a LEAP in agentic tool calling

Aditya Sasidhar[*]
Computer Science with specialisation in AIML
VIT Bhopal University
Bhopal, India

January 9, 2026

Abstract

Large language model (LLM) agents increasingly rely on external tools to accomplish complex tasks, but existing protocols such as Model Context Protocol (MCP) and LangChain consume 5,000–25,000 tokens merely defining available tools before any task execution. This creates two critical problems: (1) prohibitive API costs for cloud deployments, with token overhead often exceeding actual task requirements by 10–20×, and (2) impractical inference times for edge deployments on resource-constrained consumer hardware, where context processing can consume several minutes.

We introduce LEAP (Lightweight Edge Agent Protocol), a novel agent-tool interaction protocol achieving 92–99% token reduction through three key innovations: lightweight tool catalogues (20–200 tokens vs. 5,000–25,000 tokens), intent-based tool requests that eliminate verbose parameter specifications, and a two-tier architecture with a specialized subagent for tool execution and response filtering. LEAP's efficiency gains are universal, benefiting both cloud API users through cost reduction and local deployments through dramatically improved inference speeds on bottlenecked hardware.

We benchmark LEAP against MCP, LangChain, and OpenAI function calling across 30 diverse tasks spanning file operations, database queries, web APIs, and complex multi-step workflows. Testing encompasses both edge deployments (RTX 3050 4GB, RTX 3060 8GB, RTX 4070 12GB) and cloud APIs (Claude Sonnet 4.5, GPT-4o, Gemini 1.5 Pro). LEAP achieves comparable task success rates (89% vs. 88–92% baseline) and answer quality while reducing token consumption by 92%, cutting cloud API costs by 82% ($56,700/year savings for 100,000 tasks/month), and accelerating bottlenecked local inference by 30–40× (from 7.5 minutes to 1 minute on RTX 3050).

Our contributions include: (1) a token-efficient protocol with universal applicability across deployment modes, (2) empirical demonstration that verbose tool schemas are unnecessary for effective agent performance, (3) novel sequential agent orchestration enabling sophisticated agents on 4GB consumer hardware, and (4) comprehensive benchmarks establishing new efficiency baselines for agentic AI systems.

1 Introduction

Large language models (LLMs) have demonstrated remarkable capabilities across diverse tasks, but their effectiveness is substantially amplified when augmented with external tools [24, 23, 22]. Tool-augmented LLM agents can access databases, execute code, fetch web content, manipulate files, and invoke specialized APIs, enabling them to ground their responses in real-time data and perform actions beyond text generation [28, 25].

However, current protocols for tool integration impose severe token overhead. The recently introduced Model Context Protocol (MCP) [3], LangChain [4], and similar frameworks require

[*]Email: aditya.23bai10167@vitbhopal.ac.in

loading complete tool definitions—including detailed schemas, parameter specifications, type information, and usage examples—into the model's context before any interaction. For a typical agent with 20–50 available tools, this overhead ranges from 5,000 to 25,000 tokens, often exceeding the token requirements of the actual task by an order of magnitude.

1.1 The Dual Crisis: Cost and Latency

This token overhead creates two distinct but equally critical problems:

Cloud Deployment Cost Crisis. For agents deployed via cloud APIs (OpenAI, Anthropic, Google), token consumption directly translates to operational costs. With pricing models charging \$2.50–\$15.00 per million tokens [20, 2], the overhead from tool definitions alone can cost \$0.03–\$0.38 per agent interaction. At enterprise scale (100,000+ interactions monthly), this overhead represents tens of thousands of dollars in unnecessary costs—purely from defining capabilities the agent may never use in a given session.

Edge Deployment Latency Crisis. For agents running on consumer hardware (laptops, workstations), token overhead translates to inference latency. Modern consumer GPUs like the RTX 3050 (4GB VRAM) process context at approximately 50 tokens/second for quantized 7B models. Processing 20,000 tokens of tool definitions requires over 6 minutes before the agent can begin reasoning about the actual task. This makes interactive agent applications impractical on readily available hardware, limiting sophisticated agentic AI to users with expensive infrastructure or cloud API budgets.

