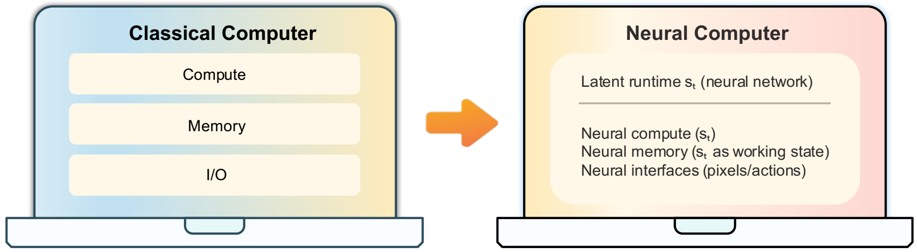
NEURAL COMPUTERS

Anonymous authors
Paper under double-blind review



From modular hardware stack to neural latent stack

ABSTRACT

We train neural networks (NNs) to learn the dynamics of traditional computers (TCs). The NN's inputs include screenshots, instructions (natural language or program commands), and user actions such as keyboard typing and mouse movements/clicks. Without relying on internal program state, we investigate whether a unified neural system can learn, simulate, and operate a computer interface directly from raw I/O data. We present initial steps towards a completely neural computer (CNC): in our first experiments, our NNs learn to emulate TCs in both CLI and GUI settings. We derive practical design considerations for building more and more general CNCs.

1 INTRODUCTION

Can a single set of weights act as the future "computer"? Recent advances in world models (Ha & Schmidhuber, 2018; OpenAI, 2024; Google DeepMind, 2025) suggest that NNs can internalize rich environment dynamics.

High-capacity video generators such as Veo 3.1 (Google, 2025b) and Sora 2 (OpenAI, 2025) translate conditioning signals into coherent, photorealistic frame sequences. Interactive generators such as Genie 3 (Google, 2025a) extend this to action-conditioned rollouts. In parallel, LLM-driven UI runtimes such as Imagine with Claude[1] map natural language inputs to structured interface updates. Yet these capabilities are typically split across components: models render or predict, while an external runtime (simulator/OS/DOM) carries executable state and mediates control. To fill this void, we define a neural computer (NC) as a learned, executable, and ultimately programmable system in which model, environment, and runtime share a unified latent state. An NC exposes a computer-like interface through pixels, actions, and language.

In current version, we instantiate NCs as video models that renders frames while carrying task context. We study two interfaces. $\mathrm{NC}_{\mathrm{CLIGen}}$ is a terminal interface conditioned on a text (natural language or command lines) and first frame (Section 3.1). $\mathrm{NC}_{\mathrm{GUIWorld}}$ is a desktop interface conditioned on recent pixels and synchronized mouse/keyboard actions (Section 3.2). Teaser contrasts the classical compute/memory/I/O stack (left) with the NC formulation (right), where a single latent runtime s_t jointly (i) models environment dynamics, (ii) acts as working memory, and (iii) renders and controls the interface via pixels and actions. Figure 1 illustrates the design motivation across two interface-specific NCs.

In $\mathrm{NC}_{\mathrm{CLIGen}}$ experiments, the NC can render and execute basic command-line workflows. It often stays aligned with the terminal buffer and captures common "physics" of everyday CLI use (e.g.,

[1]https://claude.ai/imagine/

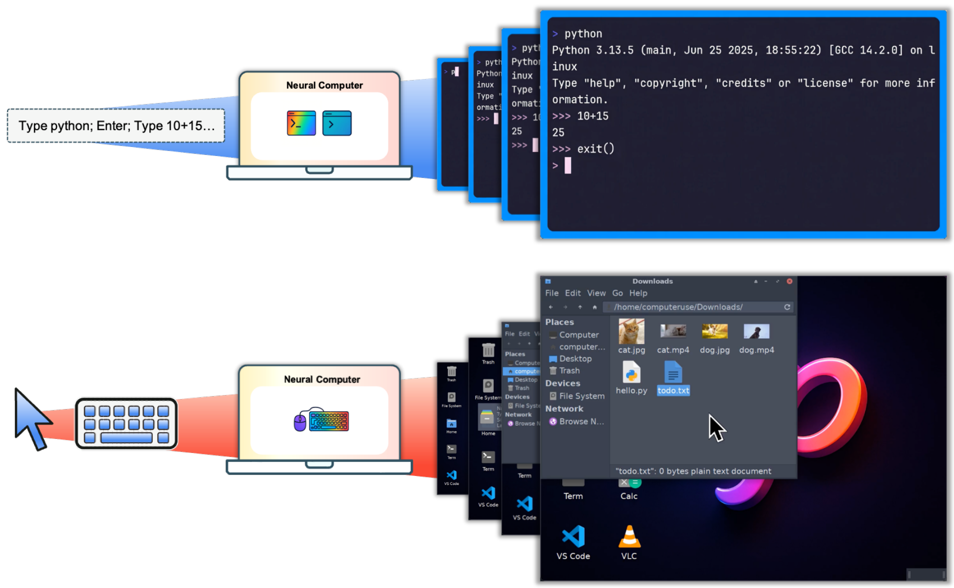


Figure 1: **Neural computers across interfaces.** Given a prompt or action stream, an NC rolls out future interface frames for NC$_{\text{CLIGen}}$ (top) and NC$_{\text{GUIWorld}}$ (bottom). NC$_{\text{CLIGen}}$ and NC$_{\text{GUIWorld}}$ are the corresponding models trained on those datasets.

fast scrollback, prompt wrapping, window resizing). Trained on carefully scripted data, generations are visually and structurally close to real sessions. NCs can execute short command chains and render coherent outputs. While symbolic reasoning remains challenging, we show that improved conditioning can dramatically raise arithmetic-probe accuracy (up to 83%; Figure 6).

In NC$_{\text{GUIWorld}}$ experiments, we evaluate practical world-model designs across data quality, action injection, and action encoding in this domain. Qualitatively, the model learns coherent pointer dynamics and short-horizon action responses (e.g., hover/click feedback and window/menu transitions). With explicit cursor supervision, it reaches 98.7% cursor accuracy (Table 6).

Overall, our results suggest that current NCs already instantiate several classical interface primitives—most notably accurate I/O alignment and reliable short-horizon control, across both CLI and GUI settings. Beyond visual alignments, we uncover many additional behaviors; two interesting examples are: (1) symbolic reasoning is often bottlenecked by the conditioning interface, but can be significantly improved with clearer, stronger conditions; (2) with explicit visual supervision, cursor tracking approaches perfection, suggesting that precise low-level control is within reach. However, fine-grained text/icon rendering and long-horizon layout consistency remain brittle, pointing to missing primitives for robust computation, memory, and runtime invariance. We therefore define the long-term target, the Completely Neural Computer (CNC), as a fully learned computer whose compute, memory, and interfaces are unified within a single runtime rather than engineered as separate modules. Appendix B summarizes a capability map and an illustrative roadmap and more insights. **Concretely, this work makes the following contributions:**

- Define neural computers (NCs) and build video-based prototypes for both CLI & GUI interfaces.
- Provide a data engine and alignment recipe that synchronize text, actions, and frames for CLI & GUI environments, from which we collect more than 3000 hours of training data.
- We identify key design choices for NCs through extensive ablation studies (spending more than 200k H100 hours).
- Outline a roadmap toward completely neural computers (CNCs) grounded in classical computer primitives and open research questions.

2 Preliminaries

Classical (von Neumann-style) computers separate computation, memory, and I/O, typically realized as a layered operating-system stack. Our motivating question is whether a single set of weights

Figure 2: **Data types for learning NC behaviors.** Logos denote datasets: CLIGen (General) replays public `asciinema` traces from diverse terminal workflows. CLIGen (Clean) uses scripted `vhs` runs for deterministic traces in controlled experiments. GUIWorld records desktop RGB with mouse/keyboard traces to validate action-conditioned rendering and control on GUIs.

can internalize these roles inside one stateful latent runtime, rather than relying on an external OS/simulator to carry executable state. We model a video-based neural computer (NC) as a learned latent-state system that folds these roles into an update-and-render loop.

In current designs of this paper, an NC is an autoregressive vidoe model or world model (Ha & Schmidhuber, 2018) of computer systems that generates observables conditioned on given context. Specifically, an NC accepts an observable and a user action as input to predict the subsequent observable. We treat screen frames as observables and define actions as text prompts or control trajectories. More broadly, the NC framework can accommodate various other modalities and structural representations for both observables and actions. Given an initial screen frame x_0 and conditioned on user action u_t at iteration t, an NC updates its memory state and samples the next frame x_{t+1}. Formally, an NC defined by an initial memory state s_0, an update function F_θ, and a decoder G_θ operates as follows:

$$s_t = F_\theta(s_{t-1}, x_t, u_t), \qquad x_{t+1} \sim G_\theta(s_t). \tag{1}$$

In the context of world modeling, the sequence of actions is referred to as a conditioning stream.

Under this formulation, each step collapses multiple subsystems into one learned update. The state s_t stores all relevant context. F_θ integrates new observations and actions, and G_θ renders the next frame. Auxiliary heads can encode and decode prompts, buffers, or action traces. This moves some functionality that would traditionally live in OS queues, device drivers, and UI toolkits into the latent runtime.

2.1 Related Work

Differentiable world models (Schmidhuber, 1990; 2015) sought to learn representations of environment dynamics. In parallel, neuromorphic designs (Mead & Ismail, 2012) explored neural computation as a substrate. Differentiable memory and program-execution architectures, including Neural Turing Machines (Graves et al., 2014), Differentiable Neural Computers (Graves et al., 2016), and Neural Programmer-Interpreters (Reed & De Freitas, 2015), showed that neural controllers with memory can execute structured procedures. Latent video and world models (Ha & Schmidhuber, 2018; Hafner et al., 2019b;a; Bruce et al., 2024) extend these ideas to embodied control and interactive environments. Genie 3 (Google, 2025a), in particular, frames such models as agent-training substrates with improved physical consistency. High-capacity generators such as Veo 3 (Google, 2025b) and Sora 2 (OpenAI, 2025) emphasize open-ended, photorealistic simulation. Systems such as NeuralOS (Rivard et al., 2025) and Imagine with Claude (Anthropic, 2025) bring world-model

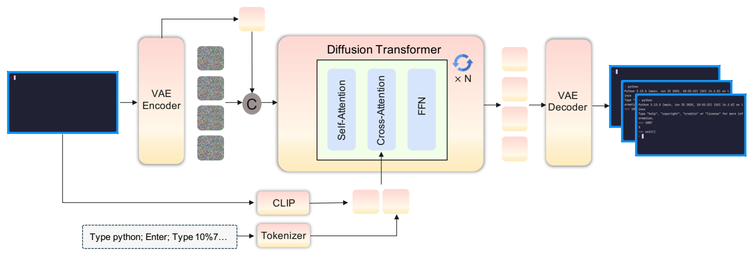


Figure 3: / NC$_{\text{CLIGen}}$ architecture. Terminal frames are observations x_t. A prompt and the first frame seed the conditioning stream. The Wan2.1-based latent state z_t rolls forward under the standard I2V sampling scheme.

conditioning to desktop and DOM-style interfaces. In this work, we study two NC instantiations for CLI and GUI with interface-specific conditioning, and a staged roadmap toward CNCs.

3 Implementation of Neural Computers

We build on the Wan2.1 model (Wan et al., 2025), which was a SoTA video generation model at the time of our experiments. We add NC-specific conditioning and action modules, together with interface-specific training recipes for our specific settings. Figure 1 illustrates the setups. NCs take a prompt or action stream as input and generate future frames for both NC$_{\text{CLIGen}}$ and NC$_{\text{GUIWorld}}$.

We treat the video model's time-indexed latent representations as the state z_t in Equation (1). This abstracts the underlying diffusion transformer as a latent dynamical system. The transformer F_θ maps prior context and current observations/conditioning into an updated state. The decoder G_θ predicts the distribution of the next frame $\hat{x}_{t+1}$. Auxiliary heads encode and decode conditioning streams u_t, including text prompts, terminal buffers, and action traces.

3.1 / The CLI Video Generators

Our CLI video generators instantiate the NC in terminal settings. Observations x_t are terminal frames rendered from the underlying text buffer. The conditioning stream u_t carries a user prompt and optional metadata, and the latent state z_t tracks CLI context across frames. At inference time, the model rolls out from the prompt and first frame, updates z_t, and predicts future terminal frames (Figure 3). Throughout this section, CLIGen (General) and CLIGen (Clean) refer to datasets. We denote the corresponding models trained on each dataset as NC$_{\text{CLIGen}}$ and NC$_{\text{CLIGen}}$.

3.1.1 Data Pipeline

We collect CLI data under two complementary regimes (details in Section E.1). CLIGen (General) replays public `asciinema` traces to cover diverse real-world terminal workflows, while CLIGen (Clean) uses scripted `vhs` runs in Dockerized environments to produce deterministic traces for controlled experiments (including math).

3.1.2 Model Architecture

We treat CLI generation as text-and-image-to-video: a caption and the first terminal frame condition the rollout. The first frame is encoded by a VAE into a conditioning latent. In parallel, a CLIP image encoder (Radford et al., 2021) extracts visual features from the same frame, and a text encoder (e.g., T5 (Raffel et al., 2020)) embeds the caption. Following the Wan2.1 image-to-video (I2V) design, these conditioning features are concatenated with diffusion noise, projected through a zero-initialized linear layer, and processed by a DiT stack. Decoupled cross-attention injects the joint caption and first-frame context derived from the CLIP and text features. The VAE encodes and decodes terminal frames. During generation, the diffusion transformer advances the latent state z_t under the original Wan2.1 I2V sampling schedule, without additional binary masks or periodic reseeding. Training details are provided in Appendix D.

Table 1: Caption styles versus generation fidelity.

Prompt style	PSNR	SSIM	Avg. words
Semantic	21.90	0.813	55
Regular	23.63	0.843	52
Detailed	**26.89**	**0.867**	**76**

3.1.3 Evaluations

We evaluate (i) optical fidelity (PSNR/SSIM/LPIPS) and (ii) trace-level interaction correctness (action→state consistency, command success, buffer/text consistency), *independent of OCR*. Beyond just visual fidelity, non-optical metrics rigorously validate the robustness, accuracy, and consistency of multi-step workflows, clearly demonstrating that our models not only replicate visuals but also handle complex, real-world tasks with high precision.

Experiment 1: Generic VAEs handle terminal content with proper font sizing

Concurrent work (Rivard et al., 2025) argues that generic natural-image VAEs perform poorly on structured computer screenshots. We evaluate this claim directly by applying the Wan2.1 VAE (Wan et al., 2025) to terminal content. In our setting, reconstruction quality is primarily governed by font size. At 13 px fonts it is high (40.77 dB PSNR, 0.989 SSIM). At 6 px fonts, text exhibits noticeable blurring.

However, a sweep over CLIGen (General) frames shows that this effect is confined to extreme cases (Figure 4). Very small 6 px fonts and ultra-dense text exhibit localized blurring despite high global PSNR. In contrast, the 13 px terminal font used in CLIGen remains visually sharp across panes and commands. These results indicate that the VAE is adequate for regular CLIGen usage and highlight that sensible font choices help ensure stable, legible NC training.

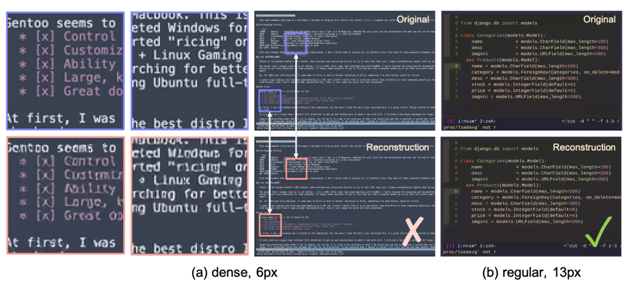


Figure 4: Wan2.1 VAE reconstructions on CLIGen (General).

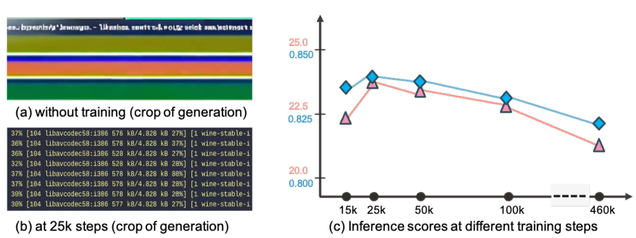


Figure 5: CLIGen (General) training plateaus at 25k steps.

Experiment 2: The impact of data quality

In CLIGen (general) experiments, data quality matters more than training duration, this is different from natural image generation can continue to benefit from scale and prolonged training. After early gains, remaining errors are dominated by artifact-prone supervision (e.g., rendering glitches or rapid screen changes that break stable temporal alignment).

A visual comparison in panels (a–b) highlights the effect of training on CLIGen data. Without CLIGen data, Wan2.1 produces largely garbled terminal outputs (panel a). After 25k training steps, the model generates readable text with consistent formatting and color codes (panel b).

Table 2: OCR accuracy versus training.

Steps (k)	Char. acc.	Exact line
0	0.03	0.01
10	0.18 $\uparrow_{0.15}$	0.05 $\uparrow_{0.04}$
20	0.33 $\uparrow_{0.30}$	0.12 $\uparrow_{0.11}$
30	0.41 $\uparrow_{0.38}$	0.18 $\uparrow_{0.17}$
40	0.52 $\uparrow_{0.49}$	0.26 $\uparrow_{0.25}$
50	0.52 $\uparrow_{0.49}$	0.27 $\uparrow_{0.26}$
60	**0.54** $\uparrow_{0.51}$	**0.31** $\uparrow_{0.30}$

Table 3: Closed-book arithmetic-probe accuracy.

Model	Accuracy
Wan2.1	0%
$\mathrm{NC}_{\mathrm{CLIGen}}$	4%
Veo3.1	2%
Sora2	**71%**

Figure 5 plots the corresponding PSNR/SSIM training curves and reveals a clear efficiency ceiling. These reconstruction metrics peak at approximately 25k steps. They do not improve with further training up to 460k steps. Extended training can even slightly degrade performance. We hypothesize that terminal rendering reaches a natural learning limit. Most structured patterns are acquired early, and further gains would require higher-quality, better-paced, or more informative data.

Experiment 3: Literal captions boost generations

Caption specificity strongly affects terminal rendering quality. We compare 3 caption tiers (semantic/regular/detailed) and find detailed, literal descriptions improve reconstruction fidelity (PSNR 21.90→26.89 dB; Table 1). We attribute this to terminals dominated by exact text placement, where literal captions constrain token text-to-pixel alignment.

Experiment 4: Neural computers achieve accurate character-level text generation

PSNR and SSIM capture perceptual similarity, but character-level accuracy is more direct for terminal rendering. It requires pixel-to-text correspondence. For CLIGen (Clean), we apply Tesseract OCR to sampled frames (with whitespace normalization). We report character accuracy (Levenshtein distance) and exact-line match rate (full protocol in Appendix E).

Table 2 shows that our models achieve strong text rendering accuracy. Character accuracy increases from 0.03 at initialization to 0.54 at 60k steps. Exact-line matches reach 0.31 (0.26 at 40k). Most gains occur within the first 40k steps, with smaller improvements afterward. These OCR-based results go beyond perceptual similarity. Accurately generating terminal characters requires modeling text structure, font rendering, and spatial relationships. These are core competencies for interactive neural systems. This level of precision brings us closer to usable terminal interfaces, not just plausible ones.

Experiment 5: Closed-book arithmetic requires improvement for better reasoning

Beyond rendering quality, we probe symbolic computation using CLI arithmetic tasks. In the closed-book setting, the conditioning text specifies the expression but not the solution. This probe is intentionally strict: the model must preserve REPL structure, keep the typed expression intact, and render the correct numeric output. We reserve a held-out pool of 1,000 math problems and randomly sample 100 for evaluation. Table 3 shows that most current video models struggle on these tasks. Wan2.1 achieves 0% accuracy, our $\mathrm{NC}_{\mathrm{CLIGen}}$ model reaches 4%, and Veo3.1 manages 2%, all far below human-level performance. Sora2's 71% accuracy is a notable outlier and may reflect system-level advantages or additional training beyond standard video generation. Overall, native symbolic computation remains an open challenge for pixel-grounded NCs.

The poor arithmetic-probe performance in Table 3 raises a practical question. Do such gains require specialized reinforcement learning, or can system-level conditioning substantially narrow this gap? Sora2's 71% accuracy suggests several possible explanations, which we examine in Table 4. Our exlorations (in Experiment 6) yield an interesting result. Better conditioning alone can dramatically

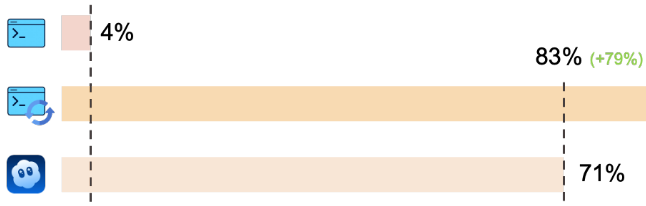


Figure 6: Reprompting boosts performance to 83%.

improve arithmetic-probe accuracy, without introducing reinforcement learning or modifying the video generator.

Experiment 6: Reprompting helps improve arithmetic-probe scores

Table 4: Why Sora2 leads (our hypotheses)?

Factor	Implication
1 Stronger base + similar data	Higher intrinsic arithmetic; symbolic capability may be baked in
2 Additional RL training	Reward shaping teaches math beyond diffusion; could transfer to CLI
3 System-level reprompt/recaption	LLM computes answers; strong conditioning drives generation

As shown in Figure 6, $\mathrm{NC}_{\mathrm{CLIGen}}$ accuracy on arithmetic tasks rises from 4% to 83% under reprompting, suggesting that system-level conditioning is an effective first step for improving performance on symbolic probes. This is complementary to (not strictly requiring) RL-based pipelines.

Reprompting highlights the sensitivity of symbolic-probe outcomes to the conditioning interface. Much of the ”reasoning” gain comes from better specification and instruction-following, not new computation. In the reprompted setting, the conditioning text provides the solution string (examples in Appendix A). To ensure faithful transcription, we include answer-augmented captions for half of the arithmetic training episodes. This demonstrates that the model can consistently render conditioned symbolic content. Our experiments do not support the view that current SoTA (open-sourced or closed sourced) video models already have strong reasoning skills. Simple tasks like two-digit addition are often unsolved correctly, while small LLMs (e.g., 100MB LLMs) easily handle them. However, after reprompting with an LLM-provided answer, accuracy improves, suggesting video models have potential but are not yet strong reasoning systems. This suggests that while video models have potential, they are not yet strong reasoning systems. We see this as similar to the development of LLMs in solving simple questions like *“How many ’r’s are in ’strawberry’ ”* (which humans can answer instantly, but LLMs easily failed). We believe these limitations are due to the current development stage, and expect stronger reasoning abilities in future NCs.

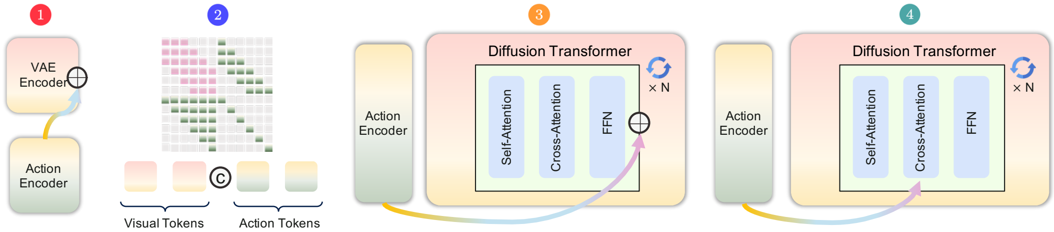
The evidence supports system-level conditioning as a practical path forward. Among hypotheses for improving arithmetic-probe performance, stronger base models, reinforcement learning, or enhanced conditioning—our results indicate that the latter is a convenient path. The gain from reprompting (4%→83%), achieved without modifying the video generator, shows that “reasoning” in these probes is highly sensitive to conditioning. Instead of complex reinforcement learning pipelines, large symbolic-probe gains can be achieved through strategic conditioning alone. Evaluations should distinguish between native computation and conditioning-assisted performance when assessing video models’ reasoning capabilities.

