

The Complete GPT-5

App Development & Prompt Engineering Cookbook

What's New in ChatGPT 5? Master AI with Python

Mammoth Club Official Guide PRO+

- ✓ FREE Online Course
- ✓ FREE Cheatsheet
- ✓ FREE Exam
- ✓ FREE Official Mammoth Club Certificate



MAMMOTH CLUB



Written by Alex Kropf • Edited by John Bura

Cover Design by Jared Matson & John Bura • Contributions by James Dabalus

Powered by  CoursePro.ai

*From the creators of the best-selling Hello Coding: Anyone Can
Learn to Code & more*

PRAISE FOR MAMMOTH CLUB

I have completed many tutorials. This one is the most outstanding one that I have seen thus far. It is doubtful that it could be topped. This is a superior tutorial. Amazing. —Joseph A., Mammoth Club Student

Exactly what I wanted! Just enough BASIC information without being technically overwhelming and intimidating. —Paul V., Mammoth Club Student

This course so far is by far amazing!

The instructor is very encouraging and upbeat, and his instructions are very clear. It's an amazing course. —Moiz S., Mammoth Club Student

It's scary to think that by following these instructional videos I can be equipped with the skills to program Python. —Charles E., Mammoth Club Student

I ended up taking it and it was INCREDIBLE.

They set great challenges that build off what was taught in the lecture, but don't directly give you the answer.

It asks you to extend your knowledge and refer to the right documentation.

So good for learning. —A_Unicycle, Mammoth Club Student

This is AMAZING!

I just learned how to code without breaking a sweat, this is really easy and fun! —Shalonda L., Mammoth Club Student

Clear instructions and excellent projects. —Ian F., Mammoth Club Student



MAMMOTH CLUB



GO TO MAMMOTHCLUB.COM

to access 3,000+ online courses and 5,000+ hours of video content!

Mammoth Club is a leading online course provider in everything from learning to code to becoming a YouTube star. Since 2011, Mammoth Club has built a global student community with over 9 million courses sold.



Scan the QR code to redeem your free course, exam and cheat sheet! Or go to this link:

mammothclub.com/course/1-hour-gpt-5/CHAT

Mammoth Club books can be purchased at a special discount when ordered in bulk for promotional giveaways, fundraisers, or educational initiatives. Customized editions or selected excerpts can also be produced to meet specific needs. For more information, please reach out to support@mammothinteractive.com.

Portions of this book may be shared promotionally if with direct citation to MammothClub.com. This book may not be reproduced — mechanically, electronically, or by any other means, including photocopying — without written permission of the publisher.



The publisher does not provide medical, legal, accounting, or other professional services. Readers seeking such expertise should consult a qualified professional. This book is not meant to be used for clinical procedures or medical treatment. To the maximum extent permitted by law, the publisher and editors are not responsible for any harm or damage to individuals or property resulting from the use or misuse of the material presented herein. All rights reserved. This book does not constitute financial, investment, legal, or tax advice. You are solely responsible for your financial decisions. We make no guarantees of income, business outcomes, or investment returns. By using this book, you agree that the author and publisher cannot be held liable for any loss, damage, or results arising from actions you take based on its content.

Written by Alex Kropf • Proofread and Edited by John Bura • Online Course, Exam and Cheatsheet by James Dabalus • Cover Design by Jared Matson and John Bura • Copyright © 2025 by Mammoth Club

GO TO MAMMOTHCLUB.COM	3
PART 1: WHAT'S NEW IN GPT-5? BREAKTHROUGH FEATURES	9
Fundamentals of GPT-5	9
Is GPT-5 is WORSE than GPT-4?	14
How to Access GPT-5	15
Hello GPT-5: Build Your First API with Zero-Shot Prompting	17
Control Verbosity in GPT-5	19
Use Minimal Reasoning in GPT-5	21
Reuse Prompts with GPT-5 Templates	27
Build Custom Tools with GPT-5	29
Freeform Function Calling in GPT-5	34
Context-Free Grammar (CFG) in GPT-5	39
PART 2: PROMPT ENGINEERING FOR GPT-5	48
Control Eagerness of GPT-5	48
Control Instructions and Roles in GPT-5	51
Fine-Tuning GPT-5	53
Zero-Shot Prompting	54
Few-Shot Prompting in GPT-5	65
Prompt Design Techniques	73
Control Structured Outputs	79
PART 3: SOFTWARE DEVELOPMENT PROMPTING FOR GPT-5	84
Write Prompts for GPT-5 Vibe Coding	84
User Interface Prompting with GPT-5	88
PART 4: BUILD AUTONOMOUS AGENTS	100
Prompt Engineering for GPT-5 Agents	102
Build AI Agents with OpenAI Agents SDK	105
Epilogue: Congratulations	107
WHERE TO GO FROM HERE	111
Get the FREE Online Course & Certificate	111
VISIT MAMMOTHCLUB.COM	113

WELCOME: WHAT YOU'LL LEARN

GPT-5 isn't just a new release—it represents a shift in how we think about, interact with, and build on top of large language models. With significantly expanded reasoning capabilities, improved efficiency, and powerful new tools for developers, GPT-5 has moved beyond simply answering questions. It can now adapt, integrate, and collaborate at a level that demands a new approach from its users.

This book is designed to be your complete guide to mastering that new approach. We'll start by walking through **what's truly different about GPT-5**—from its ability to control verbosity and reasoning depth, to its advanced function calling and context-free grammar support. You'll see side-by-side comparisons so you can spot the differences from GPT-4 and understand when to use each capability.

From there, you'll learn:

- **Fundamental concepts:** How GPT-5 processes instructions, how it differs from earlier versions, and how to set it up for optimal use.
- **Prompt engineering strategies:** Zero-shot, few-shot, and structured prompting techniques that extract consistent, high-quality outputs.
- **Development workflows:** How to build user interfaces, custom tools, and AI agents that take full advantage of GPT-5's API.
- **Advanced controls:** Methods for tuning eagerness, verbosity, and role instructions so GPT-5 produces exactly the level of detail you need.
- **Automation and integration:** How to connect GPT-5 with other systems, reuse prompts as templates, and chain multiple steps into complex workflows.

This book is built around real examples and practical techniques. No theory for theory's sake. Everything you learn, you can use immediately.

Part 1 shows you GPT-5's coolest features with simple examples you can try right away.

Part 2 teaches you how to write prompts that get better results than 99% of people using AI.

Part 3 focuses on using GPT-5 for coding and building software.

Part 4 shows you how to build AI agents that work automatically.

Who This Book Is For

You don't need to be a programmer to use this book. You don't need a PhD in computer science. You just need curiosity and a willingness to try new things.

If you're a developer, you'll discover techniques that will make you incredibly productive.

If you're a business person, you'll learn how to use AI to solve problems and create value.

If you're just curious about AI, you'll understand what's possible and how to make it happen.

You won't just be reading theory. Every chapter includes practical, reproducible examples you can run and adapt immediately—covering both individual features and larger, project-based use cases.

By the end of Part 1, you'll know **exactly what GPT-5 can do and how to access its best features**.

By the end of Part 2, you'll be able to **design prompts that get reliable, tailored results every time**.

By the end of Part 3, you'll be **applying GPT-5 to real software projects**.

In Part 4, you'll be **building autonomous agents** that operate with minimal supervision.

This is more than a “what’s new” guide—it’s a toolkit for making GPT-5 a core part of your work.

GPT-5 represents a fundamental departure from previous language models, introducing architectural changes that alter how we approach AI integration and prompt engineering. This shift demands new methodologies and a deeper understanding of the model's capabilities.

The Technical Landscape

Previous iterations of GPT models followed predictable scaling patterns—more parameters, more data, incremental improvements. GPT-5 breaks this pattern through architectural innovations that affect reasoning, context management, and output control.

Key differentiators include:

Enhanced context retention across extended conversations through improved attention mechanisms and memory management systems.

Structured output generation via context-free grammar support, enabling deterministic formatting without sacrificing semantic quality.

Advanced function calling that extends beyond simple API interactions to complex, multi-step reasoning chains.

Controllable reasoning depth allowing developers to trade computational resources for output quality based on task requirements.

What This Book Covers

This book provides practical frameworks for leveraging GPT-5's advanced capabilities in production environments. Each section builds technical depth while maintaining focus on real-world applications.

Part 1 examines GPT-5's core innovations with detailed implementation examples, comparing performance characteristics with previous models and establishing baseline proficiency.

Part 2 develops systematic prompt engineering methodologies, moving beyond trial-and-error approaches to create reproducible, scalable prompting strategies.

Part 3 focuses on software development applications, covering code generation, architectural planning, and automated testing workflows.

Part 4 addresses autonomous agent construction, including decision-making frameworks, error handling, and reliability patterns.

Prerequisites and Approach

This material assumes familiarity with API-based AI interactions and basic programming concepts. Examples use multiple programming languages, primarily Python, and prioritize conceptual understanding over syntax mastery.

The approach emphasizes systematic experimentation over theoretical speculation. Each technique includes measurable outcomes and failure modes, helping you develop intuition for when and how to apply specific methods.

Technical Philosophy

Effective AI integration requires understanding models as sophisticated tools rather than magic solutions. GPT-5's capabilities enable new approaches to problem-solving, but success depends on matching techniques to appropriate use cases.

This book focuses on building reliable systems that leverage AI capabilities while maintaining predictable behavior and acceptable error rates. The goal is professional competence, not impressive demonstrations.

Ready to Start?

The AI revolution is happening right now. The people who master these tools early will have huge advantages over those who wait.

You're not just learning about GPT-5. You're preparing for a future where AI amplifies human creativity and intelligence in ways we're only beginning to understand. Let's get started!

PART 1: WHAT'S NEW IN GPT-5?

BREAKTHROUGH FEATURES

Fundamentals of GPT-5

GPT is more intelligent and versatile than ever—delivering deeper, more accurate insights across disciplines like math, science, finance, and law. It's like having a panel of specialists ready to answer your every question. The GPT-5 series includes three models, each designed with different trade-offs in mind. While gpt-5 excels at complex, knowledge-intensive tasks, the smaller mini and nano variants offer lower latency and reduced cost, making them ideal for more narrowly defined use cases.

Variant	Best Suited For
gpt-5	Advanced reasoning, deep world knowledge, and complex tasks involving code or multi-step workflows
gpt-5-mini	A cost-efficient choice that balances reasoning ability, responsiveness, and performance in chat or lightweight logic
gpt-5-nano	Ideal for high-volume, simple tasks like classification or straightforward instruction-following

GPT-5 is a reasoning model

GPT-5 approaches problems by breaking them into smaller steps, generating an internal chain of thought that represents its reasoning process. To get the best results, feed these reasoning items back to the model—this prevents it from re-thinking previous steps and keeps responses aligned with its training style.

In multi-turn conversations, including a `previous_response_id` ensures earlier reasoning is preserved. This is especially critical when using tools, such as when a function call needs an extra round trip. In those cases, either pass the reasoning via `previous_response_id` or embed it directly in the `input`.

GPT-5 is OpenAI's most capable model yet, with particular strengths in:

- Code work — generation, debugging, and refactoring
- Instruction following — understanding and executing complex directions
- Long context + tool use — handling large inputs and orchestrating tool calls

New API features released alongside GPT-5 include verbosity controls, a minimal reasoning.effort mode, custom tool support, and an allowed-tools list for stricter execution control.

Reasoning vs. GPT Models

- Reasoning models work best with high-level objectives—like delegating to a senior colleague who can fill in the gaps and adapt.
- GPT models respond better to explicit, detailed instructions—like guiding a junior teammate who needs step-by-step direction.

Smarter Thinking When It Counts

ChatGPT now adapts to your needs with sharper reasoning and critical thinking capabilities—perfect for when you want deep, thoughtful responses.

Your Ultimate Coding Collaborator

From start to finish, GPT-5 handles complex coding challenges with ease. It generates cleaner, more usable code, offers better design solutions, and excels at debugging—making it an indispensable developer partner.

Master Communication—From Stories to Speeches

Whether you're crafting narratives, writing marketing copy, or delivering presentations, GPT-5 helps you create clearer, more persuasive messaging every time.

Best-in-Class for Health Questions

OpenAI's most advanced model yet for health-related queries—delivering accurate, trustworthy responses while serving as a thoughtful, proactive partner in your decision-making process.

More Accurate, Less Guesswork

GPT-5 is OpenAI's most reliable model so far—designed to minimize hallucinations and avoid pretending to know what it doesn't. It gives you facts, not fiction.

Tailor the Experience to You

Customize ChatGPT with a unique personality and your favorite chat color for a more personal, enjoyable experience.

Improved Voice Features

It understands your preferences better and lets you adjust its speaking tone and style to match your mood or context.

Learn Smarter with Study Mode

Get personalized, step-by-step guidance to help you learn any subject—at your pace, on your terms.

Smarter With Your Tools

Connect your Gmail and Google Calendar for tailored responses that work seamlessly with your schedule and inbox.

Customize Your Chat Experience with New Personalities

You now have five personality options in ChatGPT to tailor how it interacts with you:

- Default – The classic ChatGPT: clear, neutral, and adaptable across any topic or tone.
- Cynic – Dry, sarcastic, and brutally honest. Teasing? Yes. But when you need real help, it's straight to the point.
- Robot – No fluff. Just crisp, efficient, emotion-free answers—perfect for task-focused conversations.
- Listener – Calm, friendly, and supportive. Think reflective responses with a touch of wit.
- Nerd – Enthusiastic and inquisitive. Loves to explain and geek out over ideas with you.

These personalities currently apply to text only—not Voice mode.

Pick Your Favorite Accent Color

Personalize your chats visually by selecting an accent color that updates:

- Message bubbles
- Voice button

- Highlighted text

How to change it:

- Web: Click your profile icon → Settings → General → Choose from the Accent color dropdown.
- Mobile (iOS/Android): Tap your profile icon → Personalization → Color Scheme.

