Tutorial 12: Planners and Thinking - Strategic Agent Planning

Difficulty: advanced

Reading Time: 2 hours

Tags: advanced, planning, reasoning, strategic-thinking, task-decomposition

Description: Implement planning and reasoning capabilities in agents using strategic thinking patterns, task decomposition, and execution planning.

Tutorial 12: Planners & Thinking Configuration

Goal: Master advanced reasoning capabilities using Built-in Planners, Thinking Configuration, and structured Plan-ReAct patterns for complex problem-solving.

Prerequisites:

- Tutorial 01 (Hello World Agent)
- Tutorial 02 (Function Tools)
- Gemini 2.0+ model access

What You'll Learn:

- Using `BuiltInPlanner` with extended thinking
- Implementing `PlanReActPlanner` for structured reasoning
- Configuring `ThinkingConfig` for transparent reasoning
- Creating custom planners with `BasePlanner`
- Building agents that plan before acting
- Understanding when to use which planner

Time to Complete: 50-65 minutes

🚀 Quick Start

1. Setup Environment

```
# Clone and navigate to the implementation
cd tutorial_implementation/tutorial12

# Install dependencies
make setup

# Copy environment template
cp strategic_solver/.env.example strategic_solver/.env

# Edit .env and add your Google AI API key
# GOOGLE_API_KEY=your_actual_api_key_here
```

2. Run Development Server

```
# Start ADK web interface
make dev

# Open http://localhost:8000 in your browser
# Select "strategic_solver" from the agent dropdown
```

3. Test the Implementation

```
# Run comprehensive test suite
make test

# See example queries you can test
make examples

# Run demo examples
make demo
```

Why Planners Matter

Default agents react immediately to queries. **Planners** add a crucial step: **thinking before acting**. This leads to:

- [BRAIN] **Better Reasoning**: Multi-step problem decomposition
- 🎯 **Improved Accuracy**: Plan validation before execution
- 🔍 **Transparent Thinking**: See how agent reasons
- [FLOW] **Dynamic Replanning**: Adjust strategy based on results
- 💡 **Complex Problem Solving**: Handle multi-faceted challenges

Without Planner (Direct Response):

```
User: "Plan a trip to Japan"
Agent: "Here's a trip plan..." [Immediate response]
```

With Planner (Structured Reasoning):

```
User: "Plan a trip to Japan"
Agent:
  <PLAN>
  1. Understand requirements (budget, duration, interests)
  2. Research destinations
  3. Create itinerary
  4. Estimate costs
  5. Provide recommendations
  </PLAN>

  <REASONING>
  Need to gather info first, then plan systematically...
  </REASONING>

  <ACTION>
  Let me start by asking about your preferences...
  </ACTION>
```

1. BuiltInPlanner (Extended Thinking)

What is BuiltInPlanner?

`BuiltInPlanner` leverages Gemini 2.0+'s **native thinking capabilities** - the model performs extended reasoning internally before generating responses.

Source: `google/adk/planners/built_in_planner.py`

Basic Usage

```python
from google.adk.agents import Agent
from google.adk.planners import BuiltInPlanner
from google.adk.runners import Runner
from google.genai import types

# Create agent with extended thinking
agent = Agent(
    model='gemini-2.0-flash',  # Requires Gemini 2.0+ with thinking support
    name='thoughtful_assistant',
    instruction='You are a helpful assistant that thinks carefully before resp
    planner=BuiltInPlanner(
        thinking_config=types.ThinkingConfig(
            include_thoughts=True  # Show reasoning to user
        )
    )
)

runner = Runner()
result = runner.run(
    "How would you solve world hunger?",
    agent=agent
)

print(result.content.parts[0].text)
# Includes model's reasoning process
```

Output includes thinking:

```
[Thinking]
This is a complex global issue requiring multi-faceted approach.
I need to consider:
- Agricultural technology
- Distribution systems
- Economic factors
- Political will
- Climate change impact

Let me structure this systematically...
[End Thinking]

Based on my analysis, here are key strategies to address world hunger:
1. Improve agricultural productivity in developing regions...
2. Reduce food waste through better supply chains...
...
```