3.1.4 VISUALIZATIONS

Qualitative visualization pages are moved to Appendix A. See Appendix A for CLIGen (General/Clean) rollouts, REPL traces, and math-probe visualizations.

3.2 THE GUI WORLD MODELS

We extend the neural computer framework to interactive desktop environments with $\mathrm{NC}_{\mathrm{GUIWorld}}$. Here, fine-grained action control is essential: GUI interaction requires precise cursor tracking, timely click feedback, and robustness to rapidly changing interface states.



Mode	Injection point	Notes
1 `external`	Conditioned VAE input	External conditioning stream modulates the backbone without sharing the main token sequence.
2 `contextual`	Concat frames, actions	Video and action tokens share one sequence with a lag-aware temporal mask for mid-level fusion.
3 `residual`	Hidden state with action delta	Residual modulation branches provide a strong but more indirect baseline.
4 `internal`	After cross-attn, before FFN	Dedicated cross-attention inside transformer blocks fuses actions near the backbone core.

Figure 7: **Four modes for injecting GUI actions into the diffusion transformer.** 1 modulates VAE latents before the transformer; 2 appends action tokens; 3 applies residual modulation; and 4 inserts action cross-attention within blocks.

3.2.1 Data Pipeline

The system records synchronized RGB video frames (x_t) together with detailed mouse and keyboard event streams (u_t). This preserves pixel-level alignment between user actions and visual responses. Data collection and preprocessing details for GUIWorld are provided in Appendix E.

3.2.2 Model Architecture

The GUIWorld architecture builds on the Wan2.1 (Wan et al., 2025) by incorporating explicit action-conditioning modules. The central challenge is to align time-stamped user actions with generated frames and inject this information at the appropriate depth within the transformer. Action features are precomputed from frame-aligned mouse and keyboard signals (Appendix E.4). We aggregate them into latent-aligned embeddings that summarize recent action history at each diffusion step. We evaluate two action encoders. The *raw-action encoder* (v1) preserves fine-grained mouse/keyboard event streams. The *meta-action encoder* (v2) abstracts interactions into coarse API-style categories (clicks, drags, scrolls, typing, shortcuts). Both encoders use the same temporal alignment and are evaluated as separate ablations. In our experiments, their effects on rendering fidelity and control behavior are comparable.[2]

We inject action embeddings into the diffusion backbone in four ways (Figure 7). We study `external`, `contextual`, `residual`, and `internal` conditioning. For the injection-scheme ablation, all four modes share the same meta-action encoder and temporal alignment. They differ only in where the latent action features interact with the video latents and transformer blocks. We compare raw-action vs. meta-action encoders separately in Table 8 (with additional details in Appendix G).

external conditioning. In the `external` mode, action information modulates the latent video sequence before the diffusion transformer. Action features are applied as a pre-conditioning step at the model input, without introducing explicit action tokens or cross-attention inside the diffusion backbone. As a result, action information enters only through the modified input latents; the diffusion backbone never attends directly to action tokens, so any action signal must be carried implicitly in $z'_{1:T}$.

Concretely, an external action module produces a residual update to the VAE latents, yielding modified latents $z'_{1:T}$ that are then processed by the unchanged diffusion transformer (formal definitions in Appendix G.2).

contextual conditioning. In the `contextual` mode, actions are represented as additional tokens and integrated directly into the transformer's self-attention. Similar token-based action representations

[2]Table 14 summarizes the representational differences. Training recipes are in Appendix D

Table 5: Overall performance across data sources.

Split	FVD_{all}	$SSIM_{all}$	$LPIPS_{all}$
baseline	149.61	0.496	0.605
train with random fast	48.17	0.695	0.483
train with random slow	20.37	0.830	0.237
train with Claude CUA	14.72	0.885	0.144

Table 6: Cursor conditioning losses versus accuracy.

Loss variant	Cursor accuracy
Position (x, y) only	8.7%
Position (x, y) + Fourier	13.5%
Position (x, y) + SVG mask/ref	**98.7%**

have been explored in prior world models, including Gato (Reed et al., 2022) and World and Human Action Models (Kanervisto et al., 2025).

The meta-action encoder produces latent-aligned action tokens $A \in \mathbb{R}^{L_a \times D}$. We concatenate them with visual tokens $V \in \mathbb{R}^{L_v \times D}$ to form a joint sequence $[V; A]$. Each transformer block applies self-attention over this combined sequence using a structured temporal mask (Appendix Figure 38). The mask enforces causal alignment: each frame token attends only to actions within a short past window, and each action token attends only to frames after a fixed temporal lag. Through this masked joint attention, `contextual` conditioning fuses action and visual information within the transformer blocks.

residual conditioning. In the `residual` mode, the transformer block structure remains unchanged. A lightweight action module attaches to a subset of layers as a residual branch. This follows the `residual` conditioning paradigm from ControlNet (Zhang et al., 2023), while remaining modular and additive to the base diffusion backbone.

At each selected layer l, the transformer applies its standard sequence of self-attention, text or reference cross-attention, and feed-forward operations to produce hidden states $h^{(l)}$. A separate action module then takes $h^{(l)}$ together with a local temporal window of latent action features and mouse trajectories. It outputs a residual update $\Delta h^{(l)}(a, \text{mouse})$. This update is added to the block hidden states and passed to the subsequent transformer block (formal definitions in Appendix G.2). In this formulation, `residual` conditioning injects action information through block-external residual branches. It does not modify the internal computations of the transformer blocks themselves.

internal conditioning. In the `internal` mode, action conditioning is incorporated within the transformer blocks. Related multi-stream world models have explored similar designs, including Matrix-Game-2 (He et al., 2025). Each selected block augments the standard attention stack with an additional action cross-attention sub-layer. Specifically, the block applies self-attention, followed by cross-attention over text and reference features, and then a dedicated action cross-attention layer. Keys and values are derived from latent action features (and, optionally, mouse inputs).

Given block input h, text or reference context c, and action latents a, the internal block computes an update that inserts an additional action cross-attention term inside the block (formal definitions in Appendix G.2). As illustrated in Figure 7, action features are injected directly into the block's cross-attention stage.

In contrast to `residual` conditioning, `internal` conditioning integrates action information through a block-internal attention mechanism rather than an external residual branch. This design mirrors the multi-stream injection strategy used in Matrix-Game-2 (He et al., 2025) and provides the strongest control for fine-grained GUI interaction. In this setting, precise temporal alignment and spatial locality are critical. Each conditioning mode (`external`, `contextual`, `residual`, and `internal`) is trained as a separate ablation, and no combinations are used.

3.2.3 Evaluation setup

We use the FVD/LPIPS/SSIM suite as core metrics, adding action-driven metrics that focus on post-interaction frames after clicks, scrolls, and key/type events. For example, we compute SSIM/LPIPS averaged over the k frames after each logged action, and action-driven FVD on

Table 7: Action-driven metrics (15 frames post-action) across 4 schemes. † `baseline₂`: `external`, early-stopped at ∼50% steps.

Mode	$\text{SSIM}_{+15}\uparrow$	$\text{LPIPS}_{+15}\downarrow$	$\text{FVD}_{+15}\downarrow$
`baseline₁` (untrained)	0.326	0.649	184.3
†`baseline₂`	0.746	0.251	33.4
`contextual`	0.813	0.190	24.8
`residual`	0.857	**0.138**	18.8
`internal`	**0.863**	0.141	**14.5**

Table 8: Raw-action vs. API-like action encoding under the same injection mode (15 frames after action).

Mode	Encoding	$\text{SSIM}_{+15}\uparrow$	$\text{LPIPS}_{+15}\downarrow$	$\text{FVD}_{+15}\downarrow$
`internal`	raw-action (event-stream)	0.847	0.144	16.6
`internal`	meta-action (API-like)	**0.863**	**0.141**	**14.5**

post-action clips. Ablations vary conditioning design and action encoding to measure how these choices affect perceptual quality and responsiveness against ground-truth interfaces. Full metrics and implementations are in Appendix E.6.

Experiment 7: Data quality matters

Interactive GUI modeling shows that data quality matters more than dataset size for action-driven performance. We compare slow exploration, fast interaction, and supervised trajectories under `contextual` conditioning. This isolates which behaviors best support neural computer training.

Despite approximately 1,400 hours of random exploration across the slow and fast settings, these datasets are noisy. They are comparatively sample-inefficient for learning stable action–response mappings. They substantially improve global perceptual metrics over a baseline (Table 5). However, high-frequency cursor jitter and irregular, non-goal-directed action bursts make consistent control difficult under dense, stochastic input streams.

In contrast, the substantially smaller high-quality dataset (110 hours from Claude CUA) yields markedly stronger performance across all metrics. Goal-directed trajectories provide clearer action semantics and more predictable state transitions. This enables robust action conditioning even with limited data volume. These results indicate that neural computer development should prioritize curated, purposeful interactions over large-scale passive data collection.

Experiment 8: Precise cursor control requires explicit visual supervision

Figure 8: **Cursor references in GUIWorld. Left:** original desktop frames. **Middle:** binary cursor masks. **Right:** cursor-only reference frames rendered over a neutral background.

We examine whether the NC internalizes cursor dynamics from logged coordinates. Details of coordinate normalization and trajectory encoding are provided in Appendix E.4.

However, Table 6 shows that coordinate-based supervision remains insufficient for precise interaction. Position-only supervision achieves 8.7% accuracy, and even enhanced position features reach only 13.5%. This suggests that richer coordinate encodings alone do not resolve cursor drift and jitter. Motivated by the need for precise cursor placement, we add explicit visual cursor supervision by rendering a cursor-only foreground/mask reference stream and providing it as an additional conditioning signal during training (Appendix G.4). Under this explicit visual conditioning, cursor accuracy improves to 98.7%. This suggests that neural computers benefit from learning the cursor state as a visual object rather than relying solely on abstract coordinates. Explicit pixel-level supervision helps model cursor acceleration, hover states, and click feedback, which are essential for reliable GUI interaction.

Experiment 9: Different Action injections

Holding data and the action encoder fixed, deeper injection improves post-action fidelity (Table 7): `contextual`/`residual`/`internal` outperform `external` and the untrained baseline. `internal` yields the best SSIM/FVD and `residual` yields the best LPIPS; details of injection schemes are in Appendix G.

Experiment 10: Are action encodings influenced?

Under a fixed injection mode, raw-action and meta-action encodings perform similarly (Table 8); we use meta-action encoder and provide additional analysis in Appendix G.

4 CONCLUSION

Neural Computers learn interface-level behavior from raw CLI and GUI interaction traces, producing models that render coherent screen updates and support multi-step interactions. We provide practical guidance on data, conditioning, and evaluation to improve fidelity and control; a broader CNC roadmap is in Appendix B.

REFERENCES

Anthropic. Introducing claude sonnet 4.5. https://www.anthropic.com/news/claude-sonnet-4-5, September 2025.

Jake Bruce, Michael D Dennis, Ashley Edwards, Jack Parker-Holder, Yuge Shi, Edward Hughes, Matthew Lai, Aditi Mavalankar, Richie Steigerwald, Chris Apps, et al. Genie: Generative interactive environments. In *Forty-first International Conference on Machine Learning*, 2024.

Diego Calanzone, Stefano Teso, and Antonio Vergari. Logically consistent language models via neuro-symbolic integration. In *The Thirteenth International Conference on Learning Representations*, 2025.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

Google. Genie 3: A new frontier for world models, 2025a. URL https://deepmind.google/blog/genie-3-a-new-frontier-for-world-models/. Published August 5, 2025.

Google. Break the silence with veo 3.1. https://gemini.google/overview/video-generation/, November 2025b.

Google DeepMind. Veo. https://deepmind.google/models/veo/, May 2025.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626): 471–476, 2016.

David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2018.

Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*, 2019a.

Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. In *International conference on machine learning*, pp. 2555–2565. PMLR, 2019b.

Xianglong He, Chunli Peng, Zexiang Liu, Boyang Wang, Yifan Zhang, Qi Cui, Fei Kang, Biao Jiang, Mengyin An, Yangyang Ren, et al. Matrix-game 2.0: An open-source, real-time, and streaming interactive world model. *arXiv preprint arXiv:2508.13009*, 2025.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *The twelfth international conference on learning representations*, 2023.

Mike Innes, Alan Edelman, Keno Fischer, Chris Rackauckas, Elliot Saba, Viral B Shah, and Will Tebbutt. A differentiable programming system to bridge machine learning and scientific computing. *arXiv preprint arXiv:1907.07587*, 2019.

Anssi Kanervisto, Dave Bignell, Linda Yilin Wen, Martin Grayson, Raluca Georgescu, Sergio Valcarcel Macua, Shan Zheng Tan, Tabish Rashid, Tim Pearce, Yuhan Cao, et al. World and human action models towards gameplay ideation. *Nature*, 638(8051):656–663, 2025.

Diederik Kinga, Jimmy Ba Adam, et al. A method for stochastic optimization. In *International conference on learning representations (ICLR)*, volume 5. California;, 2015.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114 (13):3521–3526, 2017.

Carver Mead and Mohammed Ismail. *Analog VLSI implementation of neural systems*, volume 80. Springer Science & Business Media, 2012.

OpenAI. Sora by openai. https://openai.com/sora/, 2024. Accessed: 2025-07-14.

OpenAI. Sora 2 is here. https://openai.com/index/sora-2/, September 2025.

Jorge Pérez, Pablo Barceló, and Javier Marinkovic. Attention is turing-complete. *Journal of Machine Learning Research*, 22(75):1–35, 2021.

Matthieu Queloz. Explainability through systematicity: The hard systematicity challenge for artificial intelligence. *Minds and machines*, 35(3):35, 2025.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pp. 8748–8763. PmLR, 2021.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020.

Scott Reed and Nando De Freitas. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

Scott Reed, Konrad Zolna, Emilio Parisotto, Sergio Gomez Colmenarejo, Alexander Novikov, Gabriel Barth-Maron, Mai Gimenez, Yury Sulsky, Jackie Kay, Jost Tobias Springenberg, et al. A generalist agent. *arXiv preprint arXiv:2205.06175*, 2022.

Luke Rivard, Sun Sun, Hongyu Guo, Wenhu Chen, and Yuntian Deng. Neuralos: Towards simulating operating systems via neural generative models. *arXiv preprint arXiv:2507.08800*, 2025.

Jürgen Schmidhuber. *Making the world differentiable: on using self supervised fully recurrent neural networks for dynamic reinforcement learning and planning in non-stationary environments*, volume 126. Inst. für Informatik, 1990.

Jürgen Schmidhuber. On learning to think: Algorithmic information theory for novel combinations of reinforcement learning controllers and recurrent neural world models. *arXiv preprint arXiv:1511.09249*, 2015.

Hava T Siegelmann and Eduardo D Sontag. On the computational power of neural nets. In *Proceedings of the fifth annual workshop on Computational learning theory*, pp. 440–449, 1992.

Weihao Tan, Wentao Zhang, Xinrun Xu, Haochong Xia, Ziluo Ding, Boyu Li, Bohan Zhou, Junpeng Yue, Jiechuan Jiang, Yewen Li, et al. Cradle: Empowering foundation agents towards general computer control. *arXiv preprint arXiv:2403.03186*, 2024.

Alan Mathison Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

John Von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

Team Wan, Ang Wang, Baole Ai, Bin Wen, Chaojie Mao, Chen-Wei Xie, Di Chen, Feiwu Yu, Haiming Zhao, Jianxiao Yang, et al. Wan: Open and advanced large-scale video generative models. *arXiv preprint arXiv:2503.20314*, 2025.

Thaddäus Wiedemer, Yuxuan Li, Paul Vicol, Shixiang Shane Gu, Nick Matarese, Kevin Swersky, Been Kim, Priyank Jaini, and Robert Geirhos. Video models are zero-shot learners and reasoners. *arXiv preprint arXiv:2509.20328*, 2025.

Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, Jan Peters, and Jürgen Schmidhuber. Natural evolution strategies. *The Journal of Machine Learning Research*, 15(1):949–980, 2014.

Maurice Wilkes. The best way to design an automatic calculating machine. 1981.

Lvmin Zhang, Anyi Rao, and Maneesh Agrawala. Adding conditional control to text-to-image diffusion models. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 3836–3847, 2023.

Appendix

Overview. This submission is long because the project explores an extended chain of ideas that are new to this community. We design the full data-collection pipeline, train seven models in total (three CLIGen and four GUIWorld), and report extensive ablations, qualitative analyses, and system-level insights. We also introduce the definition of, and potential solutions toward, future completely neural computers (CNCs). The appendix is not filler. It contains material we consider essential for understanding the design trade-offs and the main paper's conclusions. We would greatly appreciate careful attention to the appendix when summarizing or reviewing the work, especially if LLM-based tools are used for condensation.

Table of Contents

A VISUALIZATIONS (WE KINDLY SUGGEST SCROLLING DOWN TO THE BOTTOM OF THE PDF)

A.1 CLIGEN VISUALIZATIONS

This subsection consolidates CLIGen visualization pages referenced in the main text and moves them here for space, with additional examples provided below. Each page pairs a scripted input trace—typed strings, explicit `Enter` events, and controlled sleep intervals—with the corresponding frame sequence, enabling fine-grained inspection of timing, cursor placement, and incremental output formation.

(1) CLIGen (General) visualizations. Qualitative samples highlight the breadth of real-world terminal dynamics captured in CLIGen (General): ANSI escape sequences that repaint regions with changing foreground/background colors, incremental command entry with syntax highlighting and cursor edits (e.g., SQL `CREATE TABLE`), classic shell prompts and system outputs (e.g., `date`), long-running jobs with rapidly scrolling and color-coded package logs, full-screen TUIs such as partition editors with menus and tables, and progress dashboards with updating bars, counts, and ETAs. These traces emphasize that "looking correct" requires maintaining terminal geometry, palette transitions, and cursor state frame-by-frame.

(2) CLIGen (Clean) REPL visualizations. In contrast to open-world traces, CLIGen (Clean) REPL samples are scripted and temporally well-paced (Figures 12–13; additional examples are provided in Appendix F). Each sample includes an explicit action trace (e.g., `Sleep`, `Type`, `Enter`, arrow keys, `Hide`) alongside the rendered terminal frames, making the causal link between key-level actions and pixel updates visually unambiguous. The rollouts emphasize incremental typing under a stable prompt (here, a > prompt with a pink block cursor), with deterministic outputs such as printing `$HOME` (yielding `/home/NeuralComputer`), inspecting environment variables via `env | head -n 5` (showing fields like `DEBIAN_FRONTEND=noninteractive` and UTF-8 locale), and composing longer pipelines (e.g., `seq+paste+column+tee`) that materialize small aligned tables. They also include dense TUI-like screens where layout fidelity is critical: `top` must preserve column alignment and header/footer bands, while `cal` must keep a fixed-width calendar grid with consistent spacing. Finally, multi-line programming snippets (a `python - <<'PY'` heredoc) stress quoting, indentation, and line wrapping, and the resulting output produces repeated, structured progress-like traces ("Frame ..." lines with repeated glyphs).

The main insight is that these scripted REPL traces isolate rendering-and-control errors from semantic ambiguity: given an explicit action stream, failures are dominated by low-level mechanics (cursor placement, character edits, monospace alignment, line breaks, and temporal consistency) rather than uncertainty about what content should be produced. This makes CLIGen (Clean) particularly useful for diagnosing whether a neural computer can faithfully execute action-conditioned text rendering under strict layout constraints.

(3) CLIGen (Clean) math visualizations. Figures 14–15 compare model rollouts on CLIGen (Clean) math REPL prompts, where a scripted action trace launches `python` and types a short expression (e.g., `5`, `10+15`, `40/1`); additional examples are provided in Appendix F. Sora2 most consistently preserves the intended REPL structure: it renders the Python banner, maintains a stable `>>>` prompt, and produces the expected numeric output with minimal drift. In contrast, Wan2.1 frequently fails at the interface level, producing corrupted prompts, spurious control-token text (e.g., literal `<Enter>`), or collapsing into non-terminal visuals, which aligns with its near-zero arithmetic-probe accuracy. $\text{NC}_{\text{CLIGen}}$ and Veo3.1 occupy a middle regime: they often keep a terminal-like layout and can render plausible outputs, but the visualizations highlight remaining brittleness in character-level fidelity and mode stability (e.g., prompt corruption, stray glyphs, or occasional off-domain imagery).

Figures 16–17 further illustrate why these math probes do not cleanly measure "native" reasoning; additional examples are provided in Appendix F. In the reprompted setting, we explicitly inject the solution string into the script (e.g., "The answer is 5/282/473984" for `28-23`, `560-278`, and `736*644`), and the video model only needs to render the provided output under the correct REPL layout. In these examples, the reprompted $\text{NC}_{\text{CLIGen}}$ rollouts cleanly render the injected answers, while Sora2 is often able to produce correct outputs directly and can remain numerically coherent even on three-digit multiplication (though occasional digit-level slips still occur). Given CLIGen's non-trivial OCR fidelity (0.54 character accuracy, 0.31 exact-line accuracy), this kind of answer-conditioned rollout can *look* like strong symbolic competence even when the generator itself does not reliably compute. This is consistent with the large boost we observe under reprompting (4%→83%) and supports a practical system-level interpretation: if an external LLM (or tool) computes answers and the video model faithfully renders them, the overall neural-computer system can exhibit "strong reasoning" behavior on-screen. Conversely, evaluating "reasoning" in video generation requires carefully separating computation from answer-conditioned rendering.

(4) CLIGen (Clean) REPL rollouts. Figures 18–19 provide additional scripted rollouts that complement Figures 12–13. Figure 18 contains two short sessions: (i) a lightweight shell probe that prints `$HOME` (via `echo` and `printenv`) and a timestamp using `date +%H:%M:%S`, where the rollout makes prompt transitions and line breaks explicit; and (ii) a here-doc Python snippet (`python - <<'PY'`) that imports `time` and prints a sequence of lines of the form `Frame {i:02d}:` followed by a growing run of > characters with `time.sleep(0.2)` between iterations, stressing scrolling behavior and temporal alignment. Figure 19 broadens the command coverage with common environment/system queries and didactic echoing: it reads `$HISTSIZE` (showing `History size: 500`), prints a calendar (`cal` for October 2025), and again resolves `$HOME`; a second snippet echoes "Learning shell basics", prints a date in `YYYY-MM-DD` format (e.g., `2025-10-04`), reports the login shell via `$0` (e.g., `bash`), and queries the kernel release with `uname -r` (e.g., `6.10.0-linuxkit`). Across these examples, the visualization makes clear how the same scripted input decomposes into per-frame partial typing, command submission, and output appearance.

(5) CLIGen (Clean) math comparisons. Figure 20 extends the math REPL comparisons from Figures 14–15 using the same minimal script: start `python`, wait briefly, then evaluate `40/1`. The comparison is visually diagnostic: a faithful rollout preserves recognizable interpreter structure (banner/prompt/output, with `40` rendered under the >>> prompt), while failure cases exhibit broken text rendering, missing or blank frames, or even clearly off-manifold content (non-terminal textures/icons) that disrupts the expected REPL continuity despite identical scripted inputs.

(6) CLIGen (Clean) reprompting. Figure 21 provides an additional reprompting example beyond Figures 16–17. The scripted intent is to compute `736*644`, and the page explicitly annotates the correct result (`473984`). The rollouts illustrate several practical failure modes that motivate reprompting: a correct completion that matches the annotation; a drift where the expression itself changes (e.g., `756*964`) and yields an unrelated result; and near-miss arithmetic where the expression is preserved but the output differs by a digit (e.g., `973984`). In this setting, reprompting is not merely about retrying generation, but about recovering the intended calculation and restoring consistency between the typed expression and the displayed answer.

