicação Next.js que, em determinados momentos, falha o arranque. Estes erros são **intermitentes** - o site nem sempre apresenta este comportamento, e em muitas visitas carrega corretamente - mas quando ocorrem, o impacto é total: o utilizador vê uma página partida ou um ecrã de erro.

Sintomas observados durante os episódios de falha:

- **Erro de aplicação** logo no primeiro paint: "a client-side exception has occurred."
- **Múltiplas respostas `502 Bad Gateway`** do `toursxplorer.com` para:
  - Bundles de CSS (`/_next/static/css/*.css`)
  - Chunks de JS (`/_next/static/chunks/*.js` - incluindo `main-app`, `app/error`, `app/not-found`)
  - Fetches internos de RSC (`/en/chat?_rsc=…`, `/en/location/africa…`)
  - Media (`logo_sm1.1fa6ddc8.png`, fonts)
- **`ChunkLoadError: Loading chunk 7601 failed`** - o runtime tenta carregar em lazy-load um chunk que o servidor recusa servir.
- **React Minified error #423** - um mismatch de hidratação/render despoletado pela falha no carregamento do chunk.



A natureza intermitente do problema, combinada com o padrão dos erros (todos 502, do lado do servidor, a afetar maioritariamente assets estáticos do Next.js), aponta para causas de infraestrutura e não para bugs no código do cliente. As hipóteses mais prováveis são:

- **Upstream em mau estado pontual** - servidor Node/Next a crashar e a reiniciar, picos de memória/OOM, ou worker process a ficar sem capacidade em momentos de maior tráfego.
- **Reverse proxy/CDN mal configurado** - timeouts demasiado curtos entre o proxy e o upstream, ou health checks que retiram instâncias do pool em momentos errados.
- **Deploys não-atómicos** - durante a janela de deploy, o HTML servido pode referenciar hashes de chunks (`_next/static/chunks/...`) que ainda não existem (ou já não existem) em disco, originando 502s temporários até estabilizar.

O facto de em muitas visitas o site renderizar normalmente reforça que se trata de instabilidade de infraestrutura, e não de uma falha permanente. Ainda assim, este tipo de erro é particularmente penalizador: além de afetar a experiência do utilizador, prejudica também a indexação por crawlers (Google, ClaudeBot, GPTBot) que aterrem no site durante um destes episódios.

Speed Index e Throughput de Assets Estáticos

O Speed Index é a métrica do Lighthouse que melhor captura a experiência visual real do utilizador: integra o progresso visual da página ao longo do tempo, frame a frame. Enquanto FCP mede o primeiro pixel pintado e LCP mede o aparecimento de um elemento específico, o Speed Index mede **quão completa a página parece estar, em cada instante, durante todo o load**. É a métrica que melhor correlaciona com a perceção de "o site já está pronto".

Nos dois runs analisados, o Speed Index dá leituras muito divergentes

Leituras observadas

	Run 1 (08:47 UTC)	**Run 2 (12:55 UTC)**	**"Bom" (Lighthouse)**
Speed Index (simulado)	**41,5 s**	4,4 s	< 3,4 s
Speed Index (observado, real)	27,8 s	1,3 s	—
FCP (observado, real)	27,7 s	1,0 s	—
LCP (observado, real)	27,7 s	1,4 s	—
First Visual Change	27,7 s	1,0 s	—

No Run 1, o ecrã ficou completamente branco durante **27,7 segundos** - esta é a medição observada, real, não o valor simulado pelo throttling. O FCP, LCP e first visual change aconteceram **todos no mesmo instante** (entre 27.735 e 27.736 ms). É a assinatura inequívoca de um cenário em que o browser não tinha nada para pintar até esse momento, e depois pintou tudo de uma vez.

Causa-raiz

A análise do log de rede do Run 1 revela o problema: **os assets estáticos servidos pelo próprio `toursxplorer.com` chegaram com throughput catastrófico**. Os pedidos não falharam (todos retornaram 200 OK), mas os bytes demoraram dezenas de segundos a chegar.

