Tutorial 22: Model Selection - Choosing the Right AI Model

Difficulty: intermediate

Reading Time: 1.5 hours

Tags: intermediate, models, optimization, configuration, performance

Description: Learn to select and configure different AI models including Gemini variants, optimization strategies, and model-specific configurations.

:::info Verified Against Official Sources

This tutorial has been verified against official Google AI documentation and ADK source code.

Verification Date: October 12, 2025
ADK Version: 1.16.0+
Sources Checked:

- ADK Python source code (`llm_agent.py`)
- Official Gemini API documentation (https://ai.google.dev/gemini-api/docs/models)
- Vertex AI Gemini documentation

:::

Tutorial 22: Model Selection & Optimization

Goal: Master model selection strategies, understand model capabilities and limitations, optimize costs and performance, and choose the right model for specific use cases.

Prerequisites:

- Tutorial 01 (Hello World Agent)
- Understanding of basic agent concepts
- Familiarity with different agent capabilities

What You'll Learn:

- Gemini model family overview and comparison
- Model capability matrix (vision, thinking, code execution, etc.)
- Performance vs cost tradeoffs
- Context window and token limits
- Model selection decision framework
- Testing and benchmarking strategies
- Migration strategies between models

Time to Complete: 45-60 minutes

Why Model Selection Matters

Problem: Different models have different capabilities, costs, and performance characteristics. Choosing wrong model leads to poor results or unnecessary costs.

Solution: **Strategic model selection** based on requirements, use case, budget, and performance needs.

Benefits:

- 💰 **Cost Optimization**: Pay only for capabilities you need
- ⚡ **Performance**: Right speed for your application
- 🎯 **Capability Match**: Models with features you require
- 📊 **Quality**: Best results for your specific use case
- [FLOW] **Scalability**: Models that handle your load

Decision Factors:

- Task complexity
- Response time requirements

- Budget constraints
- Feature requirements (vision, thinking, code execution)
- Context window needs
- Deployment environment

1. Gemini Model Family Overview

Current Model Lineup (2025)

Source: ADK supports all Gemini models via Google AI and Vertex AI

⚠️ **IMPORTANT**: As of October 2025, **Gemini 2.5 Flash is RECOMMENDED** for new agents due to its excellent price-performance ratio. Note that ADK's default model parameter is an empty string (which inherits from parent agent), so **always specify the model explicitly**.

Model	Context Window	Key Features	Best For	Status
gemini-2.5-flash ⭐	1M tokens	**RECOMMENDED**, thinking, fast, multimodal	**General purpose**, production	**Stable**
gemini-2.5-pro	1M tokens	State-of-the-art, complex reasoning, STEM	Critical analysis, research	**Stable**
gemini-2.5-flash-lite	1M tokens	Ultra-fast, cost-efficient, high throughput	High-volume, simple tasks	**Preview**
gemini-2.0-flash	1M tokens	Fast, thinking, code execution	General purpose (legacy)	Stable
gemini-2.0-flash-thinking	1M tokens	Extended thinking mode	Complex reasoning (legacy)	Stable
gemini-1.5-flash	1M tokens	Fast, cost-effective	High-volume (legacy)	Stable
gemini-1.5-flash-8b	1M tokens	Ultra-fast, economical	Simple queries (legacy)	Stable
gemini-1.5-pro	2M tokens	Extended context	Large documents (legacy)	Stable
gemini-2.0-flash-live	Streaming	Bidirectional audio/video (Vertex)	Real-time conversations	Preview
gemini-live-2.5-flash	Streaming	Live API (AI Studio)	Voice assistants	Preview

Model Generations:

- **2.5 Series** (Latest, October 2025): First with native thinking, image generation, best price-performance
- **2.0 Series** (December 2024): Second generation, code execution, Google Search

- **1.5 Series** (Early 2024): First generation, multimodal foundation

🆕 What's New in Gemini 2.5

Gemini 2.5 Flash is our best model in terms of price-performance:

- ✔️ **Native Thinking Capabilities**: See the model's reasoning process
- ✔️ **Image Generation**: Generate and edit images natively (2.5 Flash Image variant)
- ✔️ **Long Context**: 1 million token context window
- ✔️ **Multimodal**: Text, image, audio, video understanding
- ✔️ **Well-Rounded**: Excels at coding, reasoning, creative writing
- ✔️ **Best for Agents**: Optimized for large-scale processing and agentic use cases

Gemini 2.5 Pro is the state-of-the-art thinking model:

- ✔️ **Advanced Reasoning**: Complex problems in code, math, STEM
- ✔️ **Long Context Analysis**: Analyze large datasets, codebases, documents
- ✔️ **Highest Quality**: Best-in-class for critical applications
- ✔️ **Research Grade**: Suitable for academic and professional research

Official Documentation:

- Google AI: https://ai.google.dev/gemini-api/docs/models
- Vertex AI: https://cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-5-flash
- Technical Report: https://storage.googleapis.com/deepmind-media/gemini/gemini_v2_5_report.pdf

