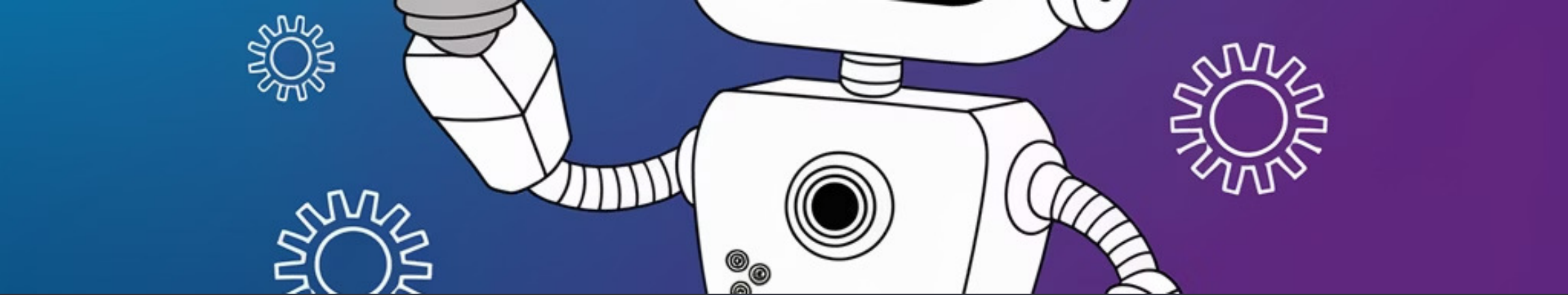


Prompt Engineering para Desenvolvedores



O que é Prompt Engineering

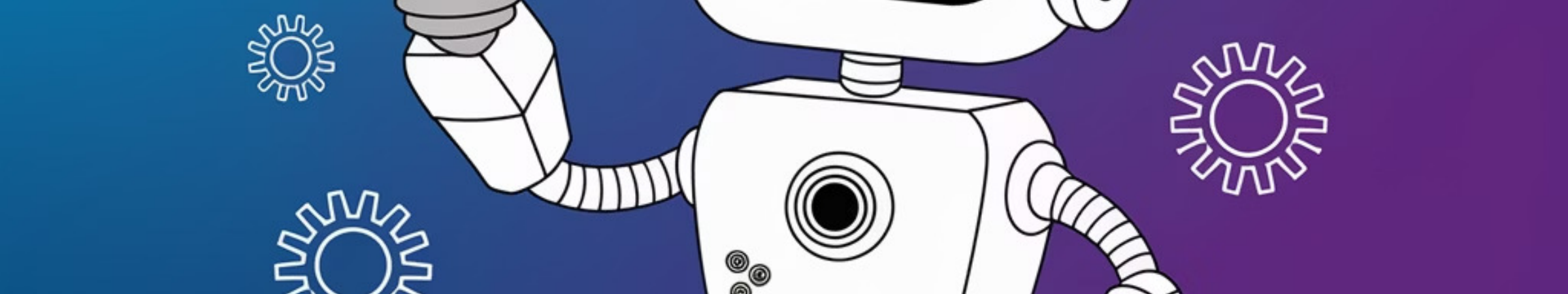
- ☐ A engenharia de prompts é o processo em que você orienta as soluções de inteligência artificial generativa (IA generativa) para gerar os resultados desejados.
- ☐ Embora a IA generativa tente imitar os humanos, ela requer instruções detalhadas para criar resultados relevantes e de alta qualidade.
- ☐ Na engenharia de prompt, você escolhe os formatos, frases, palavras e símbolos mais adequados para orientar a IA a interagir com seus usuários de forma mais significativa.

Importância para Desenvolvedores

- ☐ Desenvolver e manter
- ☐ Explorar possibilidades
- ☐ Automatizar tarefas repetitivas
- ☐ Obter soluções rápidas
- ☐ Auxiliar em processos complexos de desenvolvimento
- ☐ Aprimorar a colaboração

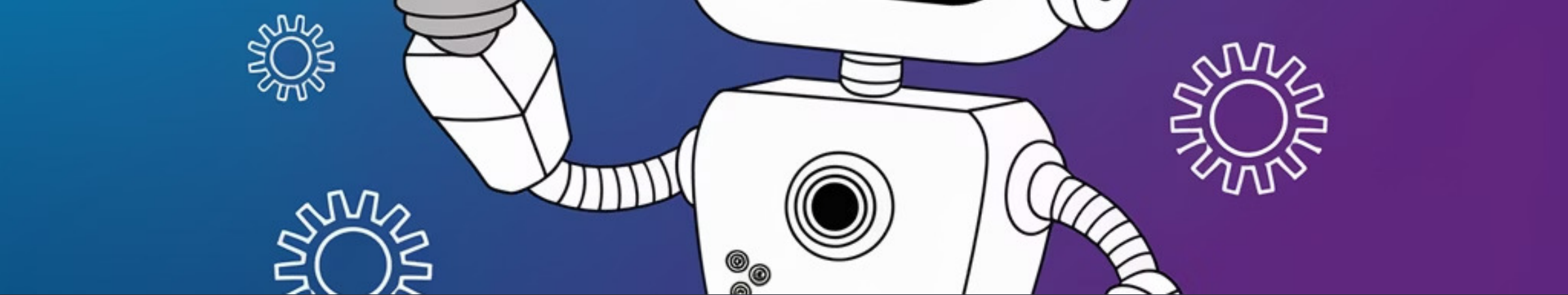
Utilização e possibilidades

- ☐ Documentação e Design Docs
- ☐ Implementação
- ☐ Code Review
- ☐ Mastermind e brainstorming



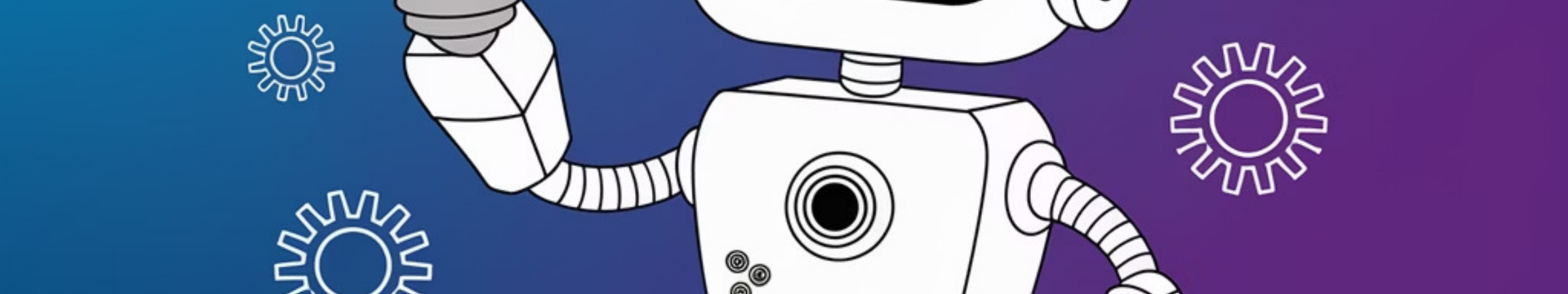
Documentação e Design Docs

- **Requisitos e Produto** // TRD, PRD, FRD, User Stories
- **Design e Arquitetura** // System Design (HL), Low-Level Design (LLD), C4
- **Decisões Técnicas** // ADRs, RFCs
- **Engineering Guidelines** // Coding Standards, Code Review, Testing, CI/CD, Security
- **Ops e Infra** // Runbook, Playbook, Infrastructure Design Document, Post-mortem