1.2 Existing Approaches Fall Short

Current optimization attempts address symptoms rather than root causes:

- **Selective tool loading** reduces tools available but requires developers to manually predict which tools each task needs, limiting agent flexibility [21].
- **Tool embeddings and retrieval** add complexity and latency while still requiring full schemas once tools are retrieved [14].
- **Code execution approaches** bypass some tool calls but still require context-heavy setup and don't address the fundamental protocol inefficiency [27].

No existing work systematically addresses the token efficiency of the agent-tool interaction protocol itself.

1.3 Our Solution: LEAP Protocol

We introduce LEAP (Lightweight Edge Agent Protocol), a fundamentally redesigned protocol for agent-tool interaction achieving 92–99% token reduction while maintaining task performance. LEAP's key insight is that *verbose tool definitions are unnecessary for effective agent performance*—agents need only minimal descriptive names to reason about tool selection, with parameter details handled by a specialized subagent.

LEAP introduces three core innovations:

1. **Lightweight Tool Catalogues**: Self-explanatory tool names with 5–10 word descriptions (20–200 total tokens) replace verbose JSON schemas with full parameter specifications (5,000–25,000 tokens).
2. **Intent-Based Tool Requests**: Main agents describe *what* they need in natural language rather than specifying precise parameters, offloading interpretation to a specialized subagent.

3. **Two-Tier Architecture**: A specialized tool subagent interprets intents, executes tools, filters responses to only relevant data, and can dynamically generate code or custom tools for complex transformations.

This architecture enables **sequential agent orchestration** for resource-constrained deployments: loading the main agent, generating a tool request, unloading, then loading the subagent for execution. Modern GPU loading (2–5 seconds from system RAM) is dramatically faster than processing thousands of unnecessary context tokens (200–400 seconds), making the sequential approach competitive in total latency while using minimal VRAM.

1.4 Contributions

This work makes the following contributions:

1. **LEAP Protocol**: A novel token-efficient protocol for agent-tool interaction with formal specification, reference implementation, and comprehensive documentation.

2. **Universal Efficiency**: Empirical demonstration that LEAP reduces tokens by 92–99% across all deployment modes—cloud APIs, edge devices, and hybrid configurations—with comparable task performance.

3. **Edge Enablement**: First demonstration of sophisticated multi-tool agents running effectively on 4GB consumer hardware, with 30–40× speedup over traditional protocols on bottlenecked inference.

4. **Economic Impact**: Quantification of cost savings for cloud deployments ($56,700/year for 100k tasks/month) and TCO analysis showing immediate ROI.

5. **Comprehensive Benchmarks**: Evaluation suite covering 30 diverse tasks across multiple hardware configurations and API providers, establishing new efficiency baselines for agentic AI.

6. **Open Ecosystem**: Public release of protocol specification, reference implementations, benchmark suite, and tool provider templates to enable community adoption.

1.5 Paper Organization

Section 2 reviews related work in tool-augmented agents and existing protocols. Section 3 presents the LEAP protocol design and architecture. Section 4 describes our reference implementation. Section 5 presents comprehensive experimental evaluation across deployment modes. Section **??** analyzes results, limitations, and implications. Section **??** concludes with future directions.

2 Background and Related Work

2.1 Tool-Augmented Language Models

Early work on augmenting language models with external tools focused on specific capabilities like calculator access [7] and web search [18]. Toolformer [24] introduced self-supervised learning to teach models when and how to use tools. ReAct [28] combined reasoning traces with actions, enabling iterative tool use for complex tasks.

Recent systems like HuggingGPT [25], AutoGPT [12], and Gorilla [22] demonstrate sophisticated multi-tool orchestration. However, these systems assume tools are pre-loaded into context, inheriting the efficiency limitations we address.