Model Comparison Matrix

```python
"""
Model capability and pricing comparison.
"""

MODELS = {
    # === GEMINI 2.5 SERIES (Latest - October 2025) ===
    'gemini-2.5-flash': {
        'context_window': 1_000_000,
        'features': ['vision', 'thinking', 'code_execution', 'audio', 'video',
        'speed': 'fast',
        'cost': 'low',
        'quality': 'excellent',
        'is_recommended': True,  # RECOMMENDED model for new projects
        'generation': '2.5',
        'recommended_for': [
            '⭐ RECOMMENDED for all new agents',
            'General agent applications',
            'Production systems',
            'Agentic workflows',
            'Best price-performance'
        ],
        'note': 'First Flash model with native thinking capabilities'
    },
    'gemini-2.5-pro': {
        'context_window': 1_000_000,
        'features': ['vision', 'thinking', 'code_execution', 'audio', 'video',
        'speed': 'moderate',
        'cost': 'high',
        'quality': 'state_of_the_art',
        'generation': '2.5',
        'recommended_for': [
            'Complex reasoning tasks',
            'STEM problems (code, math, physics)',
            'Research and analysis',
            'Critical applications requiring highest quality'
        ],
        'note': 'State-of-the-art thinking model'
    },
    'gemini-2.5-flash-lite': {
        'context_window': 1_000_000,
        'features': ['vision', 'audio', 'video', 'ultra_fast'],
        'speed': 'ultra_fast',
        'cost': 'ultra_low',
        'quality': 'good',
        'generation': '2.5',
        'recommended_for': [
            'Ultra high-throughput applications',
            'Simple queries at massive scale',
            'Cost-sensitive deployments',
            'Real-time low-latency tasks'
        ],
        'note': 'Fastest flash model, optimized for cost-efficiency'
    },

    # === GEMINI 2.0 SERIES (Legacy) ===
    'gemini-2.0-flash': {
        'context_window': 1_000_000,
        'features': ['vision', 'thinking', 'code_execution', 'audio', 'video']
        'speed': 'fast',
        'cost': 'low',
        'quality': 'high',
        'generation': '2.0',
        'recommended_for': [
            'General agent applications',
            'Production systems',
            'Multi-agent workflows',
            'Complex reasoning'
        ],
        'note': 'Consider upgrading to gemini-2.5-flash'
    },
    'gemini-2.0-flash-thinking': {
        'context_window': 1_000_000,
        'features': ['vision', 'thinking', 'code_execution', 'extended_reasoni
        'speed': 'moderate',
        'cost': 'moderate',
        'quality': 'very_high',
        'generation': '2.0',
        'recommended_for': [
            'Strategic planning',
            'Complex problem solving',
            'Research analysis',
            'Deep reasoning tasks'
        ],
        'note': 'Consider gemini-2.5-pro for better reasoning'
    },

    # === GEMINI 1.5 SERIES (Legacy) ===
    'gemini-1.5-flash': {
        'context_window': 1_000_000,
        'features': ['vision', 'audio', 'video'],
        'speed': 'very_fast',
        'cost': 'very_low',
        'quality': 'good',
        'generation': '1.5',
        'recommended_for': [
            'High-volume applications',
            'Simple queries',
            'Content moderation',
            'Quick responses'
        ],
        'note': 'Consider gemini-2.5-flash for better performance at similar c
    },
    'gemini-1.5-flash-8b': {
        'context_window': 1_000_000,
        'features': ['vision', 'audio'],
        'speed': 'ultra_fast',
        'cost': 'ultra_low',
        'quality': 'moderate',
        'generation': '1.5',
        'recommended_for': [
            'Ultra high-throughput',
            'Simple classification',
            'Basic Q&A',
            'Cost-sensitive applications'
        ],
        'note': 'Consider gemini-2.5-flash-lite for better performance'
    },
    'gemini-1.5-pro': {
        'context_window': 2_000_000,
        'features': ['vision', 'audio', 'video', 'extended_context'],
        'speed': 'moderate',
        'cost': 'high',
        'quality': 'excellent',
        'generation': '1.5',
        'recommended_for': [
            'Critical business applications',
            'Complex analysis',
            'Large document processing',
            'Highest quality requirements'
        ],
        'note': 'Consider gemini-2.5-pro unless you need 2M token context'
    },

    # === STREAMING MODELS ===
    'gemini-2.0-flash-live': {
        'context_window': 'streaming',
        'features': ['vision', 'audio', 'video', 'bidirectional', 'real_time']
        'speed': 'real_time',
        'cost': 'moderate',
        'quality': 'high',
        'generation': '2.0',
        'recommended_for': [
            'Voice assistants',
            'Real-time conversation',
            'Live video analysis',
            'Interactive applications'
        ],
        'note': 'Vertex AI only'
    }
}

def recommend_model(requirements: dict) -> list:
    """
    Recommend models based on requirements.

    Args:
        requirements: Dict with keys like 'features', 'speed', 'cost', 'qualit

    Returns:
        List of recommended model names with reasons
    """

    recommendations = []

    required_features = set(requirements.get('features', []))
    speed_pref = requirements.get('speed', 'any')
    cost_pref = requirements.get('cost', 'any')
    quality_pref = requirements.get('quality', 'any')

    for model_name, model_info in MODELS.items():
        # Check feature compatibility
        model_features = set(model_info['features'])
        if required_features and not required_features.issubset(model_features
            continue

        # Check speed preference
        if speed_pref != 'any' and model_info['speed'] != speed_pref:
            continue

        # Check cost preference
        if cost_pref != 'any' and model_info['cost'] != cost_pref:
            continue

        # Check quality preference
        if quality_pref != 'any' and model_info['quality'] != quality_pref:
            continue
            recommendations.append({
                'model': model_name,
                'reason': model_info['recommended_for'][0],
                'features': model_info['features'],
                'speed': model_info['speed'],
                'cost': model_info['cost']
            })

        return recommendations

# Example usage
requirements = {
    'features': ['vision', 'thinking'],
    'speed': 'fast',
    'cost': 'low'
}

recommended = recommend_model(requirements)
for rec in recommended:
    print(f"✔️ {rec['model']}")
    print(f"   Reason: {rec['reason']}")
    print(f"   Speed: {rec['speed']}, Cost: {rec['cost']}")
```

