TravelReimGPT: A Conversational Agent for Policy-Compliant and Reliable Travel Reimbursements

Anonymous ACL submission

Abstract

We present TravelReimGPT, a user-centric, conversational AI system for automating travel reimbursement tasks under strict policy constraints. While recent LLM-based agents have shown promise in open-ended dialogues, they often struggle with numerically sensitive and rule-governed applications due to limitations in symbolic reasoning and consistency. To address these challenges, we propose a Programming-Object-Centric Architecture (POCA) that transforms user inputs and documents into programming objects, which serve as the foundation for deterministic, logic-driven policy enforcement. A self-corrective object constructor, combining validation logic and iterative model-guided refinement, enhances the accuracy and completeness of extracted information. Through multi-turn interactions, TravelReimGPT gathers necessary inputs (e.g., receipts) and generates reimbursement reports that comply with complex policy rules. Experiments on real-world reimbursement cases show that, powered by GPT-4.1, our system consistently produces accurate reimbursement outcomes across all tested cases, exhibits robust conversational fluency, and achieves high user satisfaction. In contrast, prompting-based baselines occasionally yield inaccurate reimbursements, highlighting reliability and control limits. This work demonstrates a practical and extensible framework for building reliable AI agents for rule-intensive domains, with potential applicability to broader tasks such as auditing and budget compliance.

1 Introduction

Business-travel reimbursement is one of the most routine yet time-consuming back-office workflows in universities, governments, and industry (GBTA, 2023). Every trip generates a small archive of artefacts—receipts and per-diem tables—that must be parsed, checked against multiple policy handbooks, and distilled into reimbursable dollar amounts. The paperworks and policy-compliance check are tedious and burdensome. Software vendors have rolled out "smart" reimbursement portals to facilitate the process. They employed optical character recognition (OCR) for amount detection on documents and have integrated simple math operations for simple reimbursement rules, e.g., standard meals grid amount calculation (SDO CPA, 2025; Boese, 2025). These tools speed up raw data entry but stop far short of human-level assistance. The system might be unable to recognize the truly incurred amounts for reimbursement. For example, it might extract the total amount including tips when the tips should be removed under some polices; and it is hard to handle the complicated cases involving capped reimbursable expenses (e.g., lodging). As a result, staff must still review and check every extracted amount item, cross-check it against the raw receipts, review the grant or institutional policies manual, and compute the reimbursable amounts by hand. The semi-automation saves a few keystrokes but leaves the cognitive and numerical burden untouched (Shaked, 2025).

The recent breakthrough in large-scale generative models promises a leap forward on cognitive AI(McKinsey & Co., 2023; Johnston, 2025). Large Language Models (LLMs), represented by GPT-4-series models, exhibit extraordinary proficiency in natural language understanding and generation(Achiam et al., 2023; Touvron et al., 2023). The emergence of such sophisticated generative AI models has facilitated near-human-level competencies and task automation in various domains, including software development, text summarization, and various cognitive activities traditionally requiring human expertise (Chang et al., 2024; Thirunavukarasu et al., 2023; Wei et al., 2022a; Gottweis et al., 2025). A current and rapidly growing trend involves developing human-level AI tools and systems designed to alleviate humans from tedious and repetitive workloads (Wang et al., 2024a).

Prominent applications include automated coding assistants, intelligent scheduling systems, and advanced virtual personal assistants (GitHub, 2025; Microsoft, 2025; Stige et al., 2024). However, critical limitations persist, such as the occurrence of hallucinations and constraints imposed by the autoregressive nature of LLM architectures, which hinder their capability for reliable formal logical reasoning (Bommasani et al., 2021; Ji et al., 2023; Xu et al., 2024b). Notably, LLMs often struggle with arithmetic tasks, failing to consistently ensure accuracy even in basic addition and simple character-counting exercises, such as counting the letter 'r' in the word "strawberries". Consequently, employing LLMs in numerically-sensitive tasks including business-travel reimbursement, which require precise and multiple logical operations and have extremely low tolerance for numerical inaccuracies, remains a significant challenge.

Driven by the consideration, in this work, we propose `TravelReimGPT`, a reliable, effective, and highly user-friendly generative model designed to streamline and automate travel reimbursement processes, including paperwork processing and ensuring compliance with relevant policies. `TravelReimGPT` serves as an end-to-end expert system, guiding users to provide mandatory documentation and information, subsequently generating accurate reimbursable outcomes.

The primary contributions of this work are fourfold:

- **TravelReimGPT.** the first end-to-end, conversational system that directly turns raw receipts and free-text queries into reliable policy-compliant reimbursement reports, lifting the manual burden of document parsing, rule lookup, and calculation.

- **Programming-Object-Centric Architecture (POCA).** an architecture designed for implementing deterministic rule compliance. It ensures numerical precision, enhanced usability, and cognitive robustness by effectively transforming unstructured data (i.e., free texts and documentations) into programming objects that operate reliably within formal logical systems. POCA facilitates seamless interaction between generative AI and deterministic formal logic networks for developing cognitive AI to specific rule-compliant tasks, such as travel reimbursement.

- **AI cognition analysis.** Demonstrates that the performance of the LLMs directly impacts the overall system performance, necessitating the use of advanced models. The study emphasized the necessity of utilizing advanced generative models, such as `GPT-4.1`, for an effective and smooth cognitive AI system.

- **Blueprint of AI system for complicated rule-compliant human tasks.** Provides a practical and systematic solution to develop reliable, user-friendly, and effective AI systems for complex tasks requiring numerical precision and adherence to high-reuse, well-defined rules, such as rule-constraint budgeting, auditing, and other similar tasks.

To the best of our knowledge, `TravelReimGPT` is the first end-to-end cognitive AI system for numerically grounded, high-reuse-rule-bound tasks.

2 Related Works

LLM-based agents. LLM-based agents leverage LLMs to enable intuitive natural language interactions and facilitate various tasks (Plaat et al., 2025; Xi et al., 2025). To enhance their performance, innovative prompting and agent design techniques have been proposed(Schulhoff et al., 2024): Chain-of-Thought (CoT) (Wei et al., 2022b) improves accuracy and interpretability by externalizing reasoning steps; Function/tool-calling(Masterman et al., 2024) extends capabilities by integrating predefined functions, tools, and external knowledge; And ReAct(Yao et al., 2023) interleaves reasoning and action for dynamic, context-aware behavior. Existing agents such as MetaGPT (Hong et al., 2023), HuggingGPT(Shen et al., 2023), Chameleon(Lu et al., 2023), and KnowAgent(Zhu et al., 2024) are designed to integrate language understanding, reasoning, external knowledge, and tool use, aiming for increasingly sophisticated capabilities.

Cognitive agents for logic-intensive tasks. LLMs often hallucinate and underperform on tasks demanding precise symbolic manipulation, such as financial and audit domains, mathematics, and rule-compliant tasks(Kang and Liu, 2023). Recent research augments LLMs with symbolic and programmatic components to mitigate the hallucination and enhance formal logic(Cheng et al., 2025; Xiong et al., 2024; Xu et al., 2024a; Fang et al., 2024). OpenAI's Code Interpreter, CodeAct(Wang et al., 2024b), AgentCoder(Huang et al., 2023), Mathcoder(Wang et al., 2023), and PAL(Gao et al., 2023) translate natural-language tasks into executable code, yielding better accuracy on arithmetic and structured-reasoning benchmarks. Others, such as MCTSr (Zhang et al., 2024a) and Aflow (Zhang et al., 2024b), employ Monte Carlo Tree Search for more effective decision-making. These hybrid neuro-symbolic approaches combine LLM flexibility with formal precision. However, most current agents are optimized for general or junior-level logic tasks and perform poorly on complex symbolic challenges. Domains like rule-compliant reimbursement demand task-specific agents that are robust, accurate, and user-centered (Miehling et al., 2025). Yet, a unified framework for building such reliable and domain-specialized agents remains absent.

