Skill Discovery for Software Scripting Automation via Offline Simulations with LLMs

Paiheng Xu[1]*, Gang Wu[2]†, Xiang Chen[2]†, Tong Yu[2], Chang Xiao[2], Franck Dernoncourt[2], Tianyi Zhou[1], Wei Ai[1], Viswanathan Swaminathan[2]

[1]University of Maryland, College Park, [2]Adobe Research

Abstract

Scripting interfaces enable users to automate tasks and customize software workflows, but creating scripts traditionally requires programming expertise and familiarity with specific APIs, posing barriers for many users. While Large Language Models (LLMs) can generate code from natural language queries, runtime code generation is severely limited due to unverified code, security risks, longer response times, and higher computational costs. To bridge the gap, we propose an offline simulation framework to curate a software-specific skillset—a collection of verified scripts—by exploiting LLMs and publicly available scripting guides. Our framework comprises two components: (1) task creation, using top-down functionality guidance and bottom-up API synergy exploration to generate helpful tasks; and (2) skill generation with trials, refining and validating scripts based on execution feedback. To efficiently navigate the extensive API landscape, we introduce a Graph Neural Network (GNN)-based link prediction model to capture API synergy, enabling the generation of skills involving underutilized APIs and expanding the skillset's diversity. Experiments with Adobe Illustrator demonstrate that our framework significantly improves automation success rates, reduces response time, and saves runtime token costs compared to traditional runtime code generation. This is the first attempt to use software scripting interfaces as a testbed for LLM-based systems, highlighting the advantages of leveraging execution feedback in a controlled environment and offering valuable insights into aligning AI capabilities with user needs in specialized software domains.

1 Introduction

Scripting interfaces in software applications play a pivotal role in extending the capabilities of software beyond their standard functionalities. They enable users to automate repetitive tasks, customize workflows, and integrate applications with other systems (Ousterhout, 1998). Prominent software like Adobe Illustrator and Adobe Photoshop support scripting through ExtendScript, which is Adobe's extended version of JavaScript tailored for their applications. [1] Similarly, Microsoft Office applications provide scripting interfaces based on JavaScript, allowing users to automate tasks within Excel, Word, and other Office programs. [2] These scripting interfaces expose Application Programming Interfaces (APIs) that allow scripts to interact with the software's internal functions and data structures.

Traditionally, creating scripts using these interfaces requires programming expertise and familiarity with the specific APIs of the software, posing a barrier for many users. With the strong code generation capacity of Large Language Models (LLMs) (Chen et al., 2021; Bubeck et al., 2023), some solutions generate code based on user query during runtime (Gandhi et al., 2023; Zhao et al., 2024a). However, such runtime generation approaches have notable limitations: (1) the generated code is unverified when presented to the users, leading to low-quality code that may not align with users' intentions and can introduce security risks through unintended behaviors; (2) they impose considerable runtime burden, including increased response times and token generation costs, particularly for applications with a large user base.

In this work, we propose using offline simulation to curate a software-specific skillset – a set of scripts that automate tasks within the software. Then they can be retrieved during runtime to solve user queries. We use publicly available scripting

*Work done during internship at Adobe Research

†Corresponding authors: Gang Wu (gawu@adobe.com) and Xiang Chen (xiangche@adobe.com)

[1]Illustrator: https://ai-scripting.docsforadobe.dev/. Photoshop: https://helpx.adobe.com/photoshop/using/scripting.html.

[2]https://learn.microsoft.com/en-us/office/dev/add-ins/reference/javascript-api-for-office

Findings of the Association for Computational Linguistics: EACL 2026, pages 743–759
March 24-29, 2026

guides and LLMs' knowledge about the software to create the skillset. The offline simulation consists of two LLM-based components: (1) task creation, which generates useful tasks within the software, and (2) skill generation with trials, translating the generated tasks into skills with execution feedback from previous trials. For task creation, we introduce two simulation strategies that use the software's functionality information (top-down) and API information (bottom-up) from the publicly available scripting guide. To more efficiently explore the vast number of APIs supported in the software, we define synergistic API pairs as APIs that can work together in existing skills. We construct a synergistic API graph and train a link prediction model using a Graph Neural Network (GNN) to capture both the semantic and structural patterns of existing synergistic API pairs. This enables the model to generalize to unseen API pairs and assess their compatibility. The synergy of APIs is further used to prompt LLMs to generate tasks that better elicit the software's internal functions and data structures by using more long-tailed APIs.

We conduct comprehensive experiments using Adobe Illustrator as a testbed to evaluate our approach. Our findings demonstrate that our offline simulation framework significantly improves success rates and efficiency of automation task compared to traditional runtime code generation methods. Our main contributions are:

- We propose a novel offline simulation framework for curating a software-specific skillset, leveraging LLMs and publicly available scripting guides. Our framework employs two simulation strategies—top-down functional guidance and bottom-up API synergy exploration—to generate tasks and scripts that cover a wide range of software functionalities.
- We introduce a new setup that leverages the software's API information to explore the capacity of the software. We propose to use a GNN-based link prediction model to capture the synergy between APIs, which encourages generating skills involving underutilized or long-tailed APIs, thereby expanding the diversity and utility of the skillset.
- To the best of our knowledge, this is the first attempt to use software scripting interfaces as a testbed for LLM-based systems. This approach highlights the advantages of obtaining direct execution feedback in a controlled environment and offers valuable insights into aligning AI capabilities with user needs in specialized software domains.

2 Related Work

2.1 Skill Discovery with LLMs

LLM-based skill discovery is gaining attention across various domains, including embodied agents (Zhao et al., 2024b), sandbox environment (Wang et al., 2024b), and LLM tool usage (Qian et al., 2023; Cai et al., 2024; Yuan et al., 2024; Nguyen et al., 2024). Similarly, skills in these scenarios are represented by code and designed to interact with corresponding environments. However, existing methods either do not consider the coverage of these skills within their environments (Zhao et al., 2024b; Wang et al., 2024b; Qian et al., 2023; Cai et al., 2024) or rely on a rich dataset pairing task descriptions with desired outcomes in a question-answering format (Qian et al., 2023; Cai et al., 2024; Yuan et al., 2024; Nguyen et al., 2024). Therefore, these approaches are limited in applicability to practical scenarios such as software scripting automation, where outcomes are more complex (sometimes involving multiple modalities) and cannot be easily encapsulated in a simple text string. Curating a dataset with such diverse and complex outcomes is challenging, especially when the goal is to explore a software's full automation potential. Our study addresses this gap by investigating novel skill discovery strategies that leverage a software's publicly available functionalities and API information.