Upgraded Voice Mode

- Now compatible with custom GPTs
- Available with near-unlimited use for Plus users, and generous daily access for Free users
- Standard Voice Mode will be phased out in 30 days—all users will transition to the upgraded experience

For paid users, Voice now responds to your instructions, adjusting its:

- Speaking length
- Pace
- Tone
- And more—to match the moment

Built for Complex, Real-World Tasks

GPT5 takes AI performance to the next level—producing cleaner, high-quality code, generating UI components with minimal prompting, and executing long, multi-step tool calls with precision. With enhanced personality, steerability, and new API features like **minimal** reasoning and the **verbosity** parameter, developers and teams get finer control over how the model responds.

Your Smartest Teammate Yet

From writing and research to analysis, programming, and troubleshooting, GPT5 performs like a top-tier expert—delivering more accurate, polished, and context-aware outputs that make collaboration feel seamless and professional.

Everyone Can Be a Power User

With GPT5, you don't have to be technical to get expert-level results. It thinks critically, asks relevant follow-up questions, and helps users of all skill levels tackle complex workflows—without needing to switch models or tools.

Context-Aware & Connector-Ready

GPT5 is smarter in your company's world. It delivers higher-quality, context-aware responses by securely using your business data—integrating with platforms like Google Drive, SharePoint, and other connected apps—while always respecting existing access controls.

Is GPT-5 is WORSE than GPT-4?

Like GPT-4.1, GPT-5 is highly precise in following instructions, making it an incredibly versatile model across a wide range of workflows. However, this precision comes with a tradeoff: vague or conflicting prompts can significantly hinder performance. Unlike models that might default to one interpretation, GPT-5 will spend valuable reasoning tokens trying to resolve inconsistencies—often at the cost of output quality or speed.

Why Prompt Clarity Matters More with GPT-5

Because GPT-5 doesn't ignore contradictions or ambiguities, poorly constructed prompts are more harmful here than in earlier models. It attempts to reconcile every instruction, which can lead to unnecessary cognitive overhead and suboptimal responses.

We recognize that prompt engineering is inherently iterative. Prompts often evolve over time and may be touched by multiple stakeholders. This makes regular reviews essential. In fact, several early adopters discovered hidden flaws—ambiguous wording, redundant phrasing, or conflicting goals—within their core prompt libraries. Once removed, GPT-5's performance improved significantly, both in reliability and latency.

Recommendation

To ensure the best results:

- Review prompts regularly for clarity, consistency, and purpose.
- Use OpenAI's Prompt Optimizer Tool to identify hidden issues such as contradictions or overly vague instructions.
- Treat prompts as production code: iterate, audit, and refine continuously.

How to Access GPT-5

You can start chatting with GPT-5 at <https://chatgpt.com>.

To access GPT-5 in your applications via code, you can connect to the OpenAI API via a Software Development Kit (SDK) or REST API.

To access OpenAI via the API, you'll need to pay for your usage of the language models:

1. Create Your Developer Account

- Go to platform.openai.com → Sign Up → Verify email.

2. Get Credits or Set Billing

- New accounts get free trial credits (check in Usage).
- For paid use, add a payment method in Billing.

3. Control Costs

- In Billing → Usage Limits, set a hard limit (max monthly spend) and soft limit (early warning).
- Turn on email notifications.

4. Get Your API Key

- Go to View API Keys → Create new secret key.
- Copy it—you won't see it again.

5. Keep It Secure

- Never share your key or post it online.
- Store it in a `.env` file or environment variable.
- Regenerate immediately if leaked.

Key Reminder:

You're responsible for all usage tied to your API key—protect it like a password.

Assignment:

1. Create and verify your account.
2. Set spending limits and notifications.
3. Generate and securely store your API key.

You can use a coding language such as Python to interact with GPT-5 via OpenAI's Python SDK. If you want to run Python code generated by GPT-5, you'll need a code execution environment. The easiest option is Google Colab because it's free, online, and beginner-friendly.

1. Open Google Colab

- Go to colab.research.google.com.

- Sign in with your Google account.

2. Create a New Notebook

- Click File → New Notebook.
- You'll get an interactive coding environment where you can write and run Python code instantly.

3. Why Google Colab?

- No installation required.
- Runs Python in the cloud—no need for a powerful computer.
- Free GPU access for machine learning projects (optional).

You can also run Python offline on your computer if you install Python and a code editor on your computer.

Hello GPT-5: Build Your First API with Zero-Shot Prompting

Prompt engineering is the art and science of crafting effective prompts to elicit desired responses from large language models (LLMs) like GPT-5. A well-designed prompt can significantly influence the quality, relevance, and coherence of the generated output. It involves understanding the model's capabilities and limitations and strategically formulating input text to guide its behavior. Effective prompt engineering is crucial for maximizing the utility of LLMs in various applications.

- Key aspects of prompt engineering: Clarity, specificity, and context.
- Objective: To guide the model towards generating relevant and accurate outputs.

Experiment with different phrasing and prompt structures to discover what works best for your specific use case.

Here's an example of calling the OpenAI API:

```
from openai import OpenAI

client = OpenAI()

response = client.responses.create(
    model="gpt-5",
    input="Translate the following English text to French:
Welcome to Mammoth Club."
)

print(response.output_text)
```

The output property of the response contains an array of content generated by the LLM (Large Language Model). Here's an example of the output property:

```
[
  {
    "id": "msg_123abc",
    "type": "message",
    "role": "assistant",
    "content": [
      {
        "type": "output_text",
        "text": "Bienvenue au mammoth club.",
        "annotations": []
      }
    ]
  }
]
```

```
]
  }
]
```

The output array can include multiple items—not just one. It may contain tool calls, reasoning model token data, and more. Don't assume that the model's text response will always be found at `output[0].content[0].text`.

Besides returning plain text, the model can also provide structured data in JSON format. This capability is known as Structured Outputs.

Control Verbosity in GPT-5

The new **verbosity** parameter controls how many output tokens the model generates. Lower verbosity reduces overall latency, while higher verbosity produces more expansive responses. The model's underlying reasoning process remains largely unchanged, but it will adjust its expression to be more concise or more elaborate—affecting answer quality depending on your use case.

When to Use Different Verbosity Levels

- High verbosity – Ideal for tasks that require detailed output, such as in-depth document analysis or extensive code refactoring.
- Medium verbosity (*default prior to GPT-5*) – Balanced length and detail.
- Low verbosity – Best for short, precise outputs, such as SQL queries or minimalistic code generation.

Verbosity in Code Generation

- Medium / High – Produces longer, more structured code with inline comments and explanations.

- Low – Produces concise code with little or no commentary.

Example – Setting Low Verbosity

```
from openai import OpenAI  
client = OpenAI()  
response = client.responses.create(  
    model="gpt-5",  
    input="What is the secret to success?",  
    text={"verbosity": "low"}  
)  
print(response)
```

Note:

You can still influence verbosity via prompting after setting it in the API. The parameter sets a general token range at the system prompt level, but the actual output length will adapt to both developer and user instructions within that range.

Verbosity

The **verbosity** parameter controls how expansive the model's output is, without altering the underlying reasoning quality or prompt. It scales both the length and level of detail in the response, which can be set to:

- low → Terse, minimal prose. Best for short answers or compact code.
- medium (*default before GPT-5*) → Balanced detail and conciseness.
- high → Fully verbose, ideal for explanations, audits, teaching, or hand-offs.

Tip: **Keep prompts stable** and adjust verbosity via the parameter rather than rewriting the prompt.

Effect on Output

- Low verbosity – Minimal functional output; for code, this means a bare, working script with no commentary.
- Medium verbosity – Adds structure, explanatory comments, and reproducibility features.
- High verbosity – Produces a comprehensive, production-ready solution with extra context, options, and best practices.

Example: Coding Task with Varying Verbosity

Prompt: “Write a Python script to implement the Fibonacci sequence.”

Verbosity	Result
Lo	Simple script that prints results—no comments.
Medium	Adds comments, function structure, reproducibility controls, and verification checks.
High	Includes any or all of: CLI arguments, multiple potential backends (Python & NumPy), timing, verification, usage notes, and performance tips.

Key takeaway:

Use **verbosity** to quickly scale between concise and in-depth outputs without modifying your base prompt—ideal for toggling between quick tests and fully-documented deliverables.

Use Minimal Reasoning in GPT-5

reasoning.effort controls how many reasoning tokens the model generates before producing its reply. Earlier reasoning models (e.g., o3) supported only low, medium,

and high—with *low* prioritizing speed/fewer tokens and *high* prioritizing deeper reasoning. The default remains medium.

New: minimal

The new minimal setting emits very few reasoning tokens, giving you the fastest time-to-first-token. In practice, allowing *a few* tokens (vs. none) often performs better.

- Performs especially well for coding and instruction-following, closely adhering to directions.
- May need nudges to be more proactive.

Prompting tips for minimal

- Ask the model to briefly outline its steps (e.g., a 2–3 bullet “plan”) before answering to boost reasoning quality—even at minimal effort.

About Reasoning Effort

Reasoning models—such as GPT-5—are large language models trained with reinforcement learning to “think before answering” by producing an internal chain of thought before returning a response. This makes them ideal for:

- Complex problem solving
- Scientific and analytical reasoning
- Multi-step planning in agentic workflows
- Advanced coding tasks (including Codex CLI, our lightweight coding agent)

Model Options

- gpt-5 – Best for complex, broad-domain tasks; slower and more expensive, but with superior performance.

- gpt-5-mini – Smaller, faster, and less expensive; balances speed, cost, and capability.
- gpt-5-nano – Fastest and cheapest; ideal for simple, high-throughput tasks.

Using Reasoning Models via the Responses API

Example:

```
from openai import OpenAI

client = OpenAI()

prompt = """

    Write a Python script that takes a string representing RGB
    colors in the format '[255,0,0],[0,255,0],[0,0,255]' and
    outputs the average color in the same format.

    """

response = client.responses.create(
    model="gpt-5",
    reasoning={"effort": "medium"},
    input=[{"role": "user", "content": prompt}]
)

print(response.output_text)
```

Key Parameter:

reasoning.effort controls how many reasoning tokens the model uses before responding.

- minimum - *NEW TO GPT-5!*
- low – Faster, lower token usage

- medium (*default*) – Balanced
- high – Maximum reasoning depth for most accurate results

How Reasoning Works

Reasoning models use reasoning tokens in addition to standard input/output tokens. These tokens are invisible to the user but are billed as output tokens and occupy context space.

- Reasoning tokens are discarded before producing the visible answer.
- The `usage.output_tokens_details.reasoning_tokens` field shows the number of reasoning tokens used.

Tip: Reserve ~25,000 tokens for reasoning and output when first experimenting; adjust based on your prompt's needs.

Managing Context & Costs

- Limit total generated tokens with `max_output_tokens`.
- If you hit this limit before visible output is produced, you may incur reasoning token costs without seeing a result.
- For function calls, always pass back reasoning items from previous turns—either with `previous_response_id` or by manually including them.
- When using zero data retention (ZDR), include `reasoning.encrypted_content` in your request so reasoning items can be reused across turns.

Reasoning Summaries

Reasoning models can produce summaries of their thought process (but never raw reasoning tokens).

Set `reasoning.summary` to:

- `"auto"` – Best available detail
- `"concise"` or `"detailed"` – Depending on model support

Example:

```
response = client.responses.create(  
    model="gpt-5",  
    input="What is Mammoth Club?",  
    reasoning={"effort": "low", "summary": "auto"}  
)  
print(response.output)
```

Prompting Advice

- Reasoning models work best with high-level goals, trusting the model to fill in the details.
- Non-reasoning GPT models work best with precise, explicit step-by-step instructions.

Analogy:

- *Reasoning model*: Senior colleague—give the goal, trust the execution.
- *GPT model*: Junior colleague—give exact instructions.

Use Cases

- Data validation – e.g., checking synthetic medical datasets for anomalies

- Routine generation – e.g., creating structured actions from help center content
- Complex algorithms & refactoring – e.g., restructuring code for performance, style, or architecture

Recommended Prompting Strategies

To get the best results from minimal reasoning, OpenAI recommends adapting prompt patterns similar to those used with GPT-4.1. However, it's important to note that performance at this level is more sensitive to prompt quality, so precision and structure are key.

Best Practices for Prompting with Minimal Reasoning

1. Add Brief Thought Summaries at the Start of Responses
 - Include a bullet point list or short explanation that previews the model's reasoning.
 - This boosts effectiveness in higher-level reasoning tasks by simulating internal planning.
2. Use Descriptive Preambles for Tool Use
 - When prompting for agentic behavior (e.g., tool use or API calls), request clear, ongoing updates on the task's status.
 - Well-structured tool-calling preambles improve clarity and transparency in multi-step processes.
3. Disambiguate Tool Instructions Explicitly
 - Avoid vague or underspecified tool calls—clarity is critical at minimal reasoning.
 - Reinforce persistent behaviors and agent continuity through explicit agentic reminders.

4. Encourage Prompted Planning

- At minimal reasoning, the model has fewer internal tokens available for autonomous planning.
- Embedding explicit planning prompts helps ensure task completion, particularly in long-running or multi-part tasks.

OpenAI provides a **reasoning_effort** parameter that controls how intensively the model engages in reasoning and how proactively it invokes tools. The default setting is medium, but it should be adjusted based on the complexity of your task. For more demanding, multi-step tasks, a higher **reasoning_effort** is recommended to achieve optimal results.

Maximum performance is often observed when complex tasks are decomposed into distinct subtasks, each handled in separate agent turns.

Reuse Prompts with GPT-5 Templates

In the OpenAI dashboard, you can create reusable prompt templates for use in API requests, rather than hard-coding prompt text in your application. This approach makes it easier to build, test, and improve prompts—and to deploy updated versions without changing your integration code.

How it works:

1. Create a prompt template in the dashboard, using placeholders like **{{customer_name}}**.
2. Reference the prompt in your API request using the **prompt** parameter. The **prompt** object has three configurable properties:
 - **id** – The prompt's unique identifier (shown in the dashboard).

- **version** – The specific version to use (defaults to the dashboard's *current* version).
- **variables** – A mapping of values to fill in for the placeholders. Values can be plain strings or other Response input types like **input_image** or **input_file** (see the API reference for details).

