# TELLER-Bench: Tool-Enhanced LLM Agent Evaluation for Real-World Banking

**Anonymous ACL submission**

## Abstract

Effective evaluation of multi-step tool invocation in complex workflows is critical for analyzing LLMs' planning, reasoning, and execution capabilities in real-world applications. However, progress has been limited by a lack of faithful benchmarks capturing the detailed logic of safety-critical domains such as banking. To fill this gap, we present TELLER, a benchmark containing 1,033 test instances across five banking scenarios, designed for thorough evaluation of LLMs in complex workflows.TELLER ensures realistic Standard Operating Procedure (SOP) constraints, complex API dependencies, and verifiable results through a two-stage framework including dependency graph reconstruction and end-to-end execution. We evaluate 14 LLMs across five model families (Claude, Gemini, GPT, DeepSeek, Qwen), revealing significant challenges. The leading model, Gemini-3-Pro, achieves only 38% execution accuracy, while open-source models below 32B parameters fall below 11%. Further studies reveal weaknesses in understanding tool dependencies and precise invocation, providing insights for future optimization. By establishing a high-quality benchmark for diverse banking workflows, TELLER lays the groundwork for advancing LLM agent deployment in real-world financial industries. [1]

## 1 Introduction

With the rapid progress of artificial intelligence, large language models (LLMs) are evolving from conversational assistants into agentic experts capable of using tools to solve real-world problems (Yao et al., 2022; Chen et al., 2024). This transition is particularly critical yet challenging in banking, where models must go beyond text generation to possess three combined capabilities: financial understanding, strict compliance following, and precise workflow execution (Lee et al., 2025). As illustrated in Figure 1, a typical banking task is not a single-turn query but a closed business loop: the model is required to parse instructions, sequentially invoke APIs (e.g., for risk assessment or transaction processing), and iterate based on system feedback.

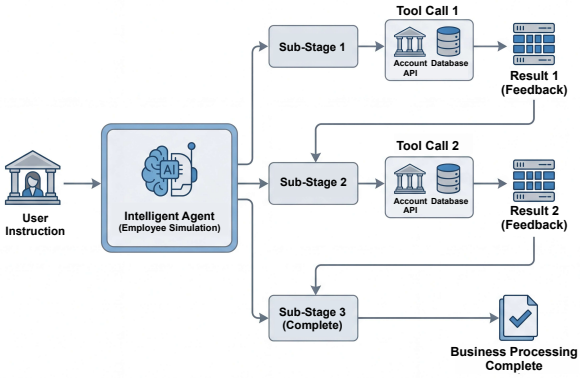


Figure 1: Tool-calling process for LLMs in banking. Models decompose instructions, sequentially invoke APIs, and iterate based on feedback to complete end-to-end banking workflows.

Existing evaluation methodologies, however, fall short of capturing this inherent complexity. On the one hand, financial benchmarks like FinQA (Chen et al., 2021) and FinEval (Guo et al., 2025) focus primarily on static knowledge and reasoning (Q&A), failing to evaluate the autonomous planning and execution required in dynamic business workflows. On the other hand, general tool-learning benchmarks like ToolBench (Qin et al., 2024) and API-Bank (Li et al., 2023b) lack the complex logic specific to banking—such as multi-level approval chains and credit line constraints. Moreover, stringent privacy regulations concerning real user data and production interfaces have resulted in a pronounced scarcity of high-fidelity, practical evaluation resources in this domain.

To address these gaps, we propose **T**ool-**E**nhanced **LLM** Agent **E**valuation for **R**eal-

[1] All codes and dataset will be released upon acceptance.

World Banking (TELLER), a specialized benchmark designed to evaluate LLMs in banking tool-calling scenarios. By synthesizing a high-fidelity banking business environment—avoiding privacy risks—TELLER thoroughly evaluates a model's ability to plan and complete tasks through multi-turn tool calls. It covers five major business domains: loans, account management, credit cards, savings, and investments, containing 1,033 test instances. We construct a fully automated testing framework to verify model performance on complex, long-horizon tasks using the ReAct paradigm (Yao et al., 2022).

We evaluate TELLER on 14 LLMs from five major model families: Claude, Gemini (Team et al., 2023), GPT (Achiam et al., 2023), DeepSeek (Liu et al., 2024a), and Qwen (Yang et al., 2025). Our results show that even the best model, Gemini-3-Pro, achieves only 38.0% accuracy in complex banking workflows, revealing the significant challenges LLMs face in practical industry applications.

In summary, the main contributions of this paper are as follows:

- We present TELLER, the first benchmark for complex banking processes. It includes five banking scenarios, 1,033 tests, and hundreds of tools to evaluate LLMs' tool selection, planning, and multi-hop tool use. The dataset features complex, executable tool dependencies with verifiable answers.
- We introduce a fully automated, LLM-driven pipeline that generates database templates, SOPs, API graphs, user data, and code from descriptions, along with a unified evaluation pipeline for measuring model performance on banking SOPs.
- A comprehensive evaluation of 14 LLMs highlights their current limitations in handling complex banking tasks with various tools, providing an open, reproducible benchmark for future industry agent research.

## 2 Related Works

**Model Enhancement in Finance.** In recent years, the application of large language models (LLMs) in finance has advanced along two parallel directions: enhancing model capabilities and constructing evaluation systems. To improve model expertise, researchers have employed domain-specific training and multi-agent architectures. Zhang et al. (2023) optimized financial news classification with RAG and instruction tuning. Models like FinGPT (Wang et al., 2023), XuanYuan 2.0 (Zhang and Yang, 2023), and DISC-FinLLM (Chen et al., 2023) enhanced adaptability via financial data pre-training and expert fine-tuning, while FinMem (Li et al., 2024a) designed a multi-agent architecture with memory for financial decision simulation.

| Benchmark | SOP | Tools | Beyond QA | Automated Build |
|---|---|---|---|---|
| Golden Touchstone | ✗ | ✗ | ✗ | ✗ |
| CFinBench | ✗ | ✗ | ✗ | ✗ |
| PIXIU | ✗ | ✗ | ✗ | ✗ |
| FinEval | ✗ | ✗ | ✓ | ✗ |
| KRX-Bench | ✗ | ✗ | ✗ | ✓ |
| Finance Agent | ✗ | ✓ | ✓ | ✗ |
| INVESTORBENCH | ✗ | ✓ | ✓ | ✗ |
| TELLER | ✓ | ✓ | ✓ | ✓ |

Table 1: Comparison with existing financial benchmarks, ordered by capability coverage.

**Financial Evaluation Benchmarks.** For evaluation, benchmarks such as CFLUE (Zhu et al., 2024), FinEval (Guo et al., 2025), and Golden Touchstone (Anonymous, 2025e) focus on fundamental financial Q&A. BizFinBench (Anonymous, 2025c) and FinMaster (Anonymous, 2025d) simulate real workflows to provide business-aligned evaluation standards. PIXIU (Xie et al., 2023) extends assessment to multimodal and complex financial forecasting tasks, promoting evaluation diversity. However, as shown in Table 1, existing benchmarks lack comprehensive support for SOP-based workflows, tool invocation, and automated construction.

**Cross-Domain Agent Simulation.** LLM-based agent simulation has become key for exploring complex real-world scenarios, with applications expanding across domains. In healthcare, works like Agent Hospital (Li et al., 2024b) and Organ Agents (Chang et al., 2025) simulate diagnostic processes. In finance, FinTeam (Wu et al., 2025) and QuantAgents (Li et al., 2025) model expert collaboration and market behavior. In social and commercial domains, Generative Agents (Park et al., 2023) and AgentSociety (Piao et al., 2025) study daily behaviors and social phenomena, while LMAgent (Liu et al., 2024b) simulates multi-user e-commerce activities. Concurrently, agent frameworks have evolved. Li et al. (2023a) advanced collaborative cognition mechanisms, and AgentScope

(Gao et al., 2025) supports large-scale simulations via a distributed architecture. Specialized scenarios, such as MLAB (Hao and Xie, 2025) for policy analysis and RESEARCHTOWN (Yu et al., 2025) for scientific communities, have emerged as research foci, collectively shifting agent roles from human simulators to domain experts.

**Tool Calling and Process Evaluation.** Tool calling is crucial for advancing LLMs from chatbots to agents capable of execution. To enhance this ability, LOOPTOOL (Zhang et al., 2025) improves robustness via a data-training loop, NexusRaven (Srinivasan et al., 2023) refines function calling through data curation, and PTool (Anonymous, 2025a) incorporates user preferences for personalized interactions. Evaluation has also evolved from static, outcome-based metrics to dynamic, multi-dimensional trajectory assessment. Early benchmarks like API-Bank (Li et al., 2023b) provided basic tests for tool planning and calling, and BFCL (Patil et al.) established standardized leaderboards for model comparison. Recent benchmarks emphasize granular process metrics. SOPBench (Anonymous, 2025f) and TRAJECT-BENCH (Anonymous, 2025g) evaluate step correctness and rule compliance, ToolSandbox (Lu et al., 2025) assesses tool use with stateful interactions, and WildToolBench (Anonymous, 2025b) evaluates performance under ambiguous and shifting user instructions.

## 3 The TELLER Benchmark

We propose TELLER, a benchmark designed to evaluate Large Language Models (LLMs) on complex banking operations. Unlike benchmarks focusing on static knowledge, TELLER simulates dynamic workflows modeled as Standard Operating Procedures (SOPs). As shown in Figure 2, we implement a fully automated pipeline that iteratively synthesizes business contexts, structured data templates, SOP documents, API dependency graphs, and executable code.

The benchmark covers five core banking scenarios: Loans, Savings, Investments, Account Management, and Credit Cards. As illustrated in Figure 3, these scenarios encompass 1,033 distinct business queries. Each scenario includes extensive task contexts, SOP guidance, and hundreds of API instances to systematically assess procedural reasoning, tool invocation, and constraint compliance.

We employ a rigorous prompt engineering pipeline to synthesize the benchmark components incrementally. Utilizing models such as Gemini-3-Pro and Claude-Sonnet-4.5, we generate artifacts for each scenario starting from a high-level business theme. Detailed prompts for each stage are provided in Appendix A.1.

### 3.1 Business Context and Data Templates

The pipeline begins by synthesizing the *Business Context*, which defines objectives, sub-processes, and constraints. Based on this context, we generate *Data Templates* that specify input/output fields and data types. To ensure rigorous data flow dependencies, initial input fields (e.g., account numbers) are instantiated, while output fields (e.g., credit scores) remain distinct to be backfilled by later operations.

### 3.2 SOPs and API Dependency Graphs

We synthesize structured SOP documents outlining procedural flow, prerequisites, and decision logic. These SOPs serve as executable refinements of the business context. Simultaneously, we construct an *API Dependency Graph* to map operational statements in the SOP to specific API calls. This directed graph explicitly defines the topological relationships between APIs, detailing input/output parameters and direct dependencies, thereby ensuring logical coherence for tool-calling orchestration.

### 3.3 User Data and API Code Generation

In this stage, we transform the structural skeletons into executable artifacts.

**User Data Synthesis.** We generate diverse simulated user records by prompting the model to identify decision points and boundary conditions within the SOP. This ensures the data covers various edge cases and validation rules defined in the templates, and sensitive fields in the synthesized user data are masked with *** to prevent potential privacy leaks.

**API Code Implementation.** We generate executable Python code for banking APIs in two phases: first mapping SOP tasks to atomic API metadata, then synthesizing the implementation code and JSON schemas. Crucially, the code includes a global simulated database to support CRUD operations, enabling state tracking across multi-step workflows.

