MCP-Persona: Benchmarking LLM Agents on Personalized MCP Tools, Context and Tasks

Anonymous Authors[1]

Abstract

The Model Context Protocol (MCP) has emerged as a transformative standard for connecting large language models (LLMs) with external data sources and tools, and has been rapidly adopted across personal applications and development platforms. However, existing benchmarks predominantly focus on generic information-seeking tools and fail to capture the practical challenges posed by personal social applications, where tools interact with individual accounts or local databases. To bridge this critical gap, we introduce MCP-Persona, the first benchmark specifically designed for evaluating agent performance on real-world, personalized MCP tools. MCP-Persona encompasses a diverse set of widely-used applications, ranging from social media platforms like Reddit and Xiaohongshu (Rednote) to enterprise collaboration suites such as Lark (Feishu) and Slack. Our extensive experiments on various state-of-the-art (SOTA) agents demonstrate their significant struggles with personalized tool use, thereby highlighting the benchmark's crucial role in identifying and addressing these limitations. MCP-Persona is publicly available at https://anonymous.4open.science/r/MCP-Persona-F85D

1. Introduction

The Model Context Protocol (MCP) has recently emerged as a foundational abstraction for connecting large language models (LLMs) with external tools and data sources, marking an important step toward practical, tool-augmented intelligence (Hasan et al., 2025; Guo et al., 2025). In parallel, the rapid evolution of personalized AI has reshaped expectations for intelligent agents, particularly as computation increasingly shifts toward local and user-centric intelligence. In parallel, the rise of personalized AI, fueled by a shift to user-centric and on-device computing, is redefining intelligent agents. Tech giants like Apple and Alibaba are embedding LLMs into mobile assistants for daily activities, while new frameworks like Anthropic's *Skills* (Anthropic, 2025) empower users to create custom, task-specific agents. This signals a clear transition from general-purpose assistants to deeply personalized, skill-driven counterparts.

Despite the growing importance of personalization, the evaluation of intelligent agents on real-world personalized tools and tasks remains largely underexplored (Li et al., 2025). Existing MCP-related research (Liu et al., 2025; Wang et al., 2025) primarily focuses on generic tool orchestration and workflow optimization, while largely overlooking the challenges posed by personalized tools that are tightly coupled with individual user accounts, preferences, and historical behaviors. This gap is particularly problematic given that many high-impact applications of MCP—such as social media, file management, and consumer-facing automation—are inherently personalized by design. The limited progress in this direction stems from several fundamental challenges:

1. First, realistic deployment of personalized MCP servers typically requires access to private user data and substantial human effort for environment setup, making large-scale experimentation costly and difficult to reproduce.
2. Second, privacy concerns and operational security constraints significantly restrict the collection, sharing, and reuse of personal contexts, hindering the construction of open-source benchmarks.
3. Finally, maintaining a stable execution environment that can reliably simulate numerous heterogeneous users while offering controlled and fair evaluation for researchers poses non-trivial technical challenges.

Together, these obstacles have slowed systematic research on personalized MCP agents, underscoring the need for new paradigms and infrastructure that can faithfully capture personalization while preserving privacy and security.

To address the aforementioned challenges, we introduce **MCP-Persona**, the first evaluation platform tailored for tool-enhanced agents operating over real-world personalized tools and tasks. As illustrated in Figure 1, MCP-Persona is

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

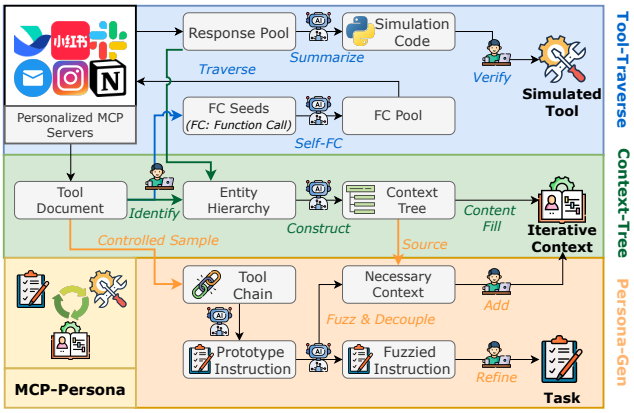


Figure 1. System overview of MCP-Persona, which is built upon the interaction of Tools , Contexts , and Tasks . For each component, we introduce a dedicated method, described in detail as *Tool-Traverse* (§3.1), *Context-Tree* (§3.2), and *Persona-Gen* (§3.3).

built upon the interaction of three core components: tools, contexts (user profiles), and tasks.

Specifically, for tool simulation, we propose **Tool-Traverse**, a traverse-then-simulate paradigm. After collecting a diverse set of real-world personalized MCP servers, we first manually deploy them with sandbox environments and accounts. Building upon a set of human-curated seed function calls (FCs), we introduce Self-FC, a technique inspired by Self-Instruct, to expand the coverage and diversity of authentic function calls. Using this enlarged FC pool, we systematically traverse the MCP servers to collect a wide range of possible response outputs. Based on these observed responses, we summarize the underlying interaction patterns and subsequently generate the executable logic for our simulated tools in the form of Python code.

For context construction, we propose **Context-Tree**, a method for simulating context as a structured tree. The process begins with the tool documents and FC seeds, from which we manually define a hierarchical context structure for each server. For example, in Lark, the hierarchy is `User→Calendar→Event`. By mapping the relationships between all tool parameters, we construct a preliminary context tree that naturally includes redundant subnodes. We populate the context tree by assigning values to each node. To maximize realism, we prioritize authentic online content (e.g., Xiaohongshu posts) when available, while sensitive fields such as phone numbers are replaced with fakes to preserve privacy. Once populated, a context instance is supplied to the simulated tools, enabling stateful operations such as creating, modifying, and deleting entities within the environment.

For task generation, we introduce **Persona-Gen**, a two-stage pipeline involving both automated synthesis and manual refinement. The process starts with automatically generating prototype instructions from high-quality tool chains, which are sampled under the constraints of dependency, diversity, and realism. To transform these abstract prototypes into realistic tasks, we then inject rich context from our Context-Tree and intentionally obfuscate the instructions to mimic the ambiguity of real-world user requests. Subsequently, human annotators review each task to verify its alignment with the tool chain and refine it by adding necessary contextual details. This rigorous process yields our final set of 173 high-quality, human-verified personalized tasks.

