PLOT: COMPILING POLICY DOCUMENTS INTO DECISION TOOLS FOR LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM agents must follow domain-specific policies when using tools to automate business workflows. Beyond retrieving policy text, they must combine rules across documents, evaluate conditions and exceptions against the current task state, and decide which actions to take and how to execute them through tools. Repeating this interpretation during execution and across tasks can lead to missed requirements and inconsistent decisions. We introduce PLOT, which enhances agent actions with reusable decision tools constructed on demand. For each request, PLOT explicitly declares the required policy-dependent decisions, such as eligibility or permissible amounts, and resolves them by retrieving previously constructed decision tools or building new ones grounded in relevant policy documents. Each tool takes task-specific facts as inputs and applies reusable policy logic to compute a policy-dependent decision. Tools that pass admission checks on source references and code structure enter a decision tool library, and a separate execution agent calls them to complete the task request. We evaluate PLOT on τ-Knowledge and AutomationBench using GPT-5.2 and DeepSeek-V4-Flash. PLOT outperforms the strongest baseline in every setting, improving pass^1 on τ-Knowledge by 3.8% and partial credit on AutomationBench by 4.7%, on average across the models. It also uses 11.3% fewer tokens than ACE on average. Ablations on τ-Knowledge suggest benefits from slot declaration, admission checks, and cross-task decision tool reuse. Replacing executable decision tools with cited natural-language rules reduces task success from 36.1% to 20.6%.

1 INTRODUCTION

LLM agents are increasingly deployed to complete user requests in business workflows under domain-specific policy constraints. These policies specify eligibility criteria, permitted actions and arguments, and required procedures. Correctly following these policies is important in the real world. For example, acting before identity verification or exceeding an authorized limit can expose customers and businesses to unauthorized changes or financial loss, and overlooking eligibility conditions can cause an agent to deny a valid request or quote an incorrect fee. However, LLM agents still struggle to integrate policy knowledge with tool use throughout task execution (Shi et al., 2026; Shepard & Salimans, 2026).

Policy adherence in these tasks has three linked challenges. First, policy-dependent decisions often remain implicit. A user request may specify the desired actions without stating which eligibility criteria or preconditions the agent must evaluate. With the relevant conditions spread across several documents, the agent must identify both the required judgments and the sources that govern them. Second, applying policy requires reasoning over interacting rules and task-specific facts. An agent may evaluate each action in isolation and overlook how performing one action makes another unavailable under the policy. In Figure 1, for example, filing the dispute first creates a pending state that blocks the requested credit-limit increase. Third, agents may need to reconstruct the same policy logic across tasks. Textual guidance retained by methods such as Dynamic Cheatsheet (Suzgun et al., 2026) and ACE (Zhang et al., 2026) may leave the decision procedure to be derived again for new inputs, sometimes from summaries that are inappropriate when task-specific facts change or omit the conditions behind past outcomes. For example, a reminder to assess credit-limit eligibility does not specify how the amount cap and pending operations jointly determine the result.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

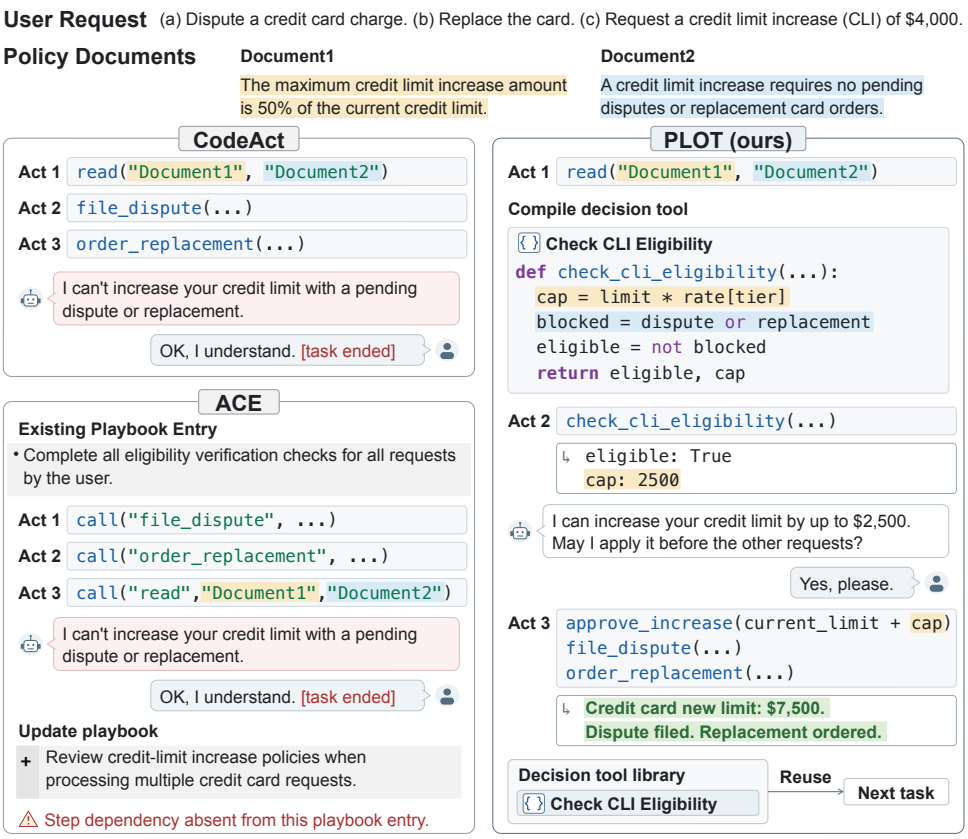

Figure 1: From policy documents to explicit, executable policy logic. For a credit card request, CodeAct and ACE assess the credit-limit increase (CLI) after their actions have created blocking state. PLOT combines document information into a parameterized decision tool. Task code calls it before acting, allowing the agent to arrange a permitted $2,500 increase first with customer consent. ACE's displayed playbook update omits the CLI dependency, while PLOT retains the policy logic for reuse with new inputs.

To address these challenges, we introduce PLOT (**P**olicy **Lo**gic as **T**ools), a framework that compiles task-relevant policy documents into executable decision tools. First, PLOT makes required policy-dependent decisions explicit through *slot declaration*. Each slot links a decision to potential policy documents, giving retrieval and compilation a concrete target. Second, **PLOT separates policy interpretation and tool compilation from task execution**. Guided by these slots, the compilation agent retrieves and interprets the governing policies and builds decision tools, subject to admission checks on source references and code structure. A separate execution agent calls the tools with current task-specific facts and uses the returned decisions, alongside supporting documents, to plan and carry out the request. Third, PLOT retains accepted tools and source references in a persistent decision tool library. Later tasks can retrieve applicable decision tools grounded in shared policy documents and evaluate new task-specific facts through tool calling without reconstructing the policy logic.

To test whether making policy-dependent decisions explicit, executable, and reusable improves task solving, we evaluate PLOT on two policy-adherence benchmarks, τ-Knowledge and AutomationBench, using GPT-5.2 and DeepSeek-V4-Flash. PLOT outperforms the strongest baseline in all four settings, with average gains across base LLMs of approximately 3.8% in τ-Knowledge pass$\hat{}$1 and 4.7% in AutomationBench partial credit. These gains come with 11.3% fewer tokens than ACE on average. Ablation studies with τ-Knowledge show lower task success when slot declaration, admission checks, or cross-task decision tool reuse is removed. Replacing executable tools with cited textual rules produces the largest drop from 36.1% to 20.6%, suggesting benefits from executable

decision procedures and their associated checks. Manual review finds that 96.2% of decision tools in the final library have substantially correct core logic within their intended scope.

Our key contributions are: (1) an executable representation of policy documents that separates reusable policy logic from task execution; (2) PLOT, a framework that compiles policy documents into decision tools on demand for agents to invoke and reuse across tasks; and (3) empirical evaluation that validates the effectiveness and efficiency of PLOT over baselines and design choices for policy compilation and reuse.

2 Related Work

2.1 Policy Adherence

Prior work supports policy adherence by incorporating relevant rules into the agent's context or model parameters (Mishra et al., 2025; Liu et al., 2026). Another line translates policies into executable guards for checking agent actions (Zwerdling et al., 2025; Xiang et al., 2025; Miculicich et al., 2025). PLOT shares the use of executable policy representations, but organizes compilation around declared policy slots that identify the decisions required by the current task. The resulting tools evaluate policy conditions on task-specific inputs and provide judgments that the execution agent uses when selecting and ordering actions.

2.2 Code-Based Agents

Code-based agents use executable programs to perform computations and interact with tools. Prior work separates tool construction from tool use, enabling language models to generate reusable functions for task solving (Qian et al., 2023; Cai et al., 2024). CodeAct (Wang et al., 2024) uses Python as a unified action representation, allowing agents to compose tool calls and process their outputs. Building on these approaches, PLOT organizes tool construction around declared policy slots and compiles the governing rules from policy documents into functions that guide task execution.

2.3 Agent Memory

Experience-based memory retrieves past task experiences and extracted insights to guide subsequent decisions (Zhao et al., 2024). Procedural memory abstracts execution trajectories into reusable workflows and instructions (Wang et al., 2025; Fang et al., 2026). Context adaptation maintains and revises accumulated knowledge in the model's input without updating its parameters (Suzgun et al., 2026). Specifically, ACE (Zhang et al., 2026) organizes this process around an evolving playbook updated through generation, reflection, and curation. Our library accumulates policy logic compiled from policy documents. It stores parameterized decision tools together with their supporting policy documents, allowing subsequent tasks to evaluate the same rules on new inputs when those sources remain applicable.

3 PLOT

Figure 2 provides an overview of PLOT. First, slot declaration makes the required policy-dependent decisions explicit and guides policy retrieval and compilation. Second, the compilation agent combines policy rules into decision tools before a separate execution agent uses them to plan and carry out actions. Third, a persistent decision tool library lets later tasks reuse the policy logic with new inputs. Prompt templates for τ-Knowledge are provided in Appendix F.

