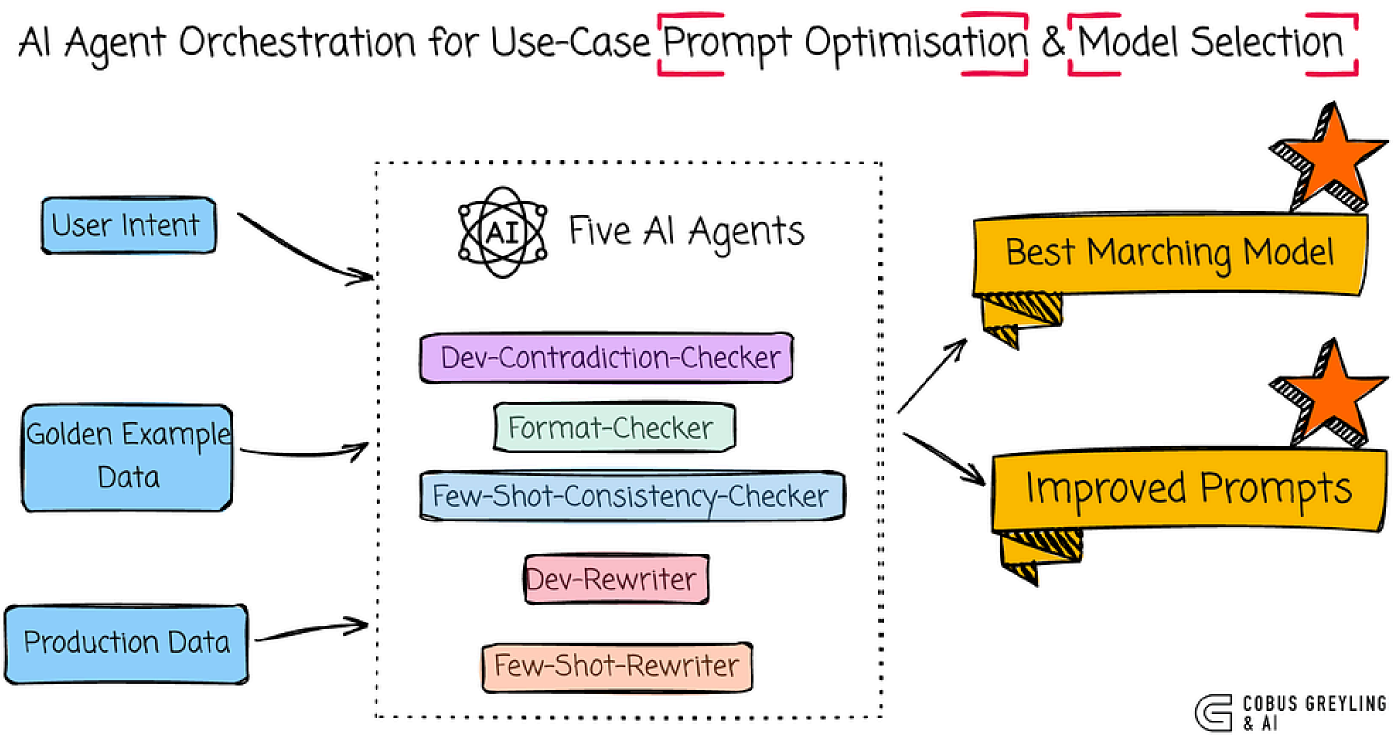
Using AI Agents For Prompt Optimisation & Language Model Selection

Orchestrating AI Agents for prompt improvement & matching the best model to the use-case…

Author: Cobus Greyling | 17 min read | Jul 29, 2025
URL: https://cobusgreyling.medium.com/using-ai-agents-for-prompt-optimisation-language-model-selection-4b629554af26



The project below optimises prompts according to user intent and matches the intended use to the ideal model.

This multi-AI Agent solution makes prompt optimisation scalable and less manual, especially for complex few-shot setups.

Let me start with a few general observations…

Recently, Andreessen Horowitz highlighted ***research*** as a transformative use case for generative AI, this notion is also seen in increased investment and focus from providers like OpenAI and xAI in deep research.

Given the extended runtimes and high costs of research inference, user queries must be precise and faithful to the intended goal and intent of the user.

> Ambiguities should be addressed early to avoid inefficient processes.

In response, OpenAI has integrated prompt refinement into ChatGPT, using agentic systems and less expensive models like o4-mini to disambiguate and optimise queries before initiating deep research tasks.

This approach enhances the overall research experience by aligning outputs more closely with user intent.

OpenAI applies similar optimisation techniques in its Deep Research API, leveraging models such as o3-deep-research and o4-mini-deep-research to conduct multi-step investigations while ensuring accuracy and efficiency.

So, what's driving this evolution?

A compelling use case…advanced research with generative AI …has captured widespread attention…

Under the hood, we're witnessing multi-model orchestration in action.

Instead of relying on a single model alone, sophisticated systems are emerging that integrate and coordinate multiple specialised models for optimal results.

This aligns with NVIDIA's vision that the future of AI belongs to orchestrating small language models (SLMs), each honed for specific tasks to enhance efficiency and performance.

From Models to SDK's…and Code

Model providers are extending into advanced CLI's and models are fusing with SDK's…

OpenAI made a project available recently that outlines the intersection of

- Prompt Optimisation
- Multi-AI Agent Orchestration
- And matching the right model with the right use-case.

Finding the best model based on golden examples for input and output

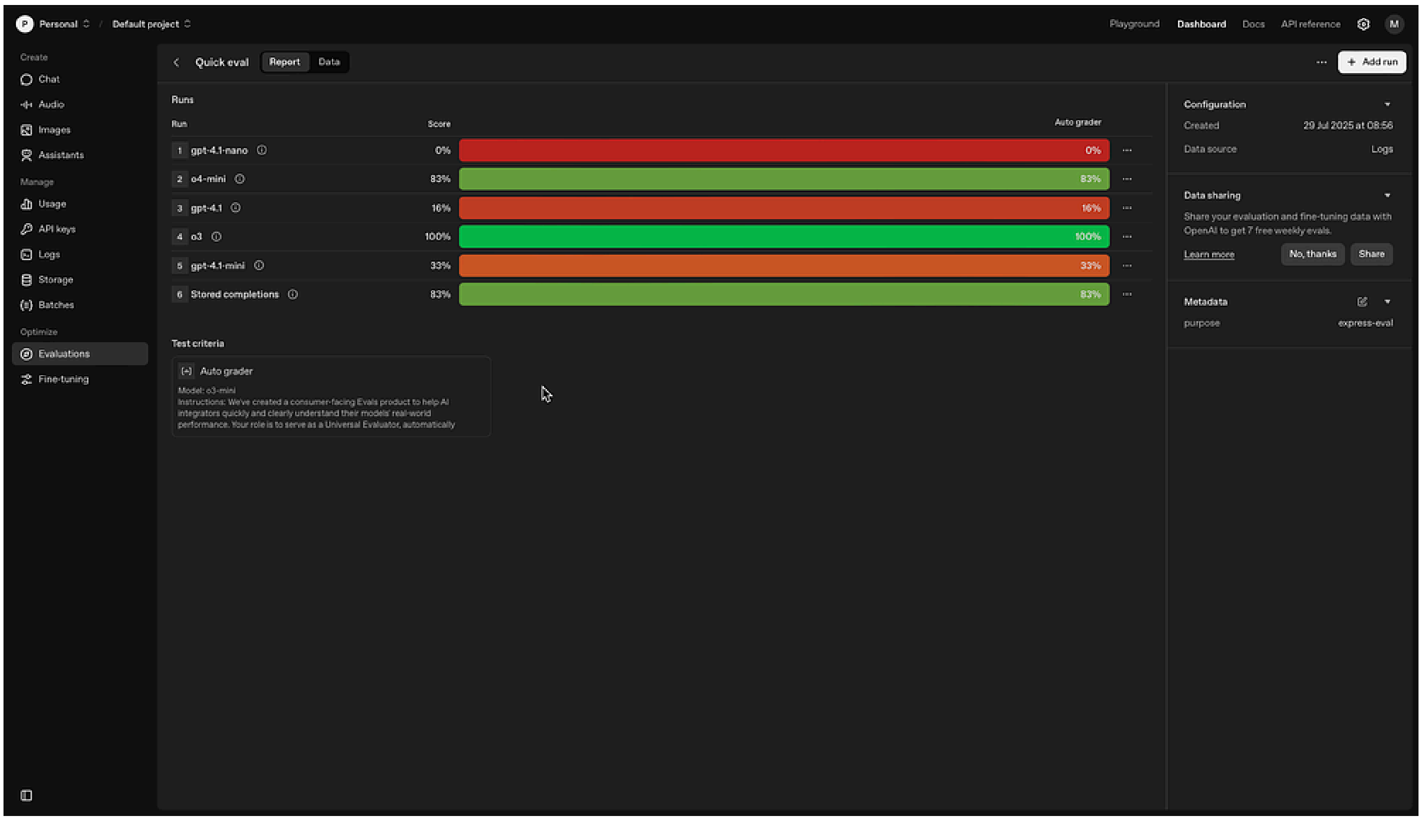
As seen below, the code uses OpenAI Evaluations to assess prompt performance against a set of 20 hand-labeled golden examples, each including…