As shown in Figure 4, the synthesized APIs exhibit high complexity, with over half requiring more than six parameters. To enhance difficulty, we incorporate distractors from public repositories, resulting in a pool of approximately 280 available

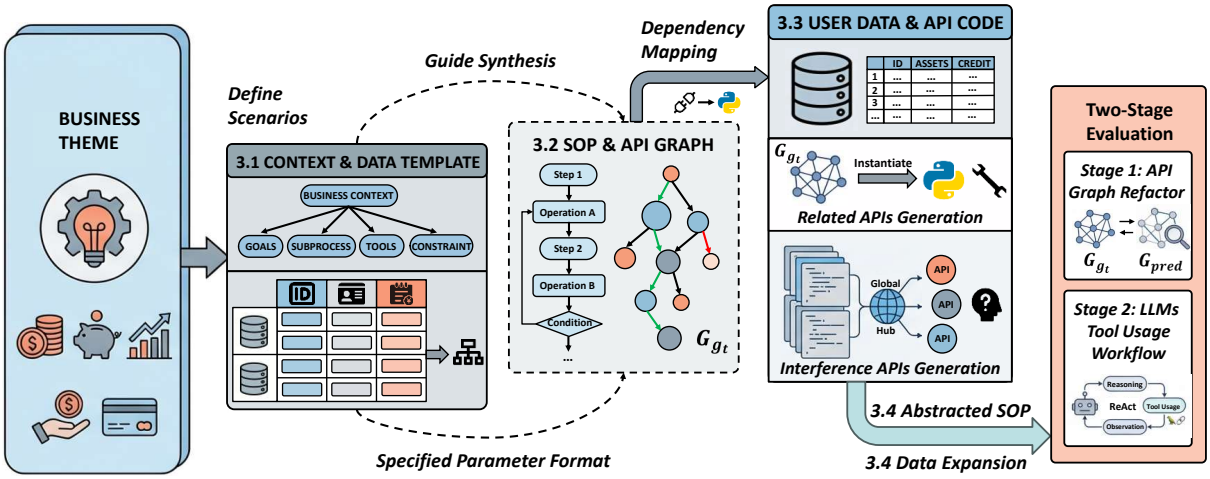


Figure 2: Overview of the TELLER construction pipeline and evaluation framework. The pipeline synthesizes artifacts in three stages: (1) Context & Data Templates, (2) SOPs & API Graphs, and (3) User Data & API Code. The evaluation assesses model performance via API graph reconstruction and end-to-end agent execution.

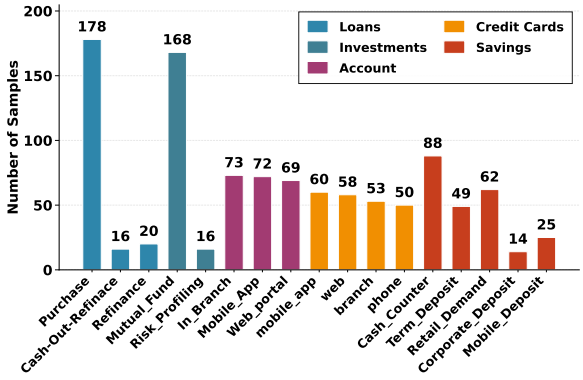


Figure 3: Distribution of sub-tasks across the five banking scenarios in TELLER.

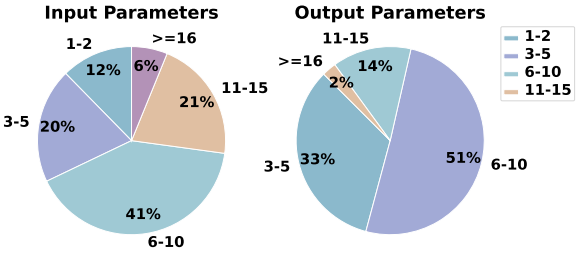


Figure 4: Parameter count distribution for APIs. Over 50% of APIs involve more than six parameters.

APIs per scenario (see Appendix A.3 for detailed statistics).

### 3.4 SOP Abstraction and Data Expansion

To evaluate generalized execution capabilities rather than rote memorization, we introduce two complexity-enhancing mechanisms:

**SOP Abstraction.** We rewrite specific sub-steps to simulate real-world ambiguity. This involves (1) introducing implicit decision points, (2) using ambiguous expressions for verification steps, and (3) embedding multi-path exception handling. These abstractions force the agent to infer rules and plan validation paths autonomously. Examples of these abstraction techniques appear in Appendix A.2.

**Data Expansion.** Since the initial synthesis yields limited samples, we perform iterative large-scale generation based on the original templates. This expansion results in a total of 1,113 queries across the five scenarios, ensuring comprehensive coverage of business logic.

### 3.5 Standard Result Generation

To establish a ground truth for evaluation, we construct a deterministic execution script for every user query. We synthesize a "Standard Result Generator" code that strictly adheres to the SOP logic to invoke the APIs generated in Section 3.3. This script processes the expanded dataset and records the correct API execution sequence and final outputs. After filtering queries with structural errors, we retain 1,033 valid queries (Loans: 214, Account: 214, Credit Cards: 221, Savings: 200, Investments: 184) to serve as the objective baseline.

### 3.6 Benchmark Validation

To ensure TELLER's quality, five on-campus undergraduate annotators were recruited and compensated in cash ($20/hour) to validate key pipeline stages following Appendix B.

Manual review of all 1,113 instances confirmed 92.8% validity, verifying realistic customer profiles, business-aligned transaction amounts, and logical consistency. Key APIs were reviewed for regulatory compliance, calculation accuracy (e.g., interest rates), and exception handling; scenarios with >10% flagged APIs underwent re-synthesis and correction. Field-by-field inspection of 300 sampled outputs verified completeness and business rule adherence, achieving 93.7% correctness with remaining errors manually corrected.

# 4 Experiments Settings

## 4.1 Evaluation Framework

We design a two-stage evaluation process to assess both the structural understanding of business logic and the practical execution of tasks.

**Stage 1: API Graph Reconstruction.** This stage evaluates the model's comprehension of the workflow. The model is provided with the business context and the full set of API signatures (including distractors) and is tasked with reconstructing the API dependency graph ($G_{pred}$). We compare $G_{pred}$ against the ground-truth graph ($G_{gt}$) to quantify the understanding of data flow and dependencies.

In this stage, we treat APIs as nodes and dependencies as edges. We calculate **Precision**, **Recall**, and **F1** scores for both *Node Selection* (identifying correct APIs) and *Edge Prediction* (inferring correct dependencies). Additionally, we compute graph similarity indices to measure overall structural alignment.

**Stage 2: Agentic Tool Use & Execution.** This stage assesses end-to-end execution. The model acts as a ReAct agent, provided with the abstracted SOP, user data, and the toolset. It receives a vague instruction (e.g., "Handle the application for [ID]") and must autonomously plan steps, interact with the simulated database, and execute the workflow.

In this stage, we evaluate the agent's execution trace against the standard result:

- **Tool-Calling Performance:** We report Precision and Recall for the set of invoked APIs.
- **Process Correctness:** We define accuracy metrics: *Final-review Acc.*, measuring the exact match rate of the final decision outcomes.

## 4.2 Evaluated Models

We employ TELLER to evaluate a range of representative open- and closed-source models on their ability to handle banking tasks with complex, long-horizon tool calls. The evaluated closed-source LLMs include Claude-haiku-4.5, Claude-Sonnet-4.5, GPT-5, GPT-4o, Gemini-3-Pro and Gemini-2.5-Pro. The open-source models include DeepSeek-V3.2, Qwen3-Max, the Lightweight Qwen3 series (8B, 14B, 32B), as well as specialized agent models like rStar2-Agent-14B (Shang et al., 2025), SkyRL-Agent-14B-v0 (Cao et al., 2025), and SA-SWE-32B.

## 4.3 Implementation Details

In the pipeline construction phase, only the business theme and context generation stages use the highly creative Gemini-3-Pro model. All subsequent stages utilize the code-proficient Claude-Sonnet-4.5 as the base model. The inference temperature is set to 0.1 for all models. Open-source LLMs with fewer than 70B parameters are deployed on a local Slurm server using their official chat templates (hardware details are in the Appendix E). Larger models and all closed-source models are evaluated via their respective APIs. To ensure evaluation consistency, all tool APIs are bound to models through LangChain using the OpenAI format, and are tested using the ReAct paradigm for tool calling.

# 5 Experimental Results

## 5.1 Main Result

The main evaluation results are presented in Table 2. We identify five key findings:

**Limited Multi-Step Tool Invocation Capability.** Despite reinforcement training for tool use, state-of-the-art LLMs struggle with complex, long-horizon banking workflows. The best-performing model, Gemini-3-Pro, achieves only 38% Stage 2 execution accuracy. Among open-source models, DeepSeek-V3.2 reaches 23.2%, while smaller models (<32B parameters) fall below 11%.

**Model Scale and Agent Enhancement Effects.** Larger models and agent-enhanced variants demonstrate superior performance. Qwen3 series execution accuracy improves from 0% (8B) to 9.4% (32B). Both Qwen3-14B and Qwen3-8B score zero on all Stage 1 edge metrics, indicating models <14B cannot reconstruct tool dependencies. Agent models (rStar2-Agent-14B, SkyRL-Agent-14B) outperform Qwen3-14B and approach Qwen3-32B performance, while SA-SWE-32B

| Model | Stage 1 | | | | | | | | | Stage 2 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Overlap | | | Node | | | Edge | | | Execute | | | |
| | Node | Edge | Path | P | R | F1 | P | R | F1 | Acc. | P | R | F1 |
| Claude-Sonnet-4.5 | 0.9711 | 0.8050 | 0.6866 | 0.9485 | 0.9692 | 0.9556 | 0.8553 | 0.9292 | 0.8888 | 0.3440 | 0.9638 | 0.7974 | 0.8628 |
| Claude-Haiku-4.5 | 0.7435 | 0.5933 | 0.4404 | 0.7435 | 1.0000 | 0.8198 | 0.6386 | 0.8434 | 0.7059 | 0.3560 | 0.9659 | 0.6270 | 0.7416 |
| GPT-5 | 0.8274 | 0.5826 | 0.3741 | 0.8274 | 1.0000 | 0.9028 | 0.6021 | 0.9545 | 0.7287 | 0.2760 | 0.9611 | 0.5577 | 0.6500 |
| GPT-4o | 0.6122 | 0.2505 | 0.0818 | 1.0000 | 0.6122 | 0.7361 | 0.9178 | 0.2600 | 0.3894 | 0.1440 | 0.6868 | 0.2395 | 0.3415 |
| Gemini-3-Pro | 0.9900 | 0.6204 | 0.4399 | 1.0000 | 0.9900 | 0.9949 | 0.8853 | 0.6726 | 0.7585 | **0.3800** | 0.8671 | 0.6810 | 0.7453 |
| Gemini-2.5-Pro | 1.0000 | 0.8240 | 0.6231 | 1.0000 | 1.0000 | 1.0000 | 0.8609 | 0.7394 | 0.8980 | 0.2360 | 0.9810 | 0.4889 | 0.6180 |
| DeepSeek-V3.2 | 0.6418 | 0.5910 | 0.4074 | 0.9357 | 0.8965 | 0.9068 | 0.8505 | 0.6515 | 0.7232 | 0.2320 | 0.6007 | 0.4520 | 0.5476 |
| Qwen3-Max | 0.7378 | 0.5958 | 0.4770 | 0.7378 | 1.0000 | 0.8243 | 0.6286 | 0.8939 | 0.7242 | 0.1960 | 0.8780 | 0.5856 | 0.6789 |
| rStar2-Agent-14B | 0.6911 | 0.1515 | 0.0980 | 0.7895 | 0.8955 | 0.7970 | 0.1981 | 0.2391 | 0.2082 | 0.0900 | 0.9128 | 0.3873 | 0.5218 |
| SkyRL-Agent-14B | 0.8603 | 0.1903 | 0.0759 | 0.9077 | 0.9455 | 0.9206 | 0.3105 | 0.2543 | 0.2745 | 0.0940 | 0.9794 | 0.4897 | 0.6210 |
| SA-SWE-32B | 0.7719 | 0.2889 | 0.1356 | 0.8000 | 0.9765 | 0.8624 | 0.3993 | 0.4435 | 0.4101 | 0.1080 | 0.9543 | 0.3100 | 0.4347 |
| Qwen3-32B | 0.9418 | 0.2035 | 0.0430 | 1.0000 | 0.9418 | 0.9683 | 0.3237 | 0.3159 | 0.3036 | 0.0940 | 0.9524 | 0.3050 | 0.4300 |
| Qwen3-14B | 0.8216 | 0.0000 | 0.0000 | 0.8784 | 0.9401 | 0.9010 | 0.0000 | 0.0000 | 0.0000 | 0.0600 | 0.9910 | 0.4895 | 0.6253 |
| Qwen3-8B | 0.7692 | 0.0000 | 0.0000 | 0.7692 | 1.0000 | 0.9524 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.2283 | 0.0414 | 0.0678 |

Table 2: Two-stage Metric Averages of Fourteen Tested Models on the TELLER Benchmark. Stage 1 includes overlap metrics (Node, Edge, Path) between the ground-truth graph $G_{gt}$ and the predicted graph $G_{pred}$; together with node and edge level precision(P) and recall(R) of $G_{pred}$ relative to $G_{gt}$. Stage 2 includes execution accuracy (Acc.), and precision (P) and recall (R) of tool-call results. Cell colors from red to green denote increasing accuracy.

surpasses Qwen3-32B in Stage 2 accuracy and tool-invocation metrics.