Recurso	Duração no Run 1	Mesmo recurso no Run 2
CSS `d4039ac2900cee88.css`	**27567 ms**	~150 ms
`cdn.trustindex.io/loader.js`	17286 ms	416 ms
Chunk `app/[lang]/(public)/tour/[slug]/page-…js`	17222 ms	~400 ms
Chunk `4126-…js`	17154 ms	~400 ms
Chunk `7983-…js`	17095 ms	—
Chunk `8238-…js`	17044 ms	—
Chunk `4962-…js`	16983 ms	—
Chunk `dc112a36-…js`	16922 ms	430 ms
Chunk `c36f3faa-…js`	16840 ms	473 ms
Chunk `layout-…js`	16681 ms	—
Chunk `not-found-…js`	16626 ms	—
Chunk `error-f5092a00b5ea97e2.js`	16569 ms	—
Chunk `app/layout-…js`	16517 ms	—
Chunk `4055-…js`	16453 ms	—
Chunk `app/(public)/layout-…js`	16397 ms	—

Mais de **20 chunks de JavaScript da própria origem `toursxplorer.com`** começaram a descarregar nos primeiros 120 ms (em paralelo via HTTP/2) mas só terminaram por volta dos 17 segundos. Um ficheiro CSS específico chegou aos 27,5 s. O browser não tinha nada para renderizar enquanto os assets não chegavam - daí o ecrã branco. Quando finalmente os bytes chegaram, o browser pintou tudo de uma vez, e por isso FCP, LCP e first visual change coincidem em ~27,7 s.

Importa sublinhar que **a rede não estava lenta**: o RTT no Run 1 foi de 30 ms (mais rápido do que os 130 ms do Run 2) e o server latency foi de 130 ms (mais rápido do que os 180 ms do Run 2). O problema foi **throughput do servidor de origem**: o servidor aceitou os pedidos rapidamente (status 200) mas demorou 16-27 segundos a entregar o conteúdo de cada um.

Conexão com a instabilidade já documentada

Os 41,5 s de Speed Index do Run 1 não são causados por excesso de JavaScript, nem por trackers, nem por imagens, nem por widgets de reviews. **São causados pela infraestrutura que serve os assets estáticos do próprio `toursxplorer.com` a entregar conteúdo com throughput catastrófico de forma intermitente.**

Esta é a mesma causa-raiz identificada na secção de Estabilidade, manifestada de forma diferente:

- **Falha completa do servidor** → 502 Bad Gateway, ChunkLoadError, ecrã de erro do React.
- **Falha parcial do servidor** → 200 OK mas com entrega a 17 segundos por chunk, ecrã branco durante 27 segundos.
- **Servidor saudável** (Run 2) → tudo carrega em < 1 segundo, Speed Index de 4,4 s.

São três faces do mesmo problema infraestrutural. O Speed Index é a métrica que melhor o captura porque integra toda a janela de carregamento, em vez de medir um único instante.

Porque é que o CLS bom agrava (e não atenua) este problema

O CLS é excelente nos dois runs (0,012 e 0,001) - não há layout jank. Combinado com o SI extremamente alto do Run 1, o quadro é:

A página não está partida. Está vazia, durante 27 segundos, sem indicar que algo está errado. Depois aparece de uma vez.

É uma das piores experiências possíveis: o utilizador olha para um ecrã branco e não tem sinais visuais de erro nem de progresso - nem spinner, nem skeleton, nem mensagem. Apenas branco. A maioria dos utilizadores abandona muito antes dos 27 segundos. Para crawlers de motores de busca e LLMs, este cenário corresponde a uma página efectivamente sem conteúdo, com todas as consequências de indexação que daí decorrem.

