AI Agent Benchmark: Performance vs. Cost Analysis

1. Overview

This experiment benchmarks the performance and cost efficiency of five AI agents across document retrieval and response generation tasks. The agents evaluated are:

- **ReAct Agent**
- **OpenAI Agent**
- **LLM Compiler Agent**
- **LLM Chain-of-Abstraction Agent**
- **Language Agent Tree Search (LATS) Agent**

Each was tested against both simple and complex queries. Evaluation criteria included:

- Execution Time
- Memory Usage
- Token Consumption
- Estimated Cost (based on OpenAI API pricing)

2. Technology Stack

- **Programming Language:**

  Python

- **Libraries & Frameworks:**
  - OpenAI API for language model execution

    - LlamaIndex for document retrieval and indexing
    - psutil for memory tracking
    - pdfplumber for PDF text extraction
    - Weights & Biases (wandb) for experiment tracking
    - ThreadPoolExecutor for parallel document loading
    - tiktoken for token counting
    - torch for GPU-based embeddings and acceleration

3. Code Implementation

1. Environment Setup

- OpenAI keys loaded from environment variables
- Weights & Biases initialized for tracking
- Token tracking via tiktoken

2. Performance Tracking

A PerformanceTracker class was created to measure:

- Execution time
- Memory usage
- Tokens used
- Estimated cost (based on OpenAI API pricing)

3. Document Processing

- Text documents and PDFs are loaded from a directory.
- Text loading optimized using parallel threads
- Extracted text is stored in a vector database (LlamaIndex).

4. Agents and Query Execution

Each agent was designed to process a query and retrieve relevant documents before generating a response:

1. **ReAct Agent:**
   - Utilized OpenAI's ReActAgent for stepwise reasoning.
   - Retrieved relevant documents and generated responses.
2. **OpenAI Agent:**
   - Used OpenAI's ChatOpenAI model to generate responses directly from retrieved documents.
3. **LLM Compiler Agent:**
   - Used a PromptTemplate and LLMChain to structure responses.
4. **LLM Chain-of-Abstraction Agent:**
   - Used a predefined SystemMessage to enforce structured abstraction in responses.
5. **LATS Agent:**
   - Implemented Language Agent Tree Search for query decomposition and synthesis
   - Broke down complex queries into sub-questions before retrieval
   - Combined parallel retrievals with hierarchical response generation

5. Execution Process

- A query ("List all the documents available.") was used for testing all agents.
- Each agent was executed sequentially.
- Performance metrics were logged to Weights & Biases.
- GPU acceleration was used where available for embedding generation.

6. Code Execution & Finalization

- After executing all agents, wandb.finish() was called to finalize tracking.

4. Document

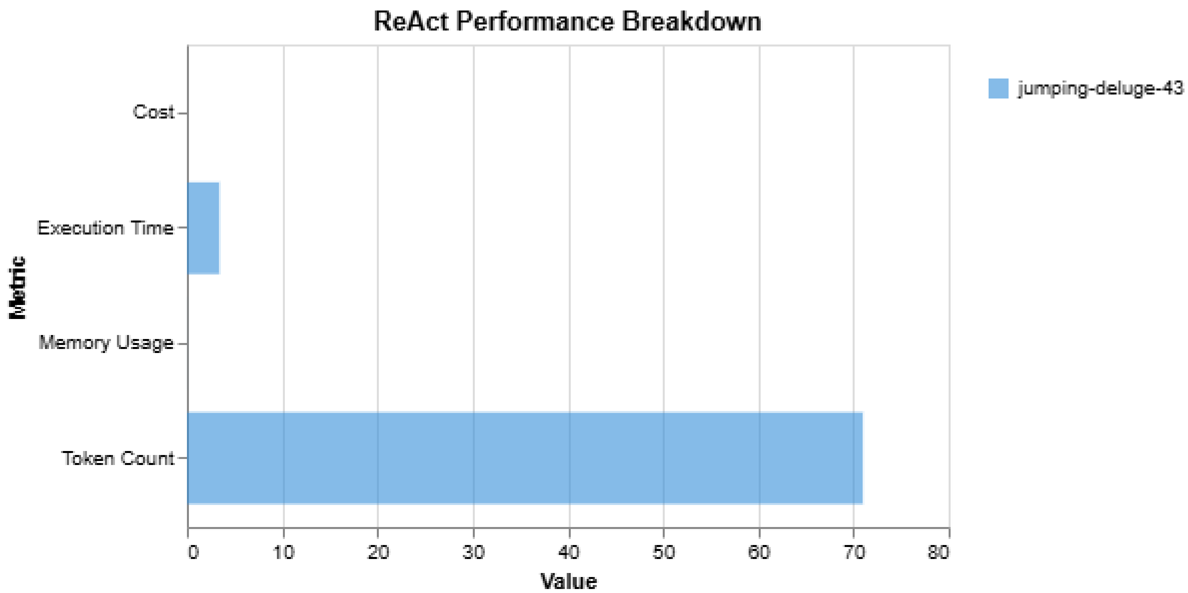
Tested on: Risk-Assessment-Report-Booklet.pdf from the BRS website
Code Repository: github.com/think-ke/AI-Experiments