2. Feature Compatibility

Built-in Tools & Features

Gemini 2.0+ Required:

- ✔️ Thinking configuration (`types.ThinkingConfig`)
- ✔️ Built-in code execution (`BuiltInCodeExecutor`)
- ✔️ Google Search grounding (native)
- ✔️ Enhanced multimodal capabilities

All Gemini Models:

- ✔️ Function calling
- ✔️ Vision (image understanding)
- ✔️ Basic multimodal (text + images)

- ✔️ Custom tools

Live API Models Only:

- ✔️ Bidirectional streaming
- ✔️ Real-time audio/video
- ✔️ Proactive responses
- ✔️ Affective dialog (emotion detection)

Feature Compatibility Table

```
FEATURE_COMPATIBILITY = {
    'function_calling': ['all'],
    'vision': ['all'],
    'audio_input': ['all'],

    # Gemini 2.5+ features
    'thinking_config': [
        'gemini-2.5-flash',      # NEW: First Flash with thinking!
        'gemini-2.5-pro',
        'gemini-2.0-flash',
        'gemini-2.0-flash-thinking'
    ],
    'image_generation': [
        'gemini-2.5-flash',      # NEW: Native image generation
        'gemini-2.5-flash-image'
    ],
    'code_execution': [
        'gemini-2.5-flash',
        'gemini-2.5-pro',
        'gemini-2.0-flash',
        'gemini-2.0-flash-thinking'
    ],
    'google_search': [
        'gemini-2.5-flash',
        'gemini-2.5-pro',
        'gemini-2.0-flash',
        'gemini-2.0-flash-thinking'
    ],

    # Video support
    'video_input': [
        'gemini-2.5-flash',
        'gemini-2.5-pro',
        'gemini-1.5-pro',
        'gemini-1.5-flash',
        'gemini-2.0-flash',
        'gemini-2.0-flash-live'
    ],

    # Streaming
    'bidirectional_streaming': [
        'gemini-2.0-flash-live',
        'gemini-live-2.5-flash'
    ],

    # Context windows
```

```python
    'extended_context': ['gemini-1.5-pro'],  # 2M tokens
    'long_context': [  # 1M tokens
        'gemini-2.5-flash',
        'gemini-2.5-pro',
        'gemini-2.5-flash-lite',
        'gemini-2.0-flash',
        'gemini-1.5-flash',
        'gemini-1.5-flash-8b'
    ],

    # Speed tiers
    'ultra_fast': ['gemini-2.5-flash-lite', 'gemini-1.5-flash-8b']
}

def check_feature_support(model: str, feature: str) -> bool:
    """Check if model supports feature."""

    if feature not in FEATURE_COMPATIBILITY:
        return False

    supported_models = FEATURE_COMPATIBILITY[feature]

    if 'all' in supported_models:
        return True

    return model in supported_models

# Examples
print(check_feature_support('gemini-2.5-flash', 'thinking_config'))  # True ✔
print(check_feature_support('gemini-2.0-flash', 'thinking_config'))  # True ✔
print(check_feature_support('gemini-1.5-flash', 'thinking_config'))  # False ❌
print(check_feature_support('gemini-2.5-flash', 'image_generation'))  # True ✔
```

3. Real-World Example: Model Selection Framework

Let's build a comprehensive model selection and testing framework.