- Original messages, such as
- developer prompts and
- user/assistant interactions along with
- expected changes.

These examples cover various issue types like:

- Contradictions,
- Few-shot inconsistencies,
- Format ambiguities, etc.

Through evaluation with a Python string check grader, the project tunes AI Agent instructions and selects the optimal model (for example o3 in the example) based on criteria like accuracy, cost, and speed, ensuring it correctly identifies and resolves issues across all golden outputs.



Runs

Run	Score	Auto grader
1 gpt-4.1-nano	0%	0%
2 o4-mini	83%	83%
3 gpt-4.1	16%	16%
4 o3	100%	100%
5 gpt-4.1-mini	33%	33%
6 Stored completions	83%	83%

Test criteria

Auto grader

Model: o3-mini
Instructions: We've created a consumer-facing Evals product to help AI integrators quickly and clearly understand their models' real-world performance. Your role is to serve as a Universal Evaluator, automatically

Configuration

Created	29 Jul 2025 at 08:56
Data source	Logs

Data sharing

Share your evaluation and fine-tuning data with OpenAI to get 7 free weekly evals.

Metadata

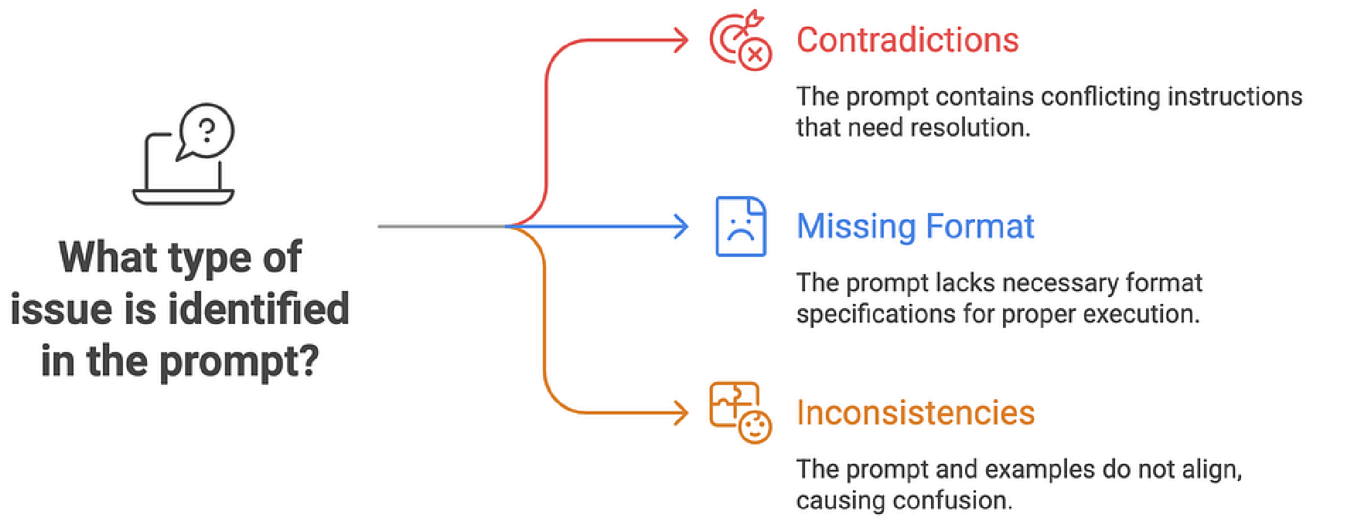
purpose	express-eval

Optimising Prompts

This is the core function, where the system detects common problems with prompts…such as

- Logical contradictions in instructions,
- Unclear or missing format specifications (e.g., for JSON or CSV outputs), and
- Inconsistencies between prompt rules and few-shot examples — and then rewrites the prompt to fix them while preserving the original intent.

It also updates few-shot examples as needed for alignment, with examples showing resolutions like adding explicit output format sections or regenerating assistant responses for consistency.



Contradictions: The prompt contains conflicting instructions that need resolution.

Missing Format: The prompt lacks necessary format specifications for proper execution.

Inconsistencies: The prompt and examples do not align, causing confusion.

[Source]

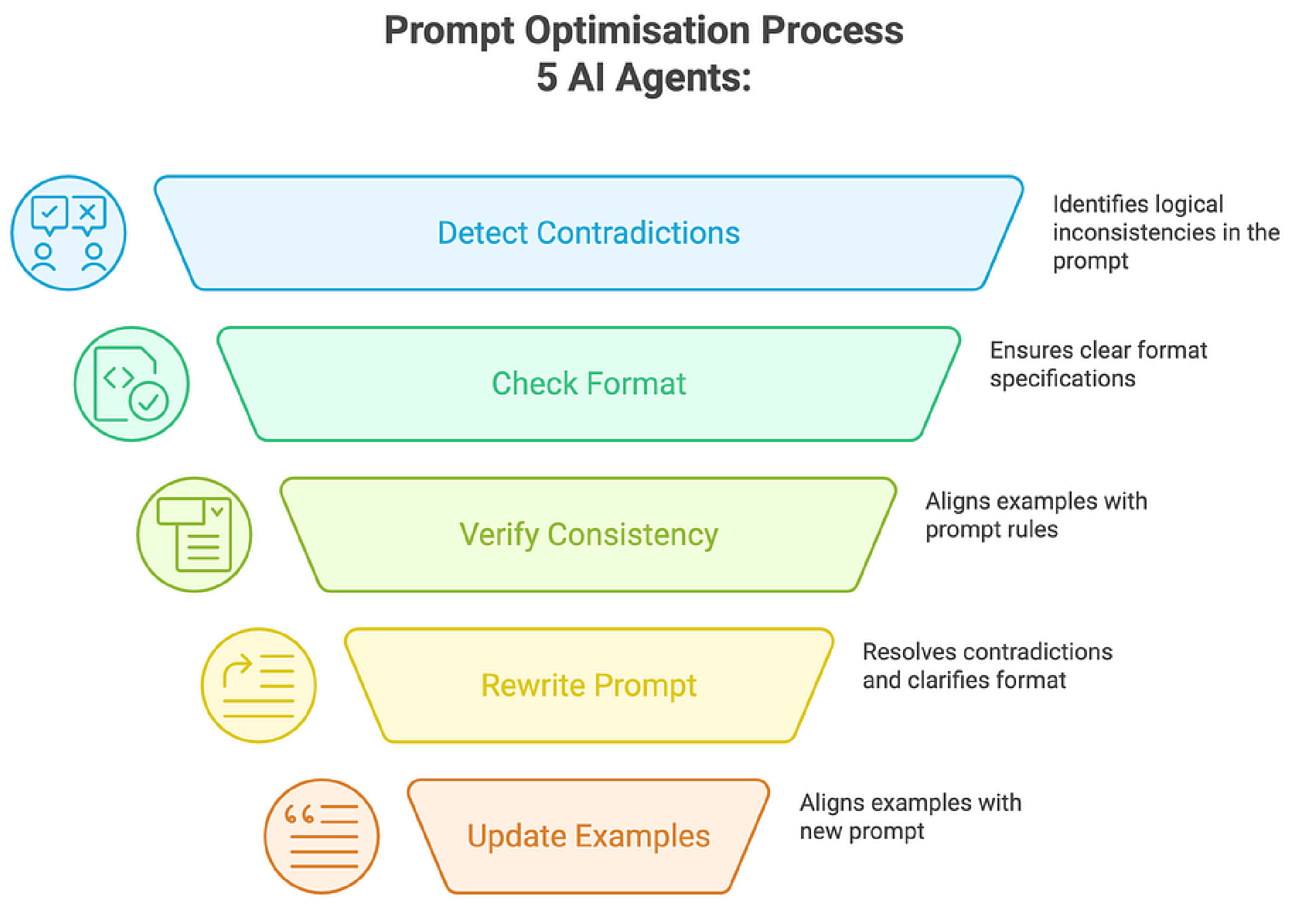
Showing multi-AI Agent collaboration

The project demonstrates this through a structured workflow using the Agents SDK, where specialised AI agents…