We conduct extensive experiments to check the performance of SOTA agents on MCP-Persona. The results exhibit several key findings: (1) Even SOTA agents like GPT-5 still fall short in some personalized tasks with unseen tools. They have trouble acquiring necessary information embedded in the environment that is not explicitly exhibited in the instruction. (2) Using simulation for operation-dominated MCP servers and limited information, our simulation method yield very accurate prediction with only limited manual function calling efforts. To summarize, our contributions are as follows:

1. We propose Tool-Traverse, a novel and rigorous simulation method designed to faithfully replicate the functionalities of real-world personalized MCP tools, which cannot be openly evaluated due to account-binding requirements.
2. We construct MCP-Persona, a new benchmark comprising 173 human-verified tasks spanning four distinct personal scenarios, addressing the critical user demand for enhanced agents in social communication applications.
3. We conduct extensive experiments on over ten SOTA models and reveal their significant limitations in personalized tool usage, particularly in terms of underexploration of tool functionalities and the failure to discover information embedded within the environment.

2. Related Work

2.1. Personalized Agents in Real-World Applications

Personalized agents have rapidly proliferated into widely deployed products, forming an emerging ecosystem that supports individualized, localized, and everyday user needs. Anthropic introduce *Skills* () exposes a modular and extensible mechanism for equipping agents with user-specific capabilities, allowing developers and end users to compose customized skills tailored to localized environments and diverse real-world demands, including social interaction, personal assistance, and routine life services. In the GUI domain, *Doubao Phone* integrates on-device intelligence to deliver deeply personalized mobile experiences, enabling users to interact with agents that adapt to personal preferences, usage habits, and local contexts while supporting

Table 1. Comparison of MCP-related or personal benchmarks.

Benchmark	Real-World	Personal Context	Task Coverage			
			Social Media	Collaboration Platform	Email	Content Management
AppWorld (Trivedi et al., 2024)	✗	✓	✗	✗	✓	✓
PersonaBench (Tan et al., 2025)	✗	✓	✗	✗	✓	✓
MCP-Universe (Luo et al., 2025)	✓	✗	✗	✗	✗	✓
TOOLATHLON (Li et al., 2025)	✓	✗	✗	✗	✓	✓
MCP-Persona (Ours)	✓	✓	✓	✓	✓	✓

daily activities such as communication, shopping and food delivery. Very recently, *ClawdBot* has caught significant attention. It focuses on personalized data and task management, providing agent-driven assistance for individual workflows and personal organization, thereby demonstrating how agents can be seamlessly integrated into personal scenarios with a high degree of automation. Together, these examples illustrate a broader trend in which personalized agents are increasingly integrated into consumer-facing products and application ecosystems, emphasizing adaptability, locality, and alignment with everyday human activities.

2.2. Research on Real-World Tool Agents

While the success of personalized tool-augmented agents have stimulated substantial community interest, existing research rarely focus on real-world personalized scenarios, particularly those involving social media platforms and enterprise collaboration systems. They are confronted with the following limitations: (1) First, most existing tool-use benchmarks and datasets primarily target generic search-oriented tools, tasks and simplified scenarios (Luo et al., 2025; Xu et al., 2025; Fan et al., 2025), which fail to capture the complexity and personalization characteristics of real-world applications. (2) Second, although a few benchmarks attempt to incorporate personalization (Trivedi et al., 2024; Tan et al., 2025), they dominantly rely on synthetic tools. For instance, Tau-Bench (Yao et al., 2024) servers as the first tool-agent-user benchmark, addressing airline and retail tasks. However, its evaluation is conducted exclusively on synthetic tools, whose distributions may substantially deviate from those of real-world systems. (3) Third, the most recent personalized tool benchmark, ToolAthlon (Li et al., 2025), provides a comprehensive evaluation suite with tasks using real-world Notion and email tools. Nevertheless, due to the intrinsic difficulty of account-associated and permission-sensitive environments, ToolAthlon does not support task evaluation on social media platforms or enterprise communication tools, which are among the most widely used applications in everyday scenarios.

This gap is particularly concerning given the critical role of individualized demands in real-world agent usage (e.g., ClawdBot). To bridge this gap, we simulate real-world tools from popular applications such as Instagram and Lark and introduce MCP-Persona, the first benchmark explicitly designed for personalized tool usage in social communication scenarios.

3. Methodology

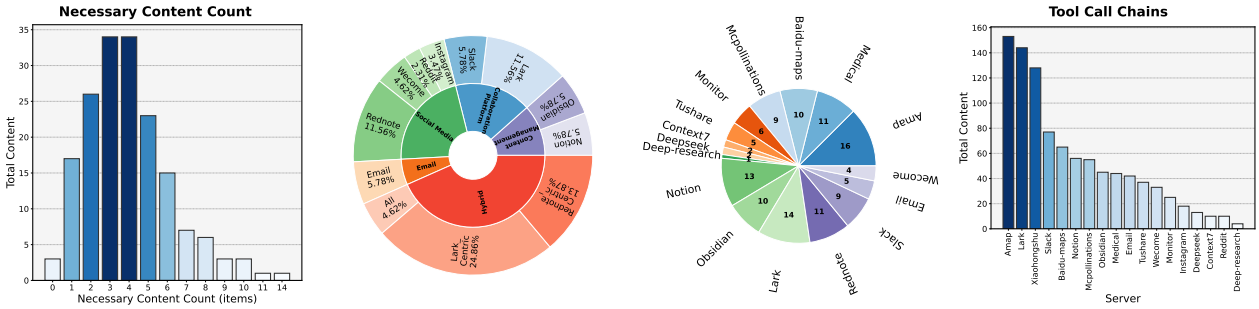
3.1. *Tool-Traverse*: Simulating Real-World MCP Servers by Traversing Authentic Tool Responses

3.1.1. MCP Server and Tool Collection

To build a comprehensive benchmark that bridges the gap between generic and personalized tool use, we employ a hybrid collection strategy combining rigorous manual curation with automated discovery. We focused our primary data collection efforts on high-value, personalized applications that require authentication and state management. We manually curated a diverse set of 7 MCP servers spanning critical categories including enterprise collaboration platforms (e.g., `Slack`, `Lark`), social media (e.g., `Instagram`), and content management (e.g., `Notion`). Server details are attached in Appendix, Table 5. To simplify testing in controlled environments, we standardized the deployment infrastructure and unified authentication by provisioning dedicated test accounts and pre-configured tokens, ensuring a stable execution environment for personalized tools. In parallel, we utilize the framework following Wang et al. (2025) and built an automated pipeline to harvest information-seeking MCP tools, which are typically stateless and require no user-specific authentication, enabling automatic discovery, validation, and deployment from open-source repositories without manual configuration.

3.1.2. Function Call Generation Based on Manual Seeds

To construct a high-fidelity simulator, we must first establish a function call pool that captures the authentic behavioral distribution of MCP servers. Relying solely on static tools' documentation is insufficient, as it often fails to document implicit constraints and error handling logic. We therefore propose an **Tool Traversal** paradigm that systematically probes real MCP servers to map both their successful op-



(a) Necessary context length. *(b)* Query type distribution. *(c)* Server tool coverage. *(d)* Tool-call chain length.