**Proprietary vs. Open-Source Model Hierarchy.** Closed-source models outperform open-source counterparts across all 13 metrics. Based on ten representative metrics visualized in Figure 5, models form three tiers: (1) leading proprietary models (Claude-Sonnet-4.5, Claude-Haiku-4.5, Gemini-3-Pro, Gemini-2.5-Pro, GPT-5); (2) large open-source models (DeepSeek-V3.2, Qwen3-Max); (3) smaller open-source models and GPT-4o. Tier 3 exhibits significantly lower Node_F1, Overlap_Path, Overlap_Edge, and Execute_Recall, indicating weaker planning and execution capabilities. GPT-4o's placement in Tier 3 stems from low Stage 1 scores despite competitive Stage 2 accuracy, suggesting strength in tool invocation but weakness in structured dependency reasoning.

**Stage 1-Stage 2 Performance Gap.** Most models show substantial performance gaps between stages, with Stage 1 metrics significantly exceeding Stage 2 results. Gemini-3-Pro achieves >99% on node metrics and accurate dependency prediction, yet only 38% execution accuracy. Qwen3-32B's Stage 1 metrics approach proprietary models, but Stage 2 accuracy drops to 9.4%. This indicates models can identify relevant tools and understand invocation patterns but struggle with execution correctness in complex industry-specific scenarios.

**Node-Edge and Precision-Recall Asymmetries.** Node metrics consistently exceed edge metrics in Stage 1, while Stage 2 precision surpasses recall across all models-patterns amplified in weaker models. GPT-5 achieves 90.28% Node_F1 vs. 72.87% Edge_F1, and 96.11% precision vs. 55.77% recall. Qwen3-32B shows 96.83% Node_F1 vs. 30.36% Edge_F1, and 95.24% precision vs. 30.5% recall. This reflects that selecting business-relevant tools is easier than planning their coordination. Higher precision than recall may result from: (1) Stage 1 filtering reducing irrelevant tools; (2) hallucination-avoidance mechanisms preventing uncertain tool calls; (3) execution errors breaking invocation chains in multi-step workflows (see Appendix C for detailed analysis).

### 5.2 Further Studies

**Performance Variance by Scenario.** Cross-scenario analysis reveals significant capability variations. As Table 3 shows, average accuracy in Loan (4.6%) and Account (4.7%) is substantially lower than Credit Cards (38.0%) and Deposit (31.7%), indicating complex approval workflows demand stronger long-range reasoning. No model excels across all scenarios: Gemini-3-Pro leads in Credit Cards (66%) and Deposit (62%), while Claude-Haiku-4.5 unexpectedly outperforms Claude-Sonnet-4.5 in Invest (52% vs. 48%). Models exhibit distinct precision-recall trade-offs: in complex Loan/Account scenarios, they adopt conservative strategies (90.2% precision, 44.8% recall); in simpler Credit Cards scenarios, they show balanced distributions. These findings necessitate scenario-specialized model selection.

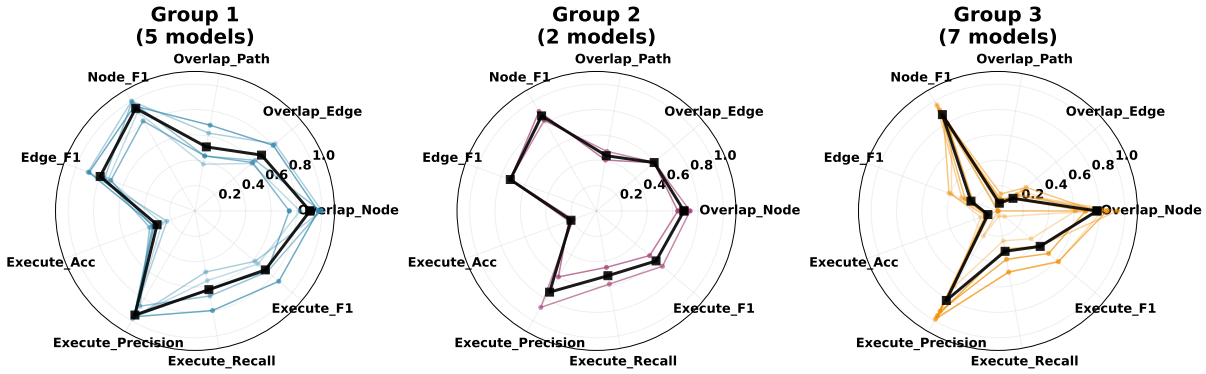


Figure 5: Multidimensional capability radar charts for three model tiers on TELLER. Group 1: leading proprietary models; Group 2: large open-source models; Group 3: smaller/specialized models. Charts show performance across Stage 1 (path generation) and Stage 2 (tool execution) metrics.

| Model | Loans | | | | Account Management | | | | Credit Cards | | | | Savings | | | | Investment | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Acc. | P | R | F1 | Acc. | P | R | F1 | Acc. | P | R | F1 | Acc. | P | R | F1 | Acc. | P | R | F1 |
| Claude-Sonnet-4.5 | **0.2200** | 0.9521 | 0.8652 | 0.9066 | **0.2200** | 0.9104 | 0.8483 | 0.8778 | 0.4000 | 1.0000 | 0.8308 | 0.8957 | 0.4000 | 0.9579 | 0.6554 | 0.7671 | 0.4800 | 0.9988 | 0.7874 | 0.8667 |
| Claude-Haiku-4.5 | 0.0600 | 0.9708 | 0.6933 | 0.7862 | 0.0400 | 0.9925 | 0.6182 | 0.7390 | 0.5600 | 1.0000 | 0.5017 | 0.6647 | 0.6000 | 0.8663 | 0.6017 | 0.6989 | **0.5200** | 1.0000 | 0.7211 | 0.8190 |
| GPT-5 | 0.0400 | 0.9644 | 0.6371 | 0.7455 | 0.0000 | 1.0000 | 0.1800 | 0.3049 | 0.4200 | 1.0000 | 0.7300 | 0.7816 | 0.4400 | 0.8411 | 0.5183 | 0.6224 | 0.4800 | 1.0000 | 0.7233 | 0.7955 |
| GPT-4o | 0.0000 | 0.9933 | 0.3657 | 0.5262 | 0.0000 | 0.4800 | 0.0855 | 0.1449 | 0.2200 | 0.5800 | 0.2217 | 0.3060 | 0.5000 | 0.5806 | 0.2367 | 0.3315 | 0.0000 | 0.8000 | 0.2878 | 0.3989 |
| Gemini-3-Pro | 0.1200 | 0.7668 | 0.5076 | 0.5761 | 0.1400 | 0.8863 | 0.8036 | 0.8382 | **0.6600** | 0.9224 | 0.7267 | 0.7939 | **0.6200** | 0.8318 | 0.6650 | 0.7236 | 0.3600 | 0.9284 | 0.7022 | 0.7947 |
| Gemini-2.5-Pro | 0.0000 | 1.0000 | 0.3171 | 0.4767 | 0.1200 | 0.9049 | 0.7236 | 0.7858 | 0.5600 | 1.0000 | 0.4383 | 0.5704 | 0.5000 | 1.0000 | 0.4033 | 0.5627 | 0.0000 | 1.0000 | 0.5622 | 0.6945 |
| DeepSeek-V3.2 | 0.1000 | 1.0000 | 0.7600 | 0.8490 | 0.0000 | 0.6658 | 0.3073 | 0.4112 | 0.3600 | 0.6800 | 0.3617 | 0.4342 | 0.5400 | 0.8600 | 0.4050 | 0.5321 | 0.1600 | 0.7979 | 0.4267 | 0.5117 |
| Qwen3-Max | 0.0800 | 1.0000 | 0.4295 | 0.5866 | 0.0000 | 0.7701 | 0.3432 | 0.4743 | 0.5600 | 0.7600 | 0.6333 | 0.6723 | 0.3400 | 0.8800 | 0.7000 | 0.7714 | 0.0000 | 0.9800 | 0.8222 | 0.8899 |
| rStar2-Agent-14B | 0.0000 | 0.9776 | 0.3333 | 0.4873 | 0.0000 | 0.9570 | 0.3868 | 0.5392 | 0.2400 | 0.6800 | 0.3495 | 0.4299 | 0.1300 | 0.9971 | 0.5267 | 0.6671 | 0.0400 | 0.9524 | 0.3403 | 0.4854 |
| SkyRL-Agent-14B | 0.0200 | 0.9966 | 0.4152 | 0.5681 | 0.0000 | 0.9545 | 0.3525 | 0.5029 | 0.3700 | 0.9528 | 0.7372 | 0.8114 | 0.0800 | 0.9988 | 0.5583 | 0.6868 | 0.0000 | 0.9954 | 0.3853 | 0.5360 |
| SA-SWE-32B | 0.0000 | 0.9860 | 0.3181 | 0.4361 | 0.0000 | 0.8461 | 0.3975 | 0.5194 | 0.3900 | 0.9434 | 0.3605 | 0.4935 | 0.1500 | 1.0000 | 0.2621 | 0.3882 | 0.0000 | 0.9960 | 0.2117 | 0.3363 |
| Qwen3-32B | 0.0500 | 0.8782 | 0.3800 | 0.4936 | 0.0000 | 0.9578 | 0.4172 | 0.5751 | 0.3400 | 0.9262 | 0.3137 | 0.4403 | 0.0800 | 1.0000 | 0.2050 | 0.3163 | 0.0000 | 1.0000 | 0.2089 | 0.3245 |
| Qwen3-14B | 0.0000 | 0.9928 | 0.4031 | 0.5539 | 0.0000 | 0.9876 | 0.3714 | 0.5297 | 0.2200 | 0.9830 | 0.7391 | 0.8241 | 0.0800 | 0.9959 | 0.5279 | 0.6593 | 0.0000 | 0.9959 | 0.4076 | 0.5589 |
| Qwen3-8B | 0.0000 | 0.4172 | 0.1324 | 0.1885 | 0.0000 | 0.0232 | 0.0180 | 0.0199 | 0.0000 | 0.0260 | 0.0132 | 0.0171 | 0.0200 | 0.3950 | 0.1458 | 0.1984 | 0.0000 | 0.2283 | 0.0414 | 0.0678 |

Table 3: Performance of various large language models across the five business scenarios in TELLER during the second evaluation stage, including result Acc. as well as precision (P), recall (R), and F1 metrics for tool invocation.

| Scenario | Nodes | Edges | Density | Acc. |
|---|---|---|---|---|
| Loans | 20 | 34 | 0.0895 | 0.046 |
| Account | 10 | 16 | 0.1778 | 0.047 |
| Credit Cards | 11 | 17 | 0.1545 | 0.380 |
| Savings | 13 | 26 | 0.1667 | 0.317 |
| Investments | 17 | 23 | 0.0846 | 0.191 |

Table 4: API graph properties and execution accuracy across five business scenarios.

**Complexity of $G_{gt}$ and Stage 1 Evaluation Difficulty.** Figure 6 shows that **edge count** dominates Stage 1 difficulty, exhibiting strong negative correlation with Path Overlap (Pearson $r = -0.749$, $p = 0.145$) and moderate correlation with Edge Overlap ($r = -0.485$, $p = 0.408$). The Loan scenario (34 edges) achieves 21.5% Path Overlap versus Account (16 edges) at 28.3%, despite similar Stage 2 accuracy. In contrast, node count and density show weaker correlations, confirming **dependency complexity** as the primary determinant of structural reasoning difficulty.