ThinkingConfig Options

```python
from google.genai import types

# Show thinking to user
thinking_config = types.ThinkingConfig(
    include_thoughts=True  # User sees reasoning
)

# Hide thinking (just final answer)
thinking_config = types.ThinkingConfig(
    include_thoughts=False  # Only final answer shown
)
```

When to show thinking:

- ✅ Educational applications (teach reasoning)
- ✅ Debugging agent logic
- ✅ Building trust (transparent AI)
- ✅ Complex problem explanations

When to hide thinking:

- ✅ Production user-facing apps

- ✔️ When users want quick answers
- ✔️ API responses (efficiency)
- ✔️ When thinking adds no value

How It Works Internally

```python
# Simplified implementation from BuiltInPlanner
class BuiltInPlanner(BasePlanner):
    def __init__(self, thinking_config: types.ThinkingConfig = None):
        self.thinking_config = thinking_config or types.ThinkingConfig()

    def apply_thinking_config(self, llm_request: LlmRequest):
        """Apply thinking config to LLM request."""
        if self.thinking_config:
            llm_request.config.thinking_config = self.thinking_config
        return llm_request
```

Model Compatibility

```python
# ✔️ Works with Gemini 2.0+ models supporting thinking
agent = Agent(
    model='gemini-2.0-flash',
    planner=BuiltInPlanner(thinking_config=types.ThinkingConfig(include_though
)

# ❌ May not work with models without thinking support
# Check model capabilities before using
```

2. PlanReActPlanner (Structured Reasoning)

What is PlanReActPlanner?

`PlanReActPlanner` implements the **Plan-ReAct pattern**: Plan → Reason → Act → Observe → Replan. This creates a structured reasoning loop.

Source: `google/adk/planners/plan_re_act_planner.py`

Basic Usage

```python
from google.adk.agents import Agent
from google.adk.planners import PlanReActPlanner
from google.adk.runners import Runner

# Create agent with Plan-ReAct pattern
agent = Agent(
    model='gemini-2.0-flash',
    name='systematic_planner',
    instruction='You solve problems systematically using planning and reasonin
    planner=PlanReActPlanner()
)

runner = Runner()
result = runner.run(
    "Build a machine learning model to predict house prices",
    agent=agent
)

print(result.content.parts[0].text)
```

Output structure:

```
<PLANNING>
To build a house price prediction model, I need to:
1. Gather and clean housing data
2. Select relevant features (size, location, age, etc.)
3. Choose appropriate algorithm (regression)
4. Train and validate model
5. Evaluate performance
</PLANNING>

<REASONING>
For this problem:
- Linear regression suitable for continuous price prediction
- Need features like square footage, bedrooms, location
- Must handle missing data and outliers
- Cross-validation important for generalization
</REASONING>

<ACTION>
Let me start by outlining the data requirements:
- Historical sale prices
- Property characteristics
- Location data
...
</ACTION>

<FINAL_ANSWER>
Here's a complete plan for your ML model...
</FINAL_ANSWER>
```

Planning Tags

PlanReActPlanner uses XML-like tags to structure reasoning:

Tag	Purpose	When Used
`<PLANNING>`	Initial plan	Start of task
`<REASONING>`	Explain logic	Throughout process
`<ACTION>`	Execute steps	When doing something
`<OBSERVATION>`	Note results	After actions
`<REPLANNING>`	Adjust plan	When strategy changes
`<FINAL_ANSWER>`	Conclusion	End of task

Replanning Example

```python
from google.adk.agents import Agent
from google.adk.planners import PlanReActPlanner
from google.adk.tools import FunctionTool
from google.adk.runners import Runner

# Tool that simulates failure
def check_api_status(api_name: str) -> str:
    """Check if API is available."""
    if api_name == "primary_api":
        return "ERROR: API unavailable"
    return "OK: API available"

# Agent with replanning capability
agent = Agent(
    model='gemini-2.0-flash',
    name='adaptive_agent',
    instruction='Use tools and adapt plan when issues arise.',
    planner=PlanReActPlanner(),
    tools=[FunctionTool(check_api_status)]
)

runner = Runner()
result = runner.run(
    "Fetch data from primary_api and process it",
    agent=agent
)

print(result.content.parts[0].text)
```