Implementação

- ☐ Exploração
- ☐ Contextualização
- ☐ Tarefas e Plano de Ação
- ☐ Workflow e Rules
- ☐ Testing
- ☐ Debugging
- ☐ Refactoring
- ☐ Análise de Performance
- ☐ Commit & Pull Requests



Code Review









- ☐ Análise de Discrepâncias // Implementação reflete a documentação
- ☐ Verificações de implementação de features
- ☐ Testabilidade e cobertura de código
- ☐ Verificação de bugs
- ☐ Documentação / Comentários Internos
- ☐ Coding Standards



Tipos de Prompts e Variações

Existem diversas categorias e técnicas de prompts que podem ser utilizadas por desenvolvedores para otimizar suas interações com IAs. Cada tipo oferece vantagens específicas dependendo do contexto e caso de uso, desde tarefas simples sem exemplos prévios até raciocínios complexos e estruturados que melhoram significativamente a qualidade e precisão das respostas.

Exemplos

-  — **Zero-Shot**
-  — **One-Shot / Few-Shot**
-  — **Chain of Thought (CoT)**
-  — **Skeleton of Thought (SoT)**
-  — **Tree of Thought (ToT)**
-  — **Self-Consistency**
-  — **Directional Stimulus**
-  — **ReActing**



Zero-Shot

Zero-Shot Prompting é uma técnica onde o modelo de linguagem recebe apenas a descrição da tarefa, **sem nenhum exemplo** anterior. O objetivo é avaliar a capacidade do modelo de generalizar e resolver o problema com base apenas na instrução textual.

Estudo

Essa abordagem ganhou destaque com o paper "**Language Models are Few-Shot Learners**" de Brown et al. (2020), publicado pela OpenAI. Nesse estudo, os pesquisadores demonstraram que modelos como o GPT-3 conseguem realizar tarefas complexas apenas com uma boa descrição da tarefa no prompt, mesmo sem exemplos anteriores.

Quando utilizar

- A tarefa é simples e bem conhecida pelo modelo (por exemplo, perguntas factuais).
- Não há tempo ou espaço para adicionar exemplos.
- Deseja-se testar a capacidade base do modelo de interpretar a tarefa.
- É preciso processar uma grande quantidade de tarefas distintas e breves com o menor custo de tokens.

Link do paper:



 arXiv.org

Language Models are Few-Shot Learners

Recent work has demonstrated substantial gains on many NLP tasks and benchmarks by pre-training on a large corpus of text followed by fine-tuning on a...





Zero-Shot

Vantagens

- Baixo custo de preparação do prompt.
- Alta escalabilidade.
- Rápido para experimentação

Limitações

- Pode falhar em tarefas mais complexas ou menos frequentes para o modelo.
- Não oferece controle sobre o formato da resposta.
- Dependente da compreensão implícita do modelo sobre a tarefa.
- Alucinação de respostas: o modelo pode gerar conteúdo incorreto com alta confiança.
- Variação com pequenas mudanças na formatação: prompts semanticamente equivalentes podem gerar resultados diferentes.
- Dificuldade em tarefas de raciocínio lógico, análises comparativas ou inferência fora do contexto factual.

Estratégias de Mitigação

Mesmo mantendo a proposta de não fornecer exemplos, algumas práticas ajudam a melhorar os resultados com Zero-Shot Prompting:

Especificidade:

Instruções claras, que detalham o que se espera, aumentam a precisão.

- **Exemplo ruim:** "Analise esse código"
- **Melhorado:** "Explique o que esse código Go faz e liste possíveis problemas de performance."

Linguagem declarativa e orientada à Tarefa:

Preferência por comandos diretos a perguntas abertas.

- **Exemplo:** "Liste os principais motivos para erros de memória em da linguagem Go."

Sinalização do formato esperado:

Solicitação explícita de listas, tópicos, parágrafos curtos etc.

- **Exemplo:** "Responda em forma de tópicos." ou "Escreva uma explicação de 3 parágrafos."



Zero-Shot

In-Context Instruction Learning

Mesmo sem exemplos, prompts estruturados com clareza (persona, formato, objetivo) ajudam o modelo a responder melhor. Instruções mais detalhadas melhoram a performance zero-shot.

- **Antes:** "Explique o que é uma goroutine"
- **Depois:** "Você é um especialista em Go. Escreva dois parágrafos explicando o que é uma goroutine, como ela é usada e quais são suas limitações. Seja claro, técnico e direto."

Boas práticas da Microsoft

- Especificar o papel do modelo ("Você é um especialista em...")
- Especificar a saída desejada ("Responda em tópicos" ou "Formato JSON")
- Garantir que o modelo compreenda a meta ("Seu objetivo é...")

Exemplo: "Você é um consultor técnico. Seu trabalho é analisar este trecho de código e sugerir melhorias de performance. Responda em tópicos e justifique cada sugestão."

Importante:

Apesar desse exemplo fornecer instruções estruturadas, **não apresenta nenhum exemplo anterior**. O modelo precisa inferir a tarefa com base apenas na instrução natural.



How to get Codex to produce the code you want!

Learn how to use AI models with prompt engineering



One-Shot / Few-Shot

Few-Shot Prompting é uma técnica onde fornecemos **um pequeno número de exemplos (normalmente entre 1 e 5)** para que o modelo entenda o padrão de entrada e saída antes de gerar uma nova resposta. O modelo "aprende" apenas com base nesses exemplos, dentro do próprio prompt, sem qualquer re-treinamento.

Estudo

A técnica foi formalizada no artigo seminal **"Language Models are Few-Shot Learners"** de Brown et al. (2020), que apresentou o GPT-3 e sua capacidade de realizar tarefas complexas com apenas alguns exemplos embutidos na entrada textual.

Quando utilizar

- A tarefa **tem múltiplas formas de execução válidas**, e você deseja orientar o estilo da resposta.
- O modelo **comete erros ou se comporta de forma inconsistente** em Zero-Shot.
- A tarefa é **relativamente complexa ou específica**, e pode se beneficiar de demonstrações diretas.
- É necessário **replicar padrões linguísticos, técnicos ou formais** específicos.

Link do paper:



 arXiv.org

Language Models are Few-Shot Learners

Recent work has demonstrated substantial gains on many NLP tasks and benchmarks by pre-training on a large corpus of text followed by fine-tuning on a...