5. Results Analysis for a simple query

Simple Query: ***"What does the document talk about?"***

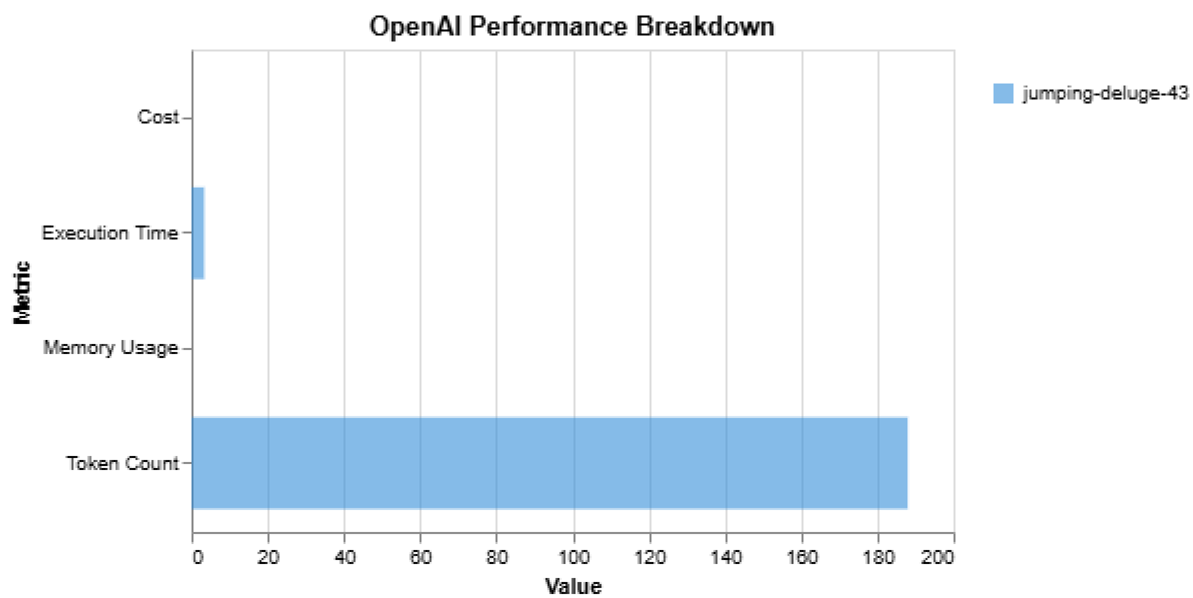
Agent	Time (s)	Memory (MB)	Tokens	Cost ($)
ReAct	3.42	0.01	71	0.00014
OpenAI	3.35	0.00	188	0.00038
Compiler	5.67	0.00	390	0.00078
Abstraction	7.96	0.00	357	0.00071
LATS	12.23	0.00	615	0.00123

ReAct Agent



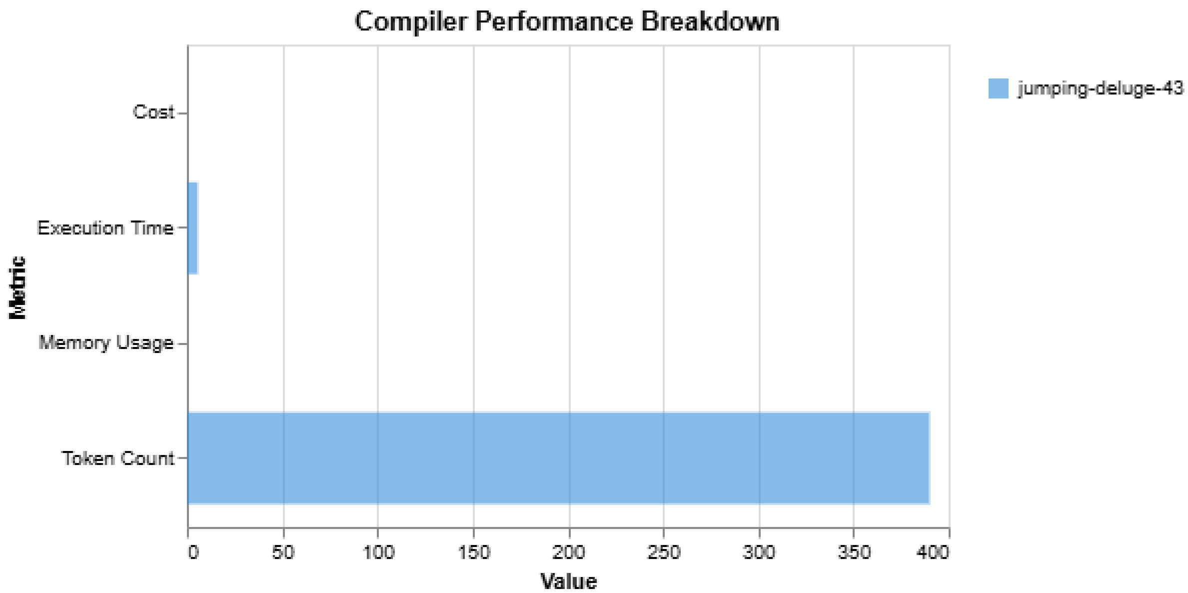
Metric	Value
Execution Time	3.420231
Memory Usage	0.011719
Token Count	71
Cost	0.000142

OpenAI Agent



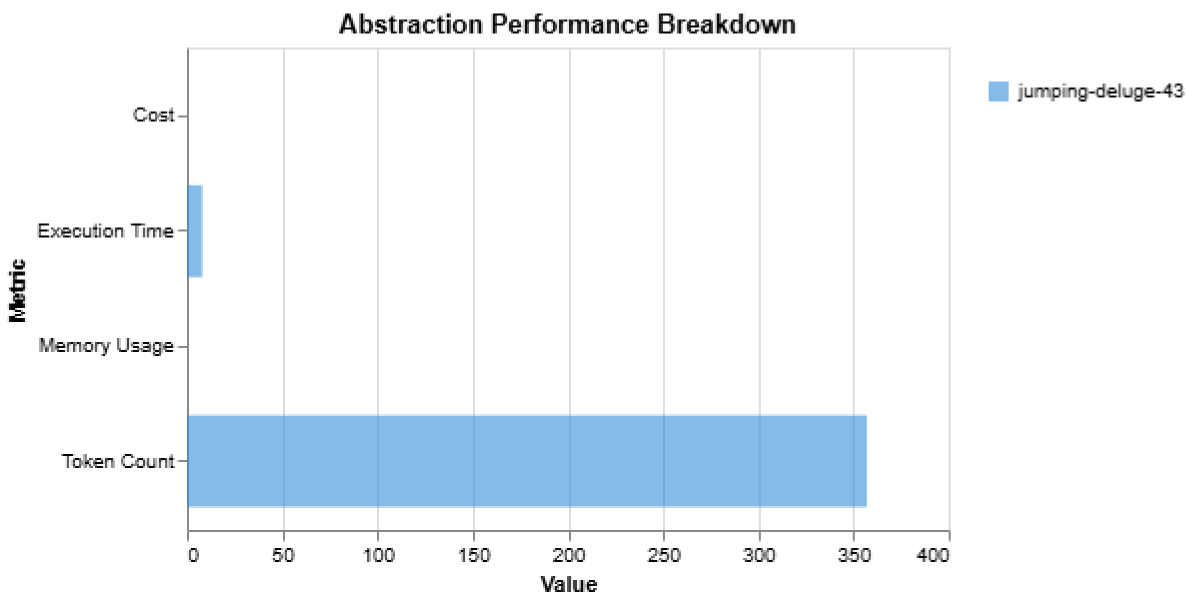
Metric	Value
Execution Time	3.353041
Memory Usage	0
Token Count	188
Cost	0.000376