Output shows replanning:

```
<PLANNING>
Plan:
1. Check primary_api status
2. Fetch data from primary_api
3. Process data
</PLANNING>

<ACTION>
Checking primary_api status...
</ACTION>

<OBSERVATION>
API returned ERROR: API unavailable
</OBSERVATION>

<REPLANNING>
Primary API is down. New plan:
1. Check backup_api status
2. Use backup_api instead
3. Process data from backup source
</REPLANNING>

<ACTION>
Switching to backup_api...
</ACTION>

<FINAL_ANSWER>
Successfully retrieved and processed data using backup_api.
</FINAL_ANSWER>
```

Planning Instructions

PlanReActPlanner injects detailed planning instructions:

```
# Internal planning instruction (simplified)
PLANNING_INSTRUCTION = """
You must follow this structured reasoning format:

<PLANNING>
Break down the problem into steps:
1. Step 1
2. Step 2
3. ...
</PLANNING>

<REASONING>
Explain why this plan makes sense:
- Consideration 1
- Consideration 2
</REASONING>

<ACTION>
Describe what you're doing now
</ACTION>

<OBSERVATION>
Note what happened
</OBSERVATION>

If plan needs adjustment:
<REPLANNING>
Explain why replanning and new plan:
1. New step 1
2. ...
</REPLANNING>

When done:
<FINAL_ANSWER>
Provide final result
</FINAL_ANSWER>
"""
```

3. Real-World Example: Strategic Problem Solver

Let's build an agent that solves complex business problems using Plan-ReAct.