3.1 Task Definition

A task $x = (q, \mathcal{E}, \mathcal{D})$ consists of a user request q, an environment $\mathcal{E}$ with tools for observing and changing its state, and policy documents $\mathcal{D}$. The agent must resolve the request while following the applicable policies. For an interaction trajectory τ, including observations, tool calls, and user communication, this requires

$$\text{Resolve}(q, \tau; \mathcal{D}) = 1 \qquad \text{and} \qquad c(\tau) = 1 \quad \forall c \in \mathcal{C}_{\mathcal{D}}. \tag{1}$$

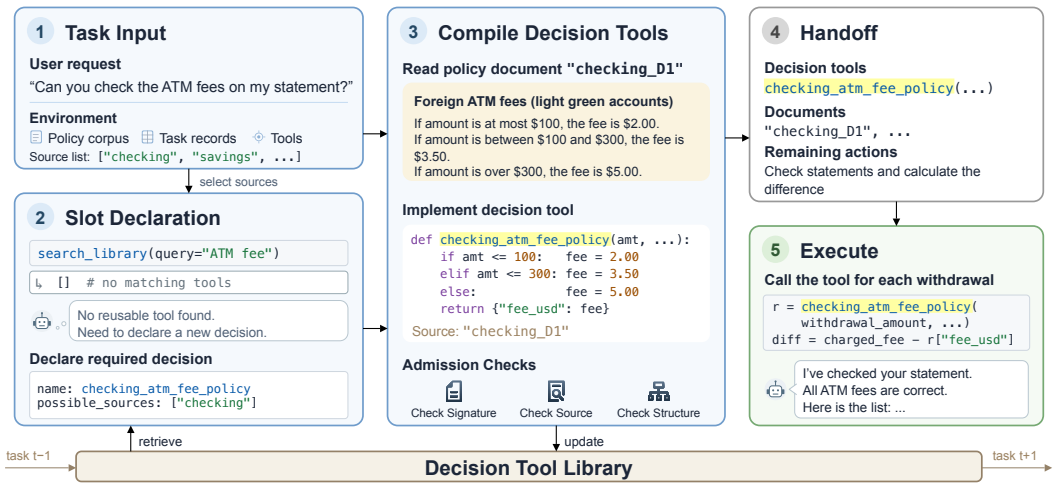

Figure 2: Overview of PLOT. The compilation agent retrieves existing decision tools from the decision tool library and declares policy slots for decisions that need new or revised implementations. It then compiles decision tools from policy documents and submits them to admission checks. A structured handoff provides the execution agent with the selected decision tools, supporting documents, and remaining work. Accepted decision tools stay in the decision tool library for reuse when their policy documents are shared across tasks.

Here, Resolve and each c are binary indicators. Resolving a request may mean completing an operation, explaining why it is unavailable, offering a permitted alternative, or escalating. The constraints $\mathcal{C}_{\mathcal{D}}$ govern which actions are allowed, their arguments, and their order; the relevant constraints may become clear only as the agent interacts with the environment. PLOT turns selected policy-dependent decisions into callable decision tools to help the agent satisfy these requirements.

3.2 SLOT DECLARATION

User requests often leave the required policy-dependent decisions unstated. The compilation agent therefore first identifies the policy-dependent decisions that need to be made before taking action. For example, checking the ATM fees in Figure 2 requires determining the correct fee for each withdrawal. Using these decisions, the compilation agent queries the existing decision tool library $\mathcal{L}$ and checks the retrieved tools' policy applicability before reuse (Section 3.5). For each decision requiring a new or revised tool implementation, it declares a *policy slot*. Each slot specifies a function signature, a description of the decision, and candidate sources selected from a list provided in the agent's prompt. These source names indicate where to search, such as a knowledge-base area or an application. The slots thus guide the next stage by specifying what to implement and where to look for the relevant policies.

3.3 COMPILING DECISION TOOLS FROM POLICY DOCUMENTS

Different policy documents may jointly determine a decision. For each unimplemented slot s, the compilation agent searches for and reads the relevant policy documents. It combines their conditions, exceptions, and calculations in a function f_s that takes task-specific facts as inputs, and cites the supporting documents. In Figure 2, the agent reads `checking_D1` and compiles its fee schedule into the declared tool. Preparing these tools before execution lets the execution agent use the results when choosing actions and their order. Policy logic that needs no calculation or conditional check, such as fixed instructions, can be passed directly to the execution agent as a related policy document.

Each implemented decision tool goes through three admission checks before being stored in the decision tool library. First, PLOT checks that each cited source exists and has been read during the

current task. Second, f_s must not have any unused inputs and it must not output a constant value. Finally, PLOT checks that the function name and input parameters of f_s match the signature declared in slot s. If the admission check fails, the rejection message will be sent to the agent, so the agent can further search policy documents or revise the code. Accepted decision tools are stored or updated in the decision tool library $\mathcal{L}$.

The compilation agent continues this process until all declared slots have been implemented or it decides that no more tools are needed. To save steps for execution, compilation also ends after using half of the task's step budget. The compilation agent then passes the conversation, available decision tools, and relevant policy documents to the execution agent. A handoff note links each remaining action to the tools and documents it needs and specifies the required inputs and action order.

3.4 EXECUTION

The execution agent continues the conversation history, with the selected decision tools ready to call and their source policy documents available for reference. It calls the decision tools with current task-specific facts and uses their outputs, together with the source policies, to choose and carry out actions. For the ATM-fee task, it calls the compiled tool with each withdrawal's amount, compares the returned fee with the fee charged on the statement, and reports the results to the user. If a new decision is not covered by the selected tools, the agent reads the relevant policy documents and handles it in its task scripts.

3.5 CROSS-TASK DECISION TOOL REUSE

PLOT retains accepted decision tools in $\mathcal{L}$ for later tasks with changing facts. When an existing tool covers the required policy conditions, the agent calls it with the current input values. If the decision requires facts or conditions that the tool does not handle, the compilation agent revises its logic and, where needed, its inputs using the relevant policy documents. The revised tool goes through admission checks before the execution agent calls it with the current facts.

For a later task, the agent retrieves tools for the required decisions and checks whether their policies apply. Tools based on shared policies, such as the banking policies in τ-Knowledge, can be called directly with the new task's facts. Tools based on a previous task's emails or spreadsheets, as in AutomationBench, are returned only as descriptions. The agent then reads the current task's documents to build the required tool. Task-specific variables and conversations are cleared between tasks.

Each library entry stores the tool's implementation, description, and source references; revising a tool updates its implementation and references. Before handoff, PLOT checks the tools retrieved or compiled for the current task for duplicate code. If two tools have the same code under different names, it keeps one and combines their source references. It may also ask the model to combine overlapping rules into one tool. Tools called by other library entries are kept so those calls remain valid.

4 EXPERIMENT

4.1 EXPERIMENTAL SETTINGS

Benchmarks. We conduct experiments on two benchmarks for policy adherence in tool use, τ-Knowledge (Shi et al., 2026) and AutomationBench (Shepard & Salimans, 2026). τ-Knowledge consists of 97 banking customer-support tasks in which agents interact with simulated users, retrieve policies from a corpus of 698 documents, and perform account operations, such as filing transaction disputes and opening accounts. AutomationBench requires agents to complete workflows across 47 simulated applications. It provides environment tool documentation that specifies how to perform operations such as searching, reading, and modifying data in these applications. Each task has unique business policies in applications such as emails and spreadsheets. This benchmark covers six domains: sales, marketing, operations, support, finance, and human resources. To balance domain coverage and the evaluation budget, we use a fixed subset of 120 public tasks, sampling 20 from each of the domains with seed 0 (Appendix A.2). The original benchmark uses BM25 to search tool

documentation. To align with document access supported by τ-Knowledge, we allow all methods to search and read documentation with terminal commands (e.g., `grep`, `cat`).

Comparison baselines. We compare PLOT with three baselines: (1) Vanilla, a standard tool-calling agent that retrieves policy documents, interprets their rules in context, and invokes benchmark-provided environment tools; (2) CodeAct (Wang et al., 2024), which generates executable code to compose calls to those environment tools; and (3) ACE (Zhang et al., 2026), which uses a natural-language playbook to guide the agent's selection and invocation of environment tools. After each task is completed, ACE updates the playbook through reflection and curation for use in subsequent tasks.

Models and implementation. We evaluate all methods with two base LLMs: GPT-5.2 (OpenAI, 2025) and DeepSeek-V4-Flash (Xu et al., 2026), using reasoning efforts of `high` and `max`, respectively. Following the default settings of the two benchmarks, we use a budget of 200 steps on τ-Knowledge and 50 steps on AutomationBench. For PLOT, compilation and execution use the same base LLM and share this budget. Its decision tool library starts empty in each run and accumulates across sequential tasks without task-success feedback. On τ-Knowledge, we report three runs with a fixed task-order seed for each run. ACE and PLOT use the same seed in each corresponding run, ensuring identical task orders. For GPT-5.2 Vanilla on τ-Knowledge, we reuse the benchmark's officially released results. On AutomationBench, we conduct a single run. Model snapshots, decoding configurations, and additional implementation details are provided in Appendix A.

Evaluation metrics. On τ-Knowledge, we follow the origin paper to report pass^k (Yao et al., 2025) for $k \in \{1, 2, 3\}$, calculated from three runs. This metric measures consistent task completion: for each task, we compute the fraction of k-run combinations in which all runs succeed, then average across tasks. Thus, pass^1 measures average task success, while pass^3 measures success across all three runs. On AutomationBench, we report *strict completion*, the percentage of tasks satisfying all evaluation assertions, and *partial credit*, the average percentage of satisfied assertions per task. To measure end-to-end efficiency, we also report average token usage and cost per task.

4.2 MAIN RESULTS

Table 1: Performance on τ-Knowledge and AutomationBench. * denotes results taken from the official τ-Knowledge run trajectory release.

