


# LLM Integration


---

**Description:** Understanding prompting patterns, grounding techniques, and reasoning frameworks for effective LLM integration

# LLM Integration

---

 **Purpose:** Master prompting, grounding, and reasoning techniques to get the best performance from language models.

 **Source of Truth:** [google/adk-python/src/google/adk/models/](https://github.com/google/adk-python/tree/main/src/google/adk/models/) (ADK 1.15) + [google/adk-python/src/google/adk/planners/](https://github.com/google/adk-python/tree/main/src/google/adk/planners/) (ADK 1.15)

---



## Prompt Engineering Fundamentals

---

### | The Prompt = Program Model

**Mental Model:** A prompt is like **programming** an LLM:

## PROMPT ANATOMY

### [INSTR] SYSTEM/INSTRUCTION (Operating System)

"You are a helpful research assistant..."

- Defines agent personality and behavior
- Sets constraints and rules
- Provides role context

### [MEM] CONTEXT (Program Data)

"Current state: {topic}, Previous: {history}"

- Injected from session state
- Dynamic data via {key} syntax
- Tools available list

### [USER] USER MESSAGE (Function Call)

"Research quantum computing trends"

- The actual task/query
- Can be text, image, audio, video

### [TOOLS] TOOL RESULTS (Return Values)

"search\_result: {...}"

- Feedback from tool execution
- Multi-turn conversation

## Instruction Patterns

### Role-Based Instructions:

```
# Clear role definition
instruction = """
You are a senior software engineer who:
- Writes clean, maintainable code
- Follows Python best practices
- Adds helpful comments
- Explains complex concepts clearly
"""

# Task-specific role
instruction = """
You are a research analyst specializing in:
- Data-driven insights
- Statistical analysis
- Clear visualization recommendations
- Executive summary writing
"""
```

### Constraint-Based Instructions:

```
# Behavioral constraints
instruction = """
Rules you must follow:
- Always cite sources for facts
- If unsure, say "I don't know" instead of guessing
- Keep responses under 3 paragraphs unless asked for detail
- Use examples when explaining concepts
- Ask clarifying questions when input is ambiguous
"""

# Output format constraints
instruction = """
Response format:
1. Executive summary (2-3 sentences)
2. Key findings (bullet points)
3. Recommendations (numbered list)
4. Next steps (if applicable)
"""
```

### State-Aware Instructions:

```
# Dynamic context injection
instruction = """
You are helping {user:name} with their {current_task}.
Previous context: {conversation_summary}
Available tools: {tool_list}

Guidelines:
- Reference previous work when relevant
- Use appropriate tools for the task
- Maintain consistency with prior responses
"""
```

---

## Grounding Techniques

---

### | Web Grounding (Real-World Facts)

**Mental Model:** Grounding connects LLM **imagination** to **reality**:

```
# Automatic grounding with Gemini 2.0+
grounded_agent = Agent(
    name="researcher",
    model="gemini-2.0-flash", # Built-in search
    instruction="Research current information using web search"
)

# Explicit search integration
search_agent = Agent(
    name="web_researcher",
    model="gemini-2.5-flash",
    tools=[google_search],
    instruction="Find and analyze current information"
)
```

#### Grounding Patterns:

- **Factual Questions:** "What is the current population of Tokyo?"
- **Current Events:** "What happened in the news today?"
- **Real-time Data:** "What's the current stock price of AAPL?"

- **Local Information:** "What's the weather like in San Francisco?"

## | Data Grounding (Internal Knowledge)

**Mental Model:** Connect to your own data sources:

```
# Database grounding
db_agent = Agent(
    name="data_analyst",
    model="gemini-2.5-flash",
    tools=[database_tool, analysis_tool],
    instruction="Analyze company data to answer questions"
)

# Document grounding
doc_agent = Agent(
    name="knowledge_assistant",
    model="gemini-2.5-flash",
    tools=[document_search_tool],
    instruction="Find relevant information in company documents"
)
```

## | Location Grounding (Spatial Intelligence)

**Mental Model:** Geographic awareness and spatial reasoning:

```
# Maps integration
location_agent = Agent(
    name="location_assistant",
    model="gemini-2.5-flash",
    tools=[google_maps_grounding],
    instruction="Help with location-based queries and directions"
)

# Capabilities:
# - Address resolution
# - Distance calculations
# - Points of interest
# - Route optimization
```