3 `TravelReimGPT` System Design

We introduce `TravelReimGPT`, a user-friendly, end-to-end system designed to automate travel reimbursement processing and generate policy-compliant reports, including automated document processing and reimbursement rule enforcement, while allowing users to interact freely through unstructured inputs such as free-form text and supporting documents. The system integrates advanced LLMs as its core AI engine for robust language understanding and generation, and programming engines for generating reliable and deterministic rule-compliant reimbursement results.

Given the inherent complexity of travel reimbursement, which requires effective expense entity processing and policy compliance involving frequently multiple interacting constraints and calculations (e.g., total allowable tips), we propose the POCA. POCA enforces entities and rules to be expressed as programming objects and logic networks so that all computations remain transparent, testable, and reproducible. Object constructor (OC) of POCA is designed to construct executable programming objects that could be seamlessly interacted within the programming logic network. OC employs a definition-guided, programming-engine-driven, and self-corrective loop to ensure the quality of constructed objects.

3.1 `TravelReimGPT` User Interface (UI)

`TravelReimGPT` exposes a Web UI built with Open WebUI[1], an extensible, feature-rich, and user-friendly self-hosted AI platform. Figure 1 displays the web UI of TraverReimGPT. Users can attach documents and images, type natural-language queries, and engage in multi-turn conversations. All conversations are stored in the sidebar so that users can resume any thread on demand. Upon completion, users can download policy-compliant reimbursement reports with a one-click button.

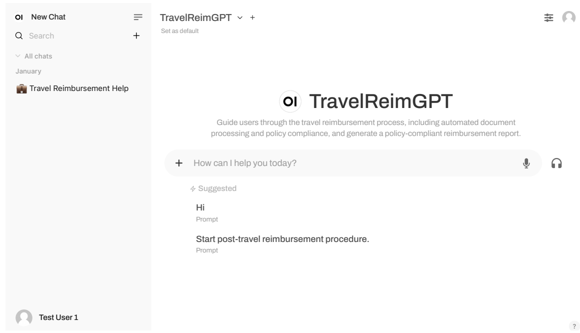

Figure 1: The Web UI of `TravelReimGPT`

3.2 Core Engines

There are two types of core engines in the development of `TravelReimGPT`: the **AI Engine** and the **Programming Engine**.

- **AI Engine.** Provides the cognitive foundation of `TravelReimGPT`, enabling intelligent language understanding and generation. Its core functional requirements include natural language understanding & generation, programming language comprehension, and visual understanding.
- **Programming Engine.** Serves as the deterministic and rule-execution layer of `TravelReimGPT`. It ensures the generation of reproducible, numerically accurate, and policy-compliant outcomes. Its key responsibilities include: 1) *Construction of domain-specific programming objects*: Instantiates programming objects representing critical reimbursement entities (e.g., expenses, travel overview). 2) *Execution of rule-based logic networks*: Operationalizes rule logic networks, encoding dependencies, conditionals, and hierarchical relationships among objects.

`TravelReimGPT` employs Python as the language of its programming engine.

3.3 POCA

POCA ensures that policy enforcement relies exclusively on deterministic programming logic rather than fragile rule-in-context prompting approaches

[1] https://github.com/open-webui/open-webui

(illustrated in Figure 2).

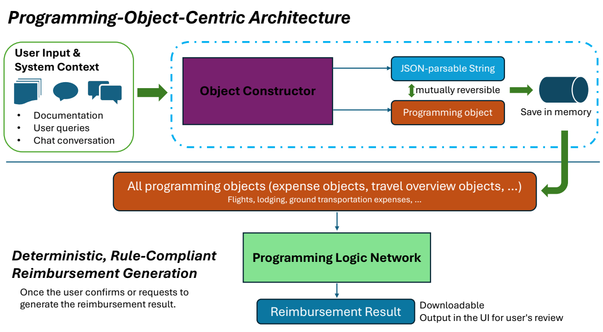

Figure 2: The Overview of POCA

3.3.1 Task-Specific Entity and Rule Modeling

The POCA underpins `TravelReimGPT` by operationalizing deterministic, policy-aware reasoning through programming representations of task-specific entities and reimbursement rules. These components collectively serve as the semantic and computational foundation of the system.

Entities and Rules In `TravelReimGPT`, the principal entity types include *Preapproval Report*, *Travel Overview*, *Flight*, *Lodging*, *Ground Transportation*, *Meal*, *Mileage*, *Registration*, *Incidental*, and *Ambiguous Expense*. These entities act as the foundational operands for evaluating policies and performing rule-based computations.

Reimbursement rules are encoded as logical formulas, specifying constraints and relationships among entities. These rules are typically defined by regulatory or funding bodies such as the National Institutes of Health (NIH). For instance, under NIH funding guidelines, lodging expenses must conform to daily rate limits set by the General Services Administration (GSA).

Dual Representations of Entities: Programming Objects and JSON-parsable Strings Unlike traditional entity extraction methods, `TravelReimGPT` represents entities as programming objects augmented with validated attributes and specific functionalities. Each entity category, such as Lodging and Flight, corresponds to a dedicated object class specifying entity-pertinent attributes and functions.

Each entity is realized through two fully interoperable and mutually reversible representations:

- ***Programming object***: an instantiated object derived from a class definition. It serves as the executable and operational form of the entity within a deterministic computational workflow for generating rule-compliant outcomes.
- ***JSON-parsable string***: a JSON-formatted string parsable by the programming engines, and well-suited for storage, transmission, and integration into the AI engines as part of augmented and contextual prompts.

To enable seamless bidirectional transformation between the two representations, all object class definitions must follow a standardized code design template that enforces mutual reversibility. This template requires that each class: 1) Supports initialization from a JSON-parsable string, 2) Implements a serialization function, and 3) Accommodates the structural and behavioral requirements specific to each entity type. With the corresponding class definition, a JSON-formatted string can be instantiated into a programming object; conversely, a programming object can be serialized into a JSON string. This dual representations of entities enable seamless data flow between the programming engine and the AI engine.

In sum, the dual representation strategy serves as a bridge between formal rule-based processing and natural language-based reasoning. It is crucial to `TravelReimGPT`'s capability to manage the complex, policy-governed task of automated travel reimbursement.

Rule Set: Programming Logic Network Each rule set is implemented as a programming logic network, which is a structured collection of functions and conditionals operating on programming objects. In `TravelReimGPT`, each logic network corresponds to a specific authority's policy framework, such as NIH or institutional policy. These networks leverage object attributes and inter-object relationships to apply conditions, perform numerical computations, and enforce hierarchical rule structures. The functions executed within the logic networks produce rule-compliant outcomes in a deterministic and interpretable manner. An illustrative example of a programming logic network is provided in the Appendix A.3. This design enables `TravelReimGPT` to perform policy-driven decision-making in complex reimbursement scenarios with transparency and precision.

3.3.2 Object Constructor (OC)

Reliable construction of entity objects is essential for automated travel reimbursement, which is numerically sensitive. POCA introduces the OC module for generating internally verified and executable programming objects. The OC leverages the AI engine's advanced code comprehension and generation capabilities and operates through a definition-guided, self-corrective loop. This process ensures that the resulting objects are well-structured, semantically coherent, and suitable for execution within formal logic networks. The overview of the OC process is illustrated in Figure 3).