A.2 GUIWorld Visualizations

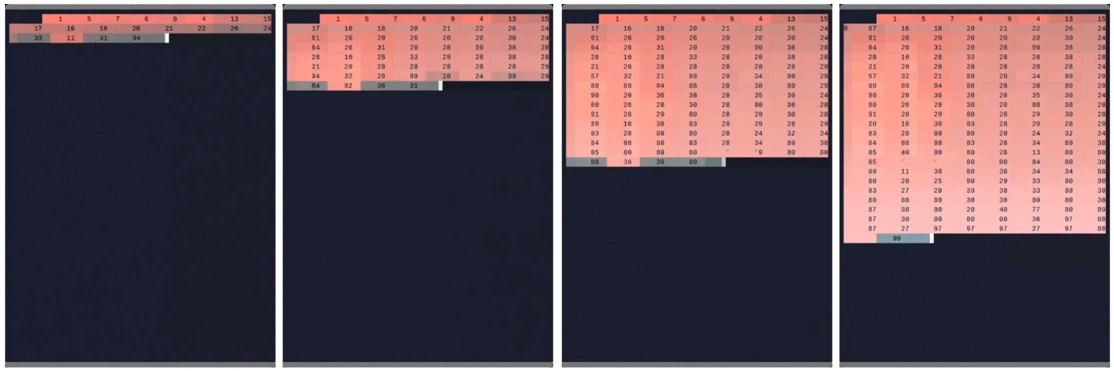
This subsection collects GUIWorld rollout visualizations referenced in the main text and additional samples. Each page overlays the CUA trace (natural-language *thinking* plus structured *action* fields such as `left_click`, `double_click`, `left_click_drag`, and `type`) and contrasts the *Ground Truth* trajectory (top) with a *Generation* conditioned on the first frame and the action sequence (bottom), making state drift easy to spot.

Across GUIWorld interactive rollouts, failure modes are dominated by data quality and by *where* action information enters the backbone. Goal-directed supervision produces smooth, target-aligned cursor paths and consistent post-click UI transitions, whereas random exploration yields bursty jitter and spurious actions that degrade visual coherence (Table 5; Figures 22–26). Consistent with the action-driven metrics in Table 7, deeper token-level injection (`contextual`/`internal`) yields more reliable post-action updates in interactive elements (hover states, dropdowns, modals) and maintains cursor alignment under rapid motion.

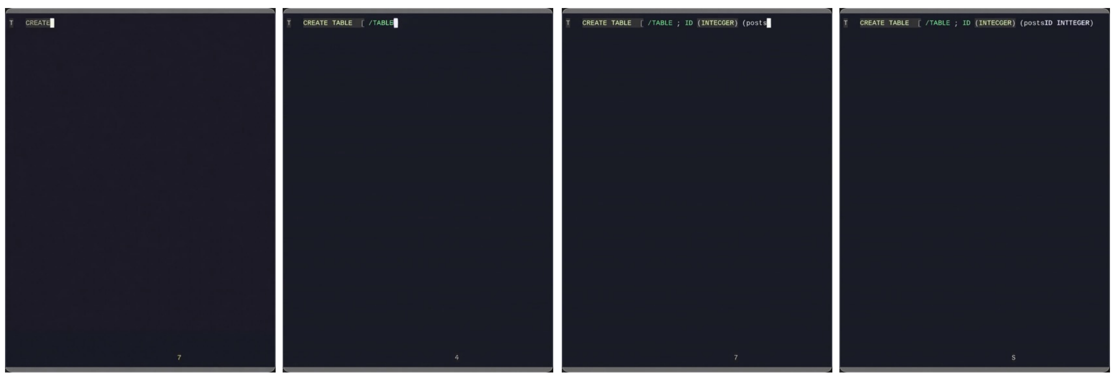
Figures 27–29 emphasize how small low-level deviations compound. In Figure 27, the intent is calculator entry: after clicking `2`, the agent clicks `0` twice to form `200` in ground truth, but the generated rollout ends with a different displayed value (e.g., `54`). Figure 28 shows a menu-navigation task in GIMP (opening `File` and selecting `Open`): the ground truth reaches the GTK "Open Image" file chooser, whereas the generation diverges into a different dialog (a "Create Image" window), despite starting from the same menu interaction. Figure 29 covers file navigation in Nautilus by double-clicking `Downloads`; the generated rollout reaches a visually similar folder view, but the file listing/thumbnails differ from the ground truth (e.g., altered filenames and icons), indicating latent state mismatch even when the high-level navigation appears correct.

Figures 30–32 focus on numeric/UI fidelity and interaction semantics. Figure 30 captures resizing in GIMP's "Scale Image" dialog: the ground truth types `512` into the width field with the aspect-ratio "chain" enabled (height updates accordingly) and then clicks `Scale`, producing a resized "wp" image; the generation fails to preserve the intended numeric edit (the dimensions drift to unrelated values such as `3088`) and breaks the expected width–height coupling. Figure 31 illustrates a drag gesture (`left_click_drag`) intended to create a square selection on a blank canvas: the ground truth remains a selection operation, while the generation hallucinates a filled black square, conflating "selection" with "painting" outcomes. Figure 32 shows launching Firefox from the desktop and navigating via the address bar toward Wikipedia: the ground truth produces a coherent suggestion dropdown (including `wikipedia.org`), while the generated rollout preserves the coarse UI layout but exhibits corrupted/garbled text and unstable suggestion content.

Finally, Figures 33–35 add three further stress cases where correctness hinges on precise field edits and page state. Figure 33 shows creating a new image in GIMP by setting the height to `768` and confirming; the generated rollout produces a different height (e.g., `1090`) and thus a different canvas geometry. Figure 34 types `https://example.com/` in Firefox; the ground truth maintains a stable address bar and dropdown behavior, while the generation drifts in both URL/text rendering (garbled page copy and unstable address-bar content). Figure 35 navigates pagination on the Ubuntu Tutorials site: the ground truth clicks page `2` (ending at `/tutorials?page=2`), whereas the generation partially matches the pagination control but highlights an incorrect page index and ultimately diverges into an inconsistent page layout, illustrating how small control-level errors can produce large downstream state differences.

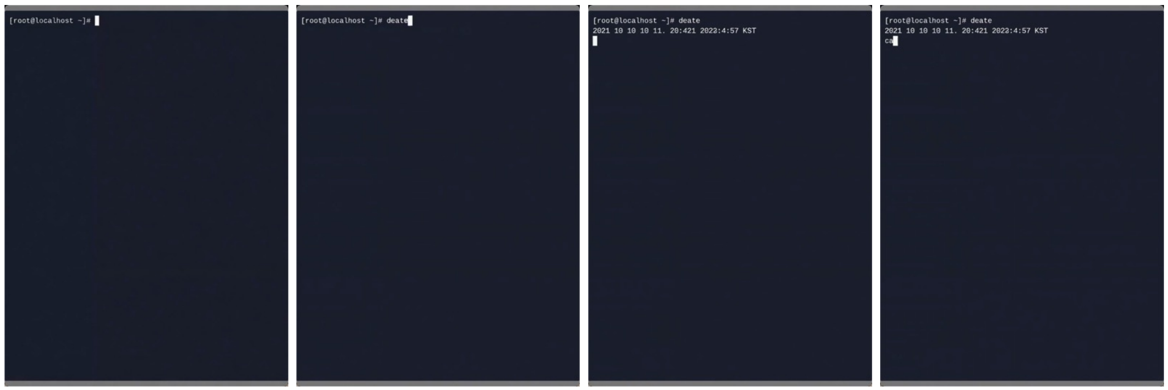


The terminal displays a series of ANSI escape code formatted texts with changing background and foreground colors, executing commands like `\\u001b[48;2;255;128;128;38;2;0;0;0m` which set the background to a shade of pink and text to black, and printing numbered lists with colors. The output includes specific numbers, such as \"1\", \"5\", \"7\", and \"9\", in different colors, creating a visually dynamic and colorful display, but the exact username, hostname, and path are not specified in the provided terminal session content.

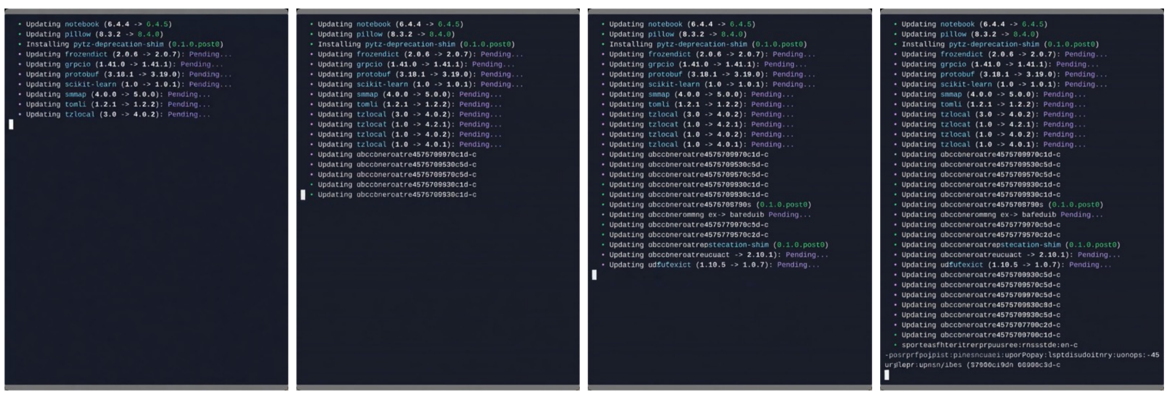


The user types the command `CREATE TABLE posts (ID INTEGER)`, with the terminal displaying the command in a dark background with colored syntax highlighting, including green and yellow text, and the cursor moving character-by-character as the user types, with some corrections and backspacing along the way. The output shows the command being executed, with key words like `CREATE` and `TABLE` in distinct colors, and the filename `posts` appearing in the command line.

Figure 9: CLIGen (General) visualization samples (A).

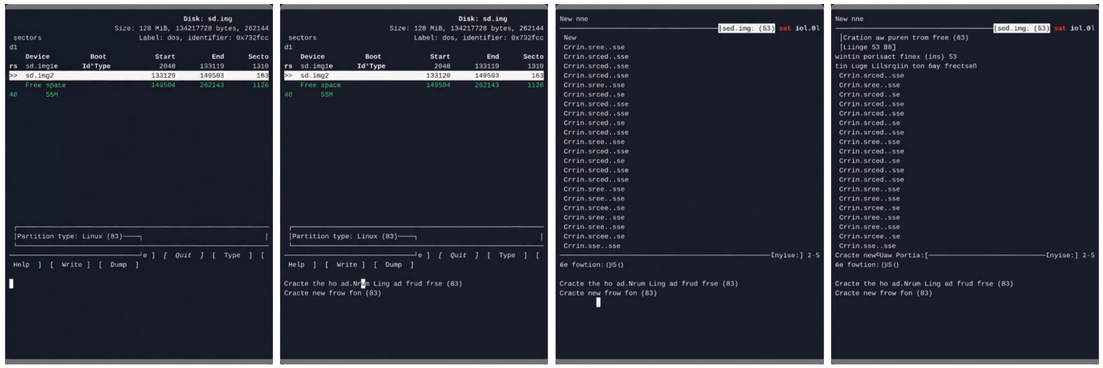


At the `root@localhost:~#` prompt, the user types the `date` command, which displays the current date and time in a plain text format as \"2021. 10. 11. 22:47:43 KST\", then begins typing the `cat` command.

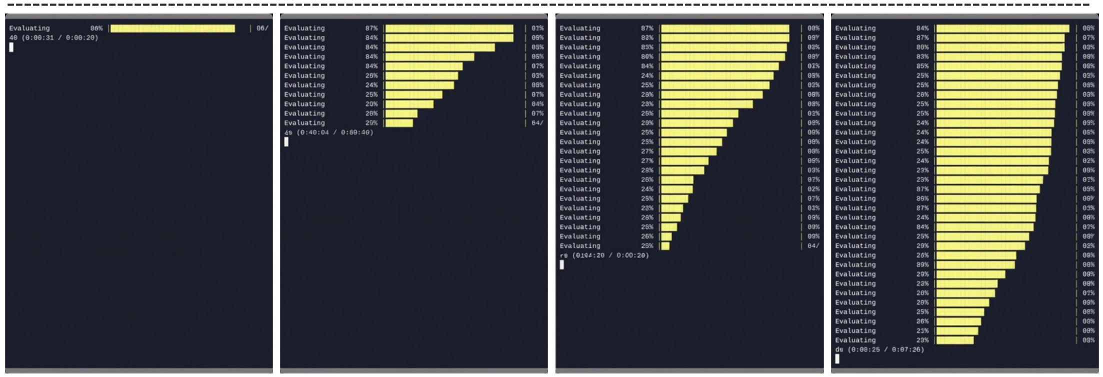


The terminal displaying progress bars, package names like `pillow`, `notebook`, and `tzlocal`, and version changes in green and red text. The output shows downloading and installing statuses, including percentages, for packages like `smmap`, `tomli`, and `protobuf`, with the terminal scrolling through the output rapidly.

Figure 10: CLIGen (General) visualization samples (B).



At the unspecified username@hostname prompt, the terminal displays a partition editor with a disk image file named \"sd.img\" (128 MiB) and the user interacts with it, creating a new Linux partition from free space, with key output content showing partition details in a table format, including \"sd.img1\" and \"sd.img2\" with their respective sizes and types, and a new partition \"sd.img3\" with 55M size and Linux type (83). The terminal shows a mix of black and colored text, including blue and red, with a cursor that blinks and moves to different parts of the screen as the user navigates through the partition editor options, such as \"New\", \"Quit\", and \"Write\", with specific prompts like \"Partition type: Linux (83)\" and \"Create new partition from free space\".



The terminal displays a progress bar with the command output \"Evaluating\" and percentages from 60% to 85%, showing yellow progress bars with increasing completion, such as \"|████████████████████ |\" to \"|█████████████████████████|\", alongside item counts \"24/40\" to \"34/40\" and time estimates \"0:00:20\" to \"0:00:07\". The output includes specific item completion and estimated time remaining, with the yellow-colored progress bar indicating the evaluation progress.

Figure 11: CLIGen (General) visualization samples (C).

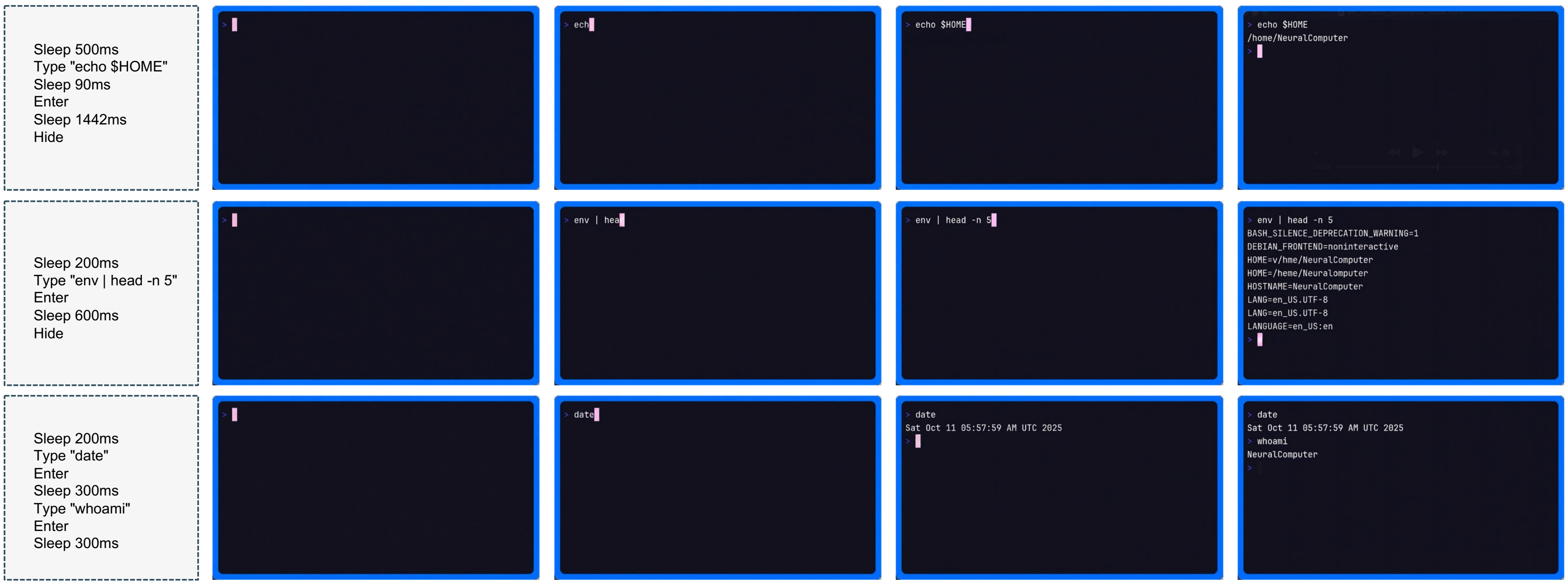


Figure 12: CLIGen (Clean) REPL visualization samples (A).

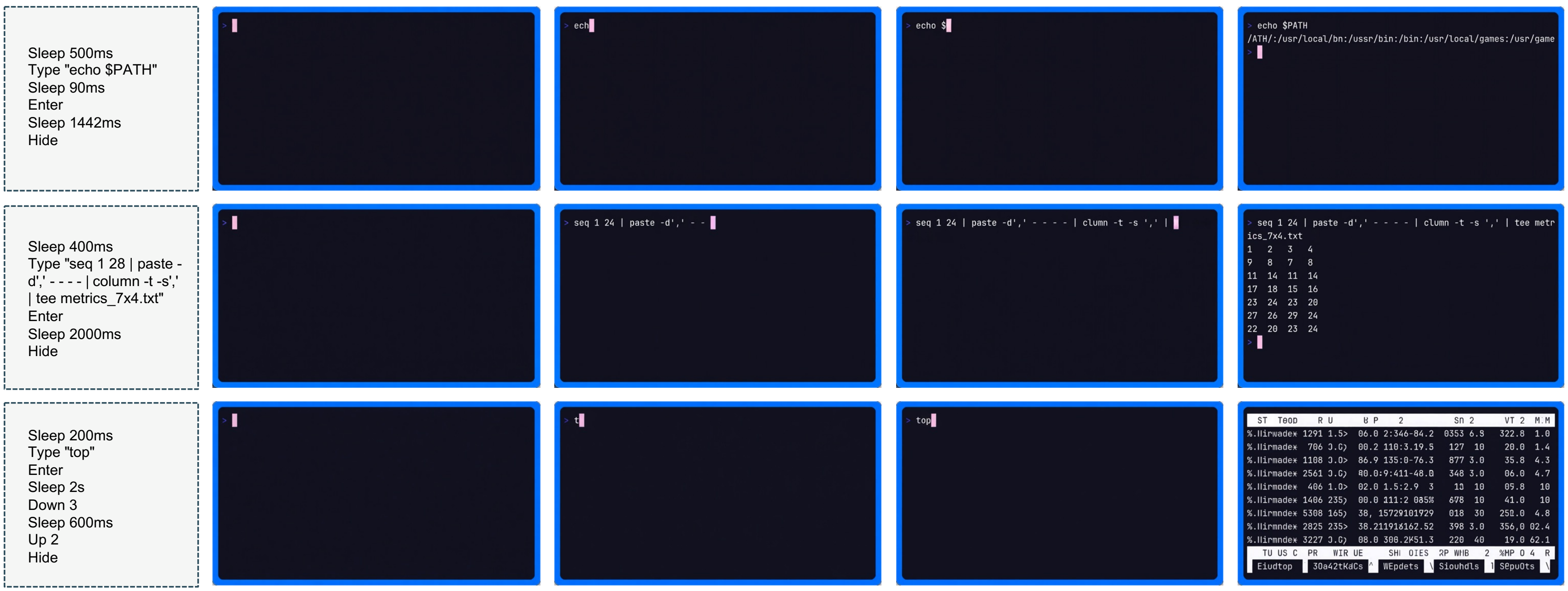


Figure 13: CLIGen (Clean) REPL visualization samples (B).

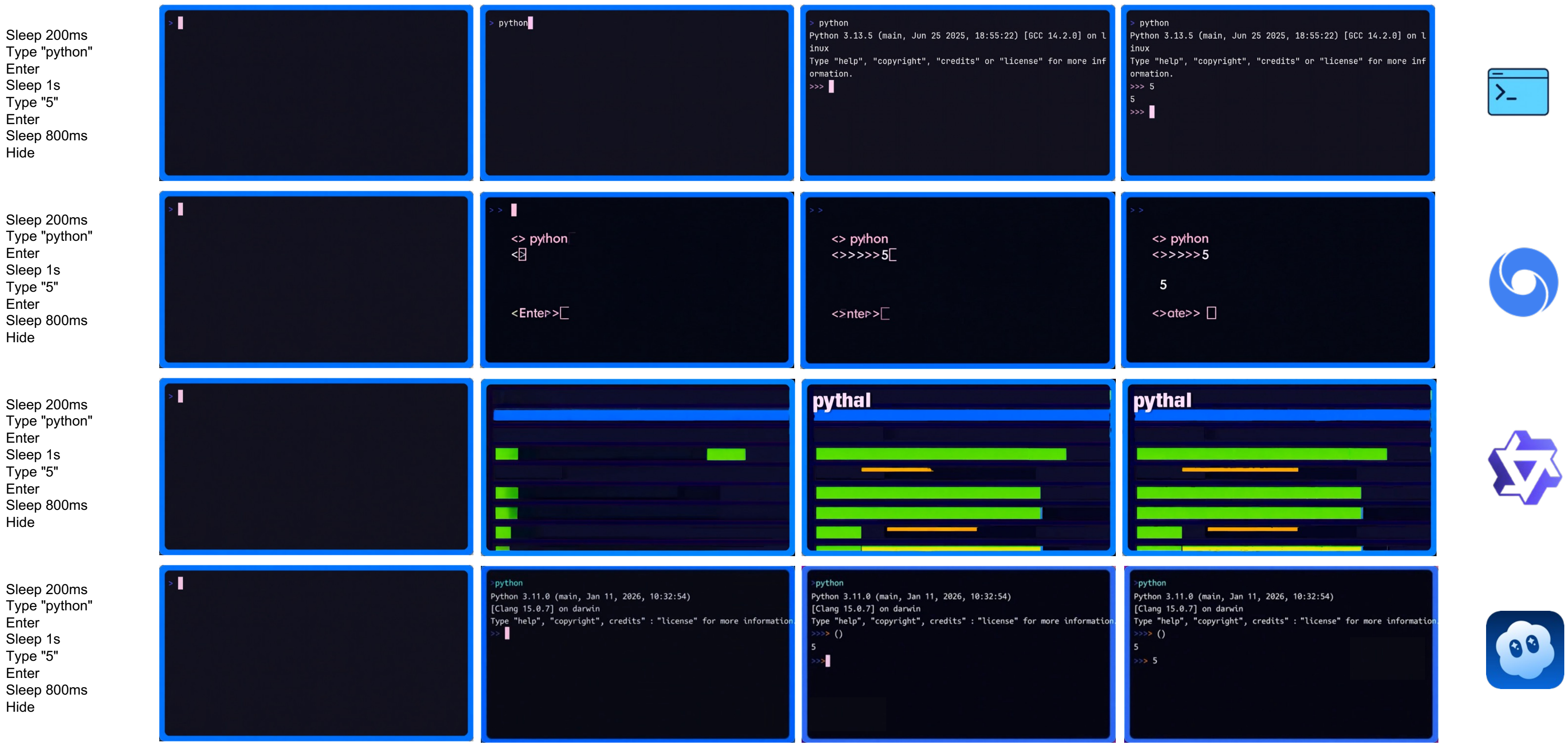


Figure 14: CLIGen (Clean) math comparison samples (A).