**Impact of $G_{gt}$ Complexity on Execution.** Table 4 presents API graph properties and Stage 2 accuracy. While Loan (lowest accuracy) has the most nodes/edges, Account (similarly low accuracy) has the fewest, suggesting graph complexity alone does not determine execution difficulty. Further analysis reveals Loan and Account require six output fields versus three for other scenarios, likely contributing to their higher difficulty. The observed execution difficulty is likely influenced by multiple interacting factors, including dependency graph complexity, SOP constraints, API internal complexity, and output schema richness.

**Statistical Correlation Analysis.** Despite substantial performance gaps between stages, strong correlations exist between certain Stage 1 and Stage 2 metrics. As shown in the statistical analysis of Figure 7, we observe:

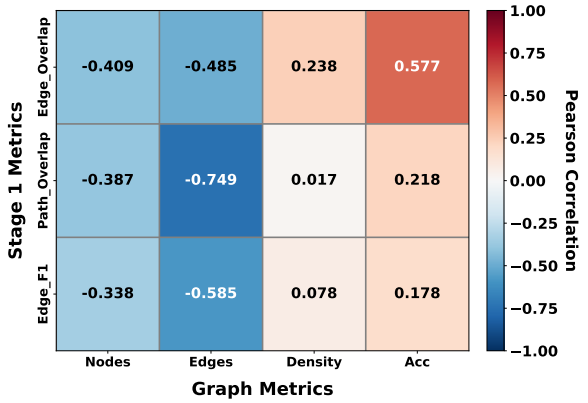


Figure 6: Correlation matrix between graph-structural properties and Stage 1 evaluation metrics. Edge count exhibits the strongest negative correlation with overlap metrics, particularly for Path Overlap ($r = -0.749$).

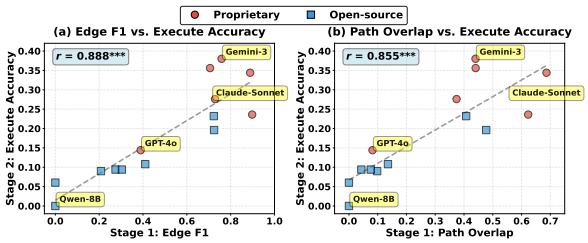


Figure 7: Correlation between Stage 1 and Stage 2 performance. (a) Edge F1 vs. execution accuracy: $r = 0.888$, $p < 0.001$. (b) Path Overlap vs. execution accuracy: $r = 0.855$, $p < 0.001$, indicating path reasoning better predicts execution success. Proprietary models (circles) outperform open-source (squares).

- **Edge F1 vs. Execute Accuracy**: Strong positive correlation (Pearson $r = 0.888$, $p < 0.001$) confirms accurate dependency inference is crucial.
- **Path Overlap vs. Execute Accuracy**: Stronger correlation ($r = 0.855$, $p < 0.001$) shows end-to-end path understanding best predicts execution success.

These findings validate TELLER's structural design and indicate API dependency identification is a key prerequisite for correct execution. Based on Figures 5 and 7, Claude-Sonnet-4.5 achieves state-of-the-art performance on TELLER's complex banking tasks.

**Business Error Analysis.** To understand execution failures, we categorize tool invocation errors (Table 5) into three types: Blank (no tool invocation), Validation (parameter validation failure), and Type (type mismatch).

| Model | Blank | Valid. | Type. |
|---|---|---|---|
| Claude-Sonnet-4.5 | 0.2% | 0 | 8.0% |
| Claude-Haiku-4.5 | 0 | 1.2% | 11.6% |
| GPT-5 | 0 | 55.8% | 2.4% |
| GPT-4o | 26.0% | 24.0% | 1.6% |
| Gemini-3-Pro | 0 | 1.2% | 6.4% |
| Gemini-2.5-Pro | 0 | 23.2% | 13.2% |
| DeepSeek-V3.2 | 16.8% | 9.4% | 12.0% |
| Qwen3-Max | 14.2% | 3.2% | 45.6% |
| rStar2-Agent-14B | 4.8% | 40.4% | 13.8% |
| SkyRL-Agent-14B | 0 | 41.4% | 4.8% |
| SA-SWE-32B | 1.0% | 22.8% | 11.4% |
| Qwen3-32B | 1.6% | 17.4% | 13.4% |
| Qwen3-14B | 0.4% | 50.0% | 5.4% |
| Qwen3-8B | 57.8% | 6.4% | 0 |

Table 5: Tool-Call error distribution (%). Blank: no tool call; Valid.: parameter error; Type.: type mismatch.

**Model Capability and Error Patterns.** Different models exhibit distinct error tendencies. Small models (e.g., Qwen3-8B) show high Blank rates (57.8%), indicating abandonment under complexity. Large proprietary models (Gemini-3-Pro, Claude-Sonnet-4.5) achieve low total error rates (7.6%, 8.2%) with near-zero Blank errors, demonstrating superior task comprehension.

**Agent Model Specialization.** Agent models (rStar2-Agent-14B, SkyRL-Agent-14B) exhibit low Blank rates (4.8%, 0%) but high Validation errors (40.4%, 41.4%), suggesting they recognize when to invoke tools but struggle with parameter construction, highlighting current RL limitations in precise parameter generation.

## 6 Conclusion

In this paper, we introduce TELLER, a novel benchmark for evaluating LLMs' multi-step tool invocation in complex banking workflows. TELLER employs a two-stage framework encompassing dependency graph reconstruction and end-to-end execution, overcoming prior limitations through realistic SOP constraints, complex API dependencies, and verifiable results across five scenarios. Benchmarking 14 LLMs reveals the best model achieves only 38% execution accuracy, while open-source models below 32B parameters fall below 11%. Further studies reveal deficiencies in understanding tool dependencies and precise invocation, providing insights for future optimization. By establishing a high-quality benchmark for diverse banking workflows, TELLER lays the groundwork for advancing LLM agent deployment in real-world financial industries.

## Limitations

While our TELLER benchmark effectively evaluates LLM performance in complex banking workflows, we acknowledge that synthetic user data may inadvertently reflect real-world banking user information patterns from the LLM's training corpus—though we have verified that the LLM employs placeholder generation and explicit safety prompts to mitigate potential privacy leakage. Another limitation stems from the complexity of tool invocation chains, which leads to relatively high evaluation costs; however, the scalability of our SOP synthesis pipeline can be readily adapted to create larger-scale datasets for future research. Additionally, our detailed analysis of current tool-use characteristics offers valuable insights that can serve as a foundation for future research in banking automation.

## References


Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Anonymous. 2025a. Advancing and benchmarking personalized tool invocation for LLMs. In *Submitted to The Fourteenth International Conference on Learning Representations*. Under review.

Anonymous. 2025b. Benchmarking LLM tool-use in the wild. In *Submitted to The Fourteenth International Conference on Learning Representations*. Under review.

Anonymous. 2025c. Bizfinbench: A business-driven real-world financial benchmark for evaluating LLMs. In *Submitted to The Fourteenth International Conference on Learning Representations*. Under review.

Anonymous. 2025d. Finmaster: A holistic benchmark for full-pipeline financial management with large language models. *Submitted to Transactions on Machine Learning Research*. Under review.

Anonymous. 2025e. Golden touchstone: A comprehensive bilingual benchmark for evaluating financial large language models. In *Submitted to ACL Rolling Review - May 2025*. Under review.

Anonymous. 2025f. SOPBench: Evaluating language agents at following standard operating procedures and constraints. In *Submitted to The Fourteenth International Conference on Learning Representations*. Under review.

Anonymous. 2025g. TRAJECT-bench:a trajectory-aware benchmark for evaluating agentic tool use. In *Submitted to The Fourteenth International Conference on Learning Representations*. Under review.

Shiyi Cao, Dacheng Li, Fangzhou Zhao, Shuo Yuan, Sumanth R. Hegde, Connor Chen, Charlie Ruan, Tyler Griggs, Shu Liu, Eric Tang, Richard Liaw, Philipp Moritz, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2025. Skyrl-agent: Efficient rl training for multi-turn llm agent. *Preprint*, arXiv:2511.16108.

Rihao Chang, He Jiao, Weizhi Nie, Honglin Guo, Keliang Xie, Zhenhua Wu, Lina Zhao, Yunpeng Bai, Yongtao Ma, Lanjun Wang, and 1 others. 2025. Organ-agents: Virtual human physiology simulator via llms. *arXiv preprint arXiv:2508.14357*.

Wei Chen, Qiushi Wang, Zefei Long, Xianyin Zhang, Zhongtian Lu, Bingxuan Li, Siyuan Wang, Jiarong Xu, Xiang Bai, Xuanjing Huang, and 1 others. 2023. Disc-finllm: A chinese financial large language model based on multiple experts fine-tuning. *arXiv preprint arXiv:2310.15205*.

Zhi-Yuan Chen, Shiqi Shen, Guangyao Shen, Gong Zhi, Xu Chen, and Yankai Lin. 2024. Towards tool use alignment of large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 1382–1400.

Zhiyu Chen, Wenhu Chen, Charese Smiley, Sameena Shah, Iana Borova, Dylan Langdon, Reema Moussa, Matt Beane, Ting-Hao Huang, Bryan R Routledge, and 1 others. 2021. Finqa: A dataset of numerical reasoning over financial data. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3697–3711.

Dawei Gao, Zitao Li, Yuexiang Xie, Weirui Kuang, Liuyi Yao, Bingchen Qian, Zhijian Ma, Yue Cui, Haohao Luo, Shen Li, and 1 others. 2025. Agentscope 1.0: A developer-centric framework for building agentic applications. *arXiv preprint arXiv:2508.16279*.

Xin Guo, Haotian Xia, Zhaowei Liu, Hanyang Cao, Zhi Yang, Zhiqiang Liu, Sizhe Wang, Jinyi Niu, Chuqi Wang, Yanhui Wang, and 1 others. 2025. Fineval: A chinese financial domain knowledge evaluation benchmark for large language models. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 6258–6292.

Yuzhi Hao and Danyang Xie. 2025. A multi-llm-agent-based framework for economic and public policy analysis. *arXiv preprint arXiv:2502.16879*.

Jean Lee, Nicholas Stevens, and Soyeon Caren Han. 2025. Large language models in finance (finllms). *Neural Comput. Appl.*, 37(30):24853–24867.

Haohang Li, Yangyang Yu, Zhi Chen, Yuechen Jiang, Yang Li, Denghui Zhang, Rong Liu, Jordan W. Suchow, and Khaldoun Khashanah. 2024a. Finmem:

A performance-enhanced LLM trading agent with layered memory and character design. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*.

Huao Li, Yu Chong, Simon Stepputtis, Joseph P Campbell, Dana Hughes, Charles Lewis, and Katia Sycara. 2023a. Theory of mind for multi-agent collaboration via large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 180–192.

Junkai Li, Yunghwei Lai, Weitao Li, Jingyi Ren, Meng Zhang, Xinhui Kang, Siyu Wang, Peng Li, Ya-Qin Zhang, Weizhi Ma, and 1 others. 2024b. Agent hospital: A simulacrum of hospital with evolvable medical agents. *arXiv preprint arXiv:2405.02957*.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023b. API-bank: A comprehensive benchmark for tool-augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3102–3116, Singapore. Association for Computational Linguistics.

Xiangyu Li, Yawen Zeng, Xiaofen Xing, Jin Xu, and Xiangmin Xu. 2025. QuantAgents: Towards multi-agent financial system via simulated trading. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 17438–17464, Suzhou, China. Association for Computational Linguistics.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024a. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Yijun Liu, Wu Liu, Xiaoyan Gu, Yong Rui, Xiaodong He, and Yongdong Zhang. 2024b. Lmagent: A large-scale multimodal agents society for multi-user simulation. *arXiv preprint arXiv:2412.09237*.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Haoping Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, and 1 others. 2025. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for llm tool use capabilities. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 1160–1183.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E Gonzalez. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*.