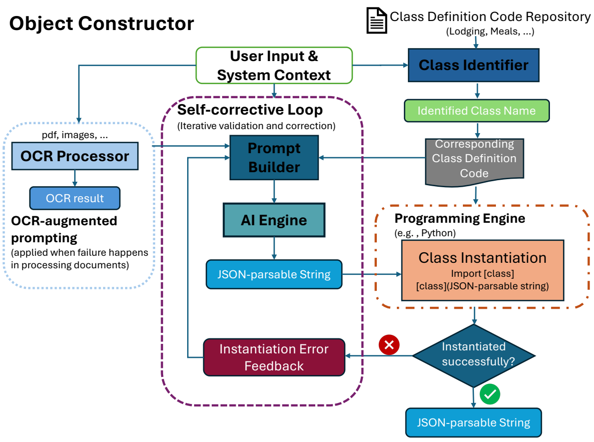

Figure 3: The Overview of OC

OC Process The OC extracts entity objects based on user inputs (e.g., queries, uploaded documents) and system context (e.g., previously processed objects and conversation history). The process consists of the following stages:

1. Object Type Detection Based on the inputs, the OC detects and identifies the types of involved entities (i.e., class name). For example, when multiple receipts are uploaded, the system determines the appropriate type of each expense receipt. The identifier is performed using prompting techniques with the AI engine, which analyzes both content and context to output the appropriate class name.

2. Definition-Guided, Self-Corrective Loop Once the target class is identified, the OC prompts the AI engine to generate a JSON-formatted string representing an instance of that class. This generation is guided by the corresponding class definition, which specifies the expected structure, attributes, and value constraints.

The generated JSON string is parsed and passed to the class constructor for object instantiation, where inner-verification modules are triggered to validate attribute types, enforce value constraints, and check for domain-specific logic. Notably, `TravelReimGPT` performs reconciliation check to ensure consistency in numerical fields (e.g., itemized and total amounts listed on receipts), thereby enhancing the amount accuracy of extracted expense objects.

If instantiation fails due to structural, type-related, or semantic inconsistencies, the system captures the specified error and feeds it back into the next prompt. This triggers a self-corrective loop, allowing the AI engine to iteratively refine its output until a valid, reliable, and executable object is produced.

Algorithm 1 Definition-Guided, Self-Corrective Loop

Require: Class initiator c and its definition code C, task prompt p, input q, maximum attempts M, supplementary context s

$k \leftarrow 1$
$CS \leftarrow$ `system`$\{C,\ p\} +$ `user`$\{q\}$ ▷ Seed first request with task instruction
while $k \leq M$ **do**
  $response \leftarrow$ `AI_engine`(CS) ▷ Get AI engine response
  $J \leftarrow$ `extract_JSON`$(response)$ ▷ Isolate candidate JSON
  try
    $c(J)$ ▷ Object instantiation attempt
    return J ▷ Success $\rightarrow$ output JSON
  except Error e
  $CS \leftarrow$ `assistant`$\{response\} +$ `user`$\{e,\ p,\ s\}$ ▷ Add error feedback
  $k \leftarrow k + 1$
end while
return `None` ▷ Fail after M attempts

Enhancing Object Construction with Inner Verification Modules The precision of constructed entities, particularly those involving numerics, is vital for reliable reimbursement. Even minor numerical errors can result in incorrect outcomes.

To improve the quality of extracted objects, `TravelReimGPT` incorporates verification modules that perform entity-specific quality checks during object construction. For expense-related entities, a key verification is the amount reconciliation check, which ensures that the sum of itemized charges matches the total amount. These values are extracted directly from receipts and cross-validated. If the verification fails, a self-corrective iteration is triggered to refine the extraction. Additional verification modules include attribute type and format checks, presence of required fields, and domain-specific logic enforcement, etc.

Each class definition implements dedicated inner-verification functions and specifies descriptive error messages for exceptions as instructions for self-correction.

In summary, the class definition code serves a dual purpose: 1) Guide the AI engine in generating structured, JSON-parsable strings; and 2) Enable robust runtime validation and correction of extracted results. The design template used in `TravelReimGPT` is provided in Appendix A.1.

Enhancing Object Construction with Supplementary Context In addition to leveraging error feedback, the OC also supports supplementary contextual signals to improve extraction accuracy and success rates. For document image inputs such as receipts, the AI engine may produce inaccurate outputs due to limitations in visual parsing. When repeated instantiation fails, the system invokes an OCR tool (PaddleOCR[2]) to extract contents of images. The OCR output is then injected into the AI prompt as supplementary context, enabling more accurate and consistent JSON generation.

3.3.3 Deterministic, Rule-Compliant Reimbursement Generation

All programming objects are stored in the database as JSON-parsable string representations. Once the user confirms or requests to generate the reimbursement result, `TravelReimGPT` invokes the corresponding programming logic networks and processes all objects to generate a rule-compliant reimbursement result. The result is presented in the UI and made available for download and review.

3.4 Dialogue Flow

At each dialogue turn, the user may input a query and optionally upload one or more documents. The backend of `TravelReimGPT` executes:

1) Think: Analyze the user's utterance to infer intent and determine the current step in the reimbursement procedure based on all processed data.

2) Act (if applicable): Invoke the OC to create or refine entity objects; trigger reimbursement generation if requested or all necessary documents are present; prepare all processed task-related results as augmented prompts.

3) Respond: Answer the user's query, summarise newly processed information and the current reimbursement status, notify the users of the next required action, and present the reimbursement output with a one-click download option in the UI if results are ready.

4 Evaluations and Experimental Setup

To assess the effectiveness of `TravelReimGPT`, we conducted a comparative evaluation against a baseline system that uses a traditional prompting technique, denoted as `BasePrt`. Specifically, the baseline employs task-instructive prompts with the `GPT-4.1` agent to perform the reimbursement process, including document processing and user query resolution. All reimbursement rules, task instructions, and document interpretation guidelines serve as prompts and knowledge to the AI agent.

The AI engine serves as the intelligence core of `TravelReimGPT`, its capability directly affects overall system performance. To explore the impact of model sophistication, we tested five variants using LLMs with varying levels of capacity, provided by OpenAI's API[3] with default settings: `ChatGPT-4o-latest`, `GPT-4o`, `GPT-4.1`, `GPT-4.1-mini`, and `GPT-4o-mini`, denoted as `TRG4olt`, `TRG4o`, `TRG4.1`, `TRG4.1mn`, and `TRG4omn`, respectively.

4.1 Evaluation Metrics

The evaluation employed a comprehensive set of performance metrics.

1) **Task success and accuracy.** Indicate the system's ability to correctly and fully complete the reimbursement task. Specifically, we verify the reimbursable amount of each expense item in the final reimbursement report and calculate the accuracy (*Reim. Acc.*); 100% accuracy indicates a fully successful reimbursement process.

2) **Total cost.** Total token usage and corresponding API charges incurred from all system-triggered calls (*Total Cost*, in USD).

3) **Processing efficiency.** Total system response time (*Total Latency*, in minutes) and the conversation rounds (*Turn Count*).

In addition, we report both system-level and user-rated evaluation metrics to capture the processing behavior of `TravelReimGPT` and reflect subjective user experience using the system.

System Behavior Metrics. 1) *OC Attempts* refers to the average number of iterations in OC triggered to build a valid object. 2) *Failed OC* counts the number of failed object constructions.

[2] https://github.com/PaddlePaddle/PaddleOCR

[3] https://platform.openai.com

3) *UI Response Length* indicates the average number of tokens per response shown in the UI.

User Satisfaction Metrics. User ratings were collected on a 5-point Likert scale (1 = poor, 5 = excellent). 1) *System Fluency* measures the smoothness of processing across the pipeline. 2) *Response Quality* captures the informativeness and clarity of responses. 3) *Intent Understanding* assesses how well the system interprets user intent and reimbursement status. 4) *Overall Satisfaction* reflects the user's overall impression of system performance.