LLM Compiler



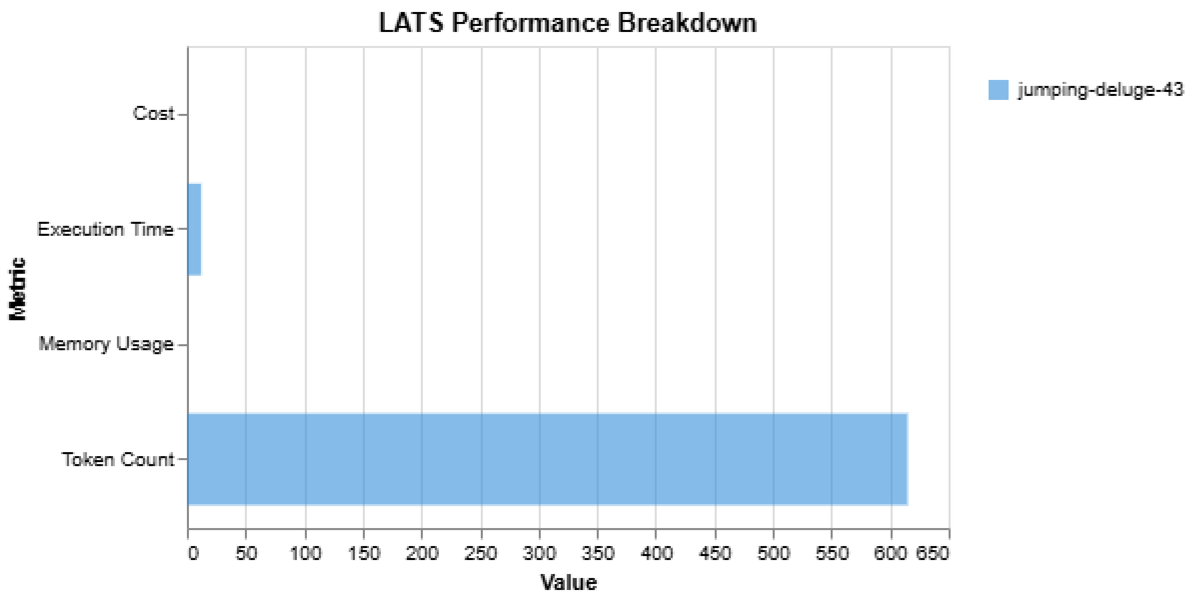
Metric	Value
Execution Time	5.667527
Memory Usage	0
Token Count	390
Cost	0.00078

LLM Chain-of-Abstraction agent



Metric	Value
Execution Time	7.961807
Memory Usage	0
Token Count	357
Cost	0.000714

Language Agent Tree Search Agent:



Metric	Value
Execution Time	12.22575
Memory Usage	0
Token Count	615

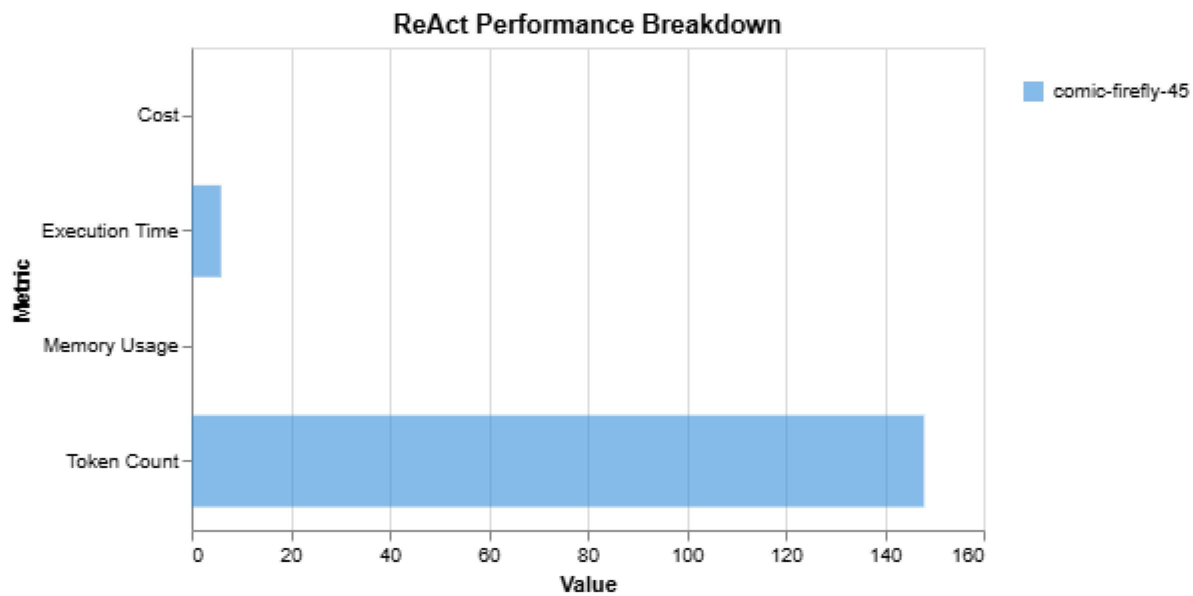
Cost	0.00123

6. Results Analysis for a complex query

Complex Query: *“Summarize the top 5 risks and mitigation strategies.”*

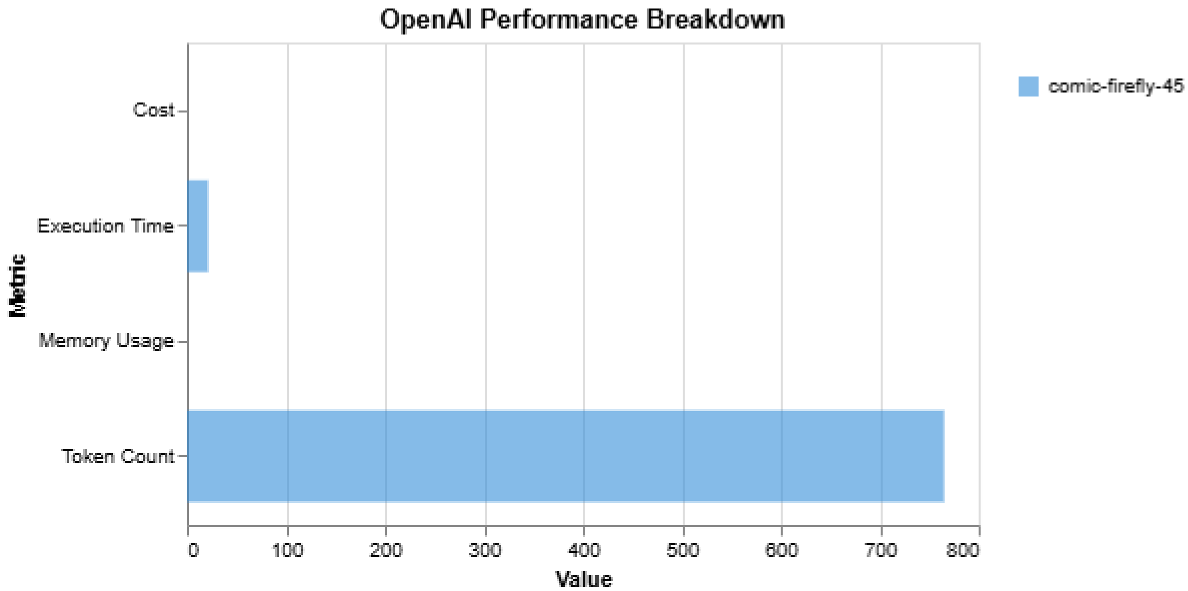
Agent	Time (s)	Memory (MB)	Tokens	Cost ($)
ReAct	5.97	0.02	148	0.00030
OpenAI	21.28	0.00	765	0.00153
Compiler	12.52	0.00	616	0.00123
Abstraction	12.99	0.00	524	0.00105
LATS	18.83	0.00	1113	0.00223

ReAct Agent



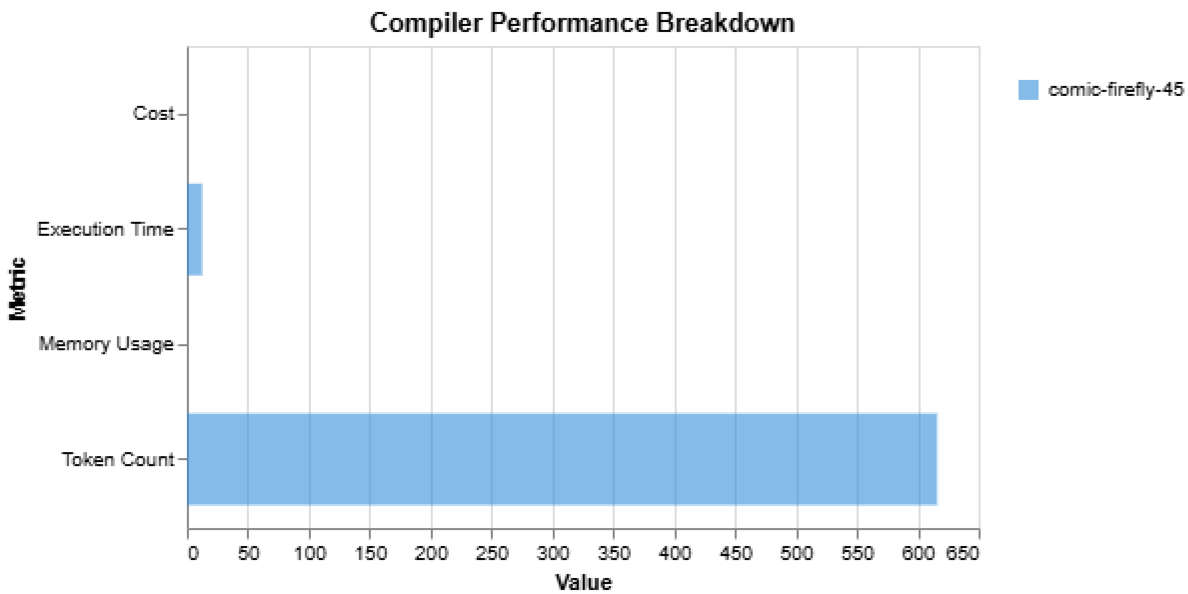
Metric	Value
Execution Time	5.967277
Memory Usage	0.023438
Token Count	148
Cost	0.000296

OpenAI Agent



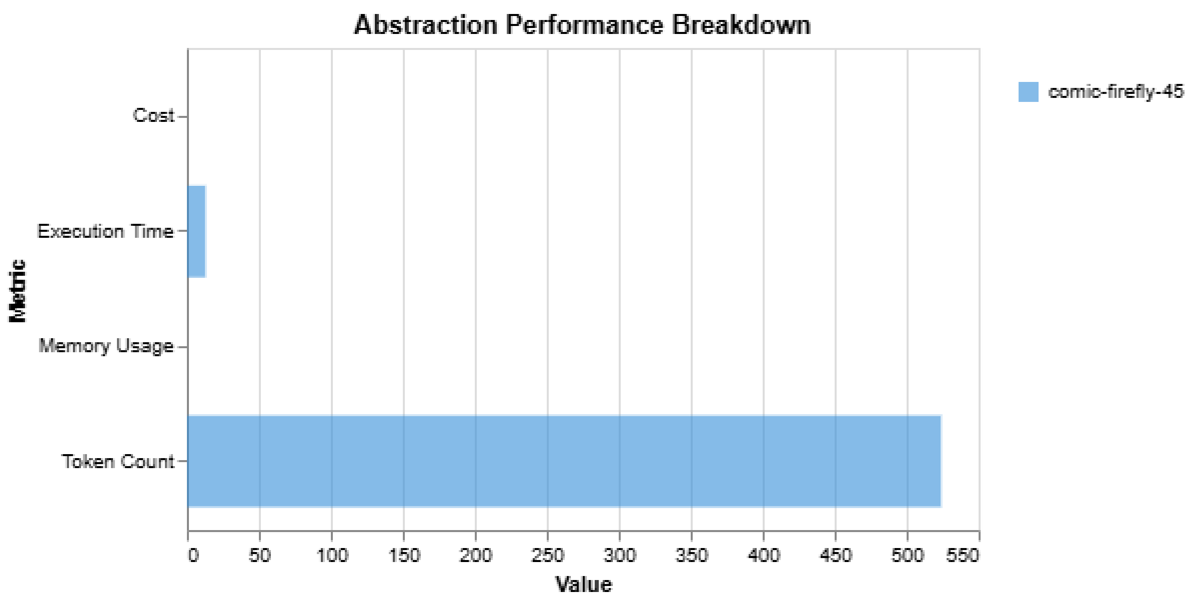
Metric	Value
Execution Time	21.27965
Memory Usage	0
Token Count	765
Cost	0.00153