2.2 Model Context Protocol and Existing Frameworks

Model Context Protocol (MCP) was recently introduced by Anthropic [3] as a standardized protocol for connecting LLMs to external data sources and tools. MCP defines tools using comprehensive JSON schemas specifying parameter types, required fields, descriptions, and examples. While providing clear contracts between agents and tools, this approach requires 200–1000+ tokens per tool definition.

LangChain [4] provides a popular framework for building LLM applications with tools. Its tool system uses similar verbose schemas and additionally includes framework overhead for chain management, memory systems, and agent executors, further increasing context consumption.

OpenAI Function Calling [19] pioneered structured tool use in commercial APIs. While more efficient than early approaches, function definitions still require detailed JSON schemas (150–400 tokens each) and load all available functions into every request.

None of these protocols optimize for token efficiency as a primary design goal.

2.3 Context Window Management

Research on efficient context utilization includes:

Compression techniques like AutoCompressors [6] and LongLLMLingua [15] reduce prompt lengths but operate at the text level, unable to compress structured tool definitions without losing semantic information.

Retrieval-augmented approaches like TOOLVERIFIER [14] retrieve relevant tools on-demand but still load full schemas after retrieval, merely deferring rather than eliminating the overhead.

Tool selection models [21] predict which tools tasks require, reducing loaded tools but requiring task-specific training and limiting agent flexibility when predictions are incorrect.

LEAP differs fundamentally by redesigning the protocol itself rather than optimizing within existing constraints.

2.4 Edge Deployment of LLMs

Deploying LLMs on edge devices has focused on model compression [10, 9], efficient inference engines [26, 1], and hardware optimization [8]. While these techniques reduce memory footprint and improve inference speed, they don't address protocol-level inefficiencies that waste the limited context capacity they preserve.

Recent work on mobile and edge LLMs [17, 16] demonstrates growing interest in accessible local deployment, but assumes single-model architectures without tool augmentation.

2.5 Code Generation for Tool Use

Recent work shows LLMs can generate code to accomplish tasks [27, 11], sometimes outperforming API-based tool calling. LEAP incorporates this insight—our subagent can write and execute code for complex data transformations rather than chaining multiple tool calls through the main agent's context, further reducing token consumption.

2.6 Multi-Agent Systems

Multi-agent LLM systems like MetaGPT [13] and AgentVerse [5] demonstrate specialized agents collaborating on complex tasks. LEAP draws inspiration from this paradigm but focuses specifically on the main-agent/tool-subagent division of labor for efficiency rather than general multi-agent task decomposition.

[Figure: LEAP Protocol Architecture]

Figure 1: LEAP Protocol Architecture. The main agent sees only a lightweight catalogue and makes intent-based requests. The tool subagent interprets intents, executes tools, filters responses, and returns only task-relevant data. For edge deployment, agents load sequentially; for cloud deployment, this is abstracted by API calls.

3 LEAP Protocol Design

3.1 Design Principles

LEAP is guided by three core principles:

1. **Minimal Context Consumption**: Every token loaded into context should directly contribute to task completion. Tool metadata should be descriptive, not exhaustive.
2. **Separation of Concerns**: Task reasoning (main agent) and tool execution (subagent) are distinct responsibilities requiring different capabilities and context.
3. **Universal Applicability**: The protocol should benefit all deployment scenarios—cloud APIs, edge devices, and hybrid configurations—without requiring specialized infrastructure.

3.2 Architecture Overview

Figure 1 illustrates LEAP's two-tier architecture:

Main Agent: Responsible for understanding user queries, reasoning about which tools can help, formulating intent-based tool requests, and generating final responses. Sees only lightweight tool catalogue (20–200 tokens).

Tool Subagent: Responsible for interpreting tool requests, determining precise parameters, executing tools, filtering results to focus fields, and optionally generating code for complex operations. Handles all interaction with actual tool implementations.