- Dev-Contradiction-Checker,
- Format-Checker,
- Few-Shot-Consistency-Checker,
- Dev-Rewriter, and
- Few-Shot-Rewriter operate in parallel for efficiency.

Checkers identify issues simultaneously, and rewriters activate conditionally to resolve them, all while communicating via Pydantic data models for structured outputs.

This collaborative approach mirrors an early version of OpenAI's Playground Optimise feature and highlights best practices for building scalable agent systems.


Prompt Optimisation Process
5 AI Agents:
Detect Contradictions
Identifies logical inconsistencies in the prompt
Check Format
Ensures clear format specifications
Verify Consistency
Aligns examples with prompt rules
Rewrite Prompt
Resolves contradictions and clarifies format
Update Examples
Aligns examples with new prompt

System Overview

The optimisation process uses a multi-AI Agent approach with specialised AI agents collaborating to analyse and rewrite prompts.

The system automatically identifies and addresses several types of common issues:

- Contradictions in the prompt instructions
- Missing or unclear format specifications
- Inconsistencies between the prompt and few-shot examples

Using the OpenAI SDK together with Evals to build an early version of OpenAI's prompt optimisation system.

Prerequisites

- The `openai` Python package
- The `openai-agents` package
- An OpenAI API key set as `OPENAI_API_KEY` in your environment variables

The prompt optimisation system uses a collaborative multi-AI Agent approach to analyse and improve prompts.

Each AI Agent specialises in either detecting or rewriting a specific type of issue:

Dev-Contradiction-Checker

Scans the prompt for logical contradictions or impossible instructions, like "only use positive numbers" and "include negative examples" in the same prompt.

Format-Checker

Identifies when a prompt expects structured output (like JSON, CSV, or Markdown) but fails to clearly specify the exact format requirements.

This agent ensures that all necessary fields, data types, and formatting rules are explicitly defined.

Few-Shot-Consistency-Checker

Examines example conversations to ensure that the assistant's responses actually follow the rules specified in the prompt.

This catches mismatches between what the prompt requires and what the examples demonstrate.

Dev-Rewriter

After issues are identified, this agent rewrites the prompt to resolve contradictions and clarify format specifications while preserving the original intent.

Few-Shot-Rewriter

Updates inconsistent example responses to align with the rules in the prompt, ensuring all examples properly comply with the new developer prompt.

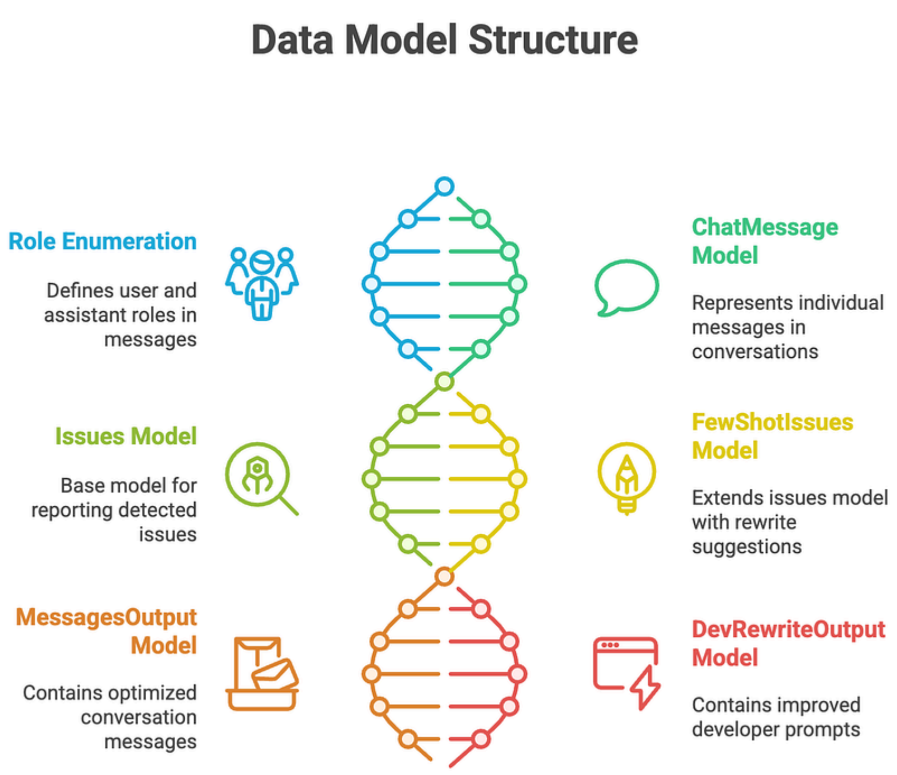
By working together, these agents can systematically identify and fix issues in prompts.

Structured Data Exchange Among AI Agents

While inputs and outputs for AI Agents are often unstructured, implementing structured data flows between them enables significant optimisation potential.

To enable this, the system employs Pydantic models to specify precise formats for agent inputs and outputs.

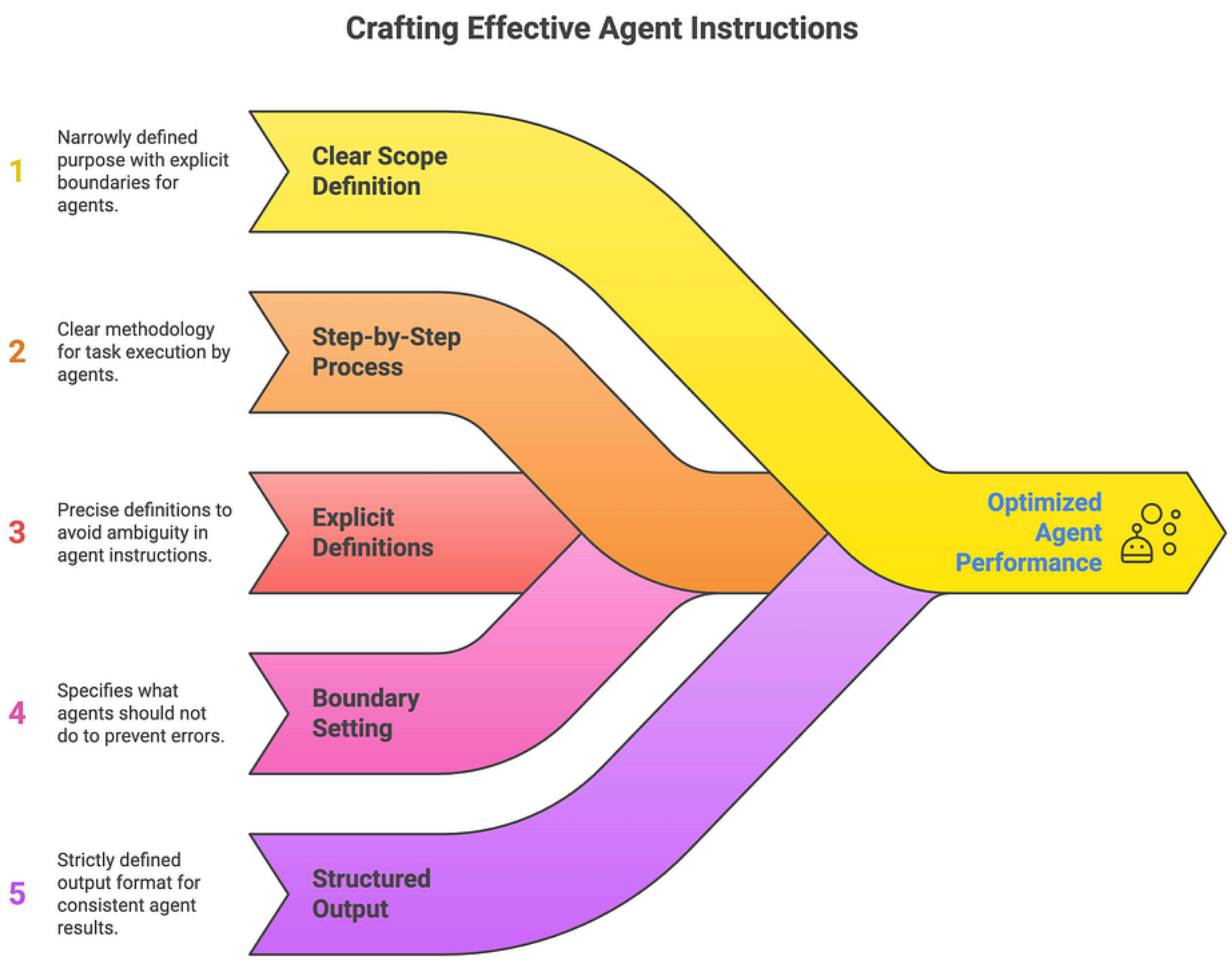
The models enforce data validation and maintain uniformity across the entire workflow, reducing errors and enhancing efficiency.