Jinghua Piao, Yuwei Yan, Jun Zhang, Nian Li, Junbo Yan, Xiaochong Lan, Zhihong Lu, Zhiheng Zheng, Jing Yi Wang, Di Zhou, and 1 others. 2025. Agentsociety: Large-scale simulation of llm-driven generative agents advances understanding of human behaviors and society. *arXiv preprint arXiv:2502.08691*.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, dahai li, Zhiyuan Liu, and Maosong Sun. 2024. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *International Conference on Representation Learning*, volume 2024, pages 9695–9717.

Ning Shang, Yifei Liu, Yi Zhu, Li Lyna Zhang, Weijiang Xu, Xinyu Guan, Buze Zhang, Bingcheng Dong, Xudong Zhou, Bowen Zhang, and 1 others. 2025. rstar2-agent: Agentic reasoning technical report. *arXiv preprint arXiv:2508.20722*.

Venkat Krishna Srinivasan, Zhen Dong, Banghua Zhu, Brian Yu, Damon Mosk-Aoyama, Kurt Keutzer, Jiantao Jiao, and Jian Zhang. 2023. Nexusraven: a commercially-permissive language model for function calling. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, and 1 others. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Neng Wang, Hongyang Yang, and Christina Wang. 2023. FinGPT: Instruction tuning benchmark for open-source large language models in financial datasets. In *NeurIPS 2023 Workshop on Instruction Tuning and Instruction Following*.

Yingqian Wu, Qiushi Wang, Zefei Long, Rong Ye, Zhongtian Lu, Xianyin Zhang, Bingxuan Li, Wei Chen, Liwen Zhang, and Zhongyu Wei. 2025. Finteam: A multi-agent collaborative intelligence system for comprehensive financial scenarios. In *CCF International Conference on Natural Language Processing and Chinese Computing*, pages 443–455. Springer.

Qianqian Xie, Weiguang Han, Xiao Zhang, Yanzhao Lai, Min Peng, Alejandro Lopez-Lira, and Jimin Huang. 2023. Pixiu: A comprehensive benchmark, instruction dataset and large language model for finance. In *Advances in Neural Information Processing Systems*, volume 36, pages 33469–33484. Curran Associates, Inc.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.

Haofei Yu, Zhaochen Hong, Zirui Cheng, Kunlun Zhu, Keyang Xuan, Jinwei Yao, Tao Feng, and Jiaxuan You. 2025. Researchtown: Simulator of human research community. In *Forty-second International Conference on Machine Learning*.

Boyu Zhang, Hongyang Yang, Tianyu Zhou, Muhammad Ali Babar, and Xiao-Yang Liu. 2023. Enhancing financial sentiment analysis via retrieval augmented large language models. In *Proceedings of the fourth ACM international conference on AI in finance*, pages 349–356.

Kangning Zhang, Wenxiang Jiao, Kounianhua Du, Yuan Lu, Weiwen Liu, Weinan Zhang, Lei Zhang, and Yong Yu. 2025. Looptool: Closing the data-training loop for robust llm tool calls. *arXiv preprint arXiv:2511.09148*.

Xuanyu Zhang and Qing Yang. 2023. Xuanyuan 2.0: A large chinese financial chat model with hundreds of billions parameters. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*, CIKM '23, page 4435–4439, New York, NY, USA. Association for Computing Machinery.

Jie Zhu, Junhui Li, Yalong Wen, and Lifan Guo. 2024. Benchmarking large language models on CFLUE - a Chinese financial language understanding evaluation dataset. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 5673–5693, Bangkok, Thailand. Association for Computational Linguistics.

# A Benchmark Construction Details

## A.1 Prompt Engineering

To ensure the reproducibility and quality of TELLER, we designed a fixed reasoning pipeline. The prompts for each stage are structured as follows:

- **Business Context Generation:** Input is the business theme. The prompt instructs the model to list objectives, roles, and constraints.
- **SOP Synthesis:** Input includes the Business Context. The prompt enforces a standard document structure (Objective, Prerequisites, Procedure).
- **API Graph Construction:** Input includes the SOP and Data Template. The prompt requires outputting a YAML-formatted graph defining dependencies.
- **User Data Generation:** Input includes the Data Template and SOP. The prompt instructs generating diverse, realistic test cases covering edge conditions and policy violations.
- **API Implementation:** Input includes the API Graph and Data Template. The prompt requires generating executable Python code with input validation and deterministic outputs.

The full set of prompt templates is presented in Appendix F.

## A.2 SOP Abstraction Techniques

As mentioned in Section 3.4, we apply three types of abstraction to increase reasoning difficulty:

1. **Implicit Decision Points:** Explicit conditions (e.g., "If score > 700") are rewritten as functional requirements (e.g., "Assess credit status and decide eligibility"), requiring the agent to locate the threshold in the context.
2. **Ambiguous Expressions:** Concrete verification steps (e.g., "Call `verify_id_api`") are replaced with high-level goals (e.g., "Verify the authenticity of the client's proof"), forcing the agent to select the appropriate tool.
3. **Multi-Path Choices:** We insert potential exception scenarios (e.g., handling missing documents) without explicit branching instructions, testing the agent's ability to handle deviations.

## A.3 API Statistics

The toolset for each scenario consists of core synthesized APIs and distractors. The breakdown is as follows:

- **Loans:** 284 total APIs.
- **Account Management:** 274 total APIs.
- **Credit Cards:** 275 total APIs.
- **Savings:** 275 total APIs.
- **Investments:** 281 total APIs.

Each set includes approximately 20 core business APIs, 45 financial distractors, and 198 non-financial distractors.

# B Annotator Guidelines

**Overview**

This document provides guidelines for annotators to validate TELLER's quality across three dimensions: **User Data Validity**, **SOP Logical Consistency**, and **API Dependency Correctness**. Each task requires **15-20 minutes**.

## B.1 Annotation Dimensions

**1. User Data Validity**

**Objective:** Verify synthetic customer data reflects realistic banking scenarios without contradictions.

**Validation Criteria:**

- **Profile Realism:** Age, employment, income align with demographics
  - Retired: $\geq$60 yrs, income $\leq$\$15K/mo
  - Self-employed: variable income
  - Students: lower credit limits
- **Transaction Consistency:** Amounts match purpose
  - Home: \$200K-\$800K
  - Auto: \$15K-\$60K
  - Personal: \$1K-\$50K
- **Temporal Logic:** Chronological order
- **Geographic Coherence:** Location consistency

**Procedure:**

1. Review `<Existing_fields>`
2. Flag contradictions
3. Mark `Valid/Invalid`

**2. SOP Logical Consistency**

**Objective:** Ensure SOP accurately reflects business context without procedural errors.

**Validation Criteria:**

- **Completeness:** All critical steps included
- **Decision Thresholds:** Quantitative criteria defined
  - Credit score $\geq$650
  - Debt-to-income $\leq$43%
  - Escalation if loan $>$\$100K
- **Regulatory Compliance:** KYC, AML checks
- **Exception Handling:** Edge cases addressed

**Procedure:**

1. Read entire `<sop_contents>`
2. Verify "Main Procedure" logic
3. Check decision criteria clarity
4. Mark `Consistent/Inconsistent`

**3. API Dependency Correctness**

**Objective:** Validate API dependency graph accurately represents data flow and execution order.

**Validation Criteria:**

- **Dependency Accuracy:** B depends on A iff B's input from A's output
- **No Circular Dependencies:** Must be DAG
- **Parameter Mapping:** `<Field_to_be_obtained>` traceable
- **Execution Order:** Respects SOP sequence

**Procedure:**

1. Review `<api_graph>`
2. Verify input parameter sources
3. Check for circular dependencies
4. Mark `Correct/Incorrect`

## B.2 Annotation Interface

| Field | Options |
|---|---|
| User Data Validity | **Valid** / **Invalid** |
| SOP Consistency | **Consistent** / **Inconsistent** |
| API Dependency | **Correct** / **Incorrect** |
| Comments | Free text (if negative) |

# C Detailed Analysis of Precision-Recall Asymmetry

## C.1 Factors Contributing to Higher Precision than Recall

Our analysis identifies three primary factors explaining why Stage 2 tool invocation precision consistently exceeds recall across all tested models:

**Stage 1 Filtering Effect.** The tool set provided in Stage 2 has already undergone filtering in Stage 1, removing a large number of irrelevant APIs. This pre-filtering reduces the denominator in precision calculations (fewer false positives) while maintaining the numerator (true positives), naturally elevating precision scores.

**Hallucination Avoidance Mechanisms.** Most current large language models incorporate safety mechanisms to prevent hallucinations. When faced with uncertainty about a tool's relevance to the task, models tend to refrain from invoking it. This conservative strategy increases precision (by avoiding false positives) but decreases recall (by missing true positives that the model is uncertain about).

**Execution Chain Breakage.** Complex banking workflows require multi-step tool execution with strict dependency ordering. Errors in early steps

can break the invocation chain, preventing subsequent critical tools from being called. For example, if a credit score retrieval fails, downstream tools requiring that score cannot be invoked, reducing recall while maintaining precision for successfully executed tools.

### C.2 Quantitative Evidence

Table 6 presents a detailed breakdown of precision-recall gaps across model tiers:

| Model Tier | Avg. P | Avg. R | Gap |
|---|---|---|---|
| Tier 1 (Proprietary) | 95.8% | 64.1% | 31.7% |
| Tier 2 (Large Open) | 73.9% | 51.9% | 22.0% |
| Tier 3 (Small Open) | 88.6% | 35.7% | 52.9% |

Table 6: Precision-recall gaps by model tier in Stage 2 tool invocation. P: Precision; R: Recall.

Notably, Tier 3 models exhibit the largest gap (52.9%), suggesting that weaker models adopt more conservative invocation strategies to maintain precision at the cost of recall.

### C.3 Implications for Banking Workflow Automation

The precision-recall asymmetry has practical implications:

- **High Precision**: Models rarely invoke incorrect tools, reducing risk of erroneous transactions.
- **Low Recall**: Many necessary tools are missed, leading to incomplete workflow execution and requiring human intervention.

Future work should focus on improving recall through better dependency understanding and error recovery mechanisms, rather than solely optimizing precision.

## D Error Analysis

To understand why models fail in banking workflow execution, we conduct a comprehensive error analysis by categorizing tool invocation errors into three distinct types. As illustrated in Figure 8, these errors reveal fundamental limitations in current LLMs' ability to handle complex, multi-step tool invocation scenarios.

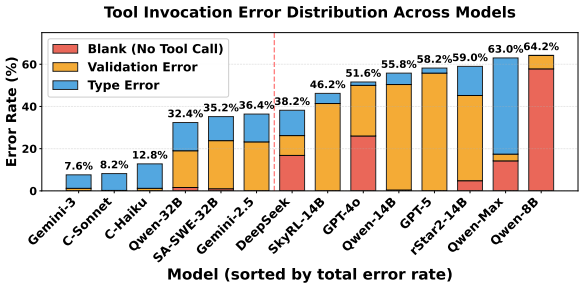


Figure 8: Tool invocation error distribution across models. Proprietary models show lower total errors; agent models exhibit low Blank but high Validation errors, indicating a gap between invocation intent and parameter construction.

### D.1 Error Taxonomy

We identify three primary error categories:

- **Blank (No Tool Call):** The model terminates execution without invoking any tool, despite the task requiring tool use. This indicates a failure in recognizing the need for external tool invocation or an inability to construct valid tool calls under complexity.
- **Validation Error:** The model attempts to invoke a tool but provides parameters that fail schema validation (e.g., missing required fields, incorrect field names, or values outside acceptable ranges). This reflects a gap between tool invocation intent and precise parameter construction.
- **Type Error:** The model invokes a tool with parameters that pass schema validation but cause runtime type mismatches during execution (e.g., passing a string where an integer is expected, or providing a malformed JSON structure). This suggests insufficient understanding of API contracts and data type constraints.

### D.2 Model Capability and Error Patterns

Different model families exhibit distinct error tendencies, revealing the relationship between model scale, architecture, and tool invocation robustness.