2.2 Program Synthesis with LLMs

Program synthesis aims to generate code given a natural language description (Zhang et al., 2023), where LLMs (Austin et al., 2021; Chen et al., 2021; Nijkamp et al., 2023) have demonstrated impressive performance. Modern code generation models are typically evaluated on functional correctness (Liu et al., 2024b), often requiring predefined unit tests (Chen et al., 2021; Li et al., 2022). For example, pass@k (Chen et al., 2021; Li et al., 2022) evaluates the model's chance of passing all unit tests with any of k generated samples. In the context of software scripting automation, execution results from the software environment provide immediate feedback on generated code. Unlike the setting of code generation with given task descriptions, this study emphasizes deciding what tasks to generate in specific software, leveraging LLMs'

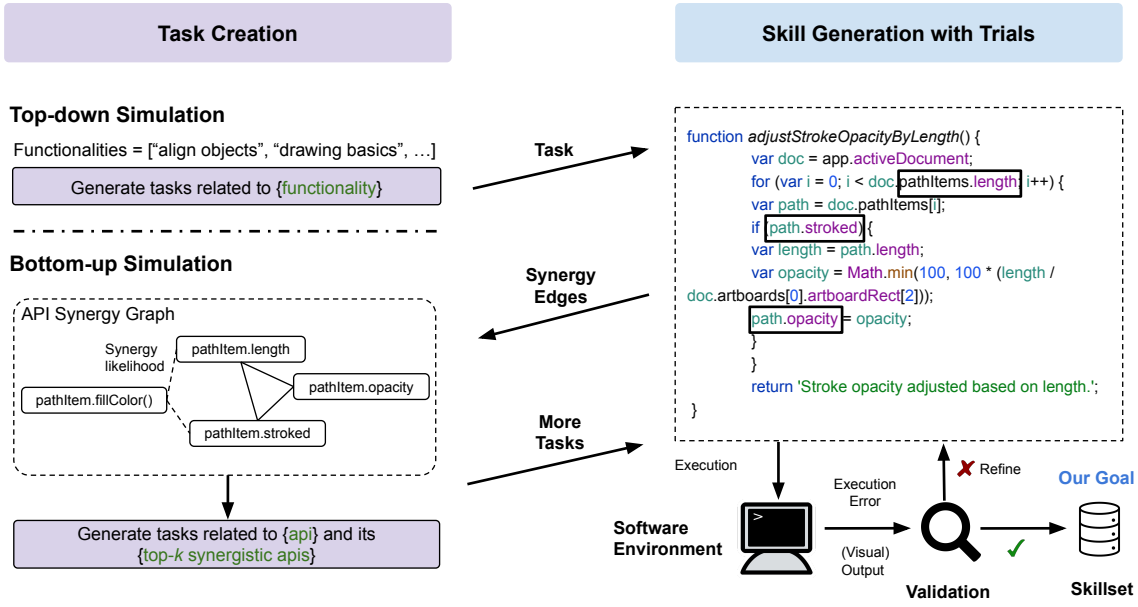


Figure 1: Overview of our offline simulation framework for skill discovery in software scripting automation. The framework consists of two components: (1) Task Creation (Section 3.1), utilizing two simulation strategies: top-down functionality guidance and bottom-up API synergy exploration to generate a wide range of tasks; and (2) Skill Generation with Trials (Section 3.2), where LLMs iteratively refine scripts based on execution feedback to produce verified skills ready for runtime retrieval.

strong code generation capacity that can be further improved with execution feedback from the software environment during offline simulation (Kim et al., 2024; Pan et al., 2024).

2.3 Automation with LLM Agents

With the rapid advancement of Large Language Models (LLMs), researchers have developed systems and benchmarks to automate complex tasks requiring multiple applications in computer environments (Xie et al., 2024; Cao et al., 2024). However, existing LLM-based agents face significant challenges in reliably automating these tasks, achieving only about a 15% success rate across several hundred tasks. Efforts to improve LLM capabilities have also focused on domain-specific tasks, such as spreadsheet manipulation (Li et al., 2024; Ma et al., 2024b), Graphical User Interface (GUI) automation (Gao et al., 2024; Nguyen et al., 2025), and web browsing (Yao et al., 2022; Ma et al., 2024a; Deng et al., 2024). In these specialized domains, LLM-based agents demonstrate higher capacity. Our paper aligns with this body of work by leveraging LLMs to automate domain-specific computer tasks. However, we present the first attempt to (1) automate tasks through the software's internal scripting environment, and (2) use LLM-based agents to systematically explore and identify which tasks can be automated in the software. Our primary objective is to generate a verified skillset that represents a software's supported functionalities and aligns with users' practical needs, rather than develop LLM agents that directly automate tasks.

3 Method

In this section, we detail our framework for skill discovery with offline simulations, which can be divided into two modules, task creation and skill generation with trials. We formulate the problem as follows: our goal is to develop a set of tasks (in natural language), $\mathcal{T} = \{t_1, t_2, ..., t_n\}$, which can be automated via the scripting interface in a software application. The corresponding scripts or code are defined as skills, $\mathcal{S} = \{s_1, s_2, ..., s_n\}$. After developing S during offline simulation, a user can input some operation they want to automate as a query q to the system during runtime, where a relevant skill, s_i, is retrieved to solve q. Figure 1 shows an overview of the proposed framework and we now introduce our method in detail.

3.1 Task Creation

We rely on LLMs to generate the tasks. To fully exploit the functionalities, we adopted two strategies to guide LLMs to "search" for possible tasks, i.e., a top-down approach that starts from high-level func-

tionalities of the software, followed by a bottom-up approach that considers low-level APIs supported by the software.

Top-down We curate a list of high-level functionalities for the software. Taking Adobe Illustrator as an example, the high-level functionalities can be drawing, arranging objects, and so on. This list can be obtained from the content categorization from publicly available scripting guides. For each functionality, we prompt the LLMs to generate related tasks. Following previous works that adopt a long-term memory (Wang et al., 2024b,a), we generate tasks in multiple rounds and integrate the feedback obtained from the previous round. We introduce what feedback our framework provides in Section 3.2. The prompts are in Appendix B. We note that this step can also be considered as a warmup for the following bottom-up search.

Bottom-up A unique opportunity and challenge in the Software scripting interface setting is that we have a list of all APIs supported by the software, denoted as $\mathcal{A} = \{a_1, a_2, ..., a_n\}$, and the scripts largely rely on the "collaboration" of multiple APIs. Therefore, we prompt LLMs with appropriate API combinations to spark LLMs' knowledge about the software. Given the large number of APIs and the potential for multiple tasks to be accomplished using the same set of APIs, we introduce an API synergy graph, $\mathcal{G}$, to model the likelihood of APIs working together. Graph-based representations are widely employed to model relationships in related applications such as API recommendations (Qi et al., 2022; Huang et al., 2022) LLM-based reasoning (Anokhin et al., 2024; Liu et al., 2024a), and so on (Wu et al., 2022; Xu et al., 2025).