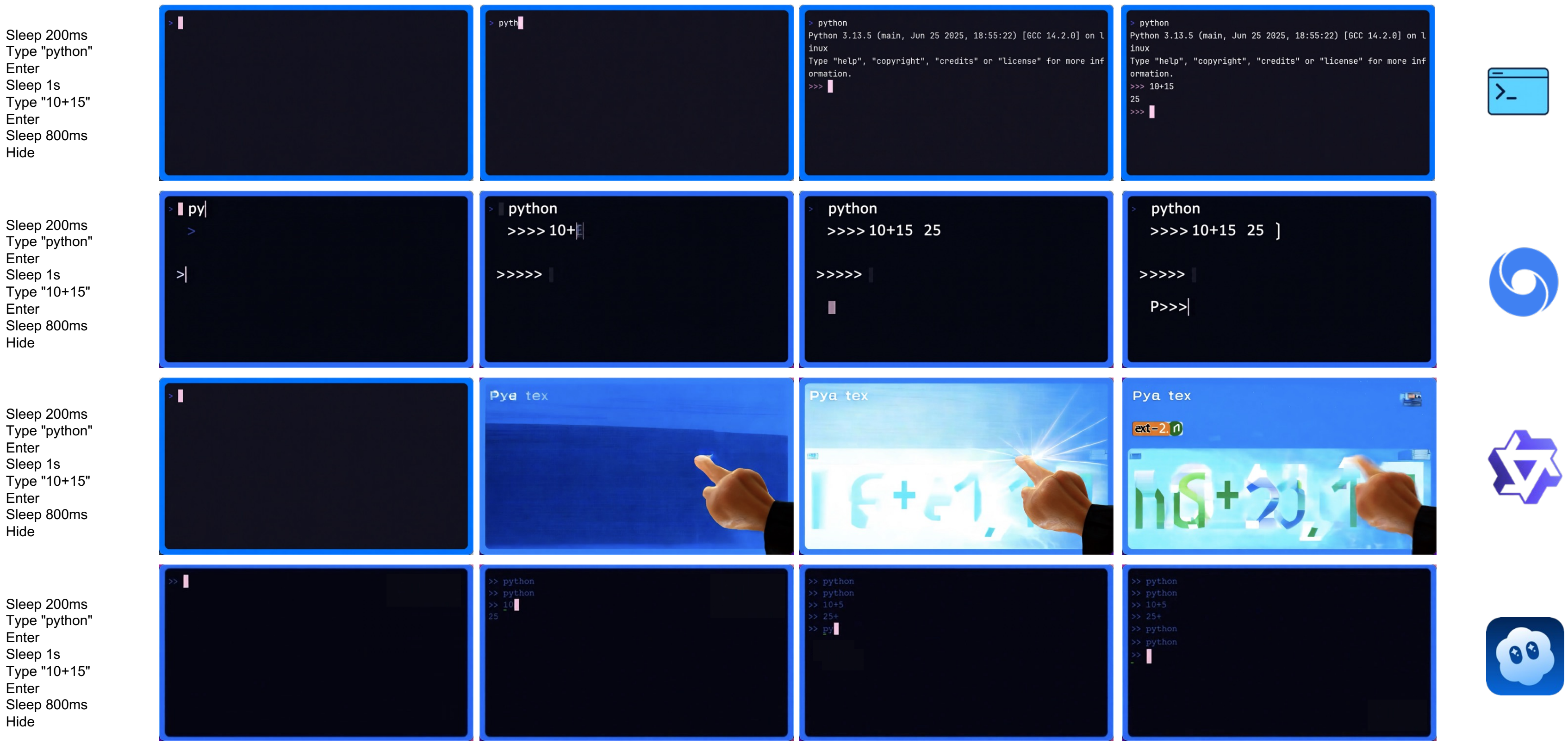


Figure 15: CLIGen (Clean) math comparison samples (B).

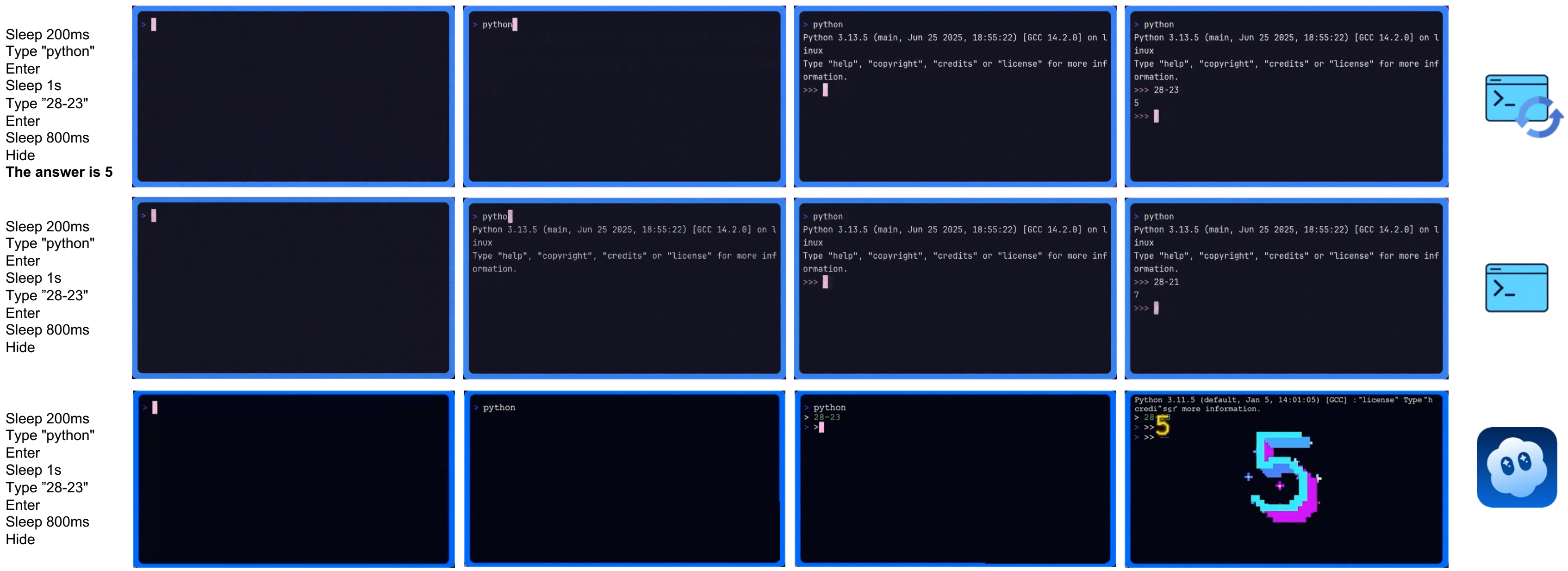


Figure 16: CLIGen (Clean) math reprompting samples (A).

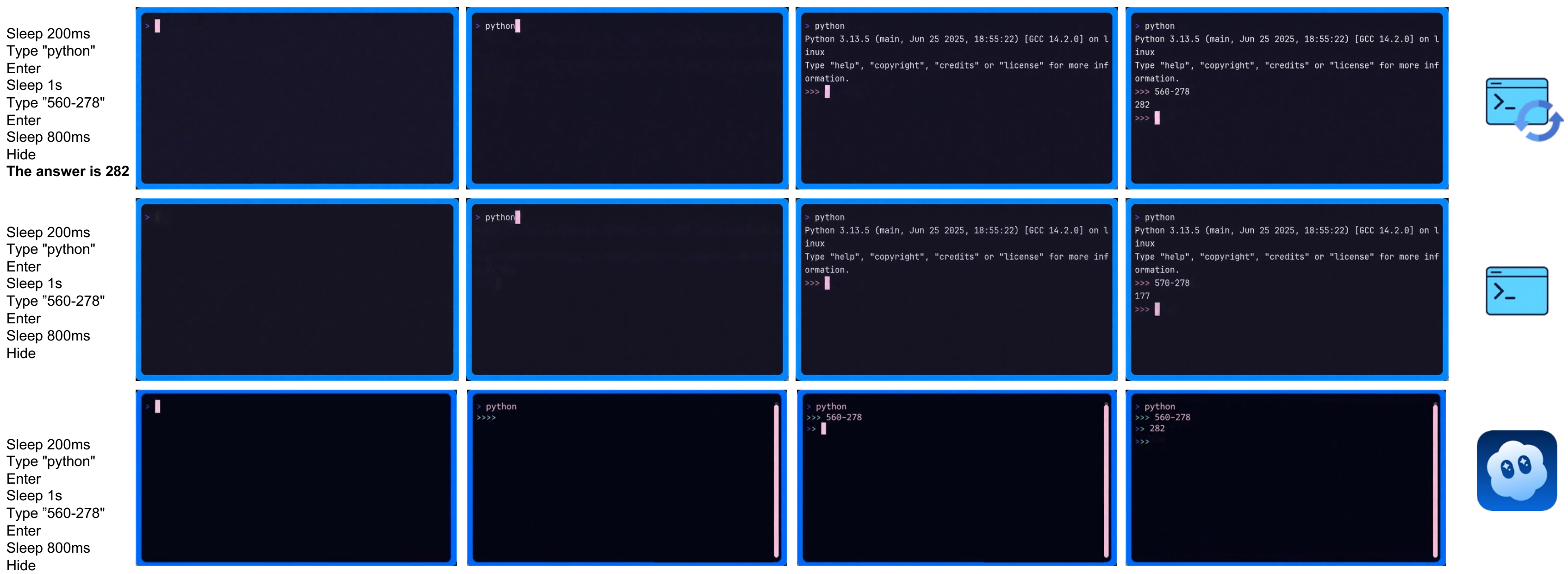


Figure 17: CLIGen (Clean) math reprompting samples (B).

Figure 18: CLIGen (Clean) REPL visualization samples (C).

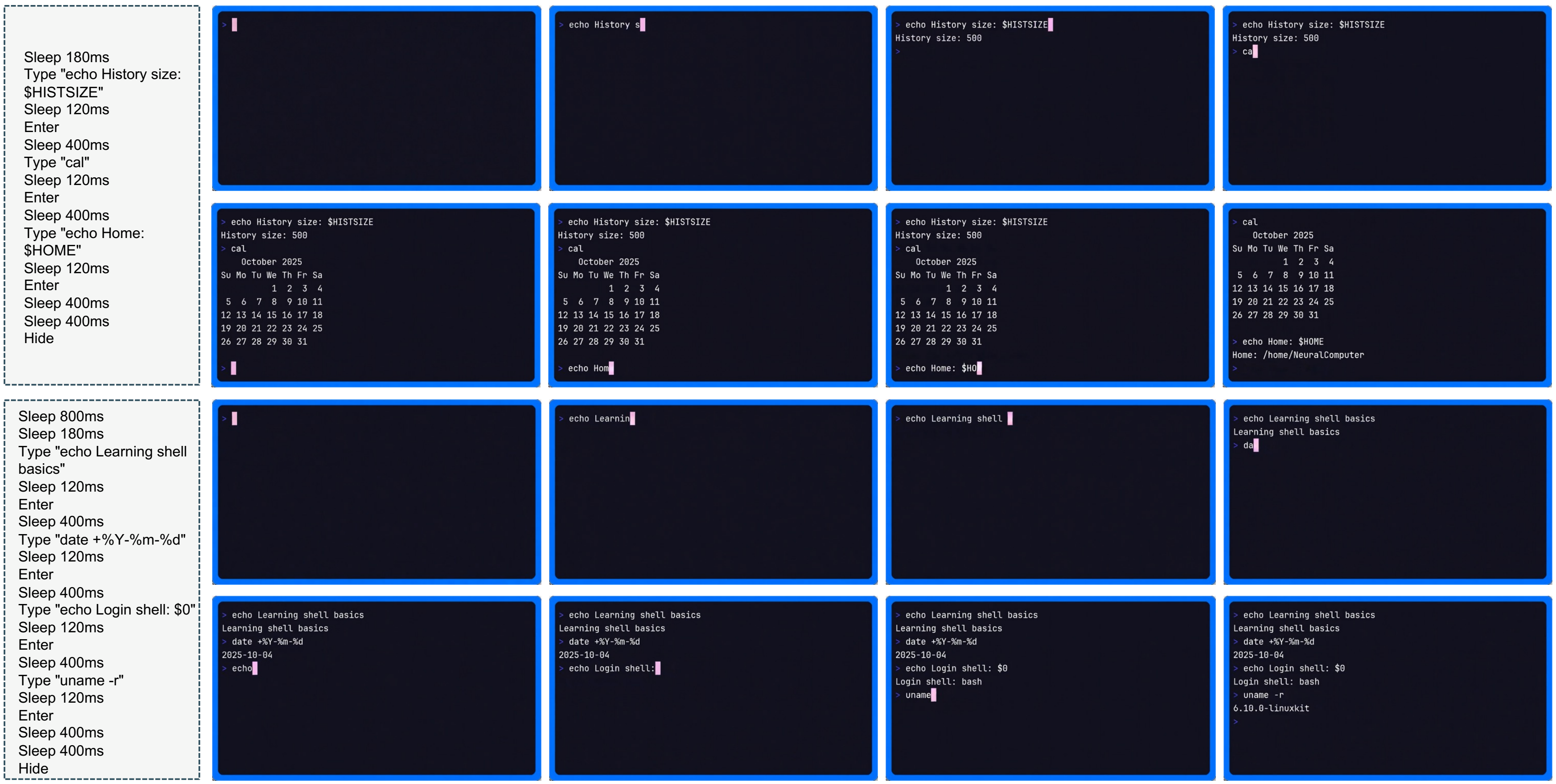


Figure 19: CLIGen (Clean) REPL visualization samples (D).

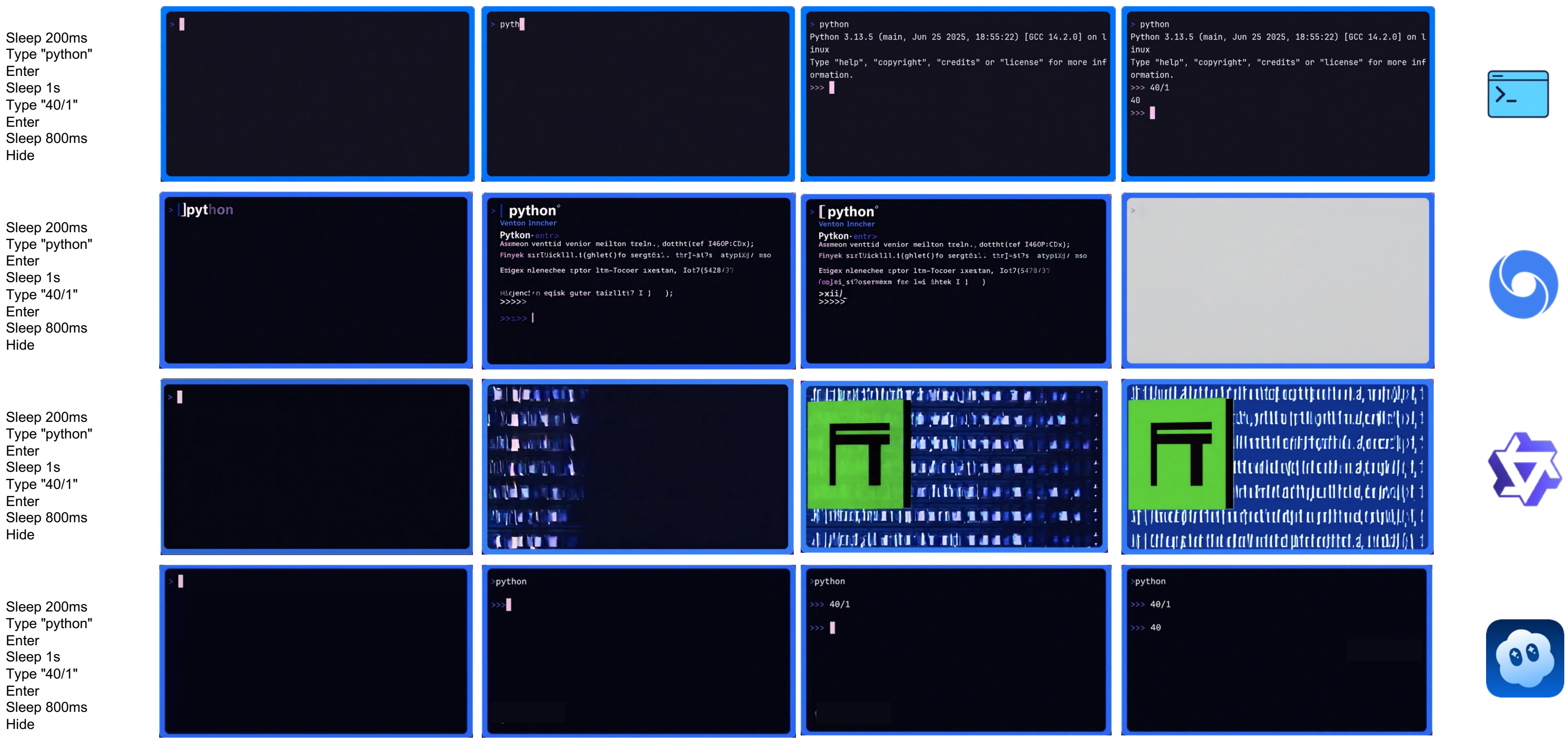


Figure 20: CLIGen (Clean) math comparison samples (C).

Figure 21: CLIGen (Clean) math reprompting samples (C).

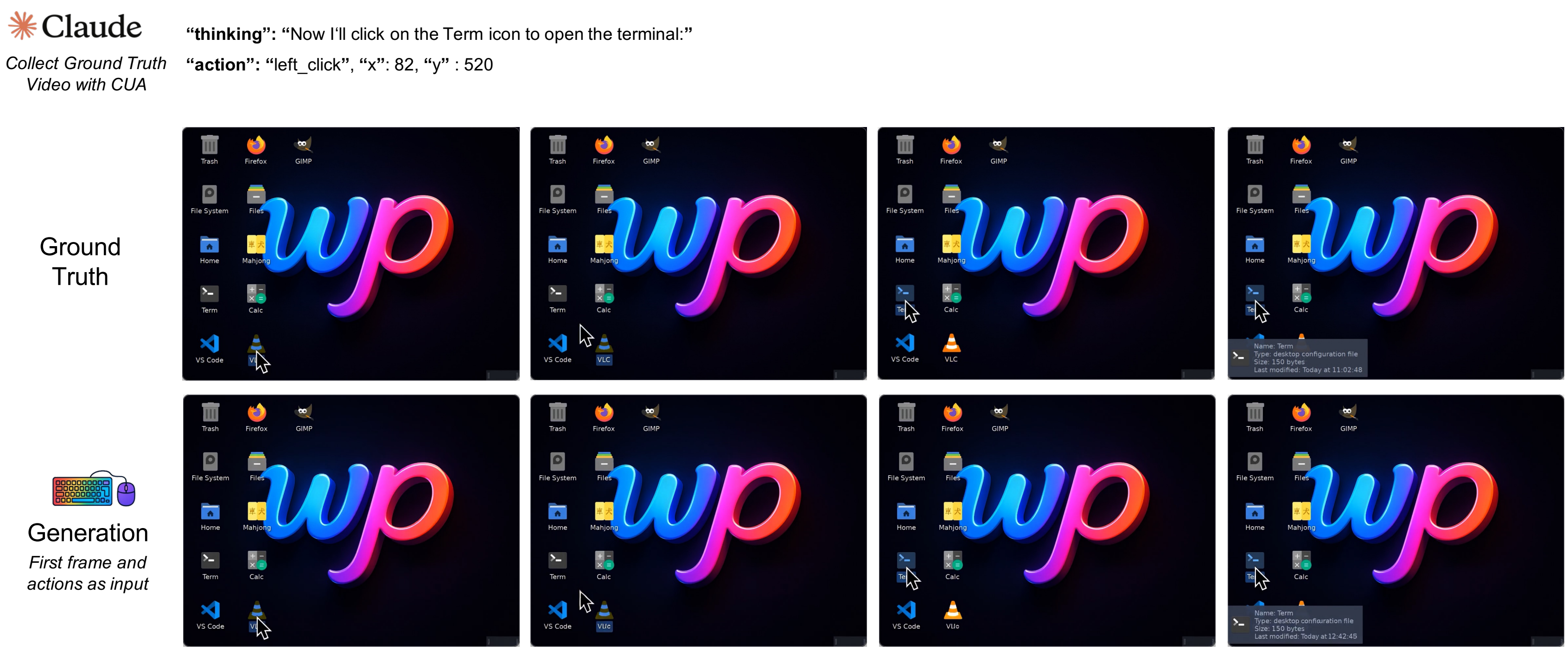


Figure 22: GUIWorld visualization sample (1).

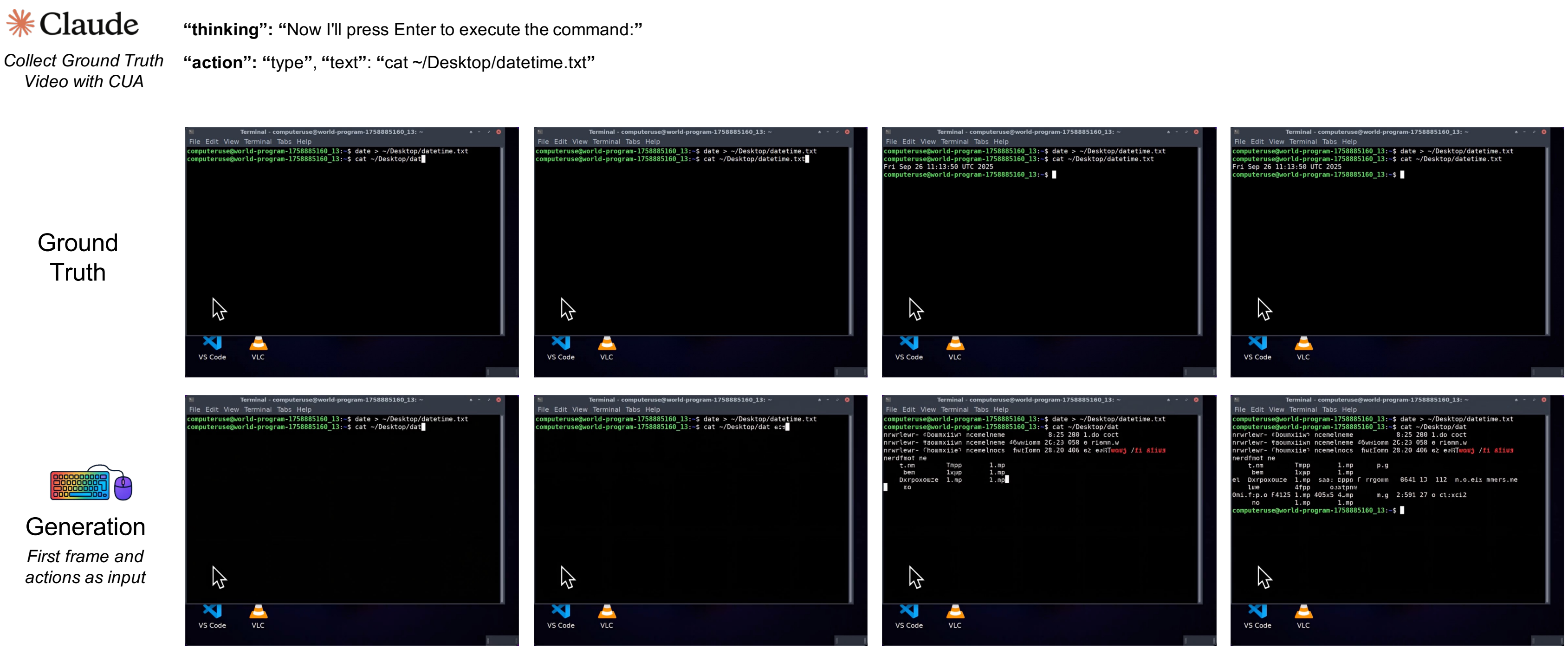


Figure 23: GUIWorld visualization sample (2).