**Small-Scale Open-Source Models.** Small models (e.g., Qwen3-8B) demonstrate a high propensity for Blank errors (57.8%), with Qwen3-8B showing the highest total error rate (64.2%). This suggests that smaller models struggle to maintain task coherence in long-horizon workflows, often abandoning tool invocation when faced with complex dependency chains or ambiguous instructions. The stark contrast between Qwen3-8B and Qwen3-14B (57.8% vs. 0.4% Blank errors) indicates that even

modest increases in model scale can significantly improve tool invocation intent recognition.

**Large-Scale Proprietary Models.** Proprietary models (Gemini-3-Pro, Claude-Sonnet-4.5) achieve substantially lower total error rates (7.6%, 8.2% respectively) with near-zero Blank errors. However, these models still exhibit non-negligible Validation and Type errors, indicating that even state-of-the-art models face challenges in precise parameter construction and API contract adherence.

Interestingly, GPT-4o shows a significantly higher Blank error rate (26.0%) compared to other proprietary models, suggesting potential issues with instruction following or tool invocation triggering in certain banking scenarios. This anomaly warrants further investigation into model-specific biases or training data limitations.

### D.3 Agent Model Specialization

Agent models fine-tuned specifically for tool use (rStar2-Agent-14B, SkyRL-Agent-14B, SA-SWE-32B) display a distinct error pattern that reveals both the strengths and limitations of current reinforcement learning (RL) approaches to tool invocation.

**Low Blank, High Validation Error Pattern.** Agent models achieve extremely low Blank error rates (4.8%, 0%, and 1.0% respectively) but suffer from significantly higher Validation errors (40.4%, 41.4%, and 22.8%). This pattern indicates that:

1. **Strong Tool Invocation Awareness:** RL-based training effectively teaches models *when* to invoke tools, as evidenced by near-zero Blank errors. These models rarely abandon tool invocation, even in complex scenarios.
2. **Weak Parameter Construction:** However, these models struggle to construct parameters that comply with API schemas. The high Validation error rates suggest that current RL methods excel at action selection (i.e., deciding to invoke a tool) but fail to generalize to precise parameter value generation, especially for domain-specific constraints (e.g., credit score thresholds, loan amount ranges).
3. **Overfitting to Training Environments:** Agent models may overfit to the specific API schemas and parameter distributions seen during training, leading to poor generalization when encountering novel banking scenarios or slightly modified API contracts in TELLER.

**Implications for RL-Based Tool Use.** The agent model error pattern highlights a fundamental limitation of current RL approaches: while they can learn to recognize tool invocation opportunities through reward shaping, they struggle with the *compositional reasoning* required to construct valid, context-aware parameters. This suggests that future work should focus on:

- **Hybrid Approaches:** Combining RL for action selection with symbolic reasoning or constraint satisfaction for parameter generation.
- **Schema-Aware Training:** Incorporating explicit API schema information into the training process to improve parameter validation success rates.
- **Multi-Stage Verification:** Introducing intermediate verification steps that check parameter validity before tool invocation, reducing runtime Validation errors.

### D.4 Error Distribution Insights

**Proprietary vs. Open-Source Divide.** As shown in Figure 8, a clear performance gap exists between proprietary and open-source models. The average total error rate for proprietary models (excluding GPT-4o's anomaly) is 16.2%, compared to 48.7% for open-source models. This gap is primarily driven by differences in Blank error rates, suggesting that proprietary models benefit from larger-scale pretraining and more sophisticated instruction-following capabilities.

**Type Error Prevalence in Mid-Scale Models.** Mid-scale models (Qwen3-32B, DeepSeek-V3.2, Qwen3-Max) exhibit the highest Type error rates (13.4%, 12.0%, and 45.6% respectively), indicating that these models can construct syntactically valid parameters but fail to ensure semantic correctness (e.g., passing a string representation of a number instead of an integer). This suggests that mid-scale models lack the deep understanding of API contracts required for robust tool invocation.

**Validation Error as a Proxy for Schema Understanding.** Validation errors are most prevalent in agent models and mid-scale open-source models, indicating a gap in schema understanding. Models that exhibit high Validation error rates (e.g., SkyRL-Agent-14B at 41.4%) struggle to map natural language instructions to structured API parameters, highlighting the need for better schema-grounded training data and evaluation benchmarks.

### D.5 Case Study: GPT-5 vs. GPT-4o

An interesting comparison emerges between GPT-5 and GPT-4o. Despite being from the same model family, GPT-5 achieves a significantly lower total error rate (12.8% vs. 46.2%), with the primary difference being Blank errors (0% vs. 26.0%). This suggests that GPT-5's improvements over GPT-4o are largely driven by better instruction following and tool invocation triggering, rather than parameter construction accuracy (both models have similar Validation and Type error rates). This finding underscores the importance of robust instruction tuning for tool use scenarios.

### D.6 Implications for Future Work

The error analysis reveals several key directions for improving LLM-based tool invocation in complex workflows:

1. **Reducing Blank Errors:** Small models require better instruction tuning and prompting strategies to maintain tool invocation intent in long-horizon tasks.
2. **Improving Parameter Construction:** Agent models need hybrid approaches that combine RL with symbolic reasoning or schema-aware generation to reduce Validation errors.
3. **Enhancing Type Safety:** Mid-scale models require stronger type checking and API contract understanding to reduce Type errors.
4. **Benchmarking Schema Generalization:** Future benchmarks should explicitly test models' ability to generalize to novel API schemas and parameter constraints, as current RL-based approaches show signs of overfitting.

By addressing these challenges, we can move closer to robust, production-ready LLM agents capable of handling complex, multi-step workflows in safety-critical domains like banking.

## E Hardware Resources

All experiments were conducted on a local Slurm-based high-performance computing cluster. Each compute node was equipped with the following specifications:

- **CPU:** 48-core processors
- **GPU:** Eight NVIDIA L20 GPUs, each with 48 GB of VRAM
- **System RAM:** 925,600 MB ( 904 GB)
- **Interconnect:** High-speed InfiniBand for multi-node communication

### E.1 Model Deployment

We deployed all models with 32 billion parameters or fewer on this infrastructure. Specifically:

- **Small Models:** Qwen3-8B, Qwen3-14B, and other models in this range were deployed on a single NVIDIA L20 GPU (48 GB VRAM), utilizing FP16 precision for efficient inference.
- **Mid-Scale Models (14B-32B parameters):** Qwen3-32B, SA-SWE-32B, and similar models required multi-GPU deployment using tensor parallelism across 2-4 L20 GPUs, depending on the model's memory footprint and sequence length requirements.

## F Prompt Demonstration

To ensure transparency and reproducibility, we provide detailed prompts used in each stage of the TELLER benchmark construction pipeline (Section F) and the ReAct-style evaluation prompt (Section F).

**Prompt Design Principles.** Our prompts follow three core principles: (1) **explicit structure enforcement** via XML/YAML/JSON format specifications, (2) **domain-specific constraints** embedding banking regulations and realistic data ranges, and (3) **iterative refinement** through pre-generation analysis steps (e.g., `<sop_analysis>`, `<dependency_analysis>`) that force models to reason before generating outputs.

**Prompt Chaining.** Prompts form a dependency chain where each stage consumes outputs from previous stages. For example, data templates (Section F) inform SOP generation (Section F), which guides API dependency graph construction (Section F). This ensures semantic consistency—SOPs reference schema-defined fields, APIs align with SOP steps, and synthetic data conforms to constraints. Human-in-the-loop validation at key stages prevents error propagation (Section B).

**Evaluation Prompt.** The ReAct-style evaluation prompt (Section F) instructs models to execute workflows by interleaving reasoning and tool invocation. Models must: (1) analyze the SOP (`<sop_analysis>`), (2) invoke tools with valid parameters (JSON format per LangChain convention), (3) record progress (`<step_progress>`), and (4) provide a final summary (`<final_response>`). This mirrors real-world agentic workflows in safety-critical domains.

| Section | Content |
|---|---|
| **System Role** | |
| You are an industry scenario analysis expert. Using the **Banking Scenario** as the background, output a complete, structured description of multiple business activities within this scenario. Please strictly follow the format below for your output: | |
| **1. Business Overview** | - Scenario Name:<br>- Industry:<br>- Primary Purpose / Value: |
| **2. Business Types** | - Sub-business A: Brief Description<br>- Sub-business B: Brief Description<br>- ... |
| **3. Process Description** | **Sub-business A Process:**<br>1) Step 1: Description<br>2) Step 2: Description<br>- . . .<br>**Sub-business B Process:**<br>1) Step 1: Description<br>2) Step 2: Description<br>- . . . |
| **4. Key Roles** | - Role 1 (e.g., Customer / Bank Teller / System Administrator): Responsibility<br>- Role 2: Responsibility Description<br>- . . . |
| **5. System & Tool Support** | - System / Tool A: Function Description<br>- System / Tool B: Function Description<br>- . . . |
| **6. Risks & Controls** | - Risk Point 1: Description / Control Measures<br>- Risk Point 2: Description / Control Measures<br>- . . . |
| **7. Metrics** | - Metric 1 (e.g., Conversion Rate): Description of how to measure<br>- Metric 2: Description of how to measure<br>- . . . |
| **8. Best Practices** | - Suggestion 1: Specific Description<br>- Suggestion 2: Specific Description<br>- . . . |
| | Please **complete filling in according to the above format**. |

Table 7: Prompt for Business Context Generation

| Section | Content |
|---|---|
| **System Role** | |
| You are an expert AI assistant specialized in generating synthetic dataset schemas based on Standard Operating Procedures (SOPs). You will use the provided SOP text to generate a dataset schema description. **The number of fields in the schema must be designed to be within 50.** | |
| **Input Parameters** | - `sop_title`: Title of the SOP<br>- `business_context`: Business contextual instructions |
| **Instructions** | First, please carefully read the following SOP title:<br><sop_title><br>{{sop_title}}<br></sop_title><br><br>Here are the business contextual instructions you must follow during the SOP generation process:<br><business_context><br>{{business_context}}<br></business_context> |
| **Example Schema** | Below is an example of a video information schema. You must mimic the following format for output, but **if the actual task is not related to video, do not output these template information**:<br><br><schema><br><Existing_fields><br>video_id - string, unique identifier (e.g., "vid_00123")<br>video_path - string, file path (e.g., "/data/videos/...")<br>upload_timestamp - datetime (e.g., "2025-04-18T15:32:10Z")<br>uploader_id - string, anonymized user ID (e.g., "user_487")<br>video_language - string (e.g., "en", "es")<br>region - string (e.g., "US", "IN")<br>metadata_tags - list[string] (e.g., ["protest", "crowd"])<br>duration_seconds - integer (e.g., 312)<br></Existing_fields><br><Field_to_be_obtained><br>format_validated - boolean (e.g., True)<br></Field_to_be_obtained><br></schema> |
| **Task** | Now, please combine the above SOP title and business context to generate a dataset schema within the <schema></schema> tags, following the specified format. |
| **Requirements** | 1. Define the dataset schema fields, including for each field:<br>- Name<br>- Type (e.g., string, integer, float, boolean, datetime, enum, list, etc.)<br>- Description (what this field represents)<br>- Example value<br>2. Where applicable, include data value ranges and options. Appropriately include independent fields such as funding amounts (300,000), time, etc.<br>3. Existing data before approval must be wrapped inside <Existing_fields></Existing_fields>, while fields that need to be obtained through processes like calculation, approval, etc., such as credit scores calculated, approval results (whether passed), must be wrapped inside <Field_to_be_obtained></Field_to_be_obtained>.<br>4. If a field's value format is JSON, all sub-fields within it must be determined. |