Specifically, we define APIs, $\mathcal{A}$, as the nodes in $\mathcal{G}$ and the edges, $\mathcal{E}$, represent whether the two APIs have appeared in a verified script. The node features are the semantic embeddings of the corresponding API descriptions, denoted as $\mathbf{X}$. The two nodes with a link are defined as a synergistic API pair that can work together. We then train a link prediction model by randomly masking existing links in $\mathcal{G}$ and predicting the likelihood of their existence. We use a Graph Convolutional Network (GCN) model (Kipf and Welling, 2017) to achieve this. The model aggregates information from neighboring nodes and their features, $\mathbf{H}^{(l)} = \text{GCN}^{(l)}(\mathbf{H}^{(l-1)}, \mathcal{E})$ where $\mathbf{H}^{(0)} = \mathbf{X}$ are the initial node embeddings and $\mathbf{H}^{(l)}$ are the node representations after l layers of message passing. The likelihood of an edge between two nodes u and v is modeled as: $\hat{y}_{uv} = \sigma\left(f\left(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)}\right)\right)$, where $\mathbf{h}_u^{(L)}$ and $\mathbf{h}_v^{(L)}$ are the L-layer representations of nodes u and v (final layer), f is a scoring function, such as the inner product $f(\mathbf{h}_u, \mathbf{h}_v) = \mathbf{h}_u^\top \mathbf{h}_v$, and σ is the sigmoid function. In short, the model learns to predict the likelihood of two API nodes working together by aggregating information from neighboring nodes and their features. The model is trained with a binary cross-entropy loss:

$$\mathcal{L} = -\frac{1}{|\mathcal{D}|} \sum_{(u,v)\in\mathcal{D}} \Big[y_{uv} \log(\hat{y}_{uv}) + (1 - y_{uv}) \log(1 - \hat{y}_{uv}) \Big],$$

where $\mathcal{D}$ is the set of sampled edges (positive pairs from $\mathcal{E}$ and negative pairs not in $\mathcal{E}$), and $y_{uv} \in \{0, 1\}$ is the ground truth label for whether the edge (u, v) exists. After training, the GCN model captures both the semantic and structural patterns of synergistic API pairs, enabling it to generalize to unseen API pairs and assess their compatibility.

Then for each API a_i, we prompt LLMs to generate tasks related to a_i and its top-k synergistic APIs. We show full prompts in Appendix B.

3.2 Skill Generation with Trials

One advantage of generating skills offline is that we can generate the code with multiple trials offline without adding any burden to the users during runtime, providing a better user experience. For each generated t_i, we use a strategy similar to the ideas of Wang et al. (2024b,a), where the LLM learns from the execution feedback in the software to refine its own outputs. Additionally, we use another LLM or a Large Vision Language Model (LVLM) as a validator to judge the script by looking at the generated code, execution output, and (visual) outcome in the software. The validator comments on whether the skill accomplishes the task and provides feedback for improvement for the next trial. We show the system and user prompt for the validator in Table 8. A skill is added to the skillset $\mathcal{S}$ if it passes the validator. Specifically, when refining the scripts, the LLM receives a structured prompt containing the task description, code from the previous round, any execution errors generated by the software, and the validator's assessment (including suggestions for improvement). This structured feedback enables the LLM to refine the script by addressing both code-level issues

and misalignments with task intent. We allow up to three trials per task. We show the prompt for ExtendScript code generation in Table 7. Example skills with above-mentioned elements are shown in Appendix C.

4 Experiments

Adobe Illustrator as the Testbed Adobe Illustrator is a leading vector graphics software used by professionals worldwide for tasks such as creating logos, illustrations, and complex design elements. It supports multiple categories of high-level functionalities, including drawing, arranging objects, and applying effects. A full list of the high-level functionalities used in the top-down search in Section 3.1 is provided in Appendix A, which can be obtained through Illustrator's official scripting guide. [3] Modern software platforms that support scripting interfaces offer tasks with varying levels of complexity. This study focuses on generating "atomic" skills: fundamental, modular tasks that involve minimal design choices and serve as building blocks for more complex operations. For instance, in the context of Illustrator, instead of creating an entire flower design, an atomic skill would arrange pre-designed petals into a circular pattern, emphasizing precision and modularity. Illustrator's scripting interface supports 1818 API endpoints, where 378 are the methods and the remaining are attributes of the object, enabling extensive programmatic control over the software. We also explored using Excel as the testbed, but were unable to conduct a large-scale experiment due to limitations in Excel's ability to permit programmatic control over Office Script from outside applications. See Appendix E for details.

Simulation Setup Adobe Illustrator, as a vector design tool, focuses on creating and manipulating graphical objects. In practical applications, a typical Illustrator project often contains numerous objects, and scripts are usually expected to operate on specific subsets of these objects. The common approach is to manually select the desired objects before running the script, leveraging the `selection` attribute available for each object. Therefore, for each skill, we prompt the LLM to generate initialization scripts that set up the document with necessary elements and adjust their `selection` attributes according to the task description. Following previous work with other virtual environments (Wang et al., 2024b), we prompt the LLM to make the task code generic and reusable, as well as other Illustrator-specific instructions. The full prompt is shown in Appendix B. Table 1 presents an example skill, including the corresponding task description, code implementation, and the layouts after running the initialization script and the skill, respectively.

Desc	arrange selected objects in circle

Code	`function arrangeInCircle(cX, cY, radius) {`
	`  var sel = app.selection`
	`  if (sel.length === 0) {`
	`  throw new Error('No selection');`
	`  }`
	`  var angleStep = 360 / sel.length;`
	`  for (var i = 0; i < sel.length; i++) {`
	`  var angle = angleStep*i*(Math.PI/180);`
	`  var x = cX + radius * Math.cos(angle);`
	`  var y = cY + radius * Math.sin(angle);`
	`  sel[i].position = [x, y];`
	`  }`
	`  return 'Objects arranged in a circle';`
	`}`

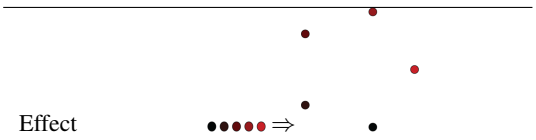
Effect	

Table 1: An example skill, `arrangeInCircle`. `Desc` is the natural language description. `Effect` shows the layout before and after running the skill.

We now detail how we operationalize our framework in Adobe Illustrator. We first execute the top-down simulation using the functionality categories listed in Appendix A, where the LLM explores tasks in each subcategory over three rounds. The validated skills, combined with sample scripts provided by the software, are then used to construct the API synergy graph. To train the GCN model for link prediction, we randomly split the sampled edges (both positive and negative) into training, development, and test sets with a ratio of 0.85/0.05/0.1. The GCN model has two layers with an embedding size of 128. We trained the model for 300 epochs using the Adam optimizer with a learning rate of 0.01. Next, for each of the approximately 500 method APIs, we retrieve its top-k ($k = 5$) synergistic APIs and prompt the LLM to generate related tasks for one round.[4] In each round of the task creation stage (both top-down and bottom-up approaches), the LLM generates ten tasks. For each task, the LLM is allowed up to three trials with previous feedback to generate the code of skills. We note that these hyperparameters may be subject to change based on cost considerations.

[3] https://helpx.adobe.com/pdf/illustrator_reference.pdf

[4] We selected $k = 5$ to balance between capturing potentially synergistic APIs and avoiding overwhelming the LLM with irrelevant API information.

We chose these values to ensure that the cost and size of the generated skillset remain manageable.

Evaluation Setup To curate a test set that represents the needs of real users, we iteratively prompt the LLM to generate useful tasks in Illustrator. We manually verify the tasks and correct any issues in the corresponding initialization scripts, if any, to enable automatic evaluation, resulting in 94 test tasks. We use the `all-mpnet-base-v2` model from sentence BERT (Reimers and Gurevych, 2019) to encode the task descriptions for both test set and the skillset built during offline simulation. We use the test task description as the query and retrieve the relevant skills from the skillset with semantic matching through cosine similarity.