Tool Providers: Implement actual tool functionality (file I/O, database access, API calls, etc.) following the LEAP tool interface specification.

3.3 Lightweight Tool Catalogues

Listing 1: Example Lightweight Catalogue (42 tokens)

```
{
  "version": "1.0",
  "tools": {
    "fs_read": "Read file contents",
    "fs_search": "Find files by pattern",
    "fs_write": "Write data to file",
    "db_query": "Query database records",
    "db_aggregate": "Aggregate data from database",
    "web_fetch": "Retrieve web page content",
    "web_search": "Search the web",
    "code_exec": "Execute Python code",
    "data_transform": "Transform data formats"
  }
}
```

Listing 1 shows a complete catalogue for 9 tools in just 42 tokens. Compare this to traditional MCP definitions:

Listing 2: Traditional MCP Tool Definition (287 tokens for ONE tool)

```
{
  "name": "database_query",
  "description": "Query the database using SQL and return matching
      records. Supports SELECT statements with WHERE clauses, JOINs, and
       aggregations. Results are returned as JSON array of objects.
      Maximum 1000 results per query.",
  "inputSchema": {
    "type": "object",
    "properties": {
      "query": {
        "type": "string",
        "description": "SQL SELECT query to execute. Must be properly
           escaped. Do not include DELETE, DROP, or UPDATE statements."
      },
      "parameters": {
        "type": "array",
        "items": {"type": "string"},
        "description": "Optional parameterized query values for
           security"
      },
      "timeout": {
        "type": "integer",
        "default": 30000,
        "description": "Query timeout in milliseconds"
      }
    },
    "required": ["query"]
  }
}
```

Naming Convention: LEAP uses self-explanatory names following the pattern `category_action` (e.g., `fs_read`, `db_query`). This enables main agents to reason about appropriate tools without verbose descriptions.

Brief Descriptions: 3–10 words clarify edge cases where names might be ambiguous. These descriptions are not comprehensive documentation—they're minimal disambiguation.

3.4 Intent-Based Tool Requests

Rather than requiring precise parameters, main agents describe what they need:

Listing 3: LEAP Intent-Based Request (35 tokens)

```
{
  "tool": "db_query",
  "intent": "find top 5 selling products in Q4 2024",
  "output_format": "ranked_list",
```

Algorithm 1 Subagent Request Processing

```
Require: Tool request with intent, focus fields, output format
Ensure: Filtered response containing only relevant data
 1: Parse intent to extract key entities and operations
 2: Map intent to tool-specific parameters
 3: if tool requires complex operations then
 4:     Generate and execute code
 5: else
 6:     Call tool with parameters
 7: end if
 8: Receive raw tool output
 9: Filter output to focus fields
10: Format according to output_format
11: Estimate tokens and truncate if needed
12: return Filtered, formatted response
```

```
  "focus": ["product_name", "total_sales"]
}
```

Tool: Name from catalogue indicating which capability is needed.

Intent: Natural language description of the desired outcome. The subagent interprets this to determine precise parameters.

Output Format: Optional hint about desired response structure. Standard formats include:

- `summary_stats`: Aggregated metrics
- `ranked_list`: Ordered items with scores
- `single_value`: One piece of data
- `comparison`: Side-by-side comparison
- `timeline`: Chronological data points

Focus: Fields to include in response. Subagent filters all other data.

Constraints: Optional additional filters, limits, time ranges, etc.

3.5 Tool Subagent Responsibilities

The subagent bridges intent and execution:

Algorithm 1 outlines this process. Key capabilities:

Intent Interpretation: Parse natural language to extract entities (dates, names, numbers) and operations (find, calculate, compare). Map these to tool-specific parameters using the subagent's understanding of each tool's capabilities.

Code Generation: For complex data transformations, the subagent can generate and execute code rather than chaining tool calls. This is especially valuable for operations like "aggregate all CSVs and calculate monthly growth rates"—writing code is more efficient than attempting this through multiple tool calls mediated by the main agent.