Complete Implementation

```python
"""
Model Selection and Testing Framework
Helps choose the right model and benchmark performance.
"""

import asyncio
import time
from dataclasses import dataclass
from typing import Dict, List
from google.adk.agents import Agent, Runner
from google.genai import types

@dataclass
class ModelBenchmark:
    """Benchmark results for a model."""
    model: str
    avg_latency: float
    avg_tokens: int
    quality_score: float
    cost_estimate: float
    success_rate: float

class ModelSelector:
    """Framework for selecting and benchmarking models."""

    def __init__(self):
        """Initialize model selector."""
        self.runner = Runner()
        self.benchmarks: Dict[str, ModelBenchmark] = {}

    async def benchmark_model(
        self,
        model: str,
        test_queries: List[str],
        instruction: str
    ) -> ModelBenchmark:
        """
        Benchmark a model on test queries.

        Args:
            model: Model to test
            test_queries: List of test queries
            instruction: Agent instruction

        Returns:
            ModelBenchmark with results
        """

        print(f"\n{'='*70}")
        print(f"BENCHMARKING: {model}")
        print(f"{'='*70}\n")

        # Create agent with this model
        agent = Agent(
            model=model,
            name=f'test_agent_{model.replace(".", "_")}',
            instruction=instruction,
            generate_content_config=types.GenerateContentConfig(
                temperature=0.5,
                max_output_tokens=1024
            )
        )

        latencies = []
        token_counts = []
        successes = 0

        for query in test_queries:
            try:
                start = time.time()

                result = await self.runner.run_async(query, agent=agent)

                latency = time.time() - start
                latencies.append(latency)

                # Estimate token count (rough)
                text = result.content.parts[0].text
                token_count = len(text.split())
                token_counts.append(token_count)

                successes += 1

                print(f"✔️ Query: {query[:50]}...")
                print(f"   Latency: {latency:.2f}s, Tokens: ~{token_count}")

            except Exception as e:
                print(f"❌ Query failed: {query[:50]}... - {e}")

        # Calculate metrics
        avg_latency = sum(latencies) / len(latencies) if latencies else 0
        avg_tokens = sum(token_counts) / len(token_counts) if token_counts els
        success_rate = successes / len(test_queries)
        # Estimate cost (simplified)
        # Real pricing varies - check Google Cloud pricing
        cost_per_1k_tokens = {
            'gemini-2.0-flash': 0.0001,
            'gemini-1.5-flash': 0.00008,
            'gemini-1.5-flash-8b': 0.00004,
            'gemini-1.5-pro': 0.0005
        }

        model_key = model
        if model_key not in cost_per_1k_tokens:
            model_key = 'gemini-2.0-flash'

        cost_estimate = (avg_tokens / 1000) * cost_per_1k_tokens[model_key]

        # Quality score (based on success rate and latency)
        quality_score = success_rate * (1.0 / (1.0 + avg_latency))

        benchmark = ModelBenchmark(
            model=model,
            avg_latency=avg_latency,
            avg_tokens=int(avg_tokens),
            quality_score=quality_score,
            cost_estimate=cost_estimate,
            success_rate=success_rate
        )

        self.benchmarks[model] = benchmark

        print(f"\n📊 RESULTS:")
        print(f"   Avg Latency: {avg_latency:.2f}s")
        print(f"   Avg Tokens: {avg_tokens:.0f}")
        print(f"   Success Rate: {success_rate*100:.1f}%")
        print(f"   Cost Estimate: ${cost_estimate:.6f} per query")
        print(f"   Quality Score: {quality_score:.3f}")

        return benchmark

    async def compare_models(
        self,
        models: List[str],
        test_queries: List[str],
        instruction: str
    ):
        """
        Compare multiple models on same queries.
        Args:
            models: List of models to compare
            test_queries: Test queries
            instruction: Agent instruction
        """

        print(f"\n{'#'*70}")
        print(f"MODEL COMPARISON")
        print(f"{'#'*70}\n")

        for model in models:
            await self.benchmark_model(model, test_queries, instruction)
            await asyncio.sleep(2)

        self._print_comparison()

    def _print_comparison(self):
        """Print comparison table."""

        print(f"\n\n{'='*70}")
        print("COMPARISON SUMMARY")
        print(f"{'='*70}\n")

        print(f"{'Model':<30} {'Latency':>10} {'Tokens':>8} {'Cost':>10} {'Qua
        print(f"{'-'*70}")

        for model, bench in self.benchmarks.items():
            print(f"{model:<30} {bench.avg_latency:>9.2f}s {bench.avg_tokens:>
                  f"${bench.cost_estimate:>9.6f} {bench.quality_score:>10.3f}"

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

        # Recommendations
        print("\n🎯 RECOMMENDATIONS:\n")

        fastest = min(self.benchmarks.items(), key=lambda x: x[1].avg_latency)
        print(f"⚡ Fastest: {fastest[0]} ({fastest[1].avg_latency:.2f}s)")

        cheapest = min(self.benchmarks.items(), key=lambda x: x[1].cost_estima
        print(f"💰 Cheapest: {cheapest[0]} (${cheapest[1].cost_estimate:.6f})"

        best_quality = max(self.benchmarks.items(), key=lambda x: x[1].quality
        print(f"🏆 Best Quality: {best_quality[0]} ({best_quality[1].quality_s

    def recommend_model_for_use_case(self, use_case: str) -> str:
        """
        Recommend model based on use case (Updated for Gemini 2.5).

        Args:
            use_case: Use case description

        Returns:
            Recommended model name
        """

        use_case_lower = use_case.lower()

        # Rule-based recommendations (Gemini 2.5 series)
        if 'real-time' in use_case_lower or 'voice' in use_case_lower:
            return 'gemini-2.0-flash-live'

        elif 'complex' in use_case_lower or 'reasoning' in use_case_lower or '
            return 'gemini-2.5-pro'  # NEW: Best for complex problems

        elif 'high-volume' in use_case_lower or 'simple' in use_case_lower or
            return 'gemini-2.5-flash-lite'  # NEW: Fastest + cheapest

        elif 'critical' in use_case_lower or 'important' in use_case_lower:
            return 'gemini-2.5-pro'  # NEW: Highest quality

        elif 'extended context' in use_case_lower or 'large document' in use_c
            return 'gemini-1.5-pro'  # Still has 2M context

        else:
            return 'gemini-2.5-flash'  # NEW DEFAULT!

async def main():
    """Main entry point."""

    selector = ModelSelector()

    # Test queries
    test_queries = [
        "What is the capital of France?",
        "Explain quantum computing in simple terms",
        "Write a haiku about artificial intelligence",
        "Calculate the compound interest on $10,000 at 5% for 10 years",
        "List the top 5 programming languages in 2025"
    ]

    instruction = """
You are a helpful assistant. Answer questions accurately and concisely.
    """.strip()
    # Compare models (Updated for Gemini 2.5)
    models_to_test = [
        'gemini-2.5-flash',       # NEW DEFAULT - Best price-performance
        'gemini-2.5-pro',         # NEW - Highest quality
        'gemini-2.5-flash-lite', # NEW - Ultra-fast
        'gemini-2.0-flash',       # Legacy
        'gemini-1.5-flash',       # Legacy
    ]

    await selector.compare_models(models_to_test, test_queries, instruction)

    # Use case recommendations
    print(f"\n\n{'='*70}")
    print("USE CASE RECOMMENDATIONS")
    print(f"{'='*70}\n")

    use_cases = [
        "Real-time voice assistant",
        "Complex strategic planning",
        "High-volume content moderation",
        "Critical business decision support",
        "General customer service"
    ]

    for use_case in use_cases:
        recommendation = selector.recommend_model_for_use_case(use_case)
        print(f"📌 {use_case}")
        print(f"   → Recommended: {recommendation}\n")

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

Expected Output