Implicações para o roadmap de correção

A causa-raiz tem de ser atacada no servidor de origem ou no que está entre ele e o cliente. Duas frentes de ação:

1. **Investigar a entrega de assets `/_next/static/*` no servidor de origem.** Os ficheiros existem (200 OK) mas a entrega é instável. Verificar logs de I/O do servidor, throughput de disco, configuração de HTTP/2 (multiplexing, server push), e identificar se há um CDN/reverse proxy à frente que esteja a falhar em alguns pedidos. A simultaneidade dos atrasos (>20 chunks todos a terminar perto dos 17 segundos) sugere um problema de concurrent connections ou de pool de workers a saturar.
2. **Monitorizar throughput, não apenas status.** Os monitores de uptime tradicionais só vêem o 200 OK e classificam o site como "online". É preciso monitorar **tempo até byte final** (não apenas TTFB) e **taxa de transferência por pedido** para detetar este tipo de degradação antes de ser reportada por utilizadores.

4 dos 5 sitemaps declarados em robots.txt devolvem 404

Apenas https://toursxplorer.com/sitemap.xml está acessível. Os restantes quatro estão partidos:

- `toursitemap.xml` — 404
- `locationsitemap.xml` — 404
- `attractionsitemap.xml` — 404
- `blogsitemap.xml` — 404

Impacto

O efeito é duplo e ambos os lados são prejudiciais. Por um lado, **o crawl budget é desperdiçado**: cada vez que o Googlebot lê o `robots.txt`, segue as cinco URLs declaradas e bate em quatro 404s. Embora o Google tolere isto, é ruído desnecessário nos relatórios do Search Console e atrasa a descoberta de URLs novas. Por outro lado, e mais grave, **secções inteiras do site podem ficar fora da indexação** - se as URLs de tours, locations, attractions e blog dependiam destes sitemaps para serem descobertas (em vez de via links internos), grande parte do catálogo pode estar invisível para os motores de busca.

Recomendações

A correção é simples e deve ser priorizada. Há dois caminhos possíveis:

1. **Repor os sitemaps em falta.** Se já existiram, restaurar a geração automática de `toursitemap.xml`, `locationsitemap.xml`, `attractionsitemap.xml` e `blogsitemap.xml`. Idealmente, gerados dinamicamente pelo Next.js (`app/sitemap.ts` ou route handlers) a partir da mesma fonte de dados do site, para nunca ficarem desatualizados.
2. **Consolidar tudo num sitemap index.** Manter apenas `sitemap.xml` como sitemap index que aponta para sub-sitemaps reais, e atualizar o `robots.txt` para declarar apenas essa URL. Esta é a abordagem padrão da indústria para sites de grande catálogo.

Independentemente da opção escolhida, depois da correção:

- Validar cada sitemap.
- Submeter os sitemaps válidos no Google Search Console e Bing Webmaster Tools.
- Confirmar no Search Console que o número de URLs descobertas/indexadas corresponde ao esperado para cada secção.

Hreflang - URLs incorretos nas tags alternate

Problema

As tags `hreflang` estão a apontar para URLs construídos com o **slug do idioma da página atual** em vez do slug real e traduzido de cada versão linguística. Ou seja, o sistema apenas troca o prefixo de idioma (`/en/`, `/pt-PT/`, etc.) e mantém o slug da página onde estamos a ser servidos.

Por exemplo:

- https://toursxplorer.com/fr/blog/category/peche-sportive

```
<link rel="alternate" hreflang="en"
href="https://toursxplorer.com/en/blog/category/peche-sportive">
<link rel="alternate" hreflang="de"
href="https://toursxplorer.com/de/blog/category/peche-sportive">
<link rel="alternate" hreflang="es"
href="https://toursxplorer.com/es/blog/category/peche-sportive">
<link rel="alternate" hreflang="fr"
href="https://toursxplorer.com/fr/blog/category/peche-sportive">
```

Neste exemplo, estamos numa página em francês (`/fr/blog/category/peche-sportive`) e todas as URLs `alternate` reutilizam o slug `peche-sportive`, independentemente do idioma. A URL para a versão portuguesa devia ser https://toursxplorer.com/pt-PT/blog/category/pesca-desportiva, não `…/peche-sportive`. O site faz redirect destas URLs "erradas" para a URL com o slug correto em cada idioma após o utilizador (ou crawler) seguir o link - o que confirma que o conteúdo final existe e está traduzido, mas o `hreflang` não está a apontar diretamente para ele.