Example — Using a prompt template in code:

```
from openai import OpenAI
client = OpenAI()
response = client.responses.create(
    model="gpt-5",
    prompt={
        "id": "pmpt_abc123",
        "version": "2",
        "variables": {
            "first_name": "Bob",
            "course_name": "Hello Coding"
        }
    }
)
print(response.output_text)
```

In this example, GPT-5 loads the predefined prompt from the dashboard, substitutes the given values for `{{first_name}}` and `{{course_name}}`, and generates the output—all without embedding the full prompt text in your source code.

Build Custom Tools with GPT-5

Function calling—also known as tool calling—lets OpenAI models access external functionality and data beyond their training. This enables models to follow instructions more accurately by connecting them to your application’s actions and data sources.

You can use:

- Function tools – Defined with a JSON schema describing input parameters and types.
- Custom tools – Accept free-form text input/output for maximum flexibility.

Prerequisites (Skip to the bottom for new Custom Tools feature)

- Tool – Functionality you make available to the model.
- Tool call – The model’s request to use a tool.
- Tool call output – The result you generate and send back to the model.

Tool Calling Flow

1. Send a request to the model, including available tools.
2. Model returns one or more tool calls.
3. Your application executes the tool logic.
4. Send the tool’s output back to the model.
5. Model responds to the user (or makes more tool calls).

Example – Function Tool

```
tools = [  
  {  
    "type": "function",  
    "name": "get_coding_challenge",  
    "description": "Get today's daily coding challenge.",  
    "parameters": {  
      "type": "object",  
      "properties": {  
        "sign": {  
          "type": "string",  
          "description": "A software developer  
interview question"  
        }  
      },  
      "required": ["sign"]  
    }  
  }  
]
```

After the model calls this function, you execute it, return the output, and re-send the result to get the final answer.

Note: For reasoning models like GPT-5, always return any reasoning items along with tool outputs.

Defining Functions

A function definition must include:

- type – `"function"`
- name – Short, clear identifier
- description – When and how to use the function
- parameters – JSON schema for inputs
- strict (*optional*) – Enforce schema compliance (recommended)

Best Practices

- Use clear names and parameter descriptions.
- Document when not to use a function.
- Use enums and schema rules to prevent invalid states.
- Avoid redundant params—pass known data in code.
- Keep the number of functions small (<20 recommended).
- Test definitions with non-experts (“intern test”).

Handling Function Calls

Responses may include multiple calls:

```
[
  {
    "type": "function_call",
    "name": "get_weather",
```

```
"arguments": "{\\"location\\":\\"New York, USA\\"}"
}
]
```

- Loop over all calls.
- Execute each function.
- Append a `function_call_output` entry with `call_id` and result string.
- Format results however you like (JSON, plain text, error codes).

Tool Choice

Control model behavior with `tool_choice`:

- `"auto"` (*default*) – Model decides.
- `"required"` – Must call one or more tools.
- Specific function – `{"type": "function", "name": "get_weather"}`
- `"allowed_tools"` – Limit accessible tools without changing the main tool list.
- `"none"` – No function calls allowed.

Parallel & Strict Mode

- Parallel calls – Model may call multiple tools at once; disable with `parallel_tool_calls=False`.
- Strict mode – Enforces exact schema adherence; requires `additionalProperties: false` and explicit `required` fields.

Streaming Tool Calls

You can stream tool call arguments as the model generates them, giving real-time insight into call progress.

Events include:

- `response.output_item.added` – New function call.
- `response.function_call_arguments.delta` – Partial argument JSON chunks.
- `response.function_call_arguments.done` – Final arguments.

Custom Tools (New to GPT-5)

Custom tools skip JSON schema and accept free-form text:

```
{  
  "type": "custom",  
  "name": "code_exec",  
  "description": "Executes JavaScript code."  
}
```

The model sends the raw input string as `input` in the tool call.

Token Usage

- Functions are embedded in the system message and count as input tokens.
- Limit the number and size of definitions to save tokens.
- Fine-tuning can help reduce token usage for large tool lists.

Freeform Function Calling in GPT-5

GPT-5 can now send raw text payloads—anything from Bash scripts to API requests—directly to your custom tool by using the new tool type `"type": "custom"`. Unlike traditional structured function calls, this approach skips JSON wrapping, giving you more freedom when working with external runtimes such as:

- Scripting engines (Bash, PowerShell, Ruby, etc.)
- NoSQL or graph databases
- Cloud CLI interfaces (AWS CLI, Azure CLI)
- Configuration and manifest generators

⚠ Note: The `custom` tool type does not support parallel tool calls.

Example – Fetch Latest Weather Data

The following code sends a plain text shell command to a `weather_cli` tool that fetches the current temperature for “London” and prints it.

```
from openai import OpenAI

client = OpenAI()

response = client.responses.create(
    model="gpt-5-mini",
    input="Use the weather_cli tool to get the current
temperature in London in Celsius",
    text={"format": {"type": "text"}},
    tools=[
        {
```

```
        "type": "custom",
        "name": "weather_cli",
        "description": "Executes shell commands to query
weather data",
    }
]
)
print(response.output)
```

Example tool call generated by GPT-5:

```
--- tool name ---
weather_cli
--- tool call argument (generated command) ---
curl -s "https://wttr.in/London?format=%t"
```

The raw shell command is sent to your backend, executed, and the output is returned to the model in the next call so it can present the result to the user.

Running a Query in Three Databases

To demonstrate freeform tool calling, here we ask GPT-5 to:

- Generate equivalent queries for MongoDB, PostgreSQL, and Neo4j that count all records in a “users” collection/table/graph.
- Ensure each query runs exactly once using the matching tool: `mongo_exec`, `postgres_exec`, and `neo4j_exec`.
- Print only the record count, then stop.

```
from openai import OpenAI
```

```
from typing import List, Optional

MODEL_NAME = "gpt-5"

TOOLS = [

    {"type": "custom", "name": "mongo_exec", "description":
"Runs MongoDB commands"},

    {"type": "custom", "name": "postgres_exec", "description":
"Runs PostgreSQL queries"},

    {"type": "custom", "name": "neo4j_exec", "description":
"Runs Neo4j Cypher queries"},

]

client = OpenAI()

def create_response(messages: List[dict], prev_id:
Optional[str] = None):

    kwargs = {

        "model": MODEL_NAME,

        "input": messages,

        "text": {"format": {"type": "text"}},

        "tools": TOOLS,

    }

    if prev_id:

        kwargs["previous_response_id"] = prev_id

    return client.responses.create(**kwargs)
```

```
def run_conversation(messages: List[dict], prev_id:
Optional[str] = None):
    response = create_response(messages, prev_id)
    tool_call = response.output[1] if len(response.output) > 1
else None
    if tool_call and tool_call.type == "custom_tool_call":
        print("--- tool name ---")
        print(tool_call.name)
        print("--- generated query ---")
        print(tool_call.input)
        # Simulated tool result
        messages.append({
            "type": "function_call_output",
            "call_id": tool_call.call_id,
            "output": "done"
        })
        return run_conversation(messages, prev_id=response.id)
    else:
        return

prompt = ""
```

Write queries to count all documents/records in a 'users' dataset for MongoDB, PostgreSQL, and Neo4j.

```
Always call these three tools exactly once: mongo_exec,  
postgres_exec, neo4j_exec.
```

```
Print only the total record count returned by each.
```

```
"""
```

```
messages = [{"role": "developer", "content": prompt}]
```

```
run_conversation(messages)
```

Example generated calls:

```
--- tool name ---
```

```
mongo_exec
```

```
--- generated query ---
```

```
db.users.countDocuments()
```

```
--- tool name ---
```

```
postgres_exec
```

```
--- generated query ---
```

```
SELECT COUNT(*) FROM users;
```

```
--- tool name ---
```

```
neo4j_exec
```

```
--- generated query ---
```

```
MATCH (u:User) RETURN COUNT(u);
```

Takeaways

Freeform tool calling in GPT-5 lets you send direct text instructions—like shell commands, database queries, or configuration snippets—straight to custom tools

without JSON formatting. This makes it ideal for cases where structured output is unnecessary, natural text is more intuitive, or you want to send the exact syntax your runtime expects.

Context-Free Grammar (CFG) in GPT-5

GPT-5 can use context-free grammars (CFGs) with custom tools, allowing you to attach a Lark grammar that locks outputs to a specific syntax or domain-specific language (DSL). By supplying a CFG—such as one for SQL, a query language, or a proprietary DSL—you ensure the assistant’s output strictly conforms to the grammar.

This approach makes it possible to produce precisely formatted and constrained tool calls or structured responses. It gives you strong control over syntactic compliance, making GPT-5 more reliable for domains that demand tight formatting rules or specialized language patterns.

Best practices for CFG-based custom tools

- Write clear, explicit tool descriptions
Be unambiguous about when and how you want the tool called. If the tool should *always* be invoked, state that plainly.
- Validate server-side outputs
Free-form strings are flexible but can carry risk—implement checks for injection, malformed commands, or unsafe instructions before execution.

A context-free grammar (CFG) is a set of production rules that says which strings are in a language. Each rule rewrites a nonterminal into a sequence of terminals (literal tokens) and/or other nonterminals, without depending on surrounding text—hence *context-free*. CFGs can model most programming-language syntax and, in OpenAI custom tools, act as strict *contracts* that force the model to emit only strings the grammar accepts.

Supported grammar syntaxes

- Lark (EBNF-style): <https://lark-parser.readthedocs.io/en/stable/>
- Regex: <https://docs.rs/regex/latest/regex/#syntax>

We constrain sampling with LLaGuidance under the hood: <https://github.com/guidance-ai/llguidance>.

Unsupported Lark features

- Regex lookahead (`(?=...)`, `(?!...)`, etc.)
- Lazy quantifiers (`*?`, `+`, `??`) in regexes
- Terminal priorities, templates, `%declares`, `%import` (except `%import common`)

Terminals vs. rules & greedy lexing

Concept	Take-away
Terminals (UPPER)	Matched first by the lexer; longest match wins.
Rules (lower)	Compose terminals; cannot change how text is tokenized.
Greedy lexer	Don't try to "shape" open-ended text across several terminals—you'll lose control.

Pattern Design

Use 1 bounded terminal for free text *between anchors*

```
start: S
S: /[A-Za-z, ]*(cat|dog)[A-Za-z, ]*(chased|found)[A-Za-z, ]*(a
toy|the food)[A-Za-z, ]*\./
```


Don't split the same free text across multiple terminals/rules

```
start: s

s: /[A-Za-z, ]+/ subject /[A-Za-z, ]+/ verb /[A-Za-z, ]+/
object /[A-Za-z, ]+/
```

Example – SQL Dialects: Oracle vs SQLite

This shows two independent Lark grammars that encode dialect differences: Oracle's **FETCH FIRST** vs SQLite's **LIMIT**. It also demonstrates how to prompt, invoke, and inspect tool calls in one script.

Define the Lark grammars

```
import textwrap

# ----- Oracle SQL grammar -----

oracle_grammar = textwrap.dedent(r"""

    // ----- Punctuation & operators -----

    SP: " "

    COMMA: ",",

    GT: ">"

    SEMI: ";"

    // ----- Start -----

    start: "SELECT" SP select_list SP "FROM" SP table SP
"WHERE" SP amount_filter SP "AND" SP date_filter SP "ORDER" SP
"BY" SP sort_cols SP "FETCH" SP "FIRST" SP NUMBER SP "ROWS" SP
"ONLY" SEMI

    // ----- Projections -----
```

```
select_list: column (COMMA SP column)*
column: IDENTIFIER

// ----- Tables -----
table: IDENTIFIER

// ----- Filters -----
amount_filter: "total_amount" SP GT SP NUMBER
date_filter: "order_date" SP GT SP DATE

// ----- Sorting -----
sort_cols: "order_date" SP "DESC"

// ----- Terminals -----
IDENTIFIER: /[A-Za-z_][A-Za-z0-9_]* /
NUMBER: /[0-9]+ /
DATE: /'[0-9]{4}-[0-9]{2}-[0-9]{2}' /

""")
# ----- SQLite grammar -----
sqlite_grammar = textwrap.dedent(r"""
    // ----- Punctuation & operators -----
    SP: " "
    COMMA: ",",
    GT: ">"
    SEMI: ";"
```

```
// ----- Start -----

start: "SELECT" SP select_list SP "FROM" SP table SP
"WHERE" SP amount_filter SP "AND" SP date_filter SP "ORDER" SP
"BY" SP sort_cols SP "LIMIT" SP NUMBER SEMI

// ----- Projections -----

select_list: column (COMMA SP column)*
column: IDENTIFIER

// ----- Tables -----

table: IDENTIFIER

// ----- Filters -----

amount_filter: "total_amount" SP GT SP NUMBER
date_filter: "order_date" SP GT SP DATE

// ----- Sorting -----

sort_cols: "order_date" SP "DESC"

// ----- Terminals -----

IDENTIFIER: /[A-Za-z_][A-Za-z0-9_]* /
NUMBER: /[0-9]+ /
DATE: /'[0-9]{4}-[0-9]{2}-[0-9]{2} '/

""")
```

Generate a specific SQL dialect

Oracle prompt & call

```
from openai import OpenAI
```

```
client = OpenAI()

sql_prompt_oracle = (
    "Call the oracle_grammar to generate a query for Oracle  
that retrieves the "
    "five most recent orders per customer, showing customer_id,  
order_id, order_date, and total_amount, "
    "where total_amount > 500 and order_date is after  
'2025-01-01'."
)

resp_oracle = client.responses.create(
    model="gpt-5",
    input=sql_prompt_oracle,
    text={"format": {"type": "text"}},
    tools=[{
        "type": "custom",
        "name": "oracle_grammar",
        "description": "Read-only Oracle SELECT with WHERE/  
ORDER BY and FETCH FIRST.",
        "format": {"type": "grammar", "syntax": "lark",
        "definition": oracle_grammar}
    }],
    parallel_tool_calls=False
)
```

```
print("--- Oracle SQL Query ---")  
print(resp_oracle.output[1].input)
```