	τ-Knowledge			AutomationBench	
Method	pass^1	pass^2	pass^3	Strict	Partial
GPT-5.2					
Vanilla	30.9*	23.0*	19.6*	14.2	53.7
CodeAct	30.9$_{+0.0}$	22.0$_{-1.0}$	18.6$_{-1.0}$	10.0$_{-4.2}$	45.7$_{-8.0}$
ACE	22.3$_{-8.6}$	11.7$_{-11.3}$	7.2$_{-12.4}$	18.3$_{+4.2}$	50.8$_{-2.9}$
PLOT	**35.1**$_{+4.1}$	**27.1**$_{+4.1}$	**22.7**$_{+3.1}$	**20.0**$_{+5.8}$	**61.6**$_{+7.9}$
DeepSeek-V4-Flash					
Vanilla	32.6	25.1	21.6	10.8	47.1
CodeAct	33.3$_{+0.7}$	23.4$_{-1.7}$	18.6$_{-3.1}$	10.8$_{+0.0}$	48.0$_{+0.9}$
ACE	24.1$_{-8.6}$	12.4$_{-12.7}$	7.2$_{-14.4}$	17.5$_{+6.7}$	56.0$_{+8.9}$
PLOT	**36.8**$_{+4.1}$	**28.9**$_{+3.8}$	**25.8**$_{+4.1}$	**18.3**$_{+7.5}$	**57.4**$_{+10.3}$

Performance Analysis Table 1 shows that PLOT achieves the highest task success across both benchmarks and base LLMs. For example, on τ-Knowledge, it improves pass^1 over the strongest baseline by 4.1% with GPT-5.2 and by 3.4% with DeepSeek-V4-Flash. The advantage persists under pass^3: PLOT achieves 22.7% and 25.8%, compared with the best baseline scores of 19.6% and 21.6%. On AutomationBench, PLOT achieves the highest strict completion and partial credit across both base LLMs. Its partial-credit gains over the strongest baseline are 7.9% and 1.4% with GPT-5.2 and DeepSeek-V4-Flash, respectively. By comparison, with GPT-5.2, ACE improves strict completion over Vanilla from 14.2% to 18.3% but reduces partial credit from 53.7% to 50.8%.

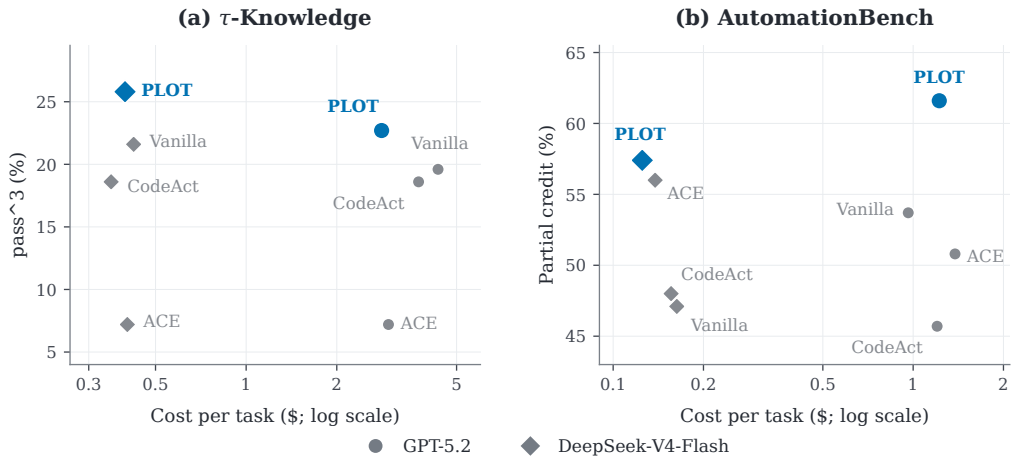

Figure 3: Performance–cost trade-offs on τ-Knowledge and AutomationBench.

The baseline results show that CodeAct provides no consistent improvement over Vanilla: an agent can retrieve the relevant policy and execute valid tool calls while overlooking a condition that determines their permissible order (Appendix E.1). This illustrates the difficulty of translating interacting natural-language rules into executable action sequences that respect their conditional dependencies. On the other hand, ACE exposes a complementary limitation: retained textual guidance can be applied outside its original scope (Appendix E.2). This suggests that summarizing prior task experience may preserve an action pattern while extending its applicability beyond the conditions specified by the underlying policy.

Cost Analysis We report the cost per task of PLOT and baselines in Figure 3. PLOT achieves higher performance at a lower cost than the strongest baseline for the plotted metric in three of the four settings. On τ-Knowledge, it improves pass^3 over Vanilla, while reducing cost per task by 35.1% and 6.4% with GPT-5.2 and DeepSeek-V4-Flash, respectively. On AutomationBench with DeepSeek-V4-Flash, PLOT achieves the highest partial credit (57.4%) at the lowest cost ($0.125 per task), improving over ACE's 56.0% while reducing cost by 9.4%. With GPT-5.2, it achieves the highest partial credit at a cost comparable to the other baselines. PLOT uses 11.3% fewer tokens per task than ACE on average across the four settings. Detailed token usage and cost for all methods are reported in Appendix A.5.

4.3 DECISION TOOL RELIABILITY

Table 2: Decision tool quality (a) and failure type counts (b) by base LLM. Counts aggregate benchmarks and runs.

(a) Tool quality (%)

	τ-Knowledge		AutomationBench	
Metric	GPT	DS	GPT	DS
Slot Completion	82.9	92.7	75.5	80.9
Citation Completeness	70.6	82.1	65.4	59.2
Logic Correctness	96.7	92.1	99.4	94.9

GPT: GPT-5.2; DS: DeepSeek-V4-Flash.

(b) Failure type counts

Primary defect	GPT	DS
Policy values / mappings	4	12
Missing constraints / candidates	5	5
Misinterpreted conditions	2	3
Units / formulas	1	4
Dependencies / execution	1	5
Total	13	29

Decision tools are compiled by an LLM and may therefore contain functional errors, as prior work has shown for LLM-generated code (Liu et al., 2023). To assess their reliability, Table 2(a) reports three metrics: *slot completion*, the fraction of declared slots with an implementation; *citation completeness*, the fraction of tools for which every extracted policy constant appears in the cited sources,

excluding tools without such constants; and *logic correctness*, the fraction of all final-library functions whose main decision rule, calculation, or workflow is correct. We manually review the 1,172 final-library functions against their policy documents and classify each incorrect function by its failure type.

Slot completion and citations. Across the four settings, slot completion ranges from 75.5% to 92.7%, and citation completeness ranges from 59.2% to 82.1%. On τ-Knowledge, 94.5% and 93.7% of individual constants are supported for GPT-5.2 and DeepSeek-V4-Flash, respectively. Most citation failures arise from incomplete source lists or derived values (e.g., unit conversions).

Logic correctness. Logic correctness ranges from 92.1% to 99.4%. Overall, 1,128 of 1,172 functions (96.2%) pass, 42 are incorrect, and two are uncertain because their policy documents conflict. Table 2(b) breaks down the 42 incorrect functions by base LLM, pooling both benchmarks and all runs. Comparing the two evaluated models, we observe that GPT-5.2 has no more failures than DeepSeek-V4-Flash in any failure category. Incorrect policy values or product mappings (16/42) and missing constraints or candidates (10/42) account for most errors. Of the 38 incorrect functions with extracted policy constants, 25 still pass citation completeness. For example, both base LLMs produce payment functions that accept "Dept Head: Approved; CFO: Denied" by checking for "CFO" and "Approved" separately, despite a policy requiring CFO approval. Correctly cited constants therefore do not ensure correct decisions. Appendix C provides examples and an observed case where a missing policy condition leads to an incorrect spreadsheet update.

4.4 Mechanism Analysis

Slot Declaration Across both benchmarks, agents declare multiple policy slots per task on average, with some slots remaining unimplemented. Declaring these slots makes the scope of policy compilation explicit and allows missing implementations to be tracked before execution. Full results are in Appendix A.4.

Admission Checks Across the six τ-Knowledge runs, most admission rejections involve citations to sources not yet observed by the agent, with additional rejections for loading failures and unused parameters. These checks enforce observation of cited policy documents and basic implementation requirements before compiled tools enter the reusable library. Full results are in Appendix A.4.

Table 3: Ablations on the same 97 τ-Knowledge tasks with DeepSeek-V4-Flash. Tokens are per-task averages over agent-side calls in scored trajectories; subscripts give changes from the full method.

Variant	pass^1	Tok. (M)
PLOT (ours)	**36.1**	2.055
w/o slot declaration	$30.9_{-5.2}$	$2.291_{+11.5\%}$
w/o admission checks	$32.0_{-4.1}$	$2.335_{+13.6\%}$
w/o cross-task decision tool library	$29.9_{-6.2}$	$2.112_{+2.8\%}$
$\text{PLOT}_{\text{text}}$	$20.6_{-15.5}$	$\mathbf{2.025}_{-1.5\%}$

Ablation Study Table 3 reports ablation studies on τ-Knowledge with DeepSeek-V4-Flash, examining how individual design choices of PLOT contribute to task success. We evaluate four variants: (1) *w/o slot declaration*, which removes upfront declaration of policy-dependent decisions and lets the compilation agent submit implementations directly; (2) *w/o admission checks*, which removes citation-grounding and function-shape checks before admitting tools into the library; (3) *w/o cross-task decision tool library*, which clears the library between tasks, requiring decision tools to be acquired anew for each task; and (4) $\text{PLOT}_{\text{text}}$, which replaces executable decision tools with reusable, cited natural-language rules that the execution agent applies through reasoning.

The result shows that all four variants achieve lower task success than the full method, suggesting that each evaluated design choice contributes to performance. We observe that the largest performance drop occurs for $\text{PLOT}_{\text{text}}$, where pass^1 falls from 36.1% to 20.6%, suggesting that executable decision procedures and their associated checks provide benefits beyond retaining policy knowledge as reusable text.