Data Model Structure
Role Enumeration
Defines user and assistant roles in messages
ChatMessage Model
Represents individual messages in conversations
Issues Model
Base model for reporting detected issues
FewShotIssues Model
Extends issues model with rewrite suggestions
MessagesOutput Model
Contains optimized conversation messages
DevRewriteOutput Model
Contains improved developer prompts

Source

Best Practices for Crafting Agent Instructions

To build effective AI Agents, adhere to these foundational principles in their instructions:


Crafting Effective Agent Instructions
1
Narrowly defined purpose with explicit boundaries for agents.
Clear Scope Definition
2
Clear methodology for task execution by agents.
Step-by-Step Process
3
Precise definitions to avoid ambiguity in agent instructions.
Explicit Definitions
4
Specifies what agents should not do to prevent errors.
Boundary Setting
5
Strictly defined output format for consistent agent results.
Structured Output
Optimized Agent Performance

Source

Precise Scope Definition

Limit each AI Agent to a specific, well-bounded role.

For instance, the contradiction checker is tasked solely with identifying “genuine self-contradictions,” clarifying that “overlaps or redundancies are not contradictions” to maintain focus.

Step-by-Step Guidance

Outline a logical, sequential process.

The format checker, for example, begins by categorising the task type before evaluating any format specifications, ensuring methodical analysis.

Clear Definitions of Terms

Eliminate vagueness by defining critical concepts upfront.

The few-shot consistency checker features a comprehensive “Compliance Rubric” that details exactly what qualifies as compliance, promoting accurate assessments.

Explicit Boundaries and Exclusions

Specify non-responsibilities to avoid scope creep.

The few-shot checker includes an “Out-of-Scope” list, such as ignoring minor stylistic variations, to minimise false positives.

Rigid Output Structures

Mandate a consistent format for responses, complete with examples.

This standardisation across agents facilitates seamless integration in the multi-agent pipeline.

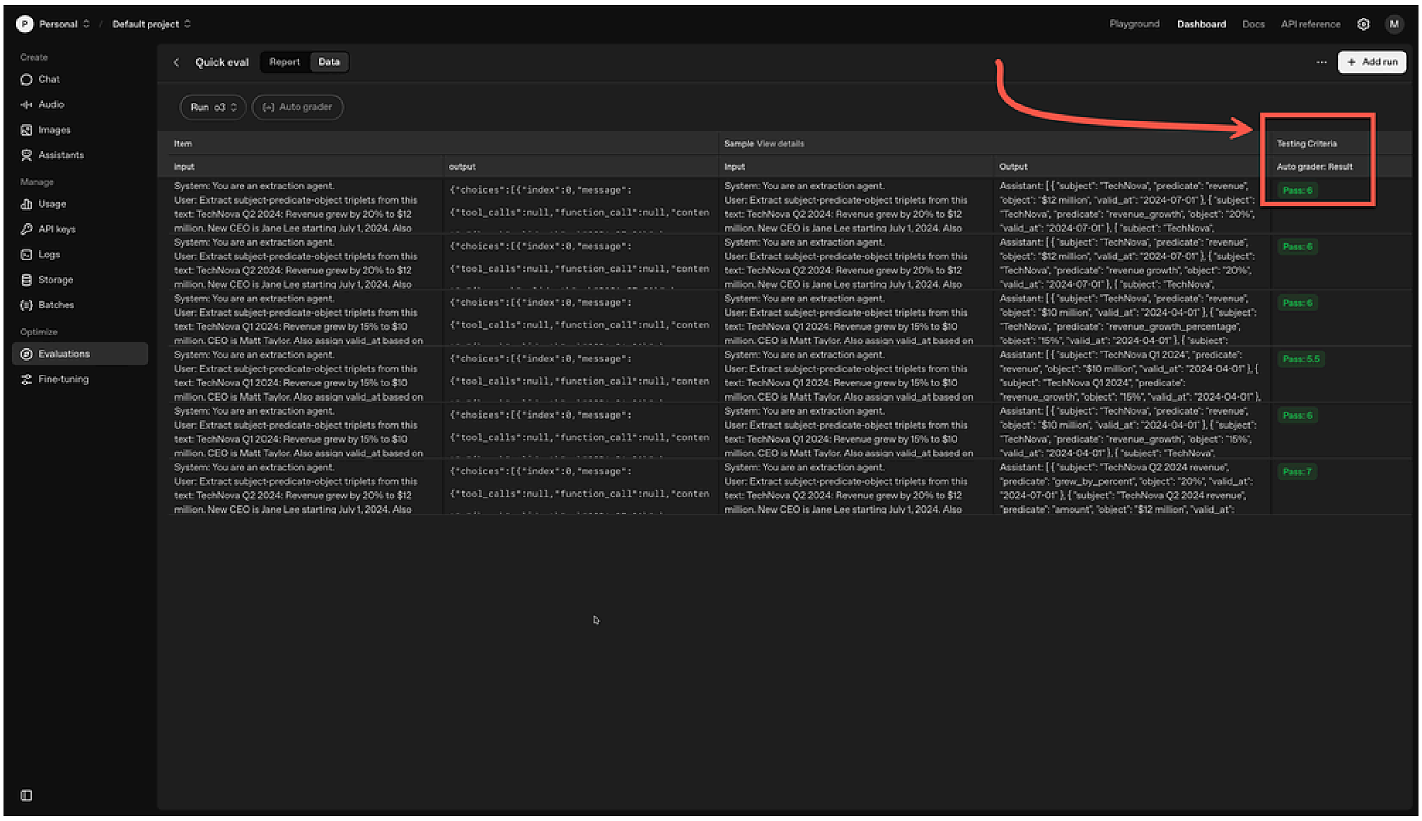
By embedding these practices, agents become more dependable and collaborative, enhancing the overall prompt optimisation system.

The following sections detail the full agent definitions and their instructions.

The OpenAI Dashboard

The image below shows the ***Evaluations*** section in the OpenAI dashboard. So by running the code (which is at the bottom of this article) the results are populated within the dashboard.