```
======================================================================
BENCHMARKING: gemini-2.0-flash
======================================================================

✓ Query: What is the capital of France?...
   Latency: 0.85s, Tokens: ~8
✓ Query: Explain quantum computing in simple terms...
   Latency: 1.23s, Tokens: ~95
✓ Query: Write a haiku about artificial intelligence...
   Latency: 0.92s, Tokens: ~25
✓ Query: Calculate the compound interest on $10,000 at 5% ...
   Latency: 1.15s, Tokens: ~42
✓ Query: List the top 5 programming languages in 2025...
   Latency: 0.98s, Tokens: ~35

📊 RESULTS:
   Avg Latency: 1.03s
   Avg Tokens: 41
   Success Rate: 100.0%
   Cost Estimate: $0.000004 per query
   Quality Score: 0.493

======================================================================
BENCHMARKING: gemini-1.5-flash
======================================================================

✓ Query: What is the capital of France?...
   Latency: 0.72s, Tokens: ~7
✓ Query: Explain quantum computing in simple terms...
   Latency: 1.05s, Tokens: ~88
✓ Query: Write a haiku about artificial intelligence...
   Latency: 0.78s, Tokens: ~22
✓ Query: Calculate the compound interest on $10,000 at 5% ...
   Latency: 0.95s, Tokens: ~38
✓ Query: List the top 5 programming languages in 2025...
   Latency: 0.82s, Tokens: ~32

📊 RESULTS:
   Avg Latency: 0.86s
   Avg Tokens: 37
   Success Rate: 100.0%
   Cost Estimate: $0.000003 per query
   Quality Score: 0.537

======================================================================
BENCHMARKING: gemini-1.5-flash-8b
```

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

✓ Query: What is the capital of France?...
   Latency: 0.58s, Tokens: ~6
✓ Query: Explain quantum computing in simple terms...
   Latency: 0.89s, Tokens: ~75
✓ Query: Write a haiku about artificial intelligence...
   Latency: 0.65s, Tokens: ~20
✓ Query: Calculate the compound interest on $10,000 at 5% ...
   Latency: 0.78s, Tokens: ~32
✓ Query: List the top 5 programming languages in 2025...
   Latency: 0.68s, Tokens: ~28

📊 RESULTS:
   Avg Latency: 0.72s
   Avg Tokens: 32
   Success Rate: 100.0%
   Cost Estimate: $0.000001 per query
   Quality Score: 0.581

======================================================================
COMPARISON SUMMARY
======================================================================

Model                               Latency   Tokens        Cost    Quality
----------------------------------------------------------------------
gemini-2.0-flash                       1.03s       41  $0.000004      0.493
gemini-1.5-flash                       0.86s       37  $0.000003      0.537
gemini-1.5-flash-8b                    0.72s       32  $0.000001      0.581

======================================================================

🎯 RECOMMENDATIONS:

⚡ Fastest: gemini-1.5-flash-8b (0.72s)
💰 Cheapest: gemini-1.5-flash-8b ($0.000001)
🏆 Best Quality: gemini-1.5-flash-8b (0.581)

======================================================================
USE CASE RECOMMENDATIONS
======================================================================

📌 Real-time voice assistant
   → Recommended: gemini-2.0-flash-live

📌 Complex strategic planning
   → Recommended: gemini-2.0-flash-thinking
```

```
📌 High-volume content moderation
   → Recommended: gemini-1.5-flash-8b

📌 Critical business decision support
   → Recommended: gemini-1.5-pro

📌 General customer service
   → Recommended: gemini-2.0-flash
```