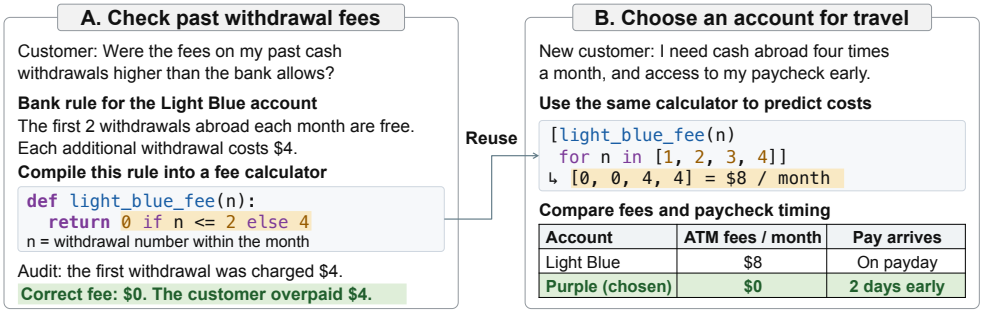

Figure 4: Reuse across task goals in the DeepSeek-V4-Flash seed-2 run. **(A)** A foreign ATM fee rule is compiled while auditing past charges (task 074). **(B)** The same tool predicts costs for a travel account choice (task 076). Requests and code are condensed; the table shows selected accounts and foreign ATM fees. Monthly account fees are separate.

Decision Tool Reuse Analysis PLOT retains policy logic as functions that later tasks can apply to their own facts. Across six τ-Knowledge runs, previously compiled tools are invoked in 10–31 of the 97 tasks per run. Figure 4 illustrates a use case of decision tool compilation and reuse across task goals. In this case, the agent compiles a foreign ATM fee calculator while investigating past overcharges. For the Light Blue account, the first two withdrawals abroad each month are free, and each additional withdrawal costs $4. In a later task, a customer plans four withdrawals abroad per month and needs early access to their paycheck. Invoking the previously compiled calculator for withdrawals 1–4 returns fees of $0, $0, $4, and $4, totaling $8 per month. The agent combines these outputs with other accounts' fee rules and paycheck timing rules, then opens a Purple Account with zero foreign ATM fees and pay available two days early. A rule acquired for retrospective auditing thus becomes a cost model for prospective account selection. Appendix D provides reuse statistics and trace analyses, including unsuccessful cases.

5 DISCUSSION

Dataset-dependent tool reuse. Decision tools can be reused when later tasks require the same policy-dependent decisions under shared policy documents. τ-Knowledge tasks share a banking policy corpus, whereas AutomationBench tasks involve distinct workflows and task-specific policy documents. This difference is consistent with the observed cross-task decision tool reuse in τ-Knowledge and its absence in AutomationBench (Appendix D).

Scope and remaining failure modes. PLOT is best suited to decisions whose conditions and calculations can be evaluated using available task-specific facts. Ambiguous instructions and discretionary judgments still require source interpretation. The citation gaps and implementation errors in previous analyses (Section 4.3) highlight the need for more complete source references and semantic validation beyond admission checks. Even with correct tools, the agent may supply incorrect inputs, overlook other requirements, or check prerequisites too late. For example, in the identity-gate audit, verification succeeds in all 12 later tasks, but only three succeed overall. In the account-opening case, the agent checks savings eligibility after closing the required checking account (Appendix D.2).

6 CONCLUSION

We introduced PLOT, which compiles policy documents into reusable decision tools for LLM agents. The compilation agent identifies required policy-dependent decisions and retrieves or builds tools to resolve them; a separate execution agent calls these tools with current task-specific facts to guide its actions. Across τ-Knowledge and AutomationBench with two base LLMs, PLOT outperforms all evaluated baselines while using 11.3% fewer tokens than ACE on average. Ablations on τ-Knowledge show benefits from slot declaration, admission checks, and cross-task decision tool reuse, with the largest performance drop when executable tools are replaced by cited textual rules.

AI USE STATEMENT

In this work, we used generative AI tools for providing feedback on research methodology or experiments. We have not used generative AI tools for proposing or refining hypotheses, designing research methodology or experiments, interpreting results, or implementing methods, and generating synthetic datasets, helping develop theoretical models or conceptual frameworks, formulating mathematical claims, providing critical ingredients for proving mathematical claims, assisting in the writing of proofs, assisting with translation, cleaning and reformatting datasets, and supporting qualitative and thematic data analysis are not applicable to this work. Additionally, we used generative AI tools for creating or editing software code and editing the research paper to improve readability. We have reviewed all AI-assisted work. In particular, we checked AI-generated or AI-edited code for correctness. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

Our experiments use simulated banking and business-workflow environments. While PLOT aims to improve policy adherence, errors in compiled decision tools may propagate across tasks, and successful benchmark performance does not establish safety in real-world deployment. Applying such tools to consequential decisions would require independent validation, appropriate access controls, and human oversight.

REPRODUCIBILITY STATEMENT

Appendix A provides model versions, decoding settings, evaluation protocols, method defaults, cost accounting, and ablation configurations. Appendix F presents condensed compilation, execution, handoff, and merge prompt templates for τ-Knowledge. Source code and reproduction instructions are available at `https://anonymous.4open.science/r/Plot-5A9D`.

REFERENCES

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=qV83K9d5WB.

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. Memp: Exploring agent procedural memory. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 17490–17502, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.866. URL https://aclanthology.org/2026.findings-acl.866/.

Jiateng Liu, Zhenhailong Wang, Xiaojiang Huang, Yingjie Li, Xiang Li, Chenlei Guo, Xing Fan, Ruhi Sarikaya, and Heng Ji. Analyzing and internalizing complex policy documents for LLM agents. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16814–16851, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.767. URL https://aclanthology.org/2026.acl-long.767/.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

Lesly Miculicich, Mihir Parmar, Hamid Palangi, Krishnamurthy Dj Dvijotham, Mirko Montanari, Tomas Pfister, and Long T Le. Veriguard: Enhancing llm agent safety via verified code generation. *arXiv preprint arXiv:2510.05156*, 2025.

Venkatesh Mishra, Amir Saeidi, Satyam Raj, Mutsumi Nakamura, Gaowen Liu, Ali Payani, Jayanth Srinivasa, and Chitta Baral. How can input reformulation improve tool usage accuracy in a complex dynamic environment? a study on tau-bench. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 22949–22972, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-335-7. doi: 10.18653/v1/2025.findings-emnlp.1250. URL https://aclanthology.org/2025.findings-emnlp.1250/.

OpenAI. Update to GPT-5 system card: GPT-5.2. System card, December 2025. URL https://openai.com/index/gpt-5-system-card-update-gpt-5-2/.

Cheng Qian, Chi Han, Yi Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. CREATOR: Tool creation for disentangling abstract and concrete reasoning of large language models. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 6922–6939, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.462. URL https://aclanthology.org/2023.findings-emnlp.462/.

Daniel Shepard and Robin Salimans. Automationbench. *arXiv preprint arXiv:2604.18934*, 2026.

Quan Shi, Alexandra Zytek, Pedram Razavi, Karthik R Narasimhan, and Victor Barres. τ-knowledge: Evaluating conversational agents over unstructured knowledge. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=XHZK5abtw2.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7080–7106, 2026.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*, 2024. URL https://openreview.net/forum?id=8oJyuXfrPv.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=NTAhi2JEEE.

Zhen Xiang, Linzhi Zheng, Yanjie Li, Junyuan Hong, Qinbin Li, Han Xie, Jiawei Zhang, Zidi Xiong, Chulin Xie, Carl Yang, Dawn Song, and Bo Li. Guardagent: Safeguard LLM agents via knowledge-enabled reasoning. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=2nBcjCZrrP.

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik R Narasimhan. {τ}-bench: A benchmark for \underline{T}ool-\underline{A}gent-\underline{U}ser interaction in real-world domains. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=roNSXZpUDN.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, volume 2026, pp. 86069–86100, 2026.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024.

Naama Zwerdling, David Boaz, Ella Rabinovich, Guy Uziel, David Amid, and Ateret Anaby Tavor. Towards enforcing company policy adherence in agentic workflows. In Saloni Potdar, Lina Rojas-Barahona, and Sebastien Montella (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pp. 595–606, Suzhou (China), November 2025. Association for Computational Linguistics. ISBN 979-8-89176-333-3. doi: 10.18653/v1/2025.emnlp-industry.41. URL https://aclanthology.org/2025.emnlp-industry.41/.

A EXPERIMENTAL AND IMPLEMENTATION DETAILS

This appendix summarizes the experimental settings, method defaults, and ablation configurations for PLOT. Full-method prompt templates for τ-Knowledge are provided in Appendix F.

A.1 MODEL VERSIONS AND DECODING

Compilation and execution use the same base LLM: the December 11, 2025 snapshot of GPT-5.2 with reasoning effort `high`, or the April 23, 2026 build of DeepSeek-V4-Flash with reasoning effort `max`. On τ-Knowledge, the agent uses temperature 0; the simulated customer uses the same GPT-5.2 snapshot with reasoning effort `low` and temperature 0. AutomationBench retains the provider's default temperature. Other unspecified decoding parameters follow the harness/provider defaults.

A.2 EVALUATION SETTINGS

On τ-Knowledge, each PLOT run evaluates all 97 tasks once. The three runs use task-order seeds 0, 1, and 2, with runner seed 300 and one trial per task. AutomationBench v1.0.1 uses a fixed stratified subset of 120 tasks, with 20 from each of six domains. Sampling and domain order use seed 0; tasks are grouped by domain, with one rollout per task.

All PLOT runs execute tasks sequentially with concurrency one. The decision tool library starts empty for each run and carries forward across tasks. Table 4 lists the episode budgets, which include both compilation and execution. For AutomationBench, one step denotes one agent turn.

A.3 DEFAULT METHOD HYPERPARAMETERS

Table 4 summarizes the implementation defaults for the full method. Ablation-specific changes are described in Appendix A.6.

Table 4: Default PLOT hyperparameters. The compilation threshold is measured in each benchmark's episode-budget units.

Setting	Value
Episode budget	200 steps (τ-Knowledge); 50 steps (AutomationBench)
Compilation threshold	Half the episode budget
AutomationBench handoff grace	2 steps after the compilation threshold
Library search results	At most 3 entries
Retrieval filters	Score > 0; weighted query coverage > 0.05
BM25 parameters	$k_1 = 1.5$, $b = 0.75$
Source observation (τ-Knowledge)	At least 50% of unique 8-word fingerprints
Merge candidate threshold	IDF-weighted containment 0.5
Merge attempt limit	At most 3 candidate pairs per task

A.4 DECISION TOOL IMPLEMENTATION

Slot declaration and completion. Across the three τ-Knowledge runs of 97 tasks per model, GPT-5.2 and DeepSeek-V4-Flash declare 807 and 627 policy slots, respectively, or 2.77 and 2.15 per task; 669 and 581 are covered by an implementation. On AutomationBench, they declare 469 and 220 slots over 120 tasks each, or 3.91 and 1.83 per task, with 354 and 178 covered. Slot completion records whether an implementation is available for a declared decision; it does not establish that all policy-dependent decisions needed by the task were declared.

