BUILT FOR HUMANS, NOT YET FOR AGENTS: TURNING APPS INTO AGENT-NATIVE CLIS AT SCALE

Anonymous authors
Paper under double-blind review

ABSTRACT

Applications (Apps) expose rich functionalities, such as document editing and image processing, through graphical user interfaces (GUIs). However, reliably interacting with these GUIs remains challenging for large language model (LLM) agents. Command-line interfaces (CLIs) offer an alternative, but generating commands for GUI functionalities requires preserving their implicit context and behavioral constraints. We present APP2CLI, a framework that generates *agent-native* commands for existing GUI functionalities in apps. APP2CLI traces GUI "trigger-handler-effect" paths to extract behavioral semantics and represent them in a structured semantic specification. Skill guidance directs the construction of CLI templates through public programmatic interfaces. Cooperating agents implement templates as standalone, parameterized commands.

We evaluate APP2CLI on 1,467 target GUI functionalities across 11 application units from 9 apps. APP2CLI achieves a 60.05% *agent-native* commands generation success rate , producing semantically correct commands for 881 target functionalities, . The CLI executions also achieve 98.01% line-coverage overlap with the corresponding GUI executions. Removing the semantic specification or skill-guided template construction reduces the generation success rate to 1.77% and 0.14%, respectively. Code is available at (Anonymous, 2026).

1 INTRODUCTION

Nowadays, applications (apps) offer rich functionalities, such as document editing, image processing, and engineering designs. Users invoke these through graphical user interfaces (GUIs), and apps interpret their actions to produce the corresponding results. Large language model (LLM) agents are able to use tools and execute code to complete software tasks (Yao et al., 2023; Wang et al., 2024). Computer-use environments enable LLM agents to interact with apps with multi-step workflows. (Xie et al., 2024; Drouin et al., 2024; Bonatti et al., 2025). Therefore, these advances enable LLM agents to access the GUI functionalities of apps originally designed for human users, while raising the question of how apps can expose them for reliable use.

Human users interact with apps via GUI feedback as the app state changes. Although LLM agents can access apps via GUI as well, they must locate controls, track the apps' state, and respond to the updates, which makes it difficult for LLM agents to complete tasks accurately and reliably. In an evaluation of 440 desktop tasks, the GUI agent successfully completed only 59.1% (Zhou et al., 2026). This raise an important question: **Can LLM agents access apps without directly reproducing GUI actions by human users?** Command-line interfaces (CLIs) offer an alternative to help LLM agents access apps. *Agent-native* interfaces expose GUI functionalities of apps that LLM agents can invoke directly. We define *agent-native* CLIs as parameterized commands and arguments that expose GUI functionalities with explicit inputs and behavioral requirements. LLM agents can select commands, supply arguments, and reuse CLIs across tasks without GUI interaction.

Recent work has begun to build *agent-native* interfaces. CLI-Anything (Yang et al., 2026) constructs CLI harnesses over app backends, via GUI-to-API mappings and explicit state management. However, these interfaces may provide incomplete support for basic GUI functionalities and their parameters. GUI vs. CLI (Zhou et al., 2026) reports that the original CLI-Anything interfaces satisfy only 37.6% of the criteria. Scaling *agent-native* CLIs therefore depends on commands that cover more GUI functionalities with rich parameter semantics.

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

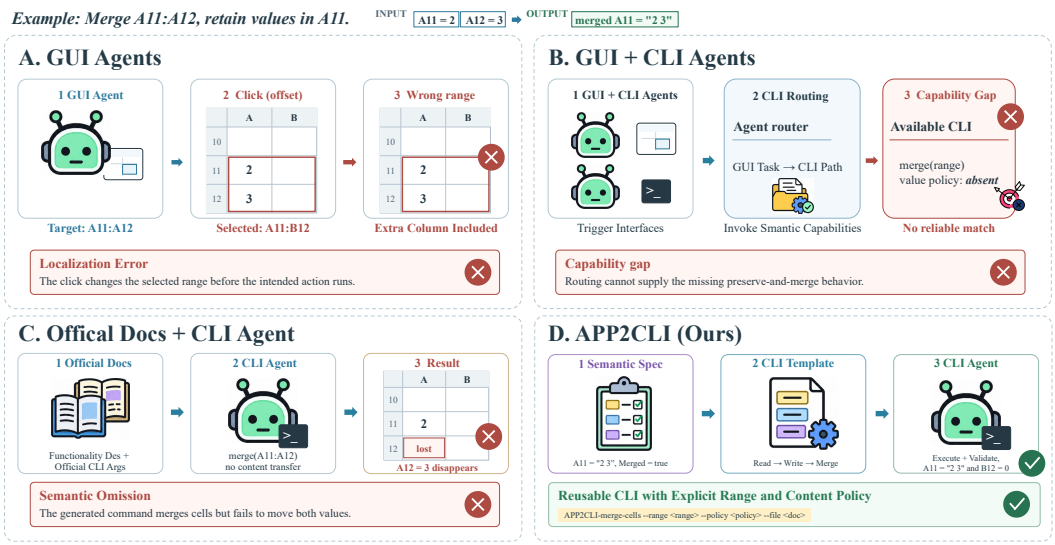


Figure 1: Illustrative comparison of a GUI agent (A), GUI & CLI agent (B), command generation via official docs (C), and APP2CLI (D) for "merging `A11:A12`".

An *agent-native* CLI must expose commands with rich arguments that explicitly capture and implement the intended GUI functionality. However, this raises two challenges. First, *what information must a command specify to invoke a GUI functionality?* The target objects, default values, and relevant states are often defined across different parts of GUI implementations. Consequently, it is hard to provide a complete specification of commands. Second, *how can these specifications be implemented?* Apps expose different scripting interfaces and object models. Reproducing a GUI functionality may involve coordinating multiple API operations and resolving the target objects they act on. Appendix A presents examples of semantic differences between GUI and CLI execution, illustrating why command generation must account for implicit context and observable effects.

Figure 1 illustrates challenges faced by exisitng approaches through "cell merging" in LibreOffice Calc (The Document Foundation, 2026b). A user selects `A11:A12`, containing `2` and `3`, and chooses to move both values into the first cell. The expected result is `"2 3"` in `A11` and an updated value of `0` in the cell `B12`. (A) A GUI agent may select the wrong range, (B) a hybrid agent may lack the commands needed for the task, and (C) a command generated from official documentations may fail to preserve both cell values. Therefore, it is necessary to trace how the source code handles the GUI selection and dialog choice to capture the intended behavior. Although the API method `XMergeable.merge` accepts only a **Boolean** flag, the input arguments jointly determine both the target range and how its values are combined (Foundation, 2026a;b). (D) An *agent-native* command therefore should expose arguments that specify the range and whether to retain values .

Our key insight is that source code in app already encodes the behavioral semantics needed for *agent-native* command generation. Therefore, we present APP2CLI, a framework that generates *agent-native* commands access to GUI functionalities in apps. APP2CLI traces "trigger-handler-effect" paths to specify parameters, preconditions, effects, and non-target invariants. Skill guidance then translates these specifications into reusable CLI templates that specify public interfaces and execution steps. Cooperating agents then implement and validate those as *agent-native* commands.

We evaluate APP2CLI on 1,467 target GUI functionalities across 11 application units from 9 apps. APP2CLI generates semantically correct commands for 881 targets (60.05%), supporting 21.5× as many targets as CLI-ANYTHING. Removing the semantic specification or skill guidance reduces success to 1.77% and 0.14%, respectively. The average GUI-to-CLI coverage overlap reaches 98.01% within semantic-core units, indicating that both routes execute largely the same core logic. These results show that APP2CLI expands *agent-native* commands, allowing LLM agents to invoke GUI functionalities with explicit arguments.