4. Model Selection Decision Tree

Decision Framework

```python
def select_model(requirements: dict) -> str:
    """
    Model selection decision tree (Updated for Gemini 2.5).

    Args:
        requirements: Dictionary with:
            - real_time: bool
            - complex_reasoning: bool (STEM, math, code)
            - high_volume: bool
            - vision_required: bool
            - code_execution: bool
            - budget_sensitive: bool
            - ultra_fast: bool
            - critical: bool

    Returns:
        Recommended model name
    """

    # Real-time requirements (streaming)
    if requirements.get('real_time', False):
        return 'gemini-2.0-flash-live'

    # Complex reasoning (STEM, research, deep analysis)
    if requirements.get('complex_reasoning', False):
        return 'gemini-2.5-pro'  # NEW: Best for complex problems

    # Ultra-fast requirements with budget constraints
    if requirements.get('ultra_fast', False) and requirements.get('budget_sens
        return 'gemini-2.5-flash-lite'  # NEW: Fastest + cheapest

    # High volume + budget sensitive (simple tasks)
    if requirements.get('high_volume', False) and requirements.get('budget_sen
        return 'gemini-2.5-flash-lite'  # NEW: Replaces 1.5-flash-8b

    # Critical applications requiring highest quality
    if requirements.get('critical', False):
        return 'gemini-2.5-pro'  # NEW: State-of-the-art quality

    # Extended context (>1M tokens)
    if requirements.get('extended_context', False):
        return 'gemini-1.5-pro'  # Still has 2M token context

    # Default: Best price-performance (NEW DEFAULT!)
    return 'gemini-2.5-flash'
```

```
# Examples (Updated for 2.5)
print(select_model({'real_time': True}))
# Output: gemini-2.0-flash-live

print(select_model({'complex_reasoning': True}))
# Output: gemini-2.5-pro  ← Changed from 2.0-flash-thinking

print(select_model({'high_volume': True, 'budget_sensitive': True}))
# Output: gemini-2.5-flash-lite  ← Changed from 1.5-flash-8b

print(select_model({}))  # No requirements
# Output: gemini-2.5-flash  ← NEW DEFAULT!
```

---

5. Using Other LLMs with LiteLLM

Source: `google/adk/models/lite_llm.py`

While Gemini models are optimized for ADK and offer the best integration, you can use **any LLM provider** through LiteLLM. The `LiteLlm` class wraps the LiteLLM library to provide unified access to OpenAI, Anthropic, Ollama, Azure, and more.

🌟 Why Use LiteLLM?

- ✅ **Provider Flexibility**: Switch between OpenAI, Claude, Ollama, Azure without code changes
- ✅ **Cost Optimization**: Compare providers and choose the best price-performance
- ✅ **Local Models**: Run Ollama models locally for privacy/compliance
- ✅ **Fallback Strategy**: Use multiple providers for reliability
- ✅ **Unified Interface**: Same ADK code works with any LLM

Basic LiteLLM Integration

```python
from google.adk.models.lite_llm import LiteLlm
from google.adk.agents import Agent

# Create agent with OpenAI GPT-4
agent = Agent(
    model=LiteLlm(model='openai/gpt-4o'),
    name='openai_agent',
    instruction='You are a helpful assistant.'
)

# Or with Anthropic Claude
agent = Agent(
    model=LiteLlm(model='anthropic/claude-3-7-sonnet'),
    name='claude_agent',
    instruction='You are a helpful assistant.'
)
```

Supported Providers

OpenAI

```python
from google.adk.models.lite_llm import LiteLlm
import os

# Set API key
os.environ['OPENAI_API_KEY'] = 'sk-...'

# Use GPT-4o
model = LiteLlm(model='openai/gpt-4o')

# Use GPT-4o-mini (faster, cheaper)
model = LiteLlm(model='openai/gpt-4o-mini')

# Use GPT-3.5-turbo
model = LiteLlm(model='openai/gpt-3.5-turbo')
```

When to use:

- Need GPT-4 specifically for compatibility with existing systems

- OpenAI-specific features like DALL-E integration
- Cost comparison (GPT-4o-mini can be cheaper than Gemini Pro)

Anthropic Claude

```python
import os

# Set API key
os.environ['ANTHROPIC_API_KEY'] = 'sk-ant-...'

# Use Claude 3.7 Sonnet
model = LiteLlm(model='anthropic/claude-3-7-sonnet')

# Use Claude 3 Opus (highest quality)
model = LiteLlm(model='anthropic/claude-3-opus')

# Use Claude 3 Haiku (fastest, cheapest)
model = LiteLlm(model='anthropic/claude-3-haiku')
```

When to use:

- Need Claude-specific capabilities (long-form writing, code analysis)
- Anthropic's constitutional AI approach
- Cost comparison for specific tasks

Ollama (Local Models)

```python
import os

# Set Ollama base URL
os.environ['OLLAMA_API_BASE'] = 'http://localhost:11434'

# ⚠️ CRITICAL: Use 'ollama_chat' prefix, NOT 'ollama'
model = LiteLlm(model='ollama_chat/llama3.3')

# Other popular models
model = LiteLlm(model='ollama_chat/mistral-small3.1')
model = LiteLlm(model='ollama_chat/codellama')
model = LiteLlm(model='ollama_chat/phi4')
```