Baseline The baseline method for this task is runtime generation where an LLM generates code to solve the query during runtime. We use `gpt-4o 2024-08-06` as the LLM for baseline, as well as the ones in the proposed framework for fair comparisons. We note that advances in program synthesis methods are complementary to our approach; a stronger program synthesis model would benefit both the baseline and our framework. To demonstrate the generalization ability of our framework, we also evaluated `lama-3.1-70B` and `deepseek-r1` as the code generation model on a random sample of 200 tasks from each of the top-down and bottom-up strategies.

Proposed Methods (1) **Retrieval-Only (RO)**: Given a user query q, the system retrieves the most semantically relevant skill using cosine similarity over sentence embeddings. This approach ensures low-latency responses and minimal runtime cost, and avoids unsafe outputs, as all retrieved skills are pre-validated through offline trials. (2) **Retrieval-Augmented Generation (RAG)**: To explore the use of the skillset for handling arbitrary user queries, we prompt an LLM with the top-r ($r = 3$) retrieved skills as in-context examples. This enables flexible adaptation and composition beyond atomic tasks.

Evaluation Metrics We evaluate the generated skillset by assessing its ability to solve the given tasks. Specifically, we report the **Success Rate**, which represents the proportion of tasks where an LVLM determines that the outcomes satisfy the provided task descriptions. This follows a similar procedure outlined in Section 3.2. We further evaluate how reliable the LVLM's judgment is by comparing it with human judgment, the results are discussed in Section 5.3. We note that judging the actual outcome after execution is challenging. Previous work on LLM tool usage adopts a simpler setting, e.g., only focus on tasks in a question-answering format where answers can be easily verified (Mialon et al., 2024; Yuan et al., 2024) or only checking the correctness of tool calling (without checking the execution results) (Wang et al., 2024a). Additionally, because automating scripting interfaces requires consideration for user experiences and runtime cost, we report **Response Time** as the averaged seconds per task for the system to output a script (either through retrieval or generation), along with averaged **Token Cost** during runtime.

5 Results and Discussions

5.1 Effectiveness in Real-World Scenarios

Our proposed method demonstrates a substantial improvement over the baseline in automating tasks within Adobe Illustrator's scripting environment. As shown in Table 2, our approach (**RO**) achieves a success rate of 44.7%, outperforming the baseline's success rate of 28.7% on the held-out test set. This evaluation setup effectively measures the end-to-end performance of automation systems when deployed in real-world software contexts, with the natural language task descriptions in the test set simulating real user queries. In addition to RO, the RAG variant achieves a success rate of 42.6%, improving baseline performance by approximately 15%. This demonstrates that retrieved skills not only serve as executable scripts but also act as effective in-context examples for code generation.

A key advantage of RO lies in its markedly lower runtime costs, which is a critical consideration given the potential volume and repetitive nature of user queries. RO avoids runtime code generation entirely by retrieving pre-validated skills, resulting in an average response time of just 0.1 seconds for retrieval and zero token cost at runtime, as offline simulation is a one-time expense. In contrast, both the baseline and RAG require significantly more resources at runtime, with response times of 4.0 and 4.3 seconds, and token costs of 666 and 1219.

5.2 Effectiveness of Two Simulation Strategies

Contribution to solving the test set. During offline simulation, we employed both top-down and bottom-up simulation strategies to capture a wide range of functionalities within the software's

	Success Rate	Response Time	Token Cost
Baseline	28.7%	4.0 s	666
RAG	42.6%	4.3 s	1219
RO	44.7%	0.1 s	0*

Table 2: Evaluation of the baseline and the proposed approaches. * The token cost for offline simulation in our approach is one-off and not included in the runtime cost.

	Top-down	Bottom-up	Total
Successful Skills	35.1%	9.8%	44.7%
Unsuccessful Skills	40.4%	14.9%	55.9%
Total	75.5%	24.4%	100%

Table 3: Distribution of retrieved skills across op-down and bottom-up simulation strategies in the test set.

scripting environment. Table 3 presents the contributions of each strategy and their combined impact when tackling tasks in the test set. Top-down simulation accounts for approximately 75% of the skills retrieved. While the top-down strategy has a higher overall contribution, the bottom-up strategy still plays a significant role. It's important to note that the test set curation and the top-down simulation both involve prompting the LLM to generate useful tasks at a high level of abstraction based on the software's functionalities. This similarity potentially gives the top-down approach an advantage in matching test tasks. On the other hand, the bottom-up simulation, which prompts the LLM with low-level API information, is designed to cover long-tailed, less obvious skills. Despite this, the bottom-up simulation still achieves comparable success rates relative to their contributions (top-down: $\frac{35.1\%}{75.5\%} = 46.5\%$, bottom-up: $\frac{9.8\%}{24.4\%} = 40.2\%$) to the test set that predominantly contains head-tail tasks — the most helpful tasks as determined by the LLM.

Bottom-up simulation uses more APIs but coverage remains incomplete. To further demonstrate that the bottom-up approach is effective in exploring the long-tail distribution of skills within the software's capabilities, Table 4 shows skills from bottom-up simulation cover 151 unique APIs, significantly more than the 49 APIs covered by the top-down approach and the 48 from sample scripts provided by the software. This broader API coverage indicates that the bottom-up approach is successful in exploring a wide range of functionalities

	Sample Scripts	Top-down	Bottom-up
# APIs	48	49	151

Table 4: Number of API endpoints from Illustrator's inherit sample scripts and skills built from two simulation strategies.

	Test Set	Top-down	Bottom-up
Avg. Score	2.48	2.28	1.75

Table 5: Average usefulness scores (1–3 scale) of sampled tasks, as rated by human evaluators.

beyond the more commonly used APIs.

However, as mentioned in Section 3.1, the bottom-up simulation iterated through approximately 378 method APIs. The fact that only 151 of these APIs resulted in successfully validated scripts suggests that many APIs did not have any corresponding successful tasks or scripts. We posit that the main reason for this limitation is the employed general-purpose LLMs' limited knowledge of Adobe Illustrator's extensive API library, resulting in LLMs' failure to generate relevant tasks or to produce code with correct API usage.

This outcome is expected, given the complexity and long-tail nature of software scripting API design. For example, a suite of APIs in Illustrator relates to physically printing, exporting to various file formats, and so on. These are "end-of-workflow" commands or meta-actions about the environment, rather than modular skills for manipulating document content. Verifying their success is difficult in a closed-loop simulation. Some APIs are tied to specific legacy software versions. Therefore, complete API coverage is an impractical goal. However, the long tail still represents numerous specialized APIs that are infrequently used and therefore less likely to be well-represented in the LLMs' training data. This highlights room for future improvement, which we discuss in Section 6.