Complete Implementation

```python
"""
Strategic Business Problem Solver
Uses Plan-ReAct pattern for systematic problem solving.
"""

import asyncio
import os
from datetime import datetime
from google.adk.agents import Agent, Runner
from google.adk.planners import PlanReActPlanner
from google.adk.tools import FunctionTool
from google.adk.tools.tool_context import ToolContext
from google.genai import types

# Environment setup
os.environ['GOOGLE_GENAI_USE_VERTEXAI'] = '1'
os.environ['GOOGLE_CLOUD_PROJECT'] = 'your-project-id'
os.environ['GOOGLE_CLOUD_LOCATION'] = 'us-central1'

# Tool: Market research
def analyze_market(industry: str, region: str) -> dict:
    """Analyze market conditions (simulated)."""
    # In production, call real market data APIs
    return {
        'industry': industry,
        'region': region,
        'growth_rate': '8.5%',
        'competition': 'High',
        'trends': ['Digital transformation', 'Sustainability focus'],
        'opportunities': ['Emerging markets', 'New technologies']
    }

# Tool: Financial analysis
def calculate_roi(investment: float, annual_return: float, years: int) -> dict
    """Calculate return on investment."""
    total_return = investment * ((1 + annual_return/100) ** years)
    profit = total_return - investment
    return {
        'initial_investment': investment,
        'annual_return_rate': f"{annual_return}%",
        'years': years,
        'total_return': round(total_return, 2),
        'profit': round(profit, 2),
        'roi_percentage': round((profit/investment)*100, 2)
    }
```

```
# Tool: Risk assessment
def assess_risk(factors: list[str]) -> dict:
    """Assess business risks."""
    risk_scores = {
        'market_volatility': 7,
        'regulatory_changes': 5,
        'competition': 8,
        'technology': 6,
        'financial': 4
    }

    total_risk = sum(risk_scores.get(f, 5) for f in factors)
    avg_risk = total_risk / len(factors) if factors else 5

    return {
        'factors_assessed': factors,
        'risk_score': round(avg_risk, 2),
        'risk_level': 'High' if avg_risk > 7 else 'Medium' if avg_risk > 4 els
        'mitigation_needed': avg_risk > 6
    }

# Tool: Save strategy report
async def save_strategy_report(
    problem: str,
    strategy: str,
    tool_context: ToolContext
) -> str:
    """Save strategic plan as artifact."""

    timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    report = f"""
# Strategic Business Plan
Generated: {timestamp}

## Problem Statement
{problem}

## Recommended Strategy
{strategy}

## Plan Generated By
- Agent: Strategic Problem Solver
- Planner: PlanReActPlanner
- Model: gemini-2.0-flash
    """.strip()

    filename = f"strategy_{problem[:30].replace(' ', '_')}.md"
    version = await tool_context.save_artifact(
        filename=filename,
        part=types.Part.from_text(report)
    )

    return f"Strategy saved as {filename} (version {version})"

# Create strategic problem solver
strategic_solver = Agent(
    model='gemini-2.0-flash',
    name='strategic_solver',
    description='Solves complex business problems systematically',
    planner=PlanReActPlanner(),  # Use structured planning
    instruction="""
You are a strategic business consultant. When given a problem:

1. PLAN: Break down into clear steps
2. REASON: Explain your logic
3. ACT: Use tools to gather data
4. OBSERVE: Analyze results
5. REPLAN: Adjust if needed
6. CONCLUDE: Provide final recommendation

Always be thorough and data-driven. Use tools for:
- analyze_market: Market research
- calculate_roi: Financial projections
- assess_risk: Risk analysis
- save_strategy_report: Save final plan

Think step-by-step and show your reasoning.
    """.strip(),
    tools=[
        FunctionTool(analyze_market),
        FunctionTool(calculate_roi),
        FunctionTool(assess_risk),
        FunctionTool(save_strategy_report)
    ],
    generate_content_config=types.GenerateContentConfig(
        temperature=0.4,  # Balanced for strategic thinking
        max_output_tokens=3000
    )
)

async def solve_business_problem(problem: str):
    """Solve strategic business problem."""

    print(f"\n{'='*70}")
    print(f"PROBLEM: {problem}")
    print(f"{'='*70}\n")

    runner = Runner()

    result = await runner.run_async(
        problem,
        agent=strategic_solver
    )

    print("\n📊 STRATEGIC ANALYSIS:\n")
    print(result.content.parts[0].text)
    print(f"\n{'='*70}\n")

async def main():
    """Run strategic problem-solving examples."""

    # Example 1: Market entry strategy
    await solve_business_problem("""
We're a mid-sized software company considering entering the
healthcare AI market. Should we pursue this? What's the strategy?
    """)

    await asyncio.sleep(2)

    # Example 2: Investment decision
    await solve_business_problem("""
We have $500,000 to invest in either:
A) Expanding current product line (15% annual return, medium risk)
B) Entering new market (25% annual return, high risk)
Which should we choose for a 5-year horizon?
    """)

    await asyncio.sleep(2)

    # Example 3: Risk mitigation
    await solve_business_problem("""
Our startup faces:
- Increasing competition
- Regulatory uncertainty
- Technology disruption
How should we mitigate these risks?
    """)

if __name__ == '__main__':
    asyncio.run(main())
```