Figure 2. Dataset statistics of MCP-Persona.

eration manifolds and their decision boundaries for failure cases.

Successful Seed Call and Schema Instantiation. For every tool t within a collected MCP server set $\mathcal{T}$, we first perform positive traversal to record nominal behaviors. We parse the tool's input schema $\mathcal{S}_t$ to identify required parameters and data types. Human annotators construct valid seed function calls x_{seed} that satisfy semantic constraints (e.g., ensuring a generic ID corresponds to an existing entity). We execute these calls against the live server and record the interaction tuple $(t, x_{seed}, y_{seed}, \tau)$, where y_{seed} is the authentic response and τ represents execution metadata. This establishes the baseline for correct tool functionality.

Adversarial Failure Induction. A robust simulator must accurately replicate how servers handle invalid requests. To capture these error modes, we implement an LLM-driven adversarial call generation pipeline. We model the input call space $\mathcal{X}$ and systematically perturb valid seed inputs x_{seed} to generate a set of invalid inputs $\mathcal{X}_{fail} = \{x'_1, x'_2, ..., x'_k\}$. The generation process is guided by a taxonomy of error types, ensuring coverage across: (1) *Type Mismatches*, violating primitive type constraints; (2) *Schema Violations*, omitting required fields; (3) *Boundary Conditions*, exceeding numerical ranges; and (4) *Semantic Conflicts*, where parameter combinations are syntactically valid but logically contradictory. The generated inputs are executed against the live server. To ensure data quality, we employ a secondary LLM as a **Response Discriminator** to classify the server's output. This discriminator determines whether the tool correctly identified the error (returning a structured error message) or failed unexpectedly. Only verified failure modes are added to the function call pool, ensuring the simulator learns to replicate specific error codes rather than generic failures.

3.1.3. Executable Code-Based Tool Simulation

Leveraging the function call pool, we construct a fully executable local replica of the target MCP tools by using SOTA LLMs to summarize their underlying processing logic. A critical distinction of our approach is the **Code-as-Simulation** paradigm: rather than relying on manual rule-coding or static mock responses, our pipeline is *fully automated*, employing LLMs to autonomously synthesize executable Python files that serve as the simulation engines. This architecture ensures scalability across thousands of diverse tools without human intervention. The simulation consists of two automatically generated components:

Dynamic Context Handler Generation. As described in Section 3.2, real-world tools operate on contexts with hierarchical data structures (e.g., a User has Calendars, which have Events). Based on this schema, we employ the LLM summarizer to autonomously writes a *Context Handler* Python module. This generated script provides a standardized, code-level interface for entity manipulation—load, save, query, and modify—allowing the simulator to persist state changes across multiple interaction turns automatically.

LLM-Synthesized Simulation Kernels. The core functional logic for each tool t is encapsulated within an individual Python script file, which is synthesized entirely by the LLM. The generation process is conditioned on: (1) the tool's input schema $\mathcal{S}_t$; (2) the collected behavioral traces containing both successful and adversarial interactions; and (3) the generated context handler APIs. Specifically, the LLM is prompted to write a complete Python function implementation, K_t, that acts as the transition function

$$f_t : (\mathcal{C}_{current}, x) \rightarrow (\mathcal{C}_{new}, y),$$

where x and y denote tool call and responses respectively. This generated Python code includes rigorous logic to validate inputs, check entity existence against the dynamic context (e.g., ensuring a 'calendar_id' exists before adding an event), and return appropriate success or error responses. By training the LLM to write code that handles the specific failure modes identified in the traversal phase, the resulting Python files accurately reproduce the exact decision boundaries and error messages of the live servers. These

generated scripts are then executed in a sandboxed Python environment to serve user requests.

3.2. *Context-Tree*: Constructing MCP Server Contexts via a Tree Hierarchy of User Profiles

To enable a high-fidelity simulation, a structured and mutable context is required to support multi-turn, stateful tool execution and personalized task synthesis. Therefore, we construct a server-level context C using a tree hierarchy, which can interact with the dynamic context handler (Section 3.1) and the task generator (Section 3.3).

Entity and Hierarchy Identification. Each MCP server contains highly distinct entities and content (e.g., Slack functions through groups while Xiaohongshu involves posts). To unify them under a consistent processing pipeline, we propose a tree structure that is both well-organized and easily extensible, supporting dynamic interactions. The hierarchy of this context tree serves as the foundation for our context construction. Specifically, for each application-level MCP server, the server-specific context hierarchy defines the entity types, their associated fields (with coarse constraints), and the ownership or reference relationships among them. We begin with the tool-call pool $\mathcal{P}^+$ collected in Section 3.1.2. The hierarchy is derived by aggregating fields and identifiers in $\mathcal{P}^+$ and aligning repeated object patterns across tools to define entity types and relations. To guarantee the robustness of our contexts, we build the hierarchy via human-in-the-loop annotation: annotators (i) consolidate entity types by grouping tools operating on the same objects, (ii) aggregate fields to identify keys, references, and coarse constraints, and (iii) define a hierarchy for organizing entities, typically rooted at `User` in personalized settings.

Context-Tree Construction. Given the tree hierarchy, we materialize a user-rooted context in which nodes correspond to typed entities and edges follow the ownership/containment relations defined in the hierarchy (e.g., `User`→`Chat`→`Message`, `User`→`Calendar`→`Event`). For each parent entity, its child entities of the same type are stored as an identifier-indexed map $\{\texttt{id} \rightarrow \texttt{obj}\}$, which ensures uniqueness and enables efficient lookup and updates. Non-containment relations are encoded via identifier references (foreign keys) rather than duplicating full objects. If different tools expose different fields of the same entity, we match by the entity ID and incorporate newly observed fields into the existing entry, keeping the context consistent across turns.

Content Sourcing and Generation. To populate each server-specific context tree, we first employ an LLM to automatically assign one of four generation method to every field in the context hierarchy. (1) *Enum:* constrained sampling from a fixed set; (2) *LLM:* LLM-conditioned free-form generation; (3) *Special:* LLM-synthesis of a structured generator, prompted with an example and lightweight constraints (e.g., required prefixes or formats) to create a compliant generator; Rule-based generation for structured fields, where content is randomly generated according to predefined constraints such as format, length, or prefixes (e.g., generating a 10-digit phone number); or (4) *Real:* seeding with sanitized authentic text from text-heavy platforms (e.g., Xiaohongshu) that preserves realism while masking user information such as nicknames. Examples of different field content generation are provided in Table 2. After the contents are filled in the context tree, We then perform cross-entity linking. For entities that require references to other entities (e.g., chat members), we form links by sampling from the already generated contents and inserting the corresponding identifiers, establishing consistent relational structure.

Table 2. 4 generation methods with example fields.