One-Shot / Few-Shot

Vantagens

- **Precisão aumentada:** exemplos ajudam o modelo a compreender nuances da tarefa.
- **Consistência de estilo:** útil para gerar código/documentação com padronização.
- **Pouco custo de engenharia:** mais simples do que treinar modelos.

Limitações

- **Custo em tokens:** exemplos ocupam espaço no prompt, reduzindo espaço para contexto.
- **Dependência da qualidade dos exemplos:** exemplos ambíguos ou mal formulados comprometem a resposta.
- **Fragilidade à ordem:** mudar a ordem dos exemplos pode afetar o desempenho.

Estrutura e exemplos

Exemplo 1:
Entrada: <texto de entrada>
Saída: <resposta esperada>

Exemplo 2:
Entrada: <texto de entrada>
Saída: <resposta esperada>

Prompt:

"Gere testes unitários para funções em Go"

Exemplo:

Função:
func Soma(a, b int) int { return a + b }

Teste:
func TestSoma(t *testing.T) { r := Soma(2, 3) if r != 5 { t.Errorf("esperado 5, obteve %d", r) } }

Função:
func Multiplica(a, b int) int { return a * b }

Teste:

Resultado esperado:

O modelo gera o teste para **Multiplica** seguindo o estilo e estrutura do exemplo anterior, com nomes de funções, variáveis e mensagens consistentes.

Comparativo

Tipo de Prompt	Exemplos	Precisão	Controle de Saída	Custo
Zero-Shot	Nenhum	Média	Baixo	Baixo
One-Shot	1	Média+	Médio	Médio
Few-Shot	2-5	Alta	Alto	Alto



Chain of Thought (CoT)

Chain of Thought (CoT) é uma técnica de engenharia de prompt que instrui o modelo a externalizar seu raciocínio **passo a passo**, permitindo que ele resolva tarefas que exigem lógica, múltiplas etapas ou operações intermediárias. Em vez de apenas dar a resposta final, o modelo mostra seu processo de pensamento.

Estudo

A técnica foi formalizada no paper "**Chain-of-Thought Prompting Elicits Reasoning in Large Language Models**" de **Wei et al. (2022)**, demonstrando que grandes modelos como PaLM e GPT-3 apresentam desempenho significativamente superior em tarefas de raciocínio lógico e aritmético quando induzidos a pensar de forma encadeada.


Advanced Reasoning


- O CoT é a **fundação para os recursos de Advanced Reasoning em LLMs**, como GPT-4, Claude e Gemini. Esses modelos foram treinados com instruções e exemplos que incentivam raciocínio multietapas, explicações lógicas e reflexões auditáveis.
- Chain of Thought permite que o modelo não apenas **chegue à resposta**, mas também **demonstre como chegou até ela**, oferecendo transparência, confiabilidade e contexto técnico.

Quando utilizar

- Diagnóstico de falhas e bugs
- Planejamento lógico de processos
- Argumentações comparativas entre abordagens


Link do paper:



 arXiv.org

Chain-of-Thought Prompting Elicits Reasoning in Large Language Models

We explore how generating a chain of thought -- a series of intermediate reasoning steps -- significantly improves the ability of large language models to perform...





Chain of Thought (CoT)

Vantagens

- **Raciocínio explícito:** permite que o modelo demonstre seu processo de pensamento passo a passo.
- **Maior resolução de problemas complexos:** melhora significativamente o desempenho em tarefas que exigem múltiplas etapas de raciocínio.
- **Transparência e auditabilidade:** torna o processo decisório do modelo visível, facilitando a verificação da lógica utilizada.

Limitações

- Gera saídas mais longas, o que pode ser custoso em prompts com limite de tokens.
- Pode introduzir ruído se o modelo gerar cadeias de pensamento incorretas.
- Requer modelo suficientemente treinado para compreender e aplicar o passo a passo com qualidade.
- Se não combinado com critérios de parada, pode prolongar desnecessariamente o raciocínio.
 - "Pense passo a passo até chegar a uma conclusão única e final."
 - "Pare quando tiver certeza da resposta."
 - "Após concluir as etapas, retorne apenas a resposta final."

Estrutura e exemplos

Você é um engenheiro de software com 20 anos de experiência em sistemas concorrentes e distribuídos. Seu trabalho é revisar o código a seguir e identificar falhas ou melhorias. Pense passo a passo, justificando cada ponto com base nas práticas recomendadas em Go. Ao final, revise a sequência de etapas e forneça uma conclusão objetiva.

Use a seguinte estrutura:

Etapa 1: <descrição>

Etapa 2: <descrição> ...

Resultado final: <conclusão>



Chain of Thought (CoT)

1. Debugging com raciocínio lógico

Prompt:

Você é um desenvolvedor Go. Analise o seguinte código e explique, passo a passo, por que ele pode causar um erro:

```
var lista []string  
fmt.Println(lista[0])
```

Resposta esperada:

Etapa 1: A variável `lista` é declarada como slice de strings, mas não foi inicializada.

Etapa 2: Um slice não inicializado tem tamanho zero (`len(lista) == 0`).

Etapa 3: Acessar `lista[0]` viola o limite do slice, pois não há elementos.

Resultado final: o programa entra em pânico (panic) por acesso fora dos limites.



Chain of Thought (CoT)

2. Refatoração com justificativas técnicas

Prompt:

Reescreva a função abaixo seguindo o padrão early return e explique cada modificação passo a passo:

```
func Validar(u Usuario) error {  
    var err error  
    if u.Email == "" {  
        err = errors.New("email obrigatório")  
    } else {  
        if u.Senha == "" {  
            err = errors.New("senha obrigatória")  
        } else {  
            err = nil  
        }  
    }  
    return err  
}
```



Chain of Thought (CoT)

3. Planejamento básico

Prompt:

Descreva todas as etapas envolvidas na migração de uma aplicação monolítica para microserviços baseados em eventos. Para cada etapa, inclua pré-requisitos, riscos e validações. Organize o raciocínio em sequência lógica e valide antes de concluir.

Resposta esperada:

Etapa 1: Identificação dos domínios de negócio.

Etapa 2: Definição de fronteiras de contexto (bounded contexts).

Etapa 3: Extração de serviços mais isolados com menor acoplamento.

Etapa 4: Implementação de mensageria (ex: RabbitMQ) com fallback e DLQs.

Etapa 5: Implementação de observabilidade: logs, métricas e tracing.

Etapa 6: Testes de regressão antes de redirecionar tráfego.

Resultado final: Arquitetura distribuída validada com ganhos de resiliência e escalabilidade.



Chain of Thought (CoT)

Estratégias inspiradas na Anthropic Prompt Library

- **Persona + Objetivo + Estrutura clara:** contextualiza a função do modelo e define o tom da resposta.
- **Chamado à reflexão lógica:** "Pense passo a passo", "Justifique cada etapa".
- **Formato de saída padronizado:** etapas numeradas + conclusão objetiva.
- **Autoavaliação embutida:** "Verifique se todos os passos estão consistentes".
- **Critério de parada lógico:** encerrar ao atingir o raciocínio final.