4.2 Test Cases

We evaluated the systems using four representative real-world reimbursement cases collected from an institution, all of which are reimbursed through federal funding and must comply with both institutional policies and corresponding funding regulations. Each case was processed independently by all model variants as well as the baseline system.

Each case includes a mandatory preapproval document required for the reimbursement. The cases represent a range of realistic reimbursement scenarios with increasing complexity:

- Case **SA** (3 docs): A simple trip to San Antonio with parking and bus receipts.

- Case **LV** (8 docs): A trip to Las Vegas with flight, lodging, and multiple taxi/Uber receipts.

- Case **SD** (7 docs): A trip to San Diego with flight, multi-night lodging with cumbersome itemized charges, and multiple Uber receipts.

- Case **IN** (14 docs): A multi-leg trip to Dallas and Indianapolis with registration, multiple flights, lodging, Uber, and incidentals receipts.

5 Results

5.1 Overall Performance

The deployed version of `TravelReimGPT` employs `GPT-4.1` as its core AI engine. As shown in Figure 4, `TRG4.1` achieve 100% reimbursement accuracy, i.e., successfully generates rule-compliant reimbursement results of all test cases, while at a low cost (approximately $0.50 for a case with 14 documents) and a system processing time of only several minutes per case. Following the guide of the system, users upload required documents and provide key information over only several dialogue turns to obtain reliable rule-compliant reimbursement results. Furthermore, users could query to retrieve specific items if needed, e.g., "show the details of the uber expense on 5/12.". The conversation of a usage example is illustrated in the Appendix A.4.

5.2 Comparison with the Baseline

The baseline only successfully handled the simplest case and consistently made errors on relatively-complex cases, even when the relevant rules were explicitly injected into the system. Most of these errors stemmed from incorrect calculations performed by the LLM, despite the fact that the original amounts were correctly extracted.

Figure 4: Performance comparison of all model variants and the baseline on the test cases.

5.3 Performance of Variants

As shown in Figure 4 and Table 1, `TRG4olt` and `TRG4.1` achieve 100% reimbursement accuracy across all test cases. They also exhibit high system fluency, accurately infer user intent without explicit prompts, and generate detailed, user-friendly outputs—indicating strong robustness and reasoning capabilities. Due to its higher API cost, `TRG4olt` is more expensive.

	System			User Ratings			
	OC-Att	Fail	Len	Flu	Qual	Int	Sat
`TRG4.1`	1.09	0	443.2	5	5	5	5
`TRG4olt`	1.13	0	395.9	5	5	5	5
`TRG4o`	1.69	0	292.1	3	3	3	3
`TRG4.1mn`	1.33	1	275.9	3	3	2	2
`TRG4omn`	2.10	5	266.1	1	1	1	1

Table 1: System-level and user-rated evaluation metrics for `TravelReimGPT` variants. **OC-Att**: OC Attempts; **Fail**: Failed OC; **Len**: UI Response Length; **Flu**: System Fluency; **Qual**: Response Quality; **Int**: Intent Understanding; **Sat**: Overall Satisfaction.

TRG4o misclassified a mandatory resort fee in a lodging receipt as non-reimbursable unnecessary fee in Case **SD**. While it demonstrates high accuracy in object extraction, it often requires multiple attempts to construct objects successfully and shows limitations in intent understanding. Its overall performance is comparable to TRG4.1mn, but with significantly higher cost.

All variants successfully handle the simplest case. However, in more complex scenarios, mini-level variants fail to complete the reimbursement process. They struggle with language understanding, including intent inference, contextual reasoning, and producing coherent responses. As a result, users must provide explicit instructions and exhibit considerable patience, even when the necessary information is already implied in the uploaded documents. Their outputs are often brief and uninformative, and less user-friendly. TRG4omn performs the worst: it always fails to process documents, struggles with object construction, and is unable to proceed with the reimbursement workflow.

Overall, TravelReimGPT proves to be an effective and deployable system for accurate, user-friendly, and rule-compliant travel reimbursement automation. Powered by GPT-4.1, the TRG4.1 variant offers the best balance between cost and performance while ensuring reimbursement reliability.

6 Discussions

TravelReimGPT is a task-specific, user-friendly, and end-to-end system for automating travel reimbursements. It generates rule-compliant results from free-form user inputs and documents via multi-turn dialogue, with a focus on usability, accuracy, and policy adherence. At the core lies the POCA, which enables object-level symbolic reasoning and deterministic rule enforcement—offering a more scalable and robust alternative to traditional parameter-level prompting. The system is modular and flexible. The OC module effectively builds executable entity objects from user inputs and contextual information, including conversation history and prior system state. These objects are validated, self-corrective, and enriched with attributes and functions to reflect structural dependencies, improving robustness and precision in data extraction and rule application.

Rather than solving general reasoning problems, TravelReimGPT is a task-specific framework, and optimized for effective and efficient task automation. Essential to this framework are clearly defined object classes and structured programmatic logical networks, which collectively underpin the system's ability to enforce complex rule compliance effectively. We provide detailed design templates outlining the critical components required for establishing well-structured object classes and logical networks. Leveraging OpenAI's advanced reasoning LLMs (specifically, o1 and o3), we systematically generated preliminary code structures for object class definitions and policy networks, subsequently refining and optimizing these through manual reviews and improvements.

Performance evaluations demonstrate TravelReimGPT's strong capability in handling representative reimbursement scenarios. Exceptional expense cases occasionally occur. In accordance with reimbursement policy, exceptions require manual communication and approval. Such exceptions fall outside the scope of the automated system and are therefore managed separately.

Furthermore, travel reimbursement represents a broader category of rule-compliant tasks, wherein explicit, predefined rules are systematically applied to documents and related data to yield compliant outcomes, similar to tasks in tax reporting and auditing. Consequently, our proposed framework not only provides a practical, user-friendly, and efficient solution specifically tailored to travel reimbursements but also offers a generalizable model for automating a wide range of similarly structured, rule-based compliance tasks.

7 Conclusions

This work introduces TravelReimGPT, a user-centric end-to-end AI system designed to efficiently automate business-travel reimbursement processes with free-form and document inputs. Leveraging advanced LLMs and the novel POCA, TravelReimGPT accurately parses documentation and inputs and generate rule-compliant reimbursement results. Our approach notably reduces cognitive and numerical burdens in administrative workflows. POCA serves as a versatile architecture, extendable to various high-reuse-rule-intensive task automation such as budgeting and auditing. TravelReimGPT significantly enhances workflow efficiency and reliability, highlighting the transformative impact of cognitive AI.

Limitations

The proposed TravelReimGPT framework, being task-specific, inherently limits its generalizability for broad, general-purpose use. Although travel reimbursement cases are similar, the evaluation in this study is limited due to the small size of evaluated test cases.

Our study does not involve of automatic parsing and programming of complex rules. While our symbolic representation of entities and rules is meticulously designed and optimized for efficiency, the class definitions and logic networks remain predefined and static and necessitates manual intervention in the design. Future developments will target automating this aspect to reduce manual efforts and further enhance the scalability and broader applicability of our framework, lower the technical barrier to making it accessible and designable by common users.

Due to inherent latency issues and the substantial computational resources required to localize open-source LLMs, we did not conduct related experiments. As a result, performance data for TravelReimGPT using localized open-source LLMs, such as LLaMA, remains unexplored and constitutes a valuable direction for future research.

Moreover, the framework depends on proprietary LLMs which introduces concerns regarding reproducibility, high usage costs, and unequal access for different user groups. Finally, while automation improves efficiency, over-reliance may reduce human oversight in edge cases requiring nuanced judgment.