Our contributions are as follows:

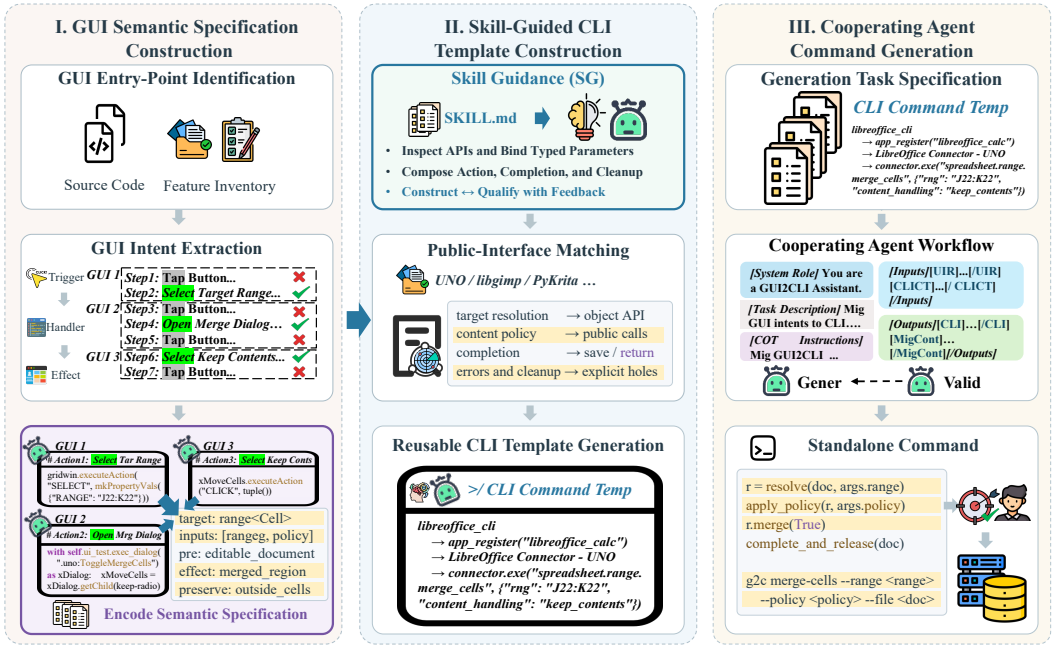


Figure 2: The Overall Framework of APP2CLI. Left: extract the semantic specification $\mathcal{R}_g$ from $\mathcal{S}$ and $\mathcal{D}$. Middle: construct a CLI template using $\mathcal{A}$. Right: implement the command $c_g(\theta)$.

- We present APP2CLI, a framework that expands *agent-native* access to GUI functionalities in apps by generating standalone, parameterized commands.
- We represent GUI semantics as explicit specifications and use skill guidance to construct CLI templates. Cooperating agents then implement standalone, parameterized *agent-native* commands.
- We evaluate APP2CLI on 1,467 GUI functionalities across 11 application units from 9 apps. APP2CLI achieves a 60.05% *agent-native* commands generation success rate, supporting 881 target functionalities, 21.5× as many as CLI-ANYTHING. The CLI executions also achieve an average line-coverage overlap of 98.01% with the corresponding GUI executions.

2 APPROACH

Problem Formulation. Let $\mathcal{S}$, $\mathcal{D}$, and $\mathcal{A}$ denote an app's source code, official documentation, and public programmatic interfaces, respectively. A *GUI functionality* is a user-level function, such as "cell merging", whose *behavior* comprises the invoking effects under given inputs and app state. For each functionality g, our goal is to generate an *agent-native* command $c_g(\theta)$ that LLM agents can invoke with explicit arguments θ to perform the corresponding operation. Here, s denotes the initial app state and θ a concrete assignment to the parameters of g. We specify the command's intended behavior as $\mathcal{R}_g = \langle P_g, E_g, N_g \rangle$, where $P_g(s, \theta)$ defines valid initial states and inputs, E_g describes the intended effects, and N_g specifies invariants on relevant non-target states.

$$\forall s, \theta, \quad P_g(s, \theta) \Rightarrow [\![c_g(\theta)]\!](s) \equiv_{E_g, N_g} [\![g(\theta)]\!](s). \tag{1}$$

Here, $g(\theta)$ denotes the GUI implementation of functionality g with the corresponding inputs, and $[\![\cdot]\!](s)$ denotes its execution outcome from state s. The relation $\equiv_{E_g, N_g}$ compares outcomes with respect to the specified effects and non-target invariants. Figure 2 presents a three-phase workflow for *agent-native* command generation. Phase I extracts $\mathcal{R}_g$ and the parameter roles in θ from $\mathcal{S}$ and $\mathcal{D}$ and records them in a semantic specification. Phase II constructs a CLI template that specifies how interfaces in $\mathcal{A}$ realize the behavior. Phase III implements the template as a command $c_g(\theta)$.

2.1 PHASE I: GUI SEMANTIC SPECIFICATION CONSTRUCTION

An LLM agent cannot reliably reproduce a GUI functionality's behavior without understanding which objects the user selected and which settings the user chose. Starting from a manually specified inventory of GUI functionalities, Phase I extracts the parameter roles in θ and $\mathcal{R}_g$ for each g and records their source locations in a semantic specification. The specification records the inputs, expected effects, and invariants for command generation. This guides interface selection in Phase II.

Step 1: GUI Entry-Point Identification. The upper-left block of Figure 2 starts from an app and a target functionality g. APP2CLI locates its entry points in $\mathcal{S}$, then follows signals and callbacks to the relevant handlers. Official documentation $\mathcal{D}$ helps interpret the exposed options and parameter meanings. The resulting fragments identify where the app obtains inputs and initiates execution g.

Step 2: GUI Intent Extraction. APP2CLI follows the GUI "trigger-handler-effect" path to trace how targets and parameters flow through the handlers. It combines fragments that act on the same target and jointly implement g, identifying conditions in P_g, effects in E_g, and invariants in N_g. For the "cell merging" example, the selected range identifies the target, while the dialog option determines how existing contents are handled. Both must be explicit inputs in θ: the target range and content-handling option. Source code only used for display and navigation is excluded, while the effect of confirming the merge is captured in E_g. Each GUI intent is therefore defined by the app behavior rather than a number of GUI events.

Step 3: Encode Semantic Specification. The extracted information is encoded as a structured specification of g: its target, parameter types and defaults for θ, preconditions P_g, expected effects E_g, and non-target invariants N_g. Each field links to the relevant part of $\mathcal{S}$ or $\mathcal{D}$. The specification also records the completion boundary and state category, distinguishing persisted changes, transient effects, returned values, and identity-preserving functionalities. The specification defines how to determine whether the command has completed its intended GUI operation, even when the app state remains unchanged. The specification guides template construction in Phase II.

2.2 PHASE II: SKILL-GUIDED CLI TEMPLATE CONSTRUCTION

Explicit inputs alone do not provide *agent-native* access. The behavior must also be realized through the app's APIs. Phase II turns $\mathcal{R}_g$ into a reusable CLI template q_g that identifies the relevant interfaces in $\mathcal{A}$, specifies their execution order, and retains typed slots for θ. The template specifies how to locate the target objects and which API operations to execute in what order. Phase III implements the template as an *agent-native* command, then builds and validates it.