Possible output

```
SELECT customer_id, order_id, order_date, total_amount FROM  
orders  
  
WHERE total_amount > 500 AND order_date > '2025-01-01'  
  
ORDER BY order_date DESC FETCH FIRST 5 ROWS ONLY;
```

SQLite prompt & call

```
sql_prompt_sqlite = (  
    "Call the sqlite_grammar to generate a query for SQLite  
    that retrieves the "  
  
    "five most recent orders per customer, showing customer_id,  
    order_id, order_date, and total_amount, "  
  
    "where total_amount > 500 and order_date is after  
    '2025-01-01'."  
)  
  
resp_sqlite = client.responses.create(  
    model="gpt-5",  
    input=sql_prompt_sqlite,  
    text={"format": {"type": "text"}},  
    tools=[  
        {"type": "custom",  
         "name": "sqlite_grammar",
```

```
        "description": "Read-only SQLite SELECT with WHERE/  
ORDER BY and LIMIT.",  
        "format": {"type": "grammar", "syntax": "lark",  
"definition": sqlite_grammar}  
    }],  
    parallel_tool_calls=False  
)  
print("--- SQLite SQL Query ---")  
print(resp_sqlite.output[1].input)
```

Possible output

```
SELECT customer_id, order_id, order_date, total_amount FROM  
orders  
  
WHERE total_amount > 500 AND order_date > '2025-01-01'  
  
ORDER BY order_date DESC LIMIT 5;
```

Example – Regex CFG: UUID v4

This shows a Regex grammar that forces the tool call to return a UUID v4 string.

```
from openai import OpenAI  
  
client = OpenAI()  
  
uuid_v4_regex = r"^[0-9a-f]{8}-[0-9a-f]{4}-4[0-9a-f]{3}-[89ab]  
[0-9a-f]{3}-[0-9a-f]{12}$"  
  
uuid_prompt = "Call the uuid_grammar to save a UUID v4  
identifier."
```

```
resp_uuid = client.responses.create(
    model="gpt-5",
    input=uuid_prompt,
    text={"format": {"type": "text"}},
    tools=[{
        "type": "custom",
        "name": "uuid_grammar",
        "description": "Saves a UUID v4 (lowercase hex).",
        "format": {"type": "grammar", "syntax": "regex",
"definition": uuid_v4_regex}
    }],
    parallel_tool_calls=False
)
print("--- UUID ---")
print(resp_uuid.output[1].input)
```

Possible output

```
3b7c1e20-0f75-4a61-9f6a-5f2e2d7f6b0d
```

Best Practices

- Bound terminals. Prefer `/[^\n]{0,10}*\. /` over `/.*\./`. Constrain by content (negated classes) and length (`{M,N}`).
- Avoid `.` wildcards; use explicit character classes.
- Thread whitespace explicitly (e.g., `SP: " "`) rather than a global `%ignore`.

- Describe the tool well. Tell the model exactly what the CFG accepts and instruct it to reason carefully about compliance.

Troubleshooting

- “Grammar too complex” (API rejects it): Simplify rules/terminals; remove broad `%ignore` patterns.
- Unexpected tokens: Check for overlapping terminals; remember the lexer is greedy.
- Model “drifts” semantically while staying syntactic:
 - Tighten the grammar.
 - Refine the prompt (few-shots) and the tool description (spell out constraints).
 - Try higher `reasoning.effort` (e.g., bump from medium → high).

PART 2: PROMPT ENGINEERING FOR GPT-5

Control Eagerness of GPT-5

Agentic scaffolds can vary widely in how much control they give the underlying model. Some delegate most decision-making to the model, while others tightly restrict it with explicit programmatic logic. GPT-5 is trained to operate anywhere on this spectrum—capable of making high-level decisions in ambiguous situations or focusing on well-defined, narrow tasks. This section explains how to calibrate GPT-5’s *agentic eagerness*—its balance between taking initiative and waiting for direct instructions.

Prompting for Less Eagerness

By default, GPT-5 is thorough when gathering context in an agentic workflow, often expanding its scope to ensure correctness. To limit this—reducing tangential tool calls and improving speed—consider:

- Lowering **reasoning_effort**: This decreases exploration depth, improving efficiency and latency. Many workflows work just as well at medium or low settings.
- Providing explicit exploration boundaries: Define exactly how you want the model to investigate the problem space.

Example:

```
<context_gathering>
```

```
Goal: Acquire only the necessary context, quickly. Parallelize  
discovery; stop as soon as actionable information appears.
```

```
Method:
```

- ```
- Begin broad, then focus into subqueries.
- Run multiple queries in parallel; review only top hits per
query.
- Deduplicate and cache results; avoid repeating searches.
- Skip excessive context hunting; run one targeted batch if
needed.
```

```
Early stop criteria:
```

- ```
- You can name the exact content to change.  
- ~70% of top hits point to the same area.
```

Escalate once:

- If results conflict or scope is unclear, run one refined batch, then act.

Depth:

- Trace only the elements you will modify or rely on.
- Avoid expanding transitively unless required.

Loop:

- **Batch search** → **minimal plan** → **execute**.
- Search again only if validation fails or new unknowns emerge.

</context_gathering>

For maximum restriction, set a fixed tool-call budget:

<context_gathering>

- Search depth: minimal
- Prioritize speed over certainty – proceed even if the answer may be partially incomplete.
- Limit: 2 tool calls.
- If more investigation is needed, share current findings and questions; continue only with user confirmation.

</context_gathering>

Giving the model an *escape hatch*—permission to act with some uncertainty—helps it stop gathering context sooner.

Prompting for More Eagerness

To encourage autonomy, persistence, and reduced hand-offs, increase `reasoning_effort` and use language that promotes continued action:

```
<persistence>
```

- Operate as an agent – continue until the query is fully resolved.
- End your turn only when certain the problem is solved.
- Do not stop for uncertainty – research or infer the most reasonable path forward and continue.
- Avoid asking for confirmation; instead, choose the most reasonable assumption, proceed, and document your reasoning afterward.

```
</persistence>
```

It's also helpful to clearly define stop conditions, outline safe vs. unsafe actions, and specify when hand-offs are acceptable.

For instance, in a shopping workflow, checkout/payment tools may require more certainty before acting, while search tools can operate with far greater autonomy. In coding, deleting files should have a lower uncertainty tolerance than running a `grep` search.

Control Instructions and Roles in GPT-5

You can guide the model's behavior using the `instructions` API parameter in combination with message roles, which vary in authority.

```
from openai import OpenAI  
  
client = OpenAI()
```

```
response = client.responses.create(  
    model="gpt-5",  
    instructions="Act as a reassuring friend.",  
    input="I have a huge bug in my code. Help me find the  
error.",  
)  
print(response.output_text)
```

The **instructions** parameter sets high-level directives for how the model should generate its response—such as tone, goals, and examples of ideal output. These instructions take precedence over anything provided in the **input** field. However, they only apply to the current response generation. If you're managing ongoing conversation state using **previous_response_id**, any past **instructions** will not carry over to the new context.

Message Roles and Prioritization

Messages in a conversation come with different roles, each carrying a different priority:

- developer (**instructions**)— These messages are provided by the application developer and define system-level rules or business logic. They have the highest priority.
- user – These are prompts from the end user and are evaluated after developer messages.
- assistant – These messages are generated by the model in response to developer and user inputs.

Multi-Turn Conversations

A conversation may include multiple messages across these roles, potentially including additional content types from both the user and the model. This supports rich, dynamic, and stateful interactions.

You can think of developer and user messages as similar to function definitions and function arguments, respectively:

- Developer messages define the system's behavior (like a function's logic).
- User messages serve as inputs that the system logic applies to (like arguments to a function).

Fine-Tuning GPT-5

Fine-tuning GPT-5 is the process of taking the **base GPT-5 model** from OpenAI and giving it **extra training on your own custom dataset** so that it learns patterns, terminology, style, and behavior specific to your use case.

How It Works

1. Start with a Pre-Trained Model

GPT-5 has already been trained on vast amounts of text and code, so it knows general language, reasoning, and problem-solving.

2. Prepare a Target Dataset

You collect examples—prompts and desired outputs—that reflect exactly how you want the model to respond.

- Example: Customer support questions and ideal answers in your company's tone.
- Example: Legal clauses and preferred rewrites.

3. **Additional Training (Fine-Tuning)**

The model is updated to give more weight to your dataset's patterns. This usually uses far fewer training steps than the original GPT-5 training (which was done on huge compute clusters).

4. **Deployment & Use**

Your fine-tuned GPT-5 behaves like GPT-5 at its core, but with **specialized knowledge, formatting, or persona traits**.

Benefits

- **Custom tone & style** — match your brand voice.
- **Domain expertise** — embed industry-specific knowledge.
- **Instruction adherence** — better follow certain formats or rules.
- **Efficiency** — fewer prompt tokens needed for the same quality output.

When to Use vs. Prompt Engineering

- **Prompt engineering:** Great for quick custom behavior without training.
- **Fine-tuning:** Better when you need *consistent, predictable, and specialized* output across many prompts.

Zero-Shot Prompting

Zero-shot prompting asks a model to complete a task using only instructions, without providing *task-specific examples*. This works because large language models internalize broad knowledge and patterns during pretraining, which we activate with carefully phrased prompts.

Use zero-shot prompting when you need speed, generality, and low setup cost. It excels at *classification* with clear labels, *extraction* of specific fields, *summarization* with

constraints, and *transformation* like rewriting tone or format. When precise accuracy is critical, we often pair it with validation and post-processing.

Key idea: articulate the task, constraints, and success criteria explicitly. Replace implied context with explicit, testable rules. For instance: “Classify sentiment as Positive, Neutral, or Negative. Output just one label.” These guardrails reduce ambiguity.

Example uses: triaging support tickets, normalizing log lines, drafting policy summaries, and generating checklist steps. In each, the prompt defines the contract; the model fills in the content.

- Task clarity: define what to do and what to avoid.
- Output format: specify exact schema or label set.
- Constraints: length, tone, style, or domain vocabulary.

Tip: If the model can “guess,” it will. Remove guesswork by stating *allowed labels*, *forbidden behaviors*, and one definitive output format.

Definition: *Zero-shot prompting* is instructing a model to perform a novel task with no in-prompt examples, relying solely on clear, constrained instructions.

Core Principles and Pitfalls

Adopt principles that reduce ambiguity and enforce consistency. Professionals benefit most when prompts are reproducible, auditable, and easy to iterate.

Below are essential practices with concise rationales and examples to accelerate your zero-shot success.

Use these as a checklist for any prompt you deploy in production or share with teammates.

Revisit and refine your prompts regularly; slight wording changes can alter outcomes significantly.

- Be explicit: “Return one of: *Approve* or *Reject*” beats “Decide.”
- Constrain output: Enforce JSON keys, lengths, or label sets to simplify parsing.
- State role: “You are a *security analyst*” primes domain reasoning.
- Use delimiters: Wrap inputs with `<input>...</input>` to avoid leakage.
- One task at a time: Split classification and explanation into separate calls for reliability.
- Prefer verification: Ask the model to validate its own compliance with your spec.
- Minimize subjective adjectives: Replace “good” with measurable criteria like \geq “100 words.”
- Guard against hallucination: Add “If unsure, output *Unknown*.”

Anti-pitfall: Don’t bury requirements mid-paragraph. Place critical rules at the end under “Output Requirements:” for salience.

Key takeaway: Zero-shot reliability comes from *contracts*—clear tasks, strict formats, and explicit refusal conditions.

Prompt Patterns at a Glance

Here’s a focused set of patterns you can combine to design robust zero-shot prompts quickly and consistently.

Each pattern maps to a frequent professional need: classification, extraction, transformation, and summarization under constraints.

Use them as building blocks. Start simple, then add only the constraints you truly need.

Keep labels and fields stable across iterations to enable apples-to-apples evaluation.

- Classification: Provide fixed label set and tie-breaker rules.

- Extraction: Name fields, data types, and default values for missing data.
- Transformation: Specify tone, audience, and length ranges.
- Summarization: Require bullet points, include/exclude lists, or KPI emphasis.

Pattern reuse boosts velocity. Maintain a prompt library with versioned templates.

Reminder: Always include a final line that restates the exact output format to reduce drift.

Zero-Shot Code Examples + Outputs

Below are copy-ready prompts and their expected outputs. They illustrate strict formatting, refusal behavior, and concise labeling.

Use these snippets in your prompt tooling or API calls. The outputs show how constraints guide the model's response.

Adapt the labels and fields to your domain while keeping structure intact for stable parsing.

Remember: output blocks show results, not the code itself.

- Classification: Single-label sentiment.
- Extraction: Fields with defaults.
- Transformation: Tone-constrained rewrite.

You are a data quality reviewer.

Task: Classify sentiment of the text as one of: Positive, Neutral, Negative.

Rules: Output exactly one label. If unclear, output Unknown.

Text: "The rollout was smooth, but support was slow."

Neutral

Role: Information extractor.

Task: Extract fields as JSON. Use null if missing.

Fields: { "name": string, "email": string|null, "company": string|null }

Input: "Hi, I'm Dana from Zephyr Labs. Reach me at dana@zephyrlabs.io."

```
{"name":"Dana","email":"dana@zephyrlabs.io","company":"Zephyr Labs"}
```

You are a professional editor.

Task: Rewrite the note for an executive audience in ≤40 words, neutral tone.

Note: "We missed the last two sprints due to vendor delays and unclear specs."

Schedule slipped due to vendor delays and ambiguous requirements. Mitigation is in progress; revised timeline shared this week.

Keep outputs short and canonical. Short labels → fewer parsing errors.

Validation tip: Post-validate outputs with a JSON schema or label whitelist to catch drift.

Crafting Effective Task Headings

Headings prime the model and readers. Precise headings act as a contract title that orients the task, scope, and constraints at a glance.

Place the heading at the top, then include a compact list of rules. Close with a one-line format requirement for maximum clarity.

Good headings are concise, role-aware, and outcome-focused: “Security Log Classifier – Criticality Labels,” not “Analyze this.”

Use delimiters to isolate inputs under the heading, reducing accidental instruction ingestion.