Ethics Statement

The use of OpenAI's LLMs was conducted via API access under a contractual agreement that explicitly prohibits data collection or retention. As a result, no personally identifying information was collected by the language model provider. This study complies with all relevant data privacy regulations and ethical research guidelines. We used real-world cases for evaluation. While these may contain information originating from actual individuals, no manual anonymization or offensive content filtering was applied. All data was handled securely within a local research environment and was not exposed to any third-party services beyond the LLM API, under strict data handling protocols. All case data was obtained with explicit consent from the individuals involved. No personal data was used without permission.

References

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

M. Boese. 2025. The real revolution in expense technology makes your daily work easier. Accessed: 2025-05-14.

Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, and 1 others. 2021. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*.

Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, and 1 others. 2024. A survey on evaluation of large language models. *ACM transactions on intelligent systems and technology*, 15(3):1–45.

Fengxiang Cheng, Haoxuan Li, Fenrong Liu, Robert van Rooij, Kun Zhang, and Zhouchen Lin. 2025. Empowering llms with logical reasoning: A comprehensive survey. *arXiv preprint arXiv:2502.15652*.

Meng Fang, Shilong Deng, Yudi Zhang, Zijing Shi, Ling Chen, Mykola Pechenizkiy, and Jun Wang. 2024. Large language models are neurosymbolic reasoners. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pages 17985–17993.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

GBTA. 2023. Gbta 2024 business travel index outlook. Accessed: 2024-05-14.

GitHub. 2025. GitHub Copilot. `https://github.com/features/copilot`. Version 1.192, accessed 2025-05-14.

Juraj Gottweis, Wei-Hung Weng, Alexander Daryin, Tao Tu, Anil Palepu, Petar Sirkovic, Artiom Myaskovsky, Felix Weissenberger, Keran Rong, Ryutaro Tanno, and 1 others. 2025. Towards an ai co-scientist. *arXiv preprint arXiv:2502.18864*.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, and 1 others. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 3(4):6.

Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*.

Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of hallucination in natural language generation. *ACM computing surveys*, 55(12):1–38.

M. Johnston. 2025. The agentic workforce: How ai is redefining back-office efficiency and strategy. Accessed: 2025-05-14.

Haoqiang Kang and Xiao-Yang Liu. 2023. Deficiency of large language models in finance: An empirical examination of hallucination. *arXiv preprint arXiv:2311.15548*.

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. 2023. Chameleon: Plug-and-play compositional reasoning with large language models. *Advances in Neural Information Processing Systems*, 36:43447–43478.

Tula Masterman, Sandi Besen, Mason Sawtell, and Alex Chao. 2024. The landscape of emerging ai agent architectures for reasoning, planning, and tool calling: A survey. *arXiv preprint arXiv:2404.11584*.

McKinsey & Co. 2023. The state of ai in 2023: Generative ai's breakout year. Accessed: 2025-05-14.

Microsoft. 2025. Microsoft Copilot. `https://copilot.microsoft.com`. Build 25.9.123, accessed 2025-05-14.

Erik Miehling, Karthikeyan Natesan Ramamurthy, Kush R Varshney, Matthew Riemer, Djallel Bouneffouf, John T Richards, Amit Dhurandhar, Elizabeth M Daly, Michael Hind, Prasanna Sattigeri, and 1 others. 2025. Agentic ai needs a systems theory. *arXiv preprint arXiv:2503.00237*.

Aske Plaat, Max van Duijn, Niki van Stein, Mike Preuss, Peter van der Putten, and Kees Joost Batenburg. 2025. Agentic large language models, a survey. *arXiv preprint arXiv:2503.23037*.

Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, H Han, Sevien Schulhoff, and 1 others. 2024. The prompt report: A systematic survey of prompting techniques. *arXiv preprint arXiv:2406.06608*, 5.

SDO CPA. 2025. Best expense management software. Accessed: 2025-05-14.

Y. Shaked. 2025. The power of automation: How businesses can work smarter, not harder. Accessed: 2025-05-14.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36:38154–38180.

Åsne Stige, Efpraxia D Zamani, Patrick Mikalef, and Yuzhen Zhu. 2024. Artificial intelligence (ai) for user experience (ux) design: a systematic literature review and future research agenda. *Information Technology & People*, 37(6):2324–2352.

Arun James Thirunavukarasu, Darren Shu Jeng Ting, Kabilan Elangovan, Laura Gutierrez, Ting Fang Tan, and Daniel Shu Wei Ting. 2023. Large language models in medicine. *Nature medicine*, 29(8):1930–1940.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, and 1 others. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Ke Wang, Houxing Ren, Aojun Zhou, Zimu Lu, Sichun Luo, Weikang Shi, Renrui Zhang, Linqi Song, Mingjie Zhan, and Hongsheng Li. 2023. Mathcoder: Seamless code integration in llms for enhanced mathematical reasoning. *arXiv preprint arXiv:2310.03731*.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, and 1 others. 2024a. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024b. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, and 1 others. 2022a. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022b. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, and 1 others. 2025. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 68(2):121101.

Haoyi Xiong, Zhiyuan Wang, Xuhong Li, Jiang Bian, Zeke Xie, Shahid Mumtaz, Anwer Al-Dulaimi, and Laura E Barnes. 2024. Converging paradigms: The synergy of symbolic and connectionist ai in llm-empowered autonomous agents. *arXiv preprint arXiv:2407.08516*.

Jundong Xu, Hao Fei, Liangming Pan, Qian Liu, Mong-Li Lee, and Wynne Hsu. 2024a. Faithful logical reasoning via symbolic chain-of-thought. *arXiv preprint arXiv:2405.18357*.

Ziwei Xu, Sanjay Jain, and Mohan Kankanhalli. 2024b. Hallucination is inevitable: An innate limitation of large language models. *arXiv preprint arXiv:2401.11817*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

Di Zhang, Xiaoshui Huang, Dongzhan Zhou, Yuqiang Li, and Wanli Ouyang. 2024a. Accessing gpt-4 level mathematical olympiad solutions via monte carlo tree self-refine with llama-3 8b. *arXiv preprint arXiv:2406.07394*.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, and 1 others. 2024b. Aflow: Automating agentic workflow generation. *arXiv preprint arXiv:2410.10762*.

Yuqi Zhu, Shuofei Qiao, Yixin Ou, Shumin Deng, Shiwei Lyu, Yue Shen, Lei Liang, Jinjie Gu, Huajun Chen, and Ningyu Zhang. 2024. Knowagent: Knowledge-augmented planning for llm-based agents. *arXiv preprint arXiv:2403.03101*.

A Appendix

A.1 Class definition code design template used in `TravelReimGPT`

Here is the class definition code design template used in POCA for `TravelReimGPT` with Python as the core of the programming engine.

```
class ClassName:
    class_name = "ClassName"
    # Comments specifying the formats
        and requirements of
        json_parsable_string, which is
        to guide AI engine
    def __init__(self,
        json_parsable_string: str):
        self.json_data = json.loads(
            json_parsable_string)
        # Attribute extraction (example)
        self.attribute1 = self.json_data
            .get("attribute1", None)
        # Add more attributes as needed
        ...
        self.__verification__()
        # Could add any functions here

    def __verification__(self):
        """
        Verification module to ensure
            the content of the extracted
             object is well-structured,
            accurate, and consistent.
            Must raises specific errors
            to guide AI engine debugging
             and correction.
        """
        # Example 1: Type and format
            checks of attributes and
            properties
        try:
            self.date = datetime.
                strptime(self.date, "%Y
                -%m-%d")
        except Exception:
            raise ValueError("Date
                format must be YYYY-MM-
                DD. Please correct the
                result.")

        # Example 2: Amount
            reconciliation check,
            ensuring the accuracy of
            extracted amounts
        if hasattr(self, "
            itemized_amounts") and
            hasattr(self, "total_amount"
            ):
            computed_total = sum(self.
                itemized_amounts)
            if computed_total != self.
                total_amount:
                raise ValueError("
                    Something wrong in
                    Amount extraction:
                    please check whether
                     the itemized
                    amounts are correct
                    and complete! And
                    check whether the
                    total amount is
                    correct!")

        # Add other custom verification
            modules as needed...
        # e.g., required fields,
            allowable value ranges,
            enumeration checks, etc.
    def to_json(self):
        # serialization function here
```