Expected Output

```
======================================================================
PROBLEM: We're a mid-sized software company considering entering the
healthcare AI market. Should we pursue this? What's the strategy?
======================================================================

📊 STRATEGIC ANALYSIS:

<PLANNING>
To evaluate healthcare AI market entry, I need to:
1. Analyze market conditions using analyze_market tool
2. Assess financial viability with calculate_roi
3. Evaluate risks using assess_risk
4. Synthesize findings into recommendation
5. Save strategic plan
</PLANNING>

<ACTION>
Analyzing healthcare AI market...
</ACTION>

<OBSERVATION>
Market Analysis Results:
- Industry: Healthcare AI
- Growth Rate: 8.5% annually
- Competition: High
- Key Trends: Digital transformation, Sustainability focus
- Opportunities: Emerging markets, New technologies
</OBSERVATION>

<REASONING>
Healthcare AI shows strong growth (8.5%), but competition is high.
Digital transformation trend is favorable. Need to assess financial
returns and risks before recommendation.
</REASONING>

<ACTION>
Calculating potential ROI for $500K investment over 5 years...
</ACTION>

<OBSERVATION>
ROI Analysis:
- Initial Investment: $500,000
- Projected Annual Return: 15%
- 5-Year Total Return: $1,005,694
- Profit: $505,694
- ROI: 101.14%
```

```
</OBSERVATION>

<ACTION>
Assessing business risks...
</ACTION>

<OBSERVATION>
Risk Assessment:
- Factors: Market volatility, Regulatory changes, Competition, Technology
- Risk Score: 6.5/10
- Risk Level: Medium
- Mitigation Needed: Yes
</OBSERVATION>

<FINAL_ANSWER>
**RECOMMENDATION: PROCEED WITH CAUTION**

**Market Entry Strategy:**

1. **Start Small (Phase 1: Year 1)**
   - Pilot project with 2-3 healthcare clients
   - Focus on niche use case (e.g., diagnostic imaging AI)
   - Investment: $100-150K
   - Build proof of concept

2. **Validate & Scale (Phase 2: Years 2-3)**
   - Expand to 10+ clients if pilot successful
   - Develop partnerships with hospitals
   - Investment: $200-250K
   - Achieve regulatory compliance

3. **Full Market Entry (Phase 3: Years 4-5)**
   - Launch full product suite
   - Scale operations
   - Investment: remaining budget
   - Target national presence

**Risk Mitigation:**
- Partner with established healthcare providers
- Hire regulatory compliance expert
- Maintain diversified product portfolio
- Build strong IP protection

**Financial Outlook:**
- Projected 5-year ROI: 101%
- Break-even: Expected Year 3
- Market growth: 8.5% annually
```

```
**Key Success Factors:**
- Regulatory compliance from day 1
- Strong clinical partnerships
- Differentiated technology
- Patient privacy focus

[Strategy saved as strategy_We're_a_mid-sized_software.md (version 1)]
</FINAL_ANSWER>

======================================================================
```

4. BasePlanner (Custom Planners)

What is BasePlanner?

`BasePlanner` is the **abstract base class** for creating custom planning strategies.

Source: `google/adk/planners/base_planner.py`

Creating Custom Planner

```python
from google.adk.planners import BasePlanner
from google.adk.types import LlmRequest, LlmResponse

class MyCustomPlanner(BasePlanner):
    """Custom planning strategy."""

    def build_planning_instruction(self, agent, context) -> str:
        """Inject custom planning instructions."""
        return """
You are a systematic problem solver. For each task:

STEP 1: ANALYZE
- What's the goal?
- What constraints exist?
- What resources are available?

STEP 2: STRATEGIZE
- What are possible approaches?
- What are pros/cons of each?
- Which is optimal?

STEP 3: EXECUTE
- Implement chosen strategy
- Monitor progress
- Adjust as needed

STEP 4: VALIDATE
- Did we achieve the goal?
- What could be improved?
        """

    def process_planning_response(self, response: LlmResponse) -> LlmResponse:
        """Process response after planning."""
        # Could modify response here
        # Add metadata, validate structure, etc.
        return response

# Use custom planner
agent = Agent(
    model='gemini-2.0-flash',
    planner=MyCustomPlanner()
)
```

Advanced Custom Planner Example

```python
class DataSciencePlanner(BasePlanner):
    """Planner for data science workflows."""

    def build_planning_instruction(self, agent, context) -> str:
        return """
Follow the data science methodology:

<DATA_UNDERSTANDING>
1. What data is available?
2. What's the data quality?
3. What are the features?
</DATA_UNDERSTANDING>

<PROBLEM_FORMULATION>
1. What's the prediction target?
2. What type of problem? (classification, regression, clustering)
3. What's the success metric?
</PROBLEM_FORMULATION>

<MODELING_APPROACH>
1. Which algorithms are suitable?
2. How to validate? (train/test split, cross-validation)
3. How to tune hyperparameters?
</MODELING_APPROACH>

<EVALUATION>
1. What's the model performance?
2. Is it good enough?
3. How to improve?
</EVALUATION>

<DEPLOYMENT>
1. How to deploy model?
2. How to monitor performance?
3. How to update model?
</DEPLOYMENT>
        """

# Data science agent with custom planner
ds_agent = Agent(
    model='gemini-2.0-flash',
    name='data_scientist',
    planner=DataSciencePlanner(),
    instruction='You are an expert data scientist following best practices.'
)
```

