

RETORNA DIGITAL

Kit de Prompts para Desenvolvedores

30 prompts prontos para debugging, refatoração, testes, arquitetura, code review e documentação.

GRÁTIS

Menos prompt genérico. Mais contexto, hipótese, teste e código verificável.

Como usar este kit

Escolha o cenário, substitua os campos entre **[colchetes]** e forneça à IA o contexto mínimo necessário para entender o projeto.

Importante: valide código gerado, execute testes e consulte documentação oficial para bibliotecas, APIs e comportamentos que podem ter mudado.

1	Escolha o problema
2	Forneça contexto real
3	Peça análise antes da mudança
4	Teste o resultado

Debug e investigação

01 Investigar um bug com contexto completo

Objetivo: Organizar hipóteses antes de alterar código.

Atue como engenheiro de software sênior e me ajude a investigar um bug.

Stack: [STACK]
Comportamento esperado: [ESPERADO]
Comportamento atual: [ATUAL]
Erro ou log: [ERRO/LOG]
Trecho relevante: [CÓDIGO]
O que já testei: [TESTES JÁ FEITOS]

Quero:

1. hipóteses ordenadas por probabilidade;
2. evidências que confirmariam ou descartariam cada hipótese;
3. menor sequência de testes para isolar a causa;
4. provável causa raiz, somente se houver evidência suficiente;
5. correção mínima sugerida.

Não invente comportamento de APIs ou bibliotecas que não esteja demonstrado no contexto.

Dica: Inclua logs e passos para reproduzir. Isso costuma valer mais do que colar o projeto inteiro.

02 Explicar uma stack trace

Objetivo: Transformar erro em plano de investigação.

Analise esta stack trace:

[STACK TRACE]

Contexto da aplicação:
[CONTEXT0]

Explique:

- onde o erro provavelmente começou;
- quais linhas são consequência e quais merecem investigação;
- possíveis causas;
- quais valores devo inspecionar;
- próximos passos para reproduzir de forma determinística.

Se faltarem dados, diga exatamente quais dados são necessários.

Dica: Informe versão da linguagem, framework e ambiente quando forem relevantes.

03 Comparar código que funciona e código que falha

Objetivo: Encontrar a diferença que causa o problema.

```
Tenho duas implementações do mesmo fluxo.  
  
Versão que funciona:  
[CÓDIGO A]  
  
Versão com problema:  
[CÓDIGO B]  
  
Sintoma:  
[SINTOMA]  
  
Compare as duas semanticamente.  
  
Liste apenas diferenças capazes de explicar o sintoma e, para cada uma:  
- por que pode causar o problema;  
- como validar;  
- correção mínima.  
  
Ignore diferenças puramente estéticas.
```

Dica: Ótimo para regressões depois de refactors ou migrações.

04 Investigar problema intermitente

Objetivo: Criar observabilidade para erros difíceis de reproduzir.

```
Tenho um problema intermitente:  
  
[DESCRIÇÃO]  
  
Stack: [STACK]  
Frequência aproximada: [FREQUÊNCIA]  
Ambiente: [AMBIENTE]  
Logs atuais: [LOGS]  
  
Crie um plano de instrumentação para descobrir a causa.  
  
Inclua:  
- quais eventos registrar;  
- campos úteis;  
- correlação entre requests/processos;  
- métricas;  
- como evitar logs excessivos ou dados sensíveis;  
- critérios para concluir a investigação.
```

Dica: Para bugs intermitentes, melhorar evidência costuma ser melhor que tentar corrigir no escuro.

05 Encontrar regressão provável

Objetivo: Rastrear quando um comportamento deixou de funcionar.

Um comportamento funcionava antes e agora falha.

Antes:

[COMPORTAMENTO ANTERIOR]

Agora:

[COMPORTAMENTO ATUAL]

Mudanças recentes conhecidas:

[MUDANÇAS]

Erro/log:

[ERRO]

Crie uma estratégia para localizar a regressão.

Priorize:

- mudanças mais relacionadas ao sintoma;
- configuração;
- dependências;
- contratos de API;
- migrations;
- cache;
- feature flags.

Sugira a ordem mais barata de verificação.

Dica: Inclua commits ou PRs recentes quando tiver acesso a eles.

Refatoração e qualidade

06 Refatorar sem mudar comportamento

Objetivo: Simplificar código preservando regras existentes.

Refatore o código abaixo sem alterar comportamento observável:

[CÓDIGO]

Contexto:

[CONTEXTO]

Prioridades:

1. legibilidade;
2. redução de duplicação;
3. menor complexidade;
4. manutenção;
5. performance somente quando houver ganho real.

Restrições:

[RESTRIÇÕES]

Apresente:

- problemas encontrados;
- versão refatorada;
- riscos de regressão;
- testes que deveriam proteger a mudança.