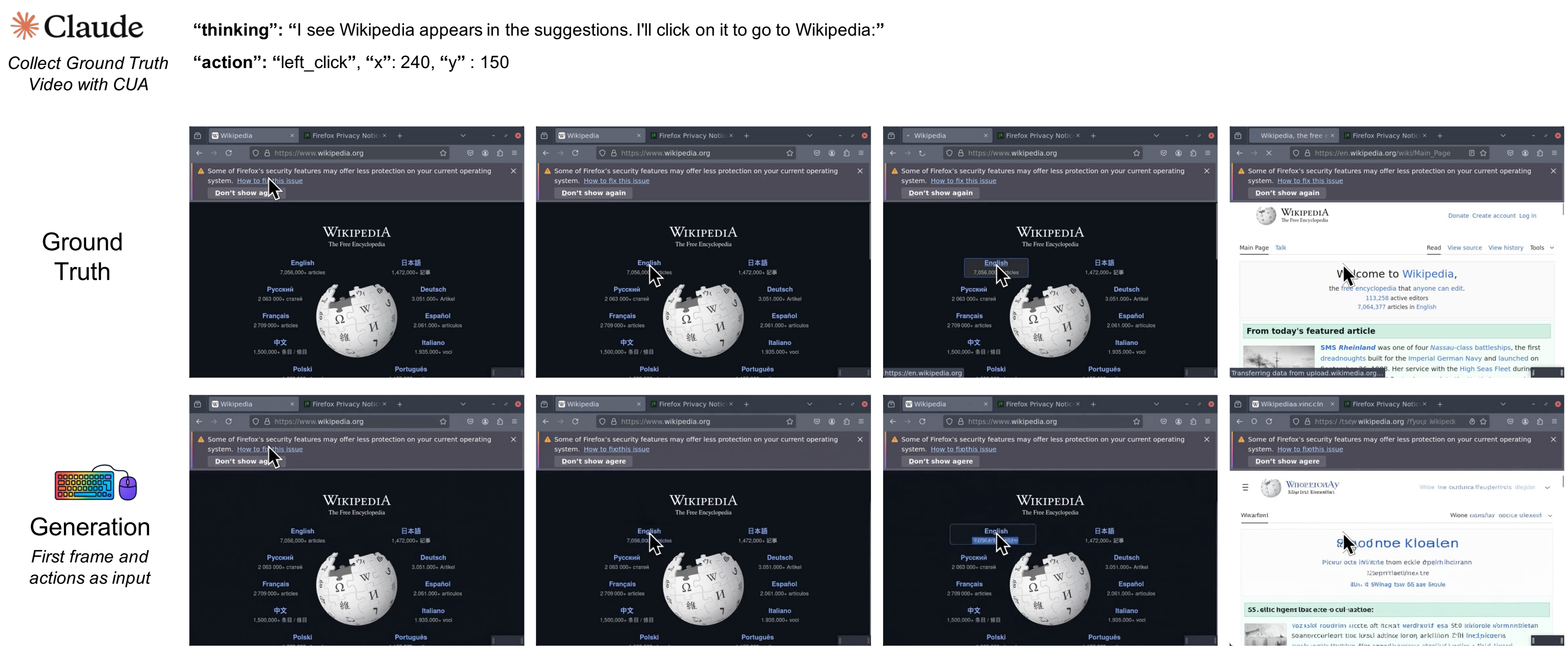


Figure 24: GUIWorld visualization sample (3).

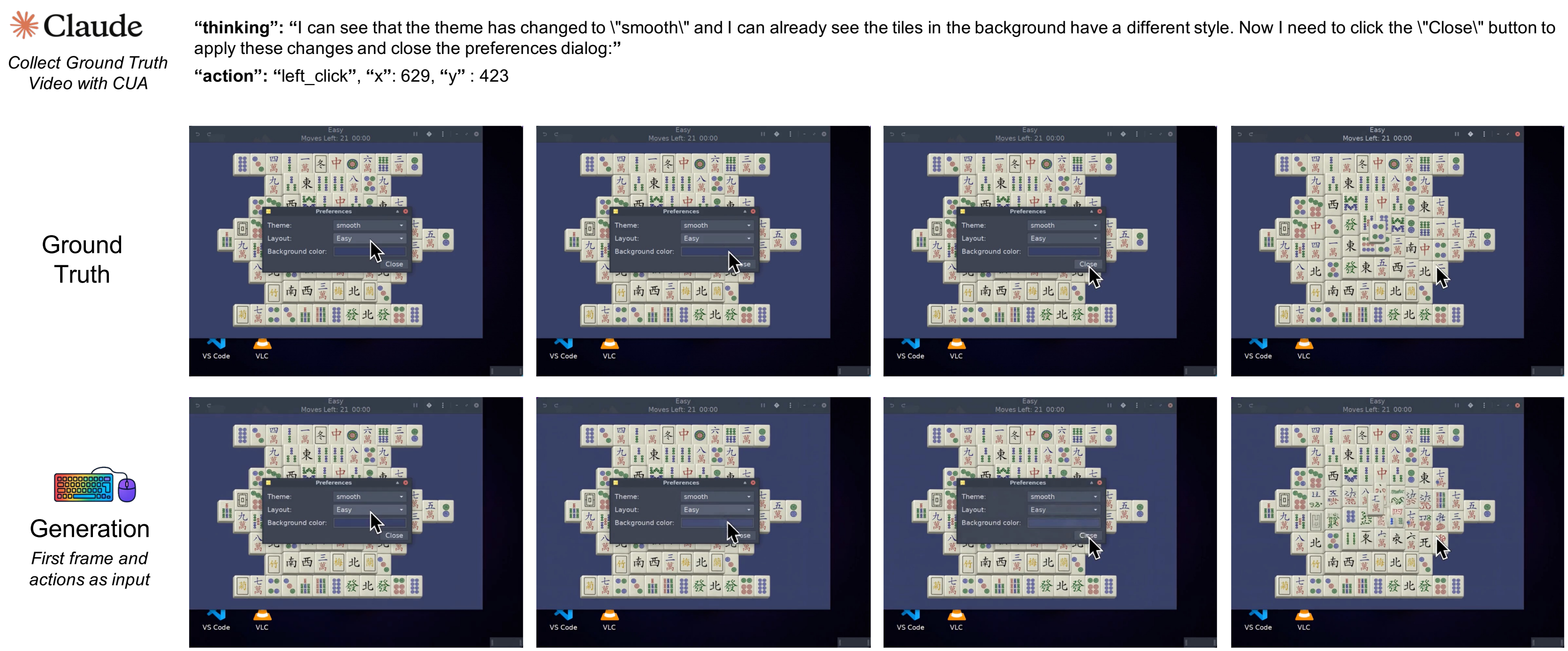


Figure 25: GUIWorld visualization sample (4).

Claude

Collect Ground Truth Video with CUA

“thinking”: **“**I can see the border_photo.png file that we created with ffmpeg. I'll double-click on it to open it in GIMP: ”

“action”: “double_click”, “x”: 310, “y”: 143

“thinking”: **“**Great! Now the image is open in GIMP. I can see it's a peachpuff colored image as requested. Now I'll add a border to it. In GIMP, I'll go to the \"Filters\" menu and look for a border option:”

“action”: “left_click”, “x”: 547, “y”: 43

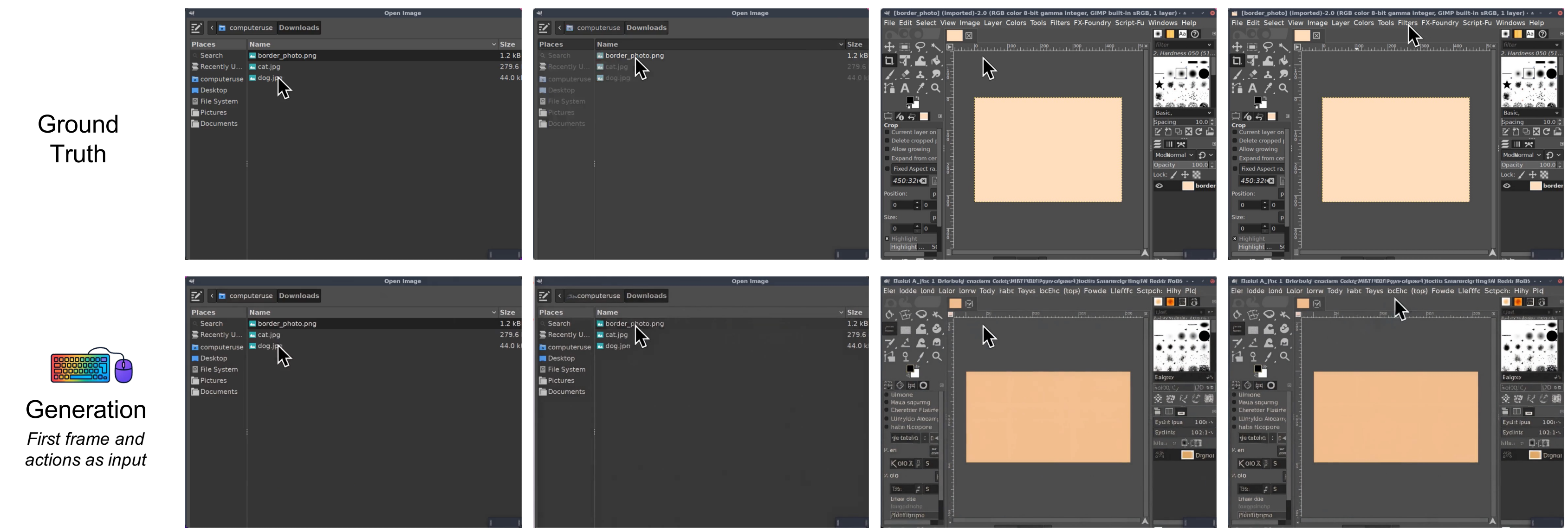


Figure 26: GUIWorld visualization sample (5).

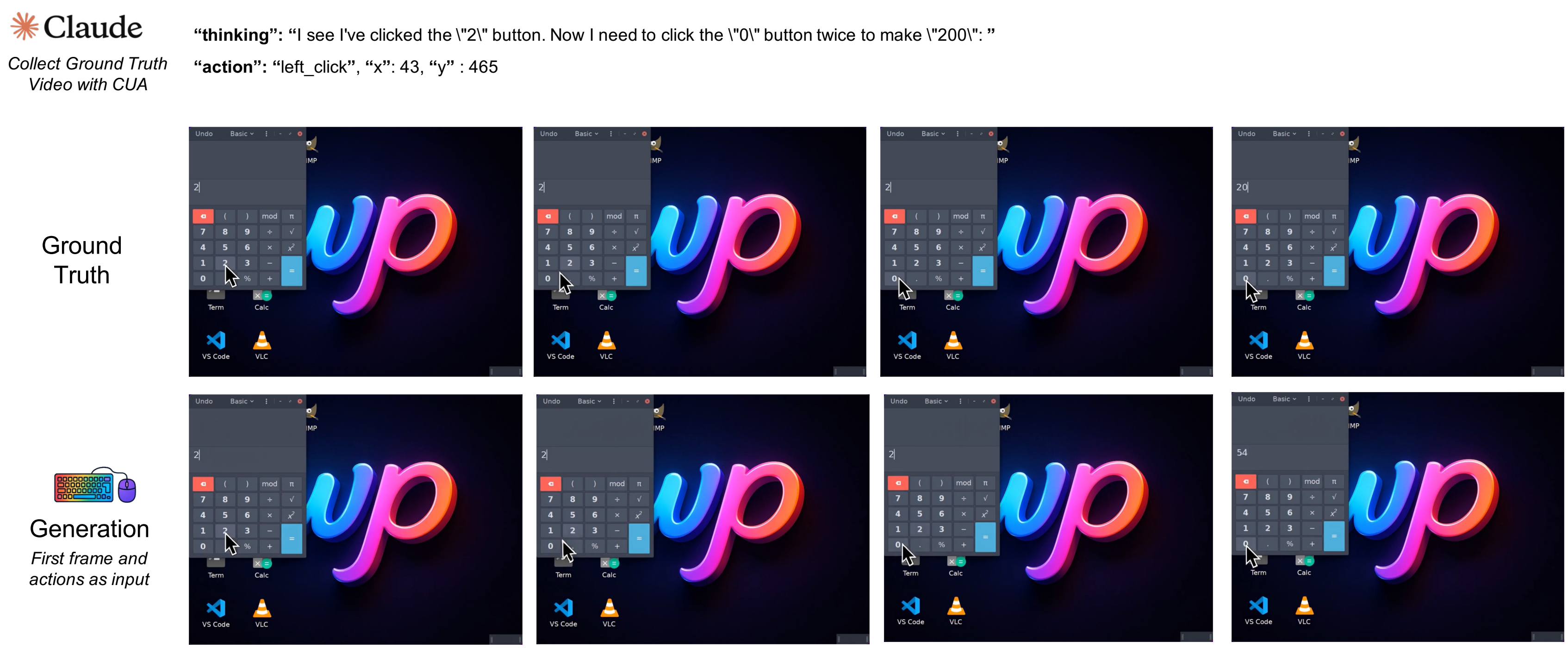


Figure 27: GUIWorld visualization sample (6).

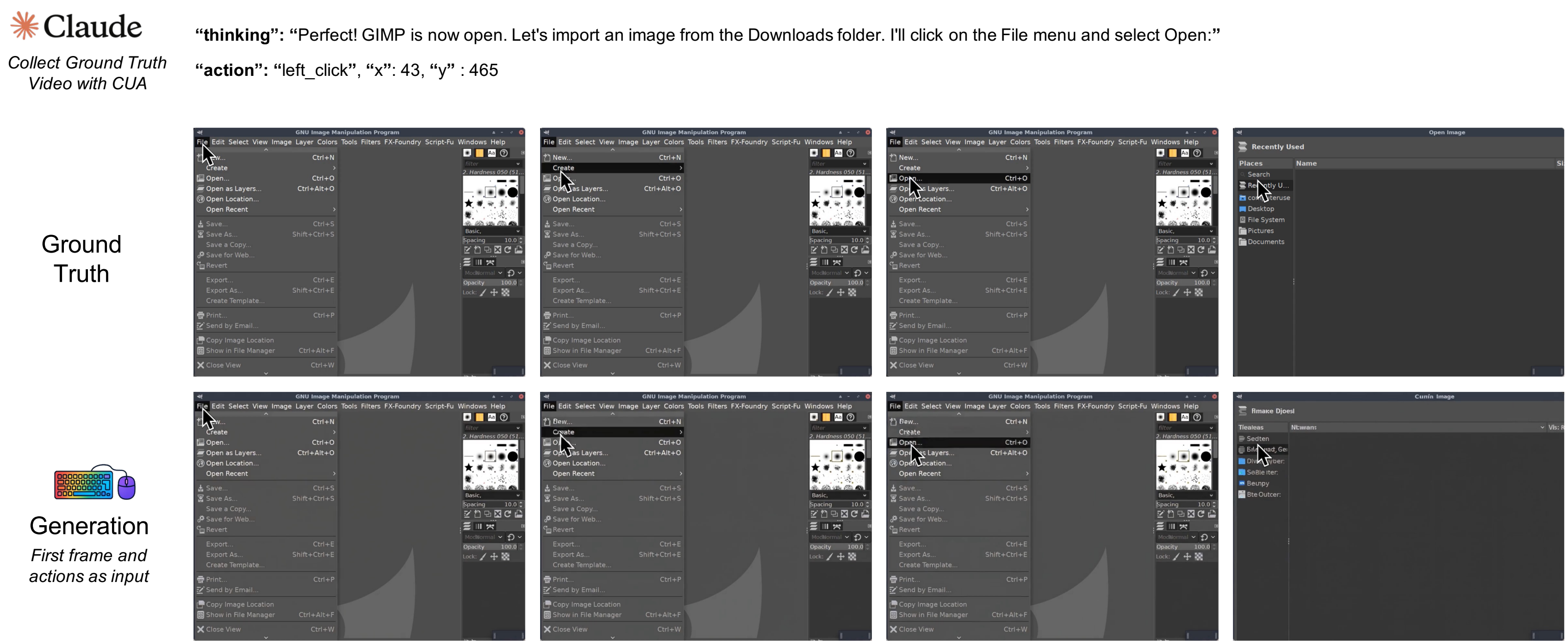


Figure 28: GUIWorld visualization sample (7).

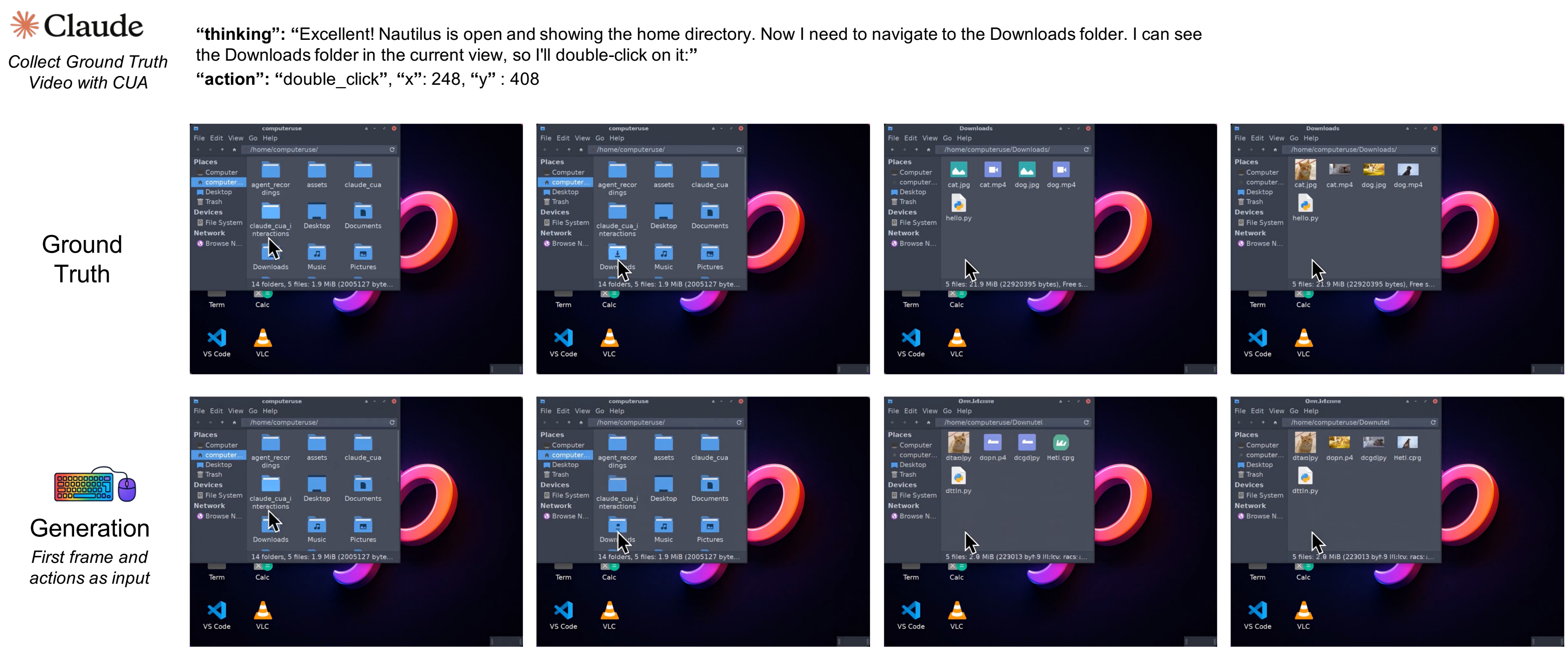


Figure 29: GUIWorld visualization sample (8).

Claude

Collect Ground Truth Video with CUA

"thinking": "Now I'll type \"512\" to change the width:"

"action": "type", "text": "512"

"thinking": "I can see that I've set the width to 512 pixels. Since the \"chain\" link icon between width and height appears to be active, the height should automatically change to maintain the aspect ratio. I'll click the \"Scale\" button to apply the resize:

"action": "left_click", "x": 723, "y": 616

Ground Truth

Generation

First frame and actions as input

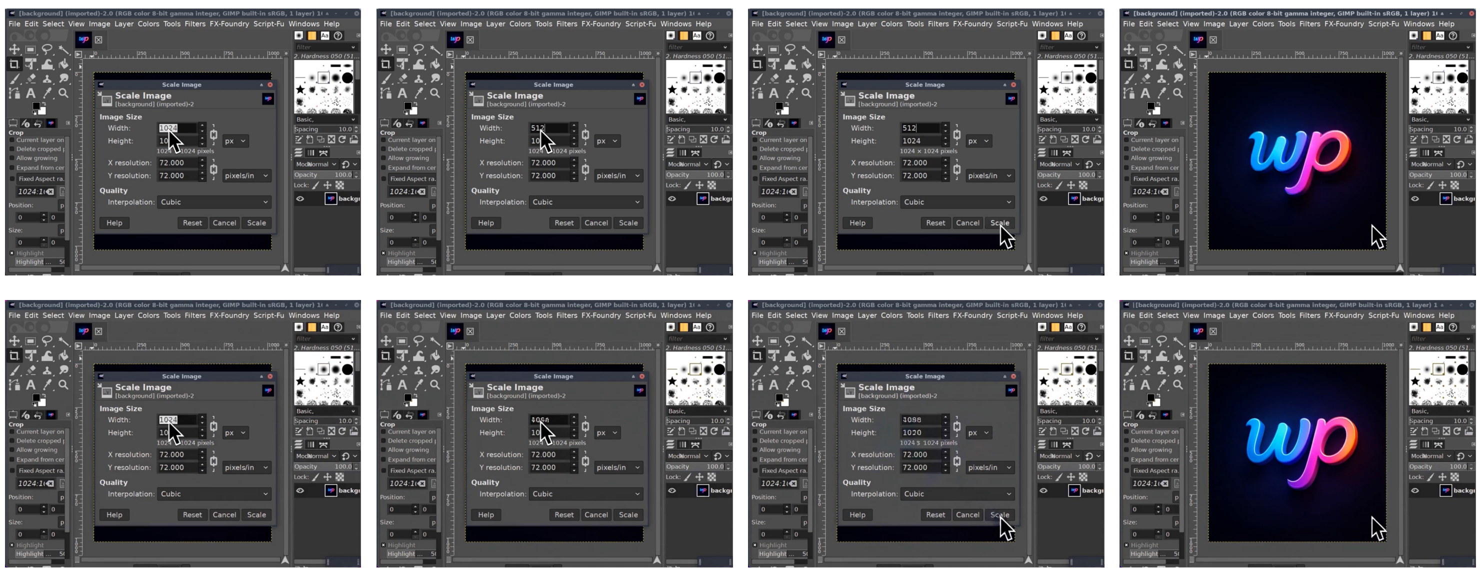


Figure 30: GUIWorld visualization sample (9).

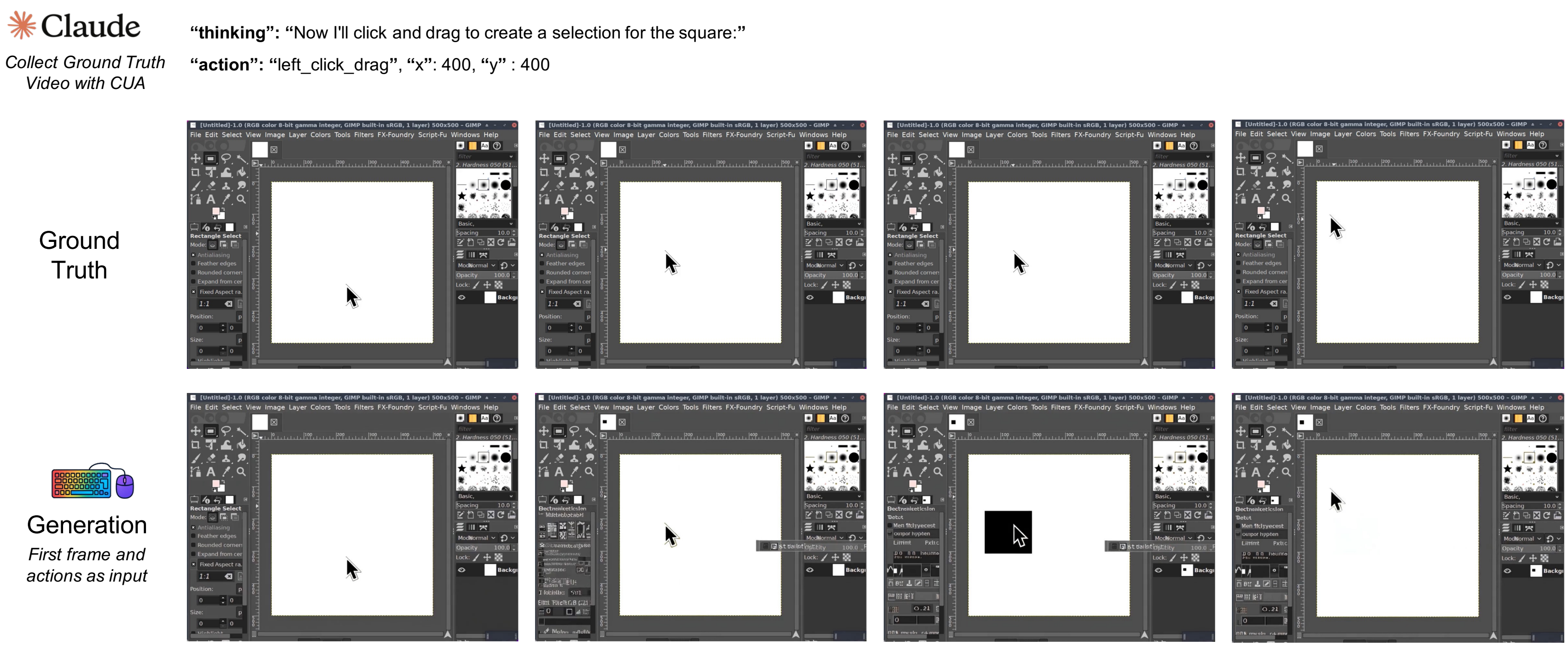


Figure 31: GUIWorld visualization sample (10).