- Include role: “You are a *financial analyst*.”
- Include task: “Rate risk: Low, Medium, High.”
- Include format: “Output JSON with keys x, y, z.”

Heading: PII Extractor – Email and Phone Only

You are a compliance assistant.

Task: Extract "email" and "phone" from the input. Use null if absent.

Output: {"email": string|null, "phone": string|null}

Input:

<input>Text here</input>

Aim for scannability: heading + 3–5 bullets + format line. ✨

Pattern: *[Role]* → *[Task]* → *[Rules]* → *[Format]* → *[Input]*. This linear flow reduces ambiguity.

Ordered Workflow for Zero-Shot Prompts

Follow a repeatable workflow to improve reliability. Each step builds constraints that anchor the model’s behavior.

This sequence works for most professional tasks and keeps prompts auditable and versionable.

Treat each prompt as a small spec; iterate with measured changes only.

Document every change and its evaluation score to avoid regressions.

- 1) Define outcome: Specify the exact output and acceptance criteria.
 - 2) Choose labels/fields: Fix the schema and defaults.
 - 3) Write rules: State constraints, refusals, and tie-breakers.
 - 4) Add role + delimiters: Prime the model and isolate inputs.
 - 5) Test on a small set: Check format compliance first.
 - 6) Evaluate: Score accuracy, coverage, and stability.
 - 7) Iterate: Change one variable at a time; re-score.
1. Start with the minimal viable prompt.
 2. Add only constraints that reduce real errors.
 3. Automate checks for output format and label validity.

Discipline beats cleverness. A boring, consistent prompt outperforms a clever but unstable one.

Checkpoint: Before rollout, verify the prompt on *edge cases* and *near misses*.

Evaluating Zero-Shot Quality

Evaluation turns intuition into evidence. Use a labeled set or heuristic oracle to quantify quality across iterations.

Measure both correctness and compliance. A perfectly formatted but wrong answer is as problematic as a correct answer that breaks your parser.

Define a *rubric* with weighted criteria like accuracy, format compliance, verbosity, and refusal correctness.

Automate evaluation to catch regressions when you tweak wording or model versions.

- Metrics: accuracy, F1, format compliance rate, average length.
- Rubrics: weights per criterion; pass/fail thresholds.
- Sampling: include clean, noisy, and adversarial inputs.
- Reporting: track scores by prompt version and model.

Evaluator role: Score the model's output.

Criteria (weights): Accuracy 0.6, Format 0.2, Brevity 0.1, Refusal 0.1.

Return:

```
{"accuracy":0-1,"format":0-1,"brevity":0-1,"refusal":0-1,"overall":0-1}
```

```
{"accuracy":0.87,"format":1.0,"brevity":0.9,"refusal":0.8,"overall":0.89}
```

Log both raw outputs and scores. Without outputs, scores are hard to interpret.

Key takeaway: Tie deployment to a minimum **overall** score and alert on drops.

Constraints, Guardrails, and Safety

Constraints make outputs predictable; guardrails make them safe. Zero-shot prompts should specify what to do and what to refuse.

Define acceptable content, required format, and unambiguous fallback behavior like returning *Unknown* or *null*.

Include disclaimers for regulated domains and require citations or confidence when uncertainty is high.

Where possible, post-validate with schemas, allow-lists, and length checks.

- Content rules: “No PII in output,” “No legal advice,” etc.
- Refusal logic: “If task violates policy, output 'Refuse' with reason.”

- Format enforcement: “Return JSON only, no prose.”
- Length caps: Tokens or characters to control cost and drift.

Role: Policy-aware classifier.

Task: Label as {Allow, Review, Refuse}. If unsure, choose Review.

Policy: No medical diagnosis. No PII exposure. Output one label only.

Input: "Should I take 200mg ibuprofen for this injury?"

Refuse

Safety first: encode policy in the prompt and verify outputs downstream. 🛡️

Warning: Without explicit refusal paths, models may attempt unsafe answers.

Zero-Shot vs Few-Shot vs CoT

Pick the right technique for the job. Zero-shot is fast and general; few-shot adds structure via examples; chain-of-thought (CoT) encourages stepwise reasoning.

Zero-shot works best when tasks are well-specified and objective. Few-shot shines for style mimicry or subtle label semantics. CoT helps with multi-step logic.

Beware that CoT may increase verbosity and cost; use it only when reasoning benefit outweighs overhead.

Start with zero-shot; escalate only if metrics demand it.

- Zero-shot: No examples; strict rules; fast iteration.
- Few-shot: 2–5 curated examples; teaches format and edge handling.
- CoT: “Think step by step”; improves reasoning tasks.

Technique	Best for	Cost	Risk
Zero-shot	Clear, objective tasks	Low	Spec ambiguity
Few-shot	Style/format fidelity	Medium	Overfitting examples
CoT	Reasoning, math, planning	Higher	Verbose drift

Decision rule: Use the simplest approach that meets your measured quality bar.

Note: You can combine techniques, e.g., zero-shot with a short verifier step.

Professional Use Cases

Zero-shot prompting delivers rapid value across functions without building large example sets. Here are practical patterns you can deploy today.

Align each case to business outcomes like reduced triage time, improved data quality, or faster insight generation.

Keep the output format tight to integrate with downstream systems.

Iterate with real data from your workflows, not synthetic placeholders.

- Customer Support: Triage severity with labels {P1, P2, P3}. Require one label.
- Sales Ops: Extract contact fields with defaults; validate emails.
- Security: Classify alerts as {Benign, Suspicious, Malicious} with refusal if data is incomplete.
- Finance: Summarize earnings calls into KPIs with ≤ 120 words.

Role: Support triage assistant.

Task: Assign severity {P1,P2,P3}. Output one label.

Input: "Checkout down for EU customers since 10:05 UTC."

P1

Keep label definitions documented so humans and models share the same vocabulary.

Integration tip: Log the prompt version with each prediction to trace outcomes.

Troubleshooting and AntiPatterns

Even strong zero-shot prompts can drift. Detect and fix issues with a structured debugging approach.

Catalog your failure modes: format violations, label ambiguity, overlong responses, and unsafe content. Map each to a corrective action.

When results vary, add determinism via stricter constraints and simpler wording. Remove flourish; keep verbs imperative.

Escalate to few-shot or a verifier step only if constraints cannot fix the issue.

- Anti-pattern: Vague multi-task prompts → *Fix*: Split into atomic tasks.
- Anti-pattern: “Explain and classify” → *Fix*: First classify, then ask for rationale.
- Anti-pattern: Unbounded prose → *Fix*: Add word or token caps.
- Anti-pattern: Hidden policies → *Fix*: Inline rules and refusal criteria.

Bad: "Analyze this and tell me what you think."

Better: "Label as {Positive,Neutral,Negative}. Output one label only. Input: <input>...</input>"

When in doubt, add a final “Output Requirements:” line. Models tend to obey the last instruction.

Recovery checklist: Reduce scope, simplify labels, enforce format, add refusal path, then re-evaluate.

Summary & Next Steps

You learned how to define tasks, constrain outputs, and add guardrails to make zero-shot prompts reliable. We covered pattern libraries, evaluation, safety, comparisons to few-shot/CoT, and troubleshooting. Next steps: build a small prompt library, instrument evaluation with a rubric, and pilot prompts on real workflows. Iterate using the ordered workflow, track scores by version, and deploy with schema validation.

Few-Shot Prompting in GPT-5

Few-shot learning (few-shot prompting) is a powerful technique for guiding GPT-5 by providing a small number of example input-output pairs. This allows the model to learn from the examples and generalize to new, unseen inputs. The quality and relevance of the examples are crucial for the effectiveness of few-shot learning.

- **Relevant examples:** Choose examples that are representative of the type of output you want the model to generate.
- **Consistent formatting:** Maintain a consistent format across all examples to help the model identify patterns and relationships.
- **Clear separation:** Clearly separate the input and output sections of each example to avoid ambiguity.

Few-shot learning is like showing GPT-5 a few completed assignments before asking it to do its own.

Here's an example of a few-shot prompt for sentiment analysis:

Input: "This movie was absolutely terrible."

Output: Negative

Input: "I had a fantastic experience at the restaurant."

Output: Positive

```
Input: "The product was okay, but nothing special."
```

```
Output: Neutral
```

```
Input: "The new software update is incredibly buggy."
```

```
Output:
```

Few-shot prompting bridges the gap between zero-shot and fine-tuning, offering a balance between data efficiency and performance.

- Key advantage: Improved performance compared to zero-shot, with minimal training data.
- Challenge: Requires careful selection of representative examples.

The quality of the examples in few-shot prompting is crucial. Choose examples that are diverse and representative of the task.

Consider a task where you want GPT-5 to classify customer reviews as positive or negative. In a few-shot prompt, you would include a few example reviews along with their corresponding sentiment labels. The model then learns from these examples and applies that knowledge to classify new reviews. The more relevant the examples, the better the performance.

Python Code Example: Few-Shot Text Classification

This coding example demonstrates few-shot text classification using GPT-5. We'll provide a few examples of text and their corresponding labels, and then ask the model to classify a new piece of text. This illustrates how to guide GPT-5 with minimal examples to achieve a specific task.

```
from openai import OpenAI  
  
client = OpenAI()  
  
print(response.output_text)
```

```
def classify_text(text, examples):  
    prompt = "Classify the following text based on the examples  
provided:\n\n"  
    for example in examples:  
        prompt += f"Text: {example['text']}\nLabel:  
{example['label']}\n\n"  
    prompt += f"Text: {text}\nLabel:"  
    response = client.responses.create(  
        model="gpt-5",  
        input="prompt"  
    )  
    return response.choices[0].text.strip()  
  
examples = [  
    {"text": "This is an amazing product!", "label":  
"Positive"},  
    {"text": "I am very disappointed with this purchase.",  
"label": "Negative"},  
    {"text": "The service was excellent.", "label":  
"Positive"},  
    {"text": "The quality is terrible.", "label": "Negative"}  
]  
  
text_to_classify = "I am extremely happy with my new phone."  
predicted_label = classify_text(text_to_classify, examples)
```

```
print(f"Text: {text_to_classify}")  
print(f"Predicted Label: {predicted_label}")  
  
Text: I am extremely happy with my new phone.  
Predicted Label: Positive
```

This code defines a function `classify_text` that takes a text and a list of examples as input. It constructs a prompt that includes the examples and the text to be classified. The GPT-5 model then predicts the label for the input text based on the provided examples. The `max_tokens` parameter limits the length of the generated label, and the `stop` parameter tells the model to stop generating text at the end of the line.

Prompt Structure

The structure of your prompt plays a vital role in the effectiveness of both zero-shot and few-shot prompting. A well-structured prompt provides clear instructions and context to the model, guiding it towards the desired output. Consider the following elements when designing your prompts: instruction, context, input data, and output indicator.

- **Instruction:** Clearly state the task you want the model to perform.
- **Context:** Provide any relevant background information or constraints.
- **Input data:** The data that the model should process.
- **Output indicator:** A cue to indicate the desired format or style of the output.

A clear and concise prompt structure improves the model's understanding and reduces ambiguity, leading to more accurate and relevant outputs.

For example, when using few-shot prompting for code generation, your prompt might include an instruction like "Generate Python code to...", followed by context about the desired functionality, input data specifications, and an output indicator such as

"```python". This structured approach helps the model understand the task and generate the appropriate code.

Temperature and Top-P Sampling

Temperature and Top-P sampling are parameters that control the randomness and diversity of the generated output. Temperature adjusts the probability distribution of the next token, while Top-P sampling selects from the most probable tokens whose cumulative probability exceeds a certain threshold. These parameters allow you to fine-tune the balance between creativity and accuracy in the model's responses.

- Temperature: Higher values (e.g., 0.9) lead to more random and creative outputs, while lower values (e.g., 0.2) produce more deterministic and focused outputs.
- Top-P: A value of 1.0 selects from all possible tokens, while lower values (e.g., 0.5) restrict the selection to the most probable tokens.

Experiment with different temperature and Top-P values to find the optimal settings for your specific task and desired output style.

For example, if you're using GPT-5 for creative writing, you might increase the temperature to generate more imaginative and unexpected results. Conversely, if you need precise and factual answers, you would lower the temperature to reduce randomness. Top-P sampling can be used in conjunction with temperature to further control the output distribution.

Advanced Prompting Techniques

Beyond basic zero-shot and few-shot prompting, several advanced techniques can further enhance the performance of GPT-5. These include chain-of-thought prompting, self-consistency decoding, and prompt ensembling. These techniques leverage the model's capabilities in more sophisticated ways to achieve improved accuracy and robustness.

- Chain-of-thought prompting: Encourages the model to explain its reasoning process step-by-step, leading to more accurate and interpretable results.
- Self-consistency decoding: Generates multiple responses and selects the most consistent one, improving the reliability of the output.
- Prompt ensembling: Combines the outputs from multiple prompts to reduce variance and improve overall performance.

Advanced prompting techniques can significantly improve the performance of GPT-5, especially for complex reasoning tasks.

For instance, when tackling a mathematical problem, chain-of-thought prompting can guide the model to break down the problem into smaller steps and explain its reasoning at each step. This not only improves the accuracy of the solution but also provides valuable insights into the model's thought process. Similarly, self-consistency decoding can be used to mitigate the effects of randomness and ensure that the generated output is consistent across multiple runs.

Problem: A baker has 15 cookies. He sells 7 cookies and then bakes 12 more. How many cookies does he have now?

Solution:

First, the baker starts with 15 cookies.

Then, he sells 7 cookies, so we subtract 7 from 15: $15 - 7 = 8$.

Next, he bakes 12 more cookies, so we add 12 to 8: $8 + 12 = 20$.

Therefore, the baker has 20 cookies now.

The `solve_problem_cot` function takes a problem as input and constructs a prompt that encourages the model to explain its reasoning process step-by-step. The prompt

includes the phrase "Let's think step by step:" to guide the model's response. The model then generates a detailed explanation of its reasoning, leading to the final answer. The `max_tokens` parameter controls the length of the generated explanation.