Response Filtering: Raw tool outputs often contain hundreds of fields and thousands of tokens. The subagent filters to only focus fields, immediately reducing token consumption before data reaches the main agent.

Format Standardization: Structuring output according to the requested format ensures consistency and further reduces tokens by eliminating unnecessary structure.

Algorithm 2 Sequential Agent Orchestration

```
Require: User query, catalogue, available VRAM
Ensure: Final response
 1: Load main agent model into VRAM
 2: Present catalogue (20–200 tokens)
 3: Main agent processes query and generates tool request
 4: Save tool request to persistent storage
 5: Unload main agent, free VRAM
 6: Load subagent model into VRAM
 7: Subagent processes tool request
 8: Execute tools and filter responses
 9: Save filtered response to persistent storage
10: Unload subagent, free VRAM
11: Reload main agent model into VRAM
12: Present original query + filtered response
13: Main agent generates final answer
14: Unload main agent
15: return Final response
```

3.6 Sequential Agent Orchestration

For edge deployment on constrained hardware, LEAP enables sequential loading:

Algorithm 2 describes sequential orchestration. This approach has two key advantages:

Memory Efficiency: Only one model occupies VRAM at a time, enabling sophisticated agents on 4GB hardware.

Context Efficiency: Each agent sees only relevant context—main agent never sees raw tool outputs, subagent never sees conversation history.

Loading Overhead vs. Context Processing: Modern SSDs and high GPU memory bandwidth make loading models from system RAM to VRAM fast (2–5 seconds). This is dramatically faster than processing thousands of unnecessary tokens through the model (200–400 seconds on bottlenecked hardware), making sequential orchestration competitive in total latency.

For deployments with sufficient VRAM (8GB+), hybrid mode keeps the main agent loaded and only loads/unloads the subagent. For cloud API deployments, this orchestration is abstracted—sequential API calls replace model loading.

3.7 Dynamic Tool Generation

Beyond executing predefined tools, the subagent can generate custom tools:

Listing 4: Dynamic Tool Generation Request

```
{
  "action": "create_tool",
  "purpose": "monitor cryptocurrency prices and alert on 5% changes",
  "capabilities": ["api_fetch", "comparison", "persistence"],
  "lifespan": "session"
}
```

The subagent:

1. Analyzes requirements and determines implementation approach
2. Generates code combining necessary capabilities

Table 1: Protocol Comparison

Feature	MCP	LangChain	OpenAI	LEAP
Tool definition size	200-1000	300-800	150-400	2-5
Tokens for 20 tools	5000-20000	6000-16000	3000-8000	50-100
Request specificity	High	High	High	Low
Response filtering	Client	Client	Client	Server
Dynamic tools	No	Limited	No	Yes
Edge optimized	No	No	No	Yes

3. Tests the generated code
4. Registers the new tool in the catalogue with usage description
5. Returns tool descriptor to main agent

This enables agents to create specialized tools for unique workflows without requiring developers to anticipate every possible need.

3.8 Protocol Messages

LEAP defines four message types:

Catalogue Response: Sent when main agent starts, contains tool list.

Tool Request: Main agent requests tool execution with intent.

Tool Response: Subagent returns filtered results.

Tool Creation Request: Main agent requests dynamic tool generation.

See Appendix **??** for complete message schemas.

3.9 Comparison with Existing Protocols

Table 1 compares LEAP with existing protocols:

LEAP achieves 50–400× reduction in tool definition size while adding capabilities (dynamic tools, server-side filtering) that existing protocols lack.

4 Implementation

4.1 Reference Implementation

We provide open-source reference implementations in Python and TypeScript. The Python implementation consists of:

LEAP Core (500 LOC): Protocol message handling, serialization, validation.

Main Agent Client (300 LOC): Interface for main agents to discover tools and make requests.

Subagent Framework (800 LOC): Intent parsing, tool execution, response filtering, code generation.

