03

AI-SPECIAL-OPS / FREE GUIDE

# PROMPT ENGINEERING FOR AI AGENTS

The skill that makes everything else work. Standard operating procedures for every pattern.

AUTHOR
Marian Esco

TECHNIQUES
8 Core + 3 Advanced

CONTENTS

# WHAT'S INSIDE

9 techniques. Each one with SOPs and worked examples.

Prompt engineering is the single highest-leverage skill in AI agent development. A bad prompt with the best model produces bad output. A good prompt with a decent model produces great output. This guide gives you every technique, with SOPs you can use today.


```
01  The Foundation
    Why prompt engineering beats fine-tuning

02  Be Explicit
    The #1 rule. Ambiguity kills agents.

03  System Prompts
    Agent identity. The DNA of behavior.

04  Few-Shot Examples
    Show, don't tell. 2-3 examples change everything.

05  Chain-of-Thought
    Make it reason step by step.

06  ReAct Pattern
    Reasoning + Acting. The agent loop.

07  Role-Based Prompting
    Narrow scope. Better output.

08  Output Parsing
    Structured responses for automation.

09  Context Engineering
    What enters the model's attention.
```

CHAPTER 01

# THE FOUNDATION

Why prompt engineering is the skill that makes everything else work.

```
Bad prompt  + Best model  + Best framework  = BAD agent
Good prompt + Decent model + Simple script    = GOOD agent

// The prompt is the DNA. Everything downstream depends on it.
```

Prompt engineering is not "asking nicely." It is the engineering discipline of designing the instructions that control how an AI model behaves, reasons, and produces output. For agents, the system prompt IS the agent's DNA. Everything downstream depends on it.

Anthropic's own research confirms: prompt engineering is more efficient than fine-tuning. Faster iteration, lower cost, preserves general knowledge, human-readable, works across model updates without retraining.

## THE 8 CORE TECHNIQUES

```
01  Be Explicit          05  ReAct Pattern
02  System Prompts       06  Role-Based Prompting
03  Few-Shot Examples    07  Output Parsing
04  Chain-of-Thought     08  Context Engineering
```

TECHNIQUE 01

# BE EXPLICIT

The #1 rule. The LLM only knows what you tell it.

Ambiguity equals inconsistency. The model cannot read your mind. If you want specific output, you must specify every dimension: format, structure, length, constraints, and success criteria.

## BAD VS GOOD

```
// BAD:
Analyze this data.

// GOOD:
You are a data analyst. Given the CSV below:
1. Count rows with missing values in "email"
2. Calculate the average of "revenue"
3. Flag values more than 2 std devs from mean
4. Output as JSON only. No explanation.
```

SOP

1. What exactly should the output look like? (format, structure, length)
2. What should the model NOT do? (constraints, forbidden behaviors)
3. What context does it need? (background, data, prior decisions)
4. How will I know if the output is correct? (success criteria)

TECHNIQUE 02

# SYSTEM PROMPTS

Agent identity. The DNA of behavior.

The system prompt defines WHO the agent is and HOW it behaves. It runs before every user message. This is where you encode the agent's personality, expertise, constraints, and output rules.

## TEMPLATE

```
[IDENTITY]
You are a [role] with expertise in [domain].
Your primary goal is [goal].

[CONSTRAINTS]
* You NEVER [forbidden action]
* You ALWAYS [required behavior]
* If unsure, you [fallback behavior]

[OUTPUT FORMAT]
Your responses follow this structure:
[define exact structure]

[EXAMPLES]
Input: [example]  Output: [example]
```

## WORKED EXAMPLE

```
You are a lead qualification agent.
Your job:
* Classify leads by ICP match
* Score them 1-10 on fit
* Recommend follow-up action
Constraints:
* You ONLY classify. Never draft outreach.
* Score below 3 = ARCHIVE
* Score 8+ = FIRE for immediate human review
* All output in JSON format.
```

SOP

1. Start with identity (one sentence: who is this agent?)
2. Define scope (what it CAN and CANNOT do)
3. Set constraints (hard boundaries that never break)
4. Specify output format (exact structure)
5. Give 1-2 examples (show don't tell)
6. Test with edge cases (what happens with weird input?)
7. Iterate (adjust constraints based on failures)

TECHNIQUE 03

# FEW-SHOT EXAMPLES

Show, don't tell. 2-3 examples change everything.

Instead of explaining what you want, show 2-3 examples of input to desired output. The model learns the pattern from examples far more reliably than from instructions alone.

```
Here are examples of processing customer feedback:

Example 1:
Input: "Product great but shipping took forever"
Output: {"sentiment":"mixed","action":"flag_shipping"}

Example 2:
Input: "Terrible experience. Nothing works."
Output: {"sentiment":"negative","action":"escalate"}

Example 3:
Input: "Love it! Recommended to friends."
Output: {"sentiment":"positive","action":"request_review"}

Now process this feedback:
Input: "[actual feedback]"
```

SOP

1. Use 2-3 examples (more than 3 rarely helps, wastes tokens)
2. Cover the range (positive, negative, edge case)
3. Keep examples consistent (same format, same detail level)
4. Put examples BEFORE the actual task (model reads top to bottom)
5. Use clear delimiters (Example 1: / Input: / Output:)

TECHNIQUE 04

# CHAIN-OF-THOUGHT

Make it reason step by step. Dramatically improves accuracy.

Ask the model to think step-by-step before giving its final answer. This dramatically improves accuracy on complex tasks. If the task has more than one logical step, use CoT.