LLM Compiler Agent



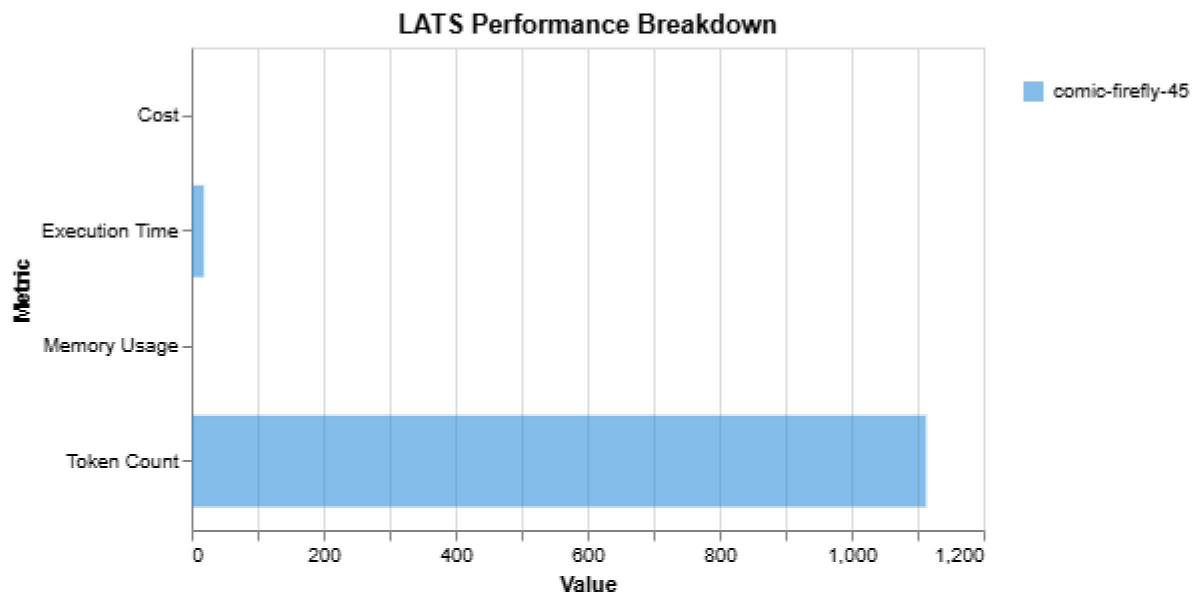
Metric	Value
Execution Time	12.51766
Memory Usage	0
Token Count	616
Cost	0.001232

LLM Chain-of-abstraction Agent



Metric	Value
Execution Time	12.99656
Memory Usage	0
Token Count	524
Cost	0.001048

Language Agent Tree Search:

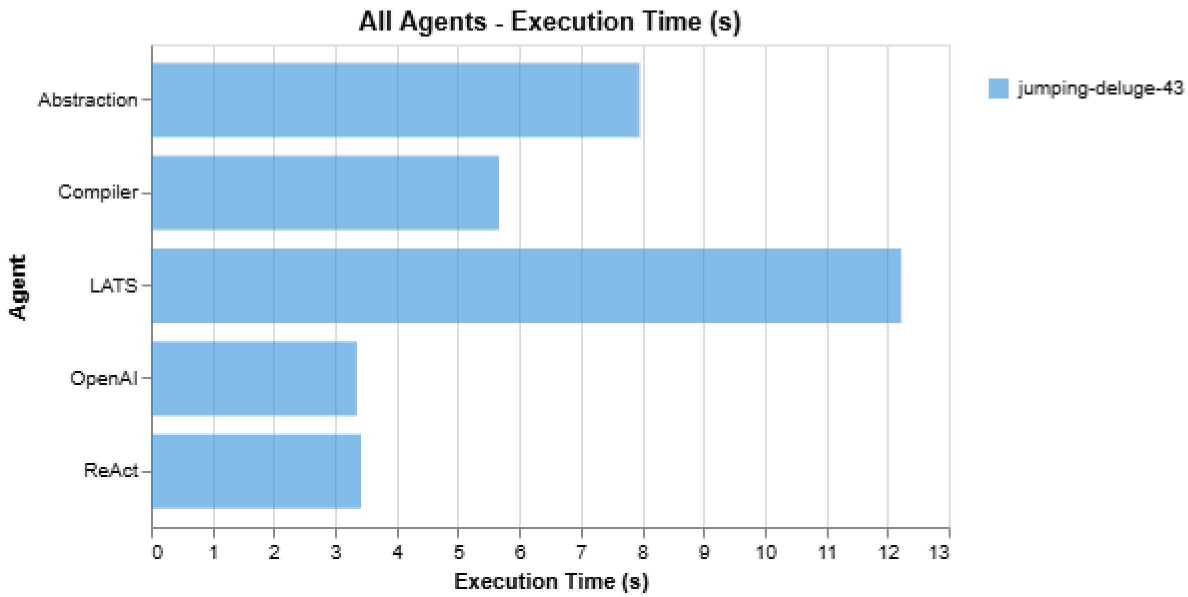


Metric	Value
Execution Time	18.83493
Memory Usage	0
Token Count	1113
Cost	0.002226