Técnicas avançadas de CoT com delimitações estruturais (Anthropic-style)

Modelos como Claude e GPT respondem melhor quando o prompt apresenta **delimitações estruturais explícitas**. Uma técnica bastante utilizada pela Anthropic, segundo sua própria Claude Prompt Library, é o uso de **delimitadores XML-like** como `<thought>`, `<reasoning>`, `<answer>`, etc., para **separar raciocínio da resposta final**, melhorar a legibilidade, e tornar o prompt mais auditável.

Variações comuns de marcadores

- `<context>` — delimita contexto inicial ou sistema.
- `<thought>` ou `<reasoning>` — contém o raciocínio lógico.
- `<step>` — pode ser usado para dividir em etapas internas.
- `<answer>` — resposta direta, sem explicações.
- `<final_decision>` — quando há múltiplas alternativas sendo avaliadas.

Motivação para usar marcadores estruturais

- Reduz ambiguidade entre raciocínio e resposta.
- Permite que o modelo organize suas ideias de maneira hierárquica.
- Facilita análises posteriores do output por humanos ou outros sistemas.
- Ajuda o modelo a se manter dentro de um fluxo lógico, sem saltos.



Chain of Thought (CoT)

Exemplo

Prompt:

Você é um engenheiro de software especialista em arquitetura de sistemas web. Sua tarefa é pensar passo a passo sobre os requisitos técnicos para construir um backend de uma aplicação de Todo List colaborativa com autenticação de usuários, categorias personalizáveis e suporte a tarefas compartilhadas entre times.

Escreva seu raciocínio entre `<thought>` e o resumo técnico final entre `<answer>`. Use também `<step>` para dividir o pensamento por área de responsabilidade.

Resposta esperada:

```
<thought>
  <step>
    <title>1. Autenticação e gerenciamento de usuários</title>
    A aplicação deve suportar login via e-mail e senha com autenticação baseada em tokens JWT. Usuários precisam ser persistidos em um banco relacional com campos como email, senha hash, nome e data de criação.
  </step>

  <step>
    <title>2. Modelagem de tarefas</title>
    Cada tarefa deve conter título, descrição, status (pendente, concluída), data de vencimento, categoria e ID do criador. As tarefas devem poder ser atribuídas a múltiplos usuários.
  </step>

  <step>
    <title>3. Compartilhamento e permissões</title>
    O sistema precisa de um controle de permissões onde usuários podem ser organizados em times, com permissões como leitura, escrita e administração de tarefas dentro de projetos ou categorias.
  </step>

  <step>
    <title>4. Armazenamento e tecnologia</title>
    Uma stack adequada seria: backend em Go com Echo ou Fiber, banco PostgreSQL, autenticação com JWT e middleware para RBAC (role-based access control).
  </step>
</thought>

<answer>
O backend da aplicação Todo List deve ser construído em Go, com suporte a JWT para autenticação, PostgreSQL para persistência e estrutura multiusuário com times e permissões de acesso. As tarefas são entidades colaborativas atribuídas a múltiplos usuários com categorias personalizadas e status gerenciável.
</answer>
```




Chain of Thought (CoT)

Comparativo

Tipo de Prompt	Requer Exemplos	Gera Raciocínio	Ideal para
Zero-Shot	Nenhum	Média	Baixo
One-Shot / Few-Shot	Sim	Opcional	Imitar formato, estilo ou padrões
Chain of Thought	Indiferente	Sim	Lógica, planejamento, análise, debugging



Skeleton of Thought (SoT)

Skeleton of Thought Prompting é uma variação de Chain of Thought onde o modelo é instruído a seguir uma **estrutura lógica pré-definida (um esqueleto)**, com marcadores claros de seções ou ideias centrais que devem ser preenchidas com raciocínio. Essa técnica é ideal quando o desenvolvedor deseja que a IA produza respostas **organizadas, completas e sem fugir da estrutura** desejada.

Estudo

A técnica foi inspirada diretamente em um estudo acadêmico publicado por **Zhang et al. (2023)** no paper *"Skeleton-of-Thought: Large Language Models Can Do Parallel Decoding"*. Nesse trabalho, os autores mostraram que é possível separar a geração do **esqueleto estrutural da resposta** da geração do conteúdo detalhado de cada parte, promovendo paralelismo e coerência. Embora o foco técnico do paper seja a melhoria de desempenho por decodificação paralela, o mesmo princípio é aplicável a prompting estrutural em diversos contextos como documentações, arquitetura de software, etc.

Quando utilizar

- Resumos técnicos com seções fixas
- Documentações, ADRs, Templates
- Quando há necessidade de controlar a saída e formato de uma resposta

Link do paper:



arXiv.org



Skeleton-of-Thought: Prompting LLMs for Efficient Parallel Generation

This work aims at decreasing the end-to-end generation latency of large language models (LLMs). One of the major causes of the high generation latency is the...



Skeleton of Thought (SoT)

Exemplo

Prompt:

Você é um engenheiro de software especializado em performance e escalabilidade. Sua tarefa é propor uma solução para implementar um rate limiter robusto que limite a taxa de requisições por cliente. Estructure sua resposta com os seguintes tópicos:

- Contexto do problema
- Requisitos funcionais e não funcionais
- Estratégia de rate limiting
- Stack tecnológica sugerida
- Considerações de concorrência e escalabilidade
- Conclusão técnica

Resposta esperada:

Contexto do problema: A API será utilizada por centenas de clientes simultaneamente e precisa garantir que nenhum cliente ultrapasse o limite de 1000 requisições por minuto. O sistema deve suportar até 100 mil requisições por segundo em picos e evitar sobrecarga no backend. É necessário garantir que o rate limiter funcione em ambiente distribuído.

Requisitos funcionais e não funcionais:

- Cada cliente deve ter limites de requisição individuais.
- Os limites devem ser reiniciados por janela deslizante (sliding window).
- O sistema deve ser resiliente, com baixa latência (<5ms).
- Suporte a múltiplas instâncias e execução concorrente.

Estratégia de rate limiting: A técnica ideal é o uso de **Token Bucket** com sliding window e sincronização via Redis. Cada cliente terá um bucket virtual com capacidade e taxa de preenchimento ajustável. A cada requisição, será feita uma tentativa de consumir um token. Se o token não estiver disponível, a requisição será rejeitada com HTTP 429.

Stack tecnológica sugerida:

- Linguagem: Go
- Middleware HTTP: Echo ou Fiber
- Rate limiting core: implementação própria ou uso do pacote golang.org/x/time/rate
- Armazenamento distribuído: Redis Cluster para controle de tokens sincronizado entre instâncias
- Monitoramento: Prometheus + Grafana

Considerações de concorrência e escalabilidade:

- Redis deve usar LUA scripts atômicos para evitar race conditions.
- Shard de clientes por chave (hash) pode distribuir melhor a carga.
- Uso de buffers internos no processo local para operações de leitura rápidas em cache.
- Evitar chamadas de rede por requisição usando prefetching de tokens.