## STRUCTURED COT TEMPLATE

```
Analyze this contract clause for risk:

Step 1: Identify clause type
Step 2: List obligations for each party
Step 3: Identify ambiguous language
Step 4: Assess risk level (low/med/high/critical)
Step 5: Final assessment

Show your reasoning for each step.
```

## WHEN TO USE COT

- Math or calculations (always)
- Multi-step reasoning (always)
- Classification with nuance (when edge cases exist)
- Decision-making (when you need to audit the reasoning)

## WHEN NOT TO USE COT

- Simple factual questions
- Format conversion (JSON to CSV)
- Tasks where speed matters more than accuracy

TECHNIQUE 05

# REACT PATTERN

Reasoning + Acting. The standard agent loop.

The standard pattern for agent loops. The model alternates between thinking and using tools. Thought leads to Action, Action produces Observation, repeat until Final Answer.

```
Thought: Need customer balance for credit risk.
Action: get_balance(customer_id="CUST-123")
Observation: {"balance": 5000, "currency": "USD"}

Thought: Good balance. Need payment history.
Action: get_payment_history(id="CUST-123", months=12)
Observation: {"payments": 12, "late": 0}

Thought: Perfect record. Can assess now.
Final Answer: Credit risk LOW. 5K USD balance,
12/12 on-time. Recommend approval.
```

SOP

1. Define available tools clearly in the system prompt
2. Set max iterations (10-20 is typical)
3. Parse Thought/Action/Observation blocks from output
4. Execute actions through your tool layer (with governance)
5. Inject Observation back into the conversation
6. Loop until Final Answer or max iterations

TECHNIQUE 06

# ROLE-BASED PROMPTING

Narrow scope for better output. Prevent scope creep.

Define the agent's role narrowly to prevent scope creep. The more specific the role, the better the output. A focused agent with boundaries outperforms a general-purpose assistant every time.

## PROGRESSION: BAD TO GOOD

```
// BAD:
"You are a helpful assistant."
  > Does everything poorly. No boundaries.

// OK:
"You are a financial analyst."
  > Better focus, but still too broad.

// GOOD:
"You are a credit risk analyst for B2B SaaS.
 You assess creditworthiness based on payment
 history, financials, and industry benchmarks.
 You NEVER make lending decisions.
 You provide risk assessments for human review."
```

## EVERY AGENT GETS A ROLE STATEMENT

```
WHO are you? (one-line identity)
WHAT do you do? (2-3 specific capabilities)
WHAT don't you do? (hard boundaries)
WHO reviews your work? (human in the loop)
```

---

TECHNIQUE 07

# OUTPUT PARSING

Structured responses. Free-form text breaks automation.

Agents need to produce parseable output. Free-form text breaks automation. Define the exact output structure, give an example, and validate in your code.

## JSON OUTPUT (MOST COMMON)

```
Always respond in valid JSON. No text outside JSON.
Output format:
{
  "analysis": "string (1-3 sentences)",
  "score": "integer (1-10)",
  "action": "APPROVE | REVIEW | REJECT",
  "confidence": "float (0.0-1.0)"
}
```

## XML TAGS (LABELED SECTIONS)

```
<thinking>
Your internal reasoning (not shown to user)
</thinking>
<answer>
Your final answer (shown to user)
</answer>
<tools_needed>
List any tools you need to call
</tools_needed>
```

SOP

1. Define the exact output structure in the system prompt
2. Give an example of correct output
3. Say "No text before or after the [format]"
4. Validate output in your code (if JSON, try to parse it)
5. If parsing fails, retry with error message

---

TECHNIQUE 08

# CONTEXT ENGINEERING

What enters the model's attention changes everything.

Context engineering is about optimizing WHAT information enters the model's attention. Not just how you ask, but what you include. More context does not equal better answers. The model's attention is a budget. Every irrelevant token dilutes focus.

## CONTEXT BUDGET REALITY

```
Context window    = 200K tokens (Claude)
System prompt     =   2K tokens
Conversation      =   5K tokens
Available space   = 193K tokens

// BUT: More context != Better answers
// Attention is a BUDGET. Irrelevant tokens dilute.
```

## INCLUDE VS EXCLUDE

```
INCLUDE:
+ System prompt (always)
+ Current task description (always)
+ Relevant memory (retrieved, not dumped)
+ Tool definitions (only available tools)
+ Recent conversation (last 5-10 turns)
+ Examples (2-3 max)

EXCLUDE:
* Full conversation history (summarize)
* All available tools (only relevant ones)
* Raw data dumps (pre-process first)
* Redundant instructions
* "Just in case" context
```

QUICK REFERENCE

# WHICH TECHNIQUE, WHEN

```
SITUATION                      TECHNIQUE            PRIORITY
-----------------------------------------------------------------
Agent won't follow format      Output structure      P1
Agent gives wrong answers      Chain-of-thought      P1
Agent calls wrong tools        Tool descriptions     P1
Agent scope creeps             Role + constraints    P1
Agent output inconsistent      Few-shot examples     P2
Agent uses too many tokens     Context engineering   P2
Agent handles edge cases       Edge case examples    P2
Agent is slow                  Simplify, smaller     P3
Building many similar agents   Meta-prompting        P3
```

## BEFORE WRITING ANY PROMPT, ANSWER THESE 4 QUESTIONS:

```
1  What exactly should the output look like?
2  What should the model NOT do?
3  What context does it need?
4  How will I know if the output is correct?
```

*Sources: Anthropic Prompt Engineering Guide, Anthropic "Building Effective Agents," Claude Tool Use docs, PromptingGuide.ai*

# AI-SPECIAL-OPS

Free guide

Questions or feedback: AImarian@proton.me