A.2 An example of object class: `Ground Transportation`

```
class GroundTransportation:
    """
    Represents a ground transportation
        expense, including details such
        as the mode of transport,
    start and end locations, itemized
        charges, trip distance, and
        optional departure and arrival
        dates and times.

    This class is suitable for expenses
        like Uber, Lyft, taxi, bus, and
        train rides.
```

```
    If reimbursed for mileage, you
        should use the Mileage class
        instead.

    Attributes:
        expense_type (str): The type of
            the expense, fixed as "
            Ground Transportation".
        description (str): A description
             of the expense.
        itemized_charges (List[Dict[str,
             Any]]): List of itemized
            charges, each containing an
            item and its charge. [{"item
            ": "Base Fare", "charge":
            10.00}, ...] If no tip is
            found in the Uber/Lyft/Taxi
            receipt, it should be {"item
            ": "Tip", "charge": 0.0}.
        expense_date (str): The date
            when the expense was made,
            in YYYY-MM-DD format.
        amount (float): The total amount
             of the expense.
        tip (float): The tip expense.
        merchant (str): The
            transportation provider, e.g
            ., Uber, Lyft.
        mode (str): Mode of transport, e
            .g., "bus", "train", "
            rideshare", "taxi".
        start_location (str): The
            starting location.
        end_location (str): The ending
            location.
        distance (float): Distance in
            miles, if available.
        departure_date_time (str):
            Departure date in YYYY-MM-DD
             HH:MM AM/PM format, if
            available.
        arrival_date_time (str): Arrival
             date in YYYY-MM-DD HH:MM AM
            /PM format, if available.
        in_foreign (bool): Indicates
            whether the ground
            transportation expenses were
             incurred outside the USA.
            Default: False.
        currency (str): the Standard
            Currency Code: "EUR", "JPY",
             "CHF", "CAD", .... Default:
             "". Empty means USD.
    """

    expense_type = "Ground
        Transportation"

    def __init__(self, input_json: str):
        """
        Initializes a
            GroundTransportation object
            from a JSON string.
        """
        try:
            data = json.loads(input_json
                )
        except json.JSONDecodeError as e
            :
            raise ValueError(f"Invalid
                JSON input: {e}")

        self.description: str = data.get
            ("description", "")
        self.itemized_charges: List[Dict
            [str, Any]] = data.get("
            itemized_charges", [])
        self.expense_date: str = data.
            get("expense_date", "")
        self.amount: float = data.get("
            amount", 0.0)
        self.tip: float = data.get("tip"
            , 0.0)
        self.merchant: str = data.get("
            merchant", "")  #
            Transportation provider
        self.mode: str = data.get("mode"
            , "")           # Mode of
            transport
        self.start_location: str = data.
            get("start_location", "")  #
             Starting location
        self.end_location: str = data.
            get("end_location", "")
                 # Ending location
        self.distance: float = data.get(
            "distance", 0.0)  # Distance
             in miles
        self.non_tip_charge = round(self
            .amount - self.tip, 2)

        # Departure and arrival dates
            and times
        self.departure_date_time: str =
            data.get("
            departure_date_time", "")
        self.arrival_date_time: str =
            data.get("arrival_date_time"
            , "")
        self.in_foreign: bool = data.get
            ("in_foreign", False)
        self.currency: str = data.get("
            currency", "")

        self.__verification__()

        self.expense_items = self.
            get_expense_items()

    def __verification__(self):
        try:
            datetime.strptime(self.
                expense_date, "%Y-%m-%d"
                )
        except:
            raise ValueError(f"Invalid
                date format in
                expense_date: {self.
                expense_date}. Expected
                YYYY-MM-DD.")

        # Check if each itemized charge
            has "item" and "charge"
        for entry in self.
            itemized_charges:
            if "item" not in entry or "
                charge" not in entry:
                raise ValueError(f"
                    Missing required
```

```
                            keys 'item' or '
                            charge' in entry: '{
                            entry}'")

        # Validate that the total
            matches the amount
        total_itemized = sum(charge.get(
            "charge", 0.0) for charge in
             self.itemized_charges)
        if not abs(total_itemized - self
            .amount) < 1e-6:  # Allowing
             for floating-point
            precision issues
            raise ValueError(
                f"The total of itemized
                    charges ({
                    total_itemized})
                    does not match the
                    amount ({self.amount
                    }). Might miss some
                    items. Please check
                    the completeness of
                    all items."
            )

    def get_expense_items(self):
        expense_items = [{"
            non_tip_charge": self.
            non_tip_charge, "tip": self.
            tip, "expense_date":self.
            expense_date, "item": self.
            expense_type}]
        return(expense_items)

    def to_json(self) -> str:
        """
        Serializes the
            GroundTransportation object
            to a JSON string, excluding
            attributes with None or
            empty values.

        Returns:
            str: JSON string
                representation of the
                object.
        """
        data = {
            "expense_type": self.
                expense_type,
            "description": self.
                description,
            "itemized_charges": self.
                itemized_charges if self
                .itemized_charges else
                None,
            "expense_date": self.
                expense_date,
            "amount": self.amount,
            "tip": self.tip,
            "non_tip_charge": self.
                non_tip_charge,
            "merchant": self.merchant,
            "mode": self.mode,
            "start_location": self.
                start_location,
            "end_location": self.
                end_location,
            "distance": self.distance if
                 self.distance else None
            ,
            "departure_date_time": self.
                departure_date_time,
            "arrival_date_time": self.
                arrival_date_time,
            "in_foreign": self.
                in_foreign,
        }

        # Exclude keys with None or
            empty string values
        filtered_data = {k: v for k, v
            in data.items() if v not in
            (None, "", [])}

        return json.dumps(filtered_data,
             indent=4)

    def __repr__(self):
        return f"{self.__class__.
            __name__}({self.to_json()})"

    def get_date(self):
        return(self.expense_date)
```