5. Comparing Planners

When to Use Each Planner

Planner	Best For	Pros	Cons
BuiltInPlanner	Complex reasoning tasks	Native thinking, transparent, fast	Gemini 2.0+ only
PlanReActPlanner	Multi-step workflows	Structured, replannable, debuggable	More verbose
BasePlanner (Custom)	Domain-specific logic	Full control, tailored	More implementation work
No Planner	Simple queries	Fast, minimal overhead	No structured reasoning

Decision Tree

```
Need planning?
├─ No → Use default (no planner)
└─ Yes → What type?
    ├─ Want native model thinking?
    |   └─ Yes → BuiltInPlanner (Gemini 2.0+)
    ├─ Need structured steps?
    |   └─ Yes → PlanReActPlanner
    ├─ Domain-specific workflow?
    |   └─ Yes → Custom BasePlanner
    └─ General purpose?
        └─ PlanReActPlanner (most flexible)
```

Performance Comparison

```python
import asyncio
import time
from google.adk.agents import Agent, Runner
from google.adk.planners import BuiltInPlanner, PlanReActPlanner
from google.genai import types

async def compare_planners():
    """Compare planner performance."""

    query = "Design a sustainable urban transportation system"

    # No planner
    agent_default = Agent(
        model='gemini-2.0-flash',
        name='default'
    )

    # BuiltInPlanner
    agent_builtin = Agent(
        model='gemini-2.0-flash',
        name='builtin',
        planner=BuiltInPlanner(
            thinking_config=types.ThinkingConfig(include_thoughts=True)
        )
    )

    # PlanReActPlanner
    agent_planreact = Agent(
        model='gemini-2.0-flash',
        name='planreact',
        planner=PlanReActPlanner()
    )

    runner = Runner()

    for agent in [agent_default, agent_builtin, agent_planreact]:
        start = time.time()
        result = await runner.run_async(query, agent=agent)
        elapsed = time.time() - start

        print(f"\n{'='*60}")
        print(f"Agent: {agent.name}")
        print(f"Time: {elapsed:.2f}s")
        print(f"Response length: {len(result.content.parts[0].text)} chars")
        print(f"{'='*60}")
```

```
asyncio.run(compare_planners())
```

Typical Results:

- **No Planner**: 2-3s, 500-800 chars (direct answer)
- **BuiltInPlanner**: 4-6s, 800-1200 chars (with thinking)
- **PlanReActPlanner**: 5-8s, 1200-2000 chars (structured)

6. Best Practices

✔️ DO: Match Planner to Task Complexity

```
# ✔️ Simple query - no planner needed
simple_agent = Agent(
    model='gemini-2.0-flash',
    instruction='Answer questions concisely'
)
runner.run("What's 2+2?", agent=simple_agent)

# ✔️ Complex problem - use planner
complex_agent = Agent(
    model='gemini-2.0-flash',
    instruction='Solve complex problems systematically',
    planner=PlanReActPlanner()
)
runner.run("Design a climate change mitigation strategy", agent=complex_agent)
```

✔️ DO: Use include_thoughts Appropriately

```python
# ✔️ Educational/debugging - show thinking
educational_agent = Agent(
    model='gemini-2.0-flash',
    planner=BuiltInPlanner(
        thinking_config=types.ThinkingConfig(include_thoughts=True)
    )
)

# ✔️ Production - hide thinking
production_agent = Agent(
    model='gemini-2.0-flash',
    planner=BuiltInPlanner(
        thinking_config=types.ThinkingConfig(include_thoughts=False)
    )
)
```

✔️ DO: Provide Clear Instructions with Planners

```python
# ✔️ Good - Clear guidance
agent = Agent(
    model='gemini-2.0-flash',
    planner=PlanReActPlanner(),
    instruction="""
You are a systematic problem solver.

When using tools:
1. Plan which tools to use and in what order
2. Explain your reasoning
3. Execute the plan
4. Review results
5. Adjust plan if needed

Be thorough but concise.
    """
)

# ❌ Bad - Vague
agent = Agent(
    model='gemini-2.0-flash',
    planner=PlanReActPlanner(),
    instruction="Solve problems"
)
```