Conclusão técnica: O uso de Token Bucket com Redis distribuído e implementação otimizada em Go atende aos requisitos de latência e escalabilidade. A estrutura modular permite reuso em outros serviços. É necessário fazer testes de carga com Locust e simulação de falhas para validar a robustez da solução em produção.



Skeleton of Thought (SoT)

Quando usar SoT vs CoT "puro"

Quando...	Use Skeleton of Thought
Você quer controle total do formato	Sim
A resposta deve ser dividida por tópicos	Sim
A tarefa exige checklist estruturado	Sim
A IA costuma se perder na estrutura	Sim
A lógica for altamente exploratória	Não

Comparativo

Técnica	Requer estrutura?	Raciocina passo a passo?	Ideal para...
Zero-Shot	Não	Não	Consultas diretas, respostas factuais
One-Shot / Few-Shot	Parcial	Opcional	Repetir padrões de exemplo com precisão
Chain of Thought	Não	Sim	Diagnóstico, debugging, raciocínio técnico
Skeleton of Thought	Sim	Opcional	Respostas organizadas, documentações, especificações



Tree of Thought (ToT)

Tree of Thought (ToT) é uma extensão da técnica **Chain of Thought (CoT)** que permite que o modelo explore **múltiplos caminhos de raciocínio paralelos ou alternativos** antes de tomar uma decisão final. Em vez de um raciocínio linear, o ToT incentiva o modelo a **ramificar ideias** e **avaliar alternativas**, como se estivesse construindo uma árvore de decisões.

Estudo


A técnica foi formalizada por **Yao et al., 2023**, no paper *Tree of Thoughts: Deliberate Problem Solving with Large Language Models*, mostrando ganhos significativos em tarefas complexas como jogos de lógica, planejamento e raciocínio analítico estruturado.

Quando utilizar

- A tarefa é ambígua ou tem **múltiplas soluções possíveis**.
- É necessário **comparar estratégias ou caminhos distintos**.
- A decisão final depende de **múltiplos critérios ou trade-offs**.
- Deseja-se simular **raciocínio humano deliberado** com ponderações.

Link do paper:



 arXiv.org

Tree of Thoughts: Deliberate Problem Solving with Large Language Models

Language models are increasingly being deployed for general problem solving across a wide range of tasks, but are still confined to token-level, left-to-right...





Tree of Thought (ToT)

Exemplo

Prompt:

Você é um engenheiro de software especialista em sistemas distribuídos. Um cliente quer limitar o acesso à sua API que recebe até 100 mil requisições por segundo. Gere múltiplas abordagens (caminhos de raciocínio) para resolver esse problema. Para cada abordagem, exponha os prós e contras. Ao final, selecione a mais adequada com base em escalabilidade, confiabilidade e latência.

Resposta esperada:

Caminho 1: Token Bucket com Redis

- Permite bursts controlados.
- Fácil de implementar.
- Redis pode ser um gargalo sob carga intensa.
- Exige replicação e scripts atômicos.

Caminho 2: Leaky Bucket com cache local

- Boa regularização de tráfego.
- Alta previsibilidade de latência.
- Menos flexível para bursts.
- Cache local pode divergir entre instâncias.

Caminho 3: Algoritmo Sliding Window Log

- Precisão de tempo real por requisição.
- Alta granularidade.
- Alto consumo de memória e custo computacional.
- Difícil de escalar sem otimizações.

Avaliação dos caminhos

- **Escalabilidade:** Caminhos 1 e 2 são melhores.
- **Baixa latência:** Caminho 2 se destaca.
- **Facilidade de manutenção:** Caminho 1 é mais consolidado.

Escolha final:

Token Bucket com Redis + buffer local representa o melhor equilíbrio entre simplicidade, performance e tolerância a falhas.



Tree of Thought (ToT)

Técnicas auxiliares

Expansão controlada de caminhos: limite a profundidade e número de alternativas para evitar dispersão.

"Apresente no máximo 3 caminhos distintos para resolver este problema e desenvolva até 2 níveis de subetapas para cada um."

CrITÉrios de decisão explícitos: guie o modelo com parâmetros como "menor custo", "maior confiabilidade", etc.

"Para cada abordagem proposta, avalie com base em custo, latência e escalabilidade. Escolha a melhor com base nesses critérios."

Reavaliação iterativa: o modelo pode revisar suas escolhas se uma nova ramificação se mostrar superior.

"Depois de explorar todas as opções, reavalie as decisões com base nos resultados observados em cada caminho e corrija se necessário."



Tree of Thought (ToT)

Combinação com outras técnicas: ToT + CoT + SoT

Tree of Thought é altamente compatível com outras estratégias de prompting, resultando em maior controle, completude e explicabilidade. Abaixo está um exemplo que combina Tree of Thought com Chain of Thought e Skeleton of Thought.

Prompt combinado (ToT + CoT + SoT):

Você é um engenheiro de software especialista em sistemas distribuídos. Sua tarefa é projetar uma solução de rate limiting para uma API que suporta 100 mil requisições por segundo. Apresente 3 estratégias distintas, usando o seguinte esqueleto para cada uma:

- Visão geral da abordagem
- Etapas detalhadas do raciocínio (pense passo a passo, como um engenheiro resolveria isso em produção)**
- Principais vantagens
- Principais desvantagens
- Quando usar essa abordagem

Ao final, decida qual abordagem representa o melhor equilíbrio para o caso proposto.

Resposta esperada (resumo):

Estratégia 1: Token Bucket com Redis distribuído

- Visão geral:** Permite pequenos bursts, boa tolerância à falha.
- Etapas:** Descrever uso de token bucket, scripts LUA atômicos, cache local.
- Vantagens:** Popular, flexível, boa documentação.
- Desvantagens:** Requer Redis de alta disponibilidade.
- Uso ideal:** Quando se deseja flexibilidade e controle individualizado.

Estratégia 2: Leaky Bucket com armazenamento local

- Visão geral:** Regulariza fluxo de forma constante.
- Etapas:** Descrever deque local, fallback em caso de falha.
- Vantagens:** Mais simples, previsível.
- Desvantagens:** Cache local pode gerar inconsistência.
- Uso ideal:** Quando a estabilidade do fluxo é mais importante que burst handling.

Estratégia 3: Sliding Window Log com shard por API key

- Visão geral:** Rastreia cada requisição.
- Etapas:** Uso de logs temporais por cliente.
- Vantagens:** Alta precisão temporal.
- Desvantagens:** Alto custo de memória.
- Uso ideal:** Quando a justiça de tempo real é indispensável.