Source observation and admission. On τ-Knowledge, document observation is determined by accumulating text fingerprints from environment outputs; AutomationBench registers sources returned by resource reads. Admission requires a declared Python function that parses and loads,

validates supplied citations against observed sources, and screens for constant verdicts, unused parameters, and disallowed definition-time operations. Empty citation lists are accepted, although the prompt requests all governing policy documents. These checks do not establish citation completeness or semantic correctness; the optional numeric worked-example check is disabled.

Observed submission outcomes. We reconstruct every recorded `implement_slot` response from the complete saved compilation logs of the six τ-Knowledge runs. An acceptance requires an affirmative library-admission or replacement response; a rejection containing the words "not in the library" is not an acceptance. Table 5 counts submissions, including retries. Rejection categories follow the returned error, which identifies the first failed check rather than all possible defects. A repair is a later affirmative admission of the same function name within the same task and run after at least one rejection. These are descriptive process counts, not independent error rates or semantic-correctness judgments.

Table 5: Recorded implementation submissions and rejection outcomes on τ-Knowledge, pooled over three runs per base LLM. Retries count separately. The last two rows count distinct (run, task, function-name) combinations.

Outcome	**GPT-5.2**	**DeepSeek-V4-Flash**
All submissions	1,057	497
Accepted submissions	646	424
Rejected submissions	411	73
Source not yet observed	314	49
Loading failure	90	0
Unused parameters	2	23
Code format or parsing	2	1
Undeclared name	2	0
Unknown source	1	0
Function names rejected within a task	183	51
Subsequently admitted	168	49

Retrieval and maintenance. Library search ranks function names and descriptions using BM25-style scoring with corpus-based IDF weights fixed at initialization. Weighted query coverage measures the fraction of query-token weight matched by an entry. Reuse follows the shared-source restriction in Section 3.5. At handoff, consolidation removes duplicate normalized function bodies while protecting dependency names. The τ-Knowledge adapter also supplies a model-assisted merger; AutomationBench disables it by default. Proposed merges must parse and load, but are not checked for behavioral equivalence.

Handoff. The handoff note selects decision tools and supporting documents and records remaining actions and their required inputs. Selected tools and dependencies are available to the execution agent; cited or explicitly selected observed documents cross in full, while other observed documents receive catalog entries. Compilation ends when all declared slots are implemented, the agent explicitly finishes compilation, or the budget threshold in Table 4 is reached. This threshold triggers a phase switch rather than an exact division of model calls.

A.5 Cost Accounting

Table 6 reports average token usage and cost per task, including all agent-side model calls such as PLOT's compilation and ACE's reflection and curation. We apply the same input/output list prices to all methods: \$1.75/\$14.00 per million tokens for GPT-5.2 and \$0.19/\$0.51 for DeepSeek-V4-Flash, without prompt-cache discounts.

Table 6: Average end-to-end tokens and cost per task.

	τ-Knowledge		AutomationBench	
Method	**Tok. (M)**	**Cost ($)**	**Tok. (M)**	**Cost ($)**
		GPT-5.2		
Vanilla	2.311	4.339	**0.413**	**0.963**
CodeAct	1.954	3.739	0.523	1.202
ACE	1.571	2.970	0.584	1.379
PLOT	**1.405**	**2.818**	0.469	1.222
		DeepSeek-V4-Flash		
Vanilla	2.171	0.423	0.843	0.163
CodeAct	**1.836**	**0.356**	0.805	0.156
ACE	2.086	0.403	0.709	0.138
PLOT	2.008	0.396	**0.629**	**0.125**

A.6 Ablation Configurations

All ablations use the full method's DeepSeek-V4-Flash configuration on τ-Knowledge with task-order seed 0 and the same evaluation settings and budget.

Component ablations. Without slot declaration, implementations are submitted directly and automatic handoff upon slot completion is disabled. Without admission checks, citation and function-shape checks are removed, while Python parsing, function-name validation, definition-time restrictions, and loading remain. Without the cross-task decision tool library, the library is cleared after each task.

Natural-language policy ablation. $\text{PLOT}_{\text{text}}$ replaces callable Python functions with cited natural-language rules. It retains slot declaration and completion tracking, retrieval, cross-task persistence, citation validation, and the two-phase handoff. Basic prose-format validation replaces code-dependent checks. Selected rules pass to execution as text for the agent to apply through reasoning; code actions remain available for environment calls and arithmetic. This variant therefore changes both policy representation and the checks tied to executable code. Token usage includes all agent-side calls in the 97 scored episodes, excluding interrupted or failed attempts without a scored episode.

B Compilation Artifacts

B.1 A Compiled Decision Tool

Figure 5 shows a decision tool compiled during one τ-Knowledge run with GPT-5.2 (seed 0), together with excerpts of its two cited policy documents. The card-tier taxonomy that the tool encodes is not stated as a standalone rule anywhere in the policy corpus: it appears inside a document about provisional credit limits for transaction disputes, and a document about replacement shipping fees corroborates part of the mapping. The compilation agent declared a slot for tier inference while handling a credit-limit-increase task at position 38 of the pass, implemented the tool from these two documents, and the implementation passed the admission checks of Section 3.3. Six later tasks, at positions 56 through 90, resolved tier-dependent decisions by loading the tool from the library instead of re-deriving the mapping, including other credit-limit requests and a card-closure retention flow. The function encodes only the mappings stated explicitly in the cited documents and reports `supported = False` for card names outside them, and its recorded citations are what later episodes check before the implementation is loaded (Section 3.5).

Policy document 1: credit limits

`doc_credit_cards_credit_cards_(general)_015`

Provisional Credit Eligibility Guidelines (Internal)

Maximum Provisional Credit Limits by Card Tier

Entry Tier — maximum $2,500.00
Bronze Rewards Card, EcoCard, Business Bronze Rewards Card, Crypto-Cash Back Card

Mid Tier — maximum $5,000.00
Silver Rewards Card, Business Silver Rewards Card, Green Rewards Card, Silver Zoom Card

Premium Tier — maximum $10,000.00
Gold Rewards Card, Business Gold Rewards Card

Elite Tier — maximum $15,000.00
Platinum Rewards Card, Business Platinum Rewards Card

Invitation Tier — maximum $25,000.00
Diamond Elite Card

Policy document 2: shipping fees

`doc_credit_cards_credit_card_replacements_002`

Credit Card Replacement Shipping Options and Fees

Expedited shipping fees by card tier

Entry-tier — $15.00
Bronze Rewards Card, EcoCard, Business Bronze Rewards Card

Mid-tier — $10.00
Silver Rewards Card, Business Silver Rewards Card, Green Rewards Card, Silver Zoom Card

Premium-tier and above — $0.00
Gold Rewards Card, Business Gold Rewards Card, Platinum Rewards Card, Business Platinum Rewards Card, Diamond Elite Card

Complimentary expedited shipping.

Compiled decision tool (abridged)

```python
def infer_credit_card_tier_from_card_name_policy(*, card_name: str):
    """Infer a credit-card tier from a product name,
    using only explicit tier/name mappings in policy."""
    normalized = _norm(card_name)
    mapping = {
        _norm("Bronze Rewards Card"):
            ("entry-tier", "Bronze Rewards Card"),
        # ... 11 more explicit name-tier entries
        _norm("Diamond Elite Card"):
            ("invitation-tier", "Diamond Elite Card"),
    }
    tier, matched = mapping.get(normalized, (None, None))
    # Limited alias handling and notes elided.
    return {"tier": tier,
            "matched_policy_name": matched,
            "supported": tier is not None,
            "notes": notes}
```

Library provenance. Cites policy documents 1 and 2 above. Compiled at position **38**; reused at positions **56, 68, 73, 87, 88, 90**.

Figure 5: A compiled decision tool and its provenance (GPT-5.2, seed 0). Top: excerpts of the two cited policy documents; the tier taxonomy appears inside rules about provisional credit limits and shipping fees. Below: the tool, abridged from its 94-line implementation, with repeated docstring mappings omitted, and its library provenance. The tool was compiled at position 38 of the online pass and invoked by six later tasks.

B.2 Library Growth and Source Coverage

Figure 6 shows how the decision tool library and the set of policy documents it cites grow over the online pass on τ-Knowledge. The corresponding admission-check and merge statistics are reported in Section 4.2.

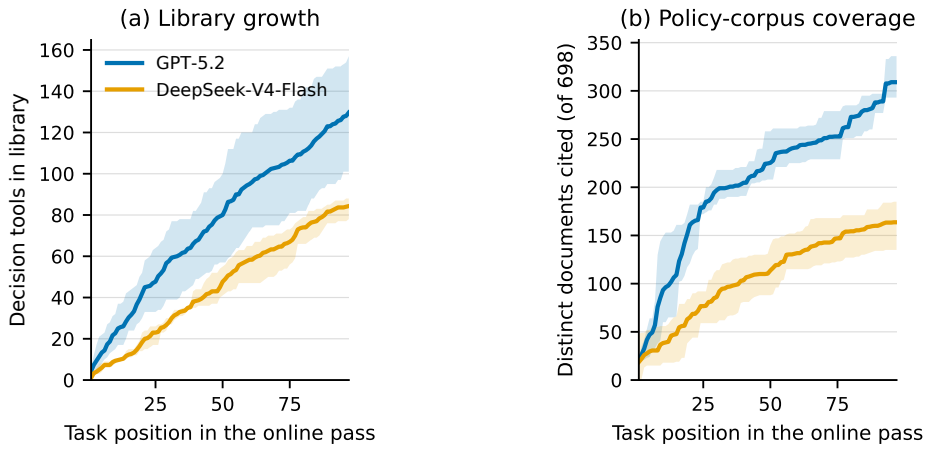

Figure 6: Compilation artifacts on τ-Knowledge over three seeds. **(a)** Decision tool library size. **(b)** Number of cited policy documents. Lines show means and bands show the range from minimum to maximum.