# [BRAIN] Thinking & Reasoning Frameworks

---

## | Built-in Thinking (Native Model Capability)

**Mental Model:** Model thinks step-by-step internally:

```
# Gemini 2.0+ thinking
thinking_agent = Agent(
    name="reasoning_assistant",
    model="gemini-2.0-flash",
    instruction="Solve complex problems with clear reasoning",
    thinking_config=ThinkingConfig(
        include_thoughts=True, # Show reasoning in response
        max_thoughts=10
    )
)

# Thinking appears in response:
# "Let me think about this step by step:
# 1. First, I need to understand the problem...
# 2. Then, I should consider the constraints...
# 3. Finally, I'll provide the solution..."
```

## | Plan-ReAct Pattern (Structured Reasoning)

**Mental Model:** Explicit planning and action framework:

```

from google.adk.planners import PlanReActPlanner

# Structured reasoning agent
reasoning_agent = Agent(
    name="strategic_planner",
    model="gemini-2.5-flash",
    planner=PlanReActPlanner(),
    tools=[research_tool, analysis_tool],
    instruction="Plan and execute complex multi-step tasks"
)

# Execution pattern:
# [PLANNING] 1. Research topic 2. Analyze data 3. Create report
# [REASONING] I should start with research to gather facts...
# [ACTION] Call research_tool("quantum computing")
# [OBSERVATION] Found 15 relevant papers...
# [REPLANNING] Now analyze the data...

```

## When to Use Each Approach

Scenario	Built-in Thinking	Plan-ReAct	None
Complex reasoning needed	✓	✓	✗
Want visible reasoning	✓	✓	✗
Multi-step problems	✓	✓	🤔
Need replanning	✗	✓	✗
Simple queries	✗	✗	✓
Speed critical	✗	✗	✓

# [FLOW] Multi-Turn Conversations

## | Context Management

**Mental Model:** Maintain conversation state across turns:

```
# State-aware agent
conversational_agent = Agent(
    name="assistant",
    model="gemini-2.5-flash",
    instruction="""
    You are a helpful assistant. Previous conversation:
    {conversation_history}

    Current user: {user:name}
    Current task: {current_task}
    """,
    output_key="response"
)

# State tracks conversation flow
state['conversation_history'] += f"User: {user_input}\nAssistant: {response}\n"
state['current_task'] = extract_task(user_input)
```

## | Tool Call Chains

**Mental Model:** LLMs can call multiple tools in sequence:



```
# Multi-tool agent
research_agent = Agent(
    name="comprehensive_researcher",
    model="gemini-2.5-flash",
    tools=[web_search, database_query, analysis_tool],
    instruction="""
    Research thoroughly using all available tools:
    1. Search the web for current information
    2. Query internal databases for company data
    3. Analyze and synthesize findings
    """
)

# LLM can generate multiple tool calls:
# 1. web_search("topic overview")
# 2. database_query("internal data")
# 3. analysis_tool("combine results")
```

## Error Recovery

**Mental Model:** Handle failures gracefully and recover:

```
# Robust agent with error handling
robust_agent = Agent(
    name="reliable_assistant",
    model="gemini-2.5-flash",
    instruction="""
    If a tool fails, try alternative approaches:
    - Web search if database is down
    - Simplified analysis if detailed data unavailable
    - Clear explanation of limitations

    Always provide value even with partial information.
    """,
    tools=[primary_tool, fallback_tool]
)
```



# Advanced Prompting Techniques

## Chain of Thought Prompting

**Mental Model:** Guide step-by-step reasoning:

```
# Explicit reasoning steps
reasoning_instruction = """
Solve this problem step by step:

1. Understand the problem: What is being asked?
2. Identify key information: What data do I have?
3. Consider approaches: What methods could work?
4. Evaluate options: Which approach is best?
5. Execute solution: Implement the chosen approach
6. Verify result: Does this make sense?

Show your work at each step.
"""
```

## Few-Shot Learning

**Mental Model:** Learn from examples in the prompt:

```
# Example-based instruction
few_shot_instruction = """
Classify the sentiment of text as positive, negative, or neutral.

Examples:
Text: "I love this product!" → positive
Text: "This is terrible quality" → negative
Text: "It's okay, nothing special" → neutral

Now classify: "{user_text}"
"""
```