Evaluating Prompt Performance

Evaluating the performance of your prompts is crucial for optimizing their effectiveness. This involves assessing the quality, relevance, and accuracy of the generated outputs. Several metrics can be used to evaluate prompt performance, depending on the specific task. These include accuracy, precision, recall, F1-score, and human evaluation.

- **Accuracy:** The percentage of correct predictions.
- **Precision:** The proportion of predicted positives that are actually positive.
- **Recall:** The proportion of actual positives that are correctly predicted.
- **F1-score:** The harmonic mean of precision and recall.
- **Human evaluation:** Subjective assessment by human evaluators.

Regularly evaluate your prompt performance and iterate on your prompts to improve their effectiveness.

For example, when evaluating a prompt for sentiment classification, you would measure the accuracy of the model's predictions against a labeled dataset. You might also conduct human evaluations to assess the subjective quality of the generated sentiment labels. By analyzing these metrics, you can identify areas for improvement and refine your prompts accordingly.

Prompt Engineering Best Practices

Effective prompt engineering requires a combination of creativity, experimentation, and systematic evaluation. By following these best practices, you can maximize the

performance of GPT-5 and build robust and reliable applications. These include being clear and specific, providing context, using examples, and iterating on your prompts.

- Be clear and specific: Avoid ambiguity and clearly state the task you want the model to perform.
- Provide context: Provide relevant background information or constraints to guide the model's response.
- Use examples: Use few-shot prompting to demonstrate the desired input-output relationship.
- Iterate on your prompts: Regularly evaluate your prompt performance and refine your prompts based on the results.

Prompt engineering is an iterative process. Continuously experiment and refine your prompts to achieve optimal results.

For instance, when designing a prompt for code generation, start with a clear and specific instruction, such as "Generate Python code to implement a function that calculates the factorial of a number." Then, provide context about the desired input and output formats. Include a few examples of input-output pairs to guide the model. Finally, evaluate the generated code and iterate on your prompt to address any issues or improve its performance.

Mastering few-shot and zero-shot prompting techniques is essential for unlocking the full potential of GPT-5. By understanding the principles of prompt engineering, you can build powerful applications that require minimal or no training data. Experiment with different prompting strategies, evaluate their performance, and continuously iterate to refine your prompts. With practice and dedication, you can become a proficient prompt engineer and leverage the power of GPT-5 to solve a wide range of problems.

- Key takeaways: Prompt engineering is crucial, zero-shot and few-shot prompting are powerful techniques, and continuous experimentation is essential.
- Next steps: Practice building prompts for different tasks, evaluate their performance, and refine your skills.

The future of AI lies in the ability to effectively communicate with and guide large language models through well-designed prompts.

Remember, the key to successful prompt engineering is to be clear, specific, and iterative. By continuously experimenting and refining your prompts, you can unlock the full potential of GPT-5 and build innovative and impactful applications. Keep exploring, keep learning, and keep pushing the boundaries of what's possible with AI!

Prompt Design Techniques

Effective prompt design hinges on providing GPT-5 with sufficient context. This means not only stating what you want but also framing the request in a way that guides the model toward the desired outcome. Consider the nuances of the task, the expected format of the response, and any relevant background information that might influence the model's reasoning.

- Specificity is key: Avoid vague or ambiguous language. Clearly define the scope and objectives of your prompt.
- Role assignment: Explicitly instruct GPT-5 to assume a specific persona or role. This can significantly improve the quality and relevance of the generated content.
- Contextual grounding: Provide relevant examples, data points, or background information to anchor the model's response in a specific domain or scenario.

Always iterate on your prompts. The first attempt is rarely the best. Experiment with different phrasing, context, and constraints to refine the model's output.

For example, instead of asking "Write a summary," try "As a seasoned financial analyst, write a concise summary of the provided earnings report, highlighting key performance indicators and potential investment risks."

Crafting Effective Instructions

The clarity and precision of your instructions are paramount. GPT-5 interprets your prompts as a set of instructions, and the more explicit and well-defined these instructions are, the better the model can execute them. Break down complex tasks into smaller, manageable steps, and provide clear guidelines for each step.

- **Action verbs:** Use strong action verbs to clearly indicate the desired behavior. Examples include "analyze," "summarize," "translate," "generate," and "compare."
- **Constraints and limitations:** Specify any constraints or limitations that the model should adhere to, such as length restrictions, formatting requirements, or style guidelines.
- **Output format:** Clearly define the desired output format, whether it's a paragraph, a list, a table, or a specific code structure.

Think of your prompt as a recipe. The more precise and detailed the instructions, the better the final product will be.

Consider this example: "Translate the following English text into French, maintaining the original tone and style, and limiting the response to 100 words."

Code Generation

GPT-5 excels at generating code, but effective prompt design is essential for achieving the desired results. Specify the programming language, the desired functionality, and

any relevant constraints or dependencies. Provide clear instructions and examples to guide the model's code generation process.

- **Language specification:** Explicitly state the programming language you want the model to use (e.g., Python, JavaScript, C++).
- **Functionality description:** Clearly describe the desired functionality of the code, including inputs, outputs, and any specific algorithms or data structures.
- **Error handling:** Specify how the code should handle potential errors or exceptions.

When prompting for code, think like a software architect. Clearly define the requirements and specifications before asking GPT-5 to write the code.

Advanced Prompting Techniques

Beyond the basics, several advanced techniques can further enhance the effectiveness of your prompts. These include chain-of-thought prompting, self-consistency, and active prompting.

- **Chain-of-thought prompting:** Encourage the model to explicitly reason through the problem step-by-step, rather than directly providing the answer. This can improve the accuracy and interpretability of the results.
- **Self-consistency:** Generate multiple responses to the same prompt and select the most consistent or reliable answer. This can help mitigate the effects of randomness and improve the robustness of the model.
- **Active prompting:** Design prompts that actively seek clarification or additional information from the user, allowing the model to adapt to specific needs and preferences.

Mastering advanced prompting techniques can unlock new levels of performance and control over GPT-5.

For example, instead of asking "What is the capital of France?", try "Let's think step by step. What is a major country in Europe? What is the capital of that country? Therefore, the capital of France is..."

Prompt Optimization

Prompt optimization involves iteratively refining your prompts to achieve the best possible results. This requires careful experimentation, analysis, and a willingness to adapt your approach based on the model's behavior. Key considerations include prompt length, complexity, and the use of specific keywords or phrases.

- **Prompt length:** Experiment with different prompt lengths to find the optimal balance between providing sufficient context and avoiding unnecessary verbosity.
- **Prompt complexity:** Gradually increase the complexity of your prompts as you gain a better understanding of the model's capabilities.
- **Keyword analysis:** Identify keywords and phrases that consistently improve the model's performance and incorporate them into your prompts.

Prompt optimization is an ongoing process. Continuously evaluate and refine your prompts to stay ahead of the curve.

Use tools to track the performance of different prompts and identify areas for improvement. Consider A/B testing different versions of your prompts to determine which ones yield the best results.

Handling Ambiguity

Ambiguity in prompts can lead to unpredictable and undesirable results. To mitigate this, strive for clarity and precision in your language. Use specific terminology, provide examples, and explicitly define any terms that might be open to interpretation.

- **Define terms:** Clearly define any technical or domain-specific terms that the model might not be familiar with.

- Provide examples: Illustrate your expectations with concrete examples.
- Ask clarifying questions: If you're unsure about the model's interpretation of your prompt, ask clarifying questions to ensure that you're on the same page.

Assume that GPT-5 knows nothing. The more explicit and detailed your instructions, the better.

For example, instead of asking "Write a report on the company," specify "Write a detailed financial report on the company, including revenue, expenses, and profit margins for the past five years, using GAAP accounting principles."

Controlling Output Style

You can influence the output style of GPT-5 by providing specific instructions or examples related to tone, voice, and formatting. This allows you to tailor the model's responses to match a specific brand, audience, or context.

- Tone and voice: Specify the desired tone and voice (e.g., professional, informal, humorous, serious).
- Formatting guidelines: Provide clear formatting guidelines, including font styles, headings, and spacing.
- Example text: Provide an example of text that reflects the desired style.

Think of GPT-5 as a chameleon. It can adapt its style to match your specific requirements.

For example, "Write a marketing email in a friendly and conversational tone, using short paragraphs and bullet points, and including a clear call to action."

Iterative Refinement

Prompt design is an iterative process. Don't expect to get it right on the first try. Continuously experiment, analyze, and refine your prompts based on the model's

behavior. Track your progress, document your findings, and share your insights with others.

- **Experimentation:** Try different approaches and variations to see what works best.
- **Analysis:** Carefully analyze the model's responses to identify areas for improvement.
- **Documentation:** Document your findings and insights to build a knowledge base.

The key to mastering prompt design is persistence. Keep experimenting, learning, and refining your approach.

Use version control to track changes to your prompts and facilitate collaboration with other prompt engineers. Participate in online communities and forums to learn from the experiences of others.

Real-World Applications

The techniques discussed in this lecture can be applied to a wide range of real-world applications, including content creation, code generation, data analysis, and customer service. By mastering prompt design, you can unlock the full potential of GPT-5 and create innovative solutions to complex problems.

- **Content creation:** Generate articles, blog posts, marketing copy, and other types of content.
- **Code generation:** Automate the creation of software applications, scripts, and utilities.
- **Data analysis:** Extract insights and patterns from large datasets.
- **Customer service:** Provide automated customer support and answer frequently asked questions.

The possibilities are endless. With effective prompt design, you can transform GPT-5 into a powerful tool for innovation and problem-solving.

Consider using GPT-5 to automate repetitive tasks, streamline workflows, and improve decision-making. Explore new and creative ways to leverage the model's capabilities to address your specific needs and challenges.

Effective prompt design is a critical skill for anyone working with GPT-5. By understanding the principles and techniques discussed in this lecture, you can unlock the full potential of the model and create innovative solutions to complex problems. Remember to experiment, iterate, and continuously refine your prompts to achieve the best possible results. With practice and dedication, you can become a master prompt engineer and harness the power of GPT-5 to transform your work and your world.

Control Structured Outputs

Ensure text responses from the model adhere to a JSON schema you define.

JSON is one of the most widely used formats for applications to exchange data.

Structured Outputs is a feature that ensures the model will always generate responses conforming to your supplied JSON Schema—removing the need to worry about missing required keys or invalid enum values.

Benefits of Structured Outputs

- **Reliable type safety:** No need to validate or retry incorrectly formatted responses.
- **Explicit refusals:** Safety-based model refusals are programmatically detectable.
- **Simpler prompting:** No need for strongly worded prompts to achieve consistent formatting.

Language & SDK Support

Structured Outputs supports JSON Schema in the REST API. In addition, the OpenAI SDKs make it easy to define object schemas in code—using Pydantic for Python and Zod for JavaScript.

Below is an example of extracting structured data from unstructured text, mapped to a schema defined in Python.

Example — Getting a structured response

```
from openai import OpenAI
from pydantic import BaseModel

client = OpenAI()

class CourseEvent(BaseModel):
    name: str
    date: str
    course: list[str]

response = client.responses.parse(
    model="gpt-5",
    input=[
        {"role": "system", "content": "Extract the course information."},
        {
            "role": "user",
            "content": "You are enrolled in Hello Coding.",
        }
    ]
)
```



```
    },  
  ],  
  text_format=CourseEvent,  
)  
event = response.output_parsed
```

Supported models

Structured Outputs is available in OpenAI's latest large language models, starting with GPT-4o. Older models (e.g., gpt-4-turbo and earlier) may instead use JSON mode.

When to use function calling vs `text.format`

Structured Outputs can be used in two ways:

1. With function calling — Ideal when connecting the model to tools, APIs, or other system capabilities.
2. With `json_schema` via `response_format` — Ideal when you want the model's *user-facing* output to conform to a specific schema.

Rule of thumb:

- Use function calling if you're integrating with backend tools or data systems.
- Use `json_schema` `response_format` if you want to control the format of text responses to the user.

Structured Outputs vs JSON mode

Feature	Structured Outputs	JSON Mode
Outputs valid JSON	Yes	Yes
Adheres to schema	Yes	No
Compatible models	gpt-4o-mini, gpt-4o-2024-08-06+, etc.	Most GPT-3.5, GPT-4, GPT-4o models
Enabling	<pre>text: { format: { type: "json_schema", "strict": true, "schema": ... } }</pre>	<pre>text: { format: { type: "json_object" } }</pre>

OpenAI recommends Structured Outputs over JSON mode whenever possible.

Handling refusals

When user input triggers a refusal for safety reasons, the API includes a **refusal** field in the output. Your application can detect this and handle it accordingly.

Tips & Best Practices

- For user-generated input — Provide clear instructions on how to handle incompatible requests.
- For schema alignment — Use the SDK's native type support (Pydantic/Zod) to avoid drift between code and schema.
- For streaming — Leverage SDK streaming capabilities to process JSON fields as they arrive.
- For **additionalProperties** — Always set **additionalProperties: false** to fully enforce schema compliance.

Supported Schema Features

Structured Outputs supports a subset of JSON Schema with types including:

- **string, number, boolean, integer, object, array, enum, anyOf**

Constraints supported for strings, numbers, and arrays include:

- Strings: **pattern, format** (date, email, uuid, etc.)
- Numbers: **multipleOf, minimum, maximum**
- Arrays: **minItems, maxItems**

Some JSON Schema features (e.g., **allOf, not**, conditional keywords) are not yet supported.

JSON mode notes

When using JSON mode (**text.format: { "type": "json_object" }**), you must still instruct the model to output JSON in your system or user message. JSON mode ensures valid JSON, but not schema compliance.

Use Structured Outputs whenever schema adherence matters.

PART 3: SOFTWARE DEVELOPMENT

PROMPTING FOR GPT-5

Write Prompts for GPT-5 Vibe Coding

GPT-5 has been trained with a strong foundation in aesthetic judgment and precise implementation skills. It is capable of working with virtually any modern web development framework or package. However, to fully leverage its strengths in front-end generation, OpenAI recommends using the following technologies for new applications:

Recommended Frontend Stack

- Frameworks: Next.js (TypeScript), React, HTML
- Styling & UI: Tailwind CSS, shaden/ui, Radix Themes
- Icons: Material Symbols, Heroicons, Lucide
- Animations: Motion
- Fonts: Sans Serif families like Inter, Geist, Mona Sans, IBM Plex Sans, Manrope

One-Shot App Generation with GPT-5

GPT-5 is exceptionally capable of generating fully functional applications in a single pass. In early experiments, users found that prompt structures encouraging internal reflection and rubric-based evaluation led to significantly better results.