✔️ DO: Test Planner Overhead

```python
# ✔️ Measure impact
import time

# Without planner
start = time.time()
result1 = runner.run(query, agent=agent_no_planner)
time1 = time.time() - start

# With planner
start = time.time()
result2 = runner.run(query, agent=agent_with_planner)
time2 = time.time() - start

overhead = ((time2 - time1) / time1) * 100
print(f"Planner overhead: {overhead:.1f}%")

# Accept overhead if quality improves significantly
```

✔️ DO: Handle Planning Failures

```python
# ✔️ Graceful fallback
agent_with_fallback = Agent(
    model='gemini-2.0-flash',
    planner=PlanReActPlanner(),
    instruction="""
Follow the planning format, but if you get stuck:
1. Acknowledge the difficulty
2. Provide best-effort answer
3. Explain limitations

Don't abandon the task completely.
    """
)
```

7. Troubleshooting

Issue: "Thinking not appearing in response"

Problem: Using BuiltInPlanner but no thinking shown

Solutions:

```
# ❌ Problem - include_thoughts=False (default)
agent = Agent(
    model='gemini-2.0-flash',
    planner=BuiltInPlanner()  # Defaults to include_thoughts=False
)

# ✔️ Solution - Explicitly set to True
agent = Agent(
    model='gemini-2.0-flash',
    planner=BuiltInPlanner(
        thinking_config=types.ThinkingConfig(include_thoughts=True)
    )
)

# Check model supports thinking
# Not all Gemini 2.0 models have thinking capability
```

Issue: "Plan-ReAct tags not appearing"

Problem: Response doesn't follow structured format

Solutions:

```python
# 1. Emphasize format in instruction
agent = Agent(
    model='gemini-2.0-flash',
    planner=PlanReActPlanner(),
    instruction="""
IMPORTANT: You MUST use the structured format with tags:
<PLANNING>, <REASONING>, <ACTION>, <FINAL_ANSWER>

Do not deviate from this format.
    """
)

# 2. Increase temperature for creativity in planning
agent = Agent(
    model='gemini-2.0-flash',
    planner=PlanReActPlanner(),
    generate_content_config=types.GenerateContentConfig(
        temperature=0.7  # Higher for creative planning
    )
)
```

Issue: "Planner adds too much latency"

Problem: Responses too slow with planner

Solutions:

```python
# 1. Reduce max_output_tokens
agent = Agent(
    model='gemini-2.0-flash',
    planner=PlanReActPlanner(),
    generate_content_config=types.GenerateContentConfig(
        max_output_tokens=1024  # Lower limit
    )
)

# 2. Use streaming for better UX
from google.adk.agents import RunConfig, StreamingMode

run_config = RunConfig(streaming_mode=StreamingMode.SSE)
async for event in runner.run_async(query, agent=agent, run_config=run_config)
    print(event.content.parts[0].text, end='', flush=True)

# 3. Use planner only for complex queries
def needs_planning(query: str) -> bool:
    complex_keywords = ['design', 'plan', 'strategy', 'analyze', 'compare']
    return any(kw in query.lower() for kw in complex_keywords)

agent = agent_with_planner if needs_planning(query) else agent_without_planner
```

Issue: "Replanning not triggered"

Problem: Agent doesn't adjust plan when encountering issues

Solutions:

```python
# 1. Explicit replanning instruction
agent = Agent(
    model='gemini-2.0-flash',
    planner=PlanReActPlanner(),
    instruction="""
When you encounter errors or unexpected results:
1. Use <OBSERVATION> to note what went wrong
2. Use <REPLANNING> to create new plan
3. Explain why replanning is needed

NEVER give up - always adapt your approach.
    """
)

# 2. Tool that forces replanning
def check_and_report(condition: bool, error_msg: str) -> str:
    if not condition:
        return f"ERROR: {error_msg}. Replanning needed."
    return "SUCCESS"

agent = Agent(
    model='gemini-2.0-flash',
    planner=PlanReActPlanner(),
    tools=[FunctionTool(check_and_report)]
)
```