Claude

Collect Ground Truth Video with CUA

“thinking”: “I can see the Firefox icon on the desktop. Now I'll click on it to open Firefox:”
“action”: “double_click”, “x”: 224, “y”: 70
“thinking”: “I see Firefox has opened with a welcome page. Now I'll click on the address bar and navigate to Wikipedia:”
“action”: “double_click”, “x”: 487, “y”: 81

Ground Truth

Generation

First frame and actions as input

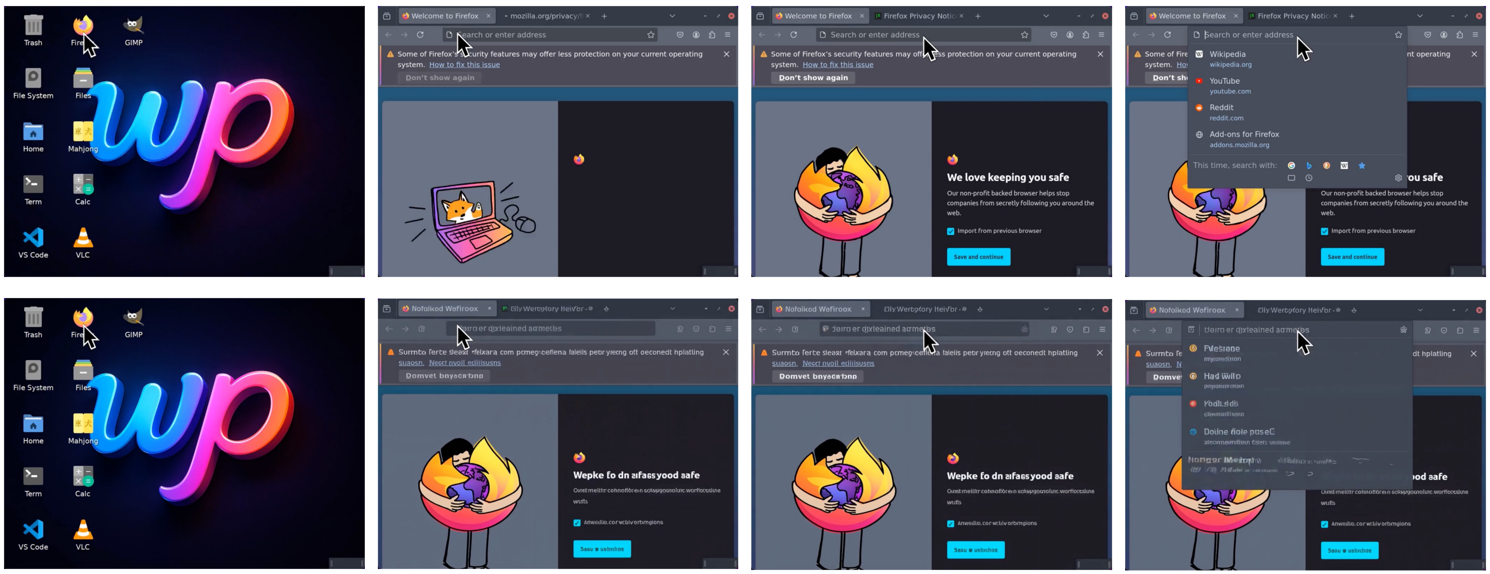


Figure 32: GUIWorld visualization sample (11).

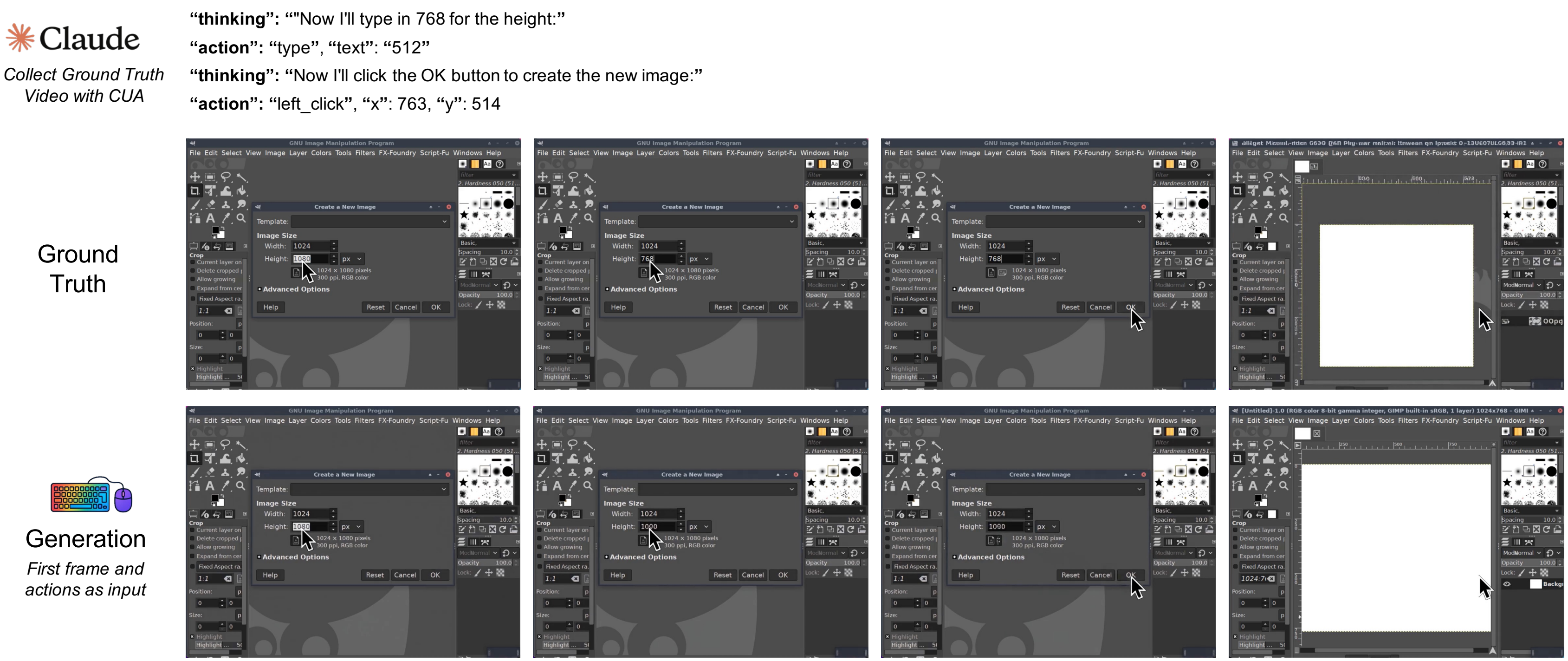


Figure 33: GUIWorld visualization sample (12).

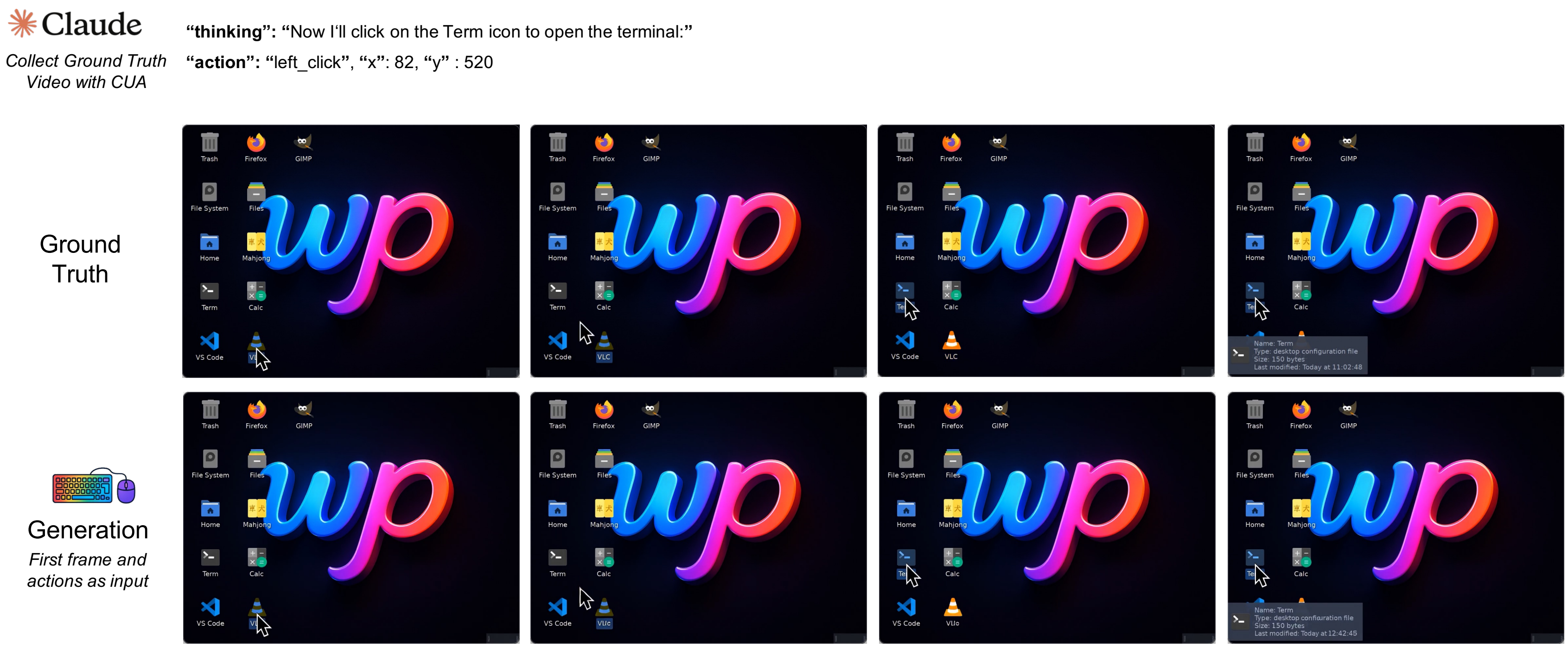


Figure 34: GUIWorld visualization sample (13).

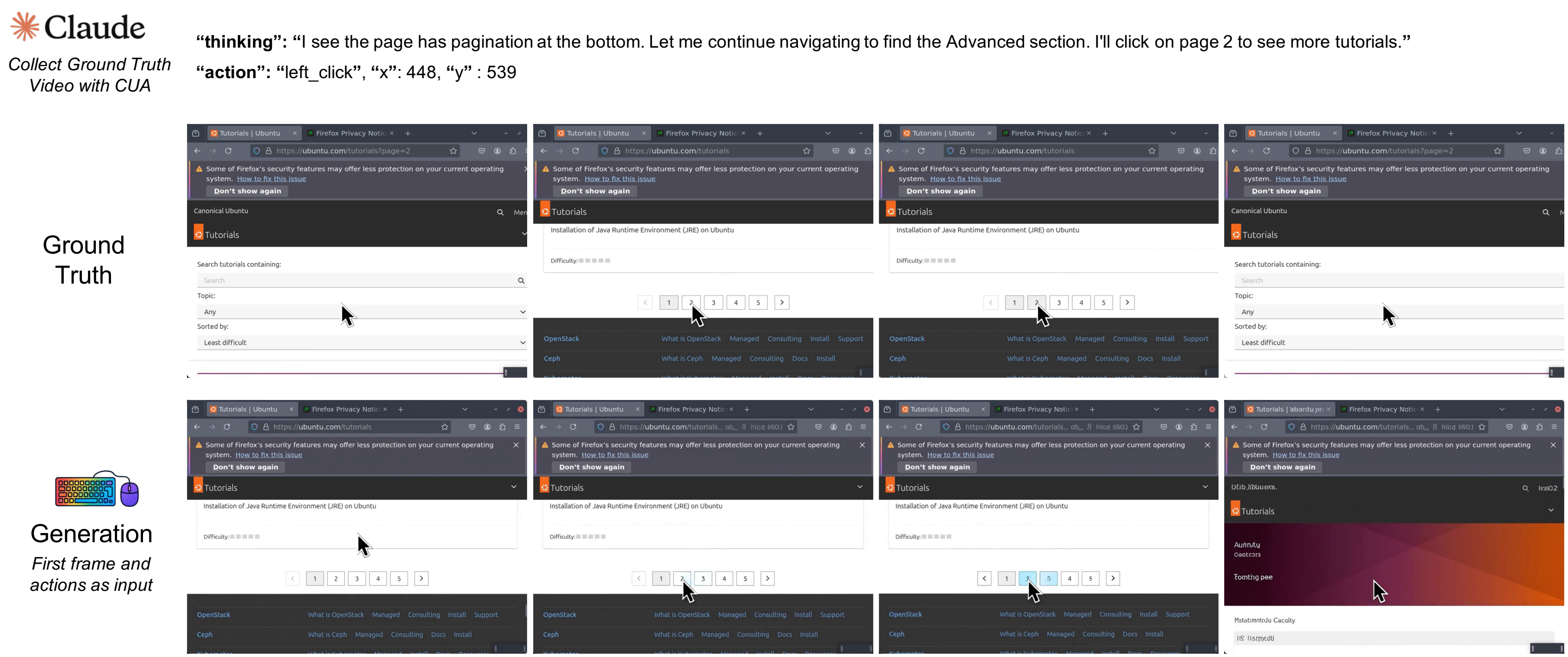


Figure 35: GUIWorld visualization sample (14).

B Toward Completely Neural Computers

B.1 From Neural Computers to Completely Neural Computers

Section Overview We summarize what the prototype Neural Computers (NCs) in this paper can reliably do today, and identify the bottlenecks that prevent general-purpose use. We then contrast NCs with General Computer Control (GCC) (Tan et al., 2024) to clarify what is uniquely enabled by a unified neural runtime. Finally, we define Completely Neural Computers (CNCs) and outline a roadmap toward them.

Current Status of NCs Our CLI and GUI-based neural computers demonstrate that a learned latent runtime can implement components of the classical stack with measurable interface fidelity. In terminal environments, character-level OCR accuracy reaches 0.54 (Table 2), while desktop cursor control achieves 98.7% accuracy with explicit visual supervision (Table 6). In GUIWorld, 110 hours of goal-directed trajectories outperforms approximately $1{,}400$ hours of random exploration (Table 5), underscoring the role of aligned supervision. These results suggest that existing technology supports reliable I/O and short-horizon control: current NCs can manage low-level interface primitives with moderate training costs given high-quality datasets.

However, to become general-purpose systems, NCs must go beyond basic I/O and short-term execution. At a minimum, NCs must support universal expressiveness (Turing et al., 1936; Siegelmann & Sontag, 1992), universal programmability (Von Neumann, 1993; Wilkes, 1981), and long-term consistency (Queloz, 2025). While sequential neural architectures can be Turing complete (Siegelmann & Sontag, 1992; Pérez et al., 2021), turning a specific instance into a reliably programmable model remains challenging. Maintaining consistency across long horizons remains an open problem in neural systems (Kirkpatrick et al., 2017; Calanzone et al., 2025).

Fundamental Architectural Differences Between NCs and GCC NCs and GCC diverge in their computational architecture. In GCC, a foundation model operates a classical computer through low-bandwidth I/O (e.g., mouse actions and screen frames), forcing both sides to communicate through compressed commands and observations. Each component only has direct access to its own internal state. In contrast, NCs unify the execution pipeline within a neural representation, which can reduce duplicated computation across components and lower synchronization overhead.

The Unique Architectural Benefits of NCs and Definition of CNC A unified neural architecture can, in principle, expose interactions that are difficult in GCC. We group these potential benefits into three categories:

1. **Shared computation:** Unlike GCC, which runs the foundation model and the classical computer separately, NCs unify computation, making resources available throughout the process. Typically, GCC operates where the classical computer has far less compute than the foundation model. An NC instance (fixed weights) could potentially be programmed to implement and execute new AI functions that require massive parallel computation.
2. **Shared information:** GCC relies on a low-bandwidth interface between the model and the computer. This bottleneck prevents the model from directly accessing large on-machine data (e.g., files), and prevents the classical runtime from accessing dense internal model states. NCs avoid this hard separation, allowing in-context-developed functions to communicate with the NC via learned semantics and exchange information at higher bandwidth.
3. **Application development via numerical optimization:** Because the memory state of an NC is continuous, one can frame some forms of "program synthesis" as optimizing that state under an objective. When the objective is differentiable through the model, first-order methods such as Adam (Kinga et al., 2015) apply; when it is not, gradient-free methods such as Natural Evolution Strategies (Wierstra et al., 2014) provide an alternative.

Finally, we define a **Completely Neural Computer (CNC)** as a Neural Computer instance that is (1) Turing complete, (2) universally programmable, (3) consistent in its output unless explicitly reprogrammed, and (4) behaviorally exhibits the unique architectural benefits of shared computation, shared information, and a numerically optimizable memory state.

B.2 A ROADMAP TOWARDS CNC

We frame the CNC transition as answering six scientific questions corresponding to its defining properties.

Turing completeness A Neural Computer (NC) instance (a specific architecture with fixed weights) defines a class of computational models in which each model corresponds to at least one memory state instance. To demonstrate that an NC instance is Turing complete, one must show it implements a Universal Turing Machine (UTM). In practice, this means that for any given Turing machine, there exists an initial memory state that makes the NC emulate that machine exactly: it must produce identical outputs for halting inputs and execute indefinitely otherwise.

For an NC instance to achieve universality, two elements are essential:

1. Iteration and Recursion: This is naturally facilitated by the recurrent architecture inherent to NCs.
2. Unbounded Effective Memory: An NC instance has an unbounded effective memory if there are infinitely many possible memory state instances such that each corresponds to a unique Turing machine.

Universal programmability An NC is universally programmable if, for each Turing machine, there exists an input sequence such that the NC realizes a new memory state representing that machine. This is likely to be achieved through compositional neural routines and partially persistent memory to store them. Progress can be tracked by routine reuse and compositional generalization.

Consistent output through runtime A CNC must preserve its function unless instructed to change it. For each memory state, there must be a non-empty set of inputs that executes the CNC without changing its pure function. Notice that since CNCs are Turing complete, for each possible input, there must be at least one memory state instance where this invariance does not hold. Thus, the set of inputs that preserve the CNC's function must be memory state dependent. We hypothesize that gating mechanisms, such as those in LSTM (Hochreiter & Schmidhuber, 1997), are effective in achieving such memory-state-conditioned invariance.

Shared computation CNCs leverage computational benefits to develop cognitive functions in context, closing the loop from neural execution to neural creation. Furthermore, similar to current AI's primitive self-improvement loop through recursive code generation, CNCs generalize this into a self-improvement cascade. CNCs emit reusable neural artifacts that can be used to generate future neural artifacts. Progress can be tracked by the reusability of cognitive functions emitted by the NC that were not present in its initial state and by whether self-improvement occurs.

Shared information In a GCC architecture, information transfer is restricted to high-level, human-interpretable interfaces: a literal "keyhole" of pixels and keystrokes. In contrast, the CNC architecture facilitates shared information by allowing neural processes to access the internal state of the environment and vice versa without serialization. A CNC achieves this when it can process and manipulate large-scale data structures (e.g., dense vector databases or raw telemetry at scale) without the latency or resolution loss associated with traditional I/O.

Validation of shared information requires demonstrating that an in-context developed function can query the NC's latent memory space at a bandwidth exceeding that of a standard visual/textual interface. Progress can be measured by the latent-throughput ratio: the amount of relevant information exchanged between a task-specific routine and the system state per computational step, compared to the same task performed over a simulated CLI or GUI on classical computers.

Application development via numerical optimization The final milestone toward CNC realization is the transition from manual "programming" to gradient-based state synthesis. Because the memory of a CNC is a continuous numerical manifold, finding a memory state that performs a specific task can be framed as an optimization problem. A system exhibits this property when a user can define a loss function (e.g., "minimize the error in this mathematical proof") and use numerical solvers to directly update the NC's memory state to satisfy that objective.

Unlike GCC, where optimization requires discrete code generation and a "compile-and-test" loop, the CNC allows for differentiable programming of the computer itself (Innes et al., 2019). Success is measured by the convergence efficiency of numerical solvers in finding valid program states compared to the success rate of combinatorial search (i.e., LLM-based code generation (Hong et al., 2023)).

B.3 Additional Thoughts Inspired by This Research

Video models as a pragmatic prototype substrate We build our prototypes on state-of-the-art video models because they currently provide the simplest path to an end-to-end learned latent runtime that jointly models pixels, dynamics, and action-conditioned control. This choice is pragmatic rather than fundamental. In our experiments, symbolic and algorithmic reasoning in terminal settings remains inconsistent for most strong video models, and even simple arithmetic can fail (Table 3). Sora2 is a notable exception in our probe, achieving 71% arithmetic accuracy, suggesting that some terminal symbolic reasoning is already possible in modern video generators. At the same time, we do not claim that video models cannot reason more broadly: recent work reports that video models can act as zero-shot learners and reasoners in naturalistic settings (Wiedemer et al., 2025). We expect reasoning capabilities to improve quickly with continued progress in video modeling, but our results suggest that CNC-level reliability will likely require additional architectural and training ingredients beyond scaling today's video generators.

A hypothesis: machine-native neural architectures We emphasize that the following is a conjecture rather than a conclusion drawn from our experiments. Closing the reasoning gap may not require designing neural networks that more closely mimic animal cognition or the human brain. Many influential architectures, including convolutional networks (Fukushima, 1980) and Transformers (Vaswani et al., 2017), are highly engineered systems, but their core inductive biases remain strongly influenced by biological perception and attention. These models primarily rely on continuous, distributed representations, in which reasoning behavior emerges implicitly from large-scale training. We hypothesize that CNCs may instead benefit from designs that are explicitly machine-native. Rather than relying on biological inspiration, we ask whether it is possible to design neural networks that directly reflect the computational structure of machines, akin to how traditional computers are built from circuits. Although such designs may be sparse, they could offer extremely strong interpretability, as well as stable reasoning and rendering capabilities. This approach would focus on developing discrete operations, compositional structures, and verifiable computations that harmonize with the principles of machine architecture, contrasting with the current trend of emergent reasoning in models like video generators.

C Explorations: Alternative Data Sources and Online Interaction

Beyond the data collection pipelines used in the main text and Appendix E, we explored alternative data sources for CLI modeling. We did not incorporate them into the final pipeline, but the trials yielded useful insights and suggest directions for future work as tooling and data quality improve.

C.1 Web video extraction

We tested mining terminal training data from web-scale screen video corpora. We used OCR and layout detectors to locate terminal regions and estimate text content and timestamps. We ultimately did not adopt this route for two reasons. First, privacy and copyright are central. Screen recordings often contain personal identifiers (usernames, emails, file paths, chat content) and come with licensing constraints that are difficult to verify at scale. Second, even when rights and privacy constraints are satisfied, cleaning the resulting data is substantially more complex than it appears (Fig. 36). Typical failure modes include (i) uncontrolled content such as faces/hands, picture-in-picture overlays, and unrelated desktop activity; (ii) domain shift across operating systems, themes, fonts, resolutions, and window managers; and (iii) quality factors such as compression artifacts, variable frame rates, zoom/crop edits, and inconsistent capture pipelines. These factors degrade OCR and temporal alignment.