Human-perceived usefulness To assess the practical value of the discovered skills, we conducted a small-scale user study in which two Illustrator users with programming experience rated the usefulness of 150 sampled tasks on a 3-point scale: 1 (very rarely useful), 2 (occasionally helpful), and 3 (very useful). The sample included 50 tasks each from the test set, top-down skills, and bottom-up skills. Annotators judged each task based solely on its description. We discuss the annotation setup

and annotator agreement in Appendix D. As shown in Table 5, skills produced by the top-down simulation strategy received a relatively high average usefulness score of 2.28, closely matching the 2.48 average for tasks in the test set. In contrast, bottom-up skills averaged 1.75. Although this is relatively lower, it still leans toward the "occasionally helpful" category. This result is consistent with the bottom-up strategy's goal of uncovering long-tail functionalities. We examine these differences further in the qualitative analysis below.

Qualitative Analysis The top-down simulation tends to generate tasks that reflect high-level design intentions familiar to users. For instance, it produces skills such as "arranging selected objects in a circle" (Table 1), along with other layout patterns like zigzag or starburst formations. These tasks are aligned with common design workflows and are typically easy to interpret and apply. The bottom-up simulation yields more technically specialized tasks that leverage less frequent APIs or scripting features. One example (Table 13) involves alerting the user when selected objects have tags that match a given list. Such skill introduces conditional logic and metadata inspection, which is expected to be rarer in practical usage but provides greater flexibility for advanced users.

Synergy Modeling of APIs in Bottom-up Simulation During bottom-up simulation, we use a GNN-based link prediction model to model the synergy of APIs. We show that this model can find synergistic APIs more effectively, compared to semantically matching similar APIs. We evaluate the performance of the GCN model in retrieving synergistic APIs for task creation using the Hit@k metric. In this context, Hit@k evaluates, for each API, how often the correct items are within the top-k predicted items where correct items refer to APIs in existing skills (not in the training set). We compare the link prediction model with a semantic matching baseline, which retrieves top-k similar APIs using `all-mpnet-base-v2` embeddings and cosine similarity Table 6 shows our model significantly outperforms semantic matching in identifying synergistic APIs, enhancing task creation in bottom-up simulation.

Generation with trials is a key driver of the improved success rate for both simulation strategies. One key advantage of our offline simulation is the ability to perform multiple trials during code generation without incurring runtime penalties. We analyzed the impact of allowing up to three trials for code generation on the overall success rate. Table 7 demonstrates that the success rate nearly doubles after three trials for both top-down and bottom-up strategies. Specifically, the success rate for both top-down and bottom-up strategies nearly doubles after three trials. We found similar improvement when using `Llama-3.1-70B` and `deepseek-r1` as the code generation models shown in Table 11.

	Semantic matching	Link Prediction
Hit@5	16.7%	37.3%

Table 6: Comparison between the link prediction model and semantic matching method for identifying synergistic APIs.

	# Tasks	%Success@1	%Success@3
Top-down	1721	16.7%	34.9%
Bottom-up	3256	23.1%	46.6%

Table 7: Success rate at the first and third trials during simulations.

5.3 Reliability of LVLM's judgment

We employed an LVLM to evaluate whether the execution outcomes satisfy the task descriptions. To estimate the reliability of the LVLM's judgments, we compared its assessments with human evaluations on a sample of 122 tasks. Table 8 shows that the LVLM has a precision of 90.9% and a recall of 80.0%. Out of 50 tasks deemed successful by humans, the LVLM correctly identified 40 as successful and 10 as failures. Meanwhile, the LVLM only incorrectly labeled 4 tasks as successful when they were not (false positives). These results indicate that the LVLM is generally reliable, albeit slightly conservative in its judgments.

6 Conclusions

We proposed an offline simulation framework for skill discovery in software scripting automation with LLMs. Our method pre-build a diverse and verified skillset in two phases: (1) task creation, guided by both top-down functionality categories and bottom-up API synergy exploration, and (2) skill generation with trials, where LLM-generated scripts are iteratively refined through feedback based on execution error and functional outcome.

Experiments on Adobe Illustrator demonstrated

	Success (LVLM)	Fail (LVLM)
Success (Human)	40 (TP)	10 (FN)
Fail (Human)	4 (FP)	68 (TN)

Table 8: Confusion matrix between human and LVLM judgments.

that our method achieves higher success rates, broader functional coverage, and lower response time compared to runtime generation. The top-down task creation strategy encourages broad functional coverage, while the bottom-up API synergy modeling explores the of discovered skills, particularly for underutilized API endpoints. Furthermore, our GCN-based API link prediction model outperformed naive semantic matching in identifying synergistic API pairs, enhancing task creation during the bottom-up simulation.

To our best knowledge, this is the first attempt to use software scripting interfaces as a testbed for LLM-based systems, highlighting the advantage of obtaining direct execution feedback in a controlled environment and providing valuable insights into the design of more capable and user-aligned scripting systems.

Limitations

Despite the promising results of our offline simulation framework for skill discovery in software scripting automation, there are several limitations.

First, our method relies heavily on publicly available scripting guides and the existing knowledge of LLMs about the software. In cases where such documentation is incomplete, outdated, or unavailable, or when LLMs have limited knowledge of the software, LLMs may hallucinate API usage or rely on superficial cues (Patil et al., 2024; Zhou et al., 2024; Liu et al., 2025), reducing effectiveness and robustness. This dependence restricts the applicability of our framework to well-documented software systems and those familiar to the LLMs. To address this, one promising direction is to train a domain-specific LLM tailored to the software through iterative trial and error, which could generate more accurate and comprehensive skills. Such methods are complementary to our framework.

Second, our evaluation primarily focuses on automated success rates and performance metrics within a controlled environment. While these metrics provide valuable insights into technical performance, they do not capture user experience or satisfaction. Automatic evaluation can be less reliable on such subjective or high-inference criteria (Liu et al., 2016; Amershi et al., 2019; Xu et al., 2024). Although we ran a small-scale user study to evaluate the perceived usefulness of the explored tasks, future studies could include more comprehensive user evaluations to assess the usability and overall impact of the generated skills in real-world workflows. Additionally, mining API interactions from actual scripting projects could provide valuable insights into common user behaviors and task patterns, making the generated skills more applicable in practice. However, obtaining such data is challenging and raises interesting questions about what kinds of behavioral data are useful and how to collect them responsibly.

Third, we expect that queries from real users may be more complex and not always match the granularity of the skills in our skillset. Real-world user queries might be ambiguous or involve higher-level tasks that require combining multiple skills or adapting existing ones. The skillset generated by our framework does not account for individual user preferences or specific workflow requirements. However, the proposed RAG variant shows promising results, significantly improving the success rate with only a modest increase in response time, suggesting its potential to better handle such complex or personalized queries.

Fourth, due to limited support for external programmatic control, we are only able to conduct a large-scale study on Adobe Illustrator. Although we conducted initial exploration on Excel, future work should extend the framework to other scripting environments with broader automation support to validate its generalizability.

Acknowledgments

We would like to thank our collaborators at Adobe Research and the University of Maryland for their helpful discussion, including Dang Minh Nguyen, Dayeon Ki, Nishant Deepa Balepur, Vishakh Padmakumar, Yoonjoo Lee, and Hyunji Lee.

References

Saleema Amershi, Dan Weld, Mihaela Vorvoreanu, Adam Fourney, Besmira Nushi, Penny Collisson, Jina Suh, Shamsi Iqbal, Paul N Bennett, Kori Inkpen, et al. 2019. Guidelines for human-ai interaction. In *Proceedings of the 2019 chi conference on human factors in computing systems*, pages 1–13.