## Meta-Prompting

**Mental Model:** Prompt about how to prompt:

```
# Self-improving prompts
meta_instruction = """
First, analyze what type of question this is:
- Factual: Look for specific information
- Analytical: Break down components
- Creative: Generate novel ideas
- Advisory: Provide recommendations

Then, choose the appropriate response strategy:
- Factual: Cite sources, be precise
- Analytical: Structure with sections
- Creative: Brainstorm multiple options
- Advisory: Consider pros/cons, provide rationale

Question: {user_question}
"""
```



## Performance Optimization

### Model Selection Strategy

**Mental Model:** Right model for the right task:

```
# Model routing based on complexity
def select_model(query):
    if len(query.split()) < 10: # Simple
        return "gemini-2.5-flash" # Fast, cheap
    elif "analyze" in query.lower(): # Complex
        return "gemini-2.5-pro" # Better reasoning
    else:
        return "gemini-2.0-flash" # Balanced

# Dynamic model selection
agent = Agent(
    name="adaptive_assistant",
    model=select_model(user_query), # Dynamic selection
    instruction="Provide the best response for this query type"
)
```

## Context Window Management

**Mental Model:** Keep relevant information, discard irrelevant:

```
# Context pruning
def prune_context(history, max_tokens=4000):
    """Keep recent and relevant messages"""
    relevant = []
    total_tokens = 0

    for msg in reversed(history):
        if total_tokens + len(msg) > max_tokens:
            break
        if is_relevant(msg): # Custom relevance function
            relevant.insert(0, msg)
            total_tokens += len(msg)

    return relevant

# Efficient context usage
agent = Agent(
    name="efficient_assistant",
    model="gemini-2.5-flash",
    instruction="Use context efficiently: {pruned_history}"
)
```

## Caching Strategies

**Mental Model:** Reuse computations for common queries:

```
# Response caching
cache = {}

def get_cached_response(query):
    key = hash(query)
    if key in cache:
        return cache[key]

    response = agent.run(query)
    cache[key] = response
    return response

# Semantic caching
def semantic_cache(query):
    """Cache based on meaning, not exact text"""
    intent = extract_intent(query)
    if intent in cache:
        return adapt_cached_response(cache[intent], query)
```



## Debugging LLM Integration

### Response Analysis

#### Track LLM Behavior:

```
# Enable detailed logging
import logging
logging.getLogger('google.adk.models').setLevel(logging.DEBUG)

# Inspect LLM calls
result = await runner.run_async(query)
for event in result.events:
    if event.type == 'llm_request':
        print(f"Prompt: {event.prompt}")
        print(f"Model: {event.model}")
    elif event.type == 'llm_response':
        print(f"Response: {event.response}")
        print(f"Tokens: {event.token_count}")
```

## Prompt Iteration

### A/B Testing Prompts:

```
# Test different prompt versions
prompts = {
    'concise': "Answer briefly: {query}",
    'detailed': "Provide comprehensive answer: {query}",
    'structured': "Structure answer with sections: {query}"
}

for version, prompt in prompts.items():
    agent = Agent(model="gemini-2.5-flash", instruction=prompt)
    result = agent.run(test_query)
    score = evaluate_response(result)
    print(f"{version}: {score}")
```

## Common Issues & Solutions

Issue	Symptom	Solution
Hallucinations	Makes up facts	Add grounding tools, fact-checking
Verbose responses	Too much text	Set length limits, structured formats
Off-topic answers	Ignores constraints	Clear instructions, role definitions
Inconsistent style	Varies between responses	Consistent persona, examples
Tool misuse	Wrong tool for task	Better tool descriptions, examples
Context loss	Forgets conversation	State management, context injection



## Related Topics

---

- **[Tools & Capabilities](#)** → ([tools-capabilities.md](#)): What LLMs can call
  - **[Workflows & Orchestration](#)** → ([workflows-orchestration.md](#)): How LLMs coordinate tasks
  - **[Decision Frameworks](#)** → ([decision-frameworks.md](#)): When to use different techniques
- 



## Key Takeaways

---

1. **Prompting:** Clear instructions, constraints, and examples
2. **Grounding:** Connect to real world via search, data, location
3. **Thinking:** Use built-in or explicit reasoning frameworks
4. **Context:** Manage conversation state and relevance
5. **Optimization:** Right model, efficient context, caching
6. **Debugging:** Monitor LLM behavior and iterate prompts

🔗 **Next:** Learn about [Production & Deployment](#) ([production-deployment.md](#)) to run these integrations at scale.

---

Generated on 2025-10-21 09:03:24 from llm-integration.md

Source: Google ADK Training Hub