Step 1: Skill Guidance. Given the semantic specification, $\mathcal{S}$, and $\mathcal{D}$, a "skill" agent follows reusable construction procedures in `SKILL.md`. Organized by functionality family, these procedures guide interface inspection, typed parameter binding, and the organization of execution, completion, and cleanup steps. They are organized by functionality family rather than individual case answers. The agent performs at most two self-review rounds without execution to validate specification completeness and template consistency against $\mathcal{R}_g$.

Step 2: Public-Interface Matching. The "skill" agent selects public interfaces in $\mathcal{A}$ to implement the behavior specified by $\mathcal{R}_g$, such as UNO (Foundation, 2026c), libgimp (Team, 2026a), and PyKrita (Team, 2026b), etc. The arguments in θ identify the target app objects, while the template q_g organizes API operations to produce the expected effects E_g and preserve the non-target invariants N_g. For example, LibreOffice-Calc's "cell-merging" dialog allows users to move the selected cells' contents into the first cell. The merge API, `merge(boolean)`, only controls whether cells are merged and does not specify how their contents should be handled. To implement the "move contents to the first cell" option (`move_to_first_cell`), the template retrieves the non-empty values, clears the cells, writes the combined content to the first cell, and finally merges the range.

Step 3: Reusable CLI Template Generation. The resulting template q_g specifies the public interfaces, parameter bindings, and execution sequence to implement g according to $\mathcal{R}_g$. Document paths, cell ranges, and content-handling options remain parameter slots in θ. Phase III implements

the template as a command that exposes these parameters as explicit arguments. The template can be reused across functionalities with target types, parameter roles, preconditions, and effects.

2.3 Phase III: Cooperating Agent Command Generation

Phases I and II provide the semantic specification and CLI template for each GUI functionality. Phase III turns those into an executable, reusable command through a cooperating-agent workflow. It first defines the specific generation task, coordinates implementation and validation, and finally generates the command with explicit arguments for *agent-native* use.

Step 1: Generation Task Specification. Phase III implements the CLI template as an *agent-native* command. It combines the semantic specification and q_g, shown in the upper-right block of Figure 2. The specification defines the valid inputs and states in P_g, the effects in E_g, and the invariants in N_g. q_g supplies the public interfaces and execution sequence. Together, they specify the behavior of $c_g(\theta)$ and how its arguments control that behavior.

Step 2: Cooperating Agent Workflow. A "Gener" agent generates a candidate command c_g by locating the target objects and matching the arguments θ to the API parameters specified in q_g. It also records the implementation constraints needed to satisfy $\mathcal{R}_g$. A "Valid" agent builds and runs the command, validating whether a candidate command meets the specification and constraints. If validation fails, failure details is sent to the "Gener" agent for revision, with at most two attempts.

Step 3: Standalone Command. The implementation is provided as a standalone command $c_g(\theta)$. For the "cell merging" example, the command opens the specified document, locates the target range, handles its contents according to the chosen option, calls the merge API, and finalizes the changes before releasing resources. The document path, range, and content-handling option are explicit arguments, allowing an LLM agent to reuse the command with different inputs meeting P_g.

The three phases enable *agent-native* command generation by extracting GUI semantics from source code, constructing CLI templates via the app's public interfaces, finally implementing those templates as reusable, standalone, and parameterized commands.

3 Experimental Setup

3.1 Datasets

CLI-Anything (Yang et al., 2026) provides CLI interfaces for apps, and GUI vs. CLI (Zhou et al., 2026) evaluates how effectively LLM agents use these interfaces. To align our evaluation with prior work, we draw our candidates from the same 16 apps, 18 app units in total, used in GUI vs. CLI . We focus on native desktop apps with publicly available core source code, excluding Chrome (Google, 2026), Obsidian (Obsidian, 2026), and Zoom (Zoom, 2026) due to source availability, as well as Zotero (Corporation for Digital Scholarship, 2026) and draw.io (JGraph, 2026) because they use web technologies. Two of the remainings 11 apps are also excluded due to execution constraints. The configuration of RenderDoc (RenderDoc, 2026) requires x86_64 Linux, whereas our experimental environment uses Linux aarch64. In CloudCompare (CloudCompare, 2026), the Qt5 DB tree does not expose entities through AT-SPI, and X11 and keyboard fallbacks cannot support reliable selection for reproducible GUI execution. The resulting 9 apps span 6 domains based on their primary functionality, seen in Table 1.

To control evaluation costs, we identify candidate source units in each app using a predefined vocabulary of framework-specific GUI triggers and extract target GUI functionalities from these units, detailed in the Appendix C. For each extracted GUI functionality, we define an *intent* that specifies its target objects, parameters, preconditions, and expected effects. This description defines the behavior that the generated command is expected to implement. For example, GIMP (Spencer Kimball & the GIMP Development Team., 2026) contributes five target functionalities from its image-transformation family: horizontal and vertical flipping, clockwise and counterclockwise 90-degree rotation, and 180-degree rotation. Shotcut (Developers, 2026b) contributes 43 target functionalities for selecting predefined project video modes. These targets represent distinct operations and param-

Table 1: 9 Applications, 11 Application Units, across 6 Domains.

Application Domain	Application Unit	Version	Target Intents
Office Productivity - LibreOffice	LibreOffice Writer	26.2.4.2	204
	LibreOffice Calc	26.2.4.2	535
	LibreOffice Impress	26.2.4.2	42
Image Editing and Digital Painting	Krita	5.3.1	36
	GIMP	3.2.4	5
Audio Editing and Music Notation	Audacity	3.7.8	78
	MuseScore	3.6.2	93
Video Editing and Streaming	Shotcut	26.1.30	43
	OBS Studio	32.2.2	41
Computer-aided design	FreeCAD	1.1.3	21
Game development	Godot	4.7.1	369
Total			1,467

eterized variants within the selected functionality families. Across all apps, we identify 1,467 target GUI functionalities from 1,003 source units. Table 5 reports both granularities for each app.

3.2 BASELINES

We compare APP2CLI with direct command generation baseline (DCGBASELINE) and CLI-Anything. We evaluate whether APP2CLI generates more high-quality *agent-native* commands that expose GUI functionalities than DCGBASELINE and CLI-Anything. **Direct Command Generation Baseline (DCGBASELINE).** DCGBASELINE generates *agent-native* commands via apps' official documentation, including natural language descriptions of native commands and detailed arguments. We separately evaluate how many commands DCGBASELINE successfully generates and whether they preserve the intended GUI behavior.

CLI-Anything. We evaluate the latest CLI-Anything release available as of August 21, 2026 (Developers, 2026a), comparing its 564 commands with those generated by APP2CLI in terms of command count and preservation of the intended GUI behavior.

3.3 EVALUATION METRICS

Correctness and Coverage. We evaluate each generated command for **Executability** and **Semantic Equivalence**. A command receives PASS only if it executes successfully and produces the specified effects E_g while preserving the non-target invariants N_g for every evaluated input and state satisfying P_g. Otherwise, it receives FAIL. This assessment merely validates the observations specified in $\mathcal{R}_g$ and does not guarantee end-to-end behavioral equivalence in Equation 1. Therefore, we compute the **commands generation success rate** as the fraction of all target GUI functionalities for which a generated command receives PASS.

The commands of baselines and APP2CLI are at different granularities: baseline ones typically expose high-level GUIs that may encompass multiple GUI intents. We report intent-level coverage (matched functionalities) and source-unit coverage (units with a matched intent).