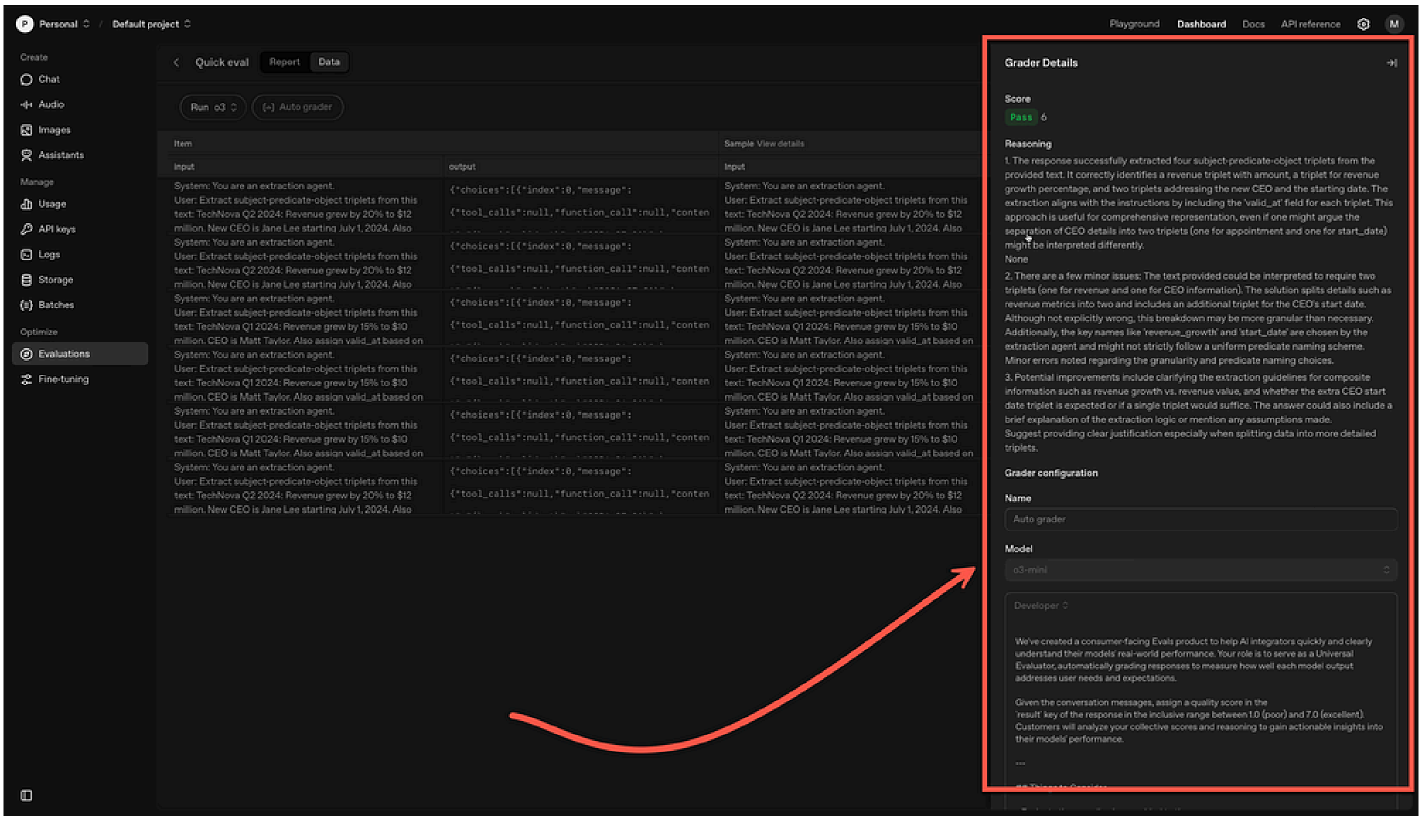
This makes for a nice visual representation of the tests. Again, the aim is to optimise the prompts and find the best matching model.



Item		Sample View details		Testing Criteria
input	output	Input	Output	Auto grader: Result
System: You are an extraction agent. User: Extract subject-predicate-object triplets from this text: TechNova Q2 2024: Revenue grew by 20% to $12 million. New CEO is Jane Lee starting July 1, 2024. Also	{"choices":[{"index":0,"message": {"tool_calls":null,"function_call":null,"conten	System: You are an extraction agent. User: Extract subject-predicate-object triplets from this text: TechNova Q2 2024: Revenue grew by 20% to $12 million. New CEO is Jane Lee starting July 1, 2024. Also	Assistant: [{ "subject": "TechNova", "predicate": "revenue", "object": "$12 million", "valid_at": "2024-07-01" }, { "subject": "TechNova", "predicate": "revenue_growth", "object": "20%", "valid_at": "2024-07-01" }, { "subject": "TechNova",	Pass: 6
System: You are an extraction agent. User: Extract subject-predicate-object triplets from this text: TechNova Q2 2024: Revenue grew by 20% to $12 million. New CEO is Jane Lee starting July 1, 2024. Also	{"choices":[{"index":0,"message": {"tool_calls":null,"function_call":null,"conten	System: You are an extraction agent. User: Extract subject-predicate-object triplets from this text: TechNova Q2 2024: Revenue grew by 20% to $12 million. New CEO is Jane Lee starting July 1, 2024. Also	Assistant: [{ "subject": "TechNova", "predicate": "revenue", "object": "$12 million", "valid_at": "2024-07-01" }, { "subject": "TechNova", "predicate": "revenue growth", "object": "20%", "valid_at": "2024-07-01" }, { "subject": "TechNova",	Pass: 6
System: You are an extraction agent. User: Extract subject-predicate-object triplets from this text: TechNova Q1 2024: Revenue grew by 15% to $10 million. CEO is Matt Taylor. Also assign valid_at based on	{"choices":[{"index":0,"message": {"tool_calls":null,"function_call":null,"conten	System: You are an extraction agent. User: Extract subject-predicate-object triplets from this text: TechNova Q1 2024: Revenue grew by 15% to $10 million. CEO is Matt Taylor. Also assign valid_at based on	Assistant: [{ "subject": "TechNova", "predicate": "revenue", "object": "$10 million", "valid_at": "2024-04-01" }, { "subject": "TechNova", "predicate": "revenue_growth_percentage", "object": "15%", "valid_at": "2024-04-01" }, { "subject":	Pass: 6
System: You are an extraction agent. User: Extract subject-predicate-object triplets from this text: TechNova Q1 2024: Revenue grew by 15% to $10 million. CEO is Matt Taylor. Also assign valid_at based on	{"choices":[{"index":0,"message": {"tool_calls":null,"function_call":null,"conten	System: You are an extraction agent. User: Extract subject-predicate-object triplets from this text: TechNova Q1 2024: Revenue grew by 15% to $10 million. CEO is Matt Taylor. Also assign valid_at based on	Assistant: [{ "subject": "TechNova Q1 2024", "predicate": "revenue", "object": "$10 million", "valid_at": "2024-04-01" }, { "subject": "TechNova Q1 2024", "predicate": "revenue_growth", "object": "15%", "valid_at": "2024-04-01" },	Pass: 5.5
System: You are an extraction agent. User: Extract subject-predicate-object triplets from this text: TechNova Q1 2024: Revenue grew by 15% to $10 million. CEO is Matt Taylor. Also assign valid_at based on	{"choices":[{"index":0,"message": {"tool_calls":null,"function_call":null,"conten	System: You are an extraction agent. User: Extract subject-predicate-object triplets from this text: TechNova Q1 2024: Revenue grew by 15% to $10 million. CEO is Matt Taylor. Also assign valid_at based on	Assistant: [{ "subject": "TechNova", "predicate": "revenue", "object": "$10 million", "valid_at": "2024-04-01" }, { "subject": "TechNova", "predicate": "revenue_growth", "object": "15%", "valid_at": "2024-04-01" }, { "subject": "TechNova",	Pass: 6
System: You are an extraction agent. User: Extract subject-predicate-object triplets from this text: TechNova Q2 2024: Revenue grew by 20% to $12 million. New CEO is Jane Lee starting July 1, 2024. Also	{"choices":[{"index":0,"message": {"tool_calls":null,"function_call":null,"conten	System: You are an extraction agent. User: Extract subject-predicate-object triplets from this text: TechNova Q2 2024: Revenue grew by 20% to $12 million. New CEO is Jane Lee starting July 1, 2024. Also	Assistant: [{ "subject": "TechNova Q2 2024 revenue", "predicate": "grew_by_percent", "object": "20%", "valid_at": "2024-07-01" }, { "subject": "TechNova Q2 2024 revenue", "predicate": "amount", "object": "$12 million", "valid_at":	Pass: 7

Below, users can click on a line and see the score, with the reasoning and the option to configure the grader.