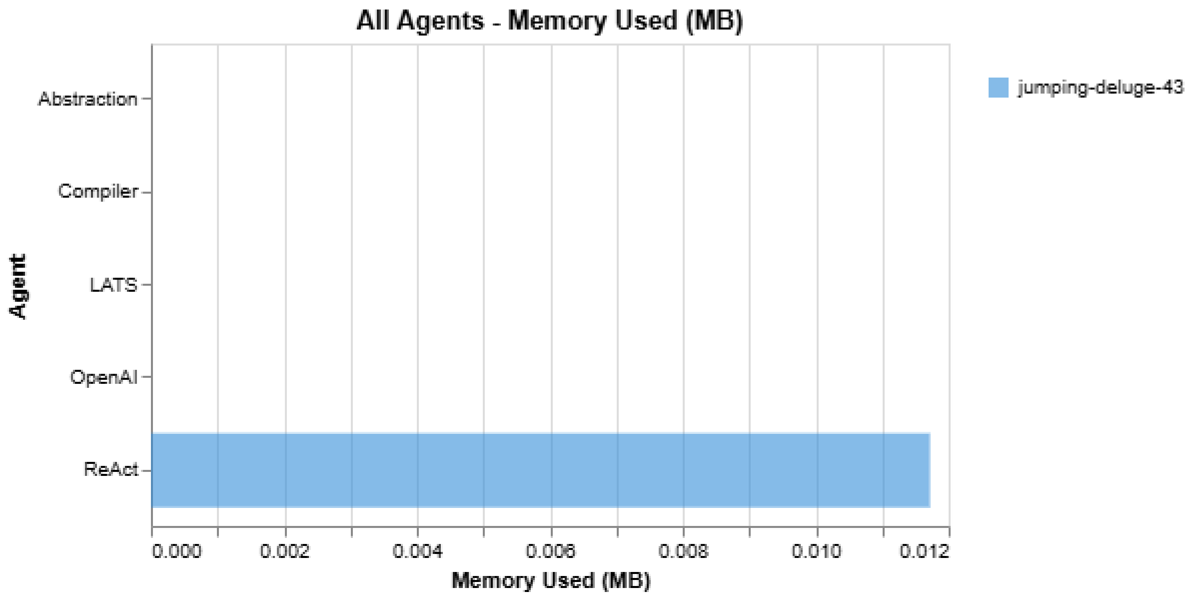
7. Aggregated Results Analysis based on Performance Metrics