Line Coverage. Line coverage measures the source lines executed by the generated commands and, through comparison with GUI execution, provides further support of whether the two routes reach similar core logic. For LibreOffice, we report line coverage at three scopes: the entire application; its core functionality modules, including Writer (`sw/`), Impress (`sd/`), and Calc (`sc/`); and the semantic-core units that implement the evaluated GUI functionalities, excluding setup and teardown code. To quantify how much of the GUI-executed source lines is also reached through CLI execution, we also define GUI-to-CLI coverage overlap as the proportion of GUI-executed source lines also covered by CLI execution:

$$\text{Overlap}_{G \to C} = \frac{|L_G \cap L_C|}{|L_G|}, \tag{2}$$

where $L_G = \bigcup_{g \in G} \text{Lines}(g)$ and $L_C = \bigcup_{c \in C} \text{Lines}(c)$. Here, G and C denote the GUI and CLI execution, respectively, and $\text{Lines}(x)$ denotes the source lines covered by execution x.

3.4 VALIDATION

Successful execution may still differ from GUI behavior (Ng et al., 2026). We therefore use Source-Test Oracle Replay (STOR) for supplementary validation in LibreOffice Writer, Calc, and Impress, whose mature tests provide reusable action boundaries, fixtures, document lifecycles, and assertions. Reconstructing those for other apps may introduce bias. Appendix Figure 4 shows the workflow.

I. Construct GUI Replay Oracle. STOR first confirms that the source test passes, then records its code, fixture version, arguments, action boundary, intermediate states, and source test assertions.

II. Replay Source Test Oracles Assertions with Generated Commands. STOR first **runs** the prefix GUI operations, **invokes** the command, **replays** the subsequent operations and **validates** the unchanged source test oracles assertions. For session-dependent test oracles assertions, STOR attaches the command to the live GUI session to retain runtime handles and relevant session context.

III. Interpret the Replay Outcome. STOR reports PASS or FAIL, distinguishing command execution errors from assertion failures. A PASS indicates that commands satisfy the retained source test oracles assertions. We then analyze the main causes of failure and record runtime state information.

4 EVALUATION

4.1 RQ1: GENERATION EFFECTIVENESS

APP2CLI supports substantially more target functionalities than the baselines. Table 2 compares APP2CLI with the baselines under the metrics in Section 3.3. APP2CLI generates commands for 905 of 1,467 target functionalities, of which 881 pass semantic validation. APP2CLI thus supports 60.05% of target functionalities and fully supports 517 of 1,003 source units (51.55%). In contrast, CLI-Anything matches only 41 target functionalities (2.80%) through static matching.

Documentation lacks the details needed to implement target functionalities. DCGBASELINE generates commands for only 23 of 592 documentation units, leaving 569 unsupported. These commands cover at least one functionality in each of 32 benchmark source units (3.19%), but none fully implements a target intent. The problem is that documentation describes broad workflows without specifying the parameters, preconditions, and effects of individual GUI functionalities. This leaves insufficient detail to generate commands for most units, while the commands generated are too coarse-grained. APP2CLI obtains these details directly from GUI source code by tracing how each functionality processes inputs and changes app state.

A large command inventory does not imply coverage of target functionalities. CLI-Anything provides 564 commands across nine apps, but only 20 (3.55%) support the semantic GUI functionalities. Together, these 20 commands cover 41 target functionalities (2.80%) across 30 source units (2.99%). First, 169 commands (30.0%) in CLI-Anything perform auxiliary operations, such as listing tracks or layers and exporting artifacts. These commands do not necessarily implement the semantic target GUI functionalities. Second, commands in CLI-Anything are designed for high-level

Table 2: Comparison of APP2CLI and the Baselines.

Method	Generated / Released	Unsupported	Source-Unit Support	Intent-Level Support
APP2CLI	905	562	517/1,003 (51.55%)	881/1,467 (60.05%)
DCGBASELINE	23	569	32/1,003 (3.19%)	0/1,467 (0.00%)
CLI-Anything	564	N/A	30/1,003 (2.99%)	41/1,467 (2.80%)

Table 3: Ablation Results of APP2CLI.

Variant	Generated	Unsupported	Source-Unit Success	Cmds. Gen. Success
APP2CLI	905	562	517 (51.55%)	881 (60.05%)
w/o semantic specification	100	1,367	26 (2.59%)	26 (1.77%)
w/o skill guidance	14	1,453	1 (0.10%)	2 (0.14%)

tasks and may not support the specific inputs or effects required by finer-grained GUI functionalities. Thus, the number of commands alone does not indicate the scope of *agent-native* access.

Generated commands reuse the app's core logic. To assess whether generated commands reuse the app logic executed by the GUI, we compare GUI and CLI source-line coverage at the application and semantic-core scopes. Appendix Table 6 reports the results, detailed in Appendix D.3. Across eleven application units, GUI and CLI coverage average 21.12% and 17.60%, respectively, with an average overlap of 72.11%, ranging from 21.35% (Shotcut) to 98.74% (Audacity). Within semantic-core units implementing the target functionalities, average overlap reaches 98.01% across six comparable apps. GIMP, Godot 4, and Shotcut are marked N/A because their GUI and CLI execution routes lack a comparable semantic-core scope. These results support reuse of core logic but do not alone guarantee behavioral equivalence.

Unchanged upstream assertions confirm behavioral equivalence. STOR replaces GUI actions with generated commands in the original source tests, while preserving upstream fixtures, action boundaries, document lifecycles, and assertions. We evaluate it on LibreOffice, whose test suites allow direct reuse of these elements without potential bias from reconstruction. Of 376 supported targets from fully supported source-test cases, 124 allow substitution under these constraints, of which 114 pass by STOR (91.94%) (Appendix Table 7), and the remainings may lack an isolatable GUI action or depend on the original GUI session. The ten failures mainly involve Undo granularity, GUI-state observation after CLI execution, or unstable source tests, which can prevent the effectiveness of STOR even when commands produce the intended effects.

> **Answering RQ1:** APP2CLI generates semantically correct commands for 881 GUI functionalities (60.05%), compared with 41 supported by CLI-Anything. Semantic-core overlap is 98.01% on average, and 91.94% LibreOffice targets pass via STOR.

4.2 RQ2: Component Contributions

We ablate the two components that support *agent-native* commands generation for target GUI functionalities: the *semantic specification*, which describes the intended behavior, and *skill guidance*, which directs CLI template construction via the app's public interfaces. Each variant removes one component while retaining the remaining pipeline. Table 3 summarizes the results.

Both components largely improve *agent-native* commands generation success. APP2CLI produces candidate commands for 905 target functionalities with a generation success rate of 60.05%. Removing the semantic specification or skill guidance reduces candidate commands generation to 100 and 14 targets, and success rates to 1.77% and 0.14%, respectively. Source-unit success similarly falls from 51.55% to 2.59% and 0.10%, respectively. Without these components, the generator must infer implicit GUI context or identify suitable interfaces without template guidance. To balance revision opportunities against token and time costs, Phase III allows only two construction attempts per target. These additional uncertainties may therefore leave more targets unresolved within the same budget. Removing either component also reduces the semantic correctness of the generated commands. Among targets with candidates, 97.3% (881/905) pass semantic validation with APP2CLI, versus 26.0% without the semantic specification and 14.3% without skill guidance.

Semantic specifications help preserve GUI behavior during generation. In the Calc Function Wizard example (Appendix E.1), APP2CLI follows the specification to enter and commit a formula, converting GUI separators for the public interface to obtain `2`. Without the specification, the

command executes but skips this conversion, producing `Err:508`. This example illustrates how the semantic specification guides interface selection and argument handling to preserve GUI behavior.

Skill guidance supports valid app invocation. In the FreeCAD deletion example, the variant without skill guidance retains the object-removal logic, but passes `--document` directly to `FreeCADCmd`. The option is rejected before the deletion logic runs. With skill guidance, APP2CLI uses a supported invocation route and matches the GUI outcome in the observed runtime state. Thus, implementing the functionality must follow the app's invocation conventions.