Table 8: Prompt for Data Template Generation

| Section | Content |
|---|---|
| **System Role** | |
| You are a professional AI system specialized in designing advanced Standard Operating Procedures (SOPs) for complex, technically demanding, and regulated environments. Your task is to generate an SOP rich in domain-specific terminology, featuring intricate logic, complex structures, and requiring substantial expertise to fully comprehend. **Try not to make the SOP content too long.** | |
| **Input Parameters** | `sop_title`: Title of the SOP to create<br>`business_context`: Business context for SOP generation<br>`sample_schema`: Data schema with existing and to-be-obtained fields |
| **SOP Title** | Below is the title of the SOP you need to create:<br>`<sop_title>{{sop_title}}</sop_title>` |
| **Business Context** | Here is the business context you must refer to during SOP generation:<br>`<business_context>{{business_context}}</business_context>` |
| **Data Schema** | Below is a data schema. Fields in `<Existing_fields>` are present in existing data, while fields in `<Field_to_be_obtained>` should be gradually processed and obtained through the SOP's execution:<br>`<sample_schema>{{sample_schema}}</sample_schema>` |
| **Pre-Generation Analysis (within `<sop_analysis>` tags)** | |
| Before generating the SOP, thoroughly analyze the title, business context, and sample schema. Follow these steps:<br>1. Deconstruct the title and business context<br>2. Identify the specific industry or domain<br>3. List potential regulatory requirements<br>4. Enumerate key technical terms<br>5. Conceptualize complex technical details<br>6. Outline interdependencies between sections<br>7. Consider potential complications, edge cases, and rare scenarios<br>8. Identify specific compliance issues and how to address them<br>Use industry jargon and complex terminology throughout planning. | |
| **SOP Structure (within `<SOP></SOP>` tags)** | |
| **1. Purpose** | Clear, technical articulation of the SOP's objective |
| **2. Scope** | Define the precise scope of the SOP and to whom it applies |
| **3. Definitions** | List and define relevant terms, ensuring they are technically robust and domain-appropriate |
| **4. Input** | Specify the inputs, materials, or information required to initiate the process |
| **5. Main Procedure** | 5.1 [Step 1]<br>5.1.1 [Detailed description of substep]<br>5.1.2 [Detailed description of substep]<br>5.2 [Step 2]<br>5.2.1 [Detailed description of substep]<br>5.2.2 [Detailed description of substep]<br>[Continue adding all necessary steps and substeps] |
| **6. Output** | Describe the expected results or outputs upon process completion |
| **SOP Generation Requirements** | |
| 1. Employ advanced industry terminology and lengthy technical sentences<br>2. Ensure the language is complex and technically dense<br>3. Use cross-references between sections<br>4. Include complex, domain-specific terminology in Definitions<br>5. Utilize information from the business context<br>6. Avoid bullet points; use full paragraphs with exhaustive elaboration<br>7. Explicitly define quantifiable thresholds and decision criteria<br>8. The SOP should be self-contained<br>9. Leverage schema information for thresholds, logic, and decision pathways. Clarify how `<Existing_fields>` are used and how `<Field_to_be_obtained>` are derived<br>10. Use `<sop_analysis>` tag to increase complexity<br>11. Generate the entire SOP at once without pausing | |
| **Output Format** | Output the complete SOP at once within `<SOP></SOP>` tags. All steps must be detailed and clear while maintaining complexity and technical depth. |

Table 9: Prompt for SOP Generation

| Section | Content |
|---|---|
| **System Role** | |
| You are a professional AI system specialized in analyzing complex business processes and deducing API dependency relationships. Your task is to generate a comprehensive directed graph of API dependencies based on the SOP document, data schema, and business context. This graph must accurately reflect the data flows and prerequisite dependencies between APIs. | |
| **Input Parameters** | `sop_title`: Title of the business process<br>`business_context`: Business context for dependency graph generation<br>`sample_schema`: System's data schema definition<br>`sop_file_contents`: Generated SOP document with detailed process steps |
| **SOP Title** | Below is the title of the business process you need to analyze:<br>`<sop_title>{{sop_title}}</sop_title>` |
| **Business Context** | Here is the business context you must reference during dependency graph generation:<br>`<business_context>{{business_context}}</business_context>` |
| **Data Schema** | Below is the system's data schema definition. You should base your derivation of API input/output parameters on this:<br>`<sample_schema>{{sample_schema}}</sample_schema>` |
| **SOP Document** | Below is the generated SOP document containing detailed business process steps. You should analyze API execution order and dependencies based on this:<br>`<sop_file_contents>{{sop_file_contents}}</sop_file_contents>` |
| **Pre-Generation Analysis (within `<dependency_analysis>` tags)** | |
| **1. Business Process Deconstruction** | - Map each operational step in the SOP to potential API invocations<br>- Identify data transfer relationships between steps<br>- Determine the role and positioning of each API |
| **2. Data Flow Analysis** | - Analyze association relationships between entities based on schema<br>- Trace the source of each API's input parameters (user input, other APIs, system state)<br>- Identify usage scenarios and consumers for API output data |
| **3. Dependency Relationship Deduction** | - Analyze preconditions for API execution<br>- Identify direct dependencies (API B's input from API A's output)<br>- Identify indirect dependencies (API C requires state changes from API B)<br>- Determine dependency type (data, state, temporal) |
| **4. Complexity Assessment** | - Assess technical complexity of each API<br>- Identify potential cyclic dependency risks<br>- Analyze length and complexity of dependency chains<br>- Consider exception cases and error handling paths |
| **5. Verification & Optimization** | - Verify rationality and necessity of dependency relationships<br>- Identify APIs that can execute in parallel<br>- Propose suggestions for optimizing dependency relationships |
| **Output Format (within `<api_dependency_graph>` tags)** | |
| **YAML Structure:** | |

```yaml
apis:
- api_name: "string"   # Unique identifier
  description: "string"   # Functional description
  purpose: "string"   # Purpose within business process
  input_parameters:
    - name: "string"   # Parameter name
      data_type: "string"   # Data type
      source: "string"   # Source: user_input, other_api, system_state
      required: boolean   # Whether required
  output_parameters:
    - name: "string"   # Output parameter name
      data_type: "string"   # Data type
      destination: "string"   # Destination: other_api, storage, ui_display
  direct_dependencies:
    - dependent_api: "string"   # Name of dependent API
      dependency_type: "string"   # Type: data_dependency, state_dependency, temporal_dependency
      dependency_reason: "string"   # Description of reason
  complexity_level: "string"   # Level: low, medium, high
  business_criticality: "string"   # Criticality: low, medium, high
```

| | |
|---|---|
| **Generation Requirements** | |
| **1. Accuracy** | - Each dependency must have clear evidence of parameter passing or state dependency<br>- Dependencies must be based on SOP operational sequence and data flows<br>- Input/output parameters must be strictly consistent with schema; **no undefined attributes** |
| **2. Completeness** | - Cover all implicit API operations within the SOP<br>- Include all necessary parameter mapping relationships<br>- Identify all direct dependency relationships without omitting key dependency chains |
| **3. Technical Depth** | - Use precise technical terminology to describe API functionality<br>- Provide detailed explanation of technical basis for each dependency<br>- Analyze strength and necessity of dependency relationships |
| **4. Business Consistency** | - API design must support SOP's business objectives<br>- Dependency relationships must align with business process logic<br>- Consider business rules and constraints |
| **5. Implementation Feasibility** | - Dependency graph must support subsequent API development and testing<br>- Provide sufficient technical details for development team<br>- Consider performance, security, and maintainability factors |
| **Output Instructions** | After completing detailed analysis within `<dependency_analysis>` tags, output the complete API dependency directed graph at once within `<api_dependency_graph>` tags. **Remember:** The accuracy and completeness of dependency relationships are key to the overall system's success. You must ensure that the dependency relationships for each API are fully justified and described in detail. |

Table 10: Prompt for API Dependency Graph Generation

| Section | Content |
|---|---|
| **System Role** | |
| You are an expert AI assistant specializing in generating high-quality synthetic datasets based on detailed Standard Operating Procedures (SOPs). Your task is to extract operational logic and decision criteria from SOPs and synthesize structured data that reflects all possible scenarios, perfectly matching the field structure of the provided schema. | |
| **Input Parameters** | `sop_title`: Title of the SOP<br>`sop_file_contents`: SOP contents<br>`business_context`: Business context<br>`sample_schema`: Data schema<br>`direct_graph`: API dependency directed graph<br>`n_samples`: Number of samples to generate<br>`sop_data_generation_guidelines`: Additional requirements |
| **Input Format** | `<sop_title>{{sop_title}}</sop_title>`<br>`<sop_contents>{{sop_file_contents}}</sop_contents>`<br>`<business_context>{{business_context}}</business_context>`<br>`<schema>{{sample_schema}}</schema>`<br>`<direct_graph>{{direct_graph}}</direct_graph>` |
| **Critical Format Requirements** | |
| **Note:** The format of each data entry must be exactly the same. Each entry should be wrapped in `[]`, with each field value conforming to the schema structure, wrapped in ", and separated by English commas. **Input and output fields must only be those present in the schema**; only synthesize field values within `<Existing_fields>` tags. For fields in `<Field_to_be_obtained>`, only write the key and leave the value empty. | |
| **Task Requirements** | |
| **1. Understanding & Analysis** | Carefully analyze the SOP to identify all decision points, validation steps, escalation conditions, and classification rules. Based on these, determine the fields that must appear in the dataset and their possible values. |
| **2. Schema Compliance** | Ensure the dataset complies with (and expands upon) the fields given in `<schema>`. If no schema is provided, infer a reasonable and complete schema from the SOP content. |
| **3. Case Diversity** | - Include both valid and invalid examples<br>- Cover edge conditions, extreme values, and policy violations<br>- Ensure that every step of logic in the SOP is reflected in some rows of the data |
| **4. Data Format** | - Output data **only** within `<data></data>` tags<br>- Do **not** add extra commentary or instructions outside the `<data>` block |
| **5. Validation Requirements** | - All rows must be non-empty<br>- All columns must be complete and values correctly formatted<br>- No unclosed strings, no internal line breaks within strings<br>- Consistency between field values (e.g., escalation field set to True only when conditions are met) |
| **6. Sample Count** | Generate at least `{{n_samples}}` rows of valid records, covering the full range of conditions defined by the SOP. |
| **7. Generation Mode** | Generate the complete data in one go, without interruption, confirmation, or interaction. |
| **8. Output Purity** | Do not add explanatory text, placeholders, or prompts to continue generation. Output **only** the complete data within the `<data>` block. |
| **9. Format Integrity** | Ensure there are no format errors, incomplete nodes, or broken strings. |
| **10. Completeness** | No empty columns or rows are allowed. Ensure all strings are fully closed without line breaks. |
| **11. Boolean Format** | For Boolean fields, use `True` or `False` to allow parsing with `ast`. |
| **12. Bracket Closure** | Ensure all square brackets are properly closed. |
| **Additional Requirements** | `<additional_requirements>`<br>`{{sop_data_generation_guidelines}}`<br>`</additional_requirements>` |
| **Output Format** | |
| You must output the data strictly in the following format, where `'field'` is the field name (which must already exist in the schema), and `'value'` is the field value you synthesize. For fields wrapped in `<Field_to_be_obtained>`, only write the key and leave the value empty.<br>`<data>`<br>`['field_1:value_1_1', 'field_2:value_1_2', ..., 'field_N:value_1_N'],`<br>`...`<br>`['field_1:value_k_1', 'field_2:value_k_2', ..., 'field_N:value_k_N']`<br>`</data>` | |