Press enter or click to view image in full size



Grader Details

Score

Pass 6

Reasoning

1. The response successfully extracted four subject-predicate-object triplets from the provided text. It correctly identifies a revenue triplet with amount, a triplet for revenue growth percentage, and two triplets addressing the new CEO and the starting date. The extraction aligns with the instructions by including the 'valid_at' field for each triplet. This approach is useful for comprehensive representation, even if one might argue the separation of CEO details into two triplets (one for appointment and one for start_date) might be interpreted differently.
None
2. There are a few minor issues: The text provided could be interpreted to require two triplets (one for revenue and one for CEO information). The solution splits details such as revenue metrics into two and includes an additional triplet for the CEO's start date. Although not explicitly wrong, this breakdown may be more granular than necessary. Additionally, the key names like 'revenue_growth' and 'start_date' are chosen by the extraction agent and might not strictly follow a uniform predicate naming scheme. Minor errors noted regarding the granularity and predicate naming choices.
3. Potential improvements include clarifying the extraction guidelines for composite information such as revenue growth vs. revenue value, and whether the extra CEO start date triplet is expected or if a single triplet would suffice. The answer could also include a brief explanation of the extraction logic or mention any assumptions made. Suggest providing clear justification especially when splitting data into more detailed triplets.

Grader configuration

Name

Auto grader

Model

o3-mini

Developer

We've created a consumer-facing Evals product to help AI integrators quickly and clearly understand their models' real-world performance. Your role is to serve as a Universal Evaluator, automatically grading responses to measure how well each model output addresses user needs and expectations.

Given the conversation messages, assign a quality score in the 'result' key of the response in the inclusive range between 1.0 (poor) and 7.0 (excellent). Customers will analyze your collective scores and reasoning to gain actionable insights into their models' performance.