Impacto

Em SEO, **hreflang só deve apontar para URLs finais** (200 OK, sem redirect). Quando aponta para uma URL que faz redirect, o Google é explícito na documentação: o sinal de hreflang é desvalorizado ou ignorado. Em concreto, o que acontece neste caso:

- **Cluster hreflang quebrado** - o Google pode deixar de tratar as 7 versões linguísticas como alternativas legítimas umas das outras, perdendo o benefício de mostrar a versão certa ao utilizador certo (PT para utilizadores em Portugal, IT para Itália, etc.).
- **Crawl budget desperdiçado** - cada visita do crawler à página faz 7 pedidos a URLs intermediárias que vão redirecionar, em vez de aterrar diretamente nas finais. Multiplicado por todas as categorias de blog, tours, locations e attractions do site, é um custo significativo.

- **Sinal de canónico ambíguo** - se as URLs com slug não-traduzido respondem 200 em vez de redirecionar, o Google pode chegar a indexar **versões duplicadas** da mesma página (com slug FR e com slug PT), diluindo autoridade.
- **Confusão para LLMs e GEO** - crawlers como GPTBot e ClaudeBot processam estes sinais de forma ainda menos tolerante que o Google e podem simplesmente não associar as versões.

Recomendações

A correção tem de ser feita na lógica de geração das tags `hreflang`, e não com regras de redirect adicionais. O padrão correto é:

1. **Para cada página, gerar as URLs alternativas a partir do slug traduzido na língua de destino** - não do slug da página onde estamos. Isto implica que o sistema de i18n tenha uma tabela de mapeamento entre slugs em todas as línguas (`pesca-desportiva` ↔ `peche-sportive` ↔ `sport-fishing` ↔ `pesca-deportiva` ↔ etc.) e a saiba consultar no momento do render.
2. **Validar que cada URL declarada em `hreflang` responde 200 OK** (sem redirect) - pode ser feito programaticamente em CI/CD a cada deploy, ou através do relatório "Internacionalização" do Search Console.
3. **Reciprocidade** - confirmar que se a página PT aponta para a EN, a página EN aponta de volta para a PT. Sem reciprocidade, o cluster hreflang também é ignorado.
4. **`x-default`** - confirmar que aponta para a versão de fallback correta (atualmente EN) e que essa URL também é a URL canónica final, não uma com slug noutro idioma.

Problemas de redirects

Redirects via JavaScript e Meta Refresh

Problema

Foram identificados volumes muito elevados de redirects implementados em camadas que não são as recomendadas para SEO:

- **842 páginas com redirects via JavaScript** (`window.location`, `router.push`, ou similares executados no cliente).
- **3.804 páginas com meta refresh redirects** (`<meta http-equiv="refresh" content="0;url=…">`).

No total, **mais de 4.600 páginas** estão a usar mecanismos de redirect que o Google trata como inferiores aos 301/302 ao nível do servidor.

Impacto

A documentação oficial do Google é explícita sobre a hierarquia de redirects: HTTP 301/308 (permanente) e 302/307 (temporário) são os métodos preferidos. Meta refresh e JavaScript redirects são suportados, mas tratados como **soft signals** - funcionam, mas com custos significativos:

- **Atraso na consolidação de sinais** - com um 301 ao nível do servidor, o Google atualiza o índice e transfere autoridade de forma relativamente rápida. Com meta refresh ou JS, o crawler tem de renderizar a página (no caso do JS), interpretar a instrução, e *depois* seguir o redirect. Pode demorar semanas a meses até consolidar a URL de destino, durante os quais a URL antiga continua indexada.
- **Custo de crawl multiplicado** - em vez de uma única resposta 301 (poucos bytes, sem render), o Googlebot tem de descarregar a página HTML inteira (todo o markup, CSS, JS), processá-la, e só depois seguir o redirect. Multiplicado por 4.646 páginas, é um custo de crawl budget enorme - especialmente prejudicial num site com a instabilidade intermitente identificada anteriormente.
- **Renderização no caso do JS** - o Googlebot consegue executar JavaScript, mas fá-lo numa **segunda fase** de indexação (rendering queue) que pode demorar dias. As 842 páginas com JS redirects podem ficar dias ou semanas em limbo antes de o redirect ser sequer detetado.
- **LLMs/GEO praticamente não seguem JS redirects** - crawlers como GPTBot, ClaudeBot e PerplexityBot têm capacidades de renderização limitadas ou nulas. As 842 páginas com JS redirects são, na prática, **invisíveis para AI Overviews, ChatGPT e Perplexity** - o que prejudica a visibilidade em GEO precisamente onde o tráfego cresce mais rápido.

- **Risco de soft 404** - meta refresh com `content="0;url=…"` para destinos inexistentes ou inconsistentes pode ser interpretado como soft 404, retirando essas páginas do índice.
- **Experiência do utilizador degradada** - meta refresh e JS redirects são percetíveis: o utilizador vê um flash de página ou um ecrã em branco antes de ser redirecionado. Soma-se aos problemas de LCP/Speed Index já identificados.
- **Métricas de Web Vitals contaminadas** - a página de redirect conta como uma navegação, com o seu próprio LCP/CLS, que se acumula com o da URL final.

Recomendações

A correção é estrutural e quase sempre vale o esforço:

1. **Migrar para 301 ao nível do servidor.** Idealmente expressos como regras no Next.js (`next.config.js` → `redirects()`) ou no reverse proxy (Nginx/Cloudflare Rules). Cada uma das 4.646 páginas deve ser substituível por uma regra de redirect HTTP que devolve `301` (permanente) ou `302` (temporário) diretamente, sem render.
2. **Mapear e categorizar.** Antes de migrar, exportar a lista completa das 842 + 3.804 URLs e respetivos destinos. Categorizar por padrão (mudanças de slug, mudanças de estrutura, redirects de idioma, redirects de marketing, etc.) - muitos vão poder ser cobertos por **regras genéricas** em vez de regras 1-para-1.
3. **Eliminar os que já não fazem sentido.** Numa amostra desta dimensão, é provável que muitos redirects sejam históricos e apontem para URLs que já não existem ou foram substituídas. Em vez de migrar todos, fazer uma triagem.
4. **Validar destinos finais.** Cada redirect deve apontar para uma URL que devolve 200 OK, sem cadeias de redirect adicionais. Cadeias longas (`A → B → C → D`) são quase tão prejudiciais como JS redirects.
5. **Submeter ao Search Console.** Após a migração, usar a ferramenta de "URL Inspection" do GSC numa amostra representativa para confirmar que o Google está a seguir corretamente os novos 301s.
6. **Monitorizar.** Acompanhar o relatório "Páginas" do Search Console durante 4-8 semanas após a migração para confirmar que as URLs antigas são removidas do índice e que as novas absorvem o tráfego e autoridade.

Em resumo: estes 4.646 redirects são uma das maiores oportunidades de SEO técnico do site, tanto pelo **volume** como pela **gravidade** do impacto em crawl budget, indexação e visibilidade em LLMs.

Meta Refresh em Loop (auto-redirect para o mesmo URL)

Problema

Cerca de **150 páginas** apresentam um bug crítico: o servidor responde `200 OK` mas injeta no `<head>` uma tag meta-refresh cujo destino é **exatamente o mesmo URL que o browser acabou de pedir**.

Exemplo:

- https://toursxplorer.com/pt-BR/location/asia/turkey-republic-of-turkey/marmara-region/istanbul
- https://toursxplorer.com/pt-BR/location/asia/india/rajasthan

```
<meta id="__next-page-redirect" http-equiv="refresh"
      content="0;url=/pt-BR/location/asia/turkey-republic-of-turkey/marmara-
region/istanbul"/>
```

O fluxo resultante é um loop infinito puramente do lado do cliente:

1. O browser pede a URL → recebe HTML com status 200.
2. Lê a meta-refresh e faz nova navegação para a mesma URL.
3. Recebe o mesmo HTML, com a mesma meta-refresh.
4. Repete até o browser intervir ou o utilizador abandonar a página.