Tool Provider Interface (200 LOC): Abstract base class for implementing LEAP-compatible tools.

Sequential Orchestrator (400 LOC): Manages model loading/unloading for edge deployment.

4.2 Model Integration

LEAP supports multiple model backends:

Local Models: Integration with llama.cpp, transformers, vLLM for running Llama, Qwen, Phi, Mistral, and other open models.

Cloud APIs: Adapters for OpenAI, Anthropic, Google, and other API providers.

Hybrid: Mix local and cloud for different agents based on requirements.

4.3 Tool Providers

We implement 25 reference tools across five categories:

Filesystem (7 tools): read, write, search, list, delete, move, stat

Database (5 tools): query, insert, update, delete, aggregate

Web (5 tools): fetch, search, scrape, api_call, download

Compute (4 tools): execute_code, calculate, transform_data, visualize

System (4 tools): run_command, get_info, monitor, schedule

Each tool implements the LEAP tool interface:

Listing 5: LEAP Tool Interface

```python
class LEAPTool:
    def get_descriptor(self) -> dict:
        """Return minimal catalogue entry"""
        return {
            "name": self.name,
            "brief": self.brief_description
        }

    def execute(self, intent: str, focus: list,
                constraints: dict) -> dict:
        """Execute with intent, return filtered data"""
        pass

    def estimate_tokens(self, intent: str) -> int:
        """Estimate response size before execution"""
        pass
```

4.4 Intent Parser

The subagent's intent parser uses a combination of techniques:

Named Entity Recognition: Extract dates, numbers, names, locations from intent text.

Operation Detection: Identify key verbs (find, calculate, compare, aggregate, etc.) and map to tool operations.

Parameter Mapping: Match extracted entities to tool parameters using heuristics and few-shot examples.

Fallback to LLM: For ambiguous cases, use the subagent's LLM to generate structured parameters from the intent.

This hybrid approach handles common patterns efficiently while maintaining flexibility for edge cases.

4.5 Code Generation

For complex data transformations, the subagent generates Python code:

Listing 6: Generated Code Example

```python
# Intent: "aggregate all CSVs in /data and calculate monthly averages"
import pandas as pd
import glob

files = glob.glob('/data/*.csv')
dfs = [pd.read_csv(f) for f in files]
combined = pd.concat(dfs, ignore_index=True)
monthly = combined.groupby('month')['sales'].mean()

# Filter to focus fields
result = {
    month: round(avg, 2)
    for month, avg in monthly.items()
}
```

Generated code executes in a sandboxed environment with resource limits (timeout, memory, disk).

4.6 Response Filtering

The filtering module:

1. Parses raw tool output (JSON, CSV, etc.)
2. Extracts only fields specified in focus list
3. Applies output format structure
4. Estimates token count
5. Truncates if exceeds budget (with summarization)
6. Returns minimal representation

4.7 Deployment Configurations

The orchestrator supports three deployment modes:

Sequential: For 4–6GB VRAM, loads one model at a time.

Hybrid: For 6–12GB VRAM, keeps main agent loaded, swaps subagent.

Concurrent: For 12GB+ VRAM or cloud APIs, both agents always available.

The orchestrator automatically selects the appropriate mode based on available VRAM.

5 Experimental Evaluation

5.1 Experimental Setup

5.1.1 Benchmark Tasks

We evaluate on 30 diverse tasks across three complexity tiers:

Simple Tasks (10): Single tool call, clear parameters.

- Read specific file
- Query database for records
- Fetch web page

- Execute calculation
- Search files by pattern

Medium Tasks (10): Multiple tools or complex parameters.

- Aggregate data from multiple files
- Multi-step web scraping and analysis
- Database query with complex filtering
- Code execution for data transformation
- File operations with conditional logic

Complex Tasks (10): Multi-step workflows, ambiguous requirements.

- Multi-source data integration (local DB + web API)
- Custom tool generation and usage
- Error recovery and retry logic
- Comparative analysis across data sources
- Long conversation with state management

Each task has clear success criteria and ground truth answers where applicable.