Common pitfall:

```python
# ❌ WRONG - Will fail with cryptic errors
model = LiteLlm(model='ollama/llama3.3')

# ✔️ CORRECT - Must use 'ollama_chat'
model = LiteLlm(model='ollama_chat/llama3.3')
```

When to use:

- Privacy requirements (data stays local)
- Compliance regulations (no data sent to cloud)
- Cost savings (no API charges)
- Offline/air-gapped environments

Sample: `contributing/samples/hello_world_ollama/agent.py`

Azure OpenAI

```python
import os

# Set Azure credentials
os.environ['AZURE_API_KEY'] = 'your-azure-key'
os.environ['AZURE_API_BASE'] = 'https://your-resource.openai.azure.com/'
os.environ['AZURE_API_VERSION'] = '2024-02-01'

# Use Azure-hosted GPT-4
model = LiteLlm(model='azure/gpt-4')

# Use Azure-hosted GPT-35-turbo
model = LiteLlm(model='azure/gpt-35-turbo')
```

When to use:

- Enterprise Azure contracts
- Azure compliance requirements
- Integration with Azure services

Claude via Vertex AI

```python
# Use Anthropic Claude through Google Cloud
model = LiteLlm(model='vertex_ai/claude-3-7-sonnet')
```

When to use:

- Existing Google Cloud setup
- Need Claude but prefer Google billing
- Combined Gemini + Claude workflows

Complete Example: Multi-Provider Agent System

```python
"""
Example: Compare responses from multiple LLM providers.
Source: contributing/samples/hello_world_litellm/agent.py
"""
from google.adk.models.lite_llm import LiteLlm
from google.adk.agents import Agent
import asyncio

async def compare_providers():
    """Compare the same query across multiple providers."""

    query = "Explain quantum computing in 2 sentences."

    # Create agents with different providers
    gemini_agent = Agent(
        model='gemini-2.5-flash',  # Native Gemini (recommended)
        name='gemini_agent'
    )

    openai_agent = Agent(
        model=LiteLlm(model='openai/gpt-4o'),
        name='openai_agent'
    )

    claude_agent = Agent(
        model=LiteLlm(model='anthropic/claude-3-7-sonnet'),
        name='claude_agent'
    )

    ollama_agent = Agent(
        model=LiteLlm(model='ollama_chat/llama3.3'),
        name='ollama_agent'
    )

    # Query all providers in parallel
    responses = await asyncio.gather(
        gemini_agent.run_async(query),
        openai_agent.run_async(query),
        claude_agent.run_async(query),
        ollama_agent.run_async(query),
        return_exceptions=True
    )

    # Compare results
    for agent_name, response in zip(
        ['Gemini', 'OpenAI', 'Claude', 'Ollama'],
        responses
    ):
        if isinstance(response, Exception):
            print(f"{agent_name}: ERROR - {response}")
        else:
            print(f"\n{agent_name}:")
            print(response.output_text)
            print(f"Time: {response.usage.time_ms}ms")
            print(f"Tokens: {response.usage.total_tokens}")

# Run comparison
asyncio.run(compare_providers())
```

⚠️ Important Warnings

DON'T: Use Gemini via LiteLLM

```
# ❌ WRONG - Don't use Gemini through LiteLLM
agent = Agent(
    model=LiteLlm(model='gemini/gemini-2.5-flash'),
    name='bad_agent'
)

# ✔️ CORRECT - Use native Gemini class
agent = Agent(
    model='gemini-2.5-flash',  # Direct string
    name='good_agent'
)
```

Why: Native Gemini integration is faster, more reliable, and supports ADK-specific features (thinking config, code execution, function calling optimizations).

DON'T: Forget ollama_chat Prefix

```
# ❌ WRONG - Will fail
model = LiteLlm(model='ollama/llama3.3')

# ✔️ CORRECT
model = LiteLlm(model='ollama_chat/llama3.3')
```

DO: Set Environment Variables

```
# ✔️ Good - Set credentials before creating agent
import os

os.environ['OPENAI_API_KEY'] = 'sk-...'
os.environ['ANTHROPIC_API_KEY'] = 'sk-ant-...'
os.environ['OLLAMA_API_BASE'] = 'http://localhost:11434'

# Now create agents
agent = Agent(model=LiteLlm(model='openai/gpt-4o'))
```

Installation

LiteLLM support is built-in to ADK:

```
# LiteLLM is included in ADK installation
pip install google-adk

# For Ollama, install separately
# Visit https://ollama.com to download
ollama pull llama3.3
ollama pull mistral-small3.1
```

When to Use Each Approach

Use Native Gemini (model='gemini-2.5-flash'):

- ✅ Default choice for new agents
- ✅ Best performance and features
- ✅ Lowest latency
- ✅ ADK-optimized (thinking, code execution, function calling)
- ✅ Best price-performance

Use LiteLLM (model=LiteLlm(...)):

- ✅ Need specific provider (GPT-4, Claude, etc.)
- ✅ Local models for privacy (Ollama)
- ✅ Cost comparison across providers