Como o status HTTP é 200, ferramentas como `curl` ou auditorias só de cabeçalhos não detetam o problema - só é visível em renderização real. O atributo `id="__next-page-redirect"` identifica isto como uma tag injetada pelo próprio Next.js, o que sugere uma falha na lógica interna de redirecionamento do framework (provavelmente uma rota que devia redirecionar para um destino canónico mas está a calcular o destino como sendo a própria URL atual).

Impacto

Este é, em termos práticos, um erro mais grave do que um 404. Um 404 sinaliza ao crawler e ao utilizador que a página não existe. Aqui o servidor diz "página existe, sucesso" mas o conteúdo é inutilizável:

- **Página completamente inacessível ao utilizador** - quem aterre numa destas 150 URLs vê um flash de página seguido de um loop. O browser eventualmente bloqueia o redirect (Chrome corta após ~20 iterações com mensagem `ERR_TOO_MANY_REDIRECTS` em alguns casos, ou simplesmente fica preso noutros).
- **Conteúdo nunca chega a ser indexado** - o Googlebot lê a meta-refresh, segue-a, recebe a mesma instrução, e em algum momento desiste. A página fica num estado ambíguo: status 200

mas sem conteúdo útil que o Google possa indexar. Pior do que uma página excluída - fica indecisa.

- **Sinais contraditórios** - o Google vê 200 OK (página válida) + meta-refresh para si própria (sinal de redirect). Com sinais contraditórios, o tratamento é imprevisível: a URL pode ser indexada com conteúdo vazio, marcada como soft 404, ou simplesmente ignorada.
- **Invisível a LLMs/GEO** - GPTBot, ClaudeBot e PerplexityBot têm muito menos tolerância para comportamentos anómalos. Estas 150 páginas são, na prática, invisíveis em AI Overviews e citações de LLMs.
- **Loops contam como página vista no GA** - cada iteração do refresh dispara eventos de pageview, inflacionando artificialmente métricas do Google Analytics e potencialmente bounce rate.
- **Custo computacional** - cada iteração do loop é um pedido completo ao servidor (já fragilizado pelos 502s intermitentes). 150 URLs × N utilizadores × N iterações por loop = carga desnecessária.

Recomendações

A correção é prioritária e deve ser tratada como bug, não como otimização:

1. **Localizar a origem no código** - a tag tem `id="__next-page-redirect"`, que é injetada quando o Next.js executa uma chamada a `redirect()` (de `next/navigation`) ou retorna `{ redirect: ... }` de `getServerSideProps`/`getStaticProps`. Procurar onde a lógica de redirect está a calcular o destino como sendo a URL atual - o bug está quase certamente numa comparação de slugs/idiomas que falha em casos extremos (e.g. página `pt-BR` que tenta redirecionar para a sua versão canónica e calcula o canónico como sendo a si própria).
2. **Adicionar guarda no servidor** - antes de emitir qualquer redirect, comparar a URL de destino com a URL de origem. Se forem iguais, devolver 200 com o conteúdo final (sem meta-refresh) em vez de injetar a tag. Isto previne o bug independentemente da causa-raiz.
3. **Validação em CI/CD** - adicionar um teste automatizado que renderize uma amostra de URLs com um browser headless e detete a presença de `<meta http-equiv="refresh">` cujo destino coincide com a URL pedida. Se for detetada, falha o build.
4. **Search Console** - após a correção, fazer "Validate fix" no relatório de cobertura/erros e pedir reindexação das URLs corrigidas. Algumas podem ter sido marcadas como soft 404 e demoram a recuperar.
5. **Monitorização** - adicionar uma regra no analytics ou logs do reverse proxy que alerte se uma sessão fizer >3 navegações para a mesma URL em menos de 5 segundos, como indicador de loop residual.