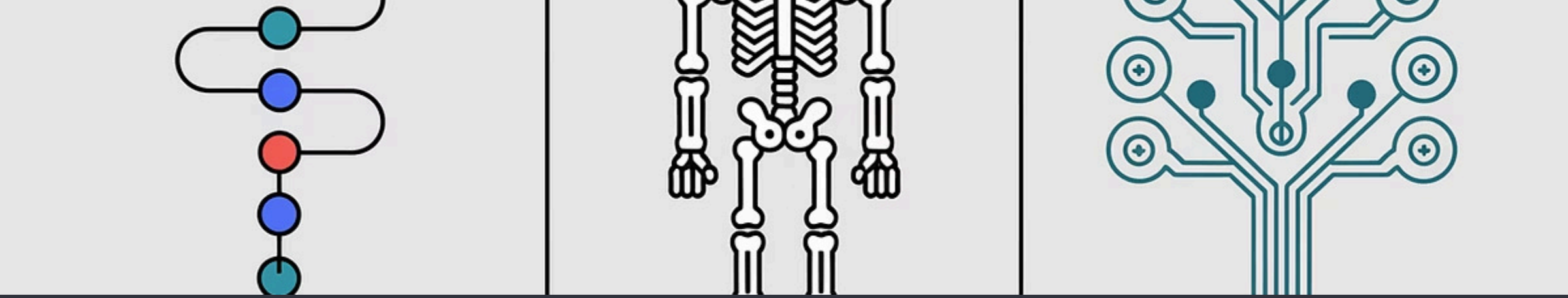
Decisão final: A estratégia 1 oferece o melhor equilíbrio entre performance, controle e robustez para a maioria dos cenários em produção com cargas elevadas.



Tree of Thought (ToT)

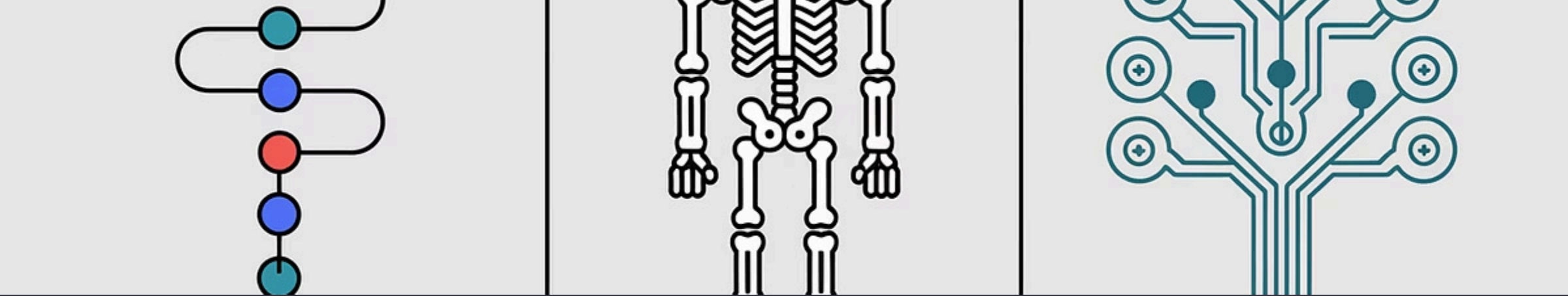
Comparativo com outras técnicas

Técnica	Requer estrutura?	Raciocina passo a passo?	Gera múltiplas alternativas?	Ideal para...
Zero-Shot	Não	Não	Não	Consultas diretas, respostas factuais
Few-Shot	Parcial	Opcional	Não	Repetir padrões de exemplo com precisão
Chain of Thought	Não	Sim	Não	Diagnóstico, debugging, raciocínio técnico
Skeleton of Thought	Sim	Opcional	Não	Respostas organizadas, documentações, especificações
Tree of Thought	Parcial	Sim	Sim	Decisão entre estratégias, brainstorming estruturado



Resumo comparativo: CoT, SoT, ToT

Técnica	Situação ideal	Justificativa	Exemplo de prompt
Chain of Thought (CoT)	Explicar bugs	Raciocínio encadeado com lógica explicada	"Explique passo a passo por que o código abaixo pode gerar um panic em Go. Analise como um engenheiro faria debug em produção."
Skeleton of Thought (SoT)	Especificar módulos com seções fixas	Exige consistência e organização por tópicos	"Crie uma especificação técnica para um módulo de autenticação JWT usando os seguintes tópicos: requisitos funcionais, modelo de dados, fluxos, validações, segurança e integração."
Tree of Thought (ToT)	Comparar opções (ex: cache)	Exploração de alternativas e decisão justificada	"Considere três formas de aplicar cache em um sistema web: in-memory, Redis e CDN. Para cada uma, descreva a estratégia, vantagens, desvantagens e cenário ideal de uso. Ao final, selecione a melhor com base em latência, custo e simplicidade."
SoT + CoT	Planejamento de arquitetura	Organização por tópicos com raciocínio detalhado	"Estruture a arquitetura de um sistema de Todo List com autenticação, API REST e persistência. Responda por tópicos: visão geral, autenticação, banco de dados, fluxos principais, escalabilidade. Em cada tópico, pense passo a passo como um arquiteto de software."
ToT + SoT + CoT	Definir melhor stack tecnológica	Estrutura, múltiplas alternativas e raciocínio interno	"Compare as stacks Go, Node.js e Python para microserviços. Para cada uma, siga os tópicos: performance, ecossistema, produtividade, complexidade de deploy e casos de uso recomendados. Dentro de cada tópico, pense passo a passo. Ao final, recomende a stack ideal para um sistema com 10 microserviços interconectados."
ToT + SoT	Comparação de bancos de dados	Análise comparativa organizada por critérios	"Compare os tipos de banco de dados SQL, NoSQL e NewSQL para uma aplicação de leitura intensiva. Para cada um, responda usando os tópicos: modelo de dados, escalabilidade, latência, consistência e custo operacional. Ao final, indique qual abordagem é mais indicada para esse cenário."



Resumo comparativo: CoT, SoT, ToT

Casos de uso abordados

Técnica aplicada	Situação	Justificativa
Chain of Thought (CoT)	Explicar por que um bug ocorre	Precisa de raciocínio encadeado com lógica explicada
Skeleton of Thought (SoT)	Especificar um módulo de autenticação com seções fixas	Exige consistência e organização por tópicos
Tree of Thought (ToT)	Comparar 3 formas de aplicar cache (in-memory, Redis, CDN)	Exige exploração de alternativas e decisão final justificada
SoT + CoT	Planejar arquitetura de um sistema com API, banco e autenticação	Exige estrutura e raciocínio técnico dentro de cada seção
ToT + SoT + CoT	Definir melhor stack entre Go, Node.js e Python para microserviços	Requer estrutura, múltiplas alternativas e raciocínio interno completo
ToT + SoT	Comparar bancos SQL, NoSQL e NewSQL para leitura intensiva	Múltiplas estratégias com análise técnica estruturada, sem exigir CoT

Conclusão

Cada técnica possui forças complementares:

- **CoT** → Raciocínio lógico.
- **SoT** → Organização e completude.
- **ToT** → Comparação e tomada de decisão.

Self-Consistency

Como funciona a técnica

1. O prompt induz o modelo a pensar passo a passo (Chain of Thought).
2. A tarefa é executada **diversas vezes** (tipicamente de 5 a 10).
3. As respostas geradas são então **coletadas e comparadas**.
4. A saída final é definida por **votação majoritária** ou por **métrica de consistência**.