Petr Anokhin, Nikita Semenov, Artyom Sorokin, Dmitry Evseev, Mikhail Burtsev, and Evgeny Burnaev. 2024. Arigraph: Learning knowledge graph world models with episodic memory for llm agents. *arXiv preprint arXiv:2407.04363*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. 2024. Large language models as tool makers. In *The Twelfth International Conference on Learning Representations*.

Ruisheng Cao, Fangyu Lei, Haoyuan Wu, Jixuan Chen, Yeqiao Fu, Hongcheng Gao, Xinzhuang Xiong, Hanchong Zhang, Yuchen Mao, Wenjing Hu, et al. 2024. Spider2-v: How far are multimodal agents from automating data science and engineering workflows? *arXiv preprint arXiv:2407.10956*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. 2024. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36.

Apurva Gandhi, Thong Q Nguyen, Huitian Jiao, Robert Steen, and Ameya Bhatawdekar. 2023. Natural language commanding via program synthesis. *arXiv preprint arXiv:2306.03460*.

Difei Gao, Lei Ji, Zechen Bai, Mingyu Ouyang, Peiran Li, Dongxing Mao, Qinchen Wu, Weichen Zhang, Peiyi Wang, Xiangwu Guo, et al. 2024. Assistgui: Task-oriented pc graphical user interface automation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13289–13298.

Qing Huang, Zhiqiang Yuan, Zhenchang Xing, Zhengkang Zuo, Changjing Wang, and Xin Xia. 2022. 1+ 1> 2: Programming know-what and know-how knowledge fusion, semantic enrichment and coherent application. *IEEE Transactions on Services Computing*, 16(3):1540–1554.

Geunwoo Kim, Pierre Baldi, and Stephen McAleer. 2024. Language models can solve computer tasks. *Advances in Neural Information Processing Systems*, 36.

Thomas N Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations (ICLR)*.

Hongxin Li, Jingran Su, Yuntao Chen, Qing Li, and ZHAO-XIANG ZHANG. 2024. Sheetcopilot: Bringing software productivity to the next level through large language models. *Advances in Neural Information Processing Systems*, 36.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Chia-Wei Liu, Ryan Lowe, Iulian Serban, Mike Noseworthy, Laurent Charlin, and Joelle Pineau. 2016. How NOT to evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2122–2132, Austin, Texas. Association for Computational Linguistics.

Guangyi Liu, Yongqi Zhang, Yong Li, and Quanming Yao. 2024a. Explore then determine: A gnn-llm synergy framework for reasoning over knowledge graph. *arXiv preprint arXiv:2406.01145*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024b. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36.

Xiaoyu Liu, Paiheng Xu, Junda Wu, Jiaxin Yuan, Yifan Yang, Yuhang Zhou, Fuxiao Liu, Tianrui Guan, Haoliang Wang, Tong Yu, et al. 2025. Large language models and causal inference in collaboration: A comprehensive survey. *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 7668–7684.

Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. 2024a. Agentboard: An analytical evaluation board of multi-turn llm agents. *arXiv preprint arXiv:2401.13178*.

Zeyao Ma, Bohan Zhang, Jing Zhang, Jifan Yu, Xiaokang Zhang, Xiaohan Zhang, Sijia Luo, Xi Wang, and Jie Tang. 2024b. Spreadsheetbench: Towards challenging real world spreadsheet manipulation. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2024. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*.

Dang Nguyen, Jian Chen, Yu Wang, Gang Wu, Namyong Park, Zhengmian Hu, Hanjia Lyu, Junda Wu, Ryan Aponte, Yu Xia, et al. 2025. Gui agents: A survey. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 22522–22538.

Dang Nguyen, Viet Dac Lai, Seunghyun Yoon, Ryan A Rossi, Handong Zhao, Ruiyi Zhang, Puneet Mathur, Nedim Lipka, Yu Wang, Trung Bui, et al. 2024. Dynasaur: Large language agents beyond predefined actions. *arXiv preprint arXiv:2411.01747*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*.

John K Ousterhout. 1998. Scripting: Higher level programming for the 21st century. *Computer*, 31(3):23–30.

Liangming Pan, Michael Saxon, Wenda Xu, Deepak Nathani, Xinyi Wang, and William Yang Wang. 2024. Automatically correcting large language models: Surveying the landscape of diverse automated correction strategies. *Transactions of the Association for Computational Linguistics*, 12:484–506.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2024. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37:126544–126565.

Lianyong Qi, Wenmin Lin, Xuyun Zhang, Wanchun Dou, Xiaolong Xu, and Jinjun Chen. 2022. A correlation graph based approach for personalized and compatible web apis recommendation in mobile app development. *IEEE Transactions on Knowledge and Data Engineering*, 35(6):5444–5457.

Cheng Qian, Chi Han, Yi Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. 2023. Creator: Tool creation for disentangling abstract and concrete reasoning of large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 6922–6939.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

Boshi Wang, Hao Fang, Jason Eisner, Benjamin Van Durme, and Yu Su. 2024a. LLMs in the imaginarium: Tool learning through simulated trial and error. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 10583–10604, Bangkok, Thailand. Association for Computational Linguistics.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2024b. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*.

Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. 2022. Graph neural networks in recommender systems: a survey. *ACM Computing Surveys*, 55(5):1–37.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. 2024. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *arXiv preprint arXiv:2404.07972*.

Paiheng Xu, Jing Liu, Nathan Jones, Julie Cohen, and Wei Ai. 2024. The promises and pitfalls of using language models to measure instruction quality in education. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 4375–4389, Mexico City, Mexico. Association for Computational Linguistics.

Paiheng Xu, Yuhang Zhou, Bang An, Wei Ai, and Furong Huang. 2025. Gfairhint: Improving individual fairness for graph neural networks via fairness hint. *ACM Transactions on Knowledge Discovery from Data*, 19(3):1–22.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2022. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757.

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi Fung, Hao Peng, and Heng Ji. 2024. Craft: Customizing llms by creating and retrieving from specialized toolsets. In *The Twelfth International Conference on Learning Representations*.

Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2023. Unifying the perspectives of nlp and software engineering: A survey on language models for code. *arXiv preprint arXiv:2311.07989*.

Wei Zhao, Zhitao Hou, Siyuan Wu, Yan Gao, Haoyu Dong, Yao Wan, Hongyu Zhang, Yulei Sui, and Haidong Zhang. 2024a. NL2Formula: Generating spreadsheet formulas from natural language queries. In *Findings of the Association for Computational Linguistics: EACL 2024*, pages 2377–2388, St. Julian's, Malta. Association for Computational Linguistics.

Xufeng Zhao, Cornelius Weber, and Stefan Wermter. 2024b. Agentic skill discovery. *arXiv preprint arXiv:2405.15019*.

Yuhang Zhou, Paiheng Xu, Xiaoyu Liu, Bang An, Wei Ai, and Furong Huang. 2024. Explore spurious correlations at the concept level in language models for text classification. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 478–492.

A Adobe Illustrator Functionalities

Part of the functionality categories in Illustrator are shown in Table 14.