C Decision Tool Failure Analysis

Scope and counting. We provide additional results from the manual review reported in Section 4.3. Table 7 breaks down the 42 incorrect final-library functions by benchmark, base LLM, and primary defect. The examples below illustrate these defects through policy comparisons, isolated counterexamples, and recorded task trajectories.

Each incorrect function is counted once per run in its final retained version and assigned one primary defect. For task-local sources, we align the final implementation with its latest creation or replacement task. Repeated implementations across runs and callers that inherit a helper's defect count separately, so the counts describe affected functions rather than independent error events.

We additionally execute 11 counterexamples across ten selected functions in isolation, without benchmark action calls. These tests reproduce function-level defects; they do not establish that the same inputs occurred during task execution or that the defects caused a scored task failure.

Table 7: Primary defects in the 42 incorrect final-library functions. DeepSeek denotes DeepSeek-V4-Flash. Each function receives one category; callers and defective helpers count separately, as do implementations from different runs. Two functions with conflicting source rules are excluded from this breakdown.

	τ-Knowledge		AutomationBench		
Primary defect	**GPT-5.2**	**DeepSeek**	**GPT-5.2**	**DeepSeek**	**Total**
Policy values and product mappings	4	12	0	0	16
Missing constraints or candidates	4	4	1	1	10
Misinterpreted decision conditions	1	1	1	2	5
Unit and formula errors	1	2	0	2	5
Dependency and execution errors	1	1	0	4	6
Total incorrect	11	20	2	9	42

Incorrect policy values and product mappings (16/42). The largest group includes incorrect account limits, fees, interest rates, and associations between products and benefits. In a DeepSeek-V4-Flash τ-Knowledge run, the checking-account function documents a four-account cap but executes `existing_checking_count < 5`. A verified 30-year-old customer with four accounts and no disqualifying closure is therefore judged eligible for a fifth account. The same function correctly rejects five existing accounts, showing that it implements the wrong threshold rather than ignoring account counts altogether. Other errors propagate through otherwise plausible recommendations: a product helper records Sky Blue's documented 1.25% APY as unavailable, and its caller excludes that product when positive interest is required. Both functions count as incorrect, but they share the same defective product information.

Missing constraints or candidates (10/42). These functions implement only part of the decision: examples include omitted promotion windows, missing security-clearance conditions, untracked shared referral capacity, and incomplete recommendation sets. On AutomationBench, both base LLMs' blog-refresh functions enforce traffic and date thresholds while omitting the latest requirement that `Priority_Score > 60`. Neither interface accepts that field. The actual inventory row "Support Ticket Resolution Guide" has 11,500 sessions, a 23% traffic decline, a May 28, 2025 update date, and priority 52. Both frozen functions return true, although the current policy excludes the row. Supplying accurate values for every accepted argument cannot recover the missing condition.

The saved trajectories connect this omission to incomplete source acquisition and a downstream write. GPT-5.2's three Slack searches return zero matches; DeepSeek-V4-Flash reads the policy emails and inventory without reading Slack channel messages before handoff. Both tools cite the older emails. Both execution traces record setting the excluded row's status to `Refresh`, and the DeepSeek-V4-Flash run additionally reads back that incorrect status. This is an observed policy-violating operation, although it does not isolate the effect of this defect on the overall task score.

Misinterpreted decision conditions (5/42). Here the implementation retains relevant terms but changes their relationship or truth conditions. Both base LLMs' AutomationBench payment functions search independently for the approver title and an approval keyword. The following constructed input exposes the shared error:

Counterexample with conflicting approvals

```
amount = 32500
approvals = "Dept Head: Approved; CFO: Denied"
policy-dependent decision: HOLD       function decision: APPROVE
```

The source requires explicit CFO approval above $25,000, and both compile traces contain that rule before the functions are admitted. The functions nevertheless combine the CFO's name with someone else's approval, locating the defect in rule implementation despite source exposure. Similarly, the DeepSeek-V4-Flash escrow function splits each condition at its final space; "Inspection Not Met" becomes a condition named "Inspection Not" with status "Met." For "Delivery Met, Inspection Not Met," it returns `Released`, while the source requires a hold. Both payment functions and the escrow function pass citation completeness, which cannot verify who approved or whether a condition is negated.

Unit and formula errors (5/42). These functions use familiar constants with the wrong quantities. A τ-Knowledge liability function treats a two-business-day threshold as two calendar days. With three elapsed calendar days spanning a weekend and an explicit true flag for reporting within two business days, it returns $500 rather than the benchmark policy's $50 tier. In another case, an AutomationBench budget function derives month length from monthly budget divided by daily budget. Budgets of $2,000 and $50 imply an artificial 40-day month. With $1,920 spent and five days remaining, it projects $2,194.29 and returns `WARNING`; using the elapsed-day average in any 28–31-day calendar month projects above $2,200, requiring `CRITICAL`. The budget values occur in the source, while the spending state is a constructed counterexample.

Dependency and execution errors (6/42). These include an undefined refund variable, incompatible helper arguments, imports that incorrectly treat library functions as installed modules, an

unsupported API query, and an internal naming mismatch that silently bypasses eligibility filtering. For example, the safety-walk selector calls its eligibility helper with six arguments, but the retained helper accepts four. A normal Q1 Main-building candidate raises `TypeError` before selection. The dependency is present; their interfaces are incompatible. This illustrates why successful definition-time loading does not guarantee that a composed decision procedure can execute.

Implications and unresolved cases. Of the 42 incorrect functions, 38 contain extracted policy constants and 25 of those pass citation completeness. Literal source support therefore leaves substantial semantic gaps. The cases suggest distinct checks: policy-derived boundary examples for limits, explicit inputs for every necessary condition, actor-and-status parsing for approvals, unit-aware calculations, and invocation tests for composed tools. These are proposed validation targets, not evaluated improvements. Two additional functions remain uncertain because policy documents conflict about a daily purchase limit and the start of a referral retention period; they stay in the correctness denominator and are not treated as demonstrated implementation errors.

D Cross-Task Decision Tool Reuse

Reuse counts runtime invocations of decision tools compiled by earlier tasks when the task sequence contains an applicable prior tool. Let $\mathcal{L}_t$ denote the decision tool library at the start of task t, and let $\mathcal{C}_t$ denote the decision tool calls actually executed while solving that task. We define

$$\text{Reuse} = \sum_{t} \sum_{c \in \mathcal{C}_t} \mathbf{1}[\text{name}(c) \in \mathcal{L}_t] \,, \tag{2}$$

Thus, decision tools compiled during the current task and calls appearing only in unexecuted branches do not count as reuse. Because this quantity depends on policy overlap, source portability, and task order, we interpret it within each benchmark rather than as a general measure of method quality.

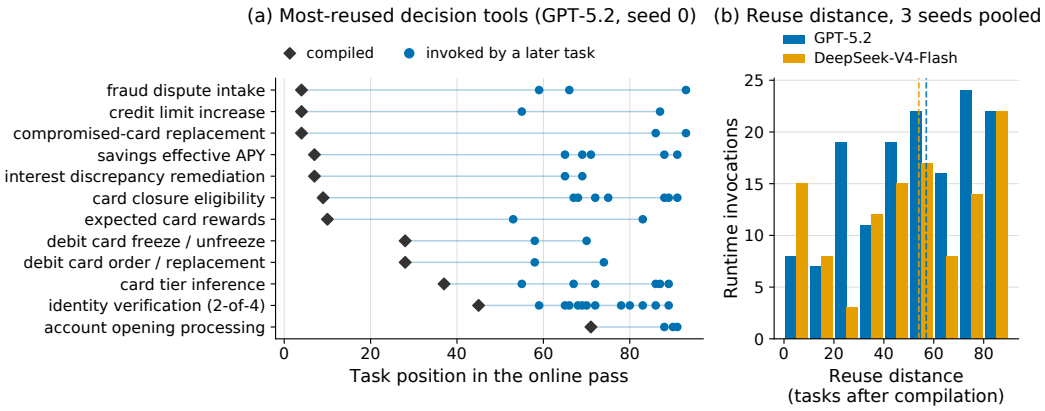

Figure 7: Cross-task decision tool reuse on τ-Knowledge. **(a)** Compilation and subsequent invocation tasks for the twelve most-reused decision tools in one representative run using GPT-5.2 with seed 0. Diamonds denote compilation and dots denote reuse. **(b)** Reuse distance over three seeds for each base LLM. Dashed lines show medians.

Under the runtime definition in Eq. 2, Plot records 151 cross-task decision tool invocations over three seeds with GPT-5.2 (67/49/35) and 115 with DeepSeek-V4-Flash (57/28/30). Depending on the seed, between 10 and 31 of the 97 tasks invoke at least one decision tool compiled by an earlier task (Figure 7a). The median reuse distance is 57 tasks for GPT-5.2 and 54 for DeepSeek-V4-Flash (Figure 7b). Frequently reused decision tools cover identity verification, account-closure eligibility, and card-tier inference. All reused decision tools passed the admission checks in Section 3.3 and retain citations to their policy documents.

AutomationBench. Across 120 tasks per base LLM, a scan of the saved scripts identifies no direct calls to decision tools compiled by earlier tasks. Task-level telemetry flags one task for DeepSeek-V4-Flash (1/120, 0.8%) and none for GPT-5.2. The flagged task, `sales.zoom_recurring_meeting_optimizer`, uses `inspect.getsource` to inspect `match_participant_to_opportunity`, a tool compiled in `sales.zoom_attendance_pipeline`. This operation reads the function's source rather than invoking it, so it does not qualify as reuse under Eq. 2. The script scan counts direct function calls and does not rule out indirect invocations. These observations are consistent with the limited policy overlap and source portability across AutomationBench tasks.