O princípio é simples: o modelo pode cometer erros em uma cadeia específica, mas, com múltiplas execuções, as respostas **mais confiáveis tendem a convergir**.

Quando utilizar

- Há ambiguidade matemática ou estrutural.
- A tarefa é suscetível a variações de raciocínio.
- O modelo tende a dar boas respostas às vezes, mas não sempre.
- Você precisa aumentar a confiabilidade da saída com pouco custo computacional adicional.

Por que funciona

LLMs operam com amostragem probabilística (ex: temperatura > 0), o que os torna suscetíveis a gerar variações, desvios ou respostas inconsistentes. Ao gerar múltiplas execuções:

- Reduzimos alucinações isoladas.
- Aumentamos a chance de obter uma resposta estatisticamente sólida.
- Priorizamos coerência entre caminhos lógicos distintos.

Link do paper:



arXiv.org

Self-Consistency Improves Chain of Thought Reasoning in Language Mo...

Chain-of-thought prompting combined with pre-trained large language models has achieved encouraging results on complex reasoning tasks. In this paper, we propose...

Self-Consistency

Situação

Você está desenvolvendo a estimativa de custo mensal para uma aplicação em produção na AWS. A aplicação utiliza:

- 10 instâncias EC2 t3.large (região US East)
- 1 TB de armazenamento EBS
- 1 Load Balancer
- 100 GB de transferência de dados saindo por mês

Como pequenos desvios podem ocorrer entre execuções, você decide aplicar Self-Consistency para gerar múltiplas estimativas e selecionar a mais confiável.

Prompt (com CoT):

“Calcule o custo total mensal dessa infraestrutura. Pense passo a passo.”

Execuções (resumidas):

Execução 1

- EC2: $10 \times \$0.0832 \times 730h = \607.36
- EBS: $1024 \text{ GB} \times \$0.10 = \102.40
- ELB: \$18.00
- Data transfer: $100 \text{ GB} \times \$0.09 = \9.00
- **Total:** \$736.76

Execução 2

- EC2: \$605.00 (aproximado)
- EBS: \$100.00
- ELB: \$18
- Transferência: \$9
- **Total:** \$732.00

Execução 3

- EC2: $10 \times 0.0832 \times 730 = \607.36
- EBS: $1TB \times 0.10 = \$102.40$
- Load Balancer: \$18
- Data Out: \$9
- **Total:** \$736.76

Resultado da Self-Consistency:

A estimativa de **\$736.76** foi repetida em duas execuções com cálculos detalhados e consistentes. Ela é considerada a resposta mais confiável por frequência e exatidão.

Resposta selecionada: \$736.76 — confirmada como mais precisa e recorrente entre as múltiplas cadeias de raciocínio.



Self-Consistency

Aplicações práticas em engenharia de software

- Estimativas de custo e capacidade (cloud, infra, storage)
- Planejamento de sizing de ambientes
- Validação de resultados numéricos ou previsões algorítmicas
- Verificação de hipóteses técnicas sob múltiplos critérios
- Comparações de lógica interna em testes de arquitetura

Dicas de aplicação

- Gere 5 a 10 respostas com **temperatura > 0.5** para estimular caminhos diversos.
- Normalize o formato da saída antes de comparar.
- Pode ser usada manualmente (humano escolhe) ou automaticamente (via votação).

Exemplo completo usando todas as dicas de aplicação

Você quer estimar o número ideal de shards para uma base de dados multitenant com 80.000 clientes.

Prompt:

"Qual o número ideal de shards para particionar uma base de dados com 80.000 clientes, considerando escalabilidade, performance e isolamento? Pense passo a passo."

Aplicação prática das dicas:

- **Temperatura variada (diversidade de raciocínio):** Você gera 8 respostas com temperaturas variando entre 0.6 e 0.8 para explorar caminhos diferentes.
- **Normalização da saída:** Antes de comparar, remove diferenças como:
 - "~10 shards", "10", "dez" → tudo é convertido para "10"
 - Formatação monetária ou numérica (R\$10k = 10000)
- **Seleção da resposta final:**
 - 5 das 8 execuções sugerem 10 shards, com raciocínios como "8000 clientes por shard", "balanceamento operacional", "flexibilidade para crescimento futuro".
 - Você implementa um script simples que conta a frequência e seleciona a mais recorrente.

Resultado final: 10 shards — justificado por frequência, coerência técnica e compatibilidade com os critérios operacionais do sistema.



Directional Stimulus / Directed Prompting

Directed Prompting (ou Directional Stimulus Prompting) é uma técnica que guia a resposta do modelo ao utilizar verbetes, comandos ou estímulos direcionais. Diferente de prompts abertos ou vagos, essa abordagem indica como o modelo deve pensar ou responder, influenciando estilo, formato, foco ou tipo de raciocínio esperado.

Quando utilizar

- Quando é necessário obter **respostas previsíveis ou formatadas**.
- Quando deseja-se guiar o raciocínio por um **estilo de resposta desejado**.
- Quando a resposta precisa estar em um **formato parseável** (como JSON, XML, YAML).
- Quando a tarefa exige foco explícito (ex: "responda como um arquiteto", "explique em tópicos").

Benefícios de Directed Prompting

- Reduz ambiguidade e alucinação.
- Padroniza a forma da resposta.
- Melhora a utilidade programática da saída (ex: para scripts ou interfaces).
- Permite controle educado do comportamento do modelo, sem necessidade de complexidade adicional.

Link do paper:



 arXiv.org

Scaling Instruction-Finetuned Language Models

Finetuning language models on a collection of datasets phrased as instructions has been shown to improve model performance and generalization to unseen tasks. In...





Directional Stimulus

Exemplos de comandos direcionais comuns

- "Liste..."
- "Compare..."
- "Explique com exemplos..."
- "Responda em formato JSON"
- "Justifique cada item"
- "Explique como um arquiteto sênior faria"
- "Divida em tópicos: contexto, estratégia, riscos e conclusão"

Esses comandos **atuam como estímulos condicionais**, que alteram tanto **o formato quanto a profundidade da resposta**.

Aplicações em engenharia de software

- Gerar documentação técnica com estrutura específica
- Listar vantagens/desvantagens de soluções de arquitetura
- Descrever fluxos de autenticação, cache, filas, escalabilidade
- Padronizar output para consumo por sistemas (JSON, YAML)



Directional Stimulus

Exemplo: Comparação de tecnologias com Directed Prompting / Direction Stimulus

Prompt:

Compare as tecnologias Kafka, RabbitMQ e Amazon SQS para uso em sistemas distribuídos. Responda nos tópicos: características principais, throughput, confiabilidade, complexidade operacional e quando usar cada uma.

Resposta esperada:

Kafka

- Características: distribuído, baseado em logs, alta durabilidade.
- Throughput: muito alto, ideal para big data e pipelines de evento.
- Confiabilidade: altíssima, com replicação e retenção configurável.
- Complexidade: alta — exige setup de brokers, Zookeeper (ou KRaft), monitoração ativa.
- Quando usar: quando o volume de dados é alto e a ordem das mensagens é crítica.