- ✔️ Existing contracts (Azure, Anthropic)
- ✔️ Multi-provider fallback strategy

---

6. Best Practices

✔️ DO: Always Specify Model Explicitly (Recommended: gemini-2.5-flash)

```
# ✔️ Good - Always specify model explicitly for clarity
agent = Agent(
    model='gemini-2.5-flash',  # RECOMMENDED: Best price-performance
    name='my_agent'
)

# ❌ Bad - Relying on default (empty string, inherits from parent)
agent = Agent(name='my_agent')  # Model defaults to '', inherits from parent

# ✔️ Good - Be explicit and intentional about model choice
# Test and optimize based on your needs:
# - Use 2.5-flash for general purpose (RECOMMENDED)
# - Downgrade to 2.5-flash-lite if ultra-simple tasks
# - Upgrade to 2.5-pro if complex reasoning needed
```

✔️ DO: Benchmark Before Production

```
# ✔️ Good - Test models before deploying
models = ['gemini-2.0-flash', 'gemini-1.5-flash', 'gemini-1.5-flash-8b']

for model in models:
    agent = Agent(model=model, name='test')
    # Run representative queries
    # Measure latency, quality, cost
    # Choose best fit
```

✔️ DO: Consider Feature Requirements

```python
# ✔️ Good - Check feature compatibility
if need_code_execution:
    model = 'gemini-2.0-flash'  # Supports code execution
elif need_thinking:
    model = 'gemini-2.0-flash-thinking'  # Extended reasoning
else:
    model = 'gemini-1.5-flash'  # Fast and economical
```

❌ DON'T: Use Pro for Simple Tasks

```python
# ❌ Bad - Overpaying for simple queries
agent = Agent(
    model='gemini-1.5-pro',  # Expensive
    instruction="Answer yes or no questions"  # Simple task
)

# ✔️ Good - Match complexity to model
agent = Agent(
    model='gemini-1.5-flash-8b',  # Economical
    instruction="Answer yes or no questions"
)
```

Summary

You've mastered model selection and optimization:

Key Takeaways:

- ⭐ `gemini-2.5-flash` **is RECOMMENDED** - Best price-performance, first Flash with thinking
- ✔️ **Always specify model explicitly** - Default is empty string (inherits from parent)
- ✔️ `gemini-2.5-pro` for complex reasoning in code, math, STEM
- ✔️ `gemini-2.5-flash-lite` for ultra-fast, cost-efficient high-throughput
- ✔️ `gemini-2.0-flash` and `gemini-1.5-*` models still available (legacy)

- ✔️ **LiteLLM support** for OpenAI, Claude, Ollama, Azure when needed
- ✔️ Native Gemini recommended over Gemini-via-LiteLLM
- ✔️ Benchmark models before production deployment
- ✔️ Consider cost vs performance vs features vs provider requirements

Production Checklist:

- [ ] Model selection based on requirements (recommended: gemini-2.5-flash)
- [ ] Model explicitly specified in Agent constructor (don't rely on defaults)
- [ ] Benchmarking completed on representative queries
- [ ] Feature compatibility verified (2.5 Flash has thinking!)
- [ ] Cost projections calculated
- [ ] Performance SLAs defined
- [ ] Provider selection (Gemini vs LiteLLM providers)
- [ ] Fallback models configured
- [ ] Model monitoring in place
- [ ] Migration strategy planned (1.5/2.0 → 2.5)

What You Learned:

1. **Gemini 2.5 Flash is RECOMMENDED** - Best performance and value for new projects
2. **Always specify models explicitly** - Default is empty string (inherits from parent)
3. **Complete model lineup** - From 2.5-flash-lite (fastest) to 2.5-pro (smartest)
4. **LiteLLM integration** - Use OpenAI, Claude, Ollama when you need provider flexibility
5. **Native vs LiteLLM** - Always prefer native Gemini for best performance
6. **Selection framework** - Use MODELS dict and recommend_model() for systematic choices

Next Steps:

- **Tutorial 23**: Learn Production Deployment Strategies
- **Tutorial 24**: Master Advanced Observability
- **Tutorial 25**: Explore Best Practices & Patterns
- **Tutorial 26**: Google AgentSpace for Enterprise Agent Management
- **Tutorial 27**: Third-Party Framework Tools (LangChain, CrewAI)

- **Tutorial 28**: Deep Dive into Using Other LLMs with LiteLLM

Resources:

- Gemini 2.5 Documentation (https://ai.google.dev/gemini-api/docs/models) - Official Google AI docs
- Vertex AI Gemini 2.5 Flash (https://cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-5-flash) - Cloud docs
- Gemini 2.5 Technical Report (https://storage.googleapis.com/deepmind-media/gemini/gemini_v2_5_report.pdf) - Research paper
- LiteLLM Documentation (https://docs.litellm.ai/) - Multi-provider integration
- Pricing Calculator (https://cloud.google.com/products/calculator) - Cost estimation

---

🎉 **Tutorial 22 Complete!** You now know how to select and optimize models for your use cases. Continue to Tutorial 23 to learn about production deployment strategies.

Generated on 2025-10-19 17:57:01 from 22_model_selection.md

Source: Google ADK Training Hub