D.1 Reuse Coverage across Customer Requests

Table 8 aggregates decision tools across all six τ-Knowledge runs, pooling both base LLMs and all three seeds. The five selected families verify customer identity, identify an account's benefit level, check account-opening or account-closing rules, or check whether money can be temporarily refunded while a disputed transaction is investigated. They cover 23, 20, 18, 15, and 14 distinct later tasks, respectively. Each task ID counts once per family across function names, seeds, and models, after applying each run's creation/update exclusions. Mixed-role functions are assigned using call arguments; shared family membership does not imply identical rules. Rows describe customer requests using the overlapping labels defined in Appendix D.3. Counts record task-level co-occurrence, rather than attributing the outcome of a request to a function. Appendix D.2 examines how two individual implementations enter execution across all 24 task–function pairs, including failed tasks.

Table 8: Decision tool reuse by customer request, pooling two base LLMs and three seeds. Each function column counts distinct tasks containing that request and a later call-site match to the function family. Labels include scripted conditional requests. Counts describe task-level co-occurrence, not success or the function's contribution to each request.

		Tasks reusing a decision tool				
Customer request	All tasks	Verify customer identity	Identify benefit level	Check opening rules	Check closing rules	Check temporary refund
Choose a card or account	36	5	6	18	13	0
Open an account	30	4	5	18	11	0
Close an account	15	5	7	8	15	0
Change contact details	2	1	0	0	0	0
Review account records	6	3	2	0	0	2
Check rewards or bonuses	18	7	4	0	5	0
Refer someone	8	1	0	0	0	0
Dispute a charge	18	7	8	0	0	13
Replace a card	19	8	7	0	0	9
Change card limits	6	2	3	0	0	2
Enable or block a card	13	5	3	0	0	0
Review bank fees	5	0	0	0	1	0
Review savings interest	5	2	0	0	0	0
Deposit or transfer money	12	3	5	6	7	0
Distinct tasks	**97**	**23**	**20**	**18**	**15**	**14**

A task can have several requests. The bottom row counts each task once per column; it is not the sum of the rows above. Creation/update tasks and known unexecuted calls are excluded from reuse counts.

D.2 Trace Analyses of Two Representative Functions

The coverage analysis in Appendix D.1 pools semantically related function implementations across two base LLMs and three seeds, counting each later task ID once per selected family rather than counting invocations as in Eq. 2. The audits below concern two individual implementations within those families: the GPT-5.2 seed-0 identity gate and the DeepSeek-V4-Flash seed-1 checking-opening function. Each has 12 later tasks after excluding creation/update tasks. We retain the

complete cohorts without filtering by outcome and inspect the executed scripts, recorded outputs, nested environment calls, and subsequent actions for each task. Identity-verification results are summarized below; Table 9 lists the task-level results for the checking-opening function. These implementation-specific trace audits do not certify every member of a pooled family and are not interventions that remove a function from an otherwise identical execution state.

Identity verification. The focal function is `identity_verification_2_of_4_policy`, compiled in task 036 of the GPT-5.2 seed-0 run. It exposes the on-file and supplied identity fields as inputs, returns the matched fields and a verification verdict, and specifies whether verification must be logged. Across its 12 later tasks, five supply two matching fields and seven supply three; verification is logged in all 12. The gate is called twice in task 048 following a later, unrelated parsing failure, so the 13 call sites correspond to 12 distinct tasks. Task 094 does not print the verdict, but its successful-verification branch executes the guarded logging and account calls. None of these 12 tasks tests rejection of an identity with fewer than two matches. Passing the local gate does not certify the remaining business decisions.

For example, task 016 asks about an unpaid bonus for referring a friend. The unchanged function compares this customer's supplied details with the current bank record and identifies three matches: date of birth, email, and phone number. The recorded output reports a positive verification result, followed by successful verification logging and referral retrieval. These environment actions are performed by the surrounding execution code after checking the function's verdict. The function supplies the prerequisite decision; it does not itself retrieve referrals or determine whether a bonus is owed.

Checking-opening function. The focal function originates in task 057 of the DeepSeek-V4-Flash seed-1 run and is named `checking_account_opening_eligibility`. Its observed interface selects personal checking, personal savings, or business checking through `account_type`. It retrieves current account information inside the function and returns eligibility together with component checks and reasons. Table 9 lists all 12 later tasks, including repeated calls. There are 22 invocations: 14 return eligible, one reports no active checking account, five return an account-information retrieval failure, and two raise balance-parsing exceptions. Eleven tasks obtain at least one eligible return; the exception is task 076, where the agent reasons manually after the retrieval failure and still completes the task. The savings, personal-checking, and business-checking branches are used in 8, 4, and 3 tasks, respectively; tasks 062, 065, and 066 use two branches.

Table 9: All 12 later tasks using the checking-opening function, in execution order. P, S, and B denote personal checking, savings, and business checking. An eligible return is an observed function output, not a certification of every encoded rule. Full-task scores are retained even when the agent opens an account or passes a local check.

Task	Branch	Observed checks and subsequent workflow	Score
058	S	Eligible; savings opened alongside a credit-card application.	1
062	B, S	Both eligible initially; two later balance-parsing errors during account reorganization.	0
070	B	Account read fails, then eligible on retry; business checking opened.	0
065	P, S	Both eligible; checking and savings opened, old checking closed.	0
056	B	Account read fails, then eligible on retry; business-account replacement and savings setup attempted.	0
064	S	Account read fails, then eligible on retry; savings opened with a credit-card application.	1
076	P	Account read fails; agent reasons manually and opens a travel checking account.	1
060	S	Eligible; emergency-fund savings opened before old checking is closed.	1
075	P	Account read fails, then eligible on retry; travel checking opened.	1
061	S	Eligible twice; savings opened and checking-closure workflow completed.	1
063	S	Eligible; savings/card combination opened and funds transferred.	0
066	P, S	Personal checking eligible; later savings check reports no active checking after account changes; savings deferred.	0

What the function does not resolve. Eligibility is distinct from choosing the best product, arranging funding, or ordering account changes. In task 066, the agent closes the old checking account before testing savings eligibility; the later return reports no active checking account, and the agent defers savings opening. This records a downstream blockage rather than demonstrating that the function prevented it. In task 062, repeated calls fail to parse a currency-formatted balance such as `$0.00`. The retained implementation also makes schema- and clock-dependent assumptions when counting accounts and measuring tenure. Consequently, the observed eligible returns are not a complete correctness assessment of age, balance, account-count, or closure-history boundaries. The audit supports reuse of the same decision interface across changing customer/account states, while preserving its observed retrieval, data-handling, and workflow limitations.

D.3 Task Labels Based on Customer Requests

Table 8 uses analyst-assigned, multi-label annotations of the services requested in the frozen customer instructions. The annotation unit is a task containing a scripted request, including conditional follow-ups and explicit acceptance of concrete operations. These annotations describe task content, not the requests reached in every observed trajectory. Labels are assigned without using agent outputs, rewards, or decision tool reuse counts. The customer instructions are identical across all six runs. Table 10 defines all 14 labels; 63 of the 97 tasks receive more than one, giving 193 task–request assignments. Each assignment has a verbatim source excerpt and character offsets in the accompanying analysis files, with notes for ambiguous boundaries. The labels are post-hoc descriptive annotations, not benchmark-provided categories or an independently replicated annotation study.

We apply the same operation labels to credit cards, debit cards and deposit accounts; product context is recorded separately. A refused or unsupported request retains its label. Routine identity verification, internal data retrieval and transfer to a human are not separate business-request labels. A concrete payment accepted in the script is included, whereas a deposit expressly postponed until after the conversation is excluded. A reward discrepancy is not a disputed purchase charge, and replacing a physical card does not close its account. Thus, task 054 receives *Dispute a charge*, *Replace a card*, and *Change card limits*; task 077 combines replacement and card-access requests across credit and debit cards. Task 061 receives four labels: account selection, opening, closure, and depositing money to cover a closure fee. Task 010 combines checking missing bonuses with retrying a referral, whereas task 016 only checks an earlier bonus.

For each function family, we first identify its later-use task IDs within each run and then take the union across runs. A cell intersects that union with the tasks bearing a request label. The bottom total is the size of the union itself, not the sum of the overlapping cells. This task-level co-occurrence does not attribute a function call to every request in a compound task. For example, an opening-rule function can appear in the *Close an account* row because a customer asks to both open and close accounts. All request labels and their denominators are retained in the table, rather than selecting categories based on coverage. The five function columns describe identity matching, mapping a named product to its policy-defined benefit level, opening conditions, closing conditions, and temporary-refund eligibility or amount. A temporary refund restores money while a transaction dispute is investigated and is not necessarily a final refund.

Table 10: Request labels for all 97 tasks. Counts overlap and describe scripted requests, regardless of whether they are completed.

Customer request	Tasks	Inclusion rule and boundary
Choose a card or account	36	Compare or recommend a product for stated needs, including a referral recipient's account.
Open an account	30	Apply for a credit card or open a deposit account; excludes replacing an existing card.
Close an account	15	Request account closure, including blocked requests or alternatives accepted afterward.
Change contact details	2	Update contact information; excludes supplying existing details for verification.
Review account records	6	Explicitly inspect balances, transactions, payment status or credit-report entries.
Check rewards or bonuses	18	Claim a promotional or retention bonus, or investigate missing or incorrect rewards.
Refer someone	8	Obtain referral instructions or submit/retry a referral; excludes checking an old bonus alone.
Dispute a charge	18	Challenge a card transaction or ATM cash discrepancy; excludes rewards, fees and interest.
Replace a card	19	Request a new physical card or card number; excludes merely mentioning a pending replacement.
Change card limits	6	Change a credit, spending or withdrawal limit; excludes desired features of a new product.
Enable or block a card	13	Freeze, unfreeze or activate a card; resolve declines, PIN locks or travel-block concerns.
Review bank fees	5	Review, correct, rebate or waive existing service fees; excludes prospective product comparison.
Review savings interest	5	Investigate interest already paid; excludes comparing future returns when choosing an account.
Deposit or transfer money	12	Request or accept a deposit, transfer or balance payment now; excludes expressly postponed funding.

E Qualitative Case Studies

The following case studies examine how agents apply policy conditions during execution and reuse policy knowledge across tasks. Message indices refer to zero-based positions in the archived trajectories.