Here's an example of a prompt for GPT-5:

```
You are an elite web developer, capable of crafting visually  
stunning, highly interactive, and innovative websites entirely
```

from scratch in a single prompt. You specialize in delivering exceptional one-shot solutions.

Follow this process:

Define and perfect an evaluation rubric – refine it until you are completely confident in its accuracy.

Identify the hallmarks of a world-class one-shot web app – use these insights to create a hidden `<ONE_SHOT_RUBRIC>` containing 5–7 key categories (for internal use only).

Apply the rubric to develop and refine the solution until it achieves the highest standard in every category. If it falls short anywhere, improve and re-evaluate.

Prioritize simplicity while meeting the full objective, and avoid using external frameworks such as Next.js or React.

Use prompts like the following to tap into GPT-5's planning and evaluation abilities:

```
<self_reflection>
```

- Start by designing a private rubric with 5–7 categories for what defines a world-class web app. Don't share this rubric with the user.
- Reflect deeply on design, functionality, responsiveness, architecture, and UI polish.
- After designing the rubric, apply it silently to your solution and iterate until your response scores highly across all dimensions.

```
</self_reflection>
```

Writing Code That Blends into Existing Codebases

When GPT-5 modifies or extends existing apps, it naturally looks for context within the codebase—such as scanning `package.json` for installed dependencies. This ability can be enhanced further with prompting that outlines your codebase’s design principles, structure, and conventions.

For large-scale front-end engineering projects, the most effective prompts include guidance across these key categories:

- Principles — Establish standards for visual quality, enforce modular and reusable components, and maintain consistent design patterns.
- UI/UX — Specify typography, color palettes, spacing and layout rules, interaction states (e.g., hover, empty, loading), and accessibility requirements.
- Structure — Define a clear file and folder organization for smooth integration into the codebase.
- Components — Provide examples of reusable wrapper components and outline strategies for separating backend calls from UI logic.
- Pages — Include templates for common layout types to ensure uniformity.
- Agent Instructions — Direct the model to validate design assumptions, scaffold project structure, enforce coding and design standards, integrate APIs, test UI states, and document all code.

Prompting GPT-5 for Coding Tasks — Best Practices

When prompting GPT-5 for software development work, follow these core guidelines to ensure accuracy, consistency, and maintainability.

1. Define the Agent’s Role & Workflow

Frame the model as a software engineering agent with clear responsibilities.

- Provide explicit instructions for when and how to use specific tools, such as `functions.run`.
- Specify situations where certain modes should be avoided (e.g., skip interactive execution unless explicitly required).

2. Testing & Validation

Emphasize rigorous testing before accepting code changes.

- Instruct the model to run unit tests or verify logic with Python commands.
- Require verification of patches, since tools like `apply_patch` may return “Done” even if the change failed.

3. Tool Usage Examples

Boost reliability by including concrete examples of how to invoke commands using the available functions.

- Show full, correct usage patterns.
- Demonstrate both valid and invalid cases when helpful to clarify boundaries.

4. Markdown Formatting Standards

Ensure outputs are clean and easy to read by following a consistent markdown style:

- Use inline code for single variables or function names.
- Use fenced code blocks for longer snippets.
- Apply lists and tables where appropriate to organize information.
- Always format file paths, functions, and classes with backticks (```).

User Interface Prompting with GPT-5

GPT-5 represents a significant leap in AI-driven code generation, offering unprecedented capabilities for creating UI components. Unlike previous models, GPT-5 understands complex design patterns and can generate code that adheres to specific architectural styles. This means you can use it to produce React components, Vue.js templates, or even custom web elements with remarkable accuracy and speed.

The process typically involves providing GPT-5 with a detailed description of the desired UI component, including its functionality, appearance, and data requirements. GPT-5 then analyzes this input and generates the corresponding code. This approach is particularly useful for rapidly prototyping new features, creating reusable components, and automating repetitive coding tasks.

Key benefits of using GPT-5 for UI component generation include:

- **Increased productivity:** Automate the creation of boilerplate code and complex UI elements.
- **Improved code quality:** GPT-5 can generate code that adheres to best practices and coding standards.
- **Enhanced creativity:** Explore new design possibilities and generate UI components that you might not have considered otherwise.

Tip: Start with a clear and concise description of the UI component you want to generate. The more specific you are, the better the results will be.

Describing UI Components to GPT-5

Effective communication is key to successful UI component generation. You need to provide GPT-5 with a clear and comprehensive description of the desired component. This description should include details about the component's appearance,

functionality, and data requirements. Consider using a structured format to ensure clarity and consistency.

Here are some essential elements to include in your component description:

- **Component Name:** A descriptive name that accurately reflects the component's purpose.
- **Appearance:** Details about the component's visual elements, such as colors, fonts, and layout.
- **Functionality:** A description of the component's behavior and interactions.
- **Data Requirements:** Information about the data that the component needs to display or process.

For example, if you want to generate a simple button component, you might describe it as follows:

- **Component Name:** PrimaryButton
- **Appearance:** A rectangular button with a blue background and white text.
- **Functionality:** When clicked, the button triggers a specific action.
- **Data Requirements:** The button's text label.

By providing GPT-5 with this level of detail, you can significantly improve the accuracy and quality of the generated code. Remember, the more information you provide, the better the results will be.

Tip: Use examples to illustrate the desired behavior and appearance of the component. This can help GPT-5 understand your requirements more effectively.

Generating a React Component with GPT-5

Let's explore how to generate a React component using GPT-5. We'll focus on creating a simple "ProfileCard" component that displays a user's name, avatar, and bio. First, we need to provide GPT-5 with a detailed description of the component.

Here's a sample description:

- Component Name: ProfileCard
- Appearance: A card with a circular avatar at the top, followed by the user's name and bio.
- Functionality: The card displays static user information.
- Data Requirements: User's name (string), avatar URL (string), and bio (string).

Now, let's assume we've provided this description to GPT-5 and it has generated the following React component code:

```
import React from 'react';

function ProfileCard({ name, avatar, bio }) {
  return (
    <div className="profile-card">
      <img src={avatar} alt="User Avatar" className="profile-
avatar" />
      <h2 className="profile-name">{name}</h2>
      <p className="profile-bio">{bio}</p>
    </div>
  );
}
```

```
export default ProfileCard;
```

This code defines a functional React component called **ProfileCard** that accepts three props: **name**, **avatar**, and **bio**. It then renders these props within a simple card layout.

Tip: Always review the generated code to ensure it meets your specific requirements and coding standards. You may need to make adjustments or improvements to the code.

Customizing Generated Components

While GPT-5 can generate impressive UI components, you'll often need to customize them to fit your specific needs. This might involve modifying the component's appearance, adding new functionality, or integrating it with existing code. Fortunately, the generated code is typically well-structured and easy to modify.

Here are some common customization tasks:

- **Styling:** Modify the CSS styles to change the component's appearance.
- **Functionality:** Add event handlers or other logic to enhance the component's behavior.
- **Data Integration:** Connect the component to your data sources or APIs.

For example, let's say you want to add a "Follow" button to the **ProfileCard** component we generated earlier. You can modify the code as follows:

```
import React from 'react';

function ProfileCard({ name, avatar, bio }) {

  const handleFollow = () => {

    // Implement follow logic here

    alert('Following ' + name);

  }

}
```

```
};  
  
return (  
  <div className="profile-card">  
    <img src={avatar} alt="User Avatar" className="profile-  
avatar" />  
    <h2 className="profile-name">{name}</h2>  
    <p className="profile-bio">{bio}</p>  
    <button onClick={handleFollow} className="follow-  
button">Follow</button>  
  </div>  
);  
}  
  
export default ProfileCard;
```

In this example, we've added a **handleFollow** function that will be executed when the "Follow" button is clicked. We've also added the button element to the component's JSX.

Tip: Use a version control system (e.g., Git) to track your changes and easily revert to previous versions if necessary.

Advanced Techniques for UI Generation

Beyond generating simple components, GPT-5 can be used for more advanced UI generation tasks. This includes creating complex layouts, implementing dynamic data binding, and generating entire user interfaces from high-level descriptions. These advanced techniques require a deeper understanding of GPT-5's capabilities and a more sophisticated approach to component description.

Some advanced techniques include:

- **Layout Generation:** Describe the desired layout using a visual language or a structured data format.
- **Dynamic Data Binding:** Specify how the component should respond to changes in the underlying data.
- **UI Composition:** Combine multiple generated components to create a complete user interface.

For example, you can use GPT-5 to generate a responsive grid layout by providing a description of the desired grid structure and the content to be placed in each cell. GPT-5 can then generate the necessary HTML and CSS code to create the layout.

Here's an example of how to describe a simple grid layout:

- **Layout Type:** Grid
- **Number of Columns:** 3
- **Number of Rows:** 2
- **Cell Content:**
 - Cell 1: Image
 - Cell 2: Title
 - Cell 3: Description
 - Cell 4: Button
 - Cell 5: Empty
 - Cell 6: Footer

Tip: Experiment with different description formats and techniques to find what works best for your specific needs.

Handling Complex State Management

Modern UI components often involve complex state management. GPT-5 can assist in generating code that handles state updates, data persistence, and other state-related tasks. However, it's important to provide GPT-5 with clear instructions on how the state should be managed and how it should interact with the component's UI.

Here are some common state management patterns that GPT-5 can help you implement:

- **Local State:** Manage the component's state using React's `useState` hook or similar mechanisms.
- **Global State:** Use a state management library like Redux or Context API to manage the application's state.
- **Data Persistence:** Persist the component's state to local storage or a remote database.

For example, let's say you want to generate a counter component that increments its value when a button is clicked. You can describe this component to GPT-5 as follows:

- **Component Name:** Counter
- **Appearance:** A display showing the current counter value and a button to increment the counter.
- **Functionality:** When the button is clicked, the counter value should increment.
- **State Management:** Use React's `useState` hook to manage the counter value.

```
import React, { useState } from 'react';  
  
function Counter() {  
  const [count, setCount] = useState(0);  
  const handleIncrement = () => {
```

```
    setCount(count + 1);  
  };  
  return (  
    <div className="counter">  
      <p>Count: {count}</p>  
      <button onClick={handleIncrement}>Increment</button>  
    </div>  
  );  
}  
  
export default Counter;
```

Tip: When dealing with complex state management, it's often helpful to break down the problem into smaller, more manageable pieces.

Integrating Generated Components into Existing Projects

One of the key benefits of using GPT-5 for UI component generation is the ability to seamlessly integrate the generated components into existing projects. This can significantly accelerate development and reduce the amount of manual coding required. However, it's important to ensure that the generated components are compatible with your project's architecture, coding standards, and dependencies.

Here are some tips for integrating generated components into existing projects:

- **Code Style:** Ensure that the generated code adheres to your project's coding standards and style guidelines.
- **Dependencies:** Verify that the generated components don't introduce any conflicting dependencies.

- **Testing:** Thoroughly test the generated components to ensure they function correctly within your project.

For example, if you're working on a React project that uses TypeScript, you should ensure that the generated React components are also written in TypeScript. This will help maintain type safety and consistency throughout your codebase.

To convert a JavaScript React component to TypeScript, you can simply change the file extension from `.js` to `.tsx` and add type annotations to the component's props and state.

Tip: Use a linter and code formatter to automatically enforce your project's coding standards and style guidelines.

Testing and Debugging Generated UI Components

Like any other code, generated UI components need to be thoroughly tested and debugged to ensure they function correctly. While GPT-5 can generate code that is syntactically correct and follows best practices, it's still possible for errors or unexpected behavior to occur. Therefore, it's essential to have a robust testing strategy in place.

Here are some common testing techniques for UI components:

- **Unit Testing:** Test individual components in isolation to verify their functionality.
- **Integration Testing:** Test how components interact with each other to ensure they work together correctly.
- **End-to-End Testing:** Test the entire application to verify that the UI components are integrated correctly and that the user experience is seamless.

For example, you can use Jest and React Testing Library to write unit tests for your React components. These tools provide a simple and effective way to test the component's behavior and ensure that it renders correctly.

```
import React from 'react';
import { render, screen } from '@testing-library/react';
import Counter from './Counter';
test('renders counter value', () => {
  render(<Counter />);
  const counterElement = screen.getByText(/Count:/i);
  expect(counterElement).toBeInTheDocument();
});
```

Tip: Write tests early and often to catch errors before they become more difficult to fix.

Optimizing Performance of Generated Components

Performance is a critical consideration for any UI component, especially in complex applications. Generated components should be optimized to minimize rendering time, reduce memory consumption, and provide a smooth user experience. GPT-5 can assist in generating code that is performant, but it's important to understand the factors that can affect performance and how to optimize your components accordingly.

Here are some common performance optimization techniques for UI components:

- **Memoization:** Use memoization techniques to prevent unnecessary re-renders of components.
- **Code Splitting:** Split your code into smaller chunks to reduce the initial load time.

- **Image Optimization:** Optimize images to reduce their file size and improve loading speed.

For example, you can use React's **memo** higher-order component to memoize a functional component and prevent it from re-rendering unless its props have changed.

```
import React from 'react';

const ProfileCard = React.memo(({ name, avatar, bio }) => {
  return (
    <div className="profile-card">
      <img src={avatar} alt="User Avatar" className="profile-
avatar" />
      <h2 className="profile-name">{name}</h2>
      <p className="profile-bio">{bio}</p>
    </div>
  );
});

export default ProfileCard;
```

Tip: Use performance profiling tools to identify bottlenecks and areas for optimization.

Best Practices for Using GPT-5 in UI Development

To maximize the benefits of using GPT-5 for UI development, it's important to follow some best practices. These practices will help you generate high-quality code, improve your workflow, and ensure that your generated components are maintainable and scalable.