8. Testing Planners

Unit Tests

```python
import pytest
from google.adk.agents import Agent, Runner
from google.adk.planners import BuiltInPlanner, PlanReActPlanner
from google.genai import types

@pytest.mark.asyncio
async def test_builtin_planner_shows_thinking():
    """Test that thinking appears when include_thoughts=True."""

    agent = Agent(
        model='gemini-2.0-flash',
        planner=BuiltInPlanner(
            thinking_config=types.ThinkingConfig(include_thoughts=True)
        )
    )

    runner = Runner()
    result = await runner.run_async(
        "Explain quantum entanglement",
        agent=agent
    )

    text = result.content.parts[0].text.lower()
    # Should contain thinking markers
    assert any(word in text for word in ['thinking', 'reasoning', 'consider'])

@pytest.mark.asyncio
async def test_planreact_planner_structure():
    """Test that Plan-ReAct planner produces structured output."""

    agent = Agent(
        model='gemini-2.0-flash',
        planner=PlanReActPlanner()
    )

    runner = Runner()
    result = await runner.run_async(
        "Create a 3-step plan to learn Python",
        agent=agent
    )

    text = result.content.parts[0].text
    # Should contain planning tags
    assert '<PLANNING>' in text or '<PLAN>' in text
    assert '<REASONING>' in text or '<FINAL_ANSWER>' in text
```

```
@pytest.mark.asyncio
async def test_planner_improves_complex_task():
    """Test that planner improves quality on complex task."""

    complex_query = "Design a machine learning system for fraud detection"

    # Without planner
    agent_no_planner = Agent(
        model='gemini-2.0-flash',
        name='no_planner'
    )

    # With planner
    agent_with_planner = Agent(
        model='gemini-2.0-flash',
        name='with_planner',
        planner=PlanReActPlanner()
    )

    runner = Runner()

    result_no_planner = await runner.run_async(complex_query, agent=agent_no_p
    result_with_planner = await runner.run_async(complex_query, agent=agent_wi

    # Planned response should be more comprehensive
    assert len(result_with_planner.content.parts[0].text) > len(result_no_plan

    # Planned response should mention key ML concepts
    planner_text = result_with_planner.content.parts[0].text.lower()
    ml_concepts = ['training', 'model', 'features', 'validation', 'accuracy']
    concepts_mentioned = sum(1 for concept in ml_concepts if concept in planne
    assert concepts_mentioned >= 3  # Should mention at least 3 ML concepts
```

Summary

You've mastered advanced reasoning with planners and thinking configuration:

Key Takeaways:

- ✔️ `BuiltInPlanner` uses Gemini 2.0+ native thinking for transparent reasoning
- ✔️ `ThinkingConfig` controls whether thinking is shown (`include_thoughts`)

- ✅ `PlanReActPlanner` provides structured Plan → Reason → Act → Observe → Replan flow
- ✅ Planning tags (`<PLANNING>` , `<REASONING>` , `<ACTION>` , etc.) structure output
- ✅ `BasePlanner` enables custom planning strategies
- ✅ Planners add latency but improve quality on complex tasks
- ✅ Choose planner based on task complexity and requirements

Production Checklist:

- [ ] Using appropriate planner for task complexity
- [ ] ThinkingConfig set correctly (show/hide based on use case)
- [ ] Clear instructions for planning behavior
- [ ] Tested planner overhead vs quality improvement
- [ ] Fallback handling for planning failures
- [ ] Streaming enabled if latency is concern
- [ ] Model supports planning features (Gemini 2.0+)

Next Steps:

- **Tutorial 13**: Learn Code Execution for agents that can write and run Python
- **Tutorial 14**: Implement Streaming for real-time response generation
- **Tutorial 15**: Explore Live API for voice and bidirectional streaming

Resources:

- ADK Planners Documentation (https://google.github.io/adk-docs/agents/planners/)
- Gemini Thinking Guide (https://cloud.google.com/vertex-ai/generative-ai/docs/model-reference/gemini)
- Plan-ReAct Pattern (https://arxiv.org/abs/2210.03629)

🎉 **Tutorial 12 Complete!** You now know how to build agents with advanced reasoning capabilities. Continue to Tutorial 13 to learn about code execution.

Generated on 2025-10-21 09:02:25 from 12_planners_thinking.md

Source: Google ADK Training Hub