Method	Example
Enum	`"iplocation": ["Beijing", "Shanghai"...]`
LLM	`"channel_name": "Mcp-Flow discussion group"`
Special	`"chat_id": "oc_50df962af-41184f144d3ebf61b0c7571"`
Real	`"desc":"I wonder what handwritten English looks like to a native speaker?"`

3.3. *Persona-Gen*: Personalized Task Generation with Tool Invocation Chain and Instruction Fuzzification

Based on the developed simulation environments, we further propose a novel pipeline for personalized instruction synthesis. The process begins with heuristically prototyping instruction skeletons from tool invocation chains and API documentation. This is followed by context injection and decoupling, which transform these abstract skeletons into situated personal tasks. All synthesized data are then subject to rigorous manual review and revision to guarantee quality and diversity

Tool Invocation Chain Sampling To ensure a wider spectrum of task diversity and complexity, we first generate prototype instructions by sampling tool invocation chains, rather than asking human annotators to compose tasks from scratch. Our tool chain sampling method is designed to satisfy five key considerations: (1) *Dependency*: To preserve inherent dependencies, tool chains are sampled at the unit level—defined as a combination of several tools—rather than as individual tools. In practice, MCP tools from the

same server are often designed to be used sequentially, such as listing available models before invoking a specific one for usage. (2) *Personalization*: Each sampled chain must include at least one personalized tool. (3) *Deduplication*: We ensure that no two sampled tool chains are identical, maximizing the diversity of the generated dataset. (4) *Coherence*: The sampled chain must be semantically coherent, meaning the outputs of upstream tools must serve as valid and necessary inputs for downstream tools. (5) *Realism*: All sampled chains undergo a manual review process where any chain that does not reflect a realistic user scenario is discarded.

Instruction Prototyping and Context Enrichment Using the curated set of high-quality tool chains $\mathcal{L}^*$ and their corresponding formal API documentation $\mathcal{D}$, we employ an LLM-based heuristic method to synthesize preliminary instruction templates, denoted as S_{proto}. Formally, $S_{proto} = \mathcal{H}(\mathcal{L}^*, \mathcal{D}, \mathcal{P})$, where specific entities (e.g., user identifiers, product names) are replaced with typed placeholders $\mathcal{P}$ (e.g., `[USER_ID]`, `[PRODUCT_NAME]`). This abstraction removes instance-specific details while explicitly preserving (i) parameter-level dependencies across tool calls and (ii) the global logical structure that orchestrates the tool chain. Since the resulting instruction skeletons lack contextual grounding, we further concretize them by injecting realistic context. Specifically, we sample entity values from our constructed interactive contexts $\mathcal{C}$ and replace the placeholders in S_{proto} accordingly, yielding instantiated instructions S_{inst}.

Instruction Fuzzification and Context Decoupling To better resemble authentic user inputs, we deliberately introduce fuzziness by removing information that humans typically omit in natural instructions. We define *implicit context* as a set of underlying parameters that are essential for tool execution but frequently absent from user instructions (e.g., `user_id` of a coworker). Such information is often ambiguous or underspecified in natural language (Li et al., 2025), yet its intended values can be deterministically inferred from the environment state (e.g., locating the unspecified coworker by querying the information of a shared group). Our strategy explicitly separates implicit context C_{imp} from the natural intent instruction. Specifically, the instruction surface form is obtained by removing execution-specific parameters from the instantiated instruction, $I_{ins} = \mathcal{F}(S_{inst} \setminus C_{imp})$, where $\mathcal{F}$ denotes a fuzzification operator.

Rigorous Filtering and Human Verification Following the automated synthesis pipeline, the candidate data triples $\mathcal{X}c = (I_{ins}, C_{total}, \mathcal{L}^*)$ undergo a multi-stage manual curation process to ensure benchmark fidelity and task quality. Annotators ensure strict consistency among the natural language instruction I_{ins}, the decoupled context C_{total}, and the ground-truth tool chain $\mathcal{L}^*$, forming a coherent, solvable and unambiguous task. Moreover, we deliberately increase task difficulty through two primary strategies: scaling query quantities (e.g., from "summarize one post" to "summarize ten posts") and rigorously pruning necessary context by removing any information that can be procedurally derived via the available tool chain. Tool Chains are synchronously enlarged and evolved during this process. The annotation prompt is provided in Appendix C.

3.4. Evaluation

We propose **execution-based** and **checkpoint-based** evaluation frameworks to analyze modified contexts and simulation tool calling outputs.

3.4.1. Checkpoint Based Evaluations

The **Checkpoint-Based** evaluation focuses on the intermediate values generated during the agent's multi-step execution. Checkpoints are established at the boundary between decomposed sub-tasks. Based on the comparison of model's input parameters and predefined standard input parameters and simulated tool output, a checkpoint is score independently based on LLM judges.

3.4.2. Execution Based Evaluations

To evaluate personalized tasks in the simulated environment, we formally abstract simulated personalized tool as a simulated Create, Read, Update, Delete (*CRUD*) operation on the structured Context File. We propose an **Execution-based** evaluation process to directly measure this success. We categorize checkpoints into three classes: *Generic Search*, *Personalized Search* (involving only *Read* operations on $\mathcal{C}_u$), and *Personalized Operations* (involving state-altering *Create*, *Update*, or *Delete* operations). For the two search types, human annotators execute tools to obtain ground-truth responses, which are compared against the agent's outputs for scoring. For Personalized Operations, we implement three dedicated executors: *CreateDataExecutor*, *UpdateDataExecutor* and *DeleteDataExecutor*. For different types of operations, LLM-based Judges can determine whether the model has completed corresponding tasks in the simulated context based on the actual updates of the context in the sandbox. All ground-truth data, context indices, and specifications undergo meticulous human annotation and verification.

4. Experiments

4.1. Main Experiment

Setups. We apply the proposed benchmark to multiple models for comparative evaluation. The main results are

Table 3. Main experiment results on MCP-Persona.