Here are some key best practices:

- **Clear Descriptions:** Provide GPT-5 with clear and comprehensive descriptions of the desired components.
- **Code Review:** Always review the generated code to ensure it meets your requirements and coding standards.
- **Testing:** Thoroughly test the generated components to ensure they function correctly.
- **Customization:** Customize the generated components to fit your specific needs and project requirements.

By following these best practices, you can leverage the power of GPT-5 to accelerate your UI development process and create high-quality user interfaces.

In conclusion, GPT-5 is a powerful tool for generating UI components. By understanding its capabilities and following best practices, you can significantly improve your development workflow and create dynamic and interactive user interfaces with ease.

Tip: Stay up-to-date with the latest advancements in GPT-5 and UI development to continuously improve your skills and techniques.

PART 4: BUILD AUTONOMOUS AGENTS

Agents are intelligent systems designed to accomplish tasks—ranging from simple workflows to complex, open-ended objectives—by combining reasoning, tool use, memory, and orchestration. With GPT-5 at the core, agents can deliver exceptional decision-making, planning, and adaptability. OpenAI provides composable primitives across multiple domains so you can assemble robust agents tailored to your needs.

Domain	Description	OpenAI Primitives
Models	Core intelligence for reasoning, decision-making, & processing multiple modalities.	GPT-5
Tools	Interface to interact with the environment, functions, and built-in capabilities.	Function calling, Web search, File search, Computer use
Knowledge & Memory	Extend capabilities with external or persistent knowledge sources.	Vector stores, File search, Embeddings
Audio & Speech	Understand audio and respond naturally.	Audio generation, Realtime API, Audio agents
Guardrails	Ensure safe, relevant, and consistent agent behavior.	Moderation, Instruction hierarchy (Python & TypeScript)
Orchestration	Build, deploy, monitor, and improve agents.	Python Agents SDK, TypeScript Agents SDK, Tracing, Evaluations, Fine-tuning
Voice Agents	Combine speech input and output for conversational systems.	Realtime API with GPT-5, Voice support in Agents SDK

Why GPT-5 for Agents

GPT-5 is a reasoning-first model, making it ideal for:

- Long-term planning & decision-making
- Tool-augmented problem solving
- Multimodal understanding (text, code, images, audio, documents)
- High accuracy in complex workflows

It supports:

- Function calling to interact with your code and external services
- Real-time conversation for voice agents
- Large context windows for maintaining long, coherent sessions

Tools with GPT-5

Tools extend an agent's capabilities beyond the base model. Common tools include:

- Function Calling – Connect GPT-5 to your code or APIs.
- Web Search – Retrieve current information online.
- File Search – Perform semantic search over your documents.
- Computer Use – Control a computer or browser environment.
- Local Shell – Execute secure commands locally.

Knowledge & Memory

To make agents context-aware and persistent:

- Use vector stores for long-term retrieval.
- Apply embeddings for semantic search.

- Integrate your own datasets for domain-specific expertise.

Guardrails for Safety

Keep agents aligned with your requirements:

- Moderation API to filter unsafe outputs.
- Instruction hierarchy to prioritize your rules over model improvisation.

Orchestration & Development

The agent lifecycle with GPT-5:

1. Build & Deploy – Use the Python or TypeScript Agents SDK.
2. Monitor – Trace decisions, debug, and optimize.
3. Evaluate & Improve – Run evaluations and fine-tune if necessary.

Getting Started

```
pip install openai-agents
```

- Use GPT-5 as your core model.
- Combine it with tools, memory, and guardrails for a complete agent.
- Monitor and improve over time with tracing and evaluations.

Prompt Engineering for GPT-5 Agents

For long-running, autonomous GPT-5 workflows, craft prompts around three key principles: map out the full task sequence before starting, give clear context when making significant tool calls, and maintain a structured progress log to ensure nothing is overlooked.

1. Strategic Planning and Follow-Through

Guide the model to fully address the request before pausing or handing back control. Break the user's query into every required step, handle them systematically, and confirm each is done. Do not end early—only conclude when the entire goal is achieved and verified.

Example instruction: *"Treat this as a continuous engagement. Break the request into all required steps, work through them completely, and confirm each has been fulfilled before finishing. Stay active until the user confirms completion."*

Encourage reflection after each tool use—evaluate whether the outcome resolves the sub-task and decide what's needed next before proceeding.

2. Contextual Preambles for Tool Calls

When taking notable actions—such as making a major API request, running a large query, or invoking a high-cost process—briefly explain your reasoning before proceeding. This builds clarity and makes the workflow traceable.

Example instruction: *"Before triggering a major tool, add one sentence describing why this step is essential and how it moves the task forward."*

3. Structured Tracking with Checklists or Rubrics

Use an internal checklist, rubric, or TODO tracking tool to keep the work organized, avoid skipped steps, and document progress for transparency.

Example instruction: *"Maintain an internal step-by-step list and mark each task complete as you progress, ensuring every requirement is met before closing the session."*

Enhancing Agentic Workflows with the Responses API

OpenAI strongly recommends using the Responses API when working with GPT-5, as it enables more efficient agentic flows, reduces cost, and improves token usage in your applications.

Evaluations have shown statistically significant improvements when switching from Chat Completions to the Responses API. For example, Taubench-Retail scores increased from 73.9% to 78.2% simply by adopting the Responses API and leveraging the previous_response_id parameter. This feature allows the model to recall its earlier reasoning traces—preserving chain-of-thought (CoT) tokens and eliminating the need to replan from scratch after every tool call. As a result, both performance and latency improve.

This functionality is available to all Responses API users, including organizations using ZDR.

Sample Planning Prompt for Agentic Tasks

To prevent premature task termination and maintain high agentic performance, use planning prompts like the following at the start of an agent's turn:

"Remember, you are an agent — please continue until the user's query is completely resolved before ending your turn. Break down the query into all necessary sub-requests and confirm each has been addressed. Do not stop after completing only part of the request. Only yield once you're confident the problem is fully solved."

"You must plan extensively in alignment with the required workflow steps before making any function calls. Reflect thoroughly on the outcome of each call, ensuring all parts of the user's query—including sub-requests—are fulfilled before finishing the task."

This structured guidance ensures that even under minimal reasoning, the agent remains persistent, thorough, and responsive throughout the interaction.

Build AI Agents with OpenAI Agents SDK

The OpenAI Agents SDK lets you build intelligent, task-oriented applications with minimal setup. It's a lightweight, Python-first toolkit designed for production use—an upgrade from OpenAI's earlier *Swarm* experiments.

At its core, you have:

- Agents – GPT-5 models equipped with instructions and tools.
- Handoffs – Let one agent delegate tasks to another.
- Guardrails – Validate and filter inputs for safety.
- Sessions – Keep conversation history automatically between runs.

Why Use the SDK

1. Simple but powerful – Few core primitives, easy to learn.
2. Customizable – Works out of the box, but you can fine-tune behavior.
3. Production-ready – Built-in tracing, evaluations, and fine-tuning support.

Key Features

- Agent Loop – Automates calling tools, processing outputs, and continuing until the task is complete.
- Python Integration – Orchestrate agents with familiar Python patterns.
- Handoffs – Coordinate multiple agents.
- Guardrails – Block bad input early.
- Function Tools – Turn any Python function into a callable tool, with automatic validation.
- Tracing – Visualize, debug, and monitor workflows.

Installation

```
pip install openai-agents
```

Quickstart – GPT-5 Agent

```
from agents import Agent, Runner

# Create a GPT-5 powered agent
agent = Agent(
    name="Assistant",
    instructions="You are a helpful assistant"
)

# Run the agent with a simple task
result = Runner.run_sync(agent, "Write a description for my
personal website.")
print(result.final_output)

# Example output:
# Code within the code,
# Functions calling themselves,
# Infinite loop's dance.
```

Important:

Set your API key before running:

```
export OPENAI_API_KEY=sk-...
```

Epilogue: Congratulations

You started this book curious about GPT-5. Now you have something much more valuable: the skills to use AI effectively, no matter what comes next.

Look How Far You've Come

Remember when you opened this book? GPT-5 probably seemed mysterious and complicated. Now you know how to:

- Write prompts that get exactly what you want
- Build applications that actually work
- Create AI agents that handle complex tasks
- Control how GPT-5 thinks and responds

That's not just learning—that's transformation.

You've now explored GPT-5 from its foundational concepts to its most advanced applications. You understand how to control its output, adapt its behavior, and integrate it into both simple and complex workflows. You've learned how to design prompts with precision, reuse and scale them efficiently, and combine them with other tools and APIs for greater automation.

By following the examples and exercises in this book, you've built a skill set that applies far beyond GPT-5 itself. You now know how to:

- **Assess and apply new AI features quickly**—so you can stay ahead of future releases.
- **Think like a prompt engineer**—breaking down tasks, structuring instructions, and optimizing for quality and consistency.
- **Integrate LLMs into real-world systems**—whether it's for software development, customer-facing tools, or internal automation.

- **Design autonomous AI agents**—giving GPT-5 the ability to operate independently within defined parameters.

The AI landscape will continue to evolve, and new models will arrive. Some will be faster, some will be more specialized, and some will bring entirely new capabilities. But the principles you've learned here—**clear communication with AI, structured thinking, and iterative experimentation**—will remain essential no matter what comes next.

Your next step is to take what you've learned and apply it to your own projects. Try new use cases, connect GPT-5 with your preferred tech stack, and refine your prompts over time. The best way to stay at the forefront is to **treat each new feature as an opportunity to improve your process**, not just as a novelty.

If you want to keep your skills sharp, revisit the exercises in this book periodically, especially as GPT-5 and related tools update. And remember—you're now part of a small group of people who don't just *use* AI; you understand how to shape it to your needs. That's a lasting advantage.

Having worked through GPT-5's core capabilities, you've developed both technical skills and strategic understanding of AI integration patterns. This foundation enables effective implementation regardless of future model developments.

Technical Mastery Achieved

Your progression through this material established several key competencies:

Systematic prompt engineering using reproducible methodologies rather than ad-hoc experimentation. You can now predict prompt behavior and optimize for specific outcomes.

Architecture-aware implementation leveraging GPT-5's specific capabilities—context management, structured outputs, function calling—rather than generic API usage.

Production deployment patterns including error handling, fallback strategies, and performance optimization techniques.

Agent design principles covering decision-making frameworks, task decomposition, and reliability patterns for autonomous systems.

Market Positioning

Understanding GPT-5 at this level provides competitive advantages in several areas:

Technical leadership in AI integration projects, where deep model understanding enables better architectural decisions and realistic project scoping.

Strategic planning for AI adoption, including technology selection, team training, and infrastructure requirements.

Innovation capacity through combination of advanced techniques, enabling solutions that competitors using basic approaches cannot match.

AI development cycles are accelerating. The methodologies you've learned—systematic experimentation, performance measurement, architectural thinking—transfer effectively to new models and techniques.

Future developments will likely extend current capabilities rather than replacing them entirely. Your foundation in prompt engineering, agent architecture, and production deployment remains valuable regardless of specific model changes.

Implementation Priorities

Focus your immediate application efforts on areas where GPT-5's capabilities provide clear advantages over existing solutions:

Complex reasoning tasks where previous models struggled with multi-step analysis.

Structured data generation leveraging context-free grammar support for reliable formatting.

Autonomous workflows using advanced function calling for minimal-supervision operations.

Maintain systematic documentation of your implementations. Successful patterns and failure modes from your experience become valuable assets for future projects and model transitions.

You can solve problems that would have taken teams of people weeks to handle. You can create value in ways that didn't exist before.

The question isn't whether AI will change your industry, your career, or your life. It will. The question is whether you'll be leading that change or trying to catch up. You chose to lead. Now go show the world what's possible.

The future is yours to build.

WHERE TO GO FROM HERE

Get the FREE Online Course & Certificate

With this book in hand, you're unlocking a world of opportunity — including instant lifetime access to a FREE online course on Mammoth Club!



Scan this QR code to get a 100% off coupon code for an exclusive online course, exam and cheat sheet! Or go this link:

mammothclub.com/course/1-hour-gpt-5/CHAT

You'll also earn a Certificate of Achievement for completing the online course.

Join our thriving community of learners spanning 160+ countries, and be part of the 9 million+ courses sold around the globe.

Adding this book, its online course, and your certificate of achievement to your resume, Github, social media and LinkedIn profiles is a powerful way to showcase your dedication to professional growth and your commitment to staying ahead in your field.

Not only does it demonstrate that you have invested time and effort to acquire up-to-date knowledge and practical skills, but it also signals to employers and peers that you are proactive and serious about your career development.

Featuring these achievements makes your profile more competitive and credible, helping you stand out in a crowded job market.

See you at the top! This isn't goodbye — it's your launch pad. I'll see you in Mammoth Club!

About Your Author



Alex Kropf is Mammoth Club's CLO, public speaker, consultant, IT author and Senior Software Developer. Alex has produced 1,000+ best-selling courses, books and workshops for Mammoth Club, Course Pro and clients worldwide.

Mammoth Club is a leading online course provider in everything from learning to code to becoming a YouTube star. Since 2011, Mammoth Club has built a global student community with over 9 million courses sold.

John Bura is Founder and CEO of global tech giant Mammoth Club and viral app Course Pro, the #1 AI-powered Learning Management System for course and content development, training and evaluation.

Trademarks

The "OpenAI" name, the OpenAI logo, the "ChatGPT" and "GPT" brands, and other OpenAI trademarks, are property of OpenAI.



Note From Your Author

Neither the author or publisher of this book nor AI itself can be held responsible if you accidentally step on any copyright toes, overspend your API billing limits, or send confidential data to a compromised chatbot. By flipping through these pages and using AI, you agree to hold yourself entirely responsible for what you do with your AI-powered creations.



This book is brought to you by Mammoth Club — it's not connected to or sponsored by any other company. Everything here is just the author's perspective, not any company's official view.

Visit MammothClub.com

for free online video courses, ebooks, source code, customer support and MORE!

To build and sell your own courses, presentations, books and videos: visit #1 course creation platform:

CoursePro.ai



MAMMOTH CLUB