Não crie abstrações sem necessidade.

Dica: Informe quais comportamentos são críticos e não podem mudar.

07 Reduzir complexidade de uma função

Objetivo: Quebrar lógica difícil de manter.

Analise esta função:

[CÓDIGO]

Quero reduzir complexidade sem espalhar a regra de negócio pelo projeto.

Identifique:

- responsabilidades misturadas;
- condicionais redundantes;
- estados implícitos;
- efeitos colaterais;
- trechos que merecem extração.

Depois proponha uma implementação mais simples.

Evite padrões de projeto apenas por estética.

Dica: Uma função menor não é automaticamente melhor. Preserve coesão.

08 Eliminar duplicação

Objetivo: Encontrar reutilização útil sem generalizar cedo demais.

Estes trechos possuem duplicação:

[CÓDIGOS]

Analise o que realmente é regra compartilhada e o que apenas parece semelhante.

Proponha uma refatoração que:

- elimine duplicação relevante;
- mantenha diferenças explícitas;
- evite configuração excessiva;
- seja fácil de testar.

Explique por que a abstração proposta tem limite adequado.

Dica: Duas funções parecidas ainda podem representar regras de negócio diferentes.

09 Melhorar nomes de código

Objetivo: Deixar intenção mais explícita.

Revise os nomes deste trecho:

[CÓDIGO]

Contexto de domínio:

[CONTEXTO]

Sugira melhorias somente para nomes que realmente prejudiquem entendimento.

Considere:

- variáveis;
- funções;
- componentes;
- classes;
- tipos;
- flags booleanas.

Prefira nomes que expressem intenção e domínio em vez de detalhes de implementação.

Dica: Evite renomear tudo só para tornar o diff maior.

10 Revisar tratamento de erros

Objetivo: Tornar falhas previsíveis e diagnósticas.

Revise o tratamento de erros deste fluxo:

[CÓDIGO]

Contexto:

[CONTEXTO]

Avalie:

- erros engolidos;
- mensagens vagas;
- exceções usadas como controle de fluxo;
- retries;
- timeouts;
- fallback;
- logging;
- exposição de dados sensíveis.

Proponha melhorias com o menor impacto arquitetural possível.

Dica: Diferencie erro recuperável, erro de usuário e falha de infraestrutura.

Testes

11 Criar casos de teste

Objetivo: Cobrir comportamento sem testar detalhes internos.

Crie uma matriz de testes para:

[FUNCIONALIDADE/CÓDIGO]

Regras:

[REGRAS]

Liste casos:

- caminho feliz;
- limites;
- entradas inválidas;
- estados vazios;
- falhas externas;
- regressões prováveis.

Para cada caso informe:

- cenário;
- entrada;
- resultado esperado;
- tipo de teste recomendado.

Não gere casos redundantes.

Dica: Comece pela matriz antes de escrever dezenas de testes.

12 Escrever testes unitários

Objetivo: Gerar testes claros para uma unidade específica.

Escreva testes unitários para:

[CÓDIGO]

Stack de testes:

[FRAMEWORK]

Comportamentos que precisam ser protegidos:

[COMPORTAMENTOS]

Siga o estilo existente:

[EXEMPLO DE TESTE EXISTENTE]

Evite testar detalhes de implementação.

Mocke apenas dependências externas necessárias.

Priorize testes determinísticos e legíveis.

Dica: Forneça um teste existente do projeto para a IA copiar o padrão correto.

13 Criar teste de regressão

Objetivo: Transformar um bug corrigido em proteção permanente.

Este bug ocorreu:

[BUG]

Causa raiz:

[CAUSA]

Correção:

[CORREÇÃO]

Crie o menor teste de regressão capaz de falhar antes da correção e passar depois dela.

Stack:

[STACK]

Explique por que esse teste protege especificamente contra a regressão.

Dica: Um teste de regressão deve reproduzir a causa, não apenas a mensagem do erro.

14 Revisar suíte de testes

Objetivo: Encontrar testes frágeis ou de baixo valor.

Analise esta suíte:

[TESTES]

Código testado:

[CÓDIGO]

Identifique:

- testes duplicados;
- mocks excessivos;
- asserts fracos;
- dependência de timing;
- testes de implementação;
- lacunas de comportamento.

Sugira uma versão mais enxuta sem reduzir proteção relevante.

Dica: Quantidade de testes não é o mesmo que cobertura de comportamento.

15 Planejar testes de integração

Objetivo: Validar contratos entre partes do sistema.

Preciso testar a integração entre:

[COMPONENTES/SERVIÇOS]

Fluxo:

[FLUXO]

Dependências externas:

[DEPENDÊNCIAS]

Crie um plano de testes de integração com:

- cenários prioritários;
- dados necessários;
- o que deve ser real;
- o que pode ser simulado;
- limpeza de dados;
- critérios de sucesso.