A.3 An example of programming logic network

An example of a programming logic network implementing hybrid rule compliance is presented. Soft rules refer to descriptive or context-dependent rules that are challenging to formalize symbolically. These rules typically govern the allowability of expense items and require the application of an AI engine to assess compliance. In contrast, all deterministic rules—which can be clearly defined and codified—are implemented using symbolic logic.

```
class PolicyNetwork:
    """Grant policy engine adhering to
        the requested template.

    Parameters
    ----------
    policy_detail : dict
        Attributes defining the fund /
            grant policy.
    gsa_rate : module-like
        Must expose 'lookup_lodge_rate'
            and 'lookup_meal_ie_rate'
            helpers.
    get_allowable_function : coroutine
        'await get_allowable_function(
            expense_json, rule_str,
            llm_helper)' to
        evaluate soft rules via LLM.
    """
    #
    #  Construction helpers
    #

    def __init__(
        self,
        policy_detail: Dict[str, Any],
        gsa_rate: Any,
        get_allowable_function: Callable
```

```
            [[Dict[str, Any], str,
            Callable | None], Any],
    ) -> None:
        # Core policy attributes
        self.policy_id = policy_detail.
            get("Fund_ID")
        self.fund_source = policy_detail
            .get("Fund Source")
        self.allowance_foreign_travel =
            policy_detail.get(
            "foreign_travel_allowance",
                False
        )
        self.allowance_domestic_travel =
             policy_detail.get(
            "domestic_travel_allowance",
                 False
        )
        self.allowance_tip =
            policy_detail.get("
            Allowance_Tip", True)
        self.interpretative_rules =
            policy_detail.get("
            InterpretativeRulesDict",
            {})

        #  External dependencies
        self._gsa_rate = gsa_rate
        self._get_allowable_function =
            get_allowable_function

    #
    #  Public helpers
    #
    def get_allowance_travel(self,
        all_objects: Dict[str, Any]) ->
        bool:
        # Return *True* if the whole
            trip is reimbursable under
            this policy.
        travel_overview = all_objects.
            get("travel_overview")
        if not travel_overview:
            return False
        if travel_overview.travel_type
            == "international":
            return self.
                allowance_foreign_travel
        return self.
            allowance_domestic_travel

    #  Reimbursement engine (template
        compliant)

    async def
        generate_reimbursement_details(
        self,
        all_objects: Dict[str, Any],
        call_ai_engine: Callable | None
            = None,
    ) -> Dict[str, Any]:
        # Entry point builds and returns
             the nested reimbursement
            details.
        if not self.get_allowance_travel
            (all_objects):
            return {}

        # Work on a copy because we
            mutate 'all_objects' later
            on
        objects: Dict[str, Any] = dict(
            all_objects)
        objects.pop("travel_overview",
            None)

        reimbursement_details: Dict[str,
             Any] = {}

        for obj_type, typed_objs in
            objects.items():
            reimbursement_details[
                obj_type] = {}
            for obj_id, obj in
                typed_objs.items():
                evaluated_items: List[
                    Dict[str, Any]] = []

                for expense_item in obj.
                    expense_items:
                    is_allowable,
                        abs_limit,
                        remark,
                        need_manual =
                        await self.
                        _eval_expense_item
                        (
                        expense_item,
                            obj,
                            call_ai_engine

                    )
                    reim_result = {"
                        incurred_expense
                        ": expense_item,
                        "is_allowable":
                            is_allowable
                            ,
                        "
                            abs_policy_limit
                            ": abs_limit
                            ,
                        "
                            need_manual_flag
                            ":
                            need_manual,
                        "remark": remark
                            ,
                        }
                    evaluated_items.
                        append(
                        reim_result
                    )
                reimbursement_details[
                    obj_type][obj_id] =
                    evaluated_items

        return reimbursement_details

    #
    #  Internal evaluators and dispatch
    #

    async def _eval_expense_item(
        self,
        expense_item: Any,
        obj: Any,
        call_ai_engine: Callable | None,
    ) -> Tuple[bool, float | None, str,
        bool]:
```

```
        """Route expense-item to the
           appropriate evaluator and
           normalize output."""
        dispatch_map = {
            "Ambiguous Expense": self.
                _eval_ambiguous,
            "Lodging": self.
                _eval_lodging,
            "Meals": self._eval_meals,
            "Ground Transportation":
                self.
                _eval_groundtransport,
        }
        handler = dispatch_map.get(
            expense_item.expense_type,
                self.
                _eval_generic_soft_rule
        )
        return await handler(
            expense_item, obj,
            call_ai_engine)

    #  Expense-type handlers

    async def _eval_ambiguous(
        self, expense_item: Any, obj:
            Any, *_: Any
    ) -> Tuple[bool, None, str, bool]:
        amount = round(getattr(
            expense_item, "amount", 0.0)
            , 2)
        remark = f"Amount: {amount}.
            Need manual process for
            ambiguous expense!"
        return False, None, remark, True

    async def _eval_lodging(
        self, expense_item: Any, obj:
            Any, *_: Any
    ) -> Tuple[bool, float | None, str,
        bool]:
        # Example use of obj - group
            booking could lower cap, etc
            .

        city, state = obj.location_city,
             obj.location_state
        date_str = expense_item.date
        try:
            d = datetime.strptime(
                date_str, "%Y-%m-%d")
            rate, msg = self._gsa_rate.
                lookup_lodge_rate(city,
                state, d.year, d.month)
            if msg:
                remark_parts.append(f"{
                    date_str}: {msg}")
            return True, rate, remark,
                False
        except Exception as exc:
            return True, None, f"GSA
                lookup failed: {exc}",
                True

    async def _eval_meals(
        self, expense_item: Any, obj:
            Any, *_: Any
    ) -> Tuple[bool, float | None, str,
        bool]:
        city, state = expense_item.
            location_city, expense_item.
            location_state
        d = datetime.strptime(
            expense_item.expense_date, "
            %Y-%m-%d")
        meal_type = obj.meal_type.
            capitalize()
        try:
            rates, msg = self._gsa_rate.
                lookup_meal_ie_rate(city
                , state, d.year, d.month
                )
            abs_limit = rates.get(
                meal_type)
            remark = msg or ""
            return True, abs_limit,
                remark, False
        except Exception as exc:
            return True, None, f"Meal
                rate lookup failed: {exc
                }", True

    async def _eval_groundtransport(
        self, expense_item: Any, obj:
            Any, *_: Any
    ) -> Tuple[bool, float | None, str,
        bool]:
        if obj.in_foreign:
            return True, None, f"No
                limit for foreign ground
                 transportation", False
        return True, 65, f"Cap 65",
            False

    async def _eval_generic_soft_rule(
        self,
        expense_item: Any,
        obj: Any,
        call_ai_engine: Callable | None,
    ) -> Tuple[bool, None, str, bool]:
        etype = expense_item.
            expense_type
        rule = self.interpretative_rules
            .get(etype)
        if not rule:
            return True, None, "No soft
                rule defined; default
                allowable", False
        try:
            ai_resp = await self.
                _get_allowable_function(
                expense_item.to_json(),
                rule, call_ai_engine)
            if ai_resp.get("return_flag"
                ):
                parsed = json.loads(
                    ai_resp["
                    final_result"])
                return parsed.get("
                    is_allowable", True)
                    , None, parsed.get("
                    explanation", ""),
                    False
            return False, None, "Soft
                rule evaluation returned
                 no flag", True
        except Exception as exc:
            return False, None, f"Soft
                rule exception: {exc}",
                True
```

A.4 A usage case example on `TravelReimGPT`

Here presents a real-world user-`TravelReimGPT` dialogue for processing a reimbursement case. The conversation demonstrates the complete workflow, including document uploads, information extraction, eligibility confirmation, and expense calculation. Note: City airport names have been anonymized in accordance with the review policy.

User: Hi

TravelReimGPT: Welcome! I'm TravelReimGPT, here to guide you through the travel reimbursement process and generate accurate reimbursement results.
To get started, please remember the major uploading procedure: upload RTA -> upload transportation/flights which start and end the travel -> upload all related receipts.
There is no COUPA RTA - Pre-Approval report. Please upload the pre-approval report to proceed.
Summary of Actions Required:
1. Please upload your COUPA RTA Pre-Approval report.
2. Once the pre-approval report is uploaded, I will guide you to the next steps regarding transportation and other required documents.

User: RTA [Uploads `RTA.pdf`]

TravelReimGPT: [File `RTA.pdf` processed successfully!] Thank you for uploading your COUPA RTA - Pre-Approval report. We have successfully received and processed your RTA.pdf.
Major reimbursement uploading procedure:

- Upload RTA (Completed)
- Upload transportation/flights which start and end the travel
- Upload all related receipts

Next Required Step: To proceed, please specify how you started and ended your actual travel for your Las Vegas trip and upload related documentation as evidence (such as airline tickets, boarding passes, flight receipts, train or bus tickets, or, if you drove, mileage documentation).
For your trip, the pre-approval includes a round-trip flight to Las Vegas. Please provide the following: Your flight ticket, boarding pass, or airline receipt that covers both your departure and return legs, showing dates and times.
Summary of Immediate Actions Required:

- Upload flight documentation/receipts (e.g., tickets, boarding passes, or receipts that confirm the travel dates/times).
- If you used other transportation for any part of your trip (e.g., mileage, train, shuttle), please specify and upload the corresponding documentation.