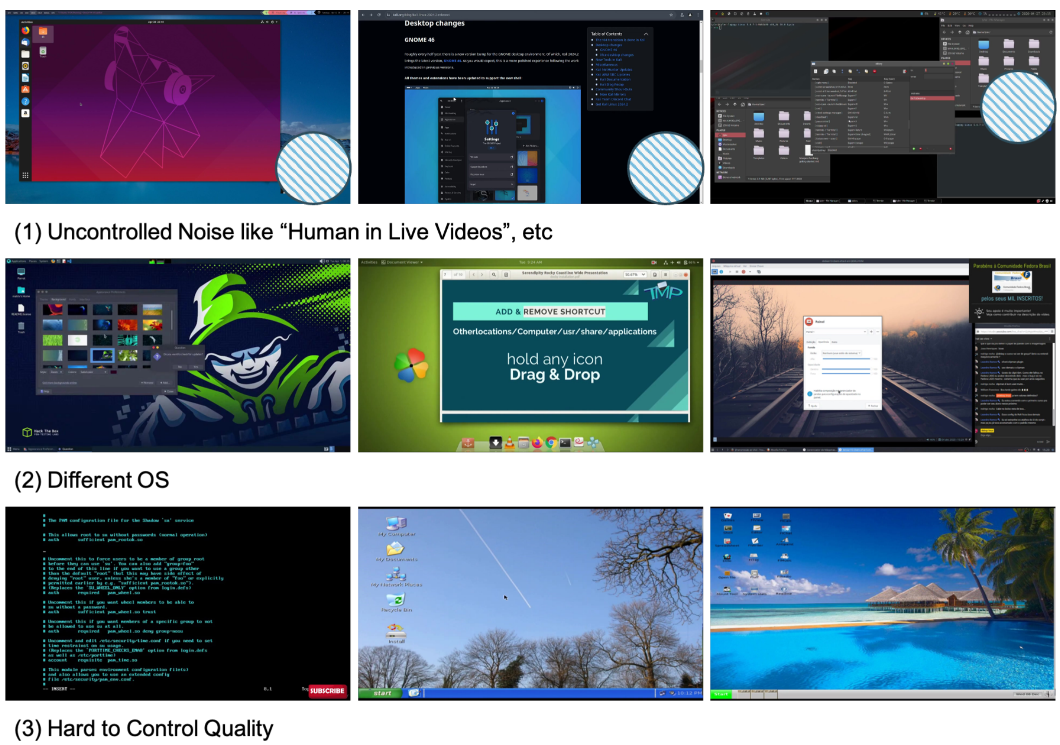


Figure 36: **Common issues in web-scale screen video mining.** Web videos mix uncontrolled content, heterogeneous OS/UI configurations, and inconsistent capture quality. This makes terminal localization, OCR, and time alignment unreliable without heavy filtering and sanitization.

Despite these challenges, web video mining may be a key lever for scaling interface experience over the long term. In this work, however, the cost–quality trade-off was unfavorable. Our setting benefits disproportionately from clean, temporally aligned text and interaction signals. Building a high-precision web filter also requires substantial upfront investment. This includes rights-cleared sourcing or licensing, privacy review and redaction, and large-scale multimodal filtering/OCR pipelines that often rely on paid APIs. Given these constraints and our emphasis on high-quality supervision, we prioritized curated CLIGen sources. Future efforts that invest in rights-respecting acquisition and stronger automated filtering could unlock web-scale data as a complementary scaling axis.

C.2 Online environment interaction

We prototyped an agentic online data and learning pipeline that separates a training/control plane from an execution/environment plane (Fig. 37). In the environment plane, an isolated container runs a live shell together with an agent and recorder exposed via a narrow port-based interface. The agent is driven by the current model checkpoint (optionally assisted by an LLM planner) and issues commands and control actions. The recorder captures synchronized terminal renders, structured terminal state when available (e.g., buffer/text), and action traces. Trajectories are streamed to the control plane for storage and optional online training, which updates model weights and pushes refreshed checkpoints back to the agent.

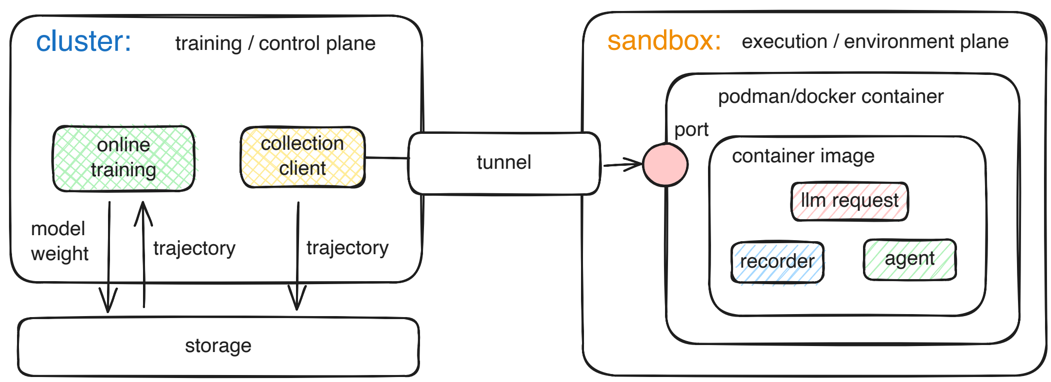


Figure 37: **Agentic online interaction pipeline (early exploration).** A control plane collects trajectories and can optionally update model checkpoints online. A sandboxed environment plane executes an agent in an isolated container and records synchronized state/action traces.

Concretely, the environment plane exposes a minimal "step/reset" interface over a port. It returns multimodal observations that can be logged deterministically (rendered screenshots plus structured state when available). The agent emits structured actions (typed command text, key/mouse events when applicable, and timing). This separation makes rollouts auditable and lets the control plane scale collection independently from training.

We explored this setup because closed-loop interaction can induce a natural curriculum by continually sampling the boundary of the current policy. It can also surface rare and safety-critical failure modes that do not appear in offline logs. It supports targeted data collection (e.g., focusing on specific tools, error recovery, or long-horizon tasks). In principle, it also offers a direct path to scaling *experience* rather than only scaling static demonstrations.

Early trials showed promise, but the end-to-end system introduced substantial engineering and safety overhead. This includes strong isolation of untrusted code execution, monitoring and abuse prevention, and deterministic resets and environment control. It also requires robust recording and serialization across heterogeneous environments. Under time and cost constraints, we therefore prioritized controlled CLIGen data for the main experiments.

Despite not being used in the final pipeline, we consider this design a general template for future work. It enables scalable multi-environment rollout collection and consistent provenance and storage of trajectories. It also supports interchangeable learning algorithms (e.g., behavior cloning, preference-based learning, or other online updates). Execution remains sandboxed and auditable.

D Implementation Details

This section summarizes training recipes. Dataset collection, preprocessing, and evaluation protocols are consolidated in Appendix E.

D.1 CLIGen training recipe

Training uses gradient checkpointing and applies dropout 0.1 to the prompt encoder, CLIP, and VAE modules. Optimization uses AdamW (learning rate 5×10^{-5}, weight decay 10^{-2}), `bfloat16` precision, and gradient clipping at 1.0. Training $\mathrm{NC}_{\mathrm{CLIGen}}$ on CLIGen (General) requires approximately 15,000 H100 GPU hours at batch size 1. Training on CLIGen (Clean) across both subsets requires approximately 7,000 H100 GPU hours in total.

D.2 GUIWorld training recipe

We train one model per injection mode (`external`, `contextual`, `residual`, `internal`), keeping the backbone and all non-action components fixed. Each run lasts about 64k steps using the CLI hyperparameters. We tune only the action encoder and learning-rate schedule. Training optimizes the diffusion loss together with a small temporal contrastive loss that aligns frame features with action and mouse embeddings (Appendix G). Runs use 64 GPUs for about 15 days, totaling about 23k GPU-hours per full pass.

E Datasets: Collection and Evaluation Protocols

This appendix summarizes collection, preprocessing, and evaluation details for the datasets used in the paper. For concrete examples of raw trajectory formats (asciinema `.cast` and `vhs` scripts), see Appendix F.

Data collection follows three coordinated stages. This ensures the CLIGen and GUIWorld datasets share synchronized timing, privacy guarantees, and consistent, well-documented artifacts.

Sourcing. CLIGen (General) episodes originate from public `asciinema .cast` archives. We replay traces with official tools to preserve terminal appearance (color schemes, cursor visibility, window dimensions) as recorded. CLIGen (Clean) episodes come from deterministic `vhs` scripts (package installs, REPLs, log filters) run in isolated environments. GUIWorld footage is captured via the rig in Section 3.2, pairing RGB video with low-latency pointer/key logs and optional accessibility cues (logged for analysis; not used as model inputs).

Alignment and sanitization. All modalities share a common clock. We align pointer/key events to the nearest frame, apply drift correction when needed, and drop clips with residual misalignment. Privacy filters remove terminal sessions with sensitive strings and redact GUI regions likely to contain private content. Frozen or repeated-frame recordings (capture artifacts) are discarded.

Episode packaging. Runs are windowed into fixed-length, overlapping episodes (window sizes and strides are specified in the released configs). Each shard stores RGB frames, terminal buffers or GUI metadata, serialized actions, the source tool (`asciinema`/`vhs`/GUI capture), and environment metadata. Structured fields (buffers/metadata) are used for alignment and evaluation, but are not provided to the video models as state inputs. Downstream dataloaders reconstruct batches directly. Released configs specify preprocessing and windowing so external users can rebuild the corpus.

E.1 / CLIGen data pipeline

The CLIGen (General) dataset is built from publicly available `asciinema .cast` trajectories[3]. The `asciinema` stack records and replays terminal sessions with synchronized timing and ANSI-faithful decoding. We replay each session with the official tools and render it into terminal frames, preserving palette transitions, cursor state, and terminal geometry. Frames, text buffers, and keyboard-event logs share a single monotonic clock. At render time, we normalize resolution and aspect ratio and apply a simple privacy filter to remove sensitive strings. We render sessions to GIF using `agg` and convert them to video with `ffmpeg`. We segment each recording into roughly five-second clips using content-aware splits. We temporally normalize each clip to a fixed length: shorter clips repeat the final frame, and longer clips are uniformly subsampled. The resulting 823,989 video streams (approximately 1,100 hours) are resampled to 15 FPS. Underlying buffers and logs are used to generate aligned textual descriptions with Llama 3.1 70B (Dubey et al., 2024) in three styles (semantic, regular, and detailed), which serve as prompts. As shown in Figure 2 (left), this split spans diverse real-world terminal use cases. [4]

The CLIGen (Clean) dataset is collected using the open-source `vhs`[5] toolkit. It enables repeatable terminal demonstrations and integration tests through scripted execution. Deterministic scripts drive Dockerized environments to capture cleaner, better-paced traces. We authored roughly 250k scripts. After filtering (51.21% retained), we keep two subsets. The first contains approximately 78k regular traces (package installation, log filtering, interactive REPL usage, etc.). The second contains approximately 50k Python math validation traces. Captions are derived directly from the raw `vhs` scripts for clarity. We standardize frame rendering by fixing one monospace font/size, using a consistent palette for success and error highlights, and locking resolution and theme to remove typography-related confounds. Each episode records its caption type and font settings for later slicing. Clips longer than five seconds are uniformly subsampled for training, while shorter clips repeat the

[3] `https://asciinema.org/`

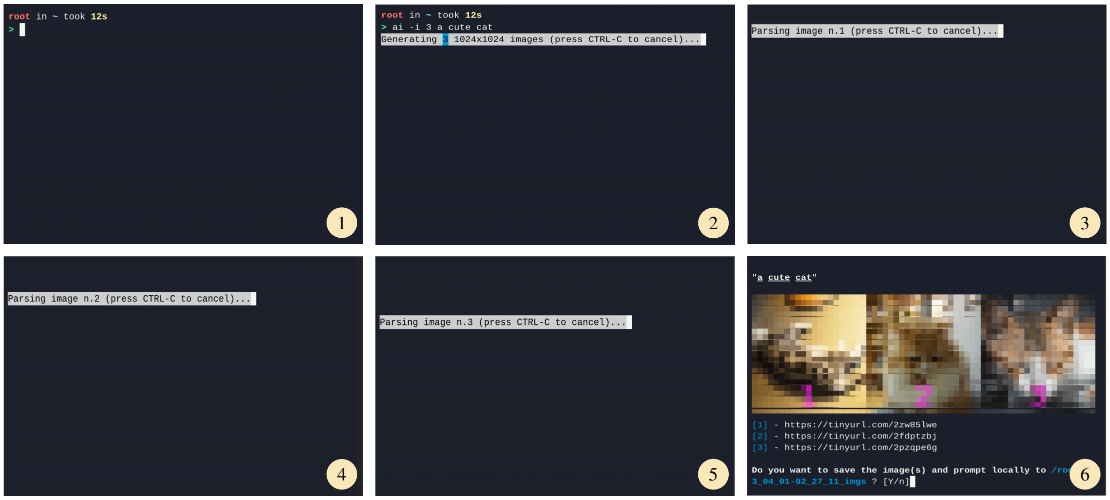
[4] Additional preprocessing details and a `.cast` example are in Section F.1, with a sample overview in Table 9.

[5] `https://github.com/charmbracelet/vhs`

final frame to normalize length. We apply the same timestamping, resolution/aspect normalization, and privacy filtering as in CLIGen (General). [6]

Table 9: $\mathrm{NC}_{\mathrm{CLIGen}}$ samples for CLIGen (General) and CLIGen (Clean).

Frames from Sample — CLIGen (General)



Semantic

A root terminal session kicks off an AI command to make three 1024x1024 cat shots, shows quick parsing for each one, then presents pixelated cat previews with numbered links and asks whether to stash them in `/root/2023_04_01-02_27_11_imgs`.

Regular

In a root shell at ~, the user runs `ai -i 3 a cute cat`, watches a green progress line announcing three 1024x1024 images, sees sequential parsing messages for images 1 through 3, and ends on a preview pane with three pixelated cat thumbnails, numbered download links, and a save prompt targeting `/root/2023_04_01-02_27_11_imgs`.

Detailed

In a dark-background terminal at the `root in ~` prompt, the user types `ai -i 3 a cute cat`. The screen prints `Generating 3 1024x1024 images (press CTRL-C to cancel)...`, shows parsing messages for images 1–3, and ends on a preview pane with three numbered thumbnails and a save prompt targeting `/root/2023_04_01-02_27_11_imgs`.

Frames from Sample — CLIGen (Clean)

Scripted caption

Type `python`; Enter; Type `values = [n*n for n in range(1, 10)]`; Enter; Type `print(values)`; Enter; Type `exit()`; Enter.

[6] Additional details are provided in Section F.2, with a representative data sample in Table 9.

E.2 Caption fields and metadata (CLIGen General)

For CLIGen (General), each replayed `.cast` fragment (example shown in Section F.1) is paired with three aligned descriptions and a compact metadata record. Table 10 summarizes the fields for clip `7****_0001`.

Table 10: Caption tiers and metadata for CLIGen (General) clip `7****_0001`.

Field	Content (abridged)
`caption`	The user, `annonomous`, logged into their account on the host `NeuralComputer`, and after some initial terminal setup and cursor movements, they started typing the command `nvim` to likely open the Neovim text editor, but the session recording ends abruptly without showing the actual command execution or any further interaction.
`caption_detailed`	At the `annonomous@NeuralComputer: $` prompt, the user types `nvim`, but before the command executes, the screen rapidly displays a colorful, pixelated animation with various RGB colors, including shades of blue, green, and purple, filling the 80x24 terminal window. The animation is briefly interrupted by a cursor blink, then the prompt returns, awaiting further input.
`caption_semantic`	The terminal displays a rapid sequence of colored text and pixel art animations, with syntax-highlighted code and prompts appearing in a mix of blue, green, and white hues, as the user types commands and the system responds with scrolling output, error messages, and success indicators.
`data_info.version`	`2` (`asciinema` v2 header)
`data_info.size`	`width=80`, `height=24`; original/target/scaled sizes all match, padding `[0,0]`.
`data_info.env`	`SHELL=/bin/bash`, `TERM=xterm-256color`.
`meta.videogen`	Automatically derived stats: visual complexity `2897.7`, interaction density `1.0`, color usage `518`, screen clears `6`, event rate `7.8` events/s, typing rhythm (avg interval `0.13`s, variance `0.34`), and `38` visual-change events.
`metadata`	Source recording on Asciinema: id `7****`, title ``` ``aneo.nvim demo'' ```, author `annonomous`, creation time `2025-05-13T23:37:53Z`, URLs for the page and raw `.cast`.

These fields make each CLIGen (General) clip self-contained. The three caption tiers provide prompts at different levels of detail. `data_info` and `metadata` preserve the structure needed to rebuild terminal geometry, environment, and source from the raw `.cast`.

A rough analysis of content categories shows the dataset covers diverse terminal usage. It includes general terminal operations (610,331 clips), file operations (81,783), programming activities (66,430), system administration (42,668), and text editing (22,777). This distribution reflects typical command-line workflows across different user tasks.

E.3 OCR evaluation protocol (CLIGen Clean)

For CLIGen (Clean), OCR-based metrics evaluate how closely generated terminal videos match reference renderings derived from the ground-truth buffers in text space rather than pixels. Each sample consists of a generated video and its paired reference video (matched by clip ID). We keep only IDs where both videos are present.

From each paired video we use at most $K{=}5$ frames. Let T_{gen} and T_{gt} be the frame counts of the generated and reference videos. We set $T = \min(T_{\text{gen}}, T_{\text{gt}})$ and uniformly sample K indices in $[0, T{-}1]$ so that evaluation frames are spread across the trajectory. For every sampled index we read the corresponding frame from both videos.

Each frame is converted to RGB and passed to Tesseract OCR. The resulting string is split into lines, leading and trailing whitespace is stripped, and internal whitespace is normalized by collapsing runs of spaces. Empty lines are dropped. We keep case and punctuation intact so that commands, paths, and symbols remain visible. This gives an ordered list of normalized lines for the ground-truth frame $(g_1, \ldots, g_{N_g})$ and the generated frame $(p_1, \ldots, p_{N_p})$.

Character accuracy pools all lines into a single multi-line string for each side and measures normalized edit distance. Let s and t be the concatenated ground-truth and generated texts and $d(s,t)$ their Levenshtein distance (insert/delete/replace cost 1). If both s and t are empty we set char_acc $= 1$; if only s is empty we set char_acc $= 0$. Otherwise,

$$\text{char\_acc} = \max\Big(0,\ 1 - \frac{d(s,t)}{\max(|s|, 1)}\Big),$$

Extra or missing characters are penalized symmetrically. Frame-level scores are averaged over the K sampled frames to yield a per-video character accuracy, and group-level scores report the mean over videos.

Exact-line accuracy treats lines as position-sensitive units and reports a recall over ground-truth lines. For a given frame, we compare line g_i to p_i at the same index. A line is counted as correct only if $i \le N_p$ and $p_i = g_i$; lines that appear in the wrong position do not count. If both lists are empty we set exact_line_acc $= 1$; if the ground-truth list is empty but the generated list is not, we set exact_line_acc $= 0$. Otherwise,

$$\text{exact\_line\_acc} = \frac{1}{N_g} \sum_{i=1}^{N_g} \mathbf{1}[i \le N_p \wedge p_i = g_i].$$

As with character accuracy, frame scores are averaged over the K sampled frames to obtain a per-video score and then averaged over videos for the reported aggregate. Unless otherwise noted, OCR results in the main text use 1,000 randomly sampled video pairs (five frame pairs per video). Together, these two metrics stress both fine-grained text fidelity and line-ordered terminal state reconstruction.

E.4 GUIWorld data pipeline

Preprocessing is implemented in the data loader in two stages. First, we normalize each recording to a fixed resolution and frame rate. This produces tensors for RGB video, per-frame cursor coordinates, and mouse/keyboard event traces (in both raw-action and meta-action views). Second, we render an SVG cursor at each logged position to produce per-frame masks and cursor-only reference frames. The first reference frame contains the full desktop with a unit mask. Later references paste only the cursor over a neutral background, with a mask restricted to arrow pixels. After VAE encoding, these references become latent slots that pin down the static GUI layout at t=0. For t>0, they supervise only a small patch around the cursor and leave the rest of the frame unconstrained. We drop clips without valid cursor or action traces to keep supervision consistent.

Cursor coordinate normalization. Raw mouse logs provide per-frame cursor positions in screen coordinates $(x_{\text{screen}}, y_{\text{screen}})$ at GUI resolution. We align these to sampled video frames and map them through the same letterbox transform applied to the RGB stream. Given source and target resolutions $(w_{\text{src}}, h_{\text{src}})$ and $(w_{\text{dst}}, h_{\text{dst}})$, we compute a uniform scale s and padding offsets (p_x, p_y). We then map each coordinate to normalized positions $(x_t, y_t) \in [0,1]^2$ as

$$x_t = \frac{s\, x_{\text{screen},t} + p_x}{w_{\text{dst}} - 1}, \qquad y_t = \frac{s\, y_{\text{screen},t} + p_y}{h_{\text{dst}} - 1}.$$

Stacking these over time yields a trajectory tensor `mouse_trajectories` $\subset [0,1]^{T\times 2}$.

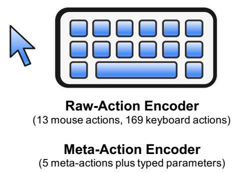
Fourier trajectory features (optional). To provide a stronger continuous conditioning signal, we optionally encode the normalized trajectories with random Fourier features. We map coordinates to $[-1,1]^2$ and project them through a fixed Gaussian matrix (followed by sine/cosine). A small MLP produces per-frame embeddings, which are aggregated into lag-aware windows aligned with the VAE stride. The resulting latent-aligned mouse features can condition the action modules and participate in the temporal contrastive loss (see Appendix G).

Table 11: Cursor/action statistics.

Split	Avg. cursor speed (px/frame)	Actions / sec
Random Slow	1.51	1.58
Random Fast	195.15	4.18
CUA (supervised)	3.79	0.10

E.5 Encoding Examples (Raw-Action vs. Meta-Action)

Table 12: Encoding examples for raw-action and meta-action encoders.



User intent	**Raw-action encoder (event stream)**	**Meta-action encoder (API-like slot)**
`ls -l`	• Key events per character (e.g., `l`, `s`, `<space>`, `-`, `l`) • Activates entries in a 169-d keyboard multi-hot • No explicit command semantics; inferred from sequence	• `type: KeyboardType` • `text: "ls -l"` • Encoded by shared text encoder
`ctrl+v`	• Separate keydown/keyup events for `ctrl` and `v` • Activated in multi-hot (or shortcut entry)	• `type: Shortcut` • `id: ctrl+v` • Embedded via shortcut table

We collect GUI data at 15 FPS with synchronized cursor and keyboard logs. The dataset includes two styles of random interaction: “Random Slow” and “Random Fast”, plus a smaller set of supervised trajectories from Claude CUA (Rivard et al., 2025). Random Slow (approximately 1,000 hours) contains longer pauses, idle gaps, and deliberate cursor movements, which can expose cursor drift after extended inactivity. Random Fast (approximately 400 hours) features denser cursor motion and typing bursts, stressing acceleration dynamics and hover timing.

We also include approximately 110 hours of supervised trajectories generated using Claude CUA[7]. These goal-directed traces provide higher-signal action–response pairs without overwhelming the exploration data. Table 11 summarizes cursor and action statistics across splits; CUA is slowest due to latency introduced by Claude’s tool API between successive actions.

All GUI data is collected inside an Ubuntu 22.04 container running XFCE4 (Arc-Dark theme, Papirus icons) on a fixed 1024×768 virtual display at 15 FPS. We render the display with Xvfb and interact through a VNC/noVNC stack. The desktop pins a small open-source app set to launchers. It includes Firefox ESR, GIMP, VLC, VS Code, Calculator, Terminal, the file manager, and the Mahjongg game, matching the environment shown in our recordings. Screen capture uses `mss` and `ffmpeg` with cursor overlays, and actions are replayed and logged via `xdotool`. We keep the recorded discontinuities and interface latency intact rather than smoothing them. For modeling, we store both raw-action and meta-action views. This lets the same encoder and loss stack train on either. [8]

E.6 Evaluation metrics and protocol (GUIWorld)

We report both global video metrics and action-driven metrics that focus on post-interaction frames. The evaluation code used for these metrics is implemented in our GUIWorld evaluation suite.