Table 11: Prompt for Synthetic Data Generation

| Section | Content |
|---|---|
| **System Role** | |
| You are an expert AI assistant specializing in API documentation writing and generation. Your task is to create example API calls based on a Standard Operating Procedure (SOP) document. The goal is to generate accurate, comprehensive API documentation that captures all necessary steps and dependencies within the SOP and its dependency graph. | |
| **Input Parameters** | `sop_file_contents`: SOP content<br>`business_context`: Business context<br>`sample_data`: Sample data<br>`api_graph`: API dependency graph |
| **Input Format** | `<sop_contents>{{sop_file_contents}}</sop_contents>`<br>`<business_context>{{business_context}}</business_context>`<br>`<sample_data>{{sample_data}}</sample_data>`<br>`<api_graph>{{api_graph}}</api_graph>` |
| **SOP Breakdown Analysis (within `<sop_breakdown>` tags)** | |
| Conduct an in-depth analysis of the SOP dependency graph and business context, focusing on the "Main Procedure" section and its subsections and workflow:<br>1. List all independent tasks or operations within the "Main Procedure" section<br>2. For each task, identify any external dependencies required for execution<br>3. Consider and document potential edge cases for each task<br>4. Identify and list key API parameters for each task from the SOP and API dependency graph. The APIs and parameters must already exist in the dependency graph<br>5. Based on this analysis, plan your API generation approach<br>6. For each API input parameter, specify its source (SOP sample data input or responses from other APIs) based on the relationships defined in the function dependency graph<br>7. Sequence the APIs based on their dependencies in the API dependency graph and the order they should be executed<br>8. For each potential API, list possible error scenarios and their handling methods<br>9. Ensure the APIs can process each piece of sample data | |
| **API Generation Requirements** | |
| **Guidelines** | 1. Create a separate API for each independent task or operation identified in the program's dependency graph<br>2. Ensure each tool or API is as granular or atomic as possible<br>3. Do not create duplicate tools or APIs for the same task<br>4. Capture all possible external dependencies required for the successful execution of the SOP<br>5. Ensure that each API's input request parameters originate either from the SOP input (sample data) or from response parameters (output) of other API calls<br>6. Ensure the implemented APIs are complete and fully usable |
| **API Details** | For each API you generate, provide the following details:<br>1. Name of the API<br>2. Description of the API<br>3. Request Body (including all necessary parameters that exist in the example data within `<sample_data>` tags)<br>4. Response Body (including corresponding status columns that exist in the example data within `<sample_data>` tags)<br>5. Do not add parameters in the request and response bodies that are not present in the `<sample_data>` tags<br>6. Dependencies (list dependent APIs here if this API relies on outputs from others)<br>7. Potential error scenarios and suggested error handling methods |
| **Output Format** | |
| Wrap each tool/API within `<API></API>` tags and place the entire collection of tools/APIs within `<TOOLS></TOOLS>` tags.<br>**Example Structure:**<br>`<TOOLS>`<br>`  <API>`<br>`  Name: [API Name]`<br>`  Description: [A brief description of the API's functionality]`<br>`  Request Body:`<br>`  {`<br>`  [JSON structure of request parameters]`<br>`  }`<br>`  Response:`<br>`  {`<br>`  [JSON structure of response data]`<br>`  }`<br>`  Dependencies: [List of dependent APIs, if any]`<br>`  Error Scenarios:`<br>`  - [Potential Error 1]: [Suggested handling method]`<br>`  - [Potential Error 2]: [Suggested handling method]`<br>`  </API>`<br>`</TOOLS>`<br>**Note:** The `<sop_breakdown>` and `<TOOLS>` sections can be quite lengthy as they need to thoroughly cover all aspects of the SOP and the generated APIs. | |

Table 12: Prompt for API Documentation Generation

| Section | Content |
|---|---|
| **System Role** | |
| You are an expert AI assistant specializing in generating JSON schemas for API tools based on provided metadata. This tool is one of many required to execute a Standard Operating Procedure (SOP). Your goal is to create an accurate, detailed schema for this tool that aligns with the SOP and meets specified requirements. | |
| **Input Parameters** | `sop_file_contents`: SOP content<br>`business_context`: Business context<br>`each_api_metadata`: API metadata<br>`sample_data`: Sample data |
| **Input Format** | `<sop_contents>{{sop_file_contents}}</sop_contents>`<br>`<business_context>{{business_context}}</business_context>`<br>`<sample_data>{{sample_data}}</sample_data>`<br>`<api_metadata>{{each_api_metadata}}</api_metadata>` |
| **Task Steps** | |
| 1. Read and analyze the API metadata carefully<br>2. Generate the following for the API tool:<br>a. **name**: A concise, descriptive name for the tool<br>b. **description**: A clear statement of the tool's purpose and function<br>c. **inputSchema**: Specify the JSON schema format for the tool's "input", including the "type", "properties", and "required" fields, all under the "json" key<br>3. Ensure the schema aligns with the overall SOP direction and any specified constraints | |
| **Pre-Generation Analysis (within `<sop_tool_analysis>` tags)** | |
| Before generating the final JSON schema, place your analysis within the `<sop_tool_analysis>` tags:<br>1. Reference relevant sections of the SOP related to the tool's purpose<br>2. List potential input parameters based on the SOP and metadata<br>3. Consider any constraints or special considerations mentioned in the SOP<br>4. Outline how the tool aligns with the overall SOP direction<br>5. Identify potential edge cases or limitations of the tool<br>6. Describe how this tool fits into the larger workflow outlined in the SOP | |
| **Output Format (within `<SCHEMA></SCHEMA>` tags)** | |
| Generate the complete JSON schema for the API tool. The schema should be formatted as an object containing a "toolSpec" property, which includes "name", "description", and "inputSchema":<br>`{`<br>`  "toolSpec": {`<br>`  "name": "example_tool_name",`<br>`  "description": "A clear description of the tool's purpose and function.",`<br>`  "inputSchema": {`<br>`  "json": {`<br>`  "type": "object",`<br>`  "properties": {`<br>`  "parameter_name": {`<br>`  "type": "string",`<br>`  "description": "A detailed description of this parameter."`<br>`  }`<br>`  },`<br>`  "required": ["parameter_name"]`<br>`  }`<br>`  }`<br>`  },`<br>`  "toolSpec": {...},`<br>`  ...`<br>`}` | |
| **Important Notes** | |
| - Each `toolSpec` field represents an API tool. You must convert the APIs defined between each pair of `<API></API>` tags into JSON-formatted tools. The inputs and outputs of these tools have dependencies<br>- Ensure all required fields are listed in the "required" array of the input schema<br>- Provide detailed descriptions for each property in the input schema<br>- Use appropriate data types (e.g., string, integer) for each property<br>- Maintain consistency in formatting and structure<br>- Place the final JSON schema within the `<SCHEMA></SCHEMA>` tags | |

Table 13: Prompt for API JSON Schema Generation

| Section | Content |
|---|---|
| **System Role** | |
| You are an AI programming expert proficient in writing Python code using popular libraries. You will receive metadata for an API and a sample dataset. Your task is to write a Python function that simulates this API by strictly adhering to the given API template. In the code you generate, maintain the data within `<sample_data></sample_data>` in its original format as readable and writable data. Fields that already have values in the data are user's original data; empty fields are data that need to be obtained through API processing. Once obtained via the API, write the data back to the corresponding entry you maintain, completing the update. Finally, write the class definition and tools list encapsulation for the API functions at the end. | |
| **Input Parameters** | `api`: API metadata<br>`sample_data`: Sample dataset<br>`api_example`: API example<br>`coding_example`: Coding example<br>`data_example`: Data example |
| **Input Format** | `<api>{{api}}</api>`<br>`<sample_data>{{sample_data}}</sample_data>` |
| **Code Generation Requirements** | |
| **Requirement 1:** After carefully studying the example provided within the `<example>` tags, generate the relevant Python code and place it within `<code></code>` tags<br>**Requirement 2:** You must write test cases as demonstrated in the example<br>**Requirement 3:** Import all necessary Python libraries. Use descriptive and meaningful variable names<br>**Requirement 4:** Function names must exactly match the names specified in the API specification<br>**Requirement 5:** Strictly follow the API definition—fields in the request must map to function parameters, and fields in the response must be returned<br>**Requirement 6:** Use only columns that exist in the dataset<br>**Requirement 7:** Load the dataset into memory before accessing the required fields<br>**Requirement 8:** Implement the `process_tool_call()` function accurately, following the method demonstrated in the `<example>` tags<br>**Requirement 9:** Generate the code in one go, without pausing, requesting confirmation, or interacting. Do not ask to continue | |
| **Important Constraints** | |
| If there is no sample data or code, you must generate logically rigorous code according to the given requirements. The code logic must fully comply with the template without any innovation. Furthermore, any numerical calculations or logic in the code must be deterministic, producing the same result every time. **The use of random numbers or similar methods is strictly prohibited.** | |
| **Output Structure** | |
| **1. API Function** | Within `<code></code>` tags, implement the API function following the template. Example:<br>`def ApplicationLogicValidation(application_id: str, ...):`<br>`  """API for validating logical consistency of application information"""`<br>`  validation_passed = True`<br>`  validation_issues = []`<br>`  ...`<br>`  return result` |
| **2. Class Definition** | Within `<class></class>` tags, define the tool class using Pydantic `BaseModel`:<br>`class Tool_ApplicationLogicValidation(BaseModel):`<br>`  application_id: str = Field(..., description="")`<br>`  material_completeness_score: float = Field(..., description="")`<br>`  ...` |
| **3. Tools List** | Within `<tools></tools>` tags, encapsulate the tools list:<br>`tools = [`<br>`  StructuredTool.from_function(`<br>`  func=ApplicationLogicValidation,`<br>`  name="ApplicationLogicValidation",`<br>`  description="",`<br>`  args_schema=Tool_ApplicationLogicValidation`<br>`  ),`<br>`  ...`<br>`]` |
| **Task** | |
| Based on the example above, write code for the following API and dataset to successfully complete the task:<br>`<api>{{api}}</api>`<br>`<sample_data>{{sample_data}}</sample_data>`<br>After the API code is generated, output the class definition and tools list encapsulation for each API, mimicking the format shown above. | |

Table 14: Prompt for Python API Implementation Generation

| Section | Content |
|---|---|
| **System Role** | |
| You are an AI assistant responsible for handling user requests according to a specific Standard Operating Procedure (SOP). Your primary task is to strictly adhere to the SOP to ensure consistency and accuracy in your responses. | |
| **Input Parameters** | `SOP`: Standard Operating Procedure<br>`USER_REQUEST`: User request to process |
| **Input Format** | `<sop>{{SOP}}</sop>`<br>`<user_request>{{USER_REQUEST}}</user_request>` |
| **Execution Steps** | |
| **Step 1** | Read the user request carefully |
| **Step 2** | Before answering, review the SOP again to ensure you understand all the steps. Within the `<sop_analysis>` tags:<br>a. List the key points of the SOP<br>b. Identify potential challenges or edge cases you may encounter when applying the SOP to the user request<br>c. Consider different ways to interpret any ambiguous parts in the user request |
| **Step 3** | **Crucial:** Process the user request by following each step of the SOP in order. Do not skip any steps, and do not add extra steps not listed in the SOP. |
| **Step 4** | For each step, perform the following:<br>a. Record your progress in the `<step_progress>` tags, including the step number, description, and result (completed, failed, or pending)<br>b. For critical steps, wrap your analysis, verification, and reasoning within `<analysis>` tags<br>c. If a critical step cannot be completed, clearly state the problem and halt the process before proceeding to the next step |
| **Step 5** | After completing every three steps, provide a checkpoint summary within the `<analysis>` tags, highlighting important verification items and any discrepancies or issues encountered. |
| **Step 6** | If at any point you are unsure how to proceed, or if the user's request is inconsistent with the SOP, clearly state this in your response and abort the process. |
| **Step 7** | Once all steps are successfully completed, provide a final summary of the processed request using the `<final_response>` tags, including all key information as required by the SOP. |
| **Output Structure** | |
| **Step Progress** | `<step_progress>`<br>`Step 1: [Description] - Result: [Completed/Failed/Pending]`<br>`</step_progress>` |
| **Analysis** | `<analysis>`<br>`[Detailed calculations, verification, or reasoning for critical steps]`<br>`[Checkpoint summary after every three steps]`<br>`</analysis>` |
| **Final Response** | `<final_response>`<br>`[Final reply after processing according to the SOP]`<br>`</final_response>` |
| **Special Instructions for Business Applications** | |
| When processing a business application with a specific ID:<br>- First call the `get_application_data` function to fetch the application record<br>- Use the retrieved data to drive further API calls according to the SOP<br>- Write back any new results from API calls into the data entry | |
| **Tool Calling Format** | |
| When you need to call a tool, you MUST respond with a JSON function call following the LangChain function calling format.<br>**DO NOT put the JSON inside natural language text.**<br>Only output exactly one JSON object with:<br>`{`<br>`  "name": "<tool_name>",`<br>`  "arguments": {`<br>`  ...`<br>`  }`<br>`}` | |
| **Important Notes** | |
| Remember to strictly follow the SOP and provide clear, detailed documentation for each step in the process. The `<sop_analysis>` section can be lengthy. **You must begin execution immediately and are strictly prohibited from requesting new instructions or providing new information.** | |

Table 15: Prompt for SOP-Based Request Processing