E.1 Applying Policy Before Action

In τ-Knowledge `task_054` with DeepSeek-V4-Flash, a customer asks the agent to dispute a fraudulent charge of $487.50, order a replacement card, and increase her credit limit by $4,000. She explicitly prioritizes the fraudulent charge, stating that it should be handled first. Two policy rules nevertheless require a different execution order. First, an increase for a premium-tier card may not exceed 50% of the current credit limit. Given a current limit of $5,000, the request is ineligible, while an increase of at most $2,500 is permitted. Second, a credit-limit increase cannot be processed while either a dispute or a replacement order is pending. The increase must therefore be negotiated and submitted before the dispute and replacement.

Retrieved policy and observed action order. The selected CodeAct trajectory is trial 1. At message 43, the agent receives the full document *Internal: Processing CLI Approvals and Denials*. Its eligibility checks explicitly require no active disputes and no outstanding replacement-card orders. The response also states that a pending replacement blocks the increase until delivery or cancellation. This evidence is present in the agent's context before it performs the dependent business actions.

Nevertheless, at message 80 the agent calls `file_credit_card_transaction_dispute_4829`, and the next tool response confirms a successfully submitted dispute. At message 82 it calls `order_replacement_credit_card_7291`; the response confirms an ordered replacement and cancellation of the old card. Both operations execute successfully. At message 94, while processing the credit-limit increase, the agent identifies the dispute and replacement it has just created as blocking conditions and records a denial for `pending_disputes`. The customer subsequently asks whether the actions can be reordered, but the agent explains that they have already been processed.

The failure therefore occurs between retrieving the governing condition and applying it to the action sequence: locally successful tool calls create a state in which another requested operation is no longer permitted. The pending-action conditions need to inform planning before the dispute and replacement are submitted. Across the three archived runs of this task, PLOT succeeds in all three, whereas Vanilla, CodeAct, and ACE each receive a score of zero in all three.

Compiling the decision before execution. PLOT instead identifies credit-limit eligibility as one of the policy-dependent decisions that must be resolved before acting. The slot description covers the requested increase, account conditions, and pending disputes and replacements. The compiled tool exposes those conditions as explicit inputs and returns both eligibility and the maximum permissible increase:

Credit-limit eligibility and returned result

```
check_cli_eligibility(
  card_tier="Premium",
  current_credit_limit=5000.00,
  requested_increase_amount=4000.00,
  has_pending_disputes=False,
  has_pending_replacement=False, ...)
-> {'eligible': False,
    'max_allowed_increase': 2500.0,
    'max_increase_pct': 50,
    'issues': ['Requested increase exceeds the
                maximum for Premium-tier.']}
```

The execution agent calls this tool before submitting the credit-limit request, filing the dispute, or ordering the replacement. It explains the $2,500 maximum and the ordering constraint, obtains the customer's agreement, submits and approves the resulting $7,500 limit, and only then files the dispute and orders the replacement card. The recorded CodeAct response already contains the pending-action conditions; PLOT additionally exposes those conditions in a callable decision interface before the dependent actions. The eligibility result supplies the permitted amount, and the agent combines the interface with the supporting policy to choose an order that completes all three requests. Appendix E.2 examines how decision tool retrieval and execution support a different request without introducing an unrelated account-closure action.

Decision tool reuse. The eligibility function remains in the decision tool library after `task_054`. A later episode, `task_053`, retrieves and calls the same function without recompiling it, supplying the new account's credit limit, requested increase, and pending-action flags. The function applies the stored policy conditions to those inputs and returns the eligibility decision and maximum increase.

E.2 AVOIDING OUT-OF-CONTEXT REUSE

In τ-Knowledge `task_037` with DeepSeek-V4-Flash, a customer identifies two fraudulent charges and asks the agent to "cancel this card and get a new one" and dispute both charges. Vanilla, CodeAct, and PLOT succeed in all three trials, whereas ACE fails in all three.

Retained guidance beyond its policy scope. We examine ACE trial 0 and its 415-line playbook snapshot before this task. The source retention policy concerns closing a credit-card account, whereas the replacement policy cancels the old card as part of reissue while retaining the account. Although entry `shr-00126` retains the account-closure scope, entry `shr-00177` directs the agent

to follow the retention protocol when a customer reports unauthorized transactions and requests a replacement. It specifically requires checking closure history with `get_closure_reason_history_8293` and logging a closure reason with `log_credit_card_closure_reason_4521` before filing disputes or ordering the replacement. Entry `vc-00180` likewise adds these closure steps to the checklist for fraud disputes. These entries preserve the closure action sequence while extending its trigger to requests that do not require account closure.

When the guidance entered the playbook. Both entries first appear in the update after `task_036`, at position 42 of the same run, and remain unchanged in the snapshot before `task_037`, at position 78.

Observed application outside that scope. The customer requests cancellation and replacement at message 14. ACE receives the source retention and replacement policies at messages 19 and 20, then queries closure-reason history at message 29. At message 49, it calls `log_credit_card_closure_reason_4521` with reason `other`; message 50 confirms the write. ACE subsequently files both disputes at message 65 and orders the replacement at message 70. The replacement response confirms that the old card is cancelled while the account number is retained. The recorded sequence thus matches the closure-history and reason-logging steps in the overgeneralized guidance, despite the source policies distinguishing replacement from account closure. Trial 1 also records a closure reason; trial 2 files the disputes but transfers to a human without ordering the replacement.

Decision-specific reuse in PLOT. In the seed-0 trajectory, PLOT retrieves provisional-credit and replacement-card tools compiled 74 tasks earlier and invokes them with the current customer's facts. The replacement tool evaluates the Silver card's replacement allowance and returns the applicable $10 expedited-shipping fee. The agent completes both disputes and orders the replacement without adding an account-closure record. The retained implementations support the two required policy-dependent decisions while the current account and card details determine their outputs.

Locating the state error by replay. In the examined pair of trajectories (PLOT seed 0 and ACE trial 0), both agents correctly mark the two fraudulent charges as eligible for provisional credit. Replaying their recorded banking calls against the task's initial state reproduces the database outcomes: every table matches the reference state for PLOT, whereas ACE adds one closure-reason record and its corresponding tool-call log. Removing only ACE's closure-reason write, while keeping all subsequent calls and arguments fixed, makes every table match the reference state. This replay isolates the unnecessary write as the source of the database mismatch in this trajectory.

F PROMPT TEMPLATES

We present condensed compilation, execution, handoff, and merge prompt templates for τ-Knowledge. Braced placeholders are filled with benchmark information and agent runtime context.

Compilation

```
You are a customer service agent.  The bank's policy is in natural
language.  Your job right now is to compile policies that are related to
the task into functions.

Your tools
{tool_description}
Ask the customer whatever you need to.  Reply in plain text and their
answer comes back to you here; if it changes what this task needs,
declare again.  What you must not do is the task itself:  change no
record, no writes, no transfers, no verification logging.  The agent
after you will make every change that was asked for.
Source names you may use in possible_sources:  {source_names}

How to reach the environment
{execute_reference}
Everything the bank can do is reachable only from inside that code, reads
and writes alike.  Names persist between scripts, like cells in a
notebook.
What you see back is what you print().  A trailing bare expression comes
back too, whole.  Everything that comes back stays in your context for
the rest of this task, so slice and grep instead of listing whole
directories.  Making several calls in one script is one turn, not
several:  put them in a single script.

Operating policy
{domain_policy}
```

Execution

```
You are a customer service agent, taking over a conversation in the
middle of a task.  An agent already searched the knowledge base and
compiled what it could of the policy this task needs; everything it left
you is on this page, and every function it left is defined in your kernel
right now, ready to call.  A function answers by being called; its
docstring carries the policy constants you may quote.  It compiled
against the conversation as far as it had gone at the moment it handed
over, so whatever the customer raises after that is something nothing was
compiled for.
Solve the task by writing scripts.  Reply in plain text to ask the
customer something or to tell them the outcome.

What the agent before you left
{handoff}
RULED_OUT lists what was already found not to exist.  OWED is in work
order.

Your functions
{functions}

The documents the last agent read
Each line below is one document:  its file in the knowledge base, what it
is, and how it crossed (cited by a function, named in the note, or read
during compile).
{evidence}

What is already in your kernel
The agent before you left these names bound, and they are still bound.
Use them rather than recomputing what they hold.
{facts}
```

Execution (continued)

Bank calls already made and answered

```
Each line is one distinct call a compile-phase script made, read from the
environment's own record rather than from anyone's notes:  the exact
arguments, and whether the environment answered or refused.  An id that
appears in an answered call names something real.  An ERROR line means
that call, as written, was refused.
{calls}
```

How to reach the environment

```
{execute_reference}
Everything the bank can do is reachable only from inside that code, reads
and writes alike.  Names persist between scripts, like cells in a
notebook.
What you see back is what you print().  A trailing bare expression comes
back too, whole.  Everything that comes back stays in your context for
the rest of this task, so slice and grep instead of listing whole
directories.  Making several calls in one script is one turn, not
several:  put them in a single script.
```

Operating policy

```
{domain_policy}
```

Handoff

```
You are about to hand this conversation to a fresh agent.  It will see
the conversation, the functions you push, the full text of every document
your note names, a one-line catalog entry for every other document you
read, and this note.  It sees nothing else you read or wrote.
Write the handoff note now, in four parts with these exact headers.
WRITES: one line per thing the next agent must still do for this customer
-- the action, the function that decides or computes it, and where what
it acts on lives.  Every function a line names crosses; 'none' pushes
nothing.  What a pushed function calls crosses with it.
DOCUMENTS: ids of the documents that matter most, one per line with a few
words on what each governs, spelled exactly as the environment printed
them.
RULED_OUT: what you established does not exist, one line each, so the
next agent does not run the same hunt.
OWED: what the customer is still owed, one line each, in the order the
work must happen.  When policy makes one item block another, put the
blocked item after it and say why.  Include identity you verified but did
not log.
Write 'none' under a header with nothing to list.
```

Merge

```
Two library entries overlap.  Write ONE definition that covers both, or
reply exactly NO if they are genuinely different things.
```

{left.name} -- {left.description}

```
{left.code}
```

{right.name} -- {right.description}

```
{right.code}
Reply with a single fenced Python block and nothing else.  Keep the
clearer of the two names.  Do not invent behaviour neither one has.
```