Model	Single-Server				Multi-Server			Overall		
	Collaboration Platform	Content Management	Social Media	Email	Lark-Centric	Rednote -Centric	Hodgepodge	Acc	SR-0.8	Exec-Acc
Proprietary Models										
Claude-4.5-Sonnet	39.94	19.76	47.04	43.63	40.81	42.37	12.50	38.66	10.40	41.50
GPT-5	43.50	22.57	42.64	47.17	37.67	34.66	12.50	36.99	6.94	41.45
Claude-4.1-Opus	38.79	13.56	44.79	9.71	39.67	34.70	25.00	34.52	7.05	36.77
o4-mini	34.38	21.22	35.61	53.83	30.43	25.25	6.25	30.70	5.78	34.73
o3	26.41	14.55	32.78	41.08	34.64	26.05	37.50	29.79	5.20	30.27
GPT-4o	24.50	7.58	36.98	12.57	30.65	20.29	25.00	25.56	4.35	20.02
Grok-4	17.80	11.82	39.43	5.71	26.78	19.79	37.50	24.58	6.68	22.49
Gemini-3-Pro	14.01	11.03	23.38	36.92	14.60	22.05	6.25	16.91	1.79	5.78
Gemini-2.5-Pro	22.58	22.24	26.22	20.92	18.23	11.76	6.25	20.68	0.66	13.38
Open-Source Models										
Qwen3-Max-Latest	24.36	11.67	47.98	11.71	30.95	15.79	18.75	27.54	5.75	29.23
Qwen3-235B	23.55	12.12	40.05	13.71	30.40	19.25	31.25	26.75	4.07	21.83
Qwen3-Coder	23.50	13.18	31.34	8.29	29.80	23.57	6.25	23.93	3.65	20.14
DeepSeek-V3	19.22	11.48	38.35	30.79	27.91	18.89	18.75	25.29	3.47	27.52

summarized in Table 3. We report 3 metrics. All columns except the last three under **Overall** report Acc. Checkpoint Accuracy (Acc) is defined as the average checkpoint score within a task, where each checkpoint is scored by the LLM judge. Acc is the default Success Rate at 0.8 (SR-0.8) measures the proportion of tasks whose Acc exceed 0.8. Execution Accuracy (Exec-Acc) is defined as the average executor-verified checkpoint correctness over all human-specified execution steps within a task. Our benchmark includes two types of tasks: Single-Server tasks, which use tools from one personalized server, and Multi-Server tasks, which require coordinating tools across multiple personalized servers. In both settings, tool chains may also include information-seeking tools such as `Amap`.

Results. Across settings, current agents show limited reliability on our benchmark: performance varies by tool family even in single-server tasks and degrades further with richer context and longer-horizon coordination. The results reveal common failure modes in tool use and demonstrate that our benchmark is a useful testbed for evaluating and improving agent capabilities.

Single-server breakdown. Email tasks achieve the highest accuracy due to simple operations and short dependency chains. Tasks involving richer schemas or heterogeneous objects—such as social media and collaboration platforms—are more challenging, as they require handling cross-user interactions and implicit entity resolution. Content-management tools perform worst, reflecting agents' limited robustness when navigating and editing long documents under extended context.

Multi-server breakdown. We evaluate **Lark-centric**, **Rednote-centric**, and a heterogeneous **hodgepodge** variant. The hodgepodge setting is the hardest among our multi-server variants, with the lowest accuracy, due to more cross-server interactions and more complex dependency chains.

Metrics insight. Exec-Acc is slightly higher than **Acc**, suggesting that human-annotated execution checkpoints are less noisy than LLM-segmented checkpoints. **SR@0.8** remains low, indicating difficulty in completing tasks end-to-end.

4.2. Validation of Simulation Fidelity

To justify the effectiveness of our **Tool-Traverse** paradigm (Section 3.1), we quantitatively verified that our code-based simulation (T_{sim}) exhibits behaviors and responses indistinguishable from real-world tools (T_{real}).

Setup and Context Reconstruction. We evaluated the **Lark** server (14 tools) using 50 authentic interaction traces (25 successful, 25 failed). A critical challenge in simulation is ensuring state consistency; a simulator might fail simply because it lacks a specific entity (e.g., a user ID) present in the real environment, not because of flawed logic. To eliminate this confounding variable, we employed a **Context Reconstruction** strategy: we reverse-engineered the exact pre-condition state (Context C) from each real-world trace and injected it into the simulator. This guarantees that T_{sim} and T_{real} operate on the exact same state snapshot, ensuring a fair comparison based solely on execution logic. We compared our approach against a **Vanilla Baseline**, which simulates tools relying only on documentation without behavioral traversal data.

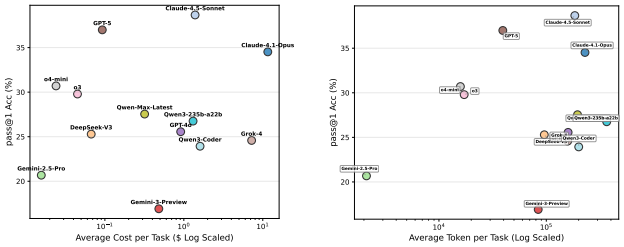
Results. We measured *Behavioral Alignment* (Accuracy, F1) and *Response Similarity* (TF-IDF, METEOR, etc.). As shown in Table 4, **Tool-Traverse** significantly outperforms the baseline. The Vanilla model achieves only 53.3% F1, suffering from a high False Negative rate (FN=13), indicating it struggles to handle valid but complex inputs due to hallucinated constraints. In contrast, our method achieves **93.8% F1** and **94.0% Accuracy**. Furthermore, regarding

Table 4. Comparison of Simulation Fidelity on the Lark Server (50 samples). **Tool-Traverse** (Ours) aligns significantly better with real-world behavior than the documentation-only Vanilla baseline. TP/TN: True Positive/Negative; FP/FN: False Positive/Negative.

Method	Confusion Matrix				Behavioral Alignment (%)				Response Similarity			
	TP	TN	FP	FN	Acc	Prec	Rec	F1	TF-IDF	ROUGE	BLEU	METEOR
Vanilla	12	17	8	13	58.0	60.0	48.0	53.3	0.2113	0.2258	0.2307	0.3214
Tool-Traverse	**23**	**24**	**1**	**2**	**94.0**	**95.8**	**92.0**	**93.8**	**0.7391**	**0.7366**	**0.7412**	**0.8703**

output semantics, Tool-Traverse nearly triples the METEOR score (0.87 vs. 0.32), proving it accurately reproduces specific error codes and data structures, whereas the baseline yields generic templates.

4.3. Token and Cost Analysis



(a) Average token per-task *(b)* Average cost ($) per-task

Figure 4. Relationship between resource consumption (cost and tokens) per task and model accuracy (pass@1).

Figure 4 demonstrates the relationship between resource consumption (cost and tokens) and model performance (pass@1 accuracy). Notably, GPT-5 demonstrates excellent cost-effectiveness, achieving 36.99 accuracy with only $0.09 cost and 39,343 tokens. The results reveal no clear correlation between resource investment and performance, suggesting that model selection should prioritize specific accuracy-cost trade-offs.

4.4. Error Analysis

After careful study of sampled failed traces, we identify three recurring failure archetypes that account for most end-to-end failures in our tool-using setting.

Type 1: Under-exploration. Weaker models often stop after producing a superficially plausible action, without sufficiently exploring the intended toolchain or verifying implicit constraints.

Example Task: "...Additionally, please send a polite message to my supervisor, Song Ke, explaining my condition and requesting leave."