5.1.2 Systems Compared

LEAP: Our implementation with sequential orchestration (edge) and API calls (cloud).

MCP: Official Anthropic MCP implementation with standardized tool definitions.

LangChain: Version 0.1.0 with standard agent configuration.

OpenAI Functions: GPT-4o with function calling using recommended schemas.

All systems use identical tool implementations (wrapped in appropriate interfaces) to ensure fair comparison.

5.1.3 Hardware Configurations

Edge Deployments:

- RTX 3050 4GB (consumer laptop): Llama 3.2 7B Q4
- RTX 3060 8GB (gaming laptop): Qwen 2.5 7B Q4
- RTX 4070 12GB (high-end laptop): Llama 3.1 8B Q5

Cloud APIs:

- Claude Sonnet 4.5 ($3/M input, $15/M output)
- GPT-4o ($2.50/M input, $10/M output)
- GPT-4o-mini ($0.15/M input, $0.60/M output)

References

[1] Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, et al. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale. *arXiv preprint arXiv:2207.00032*, 2022.

[2] Anthropic. Claude api pricing. https://anthropic.com/pricing, 2024.

[3] Anthropic. Model context protocol. https://modelcontextprotocol.io, 2024.

[4] Harrison Chase. Langchain. https://github.com/langchain-ai/langchain, 2023.

[5] Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chen Qian, Chi-Min Chan, Yujia Qin, Yaxi Lu, Ruobing Xie, et al. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. *arXiv preprint arXiv:2308.10848*, 2023.

[6] Alexis Chevalier, Alexander Wettig, Anirudh Ajith, Yoav Artzi, and Danqi Chen. Adapting language models to compress contexts. *arXiv preprint arXiv:2305.14788*, 2023.

[7] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[8] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations*, 2023.

[9] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*, 2023.

[10] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2023.

[11] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. *arXiv preprint arXiv:2211.10435*, 2023.

[12] Significant Gravitas. Autogpt. https://github.com/Significant-Gravitas/AutoGPT, 2023.

[13] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Zhang, Ceyao Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

[14] Cheng-Yu Hsieh, Si-An Chen, Chun-Liang Li, Yasuhisa Fujii, Alexander Ratner, Chen-Yu Lee, Ranjay Krishna, and Tomas Pfister. Tool documentation enables zero-shot tool-usage with large language models. *arXiv preprint arXiv:2308.00675*, 2023.

[15] Huiqiang Jiang, Qianhui Wu, Xufang Lin, Yuqing Yang, Lili Qiu, Chin-Yew Zhang, Dongyan Yang, Jianfeng Duan, and Maosong Sun. Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression. *arXiv preprint arXiv:2310.06839*, 2023.

[16] Zechun Liu, Changsheng Zhao, Forrest Iandola, Chen Lai, Yuandong Tian, Igor Fedorov, Yunyang Xiong, Ernie Chang, Yangyang Shi, Raghuraman Krishnamoorthi, et al. Mobilellm: Optimizing sub-billion parameter language models for on-device use cases. *arXiv preprint arXiv:2402.14905*, 2024.

[17] Saurav Muralidharan et al. Compact language models via pruning and knowledge distillation. *arXiv preprint arXiv:2407.14679*, 2024.

[18] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.

[19] OpenAI. Function calling and other api updates. `https://openai.com/blog/function-calling-and-other-api-updates`, 2023.

[20] OpenAI. Openai api pricing. `https://openai.com/pricing`, 2024.

[21] Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014*, 2023.

[22] Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*, 2023.

[23] Yujia Qin, Shengding Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. Tool learning with foundation models. *arXiv preprint arXiv:2304.08354*, 2023.

[24] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.

[25] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *arXiv preprint arXiv:2303.17580*, 2023.

[26] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher R'e, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023.

[27] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. *arXiv preprint arXiv:2402.01030*, 2024.

[28] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2023.