7.1 Simple Query

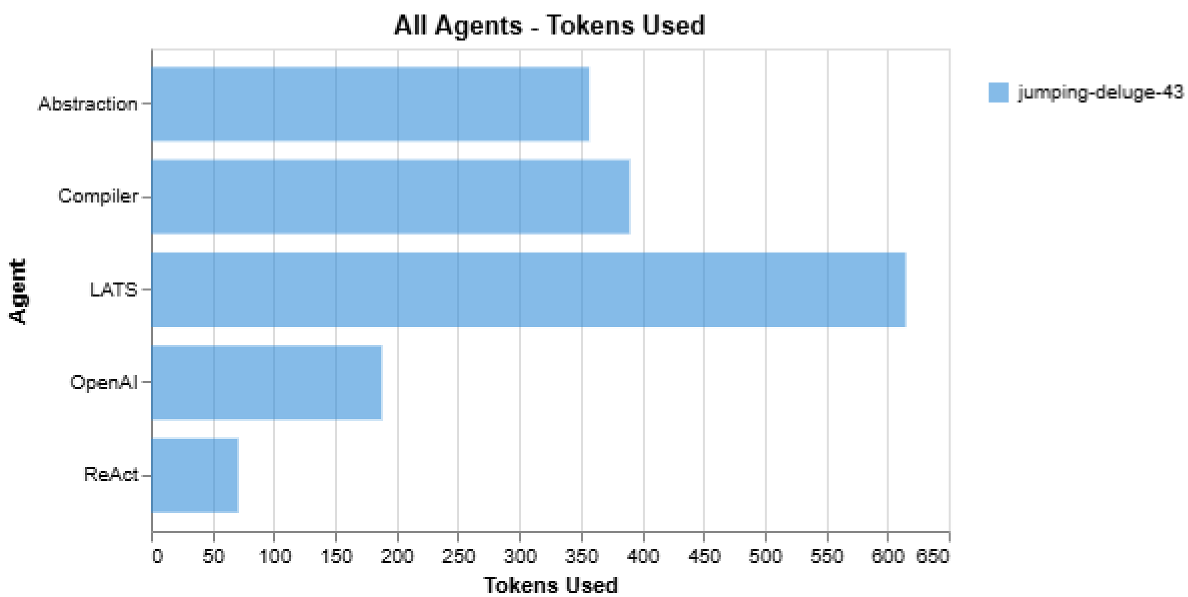
Execution Time



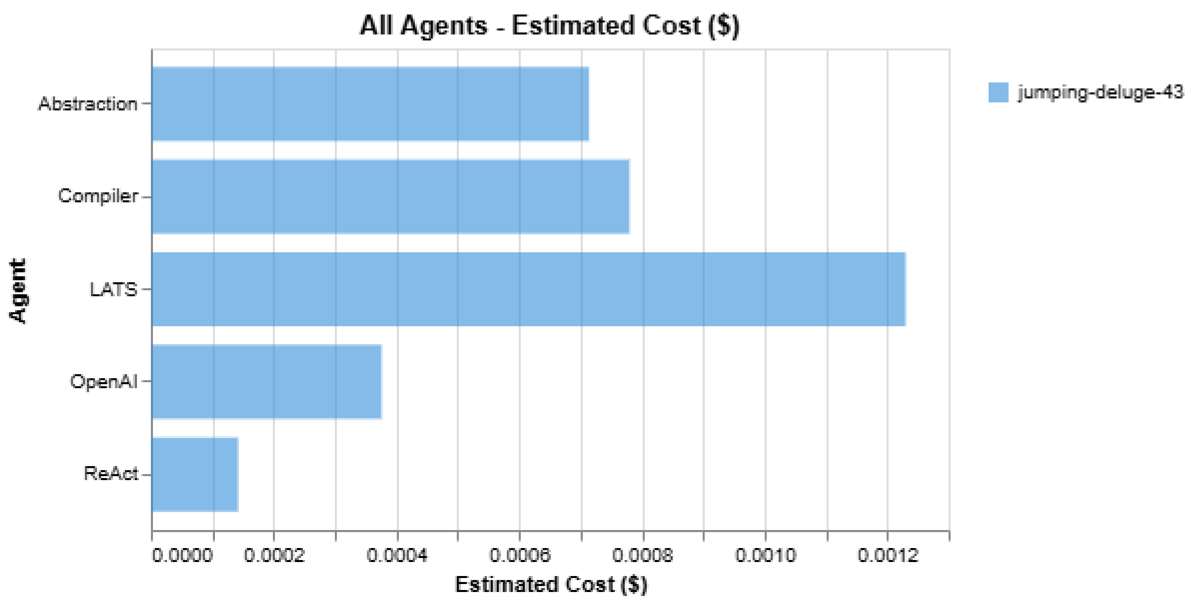
Memory used



Tokens Used

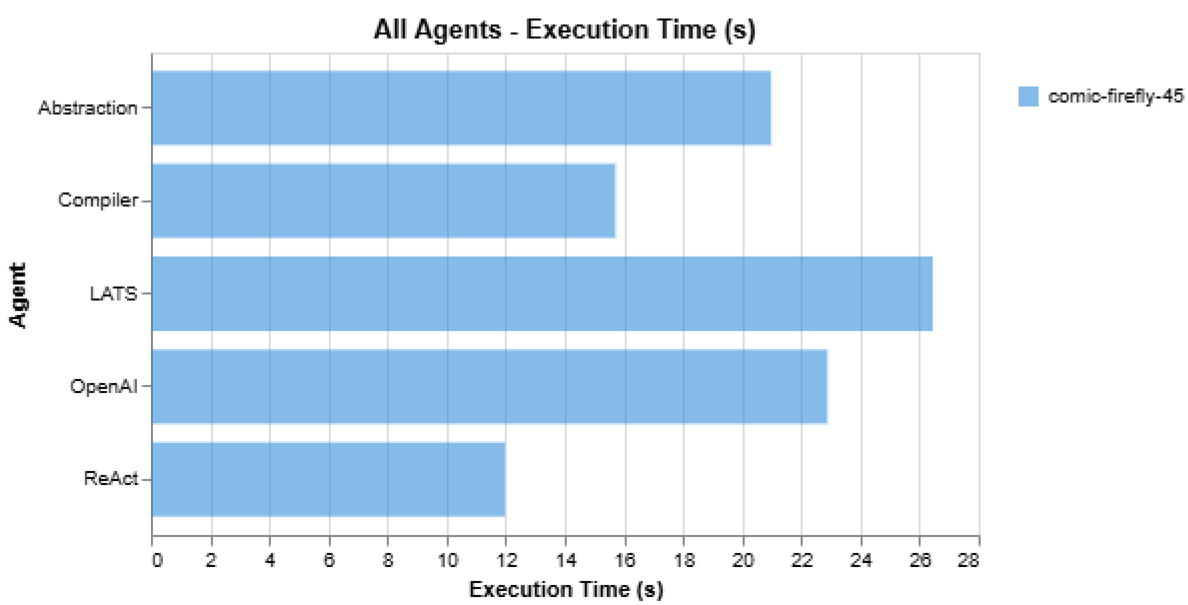


Estimated Cost

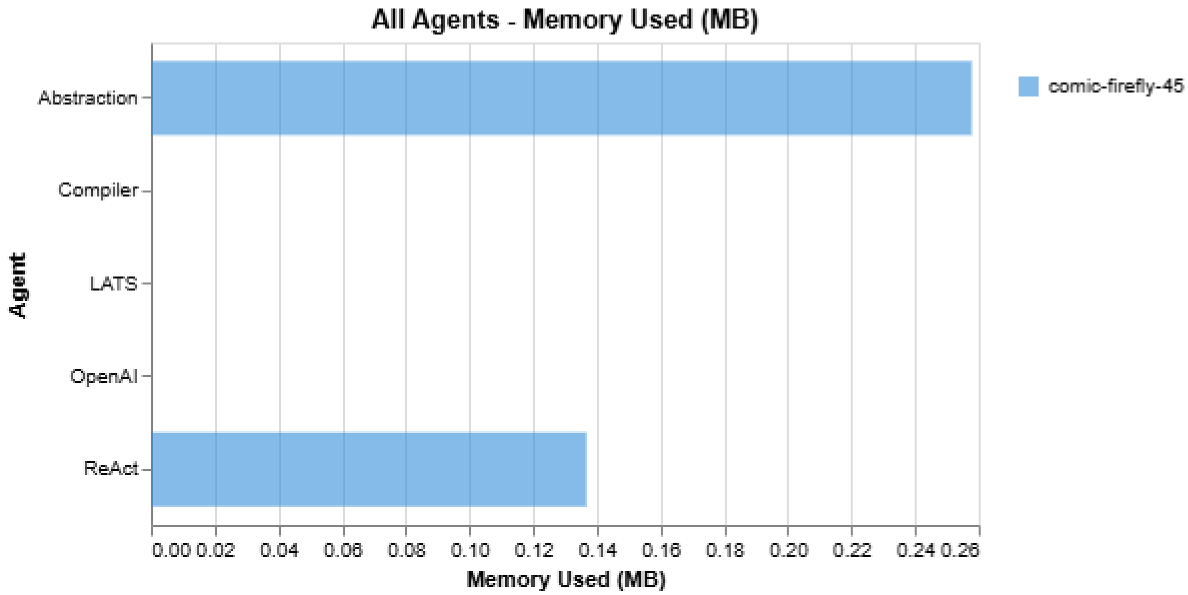


7.2 Complex Query

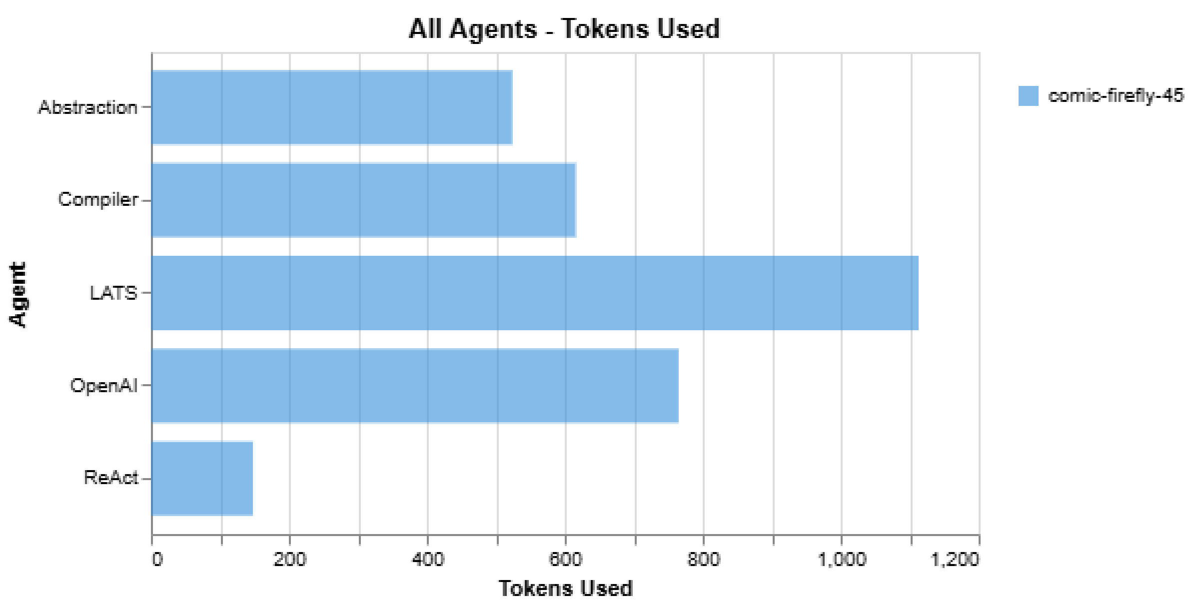
Execution Time



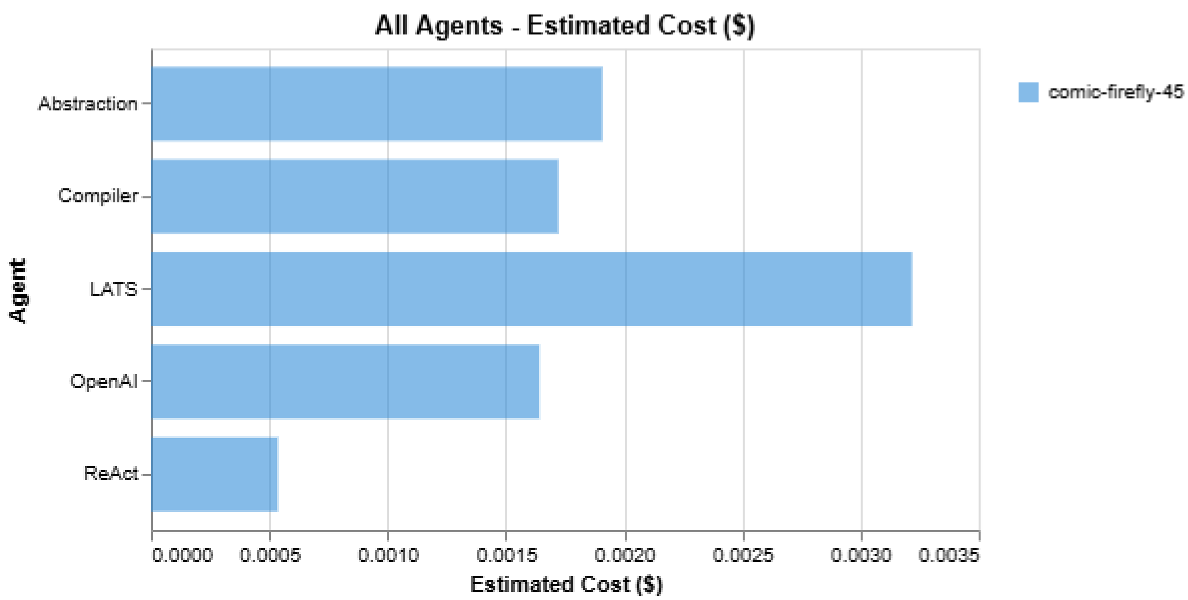
Memory Used



Tokens Used



Estimated Cost



8. Key Conclusions from the Agent Performance Experiment

8.1 Simple Queries

- **ReAct Agent** leads with the best speed-to-cost ratio.
- **OpenAI Agent** is fast but inefficient in token use (2.6x more expensive than ReAct).
- **Compiler and Abstraction** agents offer structured responses but at higher cost.
- **LATS Agent** is overkill—slowest and most expensive.

Recommendation:
Use **ReAct** for simple queries
Avoid **LATS** unless necessary for depth

8.2 Complex Queries

- **ReAct Agent** again balances performance and cost effectively.
- **OpenAI Agent** struggles: slowest and costliest relative to output quality.
- **Compiler Agent** gives structured breakdowns; suitable for technical queries.
- **Abstraction Agent** adds conceptual depth.
- **LATS** offers exhaustive breakdowns but at high cost and time.

Recommendations:

- For cost-efficiency: **ReAct**
- For structure: **Compiler**
- For layered insight and strategic-panning: **Abstraction**
- For deep synthesis: **LATS** (only if depth > budget)
- Avoid **OpenAI Agent** for complex tasks

9. Final Takeaways

Use Case	**Recommended Agent**
Fast, cheap answers	ReAct Agent
Technical summaries	Compiler Agent
Multi-layer analysis	Abstraction Agent
Deep decomposition	LATS Agent (high cost)
Simple raw outputs	OpenAI Agent (only for speed)

10. Strategic Summary:

Agent selection should be query-driven:

- prioritize ReAct for versatility
- Choose Compiler, Abstraction, or LATS for structured, deep, or complex reasoning.

11. Glossary

- **Execution Time (s)**: Time taken to process and respond to a query
- **Memory Used (MB)**: RAM usage during execution
- **Tokens Used**: Measure of text length that affects cost
- **Estimated Cost ($)**: Based on OpenAI token pricing
- **Output Structure**: Degree of logical formatting in the agent's response
- **LATS :** Language Agent Search Tree