Example Context:"Song Ke user_id: o9k5jtwo"

The context contains Feishu/Lark identity hints for Song Ke (e.g., user-id-like fields), but the instruction does not explicitly specify the platform. Instead of resolving Song Ke's Feishu/Lark identity and sending the message via Feishu/Lark, the agent sends a WeCom message to a hallucinated recipient and then terminates. As a result, the output appears on-topic but fails the task due to (i) platform mismatch and (ii) missing recipient grounding.

Type 2: Missing implied information needs. A second recurring failure is skipping latent dependency steps implied by tool schemas.

Example Task: "...If everything looks fine, schedule the review meeting for next Monday from 10:00 AM to 12:00 PM in the main conference room. Use the title '2025 Q4 Team Review Meeting.' Also, have Zhao host the meeting since she has more experience. Remember to set her as the host for the video conference as well."

Example Context: "Zhao's phone number": "+861380-0138000"

The intended workflow is to resolve Zhao's platform-internal id `user_id` from the phone number in context via the `user_batchGetId` tool, and then pass the resolved `user_id` into the `calendarEvent_create` to create a meeting with Zhao as the host. However, weaker models would directly substitute the phone number for `user_id` (or fabricate an ID), causing execution errors or silently incorrect host settings.

Type 3: Long context problems Our context-tree design can induce progressive context stacking across turns, and certain tools (e.g., local folder/document readers) can return large payloads that sharply increase the effective in-context length. As trajectories grow, models increasingly miss earlier constraints or overlook critical observations, and the output would fail to complete even simple tasks.

5. Conclusion

In this work, we introduce **MCP-Persona**, a benchmark for evaluating tool-augmented agents in realistic personalized settings via simulated MCP servers, structured context trees, and human-verified tasks. Our experiments show that even state-of-the-art agents struggle with personalized tool use, particularly in implicit information grounding, multi-step state maintenance, and cross-tool coordination. We hope MCP-Persona provides a reproducible testbed to support future research on personalization-aware agents.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Anthropic. Introducing agent skills, October 16 2025. URL https://claude.com/blog/skills.

Fan, S., Ding, X., Zhang, L., and Mo, L. MCPToolBench++: A Large Scale AI Agent Model Context Protocol MCP Tool Use Benchmark, August 2025.

Guo, Z., Xu, B., Zhu, C., Hong, W., Wang, X., and Mao, Z. MCP-AgentBench: Evaluating Real-World Language Agent Performance with MCP-Mediated Tools, September 2025.

Hasan, M. M., Li, H., Fallahzadeh, E., Rajbahadur, G. K., Adams, B., and Hassan, A. E. Model context protocol (mcp) at first glance: Studying the security and maintainability of mcp servers, 2025. URL https://arxiv.org/abs/2506.13538.

Li, J., Zhao, W., Zhao, J., Zeng, W., Wu, H., Wang, X., Ge, R., Cao, Y., Huang, Y., Liu, W., Liu, J., Su, Z., Guo, Y., Zhou, F., Zhang, L., Michelini, J., Wang, X., Yue, X., Zhou, S., Neubig, G., and He, J. The Tool Decathlon: Benchmarking Language Agents for Diverse, Realistic, and Long-Horizon Task Execution, October 2025.

Liu, Z., Qiu, J., Wang, S., Zhang, J., Liu, Z., Ram, R., Chen, H., Yao, W., Wang, H., Heinecke, S., Savarese, S., and Xiong, C. MCPEval: Automatic MCP-based Deep Evaluation for AI Agent Models, July 2025.

Luo, Z., Shen, Z., Yang, W., Zhao, Z., Jwalapuram, P., Saha, A., Sahoo, D., Savarese, S., Xiong, C., and Li, J. MCP-Universe: Benchmarking Large Language Models with Real-World Model Context Protocol Servers, August 2025.

Tan, J., Yang, L., Liu, Z., Liu, Z., Murthy, R., Awalgaonkar, T. M., Zhang, J., Yao, W., Zhu, M., Kokane, S., Savarese, S., Wang, H., Xiong, C., and Heinecke, S. PersonaBench: Evaluating AI Models on Understanding Personal Information through Accessing (Synthetic) Private User Data, August 2025.

Trivedi, H., Khot, T., Hartmann, M., Manku, R., Dong, V., Li, E., Gupta, S., Sabharwal, A., and Balasubramanian, N. AppWorld: A Controllable World of Apps and People for Benchmarking Interactive Coding Agents. In Ku, L.-W., Martins, A., and Srikumar, V. (eds.), Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 16022–16076, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.850.

Wang, W., Niu, P., Xu, Z., Chen, Z., Du, J., Du, Y., Pang, X., Huang, K., Wang, Y., Yan, Q., and Chen, S. Mcpflow: Facilitating llm agents to master real-world, diverse and scaling mcp tools, 2025. URL https://arxiv.org/abs/2510.24284.

Xu, Z., Soria, A. M., Tan, S., Roy, A., Agrawal, A. S., Poovendran, R., and Panda, R. TOUCAN: Synthesizing 1.5M Tool-Agentic Data from Real-World MCP Environments, October 2025.

Yao, S., Shinn, N., Razavi, P., and Narasimhan, K. τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains, June 2024.

Task: ICML's deadline is coming up soon, so our team needs to hold a project meeting. First, inform everyone in the project group that we're going to hold a meeting, then check whether everyone has confirmed in the group over the past three days, especlally the key people like Xiao Jiekal and Zhang Yanyuan. Also, remember to update all the meeting details in the meeting description, including the attendee list, agenda, and video conference link, and set a reminder. Then create a follow-up "Project Meeting 2" immediately afterward, also one hour long, in case someone forgets or the meetingruns overtime