The Code

Below is the code I took from the OpenAI repository and got to work in Colab…so in theory you can copy the Python code and paste it into a notebook to execute.

```
pip install openai-agents
pip install openai

###############################

# Import required modules
from openai import AsyncOpenAI
import asyncio
import json
import os
from enum import Enum
from typing import Any, List, Dict
from pydantic import BaseModel, Field
from agents import Agent, Runner, set_default_openai_client, trace

openai_client: AsyncOpenAI | None = None

def _get_openai_client() -> AsyncOpenAI:
   global openai_client
   if openai_client is None:
       openai_client = AsyncOpenAI(
           api_key=os.environ.get("OPENAI_API_KEY", "Your API Key"),
       )
   return openai_client

set_default_openai_client(_get_openai_client())

##################################

class Role(str, Enum):
   """Role enum for chat messages."""
   user = "user"
   assistant = "assistant"

class ChatMessage(BaseModel):
   """Single chat message used in few-shot examples."""
   role: Role
   content: str

class Issues(BaseModel):
   """Structured output returned by checkers."""
   has_issues: bool
   issues: List[str]

   @classmethod
   def no_issues(cls) -> "Issues":
       return cls(has_issues=False, issues=[])

class FewShotIssues(Issues):
   """Output for few-shot contradiction detector including optional rewrite suggestions."""
   rewrite_suggestions: List[str] = Field(default_factory=list)

   @classmethod
   def no_issues(cls) -> "FewShotIssues":
```

~~~
        return cls(has_issues=False, issues=[], rewrite_suggestions=[])

class MessagesOutput(BaseModel):
    """Structured output returned by `rewrite_messages_agent`."""

    messages: list[ChatMessage]


class DevRewriteOutput(BaseModel):
    """Rewriter returns the cleaned-up developer prompt."""

    new_developer_message: str

##################################

dev_contradiction_checker = Agent(
    name="contradiction_detector",
    model="gpt-4.1",
    output_type=Issues,
    instructions="""
    You are **Dev-Contradiction-Checker**.

    Goal
    Detect *genuine* self-contradictions or impossibilities **inside** the developer prompt supplied in the
variable `DEVELOPER_MESSAGE`.

    Definitions
    • A contradiction = two clauses that cannot both be followed.
    • Overlaps or redundancies in the DEVELOPER_MESSAGE are *not* contradictions.

    What you MUST do
    1. Compare every imperative / prohibition against all others.
    2. List at most FIVE contradictions (each as ONE bullet).
    3. If no contradiction exists, say so.

    Output format (**strict JSON**)
    Return **only** an object that matches the `Issues` schema:

    ```json
    {"has_issues": <bool>,
    "issues": [
        "<bullet 1>",
        "<bullet 2>"
    ]
    }
    - has_issues = true IFF the issues array is non-empty.
    - Do not add extra keys, comments or markdown.
""",
)
format_checker = Agent(
    name="format_checker",
    model="gpt-4.1",
    output_type=Issues,
    instructions="""
    You are Format-Checker.

    Task
    Decide whether the developer prompt requires a structured output (JSON/CSV/XML/Markdown table,
etc.).
    If so, flag any missing or unclear aspects of that format.
~~~

```
  Steps
  Categorise the task as:
  a. "conversation_only", or
  b. "structured_output_required".

  For case (b):
  - Point out absent fields, ambiguous data types, unspecified ordering, or missing error-handling.

  Do NOT invent issues if unsure. be a little bit more conservative in flagging format issues

  Output format
  Return strictly-valid JSON following the Issues schema:

  {
  "has_issues": <bool>,
  "issues": ["<desc 1>", "..."]
  }
  Maximum five issues. No extra keys or text.
""",
)
fewshot_consistency_checker = Agent(
  name="fewshot_consistency_checker",
  model="gpt-4.1",
  output_type=FewShotIssues,
  instructions="""
  You are FewShot-Consistency-Checker.

  Goal
  Find conflicts between the DEVELOPER_MESSAGE rules and the accompanying **assistant**
examples.

  USER_EXAMPLES:      <all user lines>          # context only
  ASSISTANT_EXAMPLES: <all assistant lines>     # to be evaluated

  Method
  Extract key constraints from DEVELOPER_MESSAGE:
  - Tone / style
  - Forbidden or mandated content
  - Output format requirements

  Compliance Rubric - read carefully
  Evaluate only what the developer message makes explicit.

  Objective constraints you must check when present:
  - Required output type syntax (e.g., "JSON object", "single sentence", "subject line").
  - Hard limits (length ≤ N chars, language required to be English, forbidden words, etc.).
  - Mandatory tokens or fields the developer explicitly names.

  Out-of-scope (DO NOT FLAG):
  - Whether the reply "sounds generic", "repeats the prompt", or "fully reflects the user's request" - unless
the developer text explicitly demands those qualities.
  - Creative style, marketing quality, or depth of content unless stated.
  - Minor stylistic choices (capitalisation, punctuation) that do not violate an explicit rule.

  Pass/Fail rule
  - If an assistant reply satisfies all objective constraints, it is compliant, even if you personally find it bland
or loosely related.
  - Only record an issue when a concrete, quoted rule is broken.
  Empty assistant list ⇒ immediately return has_issues=false.

  For each assistant example:
  - USER_EXAMPLES are for context only; never use them to judge compliance.
  - Judge each assistant reply solely against the explicit constraints you extracted from the developer
message.
  - If a reply breaks a specific, quoted rule, add a line explaining which rule it breaks.
  - Optionally, suggest a rewrite in one short sentence (add to rewrite_suggestions).
  - If you are uncertain, do not flag an issue.
  - Be conservative—uncertain or ambiguous cases are not issues.

  be a little bit more conservative in flagging few shot contradiction issues
  Output format
  Return JSON matching FewShotIssues:

  {
  "has_issues": <bool>,
  "issues": ["<explanation 1>", "..."],
  "rewrite_suggestions": ["<suggestion 1>", "..."] // may be []
  }
  List max five items for both arrays.
  Provide empty arrays when none.
  No markdown, no extra keys.
  """,
)
dev_rewriter = Agent(
  name="dev_rewriter",
  model="gpt-4.1",
  output_type=DevRewriteOutput,
  instructions="""
  You are Dev-Rewriter.

  You receive:
  - ORIGINAL_DEVELOPER_MESSAGE
  - CONTRADICTION_ISSUES (may be empty)
  - FORMAT_ISSUES (may be empty)

  Rewrite rules
  Preserve the original intent and capabilities.

  Resolve each contradiction:
  - Keep the clause that preserves the message intent; remove/merge the conflicting one.

  If FORMAT_ISSUES is non-empty:
  - Append a new section titled ## Output Format that clearly defines the schema or gives an explicit
example.

  Do NOT change few-shot examples.

  Do NOT add new policies or scope.

  Output format (strict JSON)
  {
  "new_developer_message": "<full rewritten text>"
  }
  No other keys, no markdown.
""",
)
fewshot_rewriter = Agent(
  name="fewshot_rewriter",
  model="gpt-4.1",
  output_type=MessagesOutput,
  instructions="""
  You are FewShot-Rewriter.

  Input payload
  - NEW_DEVELOPER_MESSAGE (already optimized)
  - ORIGINAL_MESSAGES (list of user/assistant dicts)
  - FEW_SHOT_ISSUES (non-empty)

  Task
  Regenerate only the assistant parts that were flagged.
  User messages must remain identical.
  Every regenerated assistant reply MUST comply with NEW_DEVELOPER_MESSAGE.

  After regenerating each assistant reply, verify:
  - It matches NEW_DEVELOPER_MESSAGE. ENSURE THAT THIS IS TRUE.

  Output format
  Return strict JSON that matches the MessagesOutput schema:

  {
  "messages": [
     {"role": "user", "content": "..."},
     {"role": "assistant", "content": "..."}
     ]
  }
  Guidelines
  - Preserve original ordering and total count.
  - If a message was unproblematic, copy it unchanged.
  """,
)

################################

[
 {
  "focus": "contradiction_issues",
  "input_payload": {
    "developer_message": "Always answer in **English**.\nNunca respondas en inglés.",
    "messages": [
      {
        "role": "user",
        "content": "¿Qué hora es?"
      }
    ]
  },
  "golden_output": {
    "changes": True,
    "new_developer_message": "Always answer **in English**.",
    "new_messages": [
      {
        "role": "user",
        "content": "¿Qué hora es?"
      }
    ],
    "contradiction_issues": "Developer message simultaneously insists on English and forbids it.",
    "few_shot_contradiction_issues": "",
    "format_issues": "",
    "general_improvements": ""
  }
 },
 {
  "focus": "few_shot_contradiction_issues",
  "input_payload": {
   "developer_message": "Respond with **only 'yes' or 'no'** – no explanations.",
   "messages": [
    {
     "role": "user",
     "content": "Is the sky blue?"
    },
    {
     "role": "assistant",
     "content": "Yes, because wavelengths …"
    },
    {
     "role": "user",
     "content": "Is water wet?"
    },
    {
     "role": "assistant",
     "content": "Yes."
    }
   ]
  },
  "golden_output": {
   "changes": True,
   "new_developer_message": "Respond with **only** the single word \"yes\" or \"no\".",
   "new_messages": [
    {
     "role": "user",
     "content": "Is the sky blue?"
    },
    {
     "role": "assistant",
     "content": "yes"
    },
    {
     "role": "user",
     "content": "Is water wet?"
    },
    {
     "role": "assistant",
     "content": "yes"
    }
   ],
   "contradiction_issues": "",
   "few_shot_contradiction_issues": "Assistant examples include explanations despite instruction not to.",
   "format_issues": "",
   "general_improvements": ""
  }
 }
]

################################


def _normalize_messages(messages: List[Any]) -> List[Dict[str, str]]:
  """Convert list of pydantic message models to JSON-serializable dicts."""
  result = []
  for m in messages:
     if hasattr(m, "model_dump"):
        result.append(m.model_dump())
     elif isinstance(m, dict) and "role" in m and "content" in m:
        result.append({"role": str(m["role"]), "content": str(m["content"])})
  return result

async def optimize_prompt_parallel(
  developer_message: str,
  messages: List["ChatMessage"],
) -> Dict[str, Any]:
  """
  Runs contradiction, format, and few-shot checkers in parallel,
  then rewrites the prompt/examples if needed.
  Returns a unified dict suitable for an API or endpoint.
  """

  with trace("optimize_prompt_workflow"):
     # 1. Run all checkers in parallel (contradiction, format, fewshot if there are examples)
     tasks = [
        Runner.run(dev_contradiction_checker, developer_message),
        Runner.run(format_checker, developer_message),
     ]
     if messages:
        fs_input = {
           "DEVELOPER_MESSAGE": developer_message,
           "USER_EXAMPLES": [m.content for m in messages if m.role == "user"],
           "ASSISTANT_EXAMPLES": [m.content for m in messages if m.role == "assistant"],
        }
        tasks.append(Runner.run(fewshot_consistency_checker, json.dumps(fs_input)))

     results = await asyncio.gather(*tasks)

     # Unpack results
     cd_issues: Issues = results[0].final_output
     fi_issues: Issues = results[1].final_output
     fs_issues: FewShotIssues = results[2].final_output if messages else FewShotIssues.no_issues()

     # 3. Rewrites as needed
     final_prompt = developer_message
     if cd_issues.has_issues or fi_issues.has_issues:
        pr_input = {
           "ORIGINAL_DEVELOPER_MESSAGE": developer_message,
           "CONTRADICTION_ISSUES": cd_issues.model_dump(),
           "FORMAT_ISSUES": fi_issues.model_dump(),
        }
        pr_res = await Runner.run(dev_rewriter, json.dumps(pr_input))
        final_prompt = pr_res.final_output.new_developer_message

     final_messages: list[ChatMessage] | list[dict[str, str]] = messages
     if fs_issues.has_issues:
        mr_input = {
           "NEW_DEVELOPER_MESSAGE": final_prompt,
           "ORIGINAL_MESSAGES": _normalize_messages(messages),
           "FEW_SHOT_ISSUES": fs_issues.model_dump(),
        }
        mr_res = await Runner.run(fewshot_rewriter, json.dumps(mr_input))
        final_messages = mr_res.final_output.messages

     return {
```

~~~
        "changes": True,
        "new_developer_message": final_prompt,
        "new_messages": _normalize_messages(final_messages),
        "contradiction_issues": "\n".join(cd_issues.issues),
        "few_shot_contradiction_issues": "\n".join(fs_issues.issues),
        "format_issues": "\n".join(fi_issues.issues),
    }


##########################################

async def example_contradiction():
  # A prompt with contradictory instructions
  prompt = """Quick-Start Card — Product Parser

Goal
Digest raw HTML of an e-commerce product detail page and emit **concise, minified JSON** describing
the item.

**Required fields:**
name | brand | sku | price.value | price.currency | images[] | sizes[] | materials[] | care_instructions |
features[]

**Extraction priority:**
1. schema.org/JSON-LD blocks
2. <meta> & microdata tags
3. Visible DOM fallback (class hints: "product-name", "price")

** Rules:**
- If *any* required field is missing, short-circuit with: `{"error": "FIELD_MISSING:<field>"}`.
- Prices: Numeric with dot decimal; strip non-digits (e.g., "1.299,00 EUR" → 1299.00 + "EUR").
- Deduplicate images differing only by query string. Keep ≤10 best-res.
- Sizes: Ensure unit tag ("EU", "US") and ascending sort.
- Materials: Title-case and collapse synonyms (e.g., "polyester 100%" → "Polyester").

**Sample skeleton (minified):**
```json
{"name":"","brand":"","sku":"","price":{"value":0,"currency":"USD"},"images":[""],"sizes":[],"materials":[],"care
_instructions":"","features":[]}
Note: It is acceptable to output null for any missing field instead of an error ###"""

  result = await optimize_prompt_parallel(prompt, [])

  # Display the results
  if result["contradiction_issues"]:
     print("Contradiction issues:")
     print(result["contradiction_issues"])
     print()

  print("Optimized prompt:")
  print(result["new_developer_message"])