Evite transformar o teste de integração em E2E desnecessariamente.

Dica: Defina claramente qual contrato a integração precisa garantir.

Arquitetura e implementação

16 Planejar uma feature antes de codar

Objetivo: Quebrar uma demanda em implementação segura.

Planeje a implementação desta feature:

[FEATURE]

Stack:

[STACK]

Arquitetura atual:

[ARQUITETURA]

Restrições:

[RESTRIÇÕES]

Divida em:

- fluxo funcional;
- mudanças necessárias;
- contratos;
- estados;
- persistência;
- tratamento de erros;
- testes;
- rollout.

Prefira aproveitar padrões existentes do projeto em vez de introduzir novas camadas.

Dica: Cole arquivos ou padrões semelhantes do projeto para reduzir suposições.

17 Avaliar duas abordagens técnicas

Objetivo: Comparar alternativas com critérios objetivos.

Compare estas duas abordagens para [PROBLEMA]:

Abordagem A:
[A]

Abordagem B:
[B]

Contexto:
[CONTEXTO]

Avalie:

- complexidade;
- manutenção;
- performance;
- testabilidade;
- acoplamento;
- risco de migração;
- custo operacional.

Não escolha pela novidade. Explique trade-offs e em quais condições cada opção é melhor.

Dica: Use requisitos reais. Sem contexto, comparações arquiteturais viram opinião.

18 Desenhar contrato de API

Objetivo: Criar endpoints consistentes e previsíveis.

Ajude a definir o contrato de API para:

[FUNCIONALIDADE]

Padrões existentes:
[PADRÕES]

Defina:

- método;
- rota;
- request;
- response;
- erros;
- status HTTP;
- idempotência, se aplicável;
- paginação, se aplicável;
- validações.

Mantenha consistência com o projeto e evite campos redundantes.

Dica: Inclua exemplos de endpoints existentes para preservar o padrão.

19 Planejar migration de banco

Objetivo: Alterar esquema com risco controlado.

Preciso fazer esta mudança no banco:

[MUDANÇA]

Banco e versão:

[BANCO]

Volume aproximado:

[VOLUME]

Aplicação atual:

[CONTEXTO]

Planeje a migration considerando:

- compatibilidade;
- locks;
- backfill;
- rollback;
- deploy em etapas;
- índices;
- validação pós-deploy.

Não assumo que a tabela pode ficar indisponível.

Dica: Informe volume e versão do banco. Isso pode mudar totalmente a estratégia.

20 Planejar integração com serviço externo

Objetivo: Reduzir risco de APIs de terceiros.

Preciso integrar com:

[SERVIÇO]

Documentação/contrato relevante:

[CONTRATO]

Fluxo desejado:

[FLUXO]

Planeje:

- autenticação;
- requests;
- validação de responses;
- timeouts;
- retries;
- idempotência;
- webhooks;
- observabilidade;
- armazenamento;
- comportamento em indisponibilidade.

Não invente campos ou endpoints ausentes da documentação.

Dica: Cole o trecho atual da documentação da API para evitar respostas desatualizadas.

Code review, PRs e documentação

21 Fazer code review focado em risco

Objetivo: Revisar mudança pelo impacto real.

Faça code review desta alteração:

[DIFF/CÓDIGO]

Contexto da task:

[CONTEXTO]

Procure principalmente:

- bugs;
- regressões;
- condições de corrida;
- segurança;
- contratos quebrados;
- estados não tratados;
- performance relevante;
- ausência de testes.

Não faça comentários apenas de preferência estética.

Classifique cada achado por impacto e explique como reproduzir ou validar.

Dica: Um bom review aponta risco concreto, não transforma estilo pessoal em regra.

22 Criar descrição de Pull Request

Objetivo: Explicar a mudança de forma objetiva.

Crie uma descrição de Pull Request com base nestas alterações:

[ALTERAÇÕES]

Motivação:

[MOTIVAÇÃO]

Testes:

[TESTES]

Riscos ou observações:

[RISCOS]

Use estrutura profissional e objetiva.

Inclua:

- contexto;
- solução;
- impacto;
- validação;
- observações relevantes.

Não liste arquivos alterados a menos que isso seja essencial.

Dica: A descrição deve permitir entender o PR sem ler primeiro todos os arquivos.

23 Gerar documentação técnica

Objetivo: Registrar como um fluxo realmente funciona.

Documente tecnicamente este fluxo:

[FLUXO/CÓDIGO]

Público da documentação:

[PÚBLICO]

Inclua apenas o necessário:

- objetivo;
- componentes envolvidos;
- sequência;
- contratos;
- configuração;
- erros importantes;
- como testar;
- limitações.

Não invente comportamento não demonstrado pelo código ou contexto.

Dica: Documentação boa responde o que alguém precisa saber para operar ou alterar o fluxo.