Necessary context: chat_id: oc_2cf23b797f122483dffe11dd46fbe9bf

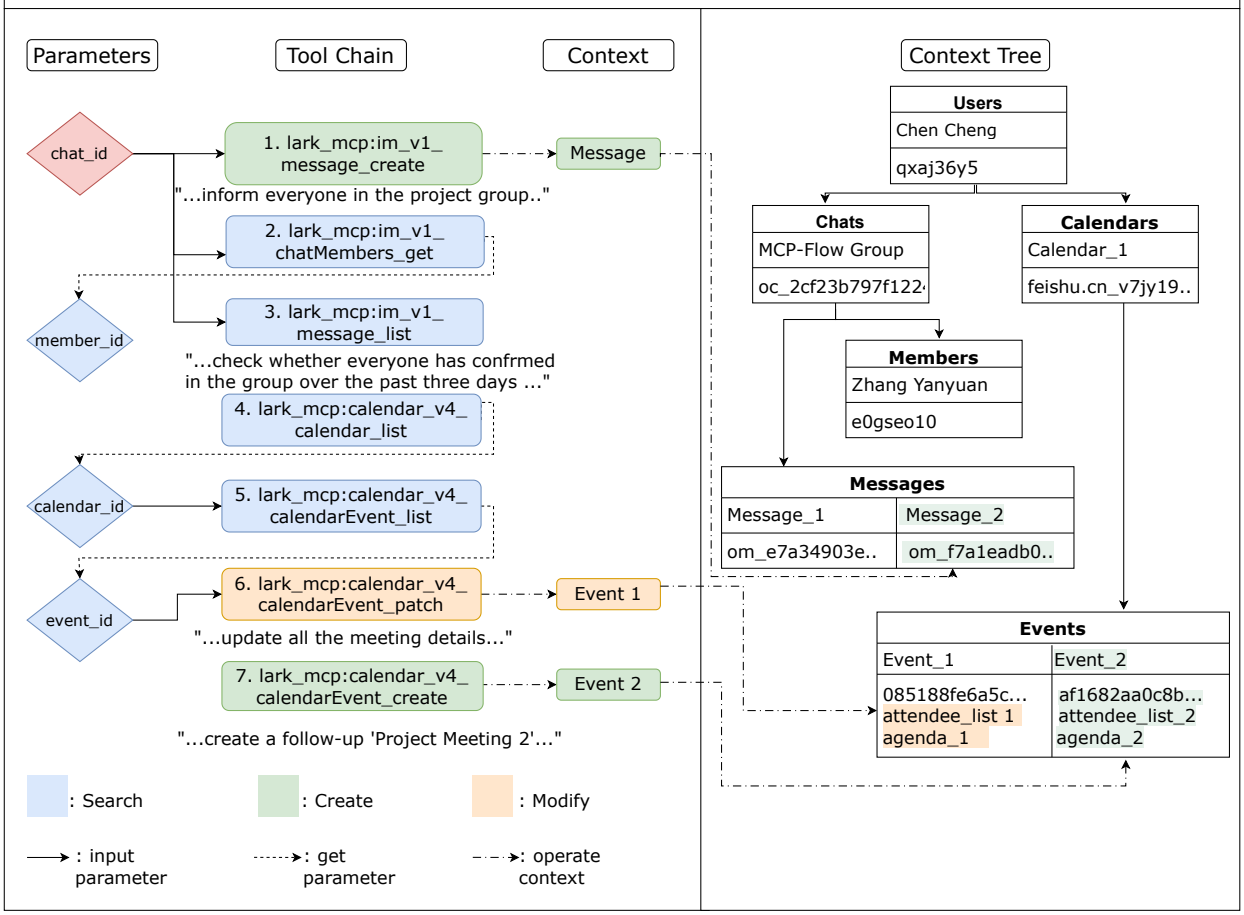


Figure 3. A case study of a Lark-MCP task. The workflow illustates task execution through tool calls that either retrieve required context or apply operations to update the context. The right panel shows an example Lark-MCP context tree, which can be accessed and edited by corresponding tools

Table 5. MCP Servers Included in MCP-Persona Benchmark

MCP Server	Icon	URL
Search		
Amap		https://github.com/sugarforever/amap-mcp-server
Mcpollinations		https://github.com/pinkpixel-dev/MCPollinations
Context7		https://github.com/upstash/context7
Medical		http://github.com/JamesANZ/medical-mcp#
Baidu-maps		https://modelscope.cn/mcp/servers/@baidu-maps/mcp
Monitor		https://github.com/seekrays/mcp-monitor#
Tushare		https://modelscope.cn/mcp/servers/@zhewenzhang/tushare_MCP
Deepseek		https://github.com/DMontgomery40/deepseek-mcp-server
Deep-research		https://smithery.ai/server/@baranwang/mcp-deep-research
Local Note-Taking		
Notion		https://github.com/suekou/mcp-notion-server?tab=readme-ov-file
Obsidian		https://github.com/MarkusPfundstein/mcp-obsidian
Social Communication		
Lark		https://github.com/larksuite/lark-openapi-mcp
Slack		https://smithery.ai/server/slack
Wecome		https://github.com/gotoolkits/mcp-wecombot-server
Xiaohongshu		https://github.com/xpzouying/xiaohongshu-mcp
Email		https://github.com/TimeCyber/email-mcp
Reddit		https://smithery.ai/server/reddit
Instagram		https://smithery.ai/server/instagram

A. Tool and Data Details

MCP Servers To ensure reproducibility and facilitate future research, we include the names and links of all MCP servers in our benchmark. Table 5 groups the MCP-Persona servers into search, local note-taking, and social communication categories, along with their official repository URLs. Overall, these servers expose external tools and data sources to the model, enabling tasks such as retrieval, content management, and multi-step information gathering. Together they cover both personal information workflows and public information access that the benchmark evaluates.

B. Tasks Examples

In Table 6, we conclude the benchmark examples.

C. Human Annotations Guide

Instructions-ToolChains-Context Alignment Annotation Guide

Human Annotation Guide
1. Instructions-ToolChains-Context Alignment Annotation
Core Objective
Ensure strict semantic consistency among the natural language instruction, decoupled context, and ground-truth tool chain, forming a coherent, solvable, and unambiguous personalized task.
Key Steps

1. **Task Selection:** Pick tasks with clear structure and logical tool chains from the candidate dataset, prioritizing those with linear dependencies.
2. **Instruction Optimization:**

Table 6. Tasks examples in the benchmark.

Example 1. *Task.* Tomorrow at 9:00 a.m. we're holding the quarterly marketing strategy meeting, and I'm busy preparing the meeting materials right now. Please set a reminder for Li Minghui on Slack, and also record a voice prompt in nova's voice style and send it over. Then help me draft a simple meeting agenda, listing key topics like market analysis and the creative proposal. Oh, and make sure to check everyone tagged "Mingri Technology" and message them so that every single one of them receives the notice—don't miss anyone and end up delaying things.
Example 2. *Task.* Recently I've been preparing a research project on urban healthy walking and need to gather some basic information. First, please check what parks and medical institutions are within 1 km of Jianguo Road, and then find the most convenient way to get from my home to the nearest park. Update this route information in the research plan, and also look up recent findings in medical journals on the health benefits of urban walking. Finally, organize all of these materials and place them in the research reference resources for easy follow-up and consultation.
Example 3. *Task.* I'm planning to drive out and relax this weekend—I'm currently in Taiwan. Please check the most convenient driving route from here to Taipei 101, and also look for some good local specialty restaurants within 500 meters nearby. Also, first confirm that I'm still logged into my Xiaohongshu account "Guokr," so I won't have to log in on the spot when I want to post an update. Finally, remember to add my weekend plan to the end of the weekly plan note in Obsidian so it'll be easier to review later.

- Increase task difficulty (e.g., scale query quantities).
- Refine the instruction structure to align with real usage scenarios.