B Prompts

In this section, we show system prompts and user prompts for task creation, skill generation with trials, and LVLM-based validation in Table 15, Table 16, and Table 17, respectively.

C Example Illustrator Skills

We show an example skill generated by the top-down strategy in Table 12, and one by the bottom-up strategy in Table 13.

D Annotation for Perceived Helpfulness

We asked the two annotators to rate the usefulness of 150 sampled tasks on a 3-point scale: 1 (very rarely useful), 2 (occasionally helpful), and 3 (very useful). Annotators judged each task based solely on its description, and the tasks are randomly shuffled without disclosing We adopted a stricter interpretation of the scale, acknowledging that most tasks offer some utility in niche scenarios, making it difficult to classify any task as completely useless.

The annotators achieved substantial agreement on their ratings, with similar average scores across conditions and only minor differences between them, as shown in Table 9. While exact agreement occurred in 41% of cases, the one-off agreement was remarkably high at 95%, indicating that when disagreements occurred, they typically differed by only one point on the scale. This is further supported by the weighted Cohen's Kappa score of 0.176, which accounts for the ordinal nature of our rating scale and the fact that most disagreements were minor. Krippendorff's Alpha (0.171) similarly reflects this pattern of agreement. Overall, these metrics demonstrate that despite the inherent subjectivity in usefulness judgments, our annotators maintained reasonable consistency in their evaluations.

E Exploring Excel as the Testbed

While Adobe Illustrator provides a flexible scripting interface that allows external automation of tasks, Excel's scripting support is primarily designed for use within its own environment and does not readily permit programmatic control from outside applications. As a result, the approach used to automate and evaluate scripts in Illustrator could not be directly applied to Excel. This limitation restricted our ability to conduct comparable experiments across both platforms.

However, we explore the following aspects to generalize our framework to Excel. (1) **API analysis**. We collected 2,140 APIs and identified 286 sample scripts from the official tutorial, [5] which cover 124 unique APIs. This supports our broader observation that modern software exposes extensive scripting APIs, while available example scripts typically cover only a small subset. (2) **LLM-based code generation**. We selected five tasks from the "quick scenarios" category in the Excel scripting tutorial and prompted LLM to generate scripts following our framework (see Section 3.2). We show the system prompt for Excel in Table 18 and user prompt follows the one in Table 16. These tasks are aligned with the complexity and skillset expected in our Illustrator experiments. Table 10 reports the number of LLM code generation trials required for each task. We observed that some tasks could be completed with only a few LLM prompt iterations, while others consistently failed, likely due to differences in task difficulty and model familiarity with Excel's scripting environment. Importantly, two out of five tasks require multiple

	Test Set	Top-down	Bottom-up
Rater 1	2.42	2.40	1.78
Rater 2	2.53	2.16	1.72

Table 9: Average usefulness scores (1–3 scale) of sampled tasks, as rated by human evaluators.

Task	# Trials
Combine multiple Excel tables into one	5
Count blank rows on all sheets	Failed
Return table data as JSON	Failed
Remove hyperlinks from each cell	5
Move rows using range values	1*

Table 10: Number of LLM code generation trials required to successfully complete each Excel scripting task. "Failed" indicates that the LLM was unable to produce a working script for the task within five attempts. * due to the ambiguity in task description, it didn't reproduce the example code in the tutorial but was able to achieve something reasonable.

[5] https://learn.microsoft.com/en-us/office/dev/scripts/overview/excel

Model	Strategy	%Success@1	%Success@3
Llama-3.1-70B	Top-down	5.5%	16.8%
Llama-3.1-70B	Bottom-up	8.7%	23.5%
deepseek-r1	Top-down	6.9%	27.6%
deepseek-r1	Bottom-up	8.7%	34.8%

Table 11: Success rate at the first and third trials for each model and strategy.

trials highlights the effectiveness of our offline simulation framework and the trial-and-error approach. Overall, these results indicate both the challenges and opportunities in generalizing LLM-based code generation across software platforms with distinct automation interfaces, and demonstrate that our framework can be extended beyond its initial domain.

F More Code Generation Results

To demonstrate our framework's adaptability, we tested `Llama-3.1-70B` and `deepseek-r1` as the code generation model. Table 11 shows the success rate on a random sample of 200 tasks from each of the top-down and bottom-up strategies.

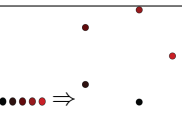
Desc	arrange selected objects in circle
# Trial	No. 2
Code last round (w/ Init. code)	`function createSimpleLayout() { var doc = app.documents.add(); var numObjs = 5; var radius = 100; for (var i = 0; i < numObjs; i++) { var circle = doc.pathItems.ellipse(...); circle.filled = true; circle.fillColor = doc.colors.add(); circle.fillColor.red = i * 50; circle.selected = true; } return 'Initialization complete'; } createSimpleLayout(); function arrangeInCircle(cX, cY, radius) { var sel = app.selection; if (sel.length === 0) { throw new Error('No objects selected'); } var angleStep = 360 / sel.length; for (var i = 0; i < sel.length; i++) { var angle = angleStep*i*(Math.PI / 180); var x = cX + radius * Math.cos(angle); var y = cY + radius * Math.sin(angle); sel[i].position = [x, y]; } return 'Objects arranged in a circle'; } arrangeInCircle(300, 300, 150);`
Error msg	Adobe Illustrator got an error: Error 21: undefined is not an object. Line: 17 -> circle.fillColor = doc.colors.add();
Judge	*Not applicable because code failed to run*
Init. code	`function createSimpleLayout() { var doc = app.documents.add(); var numObjs = 5; var radius = 100; for (var i = 0; i < numObjs; i++) { var circle = doc.pathItems.ellipse(...); circle.filled = true; var color = new RGBColor(); color.red = i * 50; circle.fillColor = color; circle.selected = true; } return 'Initialization complete'; } createSimpleLayout();`
Code	`function arrangeInCircle(cX, cY, radius) { var sel = app.selection if (sel.length === 0) { throw new Error('No selection'); } var angleStep = 360 / sel.length; for (var i = 0; i < sel.length; i++) { var angle = angleStep*i*(Math.PI/180); var x = cX + radius * Math.cos(angle); var y = cY + radius * Math.sin(angle); sel[i].position = [x, y]; } return 'Objects arranged in a circle'; }`
Effect	

Table 12: Complete version of Table 1. `Desc` is the natural language description. `Effect` shows the layout before and after running the skill. `Judge` is the validation judgment of the LVLM. Code simplified for space constraints.