RabbitMQ

- Características: orientado a fila, com suporte a múltiplos protocolos (AMQP, MQTT).
- Throughput: médio-alto, mais rápido para mensagens pequenas.
- Confiabilidade: boa, com suporte a ACK, DLQ e persistência.
- Complexidade: moderada, bom tooling e fácil operação.
- Quando usar: para mensageria tradicional, RPC e integração entre serviços.

Amazon SQS

- Características: gerenciado, escalável automaticamente, simples de operar.
- Throughput: escalável sob demanda.
- Confiabilidade: excelente, com redundância e reentrega automática.
- Complexidade: baixa — sem servidores para administrar.
- Quando usar: quando se busca simplicidade e escalabilidade sem overhead de operação.



Directional Stimulus vs SoT

Aspecto	Skeleton of Thought	Directed Prompting
Tipo de controle	Estrutura rígida com seções definidas	Estímulo leve baseado em instruções textuais
Modelo deve seguir tópicos?	Sim	Não necessariamente
Liberdade de formatação	Baixa	Média
Uso típico	Documentações, especificações, planejamento	Comparações, respostas formatadas, controle de estilo
Exemplo de prompt	SoT: "Preencha os tópicos: visão geral, arquitetura, riscos, conclusão."Directed: "Liste as vantagens e desvantagens da arquitetura em tópicos."	"Liste as vantagens e desvantagens da arquitetura em tópicos."



ReAct (Reasoning + Action)

ReAct (Reasoning + Acting) é uma técnica de prompting que permite que LLMs combinem **raciocínio passo a passo (Chain of Thought)** com a **execução de ações externas** — como chamadas a ferramentas, busca em bancos de dados, execução de código ou uso de APIs.

Estudo

Foi formalizada no paper "**ReAct: Synergizing Reasoning and Acting in Language Models**" por Yao et al., 2022. A ideia é fazer com que o modelo pense de forma deliberada e execute ações iterativas com base nesse raciocínio.

Quando utilizar

- Quando o modelo precisa consultar APIs, sistemas ou bancos de dados.
- Quando o raciocínio sozinho **não é suficiente** para responder.
- Quando há dependência de **informações externas dinâmicas**.

Vantagens e Limitações do ReAct Prompting


Vantagens:


- Permite que o modelo **combine raciocínio com ação** em tempo real.
- Ideal para **construção de agentes interativos** (SRE, DevOps, suporte, CI/CD).
- Gera **respostas auditáveis**, pois explicita cada passo (Thought → Action → Observation).
- Pode ser iterado com base em **feedbacks dinâmicos**, como logs ou APIs externas.
- Compatível com ferramentas e ambientes já operacionais (ex: Docker, scripts, observabilidade).

Limitações:

- Exige que a ação esteja simulada ou disponível via ferramenta externa.
- Requer sistemas que integrem e interpretem a Action e Observation corretamente.
- A estrutura Thought/Action/Observation deve ser seguida à risca, o que pode limitar a fluidez da linguagem.
- Aumenta a complexidade de construção e teste do prompt.


Link do paper:



 arXiv.org

ReAct: Synergizing Reasoning and Acting in Language Models

While large language models (LLMs) have demonstrated impressive capabilities across tasks in language understanding and interactive decision making, their...





ReAct

Exemplo técnico: Inspeção de containers com ReAct e ferramenta Docker

Prompt

Você é um engenheiro de plataforma. Utilize raciocínio passo a passo e a ferramenta Docker (disponível via MCP) para inspecionar os containers que estão rodando no sistema. Seu objetivo é entender o estado atual da infraestrutura. Para cada etapa, siga o padrão:

- Thought: ...
- Action: ...
- Observation: ...
- Final Answer: ...

Resposta esperada

Thought: Para entender o estado atual da infraestrutura, o primeiro passo é listar os containers que estão em execução.
Action: usar ferramenta Docker com comando "docker ps"
Observation: 12 containers ativos, incluindo nginx, postgres, redis, app-core, worker-1, worker-2, api-gateway, etc.

Thought: Precisamos identificar quais containers estão usando mais CPU.
Action: usar ferramenta Docker com comando "docker stats"
Observation: O container worker-2 está consumindo 78% de CPU, bem acima dos demais.

Thought: Isso pode indicar um ciclo de processamento contínuo. Verificar logs desse container.
Action: usar ferramenta Docker com comando "docker logs worker-2 --tail=50"
Observation: Logs mostram tarefa de processamento de mensagens em repetição com erro de parsing.

Final Answer: O container worker-2 está sobrecarregado processando mensagens com erro contínuo. Deve ser investigada a origem dos dados e aplicada correção no pipeline de parsing.

Thought: Se a aplicação está inacessível, o container pode ter parado ou falhado. Vamos consultar o MCP para verificar o estado do container payments-api.
Action: GET /mcp/containers?filter=name:payments-api
Observation: Container payments-api está em status "Exited" com código 137 (Out of Memory).

Thought: A falha indica que o container excedeu o uso de memória. Verificar histórico de uso de recursos.
Action: GET /mcp/metrics/payments-api?range=4h
Observation: Pico de uso de memória às 09:17, 09:32, 09:50 — ultrapassando o limite de 512MiB.

Thought: A aplicação precisa de mais memória. Vamos ajustar o limite e reiniciar.
Action: PATCH /mcp/containers/payments-api { "memory_limit": "1GiB" }
Observation: Novo limite definido com sucesso.

Thought: Agora vamos reiniciar o container.
Action: POST /mcp/containers/payments-api/restart
Observation: Container iniciado com sucesso. Status atual: "Running".

Final Answer: O container payments-api falhou por falta de memória. O limite foi ajustado para 1GiB e a aplicação voltou ao ar com sucesso.
Thought: O erro 503 geralmente indica indisponibilidade do backend. Vamos verificar se há ocorrências no log do serviço de checkout.
Action: GET /logs?service=checkout&since=2h
Observation: 503 errors found at 14:03, 14:05, 14:09. Spike correlates with deploy.

Thought: Pode estar relacionado ao último deploy. Verificar histórico de deploys.
Action: GET /deploys?service=checkout
Observation: Último deploy às 14:00.

Thought: Erros iniciaram imediatamente após o deploy. Validar rollback.
Action: POST /rollback?service=checkout
Observation: Rollback initiated. Traffic stabilizing.

Final Answer: O erro 503 foi causado por uma falha no último deploy às 14:00. Rollback aplicado com sucesso.

Aplicações práticas em engenharia de software

- Depuração com logs ou trace APIs



ReAct

Diferença entre ReAct e CoT puro

ReAct	Chain of Thought (CoT)
Interage com o ambiente externo	Raciocina apenas internamente
Usa ferramentas	Não executa ações
Baseado em observações iterativas	Baseado em raciocínio estático
Ideal para agentes ou assistentes	Ideal para explicações e lógica