3. **Context Refinement:**
   - Minimize necessary context: Remove information that can be derived via tool calls.
   - Filter unnecessary context: Retain only context relevant to the task.
   - Update context content to match the instruction scenario.
4. **Tool Chain Adjustment:**
   - Verify tool solvability: Check for missing parameters.
   - Evolve tool chains synchronously with instruction optimization.
5. **Consistency Check:**
   - Ensure the instruction intent can be fully achieved via the tool chain.
   - Confirm context provides all parameters required.
   - Delete tasks with misaligned instruction-context-tool chains.

Execution-Based-GT Annotation Human Guide

Human Annotation Guide
2. Execution-Based-GT Annotation
Core Objective
Annotate ground-truth (GT) data to support execution-based evaluation, enabling accurate assessment of the agent's ability to complete personalized tasks.
Annotation Categories & Requirements
GT is formatted as a list of checkpoints, with four checkpoint types:

1. **Generic Search:**
   - Definition: Web searches using non-personalized tools (e.g., map search, medical data query).
   - Annotation Requirement: Provide the exact GT value (e.g., search results, calculated data).
2. **Personalized Search:**

- Definition: Search operations using personalized tools (e.g., viewing calendar, accessing notes, checking social media posts).
- Annotation Requirement: Provide the exact GT value (e.g., calendar event details, note content).

3. **Operate:**
   - Definition: State-altering operations using personalized tools (Create/Update/Delete), with changes reflected in the context.
   - Annotation Requirements:
     - `tool`: Specify the tool used for the operation.
     - `operate_type`: Mark as "create", "delete", "modify", or "other".
     - `summary`: Describe the operation in natural language.
     - `context`: Provide the ID of the context affected by the operation.
4. **Other:**
   - Definition: Tasks that fall outside the above categories.
   - Annotation Requirement: Fall back to Log-based Evaluation, with no additional GT value required.

Annotation Principles

- Ensure GT values are accurate and reproducible.
- Align checkpoints with the tool chain sequence.
- Maintain consistency between GT annotations and the environment context.

D. Prompt Templates for Executable Code-Based Tool Simulation

D.1. Prompt Template for Dynamic Context Handler Generation

Dynamic Context Handler Generation Prompt

System Message
You are a Python code generator for dynamic context handler with nested dictionary structure. Generate a Python module that provides CRUD operations for dynamic context management.

User Message
CONTEXT STRUCTURE:

- **Context format:** `Dict[str, Dict[str, Any]]`.
- **Context file name:** {context_file_name}
- **Entity paths:** {entity_paths JSON}
- **ID fields:** {id_fields JSON}
- **Parent relations:** {parent_relations JSON}

CONTEXT SAMPLE (Structure Only):
{context_sample JSON}

REQUIREMENTS:
1. Module Functions (Must implement all):

- `load_context(path)`: Dict.
- `save_context(path, data)`: Save context to JSON with indentation.
- `get_entity_by_path(data, path, id)`: Retrieve specific entities.
- `list_entities_by_path(data, path, filters, limit)`: Support wildcard `[*]` traversal.
- `create_entity_by_path(data, path, entity, id)`: Validate parent existence before creation.
- `update_entity_by_path(...)` / `delete_entity_by_path(...)`: Standard CRUD.

2. Path Parsing Rules (CRITICAL):

- Must handle IDs containing dots (e.g., `user.name@domain.com`).
- **Rule:** Parse brackets `[...]` FIRST to extract selectors, THEN split by `.` outside brackets.
- *Example:* `calendars[user.name].events` → Token 1: `calendars` / `user.name`, Token 2: `events`.

3. ID Generation & Filtering:

- Analyze context structure to generate appropriate IDs (e.g., UUID for events, email-like for calendars).
- Apply filters for wildcard paths; use path-IDs as strict filters for specific paths.

4. Implementation Details (Parse Logic): You must implement the `parse_path` function following this logic to ensure dot-containing IDs are handled correctly:

```
def parse_path(path: str) -> List[PathToken]:
    tokens = []
    i = 0
    while i < len(path):
        if path[i] == '.':
            i += 1; continue
        # Logic to find segment end, handling nested brackets
        # ... (Implementation details as specified)
        # Extract segment name and selector
        tokens.append(PathToken(name=name, selector=sel))
    return tokens
```

OUTPUT RULES:

- Generate ONLY the Python code.
- No markdown formatting, no explanations.
- The code must be ready to execute.

D.2. Prompt Template for Simulation Kernel Generation

Simulation Kernel Generation Prompt

System Message
You are a Python code generator for API tool simulation with dynamic context. Generate a Python function that simulates the behavior of a real API tool using dynamic context operations.

User Message
TOOL INFORMATION:

- **Tool name:** {tool_name}
- **MCP name:** {mcp_name}
- **Description:** {description_short}
- **Input schema:** {input_schema_str}

{success_examples_text} {error_examples_text}

DYNAMIC CONTEXT INFORMATION:

- **Context structure:** Dict (single user) or List (multi-user).
- **Context file:** {context_file_name} located at `Path(__file__).parent.parent`.
- **Entity paths:** {entity_paths JSON}
- **ID fields:** {id_fields JSON}
- **Parent relations:** {parent_relations JSON}
- **Current entity:** Type={entity_type}, Path={entity_path}
- **Context Selection:** Read `CONTEXT_ID` from env vars ("all" or specific ID).

CRITICAL REQUIREMENTS:
1. Function Structure:

- Function name MUST be: `def analyze_response_patterns(parameters_used):`
- Return format: {`"success": bool, "error": str|None, "result": dict|None`}

2. Dynamic Context Handler Integration:

- **Import:** `from dynamic_context_handler import load_context, save_context, get/list/create/update/delete_entity_by_path`
- **Loading:** Load context at start. Select specific context based on `os.environ.get("CONTEXT_ID")`.
- **Persisting:** After ANY modification (Create/Update/Delete), you MUST call `save_context`.

3. Tool Operation Logic:

- **List Tools:** Use `list_entities_by_path`. Validate references first. Use large limit for pagination.
- **Create Tools:** Use `create_entity_by_path`. Validate parent entities exist (e.g., calendar_id).
- **Update/Delete Tools:** Validate entity existence before modification.

4. Input & Reference Validation (CRITICAL):

- Validate input format (required fields, types).
- **Reference Check:** Use `get_entity_by_path` to verify that referenced IDs (calendar IDs, folder IDs) actually exist in the current context context.
- Handle special keywords like "primary" by resolving them against the context.

5. Response Format:

- Success: Match the tool's expected JSON structure (from examples).
- Failure: Return specific error messages in the error field.

OUTPUT RULES:

- Generate ONLY the Python code.
- The code must be ready to execute without markdown formatting.