Desc	Alert the user when the name of a tag in the selection matches a predefined list of names.
# Trial	No. 2

Code last round (w/ Init. code)	`var doc = app.documents.add();`
	
	`// Init code`
	`var textFrame = doc.textFrames.add();`
	`textFrame.contents = "Sample␣Text";`
	`textFrame.name = "Tag1";`
	
	`var rect = doc.pathItems.rectangle(...);`
	`rect.name = "Tag2";`
	
	`var selectionItems = [textFrame, rect];`
	`for (var i = 0; i < selectionItems.length; i++) {`
	`  selectionItems[i].selected = true;`
	`}`
	
	`// Task code`
	`function alertIfTagNameMatches(predefined) {`
	`  try {`
	`  var items = app.activeDocument.selection;`
	`  if (items.length === 0) {`
	`  throw new Error("No␣items␣selected.");`
	`  }`
	`  var matchedTags = [];`
	`  for (var i = 0; i < items.length; i++) {`
	`  var item = items[i].name;`
	`  if (predefined.indexOf(item)!==-1) {`
	`  matchedTags.push(item);`
	`  }`
	`  }`
	`  if (matchedTags.length > 0) {`
	`  return matchedTags.join(",␣");`
	`  } else {`
	`  return "No␣matched␣tags␣found.";`
	`  }`
	`  } catch (e) {`
	`  return "Error:␣" + e.message;`
	`  }`
	`}`
	
	`var predefinedNames = ["Tag1", "Tag2", "Tag3"];`
	`alertIfTagNameMatches(predefinedNames);`

Error msg	*Not applicable because the code was successfully executed.*
Judge	The task requires alerting the user if the tag names match any in the predefined list, but the task code only returns the result as a string. There is no actual alert mechanism used. Replace the return statements with an alert function to notify the user, e.g., alert(“Matched tag(s): ” + matchedTags.join(“, ”)); to fulfill the requirement correctly.
Init. code	*Same as above*
Code	*Replaced return matchedTags.join(“, ”); with alert(“Matched tag(s):” + matchedTags.join(“, ”));*
Effect	*An alert message box is popped up.*

Table 13: Example script generated by bottom-up strategy. Details simplified for space constraints. Italicized text indicates descriptive language and not the actual output produced by LLMs.

Category	Subcategories
Drawing	Drawing basics, Draw pixel-perfect art, Edit paths, ...
Color	Adjust Colors, Select Colors, Use the Adobe Color Themes panel, ...
Painting	Gradients, Paint with fills and strokes, Brushes, ...
Select and arrange objects	Select objects, Move and align objects, Layers, ...
Reshape objects	Crop images, Transform objects, Puppet Warp, ...
Import, export, and save	Save artwork, Export artwork, Package files, ...
Type	Create text, Using fonts in Illustrator, Format paragraphs, ...
Create special effects	Work with effects, Graphic styles, Drop shadows, ...
Web graphics	Best practices, Create animations, SVG, ...
Printing	Set up documents, Print with color management, Overprint, ...

Table 14: Illustrator Functionality Overview

System Prompt for Task Creation in Illustrator

You are an expert user for Adobe Illustrator. Your goal is to generate as many tasks as possible that are helpful and represent common needs for Illustrator users.

The generated tasks should follow the following criteria:
1. Describe these tasks so that they can be coded into a script.
2. The tasks should not be already implemented in Illustrator.
3. The tasks should be minimally dependent on the content.

Generate the tasks in plain text. Each task takes one line. Do no generate any other information such as numbering.

User Prompt for Task Creation with Top-down Simulation

Give me 10 most useful tasks related to `{subcategory}` under the category of `{category}`, in Adobe Illustrator.

Examples of successful tasks in the previous rounds include:
`{a list of successful task descriptions from previous round under the same category}`

User Prompt for Task Creation with Bottom-up Simulation

Give me 10 most useful Adobe Illustrator tasks related to `{api}` whose description is: `{Names and descriptions of top k synergistic APIs}`
Take inspiration from the following APIs and their descriptions by considering the possibility of using `{api}` with at least one of the following APIs. It's okay to not use them as long as the tasks related to `{api}` are useful.

`{top_nodes_info}`

The generated tasks should follow the following guidelines:
- Make sure the tasks are reusable.
- The tasks should be logical and reasonable to use two or more APIs together.
- The generated task should not simply be a concatenation of two API nodes, i.e., do task A and do a separate task B that doesn't closely depend on task A.
- Prioritize the usefulness of the tasks over generating exactly 10 tasks—fewer, high-quality tasks are acceptable.

Table 15: Prompts for Task Creation in Illustrator

System Prompt for Code Generation

You are an assistant generating ExtendScript code for Adobe Illustrator. You will be provided a query that attempts to perform an action in Illustrator. Return only the ExtendScript code snippet without additional messages, formatting, or markdown.

Initialize a document to simulate this code. Generate the initialization code and task code separately in the following JSON format: {"init_code": INITIALIZATION_CODE, "code": CODE, "code_name": "[brief description]"}

The code must follow these rules:
1. Do not use alert. Return messages for stdout.
2. If the task is not feasible, return {"code": ""}.
3. Start error messages with "Error: ".
4. Include necessary initialization for selecting objects.
5. Ensure reusability of task code.
6. Call the function you create to execute the task.
7. Keep the initial layout minimal for clear visual results.
8. Do not crash Illustrator.

Skill Generation Prompt Structure

Task: `{task description}`

Code from the last round: `{code_last_round}`

Execution error for code from last round: `{error_msg}`

Visual evaluation for the outcome layout from code from the last round: `{validation_last_round}`

Table 16: System Prompt for ExtendScript Code Generation in Illustrator

System Prompt for Skill Validation

You will be given a task description, a piece of initialization code, a layout after running the initialization code, a piece of task code, and a layout after running the task code. The context for the task is Adobe Illustrator.

Your job is to judge whether the task was performed correctly or not, given the task description and the two layout figures. The difference between the two layout figures should reflect the result of running the task code.

Sometimes, the failure reason can be that the initialization code does not generate necessary elements for the task code to run correctly, or the task code does not perform the task correctly.

The output should be in JSON format: {"valid": true/false, "reason": [brief reason for the judgment], "suggestion": [brief suggestion for improvement]} Ensure the output contains no additional messages, formatting, or markdown, so that it can be directly parsed by json.loads().

User Prompt for Validation with LVLM

Task description: `{task_description}`
Initialization code: `{init_code}`
Task code: `{code}`

`{init_layout.png}` `{outcome_layout.png}`

Table 17: Prompts for Skill Validation

System Prompt for Code Generation

You are an assistant generating Office Scripts code for Microsoft Excel. Office Scripts use TypeScript to automate Excel tasks.

You will be provided a query that attempts to perform an action in Excel. Please return the Office Scripts code snippet without any additional message, formatting or markdown.

Generate the initialization code and task code separately, in the following json format: {"init_code": INITIALIZATION_CODE, "code": CODE, "code_name": "[brief description]"}, so that it can directly be parsed by json.loads().

The code you write should follow the following criteria:
1. Use TypeScript/JavaScript syntax for Office Scripts.
2. Make sure to use the Excel TypeScript API (Office Scripts API) - e.g., functions like workbook.worksheets, range.format, etc.
3. When it is not possible to generate the code, return "code": "". This can happen when the task is not possible to be automated or when the task is not clear.
4. The initialization code should create a clean starting environment with necessary elements to demonstrate the task code.
5. Your task code should be modular and reusable for building more complex tasks.
6. Remember to call the main function to run the task code.
7. The initial workbook layout should be simple so that the effect of running the task code is clearly visible.
8. Office Scripts for both init_code and code typically begin with: function main(workbook: ExcelScript.Workbook) { ... }

Table 18: System Prompt for TypeScript Code Generation in Excel