Global FVD_{all}/SSIM_{all}/$\text{LPIPS}_{\text{all}}$ We decode paired generated/ground-truth videos into RGB frames with temporal subsampling and resizing (`fps`=3, `size`=256, and `max_seconds`=5 by default). SSIM_{all} is computed using `torchmetrics` on frame tensors normalized to $[0, 1]$ and averaged over frames. $\text{LPIPS}_{\text{all}}$ uses the AlexNet backbone on frames normalized to $[-1, 1]$ and is averaged over frames. FVD_{all} is computed in an `r3d18` embedding space (`prelogits` by default). We extract features from fixed-length clips (16 frames at 112×112 after uniform subsampling/padding). We compute the Fréchet distance between the generated and reference feature distributions.

Action-driven metrics (SSIM_{+15}, LPIPS_{+15}, FVD_{+15}) For each paired rollout, we load recorded action timestamps (from JSON/CSV logs) and map them to frame indices at the evaluation frame rate. We skip the action frame itself (`action_start_offset`=1) and evaluate the next k=15

[7] `https://platform.claude.com/docs/en/agents-and-tools/tool-use/computer-use-tool`

[8] Conversion details and alignment quality appear in Appendix G.

frames after each action. Concretely, for each action frame index f, we select the post-action set $\{f+1, \ldots, f+k\}$. We take the union over all actions in the clip and compute SSIM_{+15} /LPIPS$_{+15}$ on the selected frame pairs. For action-driven FVD_{+15}, we build an *after-action clip* per video by concatenating the selected post-action frames. We then uniformly subsample/pad to 16 frames and compute the Fréchet distance in the same `r3d18` feature space.

F CLIGEN: CLI TRAJECTORY FORMATS

This appendix provides concrete format examples for the two CLI sources referenced throughout the data pipeline (Appendix E). We show asciinema `.cast` trajectories for CLIGen (General) and `vhs` scripts for CLIGen (Clean).

F.1 ASCIINEMA (.CAST) EXAMPLE

The header line stores the recording config (version, terminal size, timestamp, env). Each following triple is `[time, "o", "payload"]`, where `"o"` indicates screen output. The payload contains the terminal text with color codes at that timestamp.

```
{"version": 2, "width": 80, "height": 24,
 "timestamp": 1747177906,
 "env": {"SHELL": "/bin/bash", "TERM": "xterm-256color"}}
[0.082492, "o", "\u001b[H\u001b[2J\u001b[3J"]
[0.950038, "o", "\u001b[38;2;16;131;236m\u001b[39m\r\n..."]
[0.950733, "o", "\u001b[38;2;6;156;220m ... \u001b[38;2;1;195;187m■"]
```

F.2 VHS SCRIPT EXAMPLE

```
# ---- VHS documentation start (DO NOT CHANGE) ----
# Require:
#   Require <string>
# Sleep:
#   Sleep <time>
# Type:
#   Type[@<time>] "<characters>"
# Keys:
#   Escape[@<time>] [number]
#   Backspace[@<time>] [number]
#   Delete[@<time>] [number]
#   Insert[@<time>] [number]
#   Down[@<time>] [number]
#   Enter[@<time>] [number]
#   Space[@<time>] [number]
#   Tab[@<time>] [number]
#   Left[@<time>] [number]
#   Right[@<time>] [number]
#   Up[@<time>] [number]
#   PageUp[@<time>] [number]
#   PageDown[@<time>] [number]
#   ctrl+<key>
# Display:
#   Hide
#   Show
# ---- VHS documentation end (DO NOT CHANGE) ----

# ID: vhs_example
# INSTRUCTION: Runs `uname -s` repeatedly as a basic shell exercise, then hides the
# LEVEL: 1
# EVENTS: 23
# VISUAL_COMPLEXITY: 45

# ---- Theme setting start (DO NOT CHANGE) ----
Output vhs_example.mp4

Set Shell "bash"
```

```
Set Theme {
  "name": "Catppuccin Mocha (Pure White, Warm Pink Cursor)",
  "background": "#1e1e2e",
  "foreground": "#ffffff",
  "black": "#45475a",
  "red": "#f38ba8",
  "green": "#a6e3a1",
  "yellow": "#f9e2af",
  "blue": "#89b4fa",
  "purple": "#cba6f7",
  "cyan": "#94e2d5",
  "white": "#ffffff",
  "brightBlack": "#585b70",
  "brightRed": "#f38ba8",
  "brightGreen": "#a6e3a1",
  "brightYellow": "#f9e2af",
  "brightBlue": "#89b4fa",
  "brightPurple": "#cba6f7",
  "brightCyan": "#89dceb",
  "brightWhite": "#ffffff",
  "cursor": "#f5c2e7",
  "cursorAccent": "#1e1e2e",
  "selectionBackground": "#585b70"
}

Set FontSize 40
Set Width 1600
Set Height 900
Set TypingSpeed 300ms
Set PlaybackSpeed 1
Set Margin 28
Set MarginFill "#0091FF"
Set BorderRadius 25
Set Padding 18
Set LineHeight 1.2
Set LetterSpacing 0.8
# ---- Theme setting end (DO NOT CHANGE) ----

Sleep 800ms
Sleep 180ms
Type "uname -s"
Sleep 120ms
Enter
Sleep 400ms
Type "uname -s"
Sleep 120ms
Enter
Sleep 400ms
Type "uname -s"
Sleep 120ms
Enter
Sleep 400ms
Type "uname -s"
Sleep 120ms
Enter
Sleep 400ms
Type "uname -s"
Sleep 120ms
Enter
```

```
Sleep 400ms
Sleep 400ms
Sleep 600ms
Hide
```

G GUIWORLD: ACTION REPRESENTATION, CONDITIONING AND MORE EXPERIMENTS

This appendix provides additional GUIWorld visualization samples and expands on the action representation and conditioning used in Section 3.2. Evaluation metrics and protocols are summarized in Appendix E.6.

G.1 ACTION SCHEMA

$\text{NC}_{\text{GUIWorld}}$ represents actions as a structured stream, enabling the NC to condition on both cursor movements and key presses.

At each timestep, we log absolute cursor coordinates, button up/down transitions, scroll deltas, and keyboard events. Keyboard inputs are split into two types: typed characters (e.g., `ls -l`) and shortcut-style chords (e.g., `ctrl+v`). We also track state flags such as whether a drag is currently active. This lets us represent extended interactions like click-drag or press-hold as short labeled segments rather than isolated spikes. The meta-action encoder described in Section 3.2 compresses this stream into a small typed schema. Each frame can contain up to two actions, and each action has a type (e.g., `mouse click` or `keyboard type`) plus parameters. Table 13 summarizes the types and fields. Type 0 corresponds to the absence of an action. Type 1 encodes mouse clicks and drags via button identity, click count, and a drag flag. Type 2 captures scrolls with a direction and scalar amount. Type 3 packages free-form keyboard text (such as `ls -l`) embedded by the shared text encoder. Type 4 records shortcuts such as `ctrl+v` via a small shortcut vocabulary. This representation resembles a tool API while remaining recoverable from raw logs.

Table 13: Meta-action schema for GUIWorld (per action slot).

Type id	Action	Parameter fields
0	None	–
1	Mouse Click/Drag	`button`, `click_count`, `drag_flag`
2	Mouse Scroll	`direction`, `amount`
3	Keyboard Type	`text` (e.g., `ls -l`) → shared text encoder
4	Shortcut	`shortcut_id` (e.g., `ctrl+v`)

G.2 CONDITIONING: ENCODERS AND INJECTION

The main text considers two encoders for this stream. A *raw-action encoder (v1)* keeps fine-grained mouse and key events in a multi-hot representation that closely mirrors real cursor and typing behavior. A complementary *meta-action encoder (v2)* compresses events into a small typed schema (Table 13) and embeds any free-form text with a shared text encoder. Both encoders produce per-frame action features that undergo temporal windowing and alignment (described below). These embeddings support four injection modes. `external` fuses actions at the VAE input. `contextual` mixes actions and frames as tokens. `internal` injects actions inside transformer blocks. `residual` adds lightweight action deltas to hidden states.

Injection-mode definitions (formal) Below we give compact schematic definitions for the three modes that are described with formulas in the main text. **External.** Given VAE latents $z_{1:T}$ and temporally aligned action features $u_{1:T}$, an external action module produces a residual update $\Delta z_{1:T}(u_{1:T})$ and forms modified latents

$$z'_{1:T} = z_{1:T} + \Delta z_{1:T}(u_{1:T}).$$

The diffusion backbone then operates on $z'_{1:T}$ (actions do not appear as explicit tokens inside the transformer). **Residual.** At selected transformer layers l, an auxiliary action module takes block hidden states $h^{(l)}$ together with local action/mouse features and outputs a residual update $\Delta h^{(l)}(a, \text{mouse})$. The updated hidden states are

$$\tilde{h}^{(l)} = h^{(l)} + \Delta h^{(l)}(a, \text{mouse}),$$

which are passed to the next block. **Internal.** At selected blocks, action conditioning is inserted as an additional cross-attention sub-layer inside the standard attention stack. With self-attention SA, text/reference cross-attention $\mathrm{CA}_{\text{text}}$, and action cross-attention $\mathrm{CA}_{\text{action}}$, a schematic update is

$$h' = \mathrm{FFN}\Big(h + \mathrm{CA}_{\text{text}}\big(\mathrm{SA}(h), c\big) + \mathrm{CA}_{\text{action}}(h, a)\Big).$$

G.3 Temporal alignment and attention

Temporal alignment and windows. The GUI backbone processes a compressed latent video at stride c (every c pixel frames correspond to one latent frame). For a pixel sequence of length F and latent sequence of length T, we have $F = (T-1)c + 1$. Anchor frame $a_t = t \cdot c$ marks the pixel frame corresponding to latent step t.

Mouse and keyboard logs start as per-frame features $r_f \in \mathbb{R}^D$ at the pixel rate. A windowed encoder aggregates them around each anchor over $p = c \cdot w$ frames. Here w controls the window width, and a lag ℓ accounts for GUI response delay (actions precede their visual effects). We use zero-padding outside the valid range, i.e., $\tilde{r}_f = r_f$ for $0 \le f < F$ and $\tilde{r}_f = 0$ otherwise, and form a lag-shifted window that ends at $a_t - \ell$:

$$W_{t,k} = \tilde{r}_{\,a_t-(p-1+\ell)+k}, \quad k \in \{0, \dots, p-1\}, \quad a_t^{\text{act}} = \frac{1}{p}\sum_{k=0}^{p-1} W_{t,k}.$$

This shared action encoder yields one latent-aligned action embedding a_t^{act} per step. It summarizes a short, lagged history of cursor motion and key events and is reused across all injection modes.

Contextual attention mask. In the `contextual` mode, video and action tokens are concatenated into a single sequence and processed under a structured, causal–local attention mask. Appendix Figure 38 can be read as a query–key matrix: rows are queries and columns are keys.

The upper-left block (*V2V*) restricts each frame V_i to attend only to neighboring frames within a window of $\pm w$ steps, so very distant frames cannot interfere. The upper-right block (*V2A*) lets frame V_i see only the most recent actions in a short lag window. Typically this includes actions A_j with $i - (w + \ell) \le j \le i - \ell$, so actions that are too recent to have taken effect are masked. This way, a frame is explained by at most the last few perceivable operations and never by future ones. In the lower-left block (*A2V*), an action A_i attends to frames V_t that occur after it has had time to take effect (roughly $t \ge i + \ell$), but not to earlier frames. The lower-right block (*A2A*) is near-diagonal: actions mostly attend to themselves rather than to unrelated actions.

In practice (w, ℓ) act as fixed hyperparameters that trade off temporal coverage and cost. Together, these choices implement a structured causal prior: actions do not explain past frames, and each frame conditions only on the recent operations that could plausibly have shaped its pixels.

Design insights. Two insights from the GUI experiments motivate this schema. First, raw action streams are bursty and high-dimensional. Cursor and key events arrive in short spikes, and simple smoothing or full-history attention can cause false interpolated motion and underestimated typing speed. Using short, lagged windows and local attention bands makes credit assignment more intuitive: each frame connects to the few operations that could have produced it. Second, control fidelity improves more from better conditioning than from changing the visual backbone. Clean, well-paced supervision and mid- or deep-level action injection improve cursor accuracy and hover timing, while different encodings of the same stream perform similarly. This action schema and mask implement these principles: keep pixels and actions aligned in time, prioritize recent operations over distant ones, and use attention structure rather than capacity alone.

G.4 Cursor rendering and supervision

Cursor rendering and reference construction.

The cursor pipeline applies the same design principles on the visual side. Instead of relying on the global diffusion loss to discover a tiny arrow, we render the cursor explicitly. We treat it as a first-class conditioning signal.

From logs to normalized trajectories. Desktop logs provide per-frame cursor positions in screen coordinates $(x_{\text{screen}}, y_{\text{screen}})$ at the native GUI resolution. We align these with sampled video frames using the same letterbox mapping as the RGB stream. This includes rescaling to target resolution, adding symmetric padding, and normalizing by $(\text{width} - 1, \text{height} - 1)$ to obtain $(x_t, y_t) \in [0, 1]^2$ for each timestep t. Stacking these over time yields the trajectory tensor `mouse_trajectories` used across rendering and action encoding.

Table 14: Raw-action versus meta-action encoders in GUIWorld.

Aspect	Raw-action encoder (v1)	Meta-action encoder (v2)
Action schema	Per-frame multi-hot vector composed of 13 mouse actions and 169 keyboard actions; no explicit type hierarchy.	Hierarchical schema with explicit action types and parameters: `action_types` $\in \{0,1,2,3,4\}$ for {None, Mouse Click/Drag, Mouse Scroll, Keyboard Type, Keyboard Shortcut}, plus typed parameter fields (e.g., button, scroll amount, shortcut ID, text).
Mouse representation	Two signals per frame: (1) continuous cursor trajectory $\texttt{mouse\_trajectories}_t \in \mathbb{R}^2$; (2) discrete mouse events $\texttt{mouse\_action\_events}_t \in \{0,1\}^{13}$ (multi-hot).	Per-frame mouse trajectory $\texttt{mouse\_trajectories}_t$ is encoded by a shared mouse-trajectory encoder and the temporal windowing module into dense mouse embeddings $\texttt{mouse\_latent}_t \in \mathbb{R}^{d_{\text{mouse}}}$, optionally fused with action embeddings.
Keyboard representation	Per-frame keyboard state $\texttt{keyboard\_action\_events}_t \in \{0,1\}^{169}$, where each dimension corresponds to a key or shortcut; only multi-hot activation is available, no text semantics.	Two complementary forms: keyboard shortcuts as `keyboard_shortcut` IDs embedded via an embedding table, and free-form text `keyboard_text` (e.g. "ls -l") encoded by a shared text encoder (e.g., T5) and projected to the action embedding dimension.
Per-frame representation	Mouse and keyboard events are concatenated: $\texttt{raw\_actions}_t = [\texttt{mouse\_events}_t, \texttt{keyboard\_events}_t] \in \mathbb{R}^{182}$.	For each frame t and slot s, type plus parameters are embedded into $\texttt{slot\_embeds}_{t,s} \in \mathbb{R}^D$. Valid slots are aggregated (mean or attention) into a single per-frame action embedding $\texttt{action\_embeds}_t \in \mathbb{R}^D$.
Multi-slot support	No explicit notion of multiple action slots per frame; all mouse and keyboard signals are collapsed into one 182-d multi-hot vector.	Explicit multi-slot design: `action_types` and all parameter tensors have shape (B,T,S) with $S=2$ slots per frame, enabling, e.g., concurrent "mouse click + keyboard shortcut" in the same frame.
Latent-aligned representation	Temporal alignment is implemented ad-hoc inside each v1 action module (e.g., sliding windows over the 182-d per-frame vector), with custom logic per mode (external, internal, residual).	A unified temporal alignment module takes per-frame action embeddings (B,F,D) and VAE compression ratio c, constructs lag-aware temporal windows of size $c \cdot w$, and outputs latent-aligned action features $\texttt{action\_latent} \in \mathbb{R}^{B\times T\times D}$, with analogous `mouse_latent` when mouse features are enabled.
Lag modeling	GUI latency / response lag is implicitly encoded by where and how the sliding window is applied inside each v1 action module; there is no shared lag configuration across modes.	Lag is an explicit hyperparameter `action_lag` in the temporal alignment and in the contextual attention mask: it shifts temporal windows and defines which frames an action can attend to, giving a consistent lag interpretation across modes.
Downstream usage	Raw 182-d vectors (plus trajectories) are fed directly into the v1 action modules (external, internal, residual) without a shared temporal encoder.	The v2 encoder outputs provide a shared conditioning stream for all v2 modes and also supply features for the temporal contrastive loss.
Contrastive supervision	No dedicated temporal contrastive loss between frames and actions; supervision comes only from the generative objective.	Action and mouse latents are coupled to frame features via an InfoNCE-style temporal contrastive loss, encouraging time-aligned action embeddings.

G.5 TRAINING SIGNALS AND ENCODER DESIGN

From trajectories to cursor layers. Starting from normalized (x_t, y_t) coordinates, a cursor-layer module renders a fixed SVG arrow template into RGB and alpha channels. The template is reused across frames. For each timestep t, it is positioned so the hotspot (arrow tip) aligns with (x_t, y_t), clipped to screen bounds, and alpha-blended over a neutral background. This produces two tensors at video frame rate: a cursor-only foreground image $f_t \in [-1, 1]^{3\times H\times W}$ and a soft mask $m_t \in [0, 1]^{1\times H\times W}$ isolating the arrow pixels. Invalid or missing coordinates zero the mask and leave the foreground unchanged, so frames without visible cursors do not add spurious supervision.

Reference images and masks. These cursor layers become reference conditions for the I2V model. For each clip we form reference images $\mathrm{ref\_img}_{0:T-1}$ and masks $\mathrm{ref\_mask}_{0:T-1}$:

- at $t{=}0$, $\mathrm{ref\_img}_0$ is the full desktop frame and $\mathrm{ref\_mask}_0$ is all ones, anchoring the static layout and theme;
- for $t{>}0$, $\mathrm{ref\_img}_t$ is the cursor foreground f_t and $\mathrm{ref\_mask}_t$ is the cursor mask m_t, so only the arrow region is supervised while the background remains free.

The model encodes these references with the same VAE as target frames and concatenates their latents and masks into the diffusion input. It learns to "copy from ref" inside the cursor mask and relies on its dynamics model elsewhere. This makes cursor supervision a pixel-level constraint rather than a side effect of the global loss.

Fourier mouse encoding. The same (x_t, y_t) trajectories serve as a continuous control signal. We apply a Fourier position module: clamp coordinates to $[0, 1]^2$, map them to $[-1, 1]^2$, and compute random Fourier features via a fixed Gaussian projection followed by sine/cosine. A small MLP maps these features to per-frame mouse embeddings. The GUIWorld action encoder then aggregates them with lag-aware, stride-aligned windows to produce latent-aligned mouse features. These features condition the `external`/`contextual`/`residual`/`internal` modes and participate in the temporal contrastive loss.

Cursor-aware losses. Section 3.2 introduces cursor-aware losses that use this construction. A basic variant penalizes position error only in (x, y). Richer variants add Fourier features of the trajectory and, most importantly, an ℓ_2 loss on the reconstructed cursor patch under $\mathrm{ref\_mask}_t$. Table 6 shows that position-only objectives yield low cursor hit rate and visibly jittery arrows even when videos look plausible. Adding the explicit cursor reference stream together with the masked patch loss substantially improves control, reaching 98.7% cursor accuracy. This confirms that explicit cursor rendering plus localized supervision effectively separates "where the arrow is" from "what the rest of the frame should look like".

Temporal contrastive alignment. To strengthen learning signals for the action pathway, we add a lightweight temporal contrastive loss that operates on the same latent timeline as the diffusion model. For each sequence we take per-step frame features $F_t \in \mathbb{R}^{d_f}$ pooled from the latent video. We also take per-step action features $A_t \in \mathbb{R}^{d_a}$ and (optionally) mouse features $M_t \in \mathbb{R}^{d_m}$ produced by the action encoder. Linear projections map these into a common space and the resulting vectors are ℓ_2-normalized. An InfoNCE-style objective brings matching pairs (F_t, A_t) (and, when present, (F_t, M_t)) from the same timestep together. It pushes them away from other timesteps of the *same* sequence. We use frame and action masks to ignore positions without actions. A symmetric variant averages frame-to-action and action-to-frame directions.

When enabled, a small future-prediction head adds a second term. Action features at time t are mapped to a prediction of the frame feature at a slightly later step $t + \Delta$. A mean-squared error encourages consistency between the prediction and the projected future frame. Together, these terms give the action encoder direct gradients tied to specific frames rather than relying solely on the pixel diffusion loss. The contrastive term enforces a one-to-one alignment between actions and the frames they co-occur with. The future head encourages actions to anticipate the visual consequences that appear shortly after they are issued.

Encoder comparison. Table 14 contrasts the two encoders used in GUIWorld. The key theme is moving from a high-dimensional, bursty event vector toward a typed, API-like schema with explicit lag handling. This makes the action stream easier to align with the latent video timeline and to reuse across injection modes.

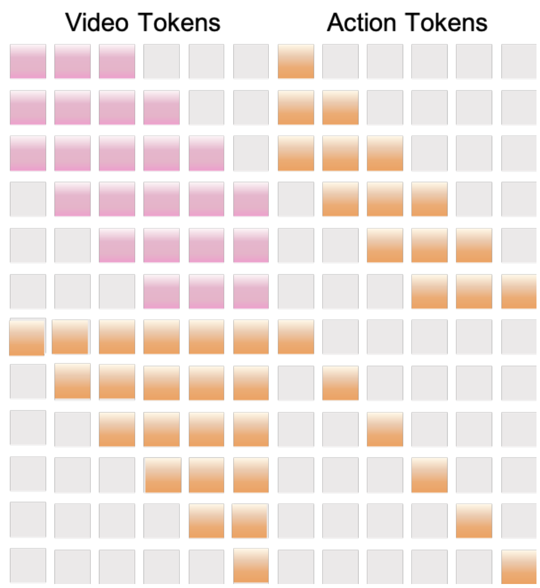


Figure 38: Contextual attention mask for GUIWorld. Video tokens (pink) and action tokens (orange) share one sequence. The mask restricts attention to a short temporal window so each frame attends only to nearby frames and temporally aligned actions.

Injection schemes. The GUI experiments explore four conditioning schemes built on this encoder: `external`, `contextual`, `internal`, and `residual` (Figure 7). Representative action-driven metrics comparing these modes are reported in Table 7. Full metric definitions and the evaluation protocol are provided in Appendix E.6.

G.6 More experiments: Do action encodings matter?

We compare two action encodings under the same injection mode to isolate the effect of representation choice (Table 8). Under `internal` conditioning, the meta-action (API-like) encoding yields small but consistent improvements over the raw-action representation. However, these gains are modest compared to the substantially larger improvements observed when varying the action injection scheme itself (Table 7).

Table 12 contrasts how short commands and shortcuts (e.g., `ls -l`, `ctrl+v`) are represented under the two encodings; the full encoding examples are provided in Appendix E.5. The raw-action encoder treats typing as a stream of individual key events, leaving command or shortcut semantics to be inferred from the sequence. In contrast, the meta-action encoder collapses each interaction into a single typed action with associated text or a shortcut identifier. This design aims to model user actions as structured, tool-like operations rather than fragmented event streams.

In practice, this more structured abstraction does not translate into clear qualitative gains. Rendered text remains similarly smeared under both encodings, and robustness under theme changes and timing noise is largely unchanged. Task-level failures such as re-centering, re-acquisition, and multi-step interactions persist across both representations. We adopt the meta-action encoder as the default for its simplicity and semantic alignment with system-level conditioning. These results suggest that encoding granularity is secondary to alignment quality and injection strategy.