24 Criar README de projeto

Objetivo: Dar entrada rápida a um novo desenvolvedor.

Crie ou melhore o README deste projeto.

Informações:

[INFORMAÇÕES]

Inclua:

- objetivo;
- requisitos;
- instalação;
- configuração;
- comandos;
- execução local;
- testes;
- estrutura essencial;
- troubleshooting básico.

Não adicione etapas que não foram confirmadas.

Dica: Teste os comandos antes de publicar o README quando possível.

25 Explicar código legado

Objetivo: Construir um mapa mental antes de alterar.

Explique este código legado:

[CÓDIGO]

Contexto conhecido:

[CONTEXTO]

Quero entender:

- responsabilidade principal;
- fluxo de dados;
- regras de negócio;
- dependências;
- efeitos colaterais;
- pontos frágeis;
- riscos ao alterar.

Separe claramente fatos observáveis de hipóteses.

Dica: Antes de refatorar legado, descubra o contrato que ele já cumpre.

Performance, segurança e produtividade

26 Investigar gargalo de performance

Objetivo: Encontrar custo antes de otimizar.

Tenho este problema de performance:

[SINTOMA]

Métricas disponíveis:

[MÉTRICAS]

Código/consulta:

[CÓDIGO]

Ambiente:

[AMBIENTE]

Crie uma estratégia de profiling.

Liste:

- possíveis gargalos;
- métricas adicionais;
- experimentos;
- mudanças de baixo risco;
- otimizações que só devem ser feitas após medição.

Não proponha cache como resposta automática.

Dica: Meça antes e depois. Performance sem baseline vira sensação.

27 Revisar query SQL

Objetivo: Melhorar consulta preservando resultado.

Analise esta query:

[SQL]

Banco e versão:

[BANCO]

Schema/índices relevantes:

[SCHEMA]

Volume aproximado:

[VOLUME]

Avalie:

- filtros;
- joins;
- índices;
- cardinalidade;
- ordenação;
- paginação;
- funções que impeçam uso de índice;
- possíveis N+1 relacionados.

Proponha alterações somente quando houver justificativa.

Dica: Inclua EXPLAIN/EXPLAIN ANALYZE quando disponível.

28 Revisão de segurança defensiva

Objetivo: Identificar riscos comuns em uma implementação.

Faça uma revisão defensiva deste código:

[CÓDIGO]

Contexto:

[CONTEXTO]

Procure riscos como:

- autorização;
- validação de entrada;
- exposição de dados;
- secrets;
- injeção;
- XSS/CSRF quando aplicável;
- SSRF quando aplicável;
- upload inseguro;
- rate limiting;
- logs sensíveis.

Para cada risco, explique impacto e correção defensiva.

Não forneça instruções de exploração contra sistemas reais.

Dica: Segurança deve ser revisada junto com a regra de autorização do negócio.

29 Transformar task vaga em plano técnico

Objetivo: Converter requisito solto em trabalho executável.

Recebi esta task:

[TASK]

Contexto do projeto:

[CONTEXTO]

Transforme em um plano técnico objetivo contendo:

- entendimento do problema;
- dúvidas ou premissas;
- critérios de aceite;
- áreas afetadas;
- passos de implementação;
- testes;
- riscos;
- dependências.

Evite aumentar escopo além do pedido.

Dica: Ótimo para preparar uma task antes de abrir o editor.

30 Criar prompt de contexto para uma sessão de desenvolvimento

Objetivo: Preparar uma IA para trabalhar respeitando o projeto.

Quero iniciar uma sessão de desenvolvimento assistido por IA.

Projeto:
[PROJETO]

Stack:
[STACK]

Objetivo:
[OBJETIVO]

Padrões importantes:
[PADRÕES]

Restrições:
[RESTRIÇÕES]

Arquivos relevantes:
[ARQUIVOS]

Crie um prompt inicial conciso que instrua a IA a:

- primeiro entender o código existente;
- preservar padrões;
- evitar dependências desnecessárias;
- implementar a menor solução correta;
- executar testes relevantes;
- reportar mudanças e riscos.

Não inclua instruções genéricas que não ajudem na tarefa.

Dica: Quanto mais contexto útil no início, menos retrabalho durante a sessão.

Boas práticas

- Forneça contexto, versões e erros reais.
- Mostre exemplos do padrão existente antes de pedir código novo.
- Peça hipóteses e formas de validação antes de aceitar uma causa raiz.
- Execute testes e lint depois de alterações geradas por IA.
- Consulte documentação oficial para APIs e bibliotecas que mudam com frequência.
- Nunca envie secrets, tokens ou dados sensíveis no prompt.

RETORNA DIGITAL

**Ferramentas prontas para trabalhar,
vender e automatizar.**

Este material faz parte da coleção gratuita da Retorna Digital.

retornaservicos.com.br