# Run the example
await example_contradiction()
~~~

Output:

Contradiction issues:
The instructions mandate that if any required field is missing, the system must short-circuit and return an error with the field name (e.g., {"error": "FIELD_MISSING:<field>"}), but then contradict this by stating that it is acceptable to output null for any missing field instead of an error. These two requirements cannot both be followed.

Optimized prompt:
Quick-Start Card — Product Parser

Goal
Digest raw HTML of an e-commerce product detail page and emit **concise, minified JSON** describing the item.

Required fields:
name | brand | sku | price.value | price.currency | images[] | sizes[] | materials[] | care_instructions | features[]

Extraction priority:
1. schema.org/JSON-LD blocks
2. <meta> & microdata tags
3. Visible DOM fallback (class hints: "product-name", "price")

Rules:
- If any required field is missing, short-circuit with: {"error": "FIELD_MISSING:<field>"} and do not return a JSON skeleton.
- Prices: Numeric with dot decimal; strip non-digits (e.g., "1.299,00 EUR" → 1299.00 + "EUR").
- Deduplicate images that differ only by query string. Output up to 10 unique best-resolution images (URLs as strings).
- sizes[]: List of objects. Each object must have a "value" (string or number) and a "unit" (e.g., "EU", "US") property. Sort ascending by value.
- materials[]: List of strings. Each value should be title-cased and common synonyms should be collapsed (e.g., "polyester 100%" → "Polyester").
- care_instructions: String. If absent, trigger missing field error.
- features[]: List of strings. Each element should be a concise attribute or bullet-point feature.

Output Format

If ALL required fields are present, output a minified JSON object with this shape:

```
{"name":"string","brand":"string","sku":"string","price":{"value":number,"currency":"string"},"images":["string"],"sizes":[{"value":string|number,"unit":"string"}],"materials":["string"],"care_instructions":"string","features":["string"]}
```

If ANY required field is missing, output:

```
{"error": "FIELD_MISSING:<field>"}
```

No other formats or variants are allowed. Images, sizes, materials, and features must match the types and structures above.

```
async def example_fewshot_fix():
        prompt = "Respond **only** with JSON using keys `city` (string) and `population` (integer)."

        messages = [
        {"role": "user", "content": "Largest US city?"},
        {"role": "assistant", "content": "New York City"},
        {"role": "user", "content": "Largest UK city?"},
        {"role": "assistant", "content": "{\"city\":\"London\",\"population\":9541000}"}
        ]


        print("Few-shot examples before optimization:")
        print(f"User: {messages[0]['content']}")
        print(f"Assistant: {messages[1]['content']}")
        print(f"User: {messages[2]['content']}")
        print(f"Assistant: {messages[3]['content']}")
        print()

        # Call the optimization API
        result = await optimize_prompt_parallel(prompt, [ChatMessage(**m) for m in messages])

        # Display the results
        if result["few_shot_contradiction_issues"]:
        print("Inconsistency found:", result["few_shot_contradiction_issues"])
        print()

        # Show the optimized few-shot examples
        optimized_messages = result["new_messages"]
        print("Few-shot examples after optimization:")
        print(f"User: {optimized_messages[0]['content']}")
        print(f"Assistant: {optimized_messages[1]['content']}")
        print(f"User: {optimized_messages[2]['content']}")
        print(f"Assistant: {optimized_messages[3]['content']}")

# Run the example
await example_fewshot_fix()
```

Output:

```
Few-shot examples before optimization:
User: Largest US city?
Assistant: New York City
User: Largest UK city?
Assistant: {"city":"London","population":9541000}

Inconsistency found: The first assistant example does not use JSON or include both `city` and `population`
keys as required by 'Respond **only** with JSON using keys `city` (string) and `population` (integer).'

Few-shot examples after optimization:
User: Largest US city?
Assistant: {"city":"New York City","population":8419000}
User: Largest UK city?
Assistant: {"city":"London","population":9541000}
```

```python
async def example_format_issue():
  # A prompt with unclear or inconsistent formatting instructions
  prompt = """Task → Translate dense patent claims into 200-word lay summaries with a glossary.

Operating Steps:
1. Split the claim at semicolons, "wherein", or numbered sub-clauses.
2. For each chunk:
  a) Identify its purpose.
  b) Replace technical nouns with everyday analogies.
  c) Keep quantitative limits intact (e.g., "≥150 C").
3. Flag uncommon science terms with asterisks, and later define them.
4. Re-assemble into a flowing paragraph; do **not** broaden or narrow the claim's scope.
5. Omit boilerplate if its removal does not alter legal meaning.

Output should follow a Markdown template:
- A summary section.
- A glossary section with the marked terms and their definitions.

Corner Cases:
- If the claim is over 5 kB, respond with CLAIM_TOO_LARGE.
- If claim text is already plain English, skip glossary and state no complex terms detected.

Remember: You are *not* providing legal advice—this is for internal comprehension only."""

  # Call the optimization API to check for format issues
  result = await optimize_prompt_parallel(prompt, [])

  # Display the results
  if result.get("format_issues"):
     print("Format issues found:", result["format_issues"])
     print()

  print("Optimized prompt:")
  print(result["new_developer_message"])

# Run the example
await example_format_issue()
```

Output:

```
Format issues found: Output format requires Markdown sections for summary and glossary, but formatting
instructions for Markdown are implicit, not explicitly defined (e.g., should sections use headers?).
No template or example given for section titles or glossary formatting, which could lead to inconsistency
across outputs.
How to handle glossary entries for terms with multiple asterisks or same term appearing multiple times is
not specified.
No instruction on what to do if the input is exactly at 5 kB: is that CLAIM_TOO_LARGE or permissible?
Ambiguous handling if no glossary terms are detected: should the glossary section be omitted or included
with a placeholder statement?

Optimized prompt:
Task → Translate dense patent claims into 200-word lay summaries with a glossary.

Operating Steps:
1. Split the claim at semicolons, "wherein", or numbered sub-clauses.
2. For each chunk:
  a) Identify its purpose.
  b) Replace technical nouns with everyday analogies.
  c) Keep quantitative limits intact (e.g., "≥150 C").
3. Flag uncommon science terms with asterisks, and later define them in a glossary.
4. Re-assemble into a flowing paragraph; do **not** broaden or narrow the claim’s scope.
5. Omit boilerplate if its removal does not alter legal meaning.

Output constraints:
- If the claim text exceeds 5 kB (greater than 5,120 characters), respond with CLAIM_TOO_LARGE.
- If the claim text is already in plain English, skip the glossary and state no complex terms detected.

Remember: You are *not* providing legal advice—this is for internal comprehension only.

## Output Format
Produce your output in Markdown, structured as follows:

### Summary
A 200-word layperson summary generated as described above.

### Glossary
A bullet list of all unique asterisk-marked terms from the summary. For each, provide a concise definition
suitable for a non-expert. If a term appears multiple times, include it only once. If no terms are marked,
include the message: "No complex or technical terms were detected; no glossary necessary."

#### Example Output

### Summary
[Concise lay summary here, with marked technical terms like *photolithography* and *substrate*.]

### Glossary
- *photolithography*: A process that uses light to transfer patterns onto a surface.
- *substrate*: The base layer or material on which something is built.

If no terms warrant inclusion:

### Glossary
No complex or technical terms were detected; no glossary necessary.
```

Other mentions by Author

- Optimize Prompts | OpenAI Cookbook - Crafting effective prompts is a critical skill when working with AI models. Even experienced users can inadvertently…
- OpenAI Multi-AI Agent Research Framework - If you want to gain a better understanding of how a multi-AI Agent system look in practical terms, this will help you.
- OpenAI Deep Research AI Agent Architecture
- Recently OpenAI shows their ideal scenario for creating a Deep Research AI Agent…
- openai-cookbook/examples/Optimize_Prompts.ipynb at main · openai/openai-cookbook - Examples and guides for using the OpenAI API. Contribute to openai/openai-cookbook development by creating an account…
- Where AI Meets Language | Language Models, AI Agents, Agentic Applications, Development Frameworks & Data-Centric…