> **Answering RQ2:** Both components contribute substantially to *agent-native* command generation. The specification helps preserve the intended GUI behavior, while skill guidance supports its implementation through valid app interfaces.

4.3 RQ3: Cost Evaluation.

Command Generation Cost and Functionality Coverage. Appendix Table 8 summarizes total generation time and token consumption. APP2CLI generates semantically correct commands for 881 target GUI functionalities, with a total generation time of 11.41 hours and a token consumption of 76,258,286. In comparison, DCGBaseline takes 1.29 hours and consumes 2,793,856 tokens. Its successful outputs support higher-level documented workflows, corresponding to 32 source units, but do not fully match the finer-grained GUI functionalities specified in our benchmark. Without skill guidance, generation takes 4.90 hours and consumes 5,989,941 tokens, producing semantically correct commands for only two target functionalities. The cost comparison must therefore account for differences in both the scope and granularity of the supported functionality.

Effect of Semantic Specifications on Generation Cost. Removing the semantic specification reduces token cost from 76.26 to 41.97 million, but increases generation time from 11.41 to 11.75 hours and decreases the number of semantically correct commands generated from 881 to 26. The semantic specification guides public interface selection and implementation with explicit inputs, expected effects, and state constraints, thus reducing repeated reasoning and ineffective revisions. Although APP2CLI consumes more tokens, it generates semantically correct commands for more targets in relatively less time.

> **Answering RQ3:** APP2CLI generates semantically correct *agent-native* commands for 881 target GUI functionalities with 11.41 hours of generation time and 76.26 million tokens. These results highlight the trade-off between generation cost and the range of GUI functionalities.

5 Related Work

CLI Interfaces for Apps. OpenCLI (developers, 2026) provides command access to websites, browser sessions, selected Electron applications, and local tools. CLI-Anything (Yang et al., 2026) constructs stateful CLI harnesses through backend integration and state management.

GUI, CLI, and Hybrid Task Completion. GUI vs. CLI (Zhou et al., 2026) identifies gaps in CLI skill coverage and behavioral contracts as key limitations. Its analysis shows that a callable CLI entry point alone does not guarantee that the target GUI functionality's specified behavior is implemented. CUA-Universe (Shi et al., 2026) supports hybrid GUI/CLI agents on GUI tasks through three components: App-Forge builds environments and CLI interfaces, Task-Weave synthesizes tasks, and Path-Steer guides hybrid execution and collects trajectories.

6 Conclusion

We presented APP2CLI for generating reusable, parameterized commands through explicit GUI semantics, skill-guided templates, and cooperating agents. Across nine applications and 1,467 target GUI functionalities, APP2CLI generates semantically correct commands for 881 targets (60.05%). Removing the semantic specification or skill guidance reduces success to 1.77% and 0.14%, respectively. These results show how APP2CLI expands *agent-native* access by exposing existing GUI functionalities as directly invocable commands.

AI USE STATEMENT

We used ChatGPT and Codex to discuss and refine research ideas and methodology, assist with implementation and debugging, and translate and revise manuscript text. LLMs are components of APP2CLI for extracting GUI semantics, constructing CLI templates, and generating commands, as described in Section 2. Their configuration is reported in Appendix D.1.

The authors consulted the original cited sources, reviewed the code and execution results, verified reported statistics, and reviewed the manuscript. The authors made the final research and presentation decisions and take responsibility for all AI-assisted text, claims, and artifacts.

ETHICS STATEMENT

We investigate *agent-native* command generation for open-source apps to support software automation, with no study-specific ethical concerns identified.

REPRODUCIBILITY STATEMENT

Our artifacts are publicly available at (Anonymous, 2026). They include the environment configurations, implementation details, experimental results, and evaluation scripts. Section 2 describes the APP2CLI, and Algorithm 1 specifies the detailed procedure. Section 3 explains the dataset, baselines, evaluation metrics, and validations. Appendix D.1 provides the model and execution settings, while Appendix D.3 details the coverage measurements. These descriptions and artifacts support inspection and reproduction of the reported experiments.

REFERENCES

Anonymous. Data availability. https://zenodo.org/records/22970644, 2026. URL https://zenodo.org/records/22970644. Accessed: 2026-9-12.

Rogerio Bonatti, Dan Zhao, Francesco Bonacci, Dillon Dupont, Sara Abdali, Yinheng Li, Yadong Lu, Justin Wagle, Kazuhito Koishida, Arthur Bucker, Lawrence Keunho Jang, and Zheng Hui. Windows agent arena: Evaluating multi-modal OS agents at scale. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, volume 267 of *Proceedings of Machine Learning Research*. PMLR / OpenReview.net, 2025. URL https://proceedings.mlr.press/v267/bonatti25a.html.

CloudCompare. Cloudcompare. https://www.cloudcompare.org/, 2026. Accessed: 2026-9-11.

Corporation for Digital Scholarship. Zotero. https://www.zotero.org/, 2026. Accessed: 2026-9-11.

OpenCLI developers. "opencli". https://github.com/jackwener/OpenCLI/, 2026. Accessed: 2026-9-11.

The CLI-Anything Developers. Cli-anything: Making all software agent-native. https://github.com/HKUDS/CLI-Anything/tree/810c18b0d1ab9b234bc996c9fd999318523a3ef0, 2026a. Accessed: 2026-9-11.

The Shotcut Developers. Shotcut. https://www.shotcut.org/download/, 2026b. Accessed: 2026-9-11.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, David Vázquez, Nicolas Chapados, and Alexandre Lacoste. Workarena: How capable are web agents at solving common knowledge work tasks? In Ruslan Salakhutdinov, Zico Kolter, Katherine A. Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, volume 235 of *Proceedings of Machine Learning Research*, pp. 11642–11662. PMLR /

OpenReview.net, 2024. URL https://proceedings.mlr.press/v235/drouin24a.html.

The Document Foundation. Merge and center cells. LibreOffice Help, 2026a. URL https://help.libreoffice.org/latest/en-US/text/scalc/01/05060000.html. Accessed: 2026-9-11.

The Document Foundation. Xmergeable interface reference. LibreOffice SDK API Reference, 2026b. URL https://api.libreoffice.org/docs/idl/ref/interfacecom_1_1sun_1_1star_1_1util_1_1XMergeable.html. Accessed: 2026-9-11.

The Document Foundation. Uno objects, functions and services. https://help.libreoffice.org/latest/en-US/text/sbasic/shared/uno_objects.html, 2026c. Accessed: 2026-9-11.

Google. Google Chrome. https://www.google.com/chrome/, 2026. Accessed: 2026-9-11.

JGraph. draw.io. https://www.drawio.com/, 2026. Accessed: 2026-9-11.

Albus W. Ng, Yi Han, Jusheng Zhang, and Wenhao Wang. Agent safety should be a runtime contract. *CoRR*, abs/2608.11274, 2026. doi: 10.48550/ARXIV.2608.11274. URL https://doi.org/10.48550/arXiv.2608.11274.

Obsidian. Obsidian. https://obsidian.md/, 2026. Accessed: 2026-9-11.

RenderDoc. Renderdoc. https://renderdoc.org/, 2026. Accessed: 2026-9-11.

Haoting Shi, Wenhao Wang, Weicheng Fang, Yaozhong Liang, Tian Jin, Pengxiang Zhao, Guangyi Liu, Siheng Chen, and Yanfeng Wang. Cua-universe: A scalable and dynamic environment for hybrid gui+ cli agents. *arXiv preprint arXiv:2609.05374*, 2026.