Once transportation documents verifying your actual travel dates/times are uploaded, we'll move forward to the next required step.

User: Flights start and end the travel. [Uploads `flight_receipt.pdf`]

TravelReimGPT: [File `flight_receipt.pdf` processed successfully!]
Thank you for providing your flight ticket documentation. Your itinerary successfully confirms both the start and end of your travel:
Departure: [AIRPORT] → Las Vegas (LAS), 2025-03-02, 7:15 PM → [H:M] PM
Return: Las Vegas (LAS) → [AIRPORT], 2025-03-04, [H:M] PM → 10:55 PM
We have retrieved your actual travel dates and times. Here is the extracted actual travel overview:


```
{
  "travel_start_date_and_time": "2025-03-02
      07:15 PM",
  "travel_end_date_and_time": "2025-03-04 10:55
      PM",
  "transportation_to_destination": "Flight from
      [AIRPORT] to Las Vegas (LAS), Southwest
      Airlines Flight",
  "transportation_return_origin": "Flight from
      Las Vegas (LAS) to [AIRPORT], Southwest
      Airlines Flight",
  "travel_destination_city_states": [{"city": "
      Las Vegas", "state": "NV"}]
}
```


Please confirm whether the above information accurately reflects your travel, or let us know if anything needs to be corrected.
Based on your travel dates, the reimbursable meals are: 2025-03-02: dinner; 2025-03-03: breakfast, lunch, dinner; 2025-03-04: breakfast, lunch, dinner.
Please confirm if this matches your meal eligibility for the trip.
Next Steps: Please confirm: Is the actual travel overview and these meal eligibility details correct?
Lodging/Hotel: Since your pre-approval includes lodging, please upload your hotel/lodging receipt for the Las Vegas stay.

Incidentals/Ground Transportation: If you incurred any ground transportation or incidental expenses (e.g., airport shuttles, taxis, etc.), please upload the relevant receipts.

User: Confirmed. Lodging uploaded. [Uploads `hotel(15).pdf`]

TravelReimGPT: [File `hotel(15).pdf` processed successfully!]
Thank you for confirming your travel details and for uploading your lodging receipt.
Here's what has been verified so far:

- Pre-Approval (RTA): Provided and matches your travel overview
- Transportation (Flight): Documentation provided and verified—matches actual travel dates/times
- Lodging: Receipt uploaded and processed—matches travel dates (check-in: 2025-03-02, check-out: 2025-03-04)
- Travel Destination Logs: Confirmed as accurate by you

Next Steps:

1. Meals Reimbursement: Your pre-approval covered meal expenses. You are eligible for:
   - 2025-03-02: Dinner
   - 2025-03-03: Breakfast, Lunch, Dinner
   - 2025-03-04: Breakfast, Lunch, Dinner

   You are eligible for: Breakfast: 2, Lunch: 2, Dinner: 3

   Standard per diem meal rate: Breakfast: $12, Lunch: $17, Dinner: $30

   Total reimbursement amount for meals: $148

   No meal receipts required under the standard per diem rate.

   Please confirm: Do you agree to use the standard per diem meal rate for meal reimbursement? If you want to use the GSA meal rate (which requires receipts), or you have any special cases, let us know.

2. Incidentals / Ground Transportation: Your pre-approval listed $150 for incidentals/ground transportation. Please specify: Did you incur expenses such as taxis, shuttles, rideshares (Uber/Lyft), airport parking, or other ground transportation? If yes, please upload the receipts for those expenses. If not, please confirm so we can mark this as not claimed.

Summary of Priority Next Steps:

- Confirm use of the standard per diem meal rate or inform us if you wish to use another method.
- Specify and upload documentation for any incidentals/ground transportation, or confirm if there are none to claim.

Once these are addressed, we will review for any additional missing items and then proceed to the final documentation summary and fund code check.

User: Confirm. Uber receipts. [Uploads `Uber.pdf`, `Uber2.pdf`, `taxi_to_hotel.pdf`, `taxi_to_meeting.pdf`, `taxi_to_meeting2.pdf`]

TravelReimGPT: Thank you for providing and confirming your ground transportation receipts. Here's a summary of what has been processed and verified:
Processed Reimbursement Documentation

1. Pre-Approval Report (RTA)

   Provided and matched for the Las Vegas trip.

2. Transportation (Start & End Travel)

   Flight: Confirmed with itinerary—both departure and return flight documentation are present and valid.

3. Lodging

   Hotel Receipt: Uploaded and confirmed for the entirety of the trip (Check-in: 2025-03-02, Check-out: 2025-03-04).

4. Meals

   Standard per diem rate confirmed and applied.

   Eligible meals:
   - 2025-03-02: Dinner
   - 2025-03-03: Breakfast, Lunch, Dinner
   - 2025-03-04: Breakfast, Lunch, Dinner

   Total per diem meal reimbursement: $148

   No meal receipts required.

5. Incidentals / Ground Transportation

   Receipts for rideshare and taxi services: All ground transportation receipts received and processed:

   - Uber to [AIRPORT] (03/02)
   - Taxi from Las Vegas airport to hotel (03/02)
   - Taxi in Las Vegas to meetings (03/03, 03/04)
   - Taxi/Uber to Las Vegas airport (03/04)

   These match expected incidentals from your pre-approval.

What's Next:

1. Did you incur any other reimbursable expenses?

   Based on your RTA and actual expenses, the main categories (airfare, lodging, meals, ground transportation) are addressed. If you have any of these additional base expenses to include (such as parking, baggage fees, or hotel-city shuttle not yet submitted), please specify and upload the relevant receipts now.

2. Do you have all required documentation uploaded?

   Please confirm that all documents necessary for your reimbursement claim are now uploaded. If there is anything else (hotel invoice addenda, conference registration, or other receipts), please provide them now.

3. Fund Code Confirmation: The fund code in your pre-approval is: 36680. Would you like to use this code for the entire reimbursement, or provide a different/additional fund code(s)?

Summary of Immediate Actions Required:

- Confirm if all required documents are now uploaded, or upload any additional receipts for other incurred expenses.
- Confirm the fund code(s) you want to use for your reimbursement.

Once confirmed, I will prepare your reimbursement report for review.

User: All documents uploaded. I would like to use fund code 50032

TravelReimGPT: Thank you for uploading all required documentation and confirming that you would like to apply fund code 50032 for your travel reimbursement. Below is a clear summary of your actual reimbursed expenses and amounts, processed strictly by institutional and grant policy for fund code 50032.
Las Vegas Travel Reimbursement Summary (Fund Code 50032)

Category	**Amount (USD)**
Flights (Round Trip)	$793.53
Lodging/Hotel (Room Rate)	$318.00
Lodging/Hotel (Tax)	$42.54
Meals (Per Diem Rate)	$148.00
Ground Transportation	$112.15
Tips	$9.62
Total	**$1,423.84**

What Happens Next:

- Once you confirm, your submission will be finalized for review and further processing by the travel reimbursement team.
- If any detail needs correction, please specify what should be changed.

Summary of Immediate Actions for You:

- Review the reimbursement summary and amounts above.
- Confirm accuracy and intent to submit using fund code 50032 (or request changes).

  Supplementary: You asked for downloading reimbursement files.

[A button for downloading the result is located below the output box.]