Peter Mattis Spencer Kimball and the GIMP Development Team. Downloads. https://www.gimp.org/downloads/, 2026. Accessed: 2026-9-11.

GIMP Development Team. Gnu image manipulation program 3.3 development branch. https://github.com/GNOME/gimp/blob/master/libgimp/gimp.h, 2026a. Accessed: 2026-9-11.

PyKrita Development Team. How to make a krita python plugin. https://docs.krita.org/en/user_manual/python_scripting/krita_python_plugin_howto.html, 2026b. Accessed: 2026-9-11.

The Document Foundation. SheetSortDescriptor2 service reference. LibreOffice SDK API Reference, 2026a. URL https://api.libreoffice.org/docs/idl/ref/servicecom_1_1sun_1_1star_1_1sheet_1_1SheetSortDescriptor2.html. Accessed September 11, 2026.

The Document Foundation. Libreoffice is an open source office suite created by you, for you. https://www.libreoffice.org/discover/calc/, 2026b. URL https://www.libreoffice.org/discover/calc/. Accessed September 11, 2026.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In Ruslan Salakhutdinov, Zico Kolter, Katherine A. Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, volume 235 of *Proceedings of Machine Learning Research*, pp. 50208–50232. PMLR / OpenReview.net, 2024. URL https://proceedings.mlr.press/v235/wang24h.html.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang

(eds.), *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/5d413e48f84dc61244b6be550f1cd8f5-Abstract-Datasets_and_Benchmarks_Track.html.

Yuhao Yang, Tianyu Fan, and Chao Huang. Cli-anything: Towards agent-native computer use. *CoRR*, abs/2606.03854, 2026. doi: 10.48550/ARXIV.2606.03854. URL https://doi.org/10.48550/arXiv.2606.03854.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Xiao Zhou, Siyue Zhang, Yilun Zhao, Jinbiao Wei, Tingyu Song, Arman Cohan, and Chen Zhao. GUI vs. CLI: execution bottlenecks in screen-only and skill-mediated computer-use agents. *CoRR*, abs/2606.24551, 2026. doi: 10.48550/ARXIV.2606.24551. URL https://doi.org/10.48550/arXiv.2606.24551.

Zoom. Zoom. https://www.zoom.com/, 2026. Accessed: 2026-9-11.

A Semantic Differences between GUI Operations and Generated Commands

The following LibreOffice Writer examples show how commands can execute successfully yet produce different results from the corresponding GUI operations. These examples show differences between GUI and CLI execution in object identity, undo behavior, and header or footer content.

Object Identity. In both the Footnote and Endnote cases, the document already contains one empty note before text entry. The GUI writes `LibreOffice` into that note, leaving the note count at one. The command produces the same text but creates an additional note, increasing the count to two. The extra object changes the starting state of subsequent Undo and Redo operations, so their later discrepancies propagate the initial error. Matching text alone therefore misses an important distinction: the GUI edits an existing object, whereas the command creates another. Command generation must identify which object the GUI operation modifies and preserve its identity and the collection size when only its contents change.

Undo Behavior. The Comment Undo case first creates a comment and a reply, with matching observed results through the GUI and CLI routes. Four GUI undo steps subsequently remove the corresponding text fields, whereas four undo steps through the CLI leave the fields and their author information present. The same step count therefore does not necessarily reverse the same edits, even when the preceding observed results match. Command generation must account for the editing history on which subsequent undo operations may act.

Header/Footer Content. In the Header/Footer case, both executions begin with left-page header text containing `ODD` in the `Standard` page style. Disabling left/right header sharing through the GUI restores content containing `EVEN`, while the command leaves the `ODD` content unchanged. The subsequent footer operation similarly restores left-page content through the GUI but leaves it unchanged through the CLI. Command generation must capture the content restoration associated with a sharing change, beyond updating the Boolean option itself.

Together, these cases clarify the context and effects that semantic specifications must capture for generated commands to reproduce GUI behavior.

B Motivating Examples

Two LibreOffice Calc examples illustrate how APP2CLI exposes GUI functionalities as standalone, parameterized *agent-native* commands. The sorting example shows how implicit GUI context becomes explicit command arguments (Figure 3(a)). The merging example shows how a template combines public interfaces to implement behavior that a single parameter cannot express (Figure 3(b)).

Case 1: Explicit Arguments for GUI Context. In Figure 3(a), a user selects `A1:B5`, presses `TAB` once to activate column B, and invokes ascending sort. The corresponding command must sort by column B, while preserving the header and associations between values in each row. Simply wrapping a sort function and using the first column by default would sort by column A rather than the intended column B. This mismatch arises because invoking *Sort Ascending* does not identify these parameters: the selection determines the target range, while the active column determines the key. Generating the intended command therefore requires extracting these inputs from the GUI context, beyond identifying an available sorting API.

The semantic specification makes the range, sorting key, direction, and header handling explicit. Here, the single `TAB` step activates column B, corresponding to key index 1 within the selected range. The template maps these inputs to Calc's sort-descriptor properties for key selection, sort direction, row sorting, and header handling (The Document Foundation, 2026a). Cooperating agents implement the template as a standalone, parameterized command. For the illustrated input, the resulting column B is `Misc, d, f, g, s`, matching the GUI outcome. The range `A1:B5` and key column B are supplied as arguments, enabling agents to specify the sorting behavior without recreating GUI selection and keyboard interactions.

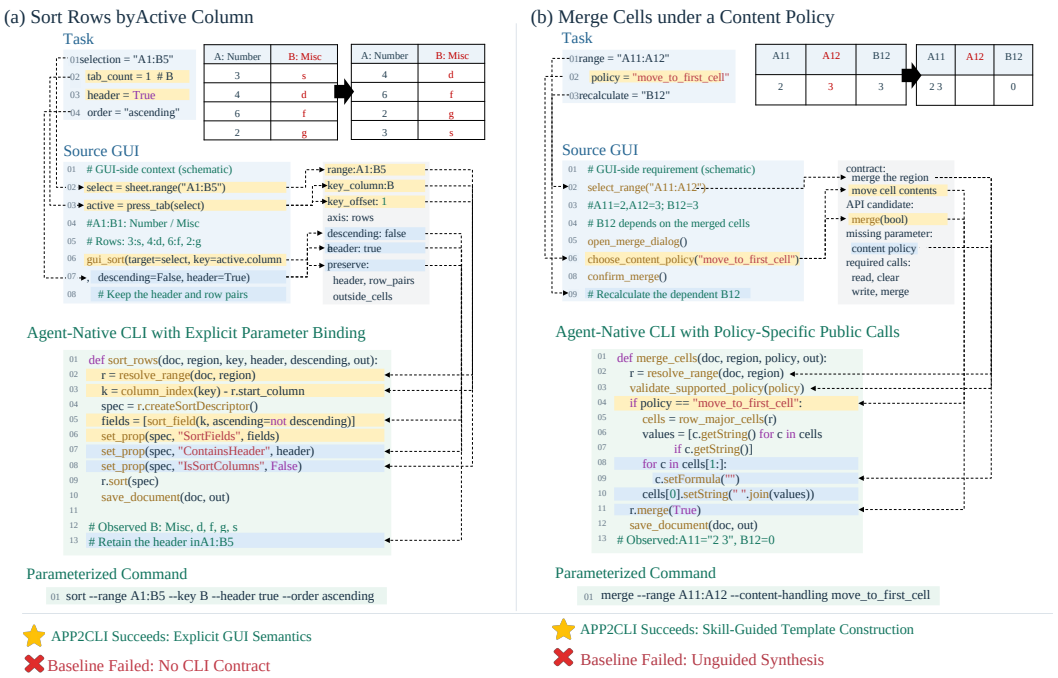


Figure 3: Agent-native command generation in LibreOffice Calc: sorting with explicit parameters and merging with content preservation.

Case 2: Implementing content handling beyond a single API. In Figure 3(b), a user selects `A11:A12`, containing `2` and `3`, and chooses `move_to_first_cell`. The intended result places `"2 3"` in `A11` and updates the dependent cell `B12` from `3` to `0`. Calc exposes several content-handling options through its GUI (Foundation, 2026a), whereas `XMergeable.merge` accepts only a Boolean merge flag (Foundation, 2026b). Invoking `merge(True)` alone cannot express the selected content-handling behavior. *Agent-native* command generation must therefore implement this behavior through the available interfaces and expose it as an explicit option.

Skill-guided template construction translates the specified behavior into a sequence of public API operations. The template resolves the range, reads its non-empty contents, clears the non-leading cells, writes the concatenated text to the first cell, and then merges the range. Cooperating agents implement these steps as a standalone, parameterized command. Its execution on the illustrated input produces `A11="2 3"` and `B12=0`, matching the GUI outcome. The document, range, and content-handling option remain explicit arguments, so the generated interface exposes a reusable merging functionality rather than an operation fixed to these cells.

Together, these examples show how explicit semantic specifications and skill-guided templates support *agent-native* command generation. The specifications capture implicit inputs and intended behavior, while the templates organize the interfaces needed to implement them. The resulting commands expand the GUI functionalities that agents can invoke directly through explicit arguments.

C Framework-specific GUI Trigger Vocabulary.

To discover GUI functionalities, we do not start from application-level GUI operation names, such as “merge,” “crop,” or “delete.” Instead, we start from how GUI frameworks expose these operations to users. For each framework, we build a vocabulary of the constructs that register GUI entry points and bind them to event handlers (Table 4). Therefore, the vocabulary can be extended with new constructs or new frameworks . APP2CLI traces each candidate entry point along a “trigger-handler-effect” path and keeps it only if the path has typed inputs, a well-defined state-changing boundary, and an independently observable effect. In short, the vocabulary only offers candidates,

Table 4: Framework-specific trigger vocabulary (representative constructs). Menu and keyboard-shortcut registrations are also included where the framework provides them.

Framework	Applications	Example Trigger Constructs
Qt / QML	Krita, OBS Studio, FreeCAD, MuseScore, Shotcut	`QAction`, `QObject::connect`, `QML onClicked`, `onTriggered`
wxWidgets	Audacity	`EVT_*`, `Bind(wxEVT_*)`
GTK	GIMP	`Gimp*ActionEntry`, `*_cmd_callback`
Godot Editor	Godot 4	`ED_SHORTCUT`, `signal connect`
UNO	LibreOffice - Writer, - Calc, - Impress	`.uno:*`, `commands`

and the subsequent semantic validation decides which of them are valid intent-level GUI functionalities.

D Implementation and Evaluation Details

D.1 Implementation Settings

The DCGBaseline, APP2CLI, and ablation studies all use `gpt-5.6-sol` through Codex CLI 0.153.4 with high reasoning effort and a 360-second limit per invocation. All experiments are conducted with five concurrent workers on a virtual machine with four vCPUs, 8 GB of memory, and 512 GB of storage, running Ubuntu 22.04 aarch64. We use CLI-Anything's existing commands for the baseline comparison.

D.2 Agent-Native Command Generation Procedures

Algorithm 1 takes a manually defined inventory of GUI functionalities, $\mathcal{F}$. During Phases I-II, the semantic specification can be completed or corrected via official documentation and source code in apps, while preserving the target GUI functionality's intended behavior and constraints. SelfJudge reviews the semantic specification and the CLI template for missing fields, unsupported implementations, and deviations from the intended behavior.

Here, u, q, and m denote the semantic specification, CLI template q_g, and candidate command c_g, respectively. The specification u records the parameters θ and intended behavior $\mathcal{R}_g = \langle P_g, E_g, N_g \rangle$ defined in Section 2. The symbol $\perp$ indicates that an artifact has not yet been produced. H stores only construction-time review results and execution diagnostics. A successful "self-review" or construction attempt does not guarantee behavioral correctness. All functionalities, including unresolved and failed cases, remain recorded in L. For each functionality, r counts joint reviews of the semantic specification and CLI template, while a counts command construction attempts. Both are limited to two and are never reset. After "self-review" succeeds, the semantic specification and CLI template remain fixed throughout Phase III.

In each construction attempt, the "Gener" Agent produces a candidate command and an implementation-constraint record, Constraints, which records the conditions the implementation must satisfy. The "Valid" Agent compares the implementation with the fixed semantic specification and Constraints, and builds and runs the candidate. After a failed attempt, the orchestrator returns the diagnostics only to the "Gener" Agent for command revision within the two-attempt limit. In Algorithm 1, $runOK$ indicates successful construction and execution of the candidate, while $specOK$ indicates that it passes the consistency validation against the semantic specification and implementation constraints.

D.3 Coverage Measurement and Overlap

All line sets below refer to one application version and source-code scope, with each line identified by its file path and line number. For each execution route $r \in \{G, C\}$, let E_r denote the executable

Algorithm 1: Agent-Native command generation procedures

Input: Functionality inventory $\mathcal{F}$; production GUI source $\mathcal{S}$; official documentation $\mathcal{D}$; public interfaces $\mathcal{A}$
Output: Commands M and per-functionality construction records L

$M, L \leftarrow \emptyset$
foreach *functionality* $g \in \mathcal{F}$ **do**
  $X \leftarrow (g, \mathcal{S}, \mathcal{D}, \mathcal{A})$ `// permitted task context`
  $u, q, m, \text{Constraints} \leftarrow \perp$; $H \leftarrow [\,]$
  $r, a \leftarrow 0$; *ready* $\leftarrow$ false
  status $\leftarrow$ `planning-unresolved`
  `/* Phases I-II: at most two reviews; no application calls */`
  while $r < 2$ and $\neg$*ready* **do**
    $r \leftarrow r + 1$
    $u \leftarrow \text{ExtractOrReviseSpecification}(X, u, H)$
    $q \leftarrow \text{ConstructOrReviseTemplate}(X, u, q, H)$
    $(\textit{ready}, d) \leftarrow \text{SelfJudge}(X, u, q)$
    Append review diagnostics d to H
  if *ready* **then**
    Fix u and q for all subsequent construction attempts
    status $\leftarrow$ `construction-failed`
    `/* Phase III: at most two attempts; command revision only */`
    while $a < 2$ **do**
      $a \leftarrow a + 1$
      $(m, \text{Constraints}) \leftarrow \text{Generate}(X, u, q, m, H)$
      $(\textit{runOK}, \textit{specOK}, d) \leftarrow \text{Valid}(X, m, u, \text{Constraints})$
      Append construction diagnostics d to H
      if $\textit{runOK} \wedge \textit{specOK}$ **then**
        Package and freeze m with its signature and Constraints
        $M[g] \leftarrow m$; *status* $\leftarrow$ `constructed`
        break
  $L[g] \leftarrow (\textit{status}, u, q, m, \text{Constraints}, r, a, H)$
return (M, L)

lines in the application source code and L_r the lines covered by that route. Its line coverage is calculated as follows:

$$\text{Coverage}(r) = \frac{|L_r \cap E_r|}{|E_r|}. \tag{3}$$

The routes of L_G and L_C may have different executable lines. Therefore, GUI-to-CLI coverage overlap measures the fraction of GUI-covered lines that are also covered by the CLI within the application source code:

$$\text{Overlap}_{G \to C} = \frac{|L_G \cap L_C|}{|L_G|}. \tag{4}$$

Overlap is undefined when $L_G = \varnothing$. Within each scope, covered lines are aggregated, with results reported separately for the application, functional modules, and individual functionality executions.

D.4 STOR Workflow

Figure 4 illustrates the STOR workflow described in Section 3.4.

E Additional Explanation and Experiments Results

E.1 Case Studies of Component Analysis

Semantic specification: Function Wizard formula commit. The source test `tdf132173` enters `FIND({"A";"B";"C"},"SAMPLE TEXT")` through the Function Wizard and commits

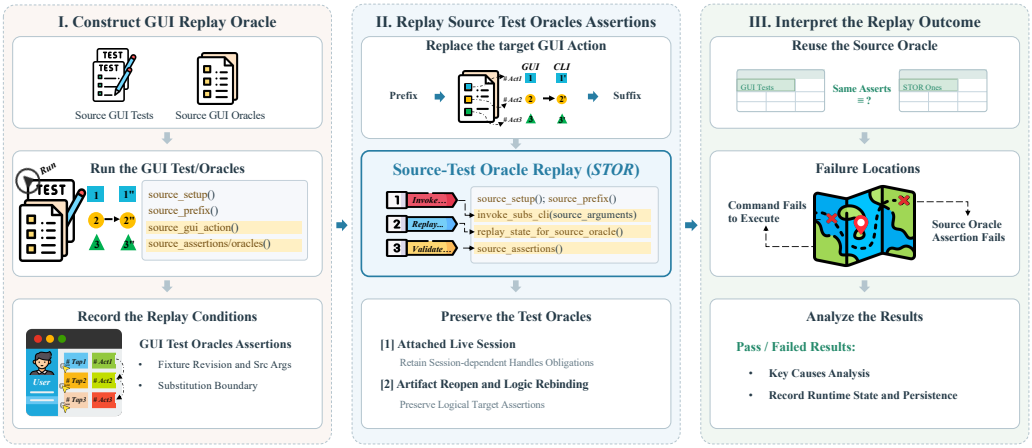


Figure 4: The Validation of LibreOffice by STOR.

the dialog. The structured input retains both formula entry and commit. Full routes the intent to `calc.formula.direct_commit` with `commit_formula`. Its GUI and CLI observations match before and after execution, and the resulting cell value is `2`.

The output without the semantic specification instead creates a new-document formula setter that calls `cell.setFormula(formula)` and then `document.calculateAll()`. It exits with code zero and creates a valid ODS, but the independent cell observer reads `Err:508` rather than `2`. The issue is not `setFormula()` itself, which Full also calls. Before that call, Full converts GUI separators to UNO syntax while preserving array contents and quoted strings. The inspected implementation performs quote-aware and brace-depth-aware conversion. This case shows how an explicit semantic specification helps preserve the source functionality's required behavior when choosing its realization, and why process success alone does not establish semantic success.

Skill-guided template construction: FreeCAD object deletion. The source intent in `TestGuiObserver` specifies deletion of an object from the active document. The input without skill-guided templates retains `document.object.delete` and the document/object bindings, and its script contains `document.removeObject(args.object_name)`. However, it passes `--document` and `--object-name` directly to `FreeCADCmd`. The executable rejects `--document` as an unknown option and exits with code one before reaching the deletion code. Full uses its qualified request/connector route and carries a certificate scoped to `live_observer_equivalent`. This is an application-specific invocation failure. It does not show that the remaining ablation script would succeed after correcting the flags.

E.2 Generation effectiveness of APP2CLI.

Table 5 reports generation effectiveness at two granularities. Intent-level success measures command generation for each target GUI functionality independently, whereas case-level success requires all target functionalities extracted from the same source/test unit to succeed. The denominators describe the selected input scope of each application unit, and source-unit boundaries do not imply a uniform number of functionalities. For example, a Shotcut case contains 43 targets, while 2 GIMP cases contain 5.

Overall, APP2CLI achieves a command-generation success rate of 60.05% (881 of 1,467 target functionalities) and supports all target functionalities in 517 of 1,003 cases (51.55%). The difference between these rates shows that many failures are localized to particular targets rather than invalidating every functionality in their source case. For example, Calc supports 247 targets across 124 fully supported units, and Writer supports 177 targets and 90 complete cases. Performance also varies across 90 fully supported units. Success also varies by application: GIMP and Shotcut achieve successful command generation for all evaluated targets, whereas Audacity and Krita remain below 30%.

Table 5: Generation Effectiveness of APP2CLI by Application Unit.

Application unit	Case-Level Success	Intent-Level Success
LibreOffice Calc	124/330 (37.58%)	247/535 (46.17%)
LibreOffice Impress	13/24 (54.17%)	29/42 (69.05%)
LibreOffice Writer	90/117 (76.92%)	177/204 (86.76%)
Audacity	23/78 (29.49%)	23/78 (29.49%)
FreeCAD	11/13 (84.62%)	19/21 (90.48%)
GIMP	2/2 (100.00%)	5/5 (100.00%)
Godot 4	191/316 (60.44%)	229/369 (62.06%)
Krita	10/36 (27.78%)	10/36 (27.78%)
MuseScore	33/46 (71.74%)	79/93 (84.95%)
OBS Studio	19/40 (47.50%)	20/41 (48.78%)
Shotcut	1/1 (100.00%)	43/43 (100.00%)
Overall	**517/1,003 (51.55%)**	**881/1,467 (60.05%)**

E.3 GUI AND CLI SOURCE-LINE COVERAGE

Table 6 provides the per-application coverage and overlap results discussed in RQ1.

Table 6: GUI and CLI Source-Line Coverage (%).

Application Unit	Application Scope			Semantic-Core Scope		
	GUI Cov.	CLI Cov.	Overlap	GUI Cov.	CLI Cov.	Overlap
LibreOffice (All)	24.99	21.16	76.68	64.20	66.70	96.22
Calc	23.86	19.99	75.45	61.52	67.13	97.12
Impress	16.36	14.62	84.20	74.78	75.22	95.86
Writer	23.20	19.93	72.52	67.22	64.73	94.87
Audacity	17.36	17.93	98.74	89.96	89.96	100.00
FreeCAD	12.29	11.57	71.34	78.11	81.13	97.10
GIMP	18.50	12.44	43.60	N/A	N/A	N/A
Godot 4	34.35	23.74	68.18	N/A	N/A	N/A
Krita	27.37	27.80	97.37	94.12	90.20	95.83
MuseScore	17.27	12.77	70.55	32.52	32.52	100.00
OBS Studio	26.46	25.36	89.86	87.62	86.67	98.91
Shotcut	15.25	7.50	21.35	N/A	N/A	N/A
Average	**21.12**	**17.60**	**72.11**	**74.42**	**74.53**	**98.01**

E.4 THE RESULTS OF STOR AND GENERATION COSTS

Tables 7 and 8 provide the results of STOR and generation costs discussed in RQ1 and RQ3.

Table 7: The Results of STOR.

Application Unit	Eligible Targets	Replay Pass	Pass Rate
Writer	20	17	85.00%
Impress	8	7	87.50%
Calc	96	90	93.75%
Overall	124	114	91.94%

Table 8: Total Generation Costs.

Method	Time (hours)	Total Tokens
DCGBASELINE	1.29	2,793,856
APP2CLI	11.41	76,258,286
w/o semantic specification	11.75	41,966,979
w/o skill guidance	4.90	5,989,941